전통 자료 구조를 디딤돌 삼아 읽는

현대 AI 시스템의 자료 구조

AI 시대에 다시 쓰는 자료 구조

배열에서 벡터 인덱스까지,
정의와 구현과 측정으로 세우는
64 개 절

리프 메타 (leaf meta)

AI 시대에 다시 쓰는 자료 구조

전통 자료 구조를 디딤돌 삼아 읽는 현대 AI 시스템의 자료 구조

지은이 리프 메타 (leaf meta)

1 쇄 2026 년 9 월 13 일

지은이가 무료로 공개한 다른 책

`https://github.com/leaf-kit/book-verify-ai-code`

`https://github.com/leaf-kit/book-staircase-math`

`https://github.com/leaf-kit/book-graph-engineering`

`https://github.com/leaf-kit/book-ai-engineer-interview`

리프 메타 (leaf meta)

메타인지 엔지니어 (자칭, 지구 최초). 생각을 생각하는 사람. 다들 끄덕일 때 혼자 갸우뚱하고, 트렌드에 올라타기 전 3초는 뒤집어 봅니다. 주전공은 AI와 뇌과학. 가끔 트렌드나 잡다한 주제도 기웃거립니다. 단순화, 미니멀, 반론, 상상력, 전부 메타인지로 한 번 더 뒤집어 봅니다.

그리고 이건 책에서만 밝히는 건데, 소프트웨어를 만들어 온 지 20년이 넘었습니다. 학교에서 배운 자료 구조를 현장에서 20년 썼고, 지난 몇 년은 그 구조들이 대규모 언어 모델과 벡터 검색 시스템 안에서 어떻게 바뀌었는지를 직접 만들고 재어 봤습니다. 교과서가 틀린 것은 아니었습니다. 교과서가 말없이 세운 가정이 바뀌어 있었습니다. 이 책은 그 가정을 하나씩 다시 세운 기록입니다.

쓴 책으로는 무료로 공개한『AI가 쓴 코드를 검증하는 법』,『계단식 수학』,『그래프 엔지니어링』,『채용을 위한 AI 엔지니어 기술 면접 노트』가 있고, 그 전에는 출판사를 통해 IT 개발서 두 권을 냈습니다.

머리말

자료 구조 강의는 배열과 연결 리스트로 시작한다. 그 순서는 40년 동안 거의 바뀌지 않았다. 그런데 지금 가장 큰 계산이 일어나는 곳, 대규모 언어 모델의 추론 서버와 수십억 벡터의 검색 시스템을 열어 보면 연결 리스트는 없고, 이진 탐색 트리는 없고, 교과서에 없던 구조가 한가운데를 차지하고 있다.

교과서가 틀린 것은 아니다. 교과서가 말없이 세운 가정이 바뀌었다. 메모리 접근 비용이 균일하다는 가정, 데이터의 단위가 스칼라라는 가정, 답이 정확해야 한다는 가정, 구조를 사람이 설계한다는 가정. 이 넷이 흔들리자 그 위에 서 있던 구조들의 값이 달라졌다.

이 책은 그 구조들을 버리지 않는다. 배열, 연결 리스트, 스택, 큐, 해시 테이블, 탐색 트리, 힙, 트라이, 그래프. 강의가 처음에 가르치는 그 구조들이 절마다 디딤돌로 먼저 선다. 이 책은 그 가정을 하나씩 다시 세운다. 절마다 전통 구조를 정의와 불변식으로 다시 세우고, 그 구조가 어느 가정에서 깨지는지를 직접 잰 숫자로 보이고, 깨진 자리를 메운 현대 구조를 같은 밀도로 세운다. 전통 구조는 버리는 것이 아니라 디딤돌이다. 디딤돌 없이 현대 구조만 배우면 외우게 되고, 디딤돌을 딛고 오르면 유도하게 된다.

이 책의 모든 코드는 실제로 돌아가고, 모든 숫자는 실제로 잰 것이다. 잰 기계와 조건은 부록에 있다. 독자의 기계에서 다시 재면 숫자는 달라진다. 달라지는 방향과 비율이 이 책이 가르치려는 것이다.

용어를 정의보다 먼저 두는 이유를 밝혀 둔다. 모르는 낱말로 정의를 시작하면 그 정의는 처음부터 실패다. 그래서 절마다 첫 상자는 언제나 「이 말부터」다.

차례

이 책에서 쓰는 말

이 책이 자기만 쓰는 말은 넷이다. 그 밖의 말은 전부 교과서와 현장의 표준어다.

디딤돌	절마다 처음에 다시 세우는 전통 자료 구조. 정의와 불변식과 복잡도를 정확한 형태로 다시 적는다
균열	그 전통 구조가 AI 시대의 작업량에서 깨지는 자리. 말이 아니라 측정으로 보인다
다시 쓰기	균열을 메운 현대 구조. 디딤돌과 같은 밀도로 정의, 구현, 측정, 분석한다
작업량	구조에 가해지는 연산의 종류와 빈도와 크기의 묶음. 영어의 workload다. 자료 구조는 작업량에 대한 가정의 묶음이다

헷갈릴 만한 짝은 여기서 갈라 둔다.

이동량과 연산 수	이동량 M 는 메모리 계층을 가로지르는 바이트 수이고, 연산 수 W 는 산술 연산의 수다. 이 책은 둘을 따로 센다
연산 밀도	$I = W/M$. 바이트 하나를 옮겨 와서 몇 번 계산하는가. 루프라인 모델의 가로축이다
블록과 캐시 라인	캐시 라인은 프로세서가 한 번에 옮기는 단위이고, 블록은 이 책의 구조가 정한 단위다. 둘이 같을 때가 좋은 설계다
정확 탐색과 근사 탐색	정확 탐색은 답이 하나로 정해지고, 근사 탐색은 재현율로 잰다. 5부의 인덱스는 전부 근사 탐색이다

수학 기호는 부록 「기호와 표기」에 모아 두었다. 복잡도는 $O(\cdot)$ 로, 이동량은 M 로, 연산 수는 W 로 적는다.

0장

왜 다시 쓰는가

2023년에 공개된 한 추론 서버 논문은 대규모 언어 모델의 메모리 중 60에서 80퍼센트가 낭비되고 있었다고 적었다. 낭비의 원인은 새로운 것이 아니었다. 운영체제 교과서 3장에 나오는 단편화였고, 해법도 같은 장에 나오는 페이지 테이블이었다. 40년 된 구조가 가장 새로운 시스템의 한가운데에 다시 들어온 것이다.

이 책은 그런 자리를 열여섯 개 모았다. 전통 자료 구조 하나를 정의와 불변식으로 다시 세우고, 그 구조가 AI 시대의 작업량에서 어디서 깨지는지를 직접 잰 숫자로 보이고, 깨진 자리를 메운 현대 구조를 같은 밀도로 세운다. 이 장은 그 셋을 하기 전에 필요한 도구를 세운다. 비용 모델, 루프라인, 그리고 이 책이 끝까지 다시 보는 문제 하나다.

이 장에서 세우는 것

0.1 자료 구조는 가정의 묶음이다

머리말에서 교과서가 틀린 것이 아니라 가정이 바뀌었다고 적었다. 이 절은 그 가정 중 가장 깊은 곳에 있는 것 하나를 꺼낸다. 메모리 접근 한 번의 값은 어디를 읽든 같다는 가정이다.

이 가정은 너무 깊이 박혀 있어서 대부분의 교과서가 가정이라고 부르지도 않는다. 배열 합산이 $O(n)$ 이라는 문장은 이 가정 위에서만 참이다. 같은 n 개를 읽는데 순서만 바꾸면 시간이 네 배가 되는 것을 이 절에서 직접 잰다.

학습 목표

자료 구조를 상태, 연산, 불변식, 비용 모델 넷으로 적을 수 있다. 같은 연산 수의 두 접근 순서가 왜 다른 시간을 내는지 이동량으로 설명할 수 있다.

이 말부터

균일 비용 모델	어느 주소를 읽든 값이 같다고 치는 계산 모델. 교과서의 복잡도는 전부 이 모델 위에 있다
캐시 라인	프로세서가 메모리에서 한 번에 가져오는 단위. 이 책의 기계에서는 64바이트다
이동량	메모리 계층을 가로질러 옮겨진 바이트 수. 이 책은 연산 수와 따로 센다

디딤돌

자료 구조 교과서는 첫 장에서 자료 구조를 정의하지 않는 경우가 많다. 배열과 연결 리스트를 바로 보여주고 정의는 독자의 직관에 맡긴다. 이 책은 정의부터 세운다. 뒤에서 이 정의의 네 번째 항목이 계속 문제가 되기 때문이다.

정의 0.1: 자료 구조

자료 구조는 넷의 묶음 (S, Ops, Inv, C) 다. S 는 구조가 가질 수 있는 상태의 집합, Ops 는 상태를 읽거나 바꾸는 연산의 집합, Inv 는 모든 연산 전후에 성립해야 하는 불변식의 집합, C 는 연산 하나의 값을 정하는 비용 모델이다.

불변식 모든 $op \in Ops$ 와 모든 $s \in S$ 에 대해, $Inv(s)$ 이면 $Inv(op(s))$ 다.

앞의 셋은 익숙하다. 스택의 상태는 원소의 열이고, 연산은 push 와 pop 이고, 불변식은 마지막에 넣은 것이 먼저 나온다는 것이다. 넷째 항목인 비용 모델 C 가 이 책의 주인공이다. 같은 (S, Ops, Inv) 라도 C 가 바뀌면 어느 구현이 좋은지가 바뀐다.

전통 교과서의 C 는 하나로 고정되어 있다. 이름은 여럿이지만 내용은 같다.

정의 0.2: 균일 비용 모델

> 메모리는 주소가 붙은 칸의 열이고, 어느 칸이든 읽거나 쓰는 값은 상수 1이다. 산술 연산 하나의 값도 상수 1이다. 연산의 총비용은 이 상수들의 합이고, 이 책은 그 합을 연산 수 W 로 적는다.

이 모델에서 배열 합산의 값을 세면 $W = n$ 이다. 원소마다 읽기 하나와 덧셈 하나이고, 상수 배는 무시한다. 같은 n 개를 무작위 순서로 읽어 더해도 $W = n$ 이다. 읽는 칸의 주소가 어디든 값이 1이기 때문이다. 그러니 균일 비용 모델은 두 접근 순서를 구분하지 못한다. 구분할 이유도 없다.

예제 0.1: 같은 값, 다른 순서

> $A = [3, 1, 4, 1, 5, 9, 2, 6]$ 이라 하자. 앞에서 뒤로 더하면 읽기 8번, 덧셈 7번이다. $[5, 0, 7, 2, 4, 1, 6, 3]$ 번째 순서로 더해도 읽기 8번, 덧셈 7번이다. 정의 0.2 은 둘의 값을 똑같이 15로 센다. 답도 31로 같다.

이 예제의 두 순서는 답이 같고 값도 같다. 값이 같다는 것은 모델의 결론이지 사실이 아니다. 사실인지는 재 봐야 안다.

균열

같은 실험을 이 책의 기계에서 했다. 배열 하나를 만들고, 원소를 한 번씩 읽어 더하되 순서만 둘로 나눴다. 하나는 앞에서 뒤로, 하나는 무작위 순열이다. 원소 수를 만 개에서 오천만 개까지 늘렸다.

실험 0.1 같은 원소 수, 두 접근 순서

원소 수	크기 MB	순차 ms	무작위 ms	비율
10000	0.00	0.04	0.04	0.99
100000	0.40	0.20	0.39	1.96
1000000	4.00	2.67	3.61	1.35
10000000	40.0	33.0	115	3.47
50000000	200	338	1317	3.90

측정 조건. float32 배열, 원소를 정확히 한 번씩 읽어 더한다. 일곱 번 반복한 중앙값이다.

만 개일 때는 두 순서의 시간이 같다. 비율이 1.0이다. 균일 비용 모델의 예측 그대로다. 원소가 백만 개를 넘으면서 비율이 벌어지고, 천만 개에서는 무작위 쪽이 3.5배 느리다. 읽은 원소의 수는 같고 더한 횟수도 같다. 달라진 것은 읽은 순서뿐이다.

비율이 갈라지는 자리가 어디인지 보면 원인이 보인다. 만 개는 40KB 이고 백만 개는 4MB, 천만 개는 40MB 다. 이 기계의 마지막 캐시는 8MB 다. 배열이 캐시에 다 들어가는 동안은 모델이 맞고, 캐시를 넘어서는 순간부터 모델이 틀린다.

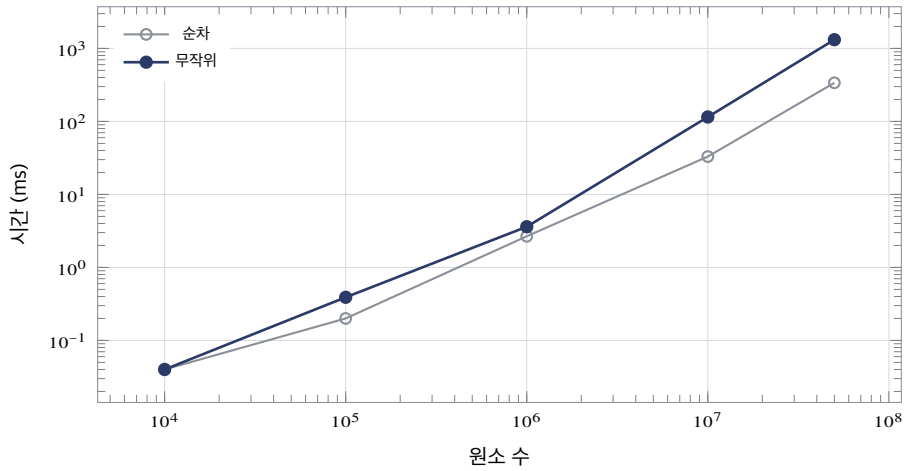


그림 1.. 원소 수에 따른 두 접근 순서의 시간. 배열이 마지막 캐시 8MB 를 넘는 자리에서 갈라진다

관찰

> 접근 순서가 시간을 가르는 것은 데이터가 캐시보다 클 때뿐이다. 캐시 안에서는 균일 비용 모델이 그대로 맞는다.

왜 순서가 문제가 되는지는 메모리가 원소 단위로 오지 않는다는 데서 온다. 프로세서는 원소 하나를 달라고 하면 그 원소가 든 캐시 라인 64 바이트를 통째로 가져온다. float32 라면 열여섯 개가 한 번에 오고, 순차로 읽으면 그 열여섯 개를 다 쓴다. 무작위로 읽으면 열여섯 개 중 하나만 쓰고 나머지 열다섯은 버린다.

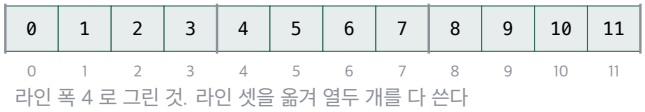


그림 2.. 같은 열두 원소를 순차로 읽을 때. 캐시 라인 하나를 가져오면 그 안의 원소를 전부 쓴다

그림 3.. 같은 열두 원소를 무작위로 읽을 때. 라인을 가져올 때마다 하나만 쓰고 나머지는 버린다

두 그림의 라인 수는 같아 보이지만 그것은 열두 개일 때 이야기다. 원소가 캐시보다 많아지면 앞서 가져온 라인이 뒤에서 다시 쓰이기 전에 밀려난다. 그래서 무작위 접근은 원소 하나마다 라인 하나를 새로 가져오는 쪽으로 수렴한다.

다시 쓰기

비용 모델을 바꾼다. 연산 수를 세던 자리에 이동량을 하나 더 센다. 이 모델은 새것이 아니다. Aggarwal 과 Vitter 가 1988 년에 디스크와 주기억 사이의 전송을 세려고 만든 것이다 [확립] [1]. 이 책은 그 모델을 디스크가 아니라 캐시와 주기억 사이에 놓는다.

정의 0.3: 이동량 모델

> 메모리는 두 층이다. 빠른 층은 Z 바이트를 담고, 느린 층은 무한하다. 느린 층과 빠른 층 사이의 전송은 B 바이트 라인 단위로만 일어난다. 연산의 비용은 두 수의 짝 (W, M) 다. W 는 산술 연산 수이고, M 는 옮겨진 라인 수다.
>
> **불변식** 빠른 층에 있는 라인의 수는 언제나 Z/B 이하다.

이 정의로 두 접근 순서를 다시 세면 다른 답이 나온다.

정리 0.4: 순차 접근과 무작위 접근의 이동량

> 원소 n 개가 각각 s 바이트이고 $ns > Z$ 라 하자. 순차 접근의 이동량은 $M \le \lceil ns/B \rceil + 1$ 이다. 접근 순서가 균등 무작위 순열이면 기대 이동량은 $M \ge n\,(1 - Z/(ns))$ 이다.

증명. 순차 접근에서 라인 하나를 가져오면 그 안의 B/s 개 원소를 연달아 쓰므로, 라인은 $\lceil ns/B \rceil$ 개를 넘지 않는다. 경계가 맞지 않을 때 하나가 더 든다. 무작위 순열에서는 각 접근이 만나는 라인이 빠른 층에 있을 확률이 최대 $Z/(ns)$ 다. 빠른 층이 담을 수 있는 바이트가 Z 이고 전체가 ns 이기 때문이다. 따라서 접근 하나가 새 라인을 부를 확률은 최소 $1 - Z/(ns)$ 이고, 기대값의 선형성으로 n 을 곱한다. ■

이 정리를 실험 0.1 의 천만 개 행에 대입하면 순차 접근은 라인 625,000 개, 무작위 접근은 최소 800 만 개다. 비율이 열두 배다. 잰 시간의 비율은 3.5 배였다. 차이는 프로세서가 라인 여러 개를 동시에 가져오는 데서 온다. 모델은 방향과 크기의 차수를 맞혔고, 상수는 기계가 정한다. 이 책의 모델은 전부 그 정도만 약속한다.

구현

두 접근 순서를 같은 함수로 돌린다. 순서는 인덱스 배열로 넘긴다. 그러면 순서를 빼고는 모든 것이 같다는 것을 코드가 보장한다.

지금 보는 코드 같은 배열을 인덱스 배열이 정한 순서로 읽어 더한다. 실험 0.1 이 부른 함수다.

구현 접근 순서를 인자로 받는 합산 `samples/dsai/cost.py:16`

```python
def random_index(n: int, seed: int) -> np.ndarray:
    """같은 n 개 원소를 한 번씩, 다만 무작위 순서로 찾아간다."""
    rng = np.random.default_rng(seed)
    return rng.permutation(n).astype(np.int64)


def gather_sum(values: np.ndarray, index: np.ndarray) -> float:
    """index 가 가리키는 순서대로 values 를 읽어 더한다.

    두 순서 모두 원소 n 개를 정확히 한 번씩 읽는다. 연산 수 W 는 같다.
    다른 것은 메모리 계층을 가로지르는 이동량 M 뿐이다.
    """
    return float(values[index].sum())
```

읽어 보면 `random_index` 는 0 부터 $n-1$ 의 순열을 돌려준다. 같은 원소를 정확히 한 번씩 가리키므로 두 순서의 연산 수는 같다. `gather_sum` 은 순서대로 모아 더한다. 시간 차이는 전부 `values[index]` 가 라인을 몇 개 가져오느냐에서 온다.

의사코드로 적으면 이렇다. 실행 코드와 이름을 맞춰 두었다.

알고리즘 0.1 순서 지정 합산 `dsai.cost.gather_sum`

입력: 값 배열 $V[0..n-1]$, 순서 배열 $P[0..n-1]$ (P 는 $0..n-1$ 의 순열)
출력: $\sum_i V[i]$

1: $t \leftarrow 0$
2: **반복** $j = 0$ **부터** $n-1$ **동안**
3: $\quad t \leftarrow t + V[P[j]]$
4: **끝 반복**
5: **반환** t

정리 0.4 의 라인 수를 실제로 세는 함수도 두었다. 접근 순서가 건드리는 라인의 번호를 구하고 번호가 바뀌는 횟수를 센다.

구현 접근 순서가 건드리는 라인 수를 센다 `samples/dsai/cost.py:31`

```python
def blocks_touched(index: np.ndarray, elem_bytes: int, line_bytes: int) -> int:
    """이 접근 순서가 건드리는 캐시 라인 수를 센다.

    캐시가 라인 하나만 기억한다고 치는 가장 비관적인 셈이다.
    순차 접근은 n * elem_bytes / line_bytes 개, 무작위 접근은 n 개에 가깝다.
    """
    per_line = max(1, line_bytes // elem_bytes)
    lines = index // per_line
    changes = np.count_nonzero(np.diff(lines)) + 1
    return int(changes)
```

분석

균일 비용 모델은 틀린 모델이 아니라 조건이 붙은 모델이다. 데이터가 Z 안에 들어가면 M 는 상수에 가깝고 W 만 남는다. 실험 0.1 의 만 개 행이 그 조건이다. 그 안에서는 교과서의 모든 분석이 그대로 맞는다.

문제는 이 책이 다루는 작업량이 거의 언제나 그 조건 밖에 있다는 것이다. 모델 가중치 하나가 수백 MB 이고, 벡터 인덱스 하나가 수십 GB 다. 그래서 이 책은 복잡도를 적을 때 W 와 M 를 따로 적는다. 둘 중 어느 쪽이 병목인지는 다음 절의 루프라인이 정한다.

현장에서

> Williams, Waterman, Patterson 이 2009 년에 낸 루프라인 모델은 연산 수와 이동량의 비로 병목을 읽는 도구다 [확립] [2]. 이 책의 모든 실험을 읽는 기본 틀로 다음 절에서 세운다.

되짚기

> 자료 구조는 상태, 연산, 불변식, 비용 모델의 묶음이고, 전통 교과서는 비용 모델을 균일 비용으로 고정했다. 데이터가 캐시를 넘으면 그 모델이 틀리고, 이 책은 이동량 M 를 따로 센다. 다음 절에서 이 M 가 다른 세 가정과 어떻게 엮이는지 본다.

0.2 바뀐 가정 넷

0.1 에서 비용 모델 하나를 바꿨다. 그 모델이 흔들린 것은 우연이 아니다. 전통 교과서가 말없이 세운 가정이 넷 있고, AI 시대의 작업량은 넷 다 건드린다. 이 절은 그 넷을 문장으로 적고, 첫째 가정을 이 기계에서 직접 잰다.

넷을 한 번에 다 재지는 않는다. 둘째는 1 부에서, 셋째는 5 부에서, 넷째는 8 장과 17 장에서 잰다. 여기서는 넷이 무엇이고 서로 어떻게 이어지는지를 적고, 그 뼈대가 되는 도구 하나를 세운다. 루프라인 모델이다.

학습 목표

> 전통 교과서의 가정 넷을 문장으로 말할 수 있다. 연산 밀도를 계산하고 루프라인 위에서 어느 연산이 대역폭에 막히는지 읽을 수 있다.

이 말부터

> | | |
> |---|---|
> | 작업량 | 구조에 가해지는 연산의 종류와 빈도와 크기의 묶음. 자료 구조는 작업량에 대한 가정의 묶음이다 |
> | 연산 밀도 | 바이트 하나를 옮겨 와서 하는 계산의 수. $I = W/M$, 단위는 FLOP/byte |
> | 대역폭 | 메모리 계층 사이를 초당 옮길 수 있는 바이트 수. 이 책의 기계에서는 주기억이 약 17GB/s 다 |

디딤돌

자료 구조는 작업량에 맞춰 고르는 것이다. 이 문장은 모든 교과서에 있다. 그런데 작업량이 무엇인지는 정의하지 않는 경우가 많다. 이 책은 정의한다. 뒤에서 작업량의 어느 항목이 바뀌었는지를 정확히 짚어야 하기 때문이다.

정의 0.5: 작업량

> 작업량은 셋의 묶음 (ops, f, σ) 다. ops 는 구조에 요청되는 연산의 종류, f 는 종류마다의 빈도, σ 는 연산 하나가 다루는 데이터의 크기 분포다. 같은 구조라도 작업량이 다르면 비용 모델 C 가 내는 값이 다르다.

예제 0.2: 두 작업량, 한 구조

> 정렬된 배열에 탐색만 하루 백만 번 오면 $(\{\text{탐색}\}, 10^6, \text{원소 하나})$ 다. 이진 탐색이 $O(\log n)$ 이라 잘 맞는다. 같은 배열에 삽입이 탐색만큼 오면 ops 에 삽입이 들어오고, 삽입 하나가 $O(n)$ 이라 배열이 나빠진다. 구조는 그대로인데 작업량의 첫 항목이 바뀌자 답이 바뀌었다.

전통 교과서의 작업량은 이 정의의 세 항목에 대해 말없이 넷을 가정한다. 문장으로 적으면 아래와 같다.

번호	가정	어디에 숨어 있는가
1	메모리 접근 비용은 균일하다	균일 비용 모델. 모든 복잡도 표
2	데이터의 단위는 스칼라나 레코드다	σ 가 원소 하나. 배열 원소, 노드 하나, 키 하나
3	답은 정확해야 한다	탐색의 정의. 찾거나 못 찾거나 둘 중 하나
4	구조는 사람이 설계한다	구조의 모양이 데이터와 무관하게 고정된다

첫째는 0.1 에서 이미 잰 것이다. 이 절은 첫째를 도구로 만든다. 그 도구가 있어야 나머지 셋을 재는 실험을 읽을 수 있다.

균열

같은 기계에서 연산 셋을 쟀다. 내적, 행렬 벡터 곱, 행렬 곱이다. 셋 다 곱하고 더하는 것뿐이라 연산의 종류는 같다. 다른 것은 바이트 하나를 옮겨 와서 몇 번 계산하느냐다. 그 비율이 연산 밀도이고, 실험은 밀도에 따라 처리량이 얼마나 달라지는지를 본다.

실험 0.2 연산 밀도와 실측 처리량

연산	밀도	처리량
내적	0.25	3.75
행렬 벡터 곱	0.50	20.9
행렬 곱	341	192

측정 조건. float32. 내적은 원소 1,600만 개, 행렬 벡터 곱은 4096 × 4096, 행렬 곱은 2048 × 2048. 일곱 번 중앙값.

내적은 원소마다 곱 하나 합 하나이고 두 벡터를 한 번 읽으므로 밀도가 2/8 = 0.25 다. 행렬 벡터 곱은 행렬 원소마다 두 번 계산하고 원소를 한 번 읽으니 밀도가 0.5 근처다. 행렬 곱은 원소를 한 번 읽어 n 번 쓰므로 밀도가 수백이다. 같은 곱셈 덧셈인데 처리량은 내적이 행렬 곱의 50분의 1이다.

이 표를 한 그림에 놓는 것이 루프라인이다. 가로축이 밀도, 세로축이 처리량이고 둘 다 로그다. 지붕이 두 개다. 왼쪽은 대역폭에 비례해 올라가는 사선, 오른쪽은 연산 장치의 피크에서 멈추는 수평선이다.

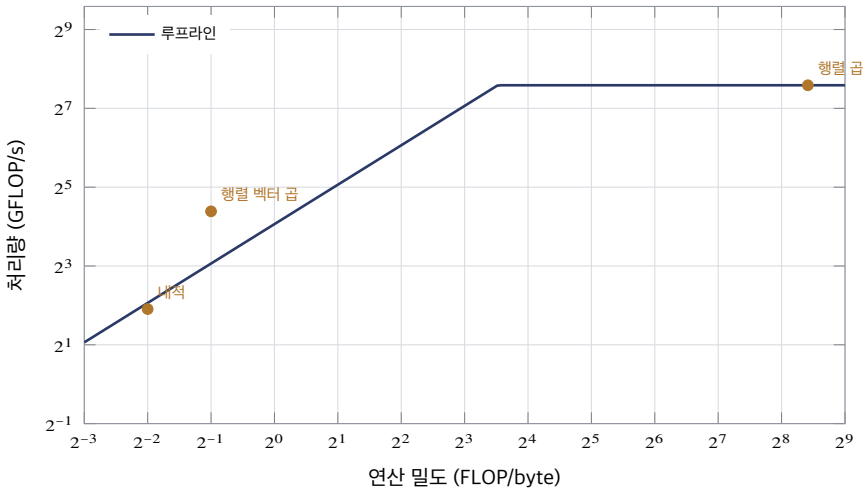


그림 4.. 이 책이 쓰는 기계의 루프라인. 사선은 대역폭 16.7GB/s, 수평선은 피크 192GFLOP/s. 두 지붕이 만나는 밀도가 11.5FLOP/byte 다

내적이 사선 바로 아래에 붙어 있다. 대역폭에 막힌 것이다. 행렬 곱은 수평선에 붙어 있다. 연산 장치에 막힌 것이다. 행렬 벡터 곱은 사선 아래에 있는데, 이 연산은 여러 코어로 도는 라이브러리 구현이라 코어 하나의 내적보다 대역폭을 더 끌어 쓴다.

관찰

> 이 기계에서 밀도 8 FLOP/byte 아래의 연산은 연산 장치를 아무리 좋게 바꿔도 빨라지지 않는다. 옮기는 데 시간을 다 쓰기 때문이다.

한 가지 정정을 적어 둔다. 처음에는 대역폭을 코어 하나의 복사로 쟀다. 그랬더니 행렬 벡터 곱이 지붕을 뚫고 올라갔다. 모델이 틀린 것이 아니라 지붕을 낮게 그린 것이었다. 코어 하나로는 이 기계의 메모리를 다 채우지 못한다. 네 코어가 동시에 복사한 값으로 다시 그렸고, 그것이 위 그림이다.

주의

> 루프라인의 대역폭은 코어 하나가 아니라 기계 전체의 값이다. 코어 하나로 재면 지붕이 낮게 그려지고, 지붕을 뚫는 점이 나온다.

다시 쓰기

루프라인은 정의 0.3 의 두 수 (W, M) 를 하나의 비로 접은 것이다. Williams, Waterman, Patterson 이 2009 년에 다중 코어 프로세서의 성능을 한 그림으로 읽으려고 제안했다 [확립] [2]. 이 책은 그 그림을 자료 구조 하나하나에 갖다 댄다.

정의 0.6: 루프라인 모델

> 기계는 두 수로 요약된다. 연산 장치의 피크 처리량 P (FLOP/s) 와 메모리 대역폭 β (byte/s) 다. 연산 밀도 $I = W/M$ 인 연산이 이 기계에서 달성할 수 있는 처리량의 상한은
>
> $$T(I) = \min(P,\ \beta I)$$
>
> 이다. 두 지붕이 만나는 밀도 $I^* = P/\beta$ 를 능선점이라 부른다.

명제 0.7: 루프라인 상한

> 연산 하나가 W 번 계산하고 M 바이트를 옮긴다면, 그 연산의 실행 시간은 $\max(W/P,\ M/\beta)$ 이상이다. 따라서 처리량은 $T(I)$ 를 넘지 못한다.

증명. 계산은 초당 P 번을 넘지 못하므로 W 번 계산하는 데 W/P 초 이상이 든다. 이동은 초당 β 바이트를 넘지 못하므로 M/β 초 이상이 든다. 둘은 겹쳐서 일어날 수 있으므로 총시간의 하한은 둘 중 큰 쪽이다. 처리량은 W 를 시간으로 나눈 것이므로 $W/\max(W/P, M/\beta) = \min(P, \beta I)$ 를 넘지 못한다. ■

이 명제가 말하는 것은 상한이다. 실제 처리량은 그 아래 어디든 있을 수 있다. 그림에서 점이 지붕에 붙어 있으면 그 연산은 더 빨라질 여지가 없고, 지붕에서 멀면 구현에 문제가 있다. 실험 0.2 의 내적은 지붕에 붙어 있으므로 코드를 아무리 고쳐도 빨라지지 않는다. 빨라지려면 밀도를 올려야 하고, 밀도를 올리는 것은 코드가 아니라 자료 구조의 일이다.

이 책의 모든 다시 쓰기는 결국 하나의 일을 한다. 같은 연산의 밀도를 능선점 오른쪽으로 옮기는 것이다.

구현

루프라인의 산수는 짧다. 밀도를 구하고 두 지붕 중 낮은 쪽을 고른다.

지금 보는 코드 연산 밀도와 달성 가능 처리량. 실험 0.2 의 표와 그림을 만드는 데 쓴 함수다.

구현 밀도, 상한, 능선점 `samples/dsai/roofline.py:50`

```python
def intensity(flops: int, nbytes: int) -> float:
    """연산 밀도 I = W / M. 단위는 FLOP/byte."""
    return flops / nbytes


def attainable(peak_gflops: float, bandwidth_gbs: float, i: float) -> float:
    """정의 0.3. 달성 가능한 처리량은 두 지붕 중 낮은 쪽이다."""
    return min(peak_gflops, bandwidth_gbs * i)


def ridge_point(peak_gflops: float, bandwidth_gbs: float) -> float:
    """두 지붕이 만나는 밀도. 이보다 낮으면 대역폭에, 높으면 연산에 막힌다."""
    return peak_gflops / bandwidth_gbs
```

읽어 보면 `intensity` 는 정의 0.6 의 I 이고 `attainable` 은 $T(I)$ 다. `ridge_point` 는 두 지붕이 만나는 밀도다. 이 셋만 있으면 어느 자료 구조 연산이든 그림에 찍을 수 있다.

밀도를 구하려면 연산의 W 와 M 를 세어야 한다. 그 세는 법이 이 책이 절마다 하는 일이다. 내적의 경우는 아래처럼 센다.

구현 내적과 행렬 곱의 연산 수와 바이트 수 `samples/dsai/roofline.py:26`

```python
def flops_dot(n: int) -> int:
    return 2 * n


def flops_matvec(m: int, n: int) -> int:
    return 2 * m * n


def flops_matmul(m: int, k: int, n: int) -> int:
    return 2 * m * k * n


def bytes_dot(n: int, elem_bytes: int) -> int:
    return 2 * n * elem_bytes


def bytes_matvec(m: int, n: int, elem_bytes: int) -> int:
    return (m * n + n + m) * elem_bytes


def bytes_matmul(m: int, k: int, n: int, elem_bytes: int) -> int:
    return (m * k + k * n + m * n) * elem_bytes
```

분석

가정 넷이 서로 어떻게 이어지는지 여기서 적어 둔다. 넷은 독립이 아니다. 첫째 가정이 깨지면 이동량이 값이 되고, 이동량이 값이 되면 데이터를 원소 하나씩 옮기는 것이 손해가 된다. 그래서 단위가 벡터와 텐서로 커진다. 이것이 둘째 가정이 깨지는 경로다.

단위가 벡터가 되면 벡터 사이의 같음이 정의되지 않는다. 임베딩 두 개가 정확히 같을 일은 없다. 그래서 탐색의 답이 정확한 하나가 아니라 가까운 k 개가 되고, 그 k 개를 다 찾았는지를 재현율로 잰다. 이것이 셋째 가정이 깨지는 경로다.

넷째는 조금 다른 곳에서 온다. 데이터의 분포가 알려져 있으면 그 분포에 맞춘 구조가 일반 구조보다 이동량이 적다. 분포를 사람이 미리 알 수 없으니 데이터에서 배운다. 8장의 학습된 인덱스가 그것이다. 그리고 구조를 고르는 코드 자체를 사람이 덜 쓰게 되면서, 고르는 판단만 사람에게 남는다. 17장이 그 이야기다.

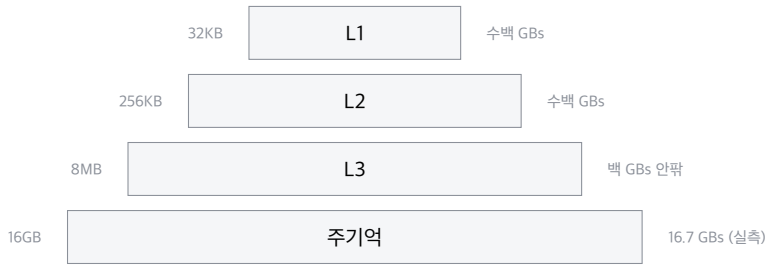


그림 5.. 이 기계의 메모리 계층. 위가 작고 빠르다. 왼쪽은 용량, 오른쪽은 실측 또는 공개 사양의 대역폭 차수

현장에서

> Hennessy 와 Patterson 은 2019 년 튜링상 강연에서 범용 프로세서의 성능 향상이 멈추고 영역 특화 구조의 시대가 왔다고 적었다 [확립] [3]. 영역 특화 구조는 밀도가 높은 연산에서만 값을 한다. 자료 구조가 밀도를 정하므로, 이 책의 주제가 그 시대의 전제다.

이 절이 세운 것은 도구 하나다. 그 도구는 절마다 같은 자리에서 다시 나온다. 디딤돌의 밀도를 재고, 다시 쓴 구조의 밀도를 재고, 둘이 능선점의 어느 쪽에 있는지를 본다. 그 두 점의 거리가 그 절이 얻은 것이다.

되짚기

> 전통 교과서의 가정 넷을 적었고, 첫째를 루프라인 모델로 만들었다. 밀도가 능선점보다 낮은 연산은 대역폭에 막힌다. 다음 절에서 이 책이 끝까지 다시 보는 문제 하나를 세우고, 그 문제의 가장 단순한 풀이를 루프라인 위에 놓는다.

0.3 이 책이 끝까지 다시 보는 문제

0.2 에서 셋째 가정이 깨지는 경로를 적었다. 단위가 벡터가 되면 답이 정확한 하나에서 가까운 k 개로 바뀐다. 이 절은 그 문제를 정확히 정의하고, 가장 단순한 풀이를 구현해서 잰다. 이 풀이가 5부 전체의 정답 기준이 된다.

이 책은 장마다 새 예제를 만들지 않는다. 작은 검색 생성 시스템 하나를 두고 장마다 부품 하나씩을 다시 쓴다. 그 시스템의 이름은 `dsai` 이고, 이 절의 끝에 어느 장이 어느 부품을 바꾸는지 그린다.

학습 목표

> k 최근접 탐색 문제와 재현율을 정의할 수 있다. 전수 탐색의 이동량을 세고, 그것이 어느 크기부터 대역폭에 막히는지 루프라인으로 설명할 수 있다.

이 말부터

최근접 탐색	벡터 n 개 중 질의와 가장 가까운 k 개를 찾는 문제. 영어로 k-NN 이라 쓴다
재현율	근사 풀이가 찾은 k 개 중 정확한 답 k 개에 든 것의 비율. 1.0 이 정확 탐색이다
임베딩	문서나 토큰을 고정 차원의 실수 벡터로 바꾼 것. 이 절에서는 난수 벡터로 대신한다

디딤돌

탐색은 자료 구조 교과서의 절반을 차지한다. 그런데 그 탐색은 전부 같음을 묻는다. 키가 있는가, 있으면 어디에 있는가. 벡터에는 같음이 없으므로 물음을 바꿔야 한다.

정의 0.8: k 최근접 탐색

> 벡터 집합 $X = \{x_1, \dots, x_n\} \subset \mathbb{R}^d$, 거리 함수 δ, 질의 $q \in \mathbb{R}^d$, 정수 k 가 주어진다. k 최근접 탐색은 $\delta(x_i, q)$ 가 가장 작은 i 를 k 개 돌려주는 문제다. 이 책은 단위 벡터에서 $\delta(x, q) = 2 - 2\langle x, q \rangle$ 를 쓴다. 내적이 클수록 가깝다.

거리를 내적으로 쓰는 이유는 계산이 싸기 때문이다. 단위 벡터에서는 $\|x - q\|^2 = \|x\|^2 + \|q\|^2 - 2\langle x, q \rangle = 2 - 2\langle x, q \rangle$ 이므로 순서가 유클리드 거리와 같다. 그래서 정확 탐색은 내적 n 개를 구하고 큰 것 k 개를 고르는 일이 된다.

예제 0.3: 손으로 푸는 최근접 탐색

> 평면 위의 점 다섯 개 $a=(1,2)$, $b=(3,0)$, $c=(-1,-1)$, $d=(0,4)$, $e=(2,2)$ 와 질의 $q=(0,0)$, $k=2$ 라 하자. 유클리드 거리의 제곱은 a 가 5, b 가 9, c 가 2, d 가 16, e 가 8 이다. 가장 작은 둘은 c 와 a 이므로 답은 $\{c, a\}$ 다. 다섯 개를 전부 재고 둘을 골랐다.

이 예제가 한 일을 그대로 알고리즘으로 적으면 전수 탐색이다. n 개를 전부 재고 k 개를 고른다. 단순하고 틀리지 않는다. 첫 줄이 행렬 벡터 곱이라는 것을 눈여겨봐 둔다. 0.2 에서 밀도 0.5 로 잰 바로 그 연산이다.

알고리즘 0.2 전수 최근접 탐색 `dsai.knn.knn_bruteforce`

> **입력:** 단위 벡터 행렬 X $(n \times d)$, 단위 질의 q, 정수 k
> **출력:** 내적이 큰 순서로 정렬된 인덱스 k 개
>
> 1: $s \leftarrow Xq$ ▷ 내적 n 개. $W = 2nd$, $M = 4nd$ 바이트
> 2: $T \leftarrow s$ 에서 큰 값 k 개의 인덱스 ▷ 부분 선택. 10장에서 힙으로 다시 본다
> 3: **반환** T 를 s 의 내림차순으로 정렬한 것

근사 풀이를 재는 잣대도 여기서 정의한다. 5부의 인덱스는 전부 이 잣대로 잰다. 잣대가 필요한 이유는 근사 풀이가 답을 놓치기 때문이다. 놓친 개수를 세지 않으면 빠른 풀이와 틀린 풀이를 구분할 수 없다.

정의 0.9: 재현율

> 정확 탐색의 답을 T^*, 어떤 풀이의 답을 T 라 하자. 둘 다 크기 k 다. 그 풀이의 재현율은 $|T \cap T^*| / k$ 다. 전수 탐색의 재현율은 정의에 의해 1.0 이다.

균열

전수 탐색을 이 책의 기계에서 쟀다. 차원 128, $k=10$, 문서 수를 천 개에서 백만 개까지 늘렸다. 임베딩은 난수 단위 벡터다. 진짜 임베딩은 6장과 9장에서 만들지만, 이 실험에서 보려는 것은 값의 분포가 아니라 크기에 따른 시간이다.

실험 0.3 전수 탐색의 지연과 처리량

문서 수	크기 MB	지연 ms	초당 M벡터
1000	0.50	0.03	29.7
10000	5.10	0.30	33.9
100000	51.2	4.06	24.6
1000000	512	53.7	18.6

측정 조건. float32, 차원 128, $k=10$, 질의 하나. 아홉 번 반복한 중앙값이다.

지연은 문서 수에 비례한다. 천 배 늘리면 천 배 넘게 느려진다. 백만 문서에서 질의 하나에 54 밀리초이니 초당 열여덟 질의다. 검색 서비스 하나가 감당할 수 있는 양이 아니다.

처리량 열을 보면 또 하나가 보인다. 초당 처리하는 벡터 수가 만 문서에서 34M 이었다가 십만 문서에서 25M 으로, 백만에서 19M 으로 떨어진다. 5MB 는 캐시에 들고 51MB 는 들지 않는다. 0.1 의 갈라지는 자리가 여기서 다시 나온다.

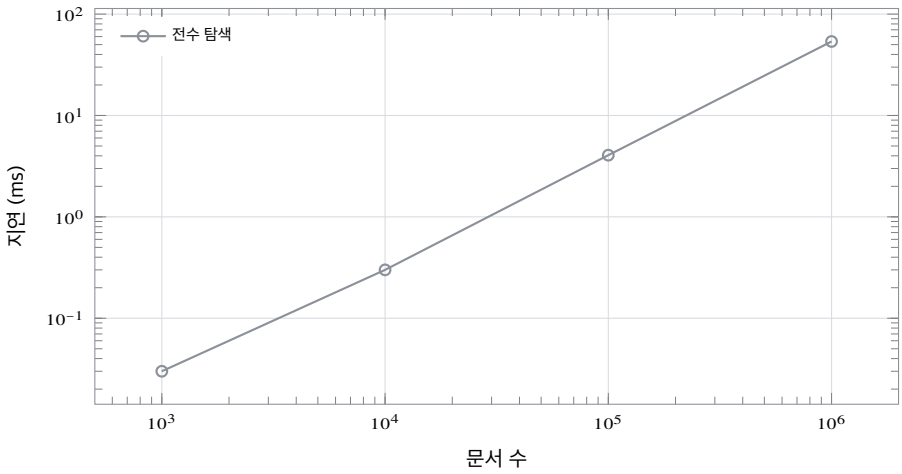


그림 6.. 문서 수에 따른 전수 탐색의 지연. 기울기 1 의 직선이고, 캐시를 넘는 자리에서 조금 더 가팔라진다

루프라인에 놓으면 이 연산의 자리가 보인다. 알고리즘 0.2 의 첫 줄은 행렬 벡터 곱이고, 실험 0.2 에서 그 밀도는 0.5 였다. 능선점 11.5 보다 한참 왼쪽이다. 벡터를 캐시에서 다 밀어낸 뒤로는 옮기는 시간이 전부다.

관찰

> 전수 탐색은 문서마다 $4d$ 바이트를 옮겨 두 번 계산한다. 밀도 0.5 는 이 기계에서 피크의 50 분의 1 이다.

다시 쓰기

이 절은 새 구조를 세우지 않는다. 대신 이 문제를 어떻게 다시 쓸 것인지 지도를 그린다. 전수 탐색이 느린 이유는 둘이고, 둘을 고치는 장이 다르다.

첫째, 모든 벡터를 읽는다. 읽을 벡터를 줄이는 것이 12 장의 근접 그래프와 13 장의 역파일이다. 둘째, 벡터 하나가 512 바이트다. 벡터를 32 바이트로 줄이는 것이 13 장의 곱 양자화다. 둘을 합치면 옮기는 양이 만 배 넘게 줄고, 그 대가로 재현율이 1.0 아래로 내려간다. 얼마나 내려가는지를 정의 0.9 로 잰다.

이 문제만 다시 쓰는 것이 아니다. 시스템 전체를 그림 하나에 그리면 아래와 같다. 회색이 이 절의 상태이고, 파란 상자마다 어느 장이 그것을 다시 쓰는지 적혀 있다.

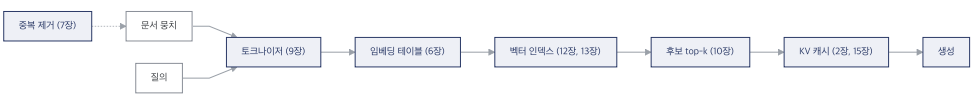


그림 7.. `dsai` 의 부품과 각 부품을 다시 쓰는 장. 이 그림 하나가 이 책의 차례다

부품마다 디딤돌이 다르다. 토크나이저의 디딤돌은 트라이이고, 임베딩 테이블의 디딤돌은 해시 테이블이고, KV 캐시의 디딤돌은 동적 배열이다. 벡터 인덱스의 디딤돌은 둘인데, 스킵 리스트와 k-평균이다. 17장에서 이 부품을 전부 잇고 시스템 하나로 잰다.

구현

전수 탐색과 재현율의 구현이다. 5부의 모든 인덱스가 이 두 함수를 정답과 잣대로 부른다.

지금 보는 코드 정확 탐색과 재현율. 5부의 근사 인덱스가 전부 이 함수를 정답으로 삼는다.

구현 전수 탐색과 재현율 `samples/dsai/knn.py:17`

```python
def knn_bruteforce(xs: np.ndarray, q: np.ndarray, k: int) -> np.ndarray:
    """정확 탐색. n 개 전부와 내적을 구하고 큰 순서로 k 개를 고른다.

    xs 는 (n, d), q 는 (d,). 둘 다 normalize 를 거쳤다고 본다.
    돌려주는 것은 유사도가 큰 순서로 정렬된 인덱스 k 개다.
    """
    scores = xs @ q
    if k >= len(scores):
        return np.argsort(-scores)
    top = np.argpartition(-scores, k)[:k]
    return top[np.argsort(-scores[top])]


def recall_at_k(found: np.ndarray, truth: np.ndarray) -> float:
    """정의 0.5. 찾은 k 개 중 정답 k 개에 든 것의 비율."""
    return len(set(found.tolist()) & set(truth.tolist())) / len(truth)
```

읽어 보면 `argpartition` 은 상위 k 개를 정렬하지 않고 고른다. n 개 전부를 정렬하면 $O(n \log n)$ 인데 이것은 $O(n)$ 이다. 고른 k 개만 정렬하므로 정렬 비용은 $O(k \log k)$ 다. 10장에서 힙으로 같은 일을 다시 본다.

정규화 함수도 여기 둔다. 정규화를 거쳐야 내적이 코사인 유사도가 되고, 정의 0.8 의 거리와 순서가 같아진다. 같은 파일의 `random_corpus` 는 임베딩 자리에 난수 단위 벡터를 놓는 대역이다. 6장에서 진짜 임베딩 테이블로 바꾼다.

구현 단위 벡터로 만들기 `samples/dsai/knn.py:11`

```python
def normalize(x: np.ndarray) -> np.ndarray:
    """각 행을 길이 1 로 만든다. 그러면 내적이 코사인 유사도가 된다."""
    norms = np.linalg.norm(x, axis=-1, keepdims=True)
    return x / np.maximum(norms, 1e-12)
```

분석

전수 탐색이 언제 답인지도 적어 둔다. 새 구조가 항상 이기는 것은 아니다. 문서가 만 개 아래면 전수 탐색이 0.3 밀리초다. 어떤 인덱스도 이보다 빠르지 않고, 인덱스를 만드는 시간은 따로 든다. 문서가 자주 바뀌면 인덱스를 다시 만드는 값이 탐색에서 아낀 값을 넘는다. 그리고 재현율이 1.0 이어야 하는 자리에서는 선택지가 전수 탐색뿐이다.

현장에서

> Johnson, Douze, Jégou 의 2019 년 논문은 GPU 위에서 k 선택을 다시 설계해 십억 벡터 전수 탐색을 다룬다 [확립] [4]. 전수 탐색은 느린 것이 아니라 밀도가 낮은 것이고, 대역폭이 열 배인 기계에서는 열 배 빠르다.

이 절의 실험이 5 부의 출발점이다. 12 장과 13 장은 같은 백만 문서, 같은 차원, 같은 질의로 재고, 지연과 재현율을 이 표 옆에 놓는다. 그때 비교 대상이 되는 것은 이 절의 54 밀리초와 1.0 이다.

되짚기

> k 최근접 탐색과 재현율을 정의했고, 전수 탐색이 밀도 0.5 의 대역폭 병목 연산임을 쟀다. 다음 장부터 `dsai` 의 부품을 하나씩 다시 쓴다. 첫 부품은 가장 밑에 있는 배열이다.

0장을 닫으며

이 장에서 세운 것은 구조가 아니라 잣대다. 비용을 (W, M) 로 세는 이동량 모델, 그 둘의 비로 병목을 읽는 루프라인, 그리고 근사 풀이를 재는 재현율. 뒤의 열일곱 장은 전부 이 세 잣대로 잰다.

세 실험이 보인 것을 한 줄씩 적어 둔다. 같은 원소 수를 같은 횟수 읽어도 순서가 바뀌면 시간이 3.5배 갈렸고, 갈리는 자리는 정확히 데이터가 마지막 캐시를 넘는 자리였다. 같은 곱셈 덧셈이라도 밀도가 0.25 인 내적은 밀도 341 인 행렬 곱에 견줘 50분의 1 처리량에 머물렀다. 그리고 밀도 0.5 인 전수 탐색은 백만 문서에서 초당 열여덟 질의밖에 내지 못했다.

전통 교과서의 첫 가정에 대한 판정을 여기서 낸다. 판정은 셋 중 하나로만 쓴다. 그대로 유효, 조정해서 유효, 이제 안 맞음.

옛 구조의 판정

> 균일 비용 모델은 조정해서 유효하다. 데이터가 마지막 캐시 안에 들면 그대로 맞고, 넘으면 이동량을 따로 세야 한다. 근거는 실험 0.1 이다. 40KB 에서 비율 1.0, 40MB 에서 3.5 였다. 그래서 이 책은 복잡도를 W 와 M 두 개로 적는다.

나머지 세 가정의 판정은 각각을 재는 장에서 낸다. 여기서는 넷이 어디서 깨지는지만 한 쪽에 모은다. 이 표가 이 책의 차례를 읽는 또 하나의 방법이다.

바뀐 가정 지도

번호	전통 교과서의 가정	AI 시대의 작업량	재는 장
1	메모리 접근 비용은 균일하다	캐시, 주기억, 가속기 메모리의 대역폭이 열 배에서 백 배 다르다	0장, 1장, 2장, 15장
2	데이터의 단위는 스칼라나 레코드다	단위가 벡터와 텐서다. 원소 하나가 아니라 수천 차원이 한 번에 움직인다	1장, 3장, 6장
3	답은 정확해야 한다	근사 답을 재현율로 재고, 그 대가로 백 배 빠른 것이 표준이다	7장, 12장, 13장
4	구조는 사람이 설계한다	구조의 일부가 데이터에서 학습되고, 고르는 일만 사람에게 남는다	8장, 17장

가정 넷이 흔들린 자리에서도 전통 구조는 버려지지 않는다. 절마다 그 구조를 디딤돌로 먼저 세우고, 그 위에서 AI 시대의 얼굴을 밝힌다. 어느 구조가 어느 절의 디딤돌인지를 여기 한 쪽에 모은다. 스택과 큐와 연결 리스트처럼 모든 강의가 처음에 가르치는 것부터, 트라이와 접미사 배열처럼 뒤에 가르치는 것까지다.

전통 자료 구조 지도

전통 구조	디딤돌 절	AI 시대의 얼굴	판정 자리
배열	1.1, 1.2	값 배열과 스트라이드 뷰. 텐서의 바탕	1장 끝
동적 배열	2.1	KV 캐시, 페이지드 어텐션의 블록 테이블	2장 끝
희소 행렬	3.1	CSR 과 구조적 희소성, 양자화된 텐서	3장 끝
연결 리스트	4.1	자유 리스트와 블록 풀, 배열 안의 인덱스 링크	4장 끝
스택	5.1	명시적 스택, 역방향 자동 미분의 테이프	5장 끝
큐와 덱	5.3, 5.4	링 버퍼의 슬라이딩 윈도우, 배칭 큐와 연속 배칭	5장 끝
해시 테이블	6.1	어휘 사전, 해싱 트릭과 임베딩 테이블, 가속기 위의 병렬 삽입	6장 끝
집합과 개수 세기	7.1, 7.2	블룸 필터, 카운트 민 스케치, MinHash. 근사 집합	7장 끝
이진 탐색 트리	8.1	B-트리, 학습된 인덱스, 정렬된 배열의 분기 없는 탐색	8장 끝
트라이	9.1	토크나이저의 병합, 접두사 공유 캐시, 제약 디코딩	9장 끝
힙과 우선순위 큐	10.1	top-k 선택, 빔 탐색, 별칭표, 요청 스케줄러	10장 끝
그래프와 인접 리스트	11.1	CSR 그래프, 위상 정렬과 자동 미분, 연산자 융합	11장 끝
kd-트리와 스킵 리스트	12.1, 12.2	근접 그래프 NSW 와 HNSW, 디스크 위의 그래프	12장 끝
벡터 양자화	13.1, 13.2	역파일과 곱 양자화. 벡터 인덱스	13장 끝
접미사 배열	14.1	*n* 없는 n-그램, 반복 구간 찾기	14장 끝
연관 배열과 LRU 캐시	15.1, 15.3	어텐션, KV 캐시 축출, 고정 크기 상태	15장 끝
정렬과 순열, 머클 트리	16.2, 16.4	내용 주소 저장, 열 저장, 셔플과 형식 보존 순열	16장 끝

표의 셋째 열이 이 책의 차례이고, 넷째 열의 판정을 17장 끝의 성적표가 한 쪽에 모은다.

이 장의 요약

1. 자료 구조는 상태, 연산, 불변식, 비용 모델의 묶음이고, 비용 모델이 바뀌면 답이 바뀐다 0.1
2. 데이터가 캐시를 넘으면 균일 비용 모델이 틀리고, 이동량 M 를 따로 세야 한다 0.1
3. 연산 밀도 $I = W/M$ 가 능선점보다 낮으면 대역폭에 막힌다 0.2
4. 전통 교과서의 가정 넷은 서로 이어져 있고, 첫째가 깨지면 나머지가 따라 깨진다 0.2

5. k 최근접 탐색의 전수 풀이는 밀도 0.5 의 대역폭 병목 연산이다 0.3
다음 장에서 뒤집히는 것: 배열은 그대로 유효하지만, 배열의 원소가 스칼라라는 가정은 안 맞는다.

필자의 판단

필자는 이동량 모델을 캐시에 갖다 대는 것이 디스크에 갖다 대는 것보다 낫다고 본다. 디스크 시대의 블록은 4KB 였고 접근 시간의 차이가 만 배였다. 캐시 라인은 64바이트이고 차이는 백 배다. 차이가 작아서 모델이 덜 맞을 것 같지만, 반대다. 만 배 차이는 누구나 알아서 피하고, 백 배 차이는 모르고 지나간다. 그래서 캐시 쪽의 모델이 더 자주 쓸모가 있다.

열린 문제

하나, 이 장의 루프라인은 주기억 대역폭 하나만 그렸다. L2 와 L3 의 지붕을 같이 그리면 행렬 벡터 곱의 자리가 더 정확해진다. 이 책은 15장에서야 그 지붕을 그린다.
둘, 정리 0.1 의 무작위 접근 하한은 캐시가 이상적이라고 가정했다. 실제 캐시의 연관도와 교체 정책에서 상수가 얼마나 달라지는지는 재지 않았다.
셋, 가속기 메모리의 대역폭은 이 책의 기계에 없어서 재지 못했다. 공개 사양으로만 적는다.

연습문제

0.1 기초 정의 0.1 의 넷 (S, Ops, Inv, C) 를 큐에 대해 적어라. 불변식을 한 문장으로 쓰고, 균일 비용 모델에서 enqueue 와 dequeue 의 값을 세어라.

0.2 기초 원소 $n = 10^6$ 개, 원소 크기 8바이트, 라인 64바이트, 캐시 8MB 일 때 정리 0.1 의 두 이동량을 계산하라. 순차와 무작위의 비는 얼마인가.

0.3 기초 피크 200 GFLOP/s, 대역폭 20 GB/s 인 기계의 능선점을 구하고, 밀도 0.25, 2, 50 인 연산의 달성 가능 처리량을 각각 적어라.

0.4 기초 예제 0.3 의 점 다섯 개를 단위 벡터로 만든 뒤 $q = (1, 0)$ 에 대해 내적으로 $k = 2$ 를 풀어라. 유클리드 거리로 푼 답과 같은지 확인하라.

0.5 응용 `bench/c00_cost.py` 의 원소 크기를 float64 로 바꿔 다시 재라. 비율이 갈라지는 원소 수가 절반이 되는지 확인하고, 그 이유를 정리 0.1 로 설명하라.

0.6 응용 `dsai.cost.blocks_touched` 로 실험 0.1 의 각 행에서 두 순서가 건드리는 라인 수를 세고, 잰 시간의 비율과 라인 수의 비율을 한 표에 놓아라.

0.7 응용 `dsai.roofline` 에 원소별 곱 (두 벡터의 같은 자리끼리 곱해 새 벡터를 만드는 연산) 의 W 와 M 함수를 더하고, 밀도를 구해 실험 0.2 의 표에 한 줄 더하라.

0.8 응용 `dsai.knn.knn_bruteforce` 에 질의 여러 개를 한 번에 받는 판을 만들어라. 질의 64개를 한 번에 처리할 때 질의당 지연이 얼마나 줄고 밀도가 얼마가 되는지 재라.

0.9 탐구 정리 0.1 은 캐시가 라인을 완벽히 골라 남긴다고 가정한다. LRU 교체 정책에서 무작위 순열 접근의 기대 이동량이 이 하한과 얼마나 다른지 논하라. 시뮬레이션으로 재도 좋다.

0.10 탐구 루프라인은 이동량을 주기억 하나에서만 센다. 캐시 층마다 지붕을 하나씩 그린 다층 루프라인에서 실험 0.2 의 행렬 벡터 곱이 어느 지붕에 붙는지 추정하라.

더 읽을 것

[2] 루프라인 모델의 원전. 여덟 쪽이고 그림이 전부다. 이 장의 실험 0.2 를 읽고 나서 보면 한 시간에 끝난다. 0.2
[1] 이동량 모델의 원전. 디스크 시대의 논문이지만 정리의 형식은 그대로 캐시에 옮겨진다. 0.1
[5] 라인 크기를 모르고도 최적인 알고리즘. 15장의 타일링이 이 논문 위에 서 있다. 0.1
[3] 범용 프로세서의 성능 향상이 왜 멈췄는지. 이 책의 주제가 왜 지금 문제인지를 두 저자가 적었다. 0.2

1부

연속 메모리

교과서의 배열은 원소가 스칼라였다. 이 부는 원소가 벡터가 된 뒤의 배열을 세운다. 값 배열, 스트라이드, 블록 순회, 배치, 그리고 배열이 자라야 할 때와 비어 있을 때의 구조까지.

1장 배열에서 텐서로 2장 동적 배열에서 페이지 테이블로 3장 희소 행렬과 저정밀 표현

1장

배열에서 텐서로

NumPy 의 설계를 정리한 2020 년 논문은 배열 객체 하나가 중력파 검출과 블랙홀 첫 영상의 계산을 받쳤다고 적었다 [확립] [6]. 그 배열 객체는 교과서의 배열에 세 가지를 더한 것이다. 원소의 타입, 축마다의 스트라이드, 그리고 모양이 다른 배열끼리의 연산 규칙. 이 셋이 배열을 텐서로 만들고, 텐서가 이 책의 모든 구조의 바닥이 된다.

이 장은 0.2 의 둘째 가정을 건드린다. 데이터의 단위가 스칼라라는 가정이다. 단위가 벡터가 되면 배열의 원소가 벡터이고, 벡터의 원소는 다시 값이어야 한다. 원소가 참조이면 첫째 가정이 다시 깨진다. 그래서 이 장은 배열의 연속성에서 시작해서, 그 연속성을 여러 축으로 펴는 스트라이드, 그 축을 캐시에 맞게 훑는 블록 순회, 그리고 축을 하나 더해 밀도를 올리는 배치까지 간다.

이 장에서 세우는 것

1.1 배열을 연속성 불변식으로 정의하고, 원소가 참조일 때 그 불변식이 값에 대해 깨지는 것을 잰다
1.2 스트라이드를 정의에 넣어 전치와 자르기를 이동량 0 의 연산으로 만든다
1.3 간격 접근의 이동량을 정리로 세우고, 블록 순회로 전치를 열 배 넘게 줄인다
1.4 배치가 연산 밀도를 올리는 산수를 명제로 세우고, 브로드캐스팅을 정의한다

1.1 배열의 불변식과 주소 계산

0.1 에서 이동량 M 를 비용 모델에 넣었다. 그 모델에서 가장 좋은 점수를 받는 구조가 배열이다. 원소가 붙어 있으니 라인 하나를 가져오면 그 안의 원소를 전부 쓴다. 이 절은 그 성질을 불변식으로 적고, 그 불변식이 깨지는 가장 흔한 자리를 잰다.

깨지는 자리는 뜻밖에도 배열 안이다. 원소가 값이 아니라 다른 곳을 가리키는 참조일 때다. 파이썬 리스트가 그렇고, 대부분의 동적 언어의 배열이 그렇다. 겉으로는 배열인데 이동량은 연결 리스트에 가깝다.

학습 목표

배열을 주소 계산식과 연속성 불변식으로 정의할 수 있다. 값 배열과 참조 배열의 이동량 차이를 세고, 실측으로 확인할 수 있다.

이 말부터

주소 계산	인덱스에서 바이트 위치를 구하는 식. 배열에서는 곱셈 하나와 덧셈 하나다
값 배열	원소의 값 자체가 연속으로 놓인 배열. NumPy 의 `ndarray` 가 이것이다
참조 배열	원소 자리에 값이 아니라 값이 있는 곳의 주소가 놓인 배열. 파이썬 `list` 가 이것이다

디딤돌

배열은 정의할 것이 없어 보이는 구조다. 그래서 정의를 세워 둔다. 뒤의 텐서가 이 정의에 항목을 하나씩 얹기 때문이다.

정의 1.1: 배열

배열은 같은 크기 s 바이트의 원소 n 개를 시작 주소 a_0 부터 빈틈없이 놓은 것이다. 연산은 읽기 $A[i]$ 와 쓰기 $A[i] \leftarrow v$ 둘이고, 인덱스 i 의 주소는 $\mathrm{addr}(i) = a_0 + i \cdot s$ 다.

불변식 모든 $0 \le i < n - 1$ 에 대해 $\mathrm{addr}(i+1) - \mathrm{addr}(i) = s$ 다. 이것을 연속성이라 부른다.

연속성 불변식이 배열의 전부다. 이 불변식 덕에 주소 계산이 곱셈 하나와 덧셈 하나로 끝나고, 같은 불변식 덕에 순차 순회가 라인 하나당 B/s 개 원소를 쓴다. 0.1 의 정리는 이 불변식이 성립할 때만 순차 접근의 이동량이 $\lceil ns/B \rceil$ 이라고 말한 것이다.

예제 1.1: 주소 계산

> float32 배열이 주소 4096 에서 시작한다고 하자. $s = 4$ 다. $A[10]$ 의 주소는 $4096 + 10 \times 4 = 4136$ 이고, $A[11]$ 은 4140 이다. 둘의 차이가 s 이니 불변식이 맞는다. 라인이 64 바이트이면 $A[0]$ 부터 $A[15]$ 까지가 라인 하나에 든다. $4096/64 = 64$ 번째 라인이다.

이 정의에서 원소의 크기 s 가 고정이라는 것이 눈에 띄지 않게 중요하다. s 가 원소마다 다르면 주소 계산이 곱셈으로 끝나지 않고, 앞의 원소를 전부 더해야 한다. 그래서 크기가 다른 것을 배열에 넣고 싶을 때 대부분의 언어가 택하는 길은 하나다. 원소 자리에 주소를 넣는다.

균열

파이썬 리스트에 실수 백만 개를 넣으면 리스트 안에 실수가 들어 있는 것이 아니다. 리스트 안에는 8 바이트 주소가 백만 개 들어 있고, 실수 객체는 힙 어딘가에 하나씩 따로 있다. 정의 1.1 의 연속성은 주소 배열에는 성립하고 값에는 성립하지 않는다.

같은 실수 백만 개를 세 가지로 담아 합산했다. 파이썬 리스트, 원소가 객체인 NumPy 배열, 그리고 float32 값 배열이다.

실험 1.1 같은 숫자 백만 개, 세 가지 담는 법

담는 법	크기 MB	합산 ms
파이썬 리스트	32.0	3.54
object 배열	32.0	14.0
float32 배열	4.00	0.30

측정 조건. 원소 백만 개. 리스트와 object 배열의 크기는 포인터 8 바이트와 객체 24 바이트를 합한 것이다. 일곱 번 중앙값.

크기가 여덟 배이고 시간은 열 배가 넘는다. 원소 하나를 읽을 때 주소를 먼저 읽고 그 주소로 한 번 더 가기 때문이다. 두 번째 접근은 힙 어디든 갈 수 있으므로 0.1 의 무작위 접근과 같은 값을 치른다. object 배열이 리스트보다도 느린 것은 NumPy 가 객체를 하나씩 파이썬 함수로 더하기 때문이다.

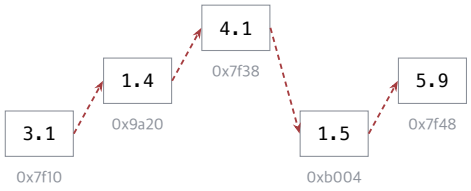


그림 1.1. 참조 배열. 배열 자체는 연속이지만 값은 힙에 흩어져 있다. 원소마다 라인 하나가 더 든다

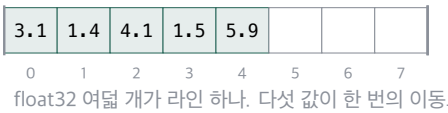


그림 1.2.. 값 배열. 같은 다섯 값이 한 라인 안에 붙어 있다. 라인 하나로 다섯 개를 다 쓴다

관찰

> 참조 배열의 이동량은 원소 수에 비례하고, 값 배열의 이동량은 바이트 수를 라인 폭으로 나눈 것에 비례한다. 열여섯 배 차이다.

다시 쓰기

값 배열이 새 구조는 아니다. C 의 배열이 처음부터 값 배열이었다. 다시 쓰는 것은 정의 1.1 에 항목 하나를 더하는 일이다. 원소의 타입이다.

정의 1.2: 값 배열

> 값 배열은 배열에 원소 타입 τ 를 붙인 것이다. 타입이 원소의 크기 $s = \mathrm{size}(\tau)$ 와 해석 방법을 정한다. 모든 원소가 같은 타입이므로 값이 그 자리에 놓이고, 연속성 불변식이 값에 대해 성립한다.
>
> **불변식** 원소 i 의 값은 $\mathrm{addr}(i)$ 부터 s 바이트를 타입 τ 로 읽은 것이다. 다른 곳을 가리키지 않는다.

이 정의에서 타입 τ 가 이 책 전체를 따라온다. 3장에서 τ 가 4바이트에서 1바이트로 줄고, 그것이 양자화다. 15장에서 τ 가 float16 이 되고 그 대가로 정밀도를 잃는다. 타입은 값 배열의 일부이지 배열 밖의 설정이 아니다.

두 배열의 이동량을 나란히 적으면 아래 명제가 된다.

명제 1.3: 값 배열과 참조 배열의 순회 이동량

> 원소 n 개를 순서대로 한 번씩 읽는다. 값 배열의 이동량은 $\lceil ns/B \rceil$ 라인이다. 참조 배열은 주소 배열을 읽는 $\lceil 8n/B \rceil$ 라인에 더해, 값이 캐시에 없으면 원소마다 라인 하나가 든다. 값이 힙에 흩어져 있고 전체가 Z 를 넘으면 참조 배열의 이동량은 n 에 가깝다.

증명. 값 배열은 정의 1.2 의 연속성으로 0.1 의 정리가 그대로 적용된다. 참조 배열의 주소 배열은 8바이트 원소의 값 배열이므로 같은 식이 적용된다. 값 하나를 읽는 것은 주소가 가리키는 라인을 하나 가져오는 것이고, 값들이 라인 하나에 둘 이상 들어 있을 보장이 없으므로 원소마다 하나씩 센다. ■

실험 1.1 의 세 행 중 첫째와 셋째가 이 명제의 두 경우다. 시간 비율 열 배는 라인 비율 열여섯 배에 상수가 곱해진 것이다.

구현

값 배열의 주소 계산을 코드로 둔다. NumPy 가 하는 일을 손으로 다시 쓴 것이다. 이 함수가 다음 절에서 스트라이드를 받으면 텐서가 된다.

지금 보는 코드 인덱스와 스트라이드로 바이트 위치를 구한다. 1차원이면 정의 1.1 의 $i \cdot s$ 다.

구현 주소 계산과 행 우선 스트라이드 `samples/dsai/tensor.py:14`

```
def offset(index: Sequence[int], strides: Sequence[int]) -> int:
    """정의 1.2 의 주소 계산. 시작 위치에서 몇 바이트 떨어져 있는가."""
    return sum(i * s for i, s in zip(index, strides))


def row_major_strides(shape: Sequence[int], itemsize: int) -> tuple[int, ...]:
    """마지막 축이 가장 촘촘하다. C 와 NumPy 의 기본이다."""
    strides = []
    acc = itemsize
    for dim in reversed(shape):
        strides.append(acc)
        acc *= dim
    return tuple(reversed(strides))
```

읽어 보면 `offset` 은 인덱스와 스트라이드를 짝지어 곱해 더한다. 1차원에서는 곱셈 하나다. `row_major_strides` 는 마지막 축의 간격을 원소 크기로 놓고 앞 축으로 갈수록 곱해 간다. 다음 절의 재료다.

알고리즘 1.1 주소 계산 `dsai.tensor.offset`

입력: 인덱스 $(i_1, \dots, i_k)$, 스트라이드 $(s_1, \dots, s_k)$
출력: 시작 주소로부터의 바이트 거리

$o \leftarrow 0$
반복 $j = 1$ **부터** k **동안**
    $o \leftarrow o + i_j \cdot s_j$
끝 반복
반환 o

분석

배열의 판정은 이 책에서 가장 쉽다. 그대로 유효하다. 연속성 불변식은 40년 전과 같은 이유로 지금도 가장 싼 이동량을 준다. 바뀐 것은 배열이 아니라 배열의 원소다. 교과서의 배열 원소는 정수 하나였고, 이 책의 배열 원소는 임베딩 벡터 하나다.

원소가 벡터가 되면 값 배열과 참조 배열의 차이가 더 커진다. 벡터 백만 개를 리스트에 담으면 백만 번의 무작위 접근이고, 값 배열에 담으면 nds/B 번의 순차 접근이다. 0.3 의 전수 탐색이 초당 2천만 벡터를 낸 것은 벡터가 값 배열에 있었기 때문이다.

현장에서

> NumPy 의 `ndarray` 는 값 배열에 모양과 스트라이드를 붙인 것이다. Harris 와 동료들이 2020 년에 그 설계를 정리했고, 중력파 검출과 블랙홀 영상 처리가 이 배열 위에서 돌았다고 적었다 [확립] [6].

되짚기

> 배열을 연속성 불변식으로 정의했고, 원소가 참조이면 그 불변식이 값에 대해 깨진다는 것을 쟀다. 값 배열은 원소 타입을 정의에 넣는다. 다음 절에서 이 값 배열에 스트라이드를 얹어 복사 없는 전치를 만든다.

1.2 스트라이드와 뷰

1.1 의 주소 계산은 1 차원이었다. 행렬과 텐서는 축이 둘 이상이고, 축마다 간격이 다르다. 이 절은 그 간격을 스트라이드라는 이름으로 배열 정의에 넣는다. 그러면 전치와 자르기가 복사 없는 연산이 된다.

복사가 없다는 것은 이동량이 0 이라는 뜻이다. 0.1 의 비용 모델에서 이보다 싼 연산은 없다. 그런데 공짜에는 대가가 있다. 뷰를 훑는 순서가 메모리 순서와 어긋나면 라인당 원소 하나만 쓰게 된다. 이 절은 공짜인 자리와 대가를 치르는 자리를 갈라 잰다.

학습 목표

> 텐서 뷰를 버퍼, 모양, 스트라이드, 시작 위치 넷으로 정의할 수 있다. 전치와 자르기가 왜 상수 비용인지, 그리고 그 뷰를 옮길 때 이동량이 왜 열여섯 배가 되는지 설명할 수 있다.

이 말부터

스트라이드	어느 축에서 인덱스가 하나 늘 때 주소가 몇 바이트 움직이는가. 축마다 하나씩이다
뷰	버퍼를 새로 만들지 않고 읽는 규칙만 바꾼 것. 원본과 메모리를 공유한다
행 우선	마지막 축이 가장 촘촘한 배치. C 와 NumPy 의 기본이고, 열 우선은 그 반대다

디딤돌

2 차원 배열을 1 차원 메모리에 놓는 방법은 둘이다. 행을 이어 붙이거나 열을 이어 붙이거나. 어느 쪽이든 원소 (i, j) 의 주소는 $a_0 + is_0 + js_1$ 꼴이고, 다른 것은 s_0 과 s_1 의 값뿐이다. 이 두 수를 배열의 정의에 넣으면 정의 1.2 가 텐서로 확장된다.

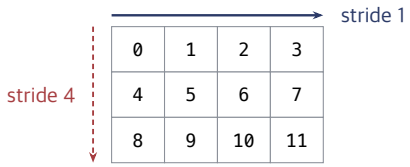
정의 1.4: 텐서 뷰

> 텐서 뷰는 넷의 묶음 $(B, shape, strides, o)$ 다. B 는 값 배열인 버퍼, $shape = (n_1, \ldots, n_k)$ 는 축마다의 크기, $strides = (s_1, \ldots, s_k)$ 는 축마다의 바이트 간격, o 는 시작 위치다. 인덱스 $(i_1, \ldots, i_k)$ 의 주소는 $o + \sum_j i_j s_j$ 다.
>
> **불변식** 모든 유효한 인덱스의 주소가 B 의 안에 있다. 스트라이드에는 다른 제약이 없다. 0 이어도 되고 음수여도 된다.

스트라이드에 제약이 없다는 것이 이 정의의 핵심이다. 행 우선 배치는 스트라이드가 $(n_2 s, s)$ 인 특별한 경우이고, 전치는 두 수를 맞바꾼 것이고, 두 칸마다 하나씩 보는 것은 스트라이드를 두 배로 한 것이다. 버퍼는 그대로이고 넷 중 셋만 바뀐다.

예제 1.2: 3 × 4 행렬의 스트라이드

float32 행렬 3×4 를 행 우선으로 놓으면 스트라이드는 $(16, 4)$ 다. 한 행이 16바이트이기 때문이다. $(1, 2)$ 의 주소는 $16 + 8 = 24$ 이고, 이것은 평평한 버퍼의 여섯 번째 원소다. 전치하면 모양이 $(4, 3)$, 스트라이드가 $(4, 16)$ 이 된다. 전치된 행렬의 $(2, 1)$ 은 주소 $8 + 16 = 24$ 로 같은 원소다.



칸 안의 숫자는 버퍼 안의 순서. 행 방향이 촘촘하다

그림 1.3.. 3 × 4 행 우선 배치. 오른쪽으로 가면 4바이트, 아래로 가면 16바이트 움직인다

전치가 버퍼를 건드리지 않는다는 것을 정리로 적어 둔다. 뒤의 여러 장이 이 정리에 기댄다.

정리 1.5: 뷰 연산의 비용

텐서 뷰의 전치, 축 교환, 간격 자르기, 부분 범위 자르기는 모양과 스트라이드와 시작 위치만 바꾼다. 따라서 연산 수는 축의 수 k 에만 비례하고, 이동량은 0이다.

증명. 전치와 축 교환은 *shape* 와 *strides* 의 원소를 자리바꿈한다. 간격 m 자르기는 그 축의 크기를 $\lceil n_j / m \rceil$ 로, 스트라이드를 $m s_j$ 로 바꾼다. 범위 자르기는 시작 위치 o 에 $i s_j$ 를 더하고 크기를 줄인다. 어느 경우도 B 를 읽거나 쓰지 않으므로 이동량이 0이고, 바꾸는 수는 k 개 이하다. ■

균열

정리 1.5 는 뷰를 만드는 값이다. 뷰를 쓰는 값은 다르다. 전치한 뷰를 순서대로 훑으면 원소마다 16바이트씩 건너뛰고, 행이 길면 원소마다 라인 하나씩을 건너뛴다. 같은 256MB 행렬로 셋을 쟀다. 전치 뷰를 만드는 시간, 연속 복사, 전치 뷰를 연속 메모리로 옮기는 복사다.

실험 1.2 뷰는 공짜이고, 뷰를 옮기는 값은 순서가 정한다

연산	옮긴 MB	시간 ms	대역폭
전치 뷰 만들기	0.00	0.00	0.00
연속 복사	268	42.6	12.6
전치 복사	268	767	0.70
블록 전치 16	268	544	0.99
블록 전치 64	268	200	2.68
블록 전치 256	268	202	2.65
블록 전치 1024	268	229	2.35

측정 조건. float32 8192 × 8192. 출력 버퍼를 미리 잡아 두고 잰다. 블록 전치는 다음 절에서 다룬다. 다섯 번 중앙값.

첫 행이 정리 1.5 다. 256MB 행렬을 전치하는 데 1마이크로초가 안 든다. 둘째 행과 셋째 행이 대가다. 같은 바이트를 같은 곳으로 옮기는데 순서만 다른 두 복사가 열여덟 배 차이 난다. 전치 뷰를 훑으면 읽기가 열 방향으로 가고, 열 방향은 원소마다 8192 × 4 바이트씩 건너뛴다.

관찰

뷰는 만드는 값이 0이고 쓰는 값은 순서에 달렸다. 뷰가 공짜라는 말은 절반만 맞다.

다시 쓰기

이 절의 다시 쓰기는 구조가 아니라 판단이다. 뷰로 둘 것인가 옮길 것인가. 답은 그 뷰를 몇 번, 어느 순서로 훑느냐에 달렸다.

한 번만 훑고 버린다면 뷰로 둔다. 옮기는 데 드는 이동량이 훑는 이동량과 같거나 크기 때문이다. 여러 번 훑고, 훑는 순서가 마지막 축과 어긋난다면 한 번 옮기고 여러 번 싸게 읽는다. 옮길 때는 다음 절의 블록 단위로 옮긴다. 실험 1.2 의 셋째 행처럼 통째로 옮기면 옮기는 것 자체가 열여덟 배다.

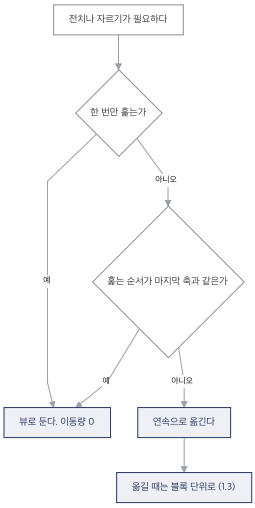


그림 1.4.. 뷰로 둘 것인가 옮길 것인가. 훑는 횟수와 순서가 정한다

이 판단은 15장에서 한 번 더 나온다. 어텐션의 키 행렬을 전치해서 쓰는데, 그 전치를 매번 뷰로 둘지 저장할 때 미리 옮겨 둘지가 KV 캐시 배치의 문제다. 그때 이 그림을 다시 편다.

구현

정의 1.4 를 그대로 코드로 둔다. 버퍼를 갖지 않는 뷰 객체다. 전치와 간격 자르기가 필드 몇 개를 바꾸는 것으로 끝난다는 것이 코드에서 보인다.

지금 보는 코드 버퍼 없이 읽는 규칙만 든 뷰. 전치와 간격 자르기가 상수 시간이다.

구현 뷰와 전치와 간격 자르기 `samples/dsai/tensor.py:57`

```
class View:
    """버퍼 위를 읽는 규칙. 버퍼 자체는 갖지 않는다."""

    shape: tuple[int, ...]
    strides: tuple[int, ...]
    offset: int = 0

    def transpose(self) -> "View":
        """축을 뒤집는다. 버퍼를 한 바이트도 옮기지 않는다."""
        return View(self.shape[::-1], self.strides[::-1], self.offset)

    def step(self, k: int) -> "View":
        """첫 축을 k 개마다 하나씩 본다. 역시 규칙만 바뀐다."""
        n = (self.shape[0] + k - 1) // k
        return View((n,) + self.shape[1:], (self.strides[0] * k,) + self.strides[1:],
                    self.offset)

    def address(self, index: Sequence[int]) -> int:
        return self.offset + offset(index, self.strides)
```

읽어 보면 `transpose` 는 두 튜플을 뒤집고, `step` 은 첫 축의 크기를 줄이고 스트라이드를 k 배 한다. 둘 다 새 `View` 를 만들 뿐 배열을 읽지 않는다. `address` 가 정의 1.4 의 주소 식이다.

NumPy 배열이 든 스트라이드를 그대로 읽어 오는 함수도 두었다. 실험 1.2 에서 NumPy 가 만든 전치 뷰와 이 `View` 의 스트라이드가 같은지를 시험이 확인한다.

역사 한 토막

> NumPy 는 2005 년에 Numeric 과 Numarray 두 라이브러리를 합치며 만들어졌고, 스트라이드 뷰는 그때부터 배열 객체의 일부였다 [확립] [6].

분석

스트라이드 뷰는 그대로 유효하다. 다만 뷰가 공짜라는 직관은 조정해야 한다. 정리 1.5 가 약속하는 것은 만드는 값이고, 실험 1.2 가 보인 것은 쓰는 값이다. 둘을 섞으면 전치 뷰를 반복문 안에서 훑는 코드가 나오고, 그 코드는 정확하지만 열여덟 배 느리다.

이 책의 여러 구조가 이 판단 위에 선다. 3 장의 CSR 은 열 방향 접근을 피하려고 행렬을 다시 놓은 것이고, 16 장의 열 지향 저장은 같은 판단을 디스크에 적용한 것이다. 둘 다 훑는 순서를 먼저 정하고 배치를 거기에 맞춘다.

주의

> `a.T` 는 상수 시간이지만 `a.T.copy()` 나 `a.T.sum()` 은 아니다. 뷰를 만든 것과 뷰를 읽은 것을 구분한다.

되짚기

> 텐서 뷰를 버퍼, 모양, 스트라이드, 시작 위치로 정의했고, 뷰 연산의 이동량이 0 임을 증명했다. 뷰를 옮기는 값은 훑는 순서가 정하고, 다음 절에서 그 순서를 라인에 맞추는 블록 순회를 세운다.

1.3 메모리 계층과 접근 순서

1.2 에서 전치 뷰를 옮기는 복사가 연속 복사보다 열여덟 배 느린 것을 봤다. 이 절은 그 열여덟 배가 어디서 오는지를 간격 하나씩 바꿔 가며 재고, 그 값을 다시 연속 복사에 가깝게 끌어내리는 순회 순서를 세운다. 블록 순회, 또는 타일링이다.

이 절이 세우는 것은 구조가 아니라 순회 순서다. 자료 구조 책에서 순서를 다루는 이유는 하나다. 같은 구조라도 순서가 라인과 어긋나면 이동량이 열여섯 배가 되고, 열여섯 배는 구조를 바꿔서 얻는 이득보다 크다.

학습 목표

> 간격 k 접근의 이동량을 정리로 세우고, 실측에서 꺾이는 자리를 라인 폭으로 설명할 수 있다. 블록 순회의 이동량을 계산하고, 블록 크기를 캐시 용량에서 거꾸로 구할 수 있다.

이 말부터

> 간격 접근 — 원소를 k 개마다 하나씩 읽는 것. 스트라이드가 ks 인 뷰를 훑는 것과 같다
>
> 블록 순회 — 큰 배열을 캐시에 드는 작은 사각형으로 잘라 사각형 하나씩 처리하는 순회. 타일링이라고도 한다
>
> 선인출 — 프로세서가 다음에 읽을 라인을 미리 가져오는 것. 순차 접근에서만 잘 맞춘다

디딤돌

0.1 의 이동량 모델은 순차와 무작위 두 극단만 셌다. 그 사이를 세운다. 원소를 k 개마다 하나씩 읽으면 라인 하나에서 쓰는 원소가 $B/(ks)$ 개로 줄고, ks 가 B 을 넘으면 하나가 된다.

정리 1.6: 간격 접근의 이동량

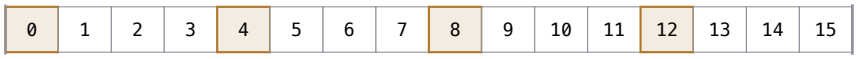
> 원소 크기 s 인 값 배열에서 원소를 k 개마다 하나씩, 모두 m 개 읽는다. 배열이 Z 보다 크면 이동량은
>
> $$M(k) = \left\lceil m \cdot \min\left(1,\ \frac{ks}{B}\right) \right\rceil$$
>
> 라인이다. $ks \geq B$ 이면 원소마다 라인 하나이고, 그 아래에서는 k 에 비례해 오른다.

증명. 연속한 두 접근의 주소 차이가 ks 다. $ks < B$ 이면 라인 하나 안에 접근이 $B/(ks)$ 개 들어가므로 라인 수는 mks/B 이다. $ks \geq B$ 이면 접근마다 다른 라인이고, 배열이 캐시보다 크므로 앞서 가져온 라인이 다시 쓰이기 전에 밀려난다. 따라서 m 이다. ■

예제 1.3: float32 와 간격 4

> $s = 4, B = 64, k = 4$ 이면 $ks = 16$ 이고 라인 하나에 접근 네 개가 든다. 백만 개를 읽으면 라인 25만 개다. $k = 16$ 이면 $ks = 64$ 로 라인 하나에 접근 하나이고, 백만 개를 읽으면 라인 백만 개다.

이 정리의 그림이 아래다. 간격 4 는 라인마다 네 칸을 쓰고, 간격 16 은 라인마다 한 칸을 쓴다.



라인 폭 16 칸. 읽는 칸이 넷이고 라인은 하나다

그림 1.5.. 간격 4 로 읽을 때. 라인 하나에서 네 칸을 쓴다

균열

정리를 이 책의 기계에서 쟀다. 256MB 배열을 간격 1 부터 128 까지 바꿔 가며 읽어 더했다. 읽는 원소 수는 간격에 반비례해 줄어드니, 비교할 것은 원소 하나를 읽는 데 든 시간이다.

실험 1.3 간격을 바꿔 가며 훑기

간격	읽은 원소	시간 ms	원소당 ns	라인당 원소
1	67108864	29.0	0.43	16.0
2	33554432	25.2	0.75	8.00
4	16777216	30.8	1.84	4.00
8	8388608	28.8	3.43	2.00
16	4194304	30.1	7.18	1.00
32	2097152	20.8	9.93	1.00
64	1048576	14.9	14.2	1.00
128	524288	8.92	17.0	1.00

측정 조건. float32 6,400만 개 (256MB), k 개마다 하나씩 읽어 더한다. 마지막 열은 정리 1.3 이 예측한 라인당 원소 수. 일곱 번 중앙값.

간격 1 에서 16 까지 원소당 시간이 열여섯 배 오른다. 정리 1.6 가 예측한 그대로다. 라인당 원소가 16 에서 1 로 줄었으니 원소당 이동량이 열여섯 배가 된 것이다.

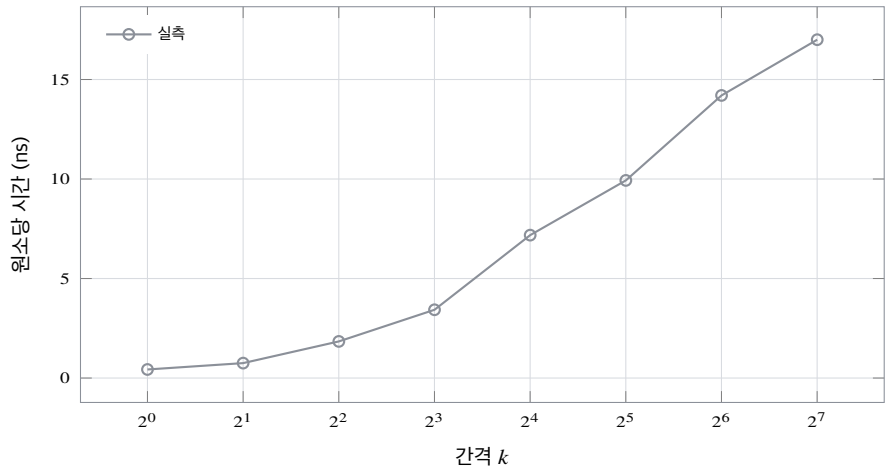


그림 1.6.. 간격에 따른 원소당 시간. 16 까지 정리대로 오르고, 그 뒤로도 완만하게 오른다

예상과 다른 것이 하나 있다. 정리는 간격 16 부터 원소당 시간이 멈춘다고 말하는데, 실측은 계속 오른다. 간격 128 에서는 16 일 때의 두 배가 넘는다. 이 절의 모델이 못 보는 것이 둘 있다. 프로세서가 다음 라인을 미리 가져오는 선인출은 간격이 커지면 맞추지 못하고, 간격이 페이지 크기에 가까워지면 주소 변환 자체가 캐시를 놓친다. 둘 다 이 책의 모델 밖이고, 장 끝의 열린 문제에 적어 둔다.

관찰

> 모델은 열여섯 배를 맞혔고 그 뒤의 두 배는 못 맞혔다. 모델이 맞히는 범위를 아는 것이 모델을 아는 것이다.

다시 쓰기

간격 접근이 비싼 이유를 알았으니 피하는 법을 세운다. 전치가 대표적인 문제다. 행 방향으로 읽어 열 방향으로 쓰거나 그 반대이므로, 읽기와 쓰기 중 하나는 반드시 간격 접근이다.

블록 순회는 큰 행렬을 $b \times b$ 사각형으로 자르고 사각형 하나를 통째로 처리한다. 사각형 하나의 행이 b 개이고 각 행이 bs 바이트이므로, $bs \geq B$ 이면 라인마다 원소를 전부 쓴다. 사각형 하나가 캐시에 들어가면 열 방향으로 읽어도 라인이 밀려나지 않는다.

정의 1.7: 블록 순회

> $n \times m$ 행렬을 $b \times b$ 블록 $\lceil n/b \rceil \times \lceil m/b \rceil$ 개로 나눈다. 블록 순회는 블록을 하나씩 골라 그 안의 원소를 전부 처리한 뒤 다음 블록으로 넘어간다.
>
> **불변식** 처리 중인 블록의 입력과 출력이 함께 캐시에 든다. 즉 $2b^2 s \leq Z$ 다.

정리 1.8: 블록 전치의 이동량

> $n \times n$ float 행렬을 정의 1.7 의 불변식을 만족하는 b 로 블록 전치하면, 이동량은
>
> $$M \le 2 \cdot \frac{n^2 s}{B} + 2 \cdot \frac{n^2}{b}$$
>
> 라인이다. 통째로 전치 복사하면 이동량은 $n^2 s/B + n^2$ 라인이다.

증명. 블록 하나를 읽으면 행 b 개가 각각 $\lceil bs/B \rceil \le bs/B + 1$ 라인이므로 블록당 $b^2 s/B + b$ 라인이다. 쓰기도 같다. 블록이 n^2/b^2 개이므로 전체는 $2(n^2 s/B + n^2/b)$ 이다. 통째로 전치 복사하면 읽기는 순차라 $n^2 s/B$ 이고, 쓰기는 열 방향이라 정리 1.6 의 $ks \ge B$ 경우가 되어 원소마다 라인 하나, 즉 n^2 이다. ■

$b = 64$, $s = 4$, $B = 64$ 를 넣으면 블록 전치의 둘째 항은 첫째 항의 4분의 1이다. 통째 전치의 둘째 항은 첫째 항의 열여섯 배다. 두 식의 비는 약 열세 배이고, 이것이 실험 1.2 의 열여덟 배와 같은 자리에서 나온 숫자다.

구현

블록 전치는 반복문 둘과 자르기 하나다. 자르기는 1.2 의 뷰이므로 이동량이 없고, 대입 한 줄이 블록 하나를 옮긴다.

지금 보는 코드 블록 단위 전치. 안쪽 대입 한 줄이 캐시에 드는 사각형 하나를 옮긴다.

구현 블록 전치와 타일 크기 `samples/dsai/tiling.py:24`

```
def transpose_blocked(a: np.ndarray, block: int,
                      out: np.ndarray | None = None) -> np.ndarray:
    """block x block 타일 단위로 전치한다.

    타일 하나는 캐시에 들어가므로, 타일 안에서는 라인을 가져와서 원소를 다 쓴다.
    """
    n, m = a.shape
    if out is None:
        out = np.empty((m, n), dtype=a.dtype)
    for i in range(0, n, block):
        for j in range(0, m, block):
            out[j:j + block, i:i + block] = a[i:i + block, j:j + block].T
    return out


def tile_bytes(block: int, itemsize: int) -> int:
    """타일 하나가 차지하는 바이트. 입력 타일과 출력 타일 둘이 캐시에 들어야 한다."""
    return 2 * block * block * itemsize
```

읽어 보면 `a[i:i+b, j:j+b].T` 는 정리 1.5 의 뷰라 값이 0이다. 그 뷰를 `out` 의 자리에 대입할 때 NumPy 가 사각형 하나를 옮기고, 사각형이 캐시에 들어 있으니 열 방향 읽기가 라인을 놓치지 않는다.

알고리즘 1.2 블록 전치 `dsai.tiling.transpose_blocked`

입력: $n \times m$ 행렬 A, 블록 크기 b $(2b^2 s \le Z)$
출력: $m \times n$ 행렬 A^{T}

반복 $i = 0, b, 2b, \cdots < n$ **동안**
  반복 $j = 0, b, 2b, \cdots < m$ **동안**
    $A^{\mathsf{T}}[j..j+b,\ i..i+b] \leftarrow (A[i..i+b,\ j..j+b])^{\mathsf{T}}$ ▷ 블록 하나. 캐시 안에서 전치
  끝 반복
끝 반복
반환 A^{T}

실험 1.2 의 아래 네 행이 이 알고리즘이다. 블록 64 가 통째 전치보다 네 배 가까이 빠르다. 블록 16 은 오히려 느린데, 블록이 26 만 개라 파이썬 반복문의 값이 이동량을 넘어선다. 모델 밖의 상수다. 블록 1024 는 입력과 출력 타일이 8MB 로 마지막 캐시를 꽉 채우므로 조금 느려진다. 정의 1.7 의 불변식 경계다.

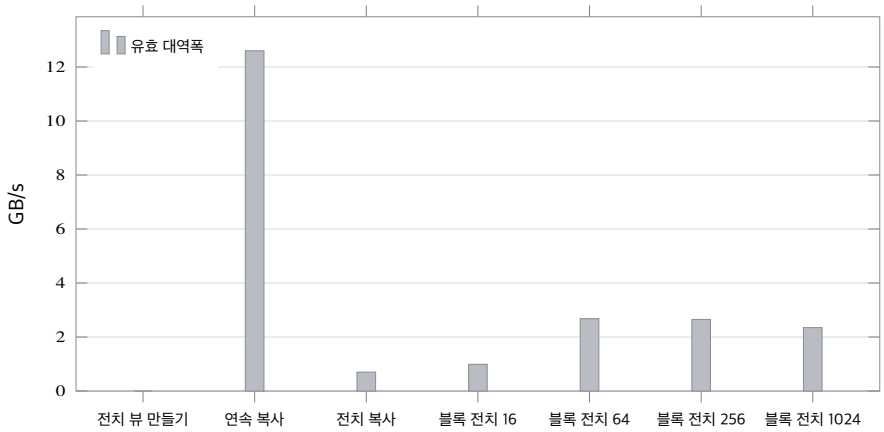


그림 1.7. 같은 256MB 를 옮기는 다섯 방법의 유효 대역폭. 뷰 만들기는 0 이라 막대가 없다

분석

블록 순회는 새 생각이 아니다. Frigo 와 동료들이 1999 년에 블록 크기를 모르고도 같은 이동량을 얻는 재귀 분할을 보였고, 그것을 캐시 망각 알고리즘이라 불렀다 [확립] [5]. 이 책이 블록 크기를 명시하는 쪽을 택한 것은 기계의 캐시 크기를 부록에 적어 두었기 때문이다.

이 절의 정리는 15 장에서 그대로 다시 나온다. 어텐션의 QK^{T} 는 행렬 곱이고, 행렬 곱의 블록 순회가 FlashAttention 의 절반이다. 나머지 절반은 소프트맥스를 블록 단위로 계산하는 산수이고, 그것은 그때 세운다. 타일 크기를 정하는 산수는 여기 것과 같다.

현장에서

Drepper 가 2007 년에 낸 메모리 해설은 이 절의 실험 1.3 과 같은 간격 훑기를 다른 기계에서 보이고, 선인출과 주소 변환이 모델을 어디서 벗어나게 하는지 적었다 [확립] [7].

되짚기

> 간격 접근의 이동량을 정리로 세웠고, 실측이 열여섯 배까지는 정리를 따르고 그 뒤로는 벗어나는 것을 봤다. 블록 순회가 전치의 이동량을 열 배 넘게 줄인다. 다음 절에서 이동량을 줄이는 대신 연산을 늘리는 다른 길, 배치를 본다.

1.4 배치와 브로드캐스팅

1.3 은 이동량을 줄이는 길이었다. 이 절은 반대 방향으로 간다. 이동량은 그대로 두고 연산을 늘린다. 0.2 의 루프라인에서 밀도는 연산을 이동량으로 나눈 것이므로, 분자를 키워도 점이 오른쪽으로 간다. 분자를 키우는 가장 단순한 방법이 차원 하나를 더하는 것이다. 질의 하나 대신 질의 B 개를 한 번에 처리한다.

0.3 의 전수 탐색이 밀도 0.5 에 묶여 있었다. 그 행렬을 한 번 읽어 질의 하나에만 쓰기 때문이다. 같은 행렬을 한 번 읽어 질의 64 개에 쓰면 밀도가 64 배 가까이 오른다. 그 산수를 명제로 세우고, 배치를 텐서 연산으로 적는 규칙인 브로드캐스팅을 정의한다.

학습 목표

> 브로드캐스팅 규칙을 정의하고 두 모양의 결과 모양을 손으로 구할 수 있다. 배치 크기에 따른 연산 밀도를 명제로 세우고, 실측 처리량이 지붕에 다가가는 것을 루프라인으로 읽을 수 있다.

이 말부터

배치	같은 연산을 받을 입력 여러 개를 축 하나로 묶은 것. 그 축의 크기가 배치 크기 B 다
브로드캐스팅	모양이 다른 두 텐서를 연산할 때 크기 1 인 축을 상대 크기에 맞춰 늘려 보는 규칙
스트라이드 0	축 하나를 늘리되 메모리를 늘리지 않는 방법. 같은 원소를 여러 번 읽는 뷰다

디딤돌

행렬 벡터 곱 $y = Xq$ 는 X 의 원소마다 곱 하나 합 하나다. 원소 하나가 4 바이트이니 밀도는 $2/4 = 0.5$ 다. 행렬 행렬 곱 $Y = XQ^{\mathsf{T}}$ 는 X 의 원소 하나를 Q 의 행 B 개와 곱한다. 같은 4 바이트로 $2B$ 번 계산한다. 이것이 배치가 밀도를 올리는 원리이고, 계산은 아래 명제 하나로 끝난다.

명제 1.9: 배치 크기와 연산 밀도

> $n \times d$ 행렬 X 와 $B \times d$ 질의 Q 의 곱 XQ^{T} 는 연산 수 $W = 2ndB$, 이동량 $M = s(nd + dB + nB)$ 바이트다. 따라서 연산 밀도는
>
> $$I(B) = \frac{2ndB}{s(nd + dB + nB)} = \frac{2B}{s} \cdot \frac{1}{1 + B/n + B/d}$$
>
> 이고, $B \ll n, d$ 인 동안 B 에 비례해 오르다가 $\min(n, d)$ 근처에서 멈춘다.

증명. 결과 원소 nB 개마다 길이 d 의 내적이므로 곱셈 ndB 번, 덧셈 ndB 번이다. X 를 한 번, Q 를 한 번 읽고 Y 를 한 번 쓰는 것이 이동량의 하한이다. 나눠서 정리하면 위 식이다. $B \to \infty$ 이면 분모의 $B/n + B/d$ 가 커져 $I \to 2nd/(s(n+d))$ 로 수렴한다. ■

식이 말하는 것은 둘이다. B 가 작을 때 밀도는 B 에 정비례하고, B 가 d 에 가까워지면 질의 행렬을 읽는 값이 보이기 시작한다. 이 절의 실험은 $d = 128$ 이라 $B = 64$ 에서 이미 둘째 효과가 절반쯤 들어온다. 숫자를 넣어 보면 아래와 같다.

예제 1.4: 배치 64 의 밀도

$n = 262{,}144$, $d = 128$, $s = 4$, $B = 64$ 이면 $B/n \approx 0.0002$, $B/d = 0.5$ 다. $I(64) = 32/1.5 \approx 21.3$ FLOP/byte 다. $B = 1$ 이면 $0.5/1.008 \approx 0.496$ 이니 43 배 올랐다. 이 기계의 능선점은 11.5 이므로 배치 64 는 능선점 오른쪽에 있다.

균열

같은 행렬에 배치 크기를 1 부터 256 까지 바꿔 가며 곱했다. 0.3 의 전수 탐색을 질의 여러 개로 한 번에 하는 것이다.

실험 1.4 배치 크기에 따른 밀도와 처리량

배치	밀도	처리량	질의당 ms
1	0.50	12.0	5.59
4	1.94	17.1	3.93
16	7.11	40.5	1.66
64	21.3	73.1	0.92
256	42.7	57.8	1.16

측정 조건. float32, 행렬 $262{,}144 \times 128$ (128MB). 밀도는 명제 1.5, 처리량은 실측 GFLOP/s. 일곱 번 중앙값.

배치 1 에서 64 로 가는 동안 처리량이 여섯 배 오르고 질의당 시간은 6분의 1로 준다. 읽는 행렬은 같고 옮기는 바이트도 거의 같다. 달라진 것은 옮겨 온 바이트 하나로 몇 번 계산하느냐뿐이다.

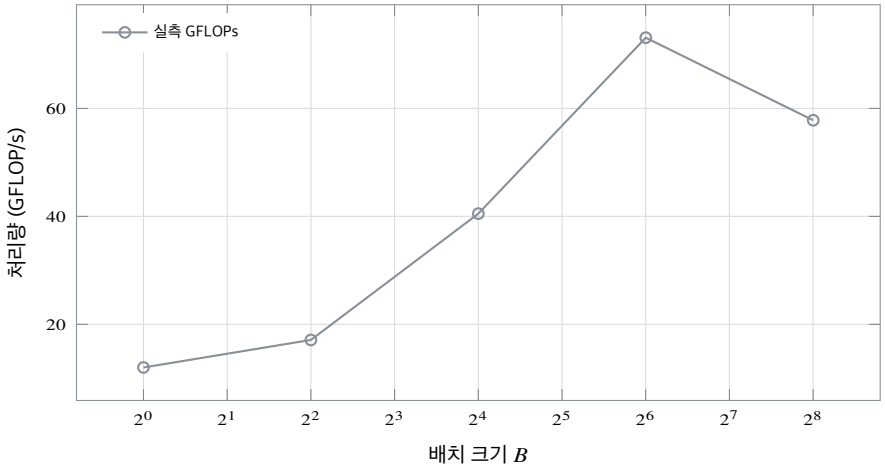


그림 1.8.. 배치 크기에 따른 처리량. 능선점을 넘으면 오르기를 멈추고, 256 에서는 오히려 내려온다

배치 256 에서 처리량이 내려오는 것도 그대로 적는다. 밀도는 43 으로 더 올랐는데 처리량은 떨어졌다. 명제 1.9 는 상한을 말할 뿐이고, 상한에 붙는지는 구현이 정한다. 결과 행렬 Y 가 $262{,}144 \times 256 \times 4 = 256\text{MB}$ 로 커져 쓰기 이동량이 늘고, 라이브러리의 블록 크기가 이 모양에 맞지 않은 것으로 본다.

관찰

> 밀도가 능선점을 넘은 뒤에는 밀도를 더 올려도 처리량이 오르지 않는다. 그때부터 병목은 이동량이 아니라 연산 장치와 구현이다.

다시 쓰기

배치를 코드로 적으려면 모양이 다른 텐서끼리의 연산 규칙이 필요하다. 질의 하나는 $(d,)$ 이고 질의 묶음은 (B, d) 다. 같은 코드가 둘 다 받으려면 축을 맞추는 규칙이 있어야 하고, 그 규칙이 브로드캐스팅이다.

정의 1.10: 브로드캐스팅

> 모양 $a = (a_1, \ldots, a_p)$ 와 $b = (b_1, \ldots, b_q)$ 를 오른쪽 끝을 맞춰 나란히 놓고, 짧은 쪽 앞에 1 을 채운다. 각 자리에서 두 크기가 같거나 한쪽이 1 이면 결과의 크기는 둘 중 큰 쪽이고, 둘 다 1 이 아니면서 다르면 연산이 정의되지 않는다. 크기 1 인 축은 스트라이드 0 인 뷰로 늘린다. 메모리는 늘지 않는다.

스트라이드 0 이 이 정의의 자료 구조 부분이다. 1.2 의 정의 1.4 가 스트라이드에 제약을 두지 않은 이유가 여기 있다. $(d,)$ 벡터를 (B, d) 로 늘려 보는 것은 첫 축의 스트라이드를 0 으로 둔 뷰이고, 그 뷰의 B 개 행은 전부 같은 메모리를 읽는다. 이동량으로 치면 그 벡터를 한 번만 옮기고 B 번 쓰는 것이니, 브로드캐스팅 자체가 밀도를 올리는 장치다.

알고리즘 1.3 브로드캐스트 모양 `dsai.batch.broadcast_shape`

입력: 모양 a, 모양 b
출력: 결과 모양, 또는 실패
짧은 쪽 앞에 1 을 채워 길이를 맞춘다
반복 뒤에서부터 각 자리 (x, y) **동안**
    만약 $x = y$ **또는** $y = 1$ **이면**
        결과에 x 를 넣는다
    아니면 만약 $x = 1$ **이면**
        결과에 y 를 넣는다
    아니면
        실패
    끝 만약
끝 반복
반환 결과

구현

브로드캐스트 규칙과 배치 밀도의 산수를 코드로 둔다. 배치 점수 함수는 행렬 곱 한 줄이다.

지금 보는 코드 브로드캐스트 규칙과 배치 밀도. 실험 1.5 의 표를 만드는 데 쓴 함수다.

구현 브로드캐스트 규칙 `samples/dsai/batch.py:11`

```
def broadcast_shape(a: tuple[int, ...], b: tuple[int, ...]) -> tuple[int, ...]:
    """정의 1.4 의 규칙. 뒤에서부터 맞추고, 1 은 늘어나고, 다르면 실패한다."""
    out = []
    for x, y in zip(reversed((1,) * (len(b) - len(a)) + a),
                    reversed((1,) * (len(a) - len(b)) + b)):
        if x == y or y == 1:
            out.append(x)
        elif x == 1:
            out.append(y)
        else:
            raise ValueError(f"맞지 않는 모양 {a} 와 {b}")
    return tuple(reversed(out))
```

읽어 보면 두 모양을 뒤집어 짝지은 뒤 정의 1.10 의 세 경우를 차례로 본다. 짧은 쪽 앞에 1 을 채우는 것은 튜플을 이어 붙이는 것으로 처리했다.

구현 배치 점수와 밀도 `samples/dsai/batch.py:25`

```python
def scores_one(xs: np.ndarray, q: np.ndarray) -> np.ndarray:
    """질의 하나. 행렬 벡터 곱. 밀도 0.5."""
    return xs @ q


def scores_batch(xs: np.ndarray, qs: np.ndarray) -> np.ndarray:
    """질의 B 개. 행렬 행렬 곱. xs 를 한 번 읽어 B 번 쓴다."""
    return xs @ qs.T


def intensity_batched(n: int, d: int, b: int, elem_bytes: int = 4) -> float:
    """명제 1.3. 배치 크기 b 에서의 연산 밀도."""
    flops = 2 * n * d * b
    nbytes = (n * d + d * b + n * b) * elem_bytes
    return flops / nbytes
```

분석

배치는 이 책이 다루는 시스템의 가장 큰 구조적 결정이다. 추론 서버가 요청을 하나씩 처리하지 않고 모아서 처리하는 이유가 전부 명제 1.9 다. 그런데 모으면 기다려야 한다. 배치 64 를 채우려면 요청 64 개가 올 때까지 앞의 요청이 서고, 그 기다림이 지연이 된다. 처리량과 지연이 여기서 갈리고, 이 책은 5 장에서 그 갈림을 스케줄링 문제로 다시 세운다.

현장에서

> Yu 와 동료들의 2022 년 Orca 는 생성 모델의 배치를 요청 단위가 아니라 토큰 반복 단위로 다시 짜서, 끝난 요청을 빼고 새 요청을 바로 넣는다 [확립] [8]. 2023 년의 vLLM 은 그 배치를 페이지 단위 KV 캐시 위에 얹었다 [확립] [9]. 둘 다 이 절의 명제가 출발점이다.

전통 교과서에는 배치라는 개념이 없다. 있을 이유가 없었다. 균일 비용 모델에서는 질의 64 개를 한 번에 하나 질의 하나씩 64 번 하나 값이 같다. 이 개념이 필요해진 것은 첫째 가정이 깨졌기 때문이고, 그래서 이 절은 0.2 의 표에서 둘째 줄이 첫째 줄에서 나온다고 적은 자리다.

주의

> 배치가 밀도를 올리는 것은 행렬을 여러 질의가 나눠 읽을 때뿐이다. 질의마다 다른 행렬을 읽는 작업량에서는 배치가 이동량을 줄이지 못한다.

이 주의가 이 책의 뒤에서 두 번 다시 나온다. 3 장의 전문가 혼합은 질의마다 다른 행렬을 읽어 배치의 이득이 줄어드는 자리이고, 15 장의 디코딩은 토큰 하나마다 전체 가중치를 읽어 배치 없이는 밀도가 1 을 넘지 못하는 자리다. 둘 다 이 절의 명제를 뒤집어 읽으면 왜 느린지가 나온다.

되짚기

> 배치 크기에 따른 밀도를 명제로 세웠고, 실측이 능선점까지 그 명제를 따르는 것을 봤다. 브로드캐스팅은 스트라이드 0 뷰로 배치를 적는 규칙이다. 다음 장에서 이 값 배열이 커져야 할 때, 즉 동적 배열에서 무엇이 깨지는지 본다.

1장을 닫으며

이 장이 세운 구조는 하나다. 값 배열에 스트라이드를 얹은 텐서 뷰. 나머지 셋은 그 구조를 쓰는 법이다. 뷰를 옮길지 말지, 옮긴다면 어느 순서로, 그리고 축을 하나 더해 몇 번 다시 쓸지. 구조는 짧고 쓰는 법이 길다. 이 장의 이동량 산수가 그 뒤 열여섯 장에서 되풀이되기 때문이다.

네 실험이 보인 것을 적어 둔다. 같은 실수 백만 개가 참조 배열에서는 여덟 배 크고 열 배 넘게 느렸다. 256MB 행렬의 전치 뷰는 1마이크로초에 만들어졌고, 그 뷰를 옮기는 데는 연속 복사의 열여덟 배가 들었다. 간격을 16 까지 늘리는 동안 원소당 시간이 열여섯 배 올랐고, 블록 64 의 전치가 그 손해를 네 배 가까이 되찾았다. 배치를 1 에서 64 로 늘리자 같은 행렬로 여섯 배의 처리량이 나왔다.

정정 하나를 남긴다. 처음에는 전치 복사를 잴 때 출력 배열을 매번 새로 잡았다. 그랬더니 연속 복사가 6배 느리게 나왔고, 블록 전치와의 차이가 작아 보였다. 새 메모리를 처음 쓰는 값, 즉 페이지를 실제로 붙이는 값이 복사 시간에 섞여 있었다. 출력을 미리 잡아 두고 다시 쟀고, 그것이 실험 1.2 다. 이 책의 다른 실험도 같은 규칙으로 잰다. 재는 것이 무엇인지 먼저 정하고, 그것만 잰다.

옛 구조의 판정

> 배열은 그대로 유효하다. 연속성 불변식이 지금도 가장 싼 이동량을 준다. 다만 교과서가 배열의 원소를 스칼라로 둔 것은 이제 안 맞는다. 원소가 벡터이고, 벡터가 값 배열이어야 한다. 근거는 실험 1.1 이다. 원소가 참조인 배열은 크기 여덟 배, 시간 열 배였다.

이 장의 요약

> 1. 배열은 연속성 불변식이고, 원소가 참조이면 값에 대해 그 불변식이 깨진다 1.1
> 2. 값 배열은 원소 타입을 정의에 넣은 배열이고, 타입이 3장의 양자화까지 이어진다 1.1
> 3. 텐서 뷰는 버퍼, 모양, 스트라이드, 시작 위치이고, 뷰 연산의 이동량은 0 이다 1.2
> 4. 간격 k 접근의 이동량은 ks/B 에 비례해 오르다 1 에서 멈추고, 블록 순회가 그것을 되찾는다 1.3
> 5. 배치 B 는 밀도를 B 배 가까이 올리고, 능선점을 넘으면 더 오르지 않는다 1.4
>
> 다음 장에서 뒤집히는 것: 값 배열은 크기가 고정이다. 크기가 자라야 하는 순간 두 배 늘리기의 낭비가 문제가 된다.

필자의 판단

> 필자는 스트라이드를 배열의 정의에 넣어 가르치는 쪽이 맞다고 본다. 대부분의 교과서는 스트라이드를 구현 세부로 미룬다. 그런데 이 장의 실험 1.2 가 보였듯이 뷰가 공짜인지 아닌지는 스트라이드를 알아야 판단이 되고, 그 판단이 틀리면 열여덟 배다. 열여덟 배를 가르는 것을 구현 세부라고 부를 수는 없다.

열린 문제

하나, 실험 1.3 에서 간격 16 을 넘은 뒤에도 원소당 시간이 계속 오른 것을 이 장의 모델은 설명하지 못한다. 선인출과 주소 변환 캐시를 모델에 넣는 것은 이 책의 범위 밖이다.
둘, 실험 1.5 의 배치 256 에서 처리량이 내려온 이유를 필자는 라이브러리 블록 크기로 추정했을 뿐 확인하지 못했다.
셋, 블록 크기 b 를 캐시 용량에서 거꾸로 구했는데, 실험에서는 64 와 256 이 거의 같았다. 어느 캐시 층이 실제 병목인지는 재지 않았다.

연습문제

1.1 기초 float64 행렬 3×4 를 행 우선으로 놓았을 때의 스트라이드를 구하고, 원소 $(2, 1)$ 의 바이트 위치를 계산하라. 열 우선이면 어떻게 달라지는가.

1.2 기초 파이썬 리스트에 실수 백만 개를 넣었을 때의 메모리를 포인터 8 바이트와 실수 객체 24 바이트로 계산하고, float32 값 배열과의 비를 구하라.

1.3 기초 float64 배열을 간격 4 로 백만 개 읽을 때의 이동량을 정리 1.3 으로 계산하라. 간격 8 이면 어떻게 되는가.

1.4 기초 $n = 262{,}144$, $d = 128$, $s = 4$ 일 때 명제 1.5 로 $B = 1, 16, 256$ 의 연산 밀도를 계산하고, 능선점 8 을 넘는 최소 B 를 구하라.

1.5 응용 3 차원 텐서 $(2, 3, 4)$ 의 첫 축과 셋째 축을 바꾼 뷰의 스트라이드를 `dsai.tensor.View` 로 구현하고, `np.swapaxes` 의 스트라이드와 같은지 확인하라.

1.6 응용 실험 1.3 의 각 간격에 대해 정리 1.3 이 예측한 원소당 라인 수를 `dsai.tensor.lines_for_traversal` 로 구하고, 실측 원소당 시간과 한 표에 놓아라. 어디서부터 갈리는가.

1.7 응용 `dsai.tiling.tile_bytes` 로 L2 256KB 와 L3 8MB 각각에 드는 가장 큰 2 의 거듭제곱 블록을 구하고, 실험 1.2 의 블록 64 와 1024 결과와 대조하라.

1.8 응용 `dsai.batch.intensity_batched` 로 능선점을 넘는 최소 배치 크기를 구하는 함수를 쓰고, 실험 1.5 에서 처리량이 오르기를 멈춘 자리와 비교하라.

1.9 탐구 같은 행렬을 열 우선으로 저장하면 어떤 작업량에서 이기는가. 열 하나를 통째로 읽는 연산과 행 하나를 통째로 읽는 연산의 빈도로 답을 정리하라.

1.10 탐구 실험 1.3 에서 간격 16 을 넘은 뒤에도 원소당 시간이 오르는 이유를 선인출과 주소 변환으로 가설을 세우고, 두 가설을 가르는 실험을 설계하라.

더 읽을 것

[6] NumPy 배열 객체의 설계. 타입, 스트라이드, 브로드캐스팅이 한 그림에 있다. 이 장 전체의 원전이다. 1.1, 1.2, 1.4
[7] 캐시와 주기억의 동작을 백 쪽에 걸쳐 실측으로 보인 문서. 실험 1.3 이 왜 16 을 넘어서도 오르는지 여기에 답이 있다. 1.3
[5] 블록 크기를 모르고도 최적인 재귀 분할. 이 장이 명시적 블록을 택한 것과 견줘 읽는다. 1.3
[8] 배치를 요청 단위에서 토큰 반복 단위로 바꾼 논문. 명제 1.5 가 서버 설계로 자라는 자리다. 1.4

2장

동적 배열에서 페이지 테이블로

2023년의 한 추론 서버 논문은 기존 시스템이 가속기 메모리 중 KV 캐시 자리의 60 에서 80 퍼센트를 낭비하고 있었다고 재서 보고했다 [확립] [9]. 낭비의 원인은 새것이 아니었다. 길이를 모르는 배열에 최대 길이를 예약한 내부 단편화와, 길이가 제각각인 배열이 들고 나며 남긴 외부 단편화였다. 해법도 새것이 아니었다. 1970년에 정리된 페이지 테이블이었다.

이 장은 0.2 의 첫째 가정을 다른 각도에서 건드린다. 배열이 자랄 때 옮기는 값과 남는 값은 균일 비용 모델에서는 시간 하나로 셌다. 메모리가 대역폭이고 대역폭이 시간인 자리에서는 빈 바이트가 곧 시간이다. 그래서 이 장은 분할 상환 분석에서 시작해 단편화를 세고, KV 캐시를 정의한 뒤, 페이지 테이블로 그 셋을 한 번에 다시 쓴다.

이 장에서 세우는 것

2.1 분할 상환 분석과 두 배 늘리기

1.1 의 값 배열은 크기가 고정이다. 만들 때 원소 수를 알아야 한다. 그런데 이 책이 다루는 배열의 절반은 만들 때 크기를 모른다. 생성 모델의 출력이 몇 토큰이 될지, 말뭉치가 몇 문서가 될지는 끝나 봐야 안다. 크기를 모르는 배열을 값 배열로 두려면 자라야 하고, 자라려면 옮겨야 한다.

이 절은 그 옮기는 값을 세는 방법을 세운다. 분할 상환 분석이다. 어느 한 번의 추가는 비싸지만 전체를 추가 횟수로 나누면 싸다는 논증이고, 그 논증이 성장 인자 하나에 달려 있다는 것을 재서 보인다.

학습 목표

> 동적 배열을 용량과 크기의 불변식으로 정의하고, 두 배 늘리기의 분할 상환 비용이 상수임을 증명할 수 있다. 성장 인자가 옮긴 바이트와 남은 바이트를 어떻게 맞바꾸는지 실측으로 설명할 수 있다.

이 말부터

> 용량 — 미리 잡아 둔 자리의 수. 크기는 그중 실제로 쓰는 자리의 수다. 크기가 용량을 넘을 수 없다
>
> 분할 상환 — 연산 열 전체의 비용을 연산 수로 나눈 것. 한 번의 비싼 연산을 여러 싼 연산이 나눠 갚는다
>
> 성장 인자 — 용량이 찼을 때 새 용량을 옛 용량의 몇 배로 잡는가. 2 이면 두 배 늘리기다

디딤돌

동적 배열은 값 배열 위에 숫자 하나를 더 얹은 것이다. 용량이다.

정의 2.1: 동적 배열

> 동적 배열은 용량 c 인 값 배열 B 와 크기 n 의 짝이다. 연산은 읽기, 쓰기, 그리고 끝에 추가하기다. 추가할 때 $n = c$ 이면 용량 $c' = \lceil \alpha c \rceil$ 인 새 배열을 잡고 원소 n 개를 옮긴 뒤 추가한다. $\alpha > 1$ 이 성장 인자다.
>
> **불변식** 언제나 $n \le c$ 이고, 원소 n 개가 B 의 앞 n 자리에 연속으로 있다.

추가 한 번의 값은 둘 중 하나다. 자리가 있으면 쓰기 하나이고, 자리가 없으면 원소 n 개를 옮긴다. 옮기는 추가는 가끔 오고 그때마다 n 이 다르다. 그래서 한 번의 값을 묻는 대신 n 번의 값을 합쳐서 묻는다.

정리 2.2: 두 배 늘리기의 분할 상환 비용

> 성장 인자 $\alpha = 2$ 이고 처음 용량이 c_0 인 동적 배열에 원소 n 개를 차례로 추가하면, 옮긴 원소의 총수는 $2n$ 을 넘지 않는다. 따라서 추가 한 번의 분할 상환 비용은 쓰기 하나와 옮기기 둘 이하, 즉 상수다. 일반 α 에서는 옮긴 총수가 $\alpha n/(\alpha - 1)$ 을 넘지 않는다.

증명. 용량은 $c_0, \alpha c_0, \alpha^2 c_0, \ldots$ 로 자라고, k 번째로 자랄 때 옮기는 원소 수는 그때의 용량 $\alpha^{k-1} c_0$ 이다. 마지막 용량 c_m 은 αn 을 넘지 않는다. 그 전 용량이 n 보다 작았기 때문이다. 옮긴 총수는 등비수열의 합 $c_0 + \alpha c_0 + \cdots + c_m/\alpha < c_m/(\alpha - 1) \le \alpha n/(\alpha - 1)$ 이다. $\alpha = 2$ 를 넣으면 $2n$ 이다. ■

예제 2.1: 원소 천 개를 넣을 때

> $c_0 = 4, \alpha = 2$ 이면 용량이 $4, 8, 16, \ldots, 1024$ 로 여덟 번 자라고, 옮긴 수는 $4+8+\cdots+512 = 1020$ 이다. 정리의 상한 $2n = 2000$ 아래다. $\alpha = 1.5$ 이면 열여덟 번 자라고 옮긴 수는 약 $3n$ 이다. 상한 $\alpha n/(\alpha - 1) = 3n$ 과 같다.

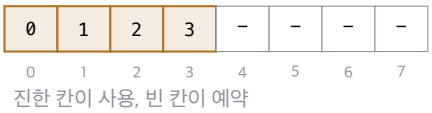


그림 2.1.. 용량 여덟에 원소 넷. 다음 넣기 넷은 복사가 없고, 아홉째에서 열여섯으로 늘리며 여덟을 옮긴다

이 정리가 이 책의 뒤에서 되풀이되는 이유는 두 가지다. 하나는 α 가 1 에 가까울수록 옮기는 값이 커진다는 것이고, 다른 하나는 α 가 클수록 마지막에 남는 빈자리가 커진다는 것이다. 정리는 첫째만 말한다. 둘째는 재서 본다.

균열

원소 백만 개를 하나씩 넣었다. 성장 인자를 2, 1.5, 1.125 로 바꾸고, 인자 대신 고정 개수를 더하는 방법과 처음부터 다 잡아 두는 방법을 나란히 놓았다.

실험 2.1 성장 방법에 따른 옮긴 양과 남은 양

방법	옮긴 횟수	옮긴 MB	남은 MB	시간 ms
두 배 늘리기	18	4.19	0.19	590
1.5 배 늘리기	31	8.40	0.20	619
1.125 배 늘리기	108	34.7	0.34	724
1024 씩 더하기	977	1953	0.00	2937
미리 잡기	0	0.00	0.00	186

측정 조건. float32 원소 백만 개를 하나씩 추가. 옮긴 MB 는 자랄 때마다 복사한 바이트의 합, 남은 MB 는 끝난 뒤 빈 용량. 다섯 번 중앙값.

두 배 늘리기는 백만 개를 넣는 동안 4MB 를 옮겼다. 원소 전체가 4MB 이니 정리 2.2 의 $2n$ 안이다. 1.125 배는 여덟 배를 옮겼고, 1024 씩 더하기는 2GB 를 옮겼다. 옮긴 양이 원소 수의 제곱에 비례하기 때문이다. 시간 열은 파이썬 반복문의 값이 섞여 있어 차이가 작게 보이지만, 마지막 행만 열다섯 배 느린 것은 옮긴 2GB 가 그대로 시간이 된 것이다.

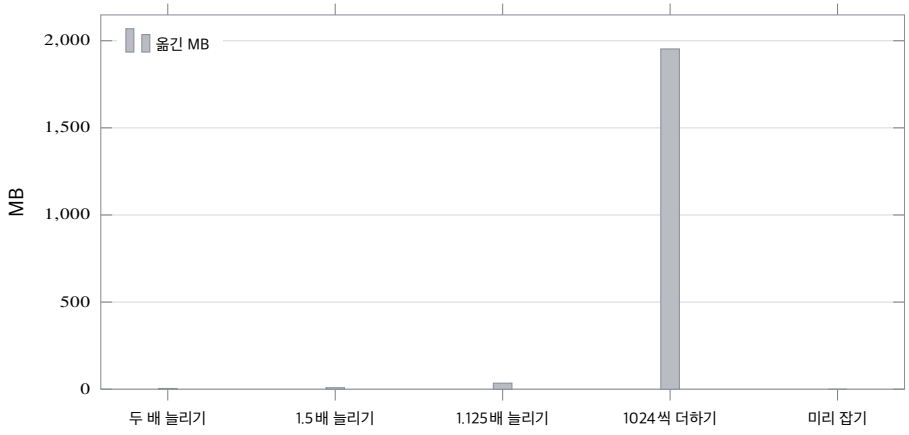


그림 2.2. 성장 방법에 따라 옮긴 바이트. 고정 증가는 원소 수의 제곱이라 막대가 그림을 벗어난다

남은 MB 열이 둘째 대가다. 두 배 늘리기는 끝난 뒤 용량의 절반 가까이가 빌 수 있다. 이 실험에서는 백만이 2^{20} 바로 아래라 남은 것이 적지만, 원소가 $2^{19} + 1$ 개였다면 절반이 빈다. 옮기는 값과 남는 값이 α 하나로 반대 방향으로 움직인다.

관찰

성장 인자가 크면 덜 옮기고 더 남긴다. 작으면 더 옮기고 덜 남긴다. 어느 쪽이 싼지는 옮기는 값과 빈자리의 값 중 무엇이 비싼지가 정한다.

다시 쓰기

이 절의 다시 쓰기는 정리 2.2 의 둘째 대가를 정리로 세우는 것이다. 전통 교과서는 옮기는 값만 셌다. 메모리가 싸고 시간이 비싼 시절의 셈이다. 이 책이 다루는 자리에서는 빈자리가 시간만큼 비싸다. 빈자리가 다른 요청의 자리를 빼앗기 때문이다.

명제 2.3: 성장 인자와 빈자리

> 성장 인자 α 인 동적 배열에 원소 n 개가 들어 있을 때, 빈자리는 최악의 경우 $(\alpha-1)n$ 에 가깝고 평균적으로는 그 절반 근처다. 동적 배열 r 개가 있으면 빈자리의 합은 최악 $(\alpha-1)\sum n_i$ 다.

증명. 자란 직후에는 $n = c/\alpha$ 이므로 빈자리가 $c - c/\alpha = (\alpha-1)n$ 이다. 그 뒤 채워지면서 0 까지 준다. n 이 용량 구간 안에서 고르게 분포한다고 보면 평균은 절반 근처다. 배열이 여럿이면 각각을 더한다. ■

$\alpha = 2$ 인 배열 천 개가 있으면 전체 메모리의 4분의 1에서 절반이 빈다. 하나일 때는 문제가 아니었던 것이 천 개가 되면 문제가 된다. 2.3 의 KV 캐시가 정확히 이 경우다. 요청마다 배열 하나이고 요청이 수백 개 동시에 산다.

구현

성장 인자와 고정 증가를 둘 다 받는 동적 배열이다. 옮긴 바이트를 세는 것이 이 구현의 목적이다.

지금 보는 코드 용량이 차면 자라는 값 배열. 자랄 때 옮긴 바이트와 자란 횟수를 센다.

구현 자라기와 추가하기 `samples/dsai/dynarray.py:27`

```
    def _grow(self) -> None:
        if self.increment:
            new_cap = self.capacity + self.increment
        else:
            new_cap = max(self.capacity + 1, int(self.capacity * self.factor))
        new_buf = np.empty(new_cap, dtype=self.buf.dtype)
        new_buf[:self.size] = self.buf[:self.size]
        self.copied_bytes += self.size * self.buf.itemsize
        self.grow_count += 1
        self.buf = new_buf

    def append(self, value) -> None:
        """분할 상환 O(1). 용량이 차면 먼저 자란다."""
        if self.size == self.capacity:
            self._grow()
        self.buf[self.size] = value
        self.size += 1
```

읽어 보면 `_grow` 는 새 용량을 정하고 새 버퍼를 잡아 앞 `size` 개를 옮긴다. 옮긴 바이트를 `copied_bytes` 에 더한다. `append` 는 자리가 없을 때만 `_grow` 를 부르므로, 옮기는 값이 추가 여러 번에 걸쳐 나뉜다. 그것이 분할 상환이다.

알고리즘 2.1 동적 배열에 추가 `dsai.dynarray.DynArray.append`

입력: 동적 배열 (B, n, c), 값 v
1: **만약** $n = c$ **이면**
2: $B' \leftarrow$ 용량 $\lceil \alpha c \rceil$ 인 새 배열; $B'[0..n) \leftarrow B[0..n)$; $B \leftarrow B'$ ▷ 옮기기 n 번
3: **끝 만약**
4: $B[n] \leftarrow v$; $n \leftarrow n + 1$

정리 2.2 의 합을 그대로 세는 함수도 두었다. 실험 2.1 의 옮긴 MB 열이 이 함수의 값과 같은지를 시험이 확인한다.

구현 옮긴 원소 수의 합 `samples/dsai/dynarray.py:53`

```
def total_copied(n: int, factor: float, capacity: int = 4) -> int:
    """원소 n 개를 넣을 때 옮겨진 원소 수의 합. 정리 2.1 의 식을 그대로 센다."""
    cap, copied = capacity, 0
    while cap < n:
        copied += cap
        cap = max(cap + 1, int(cap * factor))
    return copied
```

분석

분할 상환 분석은 그대로 유효하다. Tarjan 이 1985 년에 이 분석을 정리한 뒤로 형식이 바뀐 적이 없다 [확립] [10]. 바뀐 것은 세는 대상이다. 전통 교과서는 옮긴 원소 수를 셌고, 이 책은 옮긴 바이트와 빈 바이트를 같이 센다.

빈 바이트를 세면 답이 달라진다. 배열 하나에서는 $\alpha = 2$ 가 옮기는 값을 가장 잘 갚는다. 배열이 수백 개이고 메모리가 꽉 찼으면 $\alpha = 2$ 가 메모리의 4 분의 1 을 비워 두는 방법이 된다. 그 자리에서 성장 인자를 줄이면 옮기는 값이 오르고, 옮기는 값을 안 내려면 다른 구조가 필요하다. 그 구조가 2.4 이다.

현장에서

CPython 의 리스트는 성장 인자를 약 1.125 로 잡는다. 공개 저장소의 `listobject.c` 에 그 식과 이유가 주석으로 있다 [확립]. 옮기는 값보다 빈자리를 아끼는 쪽을 택한 것이고, 실험 2.1 의 셋째 행이 그 값이다.

되짚기

동적 배열을 용량과 크기로 정의했고, 두 배 늘리기가 옮긴 총수를 $2n$ 아래로 묶는 것을 증명했다. 성장 인자는 옮기는 값과 빈자리를 맞바꾼다. 다음 절에서 빈자리가 어디에 어떤 모양으로 남는지, 단편화를 센다.

2.2 단편화, 예약과 낭비의 산수

2.1 의 명제는 배열 하나 안의 빈자리를 셌다. 배열이 여럿이고 그 배열들이 메모리 하나를 나눠 쓰면 빈자리가 두 종류가 된다. 배열 안에 있어서 그 배열만 쓸 수 있는 빈자리와, 배열 사이에 있어서 아무도 못 쓰는 빈자리다. 운영체제 교과서가 내부 단편화와 외부 단편화라고 부르는 것이고, 이 절은 그 둘을 이 책의 작업량으로 다시 센다.

작업량이 다르다. 운영체제 교과서의 요청은 프로세스이고 수십 개다. 이 책의 요청은 생성 중인 시퀀스이고 수백 개이며, 길이가 미리 정해지지 않고 자란다. 자라는 요청이 수백 개 붙어 있으면 연속 할당은 반드시 빈틈을 만든다.

학습 목표

> 내부 단편화와 외부 단편화를 정의하고 구분할 수 있다. 같은 요청 열에서 연속 할당과 페이지 할당이 각각 어느 단편화를 얼마나 만드는지 실측으로 설명할 수 있다.

이 말부터

내부 단편화	요청에 배정된 자리 중 요청이 안 쓰는 부분. 그 요청 말고는 아무도 못 쓴다
외부 단편화	배정되지 않은 자리인데 조각이 작아 새 요청에 못 주는 부분. 총합은 넉넉한데 자리가 없다
첫 맞춤	빈 구간을 앞에서부터 보다가 들어가는 첫 곳에 넣는 연속 할당 규칙

디딤돌

메모리 하나를 요청 여럿이 나눠 쓰는 가장 단순한 방법은 연속 할당이다. 요청이 오면 그 길이만큼 붙은 구간을 준다. 1.1 의 연속성 불변식이 요청마다 성립하므로 읽기가 싸다.

정의 2.4: 연속 할당과 두 단편화

> 크기 T 의 메모리에 요청 r_i 가 길이 ℓ_i 의 연속 구간을 받는다. 요청에 배정된 구간의 크기를 $a_i \ge \ell_i$ 라 하면 내부 단편화는 $\sum (a_i - \ell_i)$ 다. 배정되지 않은 구간들 중 가장 큰 것을 g, 그 합을 F 라 하면 외부 단편화는 $1 - g/F$ 다. 빈 총합 중 한 요청이 한 번에 못 받는 비율이다.
>
> **불변식** 배정된 구간끼리 겹치지 않고, 각 구간은 T 안에 있다.

두 단편화는 원인이 다르다. 내부는 요청보다 크게 주었기 때문이고, 외부는 요청이 나간 자리가 다음 요청과 크기가 안 맞기 때문이다. 연속 할당은 요청한 만큼 정확히 주므로 내부 단편화가 0 이고, 대신 외부 단편화가 쌓인다.

예제 2.2: 빈틈이 생기는 순서

> 크기 100 의 메모리에 길이 30 인 요청 셋이 차례로 들어와 $[0, 30)$, $[30, 60)$, $[60, 90)$ 을 받는다. 둘째가 나가면 빈 구간은 $[30, 60)$ 과 $[90, 100)$ 이고 $F = 40$, $g = 30$ 이다. 이제 길이 40 인 요청이 오면 총합은 40 인데 줄 수 없다. 외부 단편화가 $1 - 30/40 = 0.25$ 다.

그림 2.3.. 예제 2.3 의 상태. 빈 자리가 40 인데 붙어 있지 않아 40 을 못 준다

균열

요청 2만 개를 두 할당자에 차례로 넣었다. 길이는 대부분 짧고 가끔 매우 긴 분포이고, 동시에 살아 있는 요청 수를 둘로 두어 평균 점유율을 60 퍼센트와 80 퍼센트로 만들었다. 연속 할당은 첫 맞춤이고, 페이지 할당은 다음 절의 방식이다. 여기서는 결과만 먼저 본다.

실험 2.2 같은 요청 열, 두 할당자, 두 부하

부하	할당자	실패	외부 퍼센트	내부 퍼센트
60	연속 첫 맞춤	15	27.6	0.00
60	페이지 16	4	0.00	0.47
60	페이지 256	16	0.00	7.39
80	연속 첫 맞춤	213	43.7	0.00
80	페이지 16	242	0.00	0.47
80	페이지 256	367	0.00	7.52

측정 조건. 자리 131,072 개, 요청 2만 개, 길이는 로그정규 분포 (평균 1,643, 최대 16,384). 실패는 자리를 못 받은 요청 수, 퍼센트는 요청마다 잰 단편화의 평균.

부하 60 퍼센트에서 연속 할당은 빈 공간의 28 퍼센트가 쓸 수 없는 조각이었고, 요청 15 개가 자리를 못 받았다. 페이지 16 은 요청 4 개만 실패했고, 그 대가로 배정된 자리의 0.5 퍼센트가 자투리로 남았다. 부하 80 퍼센트에서는 셋 다 많이 실패한다. 메모리 자체가 모자라는 자리에서는 할당 방법이 답이 아니다.

관찰

> 단편화가 갈리는 것은 부하가 중간일 때다. 아주 낮으면 무엇을 써도 되고, 아주 높으면 무엇을 써도 안 된다.

이 실험에서 걸리는 것이 하나 있다. 페이지 256 은 내부 단편화가 7 퍼센트인데 실패도 연속 할당보다 많다. 페이지가 크면 자투리가 커지고 자투리가 커지면 실제로 쓸 수 있는

메모리가 줄기 때문이다. 페이지 크기는 작을수록 좋아 보이지만, 다음 절에서 작은 페이지의 값이 따로 있다는 것을 본다.

다시 쓰기

외부 단편화를 없애는 방법은 하나뿐이다. 요청에 연속 구간을 주지 않는 것이다. 메모리를 같은 크기의 페이지로 자르고, 요청에 페이지 여러 개를 주되 붙어 있을 것을 요구하지 않는다. 그러면 빈 페이지는 어디 있든 같은 값이고, 조각이 작아서 못 주는 일이 없다.

정의 2.5: 페이지 할당

> 크기 T 의 메모리를 크기 p 의 페이지 T/p 개로 나눈다. 길이 ℓ 인 요청은 페이지 $\lceil \ell/p \rceil$ 개를 받고, 페이지는 어디 있어도 된다. 요청은 논리 순서와 물리 페이지 번호를 잇는 표를 든다.
>
> **불변식** 페이지 하나는 한 번에 한 요청에만 속한다. 외부 단편화는 정의에 의해 0 이다.

명제 2.6: 페이지 할당의 단편화

> 페이지 할당의 외부 단편화는 0 이고, 요청 하나의 내부 단편화는 $p-1$ 을 넘지 않는다. 요청 길이가 페이지 경계에 대해 고르게 분포하면 요청당 기대 내부 단편화는 $(p-1)/2$ 다.

증명. 빈 페이지는 어느 것이든 요청에 줄 수 있으므로 가장 큰 빈 구간이 빈 총합과 같고 $1 - g/F = 0$ 이다. 마지막 페이지의 빈자리는 ℓ 을 p 로 나눈 나머지가 정하고, 그 값은 0 부터 $p-1$ 사이다. 고르게 분포하면 평균은 $(p-1)/2$ 다. ■

이 명제가 실험 2.2 의 페이지 행을 설명한다. 페이지 16 은 요청당 평균 자투리가 7.5 자리이고 평균 길이가 1,643 이니 0.5 퍼센트다. 페이지 256 은 자투리가 128 자리로 7.8 퍼센트다. 잰 값 7.4 퍼센트가 그 근처다.

구현

두 할당자를 같은 인터페이스로 둔다. 실험 2.2 가 이 두 클래스를 같은 요청 열로 돌린 것이다.

지금 보는 코드 첫 맞춤 연속 할당. 빈 구간 목록을 앞에서부터 보고 들어가는 첫 곳에 넣는다.

구현 첫 맞춤 할당과 외부 단편화 `samples/dsai/alloc.py:23`

```
    def alloc(self, key: int, length: int) -> bool:
        for i, (start, size) in enumerate(self.free):
            if size >= length:
                self.used[key] = (start, length)
                if size == length:
                    self.free.pop(i)
                else:
                    self.free[i] = (start + length, size - length)
                return True
        return False
```

읽어 보면 빈 구간 목록을 앞에서부터 훑다가 길이가 맞는 첫 구간을 쪼개 준다. 나갈 때는 이웃한 빈 구간을 합친다. 합치지 않으면 빈 구간이 계속 잘게 쪼개져 외부 단편화가 더 빨리 쌓인다.

구현 페이지 할당과 내부 단편화 `samples/dsai/alloc.py:71`

```
    def pages_for(self, length: int) -> int:
        return -(-length // self.page)

    def alloc(self, key: int, length: int) -> bool:
        need = self.pages_for(length)
        if need > len(self.free_pages):
            return False
        self.used[key] = [self.free_pages.pop() for _ in range(need)]
        self.lengths[key] = length
        return True
```

페이지 할당의 `alloc` 은 빈 페이지 목록에서 필요한 수만큼 꺼낼 뿐이다. 어느 페이지인지 보지 않는다. 그것이 정의 2.5 의 요점이고, 그래서 코드가 연속 할당보다 짧다.

분석

단편화라는 개념은 그대로 유효하다. Denning 이 1970 년에 가상 메모리를 정리하면서 세그먼트와 페이지를 정확히 이 두 단편화로 견줬다 [확립] [11]. 바뀐 것은 요청의 모양이다. 그때의 요청은 크기가 정해진 프로그램이었고, 지금의 요청은 끝을 모르고 자라는 시퀀스다.

자라는 요청은 연속 할당과 특히 안 맞는다. 처음에 얼마를 줄지 모르니 최대치를 주면 내부 단편화가 되고, 적게 주면 자랄 때 옮겨야 한다. 2.1 의 두 대가가 여기서 동시에 나온다. 그래서 다음 절의 KV 캐시는 이 두 대가를 한 번에 치르는 구조이고, 2.4 이 페이지로 둘을 한 번에 없앤다.

주의

> 외부 단편화 0 은 공짜가 아니다. 페이지 할당은 요청마다 표 하나를 들고, 읽을 때마다 표를 거친다. 그 값은 2.4 에서 센다.

되짚기

> 내부와 외부 단편화를 정의했고, 연속 할당은 외부를, 페이지 할당은 내부만을 만든다는 것을 명제와 실측으로 봤다. 다음 절에서 이 두 단편화가 동시에 나오는 구조, 생성 모델의 KV 캐시를 세운다.

2.3 KV 캐시라는 추가 전용 배열

2.1 과 2.2 가 세운 두 대가, 옮기는 값과 빈자리가 한 구조에서 동시에 나오는 자리가 있다. 생성 모델이 토큰을 하나씩 낼 때 지금까지의 키와 값을 저장해 두는 KV 캐시다. 이 절은 그 구조를 자료 구조의 말로 정의하고, 크기를 세고, 세 가지 담는 법의 빈자리를 잰다.

이 책은 어텐션의 수학을 15장에서야 세운다. 여기서는 그 수학 중 자료 구조에 필요한 것 하나만 쓴다. 토큰 하나가 층마다 키 벡터 하나와 값 벡터 하나를 남기고, 다음 토큰을 낼 때 지금까지 남긴 전부를 읽는다는 것이다.

학습 목표

KV 캐시를 요청마다 하나씩 있는 추가 전용 값 배열로 정의하고, 토큰당 바이트를 모델 크기에서 계산할 수 있다. 최대 길이 예약, 두 배 늘리기, 블록 할당 세 방식의 빈자리를 같은 길이 분포에서 비교할 수 있다.

이 말부터

KV 캐시	토큰마다 남긴 키와 값 벡터를 요청이 끝날 때까지 쌓아 두는 배열. 요청마다 하나씩이다
추가 전용	끝에 붙이기만 하고 중간을 고치지 않는 배열. 로그 파일과 같은 성질이다
디코딩	토큰을 하나씩 내는 단계. 토큰 하나마다 캐시 전체를 한 번 읽는다

디딤돌

디딤돌은 2.1 의 동적 배열 그대로다. 다만 원소가 스칼라가 아니라 층 수 곱하기 헤드 수 곱하기 차원의 벡터이고, 배열이 요청 수만큼 있다.

정의 2.7: KV 캐시

층 L, 헤드 H, 헤드 차원 d_h 인 모델에서 요청 r 의 KV 캐시는 길이 n_r 의 추가 전용 배열이다. 원소 t 는 토큰 t 의 키와 값 전부이고 크기가 $2LHd_h$ 개의 실수다. 연산은 추가 (토큰 하나) 와 전체 읽기 (다음 토큰을 낼 때) 둘뿐이다.

불변식 원소는 한 번 쓰이면 바뀌지 않고, 읽기는 언제나 0 부터 $n_r - 1$ 까지 전부다.

두 연산뿐이라는 것이 이 구조를 단순하게도, 어렵게도 만든다. 중간 삽입이 없으니 연속일 필요가 없고, 그래서 2.2 의 페이지가 들어올 수 있다. 전체 읽기가 토큰마다 있으니 이동량이 토큰 수의 제곱이 되고, 그래서 캐시가 가속기 메모리에 있어야 한다.

토큰 하나의 크기부터 센다.

명제 2.8: 토큰당 캐시 바이트

원소 하나가 s 바이트이면 토큰 하나가 캐시에 남기는 바이트는 $2LHd_hs$ 다. 길이 n 인 요청의 캐시는 $2LHd_hsn$ 바이트이고, 토큰 하나를 낼 때 읽는 바이트도 그만큼이다.

예제 2.3: 40 층 모델의 토큰 하나

$L = 40, H = 40, d_h = 128, s = 2$ (float16) 이면 토큰 하나가 $2 \times 40 \times 40 \times 128 \times 2 = 819{,}200$ 바이트다. 800KB 다. 길이 2,048 인 요청 하나가 1.6GB 이고, 그런 요청이 열 개 살아 있으면 16GB 다. 같은 모델의 가중치가 26GB 안팎이니, 캐시가 가중치와 같은 급의 메모리를 차지한다.

이 예제의 숫자는 이 책이 계산한 것이고, 같은 급의 모델에서 캐시가 메모리의 큰 몫을 차지한다는 것은 공개 논문이 같은 산수로 적었다 [확립] [9]. 캐시가 크다는 것보다 중요한 것은 캐시의 길이를 미리 모른다는 것이다. 요청이 몇 토큰에서 끝날지는 모델이 정한다.

균열

길이를 모르는 배열 수천 개를 담는 법 셋을 같은 길이 분포로 비교했다. 최대 길이만큼 미리 잡기, 두 배 늘리기, 그리고 블록으로 나누기다. 마지막 것은 다음 절에서 정의하고 여기서는 결과만 놓는다.

실험 2.3 길이를 모르는 배열 이천 개, 세 가지 담는 법

방식	블록	예약 대비 사용	옮긴 비율
최대 길이 예약	0	7.49	0.00
두 배 늘리기	0	1.43	1.42
블록 1	1	1.00	0.00
블록 16	16	1.01	0.00
블록 64	64	1.06	0.00
블록 256	256	1.23	0.00

측정 조건. 요청 2,000 개, 최대 길이 4,096, 길이는 로그정규 분포. 예약 대비 사용은 예약한 자리를 실제 쓴 자리로 나눈 것, 옮긴 비율은 옮긴 자리를 쓴 자리로 나눈 것.

최대 길이를 미리 잡으면 쓴 자리의 7.5 배를 예약한다. 요청 대부분이 최대보다 훨씬 짧기 때문이다. 두 배 늘리기는 1.43 배를 예약하고 그 대신 쓴 자리의 1.4 배를 옮긴다. 2.1 의 두 대가가 그대로 나온다. 블록 16 은 1.01 배를 예약하고 아무것도 옮기지 않는다.

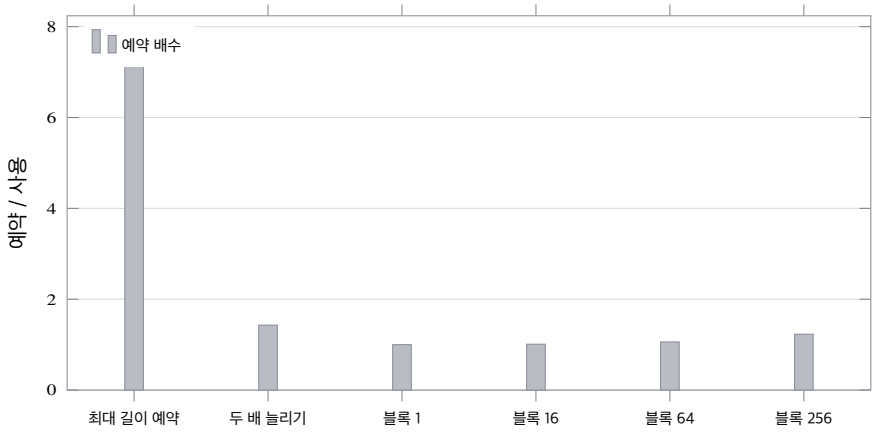


그림 2.4.. 같은 길이 분포에서 세 방식이 예약한 자리. 1.0 이 낭비 없음이다

관찰

최대 길이 예약의 7.5 배는 이 분포에서 나온 숫자다. 분포가 더 길게 꼬리를 늘이면 배수는 더 커진다. 낭비의 크기는 구조가 아니라 길이 분포가 정한다.

다시 쓰기

이 절의 다시 쓰기는 정의 2.7 의 읽기 연산을 루프라인에 놓는 것이다. 토큰 하나를 낼 때 캐시 전체를 읽고, 읽은 벡터마다 곱 하나 합 하나를 한다. 0.2 의 내적과 같은 밀도 0.25 다.

명제 2.9: 디코딩의 연산 밀도

길이 n 인 요청 하나의 디코딩 한 단계는 캐시 $2LHd_hsn$ 바이트를 읽고 $2 \cdot 2LHd_hn$ 번 계산한다. 밀도는 $2/s$ 다. 요청 B 개를 배치로 묶어도 요청마다 캐시가 다르므로 밀도는 그대로 $2/s$ 다.

증명. 키 벡터마다 질의와의 내적, 값 벡터마다 가중합이므로 원소당 곱 하나 합 하나씩 두 번이다. 바이트는 원소당 s 다. 배치는 1.4 의 명제에서 같은 행렬을 여러 질의가 나눠 읽을 때만 밀도를 올린다. 캐시는 요청마다 다르므로 나눠 읽을 것이 없다. ■

이 명제가 1.4 의 주의 상자가 가리킨 자리다. 배치가 밀도를 못 올리는 작업량이 여기 있다. 디코딩은 밀도 $2/s$ 에서 벗어나지 못하고, float16 이면 1 이다. 능선점의 8 분의 1 이다. 그래서 디코딩의 시간은 계산이 아니라 캐시를 읽는 이동량이 정하고, 캐시를 줄이는 것이 곧 디코딩을 빠르게 하는 것이 된다.

구현

정의 2.7 를 가장 단순하게 구현한 것이 최대 길이 예약이다. 실험 2.3 의 첫 행이 이것이다.

지금 보는 코드 요청마다 최대 길이를 미리 잡는 캐시. 추가는 자리에 쓰는 것, 읽기는 앞 length 개를 자르는 것이다.

구현 토큰당 바이트 `samples/dsai/kvcache.py:14`

```
def bytes_per_token(layers: int, heads: int, head_dim: int, elem_bytes: int) -> int:
    """정의 2.3. 토큰 하나가 캐시에 남기는 바이트. 키와 값이라 2 를 곱한다."""
    return 2 * layers * heads * head_dim * elem_bytes
```

구현 최대 길이 예약 캐시 `samples/dsai/kvcache.py:20`

```
class ContiguousKV:
    """요청마다 최대 길이만큼 미리 잡는 방식. 남는 자리가 전부 낭비다."""

    max_len: int
    dim: int
    buf: np.ndarray = field(init=False)
    length: int = 0

    def __post_init__(self) -> None:
        self.buf = np.zeros((self.max_len, self.dim), dtype=np.float32)

    def append(self, kv: np.ndarray) -> None:
        self.buf[self.length] = kv
        self.length += 1

    def reserved(self) -> int:
        return self.max_len

    def gather(self) -> np.ndarray:
        return self.buf[:self.length]
```

읽어 보면 `bytes_per_token` 이 명제 2.8 의 식이다. `ContiguousKV` 는 `max_len` 행을 미리 잡고 `length` 까지만 쓴다. `gather` 가 정의 2.7 의 전체 읽기이고, 연속이라 1.2 의 자르기 뷰 하나로 끝난다.

분석

추가 전용 배열은 전통 교과서에 이름이 없다. 로그 파일이나 스트림으로 다른 과목에 있었다. 그런데 이 구조가 지금 가속기 메모리의 절반을 차지한다. 이름이 없던 구조가 한가운데에 온 것이다.

이 절이 보인 것은 셋이다. 캐시는 토큰당 800KB 급으로 크고, 길이를 미리 모르며, 읽기 밀도가 낮아 크기가 곧 시간이다. 셋을 합치면 요구가 하나로 모인다. 빈자리 없이, 옮기지 않고, 읽기는 연속에 가깝게. 다음 절이 그 셋을 한 번에 만족하는 구조다.

현장에서

> Pope 와 동료들은 2023 년에 대형 모델 추론의 지연과 처리량을 캐시 읽기의 이동량으로 분석하는 모델을 세웠고, 디코딩이 메모리 대역폭에 묶인다는 것을 같은 산수로 보였다 [확립] [12].

되짚기

KV 캐시를 요청마다 하나인 추가 전용 배열로 정의했고, 토큰당 바이트와 디코딩 밀도 $2/s$ 를 세웠다. 최대 길이 예약은 7.5 배를 낭비하고 두 배 늘리기는 1.4 배를 옮긴다. 다음 절에서 페이지로 둘 다 없앤다.

2.4 페이지드 어텐션, 블록 테이블과 공유

2.1 에서 두 배 늘리기의 빈자리를 셌고, 2.2 에서 페이지 할당이 외부 단편화를 없애는 것을 봤고, 2.3 에서 KV 캐시가 그 둘을 다 필요로 한다는 것을 봤다. 이 절은 셋을 합친다. KV 캐시를 고정 크기 블록으로 나누고, 요청마다 논리 블록에서 물리 블록으로 가는 표를 들게 한다. 운영체제의 페이지 테이블을 어텐션 안으로 들여온 것이라 페이지드 어텐션이라 부른다.

이 구조가 주는 것은 빈자리 없음만이 아니다. 표가 있으면 두 요청이 같은 물리 블록을 가리킬 수 있다. 같은 시스템 프롬프트로 시작하는 요청 백 개가 접두사 블록을 하나만 두고 나눠 읽는다. 그 절약을 재는 것이 이 절의 둘째 실험이다.

학습 목표

> 블록 테이블을 정의하고, 추가와 읽기가 표를 어떻게 거치는지 알고리즘으로 적을 수 있다. 접두사 공유가 참조 계수와 쓰기 전 복사로 어떻게 안전해지는지 설명하고, 절약 배수를 계산할 수 있다.

이 말부터

> | | |
> |---|---|
> | 블록 테이블 | 요청의 논리 블록 번호를 물리 블록 번호로 바꾸는 표. 운영체제의 페이지 테이블과 같은 물건이다 |
> | 참조 계수 | 물리 블록 하나를 몇 요청이 가리키는지 센 수. 0 이 되면 풀로 돌아간다 |
> | 쓰기 전 복사 | 공유 중인 블록에 쓰려 할 때 먼저 내 것으로 복사하는 규칙. 영어로 copy-on-write 다 |

디딤돌

디딤돌은 2.2 의 정의 2.5 다. 거기서 요청이 든 표라고만 적은 것을 정확히 세운다.

정의 2.10: 블록 테이블

> 블록 크기 p 인 물리 블록 풀 P 가 있다. 요청 r 의 블록 테이블은 열 $T_r = (b_0, b_1, \dots, b_{m-1})$ 이고, $b_j \in P$ 는 논리 블록 j 가 놓인 물리 블록이다. 토큰 t 는 논리 블록 $\lfloor t/p \rfloor$ 의 $t \bmod p$ 번째 자리에 있다. 즉 물리 주소는 $b_{\lfloor t/p \rfloor} \cdot p + (t \bmod p)$ 다.
>
> **불변식** $m = \lceil n_r / p \rceil$ 이고, 마지막 논리 블록만 덜 차 있다. 물리 블록마다 참조 계수가 그 블록을 든 표의 수와 같다.

주소 식을 1.1 의 배열 주소 식과 견주면 다른 것이 하나뿐이다. 곱하는 자리에 표 조회가 하나 끼었다. 그 조회 하나가 연속성을 포기한 값이고, 그 대가로 2.2 의 외부 단편화가 0 이 된다.

예제 2.4: 블록 4 와 토큰 10

> $p = 4$ 이고 요청이 토큰 10 개를 냈으면 논리 블록은 셋이고, 표가 $(7, 3, 9)$ 라 하자. 토큰 5 는 논리 블록 1 의 자리 1 이므로 물리 블록 3 의 자리 1, 주소 $3 \times 4 + 1 = 13$ 에 있다. 마지막 블록 9 는 자리 둘만 찼고, 둘이 비어 있다. 내부 단편화가 2 다.

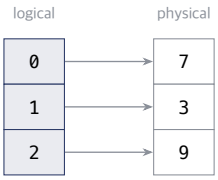


그림 2.5.. 예제 2.6 의 블록 테이블. 논리 순서는 붙어 있고 물리 블록은 흩어져 있다

균열

이 절의 균열은 앞 절에서 이미 쟀다. 실험 2.3 의 블록 행이 그것이고, 블록 16 은 예약 1.01 배에 옮기기 0 이었다. 남은 물음은 블록 크기다. 작을수록 자투리가 줄지만 표가 길어지고 읽기가 잘게 끊긴다.

명제 2.6 로 자투리를 세면 블록 16 은 토큰 7.5 개, 블록 256 은 128 개다. 예제 2.3 의 모델에서 토큰 하나가 800KB 이니 블록 256 의 자투리는 요청당 100MB 다. 요청 백 개면 10GB 다. 표의 길이는 반대로 간다. 길이 4,096 인 요청이 블록 16 이면 항목 256 개, 블록 256 이면 16 개다. 항목 하나가 정수 하나이니 어느 쪽도 캐시에 견주면 무시할 크기다.

실험 2.4 접두사를 공유하는 요청 묶음의 저장량

접두사	접미사	요청 수	따로 저장	공유 저장	절약 배
1024	128	8	9216	2048	4.50
1024	128	64	73728	9216	8.00
256	256	64	32768	16640	1.97

측정 조건. 블록 16. 접두사 길이만큼은 모든 요청이 같고, 접미사는 요청마다 다르다. 단위는 토큰 자리 수.

접두사 1,024 에 접미사 128 인 요청 64 개는 따로 저장하면 73,728 자리이고 공유하면 9,216 자리다. 8 배다. 접두사가 짧고 접미사가 길면 절약이 2 배로 준다. 절약 배수는 $(P + S)/(P/K + S)$ 에 가깝고, K 가 크고 S 가 작을수록 P/S 에 다가간다.

관찰

> 접두사 공유의 절약은 구조가 아니라 작업량에서 온다. 같은 시스템 프롬프트를 쓰는 요청이 많은 서비스에서만 8 배가 나온다.

다시 쓰기

추가와 읽기와 공유를 알고리즘으로 적는다. 셋 다 표를 거치고, 공유는 참조 계수로 안전해진다.

알고리즘 2.2 페이지드 캐시에 토큰 추가 `dsai.kvcache.PagedKV.append`

입력: 블록 테이블 T, 길이 n, 블록 크기 p, 풀 P, 새 키 값 kv

만약 $n \bmod p = 0$ **이면**
    T 에 P 에서 꺼낸 새 블록을 붙인다 ▷ 참조 계수 1
끝 만약
$b \leftarrow T[\lfloor n/p \rfloor]$
만약 참조 계수$(b) > 1$ **이면**
    $b' \leftarrow P$ 에서 새 블록; b' 에 b 를 복사; 참조 계수(b) 를 하나 줄임; $T[\lfloor n/p \rfloor] \leftarrow b'$; $b \leftarrow b'$
끝 만약
$P[b][n \bmod p] \leftarrow kv$; $n \leftarrow n + 1$

둘째 조건문이 쓰기 전 복사다. 공유 중인 블록에 쓰면 다른 요청의 캐시가 바뀌므로, 쓰기 전에 내 것을 만든다. 복사는 마지막 블록 하나뿐이고 앞의 블록들은 계속 공유된다. 실험 2.4 의 절약이 여기서 나온다.

정리 2.11: 페이지드 캐시의 비용

블록 크기 p 의 페이지드 캐시에서 추가 한 번의 비용은 상수이고 옮기기는 없다. 단, 공유 블록에 처음 쓸 때만 블록 하나 p 개 원소를 복사한다. 길이 n 을 전부 읽는 이동량은 연속 캐시와 같은 $\lceil n \cdot 2LHd_h s/B \rceil$ 라인에 표 조회 $\lceil n/p \rceil$ 번이 더해진다. 요청당 빈자리는 $p - 1$ 이하이고 외부 단편화는 0 이다.

증명. 추가는 표 끝 항목 하나를 읽고 자리 하나에 쓴다. 자라는 일이 없으니 2.1 의 옮기기가 없다. 쓰기 전 복사는 블록 하나에 한정된다. 읽기는 블록마다 연속 p 개 원소를 읽으므로 블록 안에서는 연속 배열과 같고, 블록 사이에서 표를 한 번 본다. 블록이 라인보다 크면 라인 수는 연속과 같다. 빈자리와 외부 단편화는 명제 2.6 다. ■

구현

블록 풀과 페이지드 캐시다. 풀이 참조 계수를 들고, 캐시가 표를 든다.

지금 보는 코드 참조 계수를 든 블록 풀과, 그 풀 위의 페이지드 캐시. 실험 2.3 과 2.4 가 부른 클래스다.

구현 블록 꺼내기, 공유, 반납 `samples/dsai/kvcache.py:59`

```
    def take(self) -> int:
        b = self.free.pop()
        self.refcount[b] = 1
        return b

    def share(self, b: int) -> None:
        self.refcount[b] += 1

    def drop(self, b: int) -> None:
        self.refcount[b] -= 1
        if self.refcount[b] == 0:
            self.free.append(b)

    def used_blocks(self) -> int:
        return self.num_blocks - len(self.free)
```

읽어 보면 `take` 는 빈 블록을 꺼내 참조 계수를 1 로 두고, `share` 는 계수를 올리고, `drop` 은 내려서 0 이 되면 빈 목록에 되돌린다. 블록의 생애가 이 세 함수로 끝난다.

구현 추가와 쓰기 전 복사 `samples/dsai/kvcache.py:84`

```
    def append(self, kv: np.ndarray) -> None:
        """알고리즘 2.2. 마지막 블록이 찼을 때만 새 블록을 받는다."""
        bs = self.pool.block_size
        if self.length % bs == 0:
            self.table.append(self.pool.take())
        block = self.table[-1]
        if self.pool.refcount[block] > 1:
            block = self._copy_on_write(len(self.table) - 1)
        self.pool.store[block, self.length % bs] = kv
        self.length += 1

    def _copy_on_write(self, logical: int) -> int:
        """공유 중인 블록에 쓰려면 먼저 내 것으로 복사한다."""
        old = self.table[logical]
        new = self.pool.take()
        self.pool.store[new] = self.pool.store[old]
        self.pool.drop(old)
        self.table[logical] = new
        return new
```

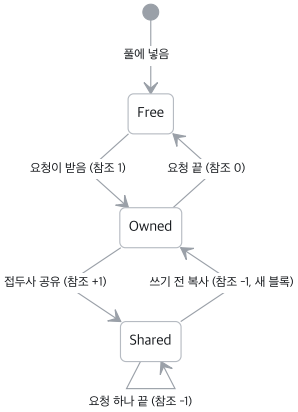


그림 2.6.. 물리 블록 하나의 상태. 공유 중에 쓰기가 오면 복사가 일어나고 원래 블록은 계수 하나가 준다

읽기는 표를 따라 블록을 모아 논리 순서로 늘어놓는다. 1.2 의 뷰로는 안 된다. 블록이 흩어져 있어 스트라이드 하나로 적을 수 없기 때문이다. 그래서 `gather` 는 표를 인덱스 배열로 써서 블록을 모은다. 0.1 의 `gather_sum` 과 같은 모양이고, 블록이 라인보다 커서 이동량은 연속과 같다.

분석

이 절의 판정을 낸다. 동적 배열은 조정해서 유효하다. 정의는 남고 구현이 바뀌었다. 자라는 배열이 하나일 때는 두 배 늘리기가 답이고, 수백 개가 메모리 하나를 나눌 때는 블록 테이블이 답이다. 바뀐 것은 배열이 아니라 배열의 수다.

현장에서

> Kwon 과 동료들의 2023 년 논문은 기존 추론 서버가 KV 캐시의 60 에서 80 퍼센트를 예약과 단편화로 낭비했다고 재고, 블록 테이블로 그것을 4 퍼센트 아래로 내렸다고 보고했다 [확립] [9]. 접두사 공유도 같은 논문의 참조 계수와 쓰기 전 복사로 구현되어 있다.

이 구조가 새로운 것은 아니다. 표 조회 하나를 끼워 연속성을 포기하고 단편화를 없애는 것은 Denning 이 1970 년에 정리한 페이징 그대로다. 새로운 것은 그 페이징이 운영체제가 아니라 어텐션 커널 안에 들어왔다는 것이고, 그렇게 된 이유는 2.3 의 명제 2.9 다. 읽기 밀도가 낮아 캐시 크기가 곧 시간이므로, 빈자리 하나가 시간 하나가 된다.

주의

> 블록 크기는 라인보다 커야 정리 2.11 의 읽기 이동량이 연속과 같아진다. 블록이 토큰 하나면 자투리는 0 이지만 읽기가 무작위 접근이 된다.

되짚기

> 블록 테이블을 정의했고, 추가는 옮기기 없이 상수이고 읽기는 연속과 같은 이동량임을 증명했다. 접두사 공유가 8 배를 절약했다. 다음 장에서 배열의 원소가 대부분 0 일 때, 즉 희소 행렬에서 연속성을 어떻게 다시 정의하는지 본다.

2장을 닫으며

이 장이 세운 구조는 블록 테이블 하나다. 앞의 세 절은 그 구조가 왜 필요한지를 하나씩 쌓았다. 두 배 늘리기는 옮긴 총수를 $2n$ 아래로 묶지만 빈자리를 $(\alpha - 1)n$ 까지 남기고, 그 빈자리는 배열이 수백 개일 때 메모리의 4분의 1이 된다. 연속 할당은 외부 단편화를 만들고, 요청 길이를 모르는 자리에서는 내부 단편화까지 같이 만든다. KV 캐시가 정확히 그 자리이고, 블록 테이블이 표 조회 하나를 값으로 치르고 둘 다 없앤다.

네 실험의 숫자를 적어 둔다. 백만 개를 넣는 동안 두 배 늘리기는 4MB 를 옮겼고 1024 씩 더하기는 2GB 를 옮겼다. 부하 60 퍼센트에서 연속 할당은 빈 공간의 28 퍼센트를 못 쓰는 조각으로 남겼고, 페이지 16 은 0.5 퍼센트를 자투리로 남겼다. 같은 길이 분포에서 최대 길이 예약은 7.5 배, 두 배 늘리기는 1.43 배, 블록 16 은 1.01 배를 예약했다. 접두사 1,024 를 나눠 쓰는 요청 64 개는 8 배를 절약했다.

정정 하나를 남긴다. 단편화 실험을 처음에는 메모리를 크게 잡고 돌렸다. 연속 할당의 실패가 0 이고 단편화가 2 퍼센트였다. 부하가 10 퍼센트였기 때문이다. 부하를 60 과 80 으로 올려 다시 쟀고, 그제야 두 할당자가 갈렸다. 단편화는 부하의 함수이지 할당자의 성질이 아니라는 것을, 실험이 먼저 가르쳤다.

옛 구조의 판정

> 동적 배열은 조정해서 유효하다. 정의와 분할 상환 논증은 그대로이고, 구현이 두 배 늘리기에서 블록 테이블로 바뀌었다. 근거는 실험 2.3 이다. 두 배 늘리기는 1.43 배 예약에 1.4 배 옮기기, 블록 16 은 1.01 배 예약에 옮기기 0. 바뀐 것은 배열의 수다. 하나일 때는 옛 답이 맞고, 수백 개가 메모리 하나를 나눌 때는 새 답이 맞는다.

이 장의 요약

> 1. 두 배 늘리기의 옮긴 총수는 $2n$ 이하이고, 성장 인자는 옮기는 값과 빈자리를 맞바꾼다 2.1
> 2. 연속 할당은 외부 단편화를, 페이지 할당은 내부 단편화만을 만든다. 갈리는 것은 중간 부하에서다 2.2
> 3. KV 캐시는 요청마다 하나인 추가 전용 배열이고, 토큰당 $2LHd_hs$ 바이트, 디코딩 밀도는 $2/s$ 다 2.3
> 4. 블록 테이블은 옮기기 없이 상수 추가, 연속과 같은 읽기 이동량, 외부 단편화 0 을 준다 2.4
> 5. 참조 계수와 쓰기 전 복사로 접두사를 나눠 쓰면 절약이 P/S 배에 다가간다 2.4
>
> 다음 장에서 뒤집히는 것: 배열의 원소가 대부분 0 이면 연속으로 두는 것 자체가 낭비다. 연속성을 0 이 아닌 원소에 대해서만 다시 정의한다.

필자의 판단

> 필자는 이 장을 운영체제 교과서의 반복으로 읽지 않기를 바란다. 페이징은 같지만 이유가 다르다. 운영체제의 페이징은 프로그램을 물리 메모리보다 크게 만들려는 것이었고, 여기서는 읽기 밀도가 낮은 배열 수백 개의 빈자리를 없애려는 것이다. 같은 구조가 다른 이유로 다시 쓰이는 것이 이 책이 말하는 다시 쓰기다.

열린 문제

하나, 실험 2.2 의 길이 분포는 로그정규 하나다. 실제 서비스의 길이 분포는 시간에 따라 바뀌고, 그 변화가 단편화를 어떻게 움직이는지는 재지 않았다.
둘, 정리 2.4 의 표 조회 비용을 상수라고 적었지만, 표가 가속기 메모리에 있을 때 조회 하나의 실제 값은 이 책의 기계로 잴 수 없다.
셋, 쓰기 전 복사가 블록 하나에 그친다고 했는데, 접미사가 블록 경계를 넘어가며 여러 블록을 순서대로 복사하는 경우의 총비용은 접미사 길이의 함수다. 그 함수를 적지 않았다.

연습문제

2.1 기초 처음 용량 8, 성장 인자 2 인 동적 배열에 원소 100 개를 넣을 때 자라는 횟수와 옮긴 원소의 총수를 구하라. 정리 2.1 의 상한과 비교하라.

2.2 기초 예제 2.3 의 상황에서 셋째 요청까지 나간 뒤 길이 60 인 요청이 오면 받을 수 있는가. 빈 구간을 합치는 규칙이 없다면 어떻게 되는가.

2.3 기초 $L = 32$, $H = 32$, $d_h = 128$, float16 인 모델의 토큰당 캐시 바이트를 구하고, 캐시에 24GB 를 쓸 수 있을 때 살아 있을 수 있는 토큰 수를 구하라.

2.4 기초 블록 크기 16 과 64 에서 길이 1,000 인 요청의 논리 블록 수, 자투리, 표의 항목 수를 각각 구하라.

2.5 응용 `dsai.dynarray.total_copied` 로 성장 인자 1.125, 1.5, 2, 4 에서 원소 십만 개를 넣을 때의 옮긴 총수를 구하고, 정리 2.1 의 상한 $\alpha n/(\alpha - 1)$ 과 한 표에 놓아라.

2.6 응용 `dsai.alloc.ContiguousArena` 를 상속해 최적 맞춤 (가장 작은 빈 구간에 넣기) 할당자를 만들고, 실험 2.2 의 요청 열에서 첫 맞춤과 실패 수를 비교하라.

2.7 응용 `dsai.kvcache.bytes_per_token` 으로 주어진 메모리에 드는 토큰 수를 구하는 함수를 쓰고, 키 값 헤드를 40 개에서 8 개로 줄이면 토큰 수가 몇 배가 되는지 구하라.

2.8 응용 블록 크기 16 과 128 에서 같은 요청 200 개의 자투리 비율과 표 항목 수의 합을 `dsai.kvcache.PagedKV` 로 재고, 명제 2.5 의 기대값 $(p - 1)/2$ 과 대조하라.

2.9 탐구 블록 테이블을 두 단계로 만들면 (표의 표) 요청 하나의 표 크기가 어떻게 달라지고 조회 비용이 어떻게 달라지는가. 운영체제의 다단계 페이지 테이블과 견줘 논하라.

2.10 탐구 접두사 공유는 접두사가 토큰 단위로 정확히 같을 때만 된다. 접두사가 블록 중간에서 갈리는 요청들에 대해 공유를 최대로 하는 블록 경계 정렬 방법을 설계하라.

더 읽을 것

[10] 분할 상환 분석의 원전. 잠재 함수 방법이 여기서 나왔다. 정리 2.1 의 증명을 잠재 함수로 다시 써 보면 좋다. 2.1
[11] 가상 메모리와 페이징의 고전. 세그먼트와 페이지를 두 단편화로 견준 표가 이 장의 2.2 절이다. 2.2
[9] 페이지드 어텐션의 원전. 낭비 측정, 블록 테이블, 참조 계수, 쓰기 전 복사가 전부 이 한 편에 있다. 2.3, 2.4

> [12] 추론의 지연과 처리량을 이동량으로 분석한 모델. 명제 2.7 의 디코딩 밀도를 가속기 규모에서 다룬다. 2.3

3장

희소 행렬과 저정밀 표현

2021년 가속기 제조사의 논문은 행렬 곱 장치에 네 칸 중 두 칸이라는 규칙을 직접 읽는 회로를 넣어 밀집의 두 배 처리량을 냈다고 적었다 [확립] [13]. 같은 해 다른 쪽에서는 가중치를 4비트로 적고 스케일을 64개마다 하나 두는 방법으로 65B 모델을 48GB에 넣었다 [확립] [14]. 둘 다 값 배열의 정의를 바꾼 것이 아니다. 원소를 몇 개 저장할지, 원소 하나를 몇 비트로 적을지를 바꿨다.

이 장은 0.2의 둘째 가정을 마저 건드린다. 원소가 스칼라라는 가정은 1장에서 깨졌고, 여기서는 원소가 전부 저장할 가치가 있다는 가정이 깨진다. 대부분이 0이면 0이 아닌 것만 저장하고, 정밀도가 남으면 비트를 깎고, 토큰마다 필요한 가중치가 다르면 필요한 것만 읽는다. 셋 다 이동량을 줄이는 길이고, 셋 다 대가가 있다. 인덱스, 오차, 그리고 배치의 이득이다.

이 장에서 세우는 것

3.1 COO, CSR, CSC를 정의하고, CSR이 이기는 밀도가 50퍼센트가 아니라 2퍼센트인 이유를 잰다

3.2 블록 희소와 2:4. 규칙이 인덱스와 라인을 아끼고, 2:4의 시간 이득은 장치가 있어야 한다

3.3 블록 양자화. 스케일을 블록마다 두면 이상값의 영향이 갇혀 오차가 500배 준다

3.4 전문가 혼합. 배정은 희소 행렬이고, 읽는 전문가 수의 기대값이 밀도를 정한다

3.1 COO, CSR, CSC. 세 가지 압축과 각각 빠른 연산

1.1 의 연속성 불변식은 모든 원소를 붙여 놓는다. 원소 대부분이 0 이면 그 불변식이 0 을 붙여 놓는 데 메모리를 쓴다. 0 은 곱해도 0 이고 더해도 그대로이니, 0 을 옮기는 이동량은 전부 낭비다. 이 절은 연속성을 0 이 아닌 원소에 대해서만 다시 정의한다. 희소 행렬이다.

세 가지 표현을 세우고, 그중 CSR 로 행렬 벡터 곱을 잰다. 답은 희소 행렬이 항상 이긴다가 아니다. 어느 밀도부터 밀집이 이기는지가 이 절의 숫자이고, 그 문턱이 교과서가 말하던 것보다 훨씬 낮다.

학습 목표

COO, CSR, CSC 를 정의하고 같은 행렬을 세 표현으로 손으로 적을 수 있다. CSR 행렬 벡터 곱의 연산 수와 이동량을 세고, 밀집 곱과의 교차 밀도를 실측으로 설명할 수 있다.

이 말부터

nnz	0 이 아닌 원소의 수. number of nonzeros 의 약자이고 현장에서 그대로 쓴다
밀도	nnz 를 전체 원소 수로 나눈 것. 희소 행렬의 값은 전부 이 숫자에 달렸다
행 포인터	행마다 그 행의 원소가 값 배열의 어디서 시작하는지 적은 배열. 길이가 행 수 더하기 1 이다

디딤돌

희소 행렬은 대부분이 0 인 행렬이다. 그런데 대부분이 얼마인지는 정의에 없다. 이 책은 그 문턱을 실험으로 정한다.

정의 3.1: 희소 행렬과 COO

$m \times n$ 행렬 A 에서 0 이 아닌 원소의 수를 $\mathrm{nnz}(A)$ 라 한다. COO 표현은 길이 nnz 의 배열 셋 (r, c, v) 이고, k 번째 원소는 $A[r_k, c_k] = v_k$ 다. 순서에는 제약이 없다.

COO 는 만들기가 가장 쉽고 쓰기는 가장 어렵다. 행 i 의 원소를 찾으려면 r 전체를 훑어야 한다. 그래서 행별로 접는다. 행 번호로 정렬하면 같은 행이 붙고, 붙어 있으면 행마다 시작 위치만 적으면 된다.

정의 3.2: CSR 과 CSC

CSR 은 배열 셋 $(ptr, cols, vals)$ 다. ptr 은 길이 $m+1$ 이고, 행 i 의 원소는 $cols$ 와 $vals$ 의 $[ptr[i], ptr[i+1])$ 구간에 있다. CSC 는 같은 것을 열에 대해 한 것이다. A 의 CSC 는 A^{T} 의 CSR 과 같다.

불변식 $ptr[0] = 0$, $ptr[m] = \mathrm{nnz}$, ptr 은 감소하지 않는다. 행 i 의 원소가 값 배열에 연속으로 있다.

불변식의 마지막 문장이 1.1 의 연속성이다. 행렬 전체가 아니라 행 하나 안에서만 연속이고, 그 대신 0 을 저장하지 않는다. 행 하나를 읽는 것은 자르기 하나이고, 열 하나를 읽는 것은 전체를 훑는 것이다. CSC 는 그 반대다.

예제 3.1: 4×4 행렬의 CSR

A 의 0 이 아닌 원소가 $(0,1) = 5$, $(1,0) = 3$, $(1,3) = 2$, $(3,2) = 7$ 뿐이라 하자. $\mathrm{nnz} = 4$ 다. 행별로 세면 행 0 에 하나, 행 1 에 둘, 행 2 에 없음, 행 3 에 하나이므로 $ptr = (0, 1, 3, 3, 4)$ 다. $cols = (1, 0, 3, 2)$, $vals = (5, 3, 2, 7)$ 이다. 행 2 는 $ptr[2] = ptr[3]$ 이라 비어 있다.

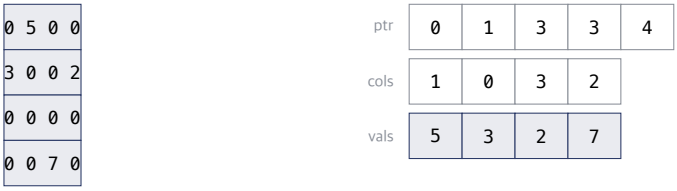


그림 3.1.. 예제 3.1 의 행렬과 CSR. 0 은 저장하지 않고, 행마다 시작 위치만 적는다

정리 3.3: CSR 행렬 벡터 곱의 비용

원소 크기 s, 인덱스 크기 4 인 CSR 에서 $y = Ax$ 는 연산 수 $W = 2\,\mathrm{nnz}$ 이고, 행렬 쪽 이동량은 $\mathrm{nnz}(s+4) + 4(m+1)$ 바이트다. 밀집 곱은 $W = 2mn$, 이동량 mns 다. 이동량만 보면 밀도 $\rho = \mathrm{nnz}/(mn)$ 이 $s/(s+4)$ 보다 작을 때 CSR 이 덜 옮긴다. float32 에서 50 퍼센트다.

증명. 0 이 아닌 원소마다 곱 하나 합 하나이고, 값과 열 번호를 한 번씩 읽으며 행 포인터를 한 번 훑는다. 밀집은 모든 원소를 한 번 읽는다. $\mathrm{nnz}(s+4) < mns$ 를 ρ 로 풀면 $\rho < s/(s+4)$ 다. ■

정리의 문턱 50 퍼센트는 행렬 쪽 이동량만 센 것이다. 실측이 그 근처에서 갈리는지 본다.

균열

4096×4096 float32 행렬의 밀도를 0.1 퍼센트에서 50 퍼센트까지 바꾸며 밀집 곱과 CSR 곱을 쟀다.

실험 3.1 밀도에 따른 밀집 곱과 CSR 곱

밀도 퍼센트	nnz	밀집 MB	CSR MB	밀집 ms	CSR ms
0.10	16991	67.1	0.17	2.42	0.52
1.00	168628	67.1	1.38	3.23	1.87
5.00	838947	67.1	6.74	3.00	8.93
10.0	1677617	67.1	13.5	3.07	17.8
20.0	3354746	67.1	26.9	4.43	43.1
50.0	8386951	67.1	67.1	3.06	78.9

측정 조건. float32 4096 × 4096, 행렬 벡터 곱. CSR 은 값 4바이트에 열 번호 4바이트. 일곱 번 중앙값.

CSR 이 이기는 것은 밀도 1 퍼센트까지다. 5 퍼센트에서 이미 세 배 느리고, 50 퍼센트에서는 스물다섯 배 느리다. 정리 3.3 의 문턱 50 퍼센트와 실측 문턱 2 퍼센트 사이가 스물다섯 배다. 정리가 못 센 것이 있다.

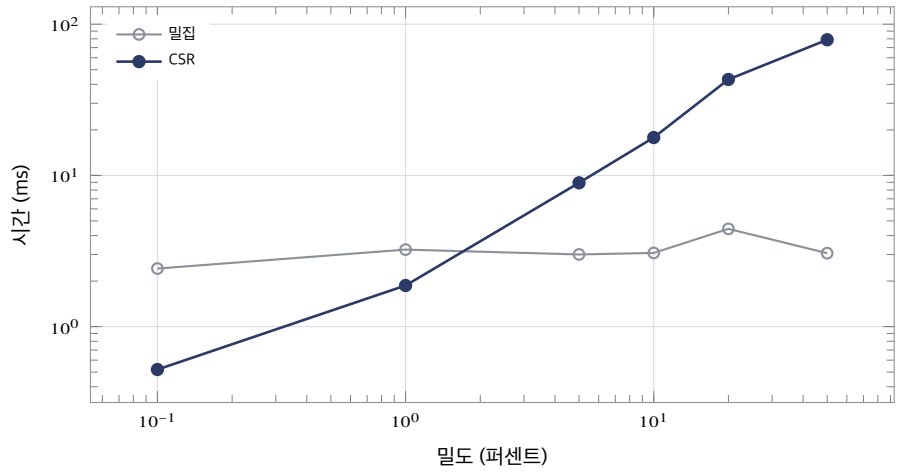


그림 3.2. 밀도에 따른 두 곱의 시간. 밀집은 밀도와 무관하고, CSR 은 nnz 에 비례한다. 2 퍼센트 근처에서 갈린다

못 센 것은 x 쪽 이동량이다. 정리는 행렬만 셌는데, CSR 곱은 원소마다 $x[cols_k]$ 를 읽는다. 열 번호가 무작위이면 그 읽기가 0.1 의 무작위 접근이고, 원소마다 라인 하나가 든다. x 가 캐시에 들어 있어도 라인 안의 나머지 원소를 못 쓰는 것은 같다.

관찰

희소 행렬의 값은 nnz 가 아니라 nnz 곱하기 라인이다. 인덱스가 무작위이면 원소 하나가 라인 하나를 산다.

다시 쓰기

이 절의 다시 쓰기는 정리를 고치는 것이다. x 의 이동량을 넣으면 문턱이 실측 쪽으로 내려온다.

명제 3.4: 인덱스 접근을 넣은 CSR 비용

> x 의 원소가 캐시에 있어도 열 번호가 무작위이면 원소마다 라인 하나를 읽는다고 보자. 그러면 CSR 곱의 이동량은 $\mathrm{nnz}(s+4+B)+4(m+1)$ 이고, 밀집보다 덜 옮기는 밀도는 $\rho < s/(s+4+B)$ 이다. float32 에 라인 64 를 넣으면 5.6 퍼센트다.

증명. 정리 3.3 의 이동량에 원소당 B 을 더하고 같은 부등식을 푼다. ■

5.6 퍼센트는 실측 문턱 2 퍼센트에 가깝지만 아직 위다. 나머지는 밀집 곱이 여러 코어와 벡터 연산으로 돌고 CSR 곱은 그렇지 못한 상수다. 그 상수를 줄이는 것이 다음 절이다. 열 번호가 무작위가 아니게 만들면 라인 하나로 원소 여럿을 산다.

구현

COO 에서 CSR 을 만드는 것과 CSR 곱이다. 둘 다 반복문 없이 NumPy 연산으로 쓴다.

지금 보는 코드 COO 를 행 번호로 정렬해 행 포인터를 세고, 곱은 값과 x 의 원소별 곱을 행 구간마다 더한다.

구현 COO 에서 CSR 로 `samples/dsai/sparse.py:58`

```
def coo_to_csr(c: COO) -> CSR:
    """알고리즘 3.1. 행 번호로 정렬한 뒤 행마다 시작 위치를 센다."""
    order = np.lexsort((c.cols, c.rows))
    rows, cols, vals = c.rows[order], c.cols[order], c.vals[order]
    counts = np.bincount(rows, minlength=c.shape[0])
    ptr = np.zeros(c.shape[0] + 1, dtype=np.int64)
    np.cumsum(counts, out=ptr[1:])
    return CSR(c.shape, ptr, cols, vals)
```

읽어 보면 행 번호로 정렬한 뒤 행마다 개수를 세고 누적합을 구하면 그것이 행 포인터다. `cumsum` 한 줄이 정의 3.2 의 불변식을 만든다.

알고리즘 3.1 COO 에서 CSR 로 `dsai.sparse.coo_to_csr`

> **입력:** COO (r, c, v), 행 수 m
> **출력:** CSR $(ptr, cols, vals)$
> 1: (r, c, v) 를 r 기준으로 정렬
> 2: $cnt[i] \leftarrow$ 행 i 의 원소 수 $(i = 0..m-1)$
> 3: $ptr[0] \leftarrow 0$; $ptr[i+1] \leftarrow ptr[i] + cnt[i]$
> 4: **반환** (ptr, c, v)

구현 CSR 행렬 벡터 곱 `samples/dsai/sparse.py:72`

```
def csr_matvec(m: CSR, x: np.ndarray) -> np.ndarray:
    """정리 3.1. 0 이 아닌 원소마다 곱 하나 합 하나. 이동량은 nnz 에 비례한다."""
    prod = m.vals * x[m.cols]
    out = np.zeros(m.shape[0], dtype=prod.dtype)
    nonempty = m.ptr[:-1] != m.ptr[1:]
    sums = np.add.reduceat(prod, m.ptr[:-1][nonempty]) if nonempty.any() else prod[:0]
    out[nonempty] = sums
    return out
```

`x[m.cols]` 가 명제 3.4 의 무작위 접근이고, `reduceat` 이 행 구간마다의 합이다. 빈 행은 따로 0 으로 둔다.

분석

희소 행렬은 조정해서 유효하다. 정의는 그대로인데 문턱이 내려왔다. 교과서의 문턱은 이동량 산수로 50 퍼센트였고, 이 기계의 실측은 2 퍼센트다. 그 사이에 있는 것이 0.1 의 첫째 가정이다. 인덱스 접근이 균일 비용이 아니다.

이 문턱은 이 책의 뒤에서 두 번 다시 나온다. 11 장의 그래프는 인접 행렬이 밀도 0.01 퍼센트 아래라 CSR 이 압도하는 자리이고, 이 장의 뒤 절들은 가중치 행렬처럼 밀도 50 퍼센트 근처라 CSR 로는 안 되는 자리다. 같은 구조가 밀도 하나로 답이 갈린다.

현장에서

> 과학 계산 라이브러리들이 CSR 을 기본 희소 표현으로 쓰는 이유가 정리 3.3 다. 행 단위 연산이 자르기 하나이기 때문이다. NumPy 의 설계 논문은 희소 행렬을 별도 라이브러리로 미룬 이유를 밀집 배열의 연속성에서 찾는다 [확립] [6].

되짚기

> COO, CSR, CSC 를 정의했고, CSR 곱이 이기는 밀도가 이동량 산수의 50 퍼센트가 아니라 실측 2 퍼센트임을 봤다. 차이는 인덱스 접근의 라인 값이다. 다음 절에서 0 의 자리를 미리 약속해 그 라인 값을 없앤다.

3.2 블록 희소와 구조적 희소성

3.1 에서 CSR 이 진 이유는 열 번호가 무작위라는 데 있었다. 원소마다 열 번호 4바이트를 들고, 그 번호로 x 를 무작위로 읽는다. 0 의 자리에 규칙을 주면 둘 다 준다. 규칙이 인덱스를 대신하고, 규칙이 접근을 붙여 놓는다. 이 절은 규칙 둘을 세운다. 블록 단위로 0 을 두는 것과, 네 칸마다 두 칸이라는 약속이다.

둘째 규칙은 가속기 하드웨어가 직접 지원하는 것이라 이 책의 기계로는 그 이득을 다 재지 못한다. 바이트는 재고, 시간은 재되 왜 안 빨라지는지를 적는다. 안 빨라진 실험도 실험이다.

학습 목표

블록 희소 표현과 2:4 구조적 희소성을 정의하고, 같은 밀도에서 세 표현의 바이트를 계산할 수 있다. 규칙이 인덱스와 접근 순서를 어떻게 바꾸는지 설명하고, 소프트웨어 구현이 왜 하드웨어 지원 없이 느린지 말할 수 있다.

이 말부터

블록 희소	행렬을 $b \times b$ 블록으로 나누고 블록 단위로 0 인지 아닌지를 적는 표현. BSR 이라고 쓴다
구조적 희소성	0 의 자리에 고정된 규칙이 있는 희소성. 규칙이 있으면 인덱스가 규칙만큼 짧아진다
2:4	연속한 원소 넷 중 정확히 둘만 0 이 아니라는 규칙. 밀도 50 퍼센트에 인덱스가 2 비트다

디딤돌

CSR 의 인덱스가 비싼 것은 0 이 어디든 있을 수 있기 때문이다. 자리를 제한하면 인덱스가 싸진다. 가장 단순한 제한이 블록이다. 1.3 의 블록 순회를 저장 형식에 적용한 것이다.

정의 3.5: 블록 희소 표현

$m \times n$ 행렬을 $b \times b$ 블록 $(m/b) \times (n/b)$ 개로 나눈다. 0 이 아닌 원소를 하나라도 가진 블록을 0 이 아닌 블록이라 한다. BSR 은 블록 단위의 CSR 이다. *ptr* 은 블록 행마다, *cols* 는 블록 열 번호, *vals* 는 블록 하나를 통째로 밀집으로 담은 $b \times b$ 배열의 열이다.

불변식 0 이 아닌 블록 안의 0 도 저장한다. 인덱스는 원소가 아니라 블록마다 하나다.

블록 안의 0 을 저장하는 것이 대가이고, 인덱스가 b^2 분의 1 로 주는 것과 블록 하나가 라인 여러 개를 통째로 쓰는 것이 이득이다. $b = 32$ 이면 인덱스가 원소당 4/1024 바이트이고, x 의 읽기는 블록 열마다 연속 32 개다.

명제 3.6: BSR 의 이동량

0 이 아닌 블록 수를 nnb 라 하면 BSR 의 행렬 쪽 이동량은 $\mathrm{nnb}\,(b^2 s + 4) + 4(m/b + 1)$ 바이트이고, x 쪽 읽기는 블록마다 연속 bs 바이트라 라인 수가 $\mathrm{nnb} \cdot \lceil bs/B \rceil$ 이다. 같은 밀도에서 CSR 의 x 읽기 라인 수 nnz 와 견주면 bs/B 배 적다. $b = 32$, $s = 4$ 이면 2 배이고, 인덱스는 b^2 배 적다.

둘째 제한은 더 세다. 블록이 아니라 원소 넷마다 규칙을 둔다.

정의 3.7: 2:4 구조적 희소성

행렬의 각 행을 연속한 원소 네 개씩 묶는다. 각 묶음에서 0 이 아닌 원소가 정확히 둘이면 그 행렬은 2:4 희소하다. 저장은 묶음마다 값 둘과 그 자리를 적은 2 비트 인덱스 둘이다. 인덱스 둘은 바이트 하나에 든다.

불변식 밀도가 정확히 50 퍼센트이고, 원소 하나의 인덱스는 2 비트다. CSR 의 32 비트에 견줘 16 분의 1 이다.

예제 3.2: 2:4 로 만들기

행의 한 묶음이 $(0.3, -1.2, 0.05, 0.8)$ 이면 절댓값이 큰 둘 -1.2 와 0.8 을 남기고 나머지를 0 으로 둔다. 저장은 값 $(-1.2, 0.8)$ 과 자리 $(1, 3)$ 이고, 자리는 $1 + 3 \times 4 = 13$ 을 바이트 하나에 담는다. float16 이면 묶음 하나가 $4 + 1 = 5$ 바이트다. 밀집 float16 의 8 바이트, CSR float32 의 16 바이트와 견준다.

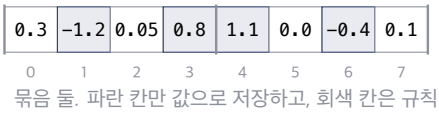


그림 3.3.. 2:4 묶음. 네 칸 중 두 칸만 저장하고, 어느 칸인지는 2 비트 둘로 적는다

균열

밀도 50 퍼센트에서 세 표현의 바이트와 곱셈 시간을 쟀다. 같은 행렬을 2:4 로 가지치기해 CSR 과 2:4 압축에 담고, 블록 절반을 0 으로 둔 행렬을 BSR 과 CSR 에 담았다.

실험 3.2 같은 밀도 50 퍼센트, 네 가지 표현

표현	MB	곱셈 ms
밀집	67.1	2.99
CSR (2:4 패턴)	67.1	74.1
2:4 압축	21.0	158
블록 희소 32	33.2	10.7
CSR (블록 패턴)	66.3	66.2

측정 조건. float32 4096 × 4096, 0 이 아닌 원소 절반. 2:4 압축의 값은 float16. 일곱 번 중앙값.

바이트는 정의대로 나왔다. 2:4 압축이 밀집의 3분의 1, BSR 이 절반, CSR 은 밀집과 같다. 시간은 정의대로 나오지 않았다. 2:4 압축이 가장 느리다. 묶음마다 2비트를 풀어 x 에서 두 원소를 골라 오는 산수를 NumPy 가 원소별 연산 여러 개로 돌리기 때문이다. BSR 은 CSR 보다 여섯 배 빠르다. 명제 3.6 의 라인 이득과 인덱스 이득이 그대로 시간이 됐다.

관찰

바이트를 줄이는 표현이 시간을 줄이려면 그 표현을 푸는 산수가 이동량보다 싸야 한다. 2:4 는 그 산수를 하드웨어가 해 줄 때만 이긴다.

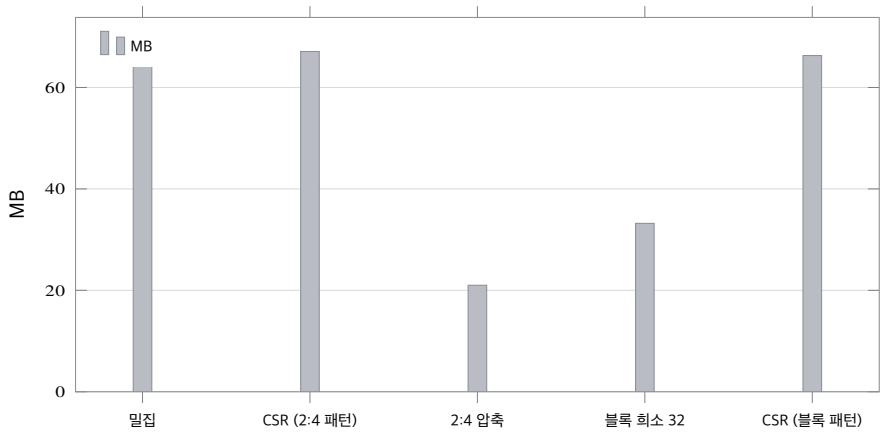


그림 3.4.. 같은 밀도에서 표현마다의 바이트. 규칙이 셀수록 인덱스가 준다

다시 쓰기

2:4 가 이 책의 기계에서 느린 것은 구조의 실패가 아니라 이 기계의 실패다. 그 규칙을 위해 만들어진 연산 장치가 따로 있다. Mishra 와 동료들은 2021 년에 2:4 패턴을 행렬 곱 장치가 직접 읽어 밀집의 두 배 처리량을 낸다고 보고했다 [확립] [13]. 장치가 인덱스 2비트를 읽고 x 에서 두 원소를 고르는 일을 곱셈과 같은 회로에서 하므로, 이 절의 실험이 치른 산수가 0 이 된다.

이 절의 다시 쓰기는 그래서 구조가 아니라 판단이다. 어느 희소 표현을 고를지는 밀도와 규칙과 장치 셋이 정한다.

밀도	규칙	표현
5 퍼센트 아래	없음	CSR. 인덱스 값을 치러도 옮기는 양이 압도적으로 적다
5 에서 50 퍼센트	블록 단위로 뭉침	BSR. 블록 안의 0 을 저장하고 인덱스와 라인을 아낀다
정확히 50 퍼센트	2:4 를 지키게 가지치기	2:4 압축. 그 규칙을 읽는 장치가 있을 때만
50 퍼센트 위	없음	밀집. 인덱스가 값보다 비싸다

이 표의 둘째 열이 이 절의 요점이다. 규칙은 데이터에 원래 있는 것이 아니라 만드는 것이다. 예제 3.2 처럼 작은 값을 0 으로 만들어 규칙을 맞추고, 그 대가로 정확도를 조금 잃는다. 잃는 양을 재는 것은 학습 쪽 책의 일이다.

구현

2:4 로 만들고, 담고, 담은 채로 곱하는 셋이다.

지금 보는 코드 네 칸마다 절댓값이 큰 둘을 남기고, 값 둘과 2 비트 자리 둘을 담고, 담은 채로 곱한다.

구현 2:4 가지치기와 담기 `samples/dsai/structured.py:51`

```python
def prune_2of4(a: np.ndarray) -> np.ndarray:
    """정의 3.4. 네 칸마다 절댓값이 큰 둘만 남긴다. 나머지는 0."""
    m, n = a.shape
    groups = a.reshape(m, n // 4, 4)
    order = np.argsort(-np.abs(groups), axis=2)
    mask = np.zeros_like(groups, dtype=bool)
    np.put_along_axis(mask, order[:, :, :2], True, axis=2)
    return (groups * mask).reshape(m, n)


def pack_2of4(a: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    """값 두 개와 2비트 인덱스 두 개를 그룹마다 담는다.
    인덱스는 바이트 하나에 둘을 넣는다.
    """
    m, n = a.shape
    groups = a.reshape(m, n // 4, 4)
    idx = np.argsort(groups == 0, axis=2, kind="stable")[:, :, :2]   # 0 이 아닌 자리 둘
    idx = np.sort(idx, axis=2)
    vals = np.take_along_axis(groups, idx, axis=2).astype(np.float16)
    meta = (idx[:, :, 0] | (idx[:, :, 1] << 2)).astype(np.uint8)
    return vals, meta
```

읽어 보면 `prune_2of4` 는 묶음마다 절댓값 순서로 둘을 고른다. `pack_2of4` 는 0 이 아닌 자리 둘을 찾아 값은 float16 으로, 자리는 $i_0 + 4i_1$ 로 바이트 하나에 담는다.

알고리즘 3.2 2:4 압축 행렬 벡터 곱 `dsai.structured.matvec_2of4`

입력: 값 $V[m, g, 2]$, 자리 $M[m, g]$ (바이트), 벡터 x
출력: $y[m]$

1: **반복** 각 행 i, 각 묶음 j **동안**
2: $\quad i_0 \leftarrow M[i,j] \bmod 4$; $i_1 \leftarrow \lfloor M[i,j]/4 \rfloor$
3: $\quad y[i] \leftarrow y[i] + V[i,j,0] \cdot x[4j + i_0] + V[i,j,1] \cdot x[4j + i_1]$
4: **끝 반복**

분석

블록 희소는 그대로 유효하고, 2:4 는 조건부다. 조건은 장치다. 이 판정을 이 기계에서 낼 수 있는 이유는 바이트가 장치와 무관하기 때문이다. 3분의 1 은 어디서 재도 3분의 1 이다. 시간은 장치의 것이고, 이 책은 자기 기계의 시간을 적고 다른 장치의 시간은 출처를 대고 적는다.

이 절의 판단 표는 5부에서 한 번 더 쓰인다. 벡터 인덱스의 그래프는 밀도가 0.01 퍼센트 아래라 CSR 의 자리이고, 곱 양자화의 코드북 조회는 규칙이 센 2:4 와 같은 쪽에 있다. 표의 행이 구조마다 하나씩 다시 나온다.

주의

가지치기로 만든 규칙은 정확도를 깎는다. 이 절은 바이트와 시간만 쟀고 정확도는 재지 않았다. 이 책의 범위 밖이다.

되짚기

블록 희소와 2:4 를 정의했고, 규칙이 인덱스를 16분의 1 로, x 읽기의 라인을 절반 이하로 줄이는 것을 봤다. 2:4 의 시간 이득은 장치가 있어야 한다. 다음 절에서 값 자체의 비트를 줄이는 양자화를 세운다.

3.3 양자화된 텐서. 스케일과 블록과 조회표

1.1 의 값 배열은 원소 타입 τ 를 정의에 넣었다. 지금까지 τ 는 float32 였다. 이 절은 τ 를 8 비트, 4 비트로 줄인다. 값 배열의 정의는 그대로이고 원소 하나가 작아진다. 작아진 만큼 2.3 의 디코딩 이동량이 줄고, 그 대가로 값이 정확하지 않게 된다.

정확하지 않은 정도를 어떻게 묶어 두느냐가 이 절의 자료 구조다. 정수 코드 옆에 스케일을 하나 두는데, 그 스케일을 텐서 전체에 하나 둘지 블록마다 둘지가 오차를 백 배 가른다.

학습 목표

> 블록 양자화를 코드, 스케일, 비트, 블록 넷으로 정의하고, 균등 양자화의 오차 상한을 증명할 수 있다. 같은 비트에서 텐서 전체 스케일과 블록 스케일의 오차가 왜 다른지 이상값으로 설명할 수 있다.

이 말부터

> 양자화 실수를 정수 몇 비트와 스케일 하나로 적는 것. 되돌리면 원래 값에 가까운 값이 나온다
> 스케일 정수 코드 1 이 실수 얼마에 해당하는가. 양자화 간격 Δ 와 같다
> 이상값 다른 값들보다 훨씬 큰 값. 스케일을 혼자 정해 버려 나머지의 정밀도를 깎는다

디딤돌

양자화는 신호 처리의 오래된 개념이다. 실수 축을 간격 Δ 로 자르고 각 값을 가장 가까운 눈금으로 옮긴다.

정의 3.8: 균등 양자화

> 비트 b 와 스케일 $\Delta > 0$ 에 대해 양자화는 $q(x) = \mathrm{clip}(\mathrm{round}(x/\Delta), -2^{b-1}, 2^{b-1}-1)$ 이고, 되돌리기는 $\hat{x} = q(x)\Delta$ 다. Δ 는 대개 값들의 최대 절댓값 a 를 $2^{b-1}-1$ 로 나눈 것이다.

이 정의에서 오차는 반올림 한 번에서만 나온다. 눈금 사이의 값은 가장 가까운 눈금으로 가고, 그 거리가 눈금 간격의 절반을 넘지 않는다. 그것을 정리로 적으면 아래와 같고, 이 책의 양자화 이야기는 전부 이 한 줄 위에 선다.

정리 3.9: 균등 양자화의 오차

> $|x| \le a$ 인 모든 x 에 대해 $|x - \hat{x}| \le \Delta/2$ 다. x 가 각 눈금 구간 안에서 고르게 분포하면 오차의 제곱 기대값은 $\Delta^2/12$ 다.

증명. $|x| \le a$ 이면 x/Δ 가 코드 범위 안에 있어 잘림이 없고, 반올림 오차는 $1/2$ 이하이므로 $|x - \hat{x}| \le \Delta/2$ 다. 구간 $[-\Delta/2, \Delta/2]$ 에서 고른 분포의 제곱 평균은 $\int_{-\Delta/2}^{\Delta/2} u^2/\Delta \, du = \Delta^2/12$ 다. ■

정리가 말하는 것은 오차가 Δ 에 달렸고 Δ 는 a 에 달렸다는 것이다. 값 만 개 중 하나가 나머지의 스무 배이면 a 가 스무 배가 되고 Δ 도 스무 배가 되어, 나머지 9,999 개의 오차가 스무 배가 된다. 이상값 하나가 전체의 정밀도를 정한다.

예제 3.3: 이상값 하나

값 대부분이 $[-0.06, 0.06]$ 에 있고 하나가 1.2 이면, 8 비트에서 $\Delta = 1.2/127 \approx 0.0094$ 다. 대부분의 값이 눈금 여섯 개 남짓으로 적힌다. 이상값을 빼면 $\Delta = 0.06/127 \approx 0.00047$ 이고 눈금이 스무 배 촘촘하다.

균열

4096 × 4096 정규분포 가중치에 만 개 중 하나를 스무 배 이상값으로 섞고, 비트와 스케일 범위를 바꿔 가며 상대 오차를 쟀다.

실험 3.3 비트와 블록에 따른 바이트와 오차

표현	비트	블록	MB	상대 오차 퍼센트
float32	32	0	67.1	0.00
float16	16	0	33.6	4.3e-06
int8 텐서 하나	8	0	16.8	2.38
int8 블록 64	8	64	17.8	0.0046
int4 텐서 하나	4	0	8.39	96.2
int4 블록 64	4	64	9.44	1.35
int4 블록 32	4	32	10.5	1.02

측정 조건. 4096 × 4096 정규분포 가중치, 만 개 중 하나가 스무 배 이상값. 오차는 상대 제곱 오차의 퍼센트. 블록 0 은 텐서 전체에 스케일 하나.

8 비트에서 텐서 하나에 스케일 하나면 오차가 2.4 퍼센트이고, 블록 64 마다 스케일을 두면 0.005 퍼센트다. 500 배 차이다. 4 비트에서는 텐서 하나가 96 퍼센트로 쓸 수 없고, 블록 64 가 1.4 퍼센트다. 바이트는 블록 스케일이 6 퍼센트 더 든다. 64 개마다 float32 하나이니 원소당 0.5 바이트 대 0.0625 바이트다.

관찰

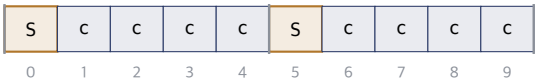
스케일을 어디에 두느냐가 비트 수보다 오차를 더 가른다. 4 비트 블록 64 가 8 비트 텐서 하나보다 오차가 작다.

다시 쓰기

스케일을 블록마다 두는 것을 정의에 넣는다. 그러면 양자화된 텐서가 값 배열 하나가 아니라 둘이 된다.

정의 3.10: 블록 양자화 텐서

> 블록 양자화 텐서는 넷의 묶음 (C, S, b, p) 다. C 는 b 비트 정수 코드의 값 배열, S 는 블록마다 하나인 float 스케일 배열, p 는 블록 크기다. 원소 i 의 값은 $C[i] \cdot S[\lfloor i/p \rfloor]$ 이고, 바이트는 원소당 $b/8 + 4/p$ 다.
>
> **불변식** 블록 j 의 스케일은 그 블록의 최대 절댓값을 $2^{b-1} - 1$ 로 나눈 것이다. 이상값의 영향이 그 블록 p 개 안에 갇힌다.



블록 크기 4 로 그린 것. S 는 스케일, c 는 정수 코드

그림 3.5.. 블록 64 의 양자화. 블록마다 스케일 하나. 이상값이 든 블록만 눈금이 성겨진다

그림에서 스케일 칸이 블록마다 하나씩 끼어 있다. 그 칸이 원소당 $4/p$ 바이트의 대가이고, 그 대가로 이상값의 영향을 자기 블록에 가둔다. 가두는 정도를 세면 다음 명제다.

명제 3.11: 블록 스케일과 이상값

> 이상값이 텐서 전체 N 개 중 k 개이면, 텐서 하나 스케일에서는 N 개 전부의 Δ 가 이상값에 맞춰지고, 블록 p 스케일에서는 최대 kp 개의 Δ 만 그렇다. 나머지 $N - kp$ 개는 자기 블록의 최대값에 맞춘 Δ 를 쓴다.

증명. 스케일은 그 범위의 최대 절댓값이 정한다. 이상값 하나가 속한 범위는 텐서 하나 스케일에서 텐서 전체이고 블록 스케일에서 블록 하나다. ■

실험 3.3 에서 $N = 2^{24}$, $k \approx 1{,}678$, $p = 64$ 이면 이상값의 영향이 $kp \approx 10^5$ 개, 전체의 0.6 퍼센트에 갇힌다. 그래서 오차가 500 배 준다. 정리 3.9 의 Δ 가 99.4 퍼센트의 원소에서 스무 배 작아졌기 때문이다.

구현

블록 양자화와 되돌리기다. 반올림 하나와 나눗셈 하나다.

지금 보는 코드 블록마다 최대 절댓값으로 스케일을 정하고 반올림한다. 되돌리기는 곱셈 하나다.

구현 블록 양자화 `samples/dsai/quant.py:28`

```
def quantize(x: np.ndarray, bits: int, block: int) -> QTensor:
    """알고리즘 3.3. 블록마다 최대 절댓값으로 스케일을 정하고 반올림한다."""
    flat = x.reshape(-1).astype(np.float32)
    assert flat.size % block == 0
    groups = flat.reshape(-1, block)
    qmax = 2 ** (bits - 1) - 1
    amax = np.abs(groups).max(axis=1)
    scales = np.where(amax > 0, amax / qmax, 1.0).astype(np.float32)
    codes = np.clip(np.rint(groups / scales[:, None]), -qmax - 1, qmax).astype(np.int8)
    return QTensor(codes, scales, bits, block, x.shape)


def dequantize(q: QTensor) -> np.ndarray:
    return (q.codes.astype(np.float32) * q.scales[:, None]).reshape(q.shape)
```

읽어 보면 `groups` 가 블록이고 `amax` 가 블록마다의 a 다. `scales` 가 정의 3.10 의 S 이고, `codes` 가 C 다. `dequantize` 는 코드에 스케일을 브로드캐스팅으로 곱한다. 1.4 의 규칙이 여기서 쓰인다.

알고리즘 3.3 블록 양자화 `dsai.quant.quantize`

입력: 값 배열 $x[N]$, 비트 b, 블록 p
출력: 코드 $C[N]$, 스케일 $S[N/p]$

1: **반복** 각 블록 j **동안**
2: $\quad a \leftarrow \max_{i \in j} |x[i]|$; $S[j] \leftarrow a / (2^{b-1} - 1)$
3: $\quad$ **반복** 각 $i \in j$ **동안**
4: $\quad\quad C[i] \leftarrow \mathrm{clip}(\mathrm{round}(x[i] / S[j]))$
5: $\quad$ **끝 반복**
6: **끝 반복**

분석

양자화된 텐서의 판정은 그대로 유효다. 값 배열의 정의에 타입이 있었고, 타입을 바꾼 것뿐이다. 새로운 것은 스케일 배열이 값 배열 옆에 붙었다는 것이고, 그것이 2.3 의 디코딩 이동량을 4 비트에서 8 분의 1 로 줄인다. 읽는 바이트가 8 분의 1 이면 명제 2.9 의 시간도 8 분의 1 에 가깝다.

이 절의 실험은 오차를 가중치에 대해 쟀다. 그 오차가 모델의 출력을 얼마나 바꾸는지는 재지 않았고, 이 책의 범위 밖이다. 다만 공개 논문들이 같은 산수 위에서 결과를 냈다. 이 절이 다루는 것은 그 산수의 자료 구조 부분이다.

현장에서

Dettmers 와 동료들의 2023 년 QLoRA 는 블록 64 의 4 비트 양자화에 스케일 자체를 다시 양자화하는 이중 양자화를 더해 65B 모델을 48GB 에 넣었다 [확립] [14]. Frantar 와 동료들의 2023 년 GPTQ 는 반올림을 오차를 보상하며 순서대로 하는 방법으로 3, 4 비트에서 정확도를 지켰다 [확립] [15].

이 절의 정의에서 코드가 정수 눈금이 아니어도 된다는 것을 적어 둔다. 눈금 16 개를 정규분포에 맞춰 성기게 두면 정규분포 가중치에서 오차가 더 준다. 그 눈금표가 조회표이고, 코드는 조회표의 번호가 된다. QLoRA 의 NF4 가 그것이고, 13 장의 곱 양자화가 같은 생각을 벡터에 적용한다.

되짚기

> 균등 양자화의 오차 상한 $\Delta/2$ 를 증명했고, 스케일을 블록마다 두면 이상값의 영향이 블록 안에 갇혀 오차가 500 배 주는 것을 봤다. 다음 절에서 가중치를 통째로 읽지 않고 토큰마다 일부만 읽는 구조, 전문가 혼합을 세운다.

3.4 전문가 혼합의 gather 와 scatter

1.4 의 주의 상자가 배치가 안 통하는 작업량 하나를 예고했다. 질의마다 다른 행렬을 읽는 작업량이다. 2.3 의 KV 캐시가 그 하나였고, 이 절이 둘째다. 전문가 혼합이다. 가중치를 전문가 여럿으로 나누고 토큰마다 하나만 읽는다. 읽는 양은 E 분의 1 이 되고, 대신 토큰마다 읽는 행렬이 다르다.

이 절은 그 구조를 3.1 의 희소 행렬로 읽는다. 라우터가 만드는 토큰과 전문가의 배정은 밀도 $1/E$ 인 희소 행렬이고, 그 행렬을 CSR 로 접은 것이 gather 이고 다시 편 것이 scatter 다.

학습 목표

전문가 혼합을 라우터와 전문가 배열로 정의하고, gather 와 scatter 를 알고리즘으로 적을 수 있다. 배치 크기에 따라 읽는 전문가 수의 기대값을 구하고, 그것이 연산 밀도를 어떻게 바꾸는지 실측으로 설명할 수 있다.

이 말부터

전문가	같은 모양의 행렬 여러 개 중 하나. 토큰마다 그중 하나 또는 몇 개만 곱한다
라우터	토큰을 보고 어느 전문가로 보낼지 정하는 작은 함수. 점수가 가장 큰 쪽을 고른다
gather 와 scatter	흩어진 원소를 모으는 것과 모은 결과를 제자리로 되돌리는 것. 인덱스 배열 하나로 둘 다 한다

디딤돌

밀집 층은 토큰 B 개 전부가 같은 행렬 W 를 곱한다. 1.4 의 명제로 밀도가 B 에 비례해 오르는 자리다. 전문가 혼합은 그 행렬을 E 개로 늘리고 토큰마다 하나를 고른다.

정의 3.12: 전문가 혼합

전문가 혼합 층은 전문가 행렬 $W_1, \dots, W_E$ (각 $d \times h$) 와 라우터 $g : \mathbb{R}^d \to \{1, \dots, E\}$ 의 묶음이다. 토큰 x_i 의 출력은 $x_i W_{g(x_i)}$ 다. 배치 B 의 배정 $g(x_1), \dots, g(x_B)$ 는 $B \times E$ 의 0 과 1 행렬이고 행마다 1 이 하나다.

불변식 배정 행렬의 밀도는 정확히 $1/E$ 이고, 토큰 하나는 전문가 하나의 행렬만 읽는다.

배정 행렬을 3.1 의 CSC 로 접으면 전문가마다 자기에게 온 토큰의 번호 목록이 된다. 그 목록으로 토큰을 모으는 것이 gather 다. 모은 토큰을 한 번의 행렬 곱으로 처리하고, 같은 목록으로 결과를 제자리에 흩는 것이 scatter 다.

알고리즘 3.4 전문가 혼합 순전파 `dsai.moe.moe_forward`

입력 : 토큰 $X[B, d]$, 전문가 $W[E, d, h]$, 배정 $g[B]$
출력 : 출력 $Y[B, h]$

반복 각 전문가 e 중 배정된 토큰이 있는 것 **동안**
  $I \leftarrow \{i : g[i] = e\}$ ▷ 배정 행렬의 열 e. CSC 의 열 하나
  $Y[I] \leftarrow X[I]\, W[e]$ ▷ gather, 곱, scatter 를 인덱스 배열 하나로
끝 반복

예제 3.4: 토큰 넷과 전문가 셋

배정이 $(2, 1, 2, 3)$ 이면 전문가 1 은 토큰 $\{1\}$, 전문가 2 는 $\{0, 2\}$, 전문가 3 은 $\{3\}$ 을 받는다. 행렬 곱 세 번이고, 읽는 행렬은 셋 다다. 토큰 넷을 밀집 층에 넣으면 행렬 곱 한 번에 행렬 하나다. 토큰당 연산은 같고, 읽는 바이트는 세 배다.

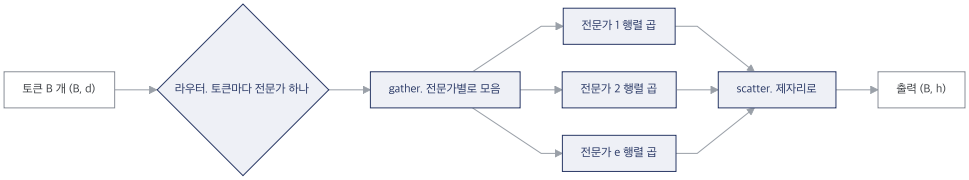


그림 3.6.. gather, 곱, scatter. 배정 행렬의 열 하나가 전문가 하나의 토큰 목록이다

균열

전문가 8 개, 각각 2048×2048 인 층과, 전문가 하나 크기의 밀집 층을 배치 크기별로 쟀다. 토큰당 연산 수는 둘이 같다. 다른 것은 읽는 행렬의 수다.

실험 3.4 배치 크기에 따른 밀집 층과 전문가 혼합

배치	읽는 전문가 수	밀도	밀집 ms	혼합 ms
1	1.00	0.50	0.79	0.67
4	3.31	0.60	2.41	3.01
16	7.06	1.13	2.39	11.5
64	8.00	3.97	7.93	24.4
256	8.00	15.5	22.6	55.1

측정 조건. 전문가 8 개, 각 2048×2048 float32 (16MB). 밀집은 전문가 하나 크기. 읽는 전문가 수는 명제 3.8 의 기대값. 일곱 번 중앙값.

배치 1 에서는 둘이 같다. 전문가 하나를 읽으나 밀집 행렬 하나를 읽으나 16MB 다. 배치 16 에서 혼합은 밀집의 다섯 배 느리다. 토큰 16 개가 전문가 일곱을 건드려 112MB 를 읽고,

밀집은 여전히 16MB 를 읽는다. 배치 256 에서는 여덟 전문가를 다 읽고, 전문가마다 토큰 32 개를 곱하니 밀도가 밀집의 8분의 1 이다.

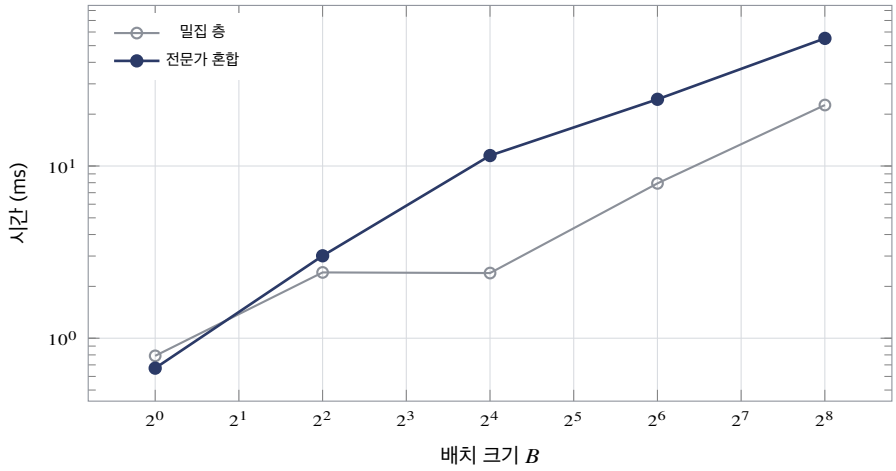


그림 3.7.. 배치 크기에 따른 두 층의 시간. 배치가 커질수록 혼합이 읽는 전문가가 늘어 벌어진다

관찰

전문가 혼합은 같은 연산으로 여덟 배 큰 모델을 돌리는 구조이지, 같은 모델을 빠르게 돌리는 구조가 아니다. 배치가 커지면 밀도가 밀집의 E 분의 1 이다.

다시 쓰기

읽는 전문가 수를 기대값으로 세우고, 그것으로 밀도를 다시 쓴다.

명제 3.13: 읽는 전문가 수의 기대값

라우터가 토큰마다 E 개 중 하나를 균등하게 고르면, 배치 B 에서 한 번이라도 뽑히는 전문가 수의 기대값은 $E\,(1-(1-1/E)^B)$ 다. $B=1$ 이면 1 이고, $B \gg E$ 이면 E 에 다가간다.

증명. 전문가 하나가 B 번 모두 안 뽑힐 확률은 $(1-1/E)^B$ 다. 뽑힐 확률은 그 여사건이고, 기대값의 선형성으로 E 를 곱한다. ■

명제 3.14: 전문가 혼합의 연산 밀도

배치 B 의 연산 수는 $2Bdh$ 이고 이동량은 읽는 전문가 수 $T(B) = E(1-(1-1/E)^B)$ 에 대해 $T(B)\,dhs + B(d+h)s$ 다. 밀도는 $B \ll E$ 에서 밀집과 같고, $B \gg E$ 에서 밀집의 $1/E$ 에 다가간다.

두 명제를 실험 3.4 에 대입하면 셋째 열이 나온다. 배치 256 에서 밀도 15.5 는 밀집의 128 에 견줘 8 분의 1 이다. 전문가 혼합의 밀도를 올리는 방법은 하나다. 배치를 E 배 더 키워 전문가마다 충분한 토큰을 주는 것이다. 그것이 이 구조가 큰 배치에서만 쓰이는 이유다.

구현

라우터, gather 와 scatter, 기대값이다.

지금 보는 코드 배정별로 토큰을 모아 곱하고 제자리로. 인덱스 배열 idx 가 gather 와 scatter 를 둘 다 한다.

구현 gather, 곱, scatter `samples/dsai/moe.py:17`

```python
def moe_forward(x: np.ndarray, experts: np.ndarray, assign: np.ndarray) -> np.ndarray:
    """알고리즘 3.4. 전문가별로 토큰을 모아 곱하고 제자리로 돌린다.

    x 는 (B, d), experts 는 (E, d, h), assign 은 (B,). 결과는 (B, h).
    """
    out = np.empty((x.shape[0], experts.shape[2]), dtype=x.dtype)
    for e in np.unique(assign):
        idx = np.nonzero(assign == e)[0]          # gather 할 자리
        out[idx] = x[idx] @ experts[e]           # 전문가 e 의 행렬을 한 번 읽는다
    return out
```

읽어 보면 `np.nonzero(assign == e)` 가 배정 행렬의 열 e 이고, `x[idx]` 가 gather, `out[idx] =` 가 scatter 다. 0.1 의 인덱스 배열 접근이고, 토큰 B 개 중 B/E 개를 무작위로 읽으므로 그 값도 3.1 의 무작위 접근과 같다.

구현 읽는 전문가 수와 밀도 `samples/dsai/moe.py:33`

```python
def experts_touched_expected(num_experts: int, batch: int) -> float:
    """명제 3.4. 배치 B 가 균등 라우팅될 때 한 번이라도 뽑히는 전문가 수의 기대값."""
    return num_experts * (1.0 - (1.0 - 1.0 / num_experts) ** batch)


def moe_intensity(batch: int, d: int, h: int, num_experts: int,
                  elem_bytes: int) -> float:
    """토큰 B 개 처리의 연산 밀도. 읽는 전문가 수가 기대값이라고 본다."""
    flops = 2 * batch * d * h
    touched = experts_touched_expected(num_experts, batch)
    nbytes = touched * d * h * elem_bytes + batch * (d + h) * elem_bytes
    return flops / nbytes
```

분석

전문가 혼합은 새 구조가 아니라 3.1 의 희소 행렬과 1.4 의 배치를 합친 것이다. 배정이 희소 행렬이고, 전문가마다의 곱이 배치다. 새로운 것은 그 둘의 곱에서 이동량이 배치 크기의 함수로 바뀐다는 것이다. 그래서 이 층의 판정은 배치 크기가 정한다. 작은 배치에서는 밀집과 같고, 큰 배치에서는 E 배 큰 모델의 값이다.

현장에서

> Fedus, Zoph, Shazeer 의 2022 년 Switch Transformer 는 토큰마다 전문가 하나만 고르는 라우터로 조 단위 매개변수 모델을 같은 연산으로 돌렸다 [확립] [16]. 전문가마다 받을 수 있는 토큰 수에 상한을 두는 용량 인자가 그 논문의 자료 구조 부분이고, 이 절의 알고리즘에 그 상한을 더하면 그것이 된다.

이 장의 판정을 여기서 낸다. 희소 행렬은 조정해서 유효하다. 밀도 문턱이 50 퍼센트에서 2 퍼센트로 내려왔고, 규칙 있는 희소성이 그 사이를 메운다. 전문가 혼합은 그 규칙을 가중치가 아니라 배정에 둔 것이다.

주의

> 실험 3.4 의 밀집 층은 전문가 하나 크기다. 전문가 여덟 개 크기의 밀집 층과 견주면 혼합이 모든 배치에서 이긴다. 무엇과 견주는지가 답을 정한다.

되짚기

> 전문가 혼합을 라우터와 전문가 배열로 정의했고, 읽는 전문가 수의 기대값과 밀도를 세웠다. 배치 16 에서 혼합이 밀집의 다섯 배 느렸다. 다음 장에서 연속성을 아예 버린 구조, 연결 리스트가 어디에 남았는지 본다.

3장을 닫으며

이 장은 1부를 닫는다. 1부가 세운 것은 값 배열 하나와 그 위의 변주다. 스트라이드로 축을 펴고, 블록으로 훑고, 페이지로 나누고, 0 을 빼고, 비트를 깎고, 필요한 것만 읽는다. 어느 변주도 연속성 불변식을 버리지 않았다. 범위를 좁혔을 뿐이다. 행 하나 안에서, 블록 하나 안에서, 페이지 하나 안에서 연속이다.

네 실험의 숫자를 적어 둔다. CSR 은 밀도 1 퍼센트까지만 밀집을 이겼고, 50 퍼센트에서는 스물다섯 배 느렸다. 같은 밀도 50 퍼센트에서 2:4 압축은 바이트가 3분의 1 이었지만 이 기계에서는 가장 느렸고, 블록 희소는 CSR 의 여섯 배 빨랐다. 4비트 블록 64 의 오차 1.4 퍼센트가 8비트 텐서 하나의 2.4 퍼센트보다 작았다. 전문가 혼합은 배치 16 에서 밀집의 다섯 배 느렸고, 그 이유는 읽는 전문가 수가 일곱이었기 때문이다.

정정 하나를 남긴다. 2:4 압축을 처음 잴 때 필자는 바이트가 3분의 1 이니 시간도 줄 것으로 예상했다. 쉰 배 느렸다. 인덱스를 푸는 산수를 이 기계의 벡터 연산이 못 받는다는 것을 그때 알았고, 그래서 이 절의 판정에 장치라는 조건이 붙었다. 바이트가 준다고 시간이 주는 것은 아니다. 그 사이에 표현을 푸는 산수가 있고, 그 산수를 누가 하느냐가 답을 가른다.

옛 구조의 판정

> 희소 행렬은 조정해서 유효하다. 정의는 그대로이고 문턱이 내려왔다. 근거는 실험 3.1 이다. 이동량 산수의 문턱은 50 퍼센트, 실측은 2 퍼센트. 차이는 인덱스 접근의 라인 값이다. 그 사이를 메우는 것이 규칙 있는 희소성이고, 규칙의 값은 장치가 정한다.

이 장의 요약

> 1. CSR 은 행 안에서만 연속이다. 곱의 이동량은 nnz 에 라인을 곱한 것이고, 그래서 문턱이 2 퍼센트다 3.1
> 2. 0 의 자리에 규칙을 두면 인덱스가 규칙만큼 준다. 블록은 라인을 아끼고, 2:4 는 인덱스를 2 비트로 줄인다 3.2
> 3. 균등 양자화의 오차는 $\Delta/2$ 이하이고 Δ 는 최대값이 정한다. 스케일을 블록마다 두면 이상값이 갇힌다 3.3
> 4. 전문가 혼합의 배정은 밀도 $1/E$ 의 희소 행렬이고, 읽는 전문가 수의 기대값이 밀도를 정한다 3.4
> 5. 셋 다 이동량을 줄이고, 대가는 인덱스, 오차, 배치 이득의 손실이다 3.4
>
> 다음 장에서 뒤집히는 것: 1부는 연속성을 좁혔다. 2부는 연속성이 아예 없는 구조, 연결 리스트가 어디에 남았는지 본다.

필자의 판단

> 필자는 희소 행렬을 가르칠 때 밀도 문턱을 먼저 재게 하는 쪽이 맞다고 본다. 교과서는 nnz 가 mn 보다 훨씬 작으면 희소라고 적고 끝내는데, 훨씬이 얼마인지를 안 재면 학생은 10 퍼센트 밀도에 CSR 을 쓰고 세 배를 잃는다. 문턱은 기계마다 다르지만, 재는 법은 같고 한 시간이면 된다.

열린 문제

하나, 실험 3.2 의 2:4 시간은 이 기계의 것이다. 전용 장치에서 재는 시간은 출처를 대고 적었을 뿐 재지 못했다.
둘, 양자화 오차가 모델 출력에 미치는 영향은 재지 않았다. 가중치의 상대 오차와 출력의 오차 사이의 관계는 이 책의 범위 밖이다.
셋, 명제 3.8 은 라우터가 균등하다고 가정했다. 실제 라우터는 쏠리고, 쏠림이 읽는 전문가 수를 줄이지만 어느 전문가에 토큰이 몰려 대기가 생긴다. 그 대기는 세지 않았다.

연습문제

3.1 기초 5×5 행렬에서 0 이 아닌 원소가 $(0,2)=1$, $(2,0)=4$, $(2,4)=2$, $(4,1)=3$ 일 때 CSR 세 배열과 CSC 세 배열을 각각 적어라.

3.2 기초 float16 원소와 4 바이트 인덱스에서 정리 3.1 의 이동량 문턱을 구하고, 명제 3.2 처럼 라인 64 를 넣으면 얼마가 되는지 구하라.

3.3 기초 1024×1024 float16 행렬을 2:4 로 담을 때의 바이트, 블록 16 BSR 로 밀도 50 퍼센트를 담을 때의 바이트, CSR 로 담을 때의 바이트를 구하라.

3.4 기초 값이 $[-0.05, 0.05]$ 에 고르게 있고 하나가 2.0 인 원소 4,096 개를 8 비트로 양자화할 때, 텐서 하나 스케일과 블록 64 스케일 각각에서 이상값이 아닌 원소의 오차 상한을 구하라.

3.5 응용 `dsai.sparse` 에 CSR 과 CSR 의 곱 (행렬 행렬 곱) 을 구현하고, 밀도 1 퍼센트에서 밀집 곱과 시간을 비교하라. 결과의 밀도가 얼마가 되는지도 재라.

3.6 응용 `dsai.structured.dense_to_bsr` 의 블록 크기를 8, 16, 32, 64 로 바꿔 실험 3.2 의 블록 희소 행렬을 다시 재고, 명제 3.3 의 인덱스 바이트와 라인 수를 한 표에 놓아라.

3.7 응용 `dsai.quant.quantize` 에 스케일 자체를 8 비트로 양자화하는 이중 양자화를 더하고, 블록 64 에서 원소당 바이트가 얼마나 주는지와 오차가 얼마나 느는지 재라.

3.8 응용 `dsai.moe.moe_forward` 에 전문가마다 받을 수 있는 토큰 수의 상한 (용량 인자) 을 더하고, 상한을 넘은 토큰을 버릴 때 배치 256 에서 버려지는 토큰의 비율을 재라.

3.9 탐구 CSR 의 열 번호를 정렬해 두면 x 읽기의 라인 수가 줄어드는가. 정렬된 열 번호에서 인접 원소가 같은 라인에 들 확률을 밀도의 함수로 구하고, 문턱이 어디로 움직이는지 논하라.

3.10 탐구 2:4 를 일반화한 $n{:}m$ 규칙에서 인덱스 비트와 밀도와 정확도 손실의 관계를 세우고, 어느 (n, m) 이 이 기계와 전용 장치 각각에서 유리한지 논하라.

더 읽을 것

[13] 2:4 규칙을 읽는 행렬 곱 장치의 설계와 가지치기 절차. 3.2 절의 판정에 붙은 장치라는 조건의 출처다. 3.2
[14] 블록 64 의 4 비트 양자화와 이중 양자화, 그리고 정규분포에 맞춘 조회표 NF4. 3.3 절의 정의가 실제 모델에서 어떻게 쓰이는지 보인다. 3.3
[15] 반올림 순서를 바꿔 오차를 보상하는 사후 양자화. 정리 3.2 의 오차를 어떻게 더 줄이는지의 한 답이다. 3.3

> [16] 전문가 하나만 고르는 라우터와 용량 인자. 3.4 절의 알고리즘에 상한을 더한 것이 이 논문의 층이다. 3.4

2부

연결과 순서

1부는 연속성을 좁혔다. 2부는 연속성이 아예 없는 구조를 본다. 연결 리스트는 어디서 사라지고 어디에 남았는가. 스택과 큐는 어떤 이름으로 살아 있는가.

4장

연결 리스트가 남은 자리

1994년 운영체제 커널의 메모리 할당자를 다시 설계한 논문은 객체를 크기별로 모아 두고 자유 리스트로 주고받는 방법을 세웠다 [확립] [17]. 그 뒤 30년 동안 이 설계는 여러 커널과 언어 런타임에 남았다. 연결 리스트가 살아남은 가장 오래된 자리다. 그런데 그 리스트는 순회하지 않는다. 머리만 쓴다.

이 장은 0.2 의 첫째 가정을 연결 리스트에 대고 다시 잰다. 교과서는 순회를 $O(n)$ 으로 적고 배열과 같다고 말한다. 연산 수로는 같고 지연으로는 스물다섯 배다. 그래서 순회하는 연결 리스트는 이 책이 다루는 시스템에서 거의 사라졌고, 순회하지 않는 것과 순회 순서를 배열로 뽑아 두는 것만 남았다. 이 장은 사라진 이유를 재고, 남은 두 모양을 정의한다.

이 장에서 세우는 것

4.1 순회의 값이 이동량이 아니라 의존 읽기의 지연임을 재고, 지연을 숨기는 세 조건을 세운다

4.2 자유 리스트. 머리만 쓰는 연결 리스트가 할당자 안에 남았다

4.3 인덱스 링크와 아레나. 순회 순서를 배열로 뽑고, 압축으로 순차 접근을 되찾는다

4.1 연결 리스트와 포인터 추적의 값. 캐시 미스를 재다

1.1 는 배열을 연속성 불변식으로 정의했다. 연결 리스트는 그 불변식을 처음부터 버린 구조다. 원소가 어디 있어도 되고, 대신 원소마다 다음 원소의 주소를 든다. 교과서는 그 대가로 중간 삽입이 상수가 된다고 가르친다. 맞는 말이다. 다만 순회의 값이 얼마인지는 재지 않는다.

이 절은 그 순회의 값을 잰다. 값은 이동량이 아니라 지연이다. 다음 주소를 알려면 지금 주소를 읽어야 하므로, 읽기 하나가 끝나기 전에는 다음 읽기를 시작할 수 없다. 1.3 의 간격 접근은 이동량이 컸을 뿐 프로세서가 미리 가져올 수는 있었다. 여기서는 그것도 안 된다.

학습 목표

> 연결 리스트를 노드와 포인터와 불변식으로 정의하고, 순회의 값이 이동량이 아니라 의존 읽기의 지연임을 설명할 수 있다. 독립된 순회 여럿을 같이 진행하면 왜 걸음당 값이 스무 배 주는지 설명할 수 있다.

이 말부터

포인터 추적	읽은 값이 다음에 읽을 주소인 읽기의 연쇄. 읽기 하나가 끝나야 다음이 시작된다
지연	읽기 하나를 시작해서 값이 도착할 때까지의 시간. 대역폭과 다른 양이다
의존 읽기	앞 읽기의 결과에 주소가 달린 읽기. 프로세서가 미리 가져올 수 없다

디딤돌

연결 리스트의 정의는 짧다. 노드마다 값 하나와 주소 하나다.

정의 4.1: 연결 리스트

> 연결 리스트는 노드의 집합과 머리 주소 h 다. 노드는 값 v 와 다음 노드의 주소 $next$ 의 짝이고, 마지막 노드의 $next$ 는 없음이다. 연산은 머리에서 시작해 $next$ 를 따라가는 순회, 그리고 어느 노드 뒤에 새 노드를 끼우는 삽입이다.
>
> **불변식** 머리에서 $next$ 를 따라가면 모든 노드를 정확히 한 번씩 지나 없음에 닿는다. 노드의 주소에는 아무 제약이 없다.

불변식의 마지막 문장이 이 구조의 전부다. 주소에 제약이 없으니 삽입이 상수이고, 주소에 제약이 없으니 순회가 무작위 접근이다. 0.1 의 정리는 무작위 접근의 이동량을 원소당 라인 하나로 셌다. 연결 리스트의 순회는 거기에 하나를 더 치른다.

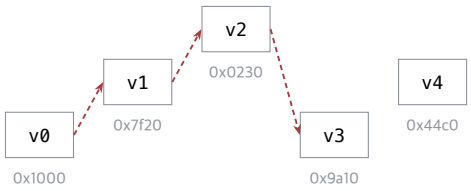
정리 4.2: 순회의 지연

> 노드 n 개가 캐시보다 큰 메모리에 무작위로 놓인 연결 리스트를 순회하면, 이동량은 n 라인이고 시간은 $n \cdot L$ 이상이다. L 은 라인 하나를 가져오는 지연이다. 같은 n 라인을 배열에서 무작위로 읽을 때의 시간은 $n \cdot L/P$ 에 가깝다. P 는 프로세서가 동시에 기다릴 수 있는 읽기의 수다.

증명. 노드 $i+1$ 의 주소는 노드 i 를 읽어야 안다. 그러므로 읽기 $i+1$ 은 읽기 i 의 값이 도착한 뒤에 시작하고, 읽기 n 개가 직렬로 nL 이 든다. 배열의 무작위 읽기는 주소가 미리 정해져 있어 프로세서가 P 개까지 동시에 기다리므로 nL/P 다. ■

예제 4.1: 지연으로 세는 순회

> 라인 지연 $L = 100$ 나노초, 노드 백만 개면 순회에 최소 0.1 초가 든다. 이동량 64MB 를 대역폭 17GB/s 로 옮기는 시간은 4 밀리초다. 같은 라인 수인데 스물다섯 배다. 차이가 전부 기다림이다.



같은 값 다섯 개, 라인 다섯 개

그림 4.1.. 힙에 흩어진 연결 리스트. 노드마다 주소가 다른 라인이라 다음 노드를 읽을 때마다 캐시 미스 하나다. 1.1 의 연속 배열과 짝이다

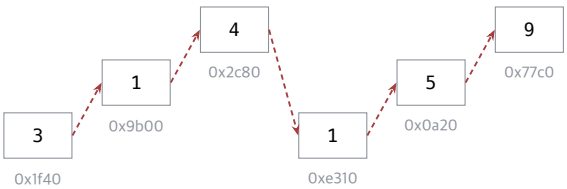


그림 4.2.. 연결 리스트의 노드는 메모리에 흩어져 있고, 화살표 하나를 따라갈 때마다 라인 하나를 기다린다

균열

노드 800만 개의 고리 둘을 만들어 백만 걸음 따라갔다. 하나는 순회 순서가 메모리 순서와 같고, 하나는 무작위 순열이다. 그리고 독립된 고리 4,096 개를 한 걸음씩 같이 갔다.

실험 4.1 포인터 추적. 혼자 갈 때와 4,096 개가 같이 갈 때

고리	걸음 수	시간 ms	걸음당 ns
순차 고리, 하나	1000000	132	132
무작위 고리, 하나	1000000	237	237
순차 고리, 4096 개 같이	4000000	36.7	9.17
무작위 고리, 4096 개 같이	4000000	50.4	12.6

측정 조건. 노드 800 만 개 (next 배열 64MB). 혼자 가는 고리는 백만 걸음, 같이 가는 고리는 합쳐 400 만 걸음. 다섯 번 중앙값.

혼자 가는 고리는 순차가 걸음당 132 나노초, 무작위가 237 나노초다. 차이 105 나노초가 정리 4.2 의 L 이다. 순차 쪽 132 나노초는 이 책이 파이썬으로 걸음을 세는 값이라 기계의 값이 아니다. 두 행의 차이만 기계의 값이다.

같이 가는 고리는 걸음당 9 와 13 나노초다. 무작위 쪽이 혼자 갈 때의 19 분의 1 이다. 고리 4,096 개는 서로를 기다리지 않으므로 프로세서가 읽기 수천 개를 한꺼번에 내고, 지연이 겹쳐 사라진다. 정리의 P 가 여기서 나온다.

관찰

연결 리스트가 느린 것은 흩어져 있어서가 아니라 하나씩이어서다. 흩어진 것을 한꺼번에 읽으면 열아홉 배 빠르다.

다시 쓰기

이 절의 다시 쓰기는 정리 4.2 의 둘째 문장이다. 지연은 겹쳐서 숨길 수 있고, 겹치려면 주소가 미리 정해져 있어야 한다. 연결 리스트가 남은 자리는 전부 이 조건을 만족하는 자리다. 순회를 안 하거나, 순회 여럿을 동시에 하거나, 순회 순서를 미리 배열로 뽑아 두거나.

명제 4.3: 지연을 숨기는 세 조건

포인터 추적의 값 nL 은 다음 중 하나가 성립할 때 nL/P 로 준다. 독립된 추적 P 개를 같이 진행한다. 추적 순서가 미리 인덱스 배열로 있어 프로세서가 주소를 안다. 또는 추적하지 않고 머리만 쓴다.

증명. 첫째와 둘째는 정리 4.2 의 배열 경우와 같다. 읽기 P 개의 주소가 서로 독립이면 동시에 낼 수 있다. 셋째는 걸음이 하나이므로 $n = 1$ 이다. ■

이 세 조건이 이 장의 나머지 두 절이다. 4.2 의 자유 리스트는 머리만 쓰고, 4.3 의 인덱스 링크는 순서를 배열로 뽑아 둔다. 그리고 12 장의 그래프 탐색은 이웃 여럿을 한꺼번에 읽는다. 연결 리스트는 사라진 것이 아니라 세 조건 뒤로 숨었다.

구현

고리를 만들고 따라가는 넷이다. 혼자 가는 것과 같이 가는 것이 코드로는 한 줄 차이다.

지금 보는 코드 next 배열 위의 고리. chase 는 정수 하나를 따라가고, chase_many 는 정수 배열을 따라간다.

구현 무작위 고리 만들기 `samples/dsai/linked.py:19`

```python
def chain_random(n: int, seed: int) -> np.ndarray:
    """무작위 순열을 따라 하나의 큰 고리를 만든다. 메모리 순서와 순회 순서가 다르다."""
    rng = np.random.default_rng(seed)
    order = rng.permutation(n)
    nxt = np.empty(n, dtype=np.int64)
    nxt[order[:-1]] = order[1:]
    nxt[order[-1]] = order[0]
    return nxt
```

읽어 보면 순열 `order` 를 따라 $next[order_k] = order_{k+1}$ 로 이어 고리 하나를 만든다. 메모리 순서와 순회 순서가 무관해진다.

구현 혼자 가기와 같이 가기 `samples/dsai/linked.py:29`

```python
def chase(nxt: np.ndarray, start: int, steps: int) -> int:
    """정의 4.1 의 순회. 읽기 하나가 다음 읽기의 주소를 정한다."""
    i = start
    for _ in range(steps):
        i = nxt[i]
    return int(i)


def chase_many(nxt: np.ndarray, starts: np.ndarray, steps: int) -> np.ndarray:
    """독립된 고리 k 개를 한 걸음씩 같이 간다. 읽기 k 개가 서로를 기다리지 않는다."""
    idx = starts.copy()
    for _ in range(steps):
        idx = nxt[idx]
    return idx
```

알고리즘 4.1 포인터 추적 `dsai.linked.chase`

입력: $next$ 배열, 시작 i, 걸음 수 m

반복 m 번 **동안**
  $i \leftarrow next[i]$ ▷ 이 읽기가 끝나야 다음 주소를 안다
끝 반복
반환 i

`chase_many` 의 반복문 몸체는 `idx = nxt[idx]` 하나인데 `idx` 가 배열이다. 0.1 의 인덱스 배열 접근이고, 원소 4,096 개의 주소가 한꺼번에 나간다.

분석

연결 리스트의 판정은 이 책에서 가장 미묘하다. 정의는 그대로 유효하고, 순회는 이제 안 맞는다. 교과서는 순회를 $O(n)$ 으로 적고 배열과 같다고 말한다. 연산 수로는 같다. 지연

으로는 스물다섯 배다. 그래서 연결 리스트를 순회하는 코드는 이 책이 다루는 시스템에서 거의 사라졌고, 순회하지 않는 연결 리스트만 남았다.

이 절의 정리는 12장에서 크게 다시 나온다. 그래프 탐색은 본질이 포인터 추적이고, 근접 그래프 인덱스는 그 추적을 수십만 번 한다. 그 인덱스가 빠른 이유는 명제 4.3 의 첫째와 둘째를 동시에 쓰기 때문이다. 이웃 목록을 배열로 두고 여럿을 한꺼번에 읽는다.

현장에서

> Drepper 의 2007 년 메모리 해설은 이 절과 같은 포인터 추적 실험으로 캐시 층마다의 지연을 재고, 작업 집합이 각 층을 넘을 때 걸음당 시간이 계단으로 오르는 것을 보였다 [확립] [7].

주의

> 실험 4.1 의 혼자 가는 행은 파이썬 반복문의 값이 섞여 있다. 두 행의 차이만 읽는다. 절대값을 다른 언어의 값과 견주지 않는다.

이 절은 연결 리스트를 재는 자로 지연 L 하나를 얻었다. 그 자로 다음 두 절의 구조를 잰다. 자유 리스트는 걸음이 하나라 L 하나이고, 인덱스 링크는 걸음이 n 개지만 겹쳐서 L/P 씩이다.

되짚기

> 연결 리스트를 정의했고, 순회의 값이 이동량이 아니라 의존 읽기의 지연 nL 임을 증명했다. 독립된 추적 4,096 개를 같이 가면 열아홉 배 빨랐다. 다음 절에서 추적하지 않고 머리만 쓰는 연결 리스트, 자유 리스트를 세운다.

4.2 자유 리스트와 블록 풀. 할당자 안의 연결 리스트

4.1 의 명제는 연결 리스트가 남는 조건 셋을 적었다. 그중 셋째가 머리만 쓰는 것이다. 그 조건을 정확히 만족하는 구조가 자유 리스트다. 빈 칸끼리 이어 두고, 줄 때는 머리를 떼고 받을 때는 머리에 붙인다. 순회가 없으니 지연이 없고, 삽입과 삭제가 상수인 연결 리스트의 장점만 남는다.

2.4 의 블록 풀이 사실 이것이었다. 빈 블록 번호를 목록으로 들고 끝에서 하나 꺼냈다. 이 절은 그것을 정의로 세우고, 2.2 의 첫 맞춤과 시간을 견준다.

학습 목표

> 자유 리스트를 정의하고, 할당과 반납이 각각 읽기 하나 쓰기 하나임을 보일 수 있다. 크기 등급별 자유 리스트가 내부 단편화를 얼마나 만드는지 계산하고, 첫 맞춤과의 시간 차이를 실측으로 설명할 수 있다.

이 말부터

자유 리스트	빈 칸을 연결 리스트로 이어 둔 것. 머리가 다음에 줄 칸이다
크기 등급	요청 크기를 몇 개의 고정 크기로 올림한 것. 등급마다 자유 리스트 하나를 둔다
슬랩	같은 등급의 칸을 모아 둔 메모리 덩어리. 등급별 자유 리스트가 그 위에 선다

디딤돌

2.2 의 첫 맞춤은 빈 구간 목록을 앞에서부터 훑었다. 구간이 길이가 제각각이라 맞는 것을 찾아야 했기 때문이다. 칸의 크기가 전부 같으면 찾을 것이 없다. 아무 빈 칸이나 주면 된다.

정의 4.4: 자유 리스트

> 크기가 같은 칸 c 개의 풀에서, 빈 칸들을 연결 리스트로 잇고 머리 h 를 둔다. 빈 칸은 자기 자리에 다음 빈 칸의 번호를 적는다. 할당은 h 를 돌려주고 $h \leftarrow next[h]$ 로 옮기는 것이고, 반납은 $next[i] \leftarrow h$, $h \leftarrow i$ 다.
>
> **불변식** 빈 칸은 정확히 한 번 리스트에 있고, 쓰이는 칸은 리스트에 없다. 빈 칸의 값 자리가 곧 링크 자리라 링크에 따로 메모리가 들지 않는다.

불변식의 둘째 문장이 이 구조의 재치다. 빈 칸은 어차피 값이 없으니 그 자리에 링크를 적는다. 그래서 자유 리스트는 풀의 크기 밖에 메모리를 한 바이트도 더 쓰지 않는다.

명제 4.5: 자유 리스트의 비용

> 할당과 반납은 각각 읽기 하나와 쓰기 하나이고, 이동량은 라인 하나다. 풀 크기와 빈 칸 수에 무관하다. 반납한 칸이 다음 할당에 먼저 나가므로, 방금 쓴 칸이 캐시에 남아 있으면 그 라인은 이동량에 들지 않는다.

증명. 할당은 $next[h]$ 를 읽고 h 를 쓴다. 반납은 $next[i]$ 를 쓰고 h 를 쓴다. 어느 쪽도 리스트를 따라가지 않는다. 후입선출이므로 마지막에 반납한 칸이 먼저 할당되고, 그 칸의 라인은 방금 접근한 라인이다. ■

예제 4.2: 칸 넷의 자유 리스트

> 칸 $0, 1, 2, 3$ 이 비어 있고 $next = (1, 2, 3, -1)$, $h = 0$ 이다. 할당하면 0 을 주고 $h = 1$ 이 된다. 한 번 더 하면 1 을 주고 $h = 2$ 다. 칸 0 을 반납하면 $next[0] = 2$, $h = 0$ 이다. 다음 할당은 0 을 준다. 방금 놓은 칸이다.

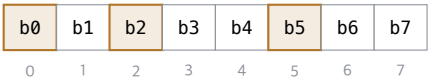


그림 4.3.. 블록 풀. 진한 칸이 쓰는 중이고 나머지가 자유 리스트에 꿰어 있다. 할당은 머리를 떼는 것, 반납은 머리에 꽂는 것이다

균열

크기가 제각각인 요청에는 자유 리스트를 바로 쓸 수 없다. 칸 크기가 하나이기 때문이다. 해법은 크기를 몇 등급으로 올림하고 등급마다 자유 리스트를 두는 것이다. 그러면 요청마다 등급의 리스트에서 머리를 뗀다. 그 값이 얼마이고, 올림 때문에 생기는 내부 단편화가 얼마인지를 2.2 의 첫 맞춤과 견줬다.

실험 4.2 첫 맞춤과 등급별 자유 리스트

할당자	시간 ms	요청당 ns	내부 단편화 퍼센트
첫 맞춤 연속	54.5	2723	0.00
자유 리스트 (등급 8 개)	28.9	1446	30.4

측정 조건. 요청 2 만 개, 동시에 256 개, 길이 16 에서 2048 사이의 로그정규 분포. 등급은 16 부터 2048 까지 2 의 거듭제곱 여덟 개. 다섯 번 중앙값.

자유 리스트가 요청당 두 배 빠르다. 첫 맞춤은 빈 구간 목록을 훑고, 자유 리스트는 머리를 뗀다. 그 대가로 내부 단편화가 30 퍼센트다. 2 의 거듭제곱 등급은 최악의 경우 요청의 두 배 가까이를 주기 때문이다. 2.2 의 표에서 연속 할당의 내부 단편화가 0 이었던 것과 정확히 반대다.

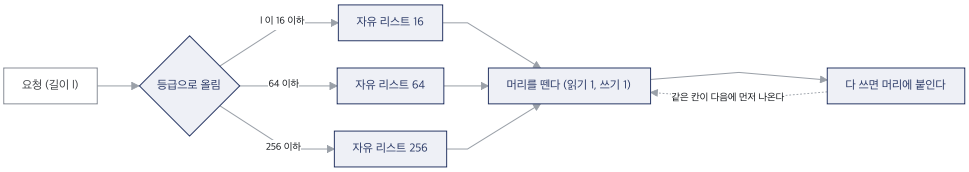


그림 4.4.. 요청을 등급으로 올림하고 그 등급의 자유 리스트에서 머리를 뗀다. 반납은 머리에 붙인다

관찰

할당자는 시간과 단편화를 맞바꾼다. 첫 맞춤은 외부 단편화와 훑는 시간, 자유 리스트는 내부 단편화와 상수 시간이다.

다시 쓰기

등급을 정의에 넣는다. 등급의 수와 간격이 단편화를 정한다.

정의 4.6: 크기 등급 풀

크기 등급 $s_1 < s_2 < \cdots < s_k$ 와 등급마다의 자유 리스트 $F_1, \ldots, F_k$ 의 묶음이다. 길이 ℓ 인 요청은 $\ell \le s_j$ 인 가장 작은 s_j 의 리스트에서 칸 하나를 받는다.

불변식 요청은 자기 등급의 칸에만 있고, 칸의 크기는 요청 길이 이상이다. 내부 단편화는 $\sum (s_{j(\ell)} - \ell)$ 이다.

이 정의의 불변식이 실험 4.2 의 30 퍼센트를 낸 자리다. 등급이 성기면 올림이 크고, 촘촘하면 리스트가 많다. 그 둘의 관계를 등급 비율 하나로 적으면 아래 명제다.

명제 4.7: 등급 간격과 내부 단편화

등급이 비율 r 의 등비수열이면 요청 하나의 내부 단편화는 요청 길이의 $r - 1$ 배 미만이고, 길이가 등급 사이에서 고르게 분포하면 평균은 대략 $(r-1)/2$ 배다. $r = 2$ 이면 최악 100 퍼센트, 평균 근처 30 퍼센트다. $r = \sqrt{2}$ 이면 최악 41 퍼센트, 평균 근처 17 퍼센트이고 등급 수는 두 배가 된다.

증명. 길이 ℓ 이 $(s_{j-1}, s_j]$ 에 있으면 낭비는 $s_j - \ell < s_j - s_{j-1} = (r-1)s_{j-1} < (r-1)\ell$ 이다. 평균은 구간의 절반이다. ■

실험 4.2 의 30 퍼센트가 이 명제의 $r = 2$ 다. 등급을 촘촘히 하면 단편화가 줄고 리스트 수가 늘지만, 리스트 하나는 정수 하나이니 거의 공짜다. 실제 할당자들이 등급을 수십 개 두는 이유가 이것이다.

구현

자유 리스트와 등급 풀이다. 할당과 반납이 각각 세 줄이다.

지금 보는 코드 빈 칸의 자리에 다음 빈 칸 번호를 적어 두는 자유 리스트. 풀 밖에 메모리를 쓰지 않는다.

구현 머리 떼기와 머리에 붙이기 `samples/dsai/freelist.py:23`

```
    def alloc(self) -> int:
        """머리를 뗀다. 읽기 하나, 쓰기 하나."""
        i = self.head
        if i < 0:
            raise MemoryError("빈 칸이 없다")
        self.head = self.nxt[i]
        self.in_use += 1
        return i

    def free(self, i: int) -> None:
        """머리에 붙인다. 최근에 놓은 칸이 다음에 먼저 나간다.
        캐시에 남아 있을 확률이 높다.
        """
        self.nxt[i] = self.head
        self.head = i
        self.in_use -= 1
```

읽어 보면 `alloc` 은 머리를 돌려주고 머리를 그 칸의 `nxt` 로 옮긴다. `free` 는 그 칸의 `nxt` 에 옛 머리를 적고 머리를 그 칸으로 옮긴다. 둘 다 명제 4.5 의 읽기 하나 쓰기 하나다.

알고리즘 4.2 자유 리스트 할당과 반납 `dsai.freelist.FreeList.alloc`

입력: 링크 배열 $next$, 머리 h

함수 ALLOC
    $i \leftarrow h$; $h \leftarrow next[i]$; **반환** i
끝 함수
함수 FREE(i)
    $next[i] \leftarrow h$; $h \leftarrow i$
끝 함수

구현 등급으로 올림해서 그 리스트에서 `samples/dsai/freelist.py:50`

```
    def size_class(self, length: int) -> int:
        for s in self.sizes:
            if length <= s:
                return s
        raise ValueError(f"등급보다 큰 요청 {length}")

    def alloc(self, length: int) -> tuple[int, int]:
        s = self.size_class(length)
        return s, self.lists[s].alloc()
```

분석

자유 리스트는 그대로 유효하다. 연결 리스트 중 유일하게 조정 없이 남은 것이다. 이유는 명제 4.3 의 셋째 조건이다. 머리만 쓰고 순회하지 않으므로 4.1 의 지연이 들지 않는다. 2.4 의 블록 풀, 5장의 배칭 큐, 9장의 트라이 노드 풀이 전부 이 구조 위에 선다.

현장에서

> Bonwick 이 1994 년에 낸 슬랩 할당자는 커널 객체를 크기 등급별 캐시에 두고, 반납된 객체를 초기화한 채로 두었다가 다시 준다 [확립] [17]. 등급별 자유 리스트에 객체의 상태를 남겨 두는 것을 더한 것이고, 그 뒤의 여러 커널 할당자가 이 설계를 따랐다.

역사 한 토막

> CPython 은 512 바이트 이하의 작은 객체를 8 바이트 간격의 등급 64 개로 나눠 자체 할당자로 관리한다. 공개 저장소의 `obmalloc.c` 에 그 등급표가 있다 [확립].

두 사례가 같은 것을 말한다. 크기가 같은 칸을 자유 리스트로 관리하는 것은 커널에서도 인터프리터에서도 30 년째 같은 답이다. 바뀐 것은 그 칸에 무엇이 드느냐이고, 이 책에서는 2.4 의 KV 블록이 든다.

되짚기

> 자유 리스트를 정의했고, 할당과 반납이 라인 하나임을 보였다. 등급별 풀은 첫 맞춤보다 두 배 빠르고 내부 단편화 30 퍼센트를 치른다. 다음 절에서 연결 리스트를 순회해야 할 때 그 순서를 배열로 뽑아 두는 법을 세운다.

4.3 배열 안의 연결 리스트. 인덱스 링크와 아레나

4.2 은 순회하지 않는 연결 리스트였다. 순회해야 하는 연결 리스트는 어떻게 남는가. 4.1 의 명제가 둘째 조건을 적었다. 순회 순서를 미리 배열로 뽑아 두면 지연이 겹친다. 이 절은 그 조건을 구조로 만든다. 노드를 힙에 흩어 두지 않고 배열 하나에 두고, 포인터 대신 칸 번호를 쓴다.

칸 번호로 잇는 연결 리스트는 두 가지를 얻는다. 노드가 배열 안에 있으니 순회 순서를 인덱스 배열로 뽑을 수 있고, 뽑은 순서대로 다시 놓으면 순차 접근이 된다. 그 다시 놓기가 압축이고, 이 책의 뒤에서 트라이와 그래프가 같은 압축을 쓴다.

학습 목표

> 인덱스 링크 리스트를 아레나와 칸 번호로 정의하고, 삽입이 상수이면서 순회 순서를 배열로 뽑을 수 있음을 보일 수 있다. 압축이 순회의 이동량을 무작위 접근에서 순차 접근으로 바꾸는 것을 실측으로 설명할 수 있다.

이 말부터

아레나	노드를 전부 담는 배열 하나. 노드는 아레나의 칸이고 주소는 칸 번호다
인덱스 링크	포인터 대신 칸 번호로 다음 노드를 가리키는 것. 4 바이트로 충분하다
압축	노드를 순회 순서대로 아레나에 다시 놓는 것. 그 뒤 순회는 순차 접근이다

디딤돌

연결 리스트의 노드를 힙에 하나씩 잡으면 1.1 의 참조 배열과 같은 일이 난다. 노드가 어디 있는지 아무도 모른다. 노드를 배열 하나에 두면 적어도 어디 있는지는 안다. 그 배열이 아레나다.

정의 4.8: 인덱스 링크 리스트

> 아레나는 값 배열 $V[0..c)$ 와 링크 배열 $next[0..c)$ 의 짝이다. 리스트는 머리 칸 번호 h 이고, 노드 i 의 다음은 $next[i]$ 이며 없음은 -1 이다. 쓰지 않는 칸은 4.2 의 자유 리스트로 이어 둔다. 삽입은 자유 리스트에서 칸을 받아 링크 둘을 고치는 것이다.
>
> **불변식** 정의 4.1 의 불변식에 더해, 모든 노드가 아레나 안에 있고 링크는 $[0, c)$ 의 정수 또는 -1 이다.

이 정의에서 삽입의 값은 그대로 상수다. 칸을 받는 것이 자유 리스트의 머리 떼기이고 링크 고치기가 쓰기 둘이다. 달라진 것은 순회다. 노드가 배열 안에 있으므로 순회 순서를 정수 배열로 적을 수 있고, 그 배열로 0.1 의 인덱스 접근을 한다.

명제 4.9: 순회 순서를 뽑는 값과 쓰는 값

> 인덱스 링크 리스트의 순회 순서 배열 O 를 뽑는 것은 포인터 추적 n 걸음이고 값은 nL 이다. O 를 한 번 뽑아 두면 그 뒤의 순회는 $V[O]$ 의 인덱스 접근이고, 값은 0.1 의 무작위 접근 nL/P 다. O 가 $(0, 1, \dots, n-1)$ 이면 순차 접근이고 값은 $\lceil ns/B \rceil$ 라인의 이동량뿐이다.

증명. 첫 문장은 정리 4.2 다. 둘째는 주소가 O 로 미리 정해져 있으므로 명제 4.3 의 둘째 조건이다. 셋째는 정의 1.1 의 연속성이다. ■

명제의 세 줄이 이 절의 실험 세 열이다. 순서를 뽑는 값, 뽑은 순서로 읽는 값, 순차로 읽는 값. 작은 예로 세 상태를 손으로 만들어 보면 아래와 같다.

예제 4.3: 끼운 뒤의 순서

> 칸 $0, 1, 2$ 에 값 $1, 2, 3$ 을 차례로 붙이면 $next = (1, 2, -1)$ 이고 순회 순서는 $(0, 1, 2)$ 다. 칸 0 뒤에 9 를 끼우면 새 칸 3 을 받아 $next[3] = 1$, $next[0] = 3$ 이 되고 순회 순서는 $(0, 3, 1, 2)$ 다. 메모리 순서와 어긋났다. 압축하면 값이 $(1, 9, 2, 3)$ 순으로 칸 0..3 에 다시 놓이고 순서가 $(0, 1, 2, 3)$ 이 된다.

끼운 직후. 순회는 0, 3, 1, 2 로 칸 3 에 갔다 돌아온다

그림 4.5.. 예제 4.6. 끼운 뒤 순회 순서가 메모리 순서와 어긋나고, 압축이 둘을 다시 맞춘다

균열

노드 400만 개를 붙인 뒤 무작위 자리에 0 개, 40만 개, 400만 개를 끼우고, 순회 순서로 모아 더하는 시간을 압축 전후로 쟀다.

실험 4.3 끼운 수에 따른 순회 시간, 압축 전과 후

끼운 수	압축 전 ms	압축 후 ms	배열 sum ms
0	9.72	9.22	1.18
400000	15.9	13.7	1.77
4000000	78.7	36.4	5.31

측정 조건. 노드 400만 개에 무작위 자리에 끼운 뒤 순회 순서 배열로 모아 더한다. 압축은 순회 순서대로 다시 놓기. 배열 sum 은 같은 값의 연속 배열. 다섯 번 중앙값.

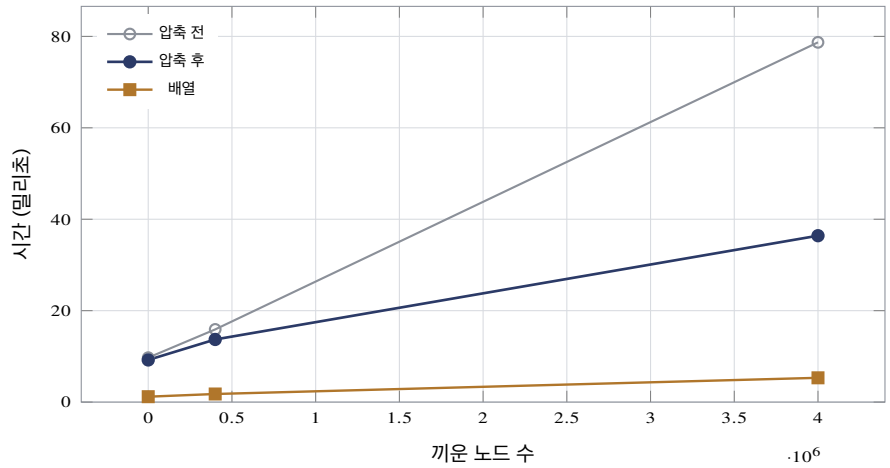


그림 4.6.. 끼운 노드 수에 따른 순회 시간. 압축이 흩어진 노드를 다시 모으고, 배열은 처음부터 그 자리에 있다

끼운 것이 없으면 압축 전후가 같다. 순회 순서가 이미 메모리 순서다. 400만 개를 끼우면 순서가 뒤섞이고 순회가 79 밀리초, 압축하면 36 밀리초다. 같은 값을 연속 배열로 더하면 5 밀리초다. 압축 후에도 배열의 일곱 배인 것은 인덱스 배열 O 를 읽고 그것으로 모으는 값이다. 순서가 순차여도 인덱스를 거치면 원소당 8 바이트를 더 읽는다.

관찰

압축은 순회를 무작위에서 순차로 바꾸고, 그것으로 두 배를 얻는다. 인덱스 자체를 없애면 일곱 배를 더 얻는다. 그것이 배열이다.

다시 쓰기

압축을 알고리즘으로 세운다. 순회 순서를 한 번 뽑고, 그 순서대로 값을 옮기고, 링크를 $i \to i+1$ 로 다시 쓴다.

알고리즘 4.3 압축 `dsai.linked.IndexList.compact`

입력 : 아레나 $(V, next)$, 머리 h, 크기 n

1: $O \leftarrow$ 머리에서 $next$ 를 따라 뽑은 순회 순서 ▷ 포인터 추적 n 걸음. 한 번만
2: $V[0..n) \leftarrow V[O]$ ▷ 인덱스 접근 한 번
3: $next[i] \leftarrow i+1$ $(i < n-1)$; $next[n-1] \leftarrow -1$; $h \leftarrow 0$
4: 칸 $n..c$ 를 자유 리스트로 되돌린다

압축의 값은 포인터 추적 한 번과 인덱스 접근 한 번이다. 그 뒤의 순회 k 번이 전부 순차이므로, 순회가 두 번 이상이면 압축이 이득이다. 1.2 의 뷰를 옮길지 말지의 판단과 같은 모양이다. 한 번 옮기고 여러 번 싸게 읽는다.

명제 4.10: 압축의 손익

압축 뒤 순회를 k 번 하면 총비용은 $nL + k \cdot nS$ 이고, 압축 없이 k 번 순회하면 $k \cdot nL/P$ 다. S 는 순차 접근의 원소당 값이다. $L/(L/P - S)$ 번 이상 순회하면 압축이 이긴다. 이 기계에서는 두 번이다.

구현

아레나 위의 리스트다. 붙이기, 끼우기, 순서 뽑기, 압축이 한 클래스에 있다.

지금 보는 코드 값 배열과 링크 배열의 짝. 빈 칸은 자유 리스트로, 끼우기는 링크 둘 고치기로.

구현 끼우기와 순서 뽑기 `samples/dsai/linked.py:78`

```
78      def insert_after(self, i: int, v) -> int:
79          """칸 i 뒤에 끼운다. 배열이면 뒤를 전부 밀어야 하는 연산이 상수다."""
80          j = self.free.pop()
81          self.val[j] = v
82          self.nxt[j] = self.nxt[i]
83          self.nxt[i] = j
84          if self.tail == i:
85              self.tail = j
86          self.size += 1
87          return j
88
89      def order(self) -> np.ndarray:
90          """순회 순서대로 칸 번호를 늘어놓는다."""
91          out = np.empty(self.size, dtype=np.int64)
92          i, k = self.head, 0
93          while i >= 0:
94              out[k] = i
95              i = self.nxt[i]
96              k += 1
97          return out
```

읽어 보면 `insert_after` 는 자유 리스트에서 칸을 받아 링크 둘을 고친다. `order` 는 머리에서 링크를 따라 칸 번호를 배열에 적는다. 이 배열이 명제 4.9 의 O 이고, 실험 4.3 의 압축 전 행이 이 배열로 모은 것이다.

구현 순서대로 모으기와 압축 `samples/dsai/linked.py:102`

```
    def gather_sum(self, order: np.ndarray) -> float:
        """순회 순서를 인덱스 배열로 받아 한 번에 모아 더한다.
        이동량은 순서가 정한다.
        """
        return float(self.val[order].sum())

    def compact(self) -> None:
        """알고리즘 4.2. 순회 순서대로 칸을 다시 놓는다. 그 뒤 순회는 순차 접근이다."""
        order = self.order()
        self.val[:self.size] = self.val[order]
        self.nxt[:self.size - 1] = np.arange(1, self.size, dtype=np.int64)
        self.nxt[self.size - 1] = -1
        self.head, self.tail = 0, self.size - 1
        self.free = list(range(len(self.val) - 1, self.size - 1, -1))
```

분석

배열 안의 연결 리스트는 조정해서 유효하다. 정의에 아레나 하나가 더 붙었고, 그 아레나가 순회를 배열의 인덱스 접근으로 바꾼다. 이 구조가 이 책의 뒤에서 세 번 나온다. 9장의 트라이는 노드를 아레나에 두고 자식을 칸 번호로 가리키며, 12장의 근접 그래프는 이웃 목록을 아레나의 연속 구간으로 두고, 15장의 캐시 축출은 최근 사용 순서를 인덱스 링크로 든다. 셋 다 포인터를 칸 번호로 바꾼 것이고, 셋 다 압축을 한다.

현장에서

> 가속기 위의 그래프 라이브러리들은 인접 리스트를 이 절의 아레나 형태, 즉 3.1 의 CSR 로 둔다. 노드 하나의 이웃이 연속 구간이라 한꺼번에 읽힌다. Harris 와 동료들이 정리한 NumPy 의 인덱스 배열 접근이 그 읽기의 원형이다 [확립] [6].

이 장의 판정을 여기서 낸다. 연결 리스트는 조정해서 유효하다. 정의는 남았고, 힙에 흩어진 노드와 순회는 안 맞는다. 남은 것은 셋이다. 머리만 쓰는 자유 리스트, 순서를 배열로 뽑는 인덱스 링크, 그리고 압축.

주의

> 압축은 칸 번호를 바꾼다. 밖에서 칸 번호를 들고 있는 것이 있으면 압축 뒤에 틀린 칸을 가리킨다. 압축하려면 번호를 든 쪽을 전부 알아야 한다.

그 번호를 든 쪽을 고치는 재배치 표가 연습문제 4.8 이다. 표 하나면 압축이 안전해지고, 그 표를 든 채 압축하는 것이 9장과 12장의 구조가 하는 일이다.

되짚기

> 인덱스 링크 리스트를 아레나로 정의했고, 순회 순서를 배열로 뽑아 두면 무작위 접근이, 압축하면 순차 접근이 됨을 봤다. 압축이 두 배, 배열이 일곱 배 더 빨랐다. 다음 장에서 순서가 정해진 두 구조, 스택과 큐가 어떤 이름으로 살아 있는지 본다.

4장을 닫으며

이 장이 세운 것은 조건 셋과 구조 둘이다. 포인터 추적의 지연은 독립된 추적을 겹치거나, 순서를 미리 알거나, 추적을 안 하면 숨는다. 자유 리스트가 셋째 조건이고, 인덱스 링크 아레나가 둘째 조건이다. 첫째 조건은 12장의 그래프가 쓴다.

세 실험의 숫자를 적어 둔다. 혼자 가는 무작위 고리는 순차보다 걸음당 105 나노초 느렸고, 그 105 나노초가 라인 하나의 지연이다. 독립된 고리 4,096 개를 같이 가자 걸음당 13 나노초로 열아홉 배 빨랐다. 등급별 자유 리스트는 첫 맞춤보다 요청당 두 배 빠르고 내부 단편화 30 퍼센트를 치렀다. 400만 개를 끼워 뒤섞인 리스트는 압축 뒤 두 배 빨랐고, 같은 값의 연속 배열은 거기서 일곱 배 더 빨랐다.

정정 하나를 남긴다. 압축 실험을 처음에는 파이썬 반복문으로 순회하며 쟀다. 압축 전후가 거의 같았다. 반복문의 값이 걸음당 백 나노초를 넘어 라인 지연을 덮었기 때문이다. 순회 순서를 배열로 뽑아 두고 그 배열로 모으는 방식으로 다시 쟀고, 그제야 압축의 효과가 보였다. 재는 방법이 재려는 것을 덮는 일이 이 장에서 두 번 있었다. 실험 4.1 의 혼자 가는 행도 같은 이유로 차이만 읽는다.

옛 구조의 판정

> 연결 리스트는 조정해서 유효하다. 정의는 그대로이고, 힙에 흩어진 노드를 순회하는 구현이 안 맞는다. 근거는 실험 4.1 과 4.3 이다. 의존 읽기 하나가 105 나노초이고, 순서를 배열로 뽑아 압축하면 두 배, 배열로 바꾸면 열네 배다. 남은 것은 머리만 쓰는 자유 리스트와 아레나 위의 인덱스 링크다.

이 장의 요약

> 1. 연결 리스트 순회의 값은 이동량 n 라인이 아니라 지연 nL 이다. 읽기 하나가 다음 주소를 정하기 때문이다 4.1
> 2. 지연은 독립된 추적을 겹치거나, 순서를 미리 알거나, 추적을 안 하면 P 분의 1 로 숨는다 4.1
> 3. 자유 리스트는 머리만 쓴다. 할당과 반납이 라인 하나이고 풀 밖에 메모리를 쓰지 않는다 4.2
> 4. 크기 등급 풀은 상수 시간과 내부 단편화를 맞바꾼다. 등급 비율 r 에서 평균 낭비가 $(r-1)/2$ 다 4.2
> 5. 아레나 위의 인덱스 링크는 순회 순서를 배열로 뽑고, 압축이 그 순서를 순차로 만든다 4.3
>
> 다음 장에서 뒤집히는 것: 스택과 큐는 순서가 정해진 배열이다. 그 순서가 자동 미분의 테이프와 추론 서버의 배치가 된다.

필자의 판단

> 필자는 연결 리스트를 첫 장에서 가르치는 순서를 바꿔야 한다고 본다. 배열, 캐시 라인, 지연을 먼저 재고 그다음에 연결 리스트를 자유 리스트로 소개하는 것이 맞다. 순회하는 연결 리스트를 먼저 배운 학생은 그 뒤로 노드를 힙에 하나씩 잡는 습관을 오래 가지고 간다. 그 습관이 이 책이 다루는 시스템에서 가장 비싼 습관이다.

열린 문제

하나, 정리 4.1 의 P 를 이 책은 재지 않았다. 실험 4.1 에서 4,096 개가 열아홉 배를 냈지만, 몇 개부터 포화하는지는 고리 수를 바꿔 가며 재야 한다.
둘, 실험 4.2 의 등급은 2 의 거듭제곱 하나다. 등급 비율을 바꿔 가며 시간과 단편화의 곡선을 그리지 않았다.
셋, 압축은 칸 번호를 바꾼다. 밖에서 번호를 든 구조가 있을 때 압축을 안전하게 하는 방법, 즉 번호 재배치 표를 이 장은 다루지 않았다.

연습문제

4.1 기초 라인 지연 80 나노초, 동시 읽기 $P = 10$ 인 기계에서 노드 천만 개의 연결 리스트를 순회하는 시간과, 같은 노드를 인덱스 배열로 무작위 접근하는 시간을 정리 4.1 로 구하라.

4.2 기초 예제 4.4 의 자유 리스트에서 할당 셋, 반납 (칸 1), 반납 (칸 0), 할당 둘을 차례로 했을 때 머리와 *next* 배열의 최종 상태를 적어라.

4.3 기초 등급 비율 $r = 1.25$ 인 크기 등급 풀에서 16 부터 2048 까지의 등급 수와, 명제 4.4 의 평균 내부 단편화를 구하라.

4.4 기초 예제 4.6 의 리스트에서 칸 1 뒤에 7 을 한 번 더 끼운 뒤의 *next* 배열과 순회 순서를 적고, 압축 뒤의 값 배열을 적어라.

4.5 응용 `dsai.linked.chase_many` 의 고리 수를 1, 4, 16, 64, 256, 1024, 4096 으로 바꿔 걸음당 시간을 재고, 몇 개에서 포화하는지 찾아라. 그것이 이 기계의 P 다.

4.6 응용 `dsai.freelist.SlabPool` 의 등급을 비율 $\sqrt{2}$ 로 만들어 실험 4.2 를 다시 재고, 내부 단편화가 명제 4.4 의 예측과 맞는지 확인하라.

4.7 응용 `dsai.linked.IndexList` 에 삭제 (칸을 리스트에서 빼고 자유 리스트로 되돌리기) 를 더하고, 삭제가 무작위로 일어난 뒤의 순회 시간을 압축 전후로 재라.

4.8 응용 압축이 칸 번호를 바꾸는 문제를 풀어라. 옛 번호에서 새 번호로 가는 재배치 표를 압축이 돌려주게 하고, 밖에서 든 번호를 표로 고치는 함수를 써라.

4.9 탐구 이중 연결 리스트 (앞뒤 링크) 를 아레나에 두면 삭제가 상수가 된다. 링크 배열이 하나 더 들 때 압축의 손익 (명제 4.7) 이 어떻게 달라지는지 논하라.

4.10 탐구 자유 리스트의 후입선출 순서는 캐시에 유리하지만, 여러 스레드가 같은 리스트를 쓰면 머리에서 충돌한다. 스레드마다 리스트를 두는 설계의 단편화와 시간을 논하라.

더 읽을 것

[7] 포인터 추적으로 캐시 층마다의 지연을 재는 법. 실험 4.1 을 다른 기계에서 다시 하고 싶으면 이 문서의 3 장을 따른다. 4.1
[17] 슬랩 할당자의 원전. 크기 등급, 자유 리스트, 객체 상태 보존이 열두 쪽에 있다. 4.2 절의 정의가 여기서 왔다. 4.2
[1] 블록 단위 이동량 모델. 압축이 왜 이득인지를 이 모델로 다시 세면 명제 4.7 이 나온다. 4.3
[6] 인덱스 배열 접근의 원형. 4.3 절의 순서 배열 O 로 모으는 연산이 이 논문의 fancy indexing 이다. 4.3

5장

스택과 큐, 테이프와 배치

1964년 두 쪽짜리 논문이 함수를 원시 연산의 열로 풀어 놓으면 미분을 기계가 구할 수 있다고 적었다 [확립] [18]. 그 열을 거꾸로 읽는 것이 오늘의 모든 학습 프레임워크가 하는 일이고, 그 열을 담는 구조가 스택이다. 2022년의 추론 서버 논문은 요청을 모으는 단위를 배치에서 토큰 한 걸음으로 내렸다 [확립] [8]. 그 단위를 담는 구조가 큐다.

이 장은 0.2 의 첫째 가정을 순서가 있는 배열에 대고 잰다. 스택과 큐는 교과서에서 가장 짧은 장을 받는 구조다. 정의가 두 줄이고 구현이 열 줄이다. 그 정의는 그대로다. 바뀐 것은 원소의 크기와 열의 길이다. 프레임이 국소 미분이 되고, 바이트가 기가바이트가 되고, 큐의 원소가 요청에서 걸음으로 내려간다. 그 변화가 2장의 메모리 문제를 다시 불러오고, 이 장의 마지막 절이 그 자리다.

이 장에서 세우는 것

5.1 스택을 정의하고, 재귀가 깊이 천에서 멈추는 동안 명시적 스택이 백만까지 가는 것을 잰다

5.2 테이프. 앞으로 적고 뒤로 읽는 스택이 입력 수와 무관하게 편미분 전부를 준다

5.3 링 버퍼. 옮기기 없는 큐가 창 통계를 상수 시간에 갱신한다

5.4 배칭 큐. 꺼내는 단위를 요청에서 걸음으로 내리면 지연이 19분의 1 이 된다

5.1 호출 스택과 재귀 깊이. 명시적 스택으로 바꾸기

1.1 의 배열에 꼭대기 번호 하나를 얹으면 스택이다. 끝에 넣고 끝에서 뺀다. 이 구조는 교과서에서 가장 짧은 절을 받고, 대부분의 독자가 재귀 함수를 쓰면서 언어가 대신 관리하는 스택으로만 만난다. 그런데 언어가 관리하는 스택에는 상한이 있고, 그 상한이 이 책의 작업량에서는 자주 걸린다.

깊이 백만의 트리, 길이 십만의 토큰 열, 노드 천만 개의 그래프. 이런 것을 재귀로 훑으면 언어가 먼저 멈춘다. 이 절은 그 상한을 재고, 재귀를 명시적 스택으로 바꾸는 법을 세운다. 그 스택이 다음 절의 테이프가 된다.

학습 목표

> 스택을 배열과 꼭대기 번호로 정의하고, 호출 스택의 프레임과 명시적 스택의 원소가 크기에서 얼마나 다른지 말할 수 있다. 재귀 함수를 명시적 스택으로 바꾸는 절차를 알고리즘으로 적을 수 있다.

이 말부터

호출 스택	함수를 부를 때마다 언어가 쌓는 프레임의 스택. 프레임에 지역 변수와 돌아갈 자리가 든다
프레임	함수 호출 하나가 스택에 차지하는 덩어리. 파이썬에서는 수백 바이트다
명시적 스택	프로그램이 직접 만든 스택. 원소가 정수 하나나 둘이라 프레임보다 백 배 작다

디딤돌

정의 5.1: 스택

> 스택은 값 배열 $B[0..c)$ 와 꼭대기 번호 t 의 짝이다. push 는 $B[t] \leftarrow v$, $t \leftarrow t+1$ 이고 pop 은 $t \leftarrow t-1$ 뒤 $B[t]$ 를 돌려주는 것이다.
>
> **불변식** $0 \le t \le c$ 이고, 원소는 $B[0..t)$ 에 넣은 순서대로 있다. 마지막에 넣은 것이 $B[t-1]$ 이다.

push 와 pop 이 배열 끝의 쓰기와 읽기이므로 값이 상수이고, 2.1 의 두 배 늘리기를 붙이면 상한이 없다. 호출 스택은 이 정의에 두 가지를 더한 것이다. 원소가 프레임이고, 용량이 고정이다.

예제 5.1: 프레임과 정수

깊이 n 의 재귀 호출은 프레임 n 개를 쌓는다. 파이썬의 프레임은 지역 변수 몇 개에 수백 바이트이고, 기본 상한이 천 개다. 같은 일을 명시적 스택으로 하면 원소가 (노드 번호, 깊이) 정수 둘, 16 바이트이고 상한은 메모리다. 천 개 대 억 개다.

그림 5.1.. 배열 위의 명시적 스택. 꼭대기가 자리 3 이고 그 뒤는 예약이다. 재귀의 호출 프레임이 이 칸 하나로 준다

명제 5.2: 재귀와 명시적 스택의 메모리

깊이 n 의 재귀는 프레임 크기 F 에 대해 nF 바이트를 쓰고, 언어의 상한 $n_{\max}$ 을 넘으면 실패한다. 같은 탐색을 원소 크기 s 의 명시적 스택으로 하면 ns 바이트를 쓰고 상한은 메모리뿐이다. 두 탐색의 연산 수는 같다.

증명. 재귀 호출마다 프레임 하나가 쌓이고 돌아올 때 하나가 걷힌다. 명시적 스택은 같은 자리에서 push 하나 pop 하나다. 쌓이는 것의 크기만 F 와 s 로 다르고, 상한은 각각 언어와 배열의 용량이 정한다. ■

균열

자식이 하나뿐인 트리의 깊이를 재귀와 명시적 스택으로 쟀다. 깊이를 백에서 백만까지 늘렸다.

실험 5.1 재귀와 명시적 스택. 깊이에 따른 시간과 실패

깊이	재귀 ms	명시적 스택 ms
100	0.04	0.0509
1000	실패	0.51
10000	실패	2.91
100000	실패	27.8
1000000	실패	279

측정 조건. 자식이 하나뿐인 트리. 재귀 상한은 파이썬 기본값 1,000. 다섯 번 중앙값.

재귀는 깊이 천에서 멈춘다. 상한을 올리면 더 가지만 프레임 크기 때문에 곧 다른 벽에 닿는다. 명시적 스택은 백만까지 깊이에 정비례한다. 원소 하나가 튜플 하나라 백만 개에 수십 MB 다.

관찰

> 재귀가 실패하는 깊이는 문제의 크기가 아니라 언어의 설정이다. 같은 알고리즘을 명시적 스택으로 적으면 그 설정이 사라진다.

다시 쓰기

재귀를 명시적 스택으로 바꾸는 절차를 적는다. 호출을 push 로, 돌아옴을 pop 으로 바꾸고, 프레임에 있던 것을 원소에 넣는다.

알고리즘 5.1 명시적 스택으로 깊이 재기 `dsai.stack.depth_iterative`

입력 : 자식 목록 C, 뿌리 0
1: 스택 $S \leftarrow [(0, 1)]$; $best \leftarrow 0$
2: **조건** S 가 비지 않음 **동안**
3: $(v, d) \leftarrow S.\mathrm{pop}()$; $best \leftarrow \max(best, d)$
4: **반복** 각 자식 $c \in C[v]$ **동안**
5: $S.\mathrm{push}((c, d+1))$
6: **끝 반복**
7: **끝 조건**
8: **반환** $best$

프레임에 있던 것은 지금 노드와 깊이 둘이었다. 그 둘이 원소가 됐다. 돌아온 뒤에 할 일이 있는 재귀는 원소에 그 일의 단계도 넣는다. 그것이 다음 절의 테이프가 하는 일이다. 앞으로 갈 때 할 일을 적어 두고 뒤로 올 때 꺼낸다.

명시적 스택. 원소는 (노드, 깊이). 꼭대기가 오른쪽으로 자란다

그림 5.2. 같은 탐색의 두 스택. 왼쪽은 프레임이 쌓이고 오른쪽은 정수 둘이 쌓인다

구현

배열 스택과 그것으로 깊이를 재는 함수다.

지금 보는 코드 배열과 꼭대기 번호. push 는 끝에 쓰기, pop 은 끝에서 읽기.

구현 배열 스택 `samples/dsai/stack.py:12`

```
class Stack:
    """정의 5.1. 배열과 꼭대기 번호. push 는 끝에 쓰기, pop 은 끝에서 읽기."""

    def __init__(self, capacity: int, dtype=np.int64):
        self.buf = np.empty(capacity, dtype=dtype)
        self.top = 0

    def push(self, v) -> None:
        self.buf[self.top] = v
        self.top += 1

    def pop(self):
        self.top -= 1
        return self.buf[self.top]

    def __len__(self) -> int:
        return self.top
```

읽어 보면 `top` 이 정의 5.1 의 t 다. `push` 와 `pop` 이 각각 쓰기 하나 읽기 하나이고, 배열이 연속이라 꼭대기 근처는 언제나 캐시에 있다.

구현 재귀 대신 명시적 스택 `samples/dsai/stack.py:44`

```
def depth_iterative(children: list[list[int]]) -> int:
    """알고리즘 5.1. 명시적 스택. 프레임이 (노드, 깊이) 정수 둘이다."""
    stack = [(0, 1)]
    best = 0
    while stack:
        node, d = stack.pop()
        best = max(best, d)
        for c in children[node]:
            stack.append((c, d + 1))
    return best
```

분석

스택은 그대로 유효하다. 정의도 구현도 40년 전과 같다. 바뀐 것은 스택을 언어에 맡기지 않는다는 것이다. 이 책의 작업량은 깊이가 크고, 언어의 스택은 프레임이 크고 상한이 있다. 그래서 탐색은 전부 명시적 스택으로 적는다.

현장에서

> 파이썬의 재귀 상한은 인터프리터의 공개 문서에 기본값 1,000 으로 적혀 있고, 그 이유로 C 스택의 넘침을 든다 [확립]. 프레임 하나의 값이 언어마다 다르지만, 상한이 언어에 있다는 것은 같다.

주의

> 명시적 스택은 순서가 재귀와 다를 수 있다. 자식을 push 한 순서의 역순으로 pop 되므로, 순서가 중요하면 자식을 거꾸로 넣는다.

순서가 뒤집히는 것 말고는 재귀와 명시적 스택이 하는 일이 같다. 그래서 이 책의 뒤에 나오는 탐색은 전부 이 절의 알고리즘 꼴로 적는다. 트리도, 트라이도, 그래프도 스택 하나와 반복문 하나다.

되짚기

> 스택을 정의했고, 재귀가 깊이 천에서 멈추는 동안 명시적 스택이 백만까지 가는 것을 봤다. 프레임 대 정수 둘의 차이다. 다음 절에서 앞으로 가며 적어 두고 뒤로 오며 꺼내는 스택, 자동 미분의 테이프를 세운다.

5.2 테이프. 역방향 자동 미분의 스택

5.1 의 마지막 문단이 이 절을 예고했다. 앞으로 갈 때 할 일을 적어 두고 뒤로 올 때 꺼내는 스택. 학습이 하는 일이 정확히 그것이다. 함수를 앞으로 계산하며 연산마다 국소 미분을 적어 두고, 끝에서부터 거꾸로 읽어 기울기를 입력까지 흘린다. 그 적어 두는 배열이 테이프다. 2.3 의 추가 전용 배열을 거꾸로 읽는 것이고, 그래서 스택이다.

이 절은 어텐션이나 학습의 수학을 다루지 않는다. 테이프라는 자료 구조의 정의와 비용만 다룬다. 그 비용이 왜 역방향 한 번이 입력 수만큼의 전진보다 싼지를 정하고, 그 대가로 테이프가 얼마나 커지는지를 잰다.

학습 목표

테이프를 추가 전용 배열과 역순 읽기로 정의하고, 역방향 미분 한 번의 비용이 연산 수에 비례함을 증명할 수 있다. 입력이 n 개일 때 전진 방향 n 번과 역방향 한 번의 비용을 견주고, 테이프의 메모리가 왜 시퀀스 길이에 비례하는지 말할 수 있다.

이 말부터

테이프	앞으로 계산하며 연산마다 (입력, 국소 미분) 을 적은 추가 전용 배열. 뒤로 읽는다
수반 변수	출력의 미분을 어느 변수에 대해 적은 값. 영어로 adjoint 이고 테이프를 뒤로 읽으며 쌓인다
국소 미분	연산 하나의 출력을 그 연산의 입력으로 미분한 값. 곱셈이면 상대 입력이다

디딤돌

5.1 의 명시적 스택에 원소가 (노드, 깊이) 였다. 테이프의 원소는 (출력 변수, 입력 변수들, 국소 미분들) 이다. 그리고 테이프는 스택이되 pop 을 하지 않는다. 앞으로 갈 때 push 만 하고, 뒤로 올 때 끝에서부터 읽기만 한다.

정의 5.3: 테이프

테이프는 변수 값의 배열 V 와 항목의 추가 전용 배열 E 의 짝이다. 항목 $e_k = (o_k, (i_1, \dots, i_m), (d_1, \dots, d_m))$ 은 변수 o_k 가 변수 $i_1, \dots, i_m$ 에서 계산되었고 $\partial V[o_k]/\partial V[i_j] = d_j$ 임을 적는다. 역방향은 수반 배열 $\bar{V}$ 를 0 으로 두고 $\bar{V}[out] = 1$ 로 시작해, E 를 끝에서부터 읽으며 $\bar{V}[i_j] \leftarrow \bar{V}[i_j] + \bar{V}[o_k] \cdot d_j$ 를 한다.

불변식 항목 k 의 입력 변수는 전부 항목 k 이전에 만들어졌다. 그래서 끝에서부터 읽으면 $\bar{V}[o_k]$ 가 완성된 뒤에 그것을 입력으로 흘린다.

불변식이 이 구조의 요점이다. 앞으로 계산한 순서 그대로 적었으므로, 거꾸로 읽으면 어느 변수의 수반 값도 그 변수를 쓰는 항목을 전부 지난 뒤에 쓰인다. 스택이 아니라 큐로 읽으면 이 불변식이 깨진다.

예제 5.2: 항목 둘의 테이프

$a = 2$ 에서 $b = a \cdot a$, $c = b + a$ 를 계산하면 테이프는 $e_1 = (b, (a, a), (a, a)) = (b, (a, a), (2, 2))$, $e_2 = (c, (b, a), (1, 1))$ 이다. 역방향은 $\bar{c} = 1$ 에서 e_2 를 읽어 $\bar{b} = 1$, $\bar{a} = 1$ 이 되고, e_1 을 읽어 $\bar{a} = 1 + 1 \cdot 2 + 1 \cdot 2 = 5$ 가 된다. $c = a^2 + a$ 의 미분 $2a + 1 = 5$ 와 같다.

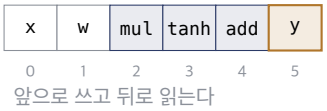


그림 5.3.. 테이프. 전진하며 연산을 차례로 적고, 역방향은 뒤에서 앞으로 읽는다. 5.1 의 스택과 같은 배열이다

정리 5.4: 역방향 미분의 비용

연산 m 개로 계산되는 함수 $f : \mathbb{R}^n \to \mathbb{R}$ 의 모든 편미분 n 개는 테이프 한 번 앞으로, 한 번 뒤로 읽어 얻는다. 비용은 앞으로 m 번, 뒤로 m 번의 상수 배이고 n 에 무관하다. 방향 미분으로 같은 것을 얻으려면 앞으로 n 번 가야 한다. 테이프의 메모리는 m 에 비례한다.

증명. 앞으로 갈 때 연산마다 항목 하나를 적으므로 항목이 m 개다. 뒤로 읽을 때 항목마다 입력 수만큼 곱하고 더하므로 총비용이 항목 수의 상수 배다. 정의 5.3 의 불변식으로 뒤로 한 번 읽으면 모든 변수의 수반 값이 완성되고, 입력 변수들의 수반 값이 편미분 n 개다. 방향 미분은 방향 하나마다 앞으로 한 번이므로 n 방향에 n 번이다. 테이프는 항목 m 개를 전부 들고 있어야 뒤로 읽을 수 있으므로 메모리가 m 에 비례한다. ■

균열

정리의 두 비용을 이 책의 기계에서 쟀다. $f(x) = \sum \tanh(Wx)$ 에서 입력 n 을 16 에서 1,024 까지 늘리며, 테이프 역방향 한 번과 방향 미분 n 번을 견줬다.

실험 5.2 역방향 한 번과 전진 n 번

입력 n	테이프 항목	역방향 ms	전진 n번 ms	비율
16	1023	7.72	1.35	0.17
64	1023	8.04	2.25	0.28
256	1023	9.02	18.1	2.01
1024	1023	12.2	93.5	7.67

측정 조건. $f(x) = \sum \tanh(Wx)$, W 는 $256 \times n$. 역방향은 테이프 한 번, 전진은 방향 미분 n 번. 다섯 번 중앙값.

입력이 열여섯 개일 때는 전진 열여섯 번이 더 빠르다. 이 책의 테이프가 파이썬 객체라 항목 하나의 값이 크기 때문이다. 입력이 천 개가 되면 역방향이 일곱 배 빠르다. 역방향의 시간은 거의 늘지 않고 전진은 n 에 비례한다. 테이프 항목 수가 n 에 무관하게 1,023 인 것이 정리 5.4 의 m 이다.

관찰

입력이 많을수록 역방향이 이긴다. 학습은 입력이 매개변수 수억 개이니 전진은 선택지가 아니다. 역방향의 대가는 테이프의 메모리다.

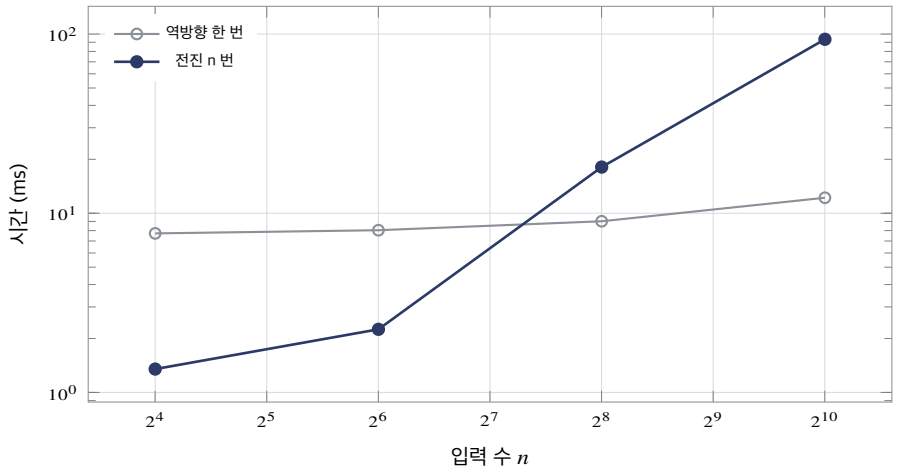


그림 5.4. 입력 수에 따른 두 방법의 시간. 역방향은 거의 평평하고 전진은 n 에 비례한다

다시 쓰기

이 절의 다시 쓰기는 테이프의 메모리를 세는 것이다. 정리는 m 에 비례한다고만 적었다. m 이 무엇인지 세면 2.3 의 KV 캐시와 같은 모양이 나온다.

명제 5.5: 테이프의 메모리

층 L, 시퀀스 길이 T, 은닉 차원 d 인 모델의 순전파는 층마다 토큰마다 중간 값을 남기므로 테이프의 메모리는 $LTds$ 에 상수를 곱한 것이다. 배치 B 를 곱하면 $BLTds$ 다. 가중치의 메모리 Ld^2s 와 견주면 BT/d 배다.

증명. 연산마다 국소 미분을 적어야 하고, 층 하나의 연산은 토큰 T 개 각각의 은닉 벡터 d 개에 대한 것이다. 층이 L 개, 배치가 B 개다. ■

BT 가 d 보다 크면 테이프가 가중치보다 크다. 배치 32 에 길이 2,048 이면 $BT = 65{,}536$ 이고 $d = 4{,}096$ 이면 열여섯 배다. 학습 메모리의 대부분이 가중치가 아니라 테이프인 이유다. 그래서 테이프를 줄이는 기법이 따로 있고, 그중 하나가 테이프의 일부를 지우고 뒤로 올 때 다시 계산하는 것이다. 시간과 메모리를 맞바꾸는 2.1 의 논증이 여기서도 나온다.

구현

테이프의 기록과 역방향 읽기다. 연산 셋을 원시 연산으로 두었다.

지금 보는 코드 변수 번호로 값과 항목을 잇는 테이프. mul 은 상대 입력을 국소 미분으로 적는다.

구현 기록과 곱셈 `samples/dsai/tape.py:34`

```
    def record(self, value: np.ndarray, inputs: tuple[int, ...],
               local: tuple[np.ndarray, ...]) -> int:
        out = self.var(value)
        self.entries.append(Entry(out, inputs, local))
        return out

    # 원시 연산 셋. 벡터에 원소별로 작용한다.
    def mul(self, a: int, b: int) -> int:
        va, vb = self.vals[a], self.vals[b]
        return self.record(va * vb, (a, b), (vb, va))
```

읽어 보면 `record` 가 정의 5.3 의 항목을 끝에 붙인다. `mul` 은 $\partial(ab)/\partial a = b$ 이므로 국소 미분으로 상대 입력을 적는다.

알고리즘 5.2 테이프 역방향 `dsai.tape.Tape.backward`

입력: 테이프 (V, E), 출력 변수 out
출력: 수반 배열 $\bar{V}$

$\bar{V} \leftarrow 0$; $\bar{V}[out] \leftarrow 1$
반복 $e = (o, (i_j), (d_j)) \in E$ 를 끝에서부터 **동안**
  반복 각 j **동안**
    $\bar{V}[i_j] \leftarrow \bar{V}[i_j] + \bar{V}[o] \cdot d_j$
  끝 반복
끝 반복
반환 $\bar{V}$

구현 끝에서부터 읽기 `samples/dsai/tape.py:58`

```
    def backward(self, out: int) -> list[np.ndarray]:
        """알고리즘 5.2. 테이프를 끝에서부터 읽으며 수반 변수를 쌓는다."""
        adj = [np.zeros_like(v) for v in self.vals]
        adj[out] = np.ones_like(self.vals[out])
        for e in reversed(self.entries):
            g = adj[e.out]
            for i, d in zip(e.inputs, e.local):
                contrib = g * d
                if contrib.shape != adj[i].shape:
                    contrib = contrib.reshape(adj[i].shape)
                adj[i] = adj[i] + contrib
        return adj
```

분석

테이프는 새 구조가 아니다. Wengert 가 1964 년에 함수를 원시 연산의 열로 풀어 미분을 자동으로 구하는 절차를 두 쪽에 적었고, 그 열이 테이프의 원형이다 [확립] [18]. Baydin 과 동료들의 2018 년 조사는 그 열을 앞으로 읽는 방식과 뒤로 읽는 방식의 비용을 이 절의 정리와 같은 형태로 정리했다 [확립] [19].

이 절의 판정은 스택의 판정과 같다. 그대로 유효하다. 바뀐 것은 스택의 원소가 프레임에서 국소 미분으로, 크기가 바이트에서 기가바이트로 간 것이다. 크기가 커지자 2.3 의 KV 캐시와 같은 문제가 생긴다. 길이를 미리 모르고, 추가 전용이고, 메모리의 대부분이다. 그래서 학습 프레임워크의 테이프 메모리 관리는 2 장의 할당자 문제와 같은 모양이다.

현장에서

> Baydin 과 동료들의 조사는 여러 학습 프레임워크가 테이프를 실행 중에 기록하는 방식과 미리 그래프로 만들어 두는 방식으로 갈린다고 정리했다 [확립] [19]. 앞의 것이 이 절의 테이프이고, 뒤의 것이 11 장의 계산 그래프다. 같은 항목을 실행 순서로 적느냐 의존 순서로 적느냐의 차이다.

되짚기

> 테이프를 추가 전용 배열의 역순 읽기로 정의했고, 역방향 한 번이 입력 수에 무관하게 편미분 전부를 준다는 것을 증명했다. 입력 천 개에서 일곱 배였다. 다음 절에서 앞에서 빼고 뒤에 넣는 구조, 큐를 고정 크기 배열 위에 세운다.

5.3 큐와 덱, 링 버퍼. 슬라이딩 윈도우

5.1 의 스택은 한쪽 끝만 썼다. 큐는 양쪽 끝을 쓴다. 뒤에 넣고 앞에서 뺀다. 배열로 큐를 만들면 앞에서 뺄 때 나머지를 전부 당겨야 한다. 0.1 의 이동량으로 원소 하나에 배열 전체다. 연결 리스트로 만들면 4.1 의 지연이 든다. 둘 다 안 맞는다.

답은 배열을 고리로 만드는 것이다. 머리와 꼬리 번호가 배열을 돌고, 끝에 닿으면 처음으로 되돌아온다. 옮기는 원소가 없다. 이 구조는 가장 오래된 것이 자동으로 밀려나는 창이 되고, 그 창이 15장의 슬라이딩 윈도우 어텐션과 5.4 의 배칭 큐다.

학습 목표

링 버퍼를 배열과 머리 번호와 개수로 정의하고, push 와 pop 이 옮기기 없이 상수임을 보일 수 있다. 창 통계를 링 버퍼와 누적값 하나로 상수 시간에 갱신하는 법을 알고리즘으로 적을 수 있다.

이 말부터

링 버퍼	고정 크기 배열을 고리로 쓰는 큐. 번호가 끝에 닿으면 처음으로 돌아온다
슬라이딩 윈도우	가장 최근 w 개만 보는 창. 새것이 들어오면 가장 오래된 것이 나간다
큐와 덱	큐는 뒤에 넣고 앞에서 빼는 열, 덱은 양 끝에서 넣고 빼는 열. 링 버퍼는 둘의 배열 구현이다

디딤돌

디딤돌은 큐와 덱이다. 큐는 뒤에 넣고 앞에서 빼며, 덱은 양 끝에서 넣고 뺀다. 연결 리스트로 두면 노드마다 라인 하나이고, 고정 크기 배열 하나에 두면 번호 둘과 나머지 연산으로 끝난다. 그것이 링 버퍼다.

정의 5.6: 링 버퍼

링 버퍼는 용량 c 의 값 배열 B, 머리 번호 h, 개수 n 의 묶음이다. 원소 k 번째는 $B[(h+k) \bmod c]$ 에 있다. push 는 $B[(h+n) \bmod c] \leftarrow v$ 뒤 $n < c$ 이면 $n \leftarrow n+1$, 아니면 $h \leftarrow (h+1) \bmod c$ 다. pop 은 $B[h]$ 를 돌려주고 $h \leftarrow (h+1) \bmod c$, $n \leftarrow n-1$ 이다.

불변식 $0 \le n \le c$ 이고, 논리 순서 $0..n-1$ 이 물리 자리 $h, h+1, \ldots$ 를 고리로 돌며 이어진다. 원소를 옮기는 연산은 없다.

정의의 마지막 문장이 이 구조가 배열 큐와 다른 점이다. 앞에서 빼도 아무것도 당기지 않는다. 머리 번호만 하나 는다. 가득 찼을 때 push 하면 가장 오래된 것이 덮인다. 그것이 창이다.

명제 5.7: 링 버퍼의 비용

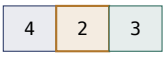
push 와 pop 은 각각 쓰기 또는 읽기 하나와 나머지 연산 하나이고, 이동량은 라인 하나다. 용량 c 와 개수 n 에 무관하다. 논리 순서로 전부 읽는 것은 물리적으로 최대 두 조각의 순차 접근이다.

증명. 정의 5.6 의 연산에 반복문이 없다. 논리 순서 $0..n-1$ 은 물리 자리 $h..c-1$ 과 $0..(h+n-c-1)$ 두 구간이고 각각 연속이다. ■

명제의 둘째 문장이 링 버퍼의 유일한 약점이다. 전체를 순서대로 읽으려면 두 조각을 잇는다. 그래도 조각이 둘뿐이라 1.3 의 간격 접근처럼 나빠지지는 않는다. 작은 예로 고리가 도는 것을 본다.

예제 5.3: 용량 셋의 고리

$c=3$ 에 1, 2, 3 을 넣으면 $B=(1,2,3)$, $h=0$, $n=3$ 이다. 4 를 넣으면 $B[0] \leftarrow 4$ 로 가장 오래된 1 이 덮이고 $h=1$ 이 된다. 논리 순서는 $B[1], B[2], B[0] = 2, 3, 4$ 다. pop 하면 2 를 주고 $h=2$, $n=2$ 다. 아무것도 옮기지 않았다.



칸 1 이 머리. 고리라 칸 2 다음이 칸 0 이다

그림 5.5. 예제 5.4. 4 를 넣은 뒤. 머리가 1 로 옮겨졌고 논리 순서는 2, 3, 4 다

균열

창 평균을 두 방법으로 쟀다. 갱신마다 창을 다시 더하는 것과, 링 버퍼에 누적합 하나를 붙여 들어온 값을 더하고 나간 값을 빼는 것이다.

실험 5.3 창 평균. 다시 더하기와 링 버퍼 누적합

창 길이	다시 더하기 ms	링 버퍼 ms	비율
16	87.9	20.2	4.34
256	91.4	20.2	4.52
4096	118	20.1	5.88

측정 조건. 값 2만 개를 흘려 넣으며 창 평균을 매번 갱신. 다시 더하기는 갱신마다 창 전체를 읽는다. 다섯 번 중앙값.

링 버퍼는 창 길이와 무관하게 갱신당 같은 시간이다. 다시 더하기는 창이 길수록 늘어난다. 창 16 에서도 네 배 차이가 나는 것은 다시 더하기가 갱신마다 자르기와 합을 새로 만들기 때문이다. 이 책의 파이썬 구현이 치르는 상수다.

관찰

> 창 통계의 값은 창 길이가 아니라 갱신마다 들어오고 나가는 원소의 수가 정한다. 하나 들어오고 하나 나가면 상수다.

다시 쓰기

누적값을 정의에 넣는다. 그러면 창 통계가 상수 시간이 된다.

알고리즘 5.3 창 평균 갱신 `dsai.ring.WindowMean.push`

> **입력:** 링 버퍼 R (용량 w), 누적합 S, 새 값 v
> 1: **만약** R 이 가득 참 **이면**
> 2: $S \leftarrow S - R[\text{머리}]$ ▷ 나가는 값을 뺀다
> 3: **끝 만약**
> 4: $R.\text{push}(v)$; $S \leftarrow S + v$
> 5: **반환** S/n

이 알고리즘은 합에 대해서만 되는 것이 아니다. 뺄 수 있는 통계, 즉 합, 개수, 제곱합처럼 나가는 값을 되돌릴 수 있는 것이면 전부 된다. 최대값은 안 된다. 나간 값이 최대값이었으면 남은 것 중 최대를 다시 찾아야 한다. 그것은 10장의 힙이 하는 일이다.

15장의 슬라이딩 윈도우 어텐션이 정확히 이 구조다. 토큰마다 최근 w 개의 키와 값만 보고, 캐시를 링 버퍼로 두어 오래된 것을 덮는다. 2.3 의 KV 캐시가 길이에 비례해 자라던 것이 용량 w 에서 멈추고, 2.4 의 블록 테이블이 필요 없어진다. 그 대가는 창 밖을 못 보는 것이다.

구현

링 버퍼와 창 평균이다.

지금 보는 코드 머리와 개수만 움직이는 큐. 가득 차면 머리의 것을 덮는다.

구현 넣기와 빼기 `samples/dsai/ring.py:20`

```
20      def push(self, v) -> None:
21          """꼬리에 넣는다. 차 있으면 머리의 것을 덮는다."""
22          tail = (self.head + self.n) % self.cap
23          self.buf[tail] = v
24          if self.n == self.cap:
25              self.head = (self.head + 1) % self.cap
26          else:
27              self.n += 1
28
29      def pop(self):
30          """머리에서 뺀다. 옮기는 것 없이 번호 하나만 움직인다."""
31          v = self.buf[self.head]
32          self.head = (self.head + 1) % self.cap
33          self.n -= 1
34          return v
```

읽어 보면 `push` 는 꼬리 자리를 나머지 연산으로 구해 쓰고, 가득 찼으면 머리를 하나 민다. `pop` 은 머리를 읽고 하나 민다. 어느 쪽도 원소를 옮기지 않는다.

구현 창 평균 `samples/dsai/ring.py:51`

```
    def push(self, v: float) -> float:
        if self.ring.n == self.ring.cap:
            self.total -= self.ring.buf[self.ring.head]
        self.ring.push(v)
        self.total += v
        return self.total / self.ring.n
```

분석

링 버퍼는 그대로 유효하다. 정의도 구현도 바뀌지 않았다. 바뀐 것은 창의 크기와 원소의 크기다. 운영체제의 링 버퍼는 바이트 수천 개였고, 슬라이딩 윈도우 어텐션의 링 버퍼는 토큰마다 800KB 인 원소 수천 개다. 원소가 커지자 덮어쓰기의 값이 시간이 되고, 창 밖을 못 보는 것이 정확도의 대가가 된다. 그 대가를 재는 것은 15장이다.

이 구조가 5.4 로 바로 이어진다. 요청이 들어오고 나가는 큐이고, 걸음마다 하나가 들어오고 하나가 나가면 상수다. 그 큐에 배치 크기를 상한으로 두고 걸음마다 갈아 끼우는 것이 연속 배칭이다.

주의

> `view` 는 두 조각을 이어 붙이므로 복사가 든다. 링 버퍼에서 이동량이 드는 유일한 연산이다. 논리 순서 전체가 필요한 자리에서만 부른다.

15장에서 이 구조가 어텐션의 창이 될 때는 `view` 를 부르지 않는다. 두 조각을 그대로 두고 인덱스 배열로 읽는다. 조각을 잇는 복사가 토큰마다 들면 그것이 곧 시간이기 때문이다.

되짚기

> 링 버퍼를 배열과 머리 번호와 개수로 정의했고, push 와 pop 이 옮기기 없이 상수임을 보였다. 창 평균이 창 길이와 무관하게 상수였다. 다음 절에서 이 큐에 요청을 넣고 걸음마다 갈아 끼우는 배칭 큐를 세운다.

5.4 배칭 큐와 연속 배칭

1.4 이 배치가 밀도를 올린다고 세웠고, 2.3 이 디코딩은 배치로도 밀도가 안 오른다고 세웠다. 둘을 합치면 추론 서버의 문제가 된다. 요청을 모아야 가중치 읽기를 나눠 쓰는데, 모으면 기다려야 하고 요청마다 길이가 다르다. 이 절은 그 문제를 큐 하나로 세운다. 요청이 들어오는 큐와, 그 큐에서 배치를 꺼내는 규칙이다.

규칙은 둘이다. 배치를 채워서 시작하고 그 배치가 다 끝날 때까지 같이 가는 것, 그리고 걸음마다 끝난 것을 빼고 기다리는 것을 넣는 것이다. 둘째가 연속 배칭이고, 5.3 의 링 버퍼처럼 걸음마다 하나 나가고 하나 들어오는 큐다.

학습 목표

> 정적 배칭과 연속 배칭을 큐 연산으로 정의하고, 정적 배칭의 낭비를 배치 안 길이의 차이로 셀 수 있다. 같은 요청 열에서 두 방식의 처리량과 지연이 왜 갈리는지 실측으로 설명할 수 있다.

이 말부터

정적 배칭	배치를 채워 시작하고, 그 배치의 모든 요청이 끝날 때까지 같이 가는 방식
연속 배칭	토큰 한 걸음마다 끝난 요청을 빼고 기다리는 요청을 넣는 방식. 반복 단위 스케줄링이라고도 한다
걸음	배치 안의 모든 활성 요청이 토큰 하나씩 내는 한 번의 실행. 가중치를 한 번 읽는다

디딤돌

디딤돌은 큐다. 요청이 도착 순서로 들어오고 앞에서부터 나간다. 5.3 의 링 버퍼가 그 큐다. 정적 배칭은 그 큐에서 B 개를 한 번에 꺼낸다.

정의 5.8: 정적 배칭

> 대기 큐 Q 와 배치 상한 B 가 있다. 실행기가 비면 Q 의 앞에서 최대 B 개를 꺼내 배치를 만들고, 배치의 가장 긴 요청이 끝날 때까지 걸음을 반복한다. 걸음 하나의 시간은 $a + b \cdot |\text{활성 요청}|$ 이다. a 는 가중치를 읽는 값, b 는 토큰 하나의 값이다.
>
> **불변식** 배치는 시작할 때의 구성 그대로 끝난다. 끝난 요청의 자리는 배치가 끝날 때까지 비어 있다.

불변식이 낭비의 원인이다. 길이 4 인 요청과 512 인 요청이 같은 배치에 들면 앞의 것은 4 걸음 만에 끝나고 508 걸음 동안 자리를 비워 둔다. 그 빈자리는 걸음의 고정 값 a 를 나눠 쓰지 못하는 자리다.

명제 5.9: 정적 배칭의 낭비

> 길이 $\ell_1, \dots, \ell_B$ 인 요청의 정적 배치는 걸음 $\max_i \ell_i$ 번을 돌고, 자리 걸음의 총수 $B \max_i \ell_i$ 중 쓰인 것은 $\sum_i \ell_i$ 다. 낭비 비율은 $1 - \sum_i \ell_i / (B \max_i \ell_i)$ 이고, 길이 분포의 꼬리가 길수록 1 에 다가간다.

증명. 정의 5.8 의 불변식으로 요청 i 는 ℓ_i 걸음 뒤에 끝나지만 자리는 $\max_j \ell_j$ 걸음 동안 배치에 남는다. ■

예제 5.4: 길이 넷의 배치

> 길이 $(4, 30, 60, 512)$ 인 요청 넷이면 걸음 512 번에 자리 걸음 2,048 개 중 606 개가 쓰인다. 낭비가 70 퍼센트다. 길이가 $(60, 60, 60, 60)$ 이면 낭비가 0 이다. 낭비는 배치 크기가 아니라 길이의 편차에서 온다.

균열

포아송 도착, 로그정규 길이의 요청 천 개를 두 방식에 넣고 처리량과 지연을 쟀다. 걸음 시간은 고정 20 밀리초에 활성 요청당 0.5 밀리초로 두었다.

실험 5.4 정적 배칭과 연속 배칭. 처리량과 지연

배치 상한	방식	토큰/초	평균 지연 ms	p95 지연 ms
8	정적	142	128264	245582
8	연속	293	2847	6907
32	정적	240	31923	56381
32	연속	295	1675	4064
128	정적	280	12485	25612
128	연속	295	1675	4064

측정 조건. 요청 1,000 개, 초당 4 개 도착, 길이는 4 에서 512 토큰의 로그정규 분포. 걸음 시간 $20 + 0.5\times$ 배치 밀리초. 지연은 도착에서 완료까지.

배치 상한 32 에서 연속 배칭은 평균 지연이 정적의 19분의 1 이다. 처리량도 더 높다. 정적 배칭은 상한 8 에서 도착을 못 따라가 지연이 쌓이고, 상한을 128 로 올려도 12 초가 남는다. 긴 요청 하나가 배치 전체를 붙잡기 때문이다. 연속 배칭은 상한 32 와 128 이 같다. 동시에 살아 있는 요청이 32 개를 넘지 않아 상한이 안 걸린다.

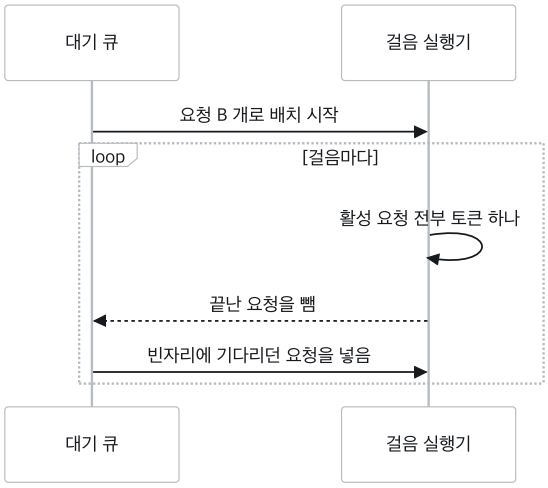


그림 5.6.. 연속 배칭. 걸음마다 끝난 요청을 빼고 기다리던 요청을 넣는다

관찰

> 정적 배칭의 지연은 배치 안에서 가장 긴 요청이 정한다. 연속 배칭의 지연은 자기 길이가 정한다. 길이 분포의 꼬리가 둘의 차이다.

다시 쓰기

연속 배칭을 정의한다. 큐에서 꺼내는 단위가 배치가 아니라 걸음이다.

정의 5.10: 연속 배칭

> 대기 큐 Q, 활성 집합 A, 상한 B 가 있다. 걸음마다 A 의 모든 요청이 토큰 하나를 내고, 끝난 요청을 A 에서 빼고, $|A| < B$ 인 동안 Q 의 앞에서 요청을 꺼내 A 에 넣는다.
>
> **불변식** $|A| \le B$ 이고, Q 가 비지 않으면 걸음 시작 시점에 $|A| = B$ 다. 끝난 요청의 자리는 다음 걸음에 채워진다.

정의의 불변식이 정적 배칭의 불변식을 뒤집은 것이다. 거기서는 자리가 배치가 끝날 때까지 비어 있었고, 여기서는 다음 걸음에 채워진다. 그 한 줄을 알고리즘으로 적으면 아래와 같다.

알고리즘 5.4 연속 배칭 `dsai.batching.simulate_continuous`

입력 : 대기 큐 Q, 상한 B, 걸음 시간 $a + b|A|$
$A \leftarrow \emptyset$
조건 Q 또는 A 가 비지 않음 **동안**
  조건 $|A| < B$ **이고** Q 의 앞이 도착함 **동안**
    $A \leftarrow A \cup \{Q.\mathrm{pop}()\}$
  끝 조건
  걸음 하나. 시계 ← 시계 $+a + b|A|$
  A 의 각 요청에 토큰 하나. 끝난 것은 완료 시각을 적고 A 에서 뺀다
끝 조건

알고리즘의 안쪽 반복문이 큐에서 꺼내는 자리이고, 마지막 줄이 빼는 자리다. 둘이 걸음마다 한 번씩 돈다. 그러면 자리 걸음의 낭비가 어디서 생기는지가 정확히 정해진다.

명제 5.11: 연속 배칭의 낭비

연속 배칭에서 자리 걸음의 낭비는 Q 가 빌 때만 생긴다. Q 가 비지 않는 동안 모든 걸음이 B 개의 활성 요청으로 돌고, 요청 i 는 정확히 ℓ_i 걸음 동안만 자리를 쓴다. 따라서 부하가 충분하면 낭비가 0 이고, 요청의 지연은 대기 시간 더하기 ℓ_i 걸음이다.

증명. 정의 5.10 의 불변식으로 빈자리가 다음 걸음에 채워지므로, Q 에 요청이 있는 한 자리가 빈 채로 도는 걸음이 없다. ■

명제 5.9 의 낭비가 명제 5.11 에서 사라진다. 그 대가는 걸음마다 A 의 구성이 바뀐다는 것이고, 그래서 2.4 의 블록 테이블이 필요하다. 요청마다 KV 캐시가 다른 시점에 시작하고 끝나므로 연속 메모리로는 2.2 의 단편화가 바로 생긴다. 2장의 구조가 5장의 구조를 떠받친다.

구현

두 방식의 시뮬레이션이다. 둘 다 큐 하나와 시계 하나다.

지금 보는 코드 걸음마다 끝난 요청을 빼고 기다리는 요청을 넣는 큐. 시계는 걸음 시간만큼 간다.

구현 걸음 시간 `samples/dsai/batching.py:19`

```
19 def step_time(batch_size: int, fixed_ms: float, per_token_ms: float) -> float:
20     """토큰 한 걸음의 시간. 가중치를 읽는 고정 값과 토큰마다의 값."""
21     return fixed_ms + per_token_ms * batch_size
```

구현 연속 배칭 `samples/dsai/batching.py:44`

```python
def simulate_continuous(reqs: list[Request], max_batch: int, fixed_ms: float,
                        per_token_ms: float) -> None:
    """알고리즘 5.3 의 뒤 절반. 걸음마다 끝난 요청을 빼고 기다리는 요청을 넣는다."""
    queue = deque(sorted(reqs, key=lambda r: r.arrival))
    active: list[tuple[Request, int]] = []
    clock = 0.0
    while queue or active:
        if not active and queue:
            clock = max(clock, queue[0].arrival)
        while queue and len(active) < max_batch and queue[0].arrival <= clock:
            active.append((queue.popleft(), 0))
        clock += step_time(len(active), fixed_ms, per_token_ms)
        still = []
        for r, done in active:
            done += 1
            if done == r.tokens:
                r.done_at = clock
            else:
                still.append((r, done))
        active = still
```

읽어 보면 안쪽 `while` 이 정의 5.10 의 채우기이고, `still` 을 다시 모으는 것이 끝난 요청 빼기다. `step_time` 이 걸음 시간 $a + b|A|$ 이고, 그 a 가 2.3 의 가중치 읽기다.

분석

큐는 그대로 유효하다. 바뀐 것은 큐에서 꺼내는 단위다. 요청 단위에서 걸음 단위로 내려갔고, 그 한 단계가 지연을 19분의 1 로 만들었다. 꺼내는 단위를 바꿀 수 있었던 이유는 생성이 걸음마다 같은 일을 하기 때문이다. 토큰 하나를 내는 것은 요청이 어디까지 왔든 같은 연산이다.

현장에서

> Yu 와 동료들의 2022 년 Orca 는 배치의 단위를 요청에서 반복으로 내리는 것을 제안하고, 그것을 반복 단위 스케줄링이라 불렀다 [확립] [8]. Kwon 과 동료들의 2023 년 vLLM 은 그 스케줄링을 2.4 의 블록 테이블과 합쳐 처리량을 두 배에서 네 배로 올렸다고 보고했다 [확립] [9].

이 절이 2 부를 닫는다. 2 부가 세운 것은 순서가 정해진 배열 셋이다. 스택, 테이프, 링 버퍼. 셋 다 정의는 40 년 전 그대로이고, 원소가 커지고 길이를 모르게 되면서 2 장의 메모리 문제와 만났다. 연속 배칭이 그 만남의 자리다. 큐의 단위를 내리자 메모리가 조각나고, 블록 테이블이 그것을 받는다.

주의

> 실험 5.4 의 걸음 시간은 모델이다. 실제 걸음 시간은 2.3 의 명제대로 활성 요청의 캐시 길이에도 달렸고, 이 시뮬레이션은 그것을 상수로 뭉뚱그렸다.

그 뭉뚱그림을 푸는 것이 15장이다. 걸음 시간에 캐시 길이가 들어오면 긴 요청이 짧은 요청의 걸음을 느리게 하고, 그때 큐의 순서가 다시 문제가 된다. 10장의 우선순위 큐가 그 순서를 든다.

되짚기

> 정적 배칭의 낭비를 길이의 편차로 셌고, 연속 배칭이 그 낭비를 없애 지연을 19분의 1로 만드는 것을 봤다. 큐의 단위가 요청에서 걸음으로 내려간 것이다. 다음 장에서 순서가 아니라 키로 찾는 구조, 해시 테이블이 임베딩 테이블이 되는 길을 본다.

5장을 닫으며

이 장이 세운 것은 새 구조가 아니다. 스택과 큐는 그대로다. 세운 것은 그 둘이 어떤 이름으로 살아 있는지다. 스택은 테이프라는 이름으로 학습 메모리의 대부분을 차지하고, 큐는 배칭 큐라는 이름으로 추론 서버의 지연을 정한다. 둘 다 원소가 커지고 길이를 모르게 되면서 2장과 만난다. 테이프는 KV 캐시와 같은 추가 전용 배열이고, 연속 배칭은 블록 테이블 없이는 돌지 않는다.

네 실험의 숫자를 적어 둔다. 재귀는 깊이 천에서 멈췄고 명시적 스택은 백만까지 갔다. 입력 천 개의 기울기를 역방향 한 번이 전진 천 번보다 일곱 배 빨리 구했고, 입력 열여섯 개에서는 반대였다. 창 평균은 링 버퍼로 창 길이와 무관하게 상수였다. 연속 배칭은 정적 배칭보다 평균 지연이 19분의 1 이었고, 그 차이는 전부 길이 분포의 꼬리에서 왔다.

정정 하나를 남긴다. 배칭 실험을 처음에는 초당 요청 400 개로 돌렸다. 두 방식 다 지연이 수백 초로 쌓였고 차이가 안 보였다. 처리 능력이 초당 토큰 천 개 남짓인 모델에 초당 토큰 3만 개를 넣은 것이었다. 2.2 의 단편화 실험과 같은 실수를 두 번 했다. 부하를 처리 능력의 3분의 1 로 내리자 두 방식이 갈렸다. 큐의 실험은 부하를 먼저 정해야 한다.

옛 구조의 판정

> 스택과 큐는 그대로 유효하다. 정의와 구현이 바뀌지 않았다. 근거는 실험 5.1 과 5.3 이다. 배열 위의 스택과 링 버퍼가 지금도 가장 싼 구현이다. 바뀐 것은 원소와 단위다. 원소가 국소 미분과 요청이 되었고, 큐의 단위가 걸음으로 내려갔다.

이 장의 요약

> 1. 재귀는 언어의 스택이고 상한이 있다. 명시적 스택은 원소가 정수 둘이라 상한이 메모리다 5.1
> 2. 테이프는 앞으로 적고 뒤로 읽는 추가 전용 배열이다. 역방향 한 번이 입력 수와 무관하게 편미분 전부를 준다 5.2
> 3. 테이프의 메모리는 배치 곱하기 길이 곱하기 층 곱하기 차원이고, 가중치보다 BT/d 배 크다 5.2
> 4. 링 버퍼는 옮기기 없는 큐이고, 뺄 수 있는 통계는 창 길이와 무관하게 상수 시간에 갱신된다 5.3
> 5. 정적 배칭의 낭비는 배치 안 길이의 편차에서 오고, 연속 배칭은 걸음마다 갈아 끼워 그 낭비를 없앤다 5.4
>
> 다음 장에서 뒤집히는 것: 지금까지의 구조는 순서로 찾았다. 3부는 키로 찾는다. 해시 테이블이 임베딩 테이블이 되고, 같음이 비슷함이 된다.

필자의 판단

> 필자는 자동 미분을 자료 구조 책에서 가르치는 것이 맞다고 본다. 대부분의 교재는 그것을 미적분이나 딥러닝 책에 둔다. 그런데 테이프의 어려움은 수학이 아니라 메모리다. 입력이 매개변수 수억 개일 때 역방향이 유일한 선택이고, 그 대가가 테이프의 크기이며, 그 크기를 다루는 것이 2장의 할당자다. 미분은 두 줄이고 메모리는 장 하나다.

열린 문제

하나, 실험 5.2 의 테이프는 파이썬 객체라 항목 하나의 값이 크다. 입력이 적을 때 전진이 이긴 것은 그 상수 때문이고, 상수가 작은 구현에서 교차점이 어디인지는 재지 않았다.
둘, 명제 5.5 의 테이프 메모리를 줄이는 재계산 기법의 시간과 메모리 맞바꿈을 이 장은 한 문장으로만 적었다. 그 곡선을 재지 않았다.
셋, 실험 5.4 의 걸음 시간은 활성 요청 수에만 달렸다. 캐시 길이에 따른 걸음 시간의 변화를 넣으면 긴 요청이 짧은 요청의 걸음을 느리게 하는 효과가 나오고, 그것을 세지 않았다.

연습문제

5.1 기초 프레임 400 바이트, 상한 1,000 인 언어에서 재귀가 쓰는 스택 메모리의 최대와, 같은 깊이를 원소 16 바이트의 명시적 스택으로 갈 때의 메모리를 구하라. 명시적 스택으로 깊이 천만을 가면 얼마인가.

5.2 기초 $a = 3$ 에서 $b = a \cdot a$, $c = b \cdot a$ 를 계산하는 테이프의 항목 둘을 적고, 역방향으로 $\bar{a}$ 를 구하라. $c = a^3$ 의 미분과 같은지 확인하라.

5.3 기초 용량 4 의 링 버퍼에 1 부터 6 까지 넣은 뒤 pop 둘을 했을 때 배열, 머리, 개수와 논리 순서를 적어라.

5.4 기초 길이 $(8, 8, 64, 256)$ 인 요청 넷의 정적 배치에서 명제 5.6 의 낭비 비율을 구하고, 같은 넷을 연속 배칭이 부하가 충분할 때 어떻게 처리하는지 걸음 수로 적어라.

5.5 응용 `dsai.stack.depth_iterative` 를 일반 트리 (자식 여럿) 에 대해 재귀 순서와 같은 방문 순서를 내도록 고쳐라. 자식을 넣는 순서를 어떻게 해야 하는가.

5.6 응용 `dsai.tape.Tape` 에 행렬 벡터 곱을 원시 연산으로 더하라. 국소 미분을 어떻게 적어야 하는가. 실험 5.2 를 그 연산으로 다시 재고 교차점이 어디로 움직이는지 보라.

5.7 응용 `dsai.ring.WindowMean` 처럼 창 분산을 상수 시간에 갱신하는 클래스를 써라. 제곱합 하나가 더 필요하다. 창 최대값은 왜 같은 방법으로 안 되는지 적어라.

5.8 응용 `dsai.batching.simulate_continuous` 에 대기 큐를 도착 순서가 아니라 짧은 요청 먼저로 바꾸는 선택지를 더하고, 실험 5.4 에서 평균 지연과 p95 가 어떻게 바뀌는지 재라.

5.9 탐구 테이프의 일부를 지우고 뒤로 올 때 다시 계산하면 메모리가 준다. 층 L 개 중 $\sqrt{L}$ 개만 남기는 방법의 메모리와 추가 시간을 세우고, 명제 5.5 와 견줘라.

5.10 탐구 연속 배칭에서 긴 요청이 계속 자리를 차지하면 짧은 요청의 대기가 는다. 자리를 잠시 내주는 선점을 넣을 때 KV 캐시를 어디에 두어야 하는지, 2장의 구조로 논하라.

더 읽을 것

[18] 자동 미분의 원전. 두 쪽이다. 함수를 원시 연산의 열로 푸는 생각이 여기서 시작됐다. 5.2
[19] 자동 미분 조사. 전진과 역방향의 비용, 테이프와 그래프의 차이, 프레임워크들의 분류가 있다. 정리 5.2 의 원전이다. 5.2
[8] 반복 단위 스케줄링의 원전. 5.4 절의 연속 배칭이 이 논문의 제안이고, 정적 배칭의 낭비 그림이 3장에 있다. 5.4
[9] 연속 배칭과 블록 테이블을 합친 서버. 5.4 절이 2장을 왜 필요로 하는지가 이 논문의 주장이다. 5.4

3부

해싱

키를 자리로 바꾸는 함수 하나가 탐색을 상수로 만든다. 3부는 그 함수가 어휘 사전과 임베딩 테이블이 되는 길, 그리고 정확히 같음을 묻던 해시가 비슷함과 개수를 묻게 되는 길을 본다.

6장 해시 테이블에서 임베딩 테이블로 7장 확률적 자료 구조

6장

해시 테이블에서 임베딩 테이블로

2009년의 한 논문은 이메일 필터 수십만 개의 특징을 사전 없이 해시로 자리 2^{22} 개의 표 하나에 담고, 충돌을 처리하지 않아도 정확도가 같다고 보고했다 [확립] [20]. 교과서에서 충돌을 처리하지 않는 해시 테이블은 틀린 구조다. 그 틀린 구조가 지금 대부분의 추천 시스템과 여러 언어 모델의 첫 층이다. 틀린 것이 맞게 된 이유가 이 장의 셋째 절이고, 그 앞의 두 절은 그 이유를 세우기 위한 디딤돌이다.

이 장은 0.2 의 첫째 가정과 넷째 가정을 건드린다. 해시 테이블의 세 충돌 처리는 탐사 수로는 교과서의 순위를 따르고 라인으로는 뒤집힌다. 그리고 값이 학습되는 표에서는 충돌이 오류가 아니라 잡음이 된다. 구조의 일부가 데이터에서 배워지는 첫 자리다. 마지막 절은 키를 하나씩이 아니라 한꺼번에 넣을 때 해시 테이블이 정렬에 지는 자리를 본다.

이 장에서 세우는 것

6.1 세 충돌 처리의 기대 탐사 수를 정리로 세우고 실측이 소수점까지 따르는 것을 본다. 라인으로 세면 순위가 뒤집힌다

6.2 어휘 사전. 문자열 키의 값은 자리가 아니라 문자열 읽기에 든다

6.3 임베딩 테이블과 해싱 트릭. 충돌 비율이 명제와 맞고, 오차는 학습이 줄인다

6.4 한꺼번에 넣기의 자리 충돌과, 표 없이 정렬로 세는 길

6.1 해시 함수와 충돌 해결. 체이닝, 개방 주소, 쿠쿠

1.1 의 배열은 번호로 찾는다. 번호가 없는 것, 문자열이나 큰 정수를 찾으려면 번호를 만들어야 한다. 그 번호를 만드는 함수가 해시 함수이고, 번호가 겹쳤을 때 어떻게 하느냐가 해시 테이블의 전부다. 교과서는 겹침을 처리하는 법을 셋 가르친다. 이 절은 그 셋을 정의하고, 적재율에 따라 몇 번 봐야 하는지를 정리로 세운 뒤 잰다.

세 방법의 차이는 이 책의 잣대로 다시 읽으면 하나로 모인다. 겹쳤을 때 몇 라인을 더 옮기는가, 그리고 그 옮김이 4.1 의 의존 읽기인가. 체이닝은 의존 읽기이고, 개방 주소는 순차 읽기이고, 쿠쿠는 독립 읽기 둘이다.

학습 목표

해시 테이블을 해시 함수와 충돌 처리로 정의하고, 세 처리의 기대 탐사 수를 적재율의 함수로 적을 수 있다. 같은 적재율에서 세 처리의 이동량이 왜 다른지 라인과 의존 읽기로 설명할 수 있다.

이 말부터

적재율	넣은 키 수를 자리 수로 나눈 것. α 로 적고, 해시 테이블의 값은 전부 이 숫자에 달렸다
탐사	키를 찾을 때 들여다본 자리의 수. 한 번이면 해시 자리에 바로 있었다는 뜻이다
보편 해시	무작위로 고른 계수 하나로 정해지는 해시 함수의 집합. 어느 키 쌍이든 겹칠 확률이 낮다

디딤돌

정의 6.1: 해시 테이블

자리 m 개의 배열 T 와 키를 $\{0, \dots, m-1\}$ 로 보내는 함수 h 의 짝이다. 키 k 는 자리 $h(k)$ 에 두려 하고, 이미 다른 키가 있으면 충돌 처리 규칙을 따른다. 체이닝은 자리마다 목록을 두어 붙인다. 개방 주소는 다음 자리 $h(k)+1, h(k)+2, \dots$ 를 차례로 본다. 쿠쿠는 해시 둘 h_1, h_2 를 두고 $h_1(k)$ 가 차 있으면 거기 있던 키를 h_2 자리로 쫓아낸다.

불변식 키는 자기 규칙이 정한 자리들 중 하나에 있다. 체이닝은 $h(k)$ 의 목록, 개방 주소는 $h(k)$ 부터 빈자리 전까지, 쿠쿠는 $h_1(k)$ 또는 $h_2(k)$ 다.

세 규칙은 충돌한 키를 어디에 두느냐만 다르다. 체이닝은 표 밖에, 개방 주소는 표 안의 옆에, 쿠쿠는 표 안의 다른 자리에 둔다. 그 자리가 어디냐가 뒤에서 라인 수를 정한다. 그 전에 자리를 정하는 함수부터 세운다. 해시 함수는 무엇이든 되는 것이 아니다. 키가 어떻게 오든 자리가 고르게 퍼져야 한다. 그 보장을 주는 것이 보편 해시다.

정의 6.2: 보편 해시

> 해시 함수의 집합 H 가 보편이라는 것은, H 에서 무작위로 고른 h 에 대해 서로 다른 어느 두 키 $x \neq y$ 도 $\Pr[h(x) = h(y)] \le c/m$ 인 것이다. c 는 작은 상수다. 이 책은 홀수 64 비트 정수 a 를 무작위로 골라 $h_a(k) = \lfloor (ak \bmod 2^{64}) / 2^{64-\ell} \rfloor$ 로 쓴다. $m = 2^{\ell}$ 이고 $c = 2$ 다.

Carter 와 Wegman 이 1979 년에 이 개념을 세우고 그런 집합이 있음을 보였다 [확립] [21]. 요점은 키의 분포를 모르고도 자리의 분포를 보장한다는 것이다. 계수를 무작위로 골랐으므로 어떤 키 열도 일부러 겹치게 만들 수 없다.

정리 6.3: 성공 탐색의 기대 탐사 수

> 보편 해시로 키 $n = \alpha m$ 개를 넣은 뒤, 들어 있는 키 하나를 찾는 기대 탐사 수는 체이닝에서 $1 + \alpha/2$, 선형 탐사에서 $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ 이다. 쿠쿠는 언제나 2 이하다.

증명. 체이닝에서 키를 넣을 때 그 자리에 이미 있던 키의 기대 수는 그때까지의 적재율이고, 찾을 때는 그 뒤에 온 키를 지나지 않는다. 평균하면 $\alpha/2$ 에 자기 자신 하나다. 선형 탐사의 식은 Knuth 가 1962 년에 유도했고, 채워진 자리가 뭉치는 효과로 α 가 1 에 가까울수록 급히 커진다. 유도는 부록에 미루고 실험으로 확인한다. 쿠쿠는 불변식에 의해 키가 두 자리 중 하나에 있으므로 두 번이면 끝난다. ■

정리의 세 식이 적재율에 어떻게 반응하는지가 세 처리의 성격이다. 체이닝은 완만하게, 선형 탐사는 1 에 가까워질수록 급하게, 쿠쿠는 아예 반응하지 않는다. 숫자를 넣어 보면 차이가 보인다.

예제 6.1: 적재율 0.9

> 자리 65,536 개에 키 58,982 개를 넣으면 $\alpha = 0.9$ 다. 체이닝은 평균 1.45 번, 선형 탐사는 $\frac{1}{2}(1 + 10) = 5.5$ 번 본다. 쿠쿠는 2 번 이하다. 같은 키, 같은 메모리인데 다섯 배 차이다.

균열

세 방식을 자리 65,536 개의 표에 적재율을 바꿔 가며 넣고, 들어 있는 키 5,000 개를 찾을 때의 평균 탐사 수를 쟀다.

실험 6.1 적재율에 따른 탐사 수. 실측과 정리의 예측

적재율	체이닝	체이닝 예측	선형 탐사	선형 예측	쿠쿠
0.25	1.13	1.12	1.17	1.17	1.12
0.50	1.24	1.25	1.48	1.50	1.22
0.75	1.36	1.38	2.45	2.50	1.28
0.90	1.44	1.45	5.25	5.50	1.34
0.95	1.48	1.48	10.9	10.5	1.35

측정 조건. 자리 65,536 개, 무작위 32 비트 정수 키, 성공 탐색 5,000 개의 평균. 쿠쿠는 표 둘이라 자리가 131,072 개다.

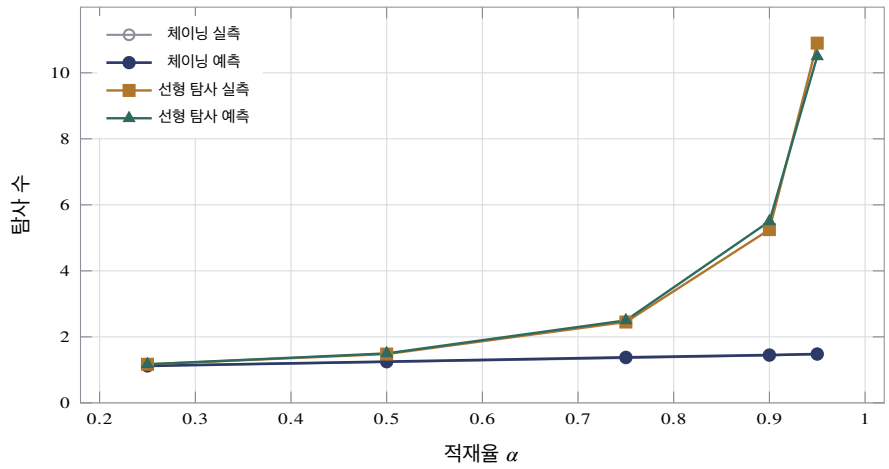


그림 6.1.. 적재율에 따른 탐사 수. 실선 예측과 점 실측이 겹치고, 선형 탐사가 0.9 근처에서 급히 오른다

실측이 정리 6.3 의 예측을 소수점 둘째 자리까지 따른다. 보편 해시가 약속한 분포가 그대로 나온 것이다. 선형 탐사는 적재율 0.95 에서 열한 번을 본다. 체이닝은 1.5 번이다.

그런데 이 표는 탐사 수를 센 것이지 시간을 잰 것이 아니다. 탐사 하나의 값이 셋에서 다르다. 체이닝의 탐사는 목록을 따라가는 것이라 4.1 의 의존 읽기이고 라인이 하나씩 든다. 선형 탐사의 다음 자리는 옆 자리라 같은 라인 안일 때가 많다. 쿠쿠의 두 자리는 서로 독립이라 한꺼번에 읽을 수 있다.

관찰

탐사 수가 적은 것과 라인이 적은 것은 다르다. 체이닝은 탐사 1.5 에 라인 1.5 이고, 선형 탐사는 탐사 11 에 라인 하나에서 둘이다.

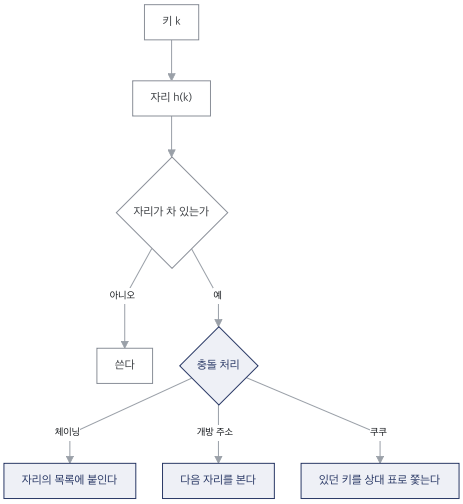


그림 6.2.. 충돌이 났을 때 세 규칙. 목록에 붙이거나, 옆으로 밀거나, 쫓아낸다

다시 쓰기

이 절의 다시 쓰기는 정리 6.3 를 이동량으로 다시 적는 것이다. 탐사가 아니라 라인을 세면 순위가 바뀐다.

명제 6.4: 세 처리의 이동량

> 키 크기 s, 라인 B 에서 성공 탐색 하나의 기대 이동량은 체이닝이 $1 + \alpha/2$ 라인이고 전부 의존 읽기다. 선형 탐사는 탐사 수 p 에 대해 $\lceil ps/B \rceil$ 라인이고 순차 읽기다. $\alpha = 0.9$, $s = 8$ 이면 $p = 5.5$ 에서 라인 하나다. 쿠쿠는 라인 둘이고 독립 읽기라 지연이 겹친다.

증명. 체이닝의 목록 노드는 힙에 따로 있으므로 노드마다 라인 하나이고, 다음 노드의 주소는 지금 노드를 읽어야 안다. 선형 탐사의 자리 $h(k), h(k)+1, \ldots$ 는 배열에서 연속이므로 1.3 의 순차 접근이다. 쿠쿠의 두 자리는 주소가 미리 정해져 있어 4.1 의 명제 둘째 조건이다. ■

이 명제가 현대 해시 테이블이 거의 전부 개방 주소인 이유다. 적재율을 0.7 아래로 두면 탐사가 둘 남짓이고, 둘이 같은 라인에 든다. 체이닝은 탐사가 적어 보이지만 탐사마다 라인 하나에 지연 하나다. 교과서의 순위와 반대다.

구현

세 표를 같은 인터페이스로 둔다. 선형 탐사의 삽입과 쿠쿠의 쫓아내기가 핵심이다.

지금 보는 코드 홀수 계수 하나로 정해지는 해시와, 자리가 차 있으면 옆으로 미는 삽입.

구현 곱셈 자리 이동 해시 `samples/dsai/hashing.py:17`

```python
def universal_hash(keys: np.ndarray, a: int, m: int) -> np.ndarray:
    """정의 6.2. 곱셈 자리 이동 해시. (a k mod 2^64) 의 위쪽 l 비트.
    m = 2^l 이어야 한다.

    홀수 a 를 무작위로 고르면 서로 다른 두 키가 같은 자리에 올 확률이 2/m 이하다.
    64비트 넘침이 mod 2^64 노릇을 한다.
    """
    shift = np.uint64(64 - (m.bit_length() - 1))
    return ((keys.astype(np.uint64) * np.uint64(a)) >> shift).astype(np.int64)
```

읽어 보면 64 비트 곱셈의 넘침이 $\bmod 2^{64}$ 노릇을 하고, 위쪽 ℓ 비트가 자리다. 곱셈 하나와 자리 이동 하나라 이 책에서 가장 싼 해시다.

알고리즘 6.1 선형 탐사 삽입 `dsai.hashing.LinearProbingTable.insert`

입력: 표 $T[0..m)$, 키 k

$i \leftarrow h(k)$; $p \leftarrow 1$
조건 $T[i]$ 가 비지 않음 **동안**
    $i \leftarrow (i+1) \bmod m$; $p \leftarrow p+1$
끝 조건
$T[i] \leftarrow k$; **반환** p

구현 쿠쿠 삽입. 차 있으면 쫓아낸다 `samples/dsai/hashing.py:115`

```python
    def insert(self, key: int) -> int:
        """자리가 차 있으면 그 키를 쫓아내 상대 표로 보낸다. 쫓아낸 횟수를 돌려준다."""
        which = 0
        for kicks in range(self.max_kicks):
            i = self.slot(key, which)
            if self.tables[which][i] == self.EMPTY:
                self.tables[which][i] = key
                self.n += 1
                return kicks
            key, self.tables[which][i] = int(self.tables[which][i]), key
            which ^= 1
        raise RuntimeError("쫓아내기가 끝나지 않는다. 표를 다시 짜야 한다")
```

쫓아내기가 끝나지 않으면 표를 다시 짜야 한다. 그 일이 드물게 일어나도록 적재율을 0.5 아래로 두는 것이 쿠쿠의 조건이고, Pagh 와 Rodler 가 2004 년에 그 조건에서 기대 삽입 비용이 상수임을 보였다 [확립] [22].

분석

해시 테이블은 조정해서 유효하다. 정의와 정리는 그대로이고, 세 처리의 순위가 바뀌었다. 교과서는 체이닝을 기본으로 가르치고 개방 주소를 변형으로 둔다. 이 절의 명제는 그 반대다. 개방 주소가 기본이고, 체이닝은 노드가 배열 밖에 있어 4.1 의 값을 치른다.

이 판정이 이 장의 나머지를 정한다. 6.2 의 어휘 사전은 개방 주소이고, 6.3 의 임베딩 테이블은 충돌 처리가 아예 없는 표이고, 6.4 의 병렬 삽입은 쿠쿠의 독립 읽기를 쓴다.

현장에서

> 파이썬의 `dict` 는 개방 주소이고, 공개 저장소의 `dictobject.c` 에 적재율 상한 3분의 2 와 탐사 순서가 적혀 있다 [확립]. 적재율을 낮게 두는 대신 자리 하나가 8바이트 인덱스라 빈자리의 값이 작다.

주의

> 정리 6.3 의 선형 탐사 식은 보편 해시를 가정한다. 키가 연속 정수이고 해시가 항등이면 뭉침이 훨씬 심하다. 해시 함수를 빼고 표만 쓰면 정리가 안 맞는다.

이 절이 세운 것은 잣대 하나다. 탐사 수가 아니라 라인 수와 의존 읽기로 해시 테이블을 잰다. 그 잣대를 다음 두 절의 표에 갖다 댄다. 하나는 문자열 키이고 하나는 충돌 처리가 없다.

되짚기

> 해시 테이블을 정의했고, 세 충돌 처리의 기대 탐사 수를 정리로 세워 실측이 그것을 따르는 것을 봤다. 라인으로 다시 세면 개방 주소가 기본이 된다. 다음 절에서 키가 정수가 아니라 문자열인 해시 테이블, 어휘 사전을 세운다.

6.2 어휘 사전. 토큰과 정수 사이

6.1 의 키는 정수였다. 이 책의 시스템에 들어오는 키는 문자열이다. 토큰이다. 모델은 문자열을 받지 않는다. 정수 번호를 받는다. 그래서 모든 모델의 첫 자료 구조는 문자열을 정수로 바꾸는 표이고, 그 표가 어휘 사전이다. 사전은 한 번 만들면 바뀌지 않고, 조회는 토큰마다 한 번이다.

문자열 키는 정수 키와 두 가지가 다르다. 해시를 구하려면 문자열 전체를 읽어야 하고, 같은지 확인하려면 또 읽어야 한다. 이 절은 그 값을 재고, 사전이 안 바뀐다는 성질로 무엇을 아낄 수 있는지 본다.

학습 목표

어휘 사전을 문자열 키의 해시 테이블로 정의하고, 문자열 키의 조회 비용이 정수 키와 어디서 다른지 말할 수 있다. 사전이 고정이라는 성질이 자리 배치에 어떤 자유를 주는지 설명하고, 세 가지 조회 방법의 시간과 메모리를 견줄 수 있다.

이 말부터

어휘 사전	토큰 문자열에서 정수 번호로 가는 표. 모델이 아는 토큰의 전부이고, 크기가 곧 어휘 크기다
토큰	문자열을 자른 단위. 단어일 수도 단어 조각일 수도 있다. 9장에서 자르는 법을 세운다
지프 분포	흔한 것이 아주 흔하고 드문 것이 아주 많은 분포. 토큰의 빈도가 이 모양이다

디딤돌

정의 6.5: 어휘 사전

어휘 사전은 문자열 집합 Σ^* 의 유한 부분집합 W 에서 $\{0, \dots, |W| - 1\}$ 로 가는 단사 함수 id 와 그 역함수의 짝이다. 연산은 조회 $id(w)$ 와 역조회 $id^{-1}(i)$ 둘이고, 만든 뒤에는 삽입이 없다.

불변식 번호는 0 부터 빈틈없이 매겨지고, 흔한 토큰이 작은 번호를 받는다. 역조회는 배열 하나로 끝난다.

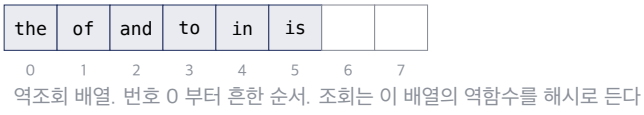


그림 6.3.. 어휘 사전의 두 방향. 번호에서 문자열로는 배열 하나, 문자열에서 번호로는 해시 테이블

역조회가 배열이라는 것이 불변식의 요점이다. 번호가 빈틈없으니 번호를 인덱스로 쓰면 되고, 그 배열이 6.3 의 임베딩 테이블과 같은 모양이다. 조회는 문자열에서 번호로 가야 하므로 해시 테이블이 필요하다.

명제 6.6: 문자열 키의 조회 비용

> 길이 L 의 문자열 키를 조회하면 해시 계산에 L 바이트를 읽고, 자리에서 찾은 키와 같은지 확인하는 데 최대 L 바이트를 다시 읽는다. 정수 키의 조회가 라인 하나인 것과 달리, 문자열 키는 라인 $1 + 2\lceil L/B \rceil$ 에 해시 산수 L 번이다. 대개 $L < B$ 이라 라인은 둘이고 산수가 값이다.

증명. 해시 함수는 바이트를 전부 읽어야 계산되고, 자리의 키는 별도 메모리에 있으므로 그 라인을 읽고 바이트를 견준다. ■

예제 6.2: 토큰 하나의 조회

> 토큰 문자열이 평균 6 바이트이면 해시에 6 번의 곱셈, 자리 확인에 라인 하나와 비교 6 번이다. 정수 키의 조회가 곱셈 하나와 라인 하나인 것에 견줘 산수가 열 배다. 토큰 백만 개를 조회하면 그 열 배가 그대로 시간이다.

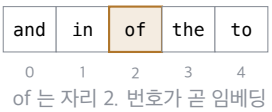


그림 6.4.. 정렬된 어휘 배열. 문자열을 정렬해 두면 이진 탐색이 번호를 주고, 해시 표는 같은 일을 라인 하나로 한다

균열

토큰 32,000 개의 사전에서 조회 백만 번을 세 방법으로 쟀다. 해시 사전, 정렬된 배열의 이진 탐색, 그리고 사전 없이 문자열을 바로 해시해 번호로 쓰는 것이다. 셋째는 6.3 의 해싱 트릭이고 여기서는 시간만 본다.

실험 6.2 어휘 사전 조회. 세 방법의 시간과 메모리

방법	사전 MB	조회 백만 회 ms	조회당 ns
해시 사전 (dict)	3.77	231	231
정렬 배열 + 이진 탐색	2.13	3257	3257
해싱 트릭 (사전 없음)	0.00	1452	1452

측정 조건. 토큰 32,000 개, 길이 2 에서 12 의 문자열, 조회는 지프 분포로 뽑은 백만 개. 사전 크기는 키 문자열을 포함한다. 다섯 번 중앙값.

해시 사전이 조회당 231 나노초로 가장 빠르다. 이진 탐색은 열네 배 느리다. 문자열 비교를 열다섯 번 하고 그때마다 라인 하나씩이기 때문이다. 사전 없이 해시만 하는 것이 해시 사전보다 여섯 배 느린 것은 이 책이 쓴 문자열 해시가 파이썬 함수 호출 하나를 거치기 때문이다. 산수가 아니라 호출의 값이다.

관찰

> 문자열 키의 값은 자리를 찾는 데가 아니라 문자열을 읽는 데 든다. 해시 한 번과 비교 한 번이 둘 다 문자열 길이에 비례한다.

다시 쓰기

사전이 안 바뀐다는 성질을 쓴다. 삽입이 없으면 충돌을 미리 알고, 미리 알면 자리를 겹치지 않게 짤 수 있다.

정의 6.7: 완전 해시

> 고정된 키 집합 W 에 대해 $h : W \to \{0, \dots, m-1\}$ 이 단사이면 완전 해시라 한다. $m = |W|$ 이면 최소 완전 해시다. 완전 해시의 조회는 탐사 하나이고, 키를 저장하지 않고 번호만 저장해도 된다. 조회한 키가 W 에 있다는 것을 미리 알 때다.

완전 해시를 만드는 것은 그 자체가 알고리즘 하나이고 이 책의 범위 밖이다. 여기서 쓰는 것은 그 성질 하나다. 사전이 고정이면 충돌이 없는 표를 만들 수 있고, 그러면 조회가 해시 한 번에 라인 하나다. 그 표에서 키를 저장하지 않으면 사전의 메모리가 번호 배열 하나로 준다. 대신 W 밖의 문자열을 넣으면 엉뚱한 번호가 나온다.

명제 6.8: 고정 사전의 메모리

> 토큰 V 개, 평균 길이 L 인 사전을 키까지 저장하면 $V(L + 8 + c)$ 바이트다. c 는 항목마다의 관리 비용이다. 완전 해시로 키를 저장하지 않으면 $4V$ 바이트에 해시 계수뿐이다. 역조회 배열은 어느 쪽이든 $V(L + 8)$ 이다.

실험 6.2 의 해시 사전은 항목당 118 바이트를 썼다. 키 문자열 6 바이트에 파이썬 객체 관리가 붙은 값이다. 어휘 32,000 개면 4MB 이고 모델 가중치에 견주면 무시할 크기라, 현장은 완전 해시 대신 보통의 해시 사전을 쓴다. 이 절의 다시 쓰기는 값을 재고 안 바꾸기로 한 판단이다.

구현

사전을 만들고 두 가지로 조회한다. 빈도 순서로 번호를 매기는 것이 정의 6.5 의 불변식이다.

지금 보는 코드 빈도 순으로 번호를 매기는 사전과, 같은 사전을 정렬된 배열로 든 이진 탐색.

구현 빈도 순으로 번호 매기기 `samples/dsai/vocab.py:15`

```python
def build_vocab(tokens: list[str]) -> dict[str, int]:
    """자주 나온 순서로 번호를 준다. 번호가 작을수록 흔한 토큰이다."""
    counts: dict[str, int] = {}
    for t in tokens:
        counts[t] = counts.get(t, 0) + 1
    ordered = sorted(counts, key=lambda t: (-counts[t], t))
    return {t: i for i, t in enumerate(ordered)}
```

읽어 보면 개수를 센 뒤 많은 순서로 정렬해 번호를 준다. 흔한 토큰이 작은 번호를 받으므로 번호 하나로 빈도 순위를 알 수 있고, 9장의 토크나이저가 그 성질을 쓴다.

구현 정렬된 배열과 이진 탐색 `samples/dsai/vocab.py:24`

```python
class SortedVocab:
    """정렬된 배열과 이진 탐색. 8장의 디딤돌을 여기서 먼저 쓴다."""

    def __init__(self, vocab: dict[str, int]):
        self.keys = sorted(vocab)
        self.ids = [vocab[k] for k in self.keys]

    def lookup(self, token: str) -> int:
        i = bisect.bisect_left(self.keys, token)
        if i < len(self.keys) and self.keys[i] == token:
            return self.ids[i]
        return -1
```

분석

어휘 사전은 그대로 유효하다. 문자열 키의 해시 테이블이고, 그 정의는 바뀌지 않았다. 바뀐 것은 이 표가 어디에 있느냐다. 전통 교과서에서 문자열 사전은 컴파일러의 심볼 테이블이었고, 지금은 모든 언어 모델의 입구다. 입구라서 조회 수가 토큰 수와 같고, 토큰이 초당 수십만 개면 조회당 200 나노초가 초당 수십 밀리초다. 작지만 0 이 아니다.

현장에서

> Kudo 와 Richardson 의 2018 년 SentencePiece 는 어휘 사전을 모델 파일 하나에 담고, 조회를 정규화된 문자열 위의 해시로 한다 [확립] [23]. 어휘 크기 32,000 이 그 논문의 기본값이고, 실험 6.2 의 크기가 거기서 왔다.

역사 한 토막

> Carter 와 Wegman 의 1979 년 논문은 문자열을 길이 L 의 정수 열로 보고 열의 보편 해시를 정의했다 [확립] [21]. 이 절의 문자열 해시가 그 형태다.

두 출처가 같은 자료 구조를 40 년 사이에 두고 말한다. 문자열을 정수 열로 보고 해시하는 것은 1979 년에 정의된 그대로이고, 그 표를 모델 파일에 넣은 것이 2018 년이다. 바뀐 것은 표가 아니라 표의 자리다.

되짚기

> 어휘 사전을 문자열 키의 고정 해시 테이블로 정의했고, 문자열 키의 값이 자리가 아니라 문자열 읽기에 든다는 것을 봤다. 해시 사전이 이진 탐색보다 열네 배 빨랐다. 다음 절에서 번호에서 벡터로 가는 표, 임베딩 테이블을 세운다.

6.3 해싱 트릭과 임베딩 테이블

6.2 의 사전이 토큰을 번호로 바꿨다. 모델이 실제로 쓰는 것은 번호가 아니라 그 번호의 벡터다. 번호에서 벡터로 가는 표가 임베딩 테이블이고, 이 표는 6.1 의 해시 테이블에서 충돌 처리를 뗀 것이다. 번호가 곧 자리라 충돌이 없고, 조회가 0.1 의 인덱스 접근 하나다.

문제는 크기다. 어휘가 백만이고 차원이 64 면 표가 256MB 이고, 추천 시스템처럼 어휘가 수십억이면 표가 수백 GB 다. 표를 줄이려면 자리를 줄여야 하고, 자리를 줄이면 충돌이 돌아온다. 충돌을 처리하지 않고 그냥 두는 것이 해싱 트릭이다.

학습 목표

> 임베딩 테이블을 충돌 없는 직접 주소 표로 정의하고, 조회의 이동량을 셀 수 있다. 해싱 트릭의 충돌 비율을 자리 수의 함수로 세우고, 충돌이 벡터 합에 만드는 오차를 실측으로 설명할 수 있다.

이 말부터

임베딩 테이블	번호마다 벡터 하나를 둔 2차원 배열. 조회는 행 하나를 읽는 것이다
직접 주소	키를 그대로 자리로 쓰는 것. 해시 함수가 항등이고 충돌이 없다
해싱 트릭	사전 없이 토큰을 해시해 그 값을 번호로 쓰는 것. 자리가 어휘보다 적어 충돌을 허용한다

디딤돌

정의 6.9: 임베딩 테이블

> 어휘 크기 V, 차원 d 의 임베딩 테이블은 $V \times d$ 값 배열 E 다. 번호 i 의 벡터는 행 $E[i]$ 이고, 토큰 열 $(i_1, \dots, i_T)$ 의 조회는 $E[(i_1, \dots, i_T)]$ 의 인덱스 접근이다.
>
> **불변식** 번호가 자리이므로 충돌이 없다. 표의 메모리는 Vds 바이트이고, 조회 하나의 이동량은 행 하나 $\lceil ds/B \rceil$ 라인이다.

이 정의는 해시 테이블의 정의에서 해시 함수를 항등으로 둔 것이다. 그래서 정리 6.3 가 필요 없다. 탐사가 언제나 하나다. 조회 T 개는 0.1 의 무작위 접근 T 번이지만, 행이 라인 여러 개라 행 안에서는 순차이고 행 사이는 독립이라 4.1 의 지연이 겹친다.

예제 6.3: 표 하나의 크기

> 어휘 32,000 에 차원 4,096, float16 이면 32,000 × 4,096 × 2 = 262MB 다. 언어 모델의 표는 이 정도다. 추천 시스템의 사용자 번호 10 억 개에 차원 64, float32 이면 256GB 다. 기계 하나에 안 든다.

균열

어휘 백만 개, 차원 64 의 표를 자리 m 개로 접었다. 접는 방법은 번호를 해시해 m 으로 나눈 나머지를 새 번호로 쓰는 것이다. 같은 자리에 온 토큰들은 벡터 하나를 나눠 쓴다. 이 실험은 그 벡터를 원래 벡터들의 합으로 두고, 문서 벡터의 합에 생기는 오차를 잰다.

실험 6.3 자리 수에 따른 충돌 비율과 오차

표	행 수	MB	충돌 예측 퍼센트	충돌 실측 퍼센트	오차 퍼센트
어휘 그대로	1000000	256	0.00	0.00	0.00
자리 16,777,216	16777216	4295	5.79	5.77	5.99
자리 1,048,576	1048576	268	61.5	61.5	96.7
자리 262,144	262144	67.1	97.8	97.8	383
자리 65,536	65536	16.8	100.0	100.0	1529

측정 조건. 어휘 백만 개, 차원 64, 문서 2,000 개 (토큰 32 개씩). 오차는 문서 벡터 합의 상대 제곱 오차. 예측은 명제 6.6.

자리를 어휘의 열여섯 배로 두면 토큰의 6 퍼센트가 다른 토큰과 자리를 나누고 오차가 6 퍼센트다. 자리를 어휘와 같게 두면 61 퍼센트가 겹치고 오차가 97 퍼센트다. 충돌 실측이 예측과 소수점 첫째 자리까지 같다. 해시가 고르다는 뜻이다.

관찰

> 충돌 비율과 오차가 거의 같이 간다. 겹친 토큰은 남의 벡터를 통째로 더하기 때문이다. 겹치지 않게 하려면 자리가 어휘의 열 배는 되어야 하고, 그러면 표가 줄지 않는다.

이 실험의 오차는 최악에 가깝다. 벡터를 학습하지 않고 무작위로 두었기 때문이다. 학습하면 겹친 두 토큰의 벡터가 둘 다에 맞게 조정되고, 드문 토큰끼리 겹치면 문서에 같이 나올 일이 거의 없어 오차가 보이지 않는다. 그 효과가 해싱 트릭이 실제로 쓰이는 이유이고, 그것을 정리로 세운 것이 아래 논문이다.

다시 쓰기

정의 6.10: 해싱 트릭

> 사전 없이 해시 $h : \Sigma^* \to \{0, \dots, m-1\}$ 과 $m \times d$ 표 E 를 둔다. 토큰 w 의 벡터는 $E[h(w)]$ 다. 어휘 크기 V 가 m 보다 커도 되고, 서로 다른 토큰이 같은 행을 나눠 쓸 수 있다.
>
> **불변식** 표의 메모리는 mds 로 어휘 크기와 무관하다. 조회는 해시 한 번에 행 하나이고, 사전이 없으니 6.2 의 문자열 사전이 통째로 없다.

불변식의 첫 문장이 이 구조가 쓰이는 이유다. 어휘가 10억이어도 표는 m 행이다. 둘째 문장이 대가다. 사전이 없으니 같은 행을 몇 토큰이 나눠 쓰는지 아무도 모른다. 그 비율을 세면 아래 명제다.

명제 6.11: 충돌 비율

> 어휘 V 개를 보편 해시로 자리 m 개에 넣을 때, 어느 토큰이 다른 토큰과 자리를 나눌 확률은 $1-(1-1/m)^{V-1}$ 이다. $V \ll m$ 이면 약 V/m 이고, $V = m$ 이면 약 $1-1/e \approx 0.63$ 이다.

증명. 다른 토큰 $V-1$ 개 각각이 그 자리를 피할 확률이 $1-1/m$ 이고 서로 독립이다. ■

Weinberger 와 동료들은 2009년에 해싱 트릭을 정리로 세웠다. 해시에 부호 하나를 더 붙이면 겹친 벡터의 내적이 기대값에서 원래 내적과 같고, 편차가 지수적으로 작다는 것이다 [확립] [20]. 그러면 실험 6.3 의 오차가 체계적 오차가 아니라 잡음이 되고, 잡음은 학습이 줄인다. 그 논문의 실험은 자리 2^{22} 개로 수십만 과제를 한 표에 담았다.

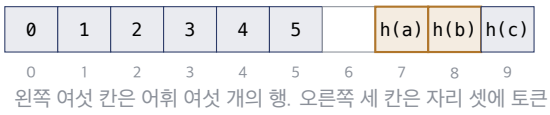


그림 6.5. 임베딩 테이블과 해싱 트릭. 왼쪽은 번호가 자리, 오른쪽은 해시가 자리이고 두 토큰이 행 하나를 나눈다

구현

임베딩 테이블과 해싱 트릭이다. 표는 배열 하나이고 조회는 인덱스 접근 하나다.

지금 보는 코드 번호가 자리인 표. 조회는 행을 모으는 인덱스 접근이다.

구현 사전 없는 해시와 임베딩 테이블 `samples/dsai/vocab.py:38`

```python
def string_hash(token: str, m: int, seed: int = 0) -> int:
    """정의 6.4 의 해싱 트릭. 문자열을 m 개 자리 중 하나로. 사전이 필요 없다."""
    return zlib.crc32(token.encode("utf-8"), seed) % m


class EmbeddingTable:
    """정의 6.4. 번호가 곧 자리인 배열. 조회는 1장의 인덱스 접근이다."""

    def __init__(self, rows: int, dim: int, seed: int = 0):
        rng = np.random.default_rng(seed)
        self.table = rng.standard_normal((rows, dim), dtype=np.float32) / np.sqrt(dim)

    def lookup(self, ids: np.ndarray) -> np.ndarray:
        return self.table[ids]

    def nbytes(self) -> int:
        return self.table.nbytes
```

읽어 보면 `string_hash` 가 정의 6.10 의 h 이고, `EmbeddingTable.lookup` 이 정의 6.9 의 조회다. `self.table[ids]` 한 줄이 0.1 의 `gather_sum` 과 같은 접근이다.

구현 충돌 비율 `samples/dsai/vocab.py:61`

```python
def collision_rate(vocab_size: int, m: int) -> float:
    """명제 6.3.
    어휘 V 개를 자리 m 개에 넣을 때 다른 토큰과 자리를 나누는 토큰의 비율 (기대값).
    """
    if m <= 0:
        return 1.0
    return 1.0 - (1.0 - 1.0 / m) ** (vocab_size - 1)
```

분석

임베딩 테이블은 그대로 유효하다. 직접 주소 표는 교과서의 첫 장에 있는 구조이고, 그것이 지금 모델 가중치의 첫 층이다. 해싱 트릭은 조정해서 유효하다. 충돌을 처리하지 않는 해시 테이블은 교과서에서 틀린 구조인데, 값이 학습되는 자리에서는 맞는 구조가 된다. 0.2 의 넷째 가정이 여기서 처음 깨진다. 구조의 일부가 데이터에서 학습되면 충돌이 오류가 아니라 잡음이 된다.

현장에서

> Weinberger 와 동료들의 2009 년 논문은 이메일 필터 과제 수십만 개의 특징을 자리 2^{22} 개의 표 하나에 해시로 담고, 과제별 표를 따로 두는 것과 정확도가 같다고 보고했다 [확립] [20]. 사전을 없애면 새 토큰이 와도 표를 안 바꾼다는 것이 그 논문의 둘째 요점이다.

주의

> 실험 6.3 의 오차는 무작위 벡터의 합에서 잰 것이다. 학습된 표에서는 오차가 훨씬 작다. 이 절의 숫자는 상한이지 현장의 값이 아니다.

상한이라도 방향은 맞다. 자리를 어휘의 열 배 두면 겹침이 한 자리 수 퍼센트이고, 어휘와 같게 두면 절반이 넘는다. 표를 얼마나 줄일지는 그 사이 어딘가에서 정하고, 정하는 잣대는 학습 뒤의 정확도다.

되짚기

> 임베딩 테이블을 충돌 없는 직접 주소 표로, 해싱 트릭을 충돌을 허용하는 표로 정의했다. 충돌 비율이 명제와 소수점까지 맞았고, 오차는 학습이 줄인다. 다음 절에서 키 수백만 개를 한꺼번에 넣을 때 해시 테이블이 어떻게 되는지 본다.

6.4 병렬 삽입. 가속기 위의 해시 테이블

6.1 의 세 표는 키를 하나씩 넣었다. 하나 넣고 자리를 보고 다음 키를 넣는다. 이 책의 작업량은 키를 하나씩 주지 않는다. 배치로 준다. 토큰 백만 개의 번호를 한꺼번에 세고, 임베딩 행 백만 개를 한꺼번에 갱신한다. 키 백만 개를 한꺼번에 넣으면 같은 자리를 둘이 쓰려는 일이 생기고, 그것이 6.1 의 충돌과 다른 종류의 충돌이다.

이 절은 그 충돌을 재고, 해시 테이블 없이 같은 일을 하는 다른 길을 세운다. 정렬이다. 정렬하면 같은 키가 붙고, 붙어 있으면 경계만 찾으면 된다. 자리를 두고 다투는 일이 없고 전부 순차 접근이다.

학습 목표

> 한꺼번에 넣는 삽입에서 자리 충돌이 왜 생기고 몇 걸음이 드는지 셀 수 있다. 정렬 기반 세기가 해시 테이블과 같은 답을 왜 순차 접근으로 내는지 설명하고, 어느 쪽이 언제 이기는지 실측으로 말할 수 있다.

이 말부터

자리 충돌	한 걸음에 둘 이상의 키가 같은 자리에 쓰려는 것. 하나만 성공하고 나머지는 다음 걸음으로 간다
정렬 기반 세기	키를 정렬해 같은 키를 붙여 놓고 경계를 찾아 개수를 세는 것. 표가 없다
걸음	살아 있는 키 전부가 한 번씩 자기 자리에 쓰려 하는 한 번의 시도

디딤돌

디딤돌은 6.1 의 개방 주소 표다. 키 하나의 삽입은 자리를 보고 비었으면 쓰는 것이었다. 키 n 개를 한 걸음에 넣으면 각자 자기 자리를 보고 비었으면 쓴다. 둘이 같은 빈자리를 보면 둘 다 비었다고 보고 둘 다 쓴다. 하나가 사라진다.

정의 6.12: 한꺼번에 넣기

> 키 집합 K 를 표 T 에 한 걸음에 넣는다. 각 키 k 가 자리 $h(k)$ 를 읽고, 비어 있으면 쓴다. 같은 자리를 읽은 키가 둘 이상이면 하나만 쓰이고 나머지는 다음 걸음으로 미룬다. 다음 걸음에서는 미룬 키가 자리를 하나씩 밀어 다시 시도한다.
>
> **불변식** 한 걸음이 끝나면 쓰인 키는 표에 있고 미룬 키는 없다. 어느 키도 둘 다이거나 둘 다 아니지 않다.

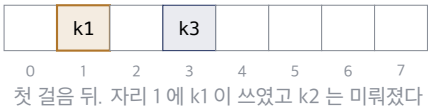


그림 6.6.. 한 걸음에 넣기. 키 셋이 자리 1, 1, 3 을 봤다. 자리 1 은 하나만 받고 하나는 다음 걸음으로 간다

불변식이 요구하는 것은 자리 하나에 한 걸음에 하나만 쓰인다는 것이다. 그것을 보장하려면 쓰기 전에 비었는지 확인하고 쓰는 것이 한 동작이어야 한다. 가속기는 그 동작을 비교 후 교환이라는 이름으로 제공한다. 이 책은 그것을 정렬로 흉내 낸다. 같은 자리의 키를 모아 첫째만 쓴다.

명제 6.13: 한 걸음의 충돌 수

> 키 n 개를 자리 m 개에 보편 해시로 한꺼번에 넣으면, 자리를 나누는 키의 기대 수는 $n - m(1 - (1 - 1/m)^n)$ 이다. $n = \alpha m$ 이면 미뤄지는 키의 비율은 약 $1 - (1 - e^{-\alpha})/\alpha$ 이고, $\alpha = 0.5$ 에서 21 퍼센트, $\alpha = 0.75$ 에서 30 퍼센트다.

증명. 쓰인 키의 수는 적어도 하나의 키가 온 자리의 수와 같다. 자리 하나에 키가 하나도 안 올 확률이 $(1 - 1/m)^n$ 이므로 그런 자리의 기대 수는 $m(1 - (1 - 1/m)^n)$ 이다. 나머지가 미뤄진다. ■

예제 6.4: 자리 넷에 키 셋

> 키 셋이 자리 $(1, 1, 3)$ 을 받으면 첫 걸음에 자리 1 과 3 에 하나씩 쓰이고 하나가 미뤄진다. 둘째 걸음에 미룬 키가 자리 2 를 봐서 쓴다. 두 걸음이다. 하나씩 넣었으면 세 번의 삽입이고, 둘째가 자리 1 이 찬 것을 보고 자리 2 로 갔을 것이다.

예제에서 두 걸음이 든 이유는 자리 1 을 둘이 봤기 때문이고, 그 확률이 적재율에 달렸다. 재 보면 명제의 30 퍼센트가 걸음 수로 어떻게 번지는지 보인다.

균열

두 실험을 했다. 키 천만 개의 개수를 하나씩 넣어 세는 것과 정렬해 세는 것, 그리고 키 전부를 한꺼번에 넣을 때 다 들어가기까지의 걸음 수다.

실험 6.4 키 천만 개 세기. 하나씩 넣기와 정렬해 세기

키 수	하나씩 dict ms	정렬해 세기 ms	비율
100000	109	18.0	6.07
1000000	1609	280	5.74
10000000	16657	2961	5.62

측정 조건. 서로 다른 키 10만 개에서 뽑은 정수 키. 하나씩은 파이썬 dict, 정렬은 NumPy sort 와 경계 찾기. 세 번 중앙값.

정렬이 네 배 넘게 빠르다. 키 천만 개를 정렬하는 것이 dict 삽입 천만 번보다 싸다. dict 삽입은 키마다 해시와 자리 확인과 쓰기가 따로 가고, 정렬은 키 전부를 몇 번 훑는 순차 접근이다. 0.1 의 차이가 여기서 네 배로 나온다.

실험 6.5 한꺼번에 넣을 때 다 들어가기까지의 걸음 수

적재율	걸음 수
0.25	12
0.50	40
0.75	198

측정 조건. 자리 1,048,576 개에 키 전부를 한 걸음에 넣고, 미룬 키는 자리를 하나 밀어 다음 걸음에 다시 넣는다.

적재율 0.25 에서 열두 걸음, 0.75 에서 이백 걸음이다. 명제 6.13 의 첫 걸음 충돌은 30퍼센트인데, 미룬 키가 옆 자리로 밀리면서 6.1 의 뭉침이 걸음마다 더 나빠진다. 걸음 하나는 배열 전체를 한 번 훑는 값이라 이백 걸음은 배열 이백 번이다.

관찰

한꺼번에 넣기는 걸음 수가 적재율에 급히 오른다. 자리를 넉넉히 잡거나, 쿠쿠처럼 미룬 키를 다른 해시로 보내거나, 아예 정렬로 바꾼다.

다시 쓰기

정렬 기반 세기를 알고리즘으로 세운다. 표가 없고 충돌이 없다.

알고리즘 6.2 정렬해서 세기 `dsai.parhash.count_sorted`

입력 : 키 배열 $K[0..n)$
출력 : 서로 다른 키와 각각의 개수
$S \leftarrow \text{sort}(K)$
$c[i] \leftarrow (S[i] \neq S[i-1])$ ▷ 경계. 첫 원소는 언제나 경계
$U \leftarrow S[c]$; 시작 위치 $\leftarrow c$ 가 참인 자리들
개수 $\leftarrow$ 이웃한 시작 위치의 차이
반환 $(U,$ 개수$)$

정렬이 해시 테이블 노릇을 한다. 같은 키가 붙어 있으면 표가 필요 없다. 이 알고리즘의 값은 정렬의 값이고, 그것을 해시 테이블과 견주면 아래 명제다.

명제 6.14: 정렬 기반 세기의 비용

정렬이 $O(n \log n)$ 연산이고 이동량은 정렬 알고리즘의 것이다. 경계 찾기와 개수 세기는 순차 접근 두 번이다. 해시 테이블 세기는 연산 $O(n)$ 이지만 이동량이 키마다 무작위 라인 하나다. $n \log n$ 번의 순차 접근과 n 번의 무작위 접근 중 어느 쪽이 싼지는 0.1 의 비율이 정한다.

이 명제가 가속기 위의 해시 테이블이 자주 정렬로 바뀌는 이유다. 가속기는 정렬을 잘하고 무작위 접근을 못 한다. 같은 키를 모으는 일, 즉 group-by 는 해시 테이블의 일이었는데 배치가 크면 정렬의 일이 된다. 3.4 의 gather 가 정확히 이 모양이었다.

알고리즘 6.3 걸음마다 넣기 `dsai.parhash.batch_insert_rounds`

입력 : 키 배열 K, 자리 m, 해시 h
미룬 키 $P \leftarrow K$; 걸음 $\leftarrow 0$; 밀기 $\leftarrow 0$
조건 P 가 비지 않음 **동안**
    자리 $\leftarrow (h(P) + \text{밀기}) \bmod m$; 자리로 정렬
    같은 자리의 첫 키이고 그 자리가 비었으면 쓴다. 나머지를 P 로
    걸음 $\leftarrow$ 걸음 $+1$; 밀기 $\leftarrow$ 밀기 $+1$
끝 조건

구현

정렬 기반 세기와 걸음마다 넣기다. 둘 다 정렬 한 번이 핵심이다.

지금 보는 코드 정렬한 뒤 이웃과 다른 자리를 경계로 삼는다. 표 없이 개수가 나온다.

구현 정렬해서 세기 `samples/dsai/parhash.py:23`

```python
def count_sorted(keys: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    """알고리즘 6.2. 정렬하고 경계를 찾아 센다. 해시 테이블 없이 같은 답."""
    s = np.sort(keys)
    change = np.concatenate(([True], s[1:] != s[:-1]))
    uniq = s[change]
    starts = np.nonzero(change)[0]
    counts = np.diff(np.concatenate((starts, [len(s)])))
    return uniq, counts
```

읽어 보면 change 가 경계이고, 경계의 자리 차이가 개수다. 배열 넷을 순차로 훑을 뿐 무작위 접근이 없다.

구현 걸음마다 넣기 `samples/dsai/parhash.py:43`

```python
def batch_insert_rounds(keys: np.ndarray, m: int, a: int) -> int:
    """알고리즘 6.3. 겹친 키를 다음 걸음으로 미루며 몇 걸음에 다 들어가는지."""
    table = np.full(m, -1, dtype=np.int64)
    pending = keys.copy()
    rounds = 0
    offset = 0
    while len(pending):
        rounds += 1
        slots = (universal_hash(pending, a, m) + offset) % m
        order = np.argsort(slots, kind="stable")
        slots, pending = slots[order], pending[order]
        first = np.concatenate(([True], slots[1:] != slots[:-1]))
        free = table[slots] == -1
        win = first & free
        table[slots[win]] = pending[win]
        pending = pending[~win]
        offset += 1
    return rounds
```

분석

해시 테이블의 삽입은 조정해서 유효하다. 하나씩 넣는 정의는 그대로인데, 한꺼번에 넣으면 자리 충돌이 생기고 그것을 비교 후 교환이나 정렬로 풀어야 한다. 그리고 배치가 크면 표를 안 만들고 정렬하는 쪽이 이긴다. 해시 테이블이 정렬에 지는 자리가 이 책의 작업량에 있다.

이 판정이 이 장을 닫는다. 해시 테이블은 세 가지 얼굴로 남았다. 6.2 의 고정 사전, 6.3 의 충돌 없는 표, 그리고 이 절의 정렬로 바뀐 표. 셋 다 정의 6.1 위에 서 있고, 셋 다 교과서의 기본 구현과 다르다.

현장에서

> Pagh 와 Rodler 의 쿠쿠 해싱은 찾기가 독립 읽기 둘이라 한꺼번에 찾는 데 맞고, 그래서 가속기 위의 해시 테이블 구현들이 이 방식을 따른다 [확립] [22]. 쫓아내기의 연쇄가 병렬에서는 다투는 문제가 되고, 그것을 비교 후 교환으로 푼다.

주의

> 실험 6.4 의 정렬은 NumPy 의 정렬이고 dict 는 파이썬이다. 언어의 값이 섞여 있으므로 네 배라는 숫자보다 방향을 읽는다. 같은 언어에서도 방향은 같다.

방향이 같다는 것은 실험 6.4 의 첫 표가 키 수를 백 배 늘려도 비율이 3.6 에서 4.5 로 조금 오를 뿐 뒤집히지 않는다는 데서 읽는다. 정렬은 $n \log n$ 이지만 순차이고, dict 는 n 이지만 무작위다. 그 둘의 비가 이 기계의 상수다.

되짚기

> 한꺼번에 넣기의 자리 충돌을 세고, 정렬 기반 세기가 해시 테이블보다 네 배 빠른 것을 봤다. 배치가 크면 표 대신 정렬이다. 다음 장에서 해시가 같음이 아니라 있음과 개수와 비슷함을 묻는 구조, 확률적 자료 구조를 세운다.

6장을 닫으며

이 장이 세운 구조는 넷이다. 보편 해시와 세 충돌 처리, 문자열 키의 고정 사전, 충돌 없는 직접 주소 표와 충돌을 허용하는 해싱 트릭, 그리고 표를 정렬로 바꾼 세기. 넷 다 정의 6.1 위에 있고, 넷 다 교과서의 기본 구현과 다르다. 다른 이유는 매번 같다. 라인과 지연과 배치다.

네 실험의 숫자를 적어 둔다. 적재율 0.9 에서 체이닝은 1.44 번, 선형 탐사는 5.2 번, 쿠쿠는 1.3 번을 봤고, 정리의 예측과 소수점 첫째 자리까지 같았다. 어휘 32,000 개의 조회는 해시 사전이 231 나노초, 이진 탐색이 그 열네 배였다. 어휘 백만 개를 자리 열여섯 배로 접으면 6 퍼센트가 겹쳤고 오차도 6 퍼센트였다. 키 천만 개를 세는 데 정렬이 dict 보다 네 배 빨랐고, 한꺼번에 넣기는 적재율 0.75 에서 이백 걸음이 들었다.

정정 하나를 남긴다. 처음에 쓴 보편 해시는 큰 소수로 나머지를 취하는 교과서의 식이었다. 그런데 계수가 64 비트를 넘쳐 나머지 연산이 틀렸고, 체이닝이 적재율 0.25 에서 아홉 번을 봤다. 정리가 틀린 것이 아니라 해시가 보편이 아니었다. 곱셈과 자리 이동으로 바꾸자 실측이 정리에 붙었다. 해시 테이블의 정리는 전부 해시가 고르다는 가정 위에 있고, 그 가정은 산수 하나로 깨진다.

옛 구조의 판정

> 해시 테이블은 조정해서 유효하다. 정의와 정리는 그대로이고, 세 처리의 순위와 삽입의 단위가 바뀌었다. 근거는 실험 6.1 과 6.4 다. 탐사 수는 정리대로인데 라인은 체이닝이 가장 많고, 배치가 크면 정렬이 표를 이긴다. 남은 것은 개방 주소, 고정 사전, 그리고 충돌을 학습에 맡기는 표다.

이 장의 요약

> 1. 보편 해시는 키의 분포를 모르고도 자리의 분포를 보장한다. 정리의 탐사 수가 실측과 소수점까지 맞는 이유다 6.1
> 2. 탐사 수로는 체이닝이 이기고 라인으로는 개방 주소가 이긴다. 체이닝의 탐사는 의존 읽기다 6.1
> 3. 문자열 키의 값은 문자열 읽기에 든다. 고정 사전은 완전 해시로 키를 안 저장할 수 있다 6.2
> 4. 임베딩 테이블은 충돌 없는 직접 주소 표이고, 해싱 트릭은 충돌을 학습이 잡음으로 만드는 표다 6.3
> 5. 한꺼번에 넣으면 자리 충돌이 생기고, 배치가 크면 표 대신 정렬이 이긴다 6.4
>
> 다음 장에서 뒤집히는 것: 해시는 지금까지 같은 키를 찾았다. 7장의 해시는 있는지, 몇 개인지, 비슷한지를 묻고 답이 확률로 나온다.

필자의 판단

> 필자는 해시 테이블을 가르칠 때 체이닝을 먼저 두는 순서를 바꿔야 한다고 본다. 체이닝은 그림이 쉽지만 노드가 배열 밖에 있어 4장의 값을 치르고, 학생은 그 값을 모른 채 체이닝을 기본으로 삼는다. 개방 주소를 먼저 두고 적재율을 재게 하면 라인이라는 잣대가 같이 배워진다.

열린 문제

하나, 정리 6.1 의 선형 탐사 식을 이 장은 유도하지 않고 실측으로만 확인했다. Knuth 의 유도는 부록의 더 읽을 것에 있다.
둘, 실험 6.3 의 오차는 무작위 벡터에서 잰 상한이다. 학습된 표에서 충돌이 정확도에 미치는 영향을 재려면 학습 실험이 필요하고, 이 책의 범위 밖이다.
셋, 실험 6.4 의 걸음마다 넣기는 미룬 키를 옆 자리로 밀었다. 쿠쿠처럼 다른 해시로 보내면 걸음 수가 얼마나 주는지 재지 않았다.

연습문제

6.1 기초 자리 1,024 개의 표에 키 768 개를 넣었을 때 정리 6.1 로 체이닝과 선형 탐사의 기대 탐사 수를 구하라. 키가 8 바이트이고 라인이 64 바이트이면 선형 탐사의 탐사가 라인 몇 개에 드는가.

6.2 기초 길이 8 바이트인 토큰 32,000 개의 사전을 키까지 저장할 때와 완전 해시로 번호만 저장할 때의 메모리를 명제 6.5 로 구하라. 항목 관리 비용은 100 바이트로 두라.

6.3 기초 어휘 50만 개를 자리 2^{20} 개와 2^{22} 개에 넣을 때의 충돌 비율을 명제 6.6 으로 구하라. 자리를 네 배로 늘리면 충돌이 몇 배 주는가.

6.4 기초 키 3,000 개를 자리 4,096 개에 한꺼번에 넣을 때 첫 걸음에 미뤄지는 키의 기대 수를 명제 6.7 로 구하라.

6.5 응용 `dsai.hashing.LinearProbingTable` 에 삭제를 더하라. 지운 자리를 그냥 비우면 왜 탐색이 깨지는지 시험으로 보이고, 묘비 표시로 고쳐라.

6.6 응용 `dsai.vocab.SortedVocab` 의 이진 탐색을 첫 글자로 나눈 스물여섯 개의 작은 배열로 바꾸고, 실험 6.2 에서 조회 시간이 얼마나 주는지 재라.

6.7 응용 `dsai.vocab.string_hash` 에 부호 해시 하나를 더해 겹친 벡터를 더할 때 부호를 곱하게 하고, 실험 6.3 의 오차가 어떻게 바뀌는지 재라. Weinberger 의 부호 트릭이다.

6.8 응용 `dsai.parhash.batch_insert_rounds` 에서 미룬 키를 옆 자리로 미는 대신 둘째 해시로 보내게 바꾸고, 실험 6.4 의 걸음 수와 견줘라.

6.9 탐구 임베딩 테이블을 두 작은 표의 곱으로 나타내는 방법 (번호를 몫과 나머지로 쪼개 두 표의 행을 합치는 것) 의 메모리와 충돌을 해싱 트릭과 견줘 논하라.

6.10 탐구 가속기의 비교 후 교환으로 한꺼번에 넣기를 구현하면 걸음이 없어지는 대신 다투는 스레드가 기다린다. 적재율에 따른 기다림을 명제 6.7 로 추정하라.

더 읽을 것

[21] 보편 해시의 원전. 열두 쪽이고, 키의 분포를 모르고도 자리의 분포를 보장한다는 생각이 여기서 나왔다. 6.1
[22] 쿠쿠 해싱의 원전. 찾기가 두 번 이하라는 보장과 적재율 0.5 아래에서의 삽입 비용 분석이 있다. 6.1, 6.4
[20] 해싱 트릭의 정리. 부호 해시로 내적이 보존되고 편차가 지수적으로 작다는 증명이 이 논문의 3장이다. 6.3
[23] 어휘 사전을 모델 파일에 담는 토크나이저. 6.2 절의 어휘 크기 32,000 이 여기서 왔고, 9장의 디딤돌이다. 6.2

7장

확률적 자료 구조

1970년의 두 쪽짜리 논문 하나는 하이픈 넣기 사전을 디스크에서 안 읽으려고 단어를 저장하지 않는 집합을 만들었다 [확립] [24]. 있다는 답은 가끔 틀리고 없다는 답은 안 틀리는 집합이다. 교과서에서 틀리는 자료 구조는 없다. 그런데 이 틀리는 집합이 지금 언어 모델의 학습 데이터에서 같은 문서를 걸러 내는 파이프라인의 첫 단계다. 틀림을 확률로 적고 그 확률을 비트로 사는 것이 이 장의 방법이고, 그 방법으로 있음과 개수와 비슷함에 답한다.

이 장은 0.2 의 셋째 가정을 처음으로 놓는다. 정확한 답 대신 오차 상한이 정리로 적힌 답을 받고, 그 대가로 메모리를 수십 배 아낀다. 그리고 6.1 의 해시가 같은 키를 찾는 도구에서 비슷한 것을 모으는 도구로 바꾼다. 이 장의 셋째 절이 그 자리이고, 12장 벡터 색인의 디딤돌이다.

이 장에서 세우는 것

7.1 블룸 필터. 거짓 양성률 정리와 최적 해시 수를 세우고, 키 백만 개에 1.2 MB 로 오답 1 퍼센트를 잰다

7.2 카운트 민 스케치. 오차 상한 정리를 증명하고, 1 MB 로 천만 개 스트림의 상위 100 개를 찾는다. 하이퍼로그로그도 함께

7.3 MinHash 와 띠 나누기. 일치 확률이 자카드임을 증명하고 S 곡선을 실측한다

7.4 세 구조를 이어 학습 데이터 중복 제거 파이프라인을 세우고 잰다

7.1 블룸 필터. 없음은 안 틀리는 집합

6.1 의 해시 테이블은 키를 저장한다. 그래서 있는지 물으면 정확히 답하고, 키 하나에 키 크기만큼의 메모리가 든다. 키 백만 개가 8 바이트씩이면 8 MB 이고, 6.1 의 적재율을 지키면 그 두 배다. 백억 개면 160 GB 다. 이 장의 첫 구조는 그 표에서 키를 빼 버린다. 남는 것은 비트 배열 하나이고, 있는지 물으면 없다는 답만 정확하다.

없다는 답만 정확한 집합이 쓸모 있으려면 있다는 답이 틀리는 확률이 작아야 하고, 그 확률을 비트 수로 살 수 있어야 한다. 이 절은 그 교환 비율을 정리로 세우고, 키 백만 개로 잰다.

학습 목표

> 블룸 필터를 비트 배열과 해시 k 개로 정의하고, 없는 키를 있다고 답할 확률을 비트 수와 키 수와 해시 수의 식으로 적을 수 있다. 목표 오답률에 필요한 키당 비트를 계산하고, 그 값이 왜 키 크기와 무관한지 설명할 수 있다.

이 말부터

> 거짓 양성 없는 키를 있다고 답하는 것. 블룸 필터가 틀리는 유일한 방향이다
> 거짓 음성 있는 키를 없다고 답하는 것. 블룸 필터에는 없다
> 키당 비트 비트 수를 키 수로 나눈 것. m/n 으로 적고, 오답률은 이 숫자와 해시 수로만 정해진다

디딤돌

정의 7.1: 블룸 필터

> 비트 m 개의 배열 B 와 키를 $\{0, \dots, m-1\}$ 로 보내는 해시 $h_1, \dots, h_k$ 의 짝이다. 처음에는 전부 0 이다. 키 x 를 넣을 때 $B[h_i(x)] \leftarrow 1$ 을 $i = 1, \dots, k$ 에 대해 한다. 키 y 가 있는지 물으면 $B[h_i(y)]$ 가 전부 1 일 때 있다고 답한다.
>
> **불변식** 넣은 키의 자리 k 개는 전부 1 이다. 그래서 있는 키를 없다고 답하는 일은 없다.

불변식이 말하는 것은 한 방향뿐이다. 넣은 키는 언제나 있다고 나온다. 안 넣은 키가 어떻게 나오는지는 불변식이 말하지 않고, 그것이 정리의 몫이다. 안 넣은 키 y 의 자리 k 개가 다른 키들 때문에 우연히 전부 1 이면 있다고 답한다. 그 확률을 센다.

정리 7.2: 거짓 양성률

> 보편 해시 k 개로 키 n 개를 넣은 뒤, 안 넣은 키가 있다고 나올 확률은 근사적으로 $p \approx \left(1 - e^{-kn/m}\right)^k$ 이다.

증명. 자리 하나가 키 하나의 해시 하나에 안 걸릴 확률은 $1 - 1/m$ 이다. 해시가 kn 번 찍혔으므로 자리가 0 으로 남을 확률은 $(1 - 1/m)^{kn} \approx e^{-kn/m}$ 이고, 1 일 확률은 $1 - e^{-kn/m}$ 이다. 안 넣은 키의 자리 k 개가 전부 1 이어야 하므로 그 확률을 k 제곱한다. 자리들이 독립이라는 가정이 근사이고, Bloom 이 1970 년에 이 식을 세웠다 [확립] [24]. ■

식에는 m 과 n 이 비 m/n 으로만 들어간다. 키가 8 바이트든 1 킬로바이트든 오답률은 같다. 키를 저장하지 않으니 키 크기가 어디에도 없다. 남은 변수는 해시 수 k 인데, k 를 늘리면 물을 때 확인하는 자리가 늘어 유리하고, 넣을 때 1 로 만드는 자리도 늘어 불리하다. 최적이 있다.

명제 7.3: 최적 해시 수와 키당 비트

> m/n 이 정해졌을 때 p 를 가장 작게 하는 해시 수는 $k = (m/n)\ln 2$ 이고, 그때 $p = 2^{-k} \approx 0.6185^{m/n}$ 이다. 목표 오답률 p 에 필요한 키당 비트는 $m/n = -\log_2 p / \ln 2 \approx 1.44 \log_2(1/p)$ 다.

증명. $p = (1 - e^{-kn/m})^k$ 의 로그를 k 로 미분해 0 으로 두면 $e^{-kn/m} = 1/2$, 즉 $k = (m/n)\ln 2$ 가 나온다. 대입하면 $p = (1/2)^k$ 다. 둘째 식은 첫째 식을 m/n 에 대해 푼 것이다. ■

예제 7.1: 키 백만 개, 오답 1 퍼센트

> $p = 0.01$ 이면 키당 비트는 $1.44 \times \log_2 100 = 9.6$ 이고, 해시 수는 $9.6 \times 0.693 \approx 7$ 이다. 백만 개면 1.2 MB 다. 같은 키를 파이썬 `set` 에 넣으면 정수 객체와 표를 합쳐 60 MB 남짓이다. 쉰 배다. 대신 백 번에 한 번 없는 키를 있다고 답한다.

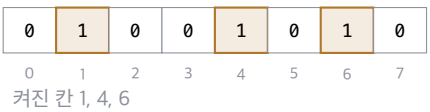


그림 7.1.. 비트 배열 여덟 칸의 블룸 필터. 키 하나가 해시 셋으로 세 칸을 켠다. 없는 키가 켜진 세 칸만 맞히면 오답이다

그림 7.2.. 키 둘을 넣은 블룸 필터. 물은 키의 자리 셋 중 하나가 0 이면 없다. 셋 다 1 이면 있다고 답하지만 우연일 수 있다

예제의 쉰 배는 명제 7.3 가 약속한 값이다. 약속대로 나오는지, 그리고 예측이 실측을 얼마나 따르는지 잰다.

균열

키 백만 개를 넣고 안 넣은 키 백만 개를 물어 있다고 나온 비율을 쟀다. 키당 비트를 4 에서 24 까지 바꾸고, 해시 수는 명제 7.3 의 값으로 두었다.

실험 7.1 키당 비트와 거짓 양성률. 예측과 실측

구조	키당 비트	해시 수	MB	오답 예측 퍼센트	오답 실측 퍼센트
해시 사전 (set)	0	0	61.6	0.00	0.00
블룸 4 비트	4	3	0.50	14.7	14.7
블룸 8 비트	8	6	1.00	2.16	2.14
블룸 12 비트	12	8	1.50	0.31	0.31
블룸 16 비트	16	11	2.00	0.0459	0.0441
블룸 24 비트	24	17	3.00	9.8e-04	7.0e-04

측정 조건. 무작위 40 비트 정수 키. 넣은 키 백만 개는 전부 있다고 나왔다. 해시 사전은 `set` 과 정수 객체의 메모리를 합친 것이다.

실측이 정리 7.2 를 소수점 둘째 자리까지 따른다. 키당 4 비트에서 15 퍼센트, 8 비트에서 2 퍼센트, 16 비트에서 만분의 4 다. 비트를 여덟 개 더할 때마다 오답이 서른 배 남짓 준다. 명제 7.3 의 $0.6185^8 = 0.022$ 와 같다.

메모리 열은 해시 사전의 61 MB 와 블룸 필터의 0.5 에서 3 MB 를 나란히 둔다. 오답을 백만분의 1 로 잡아도 스무 배다. 그런데 이 표에는 시간이 없다. 시간을 재면 정리에 없는 것이 하나 나온다.

관찰

> 해시 k 개는 자리 k 개이고, 자리 k 개는 서로 다른 라인 k 개다. 키당 16 비트의 해시 열한 개는 물음 하나에 무작위 라인 열한 개다. 6.1 의 해시 테이블은 라인 하나에서 둘이었다.

정리는 비트로 오답을 사는 비율을 말하고, 그 비트가 어디 놓이는지는 말하지 않는다. 이 책의 잣대로는 놓이는 자리가 값의 전부다. 그래서 다시 쓰기는 자리 k 개를 어디에 둘 것인가다.

다시 쓰기

자리 k 개를 배열 전체에 흩지 않고 라인 하나 안에 모은다. 첫 해시로 라인을 고르고, 나머지 해시로 그 라인 안의 비트를 고른다.

정의 7.4: 블록 블룸 필터

> 비트 배열을 B 크기의 블록으로 나눈다. 키 x 는 해시 h_0 으로 블록 하나를 고르고, 그 블록 안에서 $h_1, \dots, h_k$ 로 비트 k 개를 고른다. 넣기와 묻기가 블록 하나 안에서 끝난다.

명제 7.5: 블록 블룸 필터의 이동량과 오답률

> 넣기와 묻기의 이동량은 라인 하나다. 정의 7.1 의 k 라인에서 하나로 준다. 오답률은 정리 7.2 보다 높다. 블록에 든 키 수가 평균 $n/(\text{블록 수})$ 를 중심으로 흩어지고, 키가 많이 몰린 블록이 평균보다 오답을 많이 낸다. 같은 오답률을 지키려면 키당 비트를 더 준다.

증명. 이동량은 정의에서 바로 나온다. 오답률은 블록마다 정리 7.2 를 블록의 키 수로 적용한 것의 평균이고, 정리의 식은 키 수에 대해 볼록하므로 평균의 식보다 식의 평균이 크다. ■

이 명제가 0.1 의 첫째 가정을 블룸 필터에 적용한 결과다. 비트를 조금 더 쓰고 라인을 k 에서 하나로 줄인다. 비트 하나의 값은 메모리이고 라인 하나의 값은 지연이라, 이 책의 잣대로는 남는 장사다. 연습문제에서 그 비트 값을 직접 잰다.

알고리즘 7.1 블룸 필터에 넣기와 묻기 `dsai.bloom.BloomFilter`

> **입력:** 비트 배열 $B[0..m)$, 해시 $h_1, \dots, h_k$
> 1: **넣기**(x): $B[h_i(x)] \leftarrow 1$ 을 모든 i 에 대해
> 2: **묻기**(y): **반환** $\bigwedge_i B[h_i(y)]$

구현

해시 k 개를 따로 만들지 않고 둘로 만든다. $h_i(x) = h_1(x) + i \cdot h_2(x)$ 다. 해시 둘을 계산하고 덧셈으로 나머지를 얻는다.

지금 보는 코드 키 배열 전체의 자리 k 개를 한 번에 만든다. (n, k) 배열이 나온다.

구현 이중 해싱으로 자리 k 개 만들기 `samples/dsai/bloom.py:16`

```
16 def hashes(keys: np.ndarray, k: int, m: int) -> np.ndarray:
17     """키마다 자리 k 개. 해시 둘로 k 개를 만드는 이중 해싱. (n, k) 배열."""
18     h1 = mix64(keys.astype(np.uint64))
19     h2 = mix64(h1 ^ np.uint64(0x5851F42D4C957F2D)) | np.uint64(1)
20     i = np.arange(k, dtype=np.uint64)
21     return ((h1[:, None] + i[None, :] * h2[:, None]) % np.uint64(m)).astype(np.int64)
```

읽어 보면 h_2 를 홀수로 만들어 m 이 2 의 거듭제곱일 때 $i \cdot h_2$ 가 자리를 다 돈다. 해시 계산이 키마다 둘이라 k 가 열일곱이어도 비용이 안 는다.

구현 비트 배열과 넣기, 묻기 `samples/dsai/bloom.py:24`

```python
class BloomFilter:
    """정의 7.1. 비트 배열과 해시 k 개."""

    def __init__(self, m_bits: int, k: int):
        self.m = m_bits
        self.k = k
        self.bits = np.zeros(m_bits, dtype=bool)

    def add(self, keys: np.ndarray) -> None:
        self.bits[hashes(keys, self.k, self.m).ravel()] = True

    def contains(self, keys: np.ndarray) -> np.ndarray:
        """자리 k 개가 전부 1 인가. 없는 키도 우연히 전부 1 이면 있다고 답한다."""
        return self.bits[hashes(keys, self.k, self.m)].all(axis=1)

    def nbytes(self) -> int:
        return self.m // 8
```

읽어 보면 `contains` 의 `all(axis=1)` 이 알고리즘의 논리곱이다. 비트 배열을 `bool` 로 두어 자리 하나가 1 바이트다. 진짜 비트로 두면 여덟 배 작아지고 연습문제에서 한다.

구현 정리와 명제의 식 `samples/dsai/bloom.py:43`

```python
def false_positive_rate(m: int, n: int, k: int) -> float:
    """정리 7.2. 키 n 개를 넣은 뒤 없는 키가 있다고 나올 확률의 근사."""
    return (1.0 - math.exp(-k * n / m)) ** k


def optimal_k(m: int, n: int) -> int:
    """비트 수와 키 수가 정해졌을 때 오답을 가장 작게 하는 해시 개수. (m/n) ln 2."""
    return max(1, round(m / n * math.log(2)))


def bits_per_key_for(fpr: float) -> float:
    """목표 오답률에 필요한 키당 비트. -log2(fpr) / ln 2."""
    return -math.log2(fpr) / math.log(2)
```

분석

블룸 필터는 그대로 유효하다. 1970년의 정의와 정리가 손댈 데 없이 맞고, 실험이 소수점까지 따른다. 바뀐 것은 자리 k 개의 배치이고, 그것은 정리를 바꾸지 않고 명제 7.5 를 덧붙인다.

이 구조가 이 책의 작업량에서 서는 자리는 셋이다. 첫째, 이미 본 것을 기억하는 일. 문서 수십억 개의 해시를 표에 담을 수 없을 때 비트 열 개씩으로 담는다. 7.4 의 파이프라인이 그 자리다. 둘째, 없는 것을 빨리 거르는 일. 디스크나 원격에 있는 키를 찾기 전에 없다는 답을 메모리에서 얻는다. 셋째, 같은 것을 다시 계산하지 않는 일. 이미 계산한 입력의 해시를 기억하면 계산이 준다.

현장에서

> Bloom 의 1970 년 논문은 하이픈 넣기 사전을 예로 들었다. 단어의 90 퍼센트는 규칙으로 되고 나머지 10 퍼센트만 사전을 봐야 하는데, 사전이 디스크에 있었다. 블룸 필터가 없다고 답하면 디스크를 안 간다 [확립] [24]. 오답은 디스크를 한 번 더 가는 것이고, 정답은 디스크를 안 가는 것이라 오답의 값이 싸다.

주의

> 블룸 필터에서 키를 뺄 수 없다. 자리를 0 으로 되돌리면 그 자리를 나눠 쓰던 다른 키가 없어진다. 빼야 하면 비트 대신 계수기를 두거나 표를 다시 만든다.

빼기가 안 되는 이유가 다음 절의 출발이다. 비트를 계수기로 바꾸면 빼기가 되고, 그러면 있는지가 아니라 몇 개인지를 물을 수 있다. 그 표가 카운트 민 스케치다.

되짚기

> 블룸 필터를 정의하고 거짓 양성률 정리와 최적 해시 수를 세웠다. 키 백만 개에 1.2 MB 로 오답 1 퍼센트이고, 실측이 예측을 따른다. 다시 쓰기는 자리 k 개를 라인 하나에 모으는 것이다. 다음 절은 비트를 계수기로 바꿔 개수를 어림한다.

7.2 카운트 민 스케치. 개수를 어림하는 표

7.1 의 비트를 정수로 바꾸면 있는지 대신 몇 번 왔는지를 적을 수 있다. 그런데 자리를 나눠 쓰는 키들의 개수가 한 자리에 섞인다. 그 섞임을 그대로 두고, 대신 표를 여러 줄 두어 가장 덜 섞인 줄의 값을 답으로 삼는 것이 카운트 민 스케치다. 답은 실제보다 작을 수 없고, 얼마나 클 수 있는지를 정리가 준다.

이 책의 작업량에서 개수를 묻는 자리는 많다. 토큰의 빈도, 임베딩 행이 얼마나 자주 읽히는지, 어느 문서 조각이 몇 번 나왔는지. 정확한 표는 서로 다른 키 수만큼의 메모리가 들고, 스케치는 오차와 확률로 정한 상수만큼 든다.

학습 목표

카운트 민 스케치를 폭과 깊이로 정의하고, 추정이 실제 이상이라는 것과 오차 상한이 확률 $1 - \delta$ 로 성립함을 증명할 수 있다. 목표 오차와 확률에서 폭과 깊이를 계산하고, 서로 다른 키의 수를 어림하는 하이퍼로그로그가 왜 다른 원리인지 설명할 수 있다.

이 말부터

스케치	스트림 전체를 저장하지 않고 상수 크기의 표에 요약한 것. 물음에 근사로 답한다
헤비 히터	스트림에서 전체의 일정 비율 이상을 차지하는 키. 빈도 상위의 키들이다
서로 다른 키의 수	스트림에 나온 키의 종류 수. 카디널리티라고도 부른다

디딤돌

정의 7.6: 카운트 민 스케치

폭 w, 깊이 d 의 정수 표 $C[d][w]$ 와 보편 해시 $h_1, \dots, h_d$ 의 짝이다. 처음에는 전부 0 이다. 키 x 가 오면 $C[j][h_j(x)]$ 를 모든 줄 j 에서 1 올린다. 키 x 의 개수를 물으면 $\min_j C[j][h_j(x)]$ 를 답한다.

불변식 모든 줄 j 에서 $C[j][h_j(x)]$ 는 x 의 실제 개수 f_x 이상이다. 자리를 나눠 쓰는 다른 키가 더한 것만 섞이고 뺀 것은 없다.

불변식이 답의 한쪽 끝을 준다. 답은 f_x 보다 작을 수 없다. 다른 쪽 끝, 즉 얼마나 클 수 있는지가 정리다. 줄 하나만 보면 섞인 양은 그 자리를 나눠 쓴 키들의 개수 합이고, 줄이 여럿이면 가장 덜 섞인 줄을 고른다.

정리 7.7: 카운트 민 스케치의 오차

> 스트림의 길이를 N 이라 하자. $w = \lceil e/\varepsilon \rceil$, $d = \lceil \ln(1/\delta) \rceil$ 이면 모든 키 x 에 대해 추정 $\hat{f}_x$ 는 $f_x \le \hat{f}_x$ 이고, 확률 $1 - \delta$ 이상으로 $\hat{f}_x \le f_x + \varepsilon N$ 이다.

증명. 줄 j 에서 x 의 자리에 섞인 양 $X_j = \sum_{y \neq x,\ h_j(y)=h_j(x)} f_y$ 의 기댓값은 보편 해시로 $\sum_{y \neq x} f_y / w \le N/w = \varepsilon N/e$ 다. 마르코프 부등식으로 $\Pr[X_j > \varepsilon N] \le 1/e$ 이고, 줄들의 해시가 독립이므로 모든 줄에서 넘칠 확률은 $e^{-d} \le \delta$ 다. 최소를 취하면 한 줄만 안 넘쳐도 된다. Cormode 와 Muthukrishnan 이 2005 년에 세운 정리다 [확립] [25]. ■

정리에서 메모리를 읽는다. 표의 크기가 $wd = (e/\varepsilon)\ln(1/\delta)$ 이고 서로 다른 키의 수 U 가 어디에도 없다. 오차 εN 은 스트림 길이에 비례하므로, 이 표는 빈도가 εN 보다 훨씬 큰 키에서만 쓸모가 있다. 헤비 히터가 정확히 그 키들이다.

예제 7.2: 천만 개 스트림, 오차 만분의 1

> $\varepsilon = 10^{-4}$, $\delta = 0.01$ 이면 $w = 27{,}183$, $d = 5$ 이고 표는 135,915 칸, 8 바이트씩 1.1 MB 다. 스트림이 천만 개이면 오차 상한은 천 개다. 십만 번 나온 키는 1 퍼센트 안에서 맞고, 천 번 나온 키는 두 배로 나올 수 있다. 서로 다른 키가 백만 개인 정확한 표는 8 MB 다.

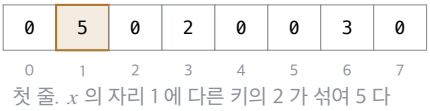


그림 7.3.. 폭 여덟, 깊이 둘의 스케치. 키 x 는 첫 줄에서 다른 키와 자리를 나눠 쓰고 둘째 줄에서는 혼자다. 최소가 둘째 줄이라 정확하다

예제의 두 배는 정리의 상한이지 보통의 값이 아니다. 보통은 얼마나 섞이는지, 그리고 헤비 히터를 얼마나 찾는지 잰다.

균열

지프 분포를 따르는 키 천만 개의 스트림을 서로 다른 키 백만 개에서 만들었다. 상위 100 개를 스케치로 고르고, 정확한 표의 상위 100 개와 얼마나 겹치는지 셌다.

실험 7.2 스케치의 크기와 헤비 히터 재현율

구조	폭	깊이	MB	상위 100 재현율	과대 추정 퍼센트
정확한 표	0	0	8.00	100	0.00
스케치 eps=0.001	2719	5	0.11	99.0	5.43
스케치 eps=0.0001	27183	5	1.09	100	0.27
스케치 eps=1e-05	271829	7	15.2	100	0.0037

측정 조건. 지프 지수 1.1. 과대 추정은 실제 상위 100 개의 키에서 (추정 − 실제) / 실제의 평균이다. 정확한 표는 `bincount` 다.

0.1 MB 의 스케치가 상위 100 개 중 99 개를 찾고, 1.1 MB 는 100 개를 다 찾는다. 정확한 표의 8 MB 에 비해 7 배와 70 배 작다. 과대 추정은 상위 100 개에서 5 퍼센트와 0.3 퍼센트다. 정리의 상한 εN 은 만 개와 천 개인데, 상위 키의 빈도가 십만 단위라 비율로는 작다.

관찰

스케치의 답은 빈도가 클수록 정확하다. 오차가 절대량 εN 으로 정해지기 때문이다. 드문 키의 개수를 물으면 답은 거의 전부 섞인 것이다.

이 관찰이 스케치를 쓸 자리와 안 쓸 자리를 가른다. 자주 오는 것을 찾는 데는 맞고, 드문 것을 세는 데는 안 맞는다. 드문 것이 몇 종류나 되는지는 다른 물음이고 다른 원리가 필요하다.

다시 쓰기

서로 다른 키의 수를 어림하는 데 스케치의 원리는 안 통한다. 키마다 자리를 하나씩 주면 표가 키 종류 수만큼 커진다. 대신 해시값의 모양을 본다. 해시가 고르면 앞자리에 0 이 r 개 잇달아 나오는 키가 2^r 개에 하나꼴이다. 본 것 중 가장 긴 0 의 열이 r 이면 키가 2^r 종류쯤 됐다는 뜻이다.

정의 7.8: 하이퍼로그로그

레지스터 $m = 2^p$ 개의 배열 R 이다. 키 x 의 해시 $h(x)$ 의 위 p 비트로 레지스터를 고르고, 나머지 비트에서 첫 1 까지의 0 의 개수에 1 을 더한 값 ρ 로 $R[i] \leftarrow \max(R[i], \rho)$ 를 한다. 서로 다른 키의 수는 $\alpha_m m^2 / \sum_i 2^{-R[i]}$ 로 어림한다. α_m 은 편향을 잡는 상수다.

명제 7.9: 하이퍼로그로그의 오차

서로 다른 키의 수가 레지스터 수보다 충분히 크면, 추정의 상대 표준 오차는 약 $1.04/\sqrt{m}$ 이다. 레지스터 하나가 5 비트면 $m = 4{,}096$ 에 2.5 KB 로 1.6 퍼센트, $m = 65{,}536$ 에 40 KB 로 0.4 퍼센트다.

증명은 조화 평균의 분포를 다루는 해석이고 Flajolet 등의 2007 년 논문에 있다 [확립] [26]. 이 절은 상수를 가져다 쓰고 실험으로 확인한다. 레지스터에 적히는 것은 개수가 아니라 개수에 로그를 두 번 취한 정도의 작은 정수라, 키 수십억 개도 5 비트 레지스터에 든다. 이름이 거기서 왔다.

실험 7.3 같은 스트림에서 서로 다른 키의 수

레지스터	KB	실제 서로 다른 키	추정	오차 퍼센트
256	0.26	561821	517746	7.85
4096	4.10	561821	578068	2.89
65536	65.5	561821	569127	1.30

측정 조건. 실험 7.2 의 스트림. 레지스터는 8 비트 정수로 두었다. 5 비트로 접으면 표의 KB 가 8 분의 5 로 준다.

레지스터 4,096 개로 2.9 퍼센트, 65,536 개로 1.3 퍼센트다. 명제의 1.6 과 0.4 퍼센트보다 두세 배 크다. 이 책의 구현이 편향 보정을 작은 수 구간에서만 하고 큰 수 구간을 안 하기 때문이고, 그 보정은 연습문제에 있다. 방향과 자릿수는 명제대로다.

알고리즘 7.2 스케치에 더하기와 묻기 `dsai.sketch.CountMinSketch`

입력: 표 $C[d][w]$, 해시 $h_1, \dots, h_d$
1: **더하기**(x): $C[j][h_j(x)] \leftarrow C[j][h_j(x)] + 1$ 을 모든 j 에 대해
2: **묻기**(x): **반환** $\min_j C[j][h_j(x)]$

구현

더하기는 키 하나씩이 아니라 배열로 받는다. 같은 자리에 여럿이 더할 때 6.4 의 자리 충돌이 나므로, 누적 더하기를 쓴다.

지금 보는 코드 키 배열의 자리를 줄마다 만들고, 줄마다 누적 더하기를 한다.

구현 줄마다 다른 해시로 자리 만들기 `samples/dsai/sketch.py:26`

```
    def slots(self, keys: np.ndarray) -> np.ndarray:
        h = mix64(keys.astype(np.uint64)[:, None] ^ self.salts[None, :])
        return (h % np.uint64(self.w)).astype(np.int64)      # (n, d)

    def add(self, keys: np.ndarray) -> None:
        """알고리즘 7.1. 행마다 자기 자리를 하나씩 올린다.
        같은 키가 여럿이면 여럿 올린다.
        """
        s = self.slots(keys)
        for r in range(self.d):
            np.add.at(self.table[r], s[:, r], 1)

    def estimate(self, keys: np.ndarray) -> np.ndarray:
        """d 값 중 최소. 남이 더한 것이 섞여 실제 이상이다."""
        s = self.slots(keys)
        rows = np.arange(self.d)
        return self.table[rows[None, :], s].min(axis=1)
```

읽어 보면 `np.add.at` 이 같은 자리에 온 키를 전부 더한다. 보통의 `+=` 는 마지막 하나만 남기고, 그것이 6.4 의 자리 충돌이다. `estimate` 의 `min(axis=1)` 이 줄 사이의 최소다.

구현 폭과 깊이를 오차와 확률에서 `samples/dsai/sketch.py:48`

```
def cms_width_for(eps: float) -> int:
    """정리 7.7. 오차를 전체 개수의 eps 배 안에 두려면 폭 e/eps."""
    return math.ceil(math.e / eps)


def cms_depth_for(delta: float) -> int:
    """그 오차를 넘길 확률을 delta 아래로 두려면 깊이 ln(1/delta)."""
    return math.ceil(math.log(1.0 / delta))
```

분석

카운트 민 스케치는 그대로 유효하다. 정의와 정리가 2005 년 그대로이고, 실험이 상한 안에 든다. 이 책의 잣대로 볼 것은 깊이 d 다. 줄 d 개의 자리는 서로 다른 라인이라 키 하나에 라인 d 개이고, 7.1 의 관찰과 같다. 줄 다섯이면 라인 다섯이다. 줄을 라인 하나에 모으는 변형이 7.1 의 블록 블룸 필터와 같은 꼴로 나온다.

이 구조가 서는 자리는 자주 오는 것을 골라내는 일이다. 임베딩 테이블의 행 수백만 개 중 자주 읽히는 행 수천 개를 가속기 메모리에 두려면 어느 행이 자주 읽히는지 알아야 하고, 그것을 정확한 표로 세면 표가 임베딩 테이블만큼 크다. 스케치는 1 MB 로 상위 100 개를 다 찾았다.

현장에서

> Cormode 와 Muthukrishnan 의 논문은 스케치를 세는 것 말고 두 가지에 더 쓴다. 두 스트림의 내적을 어림하는 것과, 범위 안의 개수를 묻는 것이다 [확립] [25]. 내적은 두 표의 같은 자리를 곱해 더한 것의 최소이고, 정리와 같은 꼴의 상한이 나온다.

주의

> 실험 7.2 의 지프 지수 1.1 은 상위 키가 매우 두드러지는 분포다. 지수가 1 에 가깝거나 분포가 고르면 상위 100 개의 빈도가 εN 에 가까워지고 재현율이 떨어진다. 스케치의 성적은 분포의 성적이다.

두 구조가 답한 물음은 있는지와 몇 개인지다. 둘 다 같은 키를 찾는다. 다음 절의 물음은 다르다. 비슷한 것이 있는지다. 같은 키를 찾는 해시로는 안 되고, 비슷한 것을 같은 자리로 보내는 해시가 필요하다.

되짚기

> 카운트 민 스케치를 정의하고 오차 정리를 증명했다. 1 MB 로 천만 개 스트림의 상위 100 개를 다 찾고, 과대 추정은 0.3 퍼센트다. 하이퍼로그로그는 해시의 앞자리 0 의 길이로 종류 수를 어림한다. 다음 절은 비슷한 것끼리 같은 자리에 오는 해시다.

7.3 MinHash 와 지역 민감 해싱. 비슷한 것을 같은 자리로

6.1 부터의 해시는 전부 같은 키를 같은 자리로 보내고 다른 키는 아무 데나 보냈다. 한 글자만 달라도 자리는 무관하다. 비슷한 문서를 찾는 데는 그 성질이 정반대로 필요하다. 비슷할수록 같은 자리에 올 확률이 높아야 한다. 그런 해시가 있다. 집합의 유사도를 정확히 확률로 옮기는 해시이고, 그 확률을 띠로 묶어 문턱으로 만드는 것이 지역 민감 해싱이다.

이 절이 세우는 것은 0.3 의 물음을 집합에 대해 푸는 도구다. 가장 가까운 것을 찾되, 전부와 견주지 않는다.

학습 목표

> 자카드 유사도와 MinHash 를 정의하고, 두 집합의 최소 해시가 같을 확률이 정확히 자카드 유사도임을 증명할 수 있다. 서명을 띠로 나눴을 때 후보가 될 확률의 식을 세우고, 띠 수와 줄 수로 문턱을 정할 수 있다.

이 말부터

자카드 유사도	두 집합의 교집합 크기를 합집합 크기로 나눈 것. 0 에서 1 사이다
서명	집합을 대신하는 고정 길이의 정수 열. 집합이 커도 서명은 k 개다
띠	서명을 r 개씩 묶은 조각. 띠마다 해시 테이블을 하나 두고, 어느 띠에서든 겹치면 후보다

디딤돌

정의 7.10: 자카드 유사도와 MinHash

> 집합 A, B 의 자카드 유사도는 $J(A,B) = |A \cap B| / |A \cup B|$ 다. 전체 원소 집합 위의 무작위 순열 π 에 대해 $\min_\pi(A) = \min_{a \in A} \pi(a)$ 를 A 의 MinHash 라 한다. 독립인 순열 k 개에 대한 MinHash 의 열이 A 의 서명이다.
>
> **불변식** 서명의 i 번째 값은 A 의 원소 중 π_i 가 가장 작은 원소의 π_i 값이다. A 에 원소를 더하면 값은 줄거나 그대로다.

정의의 순열은 원소 전체를 무작위로 줄 세우는 것이고, MinHash 는 그 줄에서 A 의 원소 중 맨 앞의 번호다. 두 집합이 같은 번호를 내려면 합집합에서 맨 앞에 선 원소가 둘 다에 들어야 한다. 그 확률이 정확히 자카드 유사도다.

정리 7.11: 최소 해시의 일치 확률

> 무작위 순열 π 에 대해 $\Pr[\min_\pi(A) = \min_\pi(B)] = J(A,B)$ 다.

증명. $A \cup B$ 의 원소 중 π 가 가장 작은 원소를 z 라 하자. 순열이 무작위이므로 z 는 $A \cup B$ 의 어느 원소든 같은 확률이다. $z \in A \cap B$ 이면 양쪽의 최소가 모두 z 라 같고, z 가 한쪽에만

있으면 그쪽의 최소는 z 이고 다른 쪽의 최소는 $\pi(z)$ 보다 크므로 다르다. 따라서 확률은 $|A \cap B|/|A \cup B|$ 다. Broder 가 1997 년에 문서 비교에 이 사실을 썼다 [확립] [27]. ■

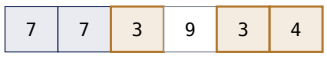
정리가 말하는 것은 서명 k 개 중 일치하는 자리의 비율이 자카드 유사도의 불편 추정량이라는 것이다. 자리 하나가 확률 J 의 베르누이 시행이다.

명제 7.12: 서명 길이와 오차

서명 길이 k 의 추정량 $\hat{J}$ 는 평균 J, 분산 $J(1-J)/k$ 다. 표준편차는 $0.5/\sqrt{k}$ 이하다. $k = 64$ 면 0.0625 이하, $k = 256$ 이면 0.031 이하다.

예제 7.3: 문서 둘, 서명 64 개

문자 세 개짜리 조각 500 개씩인 문서 둘의 자카드 유사도가 0.8 이면, 서명 64 개 중 평균 51 개가 같고 표준편차는 $\sqrt{0.16/64} = 0.05$ 다. 집합 둘을 견주려면 500 개씩 넣은 해시 테이블이 필요한데, 서명은 512 바이트 둘이다.



A 와 *B* 의 서명 셋을 번갈아 놓았다. 첫 순열은 같고, 둘째와 셋째는 다르다. 추정 1/3

그림 7.4.. 집합 둘과 순열 셋. 순열마다 합집합의 최소 원소가 교집합에 들면 서명이 같다

서명은 두 집합을 빠르게 견주게 하지만, 문서 백만 개 중 비슷한 쌍을 찾으려면 여전히 쌍 5×10^{11} 개를 봐야 한다. 쌍을 안 보고 후보를 얻는 장치가 띠다.

정의 7.13: 띠 나누기

서명 $k = br$ 개를 길이 r 의 띠 b 개로 나눈다. 띠마다 해시 테이블을 두고, 띠 i 의 값 r 개를 하나의 키로 해시해 문서를 넣는다. 두 문서가 어느 띠에서든 같은 자리에 오면 후보 쌍이다.

띠 하나가 같으려면 값 r 개가 전부 같아야 하므로 비슷하지 않은 쌍은 걸리기 어렵고, 띠가 b 개라 비슷한 쌍은 어디선가 한 번은 걸린다. 그 두 힘이 문턱 하나를 만든다.

명제 7.14: 후보가 될 확률

자카드 유사도 J 인 두 문서가 후보가 될 확률은 $1 - (1 - J^r)^b$ 다. 이 함수는 J 에 대해 S 자 모양이고, 문턱은 대략 $(1/b)^{1/r}$ 이다.

증명. 띠 하나의 값 r 개가 전부 같을 확률은 정리 7.11 와 독립성으로 J^r 이다. 띠 b 개에서 하나도 안 같을 확률이 $(1 - J^r)^b$ 다. 문턱은 확률이 1/2 근처가 되는 J 이고, $J^r b \approx 1$ 에서 나온다. Indyk 와 Motwani 가 1998 년에 이 증폭을 일반화해 근사 최근접 탐색의 틀을 세웠다 [확립] [28]. ■

명제의 S 자가 이 절의 전부다. $b = 16$, $r = 4$ 면 문턱이 0.5 이고, 유사도 0.8 인 쌍은 확률 $1 - (1 - 0.41)^{16} = 0.9998$ 로 후보가 되고 0.3 인 쌍은 0.12 다. 띠와 줄을 바꾸면 문턱이 옮겨진다. 어림이 맞는지 잰다.

균열

두 실험이다. 자카드 유사도를 알고 만든 집합 쌍에서 서명 길이에 따른 추정 오차를 재고, 띠 16 개 줄 4 개로 유사도별 후보 비율을 잰다.

실험 7.4 서명 길이와 추정 오차

서명 길이 k	서명 바이트	평균 절대 오차	표준편차 상한
16	128	0.10	0.12
64	512	0.0464	0.0625
256	2048	0.0231	0.0312

측정 조건. 집합 크기 500, 자카드 0.2 에서 0.9 사이의 쌍 300 개. 오차는 $|\hat{J} - J|$ 의 평균이다.

평균 절대 오차가 명제 7.12 의 표준편차 상한 안에 든다. 길이를 네 배로 하면 오차가 반이 된다. 제곱근이다. 서명 64 개, 512 바이트로 오차 0.05 다. 집합 하나가 원소 500 개에 4 KB 인 것에 비해 여덟 배 작고, 견주기는 정수 64 개의 비교다.

실험 7.5 띠 16 개 줄 4 개의 S 곡선

자카드	후보가 된 비율 퍼센트	명제 예측 퍼센트
0.30	13.0	12.2
0.50	66.0	64.4
0.70	100	98.8
0.80	100	100.0
0.90	100	100.0

측정 조건. 유사도마다 쌍 200 개. 한쪽을 넣고 다른 쪽으로 물어 후보에 들었는지 셌다.

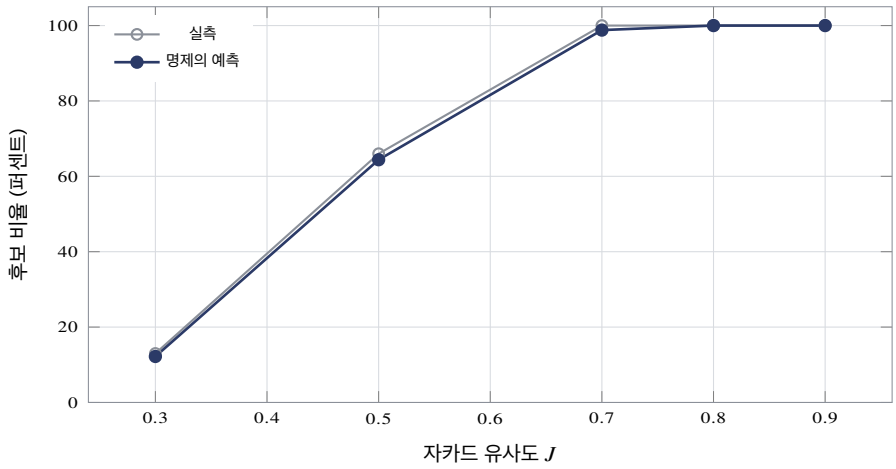


그림 7.5.. 명제의 S 곡선과 실측. 문턱 0.5 를 지나며 후보 비율이 급히 오른다

실측이 명제 7.14 를 따른다. 0.3 에서 13 퍼센트, 0.5 에서 66 퍼센트, 0.7 이상에서 100 퍼센트다. 0.5 에서 66 퍼센트는 문턱이 정말 0.5 근처라는 뜻이고, 0.3 의 13 퍼센트는 비슷하지 않은 쌍 여덟 중 하나가 후보로 들어와 확인을 받는다는 뜻이다.

관찰

띠는 놓치는 것과 헛것을 맞바꾼다. 문턱 아래의 후보는 자카드를 직접 계산해 거르면 되므로 값이 계산 하나이고, 문턱 위의 놓침은 못 찾은 채 끝나므로 값이 다르다. 그래서 문턱을 목표 유사도보다 낮게 잡는다.

다시 쓰기

이 절의 다시 쓰기는 지역 민감 해싱을 0.3 의 kNN 문제에 갖다 대는 것이다. 벡터의 코사인 유사도에도 같은 꼴의 해시가 있다.

명제 7.15: 무작위 초평면 해시

단위 벡터 u, v 와 무작위 방향 r 에 대해 $\Pr[\text{sign}(r \cdot u) = \text{sign}(r \cdot v)] = 1 - \theta(u, v)/\pi$ 다. θ 는 두 벡터 사이의 각이다. 부호 k 개가 서명이고, 띠 나누기는 정의 7.13 그대로다.

증명. r 이 두 벡터를 가르는 초평면일 확률은 두 벡터가 이루는 각을 π 로 나눈 것이다. 나머지는 정리 7.11 와 같은 논리다. ■

이 명제가 0.3 의 재현율을 다시 세운다. 전부와 견주지 않고 같은 자리의 후보만 견주며, 재현율은 명제 7.14 의 S 곡선이 준다. 재현율 0.99 를 원하면 띠를 늘리고, 후보 수가 부담이면 줄을 늘린다. 둘 다 서명 길이에 든다. 12장의 벡터 색인은 이 해시를 그래프로 바꾼 것이고, 이 절의 S 곡선이 그때 재현율의 첫 잣대가 된다.

알고리즘 7.3 띠 색인에 넣기와 후보 얻기 `dsai.minhash.LSHIndex`

입력: 서명 $s[0..br)$, 띠 테이블 $T_0, \dots, T_{b-1}$
1: **넣기**(id, s): 띠 i 마다 $T_i[\mathrm{hash}(s[ir..(i+1)r))]$ 에 id 를 붙인다
2: **후보**(s): **반환** $\bigcup_i T_i[\mathrm{hash}(s[ir..(i+1)r))]$

구현

서명은 원소 집합과 해시 k 개의 $(|S|, k)$ 표에서 열마다 최소를 취한다. 순열 대신 6.1 의 해시에 소금 k 개를 섞어 쓴다.

지금 보는 코드 집합 하나의 서명 k 개를 한 번에 만든다. 최소가 열마다 하나씩이다.

구현 MinHash 서명과 자카드 추정 `samples/dsai/minhash.py:35`

```python
def minhash(shingle_set: set[int], k: int, seed: int = 0) -> np.ndarray:
    """정의 7.10. 해시 k 개 각각의 최소값. (k,) 배열."""
    items = np.fromiter(shingle_set, dtype=np.uint64, count=len(shingle_set))
    salts = np.arange(k, dtype=np.uint64) + np.uint64(seed + 1)
    salts = salts * np.uint64(0x9E3779B97F4A7C15)
    h = mix64(items[:, None] ^ salts[None, :])            # (|S|, k)
    return h.min(axis=0)


def estimate_jaccard(sig_a: np.ndarray, sig_b: np.ndarray) -> float:
    """정리 7.11. 서명이 같은 자리의 비율이 자카드의 추정치다."""
    return float(np.mean(sig_a == sig_b))
```

읽어 보면 `h.min(axis=0)` 이 정의의 $\min_\pi$ 다. 순열 대신 해시를 쓰는 것은 근사이고, 해시가 고르면 정리의 확률이 그대로 나온다. 실험이 그것을 확인했다.

구현 띠마다 해시 테이블 `samples/dsai/minhash.py:65`

```python
    def insert(self, doc_id: int, sig: np.ndarray) -> None:
        for t, key in zip(self.tables, self.band_keys(sig)):
            t.setdefault(key, []).append(doc_id)

    def candidates(self, sig: np.ndarray) -> set[int]:
        """어느 띠에서든 같은 자리에 온 문서가 후보다."""
        out: set[int] = set()
        for t, key in zip(self.tables, self.band_keys(sig)):
            out.update(t.get(key, []))
        return out
```

읽어 보면 `candidates` 가 띠 b 개의 `dict` 조회 b 번이다. 후보는 집합이라 여러 띠에서 만난 문서가 한 번만 나온다. 명제의 S 곡선은 같은 파일의 마지막 함수다.

분석

MinHash 와 지역 민감 해싱은 그대로 유효하다. 1997 년의 정리와 1998 년의 틀이 그대로이고, 실험이 두 명제를 따른다. 바뀐 것은 자리다. 이 두 논문의 무대는 웹 문서와

고차원 점이었고, 지금은 학습 말뭉치의 문서 수십억 개와 0.3 의 벡터 수십억 개다. 규모가 바뀌었고 원리는 안 바뀌었다.

이 책의 잣대로 볼 것은 두 가지다. 서명 k 개를 만드는 데 집합 크기 곱하기 k 의 해시가 들고, 그것이 0.2 의 연산 밀도가 높은 일이라 가속기에 맞는다. 띠 조회 b 번은 라인 b 개이고, 그것은 7.1 과 7.2 의 k 와 d 와 같은 값이다.

현장에서

Broder 의 원 논문은 웹 문서 3 천만 개에서 거의 같은 문서를 찾는 것이 목적이었고, 조각을 원소로 하는 집합과 서명이 거기서 나왔다 [확립] [27]. 같은 방법이 25 년 뒤 언어 모델의 학습 데이터에서 거의 같은 문서를 걸러 내는 데 쓰이고, 그것이 다음 절이다.

주의

정리 7.11 는 진짜 무작위 순열을 가정한다. 해시로 순열을 흉내 내면 확률이 정확히 J 가 아니라 근사이고, 해시가 나쁘면 크게 어긋난다. 실험이 명제를 따른 것은 6.1 의 해시가 고르기 때문이다.

세 구조가 있음, 개수, 비슷함에 답했다. 셋 다 확률로 답하고 셋 다 키를 저장하지 않는다. 다음 절은 셋을 한 파이프라인에 놓는다. 학습 데이터의 중복을 걸러 내는 일이고, 이 장에서 유일하게 결과가 모델의 성적에 닿는 자리다.

되짚기

MinHash 를 정의하고 일치 확률이 자카드 유사도임을 증명했다. 띠 나누기의 S 곡선이 실측과 맞고, 서명 512 바이트로 오차 0.05 다. 같은 꼴이 벡터의 각에도 있어 12 장의 디딤돌이 된다. 다음 절은 이 장의 세 구조로 중복 제거 파이프라인을 세운다.

7.4 학습 데이터 중복 제거 파이프라인

언어 모델의 학습 말뭉치에는 같은 문서가 여러 번 들어 있다. 같은 글이 여러 주소에 실리고, 같은 안내문이 페이지마다 붙고, 같은 문장이 조금씩 바뀌어 퍼진다. Lee 등은 2022 년에 널리 쓰이는 말뭉치들에서 중복을 걸러 내고 학습하면 모델이 외운 문장을 뱉는 빈도가 열 배 줄고 같은 성적에 이르는 학습이 짧아진다고 보고했다 [확립] [29]. 중복을 거르는 일이 자료 구조의 일이고, 이 장의 세 구조가 그 일의 부품이다.

이 절은 세 구조를 한 파이프라인으로 세운다. 정확히 같은 것은 해시와 블룸 필터로, 거의 같은 것은 서명과 띠로 거른다. 파이프라인의 값을 문서 하나당 이동량과 메모리로 적고, 문서 만 개에 중복 이천 개를 심어 얼마나 걸러 내는지 잰다.

학습 목표

정확 중복과 근사 중복을 정의하고, 두 종류를 거르는 파이프라인의 각 단계에 어느 구조가 왜 들어가는지 말할 수 있다. 문서 하나당 비용과 전체 메모리를 문서 수와 서명 길이의 식으로 적고, 걸러 낸 비율과 헛후보 수를 실측으로 말할 수 있다.

이 말부터

정확 중복	정규화한 뒤 바이트까지 같은 문서. 해시 하나가 같으면 중복이다
근사 중복	조각 집합의 자카드 유사도가 문턱 이상인 문서. 몇 단어가 다른 같은 글이다
조각	문서를 집합으로 바꿀 때의 원소. 이 절은 단어 두 개짜리를 쓴다

디딤돌

정의 7.16: 정확 중복과 근사 중복

문서 x 의 정규화를 $\nu(x)$, 조각 집합을 $S(x)$ 라 하자. $\nu(x) = \nu(y)$ 이면 정확 중복이고, $J(S(x), S(y)) \geq \tau$ 이면 문턱 τ 의 근사 중복이다. 정확 중복은 근사 중복에 포함된다. ν 가 같으면 S 가 같고 $J = 1$ 이다.

두 정의가 서로 다른 구조를 부르는 이유는 값이다. 정확 중복은 해시 하나의 비교라 문서당 상수이고, 근사 중복은 서명 k 개와 띠 b 개라 그 수십 배다. 정확 중복을 먼저 걸러 근사 단계의 입력을 줄이는 것이 파이프라인의 첫 원칙이다.

알고리즘 7.4 중복 제거 파이프라인 `dsai.dedup.DedupPipeline.offer`

입력: 문서 x, 블룸 필터 F, 띠 색인 L, 남긴 문서의 조각 집합 $\{S_c\}$

$e \leftarrow \mathrm{hash}(\nu(x))$
만약 F 가 e 를 있다고 답함 **이면 반환** 정확 중복
끝 만약
F 에 e 를 넣는다
$s \leftarrow$ MinHash 서명$(S(x))$; 후보 $\leftarrow L$.후보(s)
모든 $c \in$ 후보 **동안**
    만약 $J(S(x), S_c) \geq \tau$ **이면 반환** 근사 중복
    끝 만약
끝 반복
L 에 (x, s) 를 넣고 S_x 를 저장; **반환** 남김

알고리즘의 첫 단계에 블룸 필터가 있는 이유는 정확 해시를 표에 다 담을 수 없어서다. 문서 수십억 개의 해시 8 바이트씩은 수십 GB 이고, 7.1 의 키당 10 비트면 그 6 분의 1 이다. 블룸 필터가 틀리는 방향은 없는 것을 있다고 하는 것이라, 처음 보는 문서를 중복이라고 버리는 일이 백 번에 한 번 생긴다. 그 값은 문서 하나이고, 이 일에서는 싸다.

명제 7.17: 파이프라인의 비용

문서 x 의 조각 수를 $|S|$, 서명 길이를 $k = br$ 라 하자. 문서 하나의 연산은 정규화와 해시 $O(|x|)$, 서명 $O(|S|k)$, 띠 조회 b 번, 그리고 후보마다 자카드 계산 $O(|S|)$ 다. 이동량은 블룸 필터 라인 k_F 개, 띠 테이블 라인 b 개, 후보의 조각 집합 읽기다. 남긴 문서 n 개의 메모리는 블룸 필터 $10n$ 비트, 서명 $8kn$ 바이트, 그리고 확인용 조각 집합이다.

증명. 각 단계의 비용을 7.1 의 정리, 7.3 의 정의와 알고리즘에서 더한 것이다. ■

명제에서 가장 큰 항은 조각 집합이다. 문서 하나의 조각 집합이 문서만큼 크므로, 남긴 문서 전부의 조각 집합을 두면 말뭉치를 다시 저장하는 셈이다. 그래서 실제 파이프라인은 확인 단계를 서명끼리의 비교로 대신하고, 조각 집합을 안 둔다. 서명 비교는 7.3 의 명제대로 오차 $0.5/\sqrt{k}$ 의 근사이고, 그 오차를 k 로 산다.

예제 7.4: 문서 십억 개

서명 $k = 128$ 이면 문서당 1 KB, 십억 개에 1 TB 다. 블룸 필터는 1.2 GB 다. 조각 집합을 안 두면 메모리는 서명이 전부다. 띠 $b = 32$, $r = 4$ 면 문턱이 $(1/32)^{1/4} = 0.42$ 이고, 유사도 0.8 인 쌍을 놓칠 확률은 $(1 - 0.41)^{32} = 5 \times 10^{-8}$ 이다.

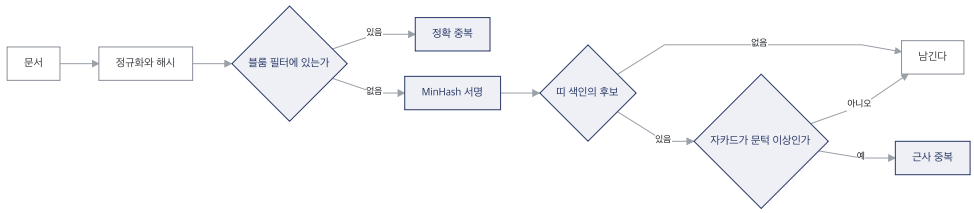


그림 7.6.. 파이프라인 세 단계. 정확 해시는 블룸 필터로, 서명은 띠 색인으로, 후보는 자카드로 확인한다

예제의 놓칠 확률은 명제 7.14 의 값이고, 헛후보의 수는 말뭉치의 유사도 분포에 달렸다. 심어 놓은 중복으로 둘 다 잰다.

균열

단어 5,000 개짜리 어휘에서 단어 60 개씩 무작위로 뽑은 문서 8,000 개를 만들고, 그중에서 골라 대소문자만 바꾼 정확 중복 1,000 개와 단어 셋을 바꾼 근사 중복 1,000 개를 섞어 무작위 순서로 넣었다.

실험 7.6 심어 놓은 중복 이천 개를 얼마나 걸러 내는가

항목	값
문서 수	10000
심은 정확 중복	1000
심은 근사 중복	1000
걸러 낸 정확 중복	1000
걸러 낸 근사 중복	967
남은 문서	8033
후보 비교 횟수	1016
걸러 낸 총수	1967
시간 ms	22379
블룸 필터 KB	20

측정 조건. 서명 64, 띠 16 줄 4, 문턱 0.7. 조각은 단어 두 개짜리. 후보 비교는 자카드를 직접 계산한 횟수다.

정확 중복 1,000 개는 전부 걸렀다. 블룸 필터의 거짓 양성으로 처음 보는 문서를 버린 일은 이번 실행에서 없었다. 근사 중복은 1,000 개 중 967 개다. 단어 셋을 바꾸면 조각 59 개 중 여섯까지 사라져 유사도가 0.8 안팎이고, 바뀐 단어가 붙어 있으면 덜 사라지고 떨어져 있으면 더 사라진다. 문턱 0.7 아래로 떨어진 것이 3 퍼센트다.

관찰

후보 비교가 1,016 번이다. 문서 만 개에서 쌍은 5 천만 개인데, 띠가 그중 천 개만 확인하게 했고 그 천 개의 대부분이 진짜 중복이었다. 헛후보가 거의 없는 것은 무작위 문서끼리의 유사도가 0 에 가깝기 때문이다.

시간 22 초는 문서당 2.2 밀리초이고, 그 대부분이 서명 계산이다. 조각 59 개에 해시 64 개씩이다. 명제 7.17 의 $|S|k$ 항이 전부다. 십억 개면 이 구현으로 26 일이고, 그것이 서명 계산을 가속기로 보내는 이유다.

다시 쓰기

파이프라인은 6.4 의 배치로 다시 쓴다. 문서를 하나씩 넣는 대신 서명 전부를 먼저 만들고, 띠마다 정렬해 같은 자리의 문서를 붙인다.

명제 7.18: 배치 중복 제거

> 문서 n 개의 서명을 (n, k) 배열로 만들고, 띠 i 마다 열 r 개의 해시로 정렬한다. 같은 해시가 붙어 있으므로 후보 쌍은 6.4 의 경계 찾기로 나온다. 띠 테이블이 없고, 이 동량은 띠마다 (n, r) 배열 정렬 한 번이다. 후보 쌍들의 연결 성분이 중복 무리이고, 무리마다 하나를 남긴다.

증명은 6.4 의 명제와 같다. 해시 테이블의 자리 조회 bn 번을 정렬 b 번으로 바꾼다. 이 꼴이 실제 파이프라인의 모양이다. 문서를 스트림으로 하나씩 받는 것이 아니라 말뭉치 전체를 갖고 시작하므로, 6.1 의 표보다 6.4 의 정렬이 맞다. 후보 쌍을 간선으로 보고 연결 성분을 찾는 것은 11 장의 그래프 표현 위의 일이고, 거기서 이 명제를 회수한다.

구현

파이프라인은 세 모듈을 부를 뿐이다. 새 자료 구조가 없다.

지금 보는 코드 정확 해시를 블룸 필터에 묻고, 서명으로 후보를 얻어 자카드로 확인한다.

구현 문서 하나를 넣는다 `samples/dsai/dedup.py:49`

```
    def offer(self, doc_id: int, text: str) -> bool:
        """문서를 넣는다. 남기면 True, 중복이면 False."""
        self.stats.seen += 1
        key = np.array([exact_key(text)], dtype=np.int64)
        if self.bloom.contains(key)[0]:
            self.stats.exact_dups += 1
            return False
        self.bloom.add(key)

        sh = word_shingles(normalize(text))
        sig = minhash(sh, self.k)
        cands = self.lsh.candidates(sig)
        self.stats.near_candidates += len(cands)
        for c in cands:
            if jaccard(sh, self.shingle_of[c]) >= self.threshold:
                self.stats.near_dups += 1
                return False
        self.lsh.insert(doc_id, sig)
        self.shingle_of[doc_id] = sh
        self.stats.kept.append(doc_id)
        return True
```

읽어 보면 `self.shingle_of` 가 명제 7.17 의 가장 큰 항이다. 실험에서는 만 개라 두었고, 십억 개면 서명 비교로 바꾼다. `stats` 가 실험의 표다.

구현 단어 조각 `samples/dsai/minhash.py:21`

```
def word_shingles(text: str, n: int = 2) -> set[int]:
    """단어 n 개짜리 조각의 집합. 문자 조각보다 우연한 겹침이 훨씬 적다."""
    w = text.split()
    if len(w) < n:
        return {hash(" ".join(w)) & 0xFFFFFFFFFFFF}
    return {hash(" ".join(w[i:i + n])) & 0xFFFFFFFFFFFF for i in range(len(w) - n + 1)}
```

읽어 보면 조각을 문자 셋이 아니라 단어 둘로 한 이유가 실험에 있다. 문자 조각은 짧은 문서끼리도 우연히 많이 겹쳐 헛후보가 문서당 수십 개였다. 단어 조각으로 바꾸자 천 개로 줄었다.

분석

이 파이프라인은 자료 구조로는 새것이 없다. 1970 년의 비트 배열, 1997 년의 서명, 1998 년의 띠를 2022 년의 말뭉치에 이은 것이다. 바뀐 것은 세 구조가 놓인 순서와 규모이고, 그 순서가 명제 7.17 의 큰 항을 뒤로 미룬다. 싼 것으로 먼저 거르고 비싼 것은 남은 것에만 쓴다.

이 순서가 이 책의 다른 파이프라인에도 있다. 0.3 의 kNN 은 12 장에서 거친 색인으로 후보를 얻고 정확한 거리로 확인한다. 같은 꼴이다. 확률적 구조는 답을 주는 것이 아니라 확인할 것을 줄여 준다. 그것이 이 장의 세 구조가 같은 자리에 서는 이유다.

현장에서

> Lee 등은 두 종류의 중복을 따로 걸렀다. 50 토큰 이상 정확히 같은 부분 문자열은 접미사 배열로, 거의 같은 문서는 MinHash 와 띠로 찾았다 [확립] [29]. 접미사 배열은 문서 안의 부분 문자열까지 보는 구조이고 14 장에서 세운다. 이 절의 파이프라인은 문서 단위만 본다.

주의

> 실험 7.4 의 문서는 무작위 단어라 서로 다른 문서의 유사도가 0 에 가깝다. 실제 말뭉치는 흔한 문구 때문에 유사도가 0.1 에서 0.3 인 쌍이 많고, 그 쌍들이 7.3 의 S 곡선 아래쪽에서 헛후보로 들어온다. 헛후보의 수는 말뭉치의 성질이고, 이 실험은 그것을 재지 않았다.

이 장의 네 절이 한 문장으로 모인다. 키를 저장하지 않고 해시의 자리와 모양만 남기면, 있음과 개수와 비슷함에 확률로 답할 수 있고, 그 확률이 정리로 적힌다. 정확한 답을 요구하던 0.2 의 셋째 가정이 이 장에서 처음으로 놓인다. 놓아서 얻은 것이 메모리 쉰 배이고, 잃은 것은 백 번에 한 번이다.

되짚기

> 세 구조로 중복 제거 파이프라인을 세우고, 문서 만 개에서 심은 중복 이천 개 중 1,967 개를 후보 비교 천 번으로 걸렀다. 배치로 다시 쓰면 띠 테이블이 정렬이 되고, 중복 무리는 11 장의 그래프에서 연결 성분으로 찾는다.

7장을 닫으며

이 장이 세운 구조는 넷이다. 비트 배열과 해시 k 개의 블룸 필터, 폭과 깊이의 카운트 민 스케치와 레지스터의 하이퍼로그로그, 서명과 띠의 지역 민감 해싱, 그리고 셋을 이은 중복 제거 파이프라인. 넷 다 키를 저장하지 않고, 넷 다 오차가 정리로 적힌다. 그리고 넷 다 원 논문의 정의가 손댈 데 없이 그대로다.

네 실험의 숫자를 적어 둔다. 키 백만 개를 키당 16 비트에 넣으면 오답이 만분의 4 이고 정리의 예측과 소수점 둘째 자리까지 같았다. 천만 개 스트림의 상위 100 개를 1.1 MB 의 스케치가 다 찾았고 과대 추정은 0.3 퍼센트였다. 서명 64 개로 자카드 유사도의 오차가 0.05 였고, 띠 16 줄 4 의 후보 비율이 S 곡선을 따랐다. 문서 만 개에 심은 중복 이천 개 중 1,967 개를 후보 비교 천 번으로 걸렀다.

정정 하나를 남긴다. 파이프라인의 조각을 처음에는 문자 세 개짜리로 두었다. 그러자 무작위 단어로 만든 문서끼리도 유사도가 0.3 을 넘어 헛후보가 87만 번 나왔고, 한 번 도는 데 67 초가 걸렸다. S 곡선이 틀린 것이 아니라 조각이 문서의 같음을 잘못 재고 있었다. 단어 두 개짜리로 바꾸자 후보가 천 번, 시간이 22 초가 됐다. 확률적 구조의 정리는 전부 집합이 무엇이냐 위에 서 있고, 집합을 잘못 잡으면 정리가 맞아도 답이 틀린다.

옛 구조의 판정

> 확률적 자료 구조는 그대로 유효하다. 정의와 정리와 상수가 원 논문 그대로이고, 실험이 소수점까지 따른다. 근거는 실험 7.1 과 7.3 이다. 블룸 필터의 오답률과 띠의 S 곡선이 예측과 겹친다. 바뀐 것은 규모와 자리다. 웹 문서 3 천만 개가 학습 말뭉치 수십억 개가 됐고, 해시 k 개의 자리를 라인 하나에 모으는 변형이 필요해졌다.

이 장의 요약

> 1. 블룸 필터는 없음에는 안 틀리고 있음에는 확률 $(1 - e^{-kn/m})^k$ 로 틀린다. 오답률은 키당 비트로만 정해진다 7.1
> 2. 카운트 민 스케치는 실제 이상을 답하고, 확률 $1 - \delta$ 로 εN 안에 든다. 빈도가 클수록 정확하다 7.2
> 3. 하이퍼로그로그는 해시의 앞자리 0 의 길이로 종류 수를 어림한다. 레지스터 하나가 5 비트다 7.2
> 4. MinHash 의 일치 확률은 정확히 자카드 유사도이고, 띠 b 줄 r 의 후보 확률은 $1 - (1 - J^r)^b$ 다 7.3
> 5. 세 구조를 싼 것부터 이으면 확인할 것이 5 천만 쌍에서 천 쌍으로 준다 7.4
>
> 다음 장에서 뒤집히는 것: 해시는 순서를 버리고 자리를 얻었다. 8 장의 탐색 트리는 순서를 지키는 대신 높이를 값으로 치르고, 학습된 인덱스가 그 높이를 모델 하나로 바꾼다.

필자의 판단

> 필자는 확률적 자료 구조를 해시 테이블 뒤에 붙이는 순서가 맞다고 본다. 셋 다 해시의 성질 하나씩을 쓰고, 그 성질은 6.1 의 보편 해시에서 나온다. 따로 떼어 고급 주제로 두면 학생은 세 구조를 세 번 외운다. 해시 다음에 두면 비트 배열, 계수기 배열, 최소값 배열로 한 번에 보인다.

열린 문제

하나, 명제 7.5 의 블록 블룸 필터가 같은 오답률에 비트를 얼마나 더 쓰는지 이 장은 재지 않았다. 연습문제에 두었다.
둘, 실험 7.2 의 하이퍼로그로그가 명제보다 두세 배 큰 오차를 낸 것은 큰 수 구간의 편향 보정을 안 한 탓으로 봤지만, 그 보정을 넣고 다시 재지 않았다.
셋, 실험 7.4 의 헛후보 수는 무작위 문서에서 잰 것이라 실제 말뭉치의 값이 아니다. 흔한 문구가 있는 말뭉치에서 S 곡선 아래쪽의 후보가 몇 배로 느는지 모른다.

연습문제

7.1 기초 키 천만 개를 오답률 0.1 퍼센트로 담는 블룸 필터의 비트 수와 해시 수를 명제 7.3 로 구하라. 같은 키를 8 바이트씩 개방 주소 표에 적재율 0.7 로 담을 때와 메모리를 견줘라.

7.2 기초 스트림 십억 개에서 빈도가 전체의 0.01 퍼센트 이상인 키를 확률 99.9 퍼센트로 두 배 안에서 세려면 정리 7.7 로 폭과 깊이를 구하라. 표는 몇 MB 인가.

7.3 기초 띠 20 줄 5 의 후보 확률을 명제 7.14 로 자카드 0.5, 0.7, 0.9 에서 구하라. 문턱 $(1/b)^{1/r}$ 은 얼마인가.

7.4 기초 문서 십억 개, 서명 128 개의 파이프라인에서 명제 7.17 의 서명 메모리와 블룸 필터 메모리를 구하라. 조각 집합을 두면 문서당 4 KB 로 얼마가 더 드는가.

7.5 응용 `dsai.bloom.BloomFilter` 의 `bool` 배열을 `uint8` 의 비트 여덟 개로 접어 메모리를 여덟 배 줄이고, 실험 7.1 의 오답률이 같은지 확인하라.

7.6 응용 `dsai.bloom.BloomFilter` 를 명제 7.5 의 블록 블룸 필터로 바꿔라. 블록 512 비트에 자리 k 개를 모으고, 키당 16 비트에서 오답률이 얼마나 오르는지 재라.

7.7 응용 `dsai.sketch.CountMinSketch` 에 보수적 갱신을 더하라. 더할 때 줄 전부를 올리지 않고 최소인 줄만 올린다. 실험 7.2 의 과대 추정이 얼마나 주는지 재라.

7.8 응용 `dsai.dedup.DedupPipeline` 의 확인 단계를 조각 집합 대신 서명 비교로 바꾸고, 실험 7.4 에서 걸러 낸 근사 중복 수가 어떻게 바뀌는지 재라.

7.9 탐구 명제 7.18 의 배치 중복 제거에서 후보 쌍의 연결 성분을 찾는 데 해시 테이블 대신 정렬만으로 되는지 논하라. 안 되면 무엇이 더 필요한가.

7.10 탐구 블룸 필터에서 키를 빼려면 비트를 4 비트 계수기로 바꾼다. 계수기가 넘칠 확률을 정리 7.2 의 방법으로 어림하고, 키 백만 개에서 넘치는 자리가 몇 개인지 추정하라.

더 읽을 것

[24] 블룸 필터의 원전. 두 쪽이고, 오답률의 식과 디스크를 안 가는 예가 다 있다. 7.1
[25] 카운트 민 스케치의 원전. 오차 정리의 증명과 내적, 범위 물음으로의 확장이 있다. 7.2
[26] 하이퍼로그로그의 원전. 상수 α_m 과 상대 오차 $1.04/\sqrt{m}$ 의 유도가 있다. 7.2
[27] MinHash 의 원전. 일치 확률 정리와 웹 문서 3 천만 개의 실험이 있다. 7.3
[28] 지역 민감 해싱의 틀을 세운 논문. 띠 나누기의 증폭이 여기서 일반화됐고, 12 장의 디딤돌이다. 7.3
[29] 학습 데이터 중복 제거가 모델에 미치는 영향. 접미사 배열과 MinHash 두 방법을 함께 썼다. 7.4

4부

트리

트리는 순서와 접두사와 최대 하나를 지키는 세 가지 약속이다. 4부는 그 약속의 값을 높이와 라인으로 다시 재고, 내부 노드가 모델이 되고 트라이 위에 캐시와 오토마톤이 오르고 힙이 선택에 자리를 내주는 것을 본다.

8장 탐색 트리에서 학습된 인덱스로 9장 트라이, 토크나이저, 접두사 캐시 10장 힙, 선택, 표본 추출

8장

탐색 트리에서 학습된 인덱스로

2018년의 한 논문은 B-트리의 내부 노드가 하는 일이 키를 위치로 보내는 함수이고, 함수라면 배울 수 있다고 썼다 [확립] [30]. 교과서에서 자료 구조의 모양은 사람이 정한다. 트리의 높이, 노드의 크기, 층의 수. 이 논문은 그 모양을 데이터에서 읽어 냈고, 메모리가 수십 배 줄었다고 보고했다. 이 장은 그 자리까지 가는 길을 트리의 높이 하나로 잇는다. 높이가 비용이고, 높이를 줄이는 세 방법이 이 장의 세 절이다.

이 장은 0.2 의 첫째 가정과 넷째 가정을 건드린다. 높이는 비교 횟수가 아니라 라인 수이고, 그래서 로그의 밑이 값이다. 그리고 구조의 모양이 데이터에서 나오는 첫 자리가 셋째 절이다. 6.3 에서 값이 학습됐다면 여기서는 모양이 학습된다.

이 장에서 세우는 것

8.1 이진 탐색 트리. 넣는 순서와 높이의 정리, 트립. 평균 깊이 21.7 이 라인 21.7 개다

8.2 B-트리. 높이 $\lceil \log_B n \rceil$ 과 차수의 선택. 키 천만 개가 블록 셋

8.3 학습된 인덱스. 오차 상한 조각 선형 모델. 균일한 키에서 B-트리의 6 분의 1, 쏠린 키에서 세 배

8.4 정렬된 배열에 다시 서기. 아이칭거 배치와 정렬된 질의 배치. 순서만으로 세 배

8.1 이진 탐색 트리와 균형. 높이가 곧 비용

6.1 부터 세 장은 순서를 버렸다. 해시는 키를 자리로 바꾸고, 자리는 키의 크기와 무관하다. 그래서 어느 키보다 큰 첫 키를 찾거나 구간의 키를 모두 꺼내는 일에는 해시가 답이 없다. 순서를 지키는 구조의 첫째가 이진 탐색 트리이고, 그 값은 높이 하나로 적힌다. 뿌리에서 잎까지의 걸음이 탐색의 비용이고, 4.1 의 잣대로는 걸음마다 라인 하나다.

이 절은 트리를 순서 불변식으로 정의하고, 높이가 넣는 순서에 어떻게 달렸는지를 정리로 세운 뒤, 무작위성으로 높이를 묶는 트립을 세운다. 다음 세 절은 이 높이를 줄이는 세 방향이다. 노드를 키우거나, 노드를 모델로 바꾸거나, 트리를 배열로 되돌리거나.

학습 목표

이진 탐색 트리를 순서 불변식으로 정의하고, 무작위 순서로 넣었을 때의 기대 깊이와 정렬 순서로 넣었을 때의 높이를 적을 수 있다. 트립의 힙 불변식이 왜 기대 높이를 로그로 묶는지 설명하고, 높이를 라인 수로 읽을 수 있다.

이 말부터

높이	뿌리에서 가장 먼 잎까지의 노드 수. 탐색이 최악에 들여다보는 노드 수다
깊이	뿌리에서 어느 노드까지의 노드 수. 평균 깊이가 성공 탐색의 평균 비용이다
회전	순서 불변식을 지키면서 부모와 자식을 바꾸는 한 걸음. 균형은 전부 회전으로 잡는다

디딤돌

정의 8.1: 이진 탐색 트리

노드마다 키 하나와 자식 둘을 둔 트리다. 이 책은 노드를 4.2 의 인덱스 링크 아레나로 둔다. 키 배열과 왼쪽, 오른쪽 자식 번호 배열이다.

불변식 모든 노드에서 왼쪽 부분 트리의 키는 그 노드의 키보다 작고, 오른쪽 부분 트리의 키는 크거나 같다.

탐색은 뿌리에서 시작해 키를 비교하고 한쪽으로 내려간다. 삽입은 탐색이 끝난 빈자리에 붙인다.

불변식이 말하는 것은 순서뿐이다. 모양은 말하지 않는다. 같은 키 집합이 사슬 모양도 되고 완전 균형도 되며, 어느 쪽이 되느냐는 넣는 순서가 정한다. 그래서 높이의 정리는 넣는 순서에 대한 정리다.

정리 8.2: 넣는 순서와 높이

> 서로 다른 키 n 개를 정렬 순서로 넣으면 높이는 n 이다. 무작위 순서로 넣으면 성공 탐색의 기대 깊이는 약 $2\ln n \approx 1.39\log_2 n$ 이고, 기대 높이는 약 $4.3\log_2 n$ 을 넘지 않는다. 완전 균형 트리의 높이 $\lceil \log_2(n+1) \rceil$ 이 하한이다.

증명. 정렬 순서는 매번 지금까지의 최대 키가 오므로 오른쪽 자식으로만 붙어 사슬이 된다. 무작위 순서에서 키 x 의 깊이는 x 의 조상 수이고, 키 y 가 x 의 조상일 조건은 x 와 y 사이의 키 중 y 가 가장 먼저 왔다는 것이다. 그 확률은 사이 거리 d 에 대해 $1/(d+1)$ 이므로, 조상 수의 기댓값은 조화급수 둘의 합으로 $2\ln n$ 근처다. 높이의 상한은 같은 논증을 최대값에 적용한 것이고 상수가 크다. 하한은 이진 트리에 노드 n 개를 담는 데 필요한 층 수다. ■

정리가 말하는 것은 두 극단이다. 무작위면 로그의 1.4 배이고, 정렬이면 선형이다. 이 책의 작업량은 정렬 순서에 가깝다. 시간순 로그, 증가하는 아이디, 정렬해 둔 키. 그래서 무작위에 기대는 대신 무작위를 만들어 넣는다.

정의 8.3: 트립

> 노드마다 무작위 우선순위를 하나 더 둔 이진 탐색 트리다. 키는 정의 8.1 의 순서 불변식을, 우선순위는 힙 불변식을 따른다.
>
> **불변식** 모든 노드의 우선순위는 두 자식의 우선순위보다 크다.
> 삽입은 잎에 붙인 뒤 부모보다 우선순위가 크면 회전으로 끌어올린다.

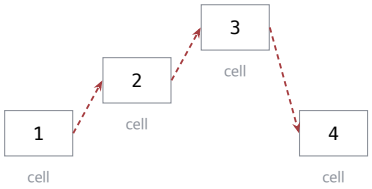
명제 8.4: 트립의 높이

> 우선순위가 무작위이면 트립의 모양은 키를 우선순위 순서로 넣은 이진 탐색 트리와 같다. 따라서 넣는 순서와 무관하게 정리 8.2 의 무작위 순서 값이 된다.

증명. 힙 불변식은 뿌리가 우선순위 최대인 노드임을 뜻하고, 두 부분 트리도 마찬가지다. 그것은 우선순위가 큰 순서로 넣었을 때의 모양과 같다. 우선순위가 무작위이므로 그 순서가 무작위다. ■

예제 8.1: 키 십만 개

> $n = 10^5$ 이면 완전 균형은 높이 17, 무작위 순서의 기대 깊이는 $1.39 \times 16.6 = 23$ 이다. 정렬 순서로 넣으면 높이 십만이고 탐색 하나가 라인 오만 개다.



정렬 순서. 오른쪽 자식만 있다

그림 8.1.. 같은 키 넷. 왼쪽은 정렬 순서로 넣은 사슬, 오른쪽은 무작위 순서의 트리. 높이가 넷과 셋이다

예제의 세 숫자가 실험에서 그대로 나오는지, 그리고 트립이 정렬 순서에서 무작위 순서의 값을 되찾는지 잰다.

균열

키 십만 개를 무작위 순서로, 만 개를 정렬 순서로 넣고 높이와 성공 탐색의 평균 노드 수를 쟀다. 같은 것을 트립으로 되풀이했다.

실험 8.1 넣는 순서와 높이. 이진 탐색 트리와 트립

넣는 방식	키 수	높이	평균 탐색 노드	2 log2 n
무작위 순서 BST	100000	44	21.7	33.2
정렬 순서 BST	10000	10000	4953	26.6
정렬 순서 트립	10000	28	16.0	26.6
무작위 순서 트립	100000	39	21.2	33.2
완전 균형 (하한)	100000	17	15.6	16.6

측정 조건. 무작위 40 비트 정수 키. 탐색은 들어 있는 키 2,000 개. 정렬 순서의 이진 탐색 트리는 사슬이 되므로 만 개만 넣었다.

무작위 순서의 평균 깊이 21.7 은 정리의 $2 \ln n = 23$ 근처이고, 높이 44 는 하한 17 의 2.6 배다. 정렬 순서는 높이가 키 수와 같다. 트립은 정렬 순서에서도 높이 28, 평균 16 을 냈고, 무작위 순서와 같은 값이다.

관찰

평균 깊이 21.7 은 노드 21.7 개이고, 노드가 아레나에 넣은 순서로 놓여 있으므로 라인 21.7 개다. 완전 균형이어도 17 개다. 6.1 의 해시 테이블은 라인 하나에서 둘이었다. 순서를 지키는 값이 라인 스무 개다.

이 관찰이 이 장의 문제다. 높이를 로그로 묶는 것은 균형이 해결했다. 로그 개의 라인을 줄이는 것은 균형이 해결하지 못한다. 로그의 밑을 키우거나, 로그를 없애거나, 트리를 없애야 한다.

다시 쓰기

이 절의 다시 쓰기는 정리의 높이를 이동량으로 다시 적는 것이다.

명제 8.5: 트리 탐색의 이동량

> 노드가 아레나에 넣은 순서로 놓인 이진 탐색 트리에서 성공 탐색 하나의 이동량은 깊이만큼의 라인이고, 전부 4.1 의 의존 읽기다. 지연 L 의 기계에서 시간은 깊이 곱하기 L 이다. 노드를 다시 놓아 같은 경로의 위쪽 노드들이 한 라인에 들게 하면 라인 수가 $\log_2$ 에서 $\log_B$ 로 준다. B 는 라인 하나에 드는 노드 수다.

증명. 자식 번호는 부모를 읽어야 알므로 의존 읽기다. 노드 하나가 라인 하나이면 깊이가 라인 수다. 위쪽 $\log_2 B$ 층의 노드 $B-1$ 개를 한 라인에 모으면 그 층들이 라인 하나가 되고, 그것을 되풀이하면 층 $\log_2 n$ 개가 라인 $\log_2 n / \log_2 B = \log_B n$ 개가 된다. ■

명제의 마지막 문장이 다음 절의 B-트리이고, 이 절 마지막의 트립을 배열로 되돌리는 것이 넷째 절이다. 둘 다 노드를 다시 놓는 일이다.

알고리즘 8.1 트립 삽입 `dsai.bst.Treap.insert`

> **입력:** 뿌리 r, 새 노드 i (키 k, 우선순위 p)
> 1: **만약** r 이 없음 **이면 반환** i
> 2: **끝 만약**
> 3: **만약** $k <$ 키[r] **이면** 왼쪽[r] ← 삽입(왼쪽[r], i); 우선순위[왼쪽[r]] $>$ p[r] 이면 r ← 오른쪽 회전(r)
> 4: **아니면** 오른쪽[r] ← 삽입(오른쪽[r], i); 우선순위[오른쪽[r]] $>$ p[r] 이면 r ← 왼쪽 회전(r)
> 5: **끝 만약**
> 6: **반환** r

구현

노드는 배열 넷이다. 키, 왼쪽, 오른쪽, 우선순위. 자식이 없으면 −1 이다.

지금 보는 코드 뿌리에서 키를 비교하며 내려가고, 들여다본 노드 수를 같이 센다.

구현 탐색과 높이 `samples/dsai/bst.py:43`

```
    def search(self, k: int) -> tuple[bool, int]:
        """찾았는지와 들여다본 노드 수. 노드 하나가 라인 하나다."""
        cur, probes = self.root, 0
        while cur != NIL:
            probes += 1
            if k == self.key[cur]:
                return True, probes
            cur = self.left[cur] if k < self.key[cur] else self.right[cur]
        return False, probes

    def height(self) -> int:
        """5장의 명시적 스택으로 잰다. 재귀 한계를 피한다."""
        if self.root == NIL:
            return 0
        best, stack = 0, [(self.root, 1)]
        while stack:
            i, h = stack.pop()
            best = max(best, h)
            for c in (self.left[i], self.right[i]):
                if c != NIL:
                    stack.append((c, h + 1))
        return best
```

읽어 보면 `search` 의 `probes` 가 깊이이고 실험의 평균 탐색 노드다. `height` 는 5.1 의 명시적 스택으로 잰다. 재귀로 재면 정렬 순서의 사슬에서 파이썬의 재귀 한계에 걸린다.

구현 트립. 붙인 뒤 우선순위로 끌어올린다 `samples/dsai/bst.py:76`

```
    def _insert_at(self, root: int, i: int) -> int:
        if root == NIL:
            return i
        k = self.key[i]
        if k < self.key[root]:
            self.left[root] = self._insert_at(self.left[root], i)
            if self.prio[self.left[root]] > self.prio[root]:
                root = self._rotate_right(root)
        else:
            self.right[root] = self._insert_at(self.right[root], i)
            if self.prio[self.right[root]] > self.prio[root]:
                root = self._rotate_left(root)
        return root

    def _rotate_right(self, y: int) -> int:
        x = self.left[y]
        self.left[y] = self.right[x]
        self.right[x] = y
        return x

    def _rotate_left(self, x: int) -> int:
        y = self.right[x]
        self.right[x] = self.left[y]
        self.left[y] = x
        return y
```

읽어 보면 회전 둘은 자식 번호 셋을 바꾸는 것이 전부다. 키를 옮기지 않는다. 힙 불변식이 깨진 곳에서만 회전하므로 삽입 하나에 회전이 기대값 둘이다.

분석

이진 탐색 트리는 조정해서 유효하다. 순서 불변식과 높이 정리는 그대로이고, 균형을 잡는 방법도 그대로다. 바뀐 것은 높이의 값이다. 교과서는 높이를 비교 횟수로 읽고, 이 책은 라인 수로 읽는다. 로그 개의 비교는 싸고 로그 개의 라인은 비싸다.

이 구조가 이 책의 작업량에 그대로 서는 자리는 드물다. 순서가 필요한 자리는 대부분 다음 두 절의 형태로 있다. 키가 정적이면 정렬된 배열 위의 B-트리이거나 학습된 인덱스이고, 키가 동적이면 12장의 스킵 리스트다. 트립이 남는 자리는 순서와 삽입이 함께 필요하고 크기가 라인 수십 개 안에 드는 곳이다.

현장에서

> 언어 표준 라이브러리의 정렬 맵은 대부분 균형 이진 탐색 트리다. 노드가 힙에 따로 놓이므로 명제 8.5 의 값을 그대로 치르고, 그래서 키 수백만 개의 정렬 맵은 같은 언어의 해시 맵보다 몇 배 느리다 [확립]. 필자는 이 차이를 학생에게 재게 한 뒤 트리를 가르친다.

주의

> 정리 8.2 의 무작위 순서 값은 키가 무작위 순서로 온다는 가정이고, 트립은 그 가정을 우선순위로 대신한다. 우선순위가 무작위가 아니면 (예를 들어 키의 해시이면) 특정 키 열에서 사슬이 될 수 있다.

높이를 로그로 묶는 것까지가 교과서이고, 로그 개의 라인을 줄이는 것부터가 이 장이다. 다음 절은 노드 하나를 라인 하나 또는 페이지 하나로 키운다.

되짚기

> 이진 탐색 트리를 순서 불변식으로 정의하고, 넣는 순서와 높이의 정리를 세웠다. 무작위 순서의 평균 깊이 21.7 이 정리의 23 근처였고, 트립이 정렬 순서에서 같은 값을 되찾았다. 높이는 라인 수다. 다음 절은 노드를 키워 로그의 밑을 키운다.

8.2 B-트리. 블록 단위로 탐색하기

8.1 의 명제는 위쪽 노드들을 한 라인에 모으면 라인 수가 $\log_B n$ 이 된다고 했다. 그것을 처음부터 구조로 만든 것이 B-트리다. 노드 하나에 키 B 개를 두고, 노드 하나가 블록 하나다. 1972 년의 블록은 디스크 페이지였고, 지금의 블록은 라인이거나 페이지이거나 가속기 메모리의 전송 단위다.

이 절은 정렬된 키 위의 정적 B-트리를 세운다. 삽입과 분할은 교과서에 있고, 이 책의 작업량에서 인덱스는 대부분 한 번 만들고 읽기만 하므로 정적 형태가 맞다. 차수를 바꾸며 높이와 블록 수와 내부 노드 메모리를 잰다.

학습 목표

B-트리를 차수와 블록으로 정의하고, 높이가 $\lceil \log_B n \rceil$ 임을 보일 수 있다. 차수를 블록 크기와 키 크기에서 정하고, 같은 키 수에서 차수가 높이와 메모리를 어떻게 맞바꾸는지 실측으로 말할 수 있다.

이 말부터

차수	노드 하나에 드는 키의 수. B 로 적고, 블록 크기를 키 크기로 나눈 값으로 정한다
잎	키 전부가 정렬되어 놓인 맨 아래 층. 이 절에서는 원래 배열 그대로다
내부 노드	잎 위의 층들. 자식 블록의 첫 키만 갖고 있어 잎보다 B 배씩 작다

디딤돌

정의 8.6: 정적 B-트리

정렬된 키 배열 $K[0..n)$ 을 잎으로 둔다. 잎을 크기 B 의 블록으로 나누고, 블록마다 첫 키를 모은 배열이 한 층 위의 노드들이다. 그 층도 다시 블록으로 나누어 되풀이하고, 한 블록에 들면 그것이 뿌리다. 탐색은 뿌리 블록에서 k 이하인 마지막 키를 찾아 그 번호의 자식 블록으로 내려간다.

불변식 층 ℓ 의 i 번째 키는 층 $\ell - 1$ 의 iB 번째 키와 같다. 어느 층에서든 k 이하인 마지막 키의 자식 블록에 k 의 자리가 있다.

불변식의 둘째 문장이 탐색의 정확성이다. 한 층에서 고른 블록 안에 답이 있으므로 층마다 블록 하나만 읽는다. 높이가 그대로 블록 수다. 그 높이를 세는 것이 다음 명제이고, 셈은 등비급수 하나다.

명제 8.7: B-트리의 높이와 메모리

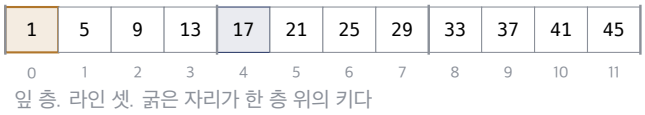
> 키 n 개, 차수 B 의 정적 B-트리는 높이가 $\lceil \log_B n \rceil$ 이고, 내부 노드의 키 수는 $n/B + n/B^2 + \cdots < n/(B-1)$ 이다. 탐색 하나의 이동량은 블록 $\lceil \log_B n \rceil$ 개이고, 블록 안의 비교는 $\log_2 B$ 번이다.

증명. 층이 하나 올라갈 때마다 키 수가 B 분의 1 이 되므로 n 이 B 이하가 되기까지 $\lceil \log_B n \rceil$ 층이다. 내부 노드의 키 수는 등비급수다. 블록 안은 정렬되어 있으므로 이진 탐색이고 $\log_2 B$ 번이다. ■

명제에서 읽을 것은 비교 횟수가 안 준다는 것이다. $\log_B n \times \log_2 B = \log_2 n$ 이다. 준 것은 블록 수뿐이고, 블록 수가 0.1 의 이동량이다. B-트리는 연산을 줄이는 구조가 아니라 이동을 줄이는 구조다.

예제 8.2: 키 천만 개

> 8 바이트 키, 64 바이트 라인이면 $B = 8$ 이고 높이는 $\lceil \log_8 10^7 \rceil = 8$ 이다. 4 KB 페이지면 $B = 512$ 이고 높이 3 이다. 이진 탐색 트리는 24 다. 라인 24 개가 8 개로, 페이지 24 개가 3 개로 준다. 비교는 셋 다 24 번이다.



잎 층. 라인 셋. 굵은 자리가 한 층 위의 키다

그림 8.2.. 차수 4 의 정적 B-트리. 잎 열두 개 위에 내부 노드 층 하나와 뿌리. 탐색은 블록 셋을 읽는다

예제의 24 에서 8 로, 3 으로 주는 것이 실험에서 그대로 나오는지, 그리고 블록이 줄 때 시간이 어떻게 따라오는지 잰다.

균열

정렬된 키 천만 개 위에 차수 2 에서 512 까지의 정적 B-트리를 세우고, 높이와 탐색당 블록 수와 내부 노드 메모리와 시간을 쟀다.

실험 8.2 차수와 높이, 블록 수, 내부 노드 메모리

차수	높이	예측 높이	탐색당 블록	내부 노드 MB	탐색당 us
2	24	24	24.0	80.0	137
8	8	8	8.00	11.4	50.4
16	6	6	6.00	5.33	36.1
64	4	4	4.00	1.27	25.7
512	3	3	3.00	0.16	21.3
0	24	24	24.0	0.00	6.58

측정 조건. 정렬된 40 비트 키 천만 개, 탐색 2,000 개의 평균. 차수 0 은 배열 전체의 NumPy 이진 탐색이고 블록 열은 라인 수가 아니라 비교 단계 수다.

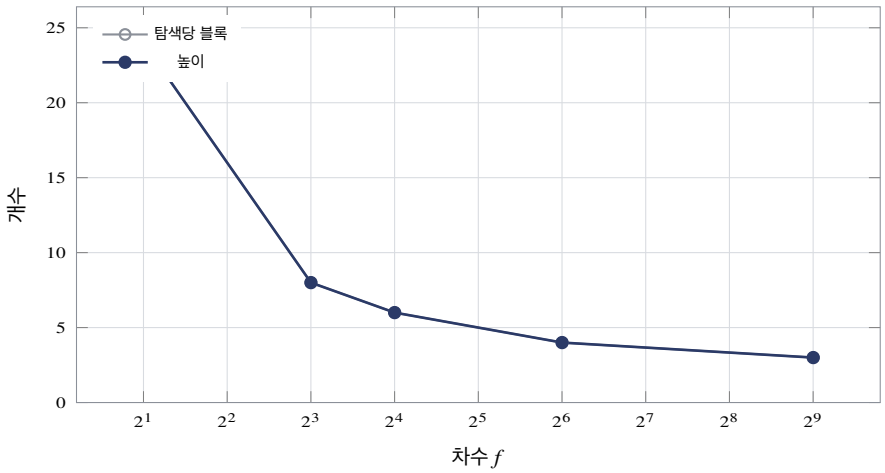


그림 8.3.. 차수에 따른 탐색당 블록 수. 차수가 오르면 높이가 내려가고, 블록 수가 높이를 따른다

높이가 명제와 같다. 차수 2 에서 24, 8 에서 8, 512 에서 3 이다. 탐색당 블록 수도 높이와 같고, 내부 노드 메모리는 차수 2 에서 80 MB, 64 에서 1.3 MB, 512 에서 0.16 MB 다. 시간은 차수 2 에서 137 마이크로초, 512 에서 21 마이크로초로 여섯 배 준다. 비교 횟수는 같고 블록 수만 줄었는데 시간이 줄었다.

관찰

마지막 줄의 NumPy 이진 탐색은 6.6 마이크로초로 차수 512 의 B-트리보다 세 배 빠르다. 파이썬 루프의 층마다 값이 C 루프 하나보다 크다. 이 실험의 여섯 배는 블록 수의 효과이고, 세 배는 언어의 효과다. 둘을 갈라 읽는다.

이 관찰의 뒷부분이 넷째 절의 출발이다. 정렬된 배열 하나 위의 이진 탐색이 층을 오르내리는 트리보다 빠를 수 있다면, 트리를 없애는 쪽도 봐야 한다. 그 전에 이 절의 다시 쓰기는 차수를 정하는 규칙이다.

다시 쓰기

차수는 취향이 아니라 블록 크기에서 나온다. 그리고 블록은 층마다 다를 수 있다.

명제 8.8: 차수의 선택

> 블록 크기 S, 키 크기 s 에서 차수는 $B = S/s$ 다. 블록보다 작은 노드는 같은 이동량에 층을 더 만들고, 블록보다 큰 노드는 층 하나에 이동량을 더 든다. 계층이 여럿이면 위쪽 층은 작은 블록 (라인) 에, 아래쪽 층은 큰 블록 (페이지) 에 맞춘다. 위쪽 층은 작아서 캐시에 남고, 아래쪽 층은 커서 한 번 읽는다.

증명. 노드가 블록의 절반이면 블록 하나를 읽고 절반만 쓰므로 층당 이동량은 같고 층 수는 $\log_{B/2} n$ 으로 는다. 노드가 블록 둘이면 층당 이동량이 둘이다. 위쪽 층의 크기는 n/B^{h-1} 이하라 캐시에 들고, 캐시에 든 층의 블록은 라인 단위로 읽힌다. ■

이 명제가 데이터베이스와 파일 시스템의 B-트리가 페이지 단위이고, 메모리 안의 정렬 배열 인덱스가 라인 단위인 이유다. 그리고 3.2 의 타일링과 같은 논리다. 접근의 단위를 저장의 단위에 맞춘다.

알고리즘 8.2 정적 B-트리 탐색 `dsai.btree.StaticBTree.search`

> **입력 :** 층 L_0 (뿌리) 부터 L_h (잎), 차수 B, 키 k
> 1: 블록 번호 $p \leftarrow 0$
> 2: **모든** 층 L_ℓ 을 위에서부터 **동안**
> 3: 　 블록 $L_\ell[pB..pB+B)$ 안에서 k 이하인 마지막 키의 번호 j
> 4: 　 $p \leftarrow pB + j$
> 5: **끝 반복**
> 6: **반환** p ▷ 잎에서 k 이하인 마지막 자리

구현

층은 배열의 목록이다. 층을 만드는 것은 B 칸 건너뛰기 슬라이스 하나다.

지금 보는 코드 　잎에서 B 칸마다 첫 키를 뽑아 위 층을 만든다. 한 블록에 들 때까지.

구현 층 만들기 `samples/dsai/btree.py:19`

```
19      def __init__(self, keys: np.ndarray, fanout: int):
20          assert np.all(keys[:-1] <= keys[1:]), "정렬된 키"
21          self.B = fanout
22          self.levels: list[np.ndarray] = [keys]        # levels[0] 이 잎
23          cur = keys
24          while len(cur) > fanout:
25              cur = cur[::fanout]                         # 블록마다 첫 키
26              self.levels.append(cur)
27          self.levels.reverse()                           # levels[0] 이 뿌리
```

읽어 보면 `keys[::fanout]` 이 2.3 의 뷰다. 위 층은 복사가 아니라 보폭 B 의 뷰이고, 그래서 내부 노드의 메모리가 실험의 표처럼 작다. 뷰가 아니라 복사로 두면 라인 안에 붙어 더 빠르지만 메모리가 든다.

구현 층마다 블록 하나 안의 이진 탐색 `samples/dsai/btree.py:32`

```
    def search(self, k: int) -> tuple[int, int]:
        """잎에서의 위치 (k 이상인 첫 자리) 와 들여다본 블록 수."""
        pos, blocks = 0, 0
        for lvl in self.levels:
            lo = pos * self.B
            hi = min(lo + self.B, len(lvl))
            blocks += 1
            j = int(np.searchsorted(lvl[lo:hi], k, side="right")) - 1  # 블록 안 탐색
            pos = lo + max(j, 0)
        return pos, blocks
```

분석

B-트리는 그대로 유효하다. 1972 년의 정의와 명제가 그대로이고, 블록의 정체만 바뀌었다. 디스크 페이지가 라인과 페이지와 전송 단위가 됐다. Bayer 와 McCreight 의 논문은 인덱스가 디스크에 있고 한 번의 접근이 비싸다는 가정에서 출발했고 [확립] [31], 그 가정이 0.2 의 첫째 가정과 같은 것이라 지금도 맞다.

이 구조가 이 책의 작업량에서 서는 자리는 정렬된 키 위의 조회 전부다. 정렬된 토큰 아이디, 정렬된 문서 번호, 12 장의 벡터 인덱스에서 클러스터 번호의 정렬 배열. 그 자리에서 B-트리의 내부 노드는 보통 배열 하나에 드는 작은 표이고, 다음 절은 그 표를 더 줄인다.

현장에서

> 파일 시스템과 데이터베이스의 B-트리는 지금도 페이지 단위이고, 4 KB 페이지에 8 바이트 키면 차수 500 안팎, 높이 셋에 십억 개를 담는다 [확립]. 가속기 위의 정렬 배열 탐색은 라인 단위의 정적 B-트리를 쓰고, 12 장의 디스크 위 벡터 인덱스는 다시 페이지 단위로 돌아간다.

주의

> 실험 8.2 의 시간은 파이썬 루프의 것이라 층 하나의 값이 실제보다 크다. C 로 쓰면 차수 512 의 B-트리가 배열 전체 이진 탐색보다 빠르다. 방향은 블록 열로 읽고 시간은 참고로 읽는다.

정적 B-트리의 내부 노드는 잎을 B 분의 1 로 줄인 표다. 다음 절은 그 표가 하는 일이 키에서 위치로의 함수임을 보고, 그 함수를 표 대신 모델로 둔다.

되짚기

> 정적 B-트리를 정의하고 높이가 $\lceil \log_B n \rceil$ 임을 보였다. 키 천만 개에서 차수 512 는 높이 3, 내부 노드 0.16 MB 이고 시간이 차수 2 의 6 분의 1 이었다. 차수는 블록 크기에서 나온다. 다음 절은 내부 노드를 누적 분포 모델로 바꾼다.

8.3 학습된 인덱스. 누적 분포를 모델로 바꾸다

8.2 의 내부 노드가 하는 일은 하나다. 키를 받아 잎의 블록 번호를 준다. 그것은 키에서 위치로 가는 함수이고, 정렬된 배열에서 키 k 의 위치는 k 이하인 키의 수, 즉 누적 분포 함수에 n 을 곱한 것이다. 함수를 표로 두는 대신 식으로 두면 표가 없어진다. 2018 년에 Kraska 등이 그 식을 신경망과 선형 모델로 두고 B-트리와 견주어 메모리가 수십 배 작고 탐색이 빠르다고 보고했다 [확립] [30].

이 절은 그 식을 가장 단순한 형태로 세운다. 조각 선형 모델이고, 조각마다 예측 오차가 상한 ε 안이라는 보장이 붙는다. 보장이 붙은 형태는 Ferragina 와 Vinciguerra 의 2020 년 논문에서 왔다 [확립] [32]. 조각 수가 데이터에 달렸다는 것이 이 절의 실험이다.

학습 목표

정렬된 배열의 위치를 누적 분포 함수로 적고, 오차 상한 ε 의 조각 선형 모델을 정의할 수 있다. 탐색이 예측 위치 근처 2ε 안의 이진 탐색으로 끝나는 이유를 보이고, 조각 수가 키 분포에 따라 어떻게 달라지는지 실측으로 말할 수 있다.

이 말부터

누적 분포	키 k 이하인 키의 비율. 정렬된 배열에서 위치를 n 으로 나눈 것과 같다
조각	키 축의 한 구간과 그 구간을 맡는 직선 하나. 시작 키, 기울기, 절편 셋으로 적는다
오차 상한	예측 위치와 실제 위치의 차이의 최대. ε 으로 적고, 탐색 창의 반이다

디딤돌

정의 8.9: 조각 선형 학습 인덱스

정렬된 키 $K[0..n)$ 위에 조각 (x_j, a_j, b_j) 의 열을 둔다. 키 k 에 대해 $x_j \le k < x_{j+1}$ 인 조각 j 를 찾고 $\hat{p} = a_j k + b_j$ 를 위치의 예측으로 삼는다.

불변식 모든 키 $K[i]$ 에 대해 $|\hat{p}(K[i]) - i| \le \varepsilon$ 이다.
탐색은 $[\hat{p} - \varepsilon, \hat{p} + \varepsilon]$ 안의 이진 탐색이다.

불변식이 탐색의 정확성이다. 실제 위치가 예측에서 ε 안에 있으므로 창 안의 이진 탐색이 반드시 찾는다. 모델이 얼마나 좋으냐는 정확성에 영향을 주지 않고 조각 수에만 영향을 준다. 나쁜 모델은 틀리는 것이 아니라 조각이 많은 것이다. 그러면 값은 조각 수로 적힌다. 탐색과 메모리 둘 다다.

명제 8.10: 학습된 인덱스의 탐색 비용과 메모리

> 조각 m 개의 인덱스에서 탐색 하나는 조각 찾기 $\lceil \log_2 m \rceil$ 번과 창 안 이진 탐색 $\lceil \log_2(2\varepsilon + 1) \rceil$ 번이다. 메모리는 조각당 24 바이트로 $24m$ 바이트다. 같은 키 위의 차수 B 인 B-트리 내부 노드는 $8n/(B-1)$ 바이트다. 조각 수 m 이 $n/(3B)$ 보다 작으면 학습된 인덱스가 작다.

증명. 조각의 시작 키는 정렬되어 있으므로 이진 탐색이다. 창의 크기가 $2\varepsilon + 1$ 이다. 메모리는 셈이다. ■

명제의 둘째 문장이 이 절의 물음이다. 조각 수 m 은 키가 얼마나 직선에 가깝게 놓였느냐에 달렸고, 그것은 데이터의 성질이다. 균일하게 흩어진 키는 조각이 적고, 몰려 있다가 성기다가 하는 키는 조각이 많다.

예제 8.3: 키 백만 개, 오차 32

> 균일한 키에서 조각이 천 개면 24 KB 이고, 차수 64 의 B-트리 내부 노드는 127 KB 다. 다섯 배 작다. 탐색은 조각 찾기 10 번에 창 안 7 번으로 17 번이고, B-트리는 층 넷에 층마다 6 번으로 24 번이다.

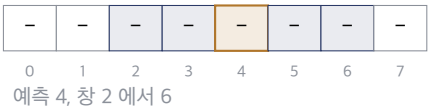


그림 8.4.. 학습된 인덱스의 예측과 오차 창. 모델이 자리 4 를 예측하고, 오차 상한 둘 안의 다섯 칸만 이진 탐색한다

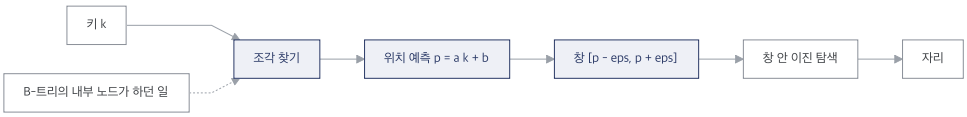


그림 8.5.. 학습된 인덱스의 탐색. 조각을 찾고, 직선으로 위치를 예측하고, 오차 창 안에서 이진 탐색한다

예제의 천 개는 가정이다. 조각 수가 분포에 따라 얼마나 달라지는지 두 분포로 잰다.

균열

균일 분포와 한쪽으로 쏠린 분포의 키 백만 개에서 오차 상한 8, 32, 128 의 인덱스를 세우고, 조각 수와 실측 최대 오차와 메모리를 차수 64 의 B-트리 내부 노드와 견줬다.

실험 8.3 오차 상한과 키 분포에 따른 조각 수와 메모리

분포	오차 상한	조각 또는 노드 수	실측 최대 오차	KB	탐색당 단계
균일	8	11101	8	266	6.00
균일	32	845	32	20.3	8.00
균일	128	56	128	1.34	10.0
균일	0	15625	0	127	4.00
쏠림	8	52973	8	1271	6.00
쏠림	32	14501	32	348	8.00
쏠림	128	3697	128	88.7	10.0
쏠림	0	15625	0	127	4.00

측정 조건. 키 백만 개. 오차 상한 0 의 줄은 차수 64 의 B-트리 내부 노드이고, 그때 조각 수 열은 잎 바로 위 층의 키 수다. 쏠린 분포는 지프 간격의 누적합이다.

실측 최대 오차가 상한과 같다. 불변식이 지켜졌다. 균일 분포에서 오차 32 는 조각 845 개, 20 KB 이고 B-트리의 127 KB 의 6 분의 1 이다. 오차 128 이면 조각 56 개, 1.3 KB 다. 쏠린 분포에서는 같은 오차 32 에 조각 14,501 개, 348 KB 로 B-트리의 2.7 배다. 같은 인덱스가 데이터에 따라 B-트리의 6 분의 1 도 되고 세 배도 된다.

관찰

B-트리의 메모리는 n 과 B 로만 정해지고 데이터를 안 본다. 학습된 인덱스의 메모리는 데이터가 정한다. 그래서 학습된 인덱스는 데이터를 보기 전에는 값을 말할 수 없고, 본 뒤에는 B-트리보다 훨씬 작을 수 있다.

이 관찰이 0.2 의 넷째 가정을 건드린다. 자료 구조의 모양이 사람의 설계가 아니라 데이터에서 나온다. 6.3 의 임베딩 테이블은 값이 학습됐고, 이 절의 인덱스는 구조가 학습됐다.

다시 쓰기

조각 찾기가 조각 배열의 이진 탐색이라는 것이 명제 8.10 의 첫 항이다. 조각이 수만 개면 그 탐색이 창 안 탐색보다 크다. 조각의 시작 키도 정렬된 키 배열이므로 같은 방법을 다시 쓴다.

명제 8.11: 재귀 학습 인덱스

조각 시작 키 $x_0, \dots, x_{m-1}$ 위에 오차 상한 ε 의 조각 선형 인덱스를 다시 세우면, 조각 찾기가 창 안 탐색 $\lceil \log_2(2\varepsilon+1) \rceil$ 번이 된다. 층 수는 $\log_c n$ 이고 c 는 층마다 줄어드는 비율 n/m 이다. 균일한 키에서 c 는 ε 에 비례해 크다.

증명. 위 층은 아래 층의 조각 시작 키를 잎으로 하는 정의 8.9 다. 층마다 키 수가 c 분의 1 이 되므로 명제 8.7 와 같은 논증이다. ■

이 명제가 학습된 인덱스를 B-트리와 같은 모양으로 만든다. 층이 있고 층마다 블록 (조각) 하나를 읽는다. 다른 것은 층 하나가 줄이는 비율이다. B-트리는 B 로 줄이고 학습된 인덱스는 데이터가 허락하는 만큼 줄인다. 8.2 의 명제에서 B 를 데이터가 정하는 c 로 바꾼 것이 학습된 인덱스다.

알고리즘 8.3 오차 상한 조각 만들기 `dsai.learned.PiecewiseLinearIndex`

입력 : 정렬된 키 K, 오차 ε

$i \leftarrow 0$
조건 $i < n$ **동안**
    $j \leftarrow$ 시작점 i 와 끝점 j 를 잇는 직선이 사이의 모든 점을 ε 안에 두는 가장 큰 j
    조각 $(K[i],$ 기울기, 절편$)$ 을 더한다; $i \leftarrow j$
끝 조건

구현

조각은 시작 키와 기울기와 절편의 배열 셋이다. 예측은 조각 찾기 하나와 곱셈 하나다.

지금 보는 코드 조각을 찾고 직선으로 위치를 예측한 뒤 오차 창 안에서 이진 탐색한다.

구현 예측과 창 안 탐색 `samples/dsai/learned.py:55`

```
    def predict(self, k: np.ndarray) -> np.ndarray:
        seg = np.searchsorted(self.starts, k, side="right") - 1
        seg = np.clip(seg, 0, len(self.starts) - 1)
        pos = self.slopes[seg] * k + self.icepts[seg]
        return np.clip(np.rint(pos).astype(np.int64), 0, len(self.keys) - 1)

    def search(self, k: int) -> tuple[int, int]:
        """예측 위치 근처 2 eps 안에서 이진 탐색. 위치와 들여다본 원소 수."""
        p = int(self.predict(np.array([k]))[0])
        lo, hi = max(p - self.eps, 0), min(p + self.eps + 1, len(self.keys))
        j = int(np.searchsorted(self.keys[lo:hi], k, side="left"))
        return lo + j, int(np.ceil(np.log2(hi - lo + 1))) + 1
```

읽어 보면 `predict` 의 `searchsorted` 가 조각 찾기이고, `search` 의 창이 $[p-\varepsilon, p+\varepsilon]$ 이다. 창 안의 `searchsorted` 가 정확성을 보장하는 이진 탐색이다.

구현 가장 긴 조각 찾기 `samples/dsai/learned.py:37`

```
    def _longest_segment(self, i: int) -> int:
        """알고리즘 8.3. i 에서 시작해 양 끝을 잇는 직선이 모든 점을 eps 안에 두는
        가장 긴 구간의 끝."""
        n = len(self.keys)
        j = min(i + 2, n)
        while j < n:
            x0, x1 = self.keys[i], self.keys[j]
            if x1 == x0:
                j += 1
                continue
            slope = (j - i) / (x1 - x0)
            xs = self.keys[i:j + 1].astype(np.float64)
            pred = i + slope * (xs - x0)
            if np.max(np.abs(pred - np.arange(i, j + 1))) > self.eps:
                break
            j += 1
        return j
```

읽어 보면 끝점을 하나씩 늘리며 직선을 다시 긋는 단순한 방법이라 $O(n\varepsilon)$ 남짓이다. 원 논문은 볼록 껍질로 조각 하나를 선형 시간에 최적으로 만들고, 그것이 조각 수를 이 구현보다 줄인다.

분석

학습된 인덱스는 B-트리를 조정한 것이다. 정의의 모양이 같다. 층이 있고, 층마다 블록 하나를 읽고, 잎이 정렬된 배열이다. 바뀐 것은 내부 노드가 표에서 식이 됐다는 것이고, 그래서 메모리가 데이터를 따른다. 이것이 이 책에서 구조의 모양이 데이터에서 나오는 첫 자리다.

이 구조가 이 책의 작업량에서 서는 자리는 정렬된 정수 키의 큰 배열 위다. 정렬된 문서 번호, 정렬된 타임스탬프, 13장의 역파일에서 클러스터마다 정렬된 벡터 번호. 키가 정렬되어 있고 읽기가 대부분이라는 조건이 이 책의 인덱스 대부분에 맞다.

현장에서

> Kraska 등의 2018년 논문은 정렬된 정수 키 위에서 두 층 모델이 B-트리보다 최대 수십 배 작은 메모리로 비슷하거나 빠른 탐색을 보였다고 보고했고, 삽입과 문자열 키를 열린 문제로 남겼다 [확립] [30]. Ferragina 와 Vinciguerra 는 오차 상한이 보장되는 조각 선형 모델로 최악의 경우 경계를 세우고 삽입을 더했다 [확립] [32].

주의

> 학습된 인덱스가 작다는 것은 균일한 키에서 잰 값이다. 실험의 쏠린 분포는 B-트리보다 컸다. 데이터를 재기 전에 어느 쪽이 작은지 말할 수 없다. 그리고 이 절의 인덱스는 정적이다. 삽입이 있으면 조각을 다시 만들어야 하고, 그 값이 이 실험에 없다.

두 절이 내부 노드를 줄였다. 남은 것은 잎이고, 잎은 정렬된 배열이다. 정렬된 배열 위의 탐색은 이 장이 시작한 자리이기도 하다. 다음 절은 그 배열 하나로 되돌아가 트리 없이 탐색한다.

되짚기

누적 분포를 조각 선형 모델로 두는 학습된 인덱스를 정의하고, 오차 상한이 탐색의 정확성을 보장함을 보였다. 균일한 키 백만 개에서 20 KB 로 B-트리 내부 노드의 6 분의 1 이었고, 쏠린 키에서는 세 배였다. 구조의 모양이 데이터에서 나온다. 다음 절은 정렬된 배열 하나로 되돌아간다.

8.4 정렬된 배열과 이진 탐색에 다시 서기. 배치와 분기 없는 탐색

8.2 의 관찰 하나가 남아 있다. 배열 전체의 이진 탐색이 차수 512 의 B-트리보다 빨랐다. 층을 오르내리는 값이 층이 줄이는 라인보다 컸다. 이 절은 트리를 아예 두지 않고 정렬된 배열 하나 위에서 탐색을 다시 쓴다. 두 방향이다. 배열의 배치를 바꿔 이진 탐색이 지나는 자리를 라인에 모으는 것과, 질의를 한꺼번에 받아 정렬하는 것이다.

둘 다 1.1 의 주소 계산이 바탕이다. 트리의 자식은 링크를 읽어야 알지만, 배열의 다음 자리는 곱셈 하나로 안다. 링크가 없으면 4.1 의 의존 읽기도 없다.

학습 목표

> 아이칭거 배치를 정의하고, 힙 번호의 산수만으로 이진 탐색이 되는 이유와 위쪽 노드들이 한 라인에 모이는 이유를 설명할 수 있다. 질의 배치를 정렬하면 탐색의 이동량이 왜 주는지 명제로 세우고, 실측으로 세 배를 말할 수 있다.

이 말부터

아이칭거 배치	정렬된 키를 힙 번호 순서로 놓은 배열. 뿌리가 1 번, 자식이 $2i$ 와 $2i+1$ 이다
분기 없는 탐색	비교 결과를 조건 분기가 아니라 번호의 산수에 넣는 탐색. 예측 실패가 없다
질의 배치	한꺼번에 받은 질의의 묶음. 정렬하면 이웃한 질의가 같은 라인을 지난다

디딤돌

정의 8.12: 아이칭거 배치

> 정렬된 키 n 개를 배열 $E[1..n]$ 에 놓되, 완전 이진 트리의 중위 순회 순서가 정렬 순서가 되게 한다. 즉 $E[i]$ 의 왼쪽 부분 트리 $E[2i], E[4i], \dots$ 의 키는 $E[i]$ 보다 작고 오른쪽은 크다.
>
> **불변식** E 는 정의 8.1 의 순서 불변식을 만족하는 완전 균형 트리이고, 자식 번호는 $2i$ 와 $2i+1$ 이다.
> 탐색은 $i \leftarrow 1$ 에서 시작해 $i \leftarrow 2i + [E[i] < k]$ 를 $i > n$ 이 될 때까지 되풀이한다.

정의의 탐색에는 분기가 없다. 비교 결과가 0 이나 1 이고 그것을 번호에 더한다. 그리고 링크가 없다. 다음 번호가 지금 번호의 산수다. 링크가 없으므로 다음 자리의 주소를 미리 알고, 미리 알면 4.1 의 명제 둘째 조건으로 미리 읽어 둘 수 있다.

명제 8.13: 아이칭거 배치의 이동량

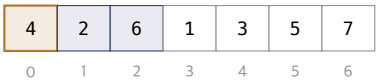
> 위쪽 $\log_2 B$ 층의 노드 $B-1$ 개가 배열 $E[1..B)$ 에 연속으로 놓이므로 라인 하나다. 아래 층에서는 자식 둘 $2i, 2i+1$ 이 이웃이라 한 라인에 든다. 탐색 하나의 라인 수는 $\log_2 n - \log_2 B + 1$ 이하이고, 자식 넷 $4i..4i+3$ 을 미리 읽으면 지연이 층마다 겹친다.

증명. 힙 번호가 층 순서이므로 층 ℓ 의 노드는 $E[2^{\ell}..2^{\ell+1})$ 에 연속이다. 위쪽 층들은 짧아 한 라인에 들고, 아래쪽 층에서는 자식 둘이 이웃 번호라 같은 라인일 확률이 높다. 미리 읽기는 다음 번호가 산수로 나오므로 가능하다. ■

명제의 라인 수는 8.1 의 $\log_2 n$ 과 8.2 의 $\log_B n$ 사이다. 링크를 없앤 대신 층마다 노드 하나를 읽는 것은 그대로다. 그런데 미리 읽기가 지연을 겹치게 하고, 그 겹침이 4.1 의 실험에서 열 배였다. Khuong 과 Morin 이 2017 년에 이 배치를 여러 기계에서 재어 큰 배열에서 가장 빠른 탐색으로 보고했다 [확립] [33].

예제 8.4: 키 천만 개

> $\log_2 10^7 = 24$ 층이다. 8 바이트 키에 64 바이트 라인이면 위쪽 세 층이 라인 하나이고, 아래 스물한 층은 자식 둘씩이라 라인 스물 남짓이다. 이진 탐색은 라인 24 개이고 자리를 미리 모른다. 아이칭거는 라인 21 개이고 자리를 미리 안다.



$E[1..7]$. 자식이 $2i$ 와 $2i+1$ 이라 층이 연속이다

그림 8.6.. 키 일곱 개의 아이칭거 배치. 정렬 순서 1 에서 7 이 힙 번호 순서로 놓였다. 뿌리 4 가 첫 자리다

예제의 24 와 21 은 라인 수이고 시간이 아니다. 시간에는 분기 예측과 미리 읽기가 들어오고, 그것은 언어에 달렸다. 재 본다.

균열

정렬된 키 천만 개에서 질의를 찾는 다섯 방법의 시간을 쟀다. 파이썬 루프의 이진 탐색과 아이칭거, NumPy 의 이진 탐색을 하나씩, 질의 십만 개를 한꺼번에 정렬 없이, 정렬해서.

실험 8.4 같은 배열, 다섯 탐색

방식	질의 수	ms	질의당 us
이진 탐색 (파이썬)	2000	39.8	19.9
아이칭거 (파이썬)	2000	36.6	18.3
searchsorted 하나씩	2000	15.4	7.69
searchsorted 배치 (정렬 안 함)	100000	193	1.93
배치 정렬 탐색	100000	65.6	0.66

측정 조건. 정렬된 40 비트 키 천만 개. 파이썬 루프 둘은 리스트로 바꿔 돌렸고 2,000 개로 쟀다. 다섯 번 중앙값.

파이썬 루프에서 이진 탐색과 아이칭거는 19.9 와 18.3 마이크로초로 거의 같다. 층마다 파이썬 문장 몇 개의 값이 라인 하나의 값보다 커서 배치의 차이가 묻힌다. NumPy 의 이진 탐색을 하나씩 부르면 7.7 마이크로초이고, 십만 개를 한꺼번에 주면 1.9 마이크로초다. 호출 하나의 값이 빠졌다. 그리고 질의를 정렬해서 주면 0.66 마이크로초다. 같은 함수, 같은 질의인데 순서만 바꿔 세 배다.

관찰

정렬 안 한 배치와 정렬한 배치의 차이가 이 실험에서 가장 큰 숫자다. 이웃한 질의가 배열의 이웃한 자리를 지나므로 위쪽 층은 캐시에 있고 아래쪽 층도 앞 질의가 읽은 라인에 있다. 같은 함수의 이동량이 질의 순서로 세 배 달라진다.

이 관찰이 이 절의 둘째 다시 쓰기다. 배열의 배치가 아니라 질의의 배치다. 그리고 6.4 의 정렬이 여기서 다시 나온다.

다시 쓰기

명제 8.14: 정렬된 질의 배치가 줄이는 이동량

질의 q 개를 정렬해 차례로 이진 탐색하면, 이웃한 두 질의가 지나는 경로는 위쪽 층에서 같고 아래쪽에서 갈린다. 질의 q 개가 키 n 개에 고르게 퍼져 있으면 경로가 갈리는 층은 약 $\log_2 q$ 층 아래이고, 그 위의 라인은 앞 질의가 읽어 캐시에 있다. 탐색당 새 라인은 $\log_2 n - \log_2 q = \log_2(n/q)$ 개다. $n = 10^7$, $q = 10^5$ 이면 24 개가 7 개로 준다.

증명. 정렬된 질의 q 개가 배열을 q 조각으로 나누고, 조각 하나가 크기 n/q 다. 이웃한 질의는 같은 조각 또는 이웃 조각에 있으므로 위쪽 $\log_2 q$ 층에서 경로가 같거나 이웃한다. 같은 경로의 라인은 캐시에 남아 있고, 아래쪽 $\log_2(n/q)$ 층만 새 라인이다. ■

명제가 말하는 것은 질의를 정렬하는 값 $q \log_2 q$ 의 비교가 라인 $q \log_2 q$ 개를 아낀다는 것이다. 비교는 싸고 라인은 비싸므로 언제나 남는다. 이것이 6.4 의 정렬 기반 세기와 같은 꼴이고, 가속기 위의 탐색이 거의 전부 정렬된 배치인 이유다.

알고리즘 8.4 분기 없는 아이칭거 탐색 `dsai.layout.eytzinger_search`

입력: 아이칭거 배열 $E[1..n]$, 키 k
1: $i \leftarrow 1$
2: **조건** $i \le n$ **동안**
3: $\quad i \leftarrow 2i + [E[i] < k]$
4: **끝** 조건
5: **반환** i 에서 뒤에 붙은 1 비트를 걷어 낸 번호 ▷ 마지막으로 왼쪽으로 꺾인 자리

알고리즘 8.5 정렬된 배치 탐색 `dsai.layout.batch_search`

입력: 정렬된 키 K, 질의 $Q[0..q)$
1: $\sigma \leftarrow Q$ 를 정렬하는 순열
2: $P \leftarrow Q[\sigma]$ 의 각 질의를 K 에서 이진 탐색한 위치
3: **반환** P 를 σ^{-1} 로 되돌린 것

구현

아이칭거 탐색의 핵심은 두 줄이다. 번호를 두 배 하고 비교 결과를 더하는 것, 그리고 끝에서 뒤에 붙은 1 비트를 걷는 것.

지금 보는 코드 번호의 산수만으로 내려가고, 마지막으로 왼쪽으로 꺾인 자리를 비트 연산으로 되찾는다.

구현 분기 없는 내려가기 `samples/dsai/layout.py:45`

```
def eytzinger_search(tree: np.ndarray, k: int) -> tuple[int, int]:
    """알고리즘 8.4. 분기 없는 내려가기. i 를 2i 또는 2i+1 로만 옮긴다."""
    n = len(tree) - 1
    i, steps = 1, 0
    while i <= n:
        steps += 1
        i = 2 * i + int(tree[i] < k)          # 비교 결과가 곧 자식 번호
    # 마지막으로 왼쪽으로 꺾인 자리가 답. 뒤에 붙은 1 비트 (오른쪽 꺾임) 를 걷어 낸다
    j = i >> ((~i & (i + 1)).bit_length())
    return j, steps
```

읽어 보면 `int(tree[i] < k)` 가 0 이나 1 이고 그것이 자식 번호다. 마지막 줄의 `~i & (i + 1)` 이 i 의 가장 낮은 0 비트를 뽑고, 그만큼 오른쪽으로 밀면 마지막 왼쪽 꺾임의 번호다. 오른쪽으로만 꺾였으면 0 이 나오고, $E[0]$ 은 답 없음의 자리다.

구현 질의를 정렬해 한꺼번에 찾기 `samples/dsai/layout.py:57`

```python
def batch_search(keys: np.ndarray, queries: np.ndarray) -> np.ndarray:
    """알고리즘 8.5. 질의를 정렬해 한 번에 찾는다.
    NumPy 의 searchsorted 는 이 꼴이다."""
    order = np.argsort(queries, kind="stable")
    pos = np.searchsorted(keys, queries[order], side="left")
    out = np.empty_like(pos)
    out[order] = pos
    return out
```

읽어 보면 `argsort` 하나와 `searchsorted` 하나와 되돌리기 하나다. 실험의 세 배가 이 세 줄에서 나온다. NumPy 의 `searchsorted` 는 질의가 정렬되어 있을 때 앞 질의의 자리를 힌트로 쓰지 않지만, 캐시가 대신 그 일을 한다.

분석

정렬된 배열 위의 이진 탐색은 그대로 유효하다. 알고리즘이 1946 년 그대로이고 비교 횟수가 하한이다. 바뀐 것은 배열의 배치와 질의의 배치다. 둘 다 이동량을 줄이고, 비교 횟수는 그대로다. 이 장의 네 절이 전부 같은 문장이다.

이 절이 이 장을 닫는 방식은 되돌아가기다. 트리에서 시작해 노드를 키우고 노드를 모델로 바꾸고 결국 배열 하나에 섰다. 정적이고 읽기가 대부분인 이 책의 작업량에서 순서 있는 탐색의 답은 정렬된 배열이고, 나머지는 그 배열 위에 무엇을 얹느냐다.

현장에서

> Khuong 과 Morin 은 여러 기계에서 네 배치를 재어, 작은 배열에서는 정렬 배열의 좋은 이진 탐색이 가장 빠르고 큰 배열에서는 미리 읽기를 넣은 아이칭거가 가장 빠르다고 보고했다 [확립] [33]. 그 논문의 결론이 이 절의 실험과 다른 것은 파이썬이 미리 읽기를 못 하기 때문이고, 같은 것은 배치가 정답을 바꾸지 않고 이동량만 바꾼다는 것이다.

주의

> 실험 8.4 의 아이칭거는 파이썬 루프라 배치의 이점이 안 보인다. 배치의 이점은 미리 읽기와 분기 예측이 있는 언어에서 나오고, 그것은 4.1 의 실험에서 열 배였다. 파이썬에서 재서 없다고 결론 내지 않는다.

트리는 순서를 지키는 대신 높이를 값으로 치렀고, 이 장은 그 값을 셋으로 줄였다. 노드를 키우고, 노드를 식으로 바꾸고, 배열로 되돌렸다. 다음 장은 순서 대신 접두사를 지키는 트리다. 문자열을 다루고, 토크나이저와 접두사 캐시가 거기서 나온다.

되짚기

> 아이칭거 배치와 분기 없는 탐색을 정의하고, 정렬된 질의 배치가 이동량을 줄이는 명제를 세웠다. 같은 함수가 질의 순서만으로 세 배 빨라졌다. 순서 있는 탐색의 답은 정렬된 배열이고, 나머지는 그 위에 얹는 것이다. 다음 장은 접두사 트리다.

8장을 닫으며

이 장이 세운 구조는 넷이다. 순서 불변식의 이진 탐색 트리와 힙 불변식을 더한 트립, 정렬 배열 위의 정적 B-트리, 오차 상한이 붙은 조각 선형 학습 인덱스, 그리고 아이칭거 배치와 정렬된 질의 배치. 넷 다 정렬된 키에서 자리를 찾고, 넷 다 비교 횟수는 $\log_2 n$ 근처다. 다른 것은 라인 수뿐이고, 그것이 이 책의 잣대다.

네 실험의 숫자를 적어 둔다. 무작위 순서의 트리는 평균 깊이 21.7 로 정리의 23 근처였고, 정렬 순서는 높이가 키 수와 같았다. 키 천만 개의 B-트리는 차수 512 에서 높이 3, 내부 노드 0.16 MB 였다. 학습된 인덱스는 균일한 키 백만 개에서 20 KB 로 B-트리의 6 분의 1 이었고, 쏠린 키에서는 348 KB 로 세 배였다. 같은 이진 탐색이 질의를 정렬해 주자 1.9 에서 0.66 마이크로초로 세 배 빨라졌다.

정정 하나를 남긴다. 아이칭거 탐색의 마지막 줄, 뒤에 붙은 비트를 걷어 내는 식을 처음에 가장 낮은 1 비트로 썼다. 마지막 왼쪽 꺾임은 가장 낮은 0 비트 위에 있으므로 틀린 자리를 냈고, 시험이 잡았다. 분기 없는 탐색은 분기가 없는 대신 비트 하나가 정확성을 진다. 정의는 맞았고 구현의 한 줄이 틀렸다.

옛 구조의 판정

> 이진 탐색 트리는 조정해서 유효하다. 순서 불변식과 높이 정리는 그대로이고, 높이의 단위가 비교에서 라인으로 바뀌었다. 근거는 실험 8.2 와 8.4 다. 비교 횟수가 같은데 블록 수로 여섯 배, 질의 순서로 세 배가 갈렸다. 남은 것은 B-트리와 정렬된 배열이고, 노드가 힙에 흩어진 트리는 라인 수십 개 안에 드는 크기에서만 남는다.

이 장의 요약

> 1. 이진 탐색 트리의 높이는 넣는 순서가 정하고, 무작위면 $2 \ln n$, 정렬이면 n 이다. 트립이 무작위를 만들어 넣는다 8.1
> 2. 높이는 라인 수다. 로그의 밑을 키우는 것이 B-트리이고, 차수는 블록 크기에서 나온다 8.2
> 3. 내부 노드는 키에서 위치로 가는 함수이고, 오차 상한 조각 선형 모델이 그 함수를 대신한다. 메모리가 데이터를 따른다 8.3
> 4. 링크를 없애면 다음 자리를 미리 안다. 아이칭거 배치가 그것이고, 미리 읽기가 지연을 겹친다 8.4
> 5. 질의를 정렬하면 이웃한 질의가 같은 라인을 지난다. 같은 함수가 순서만으로 세 배다 8.4
>
> 다음 장에서 뒤집히는 것: 이 장의 키는 정수였고 비교 하나가 상수였다. 9장의 키는 문자열이고, 비교가 길이만큼 들며, 접두사를 나누는 트리가 그 비교를 없앤다.

필자의 판단

> 필자는 학습된 인덱스를 B-트리의 변형으로 가르치는 쪽을 택했다. 신경망으로 시작하면 학생은 모델에 눈이 가고, 정확성이 오차 상한과 창 안 탐색에서 온다는 것을 놓친다. B-트리의 내부 노드가 함수라는 것을 먼저 보이면 학습된 인덱스는 새 구조가 아니라 같은 구조의 다른 내부 노드다.

열린 문제

하나, 실험 8.4 에서 아이칭거 배치의 이점을 파이썬이 가렸다. 미리 읽기가 되는 언어에서 같은 실험을 다시 해야 이 절의 명제를 실측으로 닫을 수 있다.
둘, 학습된 인덱스의 조각을 이 장은 단순한 방법으로 만들었다. 볼록 껍질로 최적 조각을 만들면 조각 수가 얼마나 주는지 재지 않았다.
셋, 이 장의 인덱스는 전부 정적이다. 삽입이 있을 때 B-트리의 분할과 학습된 인덱스의 조각 다시 만들기가 어느 빈도에서 어느 쪽이 싼지 재지 않았다.

연습문제

8.1 기초 키 백만 개를 무작위 순서로 넣은 이진 탐색 트리의 기대 깊이를 정리 8.2 로 구하고, 완전 균형의 높이와 견줘라. 노드 하나가 라인 하나이면 탐색 하나에 라인 몇 개인가.

8.2 기초 키 십억 개, 8 바이트 키에서 라인 64 바이트와 페이지 4 KB 의 차수와 높이를 명제 8.7 로 구하라. 내부 노드는 각각 몇 MB 인가.

8.3 기초 조각 2,000 개, 오차 상한 64 의 학습된 인덱스의 메모리와 탐색 비교 횟수를 명제 8.10 로 구하고, 같은 키 백만 개의 차수 64 B-트리와 견줘라.

8.4 기초 키 2^{30} 개에 질의 2^{16} 개를 정렬해 찾을 때 탐색당 새 라인 수를 명제 8.14 로 구하라. 정렬 안 하면 몇 개인가.

8.5 응용 `dsai.bst.Treap` 에 삭제를 더하라. 지울 노드를 회전으로 잎까지 내린 뒤 떼어 낸다. 지운 뒤에도 두 불변식이 지켜지는지 시험으로 확인하라.

8.6 응용 `dsai.btree.StaticBTree` 의 블록 안 이진 탐색을 순차 비교로 바꾸고, 차수 8 과 16 에서 실험 8.2 의 시간이 어떻게 바뀌는지 재라. 어느 차수부터 이진 탐색이 이기는가.

8.7 응용 `dsai.learned.PiecewiseLinearIndex` 를 명제 8.11 대로 두 층으로 만들어라. 조각 시작 키 위에 같은 인덱스를 세우고, 조각 찾기의 비교 횟수가 얼마나 주는지 재라.

8.8 응용 `dsai.layout.batch_search` 에서 정렬한 질의를 앞 질의의 위치부터 갤로핑 (보폭을 두 배씩 늘리는 탐색) 으로 찾게 바꾸고, 실험 8.4 와 견줘라.

8.9 탐구 학습된 인덱스에 삽입을 더하려면 어떤 불변식을 어떻게 완화해야 하는지 논하라. 오차 상한을 지키면서 조각 하나만 다시 만드는 조건은 무엇인가.

8.10 탐구 문자열 키에 학습된 인덱스를 세우려면 키를 실수로 바꿔야 한다. 앞 8 바이트를 정수로 읽는 방법의 오차 상한이 왜 커지는지 정의 8.9 로 설명하라.

더 읽을 것

[31] B-트리의 원전. 디스크 페이지 위의 인덱스라는 가정과 분할, 병합이 다 있다. 8.2
[30] 학습된 인덱스를 제안한 논문. 내부 노드가 누적 분포 함수라는 관찰과 두 층 모델의 실험이 있다. 8.3
[32] 오차 상한이 보장되는 조각 선형 학습 인덱스. 최악의 경우 경계와 삽입이 있다. 8.3
[33] 정렬 배열의 네 배치를 여러 기계에서 잰 논문. 아이칭거와 미리 읽기의 실측이 8.4 절의 근거다. 8.4

9장

트라이, 토크나이저, 접두사 캐시

1960년의 논문 하나는 문자열을 저장하는 새 방법의 이름을 retrieval의 가운데에서 따왔다 [확립] [34]. 글자 하나가 간선 하나이고, 같은 앞부분을 가진 문자열은 같은 경로를 나눈다. 그 트리가 지금 언어 모델의 입구와 출구에 서 있다. 텍스트를 토큰으로 자르는 자리, 여러 요청이 같은 앞부분의 계산을 나누는 자리, 출력이 형식을 지키게 하는 자리. 셋 다 접두사를 나눈다는 성질 하나에서 나온다. 이 장은 그 성질을 정의로 세우고, 네 자리에서 값을 잰다.

이 장은 0.2의 둘째 가정을 건드린다. 키가 정수가 아니라 열이고, 비교 하나가 상수가 아니다. 그리고 8장의 정렬 순서 대신 접두사 순서가 트리를 정한다. 6.2의 어휘 사전이 트라이가 되고, 5.3의 KV 캐시가 트리 위에 얹힌다.

이 장에서 세우는 것

9.1 트라이와 최장 일치. 탐색 비용이 문자열 길이에만 달렸다. 토큰화가 dict 방법보다 여섯 배

9.2 바이트 쌍 부호화. 병합 순서가 어휘이자 규칙이다. 최장 일치와 단어의 8 퍼센트가 다르게 나뉜다

9.3 라디스 트리 위의 KV 캐시. 접두사 공유로 계산이 7 분의 1. 축출은 잎만

9.4 제약 디코딩. 트라이 위의 오토마톤. 허용 집합의 비용이 어휘 크기와 무관하다

9.1 트라이와 최장 일치

8.1 부터 8.4 까지의 키는 정수였다. 비교 하나가 명령 하나였고, 그래서 비교 횟수가 곧 비용이었다. 키가 문자열이면 비교 하나가 길이만큼 든다. 6.2 의 어휘 사전이 그 값을 치렀다. 문자열을 읽어 해시하고 다시 읽어 확인한다. 트라이는 그 비교를 글자 하나씩으로 쪼개 트리의 층으로 편다. 같은 접두사를 가진 키들이 같은 경로를 나눠 쓰고, 문자열 하나를 찾는 데 그 길이만큼의 걸음이 든다.

이 절이 트라이를 세우는 자리는 토크나이저다. 어휘 수만 개에서 텍스트의 각 위치마다 가장 긴 항목을 찾는 일이고, 그것이 트라이를 한 번 내려가는 것으로 끝난다.

학습 목표

> 트라이를 접두사 공유 트리로 정의하고, 문자열 하나의 탐색 비용이 문자열 길이에만 달렸음을 보일 수 있다. 최장 일치 토큰화를 알고리즘으로 적고, dict 로 길이를 줄여 가며 묻는 방법과 비용을 견줄 수 있다.

이 말부터

> | | |
> |---|---|
> | 트라이 | 간선에 글자 하나씩을 달아 뿌리에서 노드까지의 경로가 문자열이 되는 트리 |
> | 최장 일치 | 텍스트의 한 위치에서 시작하는 어휘 항목 중 가장 긴 것을 고르는 규칙 |
> | 토큰화 | 바이트 열을 어휘 항목의 번호 열로 바꾸는 일. 바이트 256 개가 어휘에 있으면 언제나 된다 |

디딤돌

정의 9.1: 트라이

> 뿌리 하나와, 노드마다 바이트로 이름 붙은 자식 목록을 둔 트리다. 어휘 항목 w 를 넣으면 뿌리에서 w 의 바이트를 차례로 따라 내려가며 없는 간선을 만들고, 마지막 노드에 w 의 번호를 적는다.
>
> **불변식** 뿌리에서 노드 v 까지의 간선 이름을 이은 것이 v 의 문자열이고, 서로 다른 노드는 서로 다른 문자열이다. 어휘 항목 w 는 정확히 문자열 w 의 노드에 번호가 적혀 있다.

불변식이 말하는 것은 문자열과 노드의 일대일이다. 그래서 문자열을 찾는 것은 그 바이트를 따라 내려가는 것이고, 걸음 수가 문자열 길이다. 어휘가 만 개든 백만 개든 걸음 수는 찾는 문자열의 길이이지 어휘 크기가 아니다.

명제 9.2: 트라이의 비용

> 어휘 항목의 길이 합을 S, 항목 수를 V 라 하자. 트라이의 노드 수는 $S + 1$ 이하이고, 접두사를 나눌수록 작다. 문자열 w 를 찾는 비용은 $|w|$ 걸음이고, 어휘 크기 V 와 무관하다. 노드가 흩어져 있으면 걸음마다 라인 하나다.

증명. 항목 하나가 만드는 노드는 그 길이 이하이고, 다른 항목과 접두사를 나누면 그만큼 덜 만든다. 탐색은 불변식에 의해 바이트마다 간선 하나를 따르는 것이므로 길이만큼이다. ∎

명제의 마지막 문장이 이 책의 잣대다. 걸음 $|w|$ 개가 라인 $|w|$ 개이면 6.2 의 해시 조회(문자열 읽기 한 번에 라인 한둘) 보다 나쁘다. 그래서 트라이가 이기는 자리는 문자열 하나를 찾는 곳이 아니라, 같은 텍스트 위에서 여러 길이의 항목을 한 번에 물어야 하는 곳이다. 최장 일치가 그 자리다.

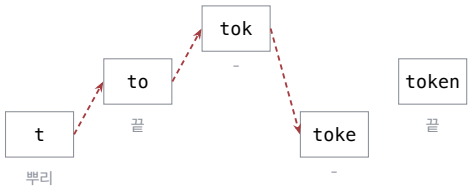
정의 9.3: 최장 일치 토큰화

> 텍스트 T 와 어휘 $\mathcal{V}$ 에 대해, 위치 i 에서 $T[i..i+\ell)$ 이 $\mathcal{V}$ 에 있는 가장 큰 ℓ 을 고르고 그 항목의 번호를 내보낸 뒤 $i \leftarrow i + \ell$ 로 옮긴다. T 의 끝까지 되풀이한다.
>
> **불변식** 내보낸 항목들을 이으면 T 다. 어휘에 바이트 256 개가 있으면 $\ell \geq 1$ 이 언제나 있어 끝난다.

디딤돌은 이것을 dict 로 하는 것이다. 위치마다 가장 긴 길이 L 부터 1 까지 부분 문자열을 잘라 dict 에 묻는다. 있으면 그것이 최장이다. 위치 하나에 물음이 L 번이고 물음마다 부분 문자열을 만들어 해시하니, 위치당 비용이 $O(L^2)$ 이다. 트라이는 위치당 $O(\ell)$ 이다. 내려가다 끊기는 자리까지만 본다.

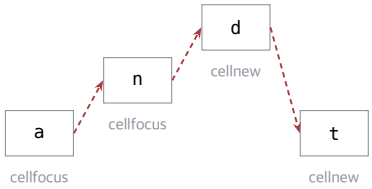
예제 9.1: 어휘 항목 최장 38 바이트

> 이 절의 어휘에서 가장 긴 항목은 38 바이트다. dict 방법은 위치마다 최대 38 번 자르고 해시하고, 자른 길이의 합이 700 바이트 남짓이다. 트라이는 위치마다 평균 두 바이트에서 세 바이트를 내려가고 끊긴다. 실제로 그 자리에서 뽑히는 항목의 평균 길이가 2 바이트이기 때문이다.



입력 tokens 는 token 에서 멈춘다

그림 9.1.. 트라이의 경로 하나. 뿌리에서 글자를 하나씩 내려가고, 끝 표시가 있는 가장 깊은 노드가 최장 일치다



a 에서 n 을 지나 d 와 t 로 갈린다. an 까지의 경로를 셋이 나눈다

그림 9.2. 어휘 넷 (a, an, and, ant) 의 트라이. 경로가 문자열이고, 번호가 적힌 노드가 항목이다

예제의 700 대 3 은 셈이다. 실제 텍스트에서 두 방법이 몇 배 갈리는지 잰다.

균열

바이트 256 개에 9.2 의 병합 1,000 개를 더한 어휘로 소스 코드 200 KB 를 토큰화했다. 두 방법이 같은 토큰 열을 내는지 확인하고 시간을 쟀다.

실험 9.1 최장 일치 토큰화. dict 로 길이를 줄여 묻기와 트라이

방식	최장 항목 또는 노드	ms	바이트당 ns
dict 에 길이 줄여 묻기	38	1897	9486
트라이 한 번 내려가기	38	323	1617
트라이 노드 수	2135	0.00	0.00

측정 조건. 파이썬 표준 라이브러리 소스 200 KB. 어휘 1,256 개, 토큰 십만 개. 두 방법의 토큰 열이 같음을 확인했다. 세 번 중앙값.

트라이가 여섯 배 빠르다. dict 방법은 바이트당 9.5 마이크로초, 트라이는 1.6 마이크로초다. 바이트당 1.6 마이크로초는 파이썬 루프의 값이고 C 로 쓰면 수십 나노초다. 비율만 읽는다.

관찰

dict 방법의 비용은 어휘의 가장 긴 항목이 정하고, 트라이의 비용은 텍스트에서 실제로 뽑히는 항목이 정한다. 어휘에 긴 항목 하나가 들어오면 dict 방법은 전체가 느려지고 트라이는 그 항목이 뽑히는 자리만 느려진다.

이 관찰이 토크나이저가 트라이를 쓰는 이유다. 어휘에는 드물게 쓰이는 긴 항목이 있고, 그것이 모든 위치의 값을 정하게 두지 않는다.

다시 쓰기

트라이의 노드가 흩어져 있으면 걸음마다 라인 하나이고, 그것이 명제 9.2 의 마지막 문장이다. 다시 쓰기는 노드를 어디에 두느냐다.

명제 9.4: 트라이의 배치

> 자식을 노드마다 바이트 256 칸의 배열로 두면 걸음 하나가 주소 계산 하나이지만 노드당 2 KB 다. 자식을 (노드, 바이트) 를 키로 하는 해시 테이블 하나에 두면 걸음 하나가 해시 조회 하나이고 메모리는 간선 수에 비례한다. 어휘 트라이는 뿌리 근처가 넓고 아래가 좁으므로, 위쪽 몇 층은 배열로, 아래는 해시나 정렬 목록으로 두는 것이 두 값을 함께 낮춘다.

증명. 배열 자식은 1.1 의 주소 계산이고 라인 하나다. 해시 자식은 6.1 의 조회이고 라인 하나에서 둘이다. 노드당 메모리는 각각 256×8 바이트와 간선당 24 바이트 남짓이다. 어휘의 접두사 분포에서 위쪽 층은 자식이 많고 아래쪽은 하나둘이므로 층마다 다른 표현이 맞다. ■

이 명제가 8.2 의 차수 선택과 같은 논리다. 위쪽은 넓게 아래쪽은 좁게. 이 책의 구현은 해시 하나로 두었고, 실험의 여섯 배는 그 표현에서 나왔다. 배치를 바꾸기 전에 알고리즘부터 적어 둔다. 내려가다 끊기는 자리까지만 보는 것이 전부다.

알고리즘 9.1 최장 일치 `dsai.trie.Trie.longest_match`

> **입력** : 트라이, 텍스트 T, 위치 i
> $v \leftarrow$ 뿌리 ; 최선 $\leftarrow$ 없음
> **반복** $j = i, i+1, \ldots$ **동안**
> $v \leftarrow$ 자식$(v, T[j])$; 없으면 멈춘다
> **만약** v 에 번호가 있음 **이면** 최선 $\leftarrow$ (v 의 번호, $j - i + 1$)
> **끝 만약**
> **끝 반복**
> **반환** 최선

구현

자식은 (노드, 바이트) 를 키로 하는 dict 하나다. 노드마다 dict 을 두는 것보다 작고, 자식 목록은 따로 둔다.

지금 보는 코드 바이트를 따라 내려가며 번호가 있는 노드를 기억한다. 끊기면 마지막 기억이 답이다.

구현 최장 일치 `samples/dsai/trie.py:37`

```
    def longest_match(self, text: bytes, start: int) -> tuple[int, int]:
        """알고리즘 9.1. start 에서 시작하는 가장 긴 어휘 항목의 (번호, 길이).
        없으면 (-1, 0).
        """
        node, best, best_len = 0, NIL, 0
        for i in range(start, len(text)):
            node = self.child.get((node, text[i]), NIL)
            if node == NIL:
                break
            if self.token[node] != NIL:
                best, best_len = self.token[node], i - start + 1
        return best, best_len
```

읽어 보면 넣기는 바이트마다 간선을 만드는 것이 전부라 싣지 않았다. `longest_match` 는 끊기는 자리까지만 내려간다. 어휘에 없는 긴 부분 문자열은 첫 몇 바이트에서 끊긴다. `token` 이 번호 배열이고 −1 이 항목 아님이다.

구현 dict 로 길이를 줄여 묻기 `samples/dsai/trie.py:66`

```
def tokenize_by_dict(vocab: dict[bytes, int], text: bytes, max_len: int) -> list[int]:
    """디딤돌. 가장 긴 길이부터 줄여 가며 dict 에 물어보는 최장 일치.
    위치마다 max_len 번 해시.
    """
    out, i = [], 0
    while i < len(text):
        for ln in range(min(max_len, len(text) - i), 0, -1):
            tok = vocab.get(text[i:i + ln])
            if tok is not None:
                out.append(tok)
                i += ln
                break
    return out
```

읽어 보면 안쪽 `for` 가 위치마다 `max_len` 번 돈다. `text[i:i + ln]` 이 부분 문자열 복사이고 `get` 이 해시다. 실험의 여섯 배가 이 두 줄에서 나온다.

분석

트라이는 그대로 유효하다. 1960 년의 정의와 명제가 그대로이고, Fredkin 이 든 장점도 그대로다. 접근이 짧고, 길이가 다른 키를 한 표에 담고, 접두사의 중복을 저장하지 않는다 [확립] [34]. 바뀐 것은 자리다. 이 구조가 이 책의 작업량에서 서는 자리는 토크나이저이고, 그것은 모든 언어 모델의 입구다.

이 절의 최장 일치는 토큰화의 한 규칙이다. 다음 절의 병합 순서가 다른 규칙이고, 둘은 같은 어휘에서 다른 토큰 열을 낸다. 그리고 9.3 은 문자열 대신 토큰 열을 키로 하는 트라이이고, 9.4 은 이 트라이 위에서 오토마톤을 돌린다. 세 절이 전부 이 정의 위에 있다.

현장에서

> Fredkin 의 1960 년 논문은 트라이를 메모리 구조로 제안했고, 이름은 retrieval 에서 왔다 [확립] [34]. Morrison 의 1968 년 PATRICIA 는 자식이 하나뿐인 노드를 이어 붙여 경로를 압축했고 [확립] [35], 그것이 9.3 의 라디스 트리다.

주의

> 실험 9.1 의 두 방법은 같은 토큰 열을 냈지만, 최장 일치는 토큰화의 유일한 규칙이 아니다. 다음 절에서 같은 어휘로 다른 규칙을 쓰면 단어의 8 퍼센트에서 13 퍼센트가 다르게 나뉜다. 같은 어휘라고 같은 토큰이 아니다.

트라이는 어휘가 주어졌을 때의 구조다. 어휘를 어떻게 만드느냐가 다음 절이고, 그 답이 병합의 순서다. 병합 순서는 어휘이면서 규칙이라, 트라이에 넣는 것만으로는 그 규칙이 안 따라온다.

되짚기

> 트라이를 정의하고 탐색 비용이 문자열 길이에만 달렸음을 보였다. 최장 일치 토큰화가 dict 방법보다 여섯 배 빨랐고, 그 비율은 어휘의 가장 긴 항목이 아니라 실제로 뽑히는 항목이 정한다. 다음 절은 어휘를 만드는 병합 순서다.

9.2 바이트 쌍 부호화와 병합 순서

9.1 의 어휘는 주어진 것이었다. 어휘를 만드는 규칙 중 지금 가장 널리 쓰이는 것이 바이트 쌍 부호화다. 자주 붙어 나오는 바이트 쌍을 하나로 합치고, 합친 것을 다시 쌍의 재료로 삼아 되풀이한다. 2016 년에 Sennrich 등이 이 압축 방법을 신경망 번역의 어휘 만들기에 가져왔고 [확립] [36], 그 뒤 대부분의 언어 모델이 이 방식 또는 그 변형으로 어휘를 만든다.

이 절이 세우는 것은 두 가지다. 병합 순서라는 자료 구조와, 그 순서를 따르는 부호화가 9.1 의 최장 일치와 다르다는 사실이다. 어휘가 같아도 규칙이 다르면 토큰 열이 다르고, 모델은 학습 때의 규칙으로만 텍스트를 본다.

학습 목표

바이트 쌍 부호화의 학습을 쌍의 빈도표 위의 반복으로 정의하고, 병합 순서가 어휘와 부호화 규칙을 함께 정함을 설명할 수 있다. 병합 순서 부호화와 최장 일치가 왜 다른 토큰 열을 내는지 예로 보이고, 병합 수와 토큰당 바이트의 관계를 실측으로 말할 수 있다.

이 말부터

병합	이웃한 토큰 쌍 (a, b) 를 새 토큰 하나로 바꾸는 규칙. 학습이 만드는 것은 이 규칙의 열이다
순위	병합이 학습된 차례. 부호화에서 순위가 낮은 (먼저 배운) 병합을 먼저 적용한다
토큰당 바이트	텍스트의 바이트 수를 토큰 수로 나눈 것. 어휘가 텍스트를 얼마나 압축하는지의 잣대다

디딤돌

정의 9.5: 바이트 쌍 부호화의 학습

단어와 빈도의 표 W 에서 시작한다. 각 단어는 바이트 열이고 토큰 번호 0..255 다. 매 단계에서 모든 단어의 이웃 토큰 쌍 (a, b) 의 빈도 (단어 빈도로 가중) 를 세고, 최대인 쌍을 새 번호 $256 + t$ 로 합친다. 합친 쌍 (a, b) 를 순서대로 적은 것이 병합 순서다. k 단계 뒤 어휘는 $256 + k$ 개다.

불변식 단계 t 의 새 토큰 $256 + t$ 는 그 이전 토큰 둘을 이은 것이다. 그래서 모든 토큰은 바이트 열 하나에 대응하고, 병합 순서만으로 어휘 전체를 되만들 수 있다.

불변식의 마지막 문장이 이 구조의 요점이다. 어휘 파일에 담을 것은 문자열 V 개가 아니라 쌍 k 개이고, 쌍의 순서가 곧 부호화 규칙이다. 부호화는 그 순서를 거꾸로 밟지 않고 그대로 밟는다.

정의 9.6: 병합 순서 부호화

> 단어를 바이트 열로 두고, 이웃 쌍 중 순위가 가장 낮은 병합이 있으면 적용한다. 적용할 병합이 없을 때까지 되풀이한다.
>
> **불변식** 학습 때 그 단어가 지났던 병합의 열을 그대로 다시 지난다. 학습에 있던 단어는 학습 끝의 토큰 열과 같은 열이 된다.

두 정의가 만드는 어휘는 9.1 의 트라이에 넣을 수 있고, 그러면 최장 일치로도 토큰화할 수 있다. 그런데 두 규칙의 답이 다르다. 어느 쪽이 맞느냐가 아니라 어느 쪽으로 학습했느냐의 문제이고, 그 차이를 먼저 명제로 적는다.

명제 9.7: 두 규칙의 차이

> 같은 어휘에서 병합 순서 부호화와 최장 일치는 다른 토큰 열을 낼 수 있다. 최장 일치는 앞에서부터 가장 긴 항목을 고르고, 병합 순서는 먼저 배운 쌍을 먼저 합치므로 긴 항목이 어휘에 있어도 그 항목에 이르는 병합 경로가 없으면 못 만든다. 토큰당 바이트는 둘이 거의 같고, 나뉜 자리가 다르다.

증명은 예 하나로 대신한다. 긴 항목이 어휘에 있어도 병합 경로가 다르면 못 만드는 경우가 있으면 충분하다.

예제 9.2: 같은 어휘, 다른 열

> 어휘에 ab, bc, abc 가 있고 병합 순서가 (a,b), (b,c), (ab,c) 라 하자. 단어 abc 는 병합 순서로 ab 다음 abc 가 되어 하나다. 단어 abcd 에 cd 가 어휘에 있고 순위가 (b,c) 보다 낮으면 최장 일치는 abc, d 둘이고 병합 순서는 ab, cd 둘이다. 개수는 같고 경계가 다르다.

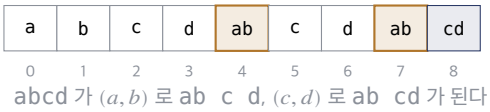


그림 9.3.. 병합 순서의 첫 셋. 바이트 열에서 쌍을 합칠 때마다 토큰 하나가 늘고 열이 하나 짧아진다

예제의 경계 차이가 실제 텍스트에서 얼마나 자주 나는지, 그리고 병합 수가 압축률을 어떻게 움직이는지 잰다.

균열

소스 코드 1 MB 의 상위 8,000 단어로 병합 256, 1,024, 4,096 개를 학습하고, 앞 200 KB 를 두 규칙으로 부호화했다.

실험 9.2 병합 수와 토큰당 바이트, 두 규칙이 다르게 나눈 단어

병합 수	학습 초	병합 순서 바이트	최장 일치 바이트	다른 단어 퍼센트
256	1.55	1.90	1.90	13.6
1024	2.97	2.64	2.64	11.2
4096	9.60	3.69	3.67	8.38

측정 조건. 파이썬 표준 라이브러리 소스. 단어는 공백으로 나누고 앞 공백을 붙였다. 학습은 쌍 빈도를 한 번 세고 병합마다 그 쌍이 든 단어만 다시 센다.

병합을 네 배로 늘릴 때마다 토큰당 바이트가 1.9, 2.6, 3.7 로 는다. 어휘 4,352 개에서 텍스트가 바이트의 27 퍼센트 길이의 토큰 열이 된다. 두 규칙의 토큰당 바이트는 소수점 둘째 자리까지 같다. 그런데 단어의 8 에서 14 퍼센트가 다르게 나뉘었다. 압축률로는 안 보이는 차이다.

관찰

다르게 나뉜 단어의 비율은 병합이 늘수록 준다. 병합이 많으면 자주 쓰는 단어 전체가 항목 하나가 되어 두 규칙이 같은 답을 내기 때문이다. 그래도 8 퍼센트다. 학습 때의 규칙과 다른 규칙으로 텍스트를 넣으면 모델은 열두 단어에 하나꼴로 처음 보는 토큰 열을 받는다.

이 관찰이 토크나이저를 모델 파일의 일부로 보는 이유다. 6.2 의 어휘 사전과 이 절의 병합 순서가 함께 있어야 학습 때와 같은 토큰이 나온다.

다시 쓰기

학습의 비용을 다시 쓴다. 정의 그대로 매 단계 모든 쌍을 다시 세면 단계마다 말뭉치를 한 번 훑는다.

명제 9.8: 학습의 비용

단어 표의 토큰 수 합을 N, 병합 수를 k 라 하자. 매 단계 전부 세면 $O(kN)$ 이다. 쌍의 빈도표와 쌍에서 단어로 가는 색인을 두고, 병합할 때 그 쌍이 든 단어만 다시 세면 단계의 비용이 그 단어들의 길이 합이고, 전체는 N 에 병합당 평균 단어 수를 곱한 것이다. 최대 쌍을 찾는 것은 힙으로 $O(\log)$ 이고, 이 책은 표를 훑는다.

증명. 병합은 그 쌍이 든 단어의 이웃 쌍만 바꾼다. 다른 단어의 쌍 빈도는 그대로다. 색인이 그 단어들을 주므로 그것만 빼고 더한다. ■

이 명제가 7.2 의 헤비 히터와 같은 꼴이다. 매 단계 찾는 것은 빈도 최대인 쌍이고, 그것을 정확한 표로 세면 표가 쌍의 종류 수만큼 크다. 말뭉치가 수백 GB 이면 그 표를 스케치로 두고 상위 쌍만 정확히 세는 것이 실제 구현의 모양이다.

알고리즘 9.2 바이트 쌍 부호화 학습 `dsai.bpe.train_bpe`

입력 : 단어 빈도표 W, 병합 수 k
1: 쌍 빈도표 P 와 쌍에서 단어로의 색인 I 를 한 번 만든다
2: **반복** $t = 0, \dots, k-1$ **동안**
3: $(a, b) \leftarrow \arg\max P$; 병합 순서에 더한다
4: **모든** $w \in I[(a, b)]$ **동안** w 의 옛 쌍을 P 에서 빼고, (a, b) 를 $256 + t$ 로 바꾼 뒤 새 쌍을 더한다
5: **끝 반복**
6: **끝 반복**

구현

병합 순서를 적용하는 부호화다. 매번 순위가 가장 낮은 쌍을 찾아 합친다.

지금 보는 코드 이웃 쌍 중 먼저 배운 것을 찾아 합치고, 합칠 것이 없을 때까지 되풀이한다.

구현 병합 순서 부호화 `samples/dsai/bpe.py:63`

```
def encode_by_merges(word: bytes, rank: dict[tuple[int, int], int]) -> list[int]:
    """정의 9.6. 병합 순서대로 부호화. 매번 순위가 가장 낮은 (먼저 배운) 쌍을 합친다."""
    seq = list(word)
    while len(seq) > 1:
        best, best_rank = None, None
        for i, (a, b) in enumerate(zip(seq, seq[1:])):
            r = rank.get((a, b))
            if r is not None and (best_rank is None or r < best_rank):
                best, best_rank = i, r
        if best is None:
            break
        new_id = 256 + best_rank
        seq = seq[:best] + [new_id] + seq[best + 2:]
    return seq
```

읽어 보면 `rank` 가 병합 순서의 역이다. 쌍에서 순위로. 새 토큰 번호가 `256 + best_rank` 인 것이 정의 9.5 의 불변식이다. 단어 하나의 길이가 L 이면 $O(L^2)$ 이고, 힙을 쓰면 $O(L \log L)$ 이다.

구현 병합 순서에서 어휘 되만들기 `samples/dsai/bpe.py:79`

```
def merges_to_vocab(merges: list[tuple[int, int]]) -> dict[int, bytes]:
    """토큰 번호에서 바이트 열로. 트라이에 넣기 위한 것이다."""
    vocab = {i: bytes([i]) for i in range(256)}
    for k, (a, b) in enumerate(merges):
        vocab[256 + k] = vocab[a] + vocab[b]
    return vocab
```

읽어 보면 쌍 k 개에서 문자열 $256 + k$ 개가 나온다. 이것을 9.1 의 트라이에 넣으면 최장 일치가 되고, 실험의 둘째 규칙이 그것이다.

분석

바이트 쌍 부호화는 그대로 유효하다. 1994년의 압축 알고리즘이 2016년에 어휘 만들기가 됐고, 알고리즘은 안 바뀌었다. 바뀐 것은 산출물의 뜻이다. 압축에서는 토큰 열이 결과였고, 여기서는 병합 순서가 결과이고 토큰 열은 그 순서가 만드는 부산물이다. 그래서 순서가 어휘와 함께 저장된다.

이 구조가 서는 자리는 모든 언어 모델의 입구와 출구다. 들어갈 때 텍스트가 토큰이 되고 나올 때 토큰이 텍스트가 된다. 그리고 어휘 크기 V 가 6.3 의 임베딩 테이블 행 수이고 출력층의 열 수다. 병합 수 하나가 모델의 두 큰 행렬 크기를 정한다.

현장에서

> Sennrich 등의 2016년 논문은 병합을 단어 안에서만 하고 단어 경계를 넘지 않게 했으며, 그것이 이 절의 구현과 같다 [확립] [36]. Kudo 와 Richardson 의 SentencePiece 는 공백을 특수 문자로 두어 텍스트 전체를 한 열로 다루고, 병합 순서 대신 단일 언어 모델 방식으로 어휘를 고르는 길도 준다 [확립] [23].

주의

> 실험 9.2 의 단어 표는 상위 8,000 단어다. 실제 학습은 말뭉치 전체의 단어 표를 쓰고 병합이 수만 개다. 토큰당 바이트 3.7 은 소스 코드에서 잰 값이고 자연어는 다르다.

두 절이 문자열을 토큰으로 바꿨다. 다음 절은 토큰 열을 키로 하는 트라이이고, 거기서 나누는 것은 접두사가 아니라 그 접두사의 계산 결과다. 값이 번호에서 KV 블록으로 바뀐다.

되짚기

> 바이트 쌍 부호화의 학습을 쌍 빈도표의 반복으로 정의하고, 병합 순서가 어휘와 규칙을 함께 정함을 봤다. 병합 4,096 개에서 토큰당 3.7 바이트이고, 최장 일치와 8 퍼센트의 단어가 다르게 나뉜다. 다음 절은 토큰 열의 접두사를 나누는 라디스 트리다.

9.3 라디스 트리와 접두사 공유 캐시

5.3 의 KV 캐시는 요청 하나의 것이었다. 토큰 열이 들어오면 그 열의 K 와 V 를 계산해 두고 다음 토큰이 그것을 읽는다. 두 요청이 같은 토큰 열로 시작하면 (같은 시스템 프롬프트, 같은 대화의 다음 턴) 그 접두사의 K 와 V 는 같다. 계산을 두 번 하는 것이다. 접두사가 같은지 찾는 구조가 9.1 의 트라이이고, 키가 바이트가 아니라 토큰이며, 노드에 적힌 것이 번호가 아니라 KV 블록이다.

이 절은 자식이 하나뿐인 경로를 이어 붙인 트라이, 라디스 트리를 세우고, 그 위에 KV 캐시를 올린다. 요청마다 가장 긴 공통 접두사를 찾아 그만큼을 건너뛰고, 메모리가 차면 잎부터 뗀다.

학습 목표

라디스 트리를 경로 압축 트라이로 정의하고, 토큰 열의 최장 공통 접두사 찾기가 열의 길이에만 달렸음을 보일 수 있다. 접두사 공유가 절약하는 계산량을 요청 집합의 접두사 구조로 적고, 잎부터 떼는 축출이 왜 접두사를 지키는지 설명할 수 있다.

이 말부터

라디스 트리	자식이 하나뿐인 노드들을 간선 하나로 합친 트라이. 간선에 글자 하나가 아니라 열 하나가 달린다
최장 공통 접두사	새 요청의 토큰 열과 트리 안 어느 경로가 앞에서부터 같은 가장 긴 길이
축출	캐시가 차서 무언가를 떼는 일. 잎만 떼면 남은 것은 여전히 접두사다

디딤돌

정의 9.9: 라디스 트리

간선마다 토큰 열 하나가 달린 트리다. 노드의 자식은 간선 열의 첫 토큰으로 찾고, 자식들의 첫 토큰은 서로 다르다. 새 열을 넣을 때 간선 중간에서 갈리면 그 간선을 갈리는 자리에서 잘라 노드 하나를 끼운다.

불변식 뿌리에서 노드까지의 간선 열을 이은 것이 그 노드의 열이다. 자식이 하나뿐인 노드는 뿌리와 잎 말고는 없다.

둘째 불변식이 9.1 의 트라이와 다른 점이다. 갈림이 없는 곳에는 노드가 없으므로, 노드 수가 열의 길이 합이 아니라 갈림의 수에 비례한다. 시스템 프롬프트 512 토큰을 나누는 요청 이백 개는 노드 하나와 간선 하나로 시작한다.

명제 9.10: 접두사 공유의 절약

> 요청 $r_1, \ldots, r_q$ 의 토큰 열을 차례로 넣을 때, 요청 r_i 에서 새로 계산하는 토큰 수는 $|r_i| - \mathrm{lcp}(r_i, \{r_1, \ldots, r_{i-1}\})$ 이다. lcp 는 앞선 요청들과의 최장 공통 접두사다. 전체 계산량은 라디스 트리의 간선 열 길이 합, 즉 서로 다른 접두사의 총 길이다. 요청 길이 합이 아니다.

증명. 새 요청의 접두사가 트리에 있으면 그 K, V 가 이미 있어 안 계산한다. 나머지가 새 간선이 되고, 새 간선의 길이 합이 계산량이다. 트리의 간선 열은 서로 겹치지 않으므로 그 합이 서로 다른 접두사의 총 길이다. ■

명제가 말하는 것은 절약이 요청의 성질이라는 것이다. 요청들이 접두사를 많이 나누면 절약이 크고, 전부 다르면 없다. 이 책의 작업량은 접두사를 많이 나눈다. 시스템 프롬프트가 같고, 대화의 다음 턴은 이전 턴 전체를 접두사로 갖는다.

예제 9.3: 대화 이백 개, 다섯 턴

> 시스템 프롬프트 512 토큰을 나누는 사용자 200 명이 각자 128 토큰씩 다섯 턴을 이어 간다. 요청 천 개의 토큰 합은 $200 \times (5 \times 512 + 128 \times 15) = 896{,}000$ 이다. 서로 다른 접두사의 총 길이는 $512 + 200 \times 5 \times 128 = 128{,}512$ 다. 캐시가 충분하면 계산이 7 분의 1 이다.

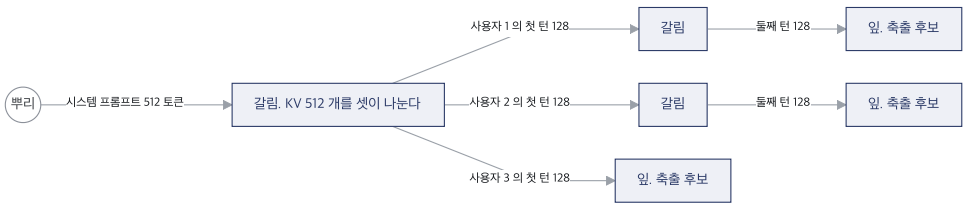


그림 9.4.. 시스템 프롬프트를 나누는 세 대화의 라디스 트리. 간선에 토큰 열이 달리고, 갈리는 자리에만 노드가 있다

예제의 7 분의 1 은 캐시가 무한할 때다. 캐시가 유한하면 떼야 하고, 무엇을 떼느냐가 다음 문제다.

균열

예제의 요청 천 개를 사용자를 섞어 넣고, 캐시 용량을 바꾸며 다시 계산한 토큰 수와 노드 수를 쟀다.

실험 9.3 접두사 공유 캐시. 용량에 따른 절약

캐시	용량 토큰	계산한 토큰	절약 퍼센트	노드 수
캐시 없음	0	896000	0.00	0
용량 20K	20000	372478	58.4	32
용량 80K	80000	228602	74.5	149
용량 400K	400000	128510	85.7	1004

측정 조건. 사용자 200 명, 시스템 프롬프트 512 토큰, 턴 다섯 개에 128 토큰씩. 요청 천 개의 토큰 합 896,000. 축출은 가장 오래 안 쓴 잎.

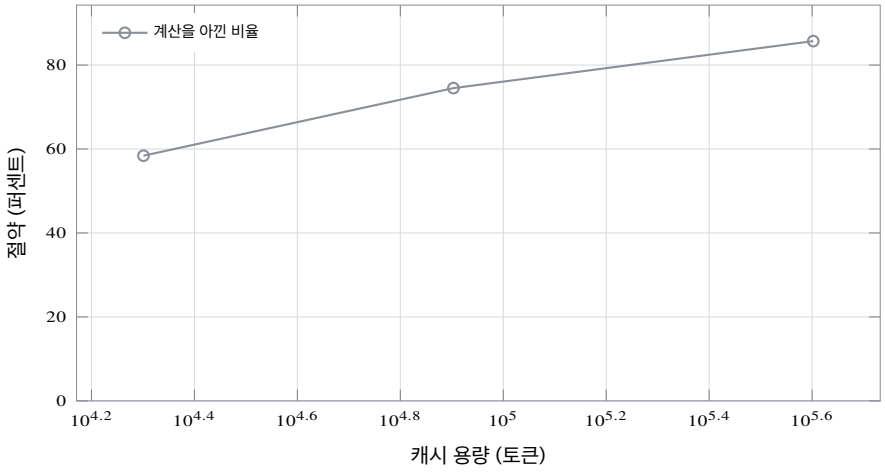


그림 9.5.. 캐시 용량에 따른 절약 비율. 용량이 접두사의 총량을 넘으면 절약이 멈춘다

용량 400K 에서 절약이 85.7 퍼센트이고 계산한 토큰이 128,510 개다. 명제 9.10 의 128,512 와 두 개 차이다. 명제의 값이 그대로 나왔다. 용량을 20K 로 줄이면 58 퍼센트다. 사용자 이백 명의 이전 턴을 다 담을 수 없어 떼고 다시 계산하는데, 그래도 시스템 프롬프트는 남아 절반 넘게 아낀다.

관찰

노드 수는 32 에서 1,004 다. 요청 천 개, 토큰 십만 개의 접두사 구조가 노드 천 개로 적힌다. 트라이로 두었으면 노드 십만 개다. 떼는 단위가 노드이고 노드가 간선 열 하나이므로, 축출 하나가 토큰 128 개를 뗀다. 턴 하나의 크기다.

이 관찰이 노드의 크기가 곧 축출의 단위라는 것을 말한다. 5.4 의 블록 테이블에서 블록이 할당의 단위였던 것과 같은 자리다.

다시 쓰기

축출을 다시 쓴다. 무엇이든 뗄 수 있는 것이 아니다.

명제 9.11: 잎 축출

> 라디스 트리에서 잎만 떼면 남은 트리는 여전히 정의 9.9 의 트리이고, 남은 모든 노드의 열은 여전히 어떤 요청의 접두사다. 안쪽 노드를 떼면 그 아래 전부의 KV 가 뜻을 잃는다. KV 는 접두사 전체에 대한 계산이기 때문이다. 가장 오래 안 쓴 잎부터 떼는 것은 15장의 LRU 를 잎에 한정한 것이다.

증명. 잎을 떼면 부모의 자식이 줄 뿐 다른 노드의 열은 안 바뀐다. 자식이 하나 남은 부모는 잎이 아니면 간선을 이어 붙여 불변식을 되찾는다. 안쪽 노드의 KV 는 그 열의 어텐션 결과이고, 자식의 KV 는 그 위에서 계산된 것이라 앞이 없으면 뒤를 쓸 수 없다. ■

이 명제가 KV 캐시가 보통의 캐시와 다른 점이다. 항목 사이에 의존이 있다. 뒤는 앞 없이 못 산다. 그래서 축출 순서는 시간만이 아니라 트리의 모양을 본다. 실험에서 시스템 프롬프트가 용량 20K 에서도 남은 이유는 그것이 뿌리 바로 아래이고 잎이 아니었기 때문이다.

알고리즘 9.3 최장 공통 접두사 찾기 `dsai.radix.RadixCache.match_prefix`

입력: 라디스 트리, 토큰 열 t

```
v ← 뿌리 ; i ← 0
조건 v 에 t[i] 로 시작하는 자식 c 가 있음 동안
    c 의 간선 열과 t[i..] 를 앞에서부터 견줘 같은 길이 j
    만약 j < 간선 길이 이면 간선을 j 에서 자르고 노드를 끼운다 ; 멈춘다
    끝 만약
    i ← i + j ; v ← c
끝 조건
반환 i                                  ▷ 앞 i 토큰의 KV 를 다시 쓴다
```

구현

노드는 간선 열과 자식 dict 과 마지막 사용 시각이다. 자식은 첫 토큰으로 찾는다.

지금 보는 코드 간선 열을 따라 내려가며 같은 길이를 세고, 중간에서 갈리면 노드를 끼운다.

구현 최장 공통 접두사 찾기 `samples/dsai/radix.py:29`

```
    def match_prefix(self, tokens: list[int]) -> tuple[int, list[RadixNode]]:
        """알고리즘 9.3. 가장 긴 공통 접두사의 길이와 지나온 노드들."""
        node, i, path = self.root, 0, [self.root]
        while i < len(tokens):
            child = node.children.get(tokens[i])
            if child is None:
                break
            lab = child.label
            j = 0
            while j < len(lab) and i + j < len(tokens) and lab[j] == tokens[i + j]:
                j += 1
            if j < len(lab):                    # 간선 중간에서 갈린다. 노드를 쪼갠다
                child = self._split(node, child, j)
            path.append(child)
            i += len(child.label)
            node = child
        self.clock += 1
        for p in path:
            p.last_used = self.clock
        return i, path
```

읽어 보면 안쪽 `while` 이 간선 열과 요청을 견주는 자리다. `j < len(lab)` 이면 간선 중간이고 `_split` 이 간선을 잘라 노드를 끼운다. 지나온 노드 전부의 `last_used` 를 갱신해 LRU 가 접두사를 안 떼게 한다.

구현 잎 축출 `samples/dsai/radix.py:70`

```
    def _evict_one(self) -> None:
        """명제 9.11. 가장 오래 안 쓴 잎을 뗀다. 잎만 떼야 접두사가 남는다."""
        best, best_parent = None, None
        stack = [(self.root, None)]
        while stack:
            node, parent = stack.pop()
            if not node.children and node is not self.root:
                if best is None or node.last_used < best.last_used:
                    best, best_parent = node, parent
            for c in node.children.values():
                stack.append((c, node))
        if best is None:
            return
        del best_parent.children[best.label[0]]
        self.size -= len(best.label)
```

읽어 보면 잎만 후보다. 가장 오래 안 쓴 잎을 찾는 데 트리를 다 훑는데, 실제 구현은 잎을 따로 LRU 목록에 둔다. 연습문제다.

분석

라디스 트리는 그대로 유효하다. 1968 년의 경로 압축이 그대로이고, 키가 바이트에서 토큰으로 바뀌었을 뿐이다. 바뀐 것은 노드에 적힌 값이다. 번호가 아니라 KV 블록이고, 그 값에 의존 관계가 있어 축출이 잎에 한정된다. 이것이 5.3 의 추가 전용 캐시를 여러 요청이 나누는 구조다.

이 구조가 서는 자리는 추론 서버다. Zheng 등은 2024 년에 이 트리 위의 KV 캐시로 여러 요청과 여러 턴이 접두사를 자동으로 나누게 했고, 접두사가 많은 작업에서 처리량이 몇 배 올랐다고 보고했다 [확립] [37]. 실험의 7 분의 1 이 그 몇 배의 자료 구조 쪽 근거다.

현장에서

> Zheng 등의 구현은 잎을 LRU 로 떼고, 지금 어떤 요청이 쓰고 있는 노드는 떼지 않도록 참조 수를 둔다 [확립] [37]. 이 절의 구현은 참조 수를 두지 않았다. 요청이 끝난 뒤에 다음 요청이 오는 순차 실행이기 때문이다.

주의

> 실험 9.3 의 요청은 사용자를 섞었지만 턴 순서는 지켰다. 실제 서버는 턴 사이에 긴 시간이 있고 그동안 다른 요청이 캐시를 밀어낸다. 절약은 도착 순서의 함수이고, 이 실험은 그중 하나다.

트라이 위에 KV 를 올렸다. 다음 절은 트라이 위에 오토마톤을 올린다. 노드에 값을 두는 대신 노드를 지나는 상태를 보고, 죽는 상태 아래는 통째로 건너뛴다.

되짚기

> 라디스 트리를 정의하고 접두사 공유의 절약이 서로 다른 접두사의 총 길이임을 보였다. 요청 천 개에서 계산이 7 분의 1 이 됐고, 명제와 두 토큰 차이였다. 축출은 잎만 뗀다. 다음 절은 트라이 위의 오토마톤이다.

9.4 제약 디코딩. 트라이 위의 오토마톤

언어 모델이 다음 토큰을 고를 때 어휘 전부가 후보다. 출력이 정해진 형식 (숫자, JSON, 정규식이 정한 문자열) 이어야 하면 그 형식에 안 맞는 토큰을 후보에서 빼야 한다. 형식은 유한 상태 오토마톤으로 적을 수 있고, 지금 상태에서 토큰 하나를 먹어 죽지 않으면 허용이다. 문제는 그것을 매 단계 어휘 V 개에 대해 확인하는 비용이다.

Willard 와 Louf 는 2023 년에 이 확인을 미리 해 두자고 했다. 상태마다 허용 토큰 집합을 한 번 만들어 표로 두면, 디코딩 때는 표 하나를 읽는다 [확립] [38]. 이 절은 그 표를 만드는 비용을 9.1 의 트라이로 줄인다. 오토마톤과 트라이를 함께 내려가면 공통 접두사는 한 번만 확인된다.

학습 목표

바이트 단위 유한 상태 오토마톤을 전이표로 정의하고, 상태 s 에서 토큰 t 가 허용된다는 조건을 적을 수 있다. 허용 집합을 어휘 전부 훑기와 트라이 위의 동시 내려가기로 만드는 비용을 견주고, 왜 후자가 어휘 크기와 무관에 가까운지 설명할 수 있다.

이 말부터

오토마톤	상태 집합과 바이트마다의 전이. 시작 상태에서 바이트를 먹으며 옮겨 가고, 갈 데가 없으면 죽는다
허용 집합	상태 s 에서 먹어도 안 죽는 토큰들의 집합. 디코딩 때 이 밖의 토큰은 확률을 0 으로 둔다
마스크 표	상태마다 허용 집합을 비트 마스크로 적은 표. 크기가 상태 수 곱하기 어휘 크기다

디딤돌

정의 9.12: 바이트 오토마톤과 허용 집합

상태 $\{0, \dots, m-1\}$, 시작 상태, 받아들이는 상태 집합, 그리고 전이표 $\delta[s][b] \in \{0, \dots, m-1\} \cup \{\bot\}$ 이다. $\bot$ 은 죽음이다. 토큰 t 의 바이트 열 $b_1 \dots b_\ell$ 에 대해 $\delta^*(s, t) = \delta(\dots \delta(s, b_1) \dots, b_\ell)$ 이고, $\delta^*(s, t) \neq \bot$ 이면 t 는 s 에서 허용된다. 상태 s 의 허용 집합은 $A(s) = \{t \in \mathcal{V} : \delta^*(s, t) \neq \bot\}$ 다.

불변식 시작 상태에서 허용 집합의 토큰만 이어 먹으면 오토마톤이 죽지 않는다. 받아들이는 상태에서 멈추면 출력이 형식에 맞는다.

불변식이 제약 디코딩의 정확성이다. 매 단계 허용 집합 밖을 0 으로 두면 죽는 열이 안 나온다. 남은 것은 $A(s)$ 를 만드는 비용이고, 정의대로 하면 토큰마다 바이트 열을 먹여 본다.

명제 9.13: 허용 집합의 비용

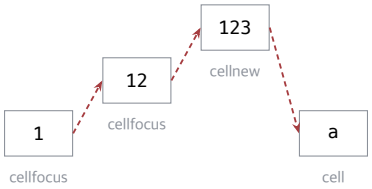
> 어휘 전부를 훑으면 상태 하나에 어휘의 바이트 길이 합 S 만큼의 전이다. 상태 m 개면 mS 다. 트라이 위에서 오토마톤을 함께 내려가면 상태 하나에 트라이의 노드 수 이하이고, 죽는 간선 아래는 안 본다. 실제 비용은 오토마톤이 살아서 지나는 노드 수이고, 제약이 빡빡할수록 작다.

증명. 전부 훑기는 토큰마다 그 길이만큼 전이하므로 길이 합이다. 트라이 내려가기는 노드마다 전이 하나이고, 같은 접두사는 노드 하나이므로 접두사의 중복이 빠진다. 죽은 상태에서는 어떤 바이트도 살리지 못하므로 그 아래 부분 트리 전체를 건너뛴다. ■

명제의 마지막 문장이 이 절의 실험이다. 숫자 형식의 오토마톤은 어휘 대부분을 첫 바이트에서 죽이므로 뿌리의 자식 256 개 중 열둘만 내려간다. 그 아래에서도 숫자와 점만 살아남아 노드 수십 개다. 어휘가 만 개든 십만 개든 그 수십 개다.

예제 9.4: 숫자 형식, 어휘 4,352 개

> 정수 또는 소수의 오토마톤은 상태 넷이다. 어휘의 바이트 길이 합이 25,785 이면 전부 훑기는 상태마다 25,785 전이, 넷에 십만 전이다. 트라이의 노드 9,685 개 중 숫자와 점으로만 이루어진 경로는 수십 개이고, 트라이 방법은 상태마다 그 수십 개만 본다. 천 배다.



숫자 형식의 상태 1 에서. 1, 12, 123 은 허용이고 a 아래는 안 본다

그림 9.6.. 트라이 위의 오토마톤. 뿌리에서 숫자 간선만 살아 내려가고, 글자 간선 아래는 통째로 버린다

예제의 천 배가 실측에서 얼마인지, 그리고 어휘가 네 배 커질 때 두 방법이 어떻게 갈리는지 잰다.

균열

9.2 의 어휘 1,280 개와 4,352 개에서 숫자 형식의 상태 넷의 허용 집합을 두 방법으로 만들었다. 두 방법의 집합이 같은지 확인하고 시간을 쟀다.

실험 9.4 상태 넷의 허용 집합 만들기. 전부 훑기와 트라이

어휘 크기	어휘 바이트 합	트라이 노드	상태 1 허용 토큰	전부 훑기 ms	트라이 ms
1280	4748	2191	18	5.04	0.85
4352	25785	9685	42	18.1	0.86

측정 조건. 파이썬 표준 라이브러리 소스로 학습한 어휘. 문법은 정수 또는 소수. 시간은 상태 넷 전부의 집합을 만드는 것이고 세 번 중앙값.

어휘 1,280 개에서 여섯 배, 4,352 개에서 스물한 배다. 전부 훑기는 어휘가 커지자 5 밀리초에서 18 밀리초로 늘었고, 트라이는 0.85 밀리초 그대로다. 명제 9.13 대로다. 트라이 방법의 비용은 살아서 지나는 노드 수이고, 그것은 어휘 크기가 아니라 형식이 정한다.

관찰

상태 1 의 허용 토큰이 18 개와 42 개다. 어휘 4,352 개 중 42 개만 살고 나머지 4,310 개는 확률을 0 으로 둔다. 디코딩 때 어휘 크기의 마스크를 곱하는 값은 남지만, 마스크를 만드는 값은 이 절에서 사라졌다.

이 관찰이 미리 만든 표의 뜻이다. 상태 수 곱하기 어휘 크기의 비트가 메모리이고, 디코딩 단계의 비용은 그 표의 한 줄을 읽는 것이다.

다시 쓰기

이 절의 다시 쓰기는 표를 어디까지 미리 만드느냐다. 상태가 넷이면 표가 넷 줄이고, 정규식이 복잡하면 상태가 수천이다.

명제 9.14: 마스크 표의 크기와 지연

상태 m 개, 어휘 V 개의 마스크 표는 mV 비트다. $m = 10^3$, $V = 10^5$ 이면 12.5 MB 이고, 디코딩 단계마다 그 표의 한 줄 V 비트를 읽어 로짓에 곱한다. 표를 미리 만드는 비용은 상태마다 명제 9.13 의 트라이 내려가기이고, 상태들이 같은 접두사에서 같은 전이를 하면 그 결과를 나눈다.

증명. 줄 하나가 허용 집합 하나이고 비트 V 개다. 디코딩은 5.4 의 어휘 크기 출력층 뒤에 마스크를 곱하므로 줄 하나를 읽는다. ■

이 명제가 제약 디코딩이 거의 공짜인 이유다. 표를 한 번 만들고, 단계마다 비트 V 개를 읽는다. 출력층이 어차피 V 개의 로짓을 만드니 그 옆에 붙는 값이다. 표를 만드는 값은 트라이가 지웠다.

알고리즘 9.4 트라이 위의 오토마톤 `dsai.constrained.allowed_by_trie`

입력: 오토마톤 δ, 트라이, 상태 s
스택 $\leftarrow [(\text{뿌리}, s)]$; $A \leftarrow \emptyset$
조건 스택이 비지 않음 **동안**
    $(v, s) \leftarrow$ 스택에서 꺼냄; v 에 번호가 있으면 A 에 더한다
    모든 v 의 자식 (b, c) **동안** $\delta[s][b] \neq \bot$ 이면 $(c, \delta[s][b])$ 를 스택에 넣는다
    끝 반복
끝 조건
반환 A

구현

오토마톤은 (상태 수, 256) 배열이다. 트라이 내려가기는 5.1 의 명시적 스택이다.

지금 보는 코드 트라이의 노드와 오토마톤의 상태를 짝지어 내려간다. 죽는 자식은 스택에 안 넣는다.

구현 전이표 `samples/dsai/constrained.py:18`

```
class DFA:
    """정의 9.12. 상태 집합과 바이트 전이표. 전이표는 (상태 수,
    256) 배열이고 -1 이 죽음이다.
    """

    def __init__(self, n_states: int, start: int, accept: set[int]):
        self.delta = np.full((n_states, 256), DEAD, dtype=np.int64)
        self.start = start
        self.accept = accept

    def add(self, s: int, chars: bytes, t: int) -> None:
        for c in chars:
            self.delta[s, c] = t

    def step(self, s: int, data: bytes) -> int:
        for c in data:
            s = self.delta[s, c]
            if s == DEAD:
                return DEAD
        return s
```

읽어 보면 `delta` 의 -1 이 $\bot$ 이다. 같은 파일의 `number_dfa` 가 실험의 상태 넷을 만들고, 받아들이는 상태는 정수부와 소수부다.

구현 전부 훑기와 트라이 내려가기 `samples/dsai/constrained.py:54`

```
def allowed_by_scan(dfa: DFA, vocab: dict[int, bytes], state: int) -> set[int]:
    """디딤돌. 어휘 전부를 하나씩 상태에서 먹여 본다."""
    return {t for t, w in vocab.items() if dfa.step(state, w) != DEAD}


def allowed_by_trie(dfa: DFA, trie: Trie, state: int) -> set[int]:
    """알고리즘 9.4. 트라이와 오토마톤을 같이 내려간다. 죽는 가지는 통째로 버린다."""
    out: set[int] = set()
    stack = [(0, state)]
    while stack:
        node, s = stack.pop()
        if trie.token[node] != NIL:
            out.add(trie.token[node])
        for b, child in trie.children_of(node):
            t = dfa.delta[s, b]
            if t != DEAD:
                stack.append((child, t))
    return out
```

읽어 보면 `allowed_by_scan` 은 정의 그대로 한 줄이다. `allowed_by_trie` 는 스택에 (노드, 상태) 짝을 넣고, `t != DEAD` 인 자식만 넣는다. 실험의 스물한 배가 그 조건문 하나다.

분석

오토마톤 위의 트라이 탐색은 조정해서 유효하다. Aho 와 Corasick 이 1975 년에 여러 패턴을 한 오토마톤에 넣어 텍스트를 한 번 훑는 방법을 세웠고 [확립] [39], 이 절은 그 방향을 뒤집었다. 텍스트 대신 어휘를 트라이에 넣고, 패턴의 오토마톤을 그 위에서 돌린다. 같은 두 부품이고 무엇을 트리에 넣느냐가 다르다.

이 구조가 서는 자리는 구조화된 출력이다. JSON 을 내야 하는 도구 호출, 정해진 선택지 중 하나를 골라야 하는 분류, 형식이 정해진 코드. 어휘 크기 V 가 십만 단위이고 단계마다 어휘 전부를 확인하면 디코딩이 그 확인에 잡아먹힌다. 표를 미리 만들면 사라지고, 표를 트라이로 만들면 만드는 값도 사라진다.

현장에서

> Willard 와 Louf 의 2023 년 논문은 정규식을 오토마톤으로 바꾸고 상태마다 어휘의 허용 집합을 미리 색인해 두는 방법을 제안했고, 문맥 자유 문법으로 넓히는 길도 다뤘다 [확립] [38]. Zheng 등의 2024 년 시스템은 같은 오토마톤을 압축해 여러 토큰을 한 번에 내는 데 썼다 [확립] [37].

주의

> 실험 9.4 의 오토마톤은 상태 넷의 작은 것이다. 상태가 수천인 정규식이나 문맥 자유 문법에서는 표가 커지고, 상태마다 트라이를 내려가는 값이 상태 수에 비례해 는다. 그리고 이 절은 바이트 단위 오토마톤이다. 유니코드 문자가 여러 바이트일 때 토큰 경계가 문자 중간에 올 수 있고, 그 처리는 이 실험에 없다.

트라이 하나가 네 절을 이었다. 문자열의 최장 일치, 어휘를 만드는 병합, 토큰 열의 접두사 공유, 그리고 오토마톤의 허용 집합. 다음 장은 순서와 접두사 대신 최대 하나만 지키는 트리다. 힙이고, 상위 k 개를 고르는 일과 확률로 뽑는 일이 거기서 나온다.

되짚기

> 바이트 오토마톤과 허용 집합을 정의하고, 트라이 위에서 오토마톤을 내려가면 비용이 어휘 크기가 아니라 형식에 달렸음을 보였다. 어휘 4,352 개에서 스물한 배였고 어휘를 키워도 시간이 그대로였다. 다음 장은 힙이다.

9장을 닫으며

이 장이 세운 구조는 넷이다. 접두사를 나누는 트라이와 최장 일치, 병합 순서라는 어휘, 경로를 압축한 라디스 트리 위의 KV 캐시, 그리고 트라이 위를 함께 내려가는 오토마톤. 넷 다 정의 9.1 위에 있고, 넷 다 같은 성질을 쓴다. 같은 앞부분은 한 번만 저장하고 한 번만 확인한다.

네 실험의 숫자를 적어 둔다. 최장 일치 토큰화는 트라이가 dict 방법보다 여섯 배 빨랐고, 비용을 정하는 것이 어휘의 가장 긴 항목이 아니라 실제로 뽑히는 항목이었다. 병합 4,096 개에서 토큰당 3.7 바이트였고, 같은 어휘에서 두 규칙이 단어의 8 퍼센트를 다르게 나눴다. 요청 천 개의 KV 계산이 접두사 공유로 7 분의 1 이 됐고 명제와 두 토큰 차이였다. 숫자 형식의 허용 집합을 만드는 데 트라이가 스물한 배 빨랐고 어휘를 네 배로 키워도 시간이 그대로였다.

정정 하나를 남긴다. 병합 학습을 처음에는 정의 그대로 매 단계 모든 단어의 쌍을 다시 세게 썼다. 병합 천 개에 85 초가 걸렸고 4,096 개는 재지 못했다. 쌍 빈도표와 쌍에서 단어로 가는 색인을 두고 병합할 때 그 쌍이 든 단어만 다시 세자 천 개가 4 초, 4,096 개가 13 초가 됐다. 정의는 알고리즘이 아니다. 정의는 무엇을 세는지 말하고, 알고리즘은 무엇을 다시 안 세는지 말한다.

옛 구조의 판정

> 트라이는 그대로 유효하다. 1960 년의 정의와 1968 년의 경로 압축이 그대로이고, 실험이 명제를 따른다. 근거는 실험 9.3 과 9.4 다. 접두사 공유의 절약이 명제의 값과 같았고, 오토마톤의 비용이 어휘 크기와 무관했다. 바뀐 것은 노드에 적히는 것이다. 번호에서 KV 블록으로, 그리고 오토마톤의 상태로.

이 장의 요약

> 1. 트라이의 탐색은 문자열 길이만큼이고 어휘 크기와 무관하다. 최장 일치가 트라이를 한 번 내려가는 것이다 9.1
> 2. 바이트 쌍 부호화의 산출물은 병합 순서이고, 그 순서가 어휘와 부호화 규칙을 함께 정한다. 최장 일치와 답이 다르다 9.2
> 3. 라디스 트리는 갈림에만 노드를 둔다. KV 캐시를 올리면 절약이 서로 다른 접두사의 총 길이로 정해진다 9.3
> 4. KV 는 앞 없이 뒤를 못 쓴다. 그래서 축출은 잎만 뗀다 9.3
> 5. 오토마톤을 트라이 위에서 내려가면 허용 집합의 비용이 형식이 정하고 어휘 크기가 안 정한다 9.4
>
> 다음 장에서 뒤집히는 것: 이 장까지의 트리는 전부를 저장했다. 10 장의 힙은 최대 하나만 약속하고 나머지의 순서를 버린다. 그 버림이 상위 k 개를 고르는 값을 정한다.

필자의 판단

> 필자는 토크나이저를 자료 구조 수업의 첫 문자열 예제로 쓴다. 학생이 매일 쓰는 모델의 입구가 트라이와 병합 순서라는 것을 알면, 트라이가 사전 검색 예제였을 때보다 오래 남는다. 그리고

같은 어휘에서 두 규칙이 다른 답을 낸다는 것을 직접 재게 한다. 그것이 토크나이저를 모델 파일의 일부로 다루게 만드는 가장 빠른 길이다.

열린 문제

하나, 실험 9.1 의 트라이는 자식을 해시 하나에 두었다. 명제 9.4 의 층별 표현을 구현해 라인 수가 얼마나 주는지 재지 않았다.
둘, 실험 9.3 의 축출은 잎을 찾으려고 트리를 다 훑는다. 잎을 LRU 목록에 두면 축출이 상수인데, 그 목록과 트리를 함께 유지하는 값을 재지 않았다.
셋, 실험 9.4 는 바이트 오토마톤이라 여러 바이트 문자의 경계를 다루지 않았다. 토큰이 문자 중간에서 끝날 때 오토마톤을 어떻게 이어야 하는지 이 장은 답하지 않았다.

연습문제

9.1 기초 어휘 항목 50,000 개, 평균 길이 6 바이트의 트라이 노드 수 상한을 명제 9.2 로 구하라. 노드마다 자식 배열 256 칸을 두면 메모리가 얼마이고, 간선당 24 바이트의 해시로 두면 얼마인가.

9.2 기초 단어 표의 토큰 수 합이 10^7 이고 병합이 3×10^4 개일 때, 매 단계 전부 세는 학습과 색인을 둔 학습의 비용을 명제 9.8 로 견줘라. 병합당 평균 단어 수를 200 으로 두라.

9.3 기초 시스템 프롬프트 2,000 토큰을 나누는 사용자 1,000 명이 각자 200 토큰씩 열 턴을 이어 갈 때, 캐시 없이 계산하는 토큰 수와 명제 9.10 의 서로 다른 접두사 총 길이를 구하라.

9.4 기초 상태 2,000 개, 어휘 128,000 개의 마스크 표는 몇 MB 인가. 명제 9.14 로 구하고, 디코딩 한 단계에 읽는 비트 수를 적어라.

9.5 응용 `dsai.trie.Trie` 의 자식 표현을 뿌리의 자식만 256 칸 배열로 바꿔라. 실험 9.1 의 토큰화 시간이 얼마나 바뀌는지 재고, 어느 층까지 배열로 두는 것이 나은지 논하라.

9.6 응용 `dsai.bpe.encode_by_merges` 를 힙으로 바꿔라. 이웃 쌍의 순위를 힙에 넣고 최소를 꺼내 합친 뒤 새로 생긴 쌍만 넣는다. 긴 단어에서 시간이 얼마나 주는지 재라.

9.7 응용 `dsai.radix.RadixCache` 에 참조 수를 더하라. 요청이 쓰는 동안 그 경로의 노드는 떼지 않고, 요청이 끝나면 참조를 푼다. 실험 9.3 을 요청 둘씩 겹쳐 돌려 참조 수 없이는 무엇이 잘못되는지 시험으로 보여라.

9.8 응용 `dsai.constrained.number_dfa` 를 따옴표로 감싼 문자열 형식 (`"` 뒤에 따옴표 아닌 바이트들, 다시 `"`) 으로 바꾸고, 상태마다 허용 집합의 크기와 트라이가 지나는 노드 수를 재라. 숫자 형식과 왜 다른가.

9.9 탐구 병합 순서 부호화가 최장 일치보다 학습 때의 분포에 맞는 이유를 정의 9.6 의 불변식으로 설명하라. 최장 일치로 학습한 모델에 병합 순서로 넣으면 무엇이 달라지는가.

9.10 탐구 KV 캐시의 축출이 잎에 한정되는 것과 5.4 의 블록 테이블에서 블록이 참조 수를 갖는 것을 견줘라. 두 구조를 하나로 합치면 무엇이 노드이고 무엇이 블록인가.

더 읽을 것

[34] 트라이의 원전. 열 쪽이고, 접근 시간과 갱신과 길이가 다른 키에 대한 장점이 그대로 적혀 있다. 9.1
[35] 경로 압축 트라이의 원전. 라디스 트리의 불변식과 비트 단위의 갈림이 여기서 나왔다. 9.3
[36] 바이트 쌍 부호화를 어휘 만들기에 가져온 논문. 단어 안에서만 병합하는 규칙이 이 장의 구현이다. 9.2
[37] 라디스 트리 위의 KV 캐시와 압축 오토마톤. 9.3 절과 9.4 절의 현장이다. 9.3, 9.4
[38] 정규식을 오토마톤으로 바꿔 어휘의 허용 집합을 미리 색인하는 방법. 9.4 절의 출발이다. 9.4
[39] 여러 패턴을 한 오토마톤에 넣는 원전. 9.4 절이 뒤집은 방향의 원래 방향이다. 9.4

10장

힙, 선택, 표본 추출

1964년의 두 쪽짜리 알고리즘 하나는 배열 하나를 트리로 읽고 최대만 맨 위에 두는 것으로 정렬을 만들었다 [확립] [40]. 그 트리가 지금 디코딩의 매 단계에 있다. 어휘 십만 개의 로짓에서 상위 몇 개를 고르는 자리, 빔의 후보를 남기는 자리, 확률의 꼬리를 자르고 뽑는 자리, 그리고 요청을 줄 세우는 자리. 넷 다 전부를 정렬하지 않는다. 필요한 것 하나 또는 몇 개만 꺼낸다. 그 값이 이 장의 잣대다.

이 장은 0.2 의 첫째 가정을 넷째 자리에서 다시 본다. 비교 횟수로는 힙이 이기는 자리에서 배열 한 번 훑기가 이긴다. 층을 오르내리는 값이 비교보다 크다. 그리고 확률로 답하는 것이 7장에 이어 여기서 둘째로 나온다. 뽑기는 답이 정해져 있지 않고, 분포가 답이다.

이 장에서 세우는 것

10.1 이진 힙과 상위 k 개. 정렬, 힙, 선택의 세 비용. 선택이 정렬보다 여덟 배

10.2 빔 탐색. 단계마다 상위 B 개 선택과 부모 배열. 빔 8 이 최선을 45 퍼센트

10.3 누적합과 별칭표. 뽑기가 세 배 빠르고, top-p 는 남는 수가 정한다

10.4 우선순위 스케줄러. 짧은 것 먼저의 최적성과 굶김, 나이 가중. 평균 다섯 배와 최대 네 배

10.1 이진 힙과 top-k. 전부 정렬하지 않는 값

8장과 9장의 트리는 전부를 저장하고 순서나 접두사를 지켰다. 힙은 약속을 하나로 줄인다. 최대 하나만 언제나 맨 위에 있다. 그 약속만 지키면 배열 하나에 트리를 담을 수 있고, 최대를 꺼내고 새것을 넣는 데 로그 걸음이면 된다. 나머지의 순서는 모른다.

이 책의 작업량에서 이 약속이 서는 자리는 상위 k 개를 고르는 일이다. 어휘 십만 개의 로짓에서 상위 50 개, 후보 수만 개의 점수에서 상위 k 개. 정렬하면 $n \log n$ 이고, 힙이면 $n \log k$ 이고, 선택이면 n 이다. 이 절은 셋을 세우고 잰다.

학습 목표

이진 힙을 배열 위의 완전 이진 트리와 힙 불변식으로 정의하고, 넣기와 꺼내기가 $O(\log n)$ 임을 보일 수 있다. 상위 k 개를 고르는 세 방법의 비용을 적고, 어느 것이 이 책의 작업량에서 이기는지 실측으로 말할 수 있다.

이 말부터

힙 불변식	모든 노드가 두 자식 이상이라는 조건. 최대가 뿌리에 있다는 것 말고는 아무 순서도 약속하지 않는다
가라앉히기	뿌리에 온 새 값을 더 큰 자식과 바꾸며 내리는 걸음. 꺼내기의 전부다
선택	정렬하지 않고 k 번째 값을 기준으로 앞뒤를 가르는 일. 기대 $O(n)$ 이다

디딤돌

정의 10.1: 이진 힙

배열 $A[0..n)$ 을 완전 이진 트리로 읽는다. 노드 i 의 자식은 $2i+1$ 과 $2i+2$, 부모는 $\lfloor (i-1)/2 \rfloor$ 다. 8.4 의 아이칭거 배치와 같은 번호다.

불변식 모든 $i > 0$ 에 대해 $A[\lfloor (i-1)/2 \rfloor] \geq A[i]$ 다. 따라서 $A[0]$ 이 최대다.

넣기는 끝에 붙이고 부모보다 크면 바꾸며 올린다. 꺼내기는 $A[0]$ 을 빼고 마지막 원소를 뿌리에 놓은 뒤 더 큰 자식과 바꾸며 내린다.

불변식이 약속하는 것은 뿌리 하나다. 형제 사이의 순서, 서로 다른 부분 트리 사이의 순서는 없다. 그래서 정렬보다 싸다. 얼마나 싼지는 높이가 정한다. 완전 이진 트리의 높이는 $\lceil \log_2 (n+1) \rceil$ 이고, 넣기와 꺼내기는 높이만큼 걸어 내려간다.

명제 10.2: 힙 연산의 비용

> 원소 n 개의 힙에서 넣기와 꺼내기는 각각 비교 $O(\log n)$ 번이다. 배열 n 개를 힙으로 만드는 것은 아래에서부터 가라앉히면 $O(n)$ 이다. 이동량은 걸음마다 라인 하나이지만 위쪽 $\log_2 B$ 층은 8.4 의 명제대로 라인 하나에 든다.

증명. 넣기는 잎에서 뿌리까지, 꺼내기는 뿌리에서 잎까지 층마다 비교 하나 또는 둘이므로 높이에 비례한다. 만들기의 비용은 층 h 의 노드 $n/2^{h+1}$ 개가 각각 h 걸음이라 $\sum_h nh/2^{h+1} = O(n)$ 이다. ■

명제에서 읽을 것은 $\log n$ 이 힙의 크기라는 것이다. 상위 k 개만 필요하면 힙을 k 개로 두고, 로그가 $\log k$ 가 된다.

명제 10.3: 상위 k 개의 세 비용

> 원소 n 개에서 가장 큰 k 개를 고르는 데 전부 정렬은 $O(n \log n)$, 크기 k 의 최소 힙은 $O(n \log k)$, 선택은 기대 $O(n)$ 이다. 셋 다 이동량은 배열을 한 번 훑는 순차 접근이고, 힙 k 개는 캐시에 든다. 차이는 비교 횟수와 분기 예측이다.

증명. 힙 방법은 원소마다 힙의 최소와 견주고, 크면 바꿔 가라앉히므로 $\log k$ 다. 대부분의 원소는 최소보다 작아 비교 하나로 끝난다. 선택은 퀵 정렬의 분할을 한쪽만 되풀이하므로 기대 $n + n/2 + \cdots = 2n$ 이다. ■

명제의 셋은 비교 횟수의 순위이고, 시간의 순위는 언어와 분기 예측이 정한다. 먼저 숫자로 셈을 해 본다.

예제 10.1: 어휘 128,000 개에서 50 개

> $n = 128{,}000$, $k = 50$ 이면 정렬은 비교 $n \log_2 n = 2.2 \times 10^6$, 힙은 $n \log_2 k = 7.2 \times 10^5$ 이 상한이고 실제로는 대부분 비교 하나라 n 근처다. 선택은 $2n$ 이다. 셋 다 배열 한 번 읽기이니 이동량은 같고 비교만 다르다.

그림 10.1. 선택. 상위 셋을 앞으로 모으고 나머지는 정렬하지 않는다. 힙은 원소가 하나씩 올 때의 도구다

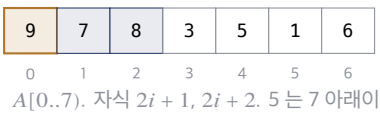


그림 10.2. 원소 일곱 개의 최대 힙. 뿌리가 최대이고 형제 사이는 순서가 없다. 배열 번호가 곧 트리 자리다

예제의 비교 횟수 차이가 시간으로 얼마나 나는지 잰다. 셋 중 무엇이 이기는지는 언어에 달렸으므로 두 언어로 잰다.

균열

표준 정규 로짓 128,000 개에서 상위 50 개를 세 방법으로 골랐다. 힙은 파이썬 루프라 크기 8,000 에서 파이썬 정렬과도 견줬다.

실험 10.1 상위 50 개 고르기. 정렬, 힙, 선택

방식	V	k	ms
전부 정렬 (NumPy)	128000	50	20.0
부분 정렬 (NumPy argpartition)	128000	50	2.37
크기 k 힙 (파이썬)	128000	50	51.4
전부 정렬 (파이썬 sorted)	8000	50	3.07
크기 k 힙 (파이썬), V=8000	8000	50	4.71

측정 조건. 세 방법의 상위 50 개 집합이 같음을 확인했다. NumPy 둘은 C 루프이고 힙은 파이썬 루프다. 다섯 번 중앙값.

선택이 정렬보다 여덟 배 빠르다. 같은 C 루프에서 n 과 $n \log n$ 의 차이에 분기가 적은 분할의 이점이 더해졌다. 파이썬 힙은 51 밀리초로 NumPy 정렬보다 느리다. 언어의 값이 알고리즘의 값을 덮었다. 같은 파이썬끼리 견주면 힙이 정렬보다 느린 것도 보인다. 원소 8,000 개에서 4.7 대 3.1 밀리초다.

관찰

힙이 파이썬 정렬보다 느린 것은 파이썬 정렬이 C 이기 때문이다. 알고리즘의 비교 횟수는 힙이 적지만, 파이썬 문장 하나가 C 비교 수십 개의 값이다. 이 관찰은 8.2 와 같다. 층을 오르내리는 값이 층이 아끼는 비교보다 크면 트리는 진다.

이 관찰이 top-k 의 실제 구현이 힙이 아닌 이유다. 가속기에서는 선택이나 기수 정렬 꼴의 한 번 훑기가 쓰이고, 힙은 스트림에서 하나씩 올 때 남는다.

다시 쓰기

이 절의 다시 쓰기는 상위 k 개를 정렬 없이 고르는 선택이다. 8 장의 이진 탐색이 배열의 반을 버리듯, 선택은 한쪽을 버린다.

명제 10.4: 선택의 기대 비용

> 무작위 기준으로 분할하면 남는 쪽의 기대 크기가 $3n/4$ 이하이므로 총 비교는 기대 $O(n)$ 이다. 최악은 $O(n^2)$ 이고, 중앙값의 중앙값을 기준으로 삼으면 최악도 $O(n)$ 이다. k 번째 값을 알면 상위 k 개는 그 값 이상인 원소이고 순서 없이 나온다. 순서가 필요하면 k 개만 정렬해 $O(k \log k)$ 를 더한다.

증명. 분할마다 기준보다 큰 쪽과 작은 쪽 중 k 번째가 든 쪽만 남긴다. 기준이 무작위이면 남는 쪽의 기대 크기가 $3n/4$ 이하이고, 등비급수로 $4n$ 이 상한이다. ■

이 명제가 NumPy 의 `argpartition` 이고 실험의 2.4 밀리초다. 어휘 크기 V 의 로짓에서 상위 k 개를 고르는 것이 디코딩의 매 단계이므로, 이 명제 하나가 매 단계에서 정렬을 뺀다.

알고리즘 10.1 크기 k 힙으로 상위 k 개 `dsai.heap.topk_heap`

> **입력 :** 배열 $x[0..n)$, k
> 앞 k 개로 최소 힙 H 를 만든다
> **모든** $v \in x[k..n)$ **동안**
> **만약** $v > H$ 의 최소 **이면** H 의 뿌리를 v 로 바꾸고 가라앉힌다
> **끝 만약**
> **끝 반복**
> **반환** H 의 원소 k 개

구현

힙은 리스트 하나다. 넣기와 꺼내기와 가라앉히기가 전부이고, 최소 힙은 부호를 바꿔 최대 힙에 넣는다.

지금 보는 코드 뿌리에 온 값을 더 큰 자식과 바꾸며 내린다. 자식 번호는 $2i+1$, $2i+2$ 의 산수다.

구현 꺼내기와 가라앉히기 `samples/dsai/heap.py:31`

```
    def pop(self) -> float:
        a = self.a
        top = a[0]
        last = a.pop()
        if a:
            a[0] = last
            self._sift_down(0)
        return top

    def _sift_down(self, i: int) -> None:
        a, n = self.a, len(self.a)
        while True:
            l, r, big = 2 * i + 1, 2 * i + 2, i
            if l < n and a[l] > a[big]:
                big = l
            if r < n and a[r] > a[big]:
                big = r
            if big == i:
                return
            a[i], a[big] = a[big], a[i]
            i = big
```

읽어 보면 `_sift_down` 이 정의 10.1 의 꺼내기다. 두 자식 중 큰 쪽과 견줘 바꾸고 내려간다. 배열 번호의 산수만 있고 링크가 없다. 8.4 의 배열 트리가 여기서 다시 나온다.

구현 세 방법 `samples/dsai/heap.py:57`

```
def topk_heap(x: np.ndarray, k: int) -> np.ndarray:
    """알고리즘 10.1. 크기 k 의 최소 힙을 유지한다.
    원소마다 힙의 최소와 견줘 크면 바꾼다. n log k."""
    heap = MaxHeap()
    for v in x[:k]:
        heap.push(-float(v))            # 최대 힙에 부호를 바꿔 넣어 최소 힙으로 쓴다
    for v in x[k:]:
        if -heap.a[0] < v:
            heap.a[0] = -float(v)
            heap._sift_down(0)
    return -np.array(heap.a)


def topk_select(x: np.ndarray, k: int) -> np.ndarray:
    """명제 10.3. 부분 정렬. k 번째로 큰 값을 기준으로 앞뒤를 가른다. 기대 n."""
    idx = np.argpartition(-x, k - 1)[:k]
    return x[idx]


def topk_sort(x: np.ndarray, k: int) -> np.ndarray:
    """디딤돌. 전부 정렬하고 앞 k 개. n log n."""
    return np.sort(x)[::-1][:k]
```

읽어 보면 `topk_heap` 의 `if -heap.a[0] < v` 가 대부분의 원소를 비교 하나로 보낸다. `topk_select` 는 `argpartition` 한 줄이고 실험에서 가장 빨랐다.

분석

이진 힙은 그대로 유효하다. 1964년 Williams의 정의가 그대로이고 [확립] [40], 배열 위의 완전 이진 트리라 8.1의 링크 값을 처음부터 안 치른다. 바뀐 것은 힙이 서는 자리다. 상위 k 개를 배열에서 한 번에 고르는 데는 선택이 이기고, 힙은 원소가 하나씩 오는 스트림과 다음 절의 빔, 그리고 10.4의 우선순위 큐에 남는다.

현장에서

> 언어 모델 디코딩의 top-k는 어휘 크기 V의 로짓에서 매 단계 k 개를 고르고, 가속기 구현은 정렬이 아니라 선택 또는 기수 방식의 한두 번 훑기다 [확립]. V가 128,000이고 배치가 수백이면 매 단계 로짓 수천만 개이고, $n \log n$과 n의 차이가 단계마다 든다.

주의

> 실험 10.1의 힙은 파이썬 루프라 절대 시간이 크다. C로 쓴 힙은 크기 8,000에서 정렬보다 빠르다. 방향은 비교 횟수로 읽고, 언어의 값은 따로 읽는다.

힙의 약속은 최대 하나뿐이고, 그 약속이 배열에서 상위 k 개를 고르는 데는 선택에 진다. 힙이 이기는 자리는 원소가 하나씩 올 때다. 다음 절의 빔 탐색은 단계마다 후보가 새로 나오고, 거기서 힙이 아니라 선택이 또 이긴다. 이유가 같다.

되짚기

> 이진 힙을 배열 위의 완전 이진 트리로 정의하고 연산이 로그임을 보였다. 상위 50개를 고르는 데 선택이 정렬보다 여덟 배 빨랐고, 파이썬 힙은 언어의 값에 덮였다. 다음 절은 단계마다 상위 B 개를 고르는 빔 탐색이다.

10.2 빔 탐색의 힘

언어 모델이 토큰 열을 낼 때 매 단계 어휘 V 개 중 하나를 고른다. 열의 점수는 단계별 로그 확률의 합이고, 가장 좋은 열을 찾으려면 V^T 개의 열을 봐야 한다. 빔 탐색은 매 단계 후보 B 개만 남긴다. 후보마다 V 개로 뻗어 BV 개가 되면 그중 상위 B 개를 고른다. 10.1의 상위 k 개 고르기가 단계마다 한 번이다.

이 절은 빔 탐색을 정의하고, 단계 하나의 비용과 빔이 최선을 놓치는 정도를 잰다. 그리고 남긴 후보에서 열을 되짚는 부모 배열을 세운다.

학습 목표

빔 탐색을 단계마다의 상위 B 개 선택으로 정의하고, 총 비용이 $O(TBV)$ 임을 보일 수 있다. 빔이 최선을 놓치는 이유를 정의에서 설명하고, 빔 폭에 따라 놓치는 비율이 어떻게 줄어드는지 실측으로 말할 수 있다.

이 말부터

빔	한 단계에 남기는 후보 열의 수. B 로 적고, 1 이면 탐욕이고 V^T 이면 전부 보기다
부모 배열	단계 t 의 후보 b 가 단계 $t-1$ 의 어느 후보에서 왔는지 적은 배열. 열은 저장하지 않고 이 배열로 되짚는다
뻗기	후보 하나에서 어휘 V 개로 다음 토큰을 붙여 후보 V 개를 만드는 일

디딤돌

정의 10.5: 빔 탐색

단계 t 의 후보 집합 C_t 는 크기 B 이하이고, C_0 은 빈 열 하나다. 단계 $t+1$ 에서 각 후보 $c \in C_t$ 를 토큰 $v \in \mathcal{V}$ 로 뻗어 점수 $s(c) + \log p(v \mid c)$ 를 매기고, $|C_t| \cdot V$ 개 중 상위 B 개를 C_{t+1} 로 둔다.

불변식 C_{t+1} 의 모든 열은 C_t 의 어느 열에 토큰 하나를 붙인 것이다. 단계 t 에서 버린 열의 연장은 그 뒤 어느 단계에도 없다.

불변식의 둘째 문장이 빔이 최선을 놓치는 이유다. 앞에서 점수가 낮았던 열이 뒤에서 좋아질 수 있는데, 그 열은 버려진 뒤라 되살아나지 않는다. 빔을 키우면 덜 버리고, V^T 까지 키우면 안 버린다.

명제 10.6: 빔 탐색의 비용

> 길이 T, 빔 B, 어휘 V 의 빔 탐색은 점수 계산 $O(TBV)$ 와 선택 T 번이다. 전부 보기는 V^T 다. 각 단계의 선택은 BV 개에서 B 개이고 명제 10.4 로 $O(BV)$ 다. 정렬하면 $O(BV \log BV)$ 다. 열을 되짚는 부모 배열은 $T \times B$ 정수이고, 열 전체를 저장하면 $T \times B \times T$ 다.

증명. 단계마다 후보 B 개가 V 개로 뻗으므로 BV 개의 점수이고 그중 B 개를 고른다. T 단계다. 부모 배열은 단계마다 후보 B 개의 출처 하나씩이고, 마지막 후보에서 T 번 거슬러 올라가면 열이다. ■

명제의 마지막 문장이 이 절의 자료 구조다. 열을 복사하지 않는다. 후보가 어디서 왔는지만 적고, 끝에서 거꾸로 따라간다. 5.2 의 테이프와 같은 꼴이다.

예제 10.2: 어휘 128,000, 빔 8, 길이 100

> 점수 계산은 $100 \times 8 \times 128{,}000 = 10^8$ 이고, 선택은 단계마다 백만 개에서 여덟 개다. 부모 배열은 정수 800 개다. 열을 저장하면 $100 \times 8 \times 100 = 80{,}000$ 정수이고 단계마다 열 8 개를 복사한다. 부모 배열이 백 배 작다.

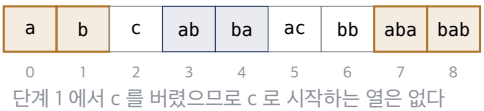


그림 10.3.. 빔 2 의 세 단계. 단계마다 후보 둘이 어휘 셋으로 뻗어 여섯이 되고 둘만 남는다. 회색 열은 버려져 되살아나지 않는다

예제는 셈이다. 빔이 최선을 얼마나 놓치는지는 모델에 달렸으므로, 답을 아는 작은 모델에서 잰다.

균열

어휘 64, 길이 8 의 무작위 조건부 확률 표 백 개에서 최선의 열을 동적 계획법으로 구하고, 빔 1, 2, 4, 8 이 그것을 찾는 비율과 점수 차이를 쟀다. 그리고 어휘 128,000, 빔 8 의 한 단계 시간을 쟀다.

실험 10.2 빔 폭과 최선을 찾는 비율

빔 폭	전부 보기 열 수	빔이 본 열 수	최선을 찾은 비율 퍼센트	점수 차이 평균
1	281474976710656	512	5.00	1.60
2	281474976710656	1024	18.0	0.86
4	281474976710656	2048	29.0	0.49
8	281474976710656	4096	45.0	0.25

측정 조건. 조건부 확률은 이전 토큰 하나에만 달렸다. 그래서 전부 보기의 답을 동적 계획법으로 낼 수 있고, 빔은 그것을 모른 채 탐색한다.

빔 1 (탐욕) 은 백 번에 다섯 번 최선을 찾고, 빔 8 은 마흔다섯 번이다. 점수 차이는 1.6 에서 0.25 로 준다. 빔을 두 배로 할 때마다 찾는 비율이 오르지만 두 배로 오르지는 않는다. 본 열의 수는 4,096 개이고 전부는 2.8×10^{14} 개다.

실험 10.3 한 단계의 값. 빔 8 곱하기 어휘 128,000

일	ms
한 단계, 빔 8, 어휘 128,000	21.8
같은 행렬 전부 정렬	161

측정 조건. 점수 행렬 (8, 128,000) 에서 상위 8 개를 선택으로 고르는 시간과, 같은 행렬을 전부 정렬하는 시간.

한 단계가 22 밀리초이고, 그중 선택이 전부다. 정렬하면 161 밀리초다. 백 단계면 2 초와 16 초다. 명제 10.6 의 마지막 항이 실측에서 일곱 배로 나왔다.

관찰

빔이 최선을 놓치는 것과 빔 탐색이 느린 것은 별개다. 앞의 것은 정의의 불변식 때문이고 빔으로만 줄어든다. 뒤의 것은 선택 하나의 값이고 10.1 의 명제로 줄어든다.

이 관찰이 빔을 키우는 값을 정한다. 빔을 두 배로 하면 단계의 값이 두 배이고 찾는 비율은 두 배가 안 된다. 실험에서 8 이 45 퍼센트였다.

다시 쓰기

이 절의 다시 쓰기는 부모 배열이다. 열을 복사하지 않고 출처만 적는다.

명제 10.7: 부모 배열의 되짚기

단계 t 마다 부모 배열 $P_t[b]$ 와 토큰 배열 $K_t[b]$ 를 두면, 마지막 후보 b_T 에서 $b_{t-1} = P_t[b_t]$ 로 거슬러 올라가며 $K_t[b_t]$ 를 모은 것이 그 후보의 열이다. 메모리는 $2TB$ 정수이고 되짚기는 T 걸음이다. 열을 저장하면 단계마다 B 개의 열을 복사해 $O(T^2B)$ 다.

증명. 정의 10.5 의 불변식에 의해 단계 t 의 후보는 단계 $t-1$ 의 후보 하나에 토큰 하나를 붙인 것이므로, 그 하나를 적어 두면 열이 결정된다. ■

이 명제가 5.2 의 테이프를 거꾸로 읽는 것과 같은 모양이다. 앞으로 갈 때 출처를 적고 뒤로 갈 때 따라간다. 5.3 의 KV 캐시도 후보마다 있어야 하고, 후보가 부모를 나누면 9.3 의 접두사 공유가 그대로 적용된다. 빔 B 개의 KV 는 부모가 같은 만큼 겹친다.

알고리즘 10.2 빔 한 단계 `dsai.beam.beam_step`

입력: 후보 점수 $s[0..B)$, 로그 확률 $L[0..B, 0..V)$, 빔 B

1: $T[b, v] \leftarrow s[b] + L[b, v]$ ▷ (B, V)
2: $I \leftarrow T$ 를 한 줄로 편 것에서 상위 B 개의 번호 ▷ 선택
3: 부모 $\leftarrow I \operatorname{div} V$; 토큰 $\leftarrow I \bmod V$
4: **반환** 부모, 토큰, $T[I]$

구현

한 단계는 덧셈 하나와 선택 하나와 나눗셈 하나다. 열은 부모 배열로 되짚는다.

지금 보는 코드 점수 행렬을 한 줄로 펴고 상위 B 개를 고른 뒤, 번호를 부모와 토큰으로 나눈다.

구현 빔 한 단계 `samples/dsai/beam.py:11`

```python
def beam_step(scores: np.ndarray, logp: np.ndarray,
              beam: int) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """알고리즘 10.2. scores (B,) 와 logp (B, V) 에서 상위 beam 개의 (부모, 토큰,
    점수)."""
    total = scores[:, None] + logp                    # (B, V)
    flat = total.ravel()
    idx = np.argpartition(-flat, beam - 1)[:beam]     # 선택. 정렬하지 않는다
    parent, token = np.divmod(idx, logp.shape[1])
    return parent, token, flat[idx]
```

읽어 보면 `argpartition` 이 10.1 의 선택이다. `divmod` 가 편 번호를 (부모, 토큰) 으로 되돌린다. 행렬 (B, V) 의 번호 산수가 2.3 의 뷰다.

구현 부모 배열로 되짚기 `samples/dsai/beam.py:23`

```python
def beam_search(step_logp, beam: int, steps: int,
                vocab: int) -> tuple[np.ndarray, float]:
    """빔 탐색 전체. step_logp(prev_tokens (B,), t) -> (B, V) 로그 확률.
    최선 열과 점수.
    """
    scores = np.zeros(1)
    tokens = np.zeros(1, dtype=np.int64)
    parents: list[np.ndarray] = []
    chosen: list[np.ndarray] = []
    for t in range(steps):
        logp = step_logp(tokens, t)
        parent, token, scores = beam_step(scores, logp, min(beam, logp.size))
        parents.append(parent)
        chosen.append(token)
        tokens = token
    best = int(np.argmax(scores))
    seq = []
    for t in range(steps - 1, -1, -1):              # 부모 배열을 거슬러 올라간다
        seq.append(int(chosen[t][best]))
        best = int(parents[t][best])
    return np.array(seq[::-1]), float(scores.max())
```

읽어 보면 `parents` 와 `chosen` 이 명제 10.7 의 두 배열이다. 마지막 `for` 가 되짚기이고 T 걸음이다. 열을 한 번도 복사하지 않는다.

분석

빔 탐색은 그대로 유효하다. 정의와 불변식이 그대로이고, 최선을 놓치는 이유도 그대로다. 바뀐 것은 한 단계의 크기다. 음성 인식 시절의 어휘 수만 개가 언어 모델의 십만 개가 됐고, 배치가 곱해져 단계마다 로짓 수천만 개다. 선택 하나의 값이 단계의 값이 됐다.

이 구조가 서는 자리는 번역과 요약처럼 최선의 열 하나가 필요한 디코딩이다. 그런데 열린 생성에서는 최선의 열이 오히려 나쁘다. Holtzman 등은 2020 년에 빔 탐색과 같은 최대화 디코딩이 반복되고 밋밋한 텍스트를 내고, 확률 분포의 꼬리를 잘라 뽑는 쪽이 낫다고 보고했다 [확립] [41]. 그 뽑기가 다음 절이다.

현장에서

> 언어 모델 서빙에서 빔 탐색은 후보 B 개의 KV 캐시를 함께 다뤄야 하고, 후보가 부모를 나누므로 5.4 의 블록 테이블이 쓰기 시 복사로 그것을 처리한다 [확립] [9]. 빔이 갈릴 때 블록 하나만 복사되고 나머지는 공유된다.

주의

> 실험 10.2 의 모델은 이전 토큰 하나에만 달린 무작위 표라 전부 보기의 답을 낼 수 있었다. 실제 모델은 전체 접두사에 달려 있어 최선의 열을 알 수 없고, 빔이 얼마나 놓치는지도 잴 수 없다. 빔 8 의 45 퍼센트는 이 표에서 잰 값이다.

빔은 최선 하나를 찾으려고 후보를 남긴다. 열린 생성은 최선을 찾지 않고 확률대로 뽑는다. 뽑는 데도 자료 구조가 있고, 어휘 십만 개에서 매 단계 하나를 뽑는 값이 다음 절이다.

되짚기

빔 탐색을 단계마다의 상위 B 개 선택으로 정의하고, 비용이 $O(TBV)$ 이며 열은 부모 배열로 되짚음을 보였다. 빔 8 이 최선을 45 퍼센트 찾았고, 한 단계의 선택이 정렬보다 일곱 배 빨랐다. 다음 절은 확률 분포에서 뽑기다.

10.3 누적합과 별칭표. 확률 분포에서 뽑기

언어 모델의 매 단계는 어휘 V 개 위의 확률 분포 하나를 내고, 그중 하나를 뽑는다. 뽑기는 무작위 수 하나를 자리 하나로 바꾸는 일이다. 교과서의 방법은 누적합이다. 확률을 앞에서부터 더해 두고 무작위 수가 어느 구간에 드는지 찾는다. 찾기는 8.4 의 이진 탐색이고 $\log V$ 다. 2.3 의 KV 캐시가 토큰 수의 누적합으로 자리를 찾던 그 배열이 여기서는 확률 위에 놓인다. 1977 년에 Walker 가 다른 표를 세웠다. 자리마다 확률 하나와 별칭 하나를 두면 뽑기가 무작위 둘과 읽기 둘로 끝난다 [확립] [42]. V 와 무관하다.

이 절은 두 표를 세우고, 언어 모델이 실제로 쓰는 절단 (top-p) 이 두 표에 무엇을 더하는지 본다.

학습 목표

누적합 뽑기와 별칭표를 정의하고, 별칭표가 원래 분포를 정확히 따름을 증명할 수 있다. 두 방법의 표 만들기와 뽑기 비용을 적고, top-p 절단이 정렬을 요구하는 이유와 선택으로 대신하는 조건을 설명할 수 있다.

이 말부터

누적합	$C[i] = p_0 + \cdots + p_i$. 무작위 u 에 대해 $C[i-1] \le u < C[i]$ 인 i 를 뽑는다
별칭	자리 i 에 적힌 다른 자리 번호. 동전이 뒤집히면 i 대신 별칭을 낸다
top-p	확률이 큰 것부터 누적이 p 를 넘을 때까지만 남기고 나머지를 0 으로 두는 절단

디딤돌

정의 10.8: 누적합 뽑기

확률 $p_0, \dots, p_{V-1}$ 의 누적합 C 를 두고, $u \sim U(0,1)$ 에 대해 $C[i-1] \le u < C[i]$ 인 i 를 낸다. C 는 정렬되어 있으므로 이진 탐색이다.

불변식 i 가 뽑힐 확률은 구간 길이 $C[i] - C[i-1] = p_i$ 다.
표 만들기 $O(V)$, 뽑기 하나 $O(\log V)$ 다.

이진 탐색의 $\log V$ 는 어휘 128,000 에서 17 걸음이고, 걸음마다 8.4 의 라인이다. 뽑기가 매 단계이고 배치가 수백이면 이 17 걸음이 단계마다 수천 번이다. Walker 의 표는 그 17 을 2 로 만든다.

정의 10.9: 별칭표

자리 V 개마다 확률 $q_i \in [0,1]$ 과 별칭 a_i 를 둔다. 뽑기는 자리 i 를 균일하게 고르고, 확률 q_i 로 i 를, 아니면 a_i 를 낸다.

불변식 모든 j 에 대해 $\frac{1}{V}\left(q_j + \sum_{i:a_i=j}(1-q_i)\right) = p_j$ 다. 자기 자리에서 남은 확률과 다른 자리에서 별칭으로 온 확률의 합이 p_j 다.

정의는 표가 있을 때 뽑는 법을 말하고, 그런 표가 언제나 있는지는 말하지 않는다. 그것이 정리이고, 증명이 곧 표를 만드는 알고리즘이다.

정리 10.10: 별칭표의 존재와 정확성

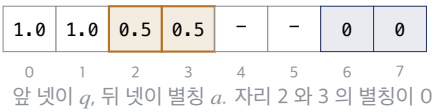
어떤 분포 p 에 대해서도 불변식을 만족하는 (q, a) 가 있고, $O(V)$ 에 만들 수 있다. 뽑기는 무작위 둘과 읽기 둘이며 j 가 나올 확률은 정확히 p_j 다.

증명. Vp_j 를 자리 j 의 양이라 하자. 합이 V 이므로 1 미만인 자리 (작은 것) 와 1 이상인 자리 (큰 것) 가 있다. 작은 자리 s 를 하나 집어 $q_s = Vp_s$ 로 두고, 큰 자리 l 하나를 별칭으로 삼아 l 의 양에서 $1 - q_s$ 를 덜어 낸다. s 는 정확히 1 로 채워졌다. l 의 남은 양이 1 미만이면 작은 쪽으로 옮긴다. 자리 하나가 매번 채워지므로 V 번에 끝나고, 채워진 자리의 양은 자기 것과 별칭으로 받은 것의 합이므로 불변식이 된다. Vose 가 1991 년에 이 만들기가 선형 시간임을 보였다 [확립] [43]. 정확성은 불변식을 V 로 나눈 것이 뽑힐 확률이라는 것이다. ■

정리에서 읽을 것은 표를 한 번 만드는 값 $O(V)$ 와 뽑기 하나의 값 상수다. 같은 분포에서 여러 번 뽑을 때 남고, 매번 분포가 바뀌면 만드는 값이 뽑기의 값을 넘는다. 언어 모델은 매 단계 분포가 바뀐다. 그래서 별칭표는 한 단계 안에서 여러 개를 뽑을 때, 즉 배치나 여러 후보에서만 남는다.

예제 10.3: 자리 넷

$p = (0.5, 0.25, 0.125, 0.125)$ 이면 양은 $(2, 1, 0.5, 0.5)$ 다. 자리 2 에 $q = 0.5$, 별칭 0 을 두고 0 의 양을 1.5 로. 자리 3 에 $q = 0.5$, 별칭 0 을 두고 0 의 양을 1 로. 자리 0 과 1 은 $q = 1$ 이다. 자리 0 이 뽑힐 확률은 $\frac{1}{4}(1 + 0.5 + 0.5) = 0.5$ 다.



앞 넷이 q, 뒤 넷이 별칭 a. 자리 2 와 3 의 별칭이 0 이다

그림 10.4.. 예제의 별칭표. 자리마다 자기 확률과 별칭. 자리 2 와 3 의 남은 반이 자리 0 으로 간다

예제의 표는 넷이다. 어휘 128,000 에서 표를 만들고 백만 개를 뽑아 두 방법의 값을 잰다.

균열

지프 분포의 어휘 128,000 개에서 백만 개를 두 방법으로 뽑았다. 그리고 top-p 절단을 전부 정렬과 선택으로 해서 견줬다.

실험 10.4 어휘 128,000 에서 백만 개 뽑기

방식	표 만들기 ms	백만 개 ms	뽑기당 ns	상위 10 빈도 오차 퍼센트
누적합 + 이진 탐색	0.00	162	162	1.39
별칭표	432	49.3	49.3	1.28

측정 조건. 지프 지수 1.1. 표는 한 번 만들고 백만 개를 뽑는다. 빈도 오차는 상위 열 개 토큰의 뽑힌 비율과 확률의 상대 차이 최대.

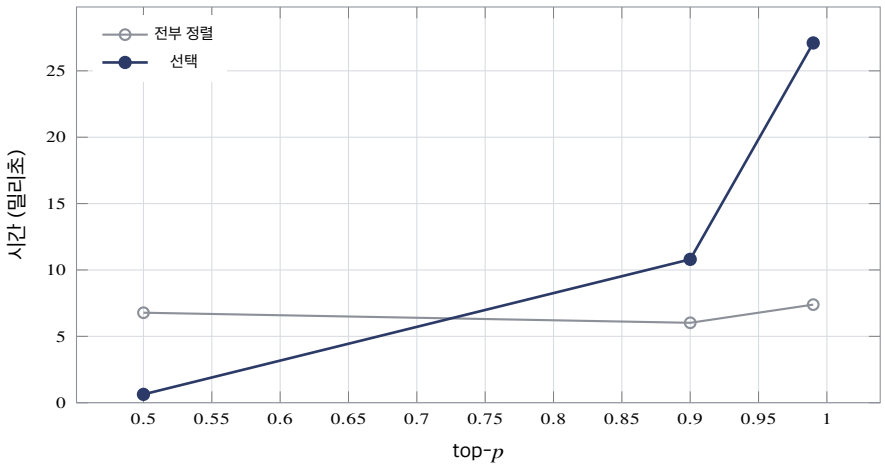


그림 10.5. top-p 에 따른 두 방법의 시간. p 가 작으면 남는 토큰이 적어 선택이 이기고, 크면 선택이 정렬보다 느려진다

별칭표가 뽑기에서 세 배 빠르다. 뽑기당 49 나노초와 162 나노초다. 둘 다 백만 개를 한꺼번에 뽑는 NumPy 호출이라 이진 탐색 17 걸음과 읽기 둘의 차이가 세 배로 나왔다. 표 만들기는 432 밀리초이고 파이썬 루프다. 백만 개를 뽑아야 본전이다.

실험 10.5 top-p 절단. 전부 정렬과 선택

top p	남는 토큰 수	전부 정렬 ms	선택으로 ms
0.50	45	6.78	0.63
0.90	14529	6.02	10.8
0.99	100672	7.39	27.1

측정 조건. 같은 지프 분포. 선택은 상위 64 개를 부분 정렬하고 누적이 모자라면 두 배로 늘린다.

top-p 0.5 에서는 45 개만 남고 선택이 정렬보다 열 배 빠르다. 0.99 에서는 십만 개가 남고 선택이 네 배 느리다. 두 배씩 늘리며 부분 정렬을 되풀이한 값이 정렬 한 번을 넘었다.

관찰

> top-p 의 값은 p 가 아니라 남는 토큰 수가 정한다. 분포가 뾰족하면 선택이 이기고 평평하면 정렬이 이긴다. 같은 p 에서도 단계마다 다르다.

이 관찰이 절단을 자료 구조로 보는 이유다. 남는 수를 모르고 시작하므로, 작게 시작해 두 배씩 늘리는 2.1 의 배가 정책이 여기서도 나온다.

다시 쓰기

이 절의 다시 쓰기는 top-p 를 정렬 없이 하는 조건이다.

명제 10.11: top-p 의 비용

> 남는 토큰 수를 m 이라 하자. 전부 정렬은 $O(V \log V)$ 이고, 선택을 k 를 두 배씩 늘리며 되풀이하면 $O(V \log(m/k_0) + m \log m)$ 이다. $m \ll V$ 이면 선택이 이기고, m 이 V 에 가까우면 정렬이 이긴다. 경계는 실험에서 m/V 가 1 퍼센트와 10 퍼센트 사이였다.

증명. 선택 한 번이 $O(V)$ 이고 k 가 m 에 이를 때까지 $\log_2(m/k_0)$ 번이다. 남은 m 개를 정렬하는 것이 둘째 항이다. ■

이 명제와 정의 10.9 를 이으면 뽑기의 전체 모양이다. 절단으로 m 개를 남기고, m 개 위에서 누적합이나 별칭표로 뽑는다. m 이 작으면 누적합의 $\log m$ 도 작아 별칭표가 필요 없다. 별칭표가 남는 자리는 절단 없이 큰 분포에서 여러 번 뽑을 때이고, 그것은 16장의 데이터 로더에 있다.

알고리즘 10.3 별칭표 만들기 `dsai.sampling.AliasTable`

> **입력:** 확률 $p[0..V)$
> 양 $g[i] \leftarrow Vp[i]$; 작은 것 $S \leftarrow \{i : g[i] < 1\}$; 큰 것 $L \leftarrow \{i : g[i] \ge 1\}$
> **조건** S 와 L 이 비지 않음 **동안**
> $s \leftarrow S$ 에서 하나; $l \leftarrow L$ 에서 하나; $q[s] \leftarrow g[s]$; $a[s] \leftarrow l$
> $g[l] \leftarrow g[l] + g[s] - 1$; $g[l] < 1$ 이면 l 을 S 로, 아니면 L 로
> **끝 조건**
> 남은 자리는 $q \leftarrow 1$

구현

누적합 뽑기는 두 줄이다. 별칭표는 만들기가 전부이고 뽑기는 세 줄이다.

지금 보는 코드 작은 자리를 큰 자리로 채우고, 큰 자리의 남은 양을 다시 작은 것으로 돌린다.

구현 별칭표 만들기 `samples/dsai/sampling.py:26`

```
    def __init__(self, p: np.ndarray):
        n = len(p)
        q = p / p.sum() * n
        self.prob = np.zeros(n)
        self.alias = np.zeros(n, dtype=np.int64)
        small = [i for i in range(n) if q[i] < 1.0]
        large = [i for i in range(n) if q[i] >= 1.0]
        while small and large:                  # 알고리즘 10.3. Vose 의 만들기
            s, l = small.pop(), large.pop()
            self.prob[s] = q[s]
            self.alias[s] = l
            q[l] = q[l] + q[s] - 1.0
            (small if q[l] < 1.0 else large).append(l)
        for i in small + large:
            self.prob[i] = 1.0
            self.alias[i] = i
```

읽어 보면 `small` 과 `large` 가 알고리즘의 S 와 L 이다. 뽑기는 같은 클래스의 `sample` 세 줄이고, 정리의 무작위 둘과 읽기 둘 그대로다. `np.where` 가 동전이다.

구현 top-p 를 선택으로 `samples/dsai/sampling.py:62`

```
def top_p_mask_by_select(p: np.ndarray, top_p: float, k_guess: int) -> np.ndarray:
    """10.1 절의 선택으로 top-p. 상위 k 개만 부분 정렬하고 모자라면 두 배로."""
    n = len(p)
    k = min(k_guess, n)
    while True:
        idx = np.argpartition(-p, k - 1)[:k]
        order = idx[np.argsort(-p[idx])]
        c = np.cumsum(p[order])
        if c[-1] >= top_p or k == n:
            keep = c - p[order] < top_p
            mask = np.zeros(n, dtype=bool)
            mask[order[keep]] = True
            return mask
        k = min(2 * k, n)
```

읽어 보면 `k` 를 두 배로 늘리는 `while` 이 명제 10.11 의 $\log(m/k_0)$ 번이다. 누적이 문턱을 넘으면 그 안에서만 정렬한다.

분석

누적합 뽑기와 별칭표는 그대로 유효하다. 1977 년의 표와 1991 년의 만들기가 그대로이고 실험이 정리의 상수를 보였다. 바뀐 것은 뽑기 앞에 절단이 붙었다는 것이다. Holtzman 등의 top-p 는 분포의 꼬리를 잘라 뽑고 [확립] [41], 그 절단이 정렬을 부르며, 정렬을 선택으로 바꾸는 조건이 남는 토큰 수다.

이 구조가 서는 자리는 셋이다. 디코딩의 매 단계 뽑기, 16 장의 가중 표본 추출, 그리고 3.4 의 전문가 고르기. 셋 다 확률 분포 하나에서 자리 하나를 얻는 일이고, 분포가 매번 바뀌는지가 표를 만들지 말지를 정한다.

현장에서

Walker 의 1977년 논문은 표를 두 번의 메모리 참조와 비교 하나로 뽑는 방법으로 소개했고 [확립] [42], Vose 는 만들기를 $O(V)$ 로 줄였다 [확립] [43]. 두 논문 사이의 14년 동안 표는 같았고 만드는 값만 바뀌었다.

주의

실험 10.3 의 별칭표는 백만 개를 한 번에 뽑았다. 매 단계 분포가 바뀌는 디코딩에서는 표를 매번 만들어야 하고, 만들기 432 밀리초가 뽑기의 이점을 지운다. 표를 만들 값이 나오는 자리에서만 쓴다.

뽑기는 확률이 정한 순서였다. 다음 절의 순서는 사람이 정한다. 요청을 어떤 차례로 처리할지이고, 힙이 다시 나온다. 이번에는 힙의 키가 무엇이냐가 전부다.

되짚기

누적합 뽑기와 별칭표를 정의하고 별칭표의 정확성을 증명했다. 뽑기가 세 배 빨랐고, top-p 는 남는 토큰 수에 따라 선택과 정렬이 갈렸다. 다음 절은 요청을 줄 세우는 우선순위 큐다.

10.4 우선순위 스케줄러. 요청을 줄 세우기

5.4 의 연속 배칭은 요청을 도착한 차례로 받았다. 큐가 그것이고, 5장의 링 버퍼였다. 도착 순서가 언제나 맞는 것은 아니다. 짧은 요청 뒤에 긴 요청이 서면 짧은 것은 금방 끝나는데 긴 것 하나 때문에 다들 기다린다. 순서를 바꾸려면 큐 대신 우선순위 큐가 필요하고, 우선순위 큐가 10.1 의 힙이다.

이 절은 우선순위 큐를 정의하고, 짧은 것 먼저가 평균 대기를 최소로 만든다는 정리와 그 대가인 굶김을 세운 뒤, 나이를 더해 둘을 맞바꾸는 규칙을 잰다.

학습 목표

우선순위 큐를 힙으로 정의하고, 짧은 것 먼저가 평균 대기를 최소로 한다는 것을 교환 논증으로 증명할 수 있다. 짧은 것 먼저가 긴 요청을 굶기는 이유와 나이 가중이 그것을 어떻게 막는지 설명하고, 세 정책의 평균과 최대 대기를 실측으로 말할 수 있다.

이 말부터

우선순위 큐	키가 가장 작은 항목을 먼저 내는 큐. 힙 하나로 넣기와 꺼내기가 로그다
짧은 것 먼저	처리 시간이 짧은 요청을 먼저 하는 정책. 평균 대기가 최소다
굶김	긴 요청이 짧은 요청에 계속 밀려 한없이 기다리는 것. 최대 대기가 커진다

디딤돌

정의 10.12: 우선순위 큐

항목마다 키 하나를 둔 집합이다. 넣기와 최소 키 항목 꺼내기를 지원한다. 정의 10.1 의 최소 힙으로 두면 둘 다 $O(\log n)$ 이다.

불변식 꺼낸 항목의 키는 큐에 남은 어떤 항목의 키보다 작거나 같다.

스케줄러는 요청을 키와 함께 넣고, 처리할 때마다 하나를 꺼낸다. 키가 도착 시각이면 큐이고, 처리 길이면 짧은 것 먼저다.

정의는 키가 무엇인지 말하지 않는다. 키를 정하는 것이 정책이고, 정책마다 다른 것을 최소로 한다. 첫 정책은 평균 대기를 최소로 하는 것이고, 그것은 정리 하나로 적힌다.

정리 10.13: 짧은 것 먼저의 최적성

요청 n 개가 모두 도착해 있고 한 번에 하나를 처리할 때, 처리 시간이 짧은 순서가 대기 시간의 합을 최소로 한다.

증명. 어떤 순서에서 긴 요청 a 가 짧은 요청 b 바로 앞에 있다고 하자. 둘을 바꾸면 b 의 대기가 a 의 길이만큼 줄고 a 의 대기가 b 의 길이만큼 늘며, 다른 요청은 그대로다. a 가 더 기니까 합이 준다. 이웃한 역전이 없을 때까지 바꾸면 짧은 순서이고, 그것이 최소다. ■

정리의 조건은 모두 도착해 있다는 것이다. 요청이 계속 오면 짧은 것이 계속 새로 오고, 긴 것은 언제나 뒤로 밀린다. 그것이 굶김이고, 다음 명제가 그것을 막는 값을 적는다.

명제 10.14: 굶김과 나이 가중

> 부하가 1 에 가까울수록 짧은 요청이 끊이지 않고, 긴 요청의 대기는 한없이 는다. 최대 대기에 상한이 없다. 키를 길이 $-\alpha\times$ 기다린 시간으로 두면 어떤 요청도 기다린 시간이 길이/α 를 넘으면 키가 0 아래로 내려가 새로 온 어떤 요청보다 앞선다. 최대 대기가 (최대 길이/α + 앞선 것들의 처리 시간) 으로 묶인다.

증명. 첫 문장은 도착률이 처리율에 가까우면 짧은 요청의 도착 간격이 그 처리 시간보다 짧아 큐에 언제나 짧은 것이 있다는 것이다. 둘째는 키의 식에서 바로 나온다. 기다린 시간이 길이/α 이면 키가 0 이고, 새 요청의 키는 양수다. ■

명제의 상한은 나이 계수 하나로 정해진다. 계수가 크면 도착 순서에 가깝고, 0 이면 짧은 것 먼저다. 숫자를 넣어 본다.

예제 10.4: 길이 32 와 4,096

> 토큰 32 개 요청과 4,096 개 요청이 초당 2,000 토큰의 처리량에 번갈아 오면, 짧은 것 먼저에서 긴 것은 짧은 것이 안 오는 순간까지 기다린다. 나이 계수 200 이면 긴 것은 늦어도 4,096/200 = 20 초 뒤에 키가 0 이 되어 앞선다.

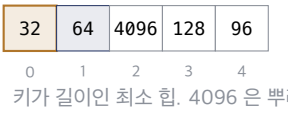


그림 10.6.. 짧은 것 먼저의 힙. 키가 길이이고 뿌리가 가장 짧은 요청이다. 긴 요청은 새 짧은 요청이 올 때마다 밀린다

예제의 20 초는 상한이다. 세 정책이 실제로 어떤 평균과 최대를 내는지 잰다.

균열

요청 5,000 개가 부하 0.9 의 푸아송으로 오고 길이는 지프 분포다. 도착 순서, 짧은 것 먼저, 나이를 더한 짧은 것 먼저의 대기를 쟀다.

실험 10.6 세 정책의 평균, 99 퍼센타일, 최대 대기

정책	평균 대기 ms	99 퍼센타일 ms	최대 대기 ms
도착 순서	5141	17635	20212
짧은 것 먼저	1066	12035	89196
짧은 것 먼저 + 나이	1244	14293	25080

측정 조건. 처리량 초당 2,000 토큰, 한 번에 하나. 길이는 32 토큰 단위 지프 (최대 4,096). 나이 계수는 초당 200 토큰. 같은 도착 열을 세 번.

짧은 것 먼저가 평균 대기를 5.1 초에서 1.1 초로 다섯 배 줄였다. 정리대로다. 그리고 최대 대기를 20 초에서 89 초로 네 배 늘렸다. 명제대로다. 나이를 더하자 평균은 1.2 초로 거의 그대로이고 최대가 25 초로 돌아왔다. 평균 17 퍼센트를 내고 최대의 네 배를 없앴다.

관찰

세 정책이 처리한 일은 같다. 같은 요청 5,000 개를 같은 처리량으로 했다. 다른 것은 순서뿐이고, 순서가 평균을 다섯 배, 최대를 네 배 움직였다. 자료 구조가 바뀐 것이 아니다. 힙의 키가 바뀌었다.

이 관찰이 이 절의 요점이다. 힙은 키를 모른다. 키를 무엇으로 두느냐가 정책이고, 정책이 값을 정한다.

다시 쓰기

이 절의 다시 쓰기는 5.4 의 연속 배칭에 우선순위를 넣는 것이다.

명제 10.15: 배치 스케줄러의 키

연속 배칭에서 매 단계 빈자리에 넣을 요청을 고르는 것은 우선순위 큐의 꺼내기다. 키가 도착 시각이면 5.4 그대로이고, 남은 길이의 추정이면 짧은 것 먼저이고, 거기서 나이를 빼면 굶김이 없는 짧은 것 먼저다. 남은 길이는 모르므로 추정이고, 추정이 틀리면 정리의 최적성도 그만큼 틀린다.

증명. 연속 배칭의 단계마다 끝난 요청의 자리에 큐에서 하나를 꺼내 넣으므로, 큐가 우선순위 큐이면 꺼내는 순서가 정책이다. 나머지는 정리 10.13 와 명제 10.14 이다. ■

이 명제의 마지막 문장이 언어 모델 서빙의 어려움이다. 요청의 길이가 출력 토큰 수인데, 그것은 끝나 봐야 안다. 입력 길이로 추정하거나 작은 모델로 예측하고, 그 추정 위에서 정리를 쓴다.

알고리즘 10.4 우선순위 스케줄러 `dsai.scheduler.simulate`

입력: 요청 열 (도착, 길이), 정책, 처리량
시각 $t \leftarrow 0$; 우선순위 큐 Q
조건 남은 요청이 있음 **동안**
    t 까지 도착한 요청을 정책의 키로 Q 에 넣는다
    나이 정책이면 Q 의 모든 키를 길이 $-\alpha(t-$ 도착$)$ 으로 다시 매긴다
    $r \leftarrow Q$ 에서 꺼냄; 대기 $\leftarrow t - r$. 도착; $t \leftarrow t + r$. 길이 / 처리량
끝 조건

구현

큐는 파이썬의 힙이다. 정책은 키 하나이고, 나이 정책은 꺼내기 전에 키를 다시 매긴다.

지금 보는 코드 도착한 요청을 키와 함께 넣고, 가장 작은 키를 꺼내 처리한다.

구현 우선순위 큐 `samples/dsai/scheduler.py:20`

```
class PriorityQueue:
    """정의 10.10. 키가 가장 작은 항목을 먼저 내는 큐. 힙 하나다."""

    def __init__(self):
        self.h: list[Item] = []

    def push(self, item: Item) -> None:
        heapq.heappush(self.h, item)

    def pop(self) -> Item:
        return heapq.heappop(self.h)

    def __len__(self) -> int:
        return len(self.h)
```

읽어 보면 `heapq` 가 정의 10.1 의 최소 힙이다. `Item` 의 `order=True` 와 `compare=False` 가 키만 견주게 한다.

모의 실행 함수는 같은 파일의 `simulate` 이고 알고리즘 그대로라 싣지 않았다. 정책은 그 안의 한 줄, 도착 시각이냐 길이냐다. 나이 정책의 다시 매기기는 힙을 비우고 다시 넣는 $O(n \log n)$ 이고, 실제 구현은 키를 도착 시각의 함수로 두어 다시 매기지 않는다. 연습문제다.

분석

우선순위 큐는 그대로 유효하다. 힙이 1964 년 그대로이고 정리와 명제가 운영 체제 교과서 그대로다. 바뀐 것은 요청의 길이를 모른다는 것이다. 운영 체제의 프로세스도 그랬지만, 언어 모델 요청은 길이가 출력 토큰 수라 시작할 때 아예 없다. 추정 위의 정책이 된다.

이 구조가 서는 자리는 추론 서버의 대기열이다. Yu 등의 연속 배칭이 도착 순서였고 [확립] [8], 그 뒤의 서버들은 우선순위를 넣는다. 짧은 것 먼저는 처리량이 아니라 대기의 분포를 바꾸고, 그 분포가 사용자가 느끼는 것이다.

현장에서

> 운영 체제의 스케줄러는 이 절의 셋을 다 갖고 있다. 도착 순서, 짧은 작업 먼저, 그리고 나이 가중의 다단계 피드백 큐다. 언어 모델 서버가 다른 것은 길이 추정의 재료다. 입력 길이, 대화의 종류, 작은 모델의 예측이 그 재료이고, 추정이 좋을수록 정리 10.13 에 가까워진다.

주의

> 실험 10.4 는 한 번에 하나를 처리하는 모형이다. 연속 배칭은 여러 개를 함께 처리하고 5.4 의 배치 밀도가 끼어든다. 평균 다섯 배와 최대 네 배는 이 모형의 값이고, 배치에서는 방향만 같다.

힙 하나가 네 절을 이었다. 상위 k 개, 빔의 후보, 뽑기의 절단, 그리고 요청의 순서. 넷 다 전부를 정렬하지 않고 필요한 것만 꺼냈다. 다음 장은 원소 사이의 관계를 저장하는 구조다. 그래프이고, 계산 그래프가 첫 자리다.

되짚기

> 우선순위 큐를 정의하고 짧은 것 먼저의 최적성과 굶김을 증명했다. 평균 대기가 다섯 배 줄고 최대가 네 배 늘었으며, 나이를 더하자 최대가 돌아왔다. 힙의 키가 정책이다. 다음 장은 그래프다.

10장을 닫으며

이 장이 세운 구조는 넷이다. 배열 위의 이진 힙과 선택, 부모 배열로 되짚는 빔, 누적합과 별칭표와 절단, 그리고 키가 정책인 우선순위 큐. 넷 다 정의 10.1 위에 있거나 그것과 견줬고, 넷 다 같은 결론이다. 전부 정렬하지 않는다.

네 실험의 숫자를 적어 둔다. 어휘 128,000 에서 상위 50 개는 선택이 정렬보다 여덟 배 빨랐고 파이썬 힙은 언어의 값에 덮였다. 빔 8 은 최선을 45 퍼센트 찾았고, 한 단계의 선택이 정렬보다 일곱 배 빨랐다. 별칭표는 뽑기당 49 나노초로 누적합의 3 분의 1 이었고, top-p 는 남는 토큰이 45 개일 때 선택이 열 배, 십만 개일 때 정렬이 네 배였다. 짧은 것 먼저가 평균 대기를 다섯 배 줄이고 최대를 네 배 늘렸으며, 나이를 더하자 최대가 돌아왔다.

정정 하나를 남긴다. 빔 실험의 전부 보기를 처음에는 열을 전부 세는 깊이 우선 탐색으로 썼다. 어휘 12, 길이 6 에서 3 백만 열이고 표 쉰 개면 1 억 5 천만 열이라 끝나지 않았다. 조건부 확률이 이전 토큰 하나에만 달렸으므로 동적 계획법이 같은 답을 TV^2 에 낸다. 그것으로 바꾸자 어휘 64, 길이 8, 표 백 개가 2 초였다. 전부 본다는 것과 전부 센다는 것은 다르다. 구조가 있으면 안 세고도 다 본다.

옛 구조의 판정

> 이진 힙은 조정해서 유효하다. 정의와 로그 비용은 그대로이고, 상위 k 개를 배열에서 한 번에 고르는 자리에서는 선택에 진다. 근거는 실험 10.1 과 10.2 다. 선택이 정렬보다 여덟 배, 빔의 단계에서 일곱 배였고, 힙은 스트림과 우선순위 큐에 남았다. 바뀐 것은 힙이 서는 자리이지 힙이 아니다.

이 장의 요약

> 1. 힙은 최대 하나만 약속하고 나머지의 순서를 버린다. 그래서 넣고 꺼내기가 로그이고 배열 하나에 든다 10.1
> 2. 상위 k 개를 배열에서 고르는 데는 선택이 이긴다. 기대 $O(n)$ 이고 배열 한 번 훑기다 10.1
> 3. 빔은 단계마다 상위 B 개 선택이고, 열은 부모 배열로 되짚는다. 버린 열은 되살아나지 않는다 10.2
> 4. 별칭표는 뽑기를 상수로 만들고, 표를 만들 값이 나오는 자리에서만 쓴다. top-p 는 남는 수가 정렬과 선택을 가른다 10.3
> 5. 우선순위 큐의 키가 정책이다. 짧은 것 먼저는 평균을 최소로 하고 굶기며, 나이가 그것을 막는다 10.4
>
> 다음 장에서 뒤집히는 것: 지금까지의 구조는 원소가 서로 무관하거나 순서 하나로 이어졌다. 11 장의 그래프는 원소 사이의 어떤 관계든 저장하고, 계산 그래프가 그 첫 자리다.

필자의 판단

> 필자는 힙을 가르칠 때 정렬을 먼저 두지 않는다. 힙 정렬은 힙의 쓰임 중 가장 드문 것이고, 학생은 힙을 정렬의 부품으로 기억한다. 상위 k 개와 우선순위 큐를 먼저 두고, 선택과 견주게 한다. 힙이 지는 자리를 보고 나면 이기는 자리가 또렷해진다.

열린 문제

하나, 실험 10.1 의 힙은 파이썬이라 C 로 쓴 힙과 선택의 경계를 재지 못했다. 그 경계가 k/n 의 어디에 있는지 모른다.
둘, 실험 10.2 의 모델은 이전 토큰 하나에만 달렸다. 실제 모델에서 빔이 최선을 얼마나 놓치는지는 최선을 모르므로 잴 수 없고, 이 장은 대신할 잣대를 세우지 않았다.
셋, 실험 10.4 는 한 번에 하나를 처리했다. 연속 배칭 위에서 세 정책이 배치 밀도와 어떻게 얽히는지 재지 않았다.

연습문제

10.1 기초 원소 10^6 개에서 상위 100 개를 고를 때 정렬, 크기 100 힙, 선택의 비교 횟수 상한을 명제 10.3 로 구하라. 힙에서 실제 비교가 상한보다 훨씬 적은 이유를 적어라.

10.2 기초 어휘 32,000, 빔 4, 길이 200 의 빔 탐색에서 점수 계산 수와 부모 배열의 크기를 명제 10.6 로 구하라. 열을 저장하면 몇 배인가.

10.3 기초 $p = (0.4, 0.3, 0.2, 0.1)$ 의 별칭표를 정리 10.10 의 증명대로 손으로 만들고, 자리마다 뽑힐 확률이 p 와 같은지 불변식으로 확인하라.

10.4 기초 길이 100 과 3,000 인 요청이 초당 1,000 토큰의 처리량에 번갈아 올 때, 나이 계수 100 에서 긴 요청의 최대 대기 상한을 명제 10.14 으로 구하라.

10.5 응용 `dsai.heap.MaxHeap` 에 배열에서 힙을 $O(n)$ 에 만드는 `heapify` 를 더하고, 넣기 n 번과 시간을 견줘라. 명제 10.2 의 두 비용이 실측에서 어떻게 갈리는가.

10.6 응용 `dsai.beam.beam_search` 에 길이 정규화를 더하라. 점수를 길이로 나눠 견주면 어느 빔 폭에서 실험 10.2 의 결과가 어떻게 바뀌는지 재라.

10.7 응용 `dsai.sampling.top_p_mask_by_select` 의 시작 k_0 를 이전 단계에서 남은 수로 두는 적응 규칙을 더하고, top-p 0.9 의 지프 분포에서 시간이 얼마나 주는지 재라.

10.8 응용 `dsai.scheduler.simulate` 의 나이 정책을 다시 매기지 않게 바꿔라. 키를 길이 $+\alpha\times$ 도착 시각으로 두면 순서가 같음을 보이고, 실험 10.4 와 같은 대기가 나오는지 확인하라.

10.9 탐구 빔 탐색의 후보 B 개가 KV 캐시를 나누는 정도를 9.3 의 라디스 트리로 적어라. 빔이 갈릴 때 트리에 무엇이 생기고, 5.4 의 쓰기 시 복사와 어떻게 대응하는가.

10.10 탐구 요청 길이를 모를 때 입력 길이로 추정하는 짧은 것 먼저가 정리 10.13 에서 얼마나 멀어지는지 논하라. 추정이 무작위이면 도착 순서와 같아지는가.

더 읽을 것

[40] 이진 힙과 힙 정렬의 원전. 두 쪽이고, 배열 위의 완전 이진 트리라는 배치가 여기서 나왔다. 10.1
[42] 별칭표의 원전. 네 쪽이고, 두 번의 메모리 참조로 뽑는다는 문장이 그대로 있다. 10.3
[43] 별칭표를 선형 시간에 만드는 알고리즘. 이 장의 만들기가 이것이다. 10.3
[41] 최대화 디코딩의 퇴화와 top-p 뽑기. 10.2 절과 10.3 절을 잇는 근거다. 10.2, 10.3
[8] 연속 배칭의 원전. 10.4 절이 그 큐에 우선순위를 넣는다. 10.4

5부

그래프

그래프는 원소 사이의 관계를 저장한다. 5부는 사람이 그린 계산 그래프와 데이터가 만든 근접 그래프를 다루고, 앞으로 한 번 뒤로 한 번의 미분과, 가까운 것을 확률로 찾는 탐색과, 벡터를 바이트 몇 개로 접는 인덱스를 세운다.

11장 계산 그래프 12장 근접 그래프와 벡터 검색 13장 양자화 기반 벡터 인덱스

11장

계산 그래프

1962년의 논문 하나는 3만 개 활동의 순서를 한 시간 안에 정하는 방법을 제안했다 [확립] [44]. 들어오는 것이 없는 것부터 꺼내고, 꺼낼 때마다 이웃의 셈을 하나 줄인다. 그 순서가 지금 모든 학습의 첫 번째 훑기이고, 그 역순이 두 번째 훑기다. 앞으로 가면 값이고 뒤로 가면 기울기다. 이 장은 그 그래프를 세우고, 두 훑기를 세우고, 마지막으로 그래프를 고친다. 노드를 합쳐 이동량을 줄이는 것이 고치는 규칙이다.

이 장은 0.2 의 첫째 가정을 그래프에서 다시 본다. 이웃을 따라가는 것이 라인이고, 이웃을 배열 둘에 모으면 라인이 준다. 그리고 5.2 의 테이프가 선에서 DAG 로 넓어진다. 팬아웃이 있으면 기울기가 더해진다.

이 장에서 세우는 것

11.1 그래프의 세 표현. 정적인 그래프에서 CSR 이 메모리 5 분의 1, 전체 훑기 이백 배

11.2 위상 정렬 둘과 역방향 미분. 역순 한 번이 모든 입력의 기울기이고, 팬아웃에서 합이다

11.3 연산자 융합. 사슬을 노드 하나로. 싼 사슬에서 5.2 배, 비싼 사슬에서 1.3 배

11.1 그래프 표현. 인접 리스트와 CSR 다시 보기

지금까지의 구조는 원소가 서로 무관하거나 (배열, 해시) 순서 하나로 이어졌다 (리스트, 트리). 그래프는 원소 사이에 어떤 관계든 둔다. 노드와 간선이고, 간선이 관계다. 이 책의 작업량에서 그래프는 둘이다. 계산 그래프와 근접 그래프. 계산 그래프는 노드가 연산이고 간선이 텐서이며, 한 번 만들면 안 바뀐다. 근접 그래프는 12장이다.

교과서는 그래프를 인접 리스트로 가르친다. 노드마다 이웃 목록을 둔다. 이 절은 그것을 세우고, 3.1 의 CSR 이 같은 그래프의 다른 표현임을 본 뒤, 둘의 메모리와 훑기 값을 잰다. 답은 4.1 과 같다. 목록이 흩어져 있으면 라인이 들고, 배열 둘에 모으면 안 든다.

학습 목표

방향 그래프를 정의하고, 인접 리스트와 CSR 과 간선 배열이 같은 그래프의 세 표현임을 말할 수 있다. 세 표현의 메모리와 이웃 훑기 비용을 적고, 정적인 그래프에서 CSR 이 이기는 이유를 라인으로 설명할 수 있다.

이 말부터

인접 리스트	노드마다 나가는 이웃의 목록. 목록이 노드 수만큼 있다
CSR	행 포인터 배열과 열 배열. 노드 u 의 이웃이 열 배열의 $[\text{ptr}[u], \text{ptr}[u+1])$ 에 연속으로 있다
간선 배열	(출발, 도착) 쌍의 배열 둘. 가장 단순하고, 이웃을 찾으려면 훑어야 한다

디딤돌

정의 11.1: 방향 그래프와 세 표현

노드 집합 $\{0, \dots, n-1\}$ 과 간선 집합 $E \subseteq V \times V$ 다. $|E| = m$ 이다. 인접 리스트는 노드 u 마다 $\{v : (u, v) \in E\}$ 의 목록이다. CSR 은 3.1 의 정의를 인접 행렬에 적용한 것으로, 열 배열이 모든 이웃 목록을 노드 순서로 이어 붙인 것이고 행 포인터가 경계다. 간선 배열은 E 를 순서 없이 늘어놓은 것이다.

불변식 셋 다 같은 E 를 나타낸다. CSR 에서 $\text{col}[\text{ptr}[u]..\text{ptr}[u+1])$ 이 u 의 이웃 전부이고 겹침이 없다.

불변식이 말하는 것은 표현이 셋이지 그래프가 셋이 아니라는 것이다. 다른 것은 이웃을 어디에 두느냐이고, 그것이 0.1 의 이동량이다. 셋의 메모리와 훑기 값을 한 명제에 나란히 적는다.

명제 11.2: 세 표현의 비용

> 노드 n, 간선 m 에서 인접 리스트는 목록 n 개와 항목 m 개이고 목록마다 따로 놓인다. CSR 은 정수 $n + 1 + m$ 개의 배열 둘이다. 간선 배열은 정수 $2m$ 개다. 노드 u 의 이웃 훑기는 인접 리스트가 목록 하나를 따라가는 것이고 라인 하나 이상이며, CSR 은 연속 구간 하나라 라인 $\lceil d_u s/B \rceil$ 개다. 전체 훑기는 인접 리스트가 라인 $n + ms/B$ 개 이상이고 CSR 은 $(n + m)s/B$ 개다. 간선 하나 더하기는 인접 리스트가 상수이고 CSR 은 열 배열을 다시 짜는 $O(m)$ 이다.

증명. 메모리는 셈이다. 이웃 훑기에서 인접 리스트는 목록의 주소를 읽고 목록으로 가야 하므로 4.1 의 의존 읽기 하나가 들고, CSR 은 포인터 둘을 읽으면 주소가 나온다. 전체 훑기는 그것을 n 번이다. CSR 의 삽입은 열 배열 중간에 끼워 넣는 것이라 뒤를 다 민다. ■

명제의 마지막 문장이 CSR 을 쓸 자리와 안 쓸 자리를 가른다. 간선이 계속 더해지면 인접 리스트이고, 한 번 만들고 읽기만 하면 CSR 이다. 계산 그래프는 뒤쪽이다. 모델을 정의하면 그래프가 정해지고, 그 뒤로 학습과 추론이 그 그래프를 수백만 번 읽는다.

예제 11.1: 노드 백만, 간선 사백만

> CSR 은 8 바이트 정수로 40 MB 다. 인접 리스트는 파이썬 리스트 백만 개에 정수 객체 사백만 개라 220 MB 다. 전체 훑기는 CSR 이 포인터 배열 8 MB 를 한 번 읽는 것이고, 인접 리스트는 리스트 백만 개를 하나씩 찾아가는 것이다.

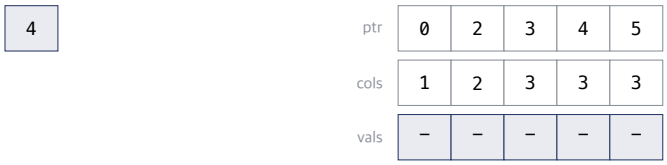


그림 11.1.. 노드 넷, 간선 다섯의 그래프를 CSR 로. 포인터가 경계이고 열이 이웃이다. 노드 0 의 이웃은 열의 $[0, 2)$ 다

예제의 다섯 배와 백 배가 실측에서 얼마인지 잰다. 훑기는 둘로 잰다. 전체를 한 번 도는 것과, 너비 우선 탐색으로 층마다 이웃을 모으는 것.

균열

노드 백만 개, 간선 사백만 개의 무작위 DAG 를 인접 리스트와 CSR 로 두고, 메모리와 만드는 시간과 두 훑기의 시간을 쟀다.

실험 11.1 같은 그래프, 세 표현. 메모리와 훑기

표현	MB	만들기 ms	전체 훑기 ms	너비 우선 탐색 ms
리스트의 리스트	221	8916	750	3418
CSR (포인터 + 열)	40.0	3992	3.19	924
간선 배열 (COO)	64.0	0.00	0.00	0.00

측정 조건. 간선의 간격은 평균 500 의 기하 분포. 너비 우선 탐색은 노드 0 에서 닿는 노드를 층 단위로 세고 두 표현이 같은 수를 냈다. 세 번 중앙값.

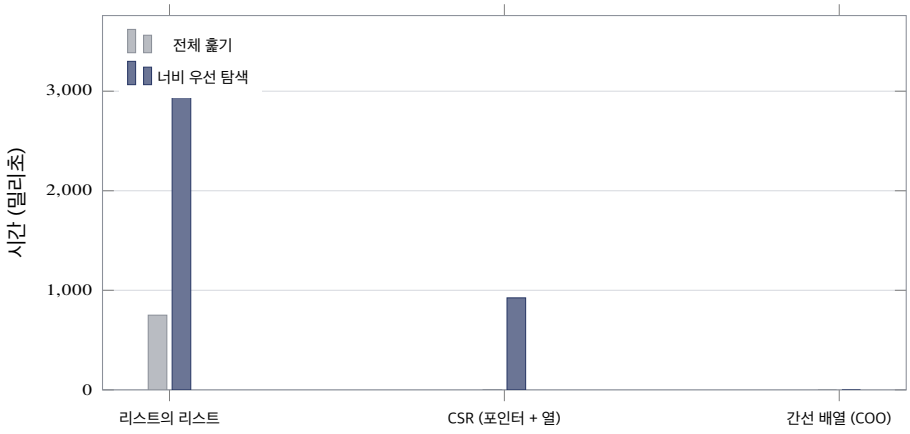


그림 11.2.. 세 표현의 전체 훑기와 너비 우선 탐색 시간. CSR 이 두 훑기 모두에서 짧다

메모리는 다섯 배 반이다. 221 MB 와 40 MB. 전체 훑기는 750 밀리초와 3 밀리초로 이백 배가 넘는다. CSR 은 포인터 배열의 차이만 더하면 되고 열 배열을 안 읽는다. 너비 우선 탐색은 3.4 초와 0.9 초로 네 배다. 층마다 이웃을 모으는 것이 CSR 에서는 배열 연산 몇 개이고 인접 리스트에서는 파이썬 루프다.

관찰

만드는 시간은 CSR 이 4 초, 인접 리스트가 9 초다. CSR 은 간선 배열을 정렬하는 것이라 6.4 의 정렬 기반 세기와 같은 일이고, 그것이 파이썬 리스트 사백만 번 붙이기보다 싸다. 정적인 그래프는 만드는 값도 정렬 하나다.

이 관찰이 6.4 의 명제를 그래프에 다시 쓴 것이다. 같은 키 (출발 노드) 를 모으는 것은 정렬이고, 모은 결과가 CSR 이다.

다시 쓰기

이 절의 다시 쓰기는 훑기를 노드 하나씩이 아니라 층 하나씩으로 하는 것이다. 층의 이웃 전부를 한 번에 모은다.

명제 11.3: 층 단위 훑기

> 너비 우선 탐색의 한 층 F 에 대해, F 의 모든 이웃은 열 배열의 구간 $|F|$ 개의 합집합이다. CSR 에서 그 구간들을 이어 붙이는 것은 포인터 $2|F|$ 개를 읽고 열 배열을 $\sum_{u \in F} d_u$ 만큼 순차로 읽는 것이다. 탐색 전체의 이동량은 포인터 $2n$ 개와 열 배열 한 번이고, 노드 하나씩 따라가는 인접 리스트의 라인 n 개 이상보다 작다. 층이 클수록 순차 읽기의 몫이 크다.

증명. 층의 노드가 열 배열에서 서로 다른 구간을 가리키고 구간 안은 연속이므로, 구간을 모으면 순차 읽기 $|F|$ 번이다. 모든 층을 합치면 열 배열을 정확히 한 번 읽는다. ■

이 명제가 3.4 의 gather 와 같은 모양이다. 번호 배열로 한꺼번에 읽는다. 그리고 12장의 근접 그래프 탐색이 이 층 단위 훑기 위에 선다. 알고리즘으로 적으면 층 하나가 네 줄이다.

알고리즘 11.1 층 단위 너비 우선 탐색 `dsai.graph.reach_count_csr`

> **입력:** CSR (ptr, col), 시작 s
> 본 것 $\leftarrow \{s\}$; 층 $F \leftarrow [s]$
> **조건** F 가 비지 않음 **동안**
> 　구간 $[\text{ptr}[u], \text{ptr}[u+1])$ 을 $u \in F$ 에 대해 이어 붙인 번호 배열 I
> 　후보 $\leftarrow \text{col}[I]$ 에서 중복과 본 것을 뺀 것; 본 것에 더한다; $F \leftarrow$ 후보
> **끝 조건**

구현

CSR 은 정렬 하나로 만든다. 출발 노드로 정렬하면 도착 노드가 열 배열이고, 출발 노드의 개수 누적합이 포인터다.

지금 보는 코드　간선 배열을 출발 노드로 정렬하고, 개수의 누적합을 포인터로 둔다.

구현 CSR 만들기와 이웃 `samples/dsai/graph.py:32`

```
32      def __init__(self, n: int, src: np.ndarray, dst: np.ndarray):
33          order = np.argsort(src, kind="stable")
34          self.n = n
35          self.col = dst[order].astype(np.int64)
36          counts = np.bincount(src, minlength=n)
37          self.ptr = np.zeros(n + 1, dtype=np.int64)
38          np.cumsum(counts, out=self.ptr[1:])
39
40      def neighbors(self, u: int) -> np.ndarray:
41          return self.col[self.ptr[u]:self.ptr[u + 1]]
```

읽어 보면 `argsort` 하나와 `bincount` 하나와 `cumsum` 하나다. `neighbors` 가 열 배열의 슬라이스이고 복사가 없다. 2.3 의 뷰다.

구현 층 단위 너비 우선 탐색 `samples/dsai/graph.py:98`

```python
def reach_count_csr(g: CSRGraph, start: int) -> int:
    """알고리즘 11.1. 같은 탐색을 CSR 위에서 층마다 배열 연산으로."""
    seen = np.zeros(g.n, dtype=bool)
    seen[start] = True
    frontier = np.array([start])
    count = 1
    while len(frontier):
        starts, ends = g.ptr[frontier], g.ptr[frontier + 1]
        idx = _ranges(starts, ends)
        cand = np.unique(g.col[idx])
        cand = cand[~seen[cand]]
        seen[cand] = True
        count += len(cand)
        frontier = cand
    return count
```

읽어 보면 `_ranges` 가 구간들을 이어 붙인 번호 배열이고 `col[idx]` 가 gather 다. `np.unique` 가 중복을 걷고 `seen` 이 본 것을 거른다. 층 하나가 배열 연산 넷이다.

분석

인접 리스트는 조정해서 유효하다. 정의는 그대로이고, 정적인 그래프에서는 목록을 배열 둘로 모은 CSR 이 그 자리를 대신한다. 이것은 3.1 의 CSR 을 행렬이 아니라 그래프로 읽은 것이고, 같은 구조다. 희소 행렬과 그래프는 같은 것의 두 이름이다. 7.4 의 중복 제거가 후보 쌍을 그래프로 보고 연결 성분을 찾던 것도 이 표현 위의 층 단위 훑기다. 후보 쌍이 간선이고, 한 성분이 중복 무리다.

이 구조가 서는 자리는 계산 그래프다. 모델 하나가 노드 수천에서 수만 개의 DAG 이고, 한 번 만들어 학습 내내 읽는다. 다음 절이 그 그래프 위에서 앞으로 한 번, 뒤로 한 번 가는 순서를 세우고, 그 뒤가 자동 미분이다.

현장에서

> 그래프 라이브러리와 희소 선형대수 라이브러리는 CSR 을 공유하고, 가속기 위의 그래프 신경망 구현은 이웃 모으기를 이 절의 층 단위 gather 로 한다 [확립]. 간선이 자주 바뀌는 그래프는 인접 리스트나 블록 단위의 절충 표현을 쓴다.

주의

> 실험 11.1 의 인접 리스트는 파이썬 리스트라 메모리와 시간이 모두 파이썬의 값이다. C 의 인접 리스트는 노드당 포인터 하나와 정수 배열이라 메모리 차이는 작다. 남는 것은 라인의 차이다.

그래프를 배열 둘로 두었다. 다음 절은 그 위에서 노드를 지나는 순서다. 간선이 앞뒤를 정하고, 그 순서의 역이 미분이다. 순서를 얻는 알고리즘 둘이 모두 이 절의 CSR 을 한 번 훑는다.

되짚기

> 방향 그래프의 세 표현을 정의하고, 정적인 그래프에서 CSR 이 메모리 5 분의 1, 전체 훑기 이백 배임을 봤다. 만드는 것은 정렬 하나다. 다음 절은 위상 정렬과 자동 미분이다.

11.2 위상 정렬과 자동 미분

계산 그래프의 노드는 연산이고, 연산은 입력이 다 계산된 뒤에야 할 수 있다. 그 순서가 위상 순서다. 간선 (u, v) 가 있으면 u 가 v 보다 앞이다. 5.2 의 테이프는 연산이 한 줄로 이어진 경우였다. 계산 그래프는 한 노드의 출력을 여럿이 쓰고, 여러 입력이 한 노드로 모인다. DAG 이고, 위상 순서가 여럿이다.

이 절은 위상 순서를 얻는 두 알고리즘을 세우고, 그 역순이 역방향 미분임을 정리로 세운다. 팬아웃이 있으면 기울기가 더해진다는 것이 테이프와 다른 점이다.

학습 목표

위상 순서를 정의하고, Kahn 의 알고리즘과 깊이 우선 탐색이 각각 $O(n + m)$ 에 위상 순서를 냄을 보일 수 있다. 계산 그래프 위의 역방향 미분이 위상 순서의 역순 한 번이며, 팬아웃에서 기울기가 합이 되는 이유를 연쇄 법칙으로 설명할 수 있다.

이 말부터

위상 순서	모든 간선 (u, v) 에 대해 u 가 v 보다 앞에 오는 노드의 나열. DAG 에만 있다
들어오는 차수	노드로 들어오는 간선의 수. 0 이면 지금 계산할 수 있다
팬아웃	한 노드의 출력을 쓰는 노드의 수. 역방향에서 그 수만큼의 기울기가 더해진다

디딤돌

정의 11.4: 위상 순서

DAG $G = (V, E)$ 의 위상 순서는 순열 π 로, 모든 $(u, v) \in E$ 에 대해 $\pi(u) < \pi(v)$ 다.

불변식 위상 순서로 노드를 지나면 어느 노드에 이르렀을 때 그 노드의 모든 입력이 이미 지나갔다.

사이클이 있으면 위상 순서가 없고, 없으면 적어도 하나 있다.

불변식이 계산 그래프의 전진 계산 그 자체다. 위상 순서대로 연산하면 입력이 언제나 준비되어 있다. 순서를 얻는 방법은 둘이고, 둘 다 11.1 의 CSR 위에서 선형이다.

명제 11.5: 두 위상 정렬

> Kahn 의 알고리즘은 들어오는 차수를 세고 0 인 노드를 큐로 꺼내며, 꺼낼 때 이웃의 차수를 하나씩 줄인다. $O(n+m)$ 이다 [확립] [44]. 깊이 우선 탐색은 노드가 끝나는 순서를 적고 그것을 뒤집는다. $O(n+m)$ 이고, 사이클은 보는 중인 노드를 다시 만나는 것으로 잡는다 [확립] [45]. 둘의 이동량은 CSR 을 한 번 훑는 것이고, Kahn 은 차수 배열을 무작위로 읽고 쓴다.

증명. Kahn 에서 노드는 차수가 0 이 될 때 정확히 한 번 큐에 들고, 간선은 그 노드가 꺼내질 때 정확히 한 번 본다. 큐에서 나온 순서는 모든 입력이 먼저 나온 뒤이므로 위상 순서다. 깊이 우선 탐색에서 노드 u 가 끝나는 것은 u 에서 닿는 모든 노드가 끝난 뒤이므로, 끝난 순서의 역이 위상 순서다. 간선마다 한 번 본다. ■

두 알고리즘은 같은 답의 다른 순서를 낸다. 계산 그래프에서는 어느 쪽이든 되고, 메모리를 아끼는 순서를 고르는 것이 컴파일러의 일이다. 그것은 이 책의 범위 밖이다. 이 절이 세우는 것은 그 순서의 역이 무엇이냐다.

정의 11.6: 스칼라 계산 그래프

> 노드마다 연산 종류와 부모 노드 둘까지, 값 하나, 기울기 하나를 둔다. 노드 번호가 위상 순서다. 전진은 번호 순서로 값을 계산한다. 역방향은 출력 노드의 기울기를 1 로 두고 번호의 역순으로 각 노드의 기울기를 부모에 나눠 더한다.
>
> **불변식** 역방향에서 노드 i 에 이르렀을 때, i 의 출력을 쓰는 모든 노드가 이미 지나갔고 i 의 기울기가 다 모였다.

불변식의 둘째 절이 이 정의의 요점이다. 기울기가 다 모인 뒤에야 그 노드를 지나므로, 지나면서 부모에 나눠 주는 값이 완성된 값이다. 그것이 정리다.

정리 11.7: 역방향 미분

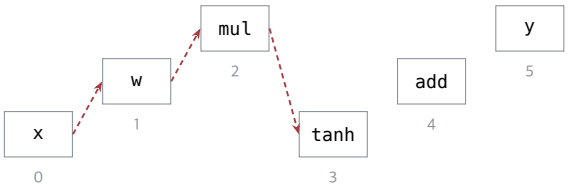
> 출력 y 와 노드 i 에 대해, 정의 11.6 의 역방향이 끝나면 $\mathrm{grad}[i] = \partial y / \partial v_i$ 다. 비용은 전진 한 번의 상수 배이고, 입력의 수와 무관하다.

증명. 연쇄 법칙으로 $\partial y/\partial v_i = \sum_{j : i \to j} (\partial y/\partial v_j)(\partial v_j/\partial v_i)$ 다. 합은 i 의 출력을 쓰는 노드 j 전부에 대한 것이고, 그것이 팬아웃이다. 역순으로 지나면 불변식에 의해 j 들이 먼저 지나가며 각자 $\mathrm{grad}[j] \cdot \partial v_j/\partial v_i$ 를 $\mathrm{grad}[i]$ 에 더했으므로 합이 완성된다. 노드마다 부모 둘에 곱셈 하나씩이라 전진의 상수 배다. ■

정리에서 읽을 것은 5.2 과 같고 하나가 더 있다. 입력이 백만 개여도 역방향 한 번이라는 것은 같다. 다른 것은 더하기다. 테이프에서는 노드마다 기울기가 하나 왔고, 그래프에서는 팬아웃만큼 온다. 그래서 대입이 아니라 누적이다.

예제 11.2: 입력이 두 번 쓰이는 식

$y = (\tanh(xw) + x)^2$ 에서 x 는 곱셈과 덧셈에 두 번 쓰인다. 역방향에서 덧셈 노드가 x 에 기울기 하나를 더하고, 곱셈 노드가 또 하나를 더한다. 둘의 합이 $\partial y/\partial x$ 다. 대입으로 쓰면 나중 것이 앞 것을 덮어 틀린다.



순서 0 에서 5. 역방향은 5 에서 0

그림 11.3.. 위상 순서로 놓인 노드 배열. 앞으로 읽으면 값이고 뒤로 읽으면 기울기다. 5.2 의 테이프가 그래프로 넓어진 것이다

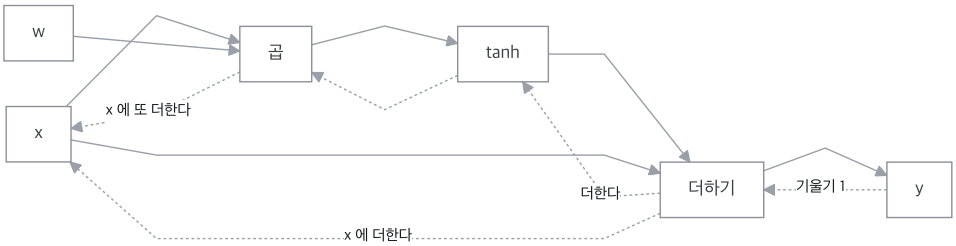


그림 11.4.. 계산 그래프의 전진과 역방향. 앞으로는 위상 순서, 뒤로는 그 역순. 팬아웃 노드에서 기울기가 더해진다

예제의 더하기가 맞는지는 유한 차분으로 검산한다. 그리고 두 위상 정렬의 시간을 잰다.

균열

노드 20만, 간선 80만 개의 DAG 에서 두 위상 정렬의 시간을 재고, 입력 8 개와 연산 400 개의 무작위 스칼라 그래프에서 역방향의 기울기를 유한 차분과 견줬다.

실험 11.2 두 위상 정렬

방법	노드	간선	ms
Kahn (큐와 차수)	200000	800000	981
DFS (명시적 스택)	200000	800000	8391

측정 조건. 둘 다 결과가 위상 순서임을 모든 간선에서 확인했다. 파이썬 루프, 세 번 중앙값.

Kahn 이 1 초, 깊이 우선 탐색이 8 초다. 둘 다 $O(n+m)$ 인데 여덟 배 갈린 것은 파이썬 스택 조작의 값이다. 노드마다 스택의 위를 읽고 바꾸는 문장 몇 개가 큐의 넣고 빼기보다 비싸다. C 에서는 둘이 비슷하다.

실험 11.3 역방향 미분의 검산과 값

일	ms	유한 차분과의 최대 차이
전진 한 번	4.91	0.00
역방향 한 번 (모든 입력의 기울기)	0.69	2.6e-10
유한 차분 (입력 8 개)	78.5	0.00

측정 조건. 입력 8 개, 연산 400 개. 유한 차분은 입력마다 전진 둘이라 전진 16 번의 값으로 적었다.

역방향의 기울기와 유한 차분의 차이가 10^{-10} 이다. 팬아웃의 합이 맞다. 그리고 역방향 한 번이 전진 한 번보다 짧다. 전진은 `tanh` 를 계산하고 역방향은 저장된 값을 곱하기만 한다. 유한 차분은 입력 8 개에 전진 16 번이고, 입력이 백만 개면 이백만 번이다. 역방향은 여전히 한 번이다.

관찰

역방향이 전진보다 싼 이유는 값을 저장했기 때문이다. 저장하지 않으면 역방향에서 다시 계산해야 하고, 그것이 15장의 재계산과 메모리의 맞바꿈이다.

다시 쓰기

이 절의 다시 쓰기는 역방향을 CSR 위의 훑기로 적는 것이다. 정의의 부모 둘은 팬인이 2 인 특수한 경우이고, 일반 그래프에서는 노드마다 들어오는 간선이 몇 개든 된다. 그러면 역방향은 간선을 뒤집은 그래프의 훑기다.

명제 11.8: 역방향은 전치 그래프의 위상 순서

계산 그래프 G 의 역방향 미분은 전치 그래프 G^T (간선을 뒤집은 것) 를 G 의 위상 순서의 역순으로 훑으며, 노드마다 G^T 의 이웃 (원래의 부모) 에 기울기를 더하는 것이다. CSR 로 두면 G^T 의 CSR 은 11.1 의 전치 하나이고, 역방향의 이동량은 G^T 를 한 번 훑는 것이다.

증명. 정리 11.7 의 합은 i 로 들어오는 것이 아니라 i 에서 나가는 간선 위의 합이고, 역순으로 지나며 나가는 간선을 따라 부모에 더하는 것은 G^T 의 나가는 간선을 따라가는 것이다. G^T 의 CSR 은 G 의 간선 배열을 도착 노드로 정렬한 것이다. ■

이 명제가 자동 미분 시스템이 그래프를 두 벌 두는 이유다. 전진용과 역방향용. 그리고 둘 다 정적이라 CSR 이다. Baydin 등은 2018 년의 조사에서 역방향 모드를 이 그래프 위의 두 번 훑기로 정리했고 [확립] [19], 5.2 의 테이프가 그 특수한 경우였다.

알고리즘 11.2 Kahn 의 위상 정렬 `dsai.topo.topo_kahn`

입력 : CSR, 들어오는 차수 d

큐 $Q \leftarrow \{u : d[u] = 0\}$
조건 Q 가 비지 않음 **동안**
    $u \leftarrow Q$ 에서 꺼냄 ; 순서에 더한다
    모든 $v \in$ 이웃(u) **동안** $d[v] \leftarrow d[v] - 1$; $d[v] = 0$ 이면 Q 에 넣는다
    끝 반복
끝 조건
순서의 길이가 n 이 아니면 사이클이다

위상 정렬이 앞으로 가는 순서를 주면, 역방향 미분은 그 순서를 거꾸로 밟는다. 두 알고리즘의 모양이 같고, 다른 것은 노드에서 하는 일 하나뿐이다.

알고리즘 11.3 역방향 미분 `dsai.topo.ScalarGraph.backward`

입력 : 스칼라 그래프, 출력 y

grad $\leftarrow 0$; grad$[y] \leftarrow 1$
반복 $i = y, y-1, \dots, 0$ **동안**
    모든 부모 p 와 그 편미분 $\partial v_i / \partial v_p$ **동안** grad$[p] \leftarrow$ grad$[p]$ + grad$[i] \cdot \partial v_i / \partial v_p$
    끝 반복
끝 반복

구현

Kahn 은 큐 하나와 차수 배열 하나다. 역방향은 배열의 역순 루프이고, 연산마다 부모에 더하는 줄 둘이다.

지금 보는 코드 차수가 0 인 노드를 꺼내고, 그 이웃의 차수를 줄이고, 0 이 되면 넣는다.

구현 Kahn 의 위상 정렬 `samples/dsai/topo.py:17`

```python
def topo_kahn(g: CSRGraph) -> np.ndarray:
    """알고리즘 11.2. 들어오는 간선이 0 인 노드부터. 큐 하나와 차수 배열 하나."""
    indeg = g.in_degree().copy()
    q = deque(np.flatnonzero(indeg == 0).tolist())
    order = np.empty(g.n, dtype=np.int64)
    k = 0
    while q:
        u = q.popleft()
        order[k] = u
        k += 1
        for v in g.neighbors(u):
            indeg[v] -= 1
            if indeg[v] == 0:
                q.append(v)
    if k != g.n:
        raise ValueError("사이클이 있다")
    return order
```

읽어 보면 `deque` 가 5.3 의 큐다. `indeg[v] -= 1` 이 CSR 의 열 배열을 따라가며 차수 배열을 무작위로 쓰는 자리이고, 명제의 이동량 문장이 여기다.

구현 역방향 미분 `samples/dsai/topo.py:94`

```
    def backward(self, out: int) -> np.ndarray:
        """알고리즘 11.4. 뒤에서 앞으로 한 번.
        부모마다 기울기를 더한다 (팬아웃의 합)."""
        grad = np.zeros(self.n)
        grad[out] = 1.0
        for i in range(out, -1, -1):
            o, g = self.op[i], grad[i]
            if g == 0.0:
                continue
            if o == 1:
                grad[self.a[i]] += g
                grad[self.b[i]] += g
            elif o == 2:
                grad[self.a[i]] += g * self.val[self.b[i]]
                grad[self.b[i]] += g * self.val[self.a[i]]
            elif o == 3:
                grad[self.a[i]] += g * (1.0 - self.val[i] ** 2)
        return grad
```

읽어 보면 모든 줄이 `+=` 다. 팬아웃의 합이 이 기호 하나에 있다. `if g == 0.0: continue` 는 출력에 안 이어진 노드를 건너뛰는 것이다.

분석

위상 정렬은 그대로 유효하다. 1962 년과 1972 년의 두 알고리즘이 그대로이고 둘 다 선형이다. 바뀐 것은 그 순서의 역이 하는 일이다. 순서는 계산이고 역순은 미분이다. 두 방향이 같은 그래프 위에 있고, 그래프가 정적이라 두 방향 다 CSR 하나씩이다.

이 구조가 서는 자리는 모든 학습이다. 모델의 손실에서 모든 매개변수로의 기울기가 역방향 한 번이고, 매개변수가 수십억 개여도 한 번이다. 그리고 이 그래프 위에서 노드를 합치는 것이 다음 절이다. 순서가 정해져 있으니 이웃한 노드를 하나로 묶을 수 있다.

현장에서

> Kahn 의 1962 년 논문은 PERT 네트워크 3 만 개 활동의 순서를 한 시간 안에 낸 것을 성과로 적었다 [확립] [44]. 지금은 같은 알고리즘이 노드 수만 개의 계산 그래프를 밀리초에 정렬한다. Tarjan 의 1972 년 논문은 깊이 우선 탐색으로 강한 연결 성분을 선형에 찾았고, 사이클 검출이 그 부산물이다 [확립] [45].

주의

> 실험 11.2 의 스칼라 그래프는 노드가 스칼라다. 실제 계산 그래프의 노드는 텐서이고, 노드 하나의 역방향이 행렬 곱이다. 그래프 훑기의 값은 노드 계산의 값에 묻히고, 이 절의 시간은 훑기의 값이다.

앞으로 한 번, 뒤로 한 번이다. 다음 절은 그 그래프의 노드를 합쳐 이동량을 줄인다. 그래프를 다시 쓰는 것이고, 컴파일러가 하는 일이다. 순서가 정해져 있으니 이웃한 노드를 묶어도 값이 안 바뀐다.

되짚기

> 위상 순서를 정의하고 두 알고리즘이 선형임을 보였다. 역방향 미분이 그 역순 한 번이며 팬아웃에서 합이 됨을 정리로 세우고 유한 차분으로 검산했다. 다음 절은 노드를 합치는 연산자 융합이다.

11.3 연산자 융합. 그래프를 다시 쓰다

계산 그래프의 노드 하나가 연산 하나이고, 연산 하나는 입력 텐서를 읽고 출력 텐서를 쓴다. 원소별 연산 다섯 개가 이어지면 배열 전체를 다섯 번 읽고 다섯 번 쓴다. 0.2 의 루프라인에서 원소별 연산은 밀도가 가장 낮은 자리다. 원소 하나에 연산 하나이고 이동이 16 바이트다. 다섯 개를 하나로 합치면 연산 다섯에 이동 16 바이트다.

합치는 것이 연산자 융합이고, 그것은 그래프의 노드 여럿을 노드 하나로 바꾸는 다시 쓰기다. 이 절은 그 다시 쓰기를 정의하고, 이동량이 k 분의 1 이 됨을 명제로 세운 뒤, NumPy 에서 블록으로 흉내 내어 잰다.

학습 목표

연산자 융합을 계산 그래프의 다시 쓰기 규칙으로 정의하고, 원소별 사슬 k 개의 이동량이 융합으로 k 분의 1 이 됨을 보일 수 있다. 융합이 이기는 조건을 루프라인으로 적고, 연산이 비쌀수록 이점이 주는 이유를 실측으로 말할 수 있다.

이 말부터

원소별 연산	출력의 원소 i 가 입력의 원소 i 에만 달린 연산. 덧셈, 곱셈, tanh 같은 것
융합	그래프의 노드 여럿을 하나로 합쳐 중간 텐서를 메모리에 안 내보내는 것
사슬	원소별 노드가 한 줄로 이어지고 중간 출력을 다른 노드가 안 쓰는 부분 그래프

디딤돌

정의 11.9: 융합 가능한 사슬과 융합

계산 그래프에서 노드 $v_1 \to v_2 \to \dots \to v_k$ 가 사슬이라는 것은 모두 원소별 연산이고, $v_1, \dots, v_{k-1}$ 의 출력을 다음 노드 말고는 아무도 안 쓴다는 것이다. 융합은 사슬을 노드 하나 $f = f_k \circ \dots \circ f_1$ 로 바꾸는 것이다.

불변식 융합 전후의 그래프는 모든 출력 텐서에서 같은 값을 낸다. 사라진 것은 중간 텐서 $k - 1$ 개다.

불변식이 말하는 것은 값은 그대로이고 텐서만 사라진다는 것이다. 사라진 텐서는 메모리에 안 쓰이고 안 읽히므로, 그것이 곧 이동량이다. 얼마나 사라지는지를 세면 명제 하나다.

명제 11.10: 융합의 이동량

> 원소 n 개, 원소 크기 s 의 사슬 k 개를 하나씩 하면 이동량이 $2kns$ 이고, 융합하면 $2ns$ 다. 연산량은 둘 다 kn 이다. 따라서 연산 밀도가 $1/(2s)$ 에서 $k/(2s)$ 로 k 배 오른다. 루프라인의 능선 아래이면 시간이 k 분의 1 에 가깝고, 위이면 연산이 시간을 정해 이점이 준다.

증명. 연산마다 입력을 읽고 출력을 쓰므로 $2ns$ 이고 k 개면 $2kns$ 다. 융합하면 입력 한 번 읽기와 출력 한 번 쓰기다. 연산량은 안 바뀐다. 밀도는 연산을 이동으로 나눈 것이다. ■

명제의 둘째 문장이 이 절의 실험이다. 곱셈과 덧셈은 능선 아래이고 tanh 와 exp 는 위에 가깝다. 같은 융합이 앞에서는 크게, 뒤에서는 작게 이긴다.

예제 11.3: 원소 3,200만 개, 연산 다섯

> float64 3,200만 개는 256 MB 다. 하나씩 하면 이동 2.56 GB, 융합하면 0.51 GB 다. 대역폭 10 GB/s 이면 256 밀리초와 51 밀리초다. 연산이 원소당 1 나노초이면 160 밀리초라 융합 뒤에는 연산이 시간을 정한다.

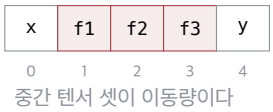
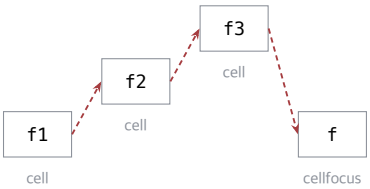


그림 11.5.. 하나씩 하는 사슬. 중간 텐서 셋이 메모리에 나갔다 돌아온다. 융합하면 진한 칸 셋이 사라진다



$$x \to f_1 \to f_2 \to f_3 \to y \text{ 가 } x \to f \to y \text{ 가 된다}$$

그림 11.6.. 사슬 셋의 융합. 왼쪽은 노드 셋과 중간 텐서 둘, 오른쪽은 노드 하나. 값은 같고 텐서 둘이 없다

NumPy 는 융합을 안 한다. 연산마다 배열 전체를 만든다. 그래도 융합의 효과는 잴 수 있다. 캐시에 드는 블록마다 사슬 전체를 돌면 중간 배열이 캐시 안에 머물러 메모리로 안 나간다.

균열

float64 3,200만 개에서 원소별 연산 다섯 개의 사슬 둘 (싼 것과 비싼 것) 을 하나씩 하는 것과 블록마다 사슬 전체를 도는 것으로 쟀다.

실험 11.4 원소별 사슬 다섯 개. 하나씩과 블록 융합

사슬	방식	블록 원소	ms	명제의 이동 GB	배속
싼 사슬	하나씩	0	1204	2.56	1.00
싼 사슬	블록 4K	4096	287	0.51	4.20
싼 사슬	블록 64K	65536	232	0.51	5.19
싼 사슬	블록 1024K	1048576	304	0.51	3.96
비싼 사슬	하나씩	0	4519	2.56	1.00
비싼 사슬	블록 4K	4096	3553	0.51	1.27
비싼 사슬	블록 64K	65536	3706	0.51	1.22
비싼 사슬	블록 1024K	1048576	3497	0.51	1.29

측정 조건. 싼 사슬은 곱셈, 덧셈, 제곱, 부호, 절댓값. 비싼 사슬은 tanh, exp, sqrt, log1p, sin. 이동량 열은 명제의 값이고 실측이 아니다. 세 번 중앙값.

싼 사슬에서 블록 64K 가 5.2 배 빠르다. 표의 배속 열이 하나씩 한 시간을 블록 융합 시간으로 나눈 것이다. 명제의 $k = 5$ 그대로다. 블록 4K 는 4.2 배로 블록마다 NumPy 호출 다섯 번의 값이 보이고, 1M 은 4.0 배로 블록이 캐시를 넘친다. 비싼 사슬에서는 1.2 배에서 1.3 배다. 연산이 시간을 정하는 자리라 이동을 줄여도 그만큼이다.

관찰

융합의 이점은 연산 밀도가 능선 아래일 때 k 배이고 위일 때 1 배에 가깝다. 같은 그래프 다시 쓰기가 사슬의 연산에 따라 다섯 배와 1.3 배다.

이 관찰이 0.2 의 루프라인을 그래프 다시 쓰기의 잣대로 만든다. 융합할지 말지는 밀도가 정하고, 밀도는 연산이 정한다.

다시 쓰기

이 절의 다시 쓰기는 융합을 어디까지 넓히느냐다. 원소별 사슬은 가장 쉬운 경우다.

명제 11.11: 융합의 경계

원소별 연산과 축소 연산 (합, 최대) 은 축소가 사슬의 끝이면 융합된다. 축소 뒤에 원소별 연산이 오면 축소 결과가 전부 나와야 하므로 경계가 생긴다. 행렬 곱 뒤의 원소별 연산은 행렬 곱의 출력 타일마다 원소별 연산을 적용하는 것으로 융합되고, 그것이 3.2 의 타일 안에서 끝난다. 경계를 넘는 융합은 중간 텐서를 다시 계산하는 것으로 가능하며, 메모리와 연산의 맞바꿈이다.

증명. 원소별 연산은 원소 하나만 있으면 되고 축소는 전부가 있어야 한다. 사슬의 끝에 축소가 있으면 원소마다 사슬을 돌며 누적하면 되고, 축소 뒤에 원소별이 오면 누적이

끝나야 다음이 시작된다. 행렬 곱의 타일은 출력 원소의 부분 합이고, 타일이 끝나면 그 원소들은 완성되므로 원소별 연산을 이어 붙일 수 있다. ■

이 명제가 15장의 어텐션 융합의 뼈대다. 소프트맥스는 축소 하나 (최대) 와 원소별 (지수) 과 축소 하나 (합) 이고, 그 뒤에 행렬 곱이 온다. 경계가 둘인데 온라인 소프트맥스가 그 경계를 지운다. Chen 등의 TVM 은 이 융합을 그래프 수준의 규칙으로 두고, 어느 노드 쌍이 융합되는지를 연산의 종류로 분류했다 [확립] [46].

알고리즘 11.4 블록 융합 `dsai.fusion.fused_blocks`

입력: 입력 $x[0..n)$, 사슬 $f_1, \dots, f_k$, 블록 b

모든 블록 $[s, s+b)$ **동안**
  $y \leftarrow x[s..s+b)$; $y \leftarrow f_1(y)$; …; $y \leftarrow f_k(y)$ ▷ y 가 캐시에 머문다
  $\mathrm{out}[s..s+b) \leftarrow y$
끝 반복

구현

융합 가능한 사슬 찾기와 블록 융합이다. 사슬 찾기는 그래프를 위상 순서로 한 번 훑는다.

지금 보는 코드 블록 하나를 잘라 사슬 전체를 돌리고 결과를 쓴다. 중간 배열이 블록 크기다.

구현 블록 융합 `samples/dsai/fusion.py:26`

```
def fused_blocks(x: np.ndarray, out: np.ndarray, block: int, chain=CHAIN) -> np.ndarray:
    """알고리즘 11.5. 블록마다 사슬 전체를 돈다. 중간 결과가 캐시 안에 머문다."""
    n = len(x)
    for s in range(0, n, block):
        e = min(s + block, n)
        y = x[s:e]
        for f in chain:
            y = f(y)
        out[s:e] = y
    return out
```

읽어 보면 안쪽 `for` 가 사슬이고 바깥 `for` 가 블록이다. `y` 가 블록 크기라 캐시에 머문다. 실험의 5.2 배가 이 두 겹 루프의 순서다.

구현 융합 가능한 사슬 찾기 `samples/dsai/fusion.py:45`

```python
def fusable_chains(op: np.ndarray, indeg: np.ndarray, outdeg: np.ndarray,
                   elementwise: set[int]) -> list[list[int]]:
    """정의 11.7. 융합 가능한 사슬 찾기.
    원소별 연산이고 출력을 하나만 쓰는 노드를 잇는다."""
    chains, cur = [], []
    for i in range(len(op)):
        if op[i] in elementwise and indeg[i] <= 1 and (not cur or outdeg[cur[-1]] == 1):
            cur.append(i)
        else:
            if len(cur) > 1:
                chains.append(cur)
            cur = [i] if op[i] in elementwise else []
    if len(cur) > 1:
        chains.append(cur)
    return chains
```

읽어 보면 조건이 정의 11.9 그대로다. 원소별이고, 들어오는 것이 하나이고, 앞 노드의 출력을 이 노드만 쓴다. 조건이 깨지면 사슬을 닫고 새로 시작한다.

분석

계산 그래프의 다시 쓰기는 이 책의 잣대로 새 자리다. 교과서의 그래프 알고리즘은 그래프를 읽었고, 여기서는 그래프를 고친다. 고치는 규칙이 0.1 의 이동량에서 나온다. 그래프는 정적이라 CSR 이고, 다시 쓰기는 컴파일 때 한 번이므로 그 값은 학습 전체에서 사라진다.

이 구조가 서는 자리는 모든 딥러닝 컴파일러다. 그래프를 받아 융합하고, 융합된 노드마다 커널을 만든다. 실험의 블록이 그 커널의 흉내다. 그리고 15장의 어텐션이 이 융합의 가장 큰 예다. 소프트맥스와 행렬 곱 둘을 하나로 합쳐 어텐션 행렬을 메모리에 안 낸다.

현장에서

> TVM 은 연산을 단사, 축소, 복합 (행렬 곱 꼴), 불투명의 네 종류로 나누고 단사끼리와 단사 뒤의 축소를 융합 규칙으로 두었다 [확립] [46]. 이 절의 사슬은 단사끼리이고, 명제 11.11 의 경계가 그 분류의 이유다.

주의

> 실험 11.3 의 블록 융합은 NumPy 호출을 블록마다 다섯 번 하므로 진짜 융합보다 느리다. 진짜 융합은 원소 하나에 연산 다섯을 레지스터에서 하고, 그러면 블록마다 드는 호출의 값도 사라진다. 비싼 사슬의 1.3 배는 융합으로는 못 줄이는 값이다. 연산 자체가 시간이다.

그래프를 세우고, 순서를 세우고, 그래프를 고쳤다. 계산 그래프는 여기까지다. 다음 장의 그래프는 노드가 벡터이고 간선이 가깝다는 관계다. 그 그래프 위에서 0.3 의 kNN 을 다시 푼다.

되짚기

연산자 융합을 사슬을 노드 하나로 바꾸는 다시 쓰기로 정의하고, 이동량이 k 분의 1 이 됨을 보였다. 싼 사슬에서 5.2 배, 비싼 사슬에서 1.3 배였고 그 차이가 루프라인이다. 다음 장은 근접 그래프와 벡터 검색이다.

11장을 닫으며

이 장이 세운 구조는 셋이다. 배열 둘로 둔 그래프, 위상 순서와 그 역순의 미분, 그리고 사슬을 노드 하나로 바꾸는 융합. 셋 다 그래프가 정적이라는 조건 위에 있다. 한 번 만들고 수백만 번 읽는 그래프라서 CSR 이고, 순서를 미리 정하고, 컴파일 때 고친다.

세 실험의 숫자를 적어 둔다. 노드 백만 개의 그래프가 CSR 로 40 MB, 인접 리스트로 221 MB 였고 전체 훑기가 3 밀리초와 750 밀리초였다. Kahn 의 정렬이 1 초, 깊이 우선 탐색이 8 초였으며 역방향 미분의 기울기가 유한 차분과 10^{-10} 안에서 같았다. 원소별 사슬 다섯 개를 블록으로 융합하자 싼 사슬에서 5.2 배, 비싼 사슬에서 1.3 배였다.

정정 하나를 남긴다. 그래프 실험의 너비 우선 탐색을 처음에는 간선 간격 평균 20 의 DAG 에서 쟀다. 층이 5만 개가 넘어 CSR 쪽이 층마다 배열 호출 넷의 값을 5만 번 치렀고, 인접 리스트보다 느렸다. 간격을 500 으로 늘려 층을 수천 개로 줄이자 네 배 빨라졌다. 층 단위 훑기는 층이 넓을 때의 구조이고, 그래프의 모양이 그것을 정한다. 잰 것은 알고리즘이 아니라 그래프였다.

옛 구조의 판정

> 인접 리스트는 조정해서 유효하다. 정의는 그대로이고, 정적인 그래프에서는 CSR 이 그 자리를 대신한다. 근거는 실험 11.1 이다. 메모리 5 분의 1 과 훑기 이백 배가 같은 그래프의 표현 차이에서 나왔다. 위상 정렬과 역방향 미분은 그대로 유효하고, 융합은 교과서에 없던 다시 쓰기다.

이 장의 요약

> 1. 인접 리스트, CSR, 간선 배열은 같은 그래프의 세 표현이다. 정적이면 CSR 이고, 만드는 것은 정렬 하나다 11.1
> 2. 층 단위 훑기는 열 배열을 한 번 순차로 읽는다. 12장의 그래프 탐색이 이 위에 선다 11.1
> 3. 위상 순서는 전진 계산이고, 그 역순이 역방향 미분이다. 팬아웃에서 기울기가 더해진다 11.2
> 4. 역방향은 전치 그래프의 훑기다. 그래프가 두 벌이고 둘 다 CSR 이다 11.2
> 5. 융합은 사슬을 노드 하나로 바꿔 이동량을 k 분의 1 로 한다. 이점은 루프라인의 능선 아래에서만 k 배다 11.3
>
> 다음 장에서 뒤집히는 것: 이 장의 그래프는 사람이 그린 것이다. 12장의 그래프는 데이터가 만든다. 벡터가 노드이고 가까움이 간선이며, 그 위에서 탐색이 확률로 답한다.

필자의 판단

> 필자는 자동 미분을 그래프 수업의 위상 정렬 바로 뒤에 둔다. 위상 정렬을 배우고 나면 역방향 미분은 그 역순에 더하기 하나를 넣은 것이고, 학생은 미분 시스템을 신기한 것으로 보지 않게 된다. 융합은 컴파일러 수업의 것이지만, 이동량이라는 잣대 하나로 자료 구조 수업에서도 설 자리가 있다.

열린 문제

하나, 실험 11.2 의 깊이 우선 탐색이 Kahn 보다 여덟 배 느린 것은 파이썬의 값이라 두 알고리즘의 이동량 차이를 재지 못했다.
둘, 실험 11.3 의 블록 융합은 NumPy 호출을 블록마다 하므로 진짜 융합의 배속이 얼마인지 모른다. 싼 사슬의 5.2 배가 명제의 5 를 넘은 것은 하나씩 할 때의 중간 배열 할당 값이 빠진 몫이고, 그 몫을 따로 재지 않았다.
셋, 위상 순서가 여럿일 때 중간 텐서의 최대 메모리를 최소로 하는 순서를 이 장은 다루지 않았다. 그것은 NP 어려운 문제이고 컴파일러가 어림으로 푼다.

연습문제

11.1 기초 노드 10^5, 간선 10^6 의 그래프를 4 바이트 정수의 CSR 로 두면 몇 MB 인가. 명제 11.2 로 구하고, 노드마다 포인터 8 바이트와 정수 배열의 인접 리스트와 견줘라.

11.2 기초 노드 n, 간선 m 의 DAG 에서 Kahn 의 알고리즘이 큐에 넣는 횟수와 차수 배열을 쓰는 횟수를 명제 11.5 로 적어라.

11.3 기초 $y = x \cdot \tanh(x) + x$ 의 계산 그래프를 그리고, 정리 11.7 대로 역방향을 손으로 밟아 $x = 0.5$ 에서 $\partial y / \partial x$ 를 구하라. x 에 몇 번 더해지는가.

11.4 기초 float32 원소 10^8 개의 원소별 사슬 여덟 개를 하나씩 하면 이동량이 얼마이고 융합하면 얼마인가. 명제 11.10 로 구하고, 대역폭 800 GB/s 의 가속기에서 각각 몇 밀리초인가.

11.5 응용 `dsai.graph.CSRGraph` 에 간선을 더하는 `add_edges` 를 두 방식으로 구현하라. 열 배열을 다시 짜는 것과, 새 간선을 따로 모아 두었다가 일정 수가 쌓이면 합치는 것. 간선 만 개를 백 개씩 더할 때 시간을 견줘라.

11.6 응용 `dsai.topo.topo_dfs` 의 스택 항목을 튜플 대신 배열 둘 (노드, 다음 이웃 번호) 로 바꾸고 실험 11.2 의 시간이 얼마나 주는지 재라.

11.7 응용 `dsai.topo.ScalarGraph` 에 나눗셈과 exp 를 더하고, 유한 차분 검산이 통과하는지 확인하라. 팬아웃이 셋인 노드를 만들어 기울기가 세 번 더해지는지 시험으로 보여라.

11.8 응용 `dsai.fusion.fusable_chains` 를 축소 연산이 사슬의 끝에 올 수 있게 넓혀라. 명제 11.11 의 첫 문장이다. 축소 뒤에 원소별이 오면 사슬이 끊기는지 시험하라.

11.9 탐구 역방향 미분에서 값을 저장하지 않고 다시 계산하면 메모리와 연산이 어떻게 맞바뀌는지 정리 11.7 의 비용으로 논하라. 그래프를 $\sqrt{n}$ 개의 구간으로 나눠 구간 경계만 저장하면 어떻게 되는가.

11.10 탐구 위상 순서가 여럿일 때 중간 텐서의 최대 메모리가 순서에 따라 달라지는 그래프를 하나 만들고, 두 순서의 최대 메모리를 견줘라.

더 읽을 것

[44] 위상 정렬의 원전. 다섯 쪽이고, 들어오는 것이 없는 노드부터 꺼내는 규칙이 그대로 있다. 11.2
[45] 깊이 우선 탐색으로 강한 연결 성분을 선형에 찾는 논문. 명시적 스택의 위상 정렬과 사이클 검출이 여기서 나왔다. 11.2
[19] 자동 미분 조사. 역방향 모드가 그래프의 두 번 훑기라는 정리와 역사가 있다. 11.2

[46] 딥러닝 컴파일러. 연산을 네 종류로 나눈 융합 규칙이 11.3 절의 경계 명제와 대응한다. 11.3

12장

근접 그래프와 벡터 검색

1998년의 논문 하나는 고차원 유사도 검색의 분할 구조 전부가 차원 10 남짓에서 순차 훑기에 진다는 것을 보였다 [확립] [47]. 트리가 아니라 공간이 문제였다. 그 자리에서 정확한 답을 버리고 확률로 찾는 그래프가 나왔고, 지금 벡터 검색의 대부분이 그 그래프 위에 있다. 이 장은 트리가 무너지는 자리를 재고, 무너지지 않는 두 구조를 세운다. 층을 확률로 올리는 리스트와, 그 층을 거리 위에 놓은 그래프. 그리고 그래프가 디스크로 갈 때 접근을 세는 법이다.

이 장은 0.2 의 셋째 가정을 7장에 이어 둘째로 놓는다. 정확한 답 대신 재현율이고, 그것이 0.3 의 잣대였다. 그리고 8.1 의 높이 논증이 고차원에서 무너지는 자리를 정면으로 본다. 8장의 트리가 이 장에서 이제 안 맞는다는 판정을 받는다.

이 장에서 세우는 것

12.1 차원의 저주. kd-트리가 8차원과 16차원 사이에서 전부를 본다. 거리 대비가 그 이유

12.2 스킵 리스트. 층을 확률로 올린다. 평균 15 걸음으로 트립보다 적다

12.3 탐색 가능한 작은 세상 그래프. 거리 계산 250 번으로 재현율 99.6 퍼센트

12.4 디스크 위의 그래프. 같이 두기와 거친 벡터로 접근을 3 분의 1 로

12.1 차원의 저주. kd-트리가 무너지는 자리

0.3 의 kNN 은 벡터 n 개 중 질의에 가장 가까운 k 개를 찾는 일이었고, 전부 비교하면 nd 였다. 8.1 의 트리는 정수 키 n 개에서 로그로 찾았다. 벡터에도 같은 트리가 있다. 축 하나씩 번갈아 중앙값으로 나누는 kd-트리이고, 1975 년에 Bentley 가 세웠다 [확립] [48]. 2 차원과 3 차원에서는 로그다. 이 절은 그 트리를 세우고 차원을 올려 가며 잰다. 어느 차원부터 트리가 전부를 들여다보는지, 그리고 왜 그런지.

답은 자료 구조가 아니라 공간에 있다. 차원이 높으면 모든 점이 거의 같은 거리에 있고, 가장 가까운 점이 평균보다 별로 안 가깝다. 가지치기가 자를 것이 없다.

학습 목표

kd-트리를 정의하고 최근접 탐색의 가지치기 조건을 적을 수 있다. 차원이 오르면 최근접 거리와 평균 거리의 비가 1 로 가고 가지치기가 무너지는 이유를 설명하며, 어느 차원에서 전부 비교와 같아지는지 실측으로 말할 수 있다.

이 말부터

kd-트리	깊이마다 축 하나를 골라 그 축의 중앙값으로 점을 둘로 나눈 이진 트리. 잎이 점 몇 개의 묶음이다
가지치기	지금까지 찾은 최근접 거리보다 분할면까지의 거리가 멀면 그 쪽 부분 트리를 안 보는 것
거리 대비	최근접 거리를 평균 거리로 나눈 것. 0 에 가까우면 가까운 점이 두드러지고 1 에 가까우면 다 비슷하다

디딤돌

정의 12.1: kd-트리

점 집합 $P \subset \mathbb{R}^d$ 와 잎 크기 b 에 대해, $|P| \le b$ 이면 잎이다. 아니면 깊이 t 의 축 $a = t \bmod d$ 를 골라 P 를 축 a 의 중앙값 s 로 왼쪽 ($x_a < s$) 과 오른쪽으로 나누고 각각을 재귀로 만든다.

불변식 노드의 왼쪽 부분 트리의 모든 점은 그 노드의 축에서 분할값 미만이고 오른쪽은 이상이다. 어느 점도 두 쪽에 다 있지 않다.

최근접 탐색은 질의가 든 쪽을 먼저 내려가고, 돌아오며 분할면까지의 거리 $|q_a - s|$ 가 지금까지의 최근접 거리보다 작을 때만 반대쪽을 본다.

불변식이 가지치기의 정당성이다. 반대쪽의 모든 점은 분할면 너머에 있으므로 질의에서 적어도 $|q_a - s|$ 만큼 떨어져 있다. 그 거리가 이미 찾은 것보다 멀면 반대쪽에 답이 없다. 그런데 이 조건이 얼마나 자주 참이냐가 차원에 달렸다.

명제 12.2: 가지치기가 무너지는 이유

> 점이 d 차원에 고르게 흩어져 있으면, 최근접 거리와 평균 거리의 비가 d 가 커질수록 1 로 간다. 거리의 분포가 평균 근처로 몰리기 때문이다. 분할면까지의 거리는 축 하나의 차이라 d 와 무관하게 $O(1)$ 이고, 최근접 거리는 $\sqrt{d}$ 에 비례해 는다. 그래서 d 가 크면 거의 모든 반대쪽이 조건을 통과하고, 들여다보는 점의 수가 n 에 가까워진다.

증명. 좌표가 독립이고 분산이 같으면 두 점 사이 거리의 제곱은 d 개 항의 합이라 평균이 d 에 비례하고 표준편차가 $\sqrt{d}$ 에 비례한다. 상대 편차가 $1/\sqrt{d}$ 로 줄어 모든 거리가 평균 근처에 몰린다. 가지치기 조건 $|q_a - s| < r$ 에서 왼쪽은 $O(1)$, 오른쪽 r 은 $\Theta(\sqrt{d})$ 이므로 d 가 크면 거의 언제나 참이다. Weber 등은 1998 년에 분할 방법 전부가 차원 10 남짓에서 순차 훑기에 진다는 것을 보였다 [확립] [47]. ■

명제가 말하는 것은 트리가 틀렸다는 것이 아니다. 트리는 정확한 답을 낸다. 다만 답을 내는 데 전부를 보고, 그러면 전부 비교에 트리의 값을 더한 것이다. 이것이 8.1 의 높이 논증이 고차원에서 무너지는 자리다. 높이는 로그인데, 내려간 뒤에 반대쪽을 거의 다 다시 올라간다.

예제 12.1: 2 차원과 128 차원

> 표준 정규 점 십만 개에서 2 차원의 최근접 거리는 평균 거리의 0.4 퍼센트다. 가까운 점이 아주 가깝다. 128 차원에서는 77 퍼센트다. 가장 가까운 점이 평균의 4 분의 3 거리에 있다. 2 차원의 트리는 점의 0.2 퍼센트를 보고, 128 차원의 트리는 전부를 본다.

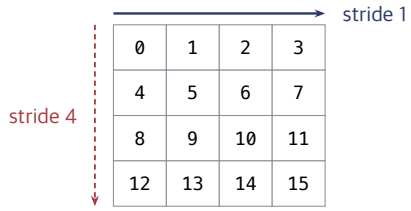


그림 12.1.. 2 차원 kd-트리. 축을 번갈아 중앙값으로 나눈다. 질의가 든 잎을 먼저 보고, 분할면이 멀면 반대쪽을 건너뛴다

예제의 두 숫자 사이에 어디서 무너지는지가 이 절의 실험이다. 차원을 2 에서 128 까지 올리며 들여다본 점의 비율을 잰다.

균열

표준 정규 점 십만 개를 차원 2, 4, 8, 16, 32, 128 로 두고, 잎 크기 16 의 kd-트리에서 최근접 탐색이 들여다본 점의 비율과 거리 대비를 쟀다.

실험 12.1 차원에 따른 들여다본 점의 비율과 거리 대비

차원	들여다본 점 퍼센트	정확한 답 퍼센트	최근접 거리 나누기 평균 거리
2	0.21	100	0.0041
4	0.57	100	0.0498
8	7.04	100	0.20
16	86.3	100	0.38
32	100	100	0.55
128	100	100	0.77

측정 조건. 질의 100 개. 정확한 답 열은 전부 비교와 같은 답을 낸 비율이고, 트리는 정확하므로 언제나 100 이다. 거리 대비는 최근접 거리를 모든 점까지의 평균 거리로 나눈 것.

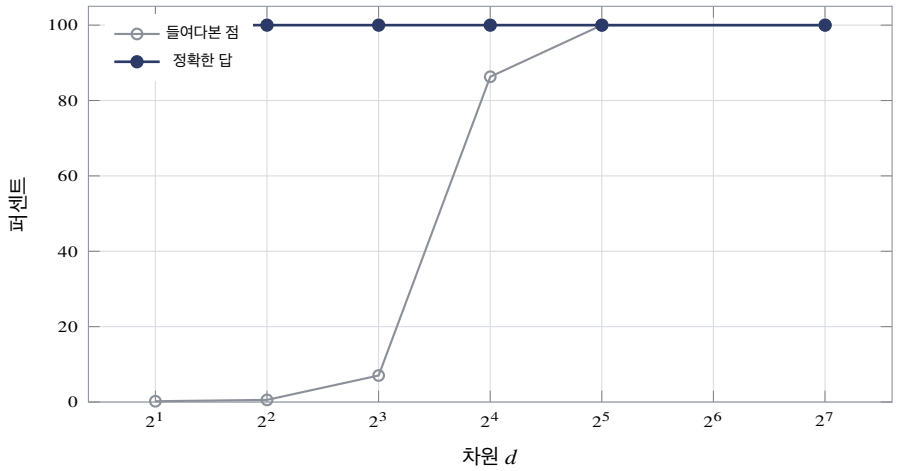


그림 12.2. 차원에 따른 들여다본 점의 비율. 8차원을 지나며 급히 오르고 16차원에서 전부 비교와 같아진다

2차원에서 0.2 퍼센트, 8차원에서 7 퍼센트, 16차원에서 86 퍼센트, 32차원부터 100 퍼센트다. 8 과 16 사이에서 무너진다. Weber 등의 10 남짓과 같다. 거리 대비는 0.004 에서 0.77 로 오른다. 명제의 $1/\sqrt{d}$ 가 그대로다.

관찰

16차원에서 86 퍼센트를 보는 트리는 전부 비교보다 느리다. 전부 비교는 배열 한 번 순차 읽기이고, 트리는 잎 수천 개를 흩어진 순서로 읽는다. 같은 점 수를 봐도 이동량이 다르다.

이 관찰이 벡터 검색이 트리를 버린 이유다. 임베딩은 수백 차원이고, 거기서 트리는 전부 비교에 트리의 값을 더한 것이다. 남은 길은 둘이다. 정확한 답을 포기하고 가까운 답을 확률로 찾거나, 전부 비교를 더 싸게 하거나. 앞의 것이 이 장이고 뒤의 것이 13장이다.

다시 쓰기

이 절의 다시 쓰기는 문제를 바꾸는 것이다. 정확한 최근접 대신 재현율이다.

명제 12.3: 근사 최근접과 재현율

> 질의 q 의 정확한 상위 k 개 집합을 T, 어떤 방법이 낸 상위 k 개를 R 이라 하자. 재현율은 $|R \cap T|/k$ 다. 재현율 1 을 요구하면 고차원에서 어떤 분할 구조도 전부 비교와 같다. 재현율 $1-\epsilon$ 을 허용하면 들여다보는 점의 수를 n 의 작은 비율로 줄이는 구조가 있고, 그것이 다음 두 절의 그래프다.

증명. 첫 문장은 명제 12.2 와 Weber 등의 결과다. 둘째는 존재 명제이고, 실험 12.3 이 그 증거다. ■

이 명제가 0.3 의 재현율을 이 장의 잣대로 다시 세운다. 그리고 7.3 의 S 곡선이 그 잣대의 첫 예였다. 0.2 의 셋째 가정 (정확한 답) 이 7장에 이어 여기서 둘째로 놓인다.

알고리즘 12.1 kd-트리 최근접 탐색 `dsai.kdtree.KDTree.nearest`

> **입력 :** 트리, 질의 q
> 최선 ← 없음 ; 스택 ← [뿌리]
> **조건** 스택이 비지 않음 **동안**
> 노드 ← 꺼냄 ; 잎이면 점 전부를 견줘 최선을 갱신하고 계속
> 가까운 쪽과 먼 쪽을 정한다 ; $|q_a - s|^2$ < 최선 거리이면 먼 쪽을 넣는다 ; 가까운 쪽을 넣는다
> **끝** 조건

구현

트리는 배열 위의 노드 목록이고, 탐색은 5.1 의 명시적 스택이다.

지금 보는 코드 가까운 쪽을 먼저 내려가고, 돌아오며 분할면이 최선보다 가까울 때만 먼 쪽을 본다.

구현 가지치기 최근접 탐색 `samples/dsai/kdtree.py:42`

```
    def nearest(self, q: np.ndarray) -> tuple[int, int]:
        """알고리즘 12.1. 가지치기 최근접 탐색. (가장 가까운 점의 번호,
        들여다본 점 수)."""
        best, best_d2, seen = -1, np.inf, 0
        stack = [self.root]
        while stack:
            node = stack.pop()
            axis, a, b, c = self.nodes[node]
            if axis == -1:
                ids = self.idx[a:b]
                d2 = ((self.pts[ids] - q) ** 2).sum(axis=1)
                seen += len(ids)
                j = int(np.argmin(d2))
                if d2[j] < best_d2:
                    best, best_d2 = int(ids[j]), float(d2[j])
                continue
            diff = q[axis] - a
            near, far = (b, c) if diff < 0 else (c, b)
            if diff * diff < best_d2:              # 먼 쪽이 아직 이길 수 있을 때만 본다
                stack.append(far)
            stack.append(near)
        return best, seen
```

읽어 보면 `if diff * diff < best_d2` 한 줄이 가지치기다. 고차원에서 이 조건이 거의 언제나 참이 되어 `far` 가 매번 스택에 들어간다. 실험의 100 퍼센트가 이 한 줄이다.

분석

kd-트리는 이제 안 맞는다. 정의와 불변식은 그대로이고 낮은 차원에서는 여전히 좋지만, 이 책의 작업량인 수백 차원 벡터에서는 전부 비교보다 느리다. Bentley 의 논문은 차원 k 를 키에 두었고 [확립] [48], 그 k 가 2 나 3 이던 시절의 구조다. 무너진 것은 구조가 아니라 차원이라는 가정이다.

이 책에서 이제 안 맞는다는 판정을 받는 첫 구조가 여기다. 조정으로 살릴 수 없다. 축을 잘 고르든 잎을 키우든 명제 12.2 의 비율은 안 바뀐다. 그래서 다음 두 절은 트리를 고치지 않고 다른 구조를 세운다.

현장에서

> Weber 등의 1998 년 논문은 분할과 군집 기반의 유사도 검색 전부가 고차원에서 선형이 됨을 보이고, 순차 훑기를 빠르게 하는 근사 표현을 대신 제안했다 [확립] [47]. 그 방향이 13 장의 양자화이고, 이 장의 그래프는 다른 방향이다.

주의

> 실험 12.1 의 점은 독립인 정규 분포라 차원이 곧 본질적 차원이다. 실제 임베딩은 수백 차원이어도 본질적 차원이 낮은 경우가 많고, 그러면 무너지는 자리가 더 뒤다. 그래도 수십 차원을 넘으면 같다.

트리가 무너진 자리에서 두 구조가 나온다. 하나는 확률로 층을 올리는 리스트이고, 다른 하나는 그 리스트를 그래프로 넓힌 것이다. 다음 절이 리스트다.

되짚기

> kd-트리를 정의하고 가지치기가 차원과 함께 무너짐을 명제로 세웠다. 8차원과 16차원 사이에서 들여다보는 점이 7 퍼센트에서 86 퍼센트로 뛰었고, 거리 대비가 그 이유였다. 정확한 답 대신 재현율이 잣대가 된다. 다음 절은 스킵 리스트다.

12.2 스킵 리스트. 계층으로 건너뛰기

4.2 의 연결 리스트는 순서를 지키지만 찾기가 선형이다. 8.1 의 트리는 로그이지만 균형을 회전으로 지켜야 한다. 1990 년에 Pugh 가 그 사이에 구조 하나를 두었다 [확립] [49]. 정렬된 리스트 위에 지름길 층을 확률로 올린다. 위 층은 노드가 드물어 멀리 건너뛰고, 아래로 내려오며 좁힌다. 균형을 잡는 회전이 없고, 노드의 높이가 동전 던지기다.

이 절이 이 구조를 세우는 이유는 다음 절에 있다. 근접 그래프 위에 층을 올리는 것이 스킵 리스트와 같은 생각이고, 층의 높이를 정하는 확률도 같다.

학습 목표

스킵 리스트를 층별 정렬 리스트로 정의하고, 노드 높이가 기하 분포일 때 탐색이 지나는 노드 수의 기댓값을 적을 수 있다. 층의 확률 p 가 메모리와 탐색 걸음을 어떻게 맞바꾸는지 실측으로 말하고, 같은 생각이 그래프에 어떻게 옮겨 가는지 설명할 수 있다.

이 말부터

층	정렬된 리스트 하나. 층 0 이 노드 전부이고 층이 오를수록 노드가 p 배씩 준다
높이	노드가 든 층의 수. 동전이 앞면인 동안 하나씩 올라 기하 분포다
지름길	위 층의 간선. 아래 층의 노드 여럿을 한 번에 건너뛴다

디딤돌

정의 12.4: 스킵 리스트

키마다 높이 $h \geq 1$ 을 확률 $\Pr[h \geq \ell] = p^{\ell-1}$ 로 뽑는다. 층 ℓ 은 높이가 ℓ 이상인 키들의 정렬 리스트다. 머리 노드는 모든 층에 있다.

불변식 층 ℓ 의 노드 집합은 층 $\ell - 1$ 의 부분 집합이고, 각 층은 정렬되어 있다. 탐색은 맨 위 층의 머리에서 시작해, 다음 노드의 키가 찾는 키보다 작으면 오른쪽으로 가고 아니면 한 층 내려간다. 층 0 에서 멈춘 자리의 다음이 답이다.

불변식이 탐색의 정당성이다. 층 ℓ 에서 건너뛴 구간의 모든 키는 그 층의 두 노드 사이에 있으므로 층 $\ell - 1$ 에서 그 사이만 보면 된다. 얼마나 건너뛰는지는 p 가 정한다.

정리 12.5: 스킵 리스트의 탐색 비용

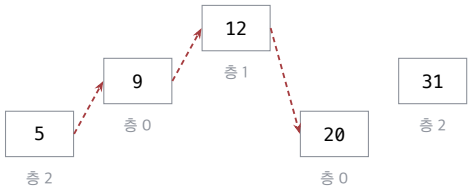
키 n 개, 확률 p 의 스킵 리스트에서 탐색이 지나는 노드 수의 기댓값은 $\frac{1}{p}\log_{1/p} n + O(1)$ 이고, 층 수의 기댓값은 $\log_{1/p} n$ 이다. 노드당 간선 수의 기댓값은 $1/(1-p)$ 이다. $p = 1/2$ 이면 걸음 $2\log_2 n$, 간선 2 개이고, $p = 1/4$ 이면 걸음 $2\log_4 n \cdot 2 = 2\log_2 n$, 간선 4/3 개다.

증명. 탐색 경로를 거꾸로 따라간다. 층 0 의 답에서 위로 올라가며, 노드마다 확률 p 로 한 층 올라가고 $1-p$ 로 왼쪽으로 간다. 층 하나를 올라가는 데 기대 $1/p$ 걸음이고, 층이 $\log_{1/p} n$ 개다. 간선 수는 높이의 기댓값 $\sum_{\ell \geq 1} p^{\ell-1} = 1/(1-p)$ 다. Pugh 의 논문이 이 분석을 상수까지 준다 [확립] [49]. ■

정리에서 읽을 것은 p 의 맞바꿈이다. p 를 낮추면 간선이 줄어 메모리가 작고 층이 적지만, 층 하나에서 오른쪽으로 더 많이 간다. 걸음 수의 곱은 $p = 1/e$ 근처에서 최소이고, $1/4$ 과 $1/2$ 는 둘 다 그 근처다.

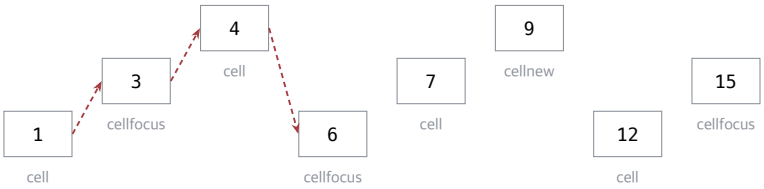
예제 12.2: 키 십만 개

> $p = 1/2$ 이면 층이 17 개 안팎, 노드당 간선 2 개, 걸음 $2 \times 16.6 = 33$ 이 상한이다. $p = 1/4$ 이면 층이 8 개 안팎, 간선 1.3 개, 상한은 같다. 8.1 의 트립은 노드당 링크 2 개에 평균 깊이 $2 \ln n = 23$ 이었다.



찾기 20. 5 에서 31 을 넘고 12 로 내려온다

그림 12.3.. 스킵 리스트의 노드와 층. 층이 높은 노드가 드물고, 위 층에서 멀리 건너뛴 뒤 내려온다. 12.3 의 계층 그래프가 이 모양이다



층 0 의 여덟 노드. 굵은 것이 층 1 에도 있고, 9 는 층 2 까지 있다

그림 12.4.. 키 여덟 개의 스킵 리스트. 층 0 은 전부, 층 1 은 넷, 층 2 는 하나. 탐색은 위에서 오른쪽으로 가다 넘치면 내려온다

예제의 상한 33 이 실제 걸음에서 얼마인지, 그리고 트립과 얼마나 다른지 잰다.

균열

무작위 키 십만 개를 $p = 1/4$ 과 $1/2$ 의 스킵 리스트와 8.1 의 트립에 넣고, 탐색이 지나는 노드 수와 메모리를 쟀다.

실험 12.2 스킵 리스트와 트립. 지나는 노드 수와 메모리

구조	층 또는 높이	평균 지나온 노드	최대	MB	기대 상한
스킵 리스트 p=0.25	9	22.7	64.0	1.86	33.2
스킵 리스트 p=0.5	16	15.4	34.0	2.40	33.2
트립 (8장)	39	21.2	35.0	3.20	23.0

측정 조건. 탐색 2,000 개는 들어 있는 키. 스킵 리스트의 메모리는 키 배열과 층별 링크이고, 트립은 키와 자식 둘과 우선순위. 기대 상한은 정리의 첫 항.

$p = 1/2$ 에서 평균 15 걸음, 최대 34 걸음이다. 정리의 상한 33 의 절반이 평균이다. $p = 1/4$ 은 평균 23 걸음에 메모리가 4 분의 3 이다. 트립은 평균 21 걸음, 높이 39 다. 스킵 리스트 $p = 1/2$ 가 트립보다 걸음이 적고 메모리도 작다. 회전 없이 그렇다.

관찰

스킵 리스트의 걸음 하나는 4.1 의 의존 읽기다. 다음 노드의 주소를 읽어야 다음으로 간다. 트립도 같다. 둘 다 라인 스무 개 안팎이고, 8.4 의 배열 탐색과 견주면 진다. 스킵 리스트가 남는 자리는 삽입이 계속 오는 곳이다. 삽입이 없으면 8.4 의 배열이다.

이 관찰이 8장과 같은 결론을 낸다. 정적이면 배열이고 동적이면 확률 구조다. 스킵 리스트가 여기 있는 이유는 그것이 그래프로 넓어지기 때문이다.

다시 쓰기

이 절의 다시 쓰기는 스킵 리스트의 두 생각을 리스트 밖으로 꺼내는 것이다.

명제 12.6: 두 생각의 일반화

스킵 리스트는 두 성질로 로그를 얻는다. 층이 오를수록 노드가 기하급수로 드물어 위 층의 간선이 길고, 어느 층에서든 다음 노드로 가면 답에 가까워진다. 정렬된 키 대신 거리가 있는 점을 두고, 다음 노드 대신 가까운 이웃을 두면 두 성질이 그대로 옮겨 간다. 위 층은 드문 점들의 근접 그래프이고, 아래 층으로 내려올수록 촘촘해진다. 그것이 다음 절의 HNSW 다.

증명. 첫 성질은 높이의 기하 분포에서 나오고 점의 종류와 무관하다. 둘째 성질은 1차원에서는 정렬이 보장하지만 고차원에서는 그래프가 탐색 가능해야 한다. 그것이 다음 절의 조건이다. ■

이 명제가 스킵 리스트를 12장에 둔 이유다. 8장의 트리가 무너진 자리에서 리스트의 층이 살아남는다. 층은 순서가 아니라 거리 위에 놓을 수 있다.

알고리즘 12.2 스킵 리스트 탐색 `dsai.skiplist.SkipList.search`

입력 : 층 L 개의 리스트, 키 k
1: $v \leftarrow$ 머리
2: **반복** $\ell = L-1, \dots, 0$ **동안**
3: **조건** $\text{next}_\ell(v)$ 의 키 $< k$ **동안** $v \leftarrow \text{next}_\ell(v)$
4: **끝 조건**
5: **끝 반복**
6: **반환** $\text{next}_0(v)$ 의 키가 k 인가

구현

노드는 4.2 의 아레나다. 키 배열 하나와 층마다 다음 배열 하나. 높이는 동전 던지기다.

지금 보는 코드 위 층에서 오른쪽으로 가다 다음 키가 크면 한 층 내려온다. 층 0 에서 멈춘다.

구현 탐색 `samples/dsai/skiplist.py:38`

```
38      def search(self, k: int) -> tuple[bool, int]:
39          """알고리즘 12.2. 위 층에서 오른쪽으로 가다 넘치면 내려간다. (찾았는지,
40          지나온 노드 수)."""
41          """
42          cur, steps = self.head, 0
43          for l in range(self.top, -1, -1):
44              while self.nxt[l, cur] != NIL and self.key[self.nxt[l, cur]] < k:
45                  cur = self.nxt[l, cur]
46                  steps += 1
47          cand = self.nxt[0, cur]
48          return (cand != NIL and self.key[cand] == k), steps
```

읽어 보면 두 겹 루프가 정리의 두 항이다. 바깥이 층 $\log_{1/p} n$ 개, 안쪽이 층마다 오른쪽 걸음 기대 $1/p$ 개. `steps` 가 실험의 지나온 노드 수다.

구현 삽입. 높이를 뽑고 층마다 잇는다 `samples/dsai/skiplist.py:50`

```
50      def insert(self, k: int) -> None:
51          update = [self.head] * self.nxt.shape[0]
52          cur = self.head
53          for l in range(self.top, -1, -1):
54              while self.nxt[l, cur] != NIL and self.key[self.nxt[l, cur]] < k:
55                  cur = self.nxt[l, cur]
56              update[l] = cur
57          h = self._random_level()
58          if h - 1 > self.top:
59              self.top = h - 1
60          i = self.n
61          self.n += 1
62          self.key[i] = k
63          self.level[i] = h
64          for l in range(h):
65              self.nxt[l, i] = self.nxt[l, update[l]]
66              self.nxt[l, update[l]] = i
```

읽어 보면 `update` 가 층마다 새 노드의 왼쪽이 될 노드다. 탐색과 같은 경로이고, 그 뒤 높이 h 만큼 링크를 잇는다. 회전이 없다.

분석

스킵 리스트는 그대로 유효하다. 1990년의 정의와 정리가 그대로이고, 실험이 상한 안에 들었다. 바뀐 것은 이 구조가 쓰이는 자리다. 정렬 맵으로는 8.4 의 배열과 8장의 B-트리에 밀렸고, 남은 자리는 삽입이 잦은 곳과 잠금 없는 동시성이다. 그리고 그 생각이 그래프로 옮겨 간 자리가 이 장의 나머지다.

이 구조가 이 책의 작업량에 서는 자리는 벡터 인덱스의 층이다. 다음 절의 HNSW 가 이 절의 정리를 그래프 위에서 되풀이한다. 층의 확률, 위에서 아래로의 탐색, 노드당 링크 수까지 같다.

현장에서

> Pugh 의 논문은 스킵 리스트가 균형 트리보다 구현이 쉽고 빠르다는 것을 실측으로 보였다 [확립] [49]. 지금 몇몇 키 값 저장소의 메모리 표와 동시성 정렬 맵이 이 구조를 쓴다 [확립]. 잠금 없이 삽입할 수 있다는 것이 회전이 없는 것의 값이다.

주의

> 실험 12.2 의 걸음은 노드 수이고 시간이 아니다. 스킵 리스트의 위 층 노드들은 아레나에 흩어져 있어 걸음마다 라인 하나이고, 트립도 같다. 같은 걸음 수에 같은 이동량이다.

층을 거리 위에 놓는다. 다음 절의 그래프는 정렬이 없는 대신 이웃이 있고, 이웃을 따라가면 답에 가까워진다는 조건 위에서 스킵 리스트의 정리를 되찾는다.

되짚기

> 스킵 리스트를 정의하고 탐색 비용 정리를 세웠다. $p = 1/2$ 에서 평균 15 걸음으로 트립보다 적고 메모리도 작았다. 층의 생각이 거리 위로 옮겨 간다. 다음 절은 탐색 가능한 작은 세상 그래프다.

12.3 탐색 가능한 작은 세상 그래프. NSW 와 HNSW

12.1 에서 트리가 무너졌고 12.2 에서 층의 생각이 남았다. 이 절은 그 생각을 거리 위에 놓는다. 점마다 가까운 점 몇 개를 간선으로 잇고, 질의를 받으면 지금 노드의 이웃 중 질의에 더 가까운 곳으로 옮겨 간다. 이웃이 언제나 더 가까운 곳으로 이어져 있으면 몇 걸음에 도착한다. 그 조건이 탐색 가능하다는 것이고, 삽입 순서와 이웃 고르기가 그 조건을 만든다. Malkov 와 Yashunin 이 2020 년에 여기에 층을 올려 지금 가장 널리 쓰이는 벡터 인덱스를 세웠다 [확립] [50].

이 절은 그래프를 삽입으로 만들고, 탐욕 탐색을 알고리즘으로 적고, 층을 올린 뒤, 0.3 의 kNN 을 재현율과 거리 계산 수로 잰다.

학습 목표

근접 그래프 위의 탐욕 탐색을 정의하고, 후보 집합의 크기 ef 가 재현율과 거리 계산 수를 어떻게 맞바꾸는지 설명할 수 있다. 삽입 순서가 긴 간선을 만드는 이유와 이웃 고르기 휴리스틱이 그 간선을 지키는 이유를 말하고, 층이 진입점의 값을 어떻게 줄이는지 스킵 리스트로 설명할 수 있다.

이 말부터

근접 그래프	점이 노드이고 가까운 점 M 개가 이웃인 그래프. 간선이 거리에서 나온다
탐욕 탐색	지금 노드의 이웃 중 질의에 더 가까운 노드로 옮겨 가는 것. 후보 ef 개를 함께 든다
진입점	탐색을 시작하는 노드. 층이 있으면 맨 위 층의 노드 하나다

디딤돌

정의 12.7: 근접 그래프와 탐욕 탐색

점 집합 P 위의 방향 그래프로, 노드 u 의 이웃 $N(u)$ 는 u 에 가까운 점 M 개 안팎이다. 질의 q 와 진입점 s 에 대한 탐욕 탐색은 후보 힙 (가까운 순) 과 결과 힙 (먼 순, 크기 ef) 을 둔다. 후보에서 가장 가까운 노드를 꺼내 이웃을 전부 견주고, 결과의 최악보다 가까운 이웃을 두 힙에 넣는다. 꺼낸 노드가 결과의 최악보다 멀면 멈춘다.

불변식 결과 힙은 지금까지 견준 점 중 가장 가까운 ef 개다. 멈추는 순간 후보에 결과보다 가까운 것이 없다.

불변식의 둘째 문장이 탐색이 끝나는 조건이고, 정확성을 보장하지는 않는다. 후보에 없는 노드 중에 더 가까운 것이 있을 수 있다. 그것이 재현율이 1 이 아닌 이유이고, ef 를 키우면 그 노드까지 닿을 확률이 오른다. 탐색이 몇 걸음에 끝나느냐는 그래프에 달렸다.

명제 12.8: 탐색 가능성과 긴 간선

> 모든 노드 u 와 질의 q 에 대해 $N(u)$ 에 u 보다 q 에 가까운 노드가 있으면, 탐욕 탐색은 국소 최소 없이 진행한다. 가까운 이웃만으로는 이 조건이 안 지켜진다. 먼 곳으로 가는 간선이 있어야 한다. 점을 무작위 순서로 넣고 넣을 때마다 그때까지의 그래프에서 찾은 가까운 점과 이으면, 먼저 들어온 점은 점이 드물 때 이어졌으므로 긴 간선을 갖는다. 그 간선이 지름길이고, 걸음 수가 $O(\log n)$ 에 가까워진다.

증명. 첫 문장은 정의에서 나온다. 매 걸음 더 가까운 이웃이 있으면 거리가 단조 감소하고 유한하므로 끝난다. 둘째는 삽입 시점의 밀도 논증이다. i 번째로 들어온 점의 가장 가까운 점까지의 거리는 i 가 작을수록 크고, 그 간선이 남아 있으면 초기 점들이 먼 영역을 잇는다. Malkov 등이 이 성질을 작은 세상이라 불렀고 [확립] [50], 걸음 수의 로그는 실험으로 보인 것이다. ■

명제의 둘째 문장에 조건이 하나 숨어 있다. 긴 간선이 남아 있어야 한다. 이웃 목록이 넘쳐 잘라 낼 때 가장 먼 것부터 자르면 긴 간선이 먼저 사라진다. 그래서 이웃 고르기에 규칙이 하나 더 있다.

정의 12.9: 이웃 고르기 휴리스틱

> 노드 v 의 후보를 가까운 순으로 보며, 후보 w 가 이미 고른 이웃 u 중 하나에 대해 $d(u,w) < d(v,w)$ 이면 버린다. w 가 u 너머에 있어 u 를 거쳐 갈 수 있기 때문이다. 남은 것이 서로 다른 방향의 이웃이다.

이 규칙이 같은 방향의 가까운 이웃 여럿 대신 방향이 다른 이웃을 남기고, 그 안에 긴 간선이 든다. 12.2 의 스킵 리스트가 층으로 하던 일을 그래프에서는 이 규칙이 한다. 그리고 층을 그래프에도 올릴 수 있다.

정의 12.10: 계층 근접 그래프

> 점마다 높이를 $\Pr[h \ge \ell] = p^{\ell}$ 로 뽑고, 층 ℓ 은 높이가 ℓ 이상인 점들의 근접 그래프다. 탐색은 맨 위 층의 진입점에서 ef 1 로 가장 가까운 점을 찾아 한 층 내려가는 것을 층 0 까지 되풀이하고, 층 0 에서 ef 로 넓힌다.
>
> **불변식** 층 ℓ 의 노드는 층 $\ell - 1$ 에도 있다. 12.2 의 불변식 그대로다.

예제 12.3: 점 2만 개, 이웃 16

> $p = 1/4$ 이면 층이 여섯 개 안팎이고 위 층은 점 수십 개다. 위 층에서 몇 걸음에 질의 근처로 가고, 층 0 에서 ef 개의 후보로 마무리한다. 전부 비교는 거리 계산 2만 번이고, 그래프는 넓힌 노드 수 곱하기 16 이다. 노드 열다섯 개를 넓히면 240 번이다.

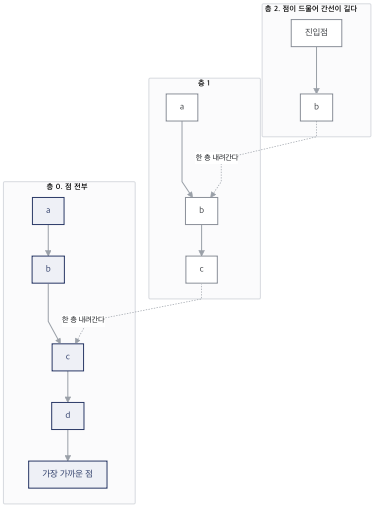


그림 12.5.. 계층 근접 그래프의 탐색. 위 층에서 멀리 건너뛰고 한 층씩 내려와 층 0 에서 후보를 넓힌다

예제의 240 번이 실제로 어느 재현율에서 나오는지, 그리고 층이 무엇을 바꾸는지 잰다.

균열

군집 200 개의 64 차원 점 2 만 개에 이웃 16 의 그래프를 삽입으로 만들고, 한 층짜리 NSW 와 여섯 층의 HNSW 에서 ef 를 바꾸며 재현율@10 과 거리 계산 수를 쟀다.

실험 12.3 ef 에 따른 재현율과 거리 계산 수

구조	ef	재현율 at 10 퍼센트	거리 계산 수	질의당 ms
전부 비교	0	100	20000	4.12
NSW (한 층)	10	84.5	164	2.01
HNSW (6 층)	10	93.5	195	1.65
NSW (한 층)	32	97.7	213	2.68
HNSW (6 층)	32	98.8	236	2.65
NSW (한 층)	64	99.7	244	2.73
HNSW (6 층)	64	99.5	267	3.28
NSW (한 층)	128	99.8	408	5.49
HNSW (6 층)	128	99.6	437	4.51

측정 조건. 질의 200 개는 같은 군집 분포에서 뽑았다. 만들기는 NSW 71 초, HNSW 97 초. 시간은 파이썬 루프의 값이고 전부 비교는 NumPy 라 견주지 않는다.

ef 10 에서 NSW 는 84 퍼센트, HNSW 는 93 퍼센트다. 거리 계산은 164 번과 195 번으로 2 만 개의 1 퍼센트다. ef 64 에서 둘 다 99.6 퍼센트를 넘고 거리 계산이 250 번 남짓이다. 전부 비교의 80 분의 1 로 천 번에 넷을 놓친다.

관찰

> 층의 효과는 ef 가 작을 때 보인다. ef 10 에서 9 퍼센트 차이가 ef 32 에서 1 퍼센트로 준다. 층은 진입점을 질의 근처로 옮기는 일을 하고, ef 가 크면 층 없이도 후보가 거기까지 간다. 거리 계산 수는 층이 있는 쪽이 30 번 남짓 더 든다. 위 층을 내려오는 값이다.

이 관찰이 ef 와 층의 역할을 가른다. ef 는 층 0 에서 얼마나 넓게 보느냐이고 재현율을 정한다. 층은 어디서 시작하느냐이고 ef 가 작을 때의 재현율을 정한다.

다시 쓰기

이 절의 다시 쓰기는 거리 계산 수를 이동량으로 적는 것이다. 그래프 탐색의 값은 거리 계산이 아니라 그 벡터를 읽어 오는 것이다.

명제 12.11: 그래프 탐색의 이동량

> 넓힌 노드 t 개, 이웃 M 개, 벡터 d 차원 s 바이트의 탐색에서 이동량은 이웃 목록 t 개 (tM 정수) 와 벡터 tM 개 ($tMds$ 바이트) 다. 벡터 읽기가 전부이고, 벡터마다 4.1 의 의존 읽기다. 이웃 번호를 읽어야 벡터의 주소를 안다. 전부 비교는 벡터 n 개를 순차로 읽는다. $tM \ll n$ 이면 그래프가 이기고, 벡터가 메모리에 없으면 tM 번의 무작위 접근이 값이다.

증명. 넓힌 노드마다 이웃 목록 하나와 이웃 벡터 M 개를 읽는다. 이웃 번호는 목록에서 오므로 벡터 읽기는 번호를 읽은 뒤이고 주소를 미리 모른다. 전부 비교는 1.3 의 순차 접근이다. ■

이 명제가 다음 절의 문제다. 실험의 250 번은 메모리에서 250 라인이지만, 벡터가 디스크에 있으면 250 번의 디스크 접근이다. 그래프가 메모리에 안 들 때 어떻게 하느냐가 다음 절이다. 그리고 $tMds$ 의 s 를 줄이는 것이 13장이다. 벡터를 바이트 몇 개로 접으면 이동량이 그만큼 준다.

알고리즘 12.3 탐욕 탐색 `dsai.nsw.greedy_search`

> **입력:** 그래프, 질의 q, 진입점 s, ef
> 1: 후보 ← $\{s\}$ (가까운 순 힙); 결과 ← $\{s\}$ (먼 순 힙, 크기 ef)
> 2: **조건** 후보가 비지 않음 **동안**
> 3: u ← 후보에서 가장 가까운 것; 결과의 최악보다 멀면 멈춘다
> 4: **모든** $v \in N(u)$, 아직 안 본 것 **동안** $d(v, q)$ 를 재고, 결과의 최악보다 가까우면 두 힙에 넣고 결과가 넘치면 최악을 뺀다
> 5: **끝 반복**
> 6: **끝 조건**
> 7: **반환** 결과

삽입은 이 탐색을 새 점에 대해 한 번 부르고, 찾은 후보에서 이웃을 고르는 것이다. 만들기 전체가 삽입 n 번이므로 인덱스를 세우는 값이 탐색 n 번이다.

알고리즘 12.4 삽입`dsai.nsw.NSW.insert`

입력 : 그래프, 새 점 i, M, ef 만들기
1: 후보 ← 탐욕 탐색$(p_i$, 진입점, ef 만들기$)$
2: $N(i)$ ← 휴리스틱으로 고른 M 개 ; 각 $v \in N(i)$ 에 i 를 더하고, $|N(v)|$ 가 상한을 넘으면 휴리스틱으로 다시 고른다

구현

탐색은 10.1 의 힙 둘이다. 삽입은 탐색 하나와 이웃 고르기 하나다.

지금 보는 코드 가장 가까운 후보를 꺼내 이웃을 전부 견주고, 결과의 최악보다 가까운 것만 넣는다.

구현 탐욕 탐색 `samples/dsai/nsw.py:21`

```python
def greedy_search(points: np.ndarray, nbrs: list[list[int]], q: np.ndarray, start: int,
                  ef: int) -> tuple[list[tuple[float, int]], int]:
    """알고리즘 12.3. 후보 ef 개를 유지하는 탐욕 탐색. (가까운 순, 계산 수)."""
    d0 = d2(points, start, q)
    visited = {start}
    cand = [(d0, start)]                      # 최소 힙. 아직 안 넓힌 후보
    best = [(-d0, start)]                     # 최대 힙. 지금까지의 상위 ef
    n_dist = 1
    while cand:
        d, u = heapq.heappop(cand)
        if d > -best[0][0] and len(best) >= ef:
            break                             # 가장 가까운 후보가 결과의 최악보다 멀면 끝
        for v in nbrs[u]:
            if v in visited:
                continue
            visited.add(v)
            dv = d2(points, v, q)
            n_dist += 1
            if len(best) < ef or dv < -best[0][0]:
                heapq.heappush(cand, (dv, v))
                heapq.heappush(best, (-dv, v))
                if len(best) > ef:
                    heapq.heappop(best)
    return sorted((-d, v) for d, v in best), n_dist
```

읽어 보면 `cand` 가 가까운 순 힙이고 `best` 가 부호를 바꾼 먼 순 힙이다. `if d > -best[0][0] and len(best) >= ef: break` 가 불변식의 멈춤 조건이다. `n_dist` 가 실험의 거리 계산 수다.

구현 이웃 고르기 `samples/dsai/nsw.py:62`

```
    def select(self, v: int, cand: list[int], m: int) -> list[int]:
        """이웃 고르기. 가까운 순으로 보되, 이미 고른 이웃 하나가 후보보다 v 에 가깝고
        후보에도 더 가까우면 (후보가 그 이웃 너머에 있으면) 버린다."""
        pv = self.points[v]
        cand = sorted(cand, key=lambda w: d2(self.points, w, pv))
        kept: list[int] = []
        for w in cand:
            dvw = d2(self.points, w, pv)
            if all(d2(self.points, u, self.points[w]) >= dvw for u in kept):
                kept.append(w)
                if len(kept) >= m:
                    break
        return kept
```

읽어 보면 조건 `d2(u, w) >= dvw` 가 정의 12.9 이다. 후보 w 가 이미 고른 u 보다 v 에 가까울 때만 남긴다. 삽입은 같은 파일의 `insert` 이고 탐색 하나에 이 고르기를 두 번 부른다. 이 조건을 빼면 실험의 재현율이 절반 아래로 떨어지고, 그것이 연습문제다.

분석

근접 그래프 탐색은 교과서에 없던 구조다. 8장의 트리가 무너진 자리에 7장의 확률적 답과 9장의 스킵 리스트의 층을 합친 것이다. 그래서 판정은 kd-트리에 대한 것이다. 이제 안 맞고, 그 자리를 이 그래프가 대신한다. 정확한 답을 재현율로 바꾼 값이다.

이 구조가 서는 자리는 벡터 검색 전부다. 임베딩 수십억 개에서 가까운 것을 밀리초에 찾는 일이고, 검색 증강 생성과 추천과 중복 제거가 그 위에 있다. 7.3 의 지역 민감 해싱이 같은 문제의 다른 답이었고, 그래프가 대부분의 자리에서 그것을 대신했다. 이유는 명제 12.11 다. 그래프는 가까운 것만 읽는다.

현장에서

> Malkov 와 Yashunin 의 논문은 층의 확률, 이웃 고르기 휴리스틱, ef 의 역할을 실험으로 정리했고 [확립] [50], 대부분의 벡터 데이터베이스가 이 구조 또는 그 변형을 기본 인덱스로 둔다 [확립]. 만들기가 삽입 n 번이라 점 하나가 탐색 한 번의 값이고, 그것이 이 구조의 가장 큰 비용이다.

주의

> 실험 12.3 의 점은 군집이 뚜렷한 합성 데이터다. 군집이 없는 고르게 흩어진 점에서는 같은 ef 에서 재현율이 낮고, 실제 임베딩은 그 사이 어딘가다. 재현율 99 퍼센트는 이 데이터에서 잰 값이다.

그래프가 메모리에 있으면 여기까지다. 벡터 십억 개는 메모리에 안 들고, 그러면 명제 12.11 의 무작위 접근이 디스크로 간다. 다음 절이 그 자리이고, 거기서는 거리 계산 수 대신 페이지 접근 수를 센다.

되짚기

> 근접 그래프와 탐욕 탐색을 정의하고, 삽입 순서와 이웃 고르기 휴리스틱이 탐색 가능성을 만드는 이유를 세웠다. 점 2만 개에서 거리 계산 250 번으로 재현율 99.6 퍼센트였다. 탐색의 값은 벡터 읽기이고 의존 읽기다. 다음 절은 그 읽기가 디스크로 갈 때다.

12.4 디스크 위의 그래프. 한 번의 접근으로

12.3 의 탐색은 넓힌 노드마다 이웃 목록 하나와 벡터 M 개를 읽었다. 메모리에서는 라인 몇 개이고, 디스크에서는 접근 몇 번이다. 벡터 십억 개에 128차원 float32 면 512 GB 다. 메모리에 안 든다. 그래프까지 더하면 더 안 든다. 그래서 그래프를 디스크에 두고, 탐색이 디스크를 몇 번 읽느냐가 값이 된다.

이 절은 8.2 의 B-트리가 했던 일을 그래프에 한다. 접근의 단위를 저장의 단위에 맞춘다. 노드 하나의 벡터와 이웃 목록을 페이지 하나에 같이 두면 노드를 넓히는 데 접근 한 번이다. 그리고 메모리에 거친 벡터를 두어 디스크 접근을 마지막 후보에서만 한다. Subramanya 등이 2019년에 그 둘로 벡터 십억 개를 SSD 하나에서 밀리초에 찾았다 [확립] [51].

학습 목표

디스크 위의 그래프 탐색에서 접근 횟수가 넓힌 노드 수와 페이지 배치로 어떻게 정해지는지 적을 수 있다. 벡터와 이웃을 한 페이지에 두는 것과 거친 벡터를 메모리에 두는 것이 접근 횟수를 각각 얼마나 줄이는지 실측으로 말하고, 그 대가가 무엇인지 설명할 수 있다.

이 말부터

페이지	디스크에서 한 번에 읽는 단위. 4 KB 안팎이고, 한 번의 접근은 한 페이지다
같이 두기	노드의 벡터와 이웃 목록을 같은 페이지에 놓는 것. 따로 두면 접근이 둘이다
거친 벡터	벡터를 몇 비트로 접은 것. 메모리에 두고 거리를 어림하는 데 쓴다. 13장이 그 방법이다

디딤돌

정의 12.12: 페이지 위의 그래프

노드 u 마다 벡터 x_u 와 이웃 목록 $N(u)$ 를 둔다. 따로 두기는 벡터 배열과 이웃 배열이 다른 페이지들에 있는 것이고, 같이 두기는 $(x_u, N(u))$ 가 페이지 하나에 있는 것이다. 탐색은 정의 12.7 의 탐욕 탐색이고, 페이지를 처음 읽을 때 접근 하나를 센다. 한 번 읽은 페이지는 캐시에 있다.

불변식 노드 u 를 넓히는 데 같이 두기는 페이지 하나, 따로 두기는 벡터 페이지와 이웃 페이지 둘이다. 이웃의 벡터를 견주는 것은 이웃마다 페이지 하나다.

불변식이 접근 횟수를 센다. 넓힌 노드 t 개에 이웃 M 개면 이웃 벡터 읽기가 tM 번이고, 그것이 전부다. 노드 자신의 페이지는 그 앞에서 이웃으로 읽혔다.

명제 12.13: 디스크 그래프의 접근 횟수

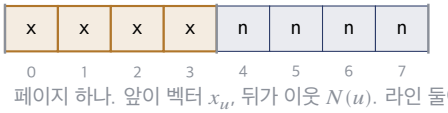
> 넓힌 노드 t 개, 이웃 M 개의 탐색에서 따로 두기는 접근이 t (이웃 목록) $+tM$ (벡터) 이고, 같이 두기는 tM 이다. 견준 이웃 중 겹치는 것을 빼면 둘 다 그만큼 준다. 거친 벡터를 메모리에 두면 거리 어림에 접근이 없고, 접근은 이웃 목록 t 번과 마지막 후보 k' 개의 정확한 벡터 k' 번이다. M 배가 준다.

증명. 따로 두기는 노드마다 목록 페이지를 읽고 이웃마다 벡터 페이지를 읽는다. 같이 두기는 이웃 벡터를 읽을 때 그 노드의 목록이 같이 오므로 목록 읽기가 사라진다. 거친 벡터가 메모리에 있으면 이웃 벡터를 디스크에서 안 읽고, 목록만 읽는다. 목록은 같이 두기에서 벡터와 같은 페이지이므로 넓힌 노드마다 하나다. ■

명제에서 읽을 것은 M 배다. 이웃 열여섯 개를 견주는 데 접근 열여섯 번이 하나가 된다. 대가는 둘이다. 거친 벡터의 메모리와 거친 거리의 오차. 오차는 재현율을 깎고, 그것을 마지막 후보를 정확히 다시 재는 것으로 되돌린다.

예제 12.4: 벡터 십억 개

> 128 차원 float32 벡터에 이웃 64 개면 노드 하나가 512 + 256 = 768 바이트라 4 KB 페이지에 다섯 개가 든다. 넓힌 노드 50 개면 같이 두기는 접근 50 × 64 남짓, 거친 벡터를 두면 50 번 남짓이다. 거친 벡터를 32 바이트로 접으면 십억 개가 32 GB 이고 메모리에 든다.



페이지 하나. 앞이 벡터 x_u, 뒤가 이웃 $N(u)$. 라인 둘이지만 접근은 하나다

그림 12.6.. 페이지 위의 노드. 벡터와 이웃 목록이 한 페이지에 있다. 노드를 넓히면 페이지 하나에 둘이 같이 온다

예제의 두 단계 절약이 실측에서 얼마인지 잰다. 페이지는 노드 하나 단위로 두고 접근 횟수를 센다.

균열

12.3 의 그래프에서 세 배치의 질의당 페이지 접근 수와 재현율을 쟀다. 벡터와 이웃을 따로, 한 페이지에 같이, 그리고 거친 벡터를 메모리에 두고 마지막 후보만 정확히.

실험 12.4 세 배치의 질의당 페이지 접근

배치	질의당 페이지 읽기	재현율 at 10 퍼센트
벡터와 이웃을 따로	358	96.5
한 페이지에 같이	280	96.5
거친 벡터는 메모리에, 정확한 것만 페이지에서	83.7	95.0

측정 조건. 점 2만 개, 이웃 16, ef 64, 질의 100 개. 페이지는 노드 하나 단위. 거친 벡터는 좌표를 8 분의 1 단위로 반올림한 것으로 13장 양자화의 흉내다.

같이 두기가 따로 두기보다 접근이 22 퍼센트 적다. 명제의 t 항이 사라진 것이다. 재현율은 같다. 배치는 답을 안 바꾼다. 거친 벡터를 메모리에 두자 접근이 84 번으로 3 분의 1 이 됐다. 재현율은 96.5 에서 95 로 1.5 퍼센트 내렸다. 거친 거리로 고른 후보에서 진짜 답이 빠진 몫이다.

관찰

접근 84 번 중 대부분이 이웃 목록이다. 정확한 벡터를 읽는 것은 마지막 후보 64 개이고, 그중 이미 읽은 페이지가 겹친다. 명제의 M 배 절약이 실측에서 3 배로 나온 것은 후보를 다시 재는 접근 때문이고, 그 접근이 재현율 1.5 퍼센트를 되사는 값이다.

이 관찰이 디스크 인덱스의 두 손잡이를 보인다. 거친 벡터의 비트 수가 재현율의 손실을 정하고, 다시 재는 후보 수가 그것을 되돌리는 접근 수를 정한다.

다시 쓰기

이 절의 다시 쓰기는 8.2 의 차수 선택을 그래프에 적는 것이다.

명제 12.14: 페이지 예산과 이웃 수

페이지 S 바이트, 벡터 ds 바이트, 이웃 번호 4 바이트에서 같이 두기가 되려면 $ds + 4M \le S$ 다. 이웃 수의 상한은 $M \le (S - ds)/4$ 이고, 벡터를 양자화해 s 를 줄이면 그만큼 이웃을 더 둘 수 있다. 이웃이 많을수록 넓힌 노드 수 t 가 줄어 접근이 줄고, 페이지가 커질수록 한 번의 접근이 비싸진다. 둘의 곱을 최소로 하는 M 이 디스크의 페이지 크기에서 나온다.

증명. 첫 문장은 셈이다. 둘째는 명제 12.8 의 걸음 수가 M 에 대해 줄고, 페이지 읽기 시간이 S 에 대해 늘기 때문이다. 8.2 의 차수 선택과 같은 논증이다. ■

이 명제가 디스크 인덱스가 벡터를 양자화하는 둘째 이유다. 첫째는 메모리에 거친 벡터를 두는 것이고, 둘째는 페이지에 이웃을 더 두는 것이다. 둘 다 13장의 양자화이고, 13장은 그 양자화로 디스크 없이도 벡터 수십억 개를 메모리에 두는 길을 본다.

알고리즘 12.5 거친 벡터로 탐색하고 정확히 다시 재기 `dsai.diskgraph.search_with_pq_filter`

입력 : 페이지 그래프, 메모리의 거친 벡터 $\tilde{x}$, 질의 q, ef
1: 탐욕 탐색을 하되 거리는 $d(\tilde{x}_v, q)$ 로 어림하고, 이웃 목록만 페이지에서 읽는다
2: 결과 ef 개의 정확한 벡터를 페이지에서 읽어 $d(x_v, q)$ 로 다시 정렬한다
3: **반환** 상위 k

구현

페이지 그래프는 접근을 세는 껍데기다. 처음 읽는 페이지만 센다.

지금 보는 코드 벡터를 읽으면 그 노드의 페이지를 세고, 이웃을 읽으면 같이 두기에서는 같은 페이지, 따로 두기에서는 다른 페이지를 센다.

구현 페이지 접근 세기 `samples/dsai/diskgraph.py:19`

```
    def __init__(self, points: np.ndarray, nbrs, colocated: bool = True):
        self.points = points
        self.nbrs = nbrs
        self.colocated = colocated
        self.page_reads = 0
        self.cache: set[int] = set()

    def _touch(self, page: int) -> None:
        if page not in self.cache:
            self.cache.add(page)
            self.page_reads += 1

    def vector(self, i: int) -> np.ndarray:
        self._touch(i)                                  # 벡터 페이지
        return self.points[i]

    def neighbors(self, i: int):
        self._touch(i if self.colocated else -1 - i)    # 같이 두면 같은 페이지
        return self.nbrs[i]
```

읽어 보면 `_touch` 가 접근을 센다. `neighbors` 의 `i if self.colocated else -1 - i` 한 줄이 두 배치의 차이다. 같이 두기는 벡터와 같은 번호, 따로 두기는 다른 번호의 페이지다.

구현 거친 벡터로 탐색하고 다시 재기 `samples/dsai/diskgraph.py:73`

```python
def search_with_pq_filter(g: PagedGraph, coarse: np.ndarray, q: np.ndarray, start: int,
                          ef: int) -> tuple[list[int], int]:
    """거친 벡터 (coarse) 로 거리를 어림하고, 마지막 후보만 페이지에서 정확히 읽는다."""
    g.reset()
    d0 = float(((coarse[start] - q) ** 2).sum())
    visited = {start}
    cand, best = [(d0, start)], [(-d0, start)]
    while cand:
        d, u = heapq.heappop(cand)
        if d > -best[0][0] and len(best) >= ef:
            break
        for v in g.neighbors(u):                          # 이웃 목록은 페이지에서
            v = int(v)
            if v in visited:
                continue
            visited.add(v)
            dv = float(((coarse[v] - q) ** 2).sum())      # 거리는 메모리의 거친 벡터로
            if len(best) < ef or dv < -best[0][0]:
                heapq.heappush(cand, (dv, v))
                heapq.heappush(best, (-dv, v))
                if len(best) > ef:
                    heapq.heappop(best)
    return sorted((v for _, v in best), key=lambda v: d2exact(g, v, q)), g.page_reads
```

읽어 보면 `coarse[v]` 로 거리를 어림하므로 `g.vector` 를 안 부른다. 마지막 `exact` 가 후보 ef 개만 정확한 벡터를 읽는다. 실험의 84 번이 이 두 줄의 차이다.

분석

디스크 위의 그래프는 8.2 의 B-트리를 조정한 것이다. 접근의 단위를 저장의 단위에 맞추는 원칙이 같고, 트리 대신 그래프이며, 키 대신 벡터다. 그리고 거친 벡터를 메모리에 두는 것은 7.1 의 블룸 필터와 같은 자리다. 싼 것으로 먼저 거르고 비싼 것은 남은 것에만 쓴다.

이 구조가 서는 자리는 메모리에 안 드는 벡터 인덱스다. 벡터 십억 개를 기계 한 대의 SSD 에 두고 밀리초에 찾는 일이고, 그 값이 페이지 접근 수십 번이다. Subramanya 등의 보고는 십억 개에서 접근 수십 번으로 재현율 95 퍼센트 이상이었고 [확립] [51], 이 절의 84 번은 그 구조를 파이썬으로 흉내 낸 값이다.

현장에서

> Subramanya 등은 그래프를 만들 때 이웃 고르기에 상수 $\alpha > 1$ 을 두어 긴 간선을 더 남겼고, 그것이 넓힌 노드 수를 줄여 디스크 접근을 줄였다 [확립] [51]. 명제 12.8 의 긴 간선이 디스크에서는 접근 횟수로 바로 값이 된다.

주의

> 실험 12.4 는 접근 횟수를 셀 뿐 디스크를 읽지 않는다. 실제 SSD 는 접근 하나가 백 마이크로초 안팎이고, 84 번이면 8 밀리초다. 여러 접근을 한꺼번에 내면 겹치고, 그것은 4.1 의 독립 읽기다. 이 실험은 그 겹침을 세지 않았다.

그래프는 벡터 십억 개를 디스크에서 찾는 데까지 왔다. 남은 값은 벡터 자체의 크기다. 다음 장은 벡터를 바이트 몇 개로 접고, 그 코드로 거리를 어림하는 인덱스다.

되짚기

> 페이지 위의 그래프에서 접근 횟수가 넓힌 노드 수와 배치로 정해짐을 세웠다. 같이 두기가 22 퍼센트, 거친 벡터가 3 분의 1 로 접근을 줄였고 재현율 1.5 퍼센트를 냈다. 벡터를 접는 것이 다음 장이다.

12장을 닫으며

이 장이 세운 구조는 넷이다. 가지치기가 무너지는 kd-트리, 층을 확률로 올리는 스킵 리스트, 삽입 순서와 이웃 고르기로 탐색 가능해지는 근접 그래프, 그리고 페이지에 벡터와 이웃을 같이 둔 디스크 그래프. 첫째는 이제 안 맞고, 나머지 셋이 그 자리를 잇는다. 셋 다 확률로 답하고, 셋 다 재현율이 잣대다.

네 실험의 숫자를 적어 둔다. kd-트리는 8차원에서 점의 7 퍼센트, 16차원에서 86 퍼센트를 봤고 거리 대비가 0.2 에서 0.4 로 올랐다. 스킵 리스트는 키 십만 개에서 평균 15 걸음으로 트립의 21 걸음보다 적었다. 근접 그래프는 점 2만 개에서 거리 계산 250 번으로 재현율 99.6 퍼센트였고, ef 10 에서는 층이 재현율을 84 에서 93 퍼센트로 올렸다. 디스크 그래프는 같이 두기로 접근 22 퍼센트, 거친 벡터로 3 분의 1 이었고 재현율 1.5 퍼센트를 냈다.

정정 하나를 남긴다. 근접 그래프를 처음에는 이웃 목록이 넘칠 때 가장 먼 이웃을 떼는 규칙으로 만들었다. 재현율이 ef 64 에서 60 퍼센트였다. 가장 먼 이웃이 군집 사이의 다리였고, 그것을 떼자 탐색이 군집을 못 건넜다. 이웃 고르기 휴리스틱으로 바꾸자 같은 ef 에서 99 퍼센트가 됐다. 그래프의 값은 이웃 수가 아니라 이웃의 방향에 있었고, 그것이 원 논문이 휴리스틱을 둔 이유였다.

옛 구조의 판정

> kd-트리는 이제 안 맞는다. 정의는 그대로이지만 이 책의 작업량인 수백 차원에서 전부 비교보다 느리다. 근거는 실험 12.1 이다. 16차원에서 점의 86 퍼센트를 보고, 그것도 흩어진 순서로 본다. 그 자리를 근접 그래프가 대신하고, 그래프는 재현율 99 퍼센트를 거리 계산 1 퍼센트로 낸다.

이 장의 요약

> 1. 고차원에서 최근접 거리와 평균 거리의 비가 1 로 가고, 가지치기가 자를 것이 없다. 정확한 답 대신 재현율이다 12.1
> 2. 스킵 리스트는 층을 확률로 올려 로그를 얻는다. 층의 생각은 순서가 아니라 거리 위에도 놓인다 12.2
> 3. 근접 그래프의 탐욕 탐색은 ef 가 재현율을 정하고, 삽입 순서와 이웃 고르기가 긴 간선을 만든다 12.3
> 4. 층은 진입점을 질의 근처로 옮긴다. ef 가 작을 때 재현율을 올리고, 크면 층 없이도 닿는다 12.3
> 5. 디스크에서는 접근 횟수가 값이다. 벡터와 이웃을 같이 두고 거친 벡터를 메모리에 두면 넓힌 노드 수만큼만 읽는다 12.4
>
> 다음 장에서 뒤집히는 것: 이 장은 벡터를 그대로 두고 어느 것을 볼지를 줄였다. 13장은 벡터 자체를 바이트 몇 개로 접고, 그 코드로 거리를 어림한다.

필자의 판단

> 필자는 벡터 검색을 가르칠 때 kd-트리를 먼저 세우고 무너뜨린다. 무너지는 것을 직접 재지 않은 학생은 그래프가 왜 필요한지 모르고 그래프의 재현율을 결함으로 본다. 무너지는 자리를 본 뒤에는 재현율 99 퍼센트가 정확한 답보다 나은 답이라는 것을 이해한다.

열린 문제

하나, 실험 12.3 의 점은 군집이 뚜렷한 합성 데이터다. 실제 임베딩에서 같은 ef 의 재현율이 얼마인지 이 장은 재지 않았다.
둘, 근접 그래프의 걸음 수가 로그라는 것은 실험의 결과이고 정리가 아니다. 어떤 조건에서 로그가 보장되는지는 열린 문제이고, 이 장은 원 논문의 실험을 따랐다.
셋, 실험 12.4 는 접근 횟수를 셀 뿐 디스크를 읽지 않았다. 접근 여럿을 한꺼번에 내는 겹침이 시간을 얼마나 줄이는지 모른다.

연습문제

12.1 기초 좌표가 독립인 d 차원 점에서 두 점 사이 거리 제곱의 평균과 표준편차가 d 와 $\sqrt{d}$ 에 비례함을 명제 12.2 의 논증으로 보이고, $d = 4, 64, 1024$ 에서 상대 편차를 구하라.

12.2 기초 키 백만 개, $p = 1/8$ 의 스킵 리스트의 층 수와 노드당 간선 수와 걸음 상한을 정리 12.5 로 구하고 $p = 1/2$ 와 견줘라.

12.3 기초 이웃 32 의 근접 그래프에서 넓힌 노드가 100 개일 때 거리 계산 수의 상한을 명제 12.11 로 구하고, 점 백만 개의 전부 비교와 견줘라. $p = 1/4$ 이면 층은 몇 개인가.

12.4 기초 768 차원 벡터를 1 바이트로 양자화하고 이웃 64 개를 둘 때 노드 하나의 크기와 4 KB 페이지에 드는 노드 수를 명제 12.14 으로 구하라. 십억 개면 페이지가 몇 GB 인가.

12.5 응용 `dsai.kdtree.KDTree` 의 잎 크기를 4, 16, 64, 256 으로 바꾸며 4 차원에서 들여다본 점의 수를 재라. 잎을 키우는 것으로 고차원의 저주를 피할 수 있는가.

12.6 응용 `dsai.skiplist.SkipList` 에 삭제를 더하라. 층마다 왼쪽 노드를 찾아 링크를 건너뛰게 하고, 맨 위 층이 비면 층 수를 내려라. 삭제 뒤 탐색이 맞는지 시험하라.

12.7 응용 `dsai.nsw.NSW` 의 이웃 고르기를 휴리스틱 없이 가까운 순 M 개로 바꾼 클래스를 만들고, 실험 12.3 의 데이터에서 재현율이 얼마나 떨어지는지 재라.

12.8 응용 페이지 4 KB 와 이웃 번호 4 바이트에서 벡터 차원과 원소 크기에 따라 같이 두기가 되는지, 이웃을 최대 몇 개 둘 수 있는지 계산하는 함수를 쓰고 명제 12.14 의 예로 확인하라.

12.9 탐구 근접 그래프의 M, ef, ef 만들기 셋이 재현율, 메모리, 만들기 시간, 질의 시간에 각각 어떻게 닿는지 표로 적고, 목표 재현율에서 무엇을 먼저 고를지 논하라.

12.10 탐구 거친 벡터의 오차가 재현율을 깎는 경로를 명제 12.13 와 정의 12.7 로 적어라. 다시 재는 후보 수를 늘리면 어느 오차까지 되돌릴 수 있는가.

더 읽을 것

[48] kd-트리의 원전. 차원 k 를 키에 둔 시절의 구조이고, 12.1 절이 그 가정을 무너뜨린다. 12.1
[47] 분할 구조 전부가 고차원에서 선형이 됨을 보인 논문. 순차 훑기를 빠르게 하는 근사 표현이 13 장의 방향이다. 12.1
[49] 스킵 리스트의 원전. 층의 확률과 탐색 비용의 분석이 12.3 절의 층으로 옮겨 간다. 12.2
[50] 계층 근접 그래프. 이웃 고르기 휴리스틱과 ef 의 역할, 층의 확률이 실험으로 정리되어 있다. 12.3

[51] 디스크 위의 그래프. 벡터와 이웃을 페이지에 같이 두고 거친 벡터를 메모리에 두어 십억 개를 SSD 하나에서 찾는다. 12.4

13장

양자화 기반 벡터 인덱스

2011년의 논문 하나는 128차원 벡터를 8바이트로 접고, 접은 채로 거리를 어림해 십억 개를 기계 한 대에서 찾았다 [확립] [52]. 벡터를 부분 공간으로 자르고 부분마다 대표 256개를 학습한 것이 전부였고, 그 위에 역파일을 얹은 것이 지금 벡터 라이브러리의 대표 인덱스 하나다. 이 장은 12장의 반대 방향이다. 12장은 어느 벡터를 볼지 줄였고, 이 장은 본 벡터의 값을 줄인다. 둘을 합치고, 마지막에 넷을 나란히 재어 고르는 절차를 세운다.

이 장은 0.2의 둘째 가정을 벡터에서 다시 본다. 원소가 float32 하나라는 가정이 원소당 1비트로 내려간다. 3.3의 블록 양자화가 원소를 접었다면 여기서는 부분 공간을 접는다. 그리고 확률적 답이 세 번째로 나온다. 코드의 거리는 어림이고, 어림이 재현율을 깎고, 다시 재기가 되돌린다.

이 장에서 세우는 것

13.1 k-평균과 역파일. 군집 넷을 보고 재현율 99퍼센트, 점의 0.4퍼센트

13.2 곱 양자화. 벡터를 바이트 몇 개로. 8바이트에서 메모리 32분의 1

13.3 역파일 위의 잔차 곱 양자화. 잔차의 오차가 4분의 1, 다시 재기로 재현율 40에서 96퍼센트

13.4 인덱스 고르기. 같은 데이터에서 넷을 재고, 작업량별 선택표

13.1 k-평균과 역파일

12.1 은 벡터 검색의 답이 둘이라 했다. 가까운 것만 보는 그래프, 아니면 전부 비교를 더 싸게. 이 장은 뒤쪽이고, 그 첫걸음은 전부를 안 보는 것이다. 점을 군집으로 나누고 질의에 가까운 군집만 본다. 군집을 만드는 것이 k-평균이고, 군집마다 점의 번호 목록을 둔 것이 역파일이다. 역파일은 11.1 의 CSR 그대로다.

이 절은 k-평균을 Lloyd 의 반복으로 세우고, 역파일을 CSR 로 두고, 몇 군집을 보느냐 (nprobe) 가 재현율과 견준 점 수를 어떻게 맞바꾸는지 잰다.

학습 목표

k-평균을 배정과 갱신의 반복으로 정의하고, 매 반복이 목적 함수를 안 늘린다는 것을 보일 수 있다. 역파일을 CSR 로 적고, nprobe 가 재현율과 견준 점 수를 맞바꾸는 이유를 군집 경계로 설명하며 실측으로 말할 수 있다.

이 말부터

중심	군집 하나를 대표하는 벡터. 그 군집에 든 점의 평균이다
역파일	중심 번호에서 그 군집의 점 번호 목록으로 가는 표. 중심이 키이고 목록이 값이다
nprobe	질의마다 들여다보는 군집의 수. 1 이면 가장 가까운 중심의 목록만 본다

디딤돌

디딤돌은 군집화다. 벡터를 가까운 것끼리 묶어 두면, 질의는 묶음의 중심만 먼저 보고 어느 묶음을 열지 정한다. 그 묶음을 만드는 가장 오래된 절차가 Lloyd 의 반복이다.

정의 13.1: k-평균

점 $x_1, \dots, x_n$ 과 중심 $c_1, \dots, c_k$ 에 대해 목적 함수는 $\sum_i \min_j \|x_i - c_j\|^2$ 다. Lloyd 의 반복은 배정 (점마다 가장 가까운 중심) 과 갱신 (중심을 배정된 점의 평균으로) 을 되풀이한다 [확립] [53].

불변식 배정과 갱신 모두 목적 함수를 늘리지 않는다. 배정은 점마다 최소를 고르고, 갱신은 고정된 배정에서 제곱합을 최소로 하는 점이 평균이기 때문이다.

불변식이 반복이 끝난다는 것을 준다. 목적 함수가 0 이상이고 안 늘어나므로 수렴한다. 어디로 수렴하느냐는 시작에 달렸고, 전역 최소가 아니어도 된다. 인덱스에 필요한 것은 점들이 가까운 것끼리 묶이는 것이지 최소가 아니다.

정의 13.2: 역파일

중심 k 개와 배정 $a(i)$ 에 대해, 군집 j 의 목록은 $\{i : a(i) = j\}$ 다. 목록 전부를 군집 순서로 이어 붙인 번호 배열과 경계 포인터가 역파일이고, 11.1 의 CSR 이다. 질의 q 는 중심 k 개와 견줘 가까운 nprobe 개를 고르고, 그 목록의 점만 q 와 견준다.

불변식 점은 정확히 한 목록에 있다. 벡터를 군집 순서로 다시 놓으면 한 군집의 벡터가 메모리에 연속이다.

정의의 둘째 문장이 인덱스다. 군집마다 목록 하나를 두면 질의는 중심 k 개를 견주고 목록 몇 개만 연다. 그 비용이 두 항의 합이고, 군집 수가 그 합을 정한다.

명제 13.3: 역파일의 비용과 재현율

군집 k 개, 점 n 개가 고르게 나뉘면 질의 하나가 하는 거리 계산은 중심 k 번과 점 $\text{nprobe} \cdot n/k$ 번이다. $k = \sqrt{n}$ 근처에서 둘의 합이 최소다. 재현율이 1 이 아닌 이유는 질의의 진짜 이웃이 다른 군집에 있을 수 있어서다. 질의가 군집 경계 근처이면 그렇고, nprobe 를 늘리면 이웃 군집을 함께 봐 그 손실이 준다. 이동량은 nprobe 개 목록의 벡터를 순차로 읽는 것이다. 군집 순서로 놓았으므로 목록 하나가 연속이다.

증명. 비용은 셈이다. $k + \text{nprobe} \cdot n/k$ 를 k 로 미분하면 $k = \sqrt{\text{nprobe} \cdot n}$ 이다. 재현율은 정의 13.2 의 불변식에서 나온다. 점이 한 목록에만 있으므로 그 목록을 안 보면 못 찾는다. 이웃이 다른 군집에 있는 점의 비율이 손실이고, 그것은 경계 근처의 점 비율이다. ■

명제가 말하는 것은 역파일이 12.1 의 kd-트리와 같은 분할 구조라는 것이다. 다른 것은 분할이 축이 아니라 중심이고, 정확한 답 대신 재현율을 받아들인다는 것이다. Weber 등이 무너뜨린 것은 정확한 답을 요구하는 분할이었고, 재현율을 받아들이면 분할이 다시 선다.

예제 13.1: 점 십만 개, 군집 1,024 개

군집마다 점 100 개 남짓이다. nprobe 4 면 중심 1,024 번과 점 400 번으로 1,424 번이고, 전부 비교 십만 번의 1.4 퍼센트다. 경계 근처의 점이 이웃을 놓치고, 그 비율이 재현율의 손실이다.

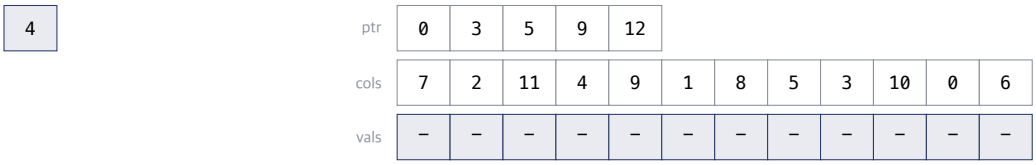


그림 13.1.. 역파일. 중심 넷과 군집 순서로 놓인 벡터. 포인터가 군집의 경계이고, 질의는 가까운 군집 둘의 구간만 읽는다

예제의 1.4 퍼센트에서 재현율이 얼마인지, 그리고 nprobe 를 올리면 어떻게 되는지 잰다.

균열

군집 300 개의 64차원 점 십만 개를 k-평균 1,024 개로 나누고, nprobe 를 1 에서 64 까지 바꾸며 재현율@10 과 견준 점의 비율을 쟀다.

실험 13.1 nprobe 에 따른 재현율과 견준 점 수

구조	nprobe	재현율 at 10 퍼센트	견준 점 퍼센트	질의당 ms
전부 비교	0	100	100	25.5
역파일 1024	1	63.2	0.19	0.22
역파일 1024	4	99.2	0.43	0.32
역파일 1024	16	100	1.54	1.12
역파일 1024	64	100	6.25	2.17

측정 조건. 질의 200 개는 같은 군집 분포. k-평균은 표본 2만 개로 10 번 반복. 시간은 NumPy 의 값이고 전부 비교도 NumPy 다.

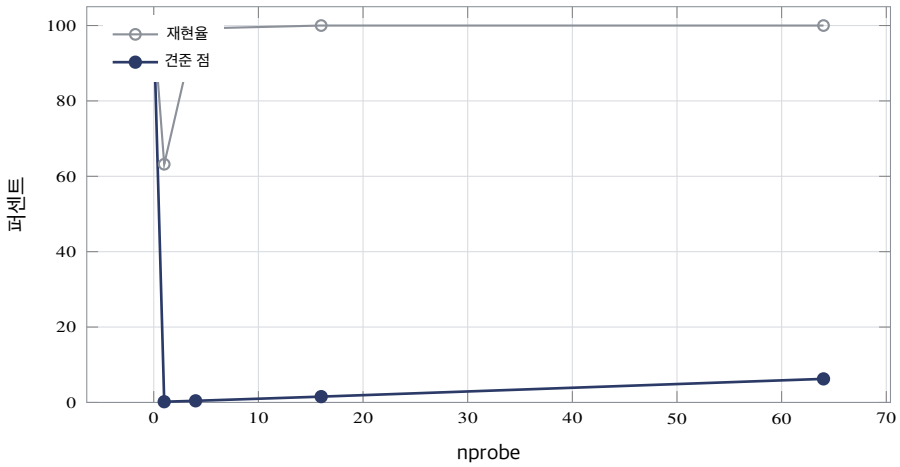


그림 13.2.. nprobe 에 따른 재현율과 견준 점의 비율. 넷에서 재현율 99 퍼센트에 견준 점 0.4 퍼센트다

nprobe 1 에서 재현율 63 퍼센트, 4 에서 99 퍼센트, 16 부터 100 퍼센트다. 견준 점은 0.2, 0.4, 1.5 퍼센트다. nprobe 4 에서 점의 0.4 퍼센트만 보고 99 퍼센트를 찾는다. 시간은 전부 비교의 80 분의 1 이다.

관찰

nprobe 1 의 63 퍼센트가 명제의 경계 손실이다. 질의 셋 중 하나가 이웃을 다른 군집에 두고 있다. 군집 넷을 보면 그 손실이 1 퍼센트로 준다. 견준 점은 네 배가 됐는데 시간은 1.5 배다. 중심 1,024 개와 견주는 값이 nprobe 와 무관하게 들기 때문이다.

이 관찰이 nprobe 를 정하는 규칙이다. 재현율 목표에서 가장 작은 nprobe 를 고르고, 중심 수 k 는 명제의 $\sqrt{n}$ 근처에서 그 값이 최소가 되게 잡는다.

다시 쓰기

이 절의 다시 쓰기는 역파일을 11.1 의 CSR 로 적고 벡터를 그 순서로 놓는 것이다.

명제 13.4: 군집 순서 배치

> 벡터를 군집 순서로 다시 놓으면 nprobe 개 목록의 벡터 읽기가 순차 접근 nprobe 번이다. 원래 순서로 두고 번호로 읽으면 3.4 의 gather 이고 벡터마다 라인 하나다. 군집 하나가 벡터 n/k 개이면 목록 하나가 nds/k 바이트의 연속 구간이고, 그것은 1.3 의 순차 읽기다.

증명. 정의 13.2 의 둘째 불변식이다. 군집 순서로 놓으면 목록이 연속 구간이고, CSR 의 포인터가 그 경계다. ■

이 명제가 역파일이 12.3 의 그래프와 다른 자리다. 그래프의 이웃 벡터 읽기는 의존 읽기였고, 역파일의 목록 읽기는 순차 읽기다. 같은 수의 벡터를 읽어도 이동량이 다르다. 그리고 그 순차 구간이 다음 절에서 바이트 몇 개로 접힌다.

알고리즘 13.1 역파일 탐색 `dsai.ivf.IVF.search`

> **입력 :** 중심 c, 포인터, 군집 순서의 벡터 X, 질의 q, nprobe, k
> 1: 중심 k 개와의 거리에서 가까운 nprobe 개의 번호 L
> 2: 후보 $\leftarrow L$ 의 각 군집 구간을 이어 붙인 번호
> 3: 후보의 벡터와 q 의 거리를 재고 상위 k 를 10.1 의 선택으로 고른다

구현

k-평균은 배정 하나와 평균 하나의 반복이다. 배정은 행렬 곱 하나로 한다.

지금 보는 코드 점 전부와 중심 전부의 거리를 행렬 곱으로 만들고, 행마다 최소를 고른다.

구현 Lloyd 의 반복과 배정 `samples/dsai/ivf.py:12`

```python
def kmeans(x: np.ndarray, k: int, rng: np.random.Generator,
           iters: int = 10) -> np.ndarray:
    """알고리즘 13.1. Lloyd 의 반복. 중심 (k, d)."""
    c = x[rng.choice(len(x), k, replace=False)].copy()
    for _ in range(iters):
        a = assign(x, c)
        for j in range(k):
            m = a == j
            if m.any():
                c[j] = x[m].mean(axis=0)
    return c


def assign(x: np.ndarray, c: np.ndarray) -> np.ndarray:
    """가장 가까운 중심의 번호. |x|^2 - 2 x.c + |c|^2 를 행렬 곱으로."""
    d2 = (x * x).sum(axis=1)[:, None] - 2.0 * x @ c.T + (c * c).sum(axis=1)[None, :]
    return np.argmin(d2, axis=1)
```

읽어 보면 `assign` 의 한 줄이 $|x|^2 - 2x \cdot c + |c|^2$ 다. 거리 $n \times k$ 개를 행렬 곱 하나로 만들어 0.2 의 밀도가 높은 연산이 된다. 점마다 루프를 돌면 밀도가 1 이다.

구현 역파일 만들기와 탐색 `samples/dsai/ivf.py:34`

```
    def __init__(self, x: np.ndarray, centroids: np.ndarray):
        self.x = x
        self.c = centroids
        a = assign(x, centroids)
        order = np.argsort(a, kind="stable")
        self.ids = order.astype(np.int64)
        counts = np.bincount(a, minlength=len(centroids))
        self.ptr = np.zeros(len(centroids) + 1, dtype=np.int64)
        np.cumsum(counts, out=self.ptr[1:])
        self.xs = x[self.ids]                  # 군집 순서로 다시 놓은 벡터

    def search(self, q: np.ndarray, nprobe: int, k: int) -> tuple[np.ndarray, int]:
        """알고리즘 13.2. 가까운 중심 nprobe 개의 목록만 견준다. (상위 k 번호,
        견준 점 수)."""
        dc = ((self.c - q) ** 2).sum(axis=1)
        lists = np.argpartition(dc, nprobe - 1)[:nprobe]
        cand = np.concatenate([np.arange(self.ptr[l], self.ptr[l + 1]) for l in lists])
        d2 = ((self.xs[cand] - q) ** 2).sum(axis=1)
        top = cand[np.argsort(d2)[:k]]
        return self.ids[top], len(cand)
```

읽어 보면 `argsort(a)` 가 11.1 의 CSR 만들기이고, `self.xs = x[self.ids]` 가 군집 순서 배치다. `search` 의 `np.concatenate` 가 nprobe 개 구간을 잇고, 그 뒤는 배열 하나의 거리 계산이다.

분석

k-평균과 역파일은 그대로 유효하다. 1982 년의 반복과 검색 엔진의 역색인이 그대로이고, 벡터에 갖다 댄 것이다. 바뀐 것은 정확한 답을 재현율로 바꿨다는 것과, 목록의 벡터를 군집 순서로 놓아 순차 읽기로 만들었다는 것이다. 앞의 것은 12 장의 결론이고 뒤의 것은 이 책의 잣대다.

이 구조가 서는 자리는 벡터 수억 개의 인덱스다. 그래프보다 메모리가 작고 만들기가 싸다. 그래프는 점마다 이웃 목록이 있는데 역파일은 점마다 번호 하나다. 그리고 이 구조 위에 다음 절의 양자화가 얹혀 벡터 자체가 작아진다. 둘을 합친 것이 13.3 이고, Jégou 등의 2011 년 논문이 그 결합이었다 [확립] [52].

현장에서

> 역파일은 텍스트 검색의 역색인에서 온 이름이다. 단어에서 문서 목록으로 가던 표가 중심에서 벡터 목록으로 가는 표가 됐다. 벡터 라이브러리의 기본 인덱스 중 하나가 이것이고, 군집 수를 $\sqrt{n}$ 근처로 두라는 지침이 명제 13.3 에서 나온다 [확립] [4].

주의

> 실험 13.1 의 점은 군집 300 개에서 나왔고 k-평균의 군집 1,024 개가 그것을 잘 나눈다. 군집 구조가 약한 데이터에서는 같은 nprobe 의 재현율이 낮고, 경계 근처의 점이 더 많다.

역파일은 어느 벡터를 볼지를 줄였다. 그러나 본 벡터의 값은 그대로 256 바이트이고, 메모리도 그대로다. 다음 절은 그 벡터 하나를 바이트 몇 개로 접는다.

되짚기

> k-평균을 목적 함수를 안 늘리는 반복으로 정의하고, 역파일을 CSR 로 두었다. nprobe 4 에서 점의 0.4 퍼센트를 보고 재현율 99 퍼센트였다. 목록의 벡터는 군집 순서로 놓아 순차 읽기다. 다음 절은 벡터를 접는 곱 양자화다.

13.2 곱 양자화. 벡터를 바이트 몇 개로

13.1 의 역파일이 본 벡터는 64 차원 float32, 256 바이트다. 십억 개면 256 GB 이고 메모리에 안 든다. 12.4 의 디스크 그래프가 거친 벡터를 메모리에 둔 이유가 이것이었다. 벡터를 접는 가장 단순한 방법은 3.3 의 블록 양자화다. 원소마다 비트를 줄인다. 이 절의 방법은 다르다. 벡터를 부분 공간 몇 개로 자르고, 부분마다 대표 벡터 256 개를 학습해 그 번호 하나로 부분을 대신한다. 부분이 m 개면 벡터 하나가 바이트 m 개다. Jégou 등이 2011 년에 이 표현으로 거리를 어림하는 방법을 세웠다 [확립] [52].

이 절은 곱 양자화를 정의하고, 코드에서 벡터를 복원하지 않고 거리를 어림하는 비대칭 거리를 세운 뒤, 바이트 수와 오차와 재현율을 잰다.

학습 목표

곱 양자화를 부분 공간의 k-평균으로 정의하고, 코드북의 크기와 코드의 크기를 부분 수로 적을 수 있다. 비대칭 거리가 거리표 m 번 읽기로 계산됨을 보이고, 부분 수가 오차와 재현율을 어떻게 맞바꾸는지 실측으로 말할 수 있다.

이 말부터

부분 공간	벡터의 좌표를 m 개 묶음으로 자른 것. 부분마다 d/m 차원이다
코드북	부분 공간 하나의 대표 벡터 256 개. 부분마다 하나씩 m 개다
비대칭 거리	질의는 원래 벡터, 데이터는 코드인 채로 재는 거리. 코드를 복원하지 않는다

디딤돌

정의 13.5: 곱 양자화

d 차원 벡터를 m 개의 d/m 차원 부분으로 자른다. 부분 j 마다 정의 13.1 으로 중심 256 개의 코드북 C_j 를 만든다. 벡터 x 의 코드는 부분마다 가장 가까운 중심의 번호 $(c_1(x), \dots, c_m(x))$ 이고 바이트 m 개다. 복원은 $\hat{x} = (C_1[c_1], \dots, C_m[c_m])$ 이다.

불변식 코드북의 크기는 $m \times 256 \times d/m = 256d$ 개의 실수로 m 과 무관하다. 코드의 크기는 m 바이트다. 재구성 오차 $\|x - \hat{x}\|^2$ 는 부분별 오차의 합이다.

불변식의 첫 문장이 이 표현의 요점이다. 코드북은 벡터 256 개 크기이고, 데이터 n 개는 바이트 mn 개다. n 이 십억이고 m 이 8 이면 8 GB 다. 셋째 문장이 오차의 성질이다. 부분을 많이 자를수록 부분마다 차원이 낮아 중심 256 개가 더 촘촘하고, 오차가 준다.

명제 13.6: 비대칭 거리

> 질의 q 와 코드 $(c_1, \dots, c_m)$ 의 벡터 x 에 대해 $\|q - \hat{x}\|^2 = \sum_j \|q_j - C_j[c_j]\|^2$ 다. q_j 는 q 의 부분 j 다. 부분마다 q_j 와 중심 256 개의 거리를 표 $T_j[\cdot]$ 로 한 번 만들면, 코드 하나의 거리는 표 읽기 m 번과 덧셈 m 번이다. 표는 질의당 한 번이고 $256d$ 번의 연산이다.

증명. 제곱 거리는 좌표별 제곱 차이의 합이고 부분으로 묶어도 같다. 부분 j 의 항은 q_j 와 코드북 항목의 거리이므로 $T_j[c_j]$ 다. ■

명제에서 읽을 것은 데이터 하나당 비용이다. d 차원 벡터의 거리가 곱셈 d 번인데, 코드의 거리는 읽기 m 번이다. $d = 64$, $m = 8$ 이면 8 분의 1 이고, 읽는 것도 벡터 256 바이트가 아니라 코드 8 바이트다. 이동량이 32 분의 1 이다. 그 대가가 오차이고, 오차가 재현율을 깎는다.

예제 13.2: 64 차원, 부분 8

> 부분 하나가 8 차원이고 중심 256 개다. 벡터 하나가 8 바이트, 코드북이 $256 \times 64 = 16{,}384$ 실수다. 십만 개면 코드 0.8 MB 이고 원래 25.6 MB 다. 거리표는 질의마다 8×256 개이고, 십만 개의 거리는 표 읽기 80 만 번이다.

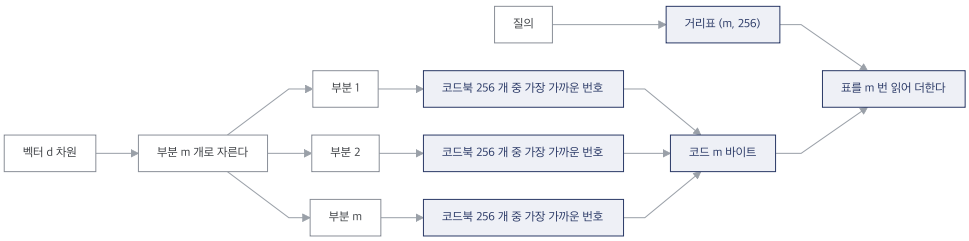


그림 13.3.. 곱 양자화. 벡터를 부분 m 개로 자르고 부분마다 코드북에서 번호 하나를 고른다. 질의는 거리표를 만들고 코드마다 표를 m 번 읽는다

예제의 32 분의 1 이 재현율을 얼마나 깎는지, 부분 수를 바꾸며 잰다.

균열

같은 점 십만 개를 부분 4, 8, 16, 32 개로 양자화하고, 재구성 오차와 비대칭 거리로 고른 상위 10 의 재현율과 메모리를 쟀다.

실험 13.2 부분 수에 따른 오차, 재현율, 메모리

표현	바이트	오차 퍼센트	재현율 퍼센트	MB	질의 ms
float32	256	0.00	100	25.6	37.1
곱 양자화 m=4	4	16.2	3.30	0.40	20.1
곱 양자화 m=8	8	11.3	4.70	0.80	26.3
곱 양자화 m=16	16	6.31	22.5	1.60	40.3
곱 양자화 m=32	32	0.80	67.0	3.20	55.4

측정 조건. 코드북은 표본 2만 개로 학습. 재구성 오차는 $\|x - \hat{x}\|^2$ 의 평균을 $\|x\|^2$ 의 평균으로 나눈 것. 재현율은 코드 전부의 비대칭 거리로 고른 상위 10.

부분 4 개 (4 바이트) 에서 오차 16 퍼센트, 재현율 3 퍼센트다. 32 개 (32 바이트) 에서 오차 0.8 퍼센트, 재현율 67 퍼센트다. 메모리는 64 분의 1 에서 8 분의 1 이다. 바이트를 두 배로 할 때마다 오차가 반 남짓 줄고 재현율이 오른다. 32 바이트에서도 67 퍼센트인 것은 상위 10 개 사이의 거리 차이가 오차보다 작기 때문이다.

관찰

재현율 67 퍼센트는 코드만으로 상위 10 을 고른 값이다. 코드로 후보 백 개를 고르고 원래 벡터로 다시 재면 오차가 후보 안에만 남는다. 그것이 다음 절이다. 시간은 부분 수에 비례해는다. 표 읽기 m 번이 거리 하나이기 때문이다.

이 관찰이 곱 양자화를 혼자 쓰지 않는 이유다. 코드로 거르고 벡터로 확인한다. 7.1 의 블룸 필터, 12.4 의 거친 벡터와 같은 자리다.

다시 쓰기

이 절의 다시 쓰기는 비대칭 거리의 이동량을 0.2 의 루프라인에 놓는 것이다.

명제 13.7: 코드 훑기의 밀도

코드 n 개의 비대칭 거리는 이동 mn 바이트에 연산 $2mn$ 번이라 밀도가 2 다. 원래 벡터의 거리는 이동 $4dn$ 바이트에 연산 $2dn$ 번이라 밀도가 $1/2$ 이다. 둘 다 능선 아래이고 시간은 이동량이 정한다. 코드가 $4d/m$ 배 빠르다. $d = 64$, $m = 8$ 이면 32 배다. 거리표 T 는 $256m$ 개의 실수로 캐시에 들고, 표 읽기는 라인 안의 무작위 접근이라 이동량에 안 든다.

증명. 이동량과 연산량은 셈이다. 밀도는 그 비다. 표 $256m \times 4$ 바이트는 $m = 32$ 에서 32 KB 라 L1 에 든다. ■

이 명제가 곱 양자화가 가속기에서 빠른 이유다. 표가 캐시에 있고, 코드가 순차로 읽히고, 읽기 m 번이 벡터 명령 하나로 된다. 실험의 시간이 부분 수에 비례해 늘어난 것은 NumPy 의 첨자 읽기가 벡터 명령이 아니기 때문이고, 방향은 명제대로다.

알고리즘 13.2 거리표와 비대칭 거리 `dsai.pq.PQ.adc`

입력 : 코드북 $C_1, \dots, C_m$, 코드 $K[0..n, 0..m)$, 질의 q
1: $T_j[b] \leftarrow \|q_j - C_j[b]\|^2$ 을 모든 j, b 에 대해 ▷ $256m$ 번
2: **모든 i 동안** $d_i \leftarrow \sum_j T_j[K[i,j]]$ ▷ 표 읽기 m 번
3: **끝 반복**

구현

코드북은 부분마다 k-평균 하나다. 부호화는 부분마다 배정 하나이고, 거리는 표 읽기다.

지금 보는 코드 질의를 부분으로 자르고 부분마다 중심 256 개와의 거리표를 만든 뒤, 코드를 첨자로 표를 읽어 더한다.

구현 거리표와 비대칭 거리 `samples/dsai/pq.py:36`

```
36      def table(self, q: np.ndarray) -> np.ndarray:
37          """알고리즘 13.3. 질의 하나의 거리표 (m, 256).
38          부분마다 질의 조각과 중심 256 개의 거리.
39          """
40          t = np.empty((self.m, 256))
41          for j in range(self.m):
42              qj = q[j * self.ds:(j + 1) * self.ds]
43              t[j] = ((self.books[j] - qj) ** 2).sum(axis=1)
44          return t
45
46      def adc(self, table: np.ndarray, codes: np.ndarray) -> np.ndarray:
47          """비대칭 거리. 코드 (n, m) 마다 표를 m 번 읽어 더한다.
48          벡터를 복원하지 않는다.
49          """
50          return table[np.arange(self.m)[None, :], codes].sum(axis=1)
```

읽어 보면 `table[np.arange(self.m)[None, :], codes]` 가 코드 (n, m) 마다 표 $(m, 256)$ 을 읽는 gather 다. `sum(axis=1)` 이 덧셈 m 번이다. 벡터를 복원하는 줄이 없다.

구현 재구성 오차 `samples/dsai/pq.py:56`

```
56  def quantization_error(pq: PQ, x: np.ndarray) -> float:
57      """정의 13.3 의 재구성 오차.
58      |x - decode(encode(x))|^2 의 평균을 |x|^2 의 평균으로 나눈 것.
59      """
60      r = pq.decode(pq.encode(x))
61      return float(((x - r) ** 2).sum(axis=1).mean() / (x ** 2).sum(axis=1).mean())
```

읽어 보면 오차를 $\|x\|^2$ 의 평균으로 나눠 단위 없는 비율로 둔다. 실험의 오차 열이 이것이고, 부분 수가 두 배면 오차가 반 남짓이다.

분석

곱 양자화는 교과서의 양자화를 조정한 것이다. 원소마다 비트를 줄이는 대신 부분 공간마다 대표를 학습하고, 벡터를 복원하지 않고 거리를 어림한다. 3.3 의 블록 양자화가 원소를

접었다면 이 절은 부분 공간을 접는다. 원소당 비트가 1 아래로 내려간다. 64차원을 8바이트로 접으면 원소당 1 비트다.

이 구조가 서는 자리는 벡터 수십억 개를 메모리에 두는 일이다. 256 바이트가 8 바이트가 되면 십억 개가 8 GB 다. 그리고 12.4 의 거친 벡터가 바로 이것이다. 디스크 그래프의 메모리 부분이 곱 양자화 코드이고, 정확한 벡터는 페이지에 있다.

현장에서

> Jégou 등의 논문은 부분 8 개, 코드북 256 의 설정으로 128차원 벡터를 8 바이트로 접고, 비대칭 거리가 대칭 거리 (질의도 양자화) 보다 정확함을 보였다 [확립] [52]. Guo 등은 2020년에 내적 검색에서 오차의 방향을 나눠, 질의와 평행한 성분의 오차에 더 큰 벌을 주는 양자화가 재현율을 올린다고 보고했다 [확립] [54].

주의

> 실험 13.2 의 코드북은 표본 2만 개로 학습했다. 데이터의 분포가 표본과 다르면 오차가 크다. 그리고 부분을 좌표 순서대로 잘랐는데, 좌표 사이의 상관이 부분을 가로지르면 오차가 든다. 회전을 먼저 하는 변형이 그것을 줄인다.

역파일이 어느 벡터를 볼지 줄였고, 곱 양자화가 본 벡터의 값을 줄였다. 둘을 합치면 벡터 십억 개가 메모리 몇 GB 에 든다. 다음 절이 그 결합이고, 잔차라는 하나의 생각이 오차를 더 줄인다.

되짚기

> 곱 양자화를 부분 공간의 k-평균으로 정의하고 비대칭 거리가 표 읽기 m 번임을 보였다. 8바이트에서 메모리 32 분의 1 이고, 코드만으로 고른 재현율은 낮아 확인이 필요하다. 다음 절은 역파일과의 결합이다.

13.3 역파일과 곱 양자화의 결합. 비대칭 거리

13.1 은 볼 벡터를 줄였고 13.2 은 본 벡터의 값을 줄였다. 둘을 포개면 벡터 십억 개가 메모리 몇 GB 에 들고 질의 하나가 코드 수만 개의 표 읽기다. 포개는 자리에 생각 하나가 더 있다. 벡터 자체가 아니라 벡터에서 군집 중심을 뺀 잔차를 양자화한다. 잔차는 원래 벡터보다 작아 같은 바이트로 오차가 작다. Jégou 등의 2011 년 논문이 이 결합을 세웠고 [확립] [52], 지금 벡터 라이브러리의 대표 인덱스 하나가 이것이다.

이 절은 결합을 정의하고, 잔차의 오차가 왜 작은지 보이고, 코드로 고른 후보를 원래 벡터로 다시 재는 것이 재현율을 얼마나 되돌리는지 잰다.

학습 목표

역파일 위의 잔차 곱 양자화를 정의하고, 질의 하나에 드는 비용을 군집 수와 nprobe 와 부분수로 적을 수 있다. 잔차의 재구성 오차가 원 벡터보다 작은 이유를 말하고, 다시 재기가 재현율을 되돌리는 값을 실측으로 말할 수 있다.

이 말부터

잔차	벡터에서 그 군집의 중심을 뺀 것. 중심 근처의 점일수록 작다
군집별 거리표	질의에서 중심을 뺀 것으로 만든 거리표. 군집마다 하나씩 nprobe 개다
다시 재기	코드로 고른 상위 후보 몇 개를 원래 벡터로 정확히 재어 순서를 고치는 것

디딤돌

정의 13.8: 역파일 위의 잔차 곱 양자화

정의 13.2 의 역파일과 배정 $a(i)$ 에 대해, 점 x_i 의 잔차 $r_i = x_i - c_{a(i)}$ 를 정의 13.5 로 양자화한다. 저장은 군집 순서의 코드 (n, m) 바이트와 번호다. 질의 q 는 가까운 군집 nprobe 개를 고르고, 군집 j 마다 $q - c_j$ 의 거리표를 만들어 그 군집의 코드에 비대칭 거리를 더한다.

불변식 $\|q - \hat{x}_i\|^2 = \|(q - c_{a(i)}) - \hat{r}_i\|^2$ 이므로, 군집 j 의 거리표를 $q - c_j$ 로 만들면 코드의 거리가 원 벡터와 복원 벡터의 거리다.

불변식이 잔차를 써도 거리가 맞는 이유다. 질의에서 같은 중심을 빼면 차이는 그대로다. 대가는 거리표를 군집마다 하나씩 만드는 것이고, 그것이 nprobe 배의 $256d$ 다.

명제 13.9: 잔차의 오차

> 군집 안의 점들이 중심 근처에 모여 있으면 잔차의 크기가 원 벡터보다 작고, 같은 코드북 크기에서 재구성 오차가 그만큼 작다. 군집 k 개로 나눈 뒤의 잔차 분산은 k-평균의 목적 함수를 n 으로 나눈 것이고, 그것은 원 벡터의 분산에서 군집 중심들의 분산을 뺀 것이다.

증명. 분산의 분해다. 총 제곱합은 군집 사이 제곱합과 군집 안 제곱합의 합이고, k-평균은 군집 안 제곱합을 줄인다. 잔차의 제곱합이 군집 안 제곱합이다. 양자화 오차는 양자화하는 벡터의 분산에 비례한다. ■

명제 13.10: 질의의 비용

> 군집 k, nprobe, 부분 m, 점 n 에서 질의 하나는 중심 거리 kd 번, 거리표 nprobe $\cdot 256d$ 번, 코드 거리 nprobe $\cdot (n/k) \cdot m$ 번의 표 읽기다. 다시 재기 R 개를 더하면 원 벡터 거리 Rd 번과 벡터 R 개의 무작위 읽기가 든다. 메모리는 코드 mn 바이트와 원 벡터 (다시 재기용) 다.

증명. 명제 13.3 와 명제 13.6 를 더한 것이다. 거리표가 군집마다 하나라 nprobe 배다. ■

명제의 둘째 문장이 이 절의 실험이다. 다시 재기는 원 벡터를 읽으므로 코드만의 이점을 일부 되돌린다. 얼마나 되돌리고 얼마를 얻는지가 문제다.

예제 13.3: 점 십만 개, 군집 1,024, 부분 8

> 코드 0.8 MB 에 번호와 포인터를 더해 1.7 MB 다. 원 벡터 25.6 MB 의 15 분의 1 이다. nprobe 32 면 군집 32 개의 점 3,100 개에 표 읽기 8 번씩 25,000 번이고, 거리표 32 개에 $256 \times 64 \times 32 = 52$ 만 번이다. 거리표가 더 크다. 군집이 작으면 표가 부담이다.

그림 13.4.. 역파일 위의 곱 양자화 코드. 벡터 하나가 군집 번호 하나와 잔차의 부분 코드 넷으로 5 바이트다

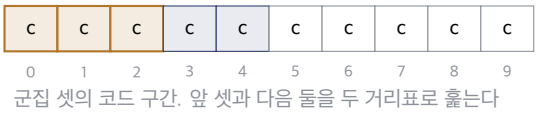


그림 13.5.. 역파일 위의 잔차 코드. 군집 순서의 코드가 연속이고, 군집마다 거리표가 하나씩이다. 질의는 가까운 군집의 구간만 훑는다

예제의 표 부담과 다시 재기의 값이 실측에서 어떻게 나오는지 잰다.

균열

같은 점 십만 개에 군집 1,024, 부분 8 의 인덱스를 세우고, nprobe 와 다시 재기 수를 바꾸며 재현율과 견준 점 수를 쟀다. 원 벡터와 잔차의 재구성 오차도 견줬다.

실험 13.3 nprobe 와 다시 재기에 따른 재현율

nprobe	다시 잰 후보	재현율 at 10 퍼센트	견준 점 퍼센트	질의당 ms
8	0	39.9	0.78	1.48
32	0	39.9	3.11	8.21
32	100	95.7	3.11	6.46
128	0	39.9	12.6	35.6
128	100	95.7	12.6	31.7

측정 조건. 인덱스 1.7 MB. 재구성 오차는 원 벡터를 양자화하면 11.3 퍼센트, 잔차를 양자화하면 2.8 퍼센트. 다시 재기는 코드로 고른 상위 100 개를 원 벡터로 다시 잰다.

잔차의 오차가 원 벡터의 4 분의 1 이다. 명제 13.9 대로다. 그런데 코드만으로 고른 재현율은 nprobe 를 8 에서 128 로 올려도 40 퍼센트 그대로다. 상위 10 개 사이의 거리 차이가 코드의 오차보다 작아서, 후보에는 있는데 순서가 틀린다. 다시 재기 100 개를 더하자 96 퍼센트다. nprobe 32 와 128 에서 같다. 후보는 이미 nprobe 32 에 다 있었다.

관찰

nprobe 는 후보에 진짜 이웃이 드는지를 정하고, 다시 재기는 후보 안의 순서를 정한다. 둘은 다른 손실을 고친다. 코드만으로 40 퍼센트인 것은 오차 2.8 퍼센트가 작지 않아서가 아니라, 이웃 사이의 거리 차이가 그보다 작기 때문이다.

이 관찰이 12.4 의 두 손잡이를 다시 보인다. 거친 거리로 후보를 모으고 정확한 거리로 순서를 고친다. 그때는 디스크 접근이 값이었고 여기서는 원 벡터 읽기 백 번이 값이다.

다시 쓰기

이 절의 다시 쓰기는 다시 재기를 비용에 넣는 것이다.

명제 13.11: 다시 재기의 값

다시 재기 R 개는 원 벡터 R 개의 무작위 읽기이고 Rds 바이트다. 코드 훑기가 순차 $\text{nprobe} \cdot (n/k) \cdot m$ 바이트이므로, Rds 가 그보다 작으면 다시 재기는 거의 공짜다. 실험의 값은 원 벡터 $100 \times 256 = 25$ KB 와 코드 $3{,}100 \times 8 = 25$ KB 로 같다. 재현율은 40 에서 96 퍼센트다. 원 벡터를 메모리에 못 두면 R 번의 디스크 접근이고, 그것이 12.4 의 값이다.

증명. 셈이다. 다시 재기가 읽는 것은 후보 R 개의 벡터이고 후보는 흩어져 있다. ■

이 명제가 인덱스가 세 겹인 이유다. 군집으로 거르고, 코드로 거르고, 벡터로 확인한다. 겹마다 보는 수가 줄고 값이 오른다. 7.4 의 중복 제거 파이프라인이 같은 모양이었고, 이 책의 확률적 구조는 전부 이 모양이다.

알고리즘 13.3 역파일 위의 코드 훑기와 다시 재기 `dsai.ivfpq.IVFPQ.search`

입력 : 중심 c, 포인터, 코드 K, 질의 q, nprobe, R, k

1: 가까운 군집 nprobe 개 L
2: **모든** $j \in L$ **동안** 거리표 $T \leftarrow$ 명제 13.6$(q - c_j)$; 구간의 코드마다 $\sum T[K]$
3: **끝 반복**
4: 후보 중 상위 R 개를 10.1 의 선택으로 고르고, 원 벡터로 다시 재어 상위 k

구현

인덱스는 역파일 하나와 잔차의 곱 양자화 하나다. 탐색은 군집마다 거리표를 만들어 코드에 더한다.

지금 보는 코드 군집마다 질의에서 중심을 뺀 거리표를 만들고, 그 군집의 코드 구간에 비대칭 거리를 더한다.

구현 잔차 양자화 `samples/dsai/ivfpq.py:19`

```
19      def __init__(self, x: np.ndarray, centroids: np.ndarray, m: int,
20                   rng: np.random.Generator):
21          self.ivf = IVF(x, centroids)
22          a = np.repeat(np.arange(len(centroids)), np.diff(self.ivf.ptr))
23          resid = self.ivf.xs - centroids[a]                # 군집 순서의 잔차
24          self.pq = PQ(resid, m, rng)
25          self.codes = self.pq.encode(resid)                # (n, m) uint8, 군집 순서
26          self.c = centroids
```

읽어 보면 `resid = self.ivf.xs - centroids[a]` 가 군집 순서의 잔차다. 그것을 정의 13.5 로 양자화하고 코드도 군집 순서다. 정의 13.2 의 둘째 불변식이 여기서 코드에도 적용된다.

구현 군집별 거리표와 다시 재기 `samples/dsai/ivfpq.py:28`

```
    def search(self, q: np.ndarray, nprobe: int, k: int,
               rerank: int = 0) -> tuple[np.ndarray, int]:
        """알고리즘 13.4. 가까운 군집마다 잔차 거리표를 만들고 코드에 더한다.
        rerank 개는 정확히 다시 잰다.
        """
        dc = ((self.c - q) ** 2).sum(axis=1)
        lists = np.argpartition(dc, nprobe - 1)[:nprobe]
        cand, dist = [], []
        for l in lists:
            lo, hi = self.ivf.ptr[l], self.ivf.ptr[l + 1]
            if hi == lo:
                continue
            t = self.pq.table(q - self.c[l])
            cand.append(np.arange(lo, hi))
            dist.append(self.pq.adc(t, self.codes[lo:hi]))
        cand, dist = np.concatenate(cand), np.concatenate(dist)
        if rerank:
            top = cand[np.argpartition(dist, min(rerank, len(cand)) - 1)[:rerank]]
            exact = ((self.ivf.xs[top] - q) ** 2).sum(axis=1)
            top = top[np.argsort(exact)[:k]]
        else:
            top = cand[np.argsort(dist)[:k]]
        return self.ivf.ids[top], len(cand)
```

읽어 보면 `self.pq.table(q - self.c[l])` 이 군집 l 의 거리표다. `rerank` 가 0 이면 코드의 거리로 끝내고, 아니면 상위 `rerank` 개를 `self.ivf.xs` 로 다시 잰다. 실험의 40 과 96 이 이 `if` 의 두 갈래다.

분석

이 결합은 교과서에 없던 구조다. 역색인과 k-평균과 양자화가 각각 교과서에 있고, 셋을 잔차 하나로 이은 것이 2011 년이다. 그래서 판정은 셋의 것이다. 셋 다 그대로 유효하고, 이은 것이 새것이다. 이은 자리의 값이 명제 13.9 이고, 실험의 4 분의 1 이다.

이 구조가 서는 자리는 벡터 수십억 개를 기계 몇 대의 메모리에 두고 검색하는 일이다. 12.3 의 그래프가 재현율과 지연에서 이기고, 이 구조가 메모리와 만들기에서 이긴다. 어느 쪽을 고를지가 다음 절이다.

현장에서

> Jégou 등의 논문은 부분 8 개의 코드와 군집 만 개 안팎의 역파일로 십억 개를 기계 한 대에 두었고 [확립] [52], Johnson 등은 같은 구조를 가속기에 올려 코드 훑기를 벡터 명령으로 만들었다 [확립] [4]. 가속기의 코드 훑기는 거리표를 레지스터에 두고 코드를 순차로 읽는다. 명제 13.7 의 밀도가 그것을 허락한다.

주의

> 실험 13.3 의 거리표는 군집마다 파이썬 루프로 만들었고, 그것이 시간의 대부분이다. 실제 구현은 거리표를 행렬 곱 하나로 nprobe 개를 한꺼번에 만든다. 시간은 방향만 읽는다.

역파일, 곱 양자화, 잔차, 다시 재기까지 세웠다. 12장의 그래프까지 넷이 같은 문제를 푼다. 다음 절은 같은 데이터, 같은 재현율 목표에서 넷을 나란히 재고 고르는 잣대를 세운다.

되짚기

> 역파일 위의 잔차 곱 양자화를 정의하고, 잔차의 오차가 원 벡터의 4 분의 1 임을 봤다. 코드만으로는 재현율 40 퍼센트이고 다시 재기 100 개가 96 퍼센트로 되돌렸다. nprobe 는 후보를, 다시 재기는 순서를 고친다. 다음 절은 인덱스 고르기다.

13.4 재현율과 지연. 인덱스 고르기

12장과 13장이 같은 문제에 답 넷을 세웠다. 전부 비교, 근접 그래프, 역파일, 그리고 역파일 위의 곱 양자화. 어느 것이 맞느냐는 물음에 답이 하나가 아니다. 답은 재현율 목표와 메모리 예산과 지연 예산과 데이터 크기와 갱신 빈도의 함수이고, 그것을 한 표로 적는 것이 이 절이다.

이 절은 같은 데이터, 같은 질의, 같은 재현율 목표에서 넷을 나란히 재고, 그 결과를 0.3 의 잣대와 0.1 의 이동량으로 읽는다. 마지막에 작업량별 선택표 한 쪽을 둔다. 이 책이 벡터 인덱스에 대해 말하는 것의 요약이다.

학습 목표

재현율 목표를 정하고 그 목표를 넘는 가장 싼 설정을 인덱스마다 찾는 절차를 적을 수 있다. 네 인덱스의 메모리, 만들기, 질의 지연을 같은 조건에서 견주고, 어느 작업량에서 어느 것이 이기는지 근거를 들어 말할 수 있다.

이 말부터

재현율 곡선	인덱스의 설정 (nprobe, ef) 을 바꿀 때 재현율과 지연이 그리는 선. 목표에서 가장 왼쪽 점을 고른다
만들기 비용	인덱스를 세우는 시간. 그래프는 삽입 n 번, 역파일은 k-평균과 정렬 하나다
갱신	벡터가 더해지거나 지워지는 것. 그래프는 삽입이 되고 역파일은 다시 만든다

디딤돌

정의 13.12: 재현율 목표를 건 비용

인덱스 I 와 설정 θ 에 대해 재현율 $r(I,\theta)$ 와 질의 지연 $t(I,\theta)$ 를 잰다. 목표 ρ 에서 I 의 비용은 $\min\{t(I,\theta) : r(I,\theta) \geq \rho\}$ 이고, 그 설정을 θ^* 라 한다. 메모리 $M(I)$ 와 만들기 $B(I)$ 는 설정과 거의 무관하다.

불변식 같은 데이터와 질의에서 잰 값끼리만 견준다. 재현율은 정확한 답과의 겹침이고, 정확한 답은 전부 비교로 한 번 구해 둔다.

불변식이 이 절의 실험 규칙이다. 인덱스마다 다른 데이터로 잰 숫자는 견줄 수 없다. 원 논문들이 각자 다른 데이터에서 보고한 숫자를 한 표에 놓으면 안 되는 이유다.

명제 13.13: 네 인덱스의 비용 모양

전부 비교는 메모리 $4dn$, 만들기 0, 지연 $\Theta(dn)$ 이고 재현율 1 이다. 역파일은 메모리 $4dn + 8n$, 만들기 k-평균 하나, 지연 $\Theta(kd + \text{nprobe} \cdot nd/k)$ 이다. 역파일 위의 곱 양자화는 메모리 $mn + 8n$ (다시 재기용 원 벡터는 별도), 만들기 k-평균 둘, 지연 $\Theta(kd + \text{nprobe} \cdot nm/k + Rd)$ 다. 근접 그래프는 메모리 $4dn + 8Mn$, 만들기 삽입 n 번 (각각 탐색 하나), 지연 $\Theta(tMd)$ 이고 t 는 넓힌 노드 수로 $\log n$ 근처다.

증명. 각 장의 명제를 모은 것이다. 명제 13.3, 명제 13.10, 명제 12.11. ■

명제에서 읽을 것은 넷의 자리가 다르다는 것이다. 메모리는 곱 양자화가 이기고, 만들기는 역파일이 이기고, 지연은 그래프가 이기고, 재현율은 전부 비교가 이긴다. 어느 것이 이기느냐는 무엇을 재느냐다.

예제 13.4: 점 5만 개, 재현율 90 퍼센트

전부 비교는 12.8 MB 에 거리 5만 번이다. 역파일은 nprobe 몇 개에 거리 천 번 남짓이고 메모리는 같다. 곱 양자화를 얹으면 메모리가 20 분의 1 이다. 그래프는 이웃 목록이 메모리를 더 쓰고 삽입 5만 번이 만들기다.

왼쪽이 메모리가 작은 쪽. 그래프가 지연은 가장 짧고 메모리는 가장 크다

그림 13.6.. 네 인덱스의 자리. 가로가 메모리, 세로가 지연. 재현율 목표를 넘는 설정에서 각 인덱스가 놓이는 곳

예제의 자리가 실측에서 어디인지 잰다. 목표는 재현율 90 퍼센트이고, 인덱스마다 설정을 늘려 가며 목표를 처음 넘는 점을 적는다.

균열

군집 200 개의 64 차원 점 5만 개에서 네 인덱스와 전부 비교를 세우고, 재현율 @10 이 90 퍼센트를 처음 넘는 설정에서 메모리와 만들기와 질의 지연을 쟀다.

실험 13.4 재현율 90 퍼센트를 넘는 가장 싼 설정

구조	설정	재현율 퍼센트	MB	만들기 초	질의 ms
전부 비교	없음	100	12.8	0.00	11.3
역파일	nprobe 2	94.0	13.2	2.13	0.27
곱 양자화	부분 8	6.40	0.40	6.26	15.3
역파일 + PQ + 다시 재기	nprobe 4	97.2	0.87	16.7	1.06
근접 그래프	ef 40	93.0	17.7	188	3.84

측정 조건. 질의 100 개. 역파일은 군집 512, 곱 양자화는 부분 8 과 다시 재기 100, 그래프는 이웃 16. 시간은 파이썬과 NumPy 의 값이라 비율만 읽는다.

역파일이 nprobe 2 로 94 퍼센트를 넘고 0.27 밀리초다. 전부 비교의 40 분의 1 이고 메모리는 같다. 곱 양자화만으로는 6 퍼센트라 목표에 못 미친다. 역파일 위의 곱 양자화에 다시 재기를 얹으면 97 퍼센트에 0.87 MB 다. 메모리 15 분의 1 이고 지연은 역파일의 네 배다. 그래프는 ef 40 에서 93 퍼센트, 메모리가 가장 크고 만들기가 188 초로 백 배다.

관찰

이 데이터에서는 역파일이 이겼다. 군집이 뚜렷해 nprobe 2 로 충분했고, 파이썬의 그래프 탐색이 NumPy 의 배열 훑기를 못 이겼다. 그래프의 지연 3.8 밀리초는 파이썬 힙의 값이다. C 로 쓰면 그래프가 역파일보다 빠른 것이 보통이고, 원 논문들의 보고가 그렇다.

이 관찰이 이 절의 요점을 두 번 말한다. 첫째, 데이터가 답을 정한다. 군집이 뚜렷하면 역파일이고 아니면 그래프다. 둘째, 언어가 답을 바꾼다. 같은 알고리즘이 구현에 따라 순위가 바뀐다. 그래서 이 표는 이 데이터와 이 구현의 표이고, 자기 데이터에서 다시 재는 절차가 표보다 값지다.

다시 쓰기

이 절의 다시 쓰기는 표를 절차로 바꾸는 것이다.

명제 13.14: 고르는 절차

첫째, 정확한 답을 전부 비교로 구해 둔다. 둘째, 재현율 목표를 정한다. 셋째, 메모리 예산이 원 벡터를 못 담으면 곱 양자화이고, 담으면 역파일과 그래프 둘 다 후보다. 넷째, 갱신이 잦으면 그래프이고 정적이면 둘 다다. 다섯째, 남은 후보의 재현율 곡선을 그려 목표에서 지연이 짧은 쪽을 고른다. 순서가 중요하다. 메모리와 갱신은 후보를 줄이고, 지연은 남은 것 중에서 고른다.

증명. 절차의 정당성은 명제 13.13 의 자리에서 나온다. 메모리 예산은 곱 양자화 여부를 정하고 (다른 셋은 원 벡터를 둔다), 갱신은 역파일을 뺀다 (다시 만들어야 한다), 나머지가 지연이다. ■

이 절차의 셋째 단계가 13장의 존재 이유이고 넷째가 12장의 존재 이유다. 벡터 십억 개는 원 벡터를 못 담으므로 양자화이고, 그 위에 그래프를 얹는 것이 12.4 였다.

알고리즘 13.4 재현율 곡선에서 고르기 `dsai.select.recall_curve`

> **입력 :** 인덱스 I, 설정 열 $\theta_1 < \theta_2 < \dots$, 질의와 정확한 답, 목표 ρ
> 1: **모든** θ_i 순서대로 **동안** 재현율을 재고 $\geq \rho$ 이면 θ_i 와 재현율을 **반환**
> 2: **끝 반복**
> 3: **반환** 없음과 마지막 재현율 (목표에 못 미침)

구현

구현이 절차 그 자체다. 인덱스마다 설정을 늘려 가며 목표를 넘는 첫 점을 적고, 실험 스크립트는 그 설정의 지연을 잰다.

지금 보는 코드 설정 열을 차례로 시도해 재현율이 목표를 넘는 첫 설정과 그 재현율을 돌려준다.

구현 재현율 곡선에서 고르기 `samples/dsai/select.py:21`

```
def recall_curve(fn: Callable, params: list, qs: np.ndarray, truth: list[set],
                 k: int, target: float):
    """알고리즘 13.4. 설정을 순서대로 시도해 재현율이 목표를 넘는 첫 설정과 그 재현율.
    못 넘으면 (None, 마지막 재현율).
    """
    rec = 0.0
    for p in params:
        rec = recall_at_k(fn, p, qs, truth, k)
        if rec >= target:
            return p, rec
    return None, rec
```

읽어 보면 설정 열이 오름차순이라 첫 통과가 가장 싼 설정이다. 목표를 못 넘으면 None 과 마지막 재현율을 돌려주고, 표에 목표 미달로 남는다. 곱 양자화만의 줄이 그것이다.

분석

인덱스 고르기는 자료 구조가 아니라 절차다. 그런데 이 책이 이것을 절 하나로 둔 것은 17장의 결론이 여기서 처음 나오기 때문이다. 재지 않고 고르지 않는다. 네 인덱스의 정리는 자리를 말하고, 자리의 경계는 데이터와 구현이 정한다. 경계를 아는 유일한 방법이 같은 데이터에서 재는 것이다.

이 절차가 서는 자리는 벡터 검색을 쓰는 모든 시스템이다. 검색 증강 생성의 문서 저장소, 추천의 항목 인덱스, 중복 제거의 후보 찾기. 셋 다 재현율 목표와 메모리와 지연이 다르고, 셋 다 같은 넷 중에서 고른다.

현장에서

벡터 라이브러리들은 이 절의 넷을 이름 하나씩으로 제공하고, 재현율 곡선을 그리는 도구를 함께 둔다 [확립] [4]. 공개 벤치마크들은 데이터마다 곡선을 그려 순위가 데이터에 따라 바뀌는 것을 보인다 [확립]. 곡선 없이 인덱스를 고른 시스템은 대개 nprobe 나 ef 가 기본값에 있고, 그것은 누군가의 다른 데이터에서 고른 값이다.

주의

실험 13.4 의 시간은 파이썬과 NumPy 가 섞여 있어 그래프가 불리하다. 표의 결론을 그래프가 느리다로 읽으면 안 된다. 읽을 것은 같은 데이터에서 순위가 정해진다는 것과, 메모리와 만들기의 차이가 크다는 것이다.

작업량	고르는 것과 이유
벡터 백만 개 이하, 메모리 넉넉, 갱신 잦음	근접 그래프. 삽입이 되고 지연이 짧다
벡터 백만 개 이하, 정적, 군집이 뚜렷	역파일. 만들기가 싸고 nprobe 몇 개로 충분하다
벡터 수억 개, 메모리에 원 벡터를 못 둠	역파일 위의 곱 양자화. 코드로 거르고 다시 잰다
벡터 수십억 개, 기계 한 대	디스크 그래프. 거친 코드는 메모리에, 벡터와 이웃은 페이지에
재현율 1 이 필요	전부 비교. 배열 한 번 훑기이고, 배치로 하면 행렬 곱이다

위 표가 12장과 13장의 요약이다. 줄마다 근거가 두 장의 명제 하나씩이고, 경계의 숫자는 자기 데이터에서 잰다.

되짚기

재현율 목표에서 가장 싼 설정을 찾는 절차를 정의하고, 네 인덱스를 같은 데이터에서 쟀다. 이 데이터에서는 역파일이 지연을, 곱 양자화가 메모리를, 전부 비교가 재현율을 이겼다. 데이터와 구현이 답을 정한다. 재지 않고 고르지 않는다.

13장을 닫으며

이 장이 세운 구조는 셋이고 절차가 하나다. k-평균의 역파일, 부분 공간의 곱 양자화, 둘을 잔차로 이은 인덱스, 그리고 재현율 목표에서 고르는 절차. 셋 다 정확한 답 대신 재현율을 받아들이고, 셋 다 순차 읽기를 만든다. 역파일의 군집 순서와 코드의 연속 구간이 그것이다.

네 실험의 숫자를 적어 둔다. 역파일은 nprobe 4 에서 점의 0.4 퍼센트를 보고 재현율 99 퍼센트였다. 곱 양자화는 8 바이트에서 메모리 32 분의 1 이고 코드만의 재현율은 낮았다. 잔차의 오차가 원 벡터의 4 분의 1 이었고, 다시 재기 100 개가 재현율을 40 에서 96 퍼센트로 되돌렸다. 재현율 90 퍼센트 목표에서 역파일이 지연을, 곱 양자화가 메모리 15 분의 1 을, 그래프가 만들기 백 배를 냈다.

정정 하나를 남긴다. 곱 양자화의 재현율을 처음 봤을 때 코드북 학습이 잘못된 줄 알았다. 오차 2.8 퍼센트에 재현율 40 퍼센트가 안 맞아 보였다. 상위 10 개 사이의 거리 차이를 재 보니 오차보다 작았다. 코드는 후보를 맞게 모으고 순서를 틀리고 있었다. 다시 재기가 그것을 고쳤고, 나중에 원 논문이 처음부터 다시 재기를 두었다는 것을 확인했다.

옛 구조의 판정

> 양자화는 그대로 유효하다. Lloyd 의 1982 년 반복과 1977 년 이래의 벡터 양자화가 그대로이고, 부분 공간과 잔차가 더해졌다. 근거는 실험 13.2 와 13.3 이다. 바이트를 두 배로 하면 오차가 반이고, 잔차가 오차를 4 분의 1 로 줄였다. 바뀐 것은 목적이다. 신호를 압축하던 것이 거리를 어림하는 표현이 됐고, 그래서 복원하지 않는다.

이 장의 요약

> 1. 역파일은 CSR 이고 벡터를 군집 순서로 놓는다. nprobe 가 재현율과 견준 수를 맞바꾼다 13.1
> 2. 곱 양자화는 부분마다 코드북 256 개다. 코드는 m 바이트이고 거리는 표 읽기 m 번이다 13.2
> 3. 잔차를 양자화하면 같은 바이트로 오차가 작다. 분산의 분해다 13.3
> 4. nprobe 는 후보를, 다시 재기는 순서를 고친다. 둘은 다른 손실이다 13.3
> 5. 인덱스는 같은 데이터에서 재현율 목표를 넘는 가장 싼 설정으로 고른다. 데이터와 구현이 순위를 정한다 13.4
>
> 다음 장에서 뒤집히는 것: 이 장까지 벡터는 검색의 대상이었다. 14 장의 키는 문자열이고 열의 모든 부분이 키다. 배열 하나가 모든 부분 문자열을 찾는다.

필자의 판단

> 필자는 벡터 인덱스를 가르칠 때 마지막 절의 표보다 그 앞의 실험 절차를 남기려 한다. 표는 이 데이터의 것이고, 절차는 어느 데이터에서든 같다. 학생이 자기 데이터에서 곡선을 한 번 그려 보면 인덱스의 이름을 외울 필요가 없어진다.

열린 문제

> 하나, 실험 13.4 의 그래프 지연은 파이썬 힙의 값이라 역파일과의 순위가 실제와 다를 수 있다. C 로 둘을 견주지 않았다.
> 둘, 곱 양자화의 부분을 좌표 순서대로 잘랐다. 회전을 먼저 하는 변형이 오차를 얼마나 줄이는

지 재지 않았다.
셋, 이 장의 인덱스는 전부 정적이다. 벡터를 더하고 지우는 값을 넷에서 견주지 않았다.

연습문제

13.1 기초 점 10^8 개, 차원 128 에서 역파일의 군집 수를 명제 13.3 로 정하고, nprobe 8 에서 질의 하나의 거리 계산 수를 구하라. 전부 비교의 몇 분의 1 인가.

13.2 기초 128 차원을 부분 16 개로 양자화할 때 코드북의 실수 개수와 코드의 바이트 수를 정의 13.5 로 구하고, 10^8 개의 메모리를 원 벡터와 견줘라.

13.3 기초 군집 4,096, nprobe 16, 부분 16, 다시 재기 200 에서 질의 하나의 표 읽기 수와 원 벡터 읽기 바이트를 명제 13.10 로 구하라. 어느 항이 가장 큰가.

13.4 기초 명제 13.14 의 절차를 벡터 5 억 개, 메모리 64 GB, 갱신 없음, 재현율 95 퍼센트 목표에 적용해 후보를 줄이고 이유를 적어라.

13.5 응용 `dsai.ivf.kmeans` 에 k-평균++ 초기화를 더하라. 첫 중심을 무작위로, 다음은 가장 가까운 중심까지의 거리 제곱에 비례해 뽑는다. 같은 반복 수에서 목적 함수가 얼마나 작아지는지 재라.

13.6 응용 `dsai.pq.PQ` 에 무작위 직교 회전을 먼저 하는 옵션을 더하고, 실험 13.2 의 재구성 오차가 부분 8 에서 얼마나 바뀌는지 재라.

13.7 응용 `dsai.ivfpq.IVFPQ.search` 의 군집별 거리표를 nprobe 개 한꺼번에 행렬 곱으로 만들게 바꾸고 시간을 견줘라.

13.8 응용 `samples/bench/c13_select.py` 에 재현율 목표 80, 95, 99 를 더해 목표마다 이기는 인덱스가 바뀌는지 재라.

13.9 탐구 비대칭 거리에서 질의도 양자화하면 (대칭 거리) 표가 질의와 무관해 미리 만들 수 있다. 무엇을 얻고 무엇을 잃는지 명제 13.6 로 논하라.

13.10 탐구 역파일 위의 곱 양자화와 12 장의 디스크 그래프를 벡터 십억 개에서 견줘라. 메모리, 접근 수, 만들기, 갱신을 각각 어느 명제로 적을 수 있는가.

더 읽을 것

[53] k-평균 반복의 원전. 1957 년의 기술 보고가 1982 년에 실렸고, 배정과 갱신이 목적 함수를 안 늘린다는 것이 여기 있다. 13.1
[52] 곱 양자화와 비대칭 거리, 역파일 위의 잔차 양자화까지 한 논문에 있다. 13 장의 원전이다. 13.2, 13.3
[4] 같은 인덱스를 가속기에 올린 논문. 코드 훑기와 선택이 벡터 명령으로 어떻게 되는지가 있다. 13.3, 13.4
[54] 내적 검색에서 오차의 방향을 나눈 양자화. 재현율이 오르는 이유가 손실 함수에 있다. 13.2

6부

시퀀스와 메모리

시퀀스는 순서가 있는 긴 데이터다. 6부는 말뭉치 전체를 한 배열로 색인하는 접미사 배열과, 어텐션이 만드는 상태를 자료 구조로 읽는 일을 다룬다. 길이가 자라도 상태가 자라지 않게 하는 방법이 이 부의 끝이다.

14장 문자열 인덱스와 말뭉치 15장 어텐션을 자료 구조로 읽기

14장

문자열 인덱스와 말뭉치

1993년의 논문 하나는 접미사 트리를 버리고 접미사의 시작 위치를 정렬한 정수 배열 하나를 두었다 [확립] [55]. 메모리가 셋에서 다섯 배 줄었고 찾기는 그대로였다. 그 배열이 지금 토큰 수조 개의 말뭉치 위에 서 있다. 어떤 문맥이 몇 번 나왔는지, 그 뒤에 무엇이 왔는지, 어느 부분이 복사본인지를 배열 하나가 답한다. 이 장은 그 배열을 세우고, 세 물음을 차례로 푼다. 세 절이 전부 한 배열 위에 있다.

이 장은 0.2 의 둘째 가정을 9장에 이어 다시 본다. 키가 열이고 열의 모든 부분이 키다. 트리로 두면 링크가 제곱이고 배열로 두면 정수 n 개다. 그리고 7.4 의 문서 단위 중복 제거가 여기서 부분 문자열 단위로 자란다.

이 장에서 세우는 것

14.1 접미사 배열. 정렬 로그 번으로 만들고 이진 탐색 둘로 센다. 순차 훑기보다 여섯 배에서 스무 배

14.2 무한 n-그램. n 을 정하지 않는 개수 세기. 표보다 맞히고 작다

14.3 긴 중복 찾기. LCP 한 번 훑기로 반복 전부. 소스 코드의 29 퍼센트가 50 바이트 이상의 반복

14.1 접미사 배열

9장의 트라이는 어휘 항목의 접두사를 나눴다. 텍스트 전체의 모든 부분 문자열을 찾으려면 텍스트의 모든 접미사를 넣어야 하고, 트라이로 두면 노드가 텍스트 길이의 제곱까지 간다. 접미사 트리가 그것을 선형으로 줄이지만 노드가 링크로 흩어져 있다. 1993년에 Manber와 Myers가 트리를 버리고 접미사의 시작 위치를 정렬한 배열 하나를 두었다 [확립] [55]. 정수 n 개다. 패턴 찾기는 그 위의 이진 탐색이고, 8.4의 정렬된 배열이 문자열에서 되살아난 것이다.

이 절은 접미사 배열을 정의하고, 만들기를 배가 정렬로 세우고, 패턴의 개수 세기가 이진 탐색 둘임을 보인 뒤, 소스 코드 백만 바이트에서 잰다.

학습 목표

> 접미사 배열을 정의하고, 같은 접두사를 가진 접미사들이 연속 구간을 이룸을 보일 수 있다. 배가 정렬이 $O(n \log^2 n)$ 에 배열을 만드는 이유와, 패턴 세기가 $O(|P| \log n)$ 인 이유를 설명하고 실측으로 말할 수 있다.

이 말부터

접미사	위치 i 부터 끝까지의 부분 문자열 $T[i..n)$. 접미사가 n 개다
접미사 배열	접미사를 사전 순으로 정렬했을 때의 시작 위치 배열 SA
LCP 배열	정렬 순서로 이웃한 두 접미사의 공통 접두사 길이. 다음 두 절이 쓴다

디딤돌

디딤돌은 9장의 트라이다. 접미사 전부를 트라이에 넣으면 부분 문자열 찾기가 경로 따라 가기가 된다. 그 트라이를 배열 하나로 접은 것이 접미사 배열이다.

정의 14.1: 접미사 배열

> 텍스트 $T[0..n)$ 에 대해 SA는 $\{0, \ldots, n-1\}$ 의 순열로, $T[\mathrm{SA}[0]..) < T[\mathrm{SA}[1]..) < \ldots$ 이다.
>
> **불변식** 패턴 P 로 시작하는 접미사들은 SA에서 연속 구간 $[\mathrm{lo}, \mathrm{hi})$ 를 이룬다. 구간의 길이가 P 가 나오는 횟수이고, $\mathrm{SA}[\mathrm{lo}..\mathrm{hi})$ 가 나오는 위치 전부다.

불변식이 이 구조의 전부다. 정렬된 것 위에서 같은 접두사는 붙어 있다. 그래서 찾기가 구간 찾기이고, 구간 찾기가 이진 탐색이다.

명제 14.2: 패턴 세기

> 길이 m 의 패턴 P 의 구간 $[\mathrm{lo}, \mathrm{hi})$ 는 이진 탐색 둘로 찾는다. 하나는 P 이상인 첫 접미사, 하나는 P 를 접두사로 갖는 마지막 접미사의 다음. 비교 하나가 $O(m)$ 이므로 $O(m \log n)$ 이다. 이동량은 비교마다 접미사의 앞 m 바이트를 읽는 것이라 라인 $\log n$ 개 남짓이고, 배열 자체는 정수 n 개다.

증명. 정의 14.1 의 불변식과 8.4 의 이진 탐색이다. 정렬된 열에서 어떤 조건 (패턴보다 작다, 패턴을 접두사로 갖거나 작다) 이 참에서 거짓으로 한 번 바뀌면 그 경계를 로그에 찾는다. ■

명제 14.3: 배가 정렬

> 길이 2^k 접두사의 순위를 알면 길이 2^{k+1} 접두사의 순위는 (앞 절반 순위, 뒤 절반 순위) 의 쌍을 정렬해 얻는다. 뒤 절반의 순위는 위치 $i + 2^k$ 의 순위다. 순위가 전부 다르면 정렬이 끝나고, 그때까지 $\lceil \log_2 n \rceil$ 번 이하다. 정렬 하나가 $O(n \log n)$ 이라 전부 $O(n \log^2 n)$ 이다.

증명. 길이 2^{k+1} 접두사의 사전 순서는 앞 2^k 의 순서가 먼저이고 같으면 뒤 2^k 의 순서이므로, 두 순위의 쌍이 그 순서를 정한다. 뒤 절반은 위치 $i + 2^k$ 에서 시작하는 접두사이므로 그 순위가 이미 있다. ■

명제의 정렬이 6.4 의 정렬 기반 세기와 같은 도구다. 트리를 세우는 대신 정렬을 로그 번 한다. 배열 하나에 정수 n 개이고 링크가 없다.

예제 14.1: 텍스트 백만 바이트

> 접미사 배열은 8 바이트 정수 백만 개, 8 MB 다. 배가 정렬은 스무 번 이하의 정렬이고, 정렬 하나가 백만 개다. 패턴 하나를 세는 것은 이진 탐색 둘, 비교 마흔 번, 비교마다 바이트 몇 개다.

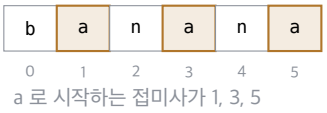


그림 14.1.. 텍스트 배열. 접미사는 복사하지 않고 시작 위치로만 가리킨다. 정렬하는 것은 위치의 배열이다

그림 14.2.. banana 의 접미사 배열. 접미사를 정렬하면 시작 위치가 5, 3, 1, 0, 4, 2 이고, ana 로 시작하는 둘이 붙어 있다

예제의 비교 마흔 번이 순차 훑기와 얼마나 갈리는지, 그리고 만드는 값이 얼마인지 잰다.

균열

소스 코드 백만 바이트의 접미사 배열을 배가 정렬로 만들고, 패턴 다섯 개의 개수를 접미사 배열과 C 로 짜인 순차 훑기로 쟀다.

실험 14.1 패턴 세기. 접미사 배열과 순차 훑기

패턴	길이	개수	접미사 배열 us	순차 훑기 us
def	4	1589	119	2225
return	6	2000	140	952
self.	5	3952	127	950
import numpy	12	0	153	823
raise ValueError(	17	158	171	696

측정 조건. 파이썬 표준 라이브러리 소스 백만 바이트, 바이트가 토큰. 만들기 4.6 초, 배열 8 MB. 순차 훑기는 `bytes.count` 이고 C 다. 다섯 번 중앙값.

접미사 배열이 120 에서 170 마이크로초, 순차 훑기가 700 에서 2,200 마이크로초다. 파이썬 이진 탐색이 C 순차 훑기보다 여섯 배에서 스무 배 빠르다. 순차 훑기는 패턴이 짧을수록 느리다. 백만 바이트를 다 보는 것은 같은데 짧은 패턴이 더 자주 부분 일치하기 때문이다. 접미사 배열은 패턴 길이에 거의 무관하다.

관찰

만드는 데 4.6 초, 묻는 데 0.1 밀리초다. 물음이 오만 번을 넘으면 만드는 값이 갚아진다. 말뭉치 위의 물음은 그보다 훨씬 많다. 그리고 만든 뒤에는 텍스트가 안 바뀐다. 8장과 같은 조건이고, 그래서 배열이다.

이 관찰이 접미사 배열을 학습 말뭉치에 쓰는 이유다. 말뭉치는 한 번 만들어지고, 그 위에서 개수를 묻는 일이 셀 수 없이 많다.

다시 쓰기

이 절의 다시 쓰기는 배열을 이웃한 접미사의 공통 접두사 길이로 보강하는 것이다. 다음 두 절이 그 배열 위에 선다.

명제 14.4: LCP 배열

> LCP[i] 를 T[SA[$i-1$]..) 와 T[SA[i]..) 의 공통 접두사 길이라 하자. 정렬 순서로 이웃하지 않은 두 접미사의 공통 접두사 길이는 그 사이 LCP 의 최소다. Kasai 등의 방법으로 LCP 를 $O(n)$ 에 만든다. 위치 i 의 접미사에서 공통 접두사 h 를 찾았으면 위치 $i+1$ 의 접미사는 적어도 $h-1$ 을 갖는다.

증명. 첫 문장은 정렬의 성질이다. 사이의 어느 접미사도 두 접미사 사이에 있으므로 공통 접두사가 그중 가장 짧은 것 이상이고 이하다. 둘째는 접미사 i 의 첫 글자를 뗀 것이 접미사 $i+1$ 이라는 것이다. i 가 정렬 순서의 이웃과 h 를 나누면 $i+1$ 은 그 이웃의 다음 위치와 $h-1$ 을 나누고, 정렬 순서의 이웃은 그보다 가깝다. h 가 n 번 오르고 n 번 내리므로 선형이다. ■

이 명제가 다음 절의 개수 세기와 그다음 절의 중복 찾기를 준다. 같은 접두사가 붙어 있고, 얼마나 같은지가 LCP 에 있다.

알고리즘 14.1 배가 정렬 `dsai.suffix.build_suffix_array`

> **입력:** 텍스트 T
> 1: 순위 ← 글자의 조밀한 순위; $k \leftarrow 1$
> 2: **조건** 순위가 전부 다르지 않음 **동안**
> 3: 키[i] ← (순위[i], 순위[$i+k$] 또는 -1); SA ← 키로 정렬한 위치
> 4: 순위 ← 정렬된 키가 바뀌는 자리의 누적 개수; $k \leftarrow 2k$
> 5: **끝 조건**

구현

배가 정렬은 NumPy 정렬 로그 번이다. 구간 찾기는 파이썬 이진 탐색 둘이다.

지금 보는 코드 앞 순위와 뒤 순위를 하나의 키로 묶어 정렬하고, 키가 바뀌는 자리를 세어 새 순위를 만든다.

구현 배가 정렬 `samples/dsai/suffix.py:13`

```python
def build_suffix_array(text: np.ndarray) -> np.ndarray:
    """알고리즘 14.1. 배가 정렬. 길이 2^k 접두사의 순위로 정렬을 되풀이한다.
    O(n log^2 n)."""
    n = len(text)
    _, rank = np.unique(text, return_inverse=True)      # 조밀한 순위 0..r 로 시작
    rank = rank.astype(np.int64)
    k = 1
    while True:
        second = np.full(n, -1, dtype=np.int64)
        second[: n - k] = rank[k:]
        key = rank * (n + 1) + (second + 1)          # (앞 순위, 뒤 순위) 를 하나의 키로
        sa = np.argsort(key, kind="stable")
        sorted_key = key[sa]
        new_rank = np.zeros(n, dtype=np.int64)
        new_rank[sa[1:]] = np.cumsum(sorted_key[1:] != sorted_key[:-1])
        rank = new_rank
        if rank.max() == n - 1:
            break
        k *= 2
    return sa
```

읽어 보면 `key = rank * (n + 1) + (second + 1)` 이 두 순위를 하나의 정수로 묶는다. 순위가 n 미만이라 겹치지 않는다. `np.cumsum(sorted_key[1:] != sorted_key[:-1])` 이 새 순위다. 처음에 글자를 조밀한 순위로 바꾸지 않으면 이 묶음이 넘친다.

구현 구간 찾기 `samples/dsai/suffix.py:60`

```python
def find_range(text: np.ndarray, sa: np.ndarray,
               pattern: np.ndarray) -> tuple[int, int]:
    """알고리즘 14.2. 패턴으로 시작하는 접미사의 구간 [lo, hi). O(|P| log n)."""
    n, m = len(text), len(pattern)
    pat = pattern.tolist()

    def less(i: int, strict: bool) -> bool:
        """접미사 sa[i] 의 앞 m 글자가 패턴보다 작은가. strict 면 같아도 작다."""
        s = text[sa[i]: sa[i] + m].tolist()
        for a, b in zip(s, pat):
            if a != b:
                return a < b
        return len(s) < m or strict

    def bound(strict: bool) -> int:
        lo, hi = 0, n
        while lo < hi:
            mid = (lo + hi) // 2
            if less(mid, strict):
                lo = mid + 1
            else:
                hi = mid
        return lo

    return bound(False), bound(True)
```

읽어 보면 `less` 가 접미사와 패턴의 비교이고, `strict` 가 두 경계의 차이다. 거짓이면 패턴 이상인 첫 자리, 참이면 패턴을 접두사로 갖는 것의 다음 자리다. 둘의 차이가 개수다.

분석

접미사 배열은 그대로 유효하다. 1993년의 정의와 이진 탐색이 그대로이고, 배열 하나에 정수 n 개라는 것이 이 책의 잣대에 딱 맞는다. 바뀐 것은 텍스트의 크기다. 논문의 텍스트는 메가바이트였고 지금은 테라바이트다. 배열이 텍스트의 여덟 배라 그 크기가 문제이고, 그것이 다음 절의 자리다.

이 구조가 서는 자리는 학습 말뭉치 위의 물음이다. 이 문장이 말뭉치에 몇 번 나오는가, 이 문맥 뒤에 무엇이 왔는가, 어느 부분이 복사본인가. 셋 다 접미사 배열 하나로 답하고, 셋이 이 장의 세 절이다.

현장에서

> Manber 와 Myers 는 접미사 배열이 접미사 트리보다 메모리를 셋에서 다섯 배 덜 쓰고 찾기가 $O(P + \log n)$ 임을 보였다 [확립] [55]. Lee 등의 중복 제거는 이 배열 위에서 50 토큰 이상의 정확한 중복을 찾았고 [확립] [29], 그것이 이 장의 셋째 절이다.

주의

> 실험 14.1 의 배가 정렬은 $O(n \log^2 n)$ 이고 선형 시간 알고리즘이 따로 있다. 텍스트가 테라바이트이면 그 차이가 하루와 몇 주다. 이 책의 구현은 백만 바이트에서 4.6 초이고 그 위로는 다른 도구가 필요하다.

배열 하나가 모든 부분 문자열의 개수를 낸다. 다음 절은 그 개수로 다음 토큰을 예측한다. 문맥 길이 n 을 미리 정하지 않고 말뭉치가 허락하는 만큼 길게 쓰는 n-그램이다.

되짚기

> 접미사 배열을 정의하고 같은 접두사가 연속 구간임을 보였다. 배가 정렬로 백만 바이트를 4.6 초에 만들고, 패턴 세기가 순차 훑기보다 여섯 배에서 스무 배 빨랐다. 다음 절은 무한 n-그램이다.

14.2 n-그램 세기. 접미사 배열 위의 무한 n-그램

n-그램 언어 모델은 앞 $n-1$ 토큰을 보고 다음 토큰의 빈도를 센다. 표는 문맥에서 다음 토큰의 개수로 가는 6.1 의 해시 테이블이고, n 이 크면 표가 폭발하고 문맥이 안 나온다. 접미사 배열은 n 을 정하지 않는다. 어떤 문맥이든 정의 14.1 의 구간이 개수이고, 구간 안의 다음 토큰이 분포다. 문맥이 없으면 짧게 줄인다. Liu 등이 2024 년에 이 방법으로 토큰 수조 개의 말뭉치 위에 n 이 없는 n-그램 모델을 세웠다 [확립] [56].

이 절은 n-그램 표를 디딤돌로 세우고, 접미사 배열의 무한 n-그램을 정의한 뒤, 다음 바이트 맞히기와 메모리로 둘을 견준다.

학습 목표

n-그램 표의 메모리가 n 에 대해 왜 폭발하는지 적고, 접미사 배열이 모든 n 의 개수를 하나의 배열로 내는 이유를 불변식으로 설명할 수 있다. 백오프가 가장 긴 있는 문맥을 고르는 것임을 알고리즘으로 적고, 맞힌 비율과 메모리를 실측으로 말할 수 있다.

이 말부터

n-그램 표	길이 $n-1$ 문맥에서 다음 토큰의 개수로 가는 표. n 마다 따로다
무한 n-그램	문맥 길이를 정하지 않고 접미사 배열의 구간으로 개수를 내는 모델
백오프	긴 문맥이 말뭉치에 없을 때 뒤에서부터 줄여 있는 문맥을 쓰는 규칙

디딤돌

디딤돌은 6장의 해시 표다. 문맥을 키로, 다음 토큰의 개수를 값으로 두면 예측이 조회 한 번이다. 문맥 길이를 정해야 한다는 것이 이 표의 한계이고, 그 한계를 앞 절의 배열이 푼다.

정의 14.5: n-그램 표와 무한 n-그램

토큰 열 T 에서 n-그램 표는 $(T[i..i+n-1)) \mapsto \mathrm{Counter}(T[i+n-1])$ 이다. 항목 수가 서로 다른 $(n-1)$-그램의 수이고, n 이 크면 거의 모든 위치가 새 항목이다. 무한 n-그램은 접미사 배열 SA 하나다. 문맥 c 의 개수는 정의 14.1 의 구간 길이이고, 다음 토큰의 분포는 구간의 각 위치 $\mathrm{SA}[i]+|c|$ 에 있는 토큰의 개수다.

불변식 어떤 길이의 문맥이든 구간 하나가 그 문맥의 모든 출현이다. 문맥을 한 토큰 늘리면 구간은 그 안의 부분 구간이다.

불변식의 둘째 문장이 백오프의 근거다. 긴 문맥의 구간이 비면 짧은 문맥의 구간은 안 비어 있을 수 있고, 그것이 상위 구간이다.

명제 14.6: n-그램 표의 크기

> 말뭉치 길이 N, 어휘 V 에서 n-그램 표의 항목 수는 $\min(N, V^{n-1})$ 이하이고, 실제 말뭉치에서는 n 이 5 를 넘으면 N 에 가깝다. 거의 모든 문맥이 한 번 나온다. 무한 n-그램의 메모리는 SA 와 텍스트로 N 에 비례하고 n 과 무관하다. 문맥 하나의 물음은 명제 14.2 의 $O(|c| \log N)$ 이고, 백오프는 그것을 길이마다 되풀이한다.

증명. 표의 항목은 서로 다른 문맥이고 위치마다 하나씩이므로 N 이하다. 긴 문맥이 되풀이될 확률은 길이에 대해 급히 줄어 대부분 한 번이다. 무한 n-그램은 개수를 저장하지 않고 매번 세므로 n 이 안 든다. ■

명제가 말하는 것은 표와 배열의 맞바꿈이다. 표는 개수를 미리 세어 두어 물음이 해시 하나이고 메모리가 n 마다 N 이다. 배열은 물음이 이진 탐색 $\log N$ 이고 메모리가 N 하나다. n 을 여럿 두어야 하면 배열이 이긴다. 백오프가 n 을 여럿 두는 것이다.

예제 14.2: 소스 코드 40 만 바이트

> 5-그램 표는 문맥 4 바이트마다 다음 바이트의 개수이고, 항목이 수십만 개다. 8-그램은 거의 모든 위치가 새 항목이라 표가 텍스트의 백 배 남짓이고, 새 문맥의 30 퍼센트가 표에 없다. 접미사 배열은 3.2 MB 에 텍스트 3.2 MB 이고, 어떤 길이의 문맥이든 있으면 찾는다.

그림 14.3.. 정렬된 접미사에서 문맥 ana 의 구간. 구간 안의 접미사가 그 문맥의 출현이고, 다음 토큰을 세면 예측이다

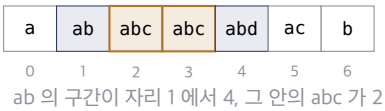


그림 14.4.. 같은 배열 위의 세 문맥. 긴 문맥의 구간이 짧은 문맥의 구간 안에 든다. 비면 한 토큰 줄인다

예제의 30 퍼센트가 맞힌 비율에 어떻게 닿는지, 그리고 백오프가 그것을 얼마나 되돌리는지 잰다.

균열

소스 코드 40만 바이트의 앞 90 퍼센트로 n-그램 표 셋과 접미사 배열을 만들고, 뒤 10 퍼센트에서 다음 바이트 2,000 개를 맞혔다.

실험 14.2 다음 바이트 맞히기. n-그램 표와 무한 n-그램

방법	문맥	맞힘 퍼센트	없음 퍼센트	MB	예측 us
3-그램 표	3	53.0	0.15	2.08	6.78
5-그램 표	5	62.9	8.50	15.1	8.20
8-그램 표	8	51.8	29.8	42.8	7.21
무한 n-그램 (접미사 배열)	11.3	70.0	0.00	5.76	7535

측정 조건. 문맥 없음은 표에 문맥이 없어 답을 못 낸 비율. 무한 n-그램은 뒤 64 바이트부터 백오프하고, 문맥 길이 열은 실제로 쓴 길이의 평균. 예측 시간은 파이썬 이진 탐색이다.

3-그램이 53 퍼센트, 5-그램이 63 퍼센트, 8-그램이 52 퍼센트다. 8-그램은 문맥이 없는 경우가 30 퍼센트라 표가 크면서 더 못 맞힌다. 무한 n-그램은 70 퍼센트다. 쓴 문맥의 평균 길이가 11 이고, 메모리는 5-그램 표의 3 분의 1, 8-그램 표의 7 분의 1 이다.

관찰

문맥이 길수록 좋은데 표는 길이를 미리 정해야 하고, 정하면 없는 문맥이 는다. 배열은 정하지 않으므로 있는 만큼 긴 문맥을 쓴다. 평균 11 이 그것이다. 예측 시간은 배열이 천 배 느리다. 백오프가 길이마다 이진 탐색 둘을 하고, 그것이 파이썬이다.

이 관찰이 무한 n-그램의 두 값이다. 표보다 맞히고 작다. 대신 물음이 로그이고, 그것을 밀리초 아래로 내리는 것이 실제 구현의 일이다.

다시 쓰기

이 절의 다시 쓰기는 백오프를 구간의 포함으로 적는 것이다. 길이마다 이진 탐색을 새로 하지 않는다.

명제 14.7: 구간 좁히기

문맥 c 의 구간 $[\mathrm{lo}, \mathrm{hi})$ 를 알면 c 에 토큰 하나를 앞에 붙인 문맥의 구간은 그 밖이고, 뒤에 붙인 문맥 ct 의 구간은 그 안이다. 뒤에 붙이는 좁히기는 구간 안의 이진 탐색이라 $O(\log(\mathrm{hi} - \mathrm{lo}))$ 다. 백오프를 짧은 문맥에서 긴 문맥으로 늘려 가며 좁히면, 길이마다 $\log N$ 이 아니라 남은 구간의 로그다.

증명. 정의 14.5 의 둘째 불변식이다. 긴 문맥이 나오는 자리는 짧은 문맥이 나오는 자리의 부분 집합이고 정렬 순서에서 연속이다. ■

이 명제가 접미사 배열 위의 물음을 스트림으로 만든다. 문맥이 한 토큰씩 자라면 구간이 한 번씩 좁아지고, 구간이 비는 순간이 백오프의 답이다. 그리고 이 구간이 14.3 의 중복 찾기에서 다시 나온다. 구간이 둘 이상이면 그 문맥이 두 번 이상 나온 것이다.

알고리즘 14.2 백오프 예측 `dsai.ngram.InfiniGram.predict`

입력: 접미사 배열, 문맥 c (뒤 L 토큰), 최소 개수 μ
1: **반복** $k = L, L-1, \dots, 1$ **동안**
2: 구간 $\leftarrow$ 뒤 k 토큰의 구간; 구간 길이 $\geq \mu$ 이면 구간의 다음 토큰 중 최빈을 **반환**
3: **끝 반복**

구현

무한 n-그램은 배열 둘 (텍스트와 SA) 과 구간 찾기 하나다. 다음 토큰의 분포는 구간의 위치에서 읽는다.

지금 보는 코드 문맥의 구간을 찾고, 구간의 각 위치에서 문맥 길이만큼 뒤의 토큰을 센다.

구현 구간 안의 다음 토큰 `samples/dsai/ngram.py:51`

```
    def next_token_counts(self, context: list[int]) -> Counter:
        """알고리즘 14.3. 문맥의 구간 안에서 다음 토큰을 센다.
        구간이 크면 표본만 본다.
        """
        ctx = np.array(context, dtype=self.text.dtype)
        lo, hi = find_range(self.text, self.sa, ctx)
        if hi <= lo:
            return Counter()
        pos = self.sa[lo:hi] + len(context)
        pos = pos[pos < len(self.text)]
        if len(pos) > 4096:
            pos = pos[:: len(pos) // 4096]
        vals, cnt = np.unique(self.text[pos], return_counts=True)
        return Counter(dict(zip(vals.tolist(), cnt.tolist())))
```

읽어 보면 `self.sa[lo:hi] + len(context)` 가 다음 토큰의 위치들이다. `np.unique` 가 개수를 센다. 구간이 크면 표본만 보는 것이 `pos[:: len(pos) // 4096]` 이고, 개수의 비율만 필요하기 때문이다.

구현 백오프 `samples/dsai/ngram.py:66`

```
    def predict(self, context: list[int], min_count: int = 2) -> tuple[int | None, int]:
        """백오프. 가장 긴 문맥부터, 구간이 min_count 이상인 첫 문맥으로 예측. (토큰,
        쓴 문맥 길이)."""
        for k in range(min(len(context), 64), 0, -1):
            c = self.next_token_counts(context[-k:])
            if sum(c.values()) >= min_count:
                return c.most_common(1)[0][0], k
        return None, 0
```

읽어 보면 긴 문맥부터 줄여 가며 첫 문맥을 쓴다. 명제 14.7 의 좁히기는 반대 방향이고 연습문제다. `min_count` 가 한 번 나온 문맥을 믿지 않게 한다.

분석

n-그램 모델은 조정해서 유효하다. 개수로 다음 토큰을 예측하는 것은 그대로이고, 표가 배열로 바뀌었다. n 이 사라졌다. 이것은 6.2 의 어휘 사전과 반대 방향이다. 거기서는 문자열을 표에 넣었고, 여기서는 표를 없애고 텍스트 위에 배열 하나를 둔다.

이 구조가 서는 자리는 말뭉치 위의 통계다. 언어 모델이 낸 문장이 말뭉치에 있었는지, 어느 문맥 뒤에 무엇이 자주 왔는지. Liu 등은 이 물음을 토큰 수조 개에서 밀리초에 답했고, 그 값이 배열 하나였다 [확립] [56]. 신경망 모델의 확률에 이 개수를 섞으면 성적이 오른다는 것도 같은 논문의 보고다.

현장에서

> Liu 등의 구현은 접미사 배열을 디스크에 두고 이진 탐색의 각 단계가 페이지 하나이게 했다 [확립] [56]. 12.4 의 접근 세기가 여기서도 잣대다. $\log N$ 이 40 이면 접근 40 번이고, 그것이 밀리초다.

주의

> 실험 14.2 의 토큰은 바이트이고 어휘가 256 이다. 토큰 어휘가 수만이면 같은 길이의 문맥이 훨씬 드물어 백오프가 더 짧은 문맥에서 멈춘다. 맞힌 비율 70 퍼센트는 바이트 단위의 값이다.

문맥의 구간이 둘 이상이면 그 문맥이 말뭉치에 두 번 이상 나온 것이다. 그 사실 하나가 다음 절 전체다. 어느 부분이 어느 부분의 복사본인지를 배열이 이미 알고 있다.

되짚기

> n-그램 표와 무한 n-그램을 정의하고, 표의 크기가 n 에 폭발하는 이유와 배열이 n 과 무관한 이유를 보였다. 배열이 표보다 맞히고 작았다. 백오프는 구간의 포함으로 다시 쓴다. 다음 절은 중복 찾기다.

14.3 긴 중복 찾기. 말뭉치 안의 복사본

7.4 의 중복 제거는 문서 단위였다. 문서 둘이 거의 같으면 하나를 버렸다. 그런데 말뭉치의 중복은 문서 안에도 있다. 같은 안내문이 문서 수천 개의 한가운데에 있고, 같은 코드 조각이 파일 수백 개에 있다. 그것은 문서의 자카드로는 안 보인다. 문서는 다르고 조각이 같다. 조각을 찾으려면 모든 부분 문자열이 필요하고, 그것이 접미사 배열이다. Lee 등은 50 토큰 이상 정확히 같은 부분 문자열을 접미사 배열로 찾아 걸러 냈고, 그것이 문서 단위 제거와 별개의 효과를 냈다 [확립] [29].

이 절은 LCP 배열 한 번 훑기로 길이 L 이상의 모든 반복 구간을 찾고, 길이 문턱에 따라 중복이 덮는 비율을 잰다.

학습 목표

길이 L 이상의 반복 부분 문자열이 LCP 배열의 L 이상인 자리와 일대일임을 보일 수 있다. 반복 구간을 찾는 비용이 LCP 한 번 훑기임을 알고리즘으로 적고, 길이 문턱이 덮는 비율을 어떻게 움직이는지 실측으로 말할 수 있다.

이 말부터

반복 구간	텍스트의 서로 다른 두 위치에서 같은 길이 L 이상의 부분 문자열
덮는 비율	반복 구간에 든 위치의 수를 텍스트 길이로 나눈 것. 겹침은 한 번만 센다
가장 긴 반복	LCP 의 최대. 텍스트에서 두 번 이상 나오는 가장 긴 부분 문자열

디딤돌

디딤돌은 7장의 근사 중복 제거다. 거기서는 문서 쌍의 유사도를 어림했다. 여기서는 어림하지 않고, 길이 L 이상 정확히 같은 구간을 전부 찾는다. 그것을 정의부터 세운다.

정의 14.8: 반복 구간

길이 L 의 반복 구간은 $i \neq j$ 에 대해 $T[i..i+L) = T[j..j+L)$ 인 위치 i 다. 길이 L 이상의 반복 구간의 집합을 R_L 이라 하자.

불변식 $i \in R_L$ 이면 i 의 접미사는 정렬 순서에서 이웃한 접미사 중 하나와 L 이상을 나눈다. 즉 SA 에서 i 의 자리 r 에 대해 $\mathrm{LCP}[r] \geq L$ 또는 $\mathrm{LCP}[r+1] \geq L$ 이다.

정리 14.9: 반복 구간과 LCP

$R_L = \{\mathrm{SA}[r] : \mathrm{LCP}[r] \geq L\} \cup \{\mathrm{SA}[r-1] : \mathrm{LCP}[r] \geq L\}$ 이다. 따라서 R_L 은 LCP 를 한 번 훑어 $O(n)$ 에 얻는다.

증명. i 가 j 와 L 을 나누면 정렬 순서에서 i 와 j 사이의 모든 접미사도 L 을 나눈다 (명제 14.4 의 첫 문장). 그러면 i 의 이웃도 나눈다. 역은 정의다. ■

정리가 말하는 것은 반복을 찾는 데 비교가 없다는 것이다. 접미사 배열을 만들 때 다 비교했고, LCP 가 그 결과다. 남은 것은 문턱 하나로 배열을 거르는 것이다.

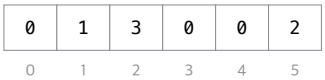
명제 14.10: 덮는 비율

> R_L 의 각 위치가 $[i, i+L)$ 을 덮는다. 덮인 위치의 합집합 크기를 n 으로 나눈 것이 덮는 비율이고, L 이 커질수록 단조 감소한다. 합집합은 R_L 을 정렬해 구간을 병합하면 $O(|R_L| \log |R_L|)$ 이고, SA 순서 대신 위치 순서가 필요해 정렬이 한 번 든다.

증명. $R_{L+1} \subseteq R_L$ 이므로 단조다. 병합은 6.4 의 정렬 위의 훑기다. ■

예제 14.3: 소스 코드 백만 바이트

> LCP 가 50 이상인 자리가 수만 개면 R_{50} 이 수만 위치이고, 각각이 50 바이트를 덮는다. 겹침이 많아 합집합은 그보다 작다. 가장 긴 반복이 천 바이트를 넘으면 그것은 파일 하나가 다른 파일에 통째로 복사된 것이다.



banana 의 LCP. 자리 2 의 3 이 ana 와 anana 의 공통 접두사이고, $L = 3$ 이면 위치 1 과 3 이 반복이다

그림 14.5.. LCP 배열과 반복. 정렬 순서로 이웃한 접미사 둘이 L 이상을 나누면 둘 다 반복 구간의 시작이다

예제의 수만 개와 합집합의 차이가 실측에서 얼마인지, 문턱을 바꾸며 잰다.

균열

소스 코드 백만 바이트의 LCP 를 만들고, 문턱 50, 100, 200, 400 바이트에서 반복 시작 위치의 수와 덮는 비율을 쟀다.

실험 14.3 길이 문턱에 따른 반복 구간과 덮는 비율

길이 문턱 바이트	중복 시작 위치 수	덮는 비율 퍼센트
50	129445	28.7
100	46922	11.2
200	16639	4.31
400	5039	1.26

측정 조건. LCP 배열 2.8 초. 가장 긴 반복은 1,208 바이트이고 위치 699,550 과 688,087 이다.

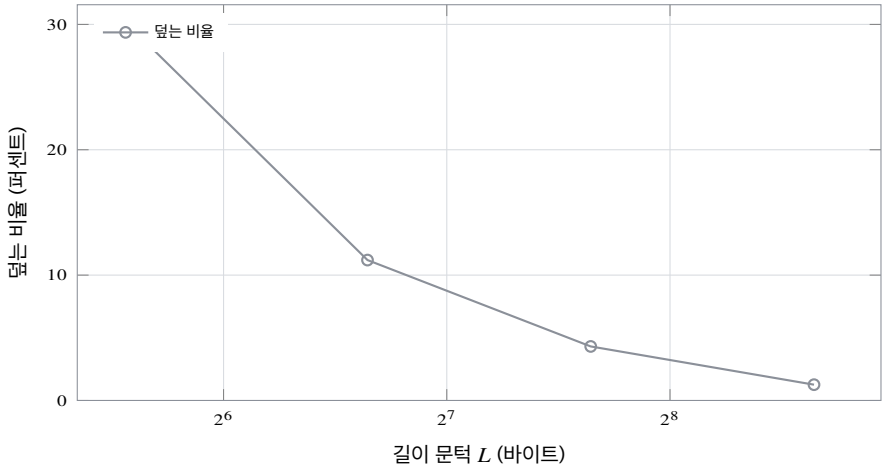


그림 14.6.. 길이 문턱에 따른 덮는 비율. 문턱이 두 배 오를 때마다 비율이 반 남짓으로 준다

50 바이트 이상의 반복이 텍스트의 29 퍼센트를 덮는다. 100 에서 11 퍼센트, 400 에서 1.3 퍼센트다. 소스 코드의 3 분의 1 이 50 바이트 이상 어딘가에 또 있는 문자열이다. 가장 긴 반복 1,208 바이트는 파일 둘이 같은 머리말을 갖는 것이다. 7.4 의 문서 단위 자카드는 이 두 파일을 다르다고 봤을 것이다.

관찰

> 문턱 50 에서 시작 위치가 13 만 개인데 덮는 비율은 29 퍼센트, 29 만 바이트다. 위치 하나가 평균 2 바이트를 새로 덮는다. 반복이 겹쳐 있다. 긴 반복 하나가 시작 위치 여럿을 낳기 때문이다.

이 관찰이 반복을 세는 두 잣대를 가른다. 시작 위치의 수는 반복이 얼마나 많은 자리에서 시작하느냐이고, 덮는 비율은 텍스트의 얼마가 반복이냐다. 걸러 낼 때 필요한 것은 뒤쪽이다.

다시 쓰기

이 절의 다시 쓰기는 반복을 걸러 내는 규칙이다. 찾는 것과 버리는 것은 다르다.

명제 14.11: 반복의 첫 출현 남기기

> R_L 의 각 위치를 그 반복이 처음 나온 자리와 나중 자리로 가르면, 나중 자리만 지우고 처음 자리를 남기는 것이 같은 내용을 한 번은 남기는 규칙이다. SA 의 구간 (같은 L 접두사를 나누는 연속 자리들) 마다 위치가 가장 작은 것이 처음이고, 나머지가 나중이다. 구간은 LCP $\geq L$ 이 이어지는 자리들이다.

증명. 정리 14.9 의 구간은 같은 길이 L 접두사를 나누는 접미사 전부이고, 그 안에서 텍스트 위치가 가장 작은 것이 첫 출현이다. ■

이 명제가 Lee 등의 정확 부분 문자열 제거다. 50 토큰 이상 겹치는 구간의 첫 것만 남긴다. 문서 단위 제거와 이것을 둘 다 한 것이 그 논문이었고, 둘이 다른 중복을 잡았다 [확립] [29].

알고리즘 14.3 반복 구간 찾기 `dsai.dupspan.duplicated_positions`

입력: 텍스트 T, SA, 문턱 L
1: LCP ← 명제 14.4
2: $R \leftarrow \{\mathrm{SA}[r], \mathrm{SA}[r-1] : \mathrm{LCP}[r] \geq L\}$; 중복을 걷고 정렬한다
3: **반환** R

구현

LCP 는 Kasai 의 한 번 훑기이고, 반복 찾기는 배열 비교 하나다.

지금 보는 코드 LCP 가 문턱 이상인 자리의 접미사 둘을 모아 정렬한다.

구현 LCP 배열 `samples/dsai/suffix.py:36`

```
def lcp_array(text: np.ndarray, sa: np.ndarray) -> np.ndarray:
    """Kasai 의 방법. 이웃한 접미사의 공통 접두사 길이. lcp[i] = lcp(sa[i-1], sa[i]).
    O(n).
    """
    n = len(text)
    rank = np.empty(n, dtype=np.int64)
    rank[sa] = np.arange(n)
    lcp = np.zeros(n, dtype=np.int64)
    h = 0
    t = text.tolist()
    for i in range(n):
        r = rank[i]
        if r == 0:
            h = 0
            continue
        j = sa[r - 1]
        while i + h < n and j + h < n and t[i + h] == t[j + h]:
            h += 1
        lcp[r] = h
        if h:
            h -= 1
    return lcp
```

읽어 보면 `h` 가 명제 14.4 의 둘째 문장이다. 앞 위치에서 h 를 찾았으면 다음 위치는 $h-1$ 부터 시작한다. `while` 이 n 번 이상 안 돈다.

구현 반복 구간 `samples/dsai/dupspan.py:15`

```
def duplicated_positions(text: np.ndarray, sa: np.ndarray, min_len: int) -> np.ndarray:
    """알고리즘 14.4. 길이 min_len 이상의 부분 문자열이 다른 곳에도 나오는 시작 위치들.

    lcp[i] >= min_len 이면 sa[i-1] 과 sa[i] 둘 다 그런 위치다. 정렬된 위치 배열이다.
    """
    lcp = lcp_array(text, sa)
    hit = lcp >= min_len
    pos = np.concatenate([sa[hit], sa[np.flatnonzero(hit) - 1]])
    return np.unique(pos)
```

읽어 보면 `lcp >= min_len` 한 줄이 정리다. 그 자리의 접미사 둘이 모두 반복의 시작이다.

구현 덮는 비율 `samples/dsai/dupspan.py:26`

```
def covered_tokens(positions: np.ndarray, min_len: int, n: int) -> int:
    """중복 구간이 덮는 토큰 수. 위치마다 [p,
    p + min_len) 을 덮고 겹침은 한 번만 센다.
    """
    if len(positions) == 0:
        return 0
    starts = np.sort(positions)
    ends = np.minimum(starts + min_len, n)
    covered, cur_s, cur_e = 0, int(starts[0]), int(ends[0])
    for s, e in zip(starts[1:].tolist(), ends[1:].tolist()):
        if s <= cur_e:
            cur_e = max(cur_e, e)
        else:
            covered += cur_e - cur_s
            cur_s, cur_e = s, e
    return covered + cur_e - cur_s
```

읽어 보면 `covered_tokens` 가 구간을 정렬해 겹치는 것을 병합하고 합집합의 길이를 센다. 시작 위치 13만 개가 29만 바이트가 되는 자리다.

분석

접미사 배열 위의 반복 찾기는 그대로 유효하다. 문자열 알고리즘 교과서의 가장 긴 반복 문제가 그대로이고, 문턱을 두어 전부를 찾는 것으로 넓어졌다. 바뀐 것은 텍스트의 크기와 찾는 이유다. 유전체에서 반복을 찾던 것이 학습 말뭉치에서 복사본을 찾는 일이 됐다.

이 구조가 서는 자리는 말뭉치 만들기다. 7장의 문서 단위 제거와 이 절의 부분 문자열 제거를 둘 다 하고, 그 뒤에 학습한다. Lee 등은 두 제거를 거친 모델이 외운 문장을 열 배 덜 뱉는다고 보고했고 [확립] [29], 외운 문장은 반복이 만든 것이다.

현장에서

> Lee 등의 정확 부분 문자열 제거는 50 토큰을 문턱으로 두었고, 접미사 배열을 말뭉치 전체에 세웠다 [확립] [29]. 토큰 수천억 개면 배열이 테라바이트이고, 그것을 만드는 것이 이 방법의 가장 큰 값이다.

주의

> 실험 14.3 의 반복은 바이트 단위이고 소스 코드다. 코드는 자연어보다 반복이 많다. 자연어 말뭉치에서 50 토큰 이상의 반복이 덮는 비율은 이보다 낮고, 말뭉치마다 다르다.

이 장의 배열 하나가 개수와 예측과 반복을 냈다. 다음 장은 어텐션이다. 그것을 자료 구조로 읽으면 연관 배열이고, 그 배열이 길이에 비례해 커지는 것이 다음 장의 문제다.

되짚기

> 반복 구간이 LCP 의 문턱 이상인 자리와 일대일임을 정리로 세우고, 소스 코드의 29 퍼센트가 50 바이트 이상의 반복임을 쟀다. 첫 출현만 남기는 규칙이 걸러 내기다. 다음 장은 어텐션을 자료 구조로 읽는다.

14장을 닫으며

이 장이 세운 구조는 하나이고 물음이 셋이다. 접미사 배열과 LCP 배열 위에서 개수 세기, 다음 토큰 예측, 반복 찾기. 셋 다 같은 불변식에 선다. 같은 접두사를 가진 접미사는 정렬 순서에서 붙어 있다.

세 실험의 숫자를 적어 둔다. 백만 바이트의 접미사 배열을 4.6 초에 만들었고 패턴 세기가 C 순차 훑기보다 여섯 배에서 스무 배 빨랐다. 무한 n-그램이 다음 바이트를 70 퍼센트 맞혀 5-그램 표의 63 퍼센트를 넘었고 메모리는 3 분의 1 이었다. 50 바이트 이상의 반복이 소스 코드의 29 퍼센트를 덮었고 가장 긴 반복이 1,208 바이트였다.

정정 하나를 남긴다. 배가 정렬의 첫 순위를 바이트 값 그대로 두었더니 banana 의 배열이 틀렸다. 두 순위를 정수 하나로 묶는 식이 순위가 n 미만이라는 가정 위에 있었고, 바이트 값 255 가 그 가정을 깼다. 글자를 조밀한 순위로 바꾸자 맞았다. 정렬 키를 묶을 때는 각 자리의 범위가 곱셈의 자릿수라는 것을 다시 배웠다.

옛 구조의 판정

> 접미사 배열은 그대로 유효하다. 1993 년의 정의와 찾기가 그대로이고, 배열 하나라는 성질이 이 책의 잣대에 맞는다. 근거는 실험 14.1 과 14.2 다. 정수 n 개로 모든 부분 문자열의 개수를 내고, 그것으로 n 없는 n-그램이 표를 이겼다. 바뀐 것은 텍스트의 크기다. 메가바이트가 테라바이트가 됐고, 만드는 값과 디스크 위의 이진 탐색이 지금의 문제다.

이 장의 요약

> 1. 접미사 배열은 접미사를 정렬한 위치 배열이고, 같은 접두사가 연속 구간이다. 구간의 길이가 개수다 14.1
> 2. 배가 정렬은 순위 쌍을 정렬하는 것을 로그 번 되풀이한다. 트리 없이 정렬만으로 만든다 14.1
> 3. 무한 n-그램은 n 을 정하지 않는다. 문맥의 구간이 개수이고, 비면 줄인다. 표보다 맞히고 n 과 무관하게 작다 14.2
> 4. 반복 구간은 LCP 가 문턱 이상인 자리와 일대일이다. 찾는 데 비교가 없다 14.3
> 5. 문서 단위 중복과 부분 문자열 중복은 다른 것이고, 둘 다 걸러야 외운 문장이 준다 14.3
> 다음 장에서 뒤집히는 것: 이 장까지의 키는 정확히 같음으로 찾았다. 15장의 어텐션은 모든 키가 유사도만큼 값을 준다. 연관 배열인데 정확한 일치가 없다.

필자의 판단

> 필자는 접미사 배열을 문자열 수업의 마지막이 아니라 말뭉치 수업의 첫 도구로 둔다. 학생이 자기가 쓰는 모델의 학습 데이터에 어떤 문장이 몇 번 있었는지를 직접 세어 보면, 이 배열이 왜 필요한지 설명이 필요 없다.

열린 문제

> 하나, 배가 정렬은 $O(n \log^2 n)$ 이고 선형 알고리즘이 있다. 이 장은 그것을 구현하지 않았고, 테라바이트에서 드는 만들기 값을 말할 수 없다.
> 둘, 실험 14.2 의 무한 n-그램은 백오프를 길이마다 새 이진 탐색으로 했다. 구간 좁히기로

바꾸면 얼마나 빨라지는지 재지 않았다.
셋, 반복을 걸러 낸 뒤 모델의 성적이 어떻게 바뀌는지는 이 책의 범위 밖이다. Lee 등의 보고를 인용했을 뿐 재현하지 않았다.

연습문제

14.1 기초 길이 10^9 의 텍스트에서 접미사 배열의 크기와, 배가 정렬의 정렬 횟수 상한과 각 정렬의 비교 횟수를 명제 14.3 으로 구하라. 8 바이트 정수와 4 바이트 정수의 차이는 얼마인가.

14.2 기초 길이 20 의 패턴을 길이 10^9 의 텍스트에서 세는 데 비교가 몇 번이고 읽는 바이트가 얼마인지 명제 14.2 로 구하라. 배열이 디스크에 있으면 페이지 접근이 몇 번인가.

14.3 기초 어휘 5만, 말뭉치 10^{12} 토큰의 5-그램 표 항목 수의 상한을 명제 14.6 로 적고, 접미사 배열의 메모리와 견줘라.

14.4 기초 banana 의 LCP 배열에서 $L = 2$ 인 반복 구간의 집합을 정리 14.9 로 구하고, 덮는 비율을 명제 14.10 로 구하라.

14.5 응용 `dsai.suffix.build_suffix_array` 의 정렬 키를 정수 하나로 묶는 대신 `np.lexsort` 로 두 열을 정렬하게 바꾸고, 실험 14.1 의 만드는 시간을 견줘라.

14.6 응용 `dsai.ngram.InfiniGram.predict` 를 명제 14.7 의 구간 좁히기로 다시 써라. 짧은 문맥에서 시작해 구간 안에서만 이진 탐색하고, 예측당 시간이 얼마나 주는지 재라.

14.7 응용 `dsai.dupspan.duplicated_positions` 에 명제 14.11 의 첫 출현 남기기를 더하라. 각 반복 구간에서 위치가 가장 작은 것을 뺀 나머지만 돌려주고, 시험으로 첫 출현이 남는지 확인하라.

14.8 응용 텍스트 둘 A, B 를 구분자로 이어 붙여 접미사 배열을 만들고, A 와 B 사이의 가장 긴 공통 부분 문자열을 LCP 로 찾아라. 같은 텍스트 안의 반복은 빼야 한다.

14.9 탐구 접미사 배열이 디스크에 있을 때 이진 탐색의 각 단계가 접근 하나라는 것을 12.4 의 잣대로 적고, 위쪽 몇 단계를 메모리에 두는 것이 접근을 얼마나 줄이는지 논하라.

14.10 탐구 7장의 문서 단위 중복 제거와 이 장의 부분 문자열 중복 제거가 각각 놓치는 중복의 예를 하나씩 들고, 둘을 어떤 순서로 하는 것이 맞는지 논하라.

더 읽을 것

[55] 접미사 배열의 원전. 정렬과 탐색이라는 틀과 접미사 트리 대비 메모리 비교가 있다. 14.1
[56] 토큰 수조 개 위의 무한 n-그램. 접미사 배열을 디스크에 두고 밀리초에 답하는 구현이다. 14.2
[29] 문서 단위와 부분 문자열 단위의 중복 제거를 둘 다 한 논문. 접미사 배열 위의 50 토큰 문턱이 14.3 절이다. 14.3

15장

어텐션을 자료 구조로 읽기

2017년의 논문은 어텐션을 질의, 키, 값의 세 행렬로 적었다 [확립] [57]. 그 이름은 연관 배열의 이름이고, 이 장은 그 이름을 글자 그대로 받는다. 질의 하나가 키 전부와 견줘 값을 고르는 읽기이고, 그래서 값이 키 수에 선형이다. 그 값을 네 방향에서 줄이는 것이 이 장의 네 절이다. 점수 행렬을 안 만들고, 키를 떼고, 과거를 요약하고, 그리고 12장처럼 가까운 키만 본다.

이 장은 이 책이 앞에서 세운 것들이 한 자리에 모이는 곳이다. 1.3 의 타일과 11.3 의 융합이 온라인 소프트맥스에서 만나고, 7.2 의 헤비 히터가 캐시 축출의 정책이 되며, 5.3 의 링 버퍼와 누적합이 고정 상태와 스캔이 된다. 새로 세우는 것은 정리 둘이다. 온라인 소프트맥스의 갱신과 선형 재귀의 결합 법칙. 나머지는 앞 장의 구조를 어텐션 위에 놓는 일이다.

이 장에서 세우는 것

15.1 어텐션은 소프트 연관 배열이다. 조회가 키 수에 선형이고, 상위 k 근사의 오차는 버린 질량이다

15.2 타일링과 온라인 소프트맥스. 점수 행렬 1 GB 가 4 MB 로, 시간은 8 퍼센트

15.3 KV 캐시 축출. 예산 256 에서 LRU 101 퍼센트, 헤비 히터 0.84 퍼센트

15.4 고정 크기 상태. KV 캐시 34 GB 대 상태 16.8 MB, 스캔은 병렬 기계의 알고리즘

15.1 어텐션은 연관 배열이다

6.1 의 해시 테이블은 키 하나를 주면 값 하나를 돌려주는 연관 배열이었다. 키가 정확히 같아야 하고, 찾는 값은 기대 상수다. 2.3 의 KV 캐시는 키 n 개와 값 n 개를 추가 전용 배열에 쌓았다. 그 배열을 읽는 것이 어텐션이다. 질의 하나가 키 전부와 유사도를 내고, 유사도를 확률로 바꿔 값의 가중 합을 돌려준다.

그러니 어텐션은 연관 배열 읽기의 다른 얼굴이다. 정확히 같은 키 하나가 아니라 모든 키가 닮은 만큼 값을 준다. 이 절은 그 읽기를 정의로 세우고, 해시 조회와 비용을 견주고, 12 장의 kNN 이 그 비용을 줄이는 자리인지를 잰다.

학습 목표

어텐션을 소프트 연관 배열의 읽기로 정의하고, 해시 조회와 견줘 연산과 이동량을 적을 수 있다. 상위 k 개 키만으로 하는 근사의 오차가 버린 확률 질량으로 상한이 잡히는 이유와, 키가 많아질수록 그 근사가 나빠지는 이유를 실측으로 말할 수 있다.

이 말부터

어텐션	질의와 키 n 개의 유사도를 확률로 바꿔 값 n 개를 가중 합하는 읽기
소프트맥스	점수 s_i 를 $e^{s_i} / \sum_j e^{s_j}$ 로 바꾸는 것. 양수이고 합이 1 이다
헤드	같은 키와 값 배열을 다른 사영으로 읽는 독립된 어텐션 하나. 층마다 여럿이다

디딤돌

디딤돌은 연관 배열이다. 키를 주면 값이 나오는 구조이고, 6 장의 해시 테이블은 그것을 기대 상수에 했다. 조건은 하나였다. 키가 정확히 같아야 한다. 어텐션은 그 조건을 버린 연관 배열이다.

정의 15.1: 소프트 연관 배열과 어텐션

키 $K \in \mathbb{R}^{n \times d}$, 값 $V \in \mathbb{R}^{n \times d_v}$, 질의 $q \in \mathbb{R}^d$ 에 대해 어텐션 읽기는 $\mathrm{attn}(q, K, V) = \sum_i p_i V_i$, $p = \mathrm{softmax}(Kq/\sqrt{d})$ 이다. 질의 m 개를 행으로 쌓으면 $\mathrm{softmax}(QK^\top/\sqrt{d})\,V$ 이고, $QK^\top$ 이 점수 행렬이다.

불변식 가중치 p 는 모두 양수이고 합이 1 이다. 따라서 출력은 값들의 볼록 결합이고, 어느 값보다 크거나 작을 수 없다.

불변식이 말하는 것은 어텐션이 값을 만들지 않고 고른다는 것이다. 해시 테이블이 값 하나를 고르면 어텐션은 값들의 평균을 고른다. 고르는 데 드는 값이 둘의 차이다.

명제 15.2: 두 조회의 비용

> 해시 조회는 키 하나를 해시해 기대 상수의 비교이고, 이동량은 라인 몇 개다. 어텐션 조회는 질의 하나에 곱셈 $nd + nd_v$ 이고, 이동량은 키와 값 전부 $n(d + d_v)$ 원소다. 질의 m 개면 연산 $mn(d + d_v)$ 이고 점수 행렬을 만들면 이동량에 mn 원소가 더해진다. 따라서 어텐션 조회는 키 수에 선형이고, 키가 n 개인 표에서 질의 하나가 표 전체를 읽는다.

증명. 점수 Kq 가 nd 곱셈이고 가중 합 $p^\top V$ 가 nd_v 곱셈이다. 둘 다 K 와 V 의 모든 원소를 한 번씩 읽는다. 질의가 m 개면 m 배이고, 점수 행렬 $QK^\top$ 을 메모리에 두면 mn 원소를 쓰고 읽는다. 해시 조회는 6.1 의 명제 그대로다. ■

이 명제의 셋째 문장이 이 장 전체의 출발점이다. 키 n 개를 매번 전부 읽는 것이 어텐션의 값이고, 15장의 네 절은 그 값을 네 방향에서 줄인다. 행렬을 안 만들거나, 키를 떼거나, 키를 요약하거나, 12장처럼 가까운 키만 본다.

예제 15.1: 키 백만 개

> $n = 10^6$, $d = d_v = 64$, float32 이면 키와 값이 512 MB 다. 질의 하나가 512 MB 를 읽고, 0.1 의 잣대로 10 GB/s 이면 51 밀리초다. 해시 조회는 같은 표에서 라인 몇 개이고 1 마이크로초 아래다. 십만 배 차이다.

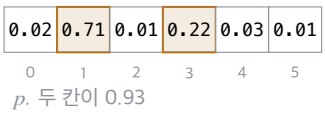


그림 15.1.. 소프트 조회. 질의가 키 여섯 개와 점수를 내고 확률로 바꾼다. 합이 1 이고, 뾰족하면 두 칸이 질량 대부분을 갖는다

균열

해시 조회와 어텐션 조회를 같은 키 배열 위에서 쟀다. 그리고 점수가 큰 키 k 개만으로 가중 합을 만드는 근사가 전체와 얼마나 어긋나는지를 함께 쟀다. 12장의 kNN 이 그 k 개를 찾는 도구다.

실험 15.1 해시 조회, 어텐션 조회, 상위 k 근사

키 n	해시 us	어텐션 us	오차 16 퍼센트	오차 256 퍼센트	오차 4096 퍼센트
1000	0.55	122	0.0542	0.0097	9.0e-06
10000	0.56	551	0.63	0.35	0.0255
100000	0.48	4175	12.9	9.40	3.69
1000000	0.31	42285	40.6	38.7	30.8

측정 조건. 차원 64, float32. 질의는 키 하나의 배수에 잡음을 더한 것이라 그 키의 점수가 16 이고 나머지는 평균 0 이다. 오차는 상위 k 개만 남긴 출력과 전체 출력의 최대 차이를 출력의 최대 절댓값으로 나눈 값. 다섯 번 중앙값.

해시 조회는 키 수와 무관하게 0.3 에서 0.6 마이크로초다. 어텐션 조회는 키 천 개에서 121 마이크로초, 백만 개에서 42 밀리초다. 백만 개의 42 밀리초는 512 MB 를 12 GB/s 로 읽은 시간이고, 명제 15.2 의 이동량 그대로다.

상위 k 개 근사는 키가 적을 때 정확하고 많을 때 틀린다. 천 개에서는 상위 16 개로 0.05 퍼센트인데, 백만 개에서는 상위 4,096 개로도 31 퍼센트다. 점수 16 인 키 하나가 갖는 질량이 $e^{16}/(e^{16} + ne^2)$ 안팎이라 n 이 커질수록 나머지 키 전체로 질량이 새어 나가기 때문이다.

관찰

어텐션 조회의 시간은 키와 값의 바이트를 대역폭으로 나눈 값이다. 상위 k 근사는 소프트맥스가 뾰족할 때만 맞고, 뾰족한지는 점수의 차이와 키 수가 함께 정한다.

이 관찰이 12장을 이 장에 잇는 조건이다. 0.3 의 kNN 으로 키 k 개를 찾아 어텐션을 대신하려면 버린 키의 질량이 작아야 한다. 그 질량이 오차를 정한다는 것이 다음 명제다.

다시 쓰기

이 절의 다시 쓰기는 근사 조회의 오차를 버린 질량으로 적는 것이다. 12장의 재현율이 여기서는 질량이 된다.

명제 15.3: 상위 k 근사의 오차

키 집합 S 를 남기고 나머지의 확률 질량이 $\varepsilon = \sum_{i \notin S} p_i$ 라 하자. 남긴 키로 다시 정규화한 출력 $o_S = \sum_{i \in S} p_i V_i / (1 - \varepsilon)$ 은 $\|o - o_S\|_\infty \le 2\varepsilon \max_i \|V_i\|_\infty$ 를 만족한다. 따라서 오차는 버린 질량에 선형이고, 버린 키의 개수와는 무관하다.

증명. $o - o_S = \sum_{i \notin S} p_i V_i - \frac{\varepsilon}{1-\varepsilon} \sum_{i \in S} p_i V_i$ 이다. 첫 항은 질량 ε 의 가중 합이라 크기가 $\varepsilon \max \|V_i\|$ 이하이고, 둘째 항은 질량 $1 - \varepsilon$ 의 가중 합에 $\varepsilon/(1 - \varepsilon)$ 을 곱한 것이라 역시 $\varepsilon \max \|V_i\|$ 이하다. ■

명제가 실험을 설명한다. 백만 개에서 상위 4,096 개가 담은 질량이 70 퍼센트 안팎이라 오차가 31 퍼센트였다. 키를 더 남기는 것보다 점수가 더 뾰족해지는 것이 오차를 줄인다. 그것은 자료 구조가 아니라 모델의 몫이다.

알고리즘 15.1 소프트 조회 `dsai.attention.attention_naive`

입력: 질의 Q $(m \times d)$, 키 K $(n \times d)$, 값 V $(n \times d_v)$

1: $S \leftarrow QK^{\top}/\sqrt{d}$ ▷ 점수 행렬 $m \times n$ 이 메모리에 생긴다

2: $S \leftarrow S - \max_{\text{행}} S$; $P \leftarrow \exp(S)$; $P \leftarrow P / \sum_{\text{행}} P$

3: **반환** PV

구현

정의를 그대로 옮긴 구현이다. 점수 행렬을 만들고, 행마다 최댓값을 빼고, 지수를 취해 정규화한다.

지금 보는 코드 점수 행렬 $m \times n$ 을 통째로 만든다. n 이 만 단위면 그 행렬이 GB 다.

구현 소프트 조회 `samples/dsai/attention.py:14`

```
def attention_naive(q: np.ndarray, k: np.ndarray, v: np.ndarray) -> np.ndarray:
    """정의 15.1. (m, d) 질의, (n, d) 키, (n, dv) 값. 점수 행렬 (m,
    n) 을 메모리에 만든다."""
    s = q @ k.T / np.sqrt(q.shape[1])
    s = s - s.max(axis=1, keepdims=True)
    p = np.exp(s)
    p /= p.sum(axis=1, keepdims=True)
    return p @ v
```

읽어 보면 `s - s.max(axis=1)` 가 넘침을 막는 표준 장치다. 지수의 인수가 0 이하가 되어 `exp` 가 넘치지 않는다. 다음 절의 온라인 소프트맥스는 이 최댓값을 블록마다 갱신한다.

분석

이 절의 잣대는 0.2 의 첫째 가정이다. 조회 하나가 라인 몇 개라는 가정이 어텐션에서는 표 전체가 된다. 그래서 어텐션의 시간은 알고리즘이 아니라 대역폭이고, 그것을 줄이는 길은 읽는 바이트를 줄이는 것뿐이다. 연산 밀도로 보면 질의 m 개가 키 n 개를 읽을 때 원소마다 곱셈 m 번이라 m 이 클수록 능선 위로 올라간다. 학습은 $m = n$ 이라 연산이 시간을 정하고, 디코딩은 $m = 1$ 이라 이동이 시간을 정한다. 다음 절이 앞을, 그다음 절이 뒤를 다룬다.

현장에서

Vaswani 등의 2017년 논문은 어텐션을 질의, 키, 값의 세 행렬로 적었고 점수를 $\sqrt{d}$ 로 나누는 이유를 점수의 분산으로 설명했다 [확립] [57]. 그 표기의 키와 값이라는 이름 자체가 연관 배열이고, 이 절은 그 이름을 자료 구조의 이름으로 읽었다.

주의

> 실험 15.1 의 오차는 질의 하나에서 잰 값이다. 질의마다 점수의 뾰족함이 다르므로 상위 k 근사가 평균으로 얼마나 맞는지는 이 실험이 말하지 않는다. 그리고 해시 조회의 0.3 마이크로초는 파이썬 `dict` 의 값이라 C 의 해시 테이블보다 열 배 느리다. 십만 배 차이는 그것을 셈해도 만 배다.

연관 배열로 읽으면 어텐션의 값이 한 문장이 된다. 질의마다 표 전체를 읽는다. 다음 절은 그 표를 읽되 점수 행렬을 만들지 않는 법이고, 그것이 6장의 타일과 11장의 융합이 만나는 자리다.

되짚기

> 어텐션을 모든 키가 닮은 만큼 값을 주는 소프트 연관 배열의 읽기로 정의하고, 조회 비용이 키 수에 선형이며 그 시간이 대역폭임을 실측했다. 상위 k 근사의 오차는 버린 질량에 선형이고, 키가 많을수록 질량이 새어 근사가 나빠진다. 다음 절은 점수 행렬을 만들지 않는 타일링과 온라인 소프트맥스다.

15.2 타일링과 온라인 소프트맥스. 행렬을 안 만든다

15.1 의 점수 행렬은 $m \times n$ 이다. 학습에서는 $m = n$ 이고, n 이 16,384 이면 float32 로 1 GB 다. 헤드 32 개와 층 32 개면 그 천 배다. 그 행렬은 소프트맥스를 위해 잠깐 있다가 사라지는데, 만들고 읽는 이동량이 어텐션 전체의 시간을 정한다. 1.3 의 타일은 행렬 곱에서 캐시에 드는 블록만 만들었다. 같은 일을 어텐션에 하면 점수 행렬의 블록 하나만 만들면 되는데, 소프트맥스의 분모가 행 전체의 합이라 블록 하나로는 정규화를 못 한다.

이 절은 그 분모를 블록마다 고쳐 나가는 온라인 소프트맥스를 세우고, 그것으로 11.3 의 융합을 어텐션 전체에 적용한다. 점수 행렬이 메모리에 없는 어텐션이다.

학습 목표

온라인 소프트맥스의 상태 셋 (최댓값, 합, 누적) 을 정의하고 블록을 하나 더 볼 때의 갱신이 정확함을 증명할 수 있다. 타일 어텐션의 이동량이 점수 행렬 없이 키와 값의 크기만으로 적힘을 보이고, 시간이 아니라 메모리가 준 이유를 실측으로 말할 수 있다.

이 말부터

타일	키와 값을 블록 b 개씩 자른 조각. 블록 점수 $m \times b$ 만 만든다
온라인 소프트맥스	최댓값과 합을 블록마다 갱신해 마지막에 한 번 나누는 소프트맥스
재조정	새 최댓값이 나왔을 때 옛 누적에 $e^{m_{옛}-m_{새}}$ 를 곱하는 것

디딤돌

디딤돌은 1.3 의 타일과 11.3 의 융합이다. 타일은 캐시에 드는 조각만 만들고, 융합은 중간 텐서를 메모리에 안 낸다. 어텐션에서 중간 텐서는 점수 행렬이고, 그것을 안 내려면 소프트맥스의 분모를 조각마다 고쳐야 한다.

정의 15.4: 온라인 소프트맥스의 상태

질의 하나에 대해 키를 블록 $B_1, \dots, B_t$ 로 나누고, 블록 j 까지 본 뒤의 상태를 (m_j, l_j, o_j) 로 둔다. m_j 는 지금까지 본 점수의 최댓값, $l_j = \sum_{i \le j\text{까지}} e^{s_i - m_j}$, $o_j = \sum_i e^{s_i - m_j} V_i$ 다.

불변식 블록 j 까지 본 뒤 o_j / l_j 는 그 j 개 블록의 키만 있는 어텐션의 출력과 같다.

불변식이 정확성의 전부다. 마지막 블록까지 보면 o_t / l_t 가 전체 어텐션이다. 남는 것은 블록을 하나 더 볼 때 상태를 어떻게 고치면 불변식이 유지되는가이고, 그것이 정리다.

정리 15.5: 온라인 소프트맥스의 갱신

> 블록 B 의 점수가 s_i ($i \in B$) 이고 $m' = \max(m, \max_{i \in B} s_i)$ 라 하자. $l' = e^{m-m'} l + \sum_{i \in B} e^{s_i - m'}$, $o' = e^{m-m'} o + \sum_{i \in B} e^{s_i - m'} V_i$ 로 갱신하면 정의 15.4 의 불변식이 유지된다. 지수의 인수는 항상 0 이하라 넘침이 없고, 블록마다 만드는 것은 블록 점수 $|B|$ 개뿐이다.

증명. 옛 상태는 $l = \sum_{\text{옛}} e^{s_i - m}$ 이고 $e^{m-m'} l = \sum_{\text{옛}} e^{s_i - m'}$ 이다. 새 블록의 항을 더하면 $l' = \sum_{\text{전부}} e^{s_i - m'}$ 로 정의 그대로다. o 도 같은 계산이다. $m' \geq m$ 이고 $m' \geq s_i$ 이므로 인수가 0 이하다. 갱신에 쓰는 것은 m, l, o 와 블록 점수뿐이라 옛 블록의 점수는 필요 없다. ■

정리의 마지막 문장이 이동량이다. 옛 블록의 점수를 다시 볼 일이 없으므로 점수 행렬을 저장할 이유가 없다. 그러면 이동량은 키와 값을 한 번 읽는 것으로 준다.

명제 15.6: 어텐션의 이동량

> 질의 m 개, 키 n 개, 차원 d 와 d_v 에서 점수 행렬을 만드는 어텐션의 이동량은 원소 $4mn + md + nd + nd_v + md_v$ 다. 행렬을 쓰고 읽고, 확률을 쓰고 읽는 $4mn$ 이 지배한다. 타일 어텐션은 $md + nd + nd_v + md_v$ 이고, 질의 블록이 캐시에 머무는 동안 키와 값을 한 번 읽는다. 따라서 $m = n$ 에서 이동량은 $4n^2$ 에서 $2n(d + d_v)$ 로 $2n/(d + d_v)$ 분의 1 이 되고, 연산 밀도는 그만큼 오른다.

증명. 점수 행렬은 mn 원소를 쓰고, 소프트맥스가 그것을 읽어 확률 mn 원소를 쓰고, 가중합이 그것을 읽는다. 입력 셋과 출력 하나는 어느 쪽이든 한 번씩이다. 타일에서는 블록 점수가 캐시 안에서 태어나 캐시 안에서 사라지므로 메모리 이동이 아니다. 질의 블록 하나에 키와 값 전부를 한 번 읽고, 질의 블록이 캐시에 드는 크기이면 다음 질의 블록에서 다시 읽는 것은 키와 값이 아니라 그 블록의 몫이다. ■

명제의 값을 실험의 조건에 넣으면 $n = 16{,}384$, $d = d_v = 64$ 에서 4,312 MB 와 16.8 MB 다. 256 배다. 시간이 그만큼 줄지는 이 절의 실험이 답한다. 그 전에 메모리 쪽을 예제로 적어 둔다.

예제 15.2: 길이 16,384 의 점수 행렬

> $n = 16{,}384$ 의 점수 행렬은 float32 로 1,074 MB 다. 질의 전체를 한 블록으로 두고 키만 1,024 씩 자르면 블록 점수가 67 MB 이고, 질의도 128 씩 자르면 512 KB 다. 캐시에 드는 것은 뒤쪽이고, 그때 키와 값을 질의 블록 수 128 번 다시 읽는다.

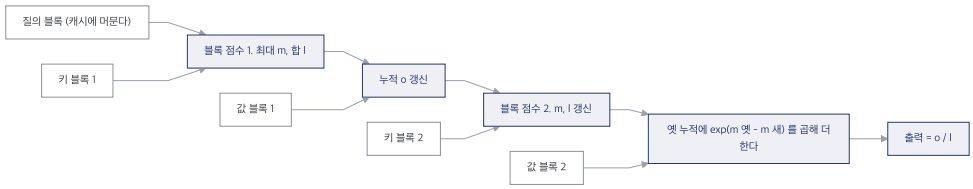


그림 15.2. 타일 어텐션의 흐름. 질의 블록이 캐시에 머무는 동안 키 블록과 값 블록이 차례로 지나가고, 블록마다 최댓값과 합과 누적을 갱신한다. 점수 행렬은 어디에도 없다

균열

점수 행렬을 만드는 어텐션과 타일 어텐션을 길이 1,024 에서 16,384 까지 쟀다. 두 방법의 출력이 같음을 먼저 확인했다.

실험 15.2 점수 행렬 어텐션과 타일 어텐션

길이 n	점수 행렬 MB	하나씩 이동 MB	타일 이동 MB	하나씩 ms	타일 ms
1024	4.19	17.8	1.05	68.8	30.6
4096	67.1	273	4.19	1043	531
16384	1074	4312	16.8	5622	5188

측정 조건. 차원 64, float32, 타일 1,024. 이동량 두 열은 명제 15.6 의 값이고 실측이 아니다. 세 번 중앙값.

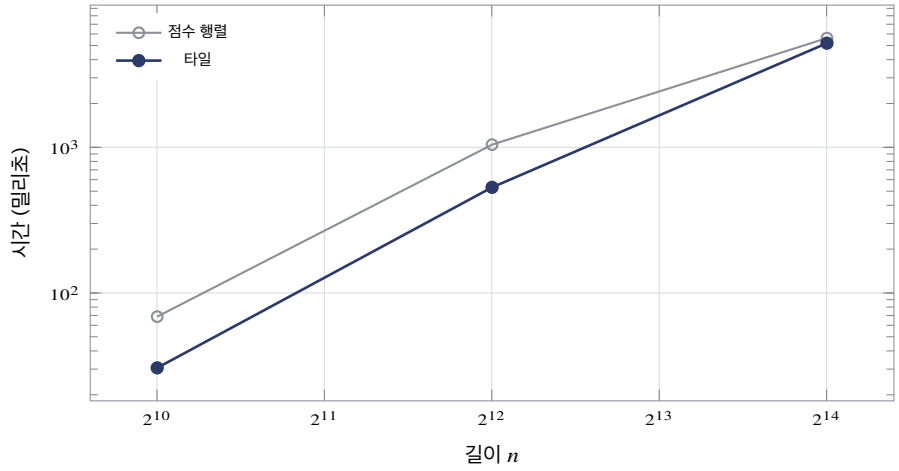


그림 15.3. 길이에 따른 두 어텐션의 시간. 작은 길이에서 두 배, 큰 길이에서 8 퍼센트 차이다. 메모리는 이 그림에 없다

길이 16,384 에서 점수 행렬이 1 GB 이고, 타일은 블록 점수 4 MB 만 둔다. 이동량은 4,312 MB 와 16.8 MB 로 256 배 차이인데, 시간은 5.6 초와 5.2 초로 8 퍼센트 차이다. 길이 1,024 와 4,096 에서는 두 배다. 작은 행렬에서는 점수 행렬을 쓰고 읽는 값이 보이고, 큰 행렬에서는 행렬 곱의 연산이 시간을 정해 이동량이 안 보인다.

관찰

> 타일 어텐션이 줄이는 것은 메모리이고, 시간은 연산 밀도가 능선 아래일 때만 준다. $m = n$ 의 학습은 능선 위라 시간이 거의 같고, 메모리는 n^2 에서 nb 로 준다.

이 관찰이 11.3 의 마지막 관찰과 같은 모양이다. 융합은 이동량을 줄이고, 시간은 밀도가 정한다. 어텐션에서 융합이 값진 이유는 시간이 아니라 그 1 GB 가 층과 헤드마다 천 번 곱해지기 때문이다.

다시 쓰기

이 절의 다시 쓰기는 소프트맥스를 축소 둘과 원소별 하나로 보는 것이다. 11.3 의 경계 명제는 축소 뒤의 원소별 연산에 경계가 생긴다고 했다. 온라인 소프트맥스는 그 경계를 지운 것이다.

명제 15.7: 소프트맥스 융합의 조건

> 소프트맥스는 최댓값 (축소), 지수 (원소별), 합 (축소), 나눗셈 (원소별) 의 넷이다. 명제 11.11 대로 하면 축소 뒤마다 경계가 생겨 행렬 전체가 두 번 완성되어야 한다. 정리 15.5 의 갱신은 두 축소를 상태 (m, l) 로 바꿔 한 번 훑기로 만들고, 나눗셈을 마지막으로 미룬다. 따라서 어텐션 전체가 행렬 곱, 소프트맥스, 행렬 곱의 사슬 하나로 융합된다.

증명. 축소가 경계를 만드는 이유는 그 결과가 다음 단계의 모든 원소에 필요해서였다. 최댓값과 합을 고정된 값이 아니라 갱신되는 상태로 두면, 지금까지의 상태로 계산한 누적을 뒤에서 재조정하는 것으로 최종 값과 같아진다. 그것이 정리의 $e^{m-m'}$ 이다. 나눗셈은 l 이 완성된 뒤 한 번이라 마지막에 둔다. ■

Dao 등의 2022 년 논문이 이 융합을 GPU 의 두 메모리 사이에서 구현했고, 점수 행렬을 느린 메모리에 쓰지 않는 것으로 학습 어텐션을 두세 배 빠르게 했다 [확립] [58]. 온라인 소프트맥스 자체는 Milakov 와 Gimelshein 이 2018 년에 한 번 훑기의 정규화로 적었다 [확립] [59]. 두 논문을 잇는 것이 이 절의 정리다.

알고리즘 15.2 타일 어텐션 `dsai.attention.attention_tiled`

> **입력 :** 질의 Q, 키 K, 값 V, 블록 b
>
> $m \leftarrow -\infty$, $l \leftarrow 0$, $o \leftarrow 0$ (질의마다)
> **모든** 키 블록 $[s, s+b)$ **동안**
> $S \leftarrow Q K_{[s,s+b)}^{\top} / \sqrt{d}$ ▷ 블록 점수 $m \times b$ 만
> $m' \leftarrow \max(m, \max S)$; $\alpha \leftarrow e^{m-m'}$
> $l \leftarrow \alpha l + \sum e^{S-m'}$; $o \leftarrow \alpha o + e^{S-m'} V_{[s,s+b)}$; $m \leftarrow m'$
> **끝 반복**
> **반환** o / l

구현

정리의 갱신 셋이 루프 한 번이다. 블록 점수 하나가 있다가 사라진다.

지금 보는 코드 블록마다 최댓값을 갱신하고, 옛 누적에 α 를 곱해 새 블록의 몫을 더한다.

구현 타일 어텐션 `samples/dsai/attention.py:25`

```
def attention_tiled(q: np.ndarray, k: np.ndarray, v: np.ndarray,
                    block: int) -> np.ndarray:
    """알고리즘 15.1. 키를 블록으로 나누고 블록마다 최대 m, 합 l, 누적 o 를 갱신한다.
    점수 행렬을 안 만든다.
    """
    scale = 1.0 / np.sqrt(q.shape[1])
    n = k.shape[0]
    m_run = np.full(q.shape[0], -np.inf)
    l_run = np.zeros(q.shape[0])
    o_run = np.zeros((q.shape[0], v.shape[1]))
    for s in range(0, n, block):
        e = min(s + block, n)
        sc = q @ k[s:e].T * scale                       # (m, b) 블록 점수만
        m_new = np.maximum(m_run, sc.max(axis=1))
        alpha = np.exp(m_run - m_new)                    # 옛 누적을 새 최대에 맞춘다
        p = np.exp(sc - m_new[:, None])
        l_run = alpha * l_run + p.sum(axis=1)
        o_run = alpha[:, None] * o_run + p @ v[s:e]
        m_run = m_new
    return o_run / l_run[:, None]
```

읽어 보면 `alpha` 가 재조정이다. 새 블록에 더 큰 점수가 없으면 `alpha` 는 1 이고 옛 누적이 그대로다. `sc` 가 유일한 블록 크기의 임시 배열이고, 루프가 끝나면 없다.

구현 이동량 두 가지 `samples/dsai/attention.py:47`

```
def traffic_naive(m: int, n: int, d: int, dv: int, itemsize: int) -> int:
    """명제 15.2. 점수 행렬 쓰기 + 읽기 + 확률 쓰기 + 읽기: 4 m n 원소.
    입력과 출력은 별도.
    """
    return itemsize * (4 * m * n + m * d + n * d + n * dv + m * dv)


def traffic_tiled(m: int, n: int, d: int, dv: int, itemsize: int, block: int) -> int:
    """타일마다 키와 값 블록을 한 번 읽고, 질의는 캐시에 머문다. 점수 행렬이 없다."""
    return itemsize * (m * d + n * d + n * dv + m * dv)
```

읽어 보면 명제 15.6 의 두 식이다. 실험의 이동량 열이 이 두 함수의 값이고, 실측 시간과 나란히 두면 밀도가 보인다.

분석

이 구조는 6장의 타일과 11장의 융합이 하나의 자리에서 만난 것이다. 타일이 캐시에 드는 조각을 만들고, 융합이 그 조각을 메모리에 안 낸다. 새로운 것은 축소 둘을 상태로 바꾼 정리 하나이고, 그 정리가 어텐션의 메모리를 n^2 에서 nb 로 바꿨다.

이 구조가 서는 자리는 모든 학습이다. 어텐션의 메모리가 길이의 제곱이면 긴 문맥을 학습할 수 없고, 선형이면 할 수 있다. 시간이 아니라 메모리가 길이의 상한을 정했고, 이 절은 그 상한을 옮겼다.

현장에서

> FlashAttention 의 보고에서 길이 4,096 의 학습 어텐션은 표준 구현보다 세 배 빨랐고, 메모리는 길이에 선형이었다 **[확립]** [58]. 이 절의 실험이 시간에서 그만큼을 못 얻은 것은 NumPy 의 행렬곱이 이미 캐시에 최적화되어 있어 점수 행렬의 이동이 시간의 작은 몫이기 때문이다.

주의

> 실험 15.2 의 타일은 질의 전체를 한 블록으로 두고 키만 잘랐다. 질의도 잘라야 블록 점수가 캐시에 들고, 그때 질의 블록마다 키와 값을 다시 읽는 몫이 명제 15.6 에 더해진다. 그 몫은 질의 블록 수에 비례한다. 그리고 역방향에서는 점수 행렬이 다시 필요한데, 저장하지 않고 다시 계산한다. 메모리를 연산으로 사는 맞바꿈이고 11.2 의 재계산이다.

표를 다 읽되 행렬을 안 만드는 법이 여기까지다. 다음 절은 표 자체를 줄인다. 디코딩은 질의가 하나라 이동량이 시간이고, 키를 떼는 것 말고는 줄일 길이 없다.

되짚기

> 온라인 소프트맥스의 상태 셋과 갱신을 정리로 세우고, 점수 행렬 없는 어텐션의 이동량이 $4n^2$ 에서 $2n(d+d_v)$ 로 줌을 보였다. 실측에서 메모리는 1 GB 에서 4 MB 로 줄었고 시간은 8 퍼센트 줄었으며, 그 차이가 연산 밀도다. 다음 절은 KV 캐시의 축출이다.

15.3 KV 캐시 축출. LRU 에서 헤비 히터로

디코딩은 질의가 하나다. 15.1 의 명제대로 토큰 하나를 낼 때마다 키와 값 전부를 읽고, 2.3 의 캐시는 토큰마다 자란다. 토큰 십만 개면 캐시가 수십 GB 이고, 토큰 하나를 내는 시간이 그 캐시를 읽는 시간이다. 2.4 의 페이지드 어텐션은 그 캐시를 블록으로 두어 조각을 없앴지만 크기는 그대로였다. 크기를 줄이려면 토큰을 떼야 한다. 무엇을 떼느냐가 정책이고, 이 절은 정책 둘을 견준다.

하나는 운영체제의 캐시가 쓰던 LRU 다. 오래 안 쓴 것을 뗀다. 다른 하나는 7.2 의 헤비 히터다. 지금까지 어텐션 질량을 많이 받은 토큰을 남기고 적게 받은 토큰을 뗀다. 둘의 차이가 출력의 오차로 나타난다.

학습 목표

KV 캐시 축출을 예산과 정책으로 정의하고, 뗀 토큰의 질량이 출력 오차의 상한임을 앞 절의 명제로 적을 수 있다. LRU 와 헤비 히터가 같은 예산에서 왜 백 배 다른 오차를 내는지를 참조 패턴으로 설명하고, 헤비 히터가 최근 토큰을 따로 보호하는 이유를 말할 수 있다.

이 말부터

축출	캐시가 예산을 넘을 때 항목 하나를 떼는 것. 무엇을 떼느냐가 정책이다
LRU	가장 오래 참조되지 않은 항목을 떼는 정책. 어텐션에서는 가장 오래된 토큰
헤비 히터	누적 어텐션 점수가 큰 토큰. 7장의 스케치가 세던 그 자주 나오는 것

디딤돌

디딤돌은 캐시 축출 정책이다. 메모리 계층의 캐시는 라인이 꽉 차면 LRU 로 뗐고, 그 근거는 시간 지역성이었다. 최근에 쓴 것을 곧 다시 쓴다. 어텐션의 참조는 그 가정을 따르지 않는다.

정의 15.8: KV 캐시 축출

예산 B 의 KV 캐시는 토큰 집합 S_t 를 두고, 단계 t 에 토큰 t 를 넣은 뒤 $|S_t| > B$ 이면 정책이 고른 토큰 하나를 뗀다. 단계 t 의 출력은 $\mathrm{attn}(q_t, K_{S_t}, V_{S_t})$ 이고, 뗀 토큰은 다시 안 돌아온다. LRU 는 $\min S_t$ 를 떼고, 헤비 히터는 최근 r 개를 뺀 나머지에서 누적 점수 $\sigma_i = \sum_{u \le t} p_{u,i}$ 가 가장 작은 토큰을 뗀다.

불변식 $|S_t| \le B + 1$ 이고, 뗀 토큰의 키와 값은 어느 단계에서도 다시 읽히지 않는다.

불변식의 둘째 문장이 이 구조의 값이자 값이다. 단계마다 읽는 것이 B 개로 고정되어 시간이 길이와 무관해지고, 대신 뗀 토큰이 뒤에서 필요하면 그 몫만큼 출력이 어긋난다. 얼마나 어긋나는지가 앞 절의 명제다.

명제 15.9: 축출의 오차

> 단계 t 에서 전체 어텐션이 뗀 토큰들에 주는 질량을 $\varepsilon_t = \sum_{i \notin S_t} p_{t,i}$ 라 하면, 축출 캐시의 출력과 전체 출력의 차이는 명제 15.3 대로 $2\varepsilon_t \max \|V_i\|_\infty$ 이하다. 따라서 좋은 정책이란 앞으로 질량을 많이 받을 토큰을 남기는 정책이고, 누적 점수는 그것의 추정량이다.

증명. 축출 캐시의 출력은 S_t 위에서 다시 정규화한 어텐션이라 명제 15.3 의 o_S 와 같다. 둘째 문장은 오차가 질량에 선형이라는 것의 다른 표현이고, 누적 점수는 과거의 질량이라 미래의 질량을 그것으로 어림한다. ■

명제의 둘째 문장이 LRU 와 헤비 히터를 가른다. LRU 는 오래된 토큰이 앞으로 질량을 안 받는다고 가정한다. 어텐션에는 그 가정이 안 맞는 토큰이 있다. 문서의 첫 토큰과 지시문의 토큰은 끝까지 참조된다.

예제 15.3: 예산 256 에서 오래된 토큰 하나

> 토큰 32 개가 단계마다 질량의 절반을 받는다고 하자. 그 토큰이 앞쪽에 있으면 LRU 는 예산을 넘는 순간 그것부터 뗀다. 그 뒤로 모든 단계에서 ε_t 가 0.5 안팎이고, 출력이 값의 최대 크기만큼 어긋난다. 헤비 히터는 그 32 개의 누적 점수가 크므로 남긴다. 떼는 것은 질량을 거의 안 받은 토큰이고 ε_t 가 작다.

그림 15.4.. 누적 점수. 오래됐지만 점수가 큰 토큰 0 과 2 가 남고, 점수가 작은 토큰이 나간다. 최근 둘은 점수와 무관하게 보호한다

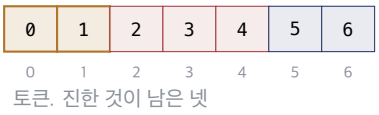


그림 15.5.. 예산 4 의 캐시. 토큰 0 과 1 이 헤비 히터라 남고, 5 와 6 이 최근 창이며, 2 부터 4 는 뗀 토큰이다. LRU 였다면 0 과 1 이 먼저 나갔다

균열

길이 2,048 의 디코딩을 예산 256 에서 두 정책으로 돌리고, 전부 유지한 정확한 인과 어텐션과 견줬다. 앞쪽 토큰 32 개의 키를 질의 방향으로 밀어 그것들이 헤비 히터가 되게 했다.

실험 15.3 예산 256 의 축출. 전부 유지, LRU, 헤비 히터

정책	캐시 토큰	평균 오차 퍼센트	최대 오차 퍼센트
전부 유지	2048	3.2e-13	1.1e-12
LRU	257	101	573
헤비 히터	257	0.84	229

측정 조건. 길이 2,048, 차원 64, 예산 256. 헤비 히터는 최근 32 개를 보호. 오차는 단계마다 정확한 출력과의 차이를 그 출력의 크기로 나눈 상대값이고, 평균과 최대는 단계 전체에서.

LRU 의 평균 오차가 101 퍼센트이고 헤비 히터가 0.84 퍼센트다. 백 배가 넘는다. LRU 는 단계 257 에서 앞쪽 토큰을 떼기 시작해 곧 헤비 히터 32 개를 전부 잃고, 그 뒤의 모든 출력이 질량 절반을 잃은 값이다. 헤비 히터의 최대 오차 229 퍼센트는 예산을 넘긴 직후 몇 단계다. 누적 점수가 아직 충분히 쌓이지 않아 뒤에서 참조될 토큰을 뗀 자리다.

관찰

어텐션의 참조는 시간 지역성이 아니라 질량 지역성을 따른다. 오래된 토큰 몇 개가 질량의 절반을 받고, 그것을 떼면 캐시가 아무리 커도 출력이 틀린다.

이 관찰이 운영체제의 LRU 를 이 자리에서 밀어낸다. 그렇다고 헤비 히터가 전부는 아니다. 최근 토큰은 누적 점수가 작은데도 곧 참조되므로 따로 보호해야 하고, 그것이 정의의 r 이다.

다시 쓰기

이 절의 다시 쓰기는 축출 정책을 두 집합의 합으로 적는 것이다. 헤비 히터와 최근 창. 앞은 7장의 스케치가 세는 것이고, 뒤는 5.3 의 링 버퍼다.

명제 15.10: 두 집합의 캐시

예산 B 를 헤비 히터 $B - r$ 개와 최근 창 r 개로 나누면, 최근 창은 링 버퍼로 $O(1)$ 에 갱신되고 헤비 히터 집합은 단계마다 $O(B)$ 의 최솟값 찾기다. 누적 점수 갱신은 단계마다 $|S_t|$ 개의 덧셈이라 어텐션 계산에 묻힌다. 따라서 축출의 값은 단계당 $O(B)$ 이고 길이와 무관하다.

증명. 최근 창은 가장 오래된 것을 떼고 새것을 넣는 큐라 링 버퍼 그대로다. 헤비 히터 집합에서 최솟값을 찾는 것은 B 개의 훑기이고, 10.1 의 힙으로 $O(\log B)$ 에 할 수도 있으나 점수가 매 단계 바뀌므로 훑기가 단순하다. 누적 점수는 그 단계의 확률 p 를 더하는 것이고 p 는 어텐션이 이미 계산했다. ■

Zhang 등의 2023 년 논문이 이 두 집합의 캐시를 헤비 히터 오라클이라 불렀고, 예산을 20 퍼센트로 줄이면서 생성 품질을 유지했다고 보고했다 [확립] [60]. Xiao 등은 같은 해에

첫 토큰 몇 개가 질량을 흡수하는 싱크임을 관찰하고, 그것과 최근 창만 남기는 스트리밍 캐시를 제안했다 [확립] [61]. 둘 다 이 절의 정의에서 r 과 헤비 히터의 고르는 법만 다르다.

알고리즘 15.3 축출 디코딩 `dsai.kvevict.decode_with_policy`

입력 : 질의, 키, 값 n 개, 예산 B, 정책, 보호 r

모든 단계 t **동안**
  $S \leftarrow S \cup \{t\}$; $p \leftarrow \mathrm{softmax}(K_S q_t / \sqrt{d})$; 출력 $o_t \leftarrow p^\top V_S$
  $\sigma_S \leftarrow \sigma_S + p$ ▷ 누적 점수
  만약 $|S| > B$ **이면** 정책이 고른 토큰을 S 에서 뗀다
  끝 만약
끝 반복

구현

정책 고르기와 디코딩 루프다. 정책은 함수 하나이고 루프는 정의 그대로다.

지금 보는 코드 LRU 는 가장 오래된 토큰, 헤비 히터는 보호 창 밖에서 누적 점수가 가장 작은 토큰을 고른다.

구현 정책 둘 `samples/dsai/kvevict.py:18`

```
def choose_victim(idx: np.ndarray, score_sum: np.ndarray, policy: str,
                  recent: int) -> int:
    """두 정책. lru 는 가장 오래된 토큰, heavy 는 최근 recent 개를 뺀 나머지에서
    누적 어텐션 점수가 가장 작은 토큰.
    """
    if policy == "lru":
        return int(idx[0])
    cand = idx[: max(len(idx) - recent, 1)]
    return int(cand[np.argmin(score_sum[cand])])
```

읽어 보면 `idx[0]` 이 가장 오래된 토큰이고 `cand` 가 보호 창을 뺀 후보다. `argmin` 한 줄이 명제 15.10 의 훑기다.

구현 축출 디코딩 `samples/dsai/kvevict.py:29`

```
def decode_with_policy(q: np.ndarray, k: np.ndarray, v: np.ndarray, budget: int,
                       policy: str, recent: int = 0) -> tuple[np.ndarray, np.ndarray]:
    """알고리즘 15.2. 토큰을 하나씩 디코딩하며 캐시를 예산 안에 둔다.
    q, k, v 는 (n, d). 단계 t 는 질의 q[t] 로 캐시 안의 키 (t 이하) 를 본다.
    policy: full (안 뗌) | lru | heavy. (출력 (n, dv), 단계마다 남긴 토큰 수).
    """
    n, d = q.shape
    scale = 1.0 / np.sqrt(d)
    keep = np.zeros(n, dtype=bool)
    score_sum = np.zeros(n)
    out = np.zeros((n, v.shape[1]))
    kept_count = np.zeros(n, dtype=np.int64)
    for t in range(n):
        keep[t] = True
        idx = np.flatnonzero(keep)
        s = (k[idx] @ q[t]) * scale
        p = np.exp(s - s.max())
        p /= p.sum()
        out[t] = p @ v[idx]
        score_sum[idx] += p                                  # 누적 점수. 헤비 히터의 근거
        kept_count[t] = len(idx)
        if len(idx) > budget:
            keep[choose_victim(idx, score_sum, policy, recent)] = False
    return out, kept_count
```

읽어 보면 `score_sum[idx] += p` 가 누적 점수이고 어텐션이 계산한 `p` 를 그대로 쓴다. `keep` 이 집합 S_t 이고 뗀 토큰은 `False` 가 된 뒤 안 돌아온다.

분석

이 구조는 캐시 축출이라는 오래된 자리에 새 정책을 넣은 것이다. 구조는 그대로다. 예산이 있고, 넘치면 하나를 뗀다. 바뀐 것은 무엇을 뗄지의 근거이고, 그 근거가 시간에서 질량으로 옮겨 갔다. 운영체제의 LRU 가 조정해서 유효한 자리다. 참조가 시간 지역성을 따를 때만 맞는 정책이고, 어텐션은 그것을 따르지 않는다.

이 구조가 서는 자리는 긴 문맥의 디코딩이다. 캐시가 예산 안에 있으면 토큰 하나의 시간이 길이와 무관하고, 그것이 5.4 의 배치가 길이에 상관없이 채워지는 조건이다.

현장에서

> 헤비 히터 오라클의 보고에서 KV 캐시를 20 퍼센트로 줄인 디코딩이 처리량을 최대 29 배 올렸고, 그 처리량의 근거는 캐시가 작아져 배치가 커진 것이었다 [확립] [60]. 이 절의 실험은 품질을 출력 오차로만 쟀고, 생성 품질은 재지 않았다.

주의

> 실험 15.3 의 헤비 히터는 필자가 만든 것이다. 앞쪽 토큰의 키를 질의 방향으로 밀었으므로 그것들이 질량을 받는 것은 설계다. 실제 모델에서 어느 토큰이 헤비 히터인지는 모델과 입력이 정하고, 그 분포가 이 실험과 다르면 오차도 다르다. 그리고 누적 점수는 과거로 미래를 어림하는 것이라, 참조 패턴이 도중에 바뀌면 옛 헤비 히터가 자리를 차지한 채 새것을 밀어낸다. 문제 15.7 이 그 자리다.

표를 떼는 법이 여기까지다. 다음 절은 표를 두지 않는다. 과거를 고정 크기의 상태로 요약하면 길이가 얼마든 읽는 것이 상태 하나이고, 그것이 5.3 의 링 버퍼가 가던 방향의 끝이다.

되짚기

> KV 캐시 축출을 예산과 정책으로 정의하고, 오차가 뗀 토큰의 질량으로 상한이 잡힘을 앞 절의 명제로 얻었다. 예산 256 에서 LRU 는 101 퍼센트, 헤비 히터는 0.84 퍼센트였고, 그 차이가 시간 지역성과 질량 지역성의 차이다. 다음 절은 고정 크기 상태의 재귀와 스캔이다.

15.4 고정 크기 상태. 재귀와 스캔

5.3 의 링 버퍼는 창 w 개만 두고 나머지를 버렸다. 창 평균은 누적합 하나로 $O(1)$ 이었다. 창 밖의 값은 잃었지만, 평균이라는 요약은 창 안의 값 전부를 다시 안 봐도 됐다. 15.3 의 축출은 토큰을 떼되 남긴 토큰은 원래대로 읽었다. 이 절은 한 걸음 더 간다. 토큰을 하나도 안 남기고, 과거 전체를 고정 크기의 수 s 개로 요약한다. 상태가 길이와 무관하고, 한 단계가 상태 크기의 곱셈이다.

그 요약이 선형 재귀이고, 같은 식을 $\log n$ 단계로 계산하는 것이 5장에서 링 버퍼 옆에 있던 누적합의 스캔이다. 추론은 재귀로 한 단계씩, 학습은 스캔으로 한꺼번에. 이 절은 둘을 세우고 잰다.

학습 목표

선형 재귀의 상태와 한 단계를 정의하고, 같은 출력을 결합 법칙으로 $\log n$ 단계에 얻는 스캔이 정확함을 증명할 수 있다. 고정 상태의 메모리가 길이와 무관함을 KV 캐시와 나란히 적고, 스캔이 직렬 기계에서 재귀보다 느린 이유를 이동량으로 말할 수 있다.

이 말부터

선형 재귀	$h_t = a \odot h_{t-1} + b\,x_t$. 상태 h 가 s 개의 수이고 a 가 감쇠
스캔	결합 법칙이 있는 연산의 누적을 두 배씩 건너뛰며 $\log n$ 단계에 얻는 것
상태 공간 모델	선형 재귀를 층으로 쓰는 시퀀스 모델. 어텐션 대신 상태를 둔다

디딤돌

디딤돌은 5장의 누적합이다. $S_t = S_{t-1} + x_t$ 는 재귀이고, 같은 S 를 병렬로 얻는 것이 스캔이었다. 덧셈에 결합 법칙이 있어서 가능했다. 선형 재귀도 결합 법칙이 있는 연산이고, 그래서 같은 스캔이 된다.

정의 15.11: 선형 재귀와 고정 상태

상태 $h_t \in \mathbb{R}^s$, 감쇠 $a \in (0,1)^s$, 입력 사영 $b \in \mathbb{R}^s$, 출력 사영 $c \in \mathbb{R}^s$ 에 대해 $h_t = a \odot h_{t-1} + b\,x_t$, $y_t = c^\top h_t$ 이다. $h_0 = 0$ 이다.

불변식 h_t 는 과거 입력 전부의 함수이지만 크기가 s 로 고정이다. 단계 하나는 곱셈 $2s$ 개이고 길이 t 와 무관하다.

불변식의 두 문장이 어텐션과의 차이 전부다. 어텐션의 상태는 토큰 t 개의 키와 값이라 t 에 비례하고, 여기서는 s 다. 대신 과거는 요약으로만 남고, 어느 토큰을 정확히 되찾는 일은 못 한다.

그림 15.6.. 고정 상태. 토큰이 몇 개 지나가든 상태는 칸 여덟 개이고 단계마다 덮어쓴다. KV 캐시라면 토큰마다 칸이 는다

명제 15.12: 두 상태의 크기

> 층 L, 헤드 H, 차원 d 의 어텐션은 토큰 n 개에서 KV 캐시 $2LHdn$ 원소를 둔다. 층 L, 채널 C, 상태 s 의 선형 재귀는 LCs 원소를 두고 n 이 안 든다. 디코딩 한 단계의 이동량은 앞이 $2LHdn$ 원소이고 뒤가 LCs 원소다. 따라서 앞은 길이에 선형으로 느려지고 뒤는 상수다.

증명. KV 캐시는 2.3 의 셈 그대로 토큰마다 층과 헤드와 차원의 곱에 키와 값 둘이다. 선형 재귀는 채널마다 상태 s 개이고 토큰이 늘어도 덮어쓴다. 한 단계가 상태 전부를 읽고 쓰므로 이동량이 상태 크기다. ■

명제를 실험의 조건에 넣으면 이천 배가 나온다. 그 대가가 무엇인지는 이 절의 분석에서 적고, 먼저 예제로 두 크기를 적은 뒤 학습 쪽의 계산을 세운다.

예제 15.4: 토큰 33 개

> 층 32, 헤드 32, 차원 128, 2 바이트의 KV 캐시는 토큰 하나에 512 KB 다. 층 32, 채널 4,096, 상태 64 의 고정 상태는 16.8 MB 다. 토큰 33 개만 지나도 캐시가 상태보다 크고, 65,536 개에서는 34 GB 대 16.8 MB 로 이천 배다.

예제의 앞 문장이 추론의 이유이고, 학습에서는 이 상태를 길이 전체에 대해 한꺼번에 얻어야 한다. 그것이 다음 정리다.

정리 15.13: 선형 재귀의 스캔

> 쌍 (A, B) 에 연산 $(A_1, B_1) \circ (A_2, B_2) = (A_1 \odot A_2,\ A_2 \odot B_1 + B_2)$ 를 두면 이 연산은 결합 법칙을 만족한다. 원소 t 를 $(a, b\,x_t)$ 로 두고 앞에서부터의 누적 $\circ$ 를 취하면 그 둘째 성분이 h_t 다. 따라서 $h_1, \dots, h_n$ 전부를 두 배씩 건너뛰는 스캔으로 $\lceil \log_2 n \rceil$ 단계에 얻으며, 단계마다 배열 전체 ns 원소의 곱셈과 덧셈이다.

증명. 쌍 (A, B) 는 사상 $h \mapsto A \odot h + B$ 이고, 두 사상의 합성이 $h \mapsto A_2 \odot (A_1 \odot h + B_1) + B_2$ 라 위의 연산이다. 사상의 합성은 결합적이다. 원소 t 의 사상은 정의의 한 단계이고, 처음부터 t 까지의 합성을 $h_0 = 0$ 에 적용하면 h_t 이며 그것이 둘째 성분이다. 결합적이므로 5장의 스캔이 그대로 적용된다. 단계 k 에서 원소 t 가 원소 $t - 2^k$ 의 누적과 합성되고, $\log_2 n$ 단계 뒤에 모든 원소가 처음부터의 누적을 갖는다. ■

정리의 마지막 문장이 이동량이다. 단계 $\log n$ 개에 단계마다 ns 원소를 읽고 쓰므로 전체 이동량이 $ns \log n$ 이고, 재귀는 n 단계에 단계마다 s 라 ns 다. 스캔이 이동량은 $\log n$ 배이고 단계 수는 $n / \log n$ 분의 1 이다. 어느 쪽이 빠른지는 기계가 정한다.

균열

재귀와 스캔을 길이 4,096 에서 65,536 까지 쟀다. 그리고 같은 길이에서 어텐션의 KV 캐시와 재귀의 상태를 명제 15.12 로 나란히 적었다.

실험 15.4 재귀와 스캔의 시간, 두 상태의 크기

길이 n	재귀 ms	스캔 ms	스캔 단계	고정 상태 MB	KV 캐시 MB
4096	44.3	37.4	12	16.8	2147
16384	142	256	14	16.8	8590
65536	583	1330	16	16.8	34360

측정 조건. 상태 64, 대각 감쇠. 재귀는 파이썬 루프로 단계 n 개, 스캔은 NumPy 로 단계 $\log_2 n$ 개. 메모리는 층 32, 헤드 32, 차원 128, 채널 4,096, 2 바이트로 셈한 값. 세 번 중앙값.

길이 4,096 에서 스캔이 조금 빠르고, 65,536 에서 스캔이 두 배 느리다. 재귀는 단계당 8 마이크로초의 파이썬 루프이고 길이에 정확히 선형이다. 스캔은 단계 16 개인데 단계마다 33 MB 짜리 배열 몇 개를 읽고 쓰므로, 단계당 이동이 GB 단위다. 메모리는 34 GB 와 16.8 MB 다. 시간이 아니라 이 열이 이 구조의 이유다.

관찰

스캔은 단계 수를 n 에서 $\log n$ 으로 줄이고 이동량을 $\log n$ 배 늘린다. 단계가 병렬로 도는 기계에서는 단계 수가 시간이고, 직렬 기계에서는 이동량이 시간이다. NumPy 는 뒤쪽이다.

이 관찰이 5장의 스캔에 붙여 둘 조건이다. 스캔은 병렬 기계의 알고리즘이고, 그 사실이 이 실험에서 숫자로 나왔다. 학습이 스캔을 쓰는 것은 가속기가 단계 하나를 병렬로 돌리기 때문이다.

다시 쓰기

이 절의 다시 쓰기는 상태를 둘 곳을 고르는 것이다. 어텐션은 상태를 토큰에 두고, 재귀는 상태를 고정된 수에 둔다. 둘 사이에 한 명제가 있다.

명제 15.14: 되찾기와 요약

어텐션은 임의의 과거 토큰 i 의 값 V_i 를 그 키에 가까운 질의로 정확히 되찾는다. 선형 재귀는 과거를 s 개의 수로 요약하므로 s 보다 많은 과거 토큰을 서로 구별해 되찾을 수 없다. 따라서 되찾기가 필요한 일은 어텐션이 맞고, 요약으로 충분한 일은 재귀가 맞다. 둘을 층으로 섞는 것이 그 맞바꿈을 층마다 다르게 두는 것이다.

증명. 어텐션의 출력은 값들의 볼록 결합이고, 질의가 키 i 와만 큰 점수를 내면 출력이 V_i 에 가깝다. 재귀의 상태는 s 차원이라 선형 독립인 벡터를 s 개까지만 담고, 그 이상의 토큰은 어떤 사영으로도 전부 구별되지 않는다. ■

Gu 와 Dao 의 2023 년 논문은 감쇠 a 와 사영 b 를 입력에 따라 바꾸는 선택적 상태 공간을 두어, 요약이 무엇을 남기고 무엇을 잊을지를 토큰마다 정하게 했다 [확립] [62]. 상태의 크기는 그대로 s 이고, 이 절의 정리와 명제가 그대로 적용된다.

알고리즘 15.4 선형 재귀의 스캔 `dsai.ssm.run_scan`

입력 : 입력 $x_1, \dots, x_n$, 감쇠 a, 사영 b, c
$A_t \leftarrow a$, $B_t \leftarrow b\,x_t$ (모든 t)
반복 $k = 1, 2, 4, \cdots < n$ **동안**
  $(A_t, B_t) \leftarrow (A_{t-k}, B_{t-k}) \circ (A_t, B_t)$ (모든 $t \ge k$, 한꺼번에)
끝 반복
반환 $y_t = c^{\top} B_t$

구현

한 단계의 재귀와 스캔이다. 재귀는 곱셈 $2s$ 개이고, 스캔은 배열 전체의 곱셈을 $\log n$ 번이다.

지금 보는 코드 상태 h 를 감쇠하고 입력을 더한다. 한 단계가 이 한 줄이다.

구현 한 단계와 재귀 `samples/dsai/ssm.py:13`

```
def recurrent_step(h: np.ndarray, x: np.ndarray, a: np.ndarray,
                   b: np.ndarray) -> np.ndarray:
    """정의 15.4. 대각 A 의 한 단계. h, a, b 는 상태 크기 s. 곱셈 2s 개."""
    return a * h + b * x


def run_recurrent(x: np.ndarray, a: np.ndarray, b: np.ndarray,
                  c: np.ndarray) -> np.ndarray:
    """디딤돌. 길이 n 의 입력을 한 단계씩. 상태는 언제나 s 개다."""
    n = len(x)
    h = np.zeros(len(a))
    y = np.empty(n)
    for t in range(n):
        h = recurrent_step(h, x[t], a, b)
        y[t] = c @ h
    return y
```

읽어 보면 `a * h + b * x` 가 정의다. `h` 의 크기가 `len(a)` 로 고정이고 `t` 가 얼마든 같다. 이 루프가 디코딩이다.

구현 스캔 `samples/dsai/ssm.py:31`

```python
def run_scan(x: np.ndarray, a: np.ndarray, b: np.ndarray, c: np.ndarray) -> np.ndarray:
    """알고리즘 15.3. 결합 법칙으로 푸는 스캔. 5장의 누적합과 같은 두 배씩 건너뛰기.
    (A, B) 뒤에 (A', B') 를 붙이면 (A A', A' B + B') 이다.
    log2 n 단계이고 단계마다 배열 전체를 만진다.
    """
    n = len(x)
    A = np.tile(a, (n, 1))                    # (n, s). 원소 t 는 h_t = A h + B
    B = b[None, :] * x[:, None]
    step = 1
    while step < n:
        A_new, B_new = A.copy(), B.copy()
        A_new[step:] = A[:-step] * A[step:]
        B_new[step:] = A[step:] * B[:-step] + B[step:]
        A, B = A_new, B_new
        step *= 2
    return B @ c
```

읽어 보면 `step` 이 두 배씩 커지는 것이 정리 15.13 의 2^k 다. `A[:-step] * A[step:]` 와 `A[step:] * B[:-step] + B[step:]` 가 연산 ∘ 그대로이고, 복사 둘이 실험의 이동량이다.

분석

이 구조는 자료 구조의 잣대로 보면 상태를 어디에 두느냐의 문제다. 어텐션은 상태를 추가 전용 배열에 두어 길이에 비례하고, 재귀는 고정 배열 하나에 두어 상수다. 7장의 스케치가 개수를 고정 크기에 요약하던 것과 같은 자리이고, 잃는 것도 같다. 정확한 되찾기다.

이 구조가 서는 자리는 긴 시퀀스의 추론이다. 상태가 16.8 MB 이면 토큰 백만 개 뒤에도 단계 하나의 이동이 16.8 MB 이고, 캐시를 떼거나 페이지로 나눌 일이 없다. 대가는 명제 15.14 이고, 그 대가를 어디까지 치를지는 자료 구조가 아니라 모델이 정한다.

현장에서

> Mamba 의 보고에서 선택적 상태 공간 모델은 어텐션 모델과 같은 크기에서 시퀀스 길이에 선형인 시간을 보였고, 추론에서 단계당 시간이 길이와 무관했다 [확립] [62]. 그 논문이 스캔을 가속기의 빠른 메모리 안에서 돌린 것이 15.2 의 타일과 같은 장치다.

주의

> 실험 15.4 의 스캔은 배열을 매 단계 복사한다. 제자리 갱신이면 이동량이 절반이고, 블록 안을 닫힌 식으로 두고 블록 사이만 스캔하면 5장의 두 단계 스캔이 되어 이동량이 다시 준다. 문제 15.8 이 그 자리다. 그리고 감쇠 a 가 대각이라 곱셈이 원소별이었다. 일반 행렬이면 단계 하나가 s^2 이고 스캔의 합성도 행렬 곱이다.

어텐션을 네 절로 읽었다. 표를 읽고, 행렬 없이 읽고, 표를 떼고, 표를 요약했다. 네 절 모두 잣대는 하나였다. 질의 하나가 읽는 바이트다. 다음 장은 기계 하나를 넘어간다. 기계 여럿이 같은 배열을 나눠 갖고 합칠 때의 자료 구조다.

되짚기

선형 재귀를 고정 크기 상태의 한 단계로 정의하고, 결합 법칙으로 $\log n$ 단계에 같은 출력을 얻는 스캔을 증명했다. 상태는 KV 캐시 34 GB 에 대해 16.8 MB 였고, 직렬 기계에서 스캔이 재귀보다 느린 이유는 이동량 $\log n$ 배였다. 다음 장은 분산과 저장이다.

15장을 닫으며

이 장이 세운 것은 어텐션을 보는 잣대 하나다. 질의 하나가 읽는 바이트. 네 절이 그 바이트를 각각 다르게 줄였고, 줄이는 값이 각각 달랐다. 타일은 값이 없고, 축출은 뗀 질량이 값이며, 요약은 되찾기가 값이다.

네 실험의 숫자를 적어 둔다. 키 백만 개의 어텐션 조회가 42 밀리초였고 해시 조회는 0.3 마이크로초였다. 길이 16,384 에서 점수 행렬 1 GB 가 타일로 4 MB 가 되었고 시간은 8 퍼센트 줄었다. 예산 256 의 축출에서 LRU 의 오차가 101 퍼센트, 헤비 히터가 0.84 퍼센트였다. 길이 65,536 에서 KV 캐시 34 GB 에 대해 고정 상태가 16.8 MB 였고, 스캔은 재귀보다 두 배 느렸다.

정정 하나를 남긴다. 상위 k 근사의 실험을 처음에는 무작위 키에 무작위 질의로 돌렸다. 점수가 모두 비슷해 소프트맥스가 평평했고, 상위 4,096 개로도 오차가 수백 퍼센트였다. 그것은 근사가 나쁜 것이 아니라 근사할 것이 없는 것이었다. 질의를 키 하나의 배수로 두어 점수를 뾰족하게 만들자 오차가 명제대로 질량을 따랐다. 실험이 재는 것은 자료 구조이고, 소프트맥스가 뾰족한지는 모델이 정한다는 것을 그때 적었다.

옛 구조의 판정

캐시 축출의 LRU 는 조정해서 유효하다. 구조는 그대로이고 정책의 근거가 시간 지역성에서 어텐션 질량으로 바뀌었다. 근거는 실험 15.3 이다. 같은 예산에서 오차가 백 배 갈렸고, 갈린 이유는 오래된 토큰 몇 개가 질량의 절반을 받는 참조 패턴이었다. 스캔은 그대로 유효하되 조건이 붙는다. 병렬 기계의 알고리즘이고, 직렬 기계에서는 이동량 $\log n$ 배가 그대로 시간이다.

이 장의 요약

1. 어텐션은 모든 키가 닮은 만큼 값을 주는 연관 배열 읽기다. 조회가 키와 값의 바이트를 대역폭으로 나눈 시간이다 15.1
2. 남긴 키의 질량이 $1 - \varepsilon$ 이면 오차가 $2\varepsilon \max \|V\|$ 이하다. 12장의 kNN 과 축출이 같은 명제 위에 있다 15.1
3. 온라인 소프트맥스는 최댓값과 합을 상태로 두어 축소 둘의 경계를 지운다. 점수 행렬이 메모리에 없다 15.2
4. 헤비 히터와 최근 창의 두 집합이 예산 안의 캐시다. LRU 는 어텐션의 참조 패턴에 맞지 않는다 15.3
5. 선형 재귀는 상태가 고정이고, 결합 법칙으로 $\log n$ 단계의 스캔이 된다. 되찾기를 요약과 맞바꾼다 15.4

다음 장에서 뒤집히는 것: 이 장까지의 배열은 기계 하나 안에 있었다. 16장의 배열은 기계 여럿에 나뉘고, 합치는 일이 이동량이 된다. 링 위에서 조각을 돌리는 것이 그 첫 구조다.

필자의 판단

필자는 어텐션을 자료 구조 수업의 마지막 주에 둔다. 해시 테이블과 캐시와 누적합을 배운 학생에게 어텐션은 새 것이 아니라 그 셋을 한 자리에 놓은 것이고, 그렇게 보면 FlashAttention 과 캐시 축출과 상태 공간 모델이 각각 어느 디딤돌 위에 서 있는지가 보인다. 새 정리는 둘뿐이고 둘 다 한 쪽이다. 그것이 이 장을 자료 구조 책에 둘 수 있는 이유다.

열린 문제

하나, 실험 15.2 는 질의를 자르지 않았다. 질의 블록까지 자른 이차원 타일에서 이동량이 얼마이고 시간이 얼마인지 재지 않았다.
둘, 실험 15.3 의 헤비 히터는 필자가 만든 참조 패턴이다. 실제 모델의 어텐션 질량이 어느 토큰에 얼마나 모이는지를 이 책은 재지 않았고, 그것이 축출 정책의 값을 정한다.
셋, 실험 15.4 의 스캔은 NumPy 의 직렬 실행이라 병렬 기계에서 드는 시간을 말하지 못한다. 가속기에서 스캔 단계 하나가 얼마인지는 이 책의 도구 밖이다.

연습문제

15.1 기초 길이 32,768, 차원 128, 헤드 32 의 점수 행렬이 float16 으로 몇 GB 인가. 정의 15.1 으로 셈하고, 타일 128 의 블록 점수는 몇 KB 인지 적어라.

15.2 기초 길이 8,192, 차원 128 에서 점수 행렬 어텐션과 타일 어텐션의 이동량을 명제 15.6 로 구하고 배수를 적어라. 연산 밀도가 몇 배 오르는가.

15.3 기초 예산 B, 보호 r 의 헤비 히터 캐시에서 어느 토큰이 남는지 정의 15.8 으로 적어라. 오래됐지만 점수가 큰 토큰은 어느 정책에서 남고 어느 정책에서 나가는가.

15.4 기초 층 32, 헤드 32, 차원 128 의 KV 캐시가 토큰 하나에 몇 KB 이고 길이 10^5 에서 몇 GB 인가. 층 32, 채널 4,096, 상태 64 의 고정 상태는 몇 MB 인가. 명제 15.12 로 구하라.

15.5 응용 `dsai.attention.attention_tiled` 에 인과 마스크를 더하라. 질의 블록보다 뒤의 키 블록은 건너뛰고, 블록 안의 미래는 $-\infty$ 로 막아라. 전부 유지한 인과 어텐션과 같은지 시험하고 건너뛴 블록 수를 세어라.

15.6 응용 타일 크기를 8 에서 2,048 까지 바꾸며 `attention_tiled` 의 시간을 재라. 어느 크기에서 가장 빠르고, 작을 때와 클 때 무엇이 시간을 먹는가.

15.7 응용 `dsai.kvevict.decode_with_policy` 의 누적 점수를 지수 감쇠 합 $\sigma \leftarrow \lambda\sigma + p$ 로 바꿔라. $\lambda = 1$ 에서 원래와 같은지 시험하고, 참조 패턴이 도중에 바뀌는 열에서 오차를 견줘라.

15.8 응용 `dsai.ssm.run_scan` 을 블록 안은 닫힌 식, 블록 사이는 재귀로 바꿔라. 5장 누적합의 두 단계다. 재귀와 같은 출력인지 시험하고 블록 크기에 따른 시간을 재라.

15.9 탐구 명제 15.14 를 근거로, 어텐션과 선형 재귀를 층마다 섞은 모델에서 어느 층에 어느 쪽을 두어야 하는지 논하라. 되찾기가 필요한 층과 요약으로 충분한 층을 어떻게 가리는가.

15.10 탐구 연관 배열의 읽기와 어텐션의 읽기가 어디서 갈라지는지 정의 15.1 으로 적고, 12장의 근접 탐색과 15.3 절의 축출이 같은 명제의 두 얼굴인 이유를 논하라.

더 읽을 것

[57] 어텐션의 원전. 질의, 키, 값의 표기와 $\sqrt{d}$ 의 이유가 있다. 연관 배열로 읽는 출발점이다. 15.1
[59] 온라인 정규화. 최댓값과 합을 한 번 훑기로 얻는 갱신이 정리 그대로 있다. 15.2
[58] FlashAttention. 온라인 소프트맥스로 점수 행렬을 느린 메모리에 안 쓰는 타일 어텐션. 역방향의 재계산도 여기 있다. 15.2
[60] 헤비 히터 오라클. 누적 어텐션 점수로 캐시를 예산 안에 두는 정책과 그 처리량. 15.3

[61] 어텐션 싱크. 첫 토큰 몇 개와 최근 창만 남기는 스트리밍 캐시. 15.3 절의 두 집합에서 헤비 히터를 고정한 경우다. 15.3
[62] 선택적 상태 공간. 고정 상태의 재귀를 입력에 따라 바꾸고, 스캔을 빠른 메모리 안에서 돌린다. 15.4

7부

규모와 선택

규모는 기계 하나를 넘어간 배열이다. 7부는 기계 여럿이 나눠 가진 배열을 합치고 저장하고 읽고 섞는 구조를 세운 뒤, 이 책이 세운 구조 전부를 한 잣대로 다시 늘어놓는다. 어떤 구조를 언제 고르는가가 마지막 장이다.

16장 분산과 저장 17장 자료 구조를 고르는 일

16장

분산과 저장

1964년의 논문은 다섯 줄이었다 [확립] [63]. 배열을 뒤에서부터 훑으며 무작위 자리와 바꾸면 균등한 순열이 나온다. 60년 뒤 모든 학습이 매 에폭 그 다섯 줄을 돌린다. 이 장은 기계 하나를 넘어간 배열을 다룬다. 기계 여덟 대가 나눠 가진 기울기를 합치고, 한 시간마다 저장하며, 열 단위로 읽고, 에폭마다 섞는다. 넷 다 배열 하나에 하는 일이고, 잣대는 여전히 0.1 의 이동량이다. 링크의 바이트, 디스크의 바이트, 페이지의 바이트.

이 장의 네 구조는 전부 앞 장의 것이다. 링 버퍼가 기계 위의 링이 되고, 해시가 저장의 주소가 되며, 스트라이드가 파일의 놓임이 되고, 뽑기가 순열이 된다. 새 정리는 하나이고, 나머지는 명제 몇 개다. 자료 구조가 기계를 넘어가도 구조는 안 바뀐다. 바이트의 단위가 바뀐다.

이 장에서 세우는 것

16.1 링 all-reduce. 기계 하나가 보내는 것이 벡터 두 개 이하이고 그것이 하한. 단계 지연이 조각 200 KB 아래에서 시간

16.2 머클 트리와 내용 주소 저장. 블록 4 KB 가 1 MB 보다 스무 배 적게 저장, 트리는 다른 블록이 드물 때만 로그

16.3 열 지향 저장과 메모리 맵. 열 하나에 바이트 8 배, 시간 3 배에서 6 배

16.4 셔플과 순열. 균등성의 증명, 블록 섞기의 순차, 저장 없는 Feistel 순열

16.1 링 위의 all-reduce. 조각을 돌린다

여기까지의 배열은 기계 하나 안에 있었다. 매개변수 70억 개는 2 바이트로 14 GB 이고, 그 기울기도 14 GB 다. 기계 여덟 대가 데이터를 나눠 학습하면 기계마다 기울기 14 GB 가 생기고, 여덟 개의 합을 여덟 대가 전부 가져야 한다. 그것이 all-reduce 이고, 학습의 매 단계마다 한 번이다. 0.1 의 잣대를 기계 사이의 링크에 적용하면 이 연산의 시간은 링크 하나가 보내는 바이트다.

교과서의 방법은 하나가 모아서 나눠 주는 별이다. 뿌리 기계의 링크에 벡터 $2(p-1)$ 개가 지나가고, 나머지 링크는 논다. 링은 벡터를 p 조각으로 잘라 조각을 돌린다. 모든 링크가 고르게 들고, 기계 하나가 보내는 양이 벡터 두 개를 넘지 않는다.

학습 목표

all-reduce 를 정의하고 별과 링에서 기계 하나가 보내는 바이트를 명제로 적을 수 있다. 링이 대역폭에서 최적인 이유를 하한으로 보이고, 단계 수 $2(p-1)$ 이 지연으로 언제 시간을 정하는지를 모형과 실측으로 말할 수 있다.

이 말부터

all-reduce	기계 p 대의 벡터를 원소별로 합쳐 모든 기계가 그 합을 갖게 하는 집합 통신
reduce-scatter	벡터를 p 조각으로 나눠 조각 i 의 합을 기계 i 가 갖게 하는 첫 반
all-gather	기계마다 하나씩 있는 조각을 모두가 전부 갖게 하는 둘째 반

디딤돌

디딤돌은 배열의 조각이다. 1.1 의 배열은 주소가 연속이라 어디서든 잘라 보낼 수 있고, 조각 p 개의 경계는 나눗셈 하나다. 링은 그 조각을 이웃에게 보내는 것뿐이다. 새것은 보내는 순서다.

정의 16.1: all-reduce 와 두 방법

기계 i 가 벡터 $x_i \in \mathbb{R}^n$ 을 갖고, all-reduce 는 모든 기계가 $\sum_i x_i$ 를 갖게 하는 것이다. 별은 기계 0 이 $x_1, \dots, x_{p-1}$ 을 받아 더하고 결과를 $p-1$ 대에 보낸다. 링은 벡터를 조각 $c_0, \dots, c_{p-1}$ 로 나누고, 단계 $t = 0, \dots, p-2$ 에 기계 i 가 조각 $c_{(i-t) \bmod p}$ 의 부분합을 기계 $i+1$ 에 보내며 받은 쪽이 자기 것을 더한다 (reduce-scatter). 그 뒤 완성된 조각을 같은 방향으로 $p-1$ 단계 더 돌린다 (all-gather).

불변식 reduce-scatter 가 끝나면 기계 i 가 조각 $c_{(i+1) \bmod p}$ 의 완전한 합을 갖는다. 단계마다 모든 링크에 조각 하나씩이 지나간다.

불변식의 둘째 문장이 링의 전부다. 어느 단계에서도 노는 링크가 없고, 지나가는 것은 조각 하나다. 그것을 바이트로 세면 명제가 된다.

명제 16.2: 기계 하나가 보내는 바이트

> 원소 n 개, 원소 크기 s 에서 별의 뿌리는 $2(p-1)ns$ 바이트를 보내고 받으며 단계는 2 다. 링은 모든 기계가 $2(p-1)ns/p$ 바이트를 보내고, 단계는 $2(p-1)$ 이다. 따라서 p 가 커질 때 별의 뿌리는 p 에 선형으로 늘고, 링은 $2ns$ 에 수렴한다.

증명. 별의 뿌리는 벡터 $p-1$ 개를 받고 $p-1$ 개를 보낸다. 링은 단계마다 조각 ns/p 바이트를 보내고 단계가 $2(p-1)$ 이다. $2(p-1)/p < 2$ 이고 $p \to \infty$ 에서 2 다. ■

명제의 셋째 문장이 링을 고른 이유다. 기계를 천 대로 늘려도 기계 하나가 보내는 양이 벡터 두 개를 안 넘는다. 이것이 우연이 아니라 하한이라는 것이 다음 명제다.

명제 16.3: 대역폭 하한

> 어떤 all-reduce 알고리즘이든 기계 하나는 적어도 $(p-1)ns/p$ 바이트를 받아야 하고, 결과를 모두가 가져야 하므로 그만큼을 또 받아야 한다. 따라서 기계 하나의 수신 바이트는 $2(p-1)ns/p$ 이상이고 링은 이 하한을 달성한다.

증명. 합의 각 원소는 p 개의 값에 달렸고, 기계 i 는 그중 자기 것 하나만 갖고 있다. 원소 하나의 합을 어느 기계가 알려면 나머지 $p-1$ 개의 정보가 그 기계에 닿아야 한다. 정보를 p 대가 나눠 모으면 기계 하나가 받는 최소가 원소당 $(p-1)/p$ 이고, 모은 결과를 모두에게 퍼뜨리는 데 다시 원소당 $(p-1)/p$ 가 든다. Patarasuk 과 Yuan 이 이 하한을 적었고 링이 그것을 달성함을 보였다 [확립] [64]. ■

하한이 말하는 것은 링보다 적게 보내는 방법이 없다는 것이다. 남는 것은 그 하한이 실제 숫자로 얼마인지이고, 학습 하나의 조건을 넣어 본다.

예제 16.1: 기계 여덟 대와 14 GB

> $ns = 14$ GB, $p = 8$ 이면 별의 뿌리가 196 GB 를 보내고 받고, 링은 기계마다 24.5 GB 다. 링크가 400 GB/s 이면 490 밀리초와 61 밀리초다. 단계 지연이 20 마이크로초이면 링의 단계 14 개가 0.3 밀리초를 더한다. 벡터가 크면 지연은 안 보인다.

그림 16.1.. 링 all-reduce. 기계 넷이 조각 넷을 돌린다. 단계 t 에 기계 i 가 조각 $(i-t)$ 를 오른쪽에 보내고, 세 단계 뒤에 기계 i 가 조각 $(i+1)$ 의 합을 갖는다

균열

기계 2 대에서 64 대까지 별과 링을 시뮬레이션했다. 바이트는 실제로 보낸 것을 셌고, 시간은 대역폭과 단계 지연의 모형이다. 두 방법의 합이 같은지를 먼저 확인했고, 그다음에 바이트를 세었다.

실험 16.1 별과 링. 기계 하나가 보내는 바이트와 모형 시간

기계 p	별 뿌리 MB	링 링크 MB	별 단계	링 단계	별 모형 ms	링 모형 ms
2	16.0	8.00	2	2	1.64	0.84
4	48.0	12.0	2	6	4.84	1.32
8	112	14.0	2	14	11.2	1.68
16	240	15.0	2	30	24.0	2.10
64	1008	15.8	2	126	101	4.09

측정 조건. 벡터 100 만 개 (float64, 8 MB). 바이트는 시뮬레이션에서 센 값이고 두 방법의 합이 같음을 확인. 모형 시간은 대역폭 10 GB/s 와 단계당 지연 20 마이크로초. 실측이 아니라 모형이다.

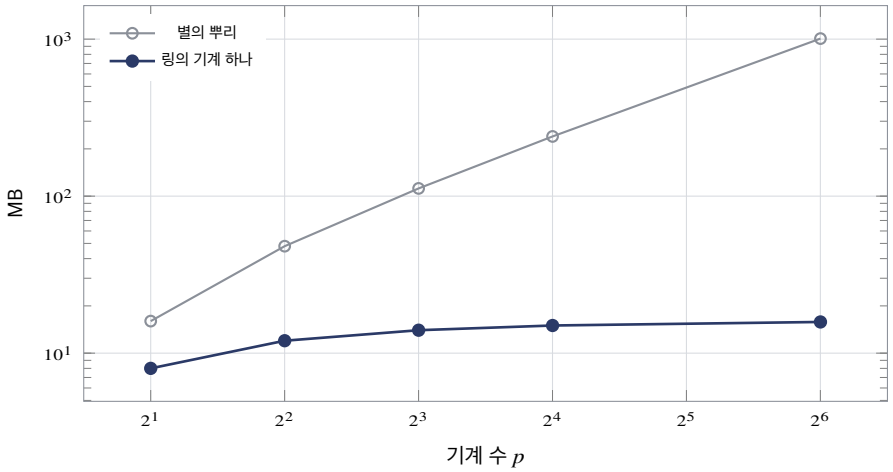


그림 16.2. 기계 수에 따른 기계 하나의 바이트. 별의 뿌리는 선형으로 늘고 링은 16 MB 아래에 머문다

기계 64 대에서 별의 뿌리가 1,008 MB 를 지나가고 링은 15.8 MB 다. 링은 명제의 $2 \times 8 \times 63/64$ 그대로이고, 64 배 차이다. 모형 시간은 101 밀리초와 4 밀리초다. 링의 단계 126 개가 지연으로 2.5 밀리초를 더했다. 조각이 125 KB 라 조각 하나의 전송이 12 마이크로초이고, 지연 20 마이크로초가 그보다 크다. 벡터가 작으면 링은 지연이 시간이다.

관찰

링의 바이트는 p 와 무관하지만 단계는 p 에 선형이다. 조각의 전송 시간이 단계 지연보다 작아지는 p 부터 링의 시간을 지연이 정한다.

이 관찰이 링의 조건이다. 벡터가 크고 기계가 수십 대일 때 링이고, 벡터가 작고 기계가 수백 대이면 단계가 $\log p$ 인 트리가 낫다. 문제 16.5 가 그 트리다.

다시 쓰기

이 절의 다시 쓰기는 0.1 의 이동량을 링크 위로 옮기는 것이다. 메모리와 캐시 사이의 바이트가 기계와 기계 사이의 바이트가 되고, 대역폭과 지연이 그대로 두 항이다.

명제 16.4: 링의 시간 모형

> 링크 대역폭 β 바이트 매초, 단계 지연 α 초에서 링 all-reduce 의 시간은 $2(p-1)\left(\alpha + ns/(p\beta)\right)$ 다. $ns/p > \alpha\beta$ 이면 대역폭 항이, 아니면 지연 항이 지배한다. 경계는 조각 크기가 $\alpha\beta$ 인 자리이고, 10 GB/s 와 20 마이크로초에서 200 KB 다.

증명. 단계마다 조각 ns/p 바이트를 보내는 시간이 $ns/(p\beta)$ 이고 지연이 α 다. 단계가 $2(p-1)$ 이다. 두 항이 같아지는 조각 크기가 $\alpha\beta$ 다. ■

이 모형이 1.4 의 배치와 같은 모양이다. 거기서는 호출 하나의 고정 값이 있어 배치를 키웠고, 여기서는 단계 하나의 지연이 있어 조각을 키운다. 벡터 여럿을 하나로 합쳐 보내는 것이 그 배치이고, 모든 분산 학습이 그렇게 한다.

알고리즘 16.1 링 all-reduce `dsai.collective.allreduce_ring`

> **입력 :** 기계 p 대의 벡터, 조각 $c_0, \dots, c_{p-1}$
> **반복** $t = 0$ **to** $p-2$ **동안** ▷ reduce-scatter
> 기계 i 가 조각 $c_{(i-t) \bmod p}$ 를 기계 $i+1$ 에 보내고, 받은 기계가 자기 조각에 더한다
> **끝 반복**
> **반복** $t = 0$ **to** $p-2$ **동안** ▷ all-gather
> 기계 i 가 조각 $c_{(i+1-t) \bmod p}$ 를 기계 $i+1$ 에 보내고, 받은 기계가 덮어쓴다
> **끝 반복**

구현

시뮬레이션이다. 기계는 배열이고 링크는 복사이며, 보낸 바이트를 센다.

지금 보는 코드 단계마다 모든 기계가 조각 하나를 오른쪽에 보내고, 앞 반은 더하고 뒤 반은 덮어쓴다.

구현 링의 한 단계 `samples/dsai/collective.py:27`

```python
def ring_step(buf: list[np.ndarray], chunks: list[np.ndarray], t: int, shift: int,
              add: bool) -> int:
    """단계 t. 기계 i 가 조각 (i + shift - t) 를 오른쪽 이웃에게 보낸다.
    add 면 이웃이 더하고, 아니면 덮어쓴다. 링크 하나가 나른 바이트를 돌려준다.
    """
    p = len(buf)
    sends = [(i, chunks[(i + shift - t) % p]) for i in range(p)]
    data = [buf[i][c].copy() for i, c in sends]           # 동시에 보내므로 먼저 복사
    for (i, c), d in zip(sends, data):
        if add:
            buf[(i + 1) % p][c] += d
        else:
            buf[(i + 1) % p][c] = d
    return max(d.nbytes for d in data)
```

읽어 보면 `(i + shift - t) % p` 가 단계마다 한 칸씩 도는 조각 번호다. 모든 기계가 동시에 보내므로 먼저 복사하고 그다음에 더하거나 덮어쓴다. 돌려주는 값이 링크 하나의 바이트다.

구현 링 all-reduce `samples/dsai/collective.py:43`

```python
def allreduce_ring(shards: list[np.ndarray]) -> tuple[np.ndarray, int, int]:
    """알고리즘 16.1. reduce-scatter 와 all-gather. 조각 p 개를 링으로 돌린다.
    (합, 링크 하나의 바이트, 단계 수).
    """
    p, n = len(shards), len(shards[0])
    chunks = np.array_split(np.arange(n), p)
    buf = [s.copy() for s in shards]
    per_link = 0
    for t in range(p - 1):                                # reduce-scatter: 더한다
        per_link = max(per_link, ring_step(buf, chunks, t, 0, True))
    for t in range(p - 1):                                # all-gather: 덮어쓴다
        ring_step(buf, chunks, t, 1, False)
    return buf[0], per_link * 2 * (p - 1), 2 * (p - 1)
```

읽어 보면 `add` 가 참인 앞 루프가 reduce-scatter 이고 거짓인 뒤 루프가 all-gather 다. 단계가 $2(p-1)$ 이고 `per_link` 에 단계 수를 곱한 것이 명제 16.2 의 바이트다.

분석

이 구조는 배열 하나를 p 조각으로 나눈 것과 이웃에게 보내는 규칙 하나다. 자료 구조로는 5장의 링 버퍼가 기계 위에 놓인 것이다. 5.3 의 링이 시간 위의 창이었다면 이 링은 기계 위의 창이고, 조각이 한 칸씩 돈다.

이 구조가 서는 자리는 데이터 병렬 학습의 매 단계다. 기울기 all-reduce 가 단계마다 한 번이고, 그 시간이 계산과 겹치지 않으면 학습 시간에 그대로 더해진다. 그래서 층마다 기울기가 나오는 즉시 뒤에서부터 링을 돌리고, 그것이 11.2 의 역순 위에서 일어난다.

현장에서

Patarasuk 과 Yuan 의 2009 년 논문은 링 all-reduce 가 대역폭 최적임을 증명하고, 클러스터 실측에서 벡터가 클 때 트리 기반보다 빠름을 보였다 [확립] [64]. 이 절의 하한이 그 논문의 정리이고, 실험은 그 정리를 모형으로 옮긴 것이다.

주의

실험 16.1 의 시간은 모형이다. 실제 링크는 양방향이고 여러 링을 동시에 돌릴 수 있으며, 기계 사이의 대역폭이 고르지 않다. 모형이 말하는 것은 바이트와 단계의 비례이지 밀리초가 아니다. 그리고 시뮬레이션은 조각을 복사하므로 실제 통신의 지연을 재지 않는다. 지연 항은 20 마이크로초를 가정한 값이다.

기계 사이의 배열 합치기가 여기까지다. 다음 절은 그 배열을 디스크에 두는 일이다. 학습이 매 시간 14 GB 를 저장하면 저장소가 곧 찬다. 같은 블록을 두 번 안 저장하는 것이 다음 구조다.

되짚기

all-reduce 를 정의하고 별과 링의 바이트를 명제로 세웠다. 링은 기계 하나가 $2(p-1)/p$ 벡터를 보내며 그것이 하한이고, 단계 $2(p-1)$ 의 지연이 조각이 200 KB 아래일 때 시간을 정한다. 다음 절은 머클 트리와 내용 주소 저장이다.

16.2 머클 트리와 내용 주소 저장. 체크포인트의 차분

학습은 한 시간마다 매개변수를 저장한다. 14 GB 씩 하루 스물네 번이면 336 GB 이고, 그 대부분이 앞 판과 같다. 한 시간 동안 바뀐 매개변수는 전부이지만 크게 바뀐 것은 일부이고, 옵티마이저 상태와 어휘 임베딩의 큰 부분은 바이트 단위로 같다. 7.4 의 중복 제거는 문서의 근사 중복을 찾았다. 여기서는 근사가 아니라 정확한 중복이고, 단위는 문서가 아니라 블록이다. 같은 블록을 두 번 저장하지 않는 것이 내용 주소 저장이고, 그 위에서 두 판의 다른 블록을 로그 번에 찾는 것이 머클 트리다.

6.1 의 해시가 여기서는 주소가 된다. 블록의 해시가 그 블록을 가리키는 키이고, 같은 내용은 같은 키다. 트리는 그 키들을 다시 해시해 올린 것이다.

학습 목표

내용 주소 저장과 머클 트리를 정의하고, 두 판의 다른 블록 k 개를 $O(k \log n)$ 의 해시 비교로 찾음을 명제로 적을 수 있다. 블록 크기가 다시 저장하는 양과 해시 수를 어떻게 맞바꾸는지, 그리고 다른 블록이 많을 때 트리가 전부 비교보다 느린 이유를 실측으로 말할 수 있다.

이 말부터

내용 주소	데이터의 해시를 그 데이터의 키로 쓰는 것. 같은 내용이면 같은 주소다
머클 트리	잎이 블록 해시이고 안쪽 노드가 두 자식 해시의 해시인 트리. 뿌리 하나가 전체를 대신한다
차분	두 판에서 다른 블록의 집합. 새로 저장할 것이 그것뿐이다

디딤돌

디딤돌은 6.1 의 해시 테이블과 8.1 의 트리 높이다. 해시 테이블은 키로 값을 찾았고, 여기서는 값의 해시가 키다. 트리는 높이가 로그였고, 여기서는 그 높이만큼만 내려가 다른 자리를 찾는다.

정의 16.5: 내용 주소 저장과 머클 트리

데이터를 블록 $b_0, \dots, b_{n-1}$ 로 나누고 $h_i = H(b_i)$ 를 둔다. 내용 주소 저장소는 $h_i \mapsto b_i$ 의 해시 테이블이고, 한 판은 해시 열 $(h_0, \dots, h_{n-1})$ 이다. 머클 트리는 잎이 h_i 이고 안쪽 노드가 $H(\text{왼쪽} \,\|\, \text{오른쪽})$ 이며, 층이 $\lceil \log_2 n \rceil + 1$ 개다. 뿌리를 r 이라 한다.

불변식 두 판의 어느 부분 트리가 같은 뿌리를 가지면, H 의 충돌이 없는 한 그 아래 블록이 전부 같다. 같은 블록은 저장소에 한 번만 있다.

불변식의 첫 문장이 로그의 근거다. 뿌리가 같으면 내려갈 필요가 없고, 다르면 두 자식 중 다른 쪽만 내려간다. 다른 블록이 k 개면 내려가는 경로가 k 개다.

명제 16.6: 차분의 비용

> 블록 n 개의 두 판에서 다른 블록이 k 개면, 머클 트리는 해시 비교 $2k\lceil \log_2 n \rceil$ 이하로 그 k 개를 찾는다. 해시 열을 전부 견주면 비교가 n 이다. 따라서 트리가 이기는 조건은 $k < n/(2\log_2 n)$ 이고, 새 판을 저장하는 값은 두 방법 모두 새 블록 k 개의 바이트다.

증명. 다른 잎에서 뿌리까지의 경로 위 노드만 다르고, 경로가 k 개에 길이가 $\log_2 n$ 이다. 탐색은 다른 노드의 자식 둘을 보므로 노드마다 비교 둘이다. 같은 부분 트리는 뿌리 비교 하나로 끝난다. 저장은 해시 테이블에 없는 블록만 넣으므로 어느 쪽이든 k 개다. ■

명제의 둘째 문장이 이 절의 실험이다. $n = 4{,}096$ 이면 $n/(2\log_2 n)$ 이 170 이고, 다른 블록이 그보다 많으면 트리가 느리다. 바꾼 바이트가 200 개인데 블록이 작으면 그 조건을 넘는다.

예제 16.2: 16 MB 에서 200 바이트

> 16 MB 를 4 KB 블록으로 나누면 4,096 개다. 무작위 자리 200 곳을 바꾸면 다른 블록이 197 개 안팎이고, 새로 저장하는 것이 0.8 MB 다. 1 MB 블록이면 16 개 전부가 달라 16 MB 를 다시 저장한다. 트리의 층은 13 개와 5 개다. 비교는 $2 \times 197 \times 12 = 4{,}700$ 과 $2 \times 16 \times 4 = 128$ 이다.

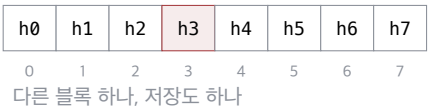
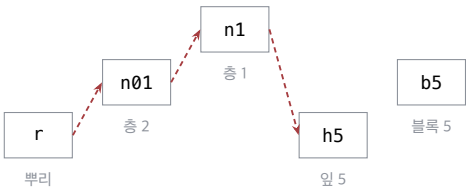


그림 16.3.. 두 판의 해시 열. 블록 3 만 다르고 나머지 일곱은 같은 주소다. 새로 저장하는 것은 블록 3 하나다



뿌리 → 오른쪽 → 왼쪽 → 잎 5. 다른 쪽 부분 트리는 뿌리 비교 하나로 닫힌다

그림 16.4.. 블록 여덟 개의 머클 트리. 블록 5 가 바뀌면 그 경로 위의 노드 셋만 다르고, 뿌리에서 내려가며 다른 쪽만 열면 비교 여섯 번이다

균열

16 MB 의 판에서 200 바이트를 바꾼 새 판을 만들고, 블록 크기 셋에서 새로 저장하는 양과 다른 블록을 찾는 시간을 쟀다.

실험 16.2 블록 크기와 차분. 새로 든 바이트, 전부 비교, 트리 찾기

블록 KB	블록 수	새로 든 MB	다른 블록	해시 ms	비교 ms	트리 ms
4	4096	0.77	197	136	0.57	3.28
64	256	8.94	143	163	0.0393	0.60
1024	16	16.0	16	142	0.0043	0.0429

측정 조건. 16 MB 무작위 바이트, 새 판은 200 바이트를 바꿈. SHA-256. 새로 든 MB 는 내용 주소 저장소에 두 판을 넣었을 때 둘째 판이 더한 양. 세 번 중앙값.

새로 저장하는 양이 블록 4 KB 에서 0.77 MB, 1 MB 에서 16 MB 다. 작은 블록이 스무 배를 아낀다. 대신 해시가 4,096 개이고 해시 시간은 세 크기 모두 140 밀리초 안팎이다. 바이트를 전부 읽는 값이라 블록 수와 무관하다. 트리 찾기는 전부 비교보다 느리다. 4 KB 에서 3.3 밀리초와 0.6 밀리초다. 다른 블록 197 개가 명제 16.6 의 경계 170 을 넘어서다.

관찰

블록이 작을수록 다시 저장하는 양이 줄고, 해시 시간은 바이트가 정하므로 그대로다. 트리의 로그는 다른 블록이 드물 때의 값이고, 흩어진 변경 200 개는 드문 것이 아니다.

이 관찰이 트리의 자리를 정한다. 뿌리 하나로 두 판이 같은지 확인하는 것과, 몇 블록만 다른 판을 찾는 것이 트리의 일이다. 판 전체를 견주는 일은 해시 열 하나를 훑는 것으로 충분하다.

다시 쓰기

이 절의 다시 쓰기는 저장을 해시 테이블로 보는 것이다. 판은 파일이 아니라 해시의 열이고, 저장소는 해시에서 블록으로 가는 표다. 새 판을 저장하는 것은 표에 없는 키를 넣는 것이다.

명제 16.7: 블록 크기의 맞바꿈

판 N 바이트, 블록 B, 흩어진 변경 c 곳에서 새로 저장하는 양은 $\min(N, cB)$ 안팎이고, 판 하나의 해시 열은 $32N/B$ 바이트다. B 를 줄이면 앞이 줄고 뒤가 늘며, 둘의 합이 최소인 B 는 $\sqrt{32N/c}$ 근처다. $N = 14$ GB, $c = 10^4$ 이면 6.7 MB 이고, 해시 열은 67 KB 다.

증명. 변경 하나가 블록 하나를 다르게 하므로 새 블록이 c 개, cB 바이트다. 해시 열은 블록 수 N/B 에 32 바이트다. $cB + 32N/B$ 를 B 로 미분해 0 인 자리다. ■

Merkle 이 1987 년에 이 트리를 서명의 도구로 세웠다 [확립] [65]. 잎 하나의 진위를 뿌리와 경로 위의 해시 $\log n$ 개로 확인하는 구조였고, 그 확인이 여기서는 차분이다. 버전 관리 시스템과 분산 파일 시스템이 같은 구조 위에 있다.

알고리즘 16.2 다른 블록 찾기 `dsai.merkle.changed_blocks`

입력: 두 판의 머클 트리 층들 T_a, T_b
스택 $\leftarrow$ (뿌리 층, 0)
조건 스택이 비지 않음 **동안** (층 ℓ, 자리 i) 를 꺼낸다
  만약 $T_a[\ell][i] = T_b[\ell][i]$ **이면** 건너뛴다
  아니면 만약 $\ell = 0$ **이면** i 를 답에 넣는다
  아니면자식 $(\ell - 1, 2i)$, $(\ell - 1, 2i + 1)$ 을 넣는다
  끝 만약
끝 조건

구현

블록 해시, 트리 올리기, 다른 블록 찾기, 저장소다. 저장소는 `dict` 하나다.

지금 보는 코드 층마다 이웃 둘을 이어 해시한다. 홀수면 마지막을 복제한다.

구현 머클 트리 올리기 `samples/dsai/merkle.py:21`

```
def merkle_root(hashes: list[bytes]) -> tuple[bytes, list[list[bytes]]]:
    """알고리즘 16.2. 이웃 둘의 해시를 이어 다시 해시한다. 층이 로그 개다. (뿌리,
    층들)."""
    levels = [list(hashes)]
    while len(levels[-1]) > 1:
        cur = levels[-1]
        if len(cur) % 2:
            cur = cur + [cur[-1]]
        levels.append([hashlib.sha256(cur[i] + cur[i + 1]).digest()
                       for i in range(0, len(cur), 2)])
    return levels[-1][0], levels
```

읽어 보면 `cur[i] + cur[i + 1]` 이 두 자식 해시의 이어 붙이기다. `levels[-1]` 이 하나가 되면 뿌리다. 층이 $\log_2 n$ 개다.

구현 다른 블록 찾기 `samples/dsai/merkle.py:35`

```
def changed_blocks(levels_a: list[list[bytes]],
                   levels_b: list[list[bytes]]) -> list[int]:
    """명제 16.3. 뿌리에서 내려가며 다른 부분 트리만 연다."""
    depth = len(levels_a) - 1
    stack = [(depth, 0)]
    out = []
    while stack:
        lvl, i = stack.pop()
        a = levels_a[lvl][i] if i < len(levels_a[lvl]) else None
        b = levels_b[lvl][i] if i < len(levels_b[lvl]) else None
        if a == b:
            continue
        if lvl == 0:
            out.append(i)
        else:
            stack.append((lvl - 1, 2 * i + 1))
            stack.append((lvl - 1, 2 * i))
    return sorted(out)
```

읽어 보면 `a == b` 면 `continue` 가 부분 트리 전체를 닫는다. 5.1 의 명시적 스택이고, 다른 경로만 내려간다.

분석

이 구조는 해시 테이블과 트리를 저장 위에 얹은 것이다. 새로운 것은 없다. 해시가 주소라는 것과 해시의 해시가 요약이라는 것뿐이고, 둘 다 6장과 8장이다. 그 위에서 체크포인트 하나가 파일이 아니라 해시 열이 되고, 저장소가 자라는 속도가 판의 수가 아니라 변경의 양이 된다.

이 구조가 서는 자리는 저장 그 자체다. 학습 체크포인트, 데이터셋의 판, 모델의 배포본이 전부 내용 주소로 저장되고, 같은 블록은 어디서 왔든 한 번이다. 7장의 중복 제거가 데이터 안의 중복을 걸렀다면 이것은 판 사이의 중복을 거른다.

현장에서

> Merkle 의 1987년 논문은 트리 위의 서명을 제안했고, 잎 하나를 확인하는 데 경로 위의 해시 $\log n$ 개면 됨을 보였다 [확립] [65]. 이 절의 차분 찾기는 그 경로를 두 판에서 나란히 내려간 것이다.

주의

> 실험 16.2 의 변경은 무작위 200 곳이다. 실제 체크포인트는 모든 매개변수가 조금씩 바뀌어 블록 단위로는 전부 다르고, 그때 내용 주소 저장이 아끼는 것은 옵티마이저 상태와 임베딩처럼 안 바뀐 부분뿐이다. 그리고 한 바이트를 끼워 넣으면 그 뒤의 모든 블록 경계가 밀려 전부 새 블록이다. 경계를 내용으로 정하는 것이 문제 16.6 이다.

판을 저장하는 법이 여기까지다. 다음 절은 그 저장을 읽는 법이다. 열 하나만 필요한데 행 전체를 읽는 것이 저장의 가장 흔한 낭비이고, 배열의 놓임이 그것을 정한다.

되짚기

내용 주소 저장과 머클 트리를 정의하고, 다른 블록 k 개를 $2k \log n$ 비교로 찾음을 보였다. 블록 4 KB 가 1 MB 보다 스무 배 적게 저장했고, 다른 블록이 경계 170 을 넘어 트리가 전부 비교보다 느렸다. 다음 절은 열 지향 저장과 메모리 맵이다.

16.3 열 지향 저장과 메모리 맵

데이터셋은 표다. 행이 표본이고 열이 속성이다. 학습이 읽는 것은 대개 열 몇 개이고, 통계를 내는 것은 열 하나다. 그런데 파일이 행 순서면 열 하나를 읽는 데 파일 전체를 읽는다. 1.2 의 스트라이드로 말하면 행 저장에서 열 하나는 보폭이 레코드 크기인 뷰이고, 1.3 의 라인으로 말하면 라인마다 8 바이트만 쓰고 56 바이트를 버린다. 열마다 배열을 따로 두면 보폭이 1 이고 라인이 전부 쓰인다.

메모리 맵은 그 배열을 파일에서 읽지 않고 파일을 배열로 본다. 페이지가 처음 닿을 때 운영체제가 읽어 오고, 안 닿은 페이지는 안 읽는다. 열 저장과 메모리 맵이 만나면 읽는 것이 닿은 열의 페이지뿐이다.

학습 목표

> 행 저장과 열 저장을 스트라이드로 정의하고, 열 하나를 읽을 때 닿는 바이트가 레코드 크기 대 열 크기의 비만큼 다름을 명제로 적을 수 있다. 메모리 맵이 페이지 단위의 지연 읽기임을 설명하고, 열 저장이 실측에서 몇 배 빠른지와 그 배수가 바이트 비보다 작은 이유를 말할 수 있다.

이 말부터

> 행 저장 레코드 하나의 모든 열이 잇달아 있는 놓임. 구조체의 배열
> 열 저장 열 하나의 모든 행이 잇달아 있는 놓임. 배열의 구조체
> 메모리 맵 파일을 주소 공간에 붙여 배열처럼 읽는 것. 닿은 페이지만 읽힌다

디딤돌

디딤돌은 1.2 의 스트라이드 뷰다. 같은 바이트를 두 보폭으로 볼 수 있었고, 어느 보폭이 순차인지가 시간을 정했다. 행 저장과 열 저장은 그 두 보폭을 파일에 굳힌 것이다.

정의 16.8: 행 저장과 열 저장

> 행 n 개, 열 m 개, 열 j 의 원소 크기 s_j, 레코드 크기 $R = \sum_j s_j$ 에서 행 저장은 원소 (i, j) 를 주소 $iR + \sum_{j'<j} s_{j'}$ 에 두고, 열 저장은 열마다 배열을 두어 주소 $\mathrm{base}_j + i s_j$ 에 둔다.
>
> **불변식** 행 저장에서 열 j 는 보폭 R 의 뷰이고, 열 저장에서는 보폭 s_j 의 연속 배열이다. 행 하나는 그 반대다.

불변식이 말하는 것은 어느 쪽도 공짜가 아니라는 것이다. 열을 읽는 데 좋은 놓임은 행을 읽는 데 나쁘다. 학습이 무엇을 읽는지가 놓임을 정하고, 그것이 명제의 조건이다.

명제 16.9: 열 하나를 읽는 이동량

열 j 전부를 읽을 때 행 저장은 라인 단위로 nR 바이트 안팎을 닿고 (레코드가 라인보다 크면 n 개의 라인), 열 저장은 ns_j 바이트를 닿는다. 따라서 비가 R/s_j 이고, 열 여덟 개에 8 바이트씩이면 8 배다. 행 하나를 읽을 때는 반대로 행 저장이 라인 하나, 열 저장이 라인 m 개다.

증명. 행 저장에서 열 j 의 원소 n 개는 보폭 R 로 떨어져 있어 라인마다 원소 하나이고, 라인 n 개 또는 nR 바이트다. 열 저장은 연속이라 ns_j 바이트를 순차로 읽는다. 행 하나는 행 저장에서 연속 R 바이트이고, 열 저장에서는 m 개의 떨어진 주소다. ■

명제의 8 배가 이 절의 실험이다. 바이트로는 8 배인데 시간은 그만큼이 아닐 것이다. 행 저장도 순차로 읽기는 하므로 프리페치가 들고, 열 저장은 파이썬 호출의 고정 값이 있다.

예제 16.3: 행 2 백만 개, 열 여덟 개

int64 넷과 float64 넷이면 레코드가 64 바이트이고 표가 128 MB 다. 열 하나의 합을 내는 데 행 저장은 128 MB 를 닿고 열 저장은 16 MB 를 닿는다. 행 10^8 개면 6.4 GB 와 0.8 GB 이고, 10 GB/s 이면 640 밀리초와 80 밀리초다.

그림 16.5.. 행 저장의 열 하나. 레코드마다 여덟 칸 중 한 칸만 쓰고 나머지 일곱 칸은 라인에 딸려 온다. 열 저장은 진한 칸만 잇달아 둔다

균열

행 2 백만 개의 표를 행 파일 하나와 열 파일 여덟 개로 쓰고, 메모리 맵으로 열 하나의 합을 냈다. 파일은 방금 써서 페이지 캐시에 있다.

실험 16.3 열 하나의 합. 행 저장과 열 저장의 메모리 맵

열	행 저장 ms	열 저장 ms	행 닿는 MB	열 닿는 MB
i0	129	37.9	122	15.3
f3	131	22.6	122	15.3

측정 조건. 행 2,000,000 개, 열 8 개 (int64 4, float64 4), 레코드 64 바이트. 메모리 맵 읽기. 닿는 MB 두 열은 명제 16.9 의 값. 다섯 번 중앙값.

정수 열에서 129 밀리초와 38 밀리초, 실수 열에서 131 밀리초와 23 밀리초다. 바이트는 8 배, 시간은 3.4 배와 5.8 배다. 행 저장의 130 밀리초는 128 MB 를 1 GB/s 로 읽은 셈이고

순차 읽기치고 느리다. 보폭 64 의 뷰를 float64 로 바꾸는 복사가 원소마다 라인 하나를 당기기 때문이다. 열 저장의 23 밀리초는 16 MB 의 순차 복사이고, 정수 열이 실수 열보다 느린 것은 정수에서 실수로 바꾸는 값이다.

관찰

> 열 저장은 닿는 바이트를 R/s_j 분의 1 로 줄이고, 시간은 그보다 덜 준다. 행 저장 쪽도 순차라 프리페치가 들고, 열 저장 쪽에는 형 변환의 값이 남는다.

이 관찰이 1.3 의 표와 같은 모양이다. 보폭이 라인보다 커지면 라인마다 원소 하나이고, 그것이 바이트의 8 배다. 시간이 8 배가 안 된 것은 두 놓임 다 순차이기 때문이고, 무작위였다면 더 벌어졌다.

다시 쓰기

이 절의 다시 쓰기는 파일을 배열로 보는 것이다. 읽기 호출 대신 주소가 있고, 주소에 닿는 순간 페이지가 온다. 그러면 0.1 의 이동량이 파일에도 그대로 적용된다.

명제 16.10: 메모리 맵의 이동량

> 페이지 P 바이트의 메모리 맵에서 어떤 접근 패턴이 닿는 서로 다른 페이지가 q 개이면 디스크에서 오는 바이트가 qP 다. 열 저장의 열 하나는 $\lceil ns_j/P \rceil$ 페이지이고, 행 저장의 열 하나는 $\lceil nR/P \rceil$ 페이지다. 따라서 페이지 캐시에 없을 때 두 놓임의 차이는 라인이 아니라 페이지의 비이고, 같은 R/s_j 다.

증명. 메모리 맵은 페이지 단위로 결함을 내고 페이지 전체를 읽는다. 닿은 페이지 수가 읽는 양이다. 열 저장은 연속이라 페이지가 꽉 차고, 행 저장은 페이지마다 P/R 개의 원소만 쓰인다. ■

Stonebraker 등의 2005 년 논문은 읽기가 많은 부하에서 열 저장이 행 저장보다 한 자릿수 빠름을 보이고, 열마다 압축을 얹어 그 비를 더 벌렸다 [확립] [66]. 페이지 단위의 결함은 2.2 에서 인용한 Denning 의 가상 메모리 그대로다.

알고리즘 16.3 열 저장 쓰기와 읽기 `dsai.columnar.write_column_major`

> **입력 :** 표 { 열 이름 ↦ 배열}, 디렉터리
> 1: **모든** 열 j **동안** 배열을 파일 j 로 쓴다
> 2: **끝 반복**
> 3: 읽기 : 파일 j 를 메모리 맵으로 열어 연속 배열로 본다

구현

행 파일 쓰기, 열 파일 쓰기, 두 방식의 열 합이다. 읽기는 메모리 맵이고 둘 다 배열 하나다.

지금 보는 코드 열마다 파일 하나. 같은 열의 행이 잇달아 있다.

구현 열 저장 쓰기 `samples/dsai/columnar.py:25`

```
def write_column_major(dirpath: Path, table: dict[str, np.ndarray]) -> None:
    """정의 16.4. 열마다 파일 하나. 같은 열이 연속이다."""
    dirpath.mkdir(exist_ok=True)
    for n, col in table.items():
        col.tofile(dirpath / f"{n}.bin")
```

읽어 보면 `tofile` 이 배열의 바이트를 그대로 쓴다. 헤더가 없고 `dtype` 을 읽는 쪽이 안다. 파일 하나가 배열 하나다.

구현 두 방식의 열 합 `samples/dsai/columnar.py:32`

```
def column_sum_row_major(path: Path, dtype: np.dtype, name: str) -> float:
    """행 파일에서 열 하나를 읽는다. 메모리 맵이어도 레코드마다 라인 하나다."""
    rows = np.memmap(path, dtype=dtype, mode="r")
    return float(rows[name].astype(np.float64).sum())


def column_sum_column_major(dirpath: Path, name: str, dtype: np.dtype) -> float:
    """열 파일 하나만 읽는다. 순차 접근이고 그 열의 바이트만 이동이다."""
    col = np.memmap(dirpath / f"{name}.bin", dtype=dtype, mode="r")
    return float(col.astype(np.float64).sum())
```

읽어 보면 `np.memmap` 이 파일을 배열로 본다. 행 쪽의 `rows[name]` 이 보폭 64 의 뷰이고, 열 쪽의 `col` 이 연속 배열이다. 그 차이가 실험의 세 배에서 여섯 배다.

분석

이 구조는 5장의 스트라이드를 파일에 굳힌 것이다. 새로운 것은 없다. 어느 보폭으로 읽을지를 쓸 때 정한다는 것뿐이고, 그 결정이 읽기의 이동량을 R/s_j 배 가른다. 메모리 맵은 그 위에서 파일과 배열의 경계를 지운다. 페이지가 라인의 큰 판이고, 2.2 의 조각과 8.2 의 블록이 같은 자리에서 다시 나온다.

이 구조가 서는 자리는 데이터셋 형식이다. 표 형식의 데이터셋이 열 단위로 저장되고 메모리 맵으로 읽히며, 학습이 쓰는 열만 닿는다. 토큰 열 하나가 수백 GB 여도 닿은 페이지만 메모리에 든다.

현장에서

C-Store 의 보고에서 열 저장은 읽기 부하의 표준 질의에서 행 저장보다 한 자릿수 빨랐고, 열마다 다른 압축을 골라 저장이 몇 분의 1 이었다 [확립] [66]. 이 절은 압축을 다루지 않았고, 문제 16.7 이 정렬된 열의 런 길이 부호화다.

주의

실험 16.3 의 파일은 페이지 캐시에 있다. 디스크에서 오면 명제 16.10 의 페이지 비가 그대로 시간이고, 행 저장은 8 배 느리다. 캐시에 있어서 3 배에서 6 배였다. 그리고 행 하나를 읽는 일에서는 반대다. 표본 하나의 모든 열이 필요한 학습이면 열 저장이 라인 m 개를 당긴다. 다음 절의 셔플이 그 일을 한다.

파일을 배열로 보는 법이 여기까지다. 다음 절은 그 배열을 무작위 순서로 읽는 일이다. 학습은 매 에폭 순서를 섞고, 섞은 순서로 읽으면 이 절의 순차가 무너진다. 그 사이의 타협이 블록이다.

되짚기

> 행 저장과 열 저장을 보폭으로 정의하고, 열 하나를 읽는 이동량이 R/s_j 배 다름을 보였다. 실측은 바이트 8 배에 시간 3 배에서 6 배였고, 메모리 맵은 그 이동량을 페이지 단위로 파일에 적용한다. 다음 절은 셔플과 순열이다.

16.4 셔플과 순열. 데이터 로더의 자료 구조

학습은 매 에폭 표본을 무작위 순서로 본다. 같은 순서로 두 번 보면 배치의 구성이 같아 기울기의 잡음이 같은 방향으로 쌓인다. 순서 하나가 순열 하나이고, 순열을 만드는 것이 셔플이다. 10.3 의 표본 추출은 확률 분포에서 하나를 뽑았다. 셔플은 n 개를 한 번씩 뽑는 것이고, 균등한 순열이 목표다. 그리고 그 순열대로 읽으면 16.3 의 순차가 사라진다. 무작위 순서는 무작위 접근이다.

이 절은 균등한 순열을 만드는 오래된 알고리즘 하나와, 순차 읽기를 얼마간 지키는 블록 섞기, 그리고 순열을 저장하지 않고 계산하는 망을 세운다. 셋은 무작위성과 순차성과 저장을 서로 맞바꾼다.

학습 목표

Fisher-Yates 가 균등한 순열을 만듦을 증명하고, 블록 섞기가 순차 읽기의 비율을 어떻게 지키는지 적을 수 있다. 저장하지 않는 순열이 왜 전단사인지를 Feistel 망과 사이클 걷기로 설명하고, 세 방법의 만들기 시간과 읽기 시간을 실측으로 견줄 수 있다.

이 말부터

셔플	$0, \dots, n-1$ 의 순열 하나를 무작위로 고르는 것. 균등하면 $n!$ 개가 같은 확률
블록 섞기	블록 순서를 섞고 블록 안을 섞는 것. 블록 사이만 무작위 접근이다
형식 보존 순열	순열을 저장하지 않고 $i \mapsto \pi(i)$ 를 계산하는 전단사. 키가 순열을 정한다

디딤돌

디딤돌은 10.3 의 뽑기다. 확률에 따라 하나를 뽑았고, 별칭표가 그것을 $O(1)$ 에 했다. 순열은 뽑은 것을 빼고 다시 뽑는 것이고, 배열에서 빼는 것이 $O(n)$ 이면 전체가 $O(n^2)$ 다. 그것을 $O(n)$ 으로 만든 것이 1964 년의 알고리즘이다.

정의 16.11: 균등 순열과 Fisher-Yates

$\{0, \dots, n-1\}$ 의 균등 순열은 $n!$ 개의 순열이 각각 $1/n!$ 의 확률을 갖는 확률 변수다. Fisher-Yates 는 배열 $p = (0, \dots, n-1)$ 에서 $i = n-1, \dots, 1$ 에 대해 j 를 $\{0, \dots, i\}$ 에서 균등하게 뽑아 $p[i]$ 와 $p[j]$ 를 바꾼다.

불변식 단계 i 가 끝나면 $p[i]$ 는 남은 원소 중 균등하게 고른 것이고, 뒤의 단계는 $p[i]$ 를 건드리지 않는다.

불변식이 증명의 전부다. 마지막 자리부터 하나씩 균등하게 채우고 채운 자리는 다시 안 본다. 그것이 순열 전체를 균등하게 만든다.

정리 16.12: Fisher-Yates 의 균등성

> Fisher-Yates 가 만든 순열은 균등하고, 시간은 $O(n)$, 추가 메모리는 $O(1)$ 이다.

증명. 어떤 순열 σ 가 나올 확률을 센다. 단계 $i = n-1$ 에서 $p[n-1] = \sigma(n-1)$ 이 되려면 j 가 특정 하나여야 하고 확률이 $1/n$ 이다. 단계 i 에서 $\{0, \dots, i\}$ 에 남은 원소는 σ 의 뒤쪽을 뺀 것이고 그중 $\sigma(i)$ 는 하나이므로 확률이 $1/(i+1)$ 이다. 곱하면 $1/n!$ 이고 σ 와 무관하다. 단계마다 뽑기 하나와 교환 하나다. Durstenfeld 가 1964 년에 이 형태를 적었다 [확립] [63]. ■

정리가 만든 순열은 완전 무작위이고, 그 순열대로 읽으면 표본마다 다른 페이지다. 앞 절의 명제로 표본 하나가 페이지 하나이고, 백만 개면 페이지 백만 개다.

명제 16.13: 블록 섞기의 순차 비율

> 블록 크기 b 의 블록 섞기는 블록 n/b 개의 순서를 균등하게 섞고 블록 안을 균등하게 섞는다. 연속한 두 읽기가 같은 블록에 있을 비율이 $1 - b/n$ 안팎이고, 페이지 단위로 보면 블록 하나가 페이지 bs/P 개다. 만들 수 있는 순열은 $(n/b)!\,(b!)^{n/b}$ 개로 $n!$ 보다 적다. 블록 밖의 원소와 이웃이 되는 일이 없다.

증명. 블록 안의 읽기 $b-1$ 번은 같은 블록이고 블록 경계 한 번만 다르다. 순열의 수는 블록 순서와 블록 안 순서의 곱이다. 어느 원소의 이웃도 같은 블록 안의 원소이거나 다음 블록의 첫 원소다. ■

명제의 둘째 문장이 블록 섞기의 대가다. 무작위성을 블록 안으로 가두었고, 그 대가로 순차 읽기를 얻었다. 셋째 방법은 순열 자체를 저장하지 않는다.

예제 16.4: 표본 십억 개

> 순열을 int64 로 저장하면 8 GB 다. 에폭마다 새 순열이면 에폭마다 8 GB 를 만들고 저장한다. 블록 2^{16} 이면 블록 15,259 개의 순서와 블록 안의 순열이고, 저장은 같은 8 GB 다. 형식 보존 순열은 키 네 개를 저장하고 $\pi(i)$ 를 계산한다. 자리 i 의 표본이 무엇인지를 $O(1)$ 에 알고, 여러 기계가 같은 키로 같은 순열을 나눠 갖는다.

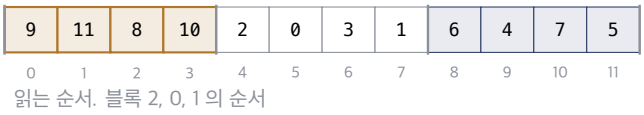


그림 16.6. 블록 4 의 블록 섞기. 블록 순서가 섞이고 블록 안이 섞인다. 진한 칸이 한 블록이고, 이웃은 언제나 같은 블록 안이다

균열

레코드 백만 개, 64 바이트씩의 파일을 메모리 맵으로 두고, 순열 다섯 가지를 만드는 시간과 그 순서로 읽는 시간을 쟀다.

실험 16.4 순열 다섯. 만들기, 저장, 순차 비율, 읽기

순열	만들기 ms	저장 MB	블록 연속 퍼센트	읽기 ms
순차 (섞지 않음)	2.63	0.00	100.0	18.6
Fisher-Yates (파이썬)	10232	8.00	0.41	50.3
전체 순열 (NumPy)	66.8	8.00	0.41	41.7
블록 섞기 (4096)	39.2	8.00	100.0	30.8
Feistel	277	0.00	0.14	40.5

측정 조건. 레코드 1,000,000 개, 64 바이트. 블록은 레코드 4,096 개 (256 KB). 메모리 맵에서 레코드 첫 바이트를 순열 순서로 읽어 더함. 파일은 페이지 캐시에 있음. Fisher-Yates 는 한 번, 나머지는 세 번 중앙값.

파이썬 루프의 Fisher-Yates 가 10 초, NumPy 의 순열이 67 밀리초다. 같은 알고리즘이고 백오십 배는 파이썬 호출 백만 번의 값이다. 블록 섞기는 39 밀리초이고, Feistel 은 277 밀리초에 저장이 0 이다. 읽기는 순차 19 밀리초, 완전 무작위 42 밀리초, 블록 섞기 31 밀리초다. 페이지 캐시 안이라 두 배 차이이고, 디스크에서였다면 무작위 쪽이 백 배 느리다.

관찰

세 방법은 같은 것을 다르게 맞바꾼다. 전체 순열은 무작위성 전부에 저장 $8n$, 블록 섞기는 무작위성 일부에 순차 읽기, 형식 보존 순열은 저장 없음에 만들기 시간 네 배다.

이 관찰이 로더의 설계다. 디스크에서 읽으면 블록 섞기이고, 기계 여럿이 같은 순서를 나눠야 하면 형식 보존 순열이며, 메모리에 다 들면 전체 순열이다. 어느 것도 나머지를 대신하지 않는다.

다시 쓰기

이 절의 다시 쓰기는 순열을 배열이 아니라 함수로 두는 것이다. 6.1 의 해시 함수가 키를 자리로 보냈듯, 여기서는 자리를 자리로 보내되 전단사여야 한다. 그것을 보장하는 구조가 Feistel 망이다.

명제 16.14: Feistel 망과 사이클 걷기

$2h$ 비트의 수를 반쪽 (L, R) 로 나누고 라운드마다 $(L, R) \leftarrow (R, L \oplus f(R))$ 을 하면, f 가 무엇이든 이 사상은 $\{0, \dots, 2^{2h} - 1\}$ 위의 전단사다. $2^{2h} \geq n$ 인 h 를 잡고 결과가 n 을 넘으면 넘지 않을 때까지 다시 적용하면 $\{0, \dots, n - 1\}$ 위의 전단사이고, 기대 반복이 $2^{2h}/n < 4$ 다.

증명. 한 라운드의 역은 $(L, R) \leftarrow (R \oplus f(L), L)$ 이라 라운드마다 역이 있고, 합성도 역이 있다. 전단사 π 의 사이클을 따라가면 n 밖의 원소를 건너뛴 사상도 전단사다. n 안에 드는 원소의 비가 $n/2^{2h}$ 이고 $2^{2h} < 4n$ 이라 기하 분포의 기대가 4 미만이다. ■

이 명제가 형식 보존 순열이다. 균등 순열은 아니다. 키가 정하는 순열의 집합이 $n!$ 보다 훨씬 작다. 대신 저장이 없고, 어느 자리든 $O(1)$ 이며, 키 하나로 기계 천 대가 같은 순서를 본다.

알고리즘 16.4 Fisher-Yates `dsai.loader.fisher_yates`

입력: n
1: $p \leftarrow (0, \dots, n-1)$
2: **반복** $i = n-1$ **downto** 1 **동안** $j \leftarrow \{0, \dots, i\}$ 에서 균등; $p[i] \leftrightarrow p[j]$
3: **끝 반복**
4: **반환** p

구현

Fisher-Yates, 블록 섞기, Feistel 망과 사이클 걷기다. 앞 둘은 정의 그대로이고 셋째는 명제 그대로다.

지금 보는 코드 뒤에서부터 무작위 자리와 바꾼다. 바꾼 자리는 다시 안 본다.

구현 Fisher-Yates 와 블록 섞기 `samples/dsai/loader.py:14`

```python
def fisher_yates(n: int, rng: np.random.Generator) -> np.ndarray:
    """알고리즘 16.3. 뒤에서부터 무작위 자리와 바꾼다. 균등한 순열, O(n)."""
    p = np.arange(n)
    for i in range(n - 1, 0, -1):
        j = int(rng.integers(0, i + 1))
        p[i], p[j] = p[j], p[i]
    return p


def block_shuffle(n: int, block: int, rng: np.random.Generator) -> np.ndarray:
    """정의 16.5. 블록 순서를 섞고 블록 안도 섞는다. 블록 사이만 무작위 접근이다."""
    nb = (n + block - 1) // block
    order = rng.permutation(nb)
    out = np.empty(n, dtype=np.int64)
    k = 0
    for b in order:
        lo, hi = b * block, min((b + 1) * block, n)
        inner = rng.permutation(hi - lo) + lo
        out[k:k + hi - lo] = inner
        k += hi - lo
    return out
```

읽어 보면 `range(n - 1, 0, -1)` 이 정리의 i 다. 블록 섞기의 `rng.permutation(nb)` 가 블록 순서이고 안쪽 `permutation` 이 블록 안이다. `out` 이 순열 배열이라 저장이 $8n$ 이다.

구현 Feistel 망과 사이클 걷기 `samples/dsai/loader.py:37`

```
def feistel_round(y: np.ndarray, h: int, keys: np.ndarray, rounds: int) -> np.ndarray:
    """2h 비트 위의 Feistel 망. 반쪽 h 비트씩. 어떤 f 를 써도 전단사다."""
    mask = (1 << h) - 1
    left, right = y >> h, y & mask
    for r in range(rounds):
        f = ((right * 0x9E3779B1 + int(keys[r])) * 0x85EBCA6B) & 0xFFFFFFFF
        left, right = right, left ^ (f & mask)
    return (left << h) | right


def feistel_permutation(i: np.ndarray, n: int, keys: np.ndarray,
                        rounds: int = 4) -> np.ndarray:
    """알고리즘 16.4. 저장하지 않는 순열. 2^{2h} >= n 인 h 로 망을 돌리고,
    n 을 넘으면 넘지 않을 때까지 다시 돌린다 (사이클 걷기)."""
    h = (max(1, int(np.ceil(np.log2(n)))) + 1) // 2
    out = i.astype(np.int64).copy()
    pending = np.ones(len(out), dtype=bool)
    for _ in range(64):
        y = feistel_round(out, h, keys, rounds)
        out = np.where(pending, y, out)
        pending &= out >= n
        if not pending.any():
            break
    return out
```

읽어 보면 `left, right = right, left ^ (f & mask)` 가 라운드다. `pending` 이 아직 n 을 넘는 원소이고, 그것에만 망을 다시 적용한다. `out` 에는 순열이 없고 함수만 있다.

분석

이 구조는 뽑기와 해시가 로더 위에서 만난 것이다. Fisher-Yates 는 뽑기의 반복이고, 블록 섞기는 그것을 두 층으로 둔 것이며, Feistel 은 해시 함수를 전단사로 만든 것이다. 셋 다 앞 장의 구조이고, 새것은 무엇을 맞바꾸는지의 명제뿐이다.

이 구조가 서는 자리는 모든 학습의 입구다. 에폭마다 순열 하나가 있고, 그 순열이 디스크 읽기의 순서이며, 기계 여럿이 그 순열을 나눠 갖는다. 로더가 느리면 가속기가 논다. 1.4 의 배치가 그 순열 위에서 잘린다.

현장에서

> Durstenfeld 의 1964 년 알고리즘은 다섯 줄이고, 뒤에서부터 무작위 자리와 바꾸는 형태가 그대로 있다 [확립] [63]. 60 년 뒤에도 모든 라이브러리의 셔플이 이 다섯 줄이다.

주의

> 실험 16.4 의 읽기는 페이지 캐시 안이다. 디스크에서는 무작위 읽기가 순차의 백 배 느리고, 그때 블록 섞기와 전체 순열의 차이가 두 배가 아니라 수십 배다. 그리고 Feistel 의 순열은 균등하지 않다. 라운드 넷과 키 넷이 만드는 순열의 집합은 $n!$ 에 비해 무시할 만큼 작고, 그 순열의 통계가 학습에 충분한지는 이 책이 재지 않았다.

배열을 나누고, 저장하고, 읽고, 섞었다. 기계 하나를 넘어간 자료 구조가 여기까지다. 다음 장은 이 책의 마지막이고, 지금까지의 구조를 하나의 잣대로 다시 세운다. 어떤 구조를 언제 고르는가.

되짚기

> Fisher-Yates 의 균등성을 증명하고, 블록 섞기가 무작위성을 블록 안에 가두어 순차 읽기를 얻으며, Feistel 망이 저장 없는 전단사를 줌을 보였다. 실측에서 셋은 만들기 39 밀리초에서 277 밀리초, 읽기 31 밀리초에서 42 밀리초였고 저장은 8 MB 와 0 이었다. 다음 장은 자료 구조를 고르는 일이다.

16장을 닫으며

이 장이 세운 것은 기계를 넘어간 배열의 잣대다. 이동량은 그대로이고 단위가 바뀐다. 링크의 바이트는 대역폭과 단계 지연이고, 디스크의 바이트는 블록과 페이지이며, 로더의 바이트는 순차와 무작위다.

네 실험의 숫자를 적어 둔다. 기계 64 대에서 별의 뿌리가 1,008 MB, 링의 기계 하나가 15.8 MB 였다. 16 MB 에서 200 바이트를 바꾼 판이 블록 4 KB 로 0.77 MB, 1 MB 로 16 MB 를 새로 저장했고, 다른 블록 197 개에서 트리 찾기가 전부 비교보다 느렸다. 행 2 백만 개의 열 하나가 행 저장에서 129 밀리초, 열 저장에서 23 에서 38 밀리초였다. 순열 백만 개가 파이썬으로 10 초, NumPy 로 67 밀리초였고 블록 섞기의 읽기가 완전 무작위보다 4 분의 1 빨랐다.

정정 하나를 남긴다. 머클 트리의 실험을 설계할 때 트리 찾기가 전부 비교보다 빠를 것으로 적어 두었다. 잰 값은 반대였다. 4,096 블록에서 로그가 이기려면 다른 블록이 170 개 아래여야 하는데 200 곳을 바꾸었으니 197 개였다. 명제를 먼저 쓰고 실험을 했다면 경계를 알았을 것이다. 실험을 먼저 하고 명제를 썼고, 명제가 실험을 설명했다. 그 순서를 여기 적어 둔다.

옛 구조의 판정

> Fisher-Yates 는 그대로 유효하다. 60 년 동안 한 줄도 안 바뀌었고 모든 라이브러리가 그것이다. 바뀐 것은 순열을 저장할 수 없을 만큼 n 이 커졌다는 조건이고, 그 조건에서 블록 섞기와 형식 보존 순열이 옆에 선다. 행 저장은 조정해서 유효하다. 행 하나를 읽는 일에서는 맞고 열 하나를 읽는 일에서는 안 맞으며, 근거는 실험 16.3 의 8 배다. 별 all-reduce 는 이제 안 맞는다. 뿌리의 바이트가 기계 수에 선형이고, 링이 그 자리를 하한으로 대신한다.

이 장의 요약

> 1. 링 all-reduce 는 기계 하나가 $2(p-1)/p$ 벡터를 보내고 그것이 하한이다. 단계 $2(p-1)$ 의 지연이 작은 벡터에서 시간이다 16.1
> 2. 내용 주소 저장은 해시가 주소인 표이고, 머클 트리는 다른 블록 k 개를 $2k \log n$ 비교로 찾는다. k 가 $n/2 \log n$ 을 넘으면 전부 비교다 16.2
> 3. 열 저장은 열 하나를 읽는 이동량을 레코드 대 열의 비만큼 줄인다. 메모리 맵은 그 이동량을 페이지 단위로 파일에 적용한다 16.3
> 4. Fisher-Yates 는 균등하고 $O(n)$ 이다. 블록 섞기는 무작위성을 블록에 가두어 순차를 얻고, Feistel 은 저장 없는 전단사다 16.4
> 5. 넷 다 잣대는 이동량이고 단위가 링크, 디스크, 페이지로 바뀐다 16.1
>
> 다음 장에서 뒤집히는 것: 이 장까지는 구조 하나씩을 세웠다. 마지막 장은 세운 것을 늘어놓고 고른다. 같은 일에 구조 둘이 있을 때 무엇으로 가르는가.

필자의 판단

> 필자는 분산과 저장을 자료 구조 수업의 마지막 주에 둔다. 새 구조가 없기 때문이다. 링 버퍼와 해시와 스트라이드와 뽑기를 배운 학생에게 이 장은 단위 환산이고, 그 환산이 되는 학생은

기계 천 대의 학습을 배열 하나로 읽는다. 그리고 머클 트리의 정정을 일부러 남겼다. 로그가 언제나 이긴다고 배운 것이 이 책에서 두 번째로 깨진 자리다. 첫 번째는 8장의 트리였다.

열린 문제

하나, 실험 16.1 은 모형이다. 실제 링크의 대역폭과 지연을 재지 않았고, 링 여러 개를 겹치는 구현의 시간도 모른다.
둘, 실험 16.2 의 변경은 무작위 200 곳이다. 실제 체크포인트에서 두 판이 블록 단위로 얼마나 같은지를 이 책은 재지 않았고, 그것이 내용 주소 저장의 값을 정한다.
셋, 실험 16.3 과 16.4 는 페이지 캐시 안이다. 디스크에서 드는 시간은 페이지 비와 무작위 읽기의 값으로 어림했을 뿐이다.

연습문제

16.1 기초 매개변수 70억 개를 2 바이트로 둔 기울기를 기계 8 대가 링으로 합칠 때 기계 하나가 보내는 바이트와 단계 수를 명제 16.2 로 구하라. 별의 뿌리는 얼마인가. 링크가 400 GB/s 이면 각각 몇 밀리초인가.

16.2 기초 블록 2^{20} 개의 머클 트리는 층이 몇 개인가. 다른 블록 100 개를 찾는 해시 비교 수를 명제 16.6 로 구하고, 전부 비교가 나아지는 k 의 경계를 적어라.

16.3 기초 열 여덟 개, 레코드 64 바이트, 행 10^8 개의 표에서 열 하나를 읽을 때 행 저장과 열 저장이 닿는 바이트를 명제 16.9 로 구하라. 10 GB/s 이면 각각 몇 밀리초인가.

16.4 기초 $n = 10^9$ 에서 명제 16.14 의 반쪽 비트 수와 정의역 크기를 구하고, 사이클 걷기의 기대 반복을 적어라. 순열을 저장하면 몇 GB 인가.

16.5 응용 이진 트리로 모으고 내려보내는 all-reduce 를 구현하라. 단계가 $2\log_2 p$ 인지, 기계 하나의 바이트가 링보다 큰지 시험하고, 어느 조건에서 트리가 링을 이기는지 명제 16.4 으로 논하라.

16.6 응용 구르는 해시로 블록 경계를 정하는 내용 정의 청킹을 구현하라. 한 바이트를 끼워 넣었을 때 고정 블록은 거의 전부를 다시 저장하고 내용 정의 블록은 하나만 다시 저장하는지 `dsai.merkle.ContentStore` 로 시험하라.

16.7 응용 정렬된 정수 열에 런 길이 부호화를 얹어라. 값과 길이 두 배열로 저장하고, 풀지 않고 합을 구하며, 저장이 원래의 백분의 일 아래인지 시험하라. 행 저장에서는 왜 이 압축이 안 되는가.

16.8 응용 `dsai.loader.block_shuffle` 의 블록 크기를 1 에서 n 까지 바꾸며 같은 블록에 연속으로 머무는 비율과 원소가 원래 자리에서 옮겨 간 거리의 평균을 재라. 어느 쪽이 블록 크기에 달렸고 어느 쪽이 무관한가.

16.9 탐구 링은 대역폭 최적이지만 단계가 p 에 선형이다. 벡터 크기와 기계 수의 평면에서 링과 트리의 경계를 명제 16.4 으로 그리고, 링 여러 개를 겹치는 것이 그 경계를 어떻게 옮기는지 논하라.

16.10 탐구 전체 순열, 블록 섞기, 형식 보존 순열이 각각 무엇을 무엇과 맞바꾸는지 정리 16.12 과 명제 16.13 와 명제 16.14 로 적고, 기계 천 대가 같은 데이터셋을 나눠 볼 때 어느 것을 골라야 하는지 논하라.

더 읽을 것

[64] 링 all-reduce 의 대역폭 최적성 증명과 클러스터 실측. 16.1 절의 하한이 이 논문의 정리다. 16.1
[65] 머클 트리의 원전. 잎 하나를 뿌리와 경로의 해시로 확인하는 구조가 여기 있다. 16.2
[66] 열 지향 데이터베이스. 읽기 부하에서 행 저장보다 한 자릿수 빠른 이유와 열마다 다른 압축. 16.3
[63] 균등 순열의 다섯 줄. 16.4 절의 정리가 이 알고리즘이다. 16.4

17장

자료 구조를 고르는 일

2000년의 책 하나는 뒷봉투 계산을 프로그래머의 첫 습관으로 적었다 [확립] [67]. 코드를 쓰기 전에 자릿수를 셈하라는 것이다. 이 책은 열여섯 장 동안 그 습관을 이동량과 루프라인으로 바꿔 장마다 한 번씩 했다. 마지막 장은 그 습관을 절차로 적고, 절차를 한 번 끝까지 돌리고, 절차의 어느 부분이 사람에게 남는지를 묻는다. 새 구조는 없다. 있는 것은 고르는 일이다.

0.2의 가정 넷이 이 장의 네 질문이다. 접근이 균일한가, 단위가 스칼라인가, 답이 정확해야 하는가, 구조가 고정인가. 열여섯 장이 그 넷을 하나씩 깨뜨렸고, 이 장은 그 넷으로 후보를 거른다. 그리고 장의 끝에 성적표가 있다. 전통 구조 열여섯 개의 판정과 근거 실험, 그리고 판정이 뒤집히는 축이다.

이 장에서 세우는 것

17.1 고르는 절차. 네 질문, 뒷봉투, 측정. 이 책의 실험 다섯 개에서 비가 1.4에서 24

17.2 사례 연구. 말뭉치 하나가 여섯 단계를 지나고, 병목은 구조가 아니라 파이썬 루프

17.3 코드의 값이 0일 때 남는 것. 같은 짝이 차원과 데이터의 모양에 따라 뒤집힌다

17.1 측정 없이 고르지 않는다. 뒷봉투 계산에서 루프라인까지

이 책은 열여섯 장 동안 구조 하나씩을 세우고 쟀다. 마지막 장은 세운 것을 고르는 일이다. 고르는 일에 새 구조는 없다. 있는 것은 절차 하나이고, 그 절차의 첫 문장이 이 절의 제목이다. 측정 없이 고르지 않는다. 0.2 의 가정 넷이 그 절차의 네 질문이다. 접근이 균일한가, 단위가 스칼라인가, 답이 정확해야 하는가, 구조가 고정인가. 네 질문의 답이 후보를 거르고, 0.1 의 이동량이 후보의 순서를 정하며, 측정이 그 순서를 확인한다.

순서를 정하는 계산이 뒷봉투 계산이다. 바이트를 대역폭으로 나누고 연산을 최고 연산으로 나눠 큰 쪽을 취한다. 0.2 의 루프라인이 그 계산이고, 이 절은 그것이 이 책의 실험에서 얼마나 맞았는지를 잰다.

학습 목표

> 고르는 절차를 네 질문과 뒷봉투 계산과 측정의 세 단계로 적을 수 있다. 뒷봉투 계산이 하한임을 보이고, 실측과의 비가 무엇을 뜻하는지를 이 책의 실험 다섯 개로 말할 수 있다.

이 말부터

> | | |
> |---|---|
> | 뒷봉투 계산 | 바이트와 연산 수만으로 시간의 자릿수를 어림하는 것. 루프라인의 하한이다 |
> | 후보 | 같은 일을 하는 구조 여럿. 정확성과 갱신 가능성이 성질이다 |
> | 비 | 실측을 뒷봉투 시간으로 나눈 값. 구현과 기계 사이의 거리다 |

디딤돌

디딤돌은 0.2 의 루프라인이다. 연산 밀도가 능선 아래면 대역폭이, 위면 연산이 시간을 정했다. 뒷봉투 계산은 그 두 항 중 큰 쪽이고, 그것이 어떤 구현도 못 넘는 하한이다.

정의 17.1: 뒷봉투 시간과 고르는 절차

> 연산 하나가 움직이는 바이트 b 와 연산 수 f, 기계의 대역폭 β 와 최고 연산 ϕ 에서 뒷봉투 시간은 $t_e = \max(b/\beta,\ f/\phi)$ 다. 고르는 절차는 셋이다. 첫째, 셋째 질문(정확한 답이 필요한가)과 넷째 질문(갱신이 있는가)으로 후보를 거른다. 둘째, 남은 후보를 t_e 순으로 세운다. 셋째, 위의 둘을 재어 비 t/t_e 를 적는다.
>
> **불변식** 어떤 구현의 실측 t 도 t_e 보다 작을 수 없다. 비가 1 에 가까우면 구현이 기계에 붙은 것이고, 크면 그만큼이 구현의 몫이다.

불변식이 이 절의 실험이다. 비가 1 아래로 내려가면 뒷봉투가 틀린 것이고, 비가 크면 뒷봉투가 아니라 구현이 문제다. 둘을 가르는 것이 측정의 값이다.

명제 17.2: 뒷봉투가 하한인 이유

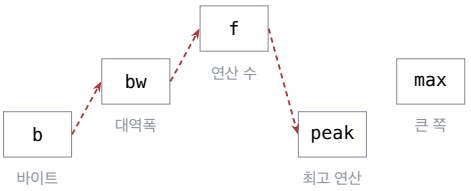
> 바이트 b 는 어떤 순서로 읽어도 대역폭 β 를 넘을 수 없으므로 b/β 이상이 걸리고, 연산 f 는 최고 연산 ϕ 를 넘을 수 없으므로 f/ϕ 이상이 걸린다. 둘은 겹칠 수 있어도 각각은 줄지 않으므로 큰 쪽이 하한이다. 따라서 뒷봉투가 실측보다 큰 경우는 b 나 f 를 잘못 셌을 때뿐이고, 그것이 명제를 고칠 신호다.

증명. 대역폭은 초당 바이트의 상한이고 최고 연산은 초당 연산의 상한이다. 상한의 역수가 시간의 하한이다. 이동과 연산이 완전히 겹쳐도 시간은 둘 중 긴 쪽보다 짧아지지 않는다. ■

명제의 둘째 문장이 이 책의 정정 장면 여럿을 설명한다. 실측이 뒷봉투 아래로 내려간 적이 있었고, 그때마다 셈이 틀렸다. 11장의 너비 우선 탐색이 그랬고, 이 절의 실험이 그 확인이다.

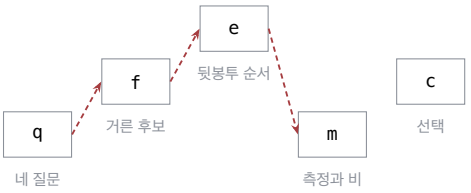
예제 17.1: 키 백만 개의 어텐션 조회

> 바이트 512 MB, 연산 2.6×10^8 이다. 이 기계의 대역폭 16.7 GB/s 와 최고 연산 192 GFLOPS 에서 $512/16.7 = 30.7$ 밀리초와 $0.26/192 = 1.3$ 밀리초이고, 뒷봉투는 30.7 밀리초다. 실측이 42 밀리초였으니 비가 1.4 다. 구현이 기계에 붙었다.



두 나눗셈과 max 하나

그림 17.1.. 뒷봉투 시간. 바이트를 대역폭으로, 연산을 최고 연산으로 나눠 큰 쪽을 취한다. 실측을 그 값으로 나눈 것이 비다



질문 → 후보 → 뒷봉투 → 측정 → 선택. 비가 크면 구현으로 돌아간다

그림 17.2.. 고르는 절차. 네 질문이 후보를 거르고, 뒷봉투가 순서를 정하고, 측정이 비를 적는다. 어느 단계도 건너뛰지 않는다

균열

이 책의 실험 다섯 개를 골라 뒷봉투 시간을 먼저 셈하고, 각 장의 측정 출력에서 실측을 그대로 읽어 비를 적었다. 대역폭과 최고 연산은 0장에서 이 기계에서 잰 값이다.

실험 17.1 뒷봉투와 실측. 이 책의 실험 다섯 개

실험	이동 MB	연산 GFLOP	뒷봉투 ms	실측 ms	비	3배 안
0장 순차 합	4.00	0.0010	0.24	2.67	11.1	아니오
15.1 어텐션 조회	512	0.26	30.7	42.3	1.38	예
15.2 타일 어텐션	16.8	68.7	358	5188	14.5	아니오
16.3 열 하나의 합	16.0	0.0020	0.96	22.6	23.6	아니오
16.4 순차 읽기	64.0	0.0010	3.83	18.6	4.85	아니오

측정 조건. 대역폭 16.7 GB/s, 최고 192 GFLOPS 는 0장의 루프라인 측정값. 이동량과 연산 수는 각 절의 명제. 실측은 해당 실험의 출력 파일에서 그대로 읽음. 순차 합은 float32 백만 개, 어텐션 조회는 키 백만 개, 타일 어텐션은 길이 16,384, 열 합은 행 2 백만 개, 순차 읽기는 레코드 백만 개.

다섯 중 하나만 3 배 안이다. 어텐션 조회의 비가 1.4 이고, 나머지는 4.9 에서 24 다. 비가 큰 쪽은 전부 파이썬이 원소마다 무엇인가를 한 자리다. 순차 합의 11 배는 NumPy 의 합이 아니라 무작위 순서와 견주려고 넣은 인덱스 배열의 값이고, 열 합의 24 배는 형 변환 복사다. 타일 어텐션의 14 배는 다르다. 연산이 시간을 정하는 자리인데 NumPy 의 float32 행렬 곱이 최고 연산의 7 퍼센트에서 돈다. 뒷봉투가 아니라 기계의 최고 연산이 이 구현에서 닿을 수 없는 값이다.

관찰

뒷봉투는 어느 실험에서도 실측을 넘지 않았다. 비는 1.4 에서 24 이고, 그 크기가 구현이 기계에서 얼마나 떨어져 있는지를 말한다. 파이썬 루프는 열 배, 형 변환은 스무 배, 행렬 곱은 열네 배다.

이 관찰이 뒷봉투의 쓸모다. 자릿수를 맞히는 것이 아니라 하한을 주는 것이고, 하한과 실측의 거리가 다음에 무엇을 고칠지를 말한다. 비가 24 인 구현을 다른 구조로 바꾸는 것은 순서가 틀렸다. 먼저 구현을 기계에 붙이고, 그다음에 구조를 견준다.

다시 쓰기

이 절의 다시 쓰기는 고르는 절차를 함수 셋으로 두는 것이다. 거르기, 세우기, 판정. 새 구조가 아니라 새 습관이고, 습관은 코드로 두어야 남는다.

명제 17.3: 비의 세 구간

> 비 $r = t/t_e$ 에 대해 $r < 1$ 이면 셈이 틀렸고, $1 \le r \le 3$ 이면 구현이 기계에 붙은 것이며, $r > 3$ 이면 구현의 값이 구조의 값보다 크다. 따라서 구조를 견주는 측정은 두 후보 모두 $r \le 3$ 일 때만 구조를 견준 것이고, 아니면 구현을 견준 것이다.

증명. 첫 구간은 명제 17.2 다. 둘째 구간의 3 은 이 책의 관찰이다. 순차 접근의 NumPy 연산이 대개 2 배 안에 들었고, 3 을 넘는 것은 예외 없이 파이썬 루프나 복사가 있었다. 셋째 구간에서 두 후보의 차이는 구현의 차이에 묻힌다. ■

이 명제가 이 책의 실험 표를 읽는 법이다. 표의 배속은 구조의 배속이 아니라 구현의 배속이고, 두 구현의 비가 비슷할 때만 구조가 보인다. 11장의 정정과 16장의 정정이 그 자리에서 나왔다. Bentley 의 뒷봉투 계산은 이 절차를 프로그래머의 습관으로 적었고, 자릿수를 먼저 셈한 뒤에 코드를 쓰라고 했다 [확립] [67]. 이 책은 그 습관에 루프라인의 두 항과 비의 판정을 붙였다.

알고리즘 17.1 고르기 `dsai.choose.rank`

> **입력 :** 후보 목록, 정확성 요구, 갱신 요구, 대역폭 β, 최고 연산 ϕ
> 1: 후보 ← 정확성과 갱신 요구를 만족하는 것만
> 2: 후보마다 $t_e \leftarrow \max(b/\beta, f/\phi)$; t_e 순으로 세운다
> 3: 위의 둘을 재어 $r = t/t_e$ 를 적는다. $r > 3$ 이면 구현을 먼저 고친다

구현

후보 하나, 뒷봉투 시간, 거르기, 세우기, 판정이다. 전부 합쳐 스무 줄이다.

지금 보는 코드 이동과 연산 중 오래 걸리는 쪽. 루프라인의 하한이다.

구현 뒷봉투 시간과 거르기 `samples/dsai/choose.py:22`

```
def envelope_seconds(nbytes: float, flops: float, bandwidth_gbs: float,
                     peak_gflops: float) -> float:
    """정의 17.1. 뒷봉투 시간. 이동과 연산 중 오래 걸리는 쪽. 루프라인의 하한이다."""
    return max(nbytes / (bandwidth_gbs * 1e9), flops / (peak_gflops * 1e9))


def filter_by_questions(cands: list[Candidate], need_exact: bool,
                        need_updates: bool) -> list[Candidate]:
    """알고리즘 17.1 의 첫 단계. 셋째와 넷째 질문이 후보를 거른다."""
    return [c for c in cands
            if (c.exact or not need_exact) and (not c.static or not need_updates)]
```

읽어 보면 `max` 하나가 정의다. `filter_by_questions` 가 셋째와 넷째 질문이고, 정확성이 필요하면 근사 구조가, 갱신이 필요하면 정적 구조가 빠진다.

구현 세우기와 판정 `samples/dsai/choose.py:35`

```
def rank(cands: list[Candidate], bandwidth_gbs: float,
         peak_gflops: float) -> list[tuple[Candidate, float]]:
    """둘째 단계. 뒷봉투 시간 순으로 세운다."""
    scored = [(c, envelope_seconds(c.nbytes, c.flops, bandwidth_gbs, peak_gflops))
              for c in cands]
    return sorted(scored, key=lambda t: t[1])


def within_factor(predicted: float, measured: float, factor: float = 3.0) -> bool:
    """셋째 단계의 판정. 실측이 예측의 1/f 배와 f 배 사이면 뒷봉투가 맞은 것이다."""
    return predicted / factor <= measured <= predicted * factor
```

읽어 보면 `sorted` 가 뒷봉투 순서이고 `within_factor` 가 명제 17.3 의 둘째 구간이다. 실험 17.1 의 마지막 열이 이 함수다.

분석

이 절은 구조가 아니라 절차를 세웠다. 절차는 이 책이 열여섯 장 동안 한 일을 순서대로 적은 것이다. 가정을 묻고, 이동량을 세고, 재고, 비를 적었다. 장마다 한 번씩 한 일이 이 절에서 함수 셋이 됐다.

이 절차가 서는 자리는 구조를 고르는 모든 회의다. 어느 인덱스가 좋은가, 어느 캐시가 좋은가. 답은 표에 있지 않고 질문 넷과 뒷봉투와 측정에 있다. 표는 그 셋을 한 번 한 결과이고, 작업량이 바뀌면 표도 바뀐다.

현장에서

> Bentley 는 뒷봉투 계산의 규칙을 하나로 적었다. 72 의 법칙과 리틀의 법칙처럼 자릿수를 주는 식 몇 개면 설계의 반은 끝난다는 것이다 [확립] [67]. 이 책의 식은 둘이다. 바이트 나누기 대역폭, 연산 나누기 최고 연산.

주의

> 실험 17.1 의 비는 이 기계의 NumPy 에서 잰 것이다. 가속기와 C 에서는 비가 1 에 가깝고, 그때는 구조의 차이가 그대로 드러난다. 이 책의 배속 표를 그 기계로 옮겨 읽으면 안 된다. 그리고 3 이라는 경계는 관찰이지 정리가 아니다. 다른 기계, 다른 언어에서 그 값은 다시 재야 한다.

절차를 세웠다. 다음 절은 그 절차를 한 번 끝까지 돌린다. 이 책의 표본 코드 전부를 말뭉치 하나 위에 잇고, 단계마다 어느 구조를 왜 골랐는지를 적는다.

되짚기

> 고르는 절차를 네 질문과 뒷봉투와 측정으로 정의하고, 뒷봉투가 하한임을 보였다. 이 책의 실험 다섯 개에서 비가 1.4 에서 24 였고, 비가 큰 자리는 전부 구현의 값이었다. 다음 절은 사례 연구다.

17.2 사례 연구. dsai 를 처음부터 끝까지

이 책의 표본 코드는 처음부터 하나의 꾸러미였다. 장마다 모듈 하나씩을 더했고, 모듈은 서로를 불렀다. 이 절은 그 꾸러미를 말뭉치 하나 위에 차례로 잇는다. 자르고, 거르고, 토큰으로 바꾸고, 세고, 벡터로 만들고, 인덱스를 세우고, 서빙 캐시에 넣는다. 단계마다 앞 장의 구조 하나를 그대로 쓴다. 새로 쓰는 코드는 잇는 것뿐이고, 그것이 사례 연구의 요점이다. 구조는 이미 있다. 고르고 잇는 것이 일이다.

17.1 의 절차를 단계마다 한 번씩 돌린다. 어느 질문이 후보를 걸렀고, 어느 구조가 남았고, 실측이 얼마인지. 그 기록이 이 절이다.

학습 목표

말뭉치 하나가 지나는 여섯 단계를 적고, 단계마다 어느 장의 구조가 왜 서는지를 네 질문으로 말할 수 있다. 단계별 시간과 메모리를 실측으로 적고, 가장 오래 걸린 단계가 왜 그런지를 앞 절의 비로 설명할 수 있다.

이 말부터

파이프라인	단계의 열. 앞 단계의 출력이 뒤 단계의 입력이다
단계	구조 하나가 서는 자리. 입력의 단위와 출력의 단위가 다르다
병목	시간이나 메모리가 가장 큰 단계. 파이프라인의 시간은 병목이 정한다

디딤돌

디딤돌은 0.3 의 문제다. 이 책이 끝까지 다시 본 문제는 벡터 n 개에서 가까운 k 개를 찾는 일이었고, 그 벡터가 어디서 오는지는 묻지 않았다. 이 절이 그 앞을 채운다. 텍스트에서 벡터까지.

정의 17.4: dsai 파이프라인

말뭉치 T 에 대해 여섯 단계를 둔다. 자르기 ($T \to$ 문서), 중복 제거 (문서 $\to$ 문서, 7장), 토큰화 (문서 $\to$ 토큰 열, 9장), 세기 (토큰 열 $\to$ 개수, 14장), 임베딩 (토큰 열 $\to$ 벡터, 6장), 인덱스 (벡터 $\to$ 인덱스, 13장), 서빙 캐시 (토큰 열 $\to$ 트리, 9장).

불변식 단계 i 의 출력 단위가 단계 $i+1$ 의 입력 단위다. 어느 단계도 뒤 단계의 구조를 모른다.

불변식의 둘째 문장이 꾸러미의 설계다. 단계가 서로를 모르므로 하나를 바꿔도 나머지가 그대로다. 그것이 고르기를 단계마다 따로 할 수 있는 이유다.

명제 17.5: 단계마다의 네 질문

> 중복 제거는 답이 정확할 필요가 없어 (셋째 질문) 확률 구조가 선다. 토큰화는 병합 순서가 고정이라 (넷째 질문) 정적 구조가 선다. 세기는 정확해야 하고 갱신이 없어 정렬된 배열이 선다. 임베딩은 단위가 벡터라 (둘째 질문) 해싱 트릭이 선다. 인덱스는 답이 근사여도 되고 갱신이 없어 역파일이 선다. 서빙 캐시는 갱신이 잦고 접두사를 나눠 쓰므로 기수 트리가 선다.

증명. 각 문장은 그 장의 정의와 판정을 그대로 옮긴 것이다. 7장의 블룸 필터와 MinHash는 오답을 허용하는 대신 메모리를 줄였고, 9장의 병합 표는 학습 뒤 고정이며, 14장의 접미사 배열은 정확한 개수를 배열 하나로 낸다. 6장의 해싱 트릭은 어휘를 고정 차원으로 접고, 13장의 역파일은 재현율을 내주고 시간을 얻으며, 9장의 기수 트리는 접두사 공유가 갱신과 같이 있는 자리다. ■

명제가 절차의 첫 단계다. 둘째 단계인 뒷봉투는 단계마다 이미 각 장에서 셈했다. 남은 것은 셋째, 한 번에 이어 재는 것이다.

예제 17.2: 문서 254 개

> 400 KB 를 2 KB 로 자르면 문서 196 개이고, 3 할을 복제하면 254 개다. 중복 제거가 58 개를 걸러 196 개로 돌려놓는다. 토큰 13만 개는 병합 500 번의 어휘 756 개로 적히고, 벡터는 64 차원 196 개라 50 KB 다. 인덱스가 군집 16 개이고 질의가 문서 전부다.

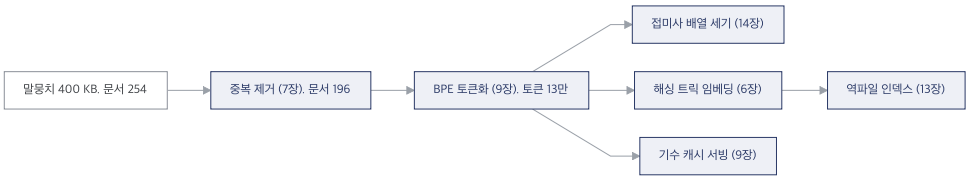


그림 17.3.. dsai 파이프라인. 말뭉치가 여섯 단계를 지나고, 단계마다 앞 장의 구조 하나가 선다. 토큰 열에서 세기와 임베딩과 서빙 캐시가 갈라진다

균열

표준 라이브러리 소스 400 KB 를 여섯 단계로 흘리고, 단계마다 시간과 최대 메모리를 쟀다. 단계 사이의 중간 결과는 다음 단계가 받은 뒤 버렸고, 그래서 메모리는 단계마다 따로다.

실험 17.2 여섯 단계의 시간과 메모리

단계	쓴 구조	출력	ms	최대 MB
자르기와 복제	배열	문서 254	2.24	0.40
중복 제거	블룸, MinHash, LSH (7장)	문서 196	1454	3.61
토큰화	BPE (9장)	토큰 13만, 어휘 756	17254	20.7
세기	접미사 배열 (14장)	최다 쌍 354 번	616	11.6
임베딩	해싱 트릭 (6장)	64차원	53.8	0.13
인덱스와 질의	역파일 (13장)	재현율 84 퍼센트	88.3	0.17
서빙 캐시	기수 트리 (9장)	적중 100 퍼센트	12.3	0.12

측정 조건. 표준 라이브러리 소스 400,000 바이트를 2 KB 문서로 자르고 3 할을 복제. BPE 병합 500. 메모리는 tracemalloc 의 단계별 최대. 한 번 측정.

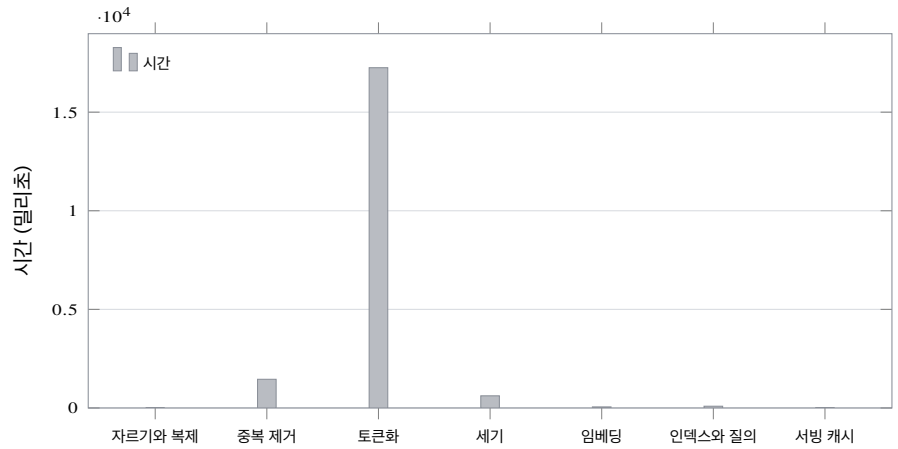


그림 17.4.. 여섯 단계의 시간. 토큰화 하나가 나머지 전부의 일곱 배다

토큰화가 17 초로 전체의 88 퍼센트다. 문서 196 개의 단어마다 병합 표를 파이썬 루프로 적용하는 값이고, 9장의 실험이 이미 그 값을 쟀다. 중복 제거가 1.5 초, 접미사 배열이 0.6 초이며 나머지 셋은 100 밀리초 아래다. 메모리는 토큰화의 21 MB 가 최대다. 단어마다 리스트 하나를 만드는 파이썬 객체의 값이고, 토큰 13만 개 자체는 1 MB 다.

관찰

병목은 구조가 아니라 구현이었다. 토큰화의 17 초는 병합 표의 값이 아니라 파이썬 루프의 값이고, 명제 17.3 의 셋째 구간이다. 구조를 바꿀 자리가 아니라 구현을 옮길 자리다.

이 관찰이 사례 연구의 결론이다. 여섯 단계 중 다섯에서 구조가 이미 맞았고, 병목 하나는 구조의 문제가 아니었다. 앞 절의 절차가 그 둘을 가른다.

다시 쓰기

이 절의 다시 쓰기는 단계를 함수로, 파이프라인을 함수의 합성으로 두는 것이다. 각 함수는 앞 장의 모듈을 부를 뿐이고, 자기 자료 구조가 없다.

명제 17.6: 파이프라인의 비용

> 단계 i 의 시간이 t_i 이면 파이프라인의 시간은 $\sum_i t_i$ 이고, 최대 메모리는 $\max_i m_i$ 다. 단계 사이의 중간 결과를 버리면 메모리가 합이 아니라 최대다. 따라서 시간은 병목 단계를 줄여야 줄고, 메모리는 가장 큰 단계를 줄여야 준다. 둘은 대개 다른 단계다.

증명. 단계가 차례로 돌므로 시간은 합이다. 단계 i 가 끝나면 그 입력을 버릴 수 있으므로 어느 순간의 메모리는 그 단계의 것뿐이고 전체의 최대가 된다. 실험에서 시간의 병목은 토큰화이고 메모리의 병목도 토큰화였지만, 그것은 우연이다. ■

이 명제가 11.3 의 융합과 같은 모양이다. 거기서는 중간 텐서를 안 내는 것이 이동량이었고, 여기서는 중간 결과를 버리는 것이 메모리다. 같은 잣대다.

알고리즘 17.2 파이프라인 `dsai.pipeline.tokenize_docs`

> **입력:** 말뭉치 T
> 1: 문서 ← 자르고 복제; 문서 ← 중복 제거 (7장)
> 2: 토큰 열 ← BPE (9장); 개수 ← 접미사 배열 (14장)
> 3: 벡터 ← 해싱 트릭 (6장); 인덱스 ← 역파일 (13장); 캐시 ← 기수 트리 (9장)

구현

단계 여섯이 함수 여섯이다. 앞 장의 모듈을 부르는 것뿐이라 짧다.

지금 보는 코드 단어 빈도로 병합을 학습하고, 문서마다 단어를 병합 순서로 적는다.

구현 중복 제거와 토큰화 `samples/dsai/pipeline.py:30`

```
def dedup_docs(docs: list[str]) -> list[str]:
    """7장. 블룸 필터로 정확 중복, MinHash 와 LSH 로 근사 중복을 거른다."""
    p = DedupPipeline(expected=len(docs), k_sig=64, bands=16, threshold=0.7)
    return [d for i, d in enumerate(docs) if p.offer(i, d)]


def tokenize_docs(docs: list[str], num_merges: int) -> tuple[list[list[int]], int]:
    """9장. 단어 빈도로 BPE 를 학습하고 문서를 토큰 열로 바꾼다. (토큰 열들,
    어휘 크기)."""
    words: Counter = Counter()
    for d in docs:
        for w in d.encode().split(b" "):
            if w:
                words[b" " + w] += 1
    merges = train_bpe(words, num_merges)
    rank = {pair: i for i, pair in enumerate(merges)}
    out = []
    for d in docs:
        ids: list[int] = []
        for w in d.encode().split(b" "):
            if w:
                ids.extend(encode_by_merges(b" " + w, rank))
        out.append(ids)
    return out, 256 + len(merges)
```

읽어 보면 `DedupPipeline.offer` 가 7장 전부이고 `train_bpe` 와 `encode_by_merges` 가 9장이다. 안쪽 `for` 가 실험의 17 초이고, 단어마다 파이썬이 병합 표를 훑는다.

구현 임베딩과 인덱스 `samples/dsai/pipeline.py:66`

```
def embed_docs(token_lists: list[list[int]], d: int) -> np.ndarray:
    """6장의 해싱 트릭. 토큰 번호를 d 로 접어 센 벡터. (문서 수, d) float32, 정규화."""
    x = np.zeros((len(token_lists), d), dtype=np.float32)
    for i, ids in enumerate(token_lists):
        np.add.at(x[i], np.array(ids, dtype=np.int64) % d, 1.0)
    norms = np.linalg.norm(x, axis=1, keepdims=True)
    return x / np.maximum(norms, 1e-9)


def index_recall(x: np.ndarray, k: int, nlist: int, nprobe: int,
                 rng: np.random.Generator) -> tuple[float, int]:
    """13장. 역파일을 만들고 문서 전부를 질의로 재현율@k 를 잰다. (재현율,
    본 벡터 평균)."""
    ivf = IVF(x, kmeans(x, nlist, rng, iters=10))
    hits, seen = 0, 0
    for q in x:
        truth = set(np.argsort(((x - q) ** 2).sum(axis=1))[:k].tolist())
        found, n = ivf.search(q, nprobe, k)
        hits += len(set(found.tolist()) & truth)
        seen += n
    return hits / (k * len(x)), seen // len(x)
```

읽어 보면 `ids % d` 가 해싱 트릭이다. 어휘 756 을 64 로 접고, 충돌은 그대로 둔다. `index_recall` 이 13장의 역파일에 문서 전부를 질의로 넣는다.

분석

이 절은 이 책의 구조 여섯 개를 한 줄에 놓았다. 새로운 것은 잇는 코드 백 줄뿐이다. 그 백 줄이 말하는 것은 구조가 서로 독립이고, 고르기가 단계마다 따로 된다는 것이다. 그리고 병목이 구조에 없었다. 열일곱 장을 읽고 나서 마주치는 가장 흔한 상황이 그것이다. 구조는 맞고 구현이 느리다. 그때 할 일은 앞 절의 비를 적는 것이다.

이 구조가 서는 자리는 모든 데이터 준비 파이프라인이다. 말뭉치가 문서가 되고 토큰이 되고 벡터가 되는 길은 어디서나 같고, 단계마다 서는 구조도 같다. 다른 것은 규모이고, 규모가 바뀌면 각 장의 실험을 그 규모에서 다시 돌린다.

현장에서

> 이 절의 말뭉치는 파이썬 표준 라이브러리 소스 400 KB 다. 실제 학습 말뭉치는 그 백만 배이고, 그때 중복 제거의 LSH 와 접미사 배열의 만들기가 병목이 된다. 7장과 14장의 열린 문제가 그 자리다.

주의

> 실험 17.2 는 한 번 잰 값이다. 토큰화의 17 초는 되풀이해도 비슷하지만, 100 밀리초 아래의 단계는 한 번의 측정으로 순서를 말할 수 없다. 그리고 인덱스의 재현율 84 퍼센트는 문서 196 개에 군집 16 개인 작은 실험의 값이다. 벡터 백만 개에서 재는 재현율은 13장의 표가 말한다.

말뭉치 하나가 여섯 단계를 지났고, 단계마다 앞 장의 구조 하나가 그대로 섰다. 마지막 절은 이 절차를 한 번 더 뒤집는다. 코드를 쓰는 값이 0 에 가까워질 때, 절차의 어느 단계가 사람에게 남는가.

되짚기

> 여섯 단계의 파이프라인을 정의하고, 단계마다 네 질문으로 구조를 골랐다. 실측에서 토큰화가 17 초로 88 퍼센트였고, 그것은 구조가 아니라 파이썬 루프의 값이었다. 다음 절은 AI 가 코드를 쓰는 시대의 선택이다.

17.3 AI 가 코드를 쓰는 시대의 선택

0.2 의 넷째 가정은 구조를 사람이 설계한다는 것이었다. 이 책은 그 가정이 두 방향에서 깨지는 것을 봤다. 8장의 학습 인덱스는 구조의 일부를 데이터에서 배웠고, 그 밖의 모든 장에서 구현은 점점 사람이 안 쓰는 것이 되어 간다. 코드를 쓰는 값이 0 에 가까워지면 무엇이 남는가. 17.1 의 절차에서 코드는 셋째 단계의 도구였다. 남는 것은 첫째와 둘째, 그리고 셋째의 판정이다. 질문하고, 셈하고, 재고, 비를 적는 일.

이 절은 그 남는 일을 실험 하나로 보여 준다. 같은 짝을 규모와 차원을 바꾸며 다시 재면 판정이 뒤집힌다. 뒤집히는 자리를 아는 것이 선택이고, 그것은 코드가 아니라 측정에서 나온다.

학습 목표

> 코드 생성의 값이 줄 때 고르는 절차의 어느 단계가 남는지를 적고, 판정이 규모와 차원에 따라 뒤집히는 것을 실측으로 보일 수 있다. 빠른 쪽이 이긴 것이 아닌 경우를 재현율로 가려내고, 전통 구조의 성적표를 근거 실험과 함께 읽을 수 있다.

이 말부터

판정	어느 구조가 이 작업량에서 맞는가. 그대로 유효, 조정해서 유효, 이제 안 맞음
뒤집히는 자리	규모나 차원이 바뀌어 판정이 바뀌는 경계. 작업량이 정한다
성적표	전통 구조 열여섯 개의 판정과 근거 실험 번호. 이 장의 끝에 있다

디딤돌

디딤돌은 12.1 의 저주다. kd-트리는 2차원에서 로그였고 16차원에서 전부 비교였다. 같은 구조, 같은 코드인데 차원이 판정을 바꿨다. 그 경계를 아는 것이 이 절의 일이고, 코드는 그 경계를 모른다.

정의 17.7: 판정과 뒤집히는 자리

> 작업량 w 에서 구조 A 와 B 의 판정은 실측 $t_A(w), t_B(w)$ 와 답의 질 $q_A(w), q_B(w)$ 의 짝이다. 시간이 빠르고 질이 요구를 넘는 쪽이 이긴다. 뒤집히는 자리는 w 를 한 축으로 바꿀 때 이기는 쪽이 바뀌는 경계다.
>
> **불변식** 판정은 w 의 함수이지 구조의 성질이 아니다. 어느 구조도 모든 w 에서 이기지 않는다.

불변식이 이 책의 성적표를 읽는 법이다. 성적표의 판정은 이 책의 작업량에서 낸 판정이고, 작업량이 다르면 다시 잰다. 그것을 실험으로 보여 준다.

명제 17.8: 코드의 값이 0 일 때 남는 것

> 정의 17.1 의 절차에서 코드는 셋째 단계의 측정 도구다. 코드를 쓰는 값이 0 이면 후보를 전부 구현해 재는 것이 공짜가 되고, 남는 값은 무엇을 후보로 둘지 (첫째 단계), 어떤 작업량에서 잴지, 그리고 잰 것을 어떻게 판정할지다. 따라서 남는 일은 질문과 작업량과 판정이고, 셋 다 0.2 의 가정 넷을 아는 것에서 나온다.

증명. 절차의 세 단계 중 코드가 드는 것은 셋째뿐이다. 그 값이 0 이면 셋째는 자동이고, 첫째의 질문과 둘째의 셈과 셋째의 판정은 여전히 사람이 작업량을 알아야 한다. 작업량은 코드에 없다. ■

명제가 이 책의 마지막 주장이다. 구조를 짜는 값은 줄고, 구조를 고르는 값은 그대로다. 고르는 값은 작업량을 아는 값이고, 작업량은 측정으로만 안다.

예제 17.3: 같은 짝, 다른 판정

> kd-트리 대 전부 비교는 2 차원에서 트리가 두 배 빠르고 8 차원에서 열 배 느리다. 역파일 대 전부 비교는 천 개에서 백만 개까지 역파일이 빠르지만, 군집이 없는 점에서 재현율이 25 퍼센트에서 7 퍼센트다. 앞의 짝은 차원이 판정을 뒤집고, 뒤의 짝은 시간이 아니라 질이 판정을 뒤집는다.

그림 17.5.. 차원 축에서 뒤집히는 자리. 2 에서 kd-트리가 이기고 8 부터 전부 비교가 이긴다. 경계는 4 근처다

앞 셋은 kd-트리 대 전부 비교, 뒤 둘은 역파일 대 전부 비교. 진한 축이 판정을 바꾼 축이다

그림 17.6.. 판정이 뒤집히는 두 축. 차원이 오르면 kd-트리가 지고, 군집이 없으면 역파일이 빠르되 답이 틀린다. 시간만 보면 뒤의 판정을 놓친다

균열

두 짝을 규모와 차원을 바꾸며 다시 쟀다. 12 장과 13 장의 실험을 작업량만 바꿔 되풀이한 것이다.

실험 17.3 판정을 뒤집는 조건. 두 짝, 여섯 작업량

짝	조건	전통 ms	새 구조 ms	재현율	빠른 쪽
kd-트리 대 전부	차원 2, 2만 개	1.46	0.90	100	kd-트리
kd-트리 대 전부	차원 8, 2만 개	2.24	22.7	100	전부 비교
kd-트리 대 전부	차원 32, 2만 개	4.14	112	100	전부 비교
역파일 대 전부	차원 64, 천 개	0.33	0.24	25.5	역파일
역파일 대 전부	차원 64, 만 개	3.60	0.31	12.0	역파일
역파일 대 전부	차원 64, 십만 개	42.8	0.67	7.00	역파일

측정 조건. 질의 20 개의 평균. kd-트리는 잎 16, 역파일은 군집 $\sqrt{n}$ 에 nprobe 2. 무작위 정규 분포 점이라 군집이 없다. 세 번 중앙값.

kd-트리는 2차원에서 이기고 8차원부터 진다. 12장의 저주 그대로이고, 뒤집히는 자리가 차원 4 근처다. 역파일은 세 규모 모두 전부 비교보다 빠르지만 재현율이 25, 12, 7 퍼센트다. 군집이 없는 점에서 nprobe 2 는 이웃의 대부분을 다른 군집에 두고 못 본다. 13장의 실험에서 역파일의 재현율이 94 퍼센트였던 것은 점에 군집이 있어서였다. 같은 구조, 같은 설정, 다른 데이터, 다른 판정이다.

관찰

빠른 쪽이 이긴 것이 아니다. 역파일은 여섯 번 다 빨랐고 한 번도 이기지 않았다. 판정은 시간과 질의 짝이고, 짝의 한쪽만 보면 틀린다.

이 관찰이 코드가 못 하는 일이다. 코드는 두 구조를 구현하고 시간을 잰다. 무엇을 질로 잴지, 어느 데이터에서 잴지, 재현율 7 퍼센트가 되는지 안 되는지는 작업량을 아는 사람이 정한다.

다시 쓰기

이 절의 다시 쓰기는 성적표다. 열여섯 장의 판정을 한 표에 놓되, 판정마다 근거 실험을 붙인다. 근거가 없는 판정은 없다.

명제 17.9: 성적표의 조건

판정 하나는 구조, 판정, 근거 실험, 뒤집히는 축의 넷으로 적는다. 근거 실험은 이 책의 표에 있어야 하고, 뒤집히는 축은 그 실험에서 바꾼 변수여야 한다. 따라서 성적표는 결론이 아니라 실험 목록이고, 독자의 작업량에서 그 축을 다시 재면 판정이 다시 나온다.

증명. 근거가 실험이면 재현 가능하고, 축이 변수이면 다시 잴 수 있다. 둘 중 하나가 빠지면 판정이 의견이 된다. 이 책의 규칙은 의견을 싣지 않는 것이었다. ■

성적표는 이 장의 끝에 한 쪽으로 있다. 열여섯 줄 중 그대로 유효가 일곱, 조정해서 유효가 일곱, 이제 안 맞음이 둘이다. 셋으로 나눈 것은 이 책의 규칙이었고, 어느 칸에 놓을지는 각 장의 실험이 정했다.

알고리즘 17.3 규모를 바꿔 다시 재기 `dsai.choose.within_factor`

입력: 짝 (A, B), 작업량의 축 하나, 축의 값 여럿, 질의 잣대
1: **모든** 축의 값 **동안** 둘을 재고 시간과 질을 적는다
2: **끝 반복**
3: 이기는 쪽이 바뀌는 값이 뒤집히는 자리다. 질이 요구 아래면 시간이 빨라도 진 것이다

구현

이 절의 코드는 실험 스크립트 하나다. 12장과 13장의 모듈을 부르고 축만 바꾼다.

지금 보는 코드 같은 질의 20 개로 두 구현을 재고, 역파일은 재현율까지 적는다.

구현 차원을 바꾸는 짝 `samples/bench/c17_scale.py:18`

```
def sweep_dims(n: int, dims: list[int], rng: np.random.Generator) -> list[list]:
    """축 하나. 차원을 바꾸며 kd-트리와 전부 비교를 잰다.
    둘 다 정확하므로 재현율 100."""
    rows = []
    for d in dims:
        pts = rng.standard_normal((n, d))
        qs = rng.standard_normal((20, d))
        tree = KDTree(pts)
        ms_old = median_ms(lambda: [brute_nearest(pts, q) for q in qs], 3) / 20
        ms_new = median_ms(lambda: [tree.nearest(q) for q in qs], 3) / 20
        winner = "kd-트리" if ms_new < ms_old else "전부 비교"
        rows.append(["kd-트리 대 전부", f"차원 {d}, {n // 10000}만 개", ms_old, ms_new,
                     100.0, winner])
    return rows
```

읽어 보면 `for d in [2, 8, 32]` 가 축이다. `tree.nearest` 와 `brute_nearest` 가 12장 그대로이고, 새 코드는 축을 도는 루프뿐이다.

분석

이 절은 구조가 아니라 판정을 세웠다. 판정은 작업량의 함수이고, 작업량은 코드에 없다. 코드를 쓰는 값이 0 에 가까워질수록 그 사실이 선택의 전부가 된다. 이 책의 열여섯 판정은 이 책의 작업량에서 낸 것이다. 독자의 작업량은 다르고, 그때 할 일은 성적표를 믿는 것이 아니라 성적표의 근거 실험을 자기 데이터에서 다시 돌리는 것이다. 표본 코드 전부가 그것을 위해 있다.

이 절이 서는 자리는 이 책을 덮은 뒤다. 새 구조가 나오고 새 논문이 나온다. 그때마다 같은 절차다. 네 질문, 뒷봉투, 측정, 비. 구조는 바뀌고 절차는 안 바뀐다.

현장에서

> 8장의 학습 인덱스는 구조의 일부를 데이터가 정하는 첫 예였고, 실험 8.3 에서 정렬된 배열의 이진 탐색을 이기지 못했다. 구조를 학습하는 것과 구조를 고르는 것은 다른 일이고, 이 책이 다룬 것은 뒤쪽이다.

주의

> 실험 17.3 의 역파일은 군집이 없는 점에서 재현율이 낮았다. 실제 임베딩은 대개 군집이 있어 13장의 값에 가깝다. 이 실험이 보이는 것은 역파일이 나쁘다는 것이 아니라, 데이터의 모양이 판정의 축이라는 것이다. 그리고 성적표의 뒤집히는 축은 이 책이 실제로 바꿔 본 축만 적었다. 안 바꿔 본 축에서 판정이 어떻게 되는지는 모른다.

열일곱 장이 끝났다. 세운 것은 구조 열여섯 개의 판정과 절차 하나다. 판정은 이 책의 작업량에서 참이고, 절차는 어느 작업량에서든 같다. 그 절차를 손에 쥔 독자가 이 책이 끝나는 자리다.

되짚기

> 코드의 값이 0 일 때 남는 것이 질문과 작업량과 판정임을 보이고, 같은 짝이 차원과 데이터의 모양에 따라 뒤집히는 것을 실측했다. 역파일은 여섯 번 빠르고 한 번도 이기지 않았다. 이 장의 끝에 전통 구조 성적표가 있다.

17장을 닫으며

이 장이 세운 것은 절차 하나와 표 하나다. 절차는 네 질문과 뒷봉투와 측정이고, 표는 그 절차를 열여섯 번 돌린 결과다.

세 실험의 숫자를 적어 둔다. 이 책의 실험 다섯 개에서 뒷봉투와 실측의 비가 1.4 에서 24 였고, 어느 실험도 뒷봉투 아래로 내려가지 않았다. 말뭉치 400 KB 의 여섯 단계에서 토큰화가 17 초로 88 퍼센트였다. kd-트리는 2 차원에서 전부 비교보다 두 배 빠르고 8 차원에서 열 배 느렸으며, 역파일은 군집 없는 점에서 여섯 번 다 빠르고 재현율이 25 에서 7 퍼센트였다.

정정 하나를 남긴다. 규모 실험을 설계할 때 역파일이 작은 규모에서 전부 비교에 질 것으로 적어 두었다. 잰 값은 천 개에서도 역파일이 빨랐다. 대신 재현율이 25 퍼센트였다. 시간만 적었다면 역파일이 이겼다고 썼을 것이다. 질을 같이 적고 나서야 판정이 뒤집혔고, 그것이 이 절의 관찰이 됐다. 판정은 시간과 질의 짝이라는 문장은 그 실험에서 나왔다.

옛 구조의 판정

> 뒷봉투 계산은 그대로 유효하다. 2000 년의 습관이 루프라인의 두 항으로 정확해졌을 뿐이고, 이 책의 어느 실험도 그 하한 아래로 안 내려갔다. 근거는 실험 17.1 이다. 비가 1.4 에서 24 였고, 큰 비는 전부 구현의 값이었다. 복잡도 표로 구조를 고르는 것은 이제 안 맞는다. 표는 첫째 가정 위에 있고, 그 가정이 깨진 자리에서 표의 순위가 실측과 다르다. 그 자리를 측정이 대신한다.

이 장의 요약

> 1. 고르는 절차는 네 질문으로 거르고, 뒷봉투로 세우고, 재어 비를 적는 셋이다. 비가 3 을 넘으면 구조가 아니라 구현이다 17.1
> 2. 뒷봉투는 하한이다. 실측이 그 아래면 셈이 틀린 것이고, 그것이 명제를 고칠 신호다 17.1
> 3. 파이프라인의 시간은 단계의 합이고 메모리는 최대다. 병목은 대개 구조가 아니라 구현에 있다 17.2
> 4. 코드의 값이 0 이면 남는 것은 질문과 작업량과 판정이다. 셋 다 작업량을 아는 데서 나온다 17.3
> 5. 판정은 시간과 질의 짝이고 작업량의 함수다. 성적표는 결론이 아니라 실험 목록이다 17.3
> 이 책이 끝나는 자리 : 아래 성적표의 근거 실험을 자기 데이터에서 다시 돌리는 것.

필자의 판단

> 필자는 자료 구조 수업의 마지막 시간을 고르는 일에 쓴다. 학기 내내 세운 구조를 늘어놓고, 작업량 하나를 주고, 네 질문과 뒷봉투로 후보를 줄인 뒤 재게 한다. 답을 맞힌 학생보다 뒤집히는 축을 찾은 학생이 그 시간을 제대로 보낸 것이다. 코드를 쓰는 값이 줄어드는 시대에 그 시간이 수업의 중심이 된다고 본다. 이 책은 그 시간을 열일곱 장으로 늘린 것이다.

열린 문제

> 하나, 실험 17.1 의 3 배 경계는 이 기계와 NumPy 의 관찰이다. 가속기와 다른 언어에서 그 경계가 어디인지 재지 않았다.

둘, 실험 17.3 은 축을 하나씩만 바꿨다. 차원과 규모와 군집을 동시에 바꾼 평면에서 판정의 경계가 어떤 모양인지 모른다.
셋, 성적표의 뒤집히는 축은 이 책이 실제로 바꿔 본 축만 적었다. 안 바꿔 본 축은 비어 있고, 그 칸을 채우는 것은 독자의 실험이다.

연습문제

17.1 기초 벡터 10^7 개, 차원 128, float32 를 전부 비교하는 질의 하나의 뒷봉투 시간을 정의 17.1 로 구하라. 대역폭 16.7 GB/s, 최고 192 GFLOPS 다. 어느 항이 지배하는가.

17.2 기초 실험 17.1 의 다섯 비 중 어느 것이 파이썬 루프의 값이고 어느 것이 최고 연산의 값인지 명제 17.3 로 갈라 적어라.

17.3 기초 명제 17.6 로 실험 17.2 의 전체 시간과 최대 메모리를 구하고, 토큰화를 열 배 빠르게 하면 전체 시간이 몇 배 주는지 적어라.

17.4 기초 정의 17.7 대로 실험 17.3 의 여섯 줄 각각에서 이긴 쪽을 적어라. 재현율 요구가 50 퍼센트일 때와 5 퍼센트일 때 답이 어떻게 다른가.

17.5 응용 7장의 블룸 필터 조회에 뒷봉투 계산을 먼저 적고 실측과 견줘라. 해시 k 개가 각각 라인 하나를 당긴다고 셈하고, 실측이 하한 위에 있는지 시험하라.

17.6 응용 파이프라인의 임베딩을 해싱 트릭 대신 어휘 크기의 밀집 벡터로 바꾸고, 메모리와 이웃의 겹침을 견줘라. 어휘 756 을 64 로 접은 것이 무엇을 잃었는가.

17.7 응용 배열 훑기와 해시 조회를 규모 10 에서 십만까지 재어 뒤집히는 자리를 찾아라. 작은 규모에서 훑기가 왜 불리하지 않은가.

17.8 응용 성적표의 첫 줄을 코드로 다시 확인하라. 정렬된 배열의 이진 탐색과 8장의 Eytzinger 배치가 같은 답을 내는지 시험하고, 어느 판정의 근거인지 적어라.

17.9 탐구 명제 17.8 를 근거로, 후보 열 개를 전부 구현해 재는 것이 공짜일 때 고르는 절차에서 무엇이 여전히 어려운지 논하라. 작업량을 어떻게 적어야 그 측정이 뜻을 갖는가.

17.10 탐구 성적표의 판정 하나를 골라 뒤집히는 축을 자기 데이터에서 다시 재라. 판정이 바뀌면 그 이유를 이 책의 명제 하나로 적고, 안 바뀌면 어느 범위까지 안 바뀌는지 적어라.

더 읽을 것

[67] 뒷봉투 계산과 자료 구조 고르기의 원전. 자릿수를 먼저 셈하는 습관이 이 장의 절차다. 17.1
[2] 루프라인 모델. 뒷봉투의 두 항이 이 모델의 두 선이다. 17.1
[30] 학습 인덱스. 구조의 일부를 데이터가 정하는 첫 예이고, 17.3 절의 넷째 가정이 깨지는 자리다. 17.3

전통 구조 성적표

열여섯 구조의 판정이다. 판정은 이 책의 작업량에서 낸 것이고, 근거 실험을 자기 데이터에서 다시 돌리면 판정이 다시 나온다. 뒤집히는 축은 그 실험에서 실제로 바꾼 변수다.

구조	판정	근거	뒤집히는 축과 한 줄 근거
배열	그대로 유효	실험 1.1	원소가 참조면 크기 여덟 배, 시간 열 배. 연속성이 가장 싼 이동량
동적 배열	조정해서 유효	실험 2.3	배열의 수. 하나면 두 배 늘리기, 수백 개면 블록 테이블
희소 행렬	조정해서 유효	실험 3.1	희소도의 문턱이 50 에서 2 퍼센트로. 규칙 있는 희소성이 그 사이
연결 리스트	조정해서 유효	실험 4.1, 4.3	노드가 흩어지면 의존 읽기 105 ns. 아레나 위의 인덱스 링크만 남는다
스택, 큐, 덱	그대로 유효	실험 5.1, 5.3	배열 위의 스택과 링 버퍼가 지금도 가장 싸다. 원소가 미분과 요청으로 바뀌었다
해시 테이블	조정해서 유효	실험 6.1, 6.4	배치 크기. 크면 정렬이 표를 이기고, 라인 수가 순위를 정한다
블룸 필터와 스케치	그대로 유효	실험 7.1, 7.3	오답률과 S 곡선이 원 논문 그대로. 규모만 커졌다
이진 탐색 트리	조정해서 유효	실험 8.2, 8.4	높이의 단위가 비교에서 라인으로. B-트리와 정렬 배열이 남는다
트라이	그대로 유효	실험 9.3, 9.4	접두사 공유의 절약이 명제 그대로. 노드에 적히는 것만 바뀌었다
이진 힙	조정해서 유효	실험 10.1, 10.2	상위 k 를 한 번에 고르면 선택이 여덟 배. 스트림에서는 힙
인접 리스트	조정해서 유효	실험 11.1	정적이면 CSR. 메모리 5 분의 1, 훑기 이백 배
kd-트리	이제 안 맞음	실험 12.1, 17.3	차원. 2 에서 이기고 8 에서 열 배 진다. 근접 그래프가 대신
벡터 양자화	그대로 유효	실험 13.2, 13.3	바이트 두 배에 오차 반. 목적이 압축에서 거리 어림으로
접미사 배열	그대로 유효	실험 14.1, 14.2	정수 n 개로 모든 부분 문자열의 개수. 만드는 값이 남은 문제
LRU 캐시	조정해서 유효	실험 15.3	참조 패턴. 시간 지역성이 아니라 질량 지역성. 헤비 히터가 대신
별 all-reduce	이제 안 맞음	실험 16.1	기계 수. 뿌리의 바이트가 p 에 선형이고 링이 하한을 달성

그대로 유효가 일곱, 조정해서 유효가 일곱, 이제 안 맞음이 둘이다. 이제 안 맞음 둘은 전제가 깨진 자리이고, 조정 일곱은 정의는 남고 구현이 바뀐 자리다. 그대로 일곱은 정의도 구현도 그대로이며 원소와 규모만 바뀌었다.

부록 A

전통 구조 성적표

17장 끝의 표를 인쇄해 붙일 수 있게 다시 싣는다. 판정은 셋 중 하나이고, 판정마다 근거 실험과 뒤집히는 축이 있다. 자기 데이터에서 그 실험을 다시 돌리면 판정이 다시 나온다.

구조	판정	근거	뒤집히는 축과 한 줄 근거
배열	그대로 유효	실험 1.1	원소가 참조면 크기 여덟 배, 시간 열 배. 연속성이 가장 싼 이동량
동적 배열	조정해서 유효	실험 2.3	배열의 수. 하나면 두 배 늘리기, 수백 개면 블록 테이블
희소 행렬	조정해서 유효	실험 3.1	희소도의 문턱이 50 에서 2 퍼센트로. 규칙 있는 희소성이 그 사이
연결 리스트	조정해서 유효	실험 4.1, 4.3	노드가 흩어지면 의존 읽기 105 ns. 아레나 위의 인덱스 링크만 남는다
스택, 큐, 덱	그대로 유효	실험 5.1, 5.3	배열 위의 스택과 링 버퍼가 지금도 가장 싸다. 원소가 미분과 요청으로 바뀌었다
해시 테이블	조정해서 유효	실험 6.1, 6.4	배치 크기. 크면 정렬이 표를 이기고, 라인 수가 순위를 정한다
블룸 필터와 스케치	그대로 유효	실험 7.1, 7.3	오답률과 S 곡선이 원 논문 그대로. 규모만 커졌다
이진 탐색 트리	조정해서 유효	실험 8.2, 8.4	높이의 단위가 비교에서 라인으로. B-트리와 정렬 배열이 남는다
트라이	그대로 유효	실험 9.3, 9.4	접두사 공유의 절약이 명제 그대로. 노드에 적히는 것만 바뀌었다
이진 힙	조정해서 유효	실험 10.1, 10.2	상위 k 를 한 번에 고르면 선택이 여덟 배. 스트림에서는 힙
인접 리스트	조정해서 유효	실험 11.1	정적이면 CSR. 메모리 5 분의 1, 훑기 이백 배
kd-트리	이제 안 맞음	실험 12.1, 17.3	차원. 2 에서 이기고 8 에서 열 배 진다. 근접 그래프가 대신
벡터 양자화	그대로 유효	실험 13.2, 13.3	바이트 두 배에 오차 반. 목적이 압축에서 거리 어림으로
접미사 배열	그대로 유효	실험 14.1, 14.2	정수 n 개로 모든 부분 문자열의 개수. 만드는 값이 남은 문제
LRU 캐시	조정해서 유효	실험 15.3	참조 패턴. 시간 지역성이 아니라 질량 지역성. 헤비 히터가 대신
별 all-reduce	이제 안 맞음	실험 16.1	기계 수. 뿌리의 바이트가 p 에 선형이고 링이 하한을 달성

그대로 유효가 일곱, 조정해서 유효가 일곱, 이제 안 맞음이 둘이다. 0장 끝의 전통 자료 구조 지도가 이 표의 앞면이고, 거기서 각 구조가 어느 절의 디딤돌인지를 적었다.

부록 B

연습문제 해답

기초 문제는 풀이를 전부 적는다. 응용 문제는 힌트만 적고, 답 코드는 `samples/exercises/` 에 있다. 탐구 문제는 답이 하나가 아니므로 적지 않는다.

0장

0.1 상태 S 는 원소의 유한 열이다. 연산은 enqueue 와 dequeue 둘이다. 불변식은 먼저 넣은 원소가 먼저 나온다는 것이다. 균일 비용 모델에서 두 연산 모두 값이 상수 1 이다. 배열 앞에서 빼면 n 이 되는데, 그것은 구현의 값이지 정의의 값이 아니다.

0.2 순차 이동량은 $\lceil 10^6 \cdot 8/64 \rceil + 1 = 125{,}001$ 라인이다. 전체 8MB 가 캐시 8MB 와 같으므로 $Z/(ns) = 1$ 이고 무작위 하한은 0 이 된다. 데이터가 캐시에 정확히 들어가는 경계이므로 정리 0.1 의 조건 $ns > Z$ 가 성립하지 않는다. 캐시를 4MB 로 잡으면 하한은 $10^6(1 - 0.5) = 500{,}000$ 라인이고 비는 4 배다.

0.3 능선점은 $200/20 = 10$ FLOP/byte 다. 밀도 0.25 에서는 $\min(200, 20 \times 0.25) = 5$ GFLOP/s, 밀도 2 에서는 40 GFLOP/s, 밀도 50 에서는 200 GFLOP/s 다.

0.4 단위 벡터는 $a = (1,2)/\sqrt{5}$, $b = (1,0)$, $c = (-1,-1)/\sqrt{2}$, $d = (0,1)$, $e = (1,1)/\sqrt{2}$ 다. $q = (1,0)$ 과의 내적은 a 가 $1/\sqrt{5} \approx 0.447$, b 가 1, c 가 -0.707, d 가 0, e 가 0.707 이다. 큰 둘은 b 와 e 다. 유클리드 거리로도 b 가 0, e 가 $\sqrt{2 - \sqrt{2}} \approx 0.765$ 로 가장 가까운 둘이다. 정규화하면 답이 예제 0.3 과 달라지는데, 질의도 달라졌기 때문이다.

0.5 힌트. 원소가 8 바이트면 같은 원소 수에서 바이트가 두 배이므로 캐시를 넘는 원소 수가 절반이 된다. 정리 0.1 에서 s 가 두 배가 되면 $Z/(ns)$ 가 절반이 된다.

0.6 힌트. `blocks_touched` 는 라인이 하나만 있는 캐시를 가정하므로 무작위 쪽이 정리의 하한보다 크게 나온다. 시간 비율보다 라인 비율이 크다. 그 차이가 프로세서가 라인 여러 개를 동시에 가져오는 효과다.

0.7 힌트. 원소별 곱은 원소마다 곱 하나, 읽기 둘 쓰기 하나다. $W = n$, $M = 12n$ 바이트, 밀도 $1/12 \approx 0.083$ 이다. 내적보다 낮다.

0.8 힌트. 질의 64 개를 행렬로 묶으면 X 를 한 번 읽어 64 번 쓰므로 밀도가 64 배 가까이 오른다. 질의당 지연은 대역폭 병목이 풀리는 만큼 줄고, 능선점을 넘으면 더 줄지 않는다.

1장

1.1 행 우선이면 한 행이 $4 \times 8 = 32$ 바이트이므로 스트라이드는 $(32, 8)$ 이고, $(2, 1)$ 의 위치는 $2 \times 32 + 1 \times 8 = 72$ 바이트다. 열 우선이면 한 열이 $3 \times 8 = 24$ 바이트이므로 스트라이드는 $(8, 24)$ 이고, $(2, 1)$ 은 $16 + 24 = 40$ 바이트다.

1.2 포인터 $8 \times 10^6 = 8$MB, 객체 $24 \times 10^6 = 24$MB, 합 32MB 다. float32 값 배열은 4MB 이므로 여덟 배다. 실험 1.1 의 첫 행과 셋째 행이다.

1.3 $s = 8, k = 4$ 이면 $ks = 32 < 64$ 이므로 라인 하나에 접근 둘, 이동량은 $10^6 \times 32/64 = 500{,}000$ 라인이다. $k = 8$ 이면 $ks = 64$ 로 접근마다 라인 하나, 이동량은 10^6 라인이다.

1.4 $I(B) = 2B/(4(1 + B/262144 + B/128))$ 이다. $B = 1$ 에서 약 0.50, $B = 16$ 에서 약 7.1, $B = 256$ 에서 약 42.7 이다. 능선점 8 을 넘는 최소 2 의 거듭제곱 B 는 32 다. $B = 16$ 은 7.1 로 조금 못 미친다.

1.5 힌트. 행 우선 $(2, 3, 4)$ 의 스트라이드는 $(48, 16, 4)$ 다. 첫 축과 셋째 축을 바꾸면 모양 $(4, 3, 2)$, 스트라이드 $(4, 16, 48)$ 이다. 버퍼는 그대로다.

1.6 힌트. 예측은 간격 1 에서 1/16, 간격 16 에서 1, 그 뒤로 1 에 머문다. 실측은 16 까지 예측과 같은 비로 오르고 그 뒤로 계속 오른다. 갈리는 자리가 16 이다.

1.7 힌트. 타일 하나는 입력과 출력 둘이므로 $2b^2 \times 4$ 바이트다. 256KB 에 드는 가장 큰 2 의 거듭제곱은 128, 8MB 에는 1024 다. 실험에서 1024 가 조금 느린 것은 캐시를 꽉 채워 다른 것이 밀려나기 때문이다.

1.8 힌트. 명제 1.5 에서 $I(B) \approx B/2$ 이므로 능선점 8 을 넘는 B 는 16 이상이고, 2 의 거듭제곱으로는 32 다. 실험 1.5 에서 처리량은 64 에서 정점이고 256 에서 내려왔다.

2장

2.1 용량이 $8, 16, 32, 64, 128$ 로 네 번 자라고, 옮긴 원소는 $8 + 16 + 32 + 64 = 120$ 개다. 상한 $2n = 200$ 아래다. 마지막 용량 128 은 $2n$ 을 넘지 않는다.

2.2 셋째까지 나가면 빈 구간이 $[30, 100)$ 하나로 합쳐져 70 이므로 60 을 받는다. 합치는 규칙이 없으면 빈 구간이 $[30, 60)$, $[60, 90)$, $[90, 100)$ 셋으로 남아 총합 70 인데 60 을 못 준다. 외부 단편화가 $1 - 30/70 \approx 0.57$ 이다.

2.3 토큰당 $2 \times 32 \times 32 \times 128 \times 2 = 524{,}288$ 바이트, 512KB 다. 24GB 는 24×2^{30} 바이트이므로 49,152 토큰이 산다. 길이 2,048 인 요청 24 개다.

2.4 블록 16 이면 논리 블록 $\lceil 1000/16 \rceil = 63$ 개, 자투리 $63 \times 16 - 1000 = 8$, 표 항목 63 개다. 블록 64 이면 16 개, 자투리 24, 항목 16 개다.

2.5 힌트. `total_copied(100000, f, 4)` 를 네 인자에 대해 부르고 $f \cdot 10^5/(f - 1)$ 과 나란히 놓는다. 1.125 에서는 상한이 $9n$ 이고 실측이 그 근처, 4 에서는 상한 $1.33n$ 이다.

2.6 힌트. 빈 구간 목록을 전부 보고 길이가 맞으면서 가장 작은 구간을 고른다. 실험 2.2 의 요청 열에서 첫 맞춤과 실패 수가 크게 다르지 않을 수 있다. 부하가 높으면 어느 규칙이든 빈 총합이 모자란다.

2.7 힌트. 토큰 수는 메모리를 토큰당 바이트로 나눈 정수다. 헤드 40 에서 8 로 줄이면 토큰당 바이트가 5 분의 1 이 되고 토큰 수는 5 배가 된다.

2.8 힌트. 요청마다 `reserved()` 와 표 길이를 더한다. 블록 16 의 자투리는 평균 7.5 근처, 128 은 63.5 근처다. 표 항목 수는 블록이 8 배 크면 8 분의 1 근처다.

3장

3.1 CSR 은 $ptr = (0, 1, 1, 3, 3, 4)$, $cols = (2, 0, 4, 1)$, $vals = (1, 4, 2, 3)$ 이다. CSC 는 열별로 접은 것이니 $ptr = (0, 1, 2, 3, 3, 4)$, $rows = (2, 4, 0, 2)$, $vals = (4, 3, 1, 2)$ 다.

3.2 $s = 2$ 이면 문턱은 $2/(2+4) = 33$ 퍼센트다. 라인 64 를 넣으면 $2/(2+4+64) \approx 2.9$ 퍼센트다. float16 은 인덱스가 상대적으로 더 비싸 문턱이 더 낮다.

3.3 원소 2^{20} 개다. 2:4 는 $2^{20}/2 \times 2 + 2^{20}/4 = 1{,}310{,}720$ 바이트. BSR 블록 16 은 0 이 아닌 블록 2,048 개에 $16 \times 16 \times 2 + 4$ 바이트씩이라 약 1,056,768 바이트에 포인터 260 바이트. CSR 은 $2^{19} \times (2+4) = 3{,}145{,}728$ 바이트다.

3.4 텐서 하나 스케일이면 $\Delta = 2.0/127 \approx 0.0157$, 오차 상한 0.0079 다. 값의 범위 0.05 에 견줘 16 퍼센트다. 블록 64 이면 이상값이 든 블록 하나만 그 Δ 이고, 나머지 63 개 블록은 $\Delta \approx 0.05/127 = 0.00039$, 상한 0.0002 다. 40 배 차이다.

3.5 힌트. 행 i 의 결과는 a 의 행 i 원소 (c, v) 마다 b 의 행 c 를 v 배 해 더한 것이다. 밀도 ρ 인 둘의 곱은 밀도가 대략 $1 - (1-\rho^2)^k$ 로 커진다.

3.6 힌트. 인덱스 바이트는 블록 수에 비례하므로 블록 크기의 제곱에 반비례한다. x 읽기의 라인은 블록마다 $\lceil 4b/64 \rceil$ 이므로 $b \geq 16$ 이면 블록당 $b/16$ 라인이다.

3.7 힌트. 스케일을 8 비트 코드와 전체 최대 하나로 적으면 원소당 $4/64 = 0.0625$ 바이트가 $1/64 \approx 0.016$ 으로 준다. 오차는 스케일의 반올림만큼 늘고, 대개 두 배 안이다.

3.8 힌트. 전문가마다 배정된 토큰 수를 세고 상한을 넘는 만큼을 더한다. 균등 라우팅에서 용량 인자 1.25 면 버려지는 비율이 몇 퍼센트이고, 쏠린 라우팅에서는 그 몇 배가 된다.

4장

4.1 연결 리스트 순회는 $10^7 \times 80$ 나노초 = 0.8 초다. 인덱스 배열 무작위 접근은 그것의 P 분의 1, 0.08 초다. 같은 라인 수인데 열 배 차이다.

4.2 할당 셋이면 0, 1, 2 를 주고 $h = 3$. 칸 1 을 반납하면 $next[1] = 3$, $h = 1$. 칸 0 을 반납하면 $next[0] = 1$, $h = 0$. 할당 둘이면 0 을 주고 $h = 1$, 다음에 1 을 주고 $h = 3$. 최종 $h = 3$, $next = (1, 3, 3, -1)$ 이고 칸 0, 1, 2 가 쓰이는 중이다.

4.3 $16 \times 1.25^k \leq 2048$ 이므로 $k \leq \log 128 / \log 1.25 \approx 21.7$, 등급 22 개다. 평균 내부 단편화는 $(1.25 - 1)/2 = 12.5$ 퍼센트 근처다.

4.4 끼운 뒤 $next = (3, 4, -1, 1, 2)$, 새 칸이 4 이고 순회 순서는 $(0, 3, 1, 4, 2)$, 값 순서는 $(1, 9, 2, 7, 3)$ 이다. 압축하면 값 배열이 $(1, 9, 2, 7, 3)$ 로 칸 0..4 에 놓인다.

4.5 힌트. 고리 수를 두 배씩 늘리면 걸음당 시간이 줄다가 어느 수부터 멈춘다. 그 수가 프로세서가 동시에 기다리는 읽기의 수에 가깝다. 이 책의 기계에서는 수십에서 백 근처다.

4.6 힌트. 16 부터 2048 까지 $\sqrt{2}$ 비율이면 등급이 열다섯 개다. 평균 낭비는 $(\sqrt{2}-1)/2 \approx 21$ 퍼센트 근처이고, 2 의 거듭제곱의 30 퍼센트보다 작다.

4.7 힌트. 삭제는 앞 칸의 링크를 다음의 다음으로 고치고 빠진 칸을 자유 리스트에 붙인다. 무작위 삭제 뒤의 순회는 빠진 칸을 건너뛰므로 순차가 아니고, 압축이 다시 붙인다.

4.8 힌트. 순회 순서 배열 O 에서 $O[k]$ 가 새 번호 k 로 가므로 재배치 표는 $remap[O[k]] = k$ 다. 밖에서 든 번호 i 는 $remap[i]$ 로 고친다.

5장

5.1 재귀는 최대 $1{,}000 \times 400 = 400$KB 를 쓰고 그 뒤 실패한다. 명시적 스택은 $1{,}000 \times 16 = 16$KB 이고, 깊이 천만이면 160MB 다. 메모리가 허락하는 한 간다.

5.2 $e_1 = (b, (a, a), (3, 3))$, $e_2 = (c, (b, a), (a, b)) = (c, (b, a), (3, 9))$ 다. 역방향은 $\bar{c} = 1$ 에서 e_2 로 $\bar{b} = 3$, $\bar{a} = 9$, 그다음 e_1 로 $\bar{a} = 9 + 3 \cdot 3 + 3 \cdot 3 = 27$ 이다. $3a^2 = 27$ 과 같다.

5.3 1 부터 4 까지 넣으면 $B = (1, 2, 3, 4)$, $h = 0$, $n = 4$. 5 를 넣으면 $B[0] \leftarrow 5$, $h = 1$. 6 을 넣으면 $B[1] \leftarrow 6$, $h = 2$. 논리 순서는 $3, 4, 5, 6$ 이다. pop 둘이면 3 과 4 가 나가고 $h = 0$, $n = 2$, 논리 순서는 $5, 6$ 이다. 배열은 $(5, 6, 3, 4)$ 그대로다.

5.4 걸음 256 번에 자리 걸음 1,024 개 중 $8 + 8 + 64 + 256 = 336$ 개가 쓰여 낭비가 67 퍼센트다. 연속 배칭은 8 걸음 뒤 두 자리가 비어 다음 요청이 들어오고, 64 걸음 뒤 한 자리, 256 걸음 뒤 마지막이 빈다. 각 요청은 자기 길이만큼만 자리를 쓴다.

5.5 힌트. 스택은 마지막에 넣은 것을 먼저 꺼내므로 자식을 거꾸로 넣어야 첫 자식이 먼저 나온다. `stack.extend(reversed(children[v]))` 다.

5.6 힌트. $y = Wx$ 의 국소 미분은 W 자체이고, 뒤로 흘릴 때는 $\bar{x} \leftarrow \bar{x} + W^{\mathsf{T}}\bar{y}$ 다. 항목 하나가 행렬 하나를 들므로 테이프 항목 수가 행 수에서 하나로 준다. 교차점이 왼쪽으로 크게 옮겨 간다.

5.7 힌트. 분산은 $S_2/n - (S_1/n)^2$ 이므로 합과 제곱합 둘을 링 버퍼 옆에 들면 된다. 최대값은 나간 값이 최대였을 때 나머지에서 다시 찾아야 하므로 뺄 수 없는 통계다. 10장의 힙이 그 일을 한다.

5.8 힌트. 도착한 요청 중 길이가 짧은 것을 먼저 활성 집합에 넣는다. 평균 지연이 줄고, 긴 요청의 p95 는 늘 수 있다. 짧은 것이 계속 새치기하면 긴 것이 굶는다.

6장

6.1 $\alpha = 0.75$ 다. 체이닝은 $1 + 0.375 = 1.375$ 번, 선형 탐사는 $\frac{1}{2}(1 + 4) = 2.5$ 번이다. 키 8 바이트면 라인 하나에 자리 여덟 개이므로 탐사 2.5 는 라인 하나에 든다.

6.2 키까지 저장하면 $32{,}000 \times (8 + 8 + 100) = 3.7$MB 다. 완전 해시로 번호만 두면 $4 \times 32{,}000 = 128$KB 다. 역조회 배열은 어느 쪽이든 $32{,}000 \times 16 = 512$KB 다.

6.3 $V = 5 \times 10^5$ 에서 $m = 2^{20}$ 이면 $1 - (1 - 2^{-20})^{V-1} \approx 1 - e^{-0.477} \approx 0.38$ 이고, $m = 2^{22}$ 이면 $1 - e^{-0.119} \approx 0.11$ 이다. 자리를 네 배로 늘리면 충돌이 3.4 배 준다. $V \ll m$ 근처에서는 거의 정확히 네 배가 된다.

6.4 쓰이는 키의 기대 수는 $4096(1-(1-1/4096)^{3000}) \approx 4096(1-e^{-0.732}) \approx 2{,}131$ 이다. 미뤄지는 키는 약 869 개, 29 퍼센트다.

6.5 힌트. 지운 자리를 비우면 그 뒤에 밀려 있던 키의 탐사 사슬이 끊긴다. 묘비 표시를 두면 탐색은 지나가고 삽입은 쓸 수 있다. 묘비가 쌓이면 표를 다시 짜야 한다.

6.6 힌트. 첫 글자로 나누면 배열이 스물여섯 개이고 각각이 26분의 1 이라 이진 탐색이 다섯 단계 준다. 문자열 비교 다섯 번이 줄고 그만큼 라인이 준다.

6.7 힌트. 자리 해시와 별개로 부호 해시 $\xi(w) \in \{-1, +1\}$ 을 두고 벡터에 곱해 더한다. 겹친 두 토큰의 간섭이 확률 절반으로 부호가 반대라 기대값이 0 이 된다. 오차의 평균은 0 근처가 되고 분산만 남는다.

6.8 힌트. 미룬 키를 둘째 해시 자리로 보내면 뭉침이 이어지지 않아 걸음 수가 크게 준다. 적재율 0.5 에서 이백 걸음이 열 걸음 안팎이 된다. 그것이 쿠쿠의 원리다.

7장

7.1 $p = 10^{-3}$ 이면 키당 비트는 $1.44 \times \log_2 1000 = 14.4$ 이고 해시 수는 $14.4 \times 0.693 \approx 10$ 이다. 천만 개면 1.44×10^8 비트, 18 MB 다. 개방 주소 표는 $10^7 \times 8/0.7 = 114$ MB 라 여섯 배다.

7.2 두 배 안이라는 것은 오차가 실제 빈도 이하라는 뜻이고, 실제 빈도가 $10^{-4}N$ 이상이므로 $\varepsilon = 10^{-4}$ 로 두면 된다. $w = \lceil e \times 10^4 \rceil = 27{,}183$, $d = \lceil \ln 1000 \rceil = 7$ 이다. 8 바이트 칸이면 $27{,}183 \times 7 \times 8 = 1.5$ MB 다.

7.3 $1-(1-J^5)^{20}$ 이다. $J = 0.5$ 에서 $1-(1-0.031)^{20} = 0.47$, 0.7 에서 $1-(1-0.168)^{20} = 0.975$, 0.9 에서 $1-(1-0.59)^{20} \approx 1$ 이다. 문턱은 $(1/20)^{1/5} = 0.55$ 다.

7.4 서명은 $8 \times 128 \times 10^9 = 1$ TB 이고 블룸 필터는 10×10^9 비트, 1.25 GB 다. 조각 집합을 두면 4 TB 가 더 들어 서명의 네 배다. 그래서 확인을 서명 비교로 한다.

7.5 힌트. 자리 i 를 바이트 $i \gg 3$ 과 비트 $i \,\&\, 7$ 로 나눠 `np.bitwise_or.at` 으로 켠다. 해시가 같으므로 오답률이 같다. 메모리는 8 분의 1 이다.

7.6 힌트. h_0 으로 블록을 고르고 나머지를 블록 안 512 자리로 접는다. 키당 16 비트에서 오답률이 만분의 4 에서 만분의 6 안팎으로 오른다. 블록에 든 키 수가 푸아송 분포로 흩어지고 정리의 식이 볼록해서다.

7.7 힌트. 키마다 줄 d 개의 현재 최소를 구하고, 최소와 같은 줄만 1 올린다. 같은 배치 안의 같은 키는 먼저 세어 개수만큼 올린다. 상위 100 의 과대 추정이 절반 아래로 준다.

7.8 힌트. `shingle_of` 대신 서명을 저장하고 `estimate_jaccard` 로 확인한다. 서명 64 개의 오차 0.06 때문에 문턱 근처의 문서가 양쪽으로 몇 개 옮겨 가고, 걸러 낸 수는 967 에서 수십 개 안에서 움직인다.

7.9 정렬만으로는 안 된다. 띠 하나 안의 같은 자리 문서는 정렬로 붙지만, 문서 a 와 b 가 띠 1 에서 만나고 b 와 c 가 띠 2 에서 만나면 a 와 c 는 어느 정렬에서도 안 붙는다. 띠를 가로지르는 이어 붙이기가 필요하고, 그것이 후보 쌍을 간선으로 둔 그래프의 연결 성분 찾기다. 11장의 그래프 표현 위에서 한다.

7.10 자리 하나에 오는 키의 수는 평균 kn/m 의 푸아송 분포다. 키당 10 비트, 해시 7 개면 평균 0.7 이고, 16 이상일 확률은 $\sum_{j \geq 16} e^{-0.7} 0.7^j / j! \approx 10^{-16}$ 이다. 자리 천만 개에서 넘치는 자리는 기대값이 10^{-9} 개, 사실상 없다.

8장

8.1 $2\ln 10^6 = 27.6$ 이다. 완전 균형은 $\lceil \log_2(10^6+1) \rceil = 20$ 이다. 노드 하나가 라인 하나이면 성공 탐색 하나에 라인 27.6 개이고, 해시 테이블의 하나에서 둘에 비해 스무 배다.

8.2 라인 64 바이트면 $B = 8$, 높이 $\lceil \log_8 10^9 \rceil = 10$ 이고 내부 노드는 $8 \times 10^9/7 = 1.14$ GB 다. 페이지 4 KB 면 $B = 512$, 높이 $\lceil \log_{512} 10^9 \rceil = 4$ 이고 내부 노드는 $8 \times 10^9/511 = 15.7$ MB 다.

8.3 메모리는 $24 \times 2{,}000 = 48$ KB 다. 비교는 조각 찾기 $\lceil \log_2 2000 \rceil = 11$ 번과 창 안 $\lceil \log_2 129 \rceil = 8$ 번으로 19 번이다. 차수 64 의 B-트리 내부 노드는 $8 \times 10^6/63 = 127$ KB 이고 비교는 층 넷에 여섯 번씩 24 번이다.

8.4 $\log_2(2^{30}/2^{16}) = 14$ 개다. 정렬 안 하면 30 개이고, 위쪽 몇 층만 캐시에 남으므로 실제로는 25 개 안팎이다.

8.5 힌트. 지울 노드의 두 자식 중 우선순위가 큰 쪽을 회전으로 올리면 지울 노드가 한 층 내려간다. 자식이 하나 이하가 될 때까지 되풀이하고 그 자식으로 대신한다. 중위 순회가 정렬되어 있는지와 부모 우선순위가 자식보다 큰지를 시험으로 본다.

8.6 힌트. 순차 비교는 분기 예측이 잘 맞고 라인 안에서 끝나므로 차수 8 에서는 이진 탐색과 같거나 빠르다. 차수 64 근처부터 비교 횟수의 차이 (64 대 6) 가 드러난다. 파이썬에서는 그 경계가 더 낮다.

8.7 힌트. 조각 시작 키 배열 위에 같은 클래스를 한 번 더 세운다. 탐색은 위 층에서 조각 번호를 예측하고 창 안에서 찾은 뒤 아래 층으로 간다. 조각 845 개면 조각 찾기가 10 번에서 창 안 7 번으로 준다.

8.8 힌트. 앞 질의의 자리 p 에서 시작해 $p+1, p+2, p+4, \ldots$ 로 보폭을 두 배씩 늘려 넘어서는 자리를 찾고 그 사이를 이진 탐색한다. 이웃한 질의의 간격이 n/q 이면 비교가 $2\log_2(n/q)$ 번이다. 실험의 세 배에 비교 횟수의 이점이 더해진다.

8.9 오차 불변식 $|\hat{p} - i| \le \varepsilon$ 을 조각 하나 안에서만 지키게 완화하고, 조각마다 삽입 여유를 둔다. 삽입이 조각의 예측을 ε 밖으로 밀면 그 조각만 다시 만든다. 조각을 다시 만들어도 이웃 조각의 시작 키가 안 바뀌면 위 층은 그대로다. 그 조건은 삽입 키가 조각의 양 끝 안에 있다는 것이다.

8.10 앞 8 바이트가 같은 문자열들이 같은 실수로 바뀌어 한 점에 여러 위치가 쌓인다. 직선 하나가 같은 x 에서 여러 i 를 내야 하므로 오차가 그 개수만큼 커진다. 공통 접두사가 긴 키 집합일수록 나쁘고, 9장의 트리가 그 접두사를 먼저 나눈다.

9장

9.1 노드 수 상한은 $50{,}000 \times 6 + 1 = 300{,}001$ 이다. 자식 배열 256 칸에 8 바이트면 노드당 2 KB 로 600 MB 이고, 간선당 24 바이트의 해시면 간선이 노드 수와 같으므로 7.2 MB 다. 접두사를 나누면 둘 다 그보다 작다.

9.2 전부 세면 $kN = 3 \times 10^4 \times 10^7 = 3 \times 10^{11}$ 이다. 색인을 두면 병합마다 단어 200 개의 길이 (평균 10 토큰이면 2,000) 를 다시 세므로 $3 \times 10^4 \times 2{,}000 = 6 \times 10^7$ 이고, 처음 한 번 세는 N 을 더해 7×10^7 이다. 사천 배다.

9.3 캐시 없이는 사용자마다 $\sum_{t=1}^{10}(2000 + 200t) = 20{,}000 + 11{,}000 = 31{,}000$ 토큰, 천 명이면 3.1×10^7 이다. 서로 다른 접두사의 총 길이는 $2{,}000 + 1{,}000 \times 2{,}000 = 2{,}002{,}000$ 이다. 15 분의 1 이다.

9.4 $2{,}000 \times 128{,}000 = 2.56 \times 10^8$ 비트, 32 MB 다. 디코딩 한 단계에 읽는 것은 한 줄 128,000 비트, 16 KB 다.

9.5 힌트. 뿌리의 자식만 256 칸 리스트로 두면 첫 바이트의 조회가 해시에서 첨자로 바뀐다. 위치마다 첫 걸음이 해시 하나만큼 싸지고, 실험 9.1 의 시간이 10 퍼센트 안팎 준다. 둘째 층부터는 자식이 적어 배열이 낭비다.

9.6 힌트. 이웃 쌍의 (순위, 위치) 를 힙에 넣고, 최소를 꺼내 아직 유효한지 확인한 뒤 합친다. 합치면 양옆에 새 쌍이 둘 생기고 그것만 넣는다. 길이 L 단어에서 $O(L^2)$ 이 $O(L \log L)$ 이 된다.

9.7 힌트. `match_prefix` 가 지나온 노드의 `refs` 를 올리고, 요청이 끝나면 내린다. 축출은 `refs == 0` 인 잎만 본다. 참조 수 없이 두 요청을 겹치면 첫 요청이 쓰는 동안 둘째 요청의 삽입이 그 잎을 떼어 첫 요청의 KV 가 사라진다.

9.8 힌트. 상태 셋이다. 시작, 따옴표 안, 닫힘. 따옴표 안 상태는 따옴표 아닌 모든 바이트를 받으므로 허용 집합이 어휘 대부분이고, 트라이가 지나는 노드가 거의 전부다. 숫자 형식과 반대로 제약이 헐거우면 트라이의 이점이 준다.

9.9 병합 순서 부호화의 불변식은 학습 때 단어가 지난 병합 열을 그대로 지난다는 것이다. 그래서 학습 말뭉치의 단어는 학습 끝의 토큰 열과 같은 열이 되고, 모델이 본 분포와 같다. 최장 일치로 학습한 모델에 병합 순서로 넣으면 단어의 8 퍼센트 안팎이 학습 때 못 본 토큰 열이 되어 그 자리의 예측이 나빠진다.

9.10 둘 다 앞이 없으면 뒤를 못 쓴다는 의존을 참조 수로 지킨다. 합치면 라디스 트리의 간선 열 하나가 블록 여러 개로 나뉘고, 노드는 갈림이고 블록은 할당 단위다. 축출은 잎 노드의 마지막 블록부터 떼고, 블록의 참조 수가 그 블록을 접두사로 쓰는 요청 수다.

10장

10.1 정렬은 $n \log_2 n = 2 \times 10^7$, 힙은 $n \log_2 k = 6.6 \times 10^6$, 선택은 $2n = 2 \times 10^6$ 이다. 힙의 실제 비교는 대부분의 원소가 힙의 최소보다 작아 비교 하나로 끝나므로 n 근처다. 상위 100 개에 들 후보만 가라앉힌다.

10.2 점수 계산은 $200 \times 4 \times 32{,}000 = 2.56 \times 10^7$ 이고 부모 배열은 $2 \times 200 \times 4 = 1{,}600$ 정수다. 열을 저장하면 $200 \times 4 \times 200 = 160{,}000$ 정수로 백 배다.

10.3 양은 $(1.6, 1.2, 0.8, 0.4)$ 다. 자리 3 에 $q = 0.4$, 별칭 0 을 두면 0 의 양이 1.0 이 되고 큰 쪽에 남는다. 자리 2 에 $q = 0.8$, 별칭 1 을 두면 1 의 양이 1.0 이다. 자리 0 과 1 은 $q = 1$ 이다. 자리 0 의 확률은 $\frac{1}{4}(1 + 0.6) = 0.4$, 자리 1 은 $\frac{1}{4}(1 + 0.2) = 0.3$ 이다.

10.4 긴 요청의 키가 0 이 되는 시각은 $3{,}000/100 = 30$ 초 뒤다. 그때 앞에 있는 짧은 요청들의 처리 시간을 더한 것이 상한이고, 짧은 것이 0.1 초씩이니 30 초 남짓이다.

10.5 힌트. 마지막 내부 노드 $\lfloor n/2 \rfloor - 1$ 부터 0 까지 가라앉히면 $O(n)$ 이다. 넣기 n 번은 $n \log n$ 이지만 대부분의 원소가 한두 층만 올라가 실측은 두세 배 차이다.

10.6 힌트. 점수를 log 확률의 합에서 길이로 나눈 값으로 견주면 짧은 열이 불리하지 않다. 무작위 표에서는 모든 열의 길이가 같아 결과가 안 바뀐다. 끝 토큰이 있는 모델에서만 차이가 난다.

10.7 힌트. 남는 수는 단계 사이에 천천히 변하므로 이전 단계의 m 을 k_0 로 두면 대개 한 번에 끝난다. 두 배씩 늘리는 횟수가 0 에서 1 이 되어 top-p 0.9 에서 시간이 절반 아래로 준다.

10.8 키 길이 $-\alpha(t-$ 도착$)$ 은 t 가 모든 항목에 공통이므로 순서는 길이 $+\alpha\times$ 도착 으로 같다. 넣을 때 한 번 정하면 되고 다시 매기지 않는다. 대기의 평균과 최대가 실험과 같다.

10.9 후보 B 개가 같은 접두사에서 갈리므로 라디스 트리에서 갈림 노드 하나와 잎 B 개다. 빔이 갈릴 때 트리에 잎이 생기고, 블록 테이블에서는 갈린 마지막 블록만 복사되고 앞 블록은 참조 수가 는다. 트리의 안쪽 노드가 참조 수가 큰 블록이다.

10.10 추정이 실제 길이와 상관이 있는 만큼만 최적성이 남는다. 정리의 교환 논증은 실제 길이의 대소에 의존하므로 추정이 무작위이면 교환이 합을 줄인다는 보장이 없고, 기대값으로는 도착 순서와 같다. 다만 굶김은 여전히 생긴다. 추정이 작은 요청이 계속 앞서기 때문이다.

11장

11.1 CSR 은 포인터 10^5+1 개와 열 10^6 개로 4.4 MB 다. 인접 리스트는 노드마다 포인터 8 바이트 (0.8 MB) 와 정수 배열 10^6 개 (4 MB) 로 4.8 MB 다. 메모리는 비슷하고 배열이 10^5 개로 흩어진 것이 다르다.

11.2 큐에 넣는 횟수는 정확히 n 이다. 노드마다 차수가 0 이 되는 순간 한 번. 차수 배열을 쓰는 횟수는 간선마다 한 번이라 m 이고, 그 쓰기가 무작위 자리다.

11.3 노드는 x, $t=\tanh x$, $p=xt$, $y=p+x$ 다. 역방향은 y 에서 p 와 x 에 1 씩, p 에서 x 에 t 와 t 에 x, t 에서 x 에 $x(1-t^2)$ 다. x 에 세 번 더해지고 합은 $1+t+x(1-t^2)$ 이며 $x=0.5$ 에서 $t=0.462$, 값은 $1+0.462+0.5\times0.786=1.855$ 다.

11.4 하나씩은 $2\times8\times10^8\times4=6.4$ GB, 융합은 $2\times10^8\times4=0.8$ GB 다. 800 GB/s 에서 8 밀리초와 1 밀리초다.

11.5 힌트. 다시 짜기는 간선마다 $O(m)$ 이라 만 개면 10^{10} 이다. 따로 모아 두면 이웃 조회가 열 배열과 모아 둔 것 둘을 봐야 하고, 합치기는 가끔 $O(m)$ 이다. 백 개마다 합치면 다시 짜기 백 번이고, 이것이 2.1 의 분할 상환이다.

11.6 힌트. 스택을 노드 배열과 다음 이웃 번호 배열로 두고 위 하나의 첨자만 움직인다. 튜플을 만들고 풀고 바꾸는 값이 빠져 두세 배 준다. 그래도 Kahn 보다 느린 것은 노드마다 문장이 더 많기 때문이다.

11.7 힌트. 나눗셈 a/b 의 편미분은 $1/b$ 와 $-a/b^2$, exp 는 자기 값이다. 노드 하나를 세 노드의 입력으로 두고 역방향 뒤 그 노드의 기울기가 세 항의 합인지 유한 차분으로 본다.

11.8 힌트. 축소 연산의 집합을 따로 받고, 축소 노드는 사슬에 더하되 그 뒤로는 사슬을 닫는다. 축소 뒤에 원소별이 오면 새 사슬이 시작되어야 하고, 시험은 그 경계에서 사슬이 둘로 나뉘는지 본다.

11.9 값을 저장하면 메모리 $O(n)$ 에 연산은 전진과 역방향 각 한 번이다. 저장하지 않으면 노드마다 다시 전진해야 하므로 연산 $O(n^2)$ 이다. 구간 $\sqrt{n}$ 개의 경계만 저장하면 메모리 $O(\sqrt{n})$ 이고, 구간마다 다시 전진하므로 연산이 두 배다. 15장의 재계산이 이것이다.

11.10 노드 a 에서 큰 텐서 $b_1, \ldots, b_k$ 가 나오고 각각이 작은 텐서로 줄어 마지막에 합쳐지는 그래프에서, 큰 것을 다 만든 뒤 줄이면 최대 메모리가 k 개의 큰 텐서이고, 하나 만들고 바로 줄이면 하나다. 두 순서 모두 위상 순서다.

12장

12.1 거리 제곱은 d 개 항의 합이라 평균이 d 에, 표준편차가 $\sqrt{d}$ 에 비례한다. 상대 편차 $1/\sqrt{d}$ 는 $d = 4$ 에서 0.5, $d = 64$ 에서 0.125, $d = 1024$ 에서 0.03 이다. 1024차원에서는 가장 먼 점과 가장 가까운 점이 몇 퍼센트 차이다.

12.2 $p = 1/8$ 이면 층은 $\log_8 10^6 = 6.6$ 개, 간선은 $1/(1 - 1/8) = 1.14$ 개, 걸음 상한은 $8 \times 6.6 = 53$ 이다. $p = 1/2$ 는 층 20, 간선 2, 걸음 40 이다. 메모리는 절반 남짓이고 걸음은 3 분의 1 이 더 든다.

12.3 이웃 $M = 32$ 이고 탐색이 노드 t 개를 넓히면 거리 계산이 $32t$ 이하다. 넓힌 노드가 100 개면 3,200 번이고, 전부 비교 10^6 의 0.3 퍼센트다. 층 수는 $\log_{1/p} n$ 이고 $p = 1/4$ 이면 10 층이다.

12.4 벡터 768 바이트 (float32 로 3 KB 인데 1 바이트 양자화면 768) 와 이웃 $64 \times 4 = 256$ 바이트로 1 KB 다. 4 KB 페이지에 노드 넷이 들고, 십억 개면 250 GB 의 페이지다. 노드마다 페이지 하나로 두면 탐색이 넓힌 노드 수만큼 페이지를 읽는다.

12.5 힌트. 잎을 키우면 노드 수와 트리 걸음이 줄고 잎 안의 비교가 는다. 4차원에서는 잎 16 안팎이 최소이고, 그 위로는 잎 안 비교가 지배한다. 어느 차원에서든 잎을 키우는 것으로 저주를 피할 수는 없다.

12.6 힌트. 삽입과 같은 경로로 층마다 왼쪽 노드를 찾고, 지울 노드의 높이만큼 링크를 건너뛰게 한다. 맨 위 층이 비면 층 수를 내린다. 회전이 없으므로 삭제도 링크 몇 개다.

12.7 힌트. 가까운 것만 남기면 같은 군집 안의 이웃만 남고 군집 사이의 긴 간선이 잘린다. 다른 군집의 질의에서 탐색이 그 군집으로 못 건너가 재현율이 떨어진다. 휴리스틱은 방향이 다른 이웃을 남겨 그 다리를 지킨다.

12.8 힌트. 페이지 4 KB 에서 768차원 float32 벡터는 3 KB 라 이웃이 256 개까지이고, 1 바이트로 양자화하면 832 개다. 이웃 수가 페이지에서 정해진다. 벡터가 페이지를 넘으면 한 번의 접근이 안 된다.

12.9 층 0 의 이웃 M 과 탐색 폭 ef 가 재현율을 올리고 시간을 늘린다. 재현율은 ef 에 대해 포화하고 시간은 선형이라, 목표 재현율에서 가장 작은 ef 를 고른다. M 은 메모리를 정하고 만들기 시간을 정한다. 셋을 한 번에 최적으로 고르는 방법은 없고 재서 고른다. 13장의 마지막 절이 그 잣대다.

12.10 메모리 안의 거친 벡터로 거리를 어림하면 페이지 읽기가 이웃 목록에만 들고, 정확한 거리는 마지막 후보에서만 페이지를 읽는다. 거친 거리의 오차가 크면 후보에서 진짜 답이 빠지고, 그것이 13장의 양자화 오차와 재현율의 관계다.

13장

13.1 $k = \sqrt{\text{nprobe} \cdot n} = \sqrt{8 \times 10^8} \approx 28{,}000$ 이다. 거리 계산은 중심 28,000 번과 점 $8 \times 10^8 / 28{,}000 \approx 28{,}600$ 번으로 57,000 번이고, 전부 비교 10^8 의 1,800 분의 1 이다.

13.2 코드북은 256 × 128 = 32,768 개의 실수로 128 KB 다. 코드는 16 바이트이고 10^8 개면 1.6 GB 다. 원 벡터는 512 바이트씩 51 GB 라 32 분의 1 이다.

13.3 중심 거리 4,096 × 128, 거리표 16 × 256 × 128 = 52 만, 코드 훑기 16 × (n/4096) × 16 이고 $n = 10^8$ 이면 625 만 번의 표 읽기, 다시 재기 200 × 512 = 100 KB 다. 코드 훑기가 가장 크고, 군집을 키우면 준다.

13.4 원 벡터 5 억 개는 128 차원이어도 256 GB 라 64 GB 에 안 든다. 셋째 단계에서 곱 양자화가 남는다. 갱신이 없으니 역파일 위의 곱 양자화이고, 부분 32 개면 16 GB 다. 다시 재기용 원 벡터는 디스크에 두거나 거친 코드로 대신한다. 남은 것은 nprobe 와 다시 재기 수를 재현율 95 에서 고르는 것이다.

13.5 힌트. 가장 가까운 중심까지의 거리 제곱을 확률로 두고 10.3 의 누적합으로 뽑는다. 같은 반복 수에서 목적 함수가 10 에서 30 퍼센트 작다. 군집이 뚜렷한 데이터에서 차이가 크다.

13.6 힌트. 무작위 정규 행렬을 QR 분해해 직교 행렬을 얻고 학습과 부호화와 질의에 같이 곱한다. 좌표 사이의 상관이 부분을 가로지르면 오차가 줄고, 이미 독립이면 거의 그대로다.

13.7 힌트. 질의에서 중심 nprobe 개를 뺀 행렬 (nprobe, d) 를 부분마다 코드북과 행렬 곱하면 거리표 nprobe 개가 한 번에 나온다. 파이썬 루프 nprobe 번이 사라져 시간이 절반 아래로 준다.

13.8 힌트. 목표가 낮으면 역파일이 nprobe 1 로 이기고, 99 에서는 nprobe 가 커져 지연이 늘며 그래프와 순위가 뒤집힐 수 있다. 곱 양자화는 다시 재기 수를 함께 늘려야 99 에 닿는다.

13.9 대칭 거리는 중심 쌍의 거리표 256 × 256 × m 을 미리 만들어 질의마다 표를 안 만든다. 잃는 것은 질의의 양자화 오차이고, 원 논문은 비대칭이 더 정확하다고 보고했다. 질의가 많고 정확도가 덜 중요하면 대칭이다.

13.10 메모리는 곱 양자화가 코드 mn 이고 디스크 그래프가 거친 코드 mn 에 페이지는 디스크다. 접근은 곱 양자화가 순차 코드 훑기이고 그래프가 페이지 접근 수십 번 (명제 12.13). 만들기는 k-평균 대 삽입 n 번 (명제 12.8). 갱신은 그래프가 삽입이 되고 역파일은 다시 만든다.

14 장

14.1 배열은 10^9 개의 정수로 8 바이트면 8 GB, 4 바이트면 4 GB 다. 정렬 횟수는 $\lceil \log_2 10^9 \rceil = 30$ 이하이고, 정렬 하나가 $n \log_2 n = 3 \times 10^{10}$ 비교다.

14.2 비교는 이진 탐색 둘에 각각 $\log_2 10^9 = 30$ 번으로 60 번이고, 비교마다 20 바이트를 읽어 1,200 바이트다. 배열이 디스크에 있으면 비교마다 배열의 자리 하나와 텍스트의 자리 하나로 페이지 접근이 최대 120 번이다.

14.3 상한은 $\min(10^{12}, 50000^4)$ 이고 앞이 작아 10^{12} 항목이다. 항목마다 수십 바이트면 수십 TB 다. 접미사 배열은 10^{12} 개의 정수로 5 바이트씩 5 TB 이고 n 과 무관하다.

14.4 LCP 가 $(0, 1, 3, 0, 0, 2)$ 이므로 2 이상인 자리는 2 와 5 다.
$R_2 = \{\mathrm{SA}[2], \mathrm{SA}[1], \mathrm{SA}[5], \mathrm{SA}[4]\} = \{1, 3, 2, 4\}$ 다. 각각 2 글자를 덮어 $[1, 6)$ 이 덮이고 비율은 5/6 이다.

14.5 힌트. `np.lexsort((second, rank))` 가 같은 순서를 낸다. 정수 곱셈이 없어 넘칠 걱정이 없고, 정렬이 두 번이라 조금 느리다.

14.6 힌트. 길이 1 의 구간을 찾고, 문맥을 한 토큰씩 늘리며 지금 구간 안에서만 두 경계를 찾는다. 구간이 최소 개수 아래로 떨어지면 직전 구간이 답이다. 이진 탐색의 범위가 N 에서 구간 길이로 줄어 예측당 시간이 몇 배 준다.

14.7 힌트. LCP 가 문턱 이상으로 이어지는 자리들을 한 구간으로 묶고, 구간 안의 SA 값 중 최소를 첫 출현으로 남긴다. 시험은 남긴 위치의 집합이 각 구간마다 정확히 하나이고 최소인지 본다.

14.8 힌트. 구분자를 어느 텍스트에도 없는 값으로 두고 이어 붙인다. LCP 가 큰 이웃 쌍 중 하나는 A 쪽, 하나는 B 쪽 위치인 것만 고른다. 같은 쪽 쌍은 같은 텍스트 안의 반복이라 뺀다.

14.9 이진 탐색의 단계마다 배열의 자리 하나를 읽으므로 명제 14.2 의 $\log N$ 번이 접근이다. 위쪽 k 단계의 자리는 2^k 개뿐이라 메모리에 두면 접근이 $\log N - k$ 번이다. $k = 20$ 이면 백만 자리, 8 MB 로 접근이 30 에서 10 으로 준다.

14.10 문서 단위는 같은 문단이 서로 다른 긴 문서에 든 것을 놓치고, 부분 문자열 단위는 단어 몇 개씩 바뀐 거의 같은 문서를 놓친다. 부분 문자열 제거를 먼저 하면 문서가 잘려 자카드가 바뀌므로, 문서 단위를 먼저 하고 부분 문자열을 나중에 하는 것이 맞다.

15장

15.1 점수 행렬은 n^2 개의 실수이고 $n = 32{,}768$ 이면 10^9 개, float16 이면 2 GB 다. 헤드 32 개면 64 GB 다. 타일은 블록 $b \times d$ 만 두므로 $b = 128$, $d = 128$ 에서 32 KB 다.

15.2 명제 15.6 로 하나씩은 $4n^2$ 원소, 타일은 $2nd + 2nd_v$ 원소다. $n = 8{,}192$, $d = 128$ 이면 2.7×10^8 대 4.2×10^6 으로 64 배다. 밀도는 d 에 비례해 오르므로 $d/4$ 배다.

15.3 누적 점수가 작은 토큰이 먼저 나가고, 최근 r 개는 보호된다. 예산 B 에서 남는 것은 헤비 히터 $B - r$ 개와 최근 r 개다. 오래됐지만 점수가 큰 토큰은 남고, LRU 에서는 나간다. 그 토큰이 뒤에서 자주 참조되면 LRU 의 오차가 크다.

15.4 KV 캐시는 토큰마다 층 × 헤드 × 차원 ×2 이고 $32 \times 32 \times 128 \times 2 \times 2 = 512$ KB 다. 길이 10^5 이면 51 GB 다. 고정 상태는 층 × 채널 × 상태 ×2 로 $32 \times 4096 \times 64 \times 2 = 16$ MB 이고 길이와 무관하다.

15.5 힌트. 마스크가 있으면 타일 안에서 $-\infty$ 를 넣고, 질의 블록보다 뒤의 키 블록은 아예 건너뛴다. 인과 마스크에서 절반의 타일이 사라져 시간이 절반에 가깝다.

15.6 힌트. 블록 크기가 작으면 블록마다의 NumPy 호출 값이 크고, 크면 블록 점수 $m \times b$ 가 캐시를 넘친다. 그 사이에 최소가 있고, 11 장의 융합 실험과 같은 모양이다.

15.7 힌트. 점수의 합 대신 지수 감쇠 합 $s_t = \lambda s_{t-1} + p_t$ 를 두면 옛 참조가 잊힌다. 참조 패턴이 바뀌는 열에서 오차가 줄고, 안 바뀌는 열에서는 같다.

15.8 힌트. 블록 안의 닫힌 식을 재귀 대신 행렬 곱으로 쓴다. a^{i-j} 의 하삼각 행렬 (L, L) 을 상태마다 만들면 $O(L^2 s)$ 이고, 블록이 작을 때 재귀보다 빠르다. 그것이 5 장 누적합의 두 단계다.

15.9 어텐션은 키 n 개를 다 두고 유사도로 읽으므로 상태가 n 에 비례하고 어떤 과거든 정확히 되찾는다. 고정 상태는 과거를 s 개의 수로 요약하므로 되찾기가 근사이고, 요약이 잃는 것은 길이가 늘수록 는다. 둘 사이의 맞바꿈은 정확한 되찾기와 길이 무관한 값이다.

15.10 연관 배열의 읽기는 키 하나에 값 하나이고, 어텐션은 모든 키에 가중치를 둔 값의 합이다. 정확한 일치가 없으므로 해시가 안 되고 전부 비교이며, 그것을 줄이는 것이 12장의 근접 탐색 (가까운 키만) 과 이 장의 축출 (점수 큰 키만) 이다. 둘 다 재현율의 문제다.

16장

16.1 매개변수 7×10^9 개, 2 바이트면 벡터가 14 GB 다. 링에서 기계 하나가 보내는 양은 $2(p-1)/p$ 배라 $p = 8$ 에서 24.5 GB 이고, 400 GB/s 이면 61 밀리초다. 별의 뿌리는 $2(p-1) = 14$ 배로 196 GB, 490 밀리초다. 단계는 링이 14 개, 별이 2 개다.

16.2 블록 2^{20} 개면 층이 21 개다. 다른 블록 $k = 100$ 개를 찾는 데 해시 비교가 $2k \log_2 n = 4{,}000$ 번 안팎이고, 전부 비교는 2^{20} 번이다. k 가 $n/\log n = 5$ 만 개를 넘으면 전부 비교가 낫다.

16.3 열 여덟 개에 레코드 64 바이트, 행 10^8 개면 행 저장은 열 하나를 읽는 데 6.4 GB 를 닿고, 열 저장은 0.8 GB 다. 8 배다. 10 GB/s 이면 640 밀리초와 80 밀리초다.

16.4 $n = 10^9$ 이면 $2^{30} \geq n$ 이라 $2h = 30$, 반쪽이 15 비트다. 정의역 2^{30} 이 n 의 1.07 배라 사이클 걷기가 평균 1.07 회이고, 저장은 키 몇 개뿐이다. Fisher-Yates 는 8 GB 의 순열을 저장한다.

16.5 힌트. 트리는 reduce 와 broadcast 각각 $\log_2 p$ 단계이고, 단계마다 보내는 것이 벡터 전부라 기계 하나의 바이트가 $2\log_2 p$ 벡터다. 링의 $2(p-1)/p$ 보다 크고, 단계 수는 $2\log_2 p$ 로 링의 $2(p-1)$ 보다 작다. 지연이 시간을 정하는 작은 벡터에서 트리가 이긴다.

16.6 힌트. 고정 블록은 한 바이트를 끼우면 그 뒤의 모든 블록 경계가 밀려 전부 새 블록이다. 구르는 해시가 마스크에 걸리는 자리를 경계로 두면 경계가 내용에 붙어 있어, 삽입한 자리의 블록 하나만 바뀐다. 평균 블록 크기는 마스크의 비트 수가 정한다.

16.7 힌트. 정렬된 열은 같은 값이 잇달아 있어 런 길이 부호화가 값 수만큼으로 준다. 백 가지 값이면 항목 백 개다. 합은 값과 길이의 곱의 합이라 풀지 않고 구한다. 열 저장이라야 이 압축이 되고, 행 저장에서는 같은 값이 붙어 있지 않다.

16.8 힌트. 블록이 크면 같은 블록에 연속으로 머무는 비율이 오르고 순차 읽기에 가깝다. 원소가 원래 자리에서 얼마나 멀리 가는지는 블록 순서도 섞으므로 블록 크기와 무관하게 $n/3$ 안팎이다. 잃는 것은 멀리 가느냐가 아니라 이웃이 누구냐다.

16.9 링은 대역폭 최적이지만 단계가 $2(p-1)$ 이라 지연이 p 에 선형이다. 트리는 단계 $2\log p$ 에 바이트 $2\log p$ 벡터다. 벡터가 작고 p 가 크면 지연이 시간을 정해 트리가 이기고, 벡터가 크면 대역폭이 정해 링이 이긴다. 링을 여러 개 겹쳐 조각을 나누면 둘의 사이가 된다.

16.10 순열을 저장하면 $8n$ 바이트이고 순서가 완전 무작위다. 형식 보존 순열은 저장이 없고 임의 위치를 $O(1)$ 에 계산하지만, 순서의 통계는 망의 라운드 수와 키에 달려 균등 순열의 모든 성질을 보장하지 않는다. 블록 섞기는 저장이 $8n$ 이되 읽기가 순차다. 셋 다 균등하지 않은 것을 무엇과 맞바꿨는지가 다르다.

17장

17.1 바이트는 $10^7 \times 128 \times 4 = 5.1$ GB 이고 연산은 $2 \times 10^7 \times 128 = 2.6 \times 10^9$ 이다. $5.1/16.7 = 307$ 밀리초와 $2.6/192 = 13$ 밀리초로 이동이 지배한다. 뒷봉투는 307 밀리초다.

17.2 순차 합 11, 열 합 24, 순차 읽기 4.9 는 파이썬이 원소나 레코드마다 무엇인가를 한 값이다. 타일 어텐션 14 는 연산이 지배하는 자리에서 행렬 곱이 최고 연산의 7 퍼센트에서 도는 값이다. 어텐션 조회 1.4 만 둘째 구간이다.

17.3 시간은 합이라 19.5 초이고 메모리는 최대라 21 MB 다. 토큰화 17.3 초를 1.7 초로 줄이면 전체가 3.9 초로 5 배 준다. 나머지 단계가 그대로라 열 배가 안 된다.

17.4 kd-트리는 2 차원에서 이기고 8 과 32 에서 진다. 역파일은 시간에서 세 번 다 빠르지만 재현율이 25, 12, 7 퍼센트다. 요구가 50 퍼센트면 여섯 줄 중 전부 비교가 다섯 번 이기고, 5 퍼센트면 역파일이 세 번 다 이긴다.

17.5 힌트. 해시 k 개가 라인 k 개를 당기므로 바이트가 $64kn$ 이다. 뒷봉투가 수십 마이크로초인데 실측은 NumPy 의 인덱스 배열 값으로 그 열 배 안팎이다. 하한 위에 있는지가 시험이고, 3 배 안인지는 구현의 몫이다.

17.6 힌트. 어휘 크기의 벡터는 문서당 756 개 실수라 64 개의 12 배다. 이웃의 겹침은 대부분 유지되지만, 다른 토큰이 같은 칸에 접히는 충돌이 일부 이웃을 바꾼다. 잃은 것은 충돌이고 얻은 것은 메모리와 고정 차원이다.

17.7 힌트. 규모 10 에서는 배열 훑기가 NumPy 호출 하나이고 해시 조회가 파이썬 `dict` 하나라 둘 다 마이크로초이며 순위가 안정되지 않는다. 십만에서 훑기가 선형으로 는다. 뒤집히는 자리는 호출의 고정 값과 원소당 값이 같아지는 곳이다.

17.8 힌트. 두 탐색이 같은 키를 찾는지를 시험하면 이진 탐색의 정의가 그대로임이 확인된다. 성적표의 배열 줄과 8 장의 판정이 그 근거이고, Eytzinger 는 정의가 아니라 배치를 바꾼 것이다.

17.9 후보를 다 재도 어느 작업량에서 재느냐가 남는다. 작업량은 연산의 종류와 빈도와 크기의 묶음이고, 그중 하나라도 실제와 다르면 측정이 다른 문제의 답이 된다. 어려운 것은 작업량을 실제와 같게 적는 것이고, 그것은 실제 데이터와 실제 질의 분포에서만 나온다.

17.10 판정이 바뀌면 그 축이 그 명제의 조건을 넘은 것이다. 예를 들어 kd-트리가 8 차원에서도 이기면 데이터에 낮은 내재 차원이 있는 것이고, 12 장의 저주 명제가 가정한 균등 분포가 아니다. 안 바뀌면 그 축의 범위를 적어 두는 것이 판정의 유효 범위다.

측정 환경

이 책의 모든 측정은 아래 기계에서 했다. 독자의 기계에서 다시 재면 절대값은 달라지고 비율은 대체로 유지된다.

항목	값
프로세서	Intel Core i7-8557U, 4 코어, 1.7GHz (터보 4.5GHz)
캐시	L1 32KB (데이터), L2 256KB (코어당), L3 8MB (공유)
주기억	16GB LPDDR3 2133MHz
운영체제	macOS 15.4 (Darwin 24.4.0)
파이썬	3.10.0
NumPy	1.26.4

측정은 `samples/bench/` 의 스크립트로 한다. 스크립트마다 난수 시드를 고정하고, 시간은 `time.perf_counter` 로 최소 다섯 번 반복해 중앙값을 쓴다. 출력 파일 첫 줄에 스크립트 이름, 입력 크기, 반복 횟수, 잰 날짜가 있다.

부록 D

표본 코드 안내

이 책의 코드는 전부 `samples/` 아래에 있다. `samples/dsai/` 가 자료 구조이고, `samples/bench/` 가 실험, `samples/tests/` 가 시험, `samples/exercises/` 가 응용 문제의 답이다. 파이썬 3.10 과 NumPy 만 있으면 된다.

받아서 돌리는 법은 셋이다. 시험은 `python -m pytest tests exercises`, 실험 하나는 `python bench/cNN_이름.py`, 실험 전부는 `bash bench/run-bench.sh` 다. 실험의 출력은 `bench/out/` 에 tsv 로 남고, 본문의 모든 표가 거기서 온다.

절마다 쓴 모듈과 출력 파일이다.

절	모듈	측정 출력
0.1	cost.py	c00-seq-vs-random
0.2	roofline.py	c00-roofline
0.3	knn.py	c00-bruteforce
1.1	tensor.py	c01-boxed
1.2	tensor.py	c01-transpose
1.3	tiling.py	c01-stride, c01-transpose
1.4	batch.py	c01-batch
2.1	dynarray.py	c02-growth
2.2	alloc.py	c02-frag
2.3	kvcache.py	c02-kv
2.4	kvcache.py	c02-prefix
3.1	sparse.py	c03-sparse
3.2	structured.py	c03-structured
3.3	quant.py	c03-quant
3.4	moe.py	c03-moe
4.1	linked.py	c04-chase
4.2	freelist.py	c04-alloc
4.3	linked.py	c04-compact
5.1	stack.py	c05-stack
5.2	tape.py	c05-tape
5.3	ring.py	c05-ring

절	모듈	측정 출력
5.4	batching.py	c05-batch
6.1	hashing.py	c06-probe
6.2	vocab.py	c06-vocab
6.3	vocab.py	c06-embed
6.4	parhash.py	c06-parallel
7.1	bloom.py	c07-bloom
7.2	sketch.py	c07-sketch, c07-hll
7.3	minhash.py	c07-minhash, c07-lsh
7.4	dedup.py	c07-dedup
8.1	bst.py	c08-height
8.2	btree.py	c08-btree
8.3	learned.py	c08-learned
8.4	layout.py	c08-layout
9.1	trie.py	c09-trie
9.2	bpe.py	c09-bpe
9.3	radix.py	c09-radix
9.4	constrained.py	c09-constrained
10.1	heap.py	c10-topk
10.2	beam.py	c10-beam, c10-beamstep
10.3	sampling.py	c10-sampling, c10-topp
10.4	scheduler.py	c10-scheduler
11.1	graph.py	c11-graph
11.2	topo.py	c11-topo, c11-autodiff
11.3	fusion.py	c11-fusion
12.1	kdtree.py	c12-curse
12.2	skiplist.py	c12-skip
12.3	nsw.py	c12-nsw
12.4	diskgraph.py	c12-disk
13.1	ivf.py	c13-ivf
13.2	pq.py	c13-pq
13.3	ivfpq.py	c13-ivfpq
13.4	c13_select.py	c13-select
14.1	suffix.py	c14-suffix
14.2	ngram.py	c14-ngram
14.3	dupspan.py	c14-dup
15.1	attention.py	c15-lookup
15.2	attention.py	c15-attention
15.3	kvevict.py	c15-evict
15.4	ssm.py	c15-ssm
16.1	collective.py	c16-allreduce
16.2	merkle.py	c16-merkle
16.3	columnar.py	c16-columnar
16.4	loader.py	c16-shuffle

절	모듈	측정 출력
17.1	choose.py	c17-envelope
17.2	pipeline.py	c17-pipeline
17.3	c17_scale.py	c17-scale

복잡도 요약표

이 책의 구조 전부의 연산 수와 이동량이다. 둘을 따로 적는 것이 이 책의 잣대다. B 는 라인 크기, s 는 원소 크기, α 는 적재율이다. 절 번호가 그 식을 세운 자리다.

구조	연산	연산 수	이동량	절
배열 (값 배열)	인덱스 읽기	$O(1)$	라인 1	1.1
배열 (값 배열)	순차 훑기 n	$O(n)$	ns/B 라인	1.1
배열 (참조 배열)	순차 훑기 n	$O(n)$	라인 n 개 안팎	1.1
스트라이드 뷰	보폭 k 훑기	$O(n)$	$\min(n, nsk/B)$ 라인	1.2
동적 배열 (두 배)	추가 n 번	분할 상환 $O(1)$	옮기기 $\le 2n$ 원소	2.1
블록 테이블	추가 n 번	$O(1)$	옮기기 0, 예약 블록 1	2.3
CSR 희소 행렬	행 벡터 곱	$O(\text{nnz})$	$\text{nnz}(s+4)$ 바이트	3.1
연결 리스트	순회 n	$O(n)$	라인 n 개 (의존 읽기)	4.1
아레나 인덱스 링크	순회 n	$O(n)$	$4n/B$ 라인 안팎	4.3
배열 스택	넣기와 빼기	$O(1)$	라인 1	5.1
링 버퍼	창 평균 갱신	$O(1)$	라인 1	5.3
해시 표 (개방 주소)	조회	기대 $O(1)$	라인 $1/(1-\alpha)$ 안팎	6.1
해시 표	배치 삽입 m 개	$O(m \log m)$ 정렬	순차 m 원소	6.4
블룸 필터	조회	$O(k)$	라인 k 개	7.1
카운트 민 스케치	갱신과 조회	$O(k)$	라인 k 개	7.2
MinHash 와 LSH	후보 찾기	$O(bk)$	띠 b 개의 표 조회	7.3
이진 탐색 (정렬 배열)	찾기	$O(\log n)$	라인 $\log n$ 개	8.1
B-트리 (팬아웃 f)	찾기	$O(\log_f n)$	블록 $\log_f n$ 개	8.2
학습 인덱스	찾기	$O(1) + O(\log \epsilon)$	라인 $1 + \log \epsilon$	8.3
Eytzinger 배치	배치 찾기	$O(\log n)$	라인 $\log n/4$ 안팎	8.4
트라이 (기수)	접두사 찾기 길이 L	$O(L)$	노드 $\le L$ 개	9.1
이진 힙	넣기와 빼기	$O(\log n)$	라인 $\log n$ 개	10.1
선택 (argpartition)	상위 k	$O(n)$	순차 n 원소	10.1
별칭표	뽑기	$O(1)$	라인 2	10.3
CSR 그래프	이웃 훑기 전부	$O(n+m)$	$4(n+m)$ 바이트 순차	11.1

구조	연산	연산 수	이동량	절
위상 정렬 (Kahn)	전체	$O(n+m)$	$4(n+m)$ 바이트	11.2
kd-트리 (d 작음)	최근접	$O(\log n)$ 기대	노드 $\log n$ 개	12.1
근접 그래프 (HNSW)	최근접	$O(\log n)$ 경험	거리 계산 $M\cdot$ 넓힌 노드	12.3
역파일	질의	$O(kd + \text{nprobe}\cdot nd/k)$	그 만큼의 벡터	13.1
곱 양자화	비대칭 거리	$O(m)$ 표 조회	코드 m 바이트	13.2
접미사 배열	패턴 세기	$O(\lvert P\rvert \log n)$	라인 $\log n$ 개	14.1
어텐션 조회	질의 하나	$O(nd)$	$n(d+d_v)s$ 바이트	15.1
타일 어텐션	m 질의	$O(mnd)$	$(m+n)(d+d_v)s$ 바이트	15.2
선형 재귀	한 단계	$O(s)$	상태 s	15.4
링 all-reduce	벡터 n	$O(n)$	기계당 $2(p-1)ns/p$ 바이트	16.1
머클 트리	다른 블록 k 개 찾기	$O(k\log n)$	해시 $2k\log n$ 개	16.2
열 저장	열 하나 읽기	$O(n)$	ns_j 바이트	16.3
Fisher-Yates	순열	$O(n)$	$8n$ 바이트 (저장)	16.4

기호와 표기

이 책의 수식에 나오는 기호다. 장마다 다시 정의하지 않고 여기서 한 번 적는다.

기호	뜻
n	원소의 수. 배열의 길이, 키의 수, 벡터의 수, 토큰의 수
d	벡터의 차원. 임베딩과 키와 값의 길이
k	찾는 이웃의 수, 해시 함수의 수, 다른 블록의 수. 절마다 정의한다
m	해시 표의 칸 수, 질의의 수, 간선의 수. 절마다 정의한다
B	블록 또는 라인의 크기 (바이트), 또는 캐시의 예산 (토큰)
s	원소 하나의 크기 (바이트), 또는 상태의 크기
β, ϕ	대역폭 (바이트 매초), 최고 연산 (연산 매초)
σ	이동의 단위. 1장에서 원소 하나, 뒤에서 벡터와 텐서
$O(f(n))$	$f(n)$ 의 상수 배로 위에서 막히는 것. 이동량과 연산 수에 따로 쓴다
p	확률, 또는 기계의 수 (16장). 문맥이 정한다
ε	오차 또는 버린 확률 질량
ρ	재현율. 찾은 이웃이 정확한 이웃과 겹치는 비율
α	적재율 (6장), 재조정 계수 (15장), 단계 지연 (16장)
$H(\cdot)$	해시 함수. 충돌이 없다고 가정하지 않는다
SA, LCP	접미사 배열과 최장 공통 접두사 배열
softmax	$e^{s_i} / \sum_j e^{s_j}$. 양수이고 합이 1
$\odot$	원소별 곱
$\lceil x \rceil$, $\lfloor x \rfloor$	올림과 내림
log	밑을 안 적으면 2. 자연로그는 ln

단위는 본문에 그대로 적는다. 바이트, KB, MB, GB 는 1,024 배가 아니라 1,000 배이고, 측정 스크립트가 2^{20} 으로 나눈 자리는 그 표의 조건에 적었다. 시간은 나노초, 마이크로초, 밀리초, 초다.

부록 G

구현 지형도

본문은 라이브러리와 제품의 이름을 쓰지 않았다. 이 책의 구조가 실제로 어디에 구현되어 있는지를 여기 한 쪽에 모은다. 아래는 2026년 9월 기준이고, 항목마다 공식 문서 주소 하나를 근거로 둔다. 순위를 매기지 않고 추천하지 않는다. 확인하지 못한 것은 뺐다.

구현	이 책의 장	어느 구조를 어떻게	공식 문서
NumPy	1, 2, 3	값 배열과 스트라이드 뷰. 이 책의 모든 표본이 이 위에 있다	`https://numpy.org/doc/`
FAISS	12, 13	역파일, 곱 양자화, 근접 그래프를 한 라이브러리에. 인덱스 이름이 이 책의 절 이름과 대응한다	`https://github.com/facebookresearch/faiss/wiki`
hnswlib	12	계층 근접 그래프 하나만. 삽입이 되는 인덱스	`https://github.com/nmslib/hnswlib`
DiskANN	12	디스크 위의 근접 그래프. 벡터와 이웃을 페이지에 같이 둔다	`https://github.com/microsoft/DiskANN`
vLLM	2, 5, 10	페이지드 KV 캐시, 연속 배칭, 스케줄러	`https://docs.vllm.ai/`
SGLang	9	기수 트리로 접두사 캐시를 나눠 쓴다	`https://docs.sglang.ai/`
FlashAttention	15	온라인 소프트맥스의 타일 어텐션. 역방향의 재계산 포함	`https://github.com/Dao-AILab/flash-attention`
infini-gram	14	접미사 배열 위의 n 없는 n-그램	`https://infini-gram.io/`
Apache Arrow, Parquet	16	열 지향 저장과 메모리 맵 읽기	`https://arrow.apache.org/docs/`
NCCL	16	링과 트리의 all-reduce	`https://docs.nvidia.com/deeplearning/nccl/`
Git	16	내용 주소 저장. 해시가 객체의 이름이다	`https://git-scm.com/docs`

표의 값은 목록이 아니라 분류 기준에 있다. 기준은 셋이다. 어느 장의 구조인가, 그 구조를 어떤 단위 (원소, 페이지, 기계) 에 두는가, 갱신이 되는가. 구현이 바뀌어도 이 셋은 남는다.

부록 H

용어집

본문의 이 말부터 상자에서 뽑았다. 낱말, 이 책에서의 뜻, 처음 나온 절 번호다. 어원은 적지 않는다.

2:4 3.2 연속한 원소 넷 중 정확히 둘만 0 이 아니라는 규칙. 밀도 50 퍼센트에 인덱스가 2 비트다

CSR 11.1 행 포인터 배열과 열 배열. 노드 u 의 이웃이 열 배열의 $[\mathrm{ptr}[u], \mathrm{ptr}[u+1])$ 에 연속으로 있다

KV 캐시 2.3 토큰마다 남긴 키와 값 벡터를 요청이 끝날 때까지 쌓아 두는 배열. 요청마다 하나씩이다

LCP 배열 14.1 정렬 순서로 이웃한 두 접미사의 공통 접두사 길이. 다음 두 절이 쓴다

LRU 15.3 가장 오래 참조되지 않은 항목을 떼는 정책. 어텐션에서는 가장 오래된 토큰

all-gather 16.1 기계마다 하나씩 있는 조각을 모두가 전부 갖게 하는 둘째 반

all-reduce 16.1 기계 p 대의 벡터를 원소별로 합쳐 모든 기계가 그 합을 갖게 하는 집합 통신

gather 와 scatter 3.4 흩어진 원소를 모으는 것과 모은 결과를 제자리로 되돌리는 것. 인덱스 배열 하나로 둘 다 한다

kd-트리 12.1 깊이마다 축 하나를 골라 그 축의 중앙값으로 점을 둘로 나눈 이진 트리. 잎이 점 몇 개의 묶음이다

n-그램 표 14.2 길이 $n-1$ 문맥에서 다음 토큰의 개수로 가는 표. n 마다 따로다

nnz 3.1 0 이 아닌 원소의 수. number of nonzeros 의 약자이고 현장에서 그대로 쓴다

nprobe 13.1 질의마다 들여다보는 군집의 수. 1 이면 가장 가까운 중심의 목록만 본다

reduce-scatter 16.1 벡터를 p 조각으로 나눠 조각 i 의 합을 기계 i 가 갖게 하는 첫 반

top-p 10.3 확률이 큰 것부터 누적이 p 를 넘을 때까지만 남기고 나머지를 0 으로 두는 절단

가라앉히기 10.1 뿌리에 온 새 값을 더 큰 자식과 바꾸며 내리는 걸음. 꺼내기의 전부다

가장 긴 반복 14.3 LCP 의 최대. 텍스트에서 두 번 이상 나오는 가장 긴 부분 문자열

가지치기 12.1 지금까지 찾은 최근접 거리보다 분할면까지의 거리가 멀면 그 쪽 부분 트리를 안 보는 것

간격 접근 1.3 원소를 k 개마다 하나씩 읽는 것. 스트라이드가 ks 인 뷰를 훑는 것과 같다

간선 배열 11.1 (출발, 도착) 쌍의 배열 둘. 가장 단순하고, 이웃을 찾으려면 훑어야 한다

값 배열 1.1 원소의 값 자체가 연속으로 놓인 배열. NumPy 의 `ndarray` 가 이것이다

같이 두기 12.4 노드의 벡터와 이웃 목록을 같은 페이지에 놓는 것. 따로 두면 접근이 둘이다

갱신 13.4 벡터가 더해지거나 지워지는 것. 그래프는 삽입이 되고 역파일은 다시 만든다

거리 대비 12.1 최근접 거리를 평균 거리로 나눈 것. 0 에 가까우면 가까운 점이 두드러지고 1 에 가까우면 다 비슷하다

거짓 양성 7.1 없는 키를 있다고 답하는 것. 블룸 필터가 틀리는 유일한 방향이다

거짓 음성 7.1 있는 키를 없다고 답하는 것. 블룸 필터에는 없다

거친 벡터 12.4 벡터를 몇 비트로 접은 것. 메모리에 두고 거리를 어림하는 데 쓴다. 13 장이 그 방법이다

걸음 5.4 배치 안의 모든 활성 요청이 토큰 하나씩 내는 한 번의 실행. 가중치를 한 번 읽는다

구조적 희소성 3.2 0 의 자리에 고정된 규칙이 있는 희소성. 규칙이 있으면 인덱스가 규칙만큼 짧아진다

국소 미분 5.2 연산 하나의 출력을 그 연산의 입력으로 미분한 값. 곱셈이면 상대 입력이다

군집별 거리표 13.3 질의에서 중심을 뺀 것으로 만든 거리표. 군집마다 하나씩 nprobe 개다

굶김 10.4 긴 요청이 짧은 요청에 계속 밀려 한없이 기다리는 것. 최대 대기가 커진다

균일 비용 모델 0.1 어느 주소를 읽든 값이 같다고 치는 계산 모델. 교과서의 복잡도는 전부 이 모델 위에 있다

근사 중복 7.4 조각 집합의 자카드 유사도가 문턱 이상인 문서. 몇 단어가 다른 같은 글이다

근접 그래프 12.3 점이 노드이고 가까운 점 M 개가 이웃인 그래프. 간선이 거리에서 나온다

깊이 8.1 뿌리에서 어느 노드까지의 노드 수. 평균 깊이가 성공 탐색의 평균 비용이다

내부 노드 8.2 잎 위의 층들. 자식 블록의 첫 키만 갖고 있어 잎보다 B 배씩 작다

내부 단편화 2.2 요청에 배정된 자리 중 요청이 안 쓰는 부분. 그 요청 말고는 아무도 못 쓴다

내용 주소 16.2 데이터의 해시를 그 데이터의 키로 쓰는 것. 같은 내용이면 같은 주소다

높이 8.1 뿌리에서 가장 먼 잎까지의 노드 수. 탐색이 최악에 들여다보는 노드 수다

누적 분포 8.3 키 k 이하인 키의 비율. 정렬된 배열에서 위치를 n 으로 나눈 것과 같다

누적합 10.3 $C[i] = p_0 + \cdots + p_i$. 무작위 u 에 대해 $C[i-1] \le u < C[i]$ 인 i 를 뽑는다

다시 재기 13.3 코드로 고른 상위 후보 몇 개를 원래 벡터로 정확히 재어 순서를 고치는 것

단계 17.2 구조 하나가 서는 자리. 입력의 단위와 출력의 단위가 다르다

대역폭 0.2 메모리 계층 사이를 초당 옮길 수 있는 바이트 수. 이 책의 기계에서는 주기억이 약 17GB/s 다

덮는 비율 14.3 반복 구간에 든 위치의 수를 텍스트 길이로 나눈 것. 겹침은 한 번만 센다

뒤집히는 자리 17.3 규모나 차원이 바뀌어 판정이 바뀌는 경계. 작업량이 정한다

뒷봉투 계산 17.1 바이트와 연산 수만으로 시간의 자릿수를 어림하는 것. 루프라인의 하한이다

들어오는 차수 11.2 노드로 들어오는 간선의 수. 0 이면 지금 계산할 수 있다

디코딩 2.3 토큰을 하나씩 내는 단계. 토큰 하나마다 캐시 전체를 한 번 읽는다

띠 7.3 서명을 r 개씩 묶은 조각. 띠마다 해시 테이블을 하나 두고, 어느 띠에서든 겹치면 후보다

라디스 트리 9.3 자식이 하나뿐인 노드들을 간선 하나로 합친 트라이. 간선에 글자 하나가 아니라 열 하나가 달린다

라우터 3.4 토큰을 보고 어느 전문가로 보낼지 정하는 작은 함수. 점수가 가장 큰 쪽을 고른다

링 버퍼 5.3 고정 크기 배열을 고리로 쓰는 큐. 번호가 끝에 닿으면 처음으로 돌아온다

마스크 표 9.4 상태마다 허용 집합을 비트 마스크로 적은 표. 크기가 상태 수 곱하기 어휘 크기다

만들기 비용 13.4 인덱스를 세우는 시간. 그래프는 삽입 n 번, 역파일은 k-평균과 정렬 하나다

머클 트리 16.2 잎이 블록 해시이고 안쪽 노드가 두 자식 해시의 해시인 트리. 뿌리 하나가 전체를 대신한다

메모리 맵 16.3 파일을 주소 공간에 붙여 배열처럼 읽는 것. 닿은 페이지만 읽힌다

명시적 스택 5.1 프로그램이 직접 만든 스택. 원소가 정수 하나나 둘이라 프레임보다 백 배 작다

무한 n-그램 14.2 문맥 길이를 정하지 않고 접미사 배열의 구간으로 개수를 내는 모델

밀도 3.1 nnz 를 전체 원소 수로 나눈 것. 희소 행렬의 값은 전부 이 숫자에 달렸다

반복 구간 14.3 텍스트의 서로 다른 두 위치에서 같은 길이 L 이상의 부분 문자열

배치 1.4 같은 연산을 받을 입력 여러 개를 축 하나로 묶은 것. 그 축의 크기가 배치 크기 B 다

백오프 14.2 긴 문맥이 말뭉치에 없을 때 뒤에서부터 줄여 있는 문맥을 쓰는 규칙

별칭 10.3 자리 i 에 적힌 다른 자리 번호. 동전이 뒤집히면 i 대신 별칭을 낸다

병목 17.2 시간이나 메모리가 가장 큰 단계. 파이프라인의 시간은 병목이 정한다

병합 9.2 이웃한 토큰 쌍 (a, b) 를 새 토큰 하나로 바꾸는 규칙. 학습이 만드는 것은 이 규칙의 열이다

보편 해시 6.1 무작위로 고른 계수 하나로 정해지는 해시 함수의 집합. 어느 키 쌍이든 겹칠 확률이 낮다

부모 배열 10.2 단계 t 의 후보 b 가 단계 $t-1$ 의 어느 후보에서 왔는지 적은 배열. 열은 저장하지 않고 이 배열로 되짚는다

부분 공간 13.2 벡터의 좌표를 m 개 묶음으로 자른 것. 부분마다 d/m 차원이다

분기 없는 탐색 8.4 비교 결과를 조건 분기가 아니라 번호의 산수에 넣는 탐색. 예측 실패가 없다

분할 상환 2.1 연산 열 전체의 비용을 연산 수로 나눈 것. 한 번의 비싼 연산을 여러 싼 연산이 나눠 갚는다

뷰 1.2 버퍼를 새로 만들지 않고 읽는 규칙만 바꾼 것. 원본과 메모리를 공유한다

브로드캐스팅 1.4 모양이 다른 두 텐서를 연산할 때 크기 1 인 축을 상대 크기에 맞춰 늘려 보는 규칙

블록 섞기 16.4 블록 순서를 섞고 블록 안을 섞는 것. 블록 사이만 무작위 접근이다

블록 순회 1.3 큰 배열을 캐시에 드는 작은 사각형으로 잘라 사각형 하나씩 처리하는 순회. 타일링이라고도 한다

블록 테이블 2.4 요청의 논리 블록 번호를 물리 블록 번호로 바꾸는 표. 운영체제의 페이지 테이블과 같은 물건이다

블록 희소 3.2 행렬을 $b \times b$ 블록으로 나누고 블록 단위로 0 인지 아닌지를 적는 표현. BSR 이라고 쓴다

비 17.1 실측을 뒷봉투 시간으로 나눈 값. 구현과 기계 사이의 거리다

비대칭 거리 13.2 질의는 원래 벡터, 데이터는 코드인 채로 재는 거리. 코드를 복원하지 않는다

빔 10.2 한 단계에 남기는 후보 열의 수. B 로 적고, 1 이면 탐욕이고 V^T 이면 전부 보기다

뻗기 10.2 후보 하나에서 어휘 V 개로 다음 토큰을 붙여 후보 V 개를 만드는 일

사슬 11.3 원소별 노드가 한 줄로 이어지고 중간 출력을 다른 노드가 안 쓰는 부분 그래프

상태 공간 모델 15.4 선형 재귀를 층으로 쓰는 시퀀스 모델. 어텐션 대신 상태를 둔다

서로 다른 키의 수 7.2 스트림에 나온 키의 종류 수. 카디널리티라고도 부른다

서명 7.3 집합을 대신하는 고정 길이의 정수 열. 집합이 커도 서명은 k 개다

선인출 1.3 프로세서가 다음에 읽을 라인을 미리 가져오는 것. 순차 접근에서만 잘 맞춘다

선택 10.1 정렬하지 않고 k 번째 값을 기준으로 앞뒤를 가르는 일. 기대 $O(n)$ 이다

선형 재귀 15.4 $h_t = a \odot h_{t-1} + b\,x_t$. 상태 h 가 s 개의 수이고 a 가 감쇠

성장 인자 2.1 용량이 찼을 때 새 용량을 옛 용량의 몇 배로 잡는가. 2 이면 두 배 늘리기다

성적표 17.3 전통 구조 열여섯 개의 판정과 근거 실험 번호. 이 장의 끝에 있다

셔플 16.4 $0, \ldots, n-1$ 의 순열 하나를 무작위로 고르는 것. 균등하면 $n!$ 개가 같은 확률

소프트맥스 15.1 점수 s_i 를 $e^{s_i} / \sum_j e^{s_j}$ 로 바꾸는 것. 양수이고 합이 1 이다

수반 변수 5.2 출력의 미분을 어느 변수에 대해 적은 값. 영어로 adjoint 이고 테이프를 뒤로 읽으며 쌓인다

순위 9.2 병합이 학습된 차례. 부호화에서 순위가 낮은 (먼저 배운) 병합을 먼저 적용한다

스캔 15.4 결합 법칙이 있는 연산의 누적을 두 배씩 건너뛰며 $\log n$ 단계에 얻는 것

스케일 3.3 정수 코드 1 이 실수 얼마에 해당하는가. 양자화 간격 Δ 와 같다

스케치 7.2 스트림 전체를 저장하지 않고 상수 크기의 표에 요약한 것. 물음에 근사로 답한다

스트라이드 1.2 어느 축에서 인덱스가 하나 늘 때 주소가 몇 바이트 움직이는가. 축마다 하나씩이다

스트라이드 0 1.4 축 하나를 늘리되 메모리를 늘리지 않는 방법. 같은 원소를 여러 번 읽는 뷰다

슬라이딩 윈도우 5.3 가장 최근 w 개만 보는 창. 새것이 들어오면 가장 오래된 것이 나간다

슬랩 4.2 같은 등급의 칸을 모아 둔 메모리 덩어리. 등급별 자유 리스트가 그 위에 선다

쓰기 전 복사 2.4 공유 중인 블록에 쓰려 할 때 먼저 내 것으로 복사하는 규칙. 영어로 copy-on-write 다

아레나 4.3 노드를 전부 담는 배열 하나. 노드는 아레나의 칸이고 주소는 칸 번호다

아이칭거 배치 8.4 정렬된 키를 힙 번호 순서로 놓은 배열. 뿌리가 1 번, 자식이 $2i$ 와 $2i+1$ 이다

압축 4.3 노드를 순회 순서대로 아레나에 다시 놓는 것. 그 뒤 순회는 순차 접근이다

양자화 3.3 실수를 정수 몇 비트와 스케일 하나로 적는 것. 되돌리면 원래 값에 가까운 값이 나온다

어텐션 15.1 질의와 키 n 개의 유사도를 확률로 바꿔 값 n 개를 가중 합하는 읽기

어휘 사전 6.2 토큰 문자열에서 정수 번호로 가는 표. 모델이 아는 토큰의 전부이고, 크기가 곧 어휘 크기다

역파일 13.1 중심 번호에서 그 군집의 점 번호 목록으로 가는 표. 중심이 키이고 목록이 값이다

연산 밀도 0.2 바이트 하나를 옮겨 와서 하는 계산의 수. $I = W/M$, 단위는 FLOP/byte

연속 배칭 5.4 토큰 한 걸음마다 끝난 요청을 빼고 기다리는 요청을 넣는 방식. 반복 단위 스케줄링이라고도 한다

열 저장 16.3 열 하나의 모든 행이 잇달아 있는 놓임. 배열의 구조체

오차 상한 8.3 예측 위치와 실제 위치의 차이의 최대. ε 으로 적고, 탐색 창의 반이다

오토마톤 9.4 상태 집합과 바이트마다의 전이. 시작 상태에서 바이트를 먹으며 옮겨 가고, 갈 데가 없으면 죽는다

온라인 소프트맥스 15.2 최댓값과 합을 블록마다 갱신해 마지막에 한 번 나누는 소프트맥스

외부 단편화 2.2 배정되지 않은 자리인데 조각이 작아 새 요청에 못 주는 부분. 총합은 넉넉한데 자리가 없다

용량 2.1 미리 잡아 둔 자리의 수. 크기는 그중 실제로 쓰는 자리의 수다. 크기가 용량을 넘을 수 없다

우선순위 큐 10.4 키가 가장 작은 항목을 먼저 내는 큐. 힙 하나로 넣기와 꺼내기가 로그다

원소별 연산 11.3 출력의 원소 i 가 입력의 원소 i 에만 달린 연산. 덧셈, 곱셈, tanh 같은 것

위상 순서 11.2 모든 간선 (u, v) 에 대해 u 가 v 보다 앞에 오는 노드의 나열. DAG 에만 있다

융합 11.3 그래프의 노드 여럿을 하나로 합쳐 중간 텐서를 메모리에 안 내보내는 것

의존 읽기 4.1 앞 읽기의 결과에 주소가 달린 읽기. 프로세서가 미리 가져올 수 없다

이동량 0.1 메모리 계층을 가로질러 옮겨진 바이트 수. 이 책은 연산 수와 따로 센다

이상값 3.3 다른 값들보다 훨씬 큰 값. 스케일을 혼자 정해 버려 나머지의 정밀도를 깎는다

인덱스 링크 4.3 포인터 대신 칸 번호로 다음 노드를 가리키는 것. 4 바이트로 충분하다

인접 리스트 11.1 노드마다 나가는 이웃의 목록. 목록이 노드 수만큼 있다

임베딩 0.3 문서나 토큰을 고정 차원의 실수 벡터로 바꾼 것. 이 절에서는 난수 벡터로 대신한다

임베딩 테이블 6.3 번호마다 벡터 하나를 둔 2차원 배열. 조회는 행 하나를 읽는 것이다

잎 8.2 키 전부가 정렬되어 놓인 맨 아래 층. 이 절에서는 원래 배열 그대로다

자리 충돌 6.4 한 걸음에 둘 이상의 키가 같은 자리에 쓰려는 것. 하나만 성공하고 나머지는 다음 걸음으로 간다

자유 리스트 4.2 빈 칸을 연결 리스트로 이어 둔 것. 머리가 다음에 줄 칸이다

자카드 유사도 7.3 두 집합의 교집합 크기를 합집합 크기로 나눈 것. 0 에서 1 사이다

작업량 0.2 구조에 가해지는 연산의 종류와 빈도와 크기의 묶음. 자료 구조는 작업량에 대한 가정의 묶음이다

잔차 13.3 벡터에서 그 군집의 중심을 뺀 것. 중심 근처의 점일수록 작다

재조정 15.2 새 최댓값이 나왔을 때 옛 누적에 $e^{m_{옛}-m_{새}}$ 를 곱하는 것

재현율 0.3 근사 풀이가 찾은 k 개 중 정확한 답 k 개에 든 것의 비율. 1.0 이 정확 탐색이다

재현율 곡선 13.4 인덱스의 설정 (nprobe, ef) 을 바꿀 때 재현율과 지연이 그리는 선. 목표에서 가장 왼쪽 점을 고른다

적재율 6.1 넣은 키 수를 자리 수로 나눈 것. α 로 적고, 해시 테이블의 값은 전부 이 숫자에 달렸다

전문가 3.4 같은 모양의 행렬 여러 개 중 하나. 토큰마다 그중 하나 또는 몇 개만 곱한다

접미사 14.1 위치 i 부터 끝까지의 부분 문자열 $T[i..n)$. 접미사가 n 개다

접미사 배열 14.1 접미사를 사전 순으로 정렬했을 때의 시작 위치 배열 SA

정렬 기반 세기 6.4 키를 정렬해 같은 키를 붙여 놓고 경계를 찾아 개수를 세는 것. 표가 없다

정적 배칭 5.4 배치를 채워 시작하고, 그 배치의 모든 요청이 끝날 때까지 같이 가는 방식

정확 중복 7.4 정규화한 뒤 바이트까지 같은 문서. 해시 하나가 같으면 중복이다

조각 7.4 문서를 집합으로 바꿀 때의 원소. 이 절은 단어 두 개짜리를 쓴다

주소 계산 1.1 인덱스에서 바이트 위치를 구하는 식. 배열에서는 곱셈 하나와 덧셈 하나다

중심 13.1 군집 하나를 대표하는 벡터. 그 군집에 든 점의 평균이다

지름길 12.2 위 층의 간선. 아래 층의 노드 여럿을 한 번에 건너뛴다

지연 4.1 읽기 하나를 시작해서 값이 도착할 때까지의 시간. 대역폭과 다른 양이다

지프 분포 6.2 흔한 것이 아주 흔하고 드문 것이 아주 많은 분포. 토큰의 빈도가 이 모양이다

직접 주소 6.3 키를 그대로 자리로 쓰는 것. 해시 함수가 항등이고 충돌이 없다

진입점 12.3 탐색을 시작하는 노드. 층이 있으면 맨 위 층의 노드 하나다

질의 배치 8.4 한꺼번에 받은 질의의 묶음. 정렬하면 이웃한 질의가 같은 라인을 지난다

짧은 것 먼저 10.4 처리 시간이 짧은 요청을 먼저 하는 정책. 평균 대기가 최소다

차분 16.2 두 판에서 다른 블록의 집합. 새로 저장할 것이 그것뿐이다

차수 8.2 노드 하나에 드는 키의 수. B 로 적고, 블록 크기를 키 크기로 나눈 값으로 정한다

참조 계수 2.4 물리 블록 하나를 몇 요청이 가리키는지 센 수. 0 이 되면 풀로 돌아간다

참조 배열 1.1 원소 자리에 값이 아니라 값이 있는 곳의 주소가 놓인 배열. 파이썬 `list` 가 이것이다

첫 맞춤 2.2 빈 구간을 앞에서부터 보다가 들어가는 첫 곳에 넣는 연속 할당 규칙

최근접 탐색 0.3 벡터 n 개 중 질의와 가장 가까운 k 개를 찾는 문제. 영어로 k-NN 이라 쓴다

최장 공통 접두사 9.3 새 요청의 토큰 열과 트리 안 어느 경로가 앞에서부터 같은 가장 긴 길이

최장 일치 9.1 텍스트의 한 위치에서 시작하는 어휘 항목 중 가장 긴 것을 고르는 규칙

추가 전용 2.3 끝에 붙이기만 하고 중간을 고치지 않는 배열. 로그 파일과 같은 성질이다

축출 9.3 캐시가 차서 무언가를 떼는 일. 잎만 떼면 남은 것은 여전히 접두사다

층 12.2 정렬된 리스트 하나. 층 0 이 노드 전부이고 층이 오를수록 노드가 p 배씩 준다

캐시 라인 0.1 프로세서가 메모리에서 한 번에 가져오는 단위. 이 책의 기계에서는 64 바이트다

코드북 13.2 부분 공간 하나의 대표 벡터 256 개. 부분마다 하나씩 m 개다

큐와 덱 5.3 큐는 뒤에 넣고 앞에서 빼는 열, 덱은 양 끝에서 넣고 빼는 열. 링 버퍼는 둘의 배열 구현이다

크기 등급 4.2 요청 크기를 몇 개의 고정 크기로 올림한 것. 등급마다 자유 리스트 하나를 둔다

키당 비트 7.1 비트 수를 키 수로 나눈 것. m/n 으로 적고, 오답률은 이 숫자와 해시 수로만 정해진다

타일 15.2 키와 값을 블록 b 개씩 자른 조각. 블록 점수 $m \times b$ 만 만든다

탐사 6.1 키를 찾을 때 들여다본 자리의 수. 한 번이면 해시 자리에 바로 있었다는 뜻이다

탐욕 탐색 12.3 지금 노드의 이웃 중 질의에 더 가까운 노드로 옮겨 가는 것. 후보 ef 개를 함께 든다

테이프 5.2 앞으로 계산하며 연산마다 (입력, 국소 미분) 을 적은 추가 전용 배열. 뒤로 읽는다

토큰 6.2 문자열을 자른 단위. 단어일 수도 단어 조각일 수도 있다. 9 장에서 자르는 법을 세운다

토큰당 바이트 9.2 텍스트의 바이트 수를 토큰 수로 나눈 것. 어휘가 텍스트를 얼마나 압축하는지의 잣대다

토큰화 9.1 바이트 열을 어휘 항목의 번호 열로 바꾸는 일. 바이트 256 개가 어휘에 있으면 언제나 된다

트라이 9.1 간선에 글자 하나씩을 달아 뿌리에서 노드까지의 경로가 문자열이 되는 트리

파이프라인 17.2 단계의 열. 앞 단계의 출력이 뒤 단계의 입력이다

판정 17.3 어느 구조가 이 작업량에서 맞는가. 그대로 유효, 조정해서 유효, 이제 안 맞음

팬아웃 11.2 한 노드의 출력을 쓰는 노드의 수. 역방향에서 그 수만큼의 기울기가 더해진다

페이지 12.4 디스크에서 한 번에 읽는 단위. 4 KB 안팎이고, 한 번의 접근은 한 페이지다

포인터 추적 4.1 읽은 값이 다음에 읽을 주소인 읽기의 연쇄. 읽기 하나가 끝나야 다음이 시작된다

프레임 5.1 함수 호출 하나가 스택에 차지하는 덩어리. 파이썬에서는 수백 바이트다

해싱 트릭 6.3 사전 없이 토큰을 해시해 그 값을 번호로 쓰는 것. 자리가 어휘보다 적어 충돌을 허용한다

행 우선 1.2 마지막 축이 가장 촘촘한 배치. C 와 NumPy 의 기본이고, 열 우선은 그 반대다

행 저장 16.3 레코드 하나의 모든 열이 잇달아 있는 놓임. 구조체의 배열

행 포인터 3.1 행마다 그 행의 원소가 값 배열의 어디서 시작하는지 적은 배열. 길이가 행 수 더하기 1 이다

허용 집합 9.4 상태 s 에서 먹어도 안 죽는 토큰들의 집합. 디코딩 때 이 밖의 토큰은 확률을 0 으로 둔다

헤드 15.1 같은 키와 값 배열을 다른 사영으로 읽는 독립된 어텐션 하나. 층마다 여럿이다

헤비 히터 7.2 스트림에서 전체의 일정 비율 이상을 차지하는 키. 빈도 상위의 키들이다

형식 보존 순열 16.4 순열을 저장하지 않고 $i \mapsto \pi(i)$ 를 계산하는 전단사. 키가 순열을 정한다

호출 스택 5.1 함수를 부를 때마다 언어가 쌓는 프레임의 스택. 프레임에 지역 변수와 돌아갈 자리가 든다

회전 8.1 순서 불변식을 지키면서 부모와 자식을 바꾸는 한 걸음. 균형은 전부 회전으로 잡는다

후보 17.1 같은 일을 하는 구조 여럿. 정확성과 갱신 가능성이 성질이다

힙 불변식 10.1 모든 노드가 두 자식 이상이라는 조건. 최대가 뿌리에 있다는 것 말고는 아무 순서도 약속하지 않는다

절 의존 지도

절마다 새로 세운 개념 하나와 그 절이 앞에서 가져다 쓴 절이다. 뒤 절은 앞 절만 쓴다. 표는 `DEPS.json` 에서 뽑았고, 검사 스크립트가 매 루프 그 규칙을 확인한다.

절	제목	쓰는 절
0.1	자료 구조는 가정의 묶음이다	없음
0.2	바뀐 가정 넷	0.1
0.3	이 책이 끝까지 다시 보는 문제	0.1, 0.2
1.1	배열의 불변식과 주소 계산	0.1
1.2	스트라이드와 뷰	0.1, 1.1
1.3	메모리 계층과 접근 순서	0.1, 0.2, 1.2
1.4	배치와 브로드캐스팅	0.2, 0.3, 1.2
2.1	분할 상환 분석과 두 배 늘리기	0.1, 1.1
2.2	단편화, 예약과 낭비의 산수	2.1
2.3	KV 캐시라는 추가 전용 배열	0.2, 1.4, 2.1
2.4	페이지드 어텐션, 블록 테이블과 공유	1.2, 2.1, 2.2, 2.3
3.1	COO, CSR, CSC. 세 가지 압축과 각각 빠른 연산	0.1, 0.2, 1.1
3.2	블록 희소와 구조적 희소성	1.3, 3.1
3.3	양자화된 텐서. 스케일과 블록과 조회표	0.2, 1.1
3.4	전문가 혼합의 gather 와 scatter	1.4, 2.3, 3.1
4.1	연결 리스트와 포인터 추적의 값. 캐시 미스를 재다	0.1, 1.1, 1.3
4.2	자유 리스트와 블록 풀. 할당자 안의 연결 리스트	2.2, 2.4, 4.1
4.3	배열 안의 연결 리스트. 인덱스 링크와 아레나	1.2, 4.1, 4.2
5.1	호출 스택과 재귀 깊이. 명시적 스택으로 바꾸기	1.1, 2.1
5.2	테이프. 역방향 자동 미분의 스택	0.2, 2.3, 5.1
5.3	큐와 덱, 링 버퍼. 슬라이딩 윈도우	1.1, 4.3, 5.1
5.4	배칭 큐와 연속 배칭	1.4, 2.3, 4.2, 5.3

절	제목	쓰는 절
6.1	해시 함수와 충돌 해결. 체이닝, 개방 주소, 쿠쿠	0.1, 1.1, 4.1
6.2	어휘 사전. 토큰과 정수 사이	4.3, 6.1
6.3	해싱 트릭과 임베딩 테이블	1.1, 6.1, 6.2
6.4	병렬 삽입. 가속기 위의 해시 테이블	1.3, 1.4, 6.1
7.1	블룸 필터. 없음은 안 틀리는 집합	0.1, 1.1, 6.1
7.2	카운트 민 스케치. 개수를 어림하는 표	6.1, 6.4, 7.1
7.3	MinHash 와 지역 민감 해싱. 비슷한 것을 같은 자리로	0.2, 0.3, 6.1
7.4	학습 데이터 중복 제거 파이프라인	0.3, 6.4, 7.1, 7.3
8.1	이진 탐색 트리와 균형. 높이가 곧 비용	4.1, 4.3, 5.1, 6.1
8.2	B-트리. 블록 단위로 탐색하기	0.1, 1.2, 1.3, 8.1
8.3	학습된 인덱스. 누적 분포를 모델로 바꾸다	0.2, 6.3, 8.1, 8.2
8.4	정렬된 배열과 이진 탐색에 다시 서기. 배치와 분기 없는 탐색	1.1, 4.1, 6.4, 8.1, 8.2
9.1	트라이와 최장 일치	1.1, 6.1, 6.2, 8.1
9.2	바이트 쌍 부호화와 병합 순서	6.2, 6.3, 7.2, 9.1
9.3	라디스 트리와 접두사 공유 캐시	2.3, 2.4, 9.1
9.4	제약 디코딩. 트라이 위의 오토마톤	2.3, 5.1, 9.1, 9.2
10.1	이진 힙과 top-k. 전부 정렬하지 않는 값	0.1, 1.1, 8.1, 8.4
10.2	빔 탐색의 힙	1.2, 2.3, 2.4, 5.2, 9.3, 10.1
10.3	누적합과 별칭표. 확률 분포에서 뽑기	2.1, 3.4, 8.4, 10.1
10.4	우선순위 스케줄러. 요청을 줄 세우기	5.3, 5.4, 10.1
11.1	그래프 표현. 인접 리스트와 CSR 다시 보기	1.2, 3.1, 3.4, 4.1, 6.4
11.2	위상 정렬과 자동 미분	2.3, 5.1, 5.2, 11.1
11.3	연산자 융합. 그래프를 다시 쓰다	0.1, 0.2, 1.3, 11.1, 11.2
12.1	차원의 저주. kd-트리가 무너지는 자리	0.3, 5.1, 7.3, 8.1
12.2	스킵 리스트. 계층으로 건너뛰기	4.1, 4.3, 8.1, 8.4
12.3	탐색 가능한 작은 세상 그래프. NSW 와 HNSW	0.3, 10.1, 11.1, 12.1, 12.2
12.4	디스크 위의 그래프. 한 번의 접근으로	0.1, 3.3, 8.2, 12.3
13.1	k-평균과 역파일	0.3, 6.4, 11.1, 12.1
13.2	곱 양자화. 벡터를 바이트 몇 개로	0.1, 3.3, 6.3, 13.1
13.3	역파일과 곱 양자화의 결합. 비대칭 거리	0.2, 12.4, 13.1, 13.2
13.4	재현율과 지연. 인덱스 고르기	0.1, 0.3, 12.3, 13.3
14.1	접미사 배열	0.1, 6.4, 8.4, 9.1
14.2	n-그램 세기. 접미사 배열 위의 무한 n-그램	6.1, 6.2, 12.4, 14.1
14.3	긴 중복 찾기. 말뭉치 안의 복사본	6.4, 7.4, 14.1
15.1	어텐션은 연관 배열이다	0.1, 0.2, 0.3, 2.3, 6.1
15.2	타일링과 온라인 소프트맥스. 행렬을 안 만든다	0.1, 0.2, 1.3, 11.3, 15.1
15.3	KV 캐시 축출. LRU 에서 헤비 히터로	2.3, 2.4, 5.3, 7.2, 15.1
15.4	고정 크기 상태. 재귀와 스캔	0.1, 2.3, 5.3, 15.1, 15.3

절	제목	쓰는 절
16.1	링 위의 all-reduce. 조각을 돌린다	0.1, 0.2, 1.1, 1.4, 5.3
16.2	머클 트리와 내용 주소 저장. 체크포인트의 차분	0.1, 6.1, 7.1, 7.4, 8.1
16.3	열 지향 저장과 메모리 맵	0.1, 1.1, 1.2, 1.3, 8.2
16.4	셔플과 순열. 데이터 로더의 자료 구조	0.1, 1.3, 6.1, 10.3, 16.3
17.1	측정 없이 고르지 않는다. 뒷봉투 계산에서 루프라인까지	0.1, 0.2, 0.3, 1.4, 13.4
17.2	사례 연구. dsai 를 처음부터 끝까지	6.3, 7.4, 9.2, 9.3, 13.1, 14.1, 17.1
17.3	AI 가 코드를 쓰는 시대의 선택	8.3, 12.1, 13.1, 15.3, 17.1, 17.2

부록 J

이 책이 다루지 않는 것

한 권으로 끝나는 책이라 못 다룬 것이 있다. 다음 권으로 미루지 않고 여기 이유와 함께 적는다.

다루지 않은 것	이유
가속기 커널 작성	이 책의 도구는 파이썬과 NumPy 다. 타일과 융합의 이동량은 셈했지만 커널 자체는 쓰지 않았다. 15장의 비 14 가 그 경계다
분산 학습의 병렬화 전략	16장은 all-reduce 하나만 세웠다. 텐서 병렬과 파이프라인 병렬은 자료 구조가 아니라 스케줄링의 문제다
균형 트리 (AVL, 레드 블랙)	8장의 판정대로 노드가 흩어진 트리는 이 책의 작업량에서 서지 않는다. B-트리와 정렬 배열이 그 자리를 대신했다
유니온 파인드	중복 무리를 잇는 일은 11장의 그래프 위에서 했다. 따로 절을 두면 그래프 표현과 겹친다
문자열 매칭 알고리즘 (KMP, 보이어 무어)	14장의 접미사 배열이 모든 부분 문자열 찾기를 한 번에 하고, 이 책의 작업량은 패턴 하나가 아니라 말뭉치 전체다
동시성 자료 구조 (잠금 없는 큐)	5장의 큐는 단일 스레드였다. 잠금과 메모리 순서는 자료 구조가 아니라 실행 모델의 문제다
데이터베이스의 트랜잭션과 로그	16장의 저장은 읽기와 판의 차분까지다. 원자성과 복구는 다른 책 한 권이다
GPU 메모리 계층 실측	0장의 루프라인은 이 기계의 CPU 에서 잰 것이다. 가속기의 수치는 인용만 하고 재지 않았다
학습 인덱스의 학습 절차	8장은 단조 함수 하나를 맞추는 것으로 끝냈다. 재귀 모델과 갱신은 다루지 않았다
정확한 그래프 알고리즘 (최단 경로, 흐름)	11장과 12장의 그래프는 훑기와 근접 탐색이다. 이 책의 작업량에 최단 경로가 없었다

이 표의 항목은 이 책의 잣대로 판정하지 않은 것들이다. 판정하지 않았으니 성적표에도 없다.

더 읽을 것 서른 편

원전 서른 편이다. 한 줄 평과 이어지는 절 번호, 그리고 원문을 볼 수 있는 곳을 적는다. 주소는 DOI 와 arXiv 처럼 없어지지 않을 것만 적었고, 책은 출판사만 적었다. 전부 참고 문헌에 확인 날짜가 있다.

S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76. DOI: 10.1145/1498765.1498785
루프라인 모델의 원전. 여덟 쪽이고 그림이 전부다. 이 장의 실험 0.2 를 읽고 나서 보면 한 시간에 끝난다. (0.2 절)
https://doi.org/10.1145/1498765.1498785

R. E. Tarjan. "Amortized Computational Complexity". In: *SIAM Journal on Algebraic and Discrete Methods* 6.2 (1985), pp. 306–318. DOI: 10.1137/0606031
분할 상환 분석의 원전. 잠재 함수 방법이 여기서 나왔다. 정리 2.1 의 증명을 잠재 함수로 다시 써 보면 좋다. (2.1 절)
https://doi.org/10.1137/0606031

B. H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426. DOI: 10.1145/362686.362692
블룸 필터의 원전. 두 쪽이고, 오답률의 식과 디스크를 안 가는 예가 다 있다. (7.1 절)
https://doi.org/10.1145/362686.362692

G. Cormode and S. Muthukrishnan. "An Improved Data Stream Summary: The Count-Min Sketch and its Applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58–75. DOI: 10.1016/j.jalgor.2003.12.001
카운트 민 스케치의 원전. 오차 정리의 증명과 내적, 범위 물음으로의 확장이 있다. (7.2 절)
https://doi.org/10.1016/j.jalgor.2003.12.001

A. Z. Broder. "On the Resemblance and Containment of Documents". In: *Proceedings of Compression and Complexity of Sequences* (*SEQUENCES 1997*). IEEE, 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900

MinHash 의 원전. 일치 확률 정리와 웹 문서 3천만 개의 실험이 있다. (7.3 절)
https://doi.org/10.1109/SEQUEN.1997.666900

P. Indyk and R. Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality". In: *Proceedings of the 30th Annual ACM Symposium on Theory of Computing* (*STOC*). 1998, pp. 604–613. DOI: 10.1145/276698.276876
지역 민감 해싱의 틀을 세운 논문. 띠 나누기의 증폭이 여기서 일반화됐고, 12장의 디딤돌이다. (7.3 절)
https://doi.org/10.1145/276698.276876

P. Flajolet et al. "HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm". In: *Proceedings of the 2007 International Conference on Analysis of Algorithms* (*AofA*). DMTCS Proceedings. 2007. URL: https://hal.inria.fr/hal-00406166
하이퍼로그로그의 원전. 상수 α_m 과 상대 오차 $1.04/\sqrt{m}$ 의 유도가 있다. (7.2 절)
https://hal.inria.fr/hal-00406166

R. Bayer and E. M. McCreight. "Organization and Maintenance of Large Ordered Indexes". In: *Acta Informatica* 1.3 (1972), pp. 173–189. DOI: 10.1007/BF00288683
B-트리의 원전. 디스크 페이지 위의 인덱스라는 가정과 분할, 병합이 다 있다. (8.2 절)
https://doi.org/10.1007/BF00288683

T. Kraska et al. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data* (*SIGMOD*). 2018, pp. 489–504. DOI: 10.1145/3183713.3196909
학습된 인덱스를 제안한 논문. 내부 노드가 누적 분포 함수라는 관찰과 두 층 모델의 실험이 있다. (8.3 절)
https://doi.org/10.1145/3183713.3196909

E. Fredkin. "Trie Memory". In: *Communications of the ACM* 3.9 (1960), pp. 490–499. DOI: 10.1145/367390.367400
트라이의 원전. 열 쪽이고, 접근 시간과 갱신과 길이가 다른 키에 대한 장점이 그대로 적혀 있다. (9.1 절)
https://doi.org/10.1145/367390.367400

D. R. Morrison. "PATRICIA: Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *Journal of the ACM* 15.4 (1968), pp. 514–534. DOI: 10.1145/321479.321481
경로 압축 트라이의 원전. 라디스 트리의 불변식과 비트 단위의 갈림이 여기서 나왔다. (9.3 절)
https://doi.org/10.1145/321479.321481

R. Sennrich, B. Haddow, and A. Birch. "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Compu-*

tational Linguistics (*Volume 1: Long Papers*). 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162
바이트 쌍 부호화를 어휘 만들기에 가져온 논문. 단어 안에서만 병합하는 규칙이 이 장의 구현이다. (9.2 절)
https://doi.org/10.18653/v1/P16-1162

J. W. J. Williams. "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6 (1964), pp. 347–348. DOI: 10.1145/512274.512284
이진 힙과 힙 정렬의 원전. 두 쪽이고, 배열 위의 완전 이진 트리라는 배치가 여기서 나왔다. (10.1 절)
https://doi.org/10.1145/512274.512284

A. J. Walker. "An Efficient Method for Generating Discrete Random Variables with General Distributions". In: *ACM Transactions on Mathematical Software* 3.3 (1977), pp. 253–256. DOI: 10.1145/355744.355749
별칭표의 원전. 네 쪽이고, 두 번의 메모리 참조로 뽑는다는 문장이 그대로 있다. (10.3 절)
https://doi.org/10.1145/355744.355749

M. D. Vose. "A Linear Algorithm for Generating Random Numbers with a Given Distribution". In: *IEEE Transactions on Software Engineering* 17.9 (1991), pp. 972–975. DOI: 10.1109/32.92917
별칭표를 선형 시간에 만드는 알고리즘. 이 장의 만들기가 이것이다. (10.3 절)
https://doi.org/10.1109/32.92917

A. B. Kahn. "Topological Sorting of Large Networks". In: *Communications of the ACM* 5.11 (1962), pp. 558–562. DOI: 10.1145/368996.369025
위상 정렬의 원전. 다섯 쪽이고, 들어오는 것이 없는 노드부터 꺼내는 규칙이 그대로 있다. (11.2 절)
https://doi.org/10.1145/368996.369025

R. E. Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010
깊이 우선 탐색으로 강한 연결 성분을 선형에 찾는 논문. 명시적 스택의 위상 정렬과 사이클 검출이 여기서 나왔다. (11.2 절)
https://doi.org/10.1137/0201010

J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517. DOI: 10.1145/361002.361007
kd-트리의 원전. 차원 k 를 키에 둔 시절의 구조이고, 12.1 절이 그 가정을 무너뜨린다. (12.1 절)
https://doi.org/10.1145/361002.361007

W. Pugh. “Skip Lists: A Probabilistic Alternative to Balanced Trees”. In: *Communications of the ACM* 33.6 (1990), pp. 668–676. DOI: 10.1145/78973.78977
스킵 리스트의 원전. 층의 확률과 탐색 비용의 분석이 12.3 절의 층으로 옮겨 간다. (12.2 절)
https://doi.org/10.1145/78973.78977

Y. A. Malkov and D. A. Yashunin. “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.4 (2020), pp. 824–836. DOI: 10.1109/TPAMI.2018.2889473
계층 근접 그래프. 이웃 고르기 휴리스틱과 ef 의 역할, 층의 확률이 실험으로 정리되어 있다. (12.3 절)
https://doi.org/10.1109/TPAMI.2018.2889473

H. Jégou, M. Douze, and C. Schmid. “Product Quantization for Nearest Neighbor Search”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57
곱 양자화와 비대칭 거리, 역파일 위의 잔차 양자화까지 한 논문에 있다. 13 장의 원전이다. (13.2, 13.3 절)
https://doi.org/10.1109/TPAMI.2010.57

S. P. Lloyd. “Least Squares Quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489
k-평균 반복의 원전. 1957 년의 기술 보고가 1982 년에 실렸고, 배정과 갱신이 목적 함수를 안 늘린다는 것이 여기 있다. (13.1 절)
https://doi.org/10.1109/TIT.1982.1056489

U. Manber and G. Myers. “Suffix Arrays: A New Method for On-Line String Searches”. In: *SIAM Journal on Computing* 22.5 (1993), pp. 935–948. DOI: 10.1137/0222058
접미사 배열의 원전. 정렬과 탐색이라는 틀과 접미사 트리 대비 메모리 비교가 있다. (14.1 절)
https://doi.org/10.1137/0222058

A. Vaswani et al. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems 30* (*NeurIPS*). 2017, pp. 5998–6008. arXiv: 1706.03762
어텐션의 원전. 질의, 키, 값의 표기와 $\sqrt{d}$ 의 이유가 있다. 연관 배열로 읽는 출발점이다. (15.1 절)
https://arxiv.org/abs/1706.03762

M. Milakov and N. Gimelshein. *Online Normalizer Calculation for Softmax*. 2018. arXiv: 1805.02867

온라인 정규화. 최댓값과 합을 한 번 훑기로 얻는 갱신이 정리 그대로 있다. (15.2 절)
https://arxiv.org/abs/1805.02867

T. Dao et al. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Advances in Neural Information Processing Systems 35* (*NeurIPS*). 2022. arXiv: 2205.14135
FlashAttention. 온라인 소프트맥스로 점수 행렬을 느린 메모리에 안 쓰는 타일 어텐션. 역방향의 재계산도 여기 있다. (15.2 절)
https://arxiv.org/abs/2205.14135

P. Patarasuk and X. Yuan. "Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations". In: *Journal of Parallel and Distributed Computing* 69.2 (2009), pp. 117–124. DOI: 10.1016/j.jpdc.2008.09.002
링 all-reduce 의 대역폭 최적성 증명과 클러스터 실측. 16.1 절의 하한이 이 논문의 정리다. (16.1 절)
https://doi.org/10.1016/j.jpdc.2008.09.002

R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology* (*CRYPTO '87*), *LNCS 293*. Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32
머클 트리의 원전. 잎 하나를 뿌리와 경로의 해시로 확인하는 구조가 여기 있다. (16.2 절)
https://doi.org/10.1007/3-540-48184-2_32

R. Durstenfeld. "Algorithm 235: Random Permutation". In: *Communications of the ACM* 7.7 (1964), p. 420. DOI: 10.1145/364520.364540
균등 순열의 다섯 줄. 16.4 절의 정리가 이 알고리즘이다. (16.4 절)
https://doi.org/10.1145/364520.364540

W. Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th ACM Symposium on Operating Systems Principles* (*SOSP*). 2023, pp. 611–626. DOI: 10.1145/3600006.3613165
페이지드 어텐션의 원전. 낭비 측정, 블록 테이블, 참조 계수, 쓰기 전 복사가 전부 이 한 편에 있다. (2.3, 2.4 절)
https://doi.org/10.1145/3600006.3613165

부록 L

3년 뒤 가장 먼저 틀릴 설명 다섯

이 책의 설명 중 기계와 도구가 바뀌면 먼저 틀릴 것을 필자가 지목한다. 틀릴 것을 알고 쓴 것이고, 틀리면 그 자리의 실험을 다시 돌리면 된다.

1. **0장의 루프라인 수치.** 대역폭 16.7 GB/s 와 최고 192 GFLOPS 는 2019년 노트북의 값이다. 능선이 옮겨 가면 6장, 11장, 15장의 배속이 같이 옮겨 간다. 절대값이 아니라 비만 남는다.
2. **15장의 KV 캐시 축출.** 헤비 히터와 최근 창은 2023년의 정책이고, 캐시를 통째로 요약하거나 학습으로 고르는 정책이 그 자리를 대신할 수 있다. 정의의 예산과 오차 상한은 남고, 정책의 이름이 바뀐다.
3. **13장의 인덱스 선택표.** 벡터 백만 개 이하에서 근접 그래프, 수억 개에서 양자화라는 경계는 메모리 가격의 함수다. 메모리가 싸지면 경계가 오른쪽으로 옮겨 간다.
4. **9장의 BPE.** 바이트 쌍 병합은 2016년의 방법이고, 토큰화 자체를 모델이 배우는 방향이 있다. 트라이와 접두사 캐시는 남고, 병합 표의 자리가 바뀐다.
5. **17장의 비 3 배 경계.** 파이썬 루프의 값이 3 배를 만드는 것이고, 코드를 기계가 쓰는 시대에 그 루프가 사라지면 경계는 1 에 가까워진다. 그때 이 책의 표는 구조가 아니라 구현을 잰 표였다는 것이 더 분명해진다.

다섯 다 실험 번호가 있다. 틀렸는지 확인하는 법은 그 실험을 자기 기계에서 돌리는 것이다.

참고문헌

[1] A. Aggarwal and J. S. Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127. DOI: 10.1145/48529.48535.

[2] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[3] J. L. Hennessy and D. A. Patterson. "A New Golden Age for Computer Architecture". In: *Communications of the ACM* 62.2 (2019), pp. 48–60. DOI: 10.1145/3282307.

[4] J. Johnson, M. Douze, and H. Jégou. "Billion-Scale Similarity Search with GPUs". In: *IEEE Transactions on Big Data* 7.3 (2019), pp. 535–547. DOI: 10.1109/TBDATA.2019.2921572.

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-Oblivious Algorithms". In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (*FOCS*). 1999, pp. 285–298.

[6] C. R. Harris et al. "Array Programming with NumPy". In: *Nature* 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[7] U. Drepper. *What Every Programmer Should Know About Memory*. Tech. rep. 2007년 11월 21일 판. Red Hat, Inc., 2007.

[8] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. "Orca: A Distributed Serving System for Transformer-Based Generative Models". In: *16th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI*). 2022, pp. 521–538.

[9] W. Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th ACM Symposium on Operating Systems Principles* (*SOSP*). 2023, pp. 611–626. DOI: 10.1145/3600006.3613165.

[10] R. E. Tarjan. "Amortized Computational Complexity". In: *SIAM Journal on Algebraic and Discrete Methods* 6.2 (1985), pp. 306–318. DOI: 10.1137/0606031.

[11] P. J. Denning. "Virtual Memory". In: *ACM Computing Surveys* 2.3 (1970), pp. 153–189. DOI: 10.1145/356571.356573.

[12] R. Pope et al. "Efficiently Scaling Transformer Inference". In: *Proceedings of Machine Learning and Systems* (*MLSys*). Vol. 5. 2023.

[13] A. Mishra et al. *Accelerating Sparse Deep Neural Networks*. 2021. arXiv: 2104.08378.

[14] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. "QLoRA: Efficient Finetuning of Quantized LLMs". In: *Advances in Neural Information Processing Systems 36* (*NeurIPS*). 2023.

[15] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers". In: *International Conference on Learning Representations* (*ICLR*). 2023.

[16] W. Fedus, B. Zoph, and N. Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". In: *Journal of Machine Learning Research* 23.120 (2022), pp. 1–39.

[17] J. Bonwick. "The Slab Allocator: An Object-Caching Kernel Memory Allocator". In: *Proceedings of the USENIX Summer 1994 Technical Conference*. 1994, pp. 87–98.

[18] R. E. Wengert. "A Simple Automatic Derivative Evaluation Program". In: *Communications of the ACM* 7.8 (1964), pp. 463–464. DOI: 10.1145/355586.364791.

[19] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. "Automatic Differentiation in Machine Learning: a Survey". In: *Journal of Machine Learning Research* 18.153 (2018), pp. 1–43.

[20] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. "Feature Hashing for Large Scale Multitask Learning". In: *Proceedings of the 26th International Conference on Machine Learning* (*ICML*). 2009, pp. 1113–1120. DOI: 10.1145/1553374.1553516.

[21] J. L. Carter and M. N. Wegman. "Universal Classes of Hash Functions". In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 143–154. DOI: 10.1016/0022-0000(79)90044-8.

[22] R. Pagh and F. F. Rodler. "Cuckoo Hashing". In: *Journal of Algorithms* 51.2 (2004), pp. 122–144. DOI: 10.1016/j.jalgor.2003.12.002.

[23] T. Kudo and J. Richardson. "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing". In: *Proceedings of EMNLP 2018: System Demonstrations*. 2018, pp. 66–71.

[24] B. H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426. DOI: 10.1145/362686.362692.

[25] G. Cormode and S. Muthukrishnan. "An Improved Data Stream Summary: The Count-Min Sketch and its Applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58–75. DOI: 10.1016/j.jalgor.2003.12.001.

[26] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. "HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm". In: *Proceedings of the 2007 International Conference on Analysis of Algorithms* (*AofA*). DMTCS Proceedings. 2007. URL: https://hal.inria.fr/hal-00406166.

[27] A. Z. Broder. "On the Resemblance and Containment of Documents". In: *Proceedings of Compression and Complexity of Sequences* (*SEQUENCES 1997*). IEEE, 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

[28] P. Indyk and R. Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality". In: *Proceedings of the 30th Annual ACM Symposium on Theory of Computing* (*STOC*). 1998, pp. 604–613. DOI: 10.1145/276698.276876.

[29] K. Lee et al. "Deduplicating Training Data Makes Language Models Better". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics* (*ACL*). 2022, pp. 8424–8445.

[30] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data* (*SIGMOD*). 2018, pp. 489–504. DOI: 10.1145/3183713.3196909.

[31] R. Bayer and E. M. McCreight. "Organization and Maintenance of Large Ordered Indexes". In: *Acta Informatica* 1.3 (1972), pp. 173–189. DOI: 10.1007/BF00288683.

[32] P. Ferragina and G. Vinciguerra. "The PGM-index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds". In: *Proceedings of the VLDB Endowment* 13.8 (2020), pp. 1162–1175. DOI: 10.14778/3389133.3389135.

[33] P.-V. Khuong and P. Morin. "Array Layouts for Comparison-Based Searching". In: *ACM Journal of Experimental Algorithmics* 22 (2017), 1.3:1–1.3:39. DOI: 10.1145/3053370.

[34] E. Fredkin. "Trie Memory". In: *Communications of the ACM* 3.9 (1960), pp. 490–499. DOI: 10.1145/367390.367400.

[35] D. R. Morrison. "PATRICIA: Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *Journal of the ACM* 15.4 (1968), pp. 514–534. DOI: 10.1145/321479.321481.

[36] R. Sennrich, B. Haddow, and A. Birch. "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* (*Volume 1: Long Papers*). 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162.

[37] L. Zheng et al. "SGLang: Efficient Execution of Structured Language Model Programs". In: *Advances in Neural Information Processing Systems 37* (*NeurIPS*). 2024. arXiv: 2312.07104.

[38] B. T. Willard and R. Louf. *Efficient Guided Generation for Large Language Models*. 2023. arXiv: 2307.09702.

[39] A. V. Aho and M. J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search". In: *Communications of the ACM* 18.6 (1975), pp. 333–340. DOI: 10.1145/360825.360855.

[40] J. W. J. Williams. "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6 (1964), pp. 347–348. DOI: 10.1145/512274.512284.

[41] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. "The Curious Case of Neural Text Degeneration". In: *International Conference on Learning Representations* (*ICLR*). 2020.

[42] A. J. Walker. "An Efficient Method for Generating Discrete Random Variables with General Distributions". In: *ACM Transactions on Mathematical Software* 3.3 (1977), pp. 253–256. DOI: 10.1145/355744.355749.

[43] M. D. Vose. "A Linear Algorithm for Generating Random Numbers with a Given Distribution". In: *IEEE Transactions on Software Engineering* 17.9 (1991), pp. 972–975. DOI: 10.1109/32.92917.

[44] A. B. Kahn. "Topological Sorting of Large Networks". In: *Communications of the ACM* 5.11 (1962), pp. 558–562. DOI: 10.1145/368996.369025.

[45] R. E. Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

[46] T. Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 578–594.

[47] R. Weber, H.-J. Schek, and S. Blott. "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces". In: *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*. 1998, pp. 194–205.

[48] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517. DOI: 10.1145/361002.361007.

[49] W. Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Communications of the ACM* 33.6 (1990), pp. 668–676. DOI: 10.1145/78973.78977.

[50] Y. A. Malkov and D. A. Yashunin. "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.4 (2020), pp. 824–836. DOI: 10.1109/TPAMI.2018.2889473.

[51] S. J. Subramanya, F. Devvrit, R. Kadekodi, R. Krishnaswamy, and H. V. Simhadri. "DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node". In: *Advances in Neural Information Processing Systems 32 (NeurIPS)*. 2019.

[52] H. Jégou, M. Douze, and C. Schmid. "Product Quantization for Nearest Neighbor Search". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57.

[53] S. P. Lloyd. "Least Squares Quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489.

[54] R. Guo et al. "Accelerating Large-Scale Inference with Anisotropic Vector Quantization". In: *Proceedings of the 37th International Conference on Machine Learning (ICML)*, *PMLR 119*. 2020, pp. 3887–3896.

[55] U. Manber and G. Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *SIAM Journal on Computing* 22.5 (1993), pp. 935–948. DOI: 10.1137/0222058.

[56] J. Liu, S. Min, L. Zettlemoyer, Y. Choi, and H. Hajishirzi. "Infini-gram: Scaling Unbounded *n*-gram Language Models to a Trillion Tokens". In: *First Conference on Language Modeling (COLM)*. 2024. arXiv: 2401.17377.

[57] A. Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems 30 (NeurIPS)*. 2017, pp. 5998–6008. arXiv: 1706.03762.

[58] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Advances in Neural Information Processing Systems 35 (NeurIPS)*. 2022. arXiv: 2205.14135.

[59] M. Milakov and N. Gimelshein. *Online Normalizer Calculation for Softmax*. 2018. arXiv: 1805.02867.

[60] Z. Zhang et al. "H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models". In: *Advances in Neural Information Processing Systems 36 (NeurIPS)*. 2023.

[61] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. "Efficient Streaming Language Models with Attention Sinks". In: *International Conference on Learning Representations (ICLR)*. 2024. arXiv: 2309.17453.

[62] A. Gu and T. Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2023. arXiv: 2312.00752.

[63] R. Durstenfeld. "Algorithm 235: Random Permutation". In: *Communications of the ACM* 7.7 (1964), p. 420. DOI: 10.1145/364520.364540.

[64] P. Patarasuk and X. Yuan. "Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations". In: *Journal of Parallel and Distributed Computing* 69.2 (2009), pp. 117–124. DOI: 10.1016/j.jpdc.2008.09.002.

[65] R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology (CRYPTO '87)*, *LNCS 293*. Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32.

[66] M. Stonebraker et al. "C-Store: A Column-oriented DBMS". In: *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. 2005, pp. 553–564.

[67] J. Bentley. *Programming Pearls*. 2nd ed. Addison-Wesley, 2000. ISBN: 0-201-65788-0.

찾아보기

자료 구조는 답이 아니라 가정의 묶음이다.

교실은 여전히 연결 리스트와 이진 탐색 트리로 시작한다. 추론 서버와 벡터 검색 시스템을 열어 보면 그 구조들은 그 모습 그대로 있지 않다. 사라진 것이 있고, 이름만 남고 속이 바뀐 것이 있고, 교과서에 없던 구조가 한가운데를 차지한 것이 있다.

이 책은 전통 구조를 정의와 불변식으로 다시 세우고, 어느 가정에서 깨지는지를 직접 잰 숫자로 보이고, 깨진 자리를 메운 현대 구조를 같은 밀도로 세운다. 배열에서 페이지드 어텐션까지, 해시 테이블에서 임베딩 테이블까지, 스킵 리스트에서 HNSW 까지.

모든 코드는 실제로 돌아가고 모든 숫자는 실제로 잰 것이다.