CS 270 Lecture Notes

Alec Li

Spring 2023 — Professor Jelani Nelson

Contents

1 Introduction, Single-Source Shortest Paths **4**
1.1 Topics in the Course 4
1.2 Single-Source Shortest Paths (SSSP) 4
1.2.1 Price Functions 5
1.2.2 Price Functions with Scaling 5

2 BNW wrap-up, Max-Flow **8**
2.1 BNW Wrap-up 8
2.1.1 `FixDAGEdges` 9
2.1.2 `ElimNeg` 9
2.1.3 Expected number of edges in E' 10
2.2 Max Flow 10

3 Blocking Flow, Link-Cut Trees **11**
3.1 Blocking Flows 12
3.1.1 Finding Blocking Flows 14
3.2 Speeding Up With Data Structures 15

4 Splay Trees **15**
4.1 Splay Tree Properties 16
4.2 Splay Trees 17

5 Link-cut Trees **19**
5.1 Link-cut Trees 19

6 Link-cut Tree Wrap-up, Flow Wrap-up **22**
6.1 Link-cut Tree Analysis (cont.) 22
6.2 Min Cost Max Flow 24
6.2.1 Algorithms 24
6.3 Special Cases 24
6.3.1 Nonnegative Costs, Unit Capacity 25
6.3.2 Negative Costs, Unit Capacity 25

7 Heaps **26**
7.1 Runtime of Heap Algorithms 26
7.2 Binomial Heap 27
7.2.1 Analysis 28
7.3 Fibonacci Heaps 28
7.3.1 Minimum Tree Sizes 29
7.3.2 Analysis 29

8 Van Emde Boas Trees, y-fast Trees **30**
8.1 Predecessor problem 30

8.2 vEB Trees 30
8.3 x-fast Tries 32

9 y-fast Tries, Fusion Trees 33
9.1 y-fast Tries 33
9.2 Fusion Trees 34
9.2.1 Sketching 34
9.2.2 Word-level Parallelism 35
9.3 Approximate Sketching 36

10 Fusion Tree Wrap-up, Hashing 37
10.1 MSSB in Constant Time 37
10.2 Hashing 38
10.2.1 Hashing with Chaining 38
10.2.2 Linear Probing 39
10.2.3 Load Balancing 39

11 k-wise Independent Families, Linear Probing 41
11.1 Hash Families 41
11.2 Linear Probing 43

12 Linear Probing with k-wise Independence, Approximate Membership 45
12.1 Linear Probing with k-wise Independence 45
12.1.1 Linear Probing with 7-wise Independence 45
12.2 Linear Probing with 5-wise Independence 48
12.3 Approximate Membership 48

13 Approximate Membership, Cuckoo Hashing, Power of Two Choices 49
13.1 Bloom Filters 49
13.2 Cuckoo Hashing 50
13.2.1 Cuckoo Hashing Analysis 51
13.3 Preview of Power of 2 Choices 54

14 Online Algorithms 54
14.1 Power of 2 Choices 54
14.2 Online Algorithms 55
14.2.1 Pot of Gold 55
14.2.2 Ski Rental 55
14.3 List Update Problem 56

15 Paging, Resource Augmentation 57
15.1 Deterministic Paging 57
15.2 Randomized Paging 59

16 k-Server Problem, LP Duality, Online Primal/Dual 62
16.1 k-server problem 62
16.2 LPs and LP Duality 62
16.3 Online Primal-Dual 63
16.3.1 Ski Rental (with primal/dual) 64
16.3.2 Ski Rental (randomized) 65

17 Online Primal/Dual (Set Cover), Approximation Algorithms 66
17.1 Online Set Cover 66
17.1.1 Primal Analysis 67
17.1.2 Primal/Dual Analysis 67
17.1.3 Integral Solutions 68

17.2 Approximation Algorithms . . . 69

18 Approximation Algorithms, Improving Approximation Ratios, PTAS, FPTAS 69
18.1 Unweighted Vertex Cover . . . 69
18.2 Barriers to Improving Approximation Ratios . . . 70
18.2.1 Vertex Cover . . . 70
18.2.2 Set Cover . . . 70
18.3 Polynomial Time Approximation Schemes . . . 71
18.3.1 Knapsack . . . 71

19 PTAS, FPTAS, FPRAS 72
19.1 Knapsack Approximations . . . 72
19.1.1 Knapsack PTAS . . . 73
19.1.2 Knapsack FPTAS . . . 73
19.2 FPRAS . . . 74

20 SDP, Approximations in Streaming 76
20.1 SDP Relaxation: Max Cut . . . 76
20.1.1 Semidefinite Programming (SDP) . . . 76
20.1.2 Max Cut as an SDP . . . 77
20.1.3 Max-Cut Integrality Gap . . . 77
20.1.4 Max-Cut Rounding . . . 78
20.2 Streaming Algorithms . . . 79
20.2.1 Count Distinct (idealized) . . . 79
20.2.2 Deterministic and Exact Count Distinct . . . 80

21 Linear Programming: Simplex Method 81
21.1 Simplex Method . . . 81
21.1.1 Finding a starting vertex . . . 82
21.1.2 OPT is at a vertex . . . 82
21.1.3 Moving to neighboring vertices . . . 84

22 Linear Programming: Strong Duality, Complementary Slackness, Ellipsoid Method 84
22.1 Strong Duality . . . 85
22.2 Simplex Runtime . . . 86
22.3 Complementary Slackness . . . 86
22.4 Ellipsoid Algorithm . . . 86
22.4.1 Optimizing LPs using Ellipsoid . . . 87

23 Interior Point Method, Newton's Method, Gradient Descent 88
23.1 Starting Point . . . 89
23.2 Number of iterations . . . 89
23.3 Continuous Optimization . . . 90
23.3.1 First-order Methods . . . 90
23.3.2 Newton's Method . . . 91

24 Multicalibration and Fairness in Prediction 91
24.1 Setup . . . 92
24.2 Calibration . . . 93
24.3 Multicalibration . . . 94
24.4 Multicalibration Boosting . . . 94

1/17/2023

Lecture 1

Introduction, Single-Source Shortest Paths

1.1 Topics in the Course

- Advanced data structures

  For example, data structures in the word RAM model: using the word RAM to design faster algorithms using the full instruction set of the machine, getting better than $O(n \log n)$ time (that's the limit for comparison sorts). Additionally, we'll also cover dictionaries (hashing), amortization, splay trees.

- Graph algorithms
  - Shortest paths (this is what we'll cover today)
  - More advanced max flow algorithms (more advanced than what is covered in CS170), along with min-cost max-flow algorithms

- Online algorithms

  With online algorithms, we make irrevocable decisions online. The ski rental problem is a classic example of this; if you're going skiing, each day you'll be making the decision of whether you keep renting skis or whether you finally end up buying the skis.

  In particular, we'll cover primal-dual algorithms

- Approximate algorithms
  - Through the primal-dual
  - Through LP/SDP rounding

- "Fast" exponential-time algorithms

  Usually exponential time algorithms are bad, but it's clear that $O(1.5^n)$ is better than $O(2^n)$; the aim is to get the fastest exponential time algorithm we can.

- Hardness within P

  Similar to NP-hardness, hardness within P is more fine-grained; for example, if we'd like to prove that for some problem, no linear or quadratic time algorithm exists, etc.

1.2 Single-Source Shortest Paths (SSSP)

Today, we'll be looking at an algorithm for SSSP with possibly negative edge weights.

The input is a directed graph $G = (V, E, w)$ where w is a weight function $w : E \to \mathbb{R}$, along with a source vertex $s \in V$.

The output should be all of the shortest paths (or the lengths of the shortest paths, or the shortest path tree) from s to all other vertices in the graph.

On the note of shortest-path trees, algorithms like Dijkstra's will give us the shortest path tree; whenever we pop a vertex off from the min-heap, we're convinced that we have the shortest path to this vertex. All we then need to remember is the most recent vertex that caused a decrease-key for this vertex; this would be the parent vertex in the tree.

Some previous algorithms:

- Dijkstra [Dij59]: $O(m + n \log n)$, with the assumption that $w \geq 0$; we can do slightly better with fibonacci heaps.
- Bellman-Ford [Bel58; For56]: $O(mn)$, via a DP algorithm with no assumptions on w. If there is a negative cycle in the graph, it'll output that it detects one, and otherwise it'll output the shortest-path tree.

Today, we'll cover an algorithm due to Bernstein–Nanongkai–Wulff-Nilsen [BNW22], with running time $\tilde{O}(m \log W)$.

Here, we'll assume that for all edges, $w(e)$ is an integer in $\{-W, \dots, W\}$; the notation $\tilde{O}(f)$ means $f \cdot \text{polylog}(f)$.

We'll talk about how this algorithm works.

1.2.1 Price Functions

The starting point is price functions and scaling; both of these topics are things we'll talk about later as well.

A price function is a function $\varphi: V \to \mathbb{R}$, and with these prices, we'll define a price-reduced weight function

$$w_\varphi(e) := w(e) + \varphi(u) - \varphi(v),$$

where $e = (u, v)$.

A couple observations:

- For all cycles C, the total weight $w(C)$ of the cycle is the same no matter what the price function is, due to telescoping; $w(C) = w_\varphi(C)$ regardless of φ.
- For all paths $P = (v_0, \dots, v_r)$, the total weight on that path under φ is $w_\varphi(P) = w(P) + \varphi(v_0) - \varphi(v_r)$, again due to telescoping.

The goal here is to find a price function such that under the price function, all the weights are nonnegative (i.e. we want $w_\varphi(e) \geq 0$ for all e). Here, all the shortest paths in G_φ will still be shortest paths in G and vice versa. This means that we can just run Dijkstra's algorithm on G under the new weights w_φ.

This invariance in shortest paths is because any path from s to v will always be its original weight w_φ, offset by some fixed constant; any path from s to v will always be offset by this same constant.

Firstly, we want to show that φ can actually exist; in fact, such a φ exists if and only if G has no negative cycles. In implementation, the program will run into an infinite loop, but we know the expected runtime of the algorithm so we can cut it off if it runs too long, indicating that there exists a negative cycle.

Lemma 1.1

The price function φ exists if and only if there are no negative cycles in G.

Proof. ($\implies$) If we know that such a φ exists, we want to show that there must not be any negative cycles. This follows immediately from our first observation. For any cycle C, $w_\varphi(C) \geq 0$, since every $w_\varphi(e) \geq 0$ for $e \in C$. However, $w(C) = w_\varphi(C)$ by the first observation, so this direction ins proven.

($\impliedby$) Suppose no negative cycles exist.

Suppose we let $\varphi(v)$ be the distance function $d(s, v)$ (i.e. the weight of the shortest path from s to v); by the triangle inequality, we know that $d(s, v) \leq d(s, u) + w(u, v)$. Rearranging, we arrive at the conclusion that $0 \leq d(s, u) - d(s, v) + w(u, v) = w_\varphi(u, v)$, showing that $w_\varphi(e)$ is nonnegative for all edges e. □

1.2.2 Price Functions with Scaling

The combination of price functions and scaling originates from Goldberg in 1995 [Gol95]. If you have an algorithm that finds a good price function (i.e. one that makes all the weights nonnegative) in time $T(m)$ for the special case of $w(e) \geq -1$ for all $e \in E$, then you can solve the general case (where $w(e) \in \{-W, \dots, W\}$) in time $O(T(m) \cdot \log(W))$.

Proof. Let us round W up to the nearest power of 2, and let this rounded $W = 2^k$. The following will be a recursive algorithm on k.

In the base case, for $k = 0$, we're done, since all edge weights are already ≥ -1.

In the recursive step, for $k > 0$, we define a new weight function $\hat{w}(e) := \left\lceil \frac{w(e)}{2} \right\rceil$, and we define $\hat{G} = (V, E, \hat{w})$.

We then recursively call the same function on $\hat{G}$ to find a feasible price function on $\hat{G}$ that makes all weights non-negative.

In pseudocode, suppose we have an algorithm `solve` that finds a good price function when $w(e) \geq -1$.

```
def A(G = (V, E, W), k):
    if k == 0: return solve(G)
    else:
        w_hat(e) = ceil(w(e) / 2)
        phi_hat = A((V, E, w_hat), k-1)
        phi = 2 * phi_hat
        return solve(G_phi)
```

We call `solve` $\log W$ times, and each call takes $T(m)$ time by assumption, so in total we have runtime $O(T(m)\log W)$.

All that is left is to show that the graph G_φ satisfies $w(e) \geq -1$.

We know that $w(e) \geq 2 \cdot \left\lceil \frac{w(e)}{2} \right\rceil - 1$; if it is odd, then the two sides are equal, but if it is even, then the LHS is strictly bigger. Replacing the ceiling, we have that $w(e) \geq 2 \cdot \hat{w}(e) - 1$.

Suppose for an edge $e = (u, v)$, we look at $w_\varphi(e)$; we want to show that this is ≥ -1 everywhere:

$$\begin{aligned} w_\varphi(e) &= w(e) + \varphi(u) - \varphi(v) \\ &\geq (2\hat{w}(e) - 1) + 2\hat{\varphi}(u) - 2\hat{\varphi}(v) \\ &= 2(\hat{w}(e) + \hat{\varphi}(u) - \hat{\varphi}(v)) - 1 \end{aligned}$$

Since $\hat{\varphi}$ comes from the recursive call, we assume that this is a price function that is non-negative for all edges; multiplying by 2 doesn't change the fact that it is non-negative, and we subtract 1 at the end, giving us the $w_\varphi(e) \geq -1$, as desired. □

(There are a lot of similar scaling ideas for other kinds of problems; we'll see that in this course. This notion of price functions also comes from duality in LPs—if we write this as an LP, a feasible solution to the dual is actually the price function that we want.)

As such, from now on, we'll always assume that $w(e) \in \{-1, \dots, W\}$ and all weights are integers. We'll focus on G_s, where s is a new dummy vertex which has a weight-0 edge to everyone else (we can't use s in any shortest path, so it's equivalent to find a price function for G_S). The source will also be this new dummy s.

For ease of notation, we'll let G be G_s. The ingredients are as follows:

1. Subroutine `LowDiamDecomp` ("low diameter decomposition"), called `LDD` from now on.

   This algorithm assumes that $w \geq 0$, and `LDD` takes in the graph G with parameter D, and outputs a set of edges $E^{\text{rem}} \subseteq E$ ("rem" for remainder) such that:

   - Every SCC of $G \setminus E^{\text{rem}}$ has weak diameter $\leq D$.

     That is, for all vertices u, v in the same SCC, the distance in G from u to v is $\leq D$.

   - For all edges e, $\mathbb{P}(e \in E^{\text{rem}}) \leq O(\frac{w(e)\cdot\log^2 n}{D} + n^{-10})$; the n^{-10} is just some constant (this can be changed to give different numbers).

   This subroutine runs in $\tilde{O}(m)$ time (nearly-linear time).

2. `FixDAGEdges` on a DAG G:

   Given a DAG, we want to find a price function for the DAG; this is a very simple problem; we can just let $\varphi(v)$ be the distance from s (our added vertex with weight-0 edges) to v, which can be computed in linear time if we have a DAG. (In context, the actual algorithm uses this to find a φ that makes $w_\varphi(e) \geq 0$ for edges going across SCCs.)

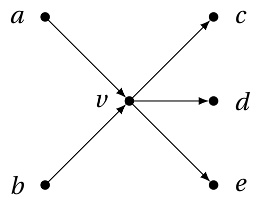
3. `ElimNeg` on a graph G:

   This subroutine finds a good φ in time $O(\log n \cdot \sum_V (1+\eta_G(v)))$. Here, $\eta_G(v)$ is the number of negative edges on the shortest path to v (if there are multiple, we take the path with the smallest number of negative edges on it).

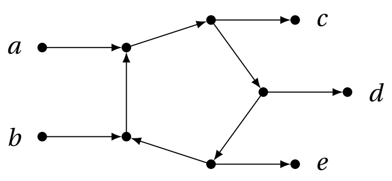
   This algorithm assumes that all in-degrees and out-degrees are $O(1)$. In fact, we can assume WLOG that all degrees are $O(1)$ (ex. ≤ 3) in the original graph.

   In the case that a lot of edges on the shortest path are negative, then this would give a bad runtime; however, if the η values are really small, then this would give a very good runtime.

   As an example, suppose that we have the following snippet of the graph:



   We can convert v into a cycle with $\deg(v)$ vertices, where all edges in the cycle have weight 0:



   The only drawback of this is that we'll blow up the number of vertices in the graph.

4. `ScaleDown`, taking G, Δ, and B:

   This algorithm assumes that $\eta(G) := \max_{v\in V} \eta(G^B)$ is $\leq \Delta$ and that for all edges e, $w(e) \geq -2B$, and returns φ such that $w_\varphi(e) \geq -B$ for all e. Here, G^B is the graph with all weights with the new weight function

$$w^B(e) := \begin{cases} w(e) & \text{if } w(e) \geq 0 \\ w(e)+B & \text{if } w(e) < 0 \end{cases}.$$

`LDD`, `FixDAGEdges`, and `Elimneg` are useful to implement `ScaleDown`, and we only use `ScaleDown` in the main algorithm.

For the main algorithm, we have the following:

- Use `LDD` to decompose the algorithm in to SCCs, where each SCC has a small η value
- Fix the edges that cross the DAG of SCCs, and then use `ElimNeg` to fix the edges inside the SCCs

```
def main(G = (V, E, w)):
    B = 2n rounded up to nearest power of 2
    $\overline{w}$ = B * w
    $\varphi_0$ = 0
    for i = 1 to log_2(B):
        $\psi_i$ = ScaleDown($G_{\varphi_{i-1}}$, $\Delta = n$, B / 2^i)
        $\varphi_i$ = $\varphi_{i-1}$ + $\psi_i$
        w* = $\overline{w}_{\varphi \log B}$ + 1
```

All the weights are integers, and we multiply them by $2n$; this means that if one path is shorter than another path, it's shorter by at least $2n$; this means all weights are at least $-2n$. Running `ScaleDown` on the graph with the blown up weights, we now have a case where all weights $w_\varphi(e) \geq -n$. We keep running `ScaleDown` to go to $\geq -\frac{n}{2}$, to $\geq -\frac{n}{4}$, etc. until we get to ≥ -1.

Remember that this is a blown-up graph, where each weight is blown up by B; if we divide by B again, now each weight is at least $-\frac{1}{2n}$. If we add $-\frac{1}{2n}$ to each weight, this makes every edge weight nonnegative, while still preserving the relative ordering of paths (any given path can only ever be by $n \cdot \frac{1}{2n} = \frac{1}{2}$, and all weights are integers, so this doesn't affect the ordering.)

1/19/2023

Lecture 2

BNW wrap-up, Max-Flow

2.1 BNW Wrap-up

Continuing on from last time, here's the pseudocode for `ScaleDown`:

```
def ScaleDown(G, $\delta$, B):
    if $\Delta \leq 2$: set $\varphi_2$ = 0, goto PHASE 3
    d = $\Delta$ / 2
    $w^B_{\geq 0}$(e) = max(0, $w^B$(e)) // all edges now nonnegative

    // PHASE 0
    E$'$ = LDD($G^B_{\geq 0}$, dB)

    // PHASE 1
    H = $\bigcup_i$ G[$V_i$] // where $V_i$'s are SCC's of G \ E'
    $\varphi_1$ = ScaleDown(H, $\Delta$ / 2, B) // claim: $\eta(H^B) \leq \Delta/2$

    // PHASE 2
    $\psi$ = FixDAGEdges($G^B_{\varphi_1}$ \ E$'$)
    $\phi_2$ = $\varphi_1$ + $\psi$

    // PHASE 3
    $\psi'$ = ElimNeg($G^B_{\varphi_2}$)

    return $\varphi_2$ + $\psi'$
```

In the base case, if every no vertex has an η value greater than 2, the runtime bound of `ElimNeg` is nearly linear, and we can just run it. Otherwise, we have to do some work (as `ElimNeg` is too slow).

Firstly, we define $w^B_{\geq 0}$ to have weights that are nonnegative, and run `LDD`, to give us an edge set E' such that each SCC in $G \setminus E'$ has weak diameter $\leq dB$.

Here, we claim that for all v, $\mathbb{E}\left[\left|P_{G^B}(v) \cap E'\right|\right] = O(\log^2 n)$; $P_{G^B}(v)$ is the path with the path with the smallest number of negative edges. Here, what we're saying is that in expectation, the number of negative edges in the shortest path to v that are also contained in E' is on the order of $O(\log^2 n)$.

Recall that `LDD` gives us E', such that if we remove these edges, and perform a SCC decomposition, any two vertices in a SCC are close together in G (possibly using edges in E').

In phase 1, H is the graph of all of the SCCs (without edges in E' or edges between SCCs), and we perform `ScaleDown` on H. That is, each edge inside of SCCs are now $\geq -B$. Then, in phase 2, we fix edges *between* SCCs, making those weights $\geq -B$.

Now, the only things left to fix are the edges in E'; this is what `ElimNeg` does in phase 3, as we've already fixed all the edges not in E'.

2.1.1 `FixDAGEdges`

We know that all edges inside of each SCC are negative already. We topologically sort the SCCs, and process them in topological order. All vertices in the source get set to the same price (ex. 0).

If we look at the edges between two SCCs (a previous processed SCC and a new SCC), we have

$$w_\varphi(u,v) = w(u,v) + \varphi(u) - \varphi(v),$$

where u is in the processed SCC, and v is in the new SCC we're currently processing. Since $w(u,v)$ and $\varphi(u)$ are both fixed already, we just need to set $\varphi(v)$ so that $w_\varphi(u,v) \ge 0$. There's always a threshold for $\varphi(v)$ (as long as it's negative enough) that makes $w_\varphi(u,v)$ nonnegative:

$$w(u,v) + \varphi(u) - \varphi(v) \ge 0 \implies \varphi(v) \le w(u,v) + \varphi(u).$$

We then take the minimum of all of these thresholds, and set this to be the price of all the vertices in the new SCC.

2.1.2 `ElimNeg`

We'll show a weaker version, with runtime $O(m\eta(G)\log n)$; to show the stronger version necessary for the algorithm, we need to do a little more, but we'll be omitting that here.

This algorithm is basically just a typical DP algorithm.

We define $h(v,k)$ to be the shortest path length from s to v using at most k negative edges, where the last edge *must* be negative. We also define $g(v,k)$ to be the shortest path length to v using at most k negative edges (here, the last edge could be negative or nonnegative).

We can then define the recurrence relation:

$$h(v,k) = \begin{cases} +\infty & \text{if } k = 0 \\ \min\left(h(v,k-1), \min\limits_{\substack{u:(u,v)\in E,\\ (u,v) \text{ is negative}}} g(u,k-1) + w(u,v)\right) & \text{otherwise} \end{cases}$$

If $k = 0$, we aren't allowed to use negative edges at all, so there's no way to have the last edge be negative. Otherwise, we have two cases; if we don't need k negative edges, we can just recurse on $h(v,k-1)$. If we do need k negative edges, then we can look at all possible last edges (u,v), which must be negative, and compute the shortest distance from s to u using $g(u,k-1)$. Here, we don't need the second-to-last edge to be negative, but we can only use $k-1$ negative edges.

With g, we have

$$g(v,k) = \min\left(g(v,k-1), \min_{u\in V}\left(h(u,k) + d^+(u,v)\right)\right).$$

Here, either the last edge is negative, or the last edge is nonnegative; in the second case, there must be some last negative edge the path goes through; we can then use h to find this shortest path, iterating through all possible last vertices u.

The runtime of h is proportional to m, since with memoization we'll be iterating through all edges. The runtime of g is a little more complicated; naively, it seems like we need to perform all-pairs shortest paths, as u and v are both arbitrary, and we need to know $d^+(u,v)$ for all such u and v.

However, we can create a new graph (Fig. 2.1) with a new vertex α, with edges to each v_i with weight $h(v_i,k)$. In the rest of the graph, we keep all nonnegative edges between v_i's.

Here, notice that the shortest path from α to v is exactly

$$\min_{u\in V} h(u,k) + d^+(u,v).$$

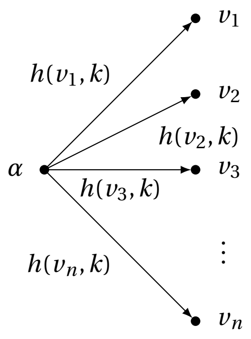


Figure 2.1: New graph used to compute $g(v, k)$; the rest of the graph is omitted

For each u, we're essentially enforcing that the first edge we traverse is (α, u), with cost $h(u, k)$, and we find the shortest path from u to v in the rest of the graph (recall we removed all of the negative edges). The minimum across all u is the shortest path from α to v.

Since we've removed all of the negative edges in the graph among $v_1, \dots, v_n$, we can just run Dijkstra's to compute this shortest path distance. However, the first edge could be negative, but we claim that Dijkstra's still works if this is the case.

2.1.3 Expected number of edges in E'

We'll now prove the claim that $\mathbb{E}[|P_{G^B}(v) \cap E'|] = O(\log^2 n)$.

We know that

$$w^B_{\geq 0}(P_{G^B}(v)) \leq w^B(P_{G^B}((v)) + B \cdot \left|P_{G^B}(v) \cap E^{\text{neg}})\right|.$$

Here, we can observe that taking $\max(0, w^B(e))$ means that we're only adding at most B to each negative edge in $P_{G^B}(v)$. We know that $\left|P_{G^B}(v) \cap E^{\text{neg}}\right| \leq \eta_{G^B}(v)$, and that $w^B(P_{G^B}(v)) \leq 0$ (since we can always take the path from s to v directly, with weight 0). This means that $w^B_{\geq 0}(P_{G^B}(v)) \leq \eta_{G^B}(v) \cdot B$.

Looking at the expectation now, we have

$$\begin{aligned}
\mathbb{E}[|P_{G^B}(v) \cap E'|] &= \mathbb{E}\left[\sum_{e \in P_{G^B}(v)} \mathbb{1}e \in E'\right] \\
&= \sum_e \mathbb{P}(e \in E') \\
&\leq O\left(\frac{w^B_{\geq 0}(P_{G^B}(v) \cdot \log^2 n)}{D} + n^{-9}\right) \\
&\leq O\left(\frac{\eta_{G^B}(v) \cdot B \cdot \log^2 n}{D} + n^{-9}\right) \\
&\leq O\left(\frac{\Delta \cdot B \cdot \log^2 n}{dB} + n^{-9}\right) \\
&\leq O\left(2 \log^2 n + n^{-9}\right) \\
&= O\left(\log^2 n\right)
\end{aligned}$$

2.2 Max Flow

With max-flow, we're given a capacitated graph $G = (V, E)$, where each edge e has some capacity u_e. We're also given a source vertex $s \in V$ and a sink vertex $t \in V$. The goal is to route as much flow from the source to the sink while obeying the capacity constraints.

That is, we want to find a flow $f \in \mathbb{R}^m$ from s to t maximizing $\text{val}(f)$. Here, we define $\text{val}(f) := \sum_{e=(s,\cdot)} f_e$.

Additionally, this flow needs to be feasible:

1. (obeys capacity constraints) $\forall e \in E$, $f_e \le u_e$
2. (nonnegative flow) $\forall e$, $f_e \ge 0$
3. (flow in = flow out) $\forall v \notin \{s,t\}$, $\sum_{e=(v,\cdot)} f_e = \sum_{e=(\cdot,v)} f_e$

We've already seen algorithms like Ford-Fulkerson for computing the max-flow; we want to do better.

Recall that Ford-Fulkerson [FF56] has runtime $O(m \cdot f^*)$, where f^* is the max flow value, assuming all capacities are integers (i.e. $u_e \in \{1, 2, \dots, U\}$).

This isn't a very good runtime, since if all edges have capacity U, the max flow could be as much as nU, so the runtime becomes $O(mnU)$.

With scaling, we can make Ford-Fulkerson run in $O(m^2 \log U)$. Since capacities are integers, each can be written as $\log U$ bit integers (padded with a leading 0). We can then start at the most-significant bit of the capacity (here, always a 0) and find the max-flow with these capacities (for the MSB, this is the zero flow).

Next, we double the flow, and add the next significant bit of the capacity. Here, when doubling the flow, we'll always still be obeying the capacity constraints, but when we potentially add 1 for the next bit, this flow may not be a max flow anymore.

However, in the residual graph, the max flow value is only m, as each edge has at most 1 more capacity compared to the previous iteration. This means that Ford-Fulkerson on this residual graph gives us a runtime $O(m^2)$. Since we iterate through all $\log U$ bits of the capacities, the total runtime is now $O(m^2 \log U)$, which is an exponentially better dependence on U.

1/24/2023

Lecture 3

Blocking Flow, Link-Cut Trees

There are two types of efficient algorithms currently to solve max-flow (assuming all capacities are integers in $\{1, \dots, U\}$):

- Pseudopolynomial, i.e. $\operatorname{poly}(m, n, U)$; these aren't actually polynomial-time algorithms
- Weakly polynomial, i.e. $\operatorname{poly}(m, n, \log U)$
- Strongly polynomial, i.e. $\operatorname{poly}(m, n)$, notably, there is no dependence on U.

  A note here is that reading in the capacities does take $\log U$ time, but there's an implicit assumption on the computational model (namely, the word RAM model)—we can read numbers in the input in constant time, we can add numbers in constant time, etc.

Ford–Fulkerson [FF56] has runtime $O(m \cdot f^*)$, and with the bound $f^* \le mU$, this gives $O(m^2 U)$, which is pseudopolynomial. Ford–Fulkerson with scaling, which we talked about last time, has runtime $O(m^2 \log U)$, which is weakly polynomial.

Blocking flow (which we'll talk about today), has runtime $O(mn^2)$, or $O(mn \log n)$ using link-cut trees, which are both strongly polynomial. Edmonds–Karp [EK72] is also strongly polynomial (recall that this is just Ford–Fulkerson but using BFS instead of DFS), as it has runtime $O(m^2 n)$.

Before we talk about blocking flows, here's a little more on the history of flow algorithms.

The best strongly polynomial algorithm we have is $O(mn)$, due to Orlin in STOC 2013 [Orl13], which isn't that much better than blocking flow with link-cut trees.

For a while, the best weakly polynomial algorithm we have is $O(m \cdot \min(\sqrt{m}, n^{\frac{2}{3}}) \log\left(\frac{n^2}{m}\right) \log U)$, due to Goldberg, Rao in 1998 [GR98]. This was improved to $\tilde{O}(m\sqrt{n} \log U)$, due to Sidford, Lee in FOCS 2014 [LS14]. Most recently, we have $m^{1+o(1)} \cdot \log U$ (the $m^{1+o(1)}$ is "almost linear"), due to Chen, Kyng, Liu, Peng, Gutenberg, and Sachdeva in 2022 [CKL+22].

The first two algorithms [Orl13; GR98] are graph algorithms, while the last two [LS14; CKL+22] are more continuous optimization and interior-point algorithms.

3.1 Blocking Flows

Blocking flows are very similar to Ford–Fulkerson; find a flow, augment using the flow, compute new residual graph, repeat. The only difference is that we're now finding a *blocking flow*.

Suppose we have some residual graph G_f.

Definition 3.1: Level Graph

For a residual graph G_f, the *level graph* only contains edges (v, w) from G_f such that $d_{G_f}(s, w) = d_{G_f}(s, v) + 1$. (Here, our graph is unweighted, so each edge has length 1 when computing distances. These distances can also be found using BFS.)

Definition 3.2: Admissible Edges, Admissible Paths

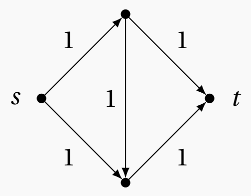
The edges in a level graph will be referred to as *admissible edges*, and *admissible paths* are paths that only contain admissible edges.

Definition 3.3: Blocking Flow

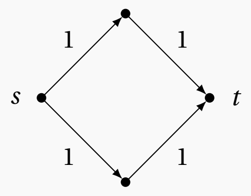
A *blocking flow* $\tilde{f}$ is a flow such that every admissible s-t path has at least one edge fully saturated by $\tilde{f}$.

Example 3.4

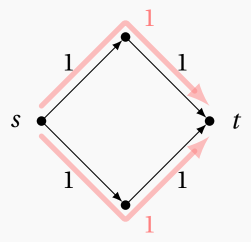
Consider the following graph:



The level graph only excludes the middle edge (as the top and bottom vertices are the same distance from s), giving us the following:

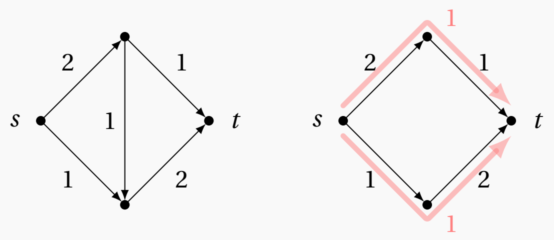


A blocking flow would then route 1 flow across the top, and 1 flow across the bottom:

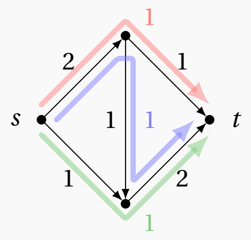


It turns out that a max-flow using only admissible edges must be a blocking flow; otherwise, if there was some edge that is not saturated, then we could just add more flow through that path.

However, a blocking flow is not always a max-flow; consider the following graph, where we tweak the capacities slightly:



We'd have the same blocking flow as before, but we could route another unit of flow down the middle of the graph as well, as a zig-zag.



Generally, a max-flow implies a blocking flow, but a blocking flow does not always imply a max-flow.

Crucially, the level graph always changes as we augment new blocking flows.

Lemma 3.5

Let L be the current level graph, and let L' be a new level graph after augmenting by a blocking flow.

We claim that $d_{L'}(s,t) > d_L(s,t)$.

Proof. Consider the level graph L:

In L, all edges are only allowed to go forward. In G, no edge can ever go forward that skip levels (otherwise, it'll be included in the graph), but it is possible for an edge to go backwards, or between vertices in the same level.

Recall that we constructed L' by finding a blocking flow in L, augmenting with this flow, and looking at the residual graph. Here, the only types of edges that can be in L' are:

- Edges $e \in L$
- $\text{rev}(e)$ for $e \in L$

- Backward edges in L; in particular, some of the forward edges could disappear after we augment with the blocking flow, making backward edges in L no longer backward in the resulting residual graph.

It should be clear that $d_{L'}(s,t) \geq d_L(s,t)$, since taking backward edges doesn't help, and we're only removing forward edges as we augment the graph with the blocking flow. The distances also must not be equal, since blocking flows must block the shortest path. This means that the distance must always be strictly increasing. □

This is an unweighted graph on n vertices, so $d(s,t)$ can never exceed $n-1$; this provides a limit to how many steps the algorithm can run. In particular, the total runtime is bounded by (number of iterations) × (time to find single blocking flow). This lemma implies that the number of iterations is at most n.

3.1.1 Finding Blocking Flows

Finding a blocking flow is basically DFS. For now, let us assume that all capacities are 1. In DFS, to get from s to t, we look at all the outgoing edges, and go to one of the neighboring vertices. If there are no outgoing edges, we go up one level of recursion.

There are three possible things we can do:

- Advance: From v, go to w such that $(v,w) \in L$.
- Retreat: No neighboring edges, so back to the predecessor of v.
- Augment: When we get to t, we add one unit of flow to every edge in the path. Since all capacities are 1, we delete all edges in the path, and go back to s.

A key point here is that even after we augment, we *do not* compute a new residual graph. We only do so after we've found our blocking flow.

The total runtime is bounded by $O(m)$ for each of advances, retreats, and augments, which means that in total it takes $O(m)$ to find the blocking flow, and $O(mn)$ to find the max flow in the unit capacity case.

Lemma 3.6

The runtime is also $O(m^{\frac{3}{2}})$; this gives another bound of $O(m \cdot \min(\sqrt{m}, n))$.

Proof. After d iterations of finding the blocking flow, since the paths from s to t are always increasing in length, each s-t path in the max flow in the residual graph must have distance at least d. Additionally, since all edges have unit capacity, each path must be disjoint and push a total of at most $\frac{m}{d}$ flow. Since each iteration will push at least one flow, we must have at most $\frac{m}{d}$ additional iterations left.

This means that we can have at most $\frac{m}{d}$ more iterations. Since d was arbitrary, we can take $d = \sqrt{m}$, giving us a bound of at most $2\sqrt{m}$ iterations. □

Lemma 3.7

With general capacities u_e, the time to find a single blocking flow is at most $O(mn)$.

Proof. We can again count the time it takes to do each of the three actions:

- Advances: $O(mn)$ time.
- Retreat: We can only ever retreat m times, so this takes $O(m)$ time.
- Augment: Whenever we do an augment, at least one edge in the path gets fully saturated. This means that we always remove at least one edge from the graph. This means that we can augment at most m times, and each augment can involve at most n vertices. This means that it takes at most $O(mn)$ time

for augments.

□

3.2 Speeding Up With Data Structures

Suppose we have a level graph, and we advance until we hit t. Since we saturate at least one edge, we then delete these saturated edges. In the next iteration, we may eventually get back to a vertex we already traversed, with edges all the way still to t.

Theoretically, if we save these path fragments in some data structure, we don't need to keep doing the DFS to t, as we already know an existing path to t. These path fragments to t end up being a tree rooted at t, which leads to the idea of link-cut trees to keep track of and manage these tree fragments.

1/26/2023

Lecture 4

Splay Trees

With the goal of deriving and motivating link-cut trees, we need to first cover splay trees, due to Sleator, Tarjan in 1985 [ST85b], which will be the primary focus of today.

Splay trees are similar to a binary search tree, which stores key-value pairs, where keys are comparable. Binary search trees can be thought of as the data structure that solves the dynamic dictionary problem. The dictionary problem requires the following methods:

- `query(k)`: return value associated with key `k`
- `insert(k, v)`: insert key `k` with value `v`
- `delete(k)`: delete key `k` along with its associated value

Optimal solutions include balanced binary search trees, red-black trees, etc., which do $O(\log n)$ work per operation in the worst case.

Splay trees use the *BST* model. At the start of every operation, our finger is on the root, and splay trees are binary trees as well; each node as at most 2 children, and the left child has smaller keys, and the right child has bigger keys.

At every time step, we can do the following, assuming we're currently at node x:

- Go to the child of node x
- Go to the parent of x
- If x is not the root, rotate x

Each of these things can be done in constant time (the first two are pointer follows, and rotation is a constant number of pointer changes. See Fig. 4.1 for an example; here, A, B, and C are sub-trees, and y is not necessarily the root. As a remark, splay trees connect to online algorithms. In online algorithms, we must make (irreversible) online decisions to satisfy requests in real-time. We don't know what will come in the future, but want to do the best we can given what we know now.

In the context of splay trees, we have a lot of operations coming in the future, but we want to choose what to do with the tree given our current knowledge of what is in the tree.

An open conjecture (dynamic optimality) is as follows: Let σ be a sequence of operations. We define $\mathrm{OPT}(\sigma)$ as the minimum cost to service σ in the BST model, given knowledge of σ ahead of time. Here, each of the three things we can do (go to child, go to parent, rotate) is of cost 1. The dynamic optimality conjecture says that for all σ, the cost of σ using splay trees (i.e. in the online case) is $O(\mathrm{OPT}(\sigma))$. That is, splay trees are dynamically optimal up to a constant factor, for any given sequence of operations σ.

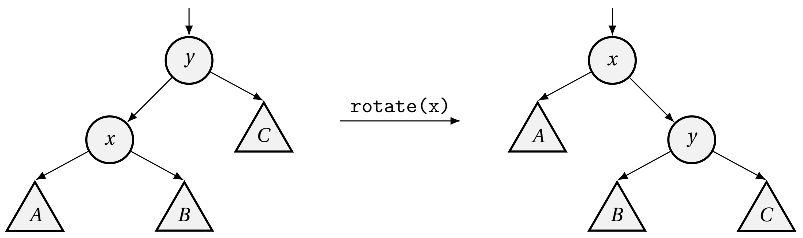


Figure 4.1: Example of a rotation

As an aside, it is trivial to have a data structure that guarantees $O(\mathrm{OPT}(\sigma)\cdot\log n)$, where n is the number of items in the data structure. This is because *any* balanced tree will give us this cost; traversing to any node takes $\log n$ time for each operation.

It is still an open problem of whether this conjecture is true. The best competitive ratio we know of (i.e. the ratio between the cost of σ and $\mathrm{OPT}(\sigma)$; the trivial case had a ratio of $\log n$) is called Tango trees, which gives a ratio of $O(\log\log n)$, due to Demaine, Harmon, Iacono, Patrascu in 2007 [DHI+07]).

4.1 Splay Tree Properties

Before we look at the details of splay trees, let us look at some properties of splay trees. (Everything from now in is in the context of using splay trees.)

Amortized Cost Firstly, the amortized cost per operation is $O(\log n)$.

Definition 4.1: Amortized Cost

The *amortized costs* of query/insert/delete are $t_q\,/\,t_I\,/\,t_d$ respectively if for all operation sequences σ, the total time to serve σ is bounded by

$$(\text{number of queries in }\sigma)\cdot t_q + (\text{number of insertions in }\sigma)\cdot t_I + (\text{number of deletions in }\sigma)\cdot t_d.$$

Intuitively, this means that on average (i.e. across the sequence σ), the runtime is $t_q\,/\,t_I\,/\,t_d$ for each of the operations.

Working Set Property Splay trees also have the *working set property*. For ease, suppose all the keys are $\{1,\ldots,n\}$. The working set properties is: for item j, let t_j be the number of operations since j was last inserted/queried. Then, the amortized cost of `query(j)` is $O(\log(t_j))$.

This means that an item that is queried recently is likely to be higher in the tree (i.e. roughly in the top $\log(t_j)$ levels in the tree).

Static Optimality Splay trees also satisfy *static optimality*.

With static optimality, suppose we have the dictionary problem, but with the additional information of how many times each item is queried (but not when they're queried). The goal is to provide a static data structure (i.e. we are not allowed to rotate after each operation) that optimizes the cost of servicing these operations. If every item is queried the same number of times, then a full binary tree would be optimal; however, if some keys are queried a lot more than others, then perhaps an unbalanced tree will be more optimal, allowing us to service the most frequent items faster. It turns out that we can find this optimal tree through dynamic programming.

Going back to static optimality, suppose the optimal tree T puts item i at level ℓ_i in T. Then, the amortized cost of `query(i)` in the splay tree is $O(\ell_i + 1$.

In this static case, the optimal tree T is only optimal with respect to the frequency distribution of the queries; it does not take order into account. This is why static optimality is different from dynamic optimality, where we must also optimize with order in mind.

Static Finger Splay trees are also optimal with a *static finger*. Here, we choose an item f and stick with it. Then, the amortized cost of querying i is $O(\log(|f - i + 1|))$.

Dynamic Finger Splay trees are also optimal with a *dynamic finger* (the proof of this is more than 100 pages long). Here, suppose the sequence of queries is $x_1, x_2, \dots, x_m$. Then, the amortized cost of querying the ith item is $O(\log|x_i - x_{i-1} + 1|)$. Here, we constantly move our finger to the item that was queried last.

4.2 Splay Trees

Here are the implementation details for each operation in a splay tree:

- `query(x)`: For a query in a splay tree for item x, we follow the child pointers down to x, and then call `splay(x)`, which essentially rotates x until it is the root.
- `insert(x)`: We follow child pointers and insert x in its corresponding position, and then call `splay(x)`
- `delete(x)`: We follow child pointers until we get to x, and call `splay(x)`. This makes x the root of the tree, and now we delete x. This creates two separate subtrees, and we take the largest child in A and call `splay` on this child; this brings it up to the root. Then we do some magic to make it a child of B.

Now, let us look at `splay(x)`. Suppose y is the parent of x, and z is the parent of y. Here, there are three cases:

1. z is null (i.e. y is the root):

   We just call `rotate(x)`.

2. x is the left child of y, and y is the left child of z, or x is the right child of y and y is the right child of z:

   We call `rotate(y)`, then `rotate(x)`.

3. In all other cases (i.e. x and y are on different sides):

   We call `rotate(x)`, then `rotate(x)`.

In the second and third cases, we look at two levels at a time, where we move x two levels up at a time, until it is finally near the root, where we'll fall into the first case and we stop, now that x is at the root.

As an example, Fig. 4.2 illustrates the rotations in the case where x is the left child of y, which is the left child of z.

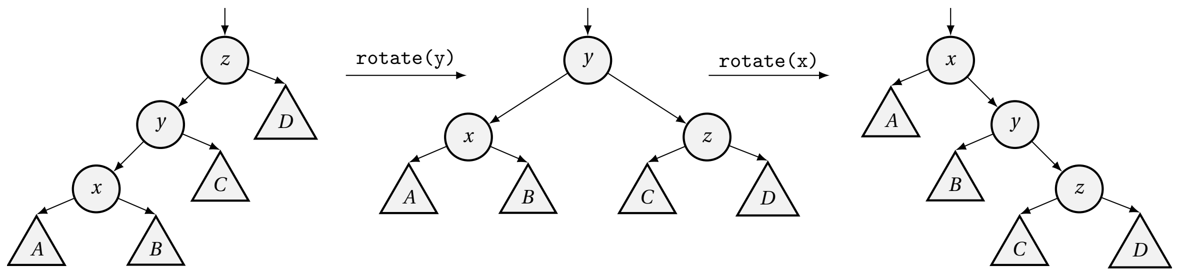


Figure 4.2: Calling `splay(x)` when x is the left child of y, and y is the left child of z

Lemma 4.2

The amortized cost of `splay(x)` is $O(\log(\frac{W}{s(x)}))$.

For each x in the database, we give it a weight $w(x)$ (arbitrary in the analysis; this applies for all weight functions $w(x)$), and we define

$$s(x) := \sum_{\substack{u \in x\text{'s subtree} \\ \text{(incl. } x\text{)}}} w(u),$$

and

$$W := \sum_{x \in \text{DB}} w(x) = s(\text{root}).$$

We then define $r(x) := \log(s(x))$.

(Here, if we want to prove the amortized cost is $O(\log n)$, we can just we the weight to all be 1; however, if we want to prove some more complicated ideas, we can set the weights to be something different.)

Proof. We will prove the claim through a *potential function argument*. For some state of our data structure, we assign a potential $\Phi : \{\text{states}\} \to \mathbb{R}$, mapping the states of the data structure to the real numbers.

We define the potential cost of an operation as

$$\text{potential cost} = \text{true cost} + \Delta\Phi.$$

Here, $\Delta\Phi$ is the difference in the potential of the data structure state before and after the operation:

$$\Delta\Phi := \Phi(\text{after operation}) - \Phi(\text{before operation}).$$

Since this telescopes when we sum across all operations, if we can say that the potential cost of each operation is low, then the amortized cost will be low.

Here, we define $\Phi = \sum_x r(x)$. We want to show that the amortized cost of `splay(x)` is at most $3(r(\text{root}) - r(x)) + 1$.

A `splay(x)` is essentially just a sequence of case 1's, 2's, and 3's (from earlier in the definition of `splay`). In each case, we have

1. Case 1: potential cost $\le 3(r'(x) - r(x)) + 1$, where r' is after the operation for case 1, and r is before the operation
2. Cases 2 and 3: potential cost $\le 3(r'(x) - r(x))$

Since we're doing these operations in a loop, this is a telescoping sum—at the end, we'll only have $r(\text{root})$ and $r(x)$ remaining.

We'll prove the bound for case 2, in the specific case where x is the left child of y, who is the left child of z. Since we're performing two rotates, the true cost is 2. The change in potential is then $(r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z))$, by definition.

We know that $r'(x) = r(z)$, since the subtrees are exactly the same set of nodes (i.e. z was at the top before the rotations, and x is at the top after the operations). We know that $r'(y) \le r'(x)$, since x is lower in the tree, and similarly, $r(y) \ge r(x)$, since y is higher in the tree. This gives a potential cost bound of

$$\text{potential cost} \le 2 + r'(x) + r'(z) - 2r(x).$$

Recall the AM-GM inequality: $\sqrt{ab} \le \frac{a+b}{2}$. Using this, we have that

$$\frac{r(x) + r'(z)}{2} \le \log\left(\frac{s(x) + s'(z)}{2}\right).$$

Here, $s(x) = s'(z)$ takes care of everything except for y, so this is at most $s'(x)$. This gives

$$\frac{r(x) + r'(z)}{2} \le r'(x) - 1.$$

This implies

$$r'(z) \leq 2r'(x) - 2 - r(x).$$

Plugging this in, we have

$$\begin{aligned} \text{potential cost} &\leq 2 + r'(x) + r'(z) - 2r(x) \\ &\leq 2 + r'(x) + 2r'(x) - 2 - r(x) - 2r(x) \\ &= 3r'(x) - 3r(x) = 3(r'(x) - r(x)) \end{aligned}$$

□

1/31/2023

Lecture 5

Link-cut Trees

5.1 Link-cut Trees

Today, we'll be talking about link-cut trees.

Recall that when finding blocking flows, we're traversing the graph to find paths from s to t. When doing so, after augmenting, we want to still remember the path fragments we've already visited. In the next iterations, if we end up on a tree we've saved, we jump to the root of the tree, as we've already found a path to the root before.

These path fragments together will be trees themselves (with singleton trees for vertices we haven't visited yet). Link-cut trees are a data structure to keep track of these tree fragments and allow for efficient jumping to tree roots.

With link-cut trees, we maintain a rooted forest subject to:

- `maketree(v)`: make a singleton tree just with node v
- `link(v, w, x)`: make w the parent of v with a respective capacity x on (v, w)

  Here, v is the root of a different tree, and we found a new edge from v to w, so we want to make w the parent of v
- `cut(v)`: remove edge $(v, \text{parent}(v))$ and return the residual capacity
- `findRoot(v)`: return the root of v's tree

  This is called when we want to jump to the root of a tree
- `findMin(v)`: return the edge with minimum capacity from v to its root, breaking ties by taking the edge closest to the root. This also returns the capacity of the edge.
- `addFlow(v, x)`: subtract x from u_e for all edges e on the path from v to its root.

Using link-cut trees, we have the following algorithm:

```
// initialize singleton trees
for each vertex v: makeTree(v)

while True:
    v = findRoot(s)
    if v = t: // augment
        (z, x) = findMin(s)
        // we found an s-t path, so add flow on this path
        addFlow(s, x)
        // while there is an edge on the path with zero capacity
        while (z, t) = findMin(s) has t == 0:
            cut(z)
            del (z, parent(z))
    else: // advance?
        if v has outgoing (v, w) in the graph:
            link(v, w, u_{v,w})
        else:
            if v == s: // done, since no outgoing from source
                break
            else: // mass retreat; cut all incoming to v
                for each child w of v:
                    cut(w)
                    del (w, v) from the graph
```

Now, we'll talk about how we implement link-cut trees using splay trees.

We'll have a helper function `access(v)`, and for now, think about this function as our intent to touch the vertex v.

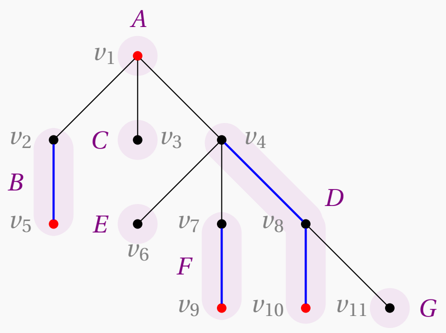
We'll use a *preferred path decomposition*, where each vertex has at most one *preferred child*. The edge to the preferred child will be called the *preferred edge*. The preferred child of v is defined to be:

$$\text{prefChild}(v) = \begin{cases} \text{null} & \text{if } v \text{ was last vertex accessed in the subtree} \\ \text{child toward the most recent access} & \text{otherwise} \end{cases}.$$

We also define a *preferred path* to be a maximal sequence of preferred edges.

Example 5.1

For example (with red vertices as recently accessed, in some ordering):

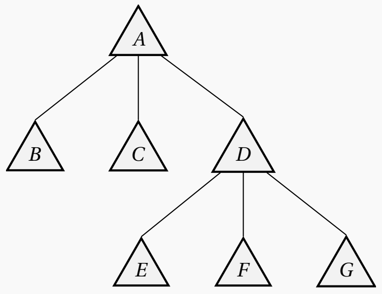


We store the preferred paths in splay trees (in particular, the vertices will be the elements in the splay tree); the natural question is what the key of each element should be. We'll take the key to be the depth of the node in the tree (here, higher up in the tree means smaller).

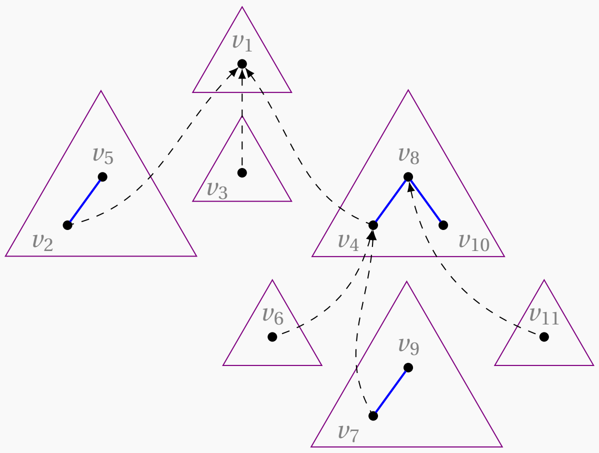
Here, we'll have multiple splay trees representing a single larger tree, and we have multiple of these larger trees used in the link-cut data structure. We call the trees in the link-cut tree *represented trees*, and we call the splay trees *auxiliary trees*.

Example 5.2

The earlier example gives the following tree of auxiliary trees:



With the splay trees drawn in, we have the following:



Consider when we access a vertex v. After this access, the preferred paths and auxiliary trees will all change—notice that v must be in the root auxiliary tree, since the path from the root to v must be preferred.

After we access v, we'll now splay v in the root auxiliary tree, so that v is actually the root of the root auxiliary tree.

To implement `link(v, w, x)`, we have:

```
def link(v, w, x):
    access(v)
    access(w)
    v.parent = w
    w.right = v
```

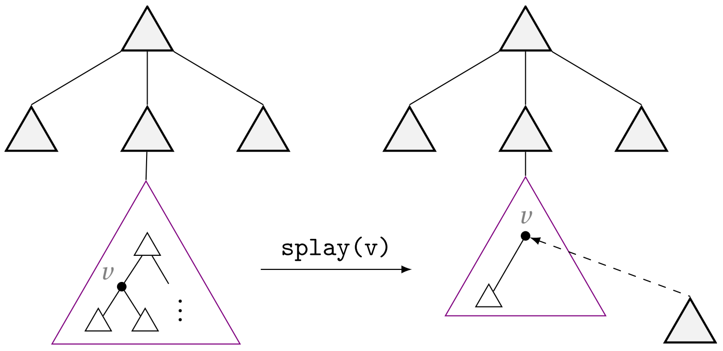
Here, we access v before w, so that w is the most recently accessed vertex in its tree. w and v are still in different auxiliary trees (in particular, they're the root of their respective auxiliary trees, and since it's now the deepest vertex in the preferred path to v), and in particular after the link, v is in its own auxiliary tree (v was the root of its represented tree, and it is now the most recently accessed in its subtree, so it has no preferred child)

To implement `cut(v)`, we have

```
def cut(v):
    access(v)
    v.left.parent = null
    v.left = null
```

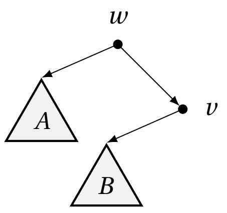
In the represented tree, we remove the parent edge from v, which would disconnect v's tree of auxiliary trees—this makes v a singleton auxiliary tree.

Now, we'll talk about how we implement `access(v)`. Suppose we have the following tree of auxiliary trees:



After splaying v, v gets moved to the top of its own auxiliary tree. Since now v is the most recently accessed, everything to the right of v becomes a new auxiliary tree. Further, there must be some highest vertex t in the preferred path ending at v; this vertex is connected to some other vertex w in another auxiliary tree.

We splay w now, and this moves w to the root of its auxiliary tree, with some potential right child in the auxiliary tree. This right child was the preferred child path of w, but now that we've accessed v, the preferred child of w is now t (since t leads down to v). This means that the right child of w gets kicked out into its own auxiliary tree, and it's replaced by v.



For every preferred child change, we do two splays (we splay v and w). This means that the cost of calling `access` is the cost of doing PCC (preferred child change) splays (+1). This means that the total runtime is $O(\text{cost(splay)} \cdot (1 + \text{PCC}))$.

We know that the amortized cost of `splay` is $O(\log n)$, and we'll see that the number of preferred child changes is $O(m \log n)$. In total, this gives an amortized runtime per operation of $O(m \log^2 n)$.

To analyze the number of preferred child changes, we'll make use of a *heavy-light decomposition*; this is only used in the analysis, as the algorithm never knows what is heavy and what is light.

Here, we define $\text{size}(v)$ to be the number of nodes in v's subtree (including v). We define an edge from v to its parent is *heavy* if $\text{size}(v) > \frac{1}{2}\,\text{size}(w)$, and we define the edge to be *light* if $\text{size}(v) \le \frac{1}{2}\,\text{size}(w)$.

In the represented tree, suppose the preferred child of w switched from v to v'. Here, we say that (v, w) is a preferred child destruction, and (v', w) is a preferred child creation. The fact that there is a destruction and a creation means that there is a preferred child change.

We want to bound the number of preferred child changes. This bound turns out to be that the number of preferred child changes is bounded by $2 \cdot \text{LPCC} + m + n$ (light preferred child changes), and that $\text{LPCC} \le O(m \log n)$.

2/2/2023

Lecture 6

Link-cut Tree Wrap-up, Flow Wrap-up

6.1 Link-cut Tree Analysis (cont.)

Today we'll continue with the runtime analysis for link-cut trees.

Recall that in a heavy-light decomposition, if we have a child w of v, we say w is *light* if $\text{size}(w) \le \frac{1}{2}\text{size}(v)$, and w is *heavy* if $\text{size}(w) > \frac{1}{2}\text{size}(v)$.

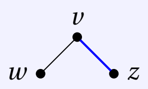
Lemma 6.1

$\text{PCC} \le 2 \cdot \text{LPCC} + m + n$.

That is, the number of preferred child changes is at most twice the number of light preferred child creations, plus $m + n$.

Proof. We know that PCC = LPCC + HPCC, since any preferred child change must involve either a light preferred child creation or a heavy preferred child creation. It then suffices to show that $\text{HPCC} \le \text{LPCC} + m + n$.

Suppose we have a preferred child change that caused a heavy preferred child creation of z.



There are a couple cases:

- w is null.

  This can happen in a couple cases:

  – The last access in v's subtree was v; in this case, v had no preferred child, so z became a new heavy preferred child. This can only happen at most m times across the entire operation sequence, since there's only m possible operations.

  – This is the first ever access in v's subtree. This can only happen at most once per v, and there are n possible v's, so this can only happen at most n times.

- w is not null.

  Since z is heavy, it has strictly more than v's mass, which must make w a light child. This means that one light preferred child was destroyed. We can charge this destruction along with the creation for w. This can only happen at most LPCC times, since we can't have more destructions than creations.

Together, this gives us our desired bound. □

Lemma 6.2

$\text{LPCC} \le O(m \log n)$

Proof. Light preferred child creations can only happen as a result of an operation. We can show that for any operation, the number of light preferred child creations is bounded by $O(m \log n)$.

Suppose we look at `access(v)` first. When we access a vertex v, there could be a lot of child creations along the route from the root to v. However, we're only considering light preferred child creations.

Every time we follow a light edge, the size of the tree must have been halved; this means that the number of light preferred child changes must be at most $O(\log n)$.

Suppose we look at `link(v, w)`. When we link v and w, this means that we make v one of w's children. This makes $\text{size}(w)$ larger, and this also propagates along w's parents (all of w's parents to the root must also get heavier).

`link` does at most 2 accesses (each has at most $O(\log n)$ LPCC's). We then add a single edge from v to w; this doesn't change preferred edges, but it can change which edges are light or heavy. This means that some light edges could have changed to be heavy, but we don't care about heavy preferred child changes, so this doesn't affect our bound on light preferred child creations. Additionally, no light preferred edges can be created here, elsewhere in the graph, since v and w were the last accessed in the subtrees. This means that the only LPCC's created are from the 2 accesses, which is at most $O(\log n)$. □

6.2 Min Cost Max Flow

With min-cost max-flow, we have the same setup as max-flow, except now edges have costs $c_e \in \{-C, -C+1, \dots, C\}$

The goal is to find a max-flow of minimum cost. That is, the priority is to find a max-flow, but we want a max-flow of the minimum cost. Here, we define a cost of a flow f to be $\text{cost}(f) = \sum_{e\in E} f_e \cdot c_e$.

Chen et al., in FOCS 2022 [CKL+22] showed a near-linear runtime for min-cost max-flow of $m^{1+o(1)} \cdot \log(C) \cdot \log(U)$.

A related problem is *minimum cost circulation*. Here, a circulation is a flow with zero value (so we're only using cycles). The goal is to find a circulation of minimum cost.

These two problems are related to each other; a fast algorithm for min-cost max-flow will imply a fast algorithm for min-cost circulation (up to some linear time changing of the input).

If we have an algorithm for min-cost max-flow, we can solve min-cost circulation on a graph G by adding a source and a sink that is disconnected from G, and running min-cost max-flow on this graph.

If we have an algorithm for min-cost circulation, we can solve min-cost max-flow by adding an edge from t back to s with infinite capacity and very negative cost. The negative cost causes the min-cost circulation to prefer to push as much flow as possible from t to s (this then means that the same amount of flow goes from s to t, and we'd be maximizing the flow from s to t).

Another way to solve min-cost max-flow with min-cost circulation is as follows. Suppose we have any max-flow from s to t. We can then augment it with a min-cost circulation in the residual graph, which does not affect the amount of flow from s to t, but minimizes the cost.

6.2.1 Algorithms

We can observe that any circulation decomposes in to a sum of at most m cycles. For each of the cycles, specifically in the min-cost circulation, must have a negative cost (otherwise, we can just drop the cycle and we have a smaller cost circulation).

This means that we want to find as many negative cycles in the graph as possible. Using this observation, we have the following algorithm:

1. Find max flow
2. Compute residual graph
3. Augment by a negative cost cycle, go back to previous step

We halt when there are no negative cycles left.

The runtime for this algorithm is $O(\text{time to find max-flow} + (\text{num iterations}) \cdot \tilde{O}(m \log C)$; here, we assume we use the BNC algorithm from before. The number of iterations is bounded by $2mUC$, since we can send at most mU flow through all edges, and each can have at most C cost.

This is a pseudopolynomial time algorithm, since it depends on U and C.

6.3 Special Cases

Here, we'll go back to price functions $\varphi : V \to \mathbb{R}$. Suppose we define the reduced cost of $e = (v, w)$ to be $c_\varphi(e) := c(e) + \varphi(v) - \varphi(w)$.

Here, we can break out of the loop of finding negative cost cycles when there exists a price function where all of the reduced costs are nonnegative. Why? The reduced cost of a cycle is the same as the actual cost of the cycle; this means that if all reduced costs are nonnegative, then all costs of cycles are nonnegative.

Similarly, if there are no negative cycles, then there must be a good price function (i.e. create dummy source to every vertex, and let the price be the shortest path distance (by cost) to each vertex, and the triangle inequality shows that it's a good price function).

6.3.1 Nonnegative Costs, Unit Capacity

In min-cost max-flow, what if all costs were nonnegative to start out with, and all edges have unit capacity?

In this case, we can initially set the flow f to 0 and the price $\varphi(v) = 0$ for all v. We then find the shortest augmenting path (SAP) from s to t (using the cost c_φ), and augment along this path.

For each vertex v in the SAP, we then update $\varphi(v) = \varphi(v) + d_\varphi(s, v)$.

We claim that throughout this entire process, we never introduce any negative cost edges. Whenever we augment along a path, the reverse edge gets introduced with negative reduced cost (we're always looking at cost with respect to φ); we'll show that this cost is always zero.

If we look at some shortest augmenting path from s to t, and look at some edge (v, w) along this path, we know that

$$d_\varphi(s, w) = d_\varphi(s, v) + c_\varphi(v, w),$$

since the shortest path to w must go through v by definition.

After augmenting along the path, we claim that the new cost of the edge is zero under the updated price function. Here, the new reduced cost of the edge is:

$$\begin{aligned} c_{\varphi_{new}}(v, w) &= c_{\varphi_{old}}(v, w) + d_{\varphi_{old}}(s, v) - d_{\varphi_{old}}(s, w) \\ &= c_{\varphi_{old}}(v, w) + d_{\varphi_{old}}(s, v) - \left(d_{\varphi_{old}}(s, v) + c_{\varphi_{old}}(v, w)\right) \\ &= 0 \end{aligned}$$

Because this never encounters negative cost edges, we can just greedily push more and more flow.

6.3.2 Negative Costs, Unit Capacity

What if we had unit capacities, but some costs can be negative?

First, we find a max-flow with no consideration of cost. Now, we want to find a min-cost circulation in the residual graph.

In the graph G, we can identify which edges have negative cost, and we initialize our flow by saturating all of these edges with negative cost (see Fig. 6.1 for an example). This isn't a valid flow or circulation, since it doesn't obey conservation of flow.

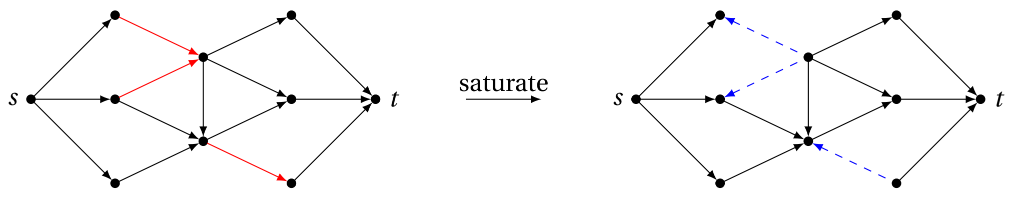


Figure 6.1: Saturating all negative edges (in red) in a graph G

In particular, some vertices have excess flow leaving them, and some have a deficit. Additionally, notice that when we saturate the negative edges, the reverse edge appears in the residual graph with the same capacity and positive cost.

We then create a new super source s' and a new super sink t', and in the residual graph (i.e. with the reverse edges from the saturated negative edges), we introduce new edges from s' to all vertices with an excess flow leaving them, and we introduce new edges to t' from all vertices with an excess flow going in (see Fig. 6.2).

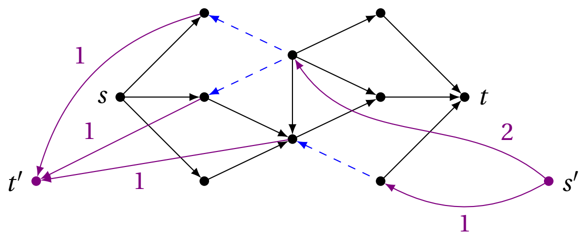


Figure 6.2: Creation of a super-source s' and super-sink t' for the graph in Fig. 6.1

Now, since all edges are now nonnegative cost, we can find the min-cost max-flow from s' to t'. This min-cost max-flow will always fully saturate our new edges from s' and t', since the trivial max-flow just reverses our newly added saturated edges. However, since we're finding the min-cost max-flow, pushing this flow from s' to t' will also automatically minimize the cost of flow through the rest of the graph (which would all be composed of circulations).

Crucially though, saturating these negative edges means that all edges in our graph now have nonnegative cost, and we can use the algorithm described earlier to find the min-cost max-flow efficiently.

2/7/2023

Lecture 7

Heaps

In the next five or so lectures, we'll be focusing on various kinds of data structures. Today in particular we'll be talking about heaps, or priority queues.

Heaps store a database of key-value pairs (where keys are comparable) subject to the following operations:

- `insert(k, v)`: Insert a key-value pair to the database
- `decKey(*v, k)`: Given a pointer to the item, this reduces the old key of `v` to become the new key `k`; do nothing if the new key is larger.

  Here, the pointer is technically a pointer to the node in the heap; this is technically an abstraction violation, but for our purposes all the data structures we talk about today will have nodes corresponding to each item.

  We'll also assume that the values are distinct, otherwise this may pose a problem referencing the values.
- `delMin()`: Return the item with the smallest key, and delete it from the database.

7.1 Runtime of Heap Algorithms

In Dijkstra, we have n `insert` operations (one for each vertex), at most n `delMin` operations, and at most m `decKey` operations. This gives a runtime of $n \cdot t_I + n \cdot t_{DM} + m \cdot t_{DK}$.

Prim's algorithm is basically the same; we have n `insert` operations (one for each vertex), n `delMin` operations (one for each vertex as we add it to the MST), and m `decKey` operations (one per neighbor of a vertex). This gives the same runtime of $n \cdot t_I + n \cdot t_{DM} + m \cdot t_{DK}$.

Today, we'll see heaps with the runtimes per operation in Table 7.1.

Binary heaps are from Williams in 1964 [Wil64], Binomial heaps are from Vuillemin in 1978 [Vui78], and Fibonacci heaps are from Fredman and Tarjan in 1987 [FT87]. We won't be talking about binary heaps today (you've seen them before), but we'll be talk about binomial and Fibonacci heaps.

Name	t_I	t_{DM}	t_{DK}
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binomial Heap	$O(1)^\dagger$	$O(\log n)$	$O(\log n)$
Fibonacci Heap	$O(1)^\dagger$	$O(\log n)^\dagger$	$O(1)^\dagger$

† amortized time

Table 7.1: Various heap algorithms and their runtimes per operation

More recently, Brodal in 1996 [Bro96] matched the Fibonacci bounds but all in worst-case, without amortization. The caveat is that it's not a "pointer-machine" data structure. (A pointer-machine data structure is one that has nodes and pointers to nodes.) Brodal, Lagogianis, and Tarjan in STOC 2012 [BLT12] matched the Fibonacci heap bounds in worst-case with a pointer-machine data structure.

Our analyses of these data structures will be based on potential function arguments; again, recall that a potential function Φ maps the state of the data structure to a real number. In particular, the Φ-cost is defined to be the true cost of the operation, plus $\Delta\Phi$, i.e. the change in potential. The total Φ-cost then becomes

$$\text{total } \Phi\text{-cost} = \text{total cost} + \Phi_{\mathit{final}} - \Phi_{\mathit{init}}.$$

Today, we'll define Φ for an empty data structure to be 0 (making $\Phi_{init} = 0$), and that in general $\Phi \geq 0$, so the total Φ-cost is an upper bound on the total cost. This means that the Φ-cost for an operation is a valid upper bound on the amortized cost of that operation.

7.2 Binomial Heap

In contrast to binary heaps (where we store items in a single tree), in a binomial heap, we store items in a forest of trees. Here, each tree is heap-ordered; that is, the key of an item is at least as large as the key of the item's parent.

We'll define the "rank" of a tree to be the number of children of the root (i.e. the degree of the root). The invariant here is that there is always at most one tree of each rank.

Figure 7.1 gives examples for rank 0, 1, 2, and 3 trees in a binomial heap.

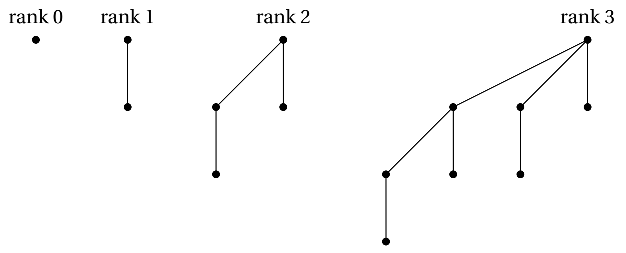


Figure 7.1: Examples of rank 0 through rank 3 trees in a binomial heap. Note that this forest is a 15-element binomial heap.

Further, note that a rank k tree will *always* have rank j trees hanging off the root for all $j = 0, 1, \dots, k-1$. In particular, any binomial heap with n items will always look the same.

When we insert an item, we add it as a singleton tree (of rank 0), and merge any duplicate trees—to merge two trees, we compare the roots of the duplicate rank trees, and make the larger root the parent of the other tree. This merges to rank k trees into one rank $k+1$ tree. We can keep doing this until there are no duplicate rank trees; see Fig. 7.2 for an example.

Note that this is equivalent to the process of binary addition; if we write the number of items n in binary, the bits at index k with a 1 denote that we have a rank k tree in the binary heap with n items. Inserting an element would then correspond to adding 1 to this binary number; the number of carry bits that arise is equivalent to the number of merges we do (each carry bit corresponds to a duplicate rank k tree).

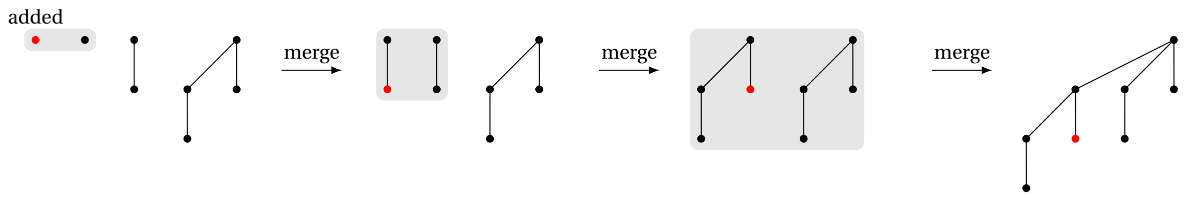


Figure 7.2: Cascading merges when adding an item to a binomial heap

When we call `decKey(*v, k)`, we have a pointer to a node, so we just change the key and make it smaller. If the new key violates the heap order property (i.e. the new key is smaller than its parent), then we just swap the item and its parent. We can keep doing this until there are no violations.

Note that a rank k tree has 2^k items, and the depth of the tree is k. As such, this means that the largest rank tree is at most $\log n$, and the depth of this tree is at most $\log n$. This means that `decKey` takes at most $O(\log n)$ time.

When we call `delMin()`, we scan through all of the roots, and delete the minimum root. By deleting this root, we now have a lot of duplicate rank trees (as these trees were children of the root we deleted). We can then iteratively merge trees of duplicate ranks. (Observe that this is essentially a binary addition.)

7.2.1 Analysis

Here, we define the potential function as

$$\Phi(H) := \text{number of trees in } H = T(H).$$

Recall that the potential cost is the actual cost, plus the difference in potential.

Consider the potential cost for `insert`. Before the insertion, suppose we have T trees before the operation, and t is the number of trees after the operation. The difference $T - t$ is essentially the number of carry bits from adding 1, which is the actual cost of inserting an item (plus some constant). The difference in potential by definition is $t - T$, so the potential cost is

$$\Phi\text{-cost} = \underbrace{1 + (T - t)}_{\text{actual cost}} + \underbrace{(t - T)}_{\Delta\Phi} = O(1).$$

For `decKey`, the actual cost is $O(\log n)$, since we have to swap up the tree, and the difference in potential is 0, so this gives

$$\Phi\text{-cost} = \underbrace{O(\log n)}_{\text{actual cost}} + \underbrace{O(\log n)}_{\Delta\Phi} = O(\log n).$$

For `delMin`, the actual cost is $O(\log n)$ to scan for the minimum, and we have to do some merging, which is also $O(\log n)$. The difference in potential is nonnegative, but is at most $\log n$, since we have at most $\log n$ trees. This means that the potential cost is

$$\Phi\text{-cost} = \underbrace{O(\log n)}_{\text{actual cost}} + \underbrace{O(\log n)}_{\leq\Delta\Phi} = O(\log n).$$

7.3 Fibonacci Heaps

The idea here is to start with binomial heaps, and make them as lazy as possible.

- `insert`: Add a new rank 0 tree

  This is just the lazy version from binomial heaps; we don't merge any duplicate trees.

- `decKey`: Change the key, and remove the parent edge, making the item the root of a new tree.

  We can slightly optimize this too, not doing anything if it doesn't violate the heap property.

- `delMin`: Scan through roots, find the min, remove the root from the tree, and consolidate the trees, by merging equal rank trees.

Here, technically decrease-key isn't quite right. Recall that we used the fact that the maximum rank is $\log n$, since each tree has 2^k nodes. This won't always be the case with this version of `decKey`, if we just keep moving nodes to new trees.

For a better `decKey` (actually used in Fibonacci heaps), the first time a node loses a child, we just cut the child from the parent and do nothing else. The second time a node x loses a child, we also cut x from its parent (this makes x the root of a new tree along with its remaining children). Note that this may cascade up the tree (as it may now be the second time the parent of x loses a child).

To implement this, each node maintains a "mark" bit; x's mark is 1 if x has lost a child, and when x loses two children, then we make x the root of its own tree, and set the mark bit back to 0.

7.3.1 Minimum Tree Sizes

Before we analyze the runtime, let's look at some intuition for the maximum possible size of a tree as a function of its rank. This isn't a formal proof, but it'll give some intuition.

The minimum possible rank 0 tree has 1 node (we can't do anything). The minimum possible rank 1 tree has 2 nodes (we can't cut any children off). The minimum possible rank 2 tree has 3 nodes (we can cut off one grandchild of the root). The minimum possible rank 3 tree has 5 nodes, and the minimum possible rank 4 tree has 8 nodes, etc.

From this pattern, we can claim that a rank k tree has size at least F_{k+2} (if we let $F_0 = 0$ and $F_1 = 1$).

Recall that for Fibonacci numbers, we have $F_k = F_{k-1} + F_{k-2}$, and since the Fibonacci numbers are increasing, this is at least $2 \cdot F_{k-2}$. If we keep iterating this, we have $F_k \geq 2 \cdot F_{k-2} \geq 2^2 \cdot F_{k-4} \geq \cdots \geq 2^{\frac{k}{2}}$.

This means that the Fibonacci numbers are actually growing exponentially, so the number of trees must be at most $O(\log n)$. This essentially fixes the problem of trees being too small, allowing us to assert that there are at most $\log n$ trees in the forest.

7.3.2 Analysis

When defining potential functions, we tend to pick ones that cancel out the actual work in some beneficial way. A potential function that works in this case is

$$\Phi(H) := T(H) + 2 \cdot M(H).$$

Here, $T(H)$ is the number of trees in the heap, and $M(H)$ is the number of items in the heap that are marked.

For `insert`, the actual cost is constant (we just add the item as a singleton tree), and the potential difference is $\Delta T(H) + 2 \cdot \Delta M(H)$. Since we added one more tree, $\Delta T(H) = 1$, and we don't change any marked items, so $\Delta M(H) = 0$. This gives a potential cost of

$$\Phi\text{-cost} = \underbrace{O(1)}_{\text{actual cost}} + \underbrace{1}_{\Delta T(H)} + \underbrace{2 \cdot 0}_{2\Delta M(H)}.$$

For `decKey`, suppose there are C cascaded cuts as a result from the decrease key. The actual cost is $O(1) + C$. We added C new trees, since each cut added a new tree. For the change in marked items, the item we call `decKey` on does not change its mark bit, but all of the other parent nodes we cut off in the cascade got unmarked, and the highest node could have gotten marked. This gives a delta of $-(C-1)+1$, which gives

$$\begin{aligned}\Phi\text{-cost} &= \underbrace{O(1) + C}_{\text{actual cost}} + \underbrace{C}_{\Delta T(H)} + \underbrace{2(-C+2)}_{2\Delta M(H)} \\ &= O(1) + C + C - 2C + 4 \\ &= O(1)\end{aligned}$$

For `delMin`, the actual cost is proportional to the number of trees. The change in the number of trees is bounded by $O(\log n)$, since this is the maximum rank. The markedness also doesn't change, so this gives an $O(\log n)$ potential cost.

2/9/2023

Lecture 8

Van Emde Boas Trees, y-fast Trees

Today and next lecture, we'll focus on the word RAM model, and in particular, we'll look at the predecessor problem.

8.1 Predecessor problem

In the predecessor problem, we are given a database S of key-value pairs (we're really only going to be focusing on the keys today), where keys are in $\{0,1,2,\dots,u-1\}$, and a query is a *predecessor query*. That is, `pred(x)` returns the maximum $z \in S$ such that $z < x$. We also want to support `insert(x)` and `delete(x)` in the dynamic case.

In the static version of the problem, the simplest solution is to sort the database of keys, and binary search for a query. This would give a $O(\log n)$ query time.

For the dynamic version of the problem, we'd use balanced binary search trees, red-black trees, splay trees, etc.

Today, we'll use the *word RAM model*, utilizing the fact that the keys are integers to do better.

In the word RAM model, the CPU has some built-in operations (ex. `LOAD`, `STORE`, $+$, $-$, $\times$, $/$, $\ll$, &, $\|$, etc.), all operating on words of w bits, in constant time. For us, $u = 2^w$, i.e. the largest integer we can perform operations on is 2^w.

Van Emde Boas in FOCS 1975 [Emd75] gave a dynamic data structure with $O(\log\log u)$ per operation, and $O(u)$ space; this is called a van Emde Boas tree. It's also possible to reduce the space to $O(n)$, with randomization (i.e. hash tables).

Willard in IPL 1983 [Wil83] gave a different data structure with the same bounds; these are called "y-fast tries".

Fredman and Willard in STOC 1990 [FW90] invented "Fusion trees", which is a static database with query time $O(\log_w n)$. Raman in ESA 1996 [Ram96] showed how to get dynamic fusion trees, where all operations are $O(\log_w n)$ time.

Note that if we have a good dynamic data structure for the predecessor problem, then we also have a good sorting algorithm; simply add all the items into the database, keeping track of the maximum, and then keep removing the predecessors starting from the max.

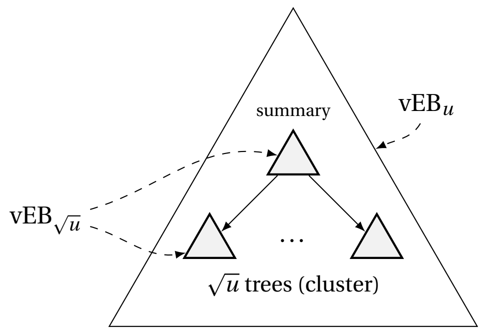
In particular, vEB trees give a sorting bound of $O(n \log w)$, and Fusion trees give a sorting bound of $O(n \log_w n) = O\left(n \cdot \frac{\log n}{\log w}\right)$. Since w is CPU-dependent (and fixed for a given CPU), we can get $O\left(n \cdot \min\left(\log w, \frac{\log n}{\log w}\right)\right)$, which is also $O(n\sqrt{\log n})$.

Han in 2002 [Han02] showed you can sort in $O(n \log\log n)$ deterministically, and Han and Thorup in 2002 [HT02] showed you can sort in $O(n\sqrt{\log\log n})$ with randomization, and it's an open problem whether you can sort in $O(n)$ time. These two sorting algorithms do not involve better predecessor data structures; there is a lower bound for sorting with predecessor data structures of $\min(\text{vEB}, \text{Fusion})$, due to Patrascu and Thorup in STOC 2006 [PT06] and SODA 2007 [PT07].

Recall that with B-trees, the main motivation was to fit more information into the same page in memory, allowing for more information to be retrieved at the same time—Fusion trees have a similar motivation, which we'll see next time.

8.2 vEB Trees

vEB Trees are parametrized by a universe size (initially u), and are recursively defined.



The fields stored by vEB_u are:

- `max`: largest key in S
- `min`: smallest key in S
- `cluster[0, 1, ..., `$\sqrt{u}-1$`]`: an array of pointers to $\text{vEB}_{\sqrt{u}}$ trees
- `summary`: a pointer to a single $\text{vEB}_{\sqrt{u}}$ tree.

We'll treat all $x \in \{0, \dots, u-1\}$ as $x = \langle c, i \rangle$, where $c, i \in [\sqrt{u}]$, and $[N]$ means $\{0, 1, \dots, N-1\}$. The most significant $\frac{w}{2}$ bits are stored in c, and the least significant $\frac{w}{2}$ bits are stored in i.

If x is in the database, we'll store i recursively in `V.cluster[c]`. The first time we store an element into a tree, we'll store it in the min and max fields, without doing any recursive storing.

Then, c is inserted into `V.summary` if S contains any key that starts with c. That is, the summary tells us which of the clusters are non-empty.

A predecessor query then proceeds as follows.

1. We first write $x = \langle c, i \rangle$, which can be done in constant time (i can be gotten with a bit-mask, and c is a mask and a shift downward).
2. If $x >$ `V.max`, then we return `V.max` (the largest element is the predecessor of x).
3. Otherwise, there are two choices: either x's predecessor lives in c, or it lives in the predecessor cluster of c.

   If `V.summary` contains c (a constant time query, since we can just check if the tree exists at the array index), then there is a possibility that the predecessor lives in the cth cluster.

   To check whether this is the case, we can check if $i >$ `V.cluster[c].min`, in which case we would return $\langle c, \texttt{V.cluster[c].pred(i)} \rangle$. This recurses into the child tree for the predecessor.

   Otherwise (if the summary doesn't contain c, or if i is less than the minimum, then we retrieve c' as `V.summary.pred(c)`, and return $\langle c', \texttt{V.cluster[}c'\texttt{].max} \rangle$.

Note that it could be possible that there is no predecessor (i.e. there's nothing in the summary, etc.), in which case we'd need some special null handling.

If we analyze the runtime of `pred(x)`, we have some constant time operations, with one recursive query into the child tree of size $\sqrt{u}$. This gives a recurrence relation of

$$T(u) \le T(\sqrt{u}) + O(1) \implies T(w) \le T\left(\frac{w}{2}\right) + O(1).$$

This solves to $T(w) = O(\log w) = O(\log\log u)$.

For `insert(V, x = <c, i>`, we have:

```
def insert(V, x = <c, i>):
    if V.min == null:
        V.min = V.max = x
        return
    if x < V.min: swap(x, V.min)
    if x > V.max: V.max = x
    if V.cluster[c] == null: V.summary.insert(c)
    insert(V.cluster[c], i)
```

Here, if there are no elements in V, we just set the minimum to x and we're done (we don't want to recurse for the first item). Otherwise, if we have a new minimum, we can swap x and `V.min` and continue on to store the old min (now x). If x is larger than the current maximum, then we overwrite the maximum and continue on to store x. In storing x, we first check if `cluter[c]` is defined; if not, then we insert c to the summary tree. Once we know that c is kept track of in the summary tree, we insert i recursively into `cluster[c]`.

For the runtime analysis, we do two recursive calls into the smaller trees, so we have

$$T(u) \le 2 \cdot T(\sqrt{u}) + O(1) \implies T(w) \le 2T\left(\frac{w}{2}\right) + O(1).$$

However, this solves to $T(w) = O(w)$, which is not what we want. It turns out that there shouldn't be a factor of 2. In particular, if we call `V.summary.insert(c)`, this is the first time we insert anything into the tree, so there are no further recursive calls in the next line with `insert(V.cluster[c], i)`. This gives the better recurrence

$$T(u) \le T(\sqrt{u}) + O(1) \implies T(w) \le T\left(\frac{w}{2}\right) + O(1).$$

This gives $O(\log w) = O(\log\log u)$ runtime.

For space, we have

$$S(u) \le \underbrace{(\sqrt{u}+1)}_{\text{subtrees}} S(\sqrt{u}) + \underbrace{O(\sqrt{u})}_{\text{fields, pointers, etc}}.$$

This solves to $O(u)$. To see why, this can be rewritten as

$$S(w) \le 2^{\frac{w}{2}} S\left(\frac{w}{2}\right) + O(2^{\frac{w}{2}}).$$

This then solves to $S(w) = O(2^w)$.

However, this is a pretty bad space complexity, but we can improve the space by simply storing the clusters as a hash table (i.e. a solution to the dynamic dictionary problem) instead of an array.

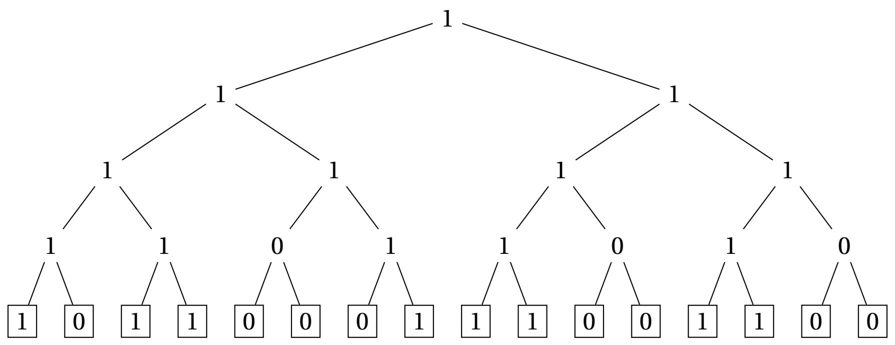
If we assume we have a good hash table (i.e. linear space, constant time queries), then the space it takes to store clusters is proportional to the number of clusters we store. The key to the hash table should be c, and the value should be a pointer to the child tree.

The amount of memory we need now is proportional to the sum of the sizes of non-empty VB trees throughout the recursion (i.e. the total number of insertions we do, including all recursive insertions).

We claim that any given item stored into the database only causes insertions into $\log(w)$ trees. Why is this? If we follow the insertion code, we're either recursing into the summary or recursing into the cluster, so we have $\log(w)$ levels of recursion (each level halves the bits of the number). This gives $O(n \log w)$ memory.

8.3 x-fast Tries

Suppose the database is $\{0,2,3,7,8,9,12,13\}$, with $w = 4$ and $u = 16 = 2^4$. To store these items, we can use a bit array of length u. In particular, we'd have the bit array 1011000111001100.



We'll support this with a perfect binary tree, where each internal node is the bitwise OR of its children. Further, we'll store the locations of all of the 1's in a doubly linked list, so we can jump between 1's in constant time.

To find the predecessor, we can go up the tree until we find a 1. If we found the 1 from its left child, then we can find the smallest element in the right subtree for the successor, and jump to the predecessor. If we found the 1 from its right child, then we can find the largest element in the left subtree for the predecessor.

This gives us $O(\log(u)) = O(w)$ time. However, if we can get to the kth ancestor in constant time, then we can reduce this time, by using a binary search—we can do this, because the path from a node up its ancestors will always be monotonically increasing.

To do this, we can store the tree in a length $2u-1$ array. For a node at index x (1-indexed), the left child is at $2x$, and the right child is at $2x+1$. This also means that finding a parent is a bit shift (dividing by 2). This means that finding the kth parent is just a right bit shift by k, which can be done in constant time in the word RAM model.

This means we can binary search through a $O(\log u)$ length path, which takes $O(\log\log u) = O(\log w)$ time, which is a fast query.

The space is bad (though it can be improved using hash tables at each level to store only the 1's; there are $O(w)$ levels, and $O(n)$ per level, so this is $O(nw)$ space), and insertion is slow (since we need to maintain the tree). This gives $O(nw)$ space, $O(\log\log u)$ query, and $O(\log u)$ update.

We can improve this to create the y-fast trie, which we'll finish up next lecture.

2/14/2023

Lecture 9

y-fast Tries, Fusion Trees

9.1 y-fast Tries

Compared to an x-fast trie, y-fast tries take $O(n)$ space, $O(\log\log u)$ query, and $O(\log\log u)$ amortized update.

With a y-fast trie, we'll group keys in the database in groups of size $\Theta(w)$ each; we'll have group sizes in $[\frac{w}{4}, 2w]$.

Each group will store elements in a balanced binary search tree (ex. red-black tree, etc.). We'll then take one representative from each group (for example, the smallest element in each group). These representative elements will be elements in the x-fast trie, which we'll build on top of this.

This means that although we actually have n elements in the database, in the perspective of the x-fast trie, we'll only have $\frac{n}{w}$ elements in the database.

If we do a predecessor query, we'll first find the representative that is the predecessor of x. If we take the representative to be the smallest element, the actual predecessor must be in the same group as the representative (the representative in the next group is the smallest element in that group, which is larger than x). We'd then search in the binary tree for the true predecessor. This means that a query takes $O(\log\log u)$ time total.

For space, balanced binary trees take linear space, the BSTs take $O(n)$ space, and the x-fast trie takes $O(\frac{n}{w} \cdot w) = O(n)$ space, since there are $O(w)$ levels, and $O(\frac{n}{w})$ 1's per level to store in a hash table.

For an insertion, we can just add the element into the corresponding binary tree; if this doesn't change the representative, then the ancestors don't need to be updated. However, if we exceed the $2w$ group size, then we can split the bottom half and top half of the BST into two separate groups, and we add a new representative element for the new group to the x-fast trie. This would cause an update for all of the ancestors of the new groups.

Whenever we do this split, we can charge the cost of updating the ancestors to the ancestors themselves; since we must have added at least w items, each update is essentially constant time amortized.

This trick of using a simpler data structure for leaves is called *indirection*.

9.2 Fusion Trees

Fusion trees solve the static predecessor problem in $O(\log_w n)$ time [FW90]; it is possible to make dynamic fusion trees [Ram96], but we won't cover them in class.

The idea is to use a B-tree/2-3-4-tree type data structure. A 2-3-4-tree is like a binary tree, except each node can have either 2, 3, or 4 children. Within each node, elements are sorted, and each branch covers an interval between the elements in the node.

In a fusion tree, every node has $\Theta(k)$ children, and we'll choose $k = w^{\frac{1}{5}}$.

Naively, we have $O(\log_k n)$ levels, but we'd need to binary search to find which branch to go down at a given node, which costs $O(\log k)$ time. This gives a naive runtime of $O(\log k \cdot \log_k n) = O(\log n)$.

Fusion trees gives a constant $O(1)$ time search for each node, which results in $O(\log_k n) = O(\log_w n)$ time.

The main ingredients are (1) sketching, (2) word-level parallelism (specifically, parallel comparison), (3) the power of integer multiplication, and (4) finding the most significant set bit in $O(1)$ time.

9.2.1 Sketching

Sketching here is a kind of compression. The main idea is that we want each fusion node to fit in a single machine word, but if we have k children, then we have $k-1$ items, which takes $(k-1)w$ bits. No matter what k is, this will take more than one machine word. We want some compact way to store this information in one word—we'll store *sketches* of all of the keys in the node.

Suppose $u = 16$, so $w = 4$. Further, suppose we have the elements 0000, 0010, 1100, and 1111 in a node. In a perfect

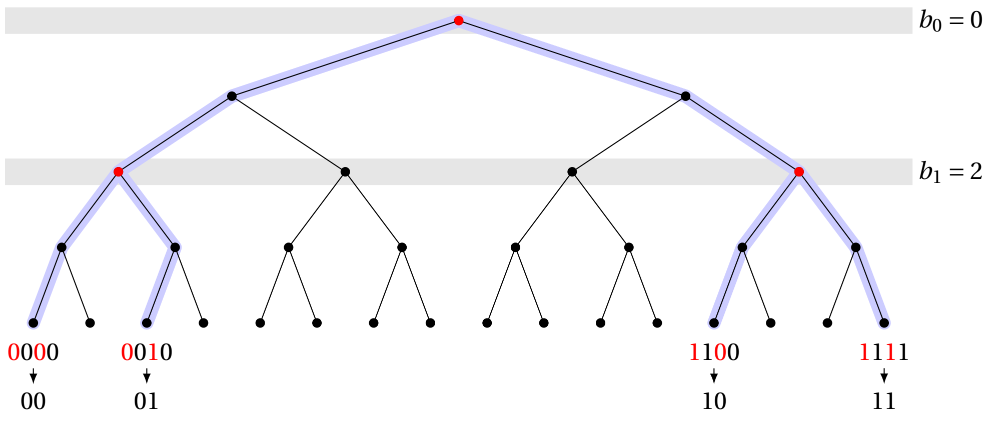


Figure 9.1: Perfect binary tree to compute sketches

binary tree (Fig. 9.1), these would correspond to 4 leaves. Looking at the tree we take to get to these leaves, we'll consider only the bits where there are branches in this tree. Here, the branch bits are the first and third bits.

This means the sketch of 0000 is 00, the sketch of 0010 is 01, the sketch of 1100 is 10, and the sketch of 1111 is 11. We'll use these sketches to perform more efficient computations on the word level.

How many branch bits are there? Each time we branch, we'll get a new leaf, so this is strictly less than $k = w^{\frac{1}{5}}$ (it could be the case that two branches coincide in the same bit, as in Fig. 9.1, so there could be fewer than k branch bits). Now, we have hope—just concatenating the sketches in a word will use $w^{\frac{2}{5}}$ space, which fits in a word.

Here, also note that if $x_0 < x_1 < \cdots < x_{k-1}$, then $\text{sk}(x_0) < \text{sk}(x_1) < \cdots < \text{sk}(x_{k-1})$. Namely, the ordering of elements is preserved when we look at the sketches.

To find the predecessor of q, the most natural idea is to take $\text{sk}(q)$, and find the predecessor of $\text{sk}(q)$. However, the sketching was based on the branches of x_i's, not of q, so we may not find the correct predecessor—q could have additional branch bits.

For example, let $q = 0101$. Here, we'd have $\text{sk}(q) = 00$, so the predecessor would be chosen to be $\text{sk}(x_0) = 00$, whereas its true predecessor is 0010 (see Fig. 9.2).

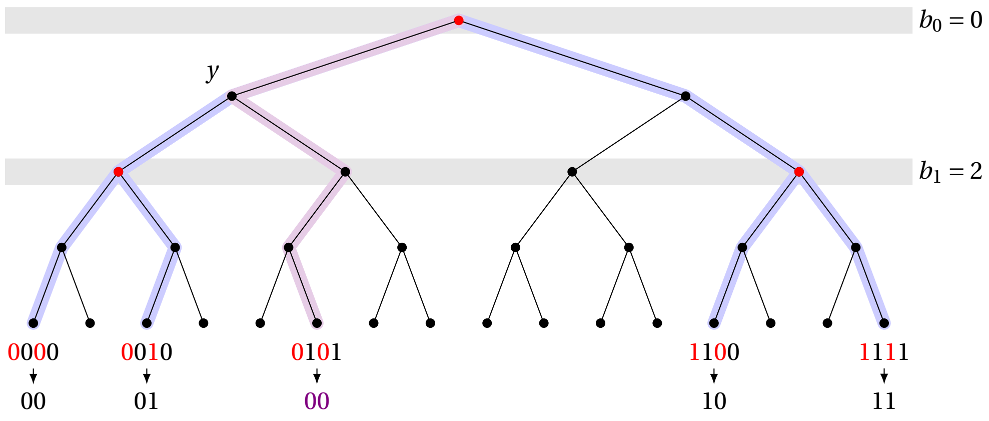


Figure 9.2: Searching for the predecessor of $q = 0101$

To fix this, suppose $\text{sk}(x_i) = \text{pred}(\text{sk}(q))$, and we'll look at $\text{sk}(x_i)$ and $\text{sk}(x_{i+1})$. (That is, we'll look at the two elements we *think* q is between).

Now, we'll look at the highest point in which the two paths diverge, and we'll call this node y. At this point, we'll remember the substring we took to get to y (in our example above, it'd be 0, since we took the left branch from the root), and we'll also remember which direction we exited y from to get to q (in the example above, we exited y to the right). If we exited y to the left, we'll form a new word $e = y10000\cdots$; if we exited y to the right, we'll form a new word $e = y01111\cdots$.

The claim is that if we search for $\text{pred}(\text{sk}(e))$, then we'll find the correct child branch. Note that we can create e in constant time if we know y (i.e. bit shift and potentially subtract 1), but how do we compute y in constant time? We can get this node from x_i and x_{i+1} via $\max(\text{MSSB}(x_i \oplus q), \text{MSSB}(x_{i+1} \oplus q))$, using the subroutine to find the most significant set bit in $O(1)$ time. Here, $x_i \oplus q$ gives the bits where x_i and q are different; the left-most bit where x_i and q are different tells us what level we branch.

9.2.2 Word-level Parallelism

How do we actually compute the predecessor of q within the node in constant time? In preprocessing, we'll calculate all branch bits b_i for each node (here, $b_0 < \cdots < b_{r-1}$ for $r < k$ denoting the number of branch bits for the node).

We'll also compute

$$\text{sk}(\text{node}) = \boxed{1\text{sk}(x_0)}\boxed{1\text{sk}(x_1)}\boxed{1\text{sk}(x_2)}\cdots.$$

There are k elements, and each chunk takes $r < k$ bits, so this takes at most $k^2 = w^{\frac{2}{5}}$ bits, which still fits in a word.

If we know $\text{sk}(q)$, we can multiply by $\boxed{0\cdots01}\boxed{0\cdots01}\boxed{0\cdots01}\cdots$ (i.e. r 0's followed by a 1, repeated) to get

$$\text{sk}(q_{node}) = \boxed{0\text{sk}(q)}\boxed{0\text{sk}(q)}\boxed{0\text{sk}(q)}\cdots.$$

If we now subtract the two quantiles, we get something like

$$\text{sk}(\text{node}) - \text{sk}(q_{node}) = \boxed{0\cdots}\boxed{0\cdots}\cdots\boxed{1\cdots}\boxed{1\cdots}.$$

In particular, we have some sequence of chunks starting with a 0 (the rest of the chunk has bits we don't care about), followed by some sequence of chunks starting with a 1. Once we mask the bits we don't care about, we end up with a word of the form

$$\text{mask}(\text{sk}(\text{node}) - \text{sk}(q_{node})) = \boxed{00\cdots0}\boxed{00\cdots0}\cdots\boxed{10\cdots0}\boxed{10\cdots0}.$$

The predecessor of q is now the chunk containing the most significant set bit. However, we can compute this number without using the MSSB routine we'll talk about later, utilizing the fact that the bits are monotone. Here, we only need to count the number of 1's in the word, as this indicates how many chunks from the right we need to look.

If we mask the bits we don't care about with a 0, and multiply by $\boxed{0\cdots01}\boxed{0\cdots01}\boxed{0\cdots01}\cdots$ once more, and look at the left most block in the product (by shifting right and ignoring everything to the left of the original leftmost block); this will be the count of the number of 1's. This is because each 1 in $\text{mask}(\text{sk}(\text{node}) - \text{sk}(q_{node}))$ contributes a 1 in this left-most block (with some other bits we don't care about to the left), and we end up adding up all of these 1's in the multiplication.

Example 9.1

For example, in our prior example of $x_0 = 0000$, $x_1 = 0010$, $x_2 = 1100$, and $x_3 = 1111$, if we want to find the predecessor of $q = 1000$ (here, $\text{sk}(q) = 10$), we'd have

$$\begin{aligned}
\text{sk}(\text{node}) &= \boxed{100}\boxed{101}\boxed{110}\boxed{111} \\
\text{sk}(q_{node}) &= \boxed{010}\boxed{010}\boxed{010}\boxed{010} \\
\text{sk}(\text{node}) - \text{sk}(q_{node}) &= \boxed{010}\boxed{011}\boxed{100}\boxed{101} \\
\text{mask}(\text{sk}(\text{node}) - \text{sk}(q_{node})) &= \boxed{000}\boxed{000}\boxed{100}\boxed{100} \\
\text{mask}(\text{sk}(\text{node}) - \text{sk}(q_{node})) \cdot \boxed{001}\boxed{001}\boxed{001}\boxed{001} &= 1\boxed{010}\boxed{010}\boxed{010}\boxed{001}00
\end{aligned}$$

9.3 Approximate Sketching

One last issue is to sketch q; naively, we'd compute $B = \sum_i 2^{b_i}$, and mask q with an AND. However, this ends up giving us a lot of extraneous 0's between the branch bits, and it's still a w-bit number, not r-bit. We were able to sketch x_i's in preprocessing using as much time as we want, but we don't have the luxury here to spend that much time on sketching q.

The idea is to settle for imperfect sketching, i.e. we allow for some extra 0's in between the bits we care about, but not too many.

We claim that given $x_0 < x_1 < \cdots < x_{k-1}$, and branch bits $b_0 < \cdots b_{r-1}$ for $r < k$, there exists a number $m \in \{0, \ldots, u-1\}$, which can be written as $m = \sum_{i=0}^{r-1} 2^{m_i}$ such that:

1. For all $(i, j) \neq (i', j')$, $m_i + b_j \neq m_{i'} + b_{j'}$.
2. $m_0 + b_0 < m_1 + b_1 < m_2 + b_2 < \cdots$
3. $m_{r-1} + b_{r-1} - (m_0 + b_0) = O(r^4)$.

The idea is that there are only r branch bits, so a perfect sketch would use r bits to store $\text{sk}(x_i)$. However, we can settle for an imperfect sketch with $x_i m$. We'll mask by $m_0 + b_0$, $m_1 + b_1$, etc., and shift right, so that it fits in r^4 bits, which is still less than a word.

This means that our new sketch is

$$\tilde{\text{sk}}(x) = \left(((x\&B)\cdot m)\&\sum_i 2^{m_i+b_i}\right) \gg (m_0+b_0)$$

Here, $x\&B = \sum_{i=0}^{r-1} x_{b_i} 2^{b_i}$, so if we multiply by m, we have

$$(x\&B)\cdot m = \sum_{i=0}^{r-1}\sum_{j=0}^{r-1} x_{b_i} 2^{b_i+m_j}.$$

By item (1), none of the exponents are equal to each other, so there are no carries.

After taking the bitwise-AND with $\sum_i 2^{m_i+b_i}$, we no longer care when $i \neq j$, so we end up with $\sum_{i=0}^{r-1} x_{b_i}\cdot 2^{b_i+m_i}$.

The least significant bit we care about is at bit m_0+b_0, and the most significant bit we care about is at bit $m_{r-1}+b_{r-1}$, so we can right shift by m_0+b_0 to get rid of the lower significant bits we don't care about.

Now, we can prove the claim. We first find $m_0',\dots,m_{r-1}'$ such that (1) holds under mod r^3. That is,

$$\forall i,j \neq (i',j'), m_i'+b_j' \neq m_i'+b_j' \pmod{r^3}.$$

We can find these greedily. That is, $m_0'=0$; suppose we've found $m_0',\dots,m_i'$ and we want to set m_{i+1}'. We want to make sure that we don't introduce any violations of the inequality mod r^3 with m_{i+1}'. That is, we don't want $m_{i+1}' = m_t' + b_{j'} - b_j$ for any j, j', or $t \le i$. Around ir^2 values that are not allowed, so we can pick any of the $r^3 - ir^2$ values that are still allowed.

To satisfy (2) and (3), we'll define $m_i \coloneqq m_i' + 2ir^3 + \lfloor w - b_i \rfloor_{r^3}$, where the floor rounds down to the nearest multiple of r^3.

After this transformation, the m_i+b_i's will occupy different blocks of size $2r^3$. Since there are r of these m_i+b_i values, and each block has size $2r^3$, this will give a total size of $O(r^4)$. Repeating this k times for sk(node), this gives a $O(r^5) = O(w)$ size result, which fits in a constant number of machine words.

2/16/2023

Lecture 10

Fusion Tree Wrap-up, Hashing

As a brief recap of last lecture, we have n keys $x_1,\dots,x_n$, put as leaves of a k-ary tree. In particular, each internal node of the tree has $\Theta(k)$ children. Starting at the root, we need to search each node to find the predecessor in the node to find which branch to go down. In fusion trees, each internal node can be searched through in constant time, which gives us $O(\log_k n)$ time.

For a particular node, we use the sketch of a node (using only bits at branch locations) and perform bitwise operations on the concatenation of all elements in the node. The result of the bitwise operations is a sequence of chunks of 0's followed by a sequence of chunks with an initial 1. Counting the number of 1's gives us the predecessor.

The issue is that we can't calculate the ideal sketch in constant time; as such, we settle for an approximate sketch, allowing for some 0's between the branch bits, but not too many (i.e. $O(r^4)$ bits long instead of $O(r)$).

10.1 MSSB in Constant Time

The last piece to fusion trees is to find the most significant set bit in $O(1)$ time.

The trivial solution is a linear search in $O(w)$ time, and binary search takes $O(\log w)$ time. Realistically, in a 64-bit computer this would only take $\log(64) = 6$ time, but this will be more than 6 operations, so it's probably not something that is realistic.

The idea is to break up the bits of x into $\sqrt{w}$ blocks, each of size $\sqrt{w}$. We'll first find the most significant block that is nonzero in constant time, and we'll then find the most significant set bit in that block in constant time.

To do the first task, our goal is to make a 1 in the first position in the block if there are any nonzero bits in the block, and have a 0 otherwise.

Suppose we define

$$F = \boxed{10\cdots0}\boxed{10\cdots0}\cdots\boxed{10\cdots0}.$$

If we compute $x\&F$, this tells us whether the first bit of each block is a 0.

Consider $F-(x\&\neg F)$. Here, $x\&\neg F$ zeroes out all of the first bits, and subtracting $F-(x\&\neg F)$ is essentially what we did before with sketches. In particular, this gives us a 0 if there were some carrying, and a 1 if no carrying happened. However, we want the opposite of this (1 if some carrying happened, 0 otherwise, as carrying indicates there were some nonzero bits in the rest of the block), so we'll take the inverse $\neg(F-(x\&\neg F))$. To zero out the rest of the junk, we have $\neg(F\&(F-(x\&\neg F)))$, telling us whether there are any nonzero bits in the rest of the block.

This means that

$$(x\&F) \mid (\neg(F\&(F-(x\&\neg F))))$$

gives us a 1 in the first bit of the block is nonzero, and a 0 otherwise. This gives us something of the form

$$z = \boxed{{}^1\!/\!_0 0\cdots0}\boxed{{}^1\!/\!_0 0\cdots0}\cdots\boxed{{}^1\!/\!_0 0\cdots0}.$$

We claim that there exists a m in this specific case (where all the bits we care about are perfectly spaced), in which case

$$((z\cdot m)\&\text{mask}) \gg \text{shift} = \boxed{00\cdots0}\boxed{{}^1\!/\!_0{}^1\!/\!_0\cdots{}^1\!/\!_0}.$$

Suppose the last block is g. We can now do a comparison with

$$c = \boxed{1100\cdots000}\boxed{1010\cdots000}\cdots\boxed{1000\cdots100}\boxed{1000\cdots010}\boxed{1000\cdots001}$$

(that is, block i has a 1 followed by a 1 in the ith position afterward) and $\boxed{0g}\boxed{0g}\cdots\boxed{0g}$.

We can do this comparison with the subtraction and counting the number of 1's in the result, like we've done before.

This will find the first block that is nonzero, but once we've found this block, we can take this block as g and do the same exact parallel comparison.

10.2 Hashing

Hashing is used for many different things; the dictionary problem, load balancing, among others.

10.2.1 Hashing with Chaining

In hashing with chaining, we have a random hash function $h : [u] \to [m]$, and we have an array of length m, where each element is a pointer to a linked list. Each linked list contains elements that have the same hash.

To query for the value for a key k, we hash k, and iterate through the linked list for the associated node.

The claim is that for all x, the expected time to query x is $O(1+\frac{n}{m})$. As long as m is linear in n, this gives expected constant time query.

There are a few reasons why people prefer to implement dictionaries in other ways. The memory for the linked list is allocated arbitrarily, so this is not cache friendly. Further, what if we care about worst-case time instead of expected time? If we don't care about space, then just a single array would work.

In the static case, there is actually a linear space worst case constant time solution, but in the dynamic setting, it's still an open problem whether we can get worst case constant time for all operations, with linear space.

10.2.2 Linear Probing

Here, we'll still have a hash function $h : [u] \to [m]$, and we still have an array of length m. Here, we'll store items directly in the array.

If we want to insert x into the array, we hash to get $h(x)$ to index into the array, and if there is already an element there, we walk down the array until we find an empty spot. Querying has the same process, where we hash and walk along the array until we find x.

If n gets close to m, then this doesn't work, and becomes very inefficient. In practice, we'd reallocate a new size array for the items when n gets close to m. Further, deletions also cause issues, and this requires additions to the data structure (called tombstones).

Linear probing was introduced in the 1950s, and Knuth in 1963 [Knu63] showed that if $m = (1+\varepsilon)n$, the expected query time is $\Theta\left(\frac{1}{\varepsilon^2}\right)$.

This solves the problem of cache-friendliness, but the query time is still a random variable and we bound the expectation.

10.2.3 Load Balancing

Load balancing is another hashing application. Here, we have m machines and n jobs to assign to machines, and we don't want any machine to be too overloaded. Ideally, each machine would get $\frac{n}{m}$ jobs in a perfect balance.

If load balancing were centralized, then we can look at all machines and assign the incoming jobs to the machine with the least jobs. However, we'd like to do this individually, where each job can compute where they need to go.

A simple solution is to send each job to a uniformly random machine. We want, for example, that the probability that the max load is large is small.

To prove this, we need the Chernoff bound.

Theorem 10.1: Chernoff Bound

Suppose $X_1, \dots, X_n$ are independent random variables, each in $\{0,1\}$, where $\mathbb{E}[X_i] = p_i$. We define $X = \sum_i X_i$, which means $\mu = \mathbb{E}[X] = \sum_{i=1}^n p_i$.

Then, for all $\varepsilon > 0$,

$$\mathbb{P}\big(X > (1+\varepsilon)\mu\big) < \left(\frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}}\right)^{\mu}.$$

Proof. The proof of the Chernoff bound relies on Markov's inequality; if Z is a nonnegative random variable, then for all $\lambda > 0$, we have

$$\mathbb{P}(Z > \lambda) < \frac{\mathbb{E}[Z]}{\lambda}.$$

The Chernoff bound is an example of a "concentration" inequality; that is, we want to say that a random variable "concentrates" around its mean.

We have

$$\mathbb{P}\big(X > (1+\varepsilon)\mu\big) = \mathbb{P}\big(e^{tX} > e^{t(1+\varepsilon)\mu}\big)$$

for any $t > 0$. We can optimize t later to get the best bound we can.

By Markov, we have a bound

$$\mathbb{P}\big(e^{tX} > e^{t(1+\varepsilon)\mu}\big) < e^{-t(1+\varepsilon)\mu} \cdot \mathbb{E}\big[e^{tX}\big].$$

Here, $\mathbb{E}\left[e^{tX}\right]$ is the moment generating function (MGF) of X. Looking at this MGF, we have

$$\begin{aligned}
\mathbb{E}\left[e^{tX}\right] &= \mathbb{E}\left[e^{\sum_{i=1}^n tX_i}\right] \\
&= \mathbb{E}\left[\prod_{i=1}^n e^{tX_i}\right] \\
&= \prod_{i=1}^n \mathbb{E}\left[e^{tX_i}\right] && (X_i \text{ indep.}) \\
&= \prod_{i=1}^n \left((1-p_i)\cdot 1 + (p_i)\cdot e^t\right) \\
&= \prod_{i=1}^n \left(1 + p_i(e^t - 1)\right) \\
&\le \prod_{i=1}^n e^{p_i(e^t-1)} && (1+z \le e^z) \\
&= \exp\left(\sum_{i=1}^n p_i(e^t-1)\right) \\
&= \exp\left(\mu(e^t-1)\right)
\end{aligned}$$

Plugging into our earlier Markov bound, we have

$$\begin{aligned}
\mathbb{P}\left(e^{tX} > e^{t(1+\varepsilon)\mu}\right) &< e^{-t(1+\varepsilon)\mu}\cdot e^{\mu(e^t-1)} \\
&= e^{\mu(e^t-1-t(1+\varepsilon))}
\end{aligned}$$

If we minimize this quantity, this is equivalent to finding the minimum of the exponent; differentiating, we want to find

$$\begin{aligned}
e^t - (1+\varepsilon) &= 0 \\
t &= \ln(1+\varepsilon)
\end{aligned}$$

(We'd also need to take the second derivative to confirm that this is a minimizer and not a maximizer.) Plugging this in, we have

$$\mathbb{P}\left(e^{\ln(1+\varepsilon)X} > e^{\ln(1+\varepsilon)(1+\varepsilon)\mu}\right) < e^{-\ln(1+\varepsilon)(1+\varepsilon)\mu}\cdot e^{\mu(e^{\ln(1+\varepsilon)}-1)}$$

□

To digest the RHS, we have two cases:

- $\varepsilon \gg 1$: Here, the denominator is larger than the numerator, which gives a bound of approximately $\varepsilon^{-\varepsilon\mu}$.
- $\varepsilon \ll 1$: Here, we have
$$\frac{e^\varepsilon}{(1+\varepsilon)^{1+\varepsilon}} = \frac{e^\varepsilon}{e^{(1+\varepsilon)\ln(1+\varepsilon)}} \approx \frac{e^\varepsilon}{e^{\varepsilon-\frac{\varepsilon^2}{2}+\varepsilon^2}-\frac{\varepsilon^3}{2}}.$$

  Here, the last bound comes from the Taylor expansion. the $\frac{\varepsilon^3}{2}$ is a very small term, so we can ignore it; this gives
$$\frac{e^\varepsilon}{(1+\varepsilon)^{1+\varepsilon}} \approx \frac{1}{e^{\frac{\varepsilon^2}{2}}},$$

  or a probability bound of $e^{-\varepsilon^2\mu/2}$.

With regard to load balancing, suppose we fix a particular machine. We'll define

$$X_i = \begin{cases} 1 & \text{if } i\text{th job maps to the machine} \\ 0 & \text{otherwise} \end{cases}.$$

The load is then $\sum_{i=1}^n X_i$. p_i is then the probability we map to the machine, so $\mu = \frac{m}{n}$. In the special case where $m = n$, then the expected load is $\mu = 1$.

The Chernoff bound for $(1+\varepsilon)\mu = C\frac{\log n}{\log\log n} = k$ (here, $\varepsilon \gg 1$) gives an approximate bound of k^{-k}.

If we want $k^k = t$, then we should choose $k = C\frac{\log t}{\log\log t}$. (This implies the probability is bounded by approximately $\frac{1}{n^{10}}$.)

Why is this true? Suppose we define $f(x) = x^x$, and $f(x_1) < t$ and $f(x_2) > t$. The x such that $f(x) = t$ must then be in the range $[x_1, x_2]$. If we pick $x_1 = 0.00001\frac{\log t}{\log\log t}$ and $x_2 = 10000\frac{\log t}{\log\log t}$, then the first is less than t and the second is larger than t. This means that there is some C in between that makes $k^k = t$.

By the union bound, the probability that *some* machine has a high load is $n \cdot \frac{1}{n^{10}} = \frac{1}{n^9}$, which means that with high probability, no machine has high load.

2/21/2023

Lecture 11

k-wise Independent Families, Linear Probing

Recall the load balancing problem from last time; we hashed n jobs to m servers, and proved that the probability the maximum server load is larger than $C\frac{\log n}{\log\log n}$ is small for $n = m$, using the Chernoff bound.

This same analysis also applies to hashing with chaining. Recall that in hashing with chaining, we have an array of linked lists containing all elements with the same hash. The expected time to query in a hash table with chaining is constant as long as n is close to m. However, what is the probability that no linked list is large? This is the same analysis as with load balancing.

However, if we wanted to have a perfectly random hash function, the most straightforward way to store this hash function is to have an array of length U with all the numbers for what $h(x)$ evaluates to. However, this is pretty bad space, ideally, we want statements like the above about the worst case server load or worst case linked list length, even with pseudorandom hash functions using $o(U)$ memory to represent.

11.1 Hash Families

It turns out that we don't actually need to use the Chernoff bound in our load balancing analysis. Note that

$$\begin{aligned}
\mathbb{P}(\exists \text{ server with load} \geq \lambda) &= \mathbb{P}\left(\bigvee_{i=1}^m (\text{server } i \text{ has load} \geq \lambda)\right) \\
&\leq \sum_{i=1}^m \mathbb{P}(\text{server } i \text{ has load} \geq \lambda) && \text{(union bound)} \\
&= n \cdot \mathbb{P}(\text{server 1 has load} \geq \lambda) && (m = n) \\
&= n\mathbb{P}\big(\exists \text{ set } T \text{ of } \lambda \text{ jobs mapping to } 1\big) \\
&= n \sum_{\substack{T\subseteq[n] \\ |T|=\lambda}} \mathbb{P}\big(\text{all jobs in } T \text{ map to } 1\big) \\
&= n \cdot \binom{n}{\lambda} \cdot \frac{1}{n^\lambda} && \left(\binom{n}{\lambda} \text{ sets, each } \frac{1}{n^\lambda} \text{ probability}\right) \\
&\leq n \cdot \left(\frac{en}{\lambda}\right)^\lambda \cdot \frac{1}{n^\lambda} && \text{(Stirling's inequality)}
\end{aligned}$$

$$\leq n \cdot \left(\frac{e}{\lambda}\right)^{\lambda}$$

This is the same thing we wanted last time, and this quantity is very small if $\lambda = C\frac{\log n}{\log\log n}$.

Notice that this does not require independence; we only used the fact that all mappings of jobs in T (for all T) are independent. In particular, this requires only λ-wise independence.

We'll talk about what λ-wise independence means next, but it turns out that we only require λ space rather than U space to store this hash function; that is, we need only $C\frac{\log n}{\log\log n}$ space to store the hash function.

Definition 11.1: k-wise independence

Random variables $Y_1, \dots, Y_n$ are *k-wise independent* if for all subsets $Y_{i_1}, \dots, Y_{i_k}$ of k of the RVs, and all values $y_1, \dots, y_k$ of these random variables,

$$\mathbb{P}\left(\bigwedge_{j=1}^{k} Y_{i_j} = y_j\right) = \prod_{j=1}^{k} \mathbb{P}\left(Y_{i_j} = y_j\right).$$

Note that k-wise independence implies $(k-1)$-wise independence: given a subset of $k-1$ RVs, we can just pick some other RV Y_k to get

$$\begin{aligned}\mathbb{P}\left(\bigwedge_{j=1}^{k-1} Y_{i_j} = y_j\right) &= \sum_{y} \mathbb{P}\left(\bigwedge_{j=1}^{k-1} Y_{i_j} = y_j \wedge Y_{i_k} = y\right) \\ &= \sum_{y} \prod_{j=1}^{k-1} \mathbb{P}\left(Y_{i_j} = y_j\right) \cdot \mathbb{P}(Y_{i_k} = y) \\ &= \prod_{j=1}^{k-1} \mathbb{P}\left(Y_{i_j} = y_j\right) \cdot \sum_{y} \mathbb{P}(Y_{i_k} = y) \\ &= \prod_{j=1}^{k-1} \mathbb{P}\left(Y_{i_j} = y\right) j)\end{aligned}$$

Definition 11.2: Hash Family

A *hash family* is a set of functions mapping $[U] \to [m]$.

Definition 11.3: k-wise Independent Hash Family

A hash family is k-wise independent if for an h chosen uniformly at random from the hash family, the random variables $h(0), h(1), \dots, h(U-1)$ are k-wise independent.

That is, if we look at any k elements in the domain, they look like they are hashing to independent places.

How do we get k-wise independent hash families? Observe that specifying a hash function h in a hash family $\mathcal{H}$ takes $\log_2 |\mathcal{H}|$ bits; we can just label each each element with a number from 1 to $|\mathcal{H}|$. If we want to minimize space, this means that the goal is to make $|\mathcal{H}|$ as small as possible.

One example is to take $\mathcal{H}$ as the set of all functions mapping $[U] \to [m]$. If we pick $h \in \mathcal{H}$ uniformly at random, we claim that this is equivalent to perfectly random hashing. This is because for any given element, the probability that we map some input x to an output y is $\frac{1}{m}$ (this can be rigorously shown with conditional probability); put another way, the probability that we get some specific mapping of $[U] \to [m]$ is $\frac{1}{m^U}$, which is the same as just randomly hashing each element.

This takes $\log|\mathcal{H}| = U \log m$ space, which is the same amount of space it takes to just store each mapping in an array (i.e. U entries, $\log m$ bits per entry).

To get better, suppose $U = m = p$ for p prime. We can then define

$$\mathcal{H}_{\text{poly}(k)} = \left\{ h(x) : h(x) = \sum_{i=0}^{k-1} a_i x^i \pmod{p} \right\}.$$

Here, there are p possibilities for each of the k coefficients, and we have $\left|\mathcal{H}_{\text{poly}(k)}\right| = p^k = U^k$, so $\log\left|\mathcal{H}_{\text{poly}}(k)\right| = O(k \log m)$. We also claim that this is k-wise independent.

To show k-wise independence, suppose we take indices $i_0, \dots, i_{k-1}$, and values $y_0, \dots, y_{k-1} \in [m]$. We want to show that

$$\mathbb{P}_{h \in \mathcal{H}_{\text{poly}(k)}}\left(\bigwedge_{j=0}^{k-1} h(i_j) = y_j \right) = \prod_{j=0}^{k-1} \mathbb{P}\big(h(i_j) = y_j\big) = \frac{1}{p^k}.$$

Here, we have that

$$\mathbb{P}_h\left(\bigwedge_{j=0}^{k-1} h(i_j) = y_j \right) = \frac{\left|h \text{ such that } h(i_j) = y_j \text{ for all } j\right|}{\left|\mathcal{H}_{\text{poly}(k)}\right|} = \frac{1}{\left|\mathcal{H}_{\text{poly}(k)}\right|} = \frac{1}{p^k}.$$

Here, this is because a polynomial of degree $k-1$ is uniquely determined by k points in a field; we give k points on the polynomial h, so there is only one h that satisfies these constraints.

However, we don't really care when $U = m$, as we're usually mapping some large universe into a much smaller range. To fix this, suppose that for any U and m, and some prime $p \gg U$, we define

$$\hat{\mathcal{H}} = \left\{ h(x) : h(x) = \left(\sum_{i=0}^{k-1} a_i x^i \pmod{p} \right) \pmod{m} \right\}.$$

However, since m doesn't evenly divide p, this doesn't exactly match the definition of k-wise independence; some values will wrap around near the end of the range, making some values in $[m]$ very slightly more probable. However, if we make p large enough, this difference is very small, which is good enough for our use cases.

We could also solve this issue differently by working over a finite field of size 2^r for some r (there always exists such finite fields), though this tends to be slower on computers, as operations under such finite fields require more complex algebra.

11.2 Linear Probing

Here, we'll analyze linear probing, first with full independence, then with 7-wise independence, then with 5-wise independence. Note that the only difference between the 7-wise independence and 5-wise independence is just the constant factor. In fact, we can't do better than 5-wise independent hash families; Patrascu, Thorup in 2010 [PT15] shows that there exists a 4-wise family such that the expected query time is $\omega(1)$, i.e. not $o(1)$.

Recall that with linear probing, we have a hash function $h : [U] \to [m]$ and a hash table. We store an item y into index $h(y)$, and we do a linear search to the right if this spot is taken. We want to find a bound on the runtime for linear probing.

Definition 11.4: Full Interval

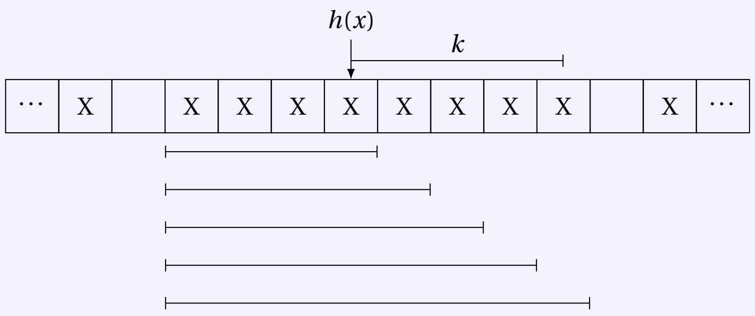
An contiguous interval $I \subseteq [m]$ is *full* if the number of keys hashing to I is at least $|I|$.

Lemma 11.5

Suppose `query(x)` took k steps. Then, $h(x)$ is contained in at least k full intervals of all different lengths.

Proof. Suppose we look at $h(x)$. We can't just take the intervals starting from $h(x)$, since there is no guarantee that the intervals are actually full (some element earlier in the array could have been put in the interval instead, so it is possible that the number of keys actually hashing to the interval is less than the size of the interval).

However, we can go backward to the first empty cell prior to $h(x)$, and start the intervals from there, since this guarantees that every item in the contiguous block must hash into the interval:



□

Last time, we saw that if $m = (1+\varepsilon)n$, then the expected query time is $\Theta(\frac{1}{\varepsilon^2})$, showed by Knuth in the 60s.

For today, suppose we look at the case where $m = 2n$. Here, we have that

$$\text{number of probes to query } x \le \sum_{k=1}^{\infty} \mathbf{1}\{\exists \text{ length } k \text{ full interval containing } h(x)\}.$$

This is because the number of distinct length full intervals containing $h(x)$ is an upper bound on the time it takes to probe k.

Taking the expectation, we have

$$\begin{aligned}
\mathbb{E}\big[\text{number of probes to query } x\big] &\le \sum_{k=1}^{\infty} \mathbb{E}\big[\mathbf{1}\{\exists \text{ length } k \text{ full interval containing } h(x)\}\big] \\
&= \sum_{k=1}^{\infty} \mathbb{P}\big(\exists \text{ length } k \text{ full interval containing } h(x)\big) \\
&\le \sum_{k=1}^{\infty} \mathbb{P}\left(\bigvee_{i=1}^{k} \{i\text{th length } k \text{ interval containing } h(x) \text{ is full}\}\right)
\end{aligned}$$

With the union bound, we have

$$\le \sum_{k=1}^{\infty} k \cdot \mathbb{P}\big(\text{a particular length } k \text{ interval containing } h(x) \text{ is full}\big)$$

If we fix $|I| = k$, then

$$\begin{aligned}
\mathbb{E}[\#\text{ items hash to } I] &= \mathbb{E}\left[\sum_{i=1}^{n} \{\text{item } i \text{ hashed to } I\}\right] \\
&= \sum_{i=1}^{n} \mathbb{P}(\text{item } i \text{ hashed to } I) \\
&\le \sum_{i=1}^{n} \frac{|I|}{m} = \frac{n}{m} \cdot |I| = \frac{1}{2}|I|
\end{aligned}$$

Now, using the Chernoff bound on the previous probability, we have

$$\mathbb{E}[\text{query time}] \leq \sum_{k=1}^{\infty} k \cdot e^{-\Omega(k)} = O(1),$$

since the series converges.

2/23/2023

Lecture 12

Linear Probing with k-wise Independence, Approximate Membership

12.1 Linear Probing with k-wise Independence

Last time, we showed that linear probing was expected constant time if we have perfect hashing, but we'd like to improve this to only use k-wise independent hash families. Today, we'll show that 7-wise independence suffices, and we'll improve this to use 5-wise independent hash families, which turns out to be optimal (there exist 4-wise independent hash families which do not work).

Recall that last time we showed that

$$\mathbb{E}[\text{query time}] \leq \sum_{k=1}^{\infty} k \cdot e^{-\Omega(k)} = O(1)$$

with the Chernoff bound. Today, we'll show

$$\mathbb{E}[\text{query time}] \leq \sum_{k=1}^{\infty} k \cdot O\left(\frac{1}{k^3}\right) = O(1)$$

with only 7-wise independence. To get down to 5-wise independence, we want to avoid using the union bound (there's a lot of overlap, which is wasteful). Instead, we can identify a constant number of intervals such that if some length k interval containing $h(z)$ is full, then one of these constantly many intervals is at least $\frac{3}{4}$ full. This probability can then be bounded by $O\left(\frac{1}{k^2}\right)$, which still converges.

12.1.1 Linear Probing with 7-wise Independence

We want to bound the probability that the first interval of length k containing $h(z)$ is full using 7-wise independence. Suppose we call this first interval of length k containing $h(z)$ interval I.

Let us define

$$X_i = \begin{cases} 1 & \text{if } h(i\text{th key}) \in I \\ 0 & \text{otherwise} \end{cases}$$

Here, notice that $\mathbb{E}[X_i] = \mathbb{P}(X_i = 1) = \frac{k}{m}$, since item i needs to be hashed into an interval of length k.

Let us define the *load* of I to be

$$L(I) := \sum_{i=1}^{n} X_i.$$

We want to understand

$$\mathbb{P}\left(|L(I) - \mathbb{E}[L(I)]| > \frac{k}{2}\right).$$

This upper bounds the probability that I is full, since it must have deviated by more than $\frac{k}{2}$. By (generalized) Markov, we have

$$\begin{aligned} \mathbb{P}\left(|L(I) - \mathbb{E}[L(I)]| > \frac{k}{2}\right) &\leq \left(\frac{k}{2}\right)^{-6} \mathbb{E}\left[(L(i) - \mathbb{E}[L(I)])^6\right] \\ &= \left(\frac{k}{2}\right)^{-6} \mathbb{E}\left[\left(\sum_{i=1}^{n} X_i - \frac{k}{2}\right)^6\right] \end{aligned} \tag{12.1}$$

We claim that this expectation is determined fully by 7-wise independence.

Notice that if we actually expand this sum and use expectation, each term will involve the product of at most 6 X_i's. This product will be 1 if all 6 keys hash to the same interval. There are a total of 7 keys involved in this calculation; z, and these 6 keys. However, with 7-wise independence, hashing these 7 keys behaves like they were perfectly random.

Before we continue, let us take a probability detour. Recall ℓ_p norms; we define $\|\vec{x}\|_p := (\sum |x_i|^p)^{\frac{1}{p}}$.

It turns out that we can also take ℓ_p norms of random variables; for a random variable X, we define

$$\|X\|_p := \left(\mathbb{E}\left[|X|^p\right]\right)^{\frac{1}{p}}.$$

Theorem 12.1: Minkowski's Inequality

If $p \geq 1$, then $\|X\|_p$ is a norm on the random variable X.

The reason why we want to look into norms is because $\mathbb{E}\left[Z^6\right] = \|Z\|_6^6$. While we could actually expand out the summation, and do some combinatorics to compute the expectation explicitly, this becomes very messy, but it is doable. The trick we'll see makes this a much more manageable computation.

We want to compute

$$\mathbb{E}\left[\left(\sum_{i=1}^n X_i - \frac{k}{2}\right)^6\right] = \mathbb{E}\left[\left(\sum_{i=1}^n X_i - \sum_{i=1}^n \mathbb{E}[X_i]\right)^6\right]$$

Here, we can notice that $\mathbb{E}[X_i]$ is just a constant; as such, we can define X_i' to be drawn from the same distribution as X_i, but independent with X_i. All this does is decouple the two instances of X_i in the expression, so that we can take the expectation out further; X_i is now constant with respect to the expectation over X_i'. The outer expectation will also become only an expectation over X_i:

$$\begin{aligned} &= \mathbb{E}_{\{X_i\}}\left[\left(\sum_{i=1}^n X_i - \sum_{i=1}^n \mathbb{E}_{\{X_i'\}}[X_i']\right)^6\right] \\ &= \mathbb{E}_{\{X_i\}}\left[\left(\mathbb{E}_{\{X_i'\}}\left[\sum_{i=1}^n X_i - \sum_{i=1}^n X_i'\right]\right)^6\right] \end{aligned}$$

By Jensen's inequality, since the function z^6 is convex, we know that $\mathbb{E}[X]^6 \leq \mathbb{E}\left[X^6\right]$, and we can take the inner expectation over X_i' outside of the exponent:

$$\leq \mathbb{E}_{\{X_i\}}\left[\mathbb{E}_{\{X_i'\}}\left[\left(\sum_{i=1}^n (X_i - X_i')\right)^6\right]\right]$$

Since we're now essentially taking the expectation over all X_i and X_i', this can be rewritten into a norm:

$$\begin{aligned} &= \mathbb{E}\left[\left(\sum_{i=1}^n (X_i - X_i')\right)^6\right] \\ &= \left\|\sum_{i=1}^n (X_i - X_i')\right\|_6^6 \end{aligned}$$

Suppose we look at this difference $X_i - X_i'$ as some random variable. Since X_i and X_i' could be swapped by symmetry, this means that their difference is symmetric about 0 (i.e. we can flip signs of terms in the summation whenever we want, without changing probabilities).

As such, we can define iid σ_i drawn uniformly from $\{-1,1\}$, and multiply by σ_i for each term in the summation:

$$\begin{aligned}
\left\|\sum_{i=1}^{n}\left(X_i - X_i'\right)\right\|_6^6 &= \left\|\sum_{i=1}^{n}\sigma_i\left(X_i - X_i'\right)\right\|_6^6 \\
&= \left\|\sum_{i=1}^{n}\sigma_i X_i - \sum_{i=1}^{n}\sigma_i X_i'\right\|_6^6 \\
&\le \left\|\sum_{i=1}^{n}\sigma_i X_i\right\|_6^6 + \left\|\sum_{i=1}^{n}\sigma_i X_i'\right\|_6^6 && \text{(triangle inequality)} \\
&= 2\left\|\sum_{i=1}^{n}\sigma_i X_i\right\|_6^6 && (X_i \text{ and } X_i' \text{ identical})
\end{aligned}$$

If we expand out the expectation from the norm, we have

$$\mathbb{E}\left[\left(\sum_{i=1}^{n}\sigma_i X_i\right)^6\right] = \sum_{i_1,i_2,\dots,i_6}\left(\mathbb{E}\left[X_{i_1}\cdots X_{i_6}\right]\right)\cdot\left(\mathbb{E}\left[\sigma_{i_1}\cdots\sigma_{i_6}\right]\right)$$

In the expectation for the product of σ_i's, if we collect the σ_i's with the same index within this term, we'd have terms of the form $\mathbb{E}\left[\sigma_{j_1}^{d_1}\right]\mathbb{E}\left[\sigma_{j_2}^{d_2}\right]\cdots$.

Since σ_i is drawn from $\{1,-1\}$, the expectation when the exponent is odd is zero (since we have the expectation of an odd function), which makes the entire term 0. Otherwise, if we have an even exponent d, $\sigma_i^d = 1$, which is 1.

This essentially filters all products with an odd multiplicity of some σ_i, leaving only products where all terms have even multiplicity.

This leaves a couple cases: everything has multiplicity 2, or one has multiplicity 4, or one has multiplicity 6.

- If everything has multiplicity 2, then there are $O(n^3)$ different terms of the form $\mathbb{E}\left[X_{i_1}^2 X_{i_2}^2 X_{i_3}^2\right]$. Further, since we have 7-wise independence, the expectation is $\left(\frac{k}{m}\right)^3$, since the exponents don't change the expectations.

  These terms give a total of $O\left(n^3\left(\frac{k}{m}^3\right)\right)$ in the summation.

- If one variable has multiplicity 2 and another has multiplicity 4, there are $O(n^2)$ different ways to choose the X_i's, and the expectation is $\left(\frac{k}{m}\right)^2$.

  These terms give a total of $O\left(n^2\left(\frac{k}{m}\right)^2\right)$ in the summation.

- If one term has multiplicity 6, we have n ways of choosing X_i, with an expectation of $\frac{k}{m}$.

  These terms give a total of $O\left(n\cdot\frac{k}{m}\right)$ in the summation.

This simplifies to

$$\mathbb{E}\left[\left(\sum_{i=1}^{n}\sigma_i X_i\right)^6\right] = O\left(n^3\left(\frac{k}{m}\right)^3\right) + O\left(n^2\left(\frac{k}{m}\right)^2\right) + O\left(n\cdot\frac{k}{m}\right) = O(k^3),$$

since the first term dominates.

Plugging this in to our expression from Eq. (12.1), we have

$$\left(\frac{k}{2}\right)^{-6}\mathbb{E}\left[\left(\sum_{i=1}^{n}X_i - \frac{k}{2}\right)^6\right] = O(k^{-6})O(k^3) = O(k^{-3}).$$

This gives an expected query time of

$$\mathbb{E}[\text{query time}] \le \sum_{k=1}^{\infty} k\cdot O(k^{-3}) = O(1),$$

since the summation still converges.

12.2 Linear Probing with 5-wise Independence

Linear probing with 5-wise independence was first shown by Pagh, Pagh, and Ruzic [PPR09].

Here, suppose we have a perfect binary tree built on top of the array, and we consider the set of all length k intervals containing $h(z)$ (rounding up to the nearest power of 2). We can go up to the level corresponding to k (i.e. the level that divides up the array into length k chunks). If we go down two more levels, suppose we consider the number of nodes that intersect with these intervals at all.

There are $O(1)$ such nodes, since each node covers intervals of size $\frac{k}{4}$, whereas the union of all intervals containing $h(z)$ is of size $2k$.

If one of the intervals containing $h(z)$ is full, let us consider the (at most) five nodes (and intervals) that touch this interval. We claim that at least one of these subintervals is at least $\frac{3}{5}$ full if the parent interval is full.

Since we only expect any interval to be half full, we deviate from the expectation by a large amount, which happens with probability $\le O(\frac{1}{k^2})$. The same computation as before but with 5-wise independence then holds, and we can now union bound over just these constantly many subintervals. In particular, we have

$$\begin{aligned}
\mathbb{E}[\text{query time}] &\le \sum_{k=1}^{\infty} \mathbb{P}\big(\exists \text{ length } k \text{ full interval containing } h(z)\big) \\
&\le \sum_{k=1}^{\infty} \mathbb{P}(\exists \text{ subinterval almost full}) \\
&\le \sum_{k=1}^{\infty} O(1) \cdot \mathbb{P}\big(\text{particular subinterval almost full}\big) && \text{(union bound)}
\end{aligned}$$

Bounding the probability that a particular subinterval is full can be done exactly as with 7-wise independence, but now with 5-wise independence.

12.3 Approximate Membership

We've been talking about the dictionary problem so far; we're hoping to have a space with $O(nw)$ bits (i.e. linear in the number of elements, with w bits per item). Now, we're looking at the membership problem, but we can hope for $O(n)$ space (i.e. we can't even store the keys).

The membership problem has a database of keys (no values), with the following operations:

- `insert(x)`: adds x to the database
- `query(x)`: tells us whether x is in the database

In the *approximate membership* problem, we want to guarantee that if x is in the database, then we say yes with probability 1. If x is not in the database, then we say yes with probability at most ε. That is, there is some false positive rate, but no false negatives.

The goal is to use $O(n \log \frac{1}{\varepsilon})$ bits of memory. If ε is constant, this is roughly $O(n)$ bits. This is achieved using Bloom filters.

Approximate dictionary is similar, where we want to guarantee that if x is in the database, then we return the value with probability 1, and if it is not int he database, we output junk with some small probability, using $O(nr)$ bits, where r is the number of the bits in the values. This is achieved using Bloomier filters.

2/28/2023

Lecture 13

Approximate Membership, Cuckoo Hashing, Power of Two Choices

13.1 Bloom Filters

Recall the approximate membership problem; we want to maintain a set $S \subseteq [U]$ (where $n := |S|$), subject to:

- `insert(x)`: Updates S to include x, i.e. $S \leftarrow S \cup \{x\}$
- `query(x)`: Query the system to ask if $x \in S$; i.e. return true if $x \in S$, false if $x \notin S$.

Here, we want the probability of a wrong answer to be at most ε.

We'll look at one (Monte Carlo) randomized data structure to solve this problem; it'll always be efficient, but it may give the wrong answer with some probability. In particular, we'll look at Bloom filters (from the 1970s)[Blo70].

A natural question is: what's the point of approximate membership? We've already seen dynamic hashing and linear probing, which gives linear space and expected constant time operations, and we can use these solutions to the dynamic dictionary problem to solve approximate membership.

The advantage of Bloom filters is that it uses only $o(nw)$ bits of memory; we use less space than just storing the keys themselves. In particular, it uses approximately $1.44n \ln \frac{1}{\varepsilon}$ bits (the 1.44 comes from $\log_2(e)$).

The lower bound for the space complexity is $\Omega(n \ln \frac{1}{\varepsilon})$ bits of space (i.e. removing the 1.44 constant multiple), which is actually achievable in the static case; it turns out it is impossible to achieve this space for the dynamic case, shown by Lovett and Porat in 2013 [LP13].

With Bloom filters, we first initialize a bit array A of m bits to all 0's. We then pick k independent fully random hash functions mapping $[U] \to [m]$, and implement the operations as follows:

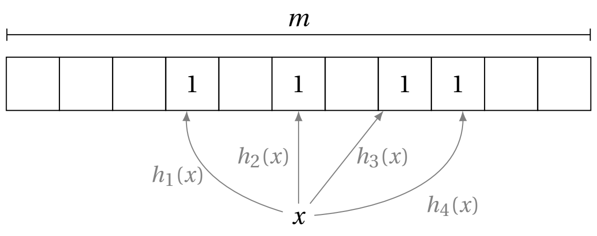
- `insert(x)`: set $A[h_i(x)] = 1$ for each i from 1 to k.
- `query(x)`: query $A[h_i(x)]$ for each i from 1 to k, and take the AND of all of them.

Note that if x was truly in the set, then all of these locations would be set to 1; otherwise, at least one of these location is probably 0, and we'd say no (there is the possibility that other elements have hashed to some of the same locations).

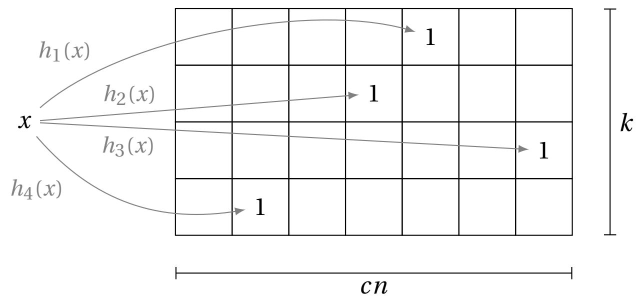
We could also analyze this algorithm to find the optimal values for m and k; the optimal value of m turns out to be $1.44n \ln \frac{1}{\varepsilon}$, and k to be $\Theta(\ln \frac{1}{\varepsilon})$.

We won't show the analysis of traditional Bloom filters here, since we run into independence problems, but we'll analyze a slightly different version, which is easier to explain.

In the traditional Bloom filter, we have an array of length $m = \Theta(n \log \frac{1}{\varepsilon})$, and we set k locations in the array to 1. (Note that we can possibly have two hash functions hashing to the same location.)



Today, we'll analyze a slightly different formulation; we have instead a 2D array with k rows and cn columns. When we insert x, we set the location $h_i(x)$ in row i to 1, for each $i = 1, \ldots, k$. Here, the crucial difference is that each row is independent from one other.



Notice that (in both versions) we never have any false negatives; we only have false positives. That is, we'll always be certain that an item is *not* in the set, but we could incorrectly say that an item *is* in the set.

Lemma 13.1

If $x \notin S$, then $\mathbb{P}\big(\text{say } x \in S\big) \leq \varepsilon$.

Proof. Suppose we fix a row i. We have a false positive if each $A[i][h_i(x)]$ is already occupied for $i = 1, \dots, k$. This means the false positive probability is $\mathbb{P}(A[i][h_i(x)] = 1)^k$. To compute this value, suppose we look at the expected number of items in S that collide under x:

$$\mathbb{E}\big[\text{\# items in } S \text{ colliding with } x \text{ under } h_i\big] = \sum_{y \in S} \underbrace{\mathbb{P}\big(h_i(x) = h_i(y)\big)}_{1/cn} = \frac{1}{c}.$$

Notice that the probability that at least 1 item collides with x is at most $\frac{1}{c}$ by Markov's inequality. This is because the random variable for the number of items that collided with x takes a value c times its expected value. (We can improve this analysis to get a better constant, but we omit it here.)

This probability we computed is the probability we get fooled in row i, so the probability we got fooled in *all* k rows is $\frac{1}{c^k}$, which we want to be at most ε. This gives $k = \Theta(\log \frac{1}{\varepsilon})$.

Notice that although we got a worse constant factor, we only needed 2-wise independence here, to reduce $\mathbb{P}\big(h_i(x) = h_i(y)\big) = \frac{1}{cn}$. $\square$

If we wanted to get a better constant multiple for the space, we can reduce to the dictionary problem; we can map to a range where there is a small probability of collisions, and then store the set of $h(x)$'s using a dictionary that uses as little space as possible. Notice that the main reason why we were able to get away with using less memory in bloom filters is because we just set k bits to 1, instead of storing x. Here, we're doing something similar, storing $h(x)$ instead of storing x in the dictionary.

13.2 Cuckoo Hashing

The goal of Cuckoo hashing [PR01] is to solve the dynamic dictionary problem. In hashing with chaining and linear probing, the query and insertion time are both expected constant time, whereas in cuckoo hashing, query time is *worst case* constant (we only look in two locations), and insertion time is expected constant. It's an open problem to get a solution to the dynamic dictionary problem that is worst case constant in both query and insertion. (It's theoretically possible to get linear space and constant time query and insertion time, deterministically, but the best way we know of is to use a balanced binary search tree.)

To implement Cuckoo hashing, we pick two fully independent hash functions $h, g : [U] \to [m]$, where m is the size of the hash table. We then implement the operations as follows.

To query x, we check both $A[h(x)]$ and $A[g(x)]$. The invariant we'll maintain is that x will always be in either $A[h(x)]$ or $A[g(x)]$. If x is in neither location, then we say that x is not in the database.

For insertion, we put x in $A[g(x)]$. If $A[g(x)]$ was null, then we're done. However, if $A[g(x)]$ is already occupied, let

x' be the old key in $A[g(x)]$ (let $j = g(x)$ for ease of notation). If $j = g(x')$, then we move x' into $h(x')$; otherwise, move x' to $A[g(x')]$ (since here $j = h(x')$). We then recurse in these two cases to insert x' if there are any additional conflicts.

Intuitively, we insert x into $A[g(x)]$, and move any existing element into their other hash location. If this element *also* conflicts with another element, we continue moving these elements until there are no conflicts. However, it is possible for this to go on forever (if there's a cycle).

To resolve this, if this process goes on for longer than say $10\log n$ steps, we rebuild the entire hash table from scratch. That is, we pick new hash functions h and g and re-insert the items. We then repeat this process until it works, i.e. until we successfully insert every element back into the hash table. The main idea is that the probability of this occurring is very tiny, so it's unlikely this process will happen or go on for to long.

(This insertion algorithm is why this algorithm is called "cuckoo hashing"; a cuckoo chick pushes other eggs or young out of the nest when it hatches, much like how this algorithm pushes existing elements out when inserting a new item.)

It's clear that query time is worst case constant, but we'll show that the query time is expected constant in a little bit.

As a little detour, cuckoo hashing can also be used to solve the static approximate dictionary problem in $O(nr)$ bits of memory.

The approximate dictionary problem is similar to approximate membership; the only difference here is that we store a set of n key-value pairs, where all the values are r-bit strings. Notice that here we'll never actually store the keys. (This is in problem set 4; a hint is to use cuckoo hashing when the cuckoo graph has no cycles, where $\mathbb{P}(\text{no cycles}) \geq \frac{1}{2}$.)

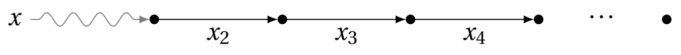
13.2.1 Cuckoo Hashing Analysis

Definition 13.2: Cuckoo Graph

A *Cuckoo graph* is a multigraph (i.e. there could be multiple edges between the same vertices) illustrating the process of Cuckoo hashing. Vertices are locations in the hash table (so there are m vertices), and edges connect between $h(x)$ and $g(x)$ for each key $x \in S$, where S is the set of keys in the database (so there are n edges).

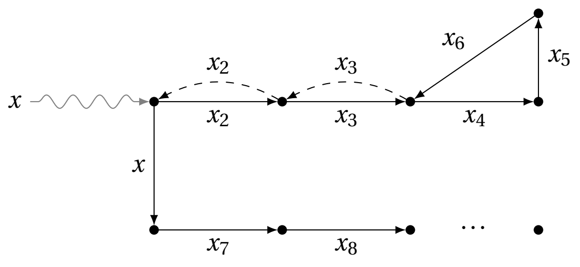
Let's look at the possible cases that can happen during an insertion.

- One case is that we have a path. (The squiggly line is the first hash of x, and other arrows denote the movement of elements due to conflicts.)



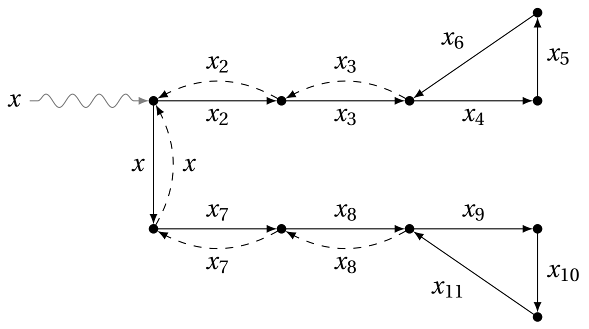
  Here, x hashes to where x_2 is originally, so x_2 moves to where x_3 is originally, and this continues until there are no collisions.

- Another case is that we have a single cycle. (Solid arrows denote the first movement of an element due to a collision, and dashed arrows denote a second movement due to another collision after the cycle.)



Here, the movement of x_6 causes x_3 to get moved once more, which propagates back to x. This means that we now try hashing x using $h(x)$, which displaces x_7, etc., until we have no collisions along this second path.

- A last case is that we have a double cycle, which does actually go on infinitely.



Here, after we go through the first cycle, we hit another cycle in the second path; this causes an infinite loop of collisions that never resolves.

For the analysis, suppose we define:

- T: the runtime to do an insert
- P_k: the indicator $\mathbf{1}\{\text{have path of length} \geq k\}$
- C_k: the indicator $\mathbf{1}\{\text{have cycle of length} \geq k\}$
- D: the indicator $\mathbf{1}\{\text{have a double cycle}\}$

Our expected runtime is

$$\mathbb{E}[T] \leq \underbrace{\mathbb{E}\left[\sum_{k=1}^{\infty} P_k\right]}_{\text{path case}} + \underbrace{\mathbb{E}\left[\sum_{k=1}^{\infty} C_k\right]}_{\text{cycle case}} + \underbrace{\mathbb{P}(D=1)}_{\text{double cycle}} \cdot \underbrace{\big(10\log n + n\mathbb{E}[T]\big)}_{\text{rebuild table}}$$
$$+ \underbrace{\mathbb{P}\big(\text{path/cycle of length} \geq 10\log n\big)}_{\text{took too long}} \cdot \underbrace{\big(10\log n + n\mathbb{E}[T]\big)}_{\text{rebuild table}}$$

We can show that $\mathbb{P}(P_k = 1) = \exp(-\Omega(k))$ for m sufficiently large (say $m = 4n$), and we'll also show that $\mathbb{P}(C_k = 1) = \exp(-\Omega(k))$. This means the sums in the first two expectations converge to a constant.

We'll also show that $\mathbb{P}(D = 1)$ and $\mathbb{P}\big(\text{path/cycle of length} \geq 10\log n\big)$ are both $O(\frac{1}{n^2})$. This simplifies the runtime to

$$\begin{aligned}
\mathbb{E}[T] &\leq O(1) + O(1) + O\left(\frac{1}{n^2}\right)\big(10\log n + n\mathbb{E}[T]\big) + O\left(\frac{1}{n^2}\right)\big(10\log n + n\mathbb{E}[T]\big) \\
\mathbb{E}[T] &\leq O(1) + O\left(\frac{1}{n}\right)\mathbb{E}[T] \\
\left(1 - O\left(\frac{1}{n}\right)\right)\mathbb{E}[T] &\leq O(1) \\
\mathbb{E}[T] &\leq O(1)
\end{aligned}$$

Lemma 13.3

$\mathbb{P}(P_k = 1) = \exp(-\Omega(k))$, in particular $\mathbb{P}(P_k = 1) \leq \frac{1}{2^k}$.

Proof. Notice that we can have a lot of possibilities for paths of length k; we have a choice of what elements are involved in the path. (We'll call each of these possibilities a "realization" of a path.)

The probability by union bound gives

$$\mathbb{P}(P_k = 1) \le \sum_{\substack{\text{all possible paths} \\ P \text{ of length } k}} \mathbb{P}(\text{have } P)$$

Notice that the number of possible realizations is at most $m^{k+1} \cdot n^k$, since there are at most m possibilities for each of the $k+1$ vertices in the path, and n possibilities for each of the k edges in the path.

The probability of a fixed realization is

$$\mathbb{P}(\text{a fixed realization}) \le \frac{1}{m} \cdot \frac{2^k}{m^{2k}}.$$

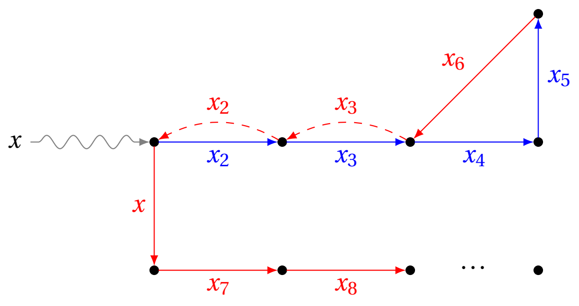
Here, the $\frac{1}{m}$ factor is because x must hash to the start of the path, and for each of the k edges along the path, we have a probability at most $\frac{2}{m^2}$ for the pair of hash functions to hash to the two locations incident to the edge.

If we fix $m = 4n$, then this simplifies to

$$\begin{aligned}\mathbb{P}(P_k = 1) &\le (\#\text{ possible realizations}) \cdot \mathbb{P}(\text{fixed realization}) \\ &\le m^{k+1} \cdot n^k \cdot \frac{2^k}{m^{2k+1}} \\ &= \left(\frac{2n}{m}\right)^k = \frac{1}{2^k}\end{aligned}$$

□

To show $\mathbb{P}(C_k = 1) \le \exp(-\Omega(k))$, we break up the cycle into two parts; edges before the cycle (in blue), and edges after the cycle (in red).



If the entire cycle is of length k, then at least one part must be of length at most $\frac{k}{2}$. We can then reuse the same analysis we did for paths to get a similar bound for cycles.

The probability of a path or cycle of length at most $\ge 10 \log n$ is similar; it's also $\exp\big(-10 \log n\big) = O(\frac{1}{n^{10}})$, which is definitely $O\left(\frac{1}{n^2}\right)$.

Lemma 13.4

$\mathbb{P}(D = 1) = O\left(\frac{1}{n^2}\right)$.

Proof. □

13.3 Preview of Power of 2 Choices

Think about hashing with chaining; the expected length of a linked list is $O(1)$, but the worst case load is on the order of $\frac{\log n}{\log\log n}$. To reduce this load, we can instead pick two random hash functions $h, g : [U] \to [m]$. We'll also augment the hash table to keep track of the length of the linked list.

When we insert x, we look at both $h(x)$ and $g(x)$, and choose the linked list with fewer items. Intuitively, this can only do better than with only one hash function. With high probability, max load turns out to be at most $\frac{\ln\ln n}{\ln 2} + \Theta(1)$. (This essentially goes from a max load of $O(\ln n)$ to $O(\ln\ln n)$.)

3/2/2023

Lecture 14

Online Algorithms

14.1 Power of 2 Choices

The power of 2 choices is due to Azar, Brodeer, Karlin, and Upfal in 1999 [ABK+99], see also a survey by Mitzenmacher, Rita, and Sitaraman [MRS01].

Here, there are two hash functions $h, g : [U] \to [n]$ (think hashing with chaining); if we do this, with high probability, the max load is $\frac{\ln\ln n}{\ln 2} + O(1)$. With d hash functions, the denominator would change to $\ln d$, so it'd be a constant improvement.

Vöcking in JACM 2003 [Vöc03] considered the case where there are d hash functions, and we split the n slots in the hash table into d blocks, each with $\frac{n}{d}$ slots. We then have a hash function for each group $(h_1, h_2, \dots, h_d)$. Whenever we see an item, we hash it to a particular slot in each bucket, and put it in the least loaded bucket, and if there are ties, we put it in the left-most slot.

Here, with high probability, the max load is $\frac{\ln\ln n}{d\cdot\varphi_d} + O(1)$, where $\varphi_1, \dots$ is a sequence of numbers all in the range $[1.61, 2]$ (the lower bound is the golden ratio). As such, slight alterations in how we use these hash functions can lead to a big difference in performance.

The rough idea of the proof is as follows.

Given an x in the database, the height of x is h if it is the hth item put in its bucket at the time of insertion. We want to consider how many people have height $1, 2, \dots$, and show a bound on the number of items with a given height after we reach some threshold.

We define B_i to be the number of slots with at least i items. We'll now look at the ratios $\frac{B_i}{n}$, i.e. the fraction of buckets with at least i items in them. When this fraction is < 1, then there are no bins with at least i items.

We know that $\frac{B_1}{n} \le 1$ (since B_1 is upper bounded by n). What is $\mathbb{E}[B_{i+1}]$? We know that

$$B_{i+1} = \sum_{x\in\text{database}} \mathbf{1}\{\text{height}(x) \ge i+1\}.$$

This is because if the load in a bucket is more than $i+1$, then we're adding one for each element in the bucket with height at least $i+1$, so this is an upper bound on the true value of B_{i+1}.

With linearity of expectation, we then have

$$\mathbb{E}[B_{i+1}] \le n \cdot \mathbb{P}\big(\text{height}(x) \ge i+1\big).$$

The next part is the hand-wavy bit. Each B_i is a random variable, and they are all related to each other. Suppose we know the values of $B_1, \dots, B_i$ (i.e. they are fixed), and we want to figure out the expectation of B_{i+1}.

We only have height$(x) \ge i+1$ if both h and g hash to a bucket with load at least i. The probability that a single hash function hashes to a bucket with load at least i is $\frac{B_i}{n}$, so the probability that both hash functions hash to a bucket of load at least i is $\frac{B_i^2}{n^2}$, giving an upper bound of

$$\mathbb{E}[B_{i+1}] \leq \frac{B_i^2}{n} \implies \frac{\mathbb{E}[B_{i+1}]}{n} \leq \left(\frac{B_i}{n}\right)^2.$$

Notice that we have to be careful in doing the analysis, since we're ignoring the randomness in previous B_i's. However, if everything works out, then the expectations decrease by the square of the previous.

A trivial upper bound on $\frac{B_2}{n}$ is $\frac{1}{2}$; this means that

$$\frac{B_3}{n} \leq \frac{1}{2^2} \implies \frac{B_4}{n} \leq \frac{1}{2^4} \implies \cdots$$

Generally, we have

$$\frac{B_{2+j}}{n} \leq \frac{1}{2^{2^j}}.$$

We want $2^{-2^j} < \frac{1}{n}$ (as here we'd have $B_{2+j} < 1$ and thus there are no buckets of load at least $2+j$), which gives $j \geq \log\log n$.

14.2 Online Algorithms

The next topic we'll cover is online algorithms, which will take us the next couple weeks.

Online algorithms involve decision making in the face of uncertainty; two example toy problems that we'll consider are (1) the pot of gold problem, and (2) the ski rental problem. The goal in these problems is to be competitive with an omniscient being who knows the future.

14.2.1 Pot of Gold

Suppose we're in a very long hallway, and there are treasure chests that are equally spaced (1 yard apart) in the hallway in both directions. One of the treasure chests has gold inside, and the others are all empty.

We can walk to a chest, and open it (the only thing we pay for is walking); the goal is to walk as few yards as possible to find the gold.

Suppose the gold is at some position t. OPT will pay $|t|$ yards, since we can just go straight to the gold.

One idea is to zig-zag between the two sides; we go to $-1 \to +1 \to -2 \to 2 \to \cdots$. This isn't quite good, since we're overlapping a lot. Instead, we could travel a little further each time, going $-1 \to 2 \to -4 \to 8 \to -16 \to \cdots$. In an amortized sense, this makes sense, since we're walking more each time, spending less time retracing our steps.

Suppose $t \in (2^m, 2^{m+1}]$. Here, we'll pay something of the form $2 \cdot (1 + 2 + 4 + \cdots + 2^k) + |t|$. If $|t| = 2^m + 1$, what can happen is that we were on the correct side, but just missed t by 1; here, in the worst case $k = m+1$.

We can bound the geometric series by $4t$, so we pay at most $9|t|$. This means that this algorithm is "9-competitive".

Definition 14.1: C-competitive Algorithm

An algorithm A is C-competitive if for all inputs σ, $\text{cost}(A(\sigma)) \leq C \cdot \text{OPT} \cdot \sigma + O(1)$.

14.2.2 Ski Rental

In the ski rental problem, we're at a ski resort, and every day you ask your friends if we should stay another day. In order to ski, we need skis, and we can either rent the skis for a day, or we can buy them. Specifically, renting skis is \$1 per day, and buying skis is \$$B$; we don't know when the vacation will end.

OPT will buy the skis on day 1 if we stay for more than B days, and will rent every day if we stay for shorter. This means that OPT pays $\min(D, B)$, where D is the number of days we stayed.

A good strategy here is to rent the first $B-1$ days, and buy on day B. The point here is that if $D < B$, then we pay the same amount as OPT. Otherwise, if $D > B$, then we'll pay roughly $2B$. As such, this is a 2-competitive algorithm.

14.3 List Update Problem

This problem is a warm-up for paging, which we'll study later.

In the list update problem, we maintain a list of items, subject to 3 operations:

- `access(x)`: We start at the beginning of the linked list, and we follow pointers until we get to x. The cost is the position of x in the list (i.e. if it's the kth item, we pay k).

  We also allow you to move x closer to the front of the list for free after you access x.

- `insert(x)`: Append x to the end of the linked list, and the cost is the length of the list.

  We allow you to move x closer to the front for free here as well.

- `del(x)`: Walk to x, and remove it. Just like access, the cost is the position of x.

Our goal is to minimize the total cost for a sequence of operations.

There are a few natural heuristics here:

1. MF (move to front): we move x to the front whenever we access x. We do the same for insert (insert x, then move it all the way to the front).
2. Transpose: after touching x (whether an insert or delete), swap it one closer to the front.
3. FC (frequency count): here, we try to keep items in decreasing order of frequency. Whenever we access x, we move it closer to the front to keep it in decreasing order of frequency.
4. SFC (static frequency count): Suppose there are no insertions or deletions; we look into the future and keep the items in decreasing order of the final frequency.

   This doesn't fit in the model, but it's something we'll compare against.

Bently and McGeockh in 1985 [BM85] showed that if the items are initially sorted by time of first access, then in the situation where there are only accesses, for all sequence of operations σ, $\text{cost}(\text{MF}(\sigma)) \le 2 \cdot \text{cost}(\text{SFC}(\sigma))$. (Here, we're also allowed to transpose any other items in the front of x when we access it for a cost of 1.)

Sleator and Tarjan in 1985 [ST85a] showed that MF is actually dynamically optimal as well (something we didn't know for splay trees and binary search trees).

Theorem 14.2

For all A and all σ, assuming A and MF start with the same ordering,

$$\text{cost}(\text{MF}(\sigma)) \le 2 \cdot \text{cost}(A(\sigma)) + P(A(\sigma)) - F(A(\sigma)) - m,$$

where $P(A(\sigma))$ is the number of non-free transpositions, and $F(A(\sigma))$ is the number of free transpositions, and $m = |\sigma|$.

Proof. We'll use a potential function argument. The potential of a state is the number of inversions in MF's list according to the ordering in A's list. Here, a state is an ordering of items in MF's list along with an ordering of items in A's list (i.e. we treat A's list as the correctly sorted order, and count the inversions in MF's list).

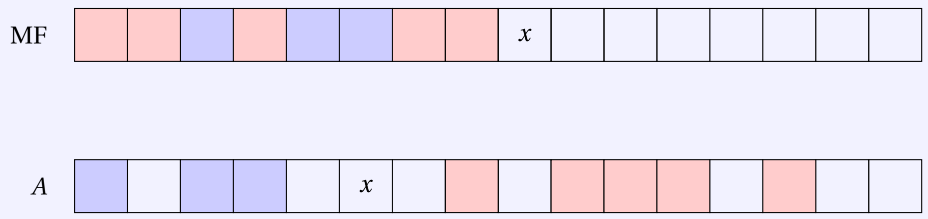
As usual, the Φ-cost of an operation is defined to be the actual cost plus $\Delta\Phi$. This means that

$$\text{total } \Phi\text{-cost} = \text{total actual cost} + \Phi_{\mathit{final}} - \Phi_{\mathit{initial}},$$

or

$$\text{total actual cost} = \text{total } \Phi\text{-cost} + \Phi_{\mathit{init}} - \Phi_{\mathit{final}}.$$

Since MF and A start with the same ordering, then we know that $\Phi_{\mathit{init}} = 0$, and $\Phi_{\mathit{final}} \ge 0$. This means that the total actual cost is at most the total Φ-cost; to bound the total cost, it's enough to bound the total Φ-cost.



Here, suppose x is in position k in MF, and in position i in A, and suppose there are t items before x in MF and after x in A (in red above). This means that the other $k-t-1$ items are before x in both MF and A (in blue above).

The actual cost is k, since it costs k to visit x.

After an access, the t items that were in incorrect relative locations are now in the correct relative locations, and the $k-t-1$ items are now incorrect. This means that the change in potential is

$$\Phi\text{-cost} = \text{actual cost} + \Delta\Phi = (k-t-1) - t = 2(k-t) - 1.$$

We know that $k-t-1 \le i-1$, so the Φ-cost is at most $2i-1$. Summing this up, we have

$$\text{total } \Phi\text{-cost} \le \sum 2i - m = 2\,\text{cost}(A(\sigma)) - m.$$

$\square$

3/7/2023

Lecture 15

Paging, Resource Augmentation

Today, we'll continue with more online algorithms, mainly in regard to paging.

15.1 Deterministic Paging

In the paging problem, we have a cache (with bounded size), and we have disk (with infinite size). Whenever we want to access a particular address, if it is already in cache, then we can access it for free. Otherwise, we need to fetch it with a cost of 1; if the cache has a blank line, we put it there, but if it is full, then we evict an existing line back to the disk.

The user makes requests for a certain sequence of addresses, and we must make these decisions online, of which cache items we evict.

This problem is related to the list update problem from last time; recall that with the list update problem, accessing an element incurs a cost of i. Paging is similar to list update with a modified cost function; that is, the cost of accessing an item at position i is 1 if $i > k$ and 0 otherwise. That is, we can think of our cache as the first k positions in the list, with the rest being disk, and we only pay if we have to touch the disk. (The paging problem also forces you to bring the item closer to the front, and we allow you to choose which item to evict.)

Common algorithms include:

- **Least Recently Used** (LRU): Each page in the cache has a timestamp, storing the most recent time the page was requested, and we evict the page with the oldest timestamp.
- **First In First Out** (FIFO): Each page in the cache has a timestamp storing the time it was put into the cache, and we evict the page that was in the cache the longest.
- **Least Frequently Used** (LFU): Each page in the cache has a counter, keeping track of how many times it has

been accessed, and we evict the page with the smallest counter.

- **Belady's Algorithm** (1966): Each time we have to evict a page, we evict the page that will be requested the furthest in the future. This isn't actually able to be implemented, since we don't know the future. However, it can be shown that this is optimal, and we'll be comparing algorithms this this one.

Slader and Tarjan in 1985 [ST85a] show that LRU and FIFO are k-competitive, and no deterministic algorithm can have competitive ratio $< k$. We'll prove both of these claims today.

It also turns out that least frequently used is not k-competitive; there exists some sequence of operations in which OPT will incur a low constant cost, but LFU will incur an unbounded cost (arbitrarily large in the number of requests).

This doesn't sound too good on first glance; since RAM typically has around 16GB of memory, LRU and FIFO would then be 16 billion competitive. One saving grace is that if the cache is not fully associative, the k would actually be the associativity of the cache, and another is *resource augmentation*.

In particular, it can be shown that

$$\text{cost}(\text{LRU}_k) \le \frac{k}{k-h+1} \cdot \text{OPT}_h.$$

Here, LRU_k means we use LRU on a sequence of operations with a cache of size k, and OPT_h means we use OPT on the same sequence of operations with a cache of size h. This means that we aren't comparing the two algorithms directly; we give OPT a smaller cache size (here $h < k$).

Even with a constant factor smaller memory for OPT, say $h = \frac{k}{2}$, then LRU now becomes 2-competitive.

Theorem 15.1

There is no deterministic algorithm with competitive ratio less than k.

Proof. Consider an algorithm A. Suppose there are $k+1$ different memory addresses in the machine, and suppose OPT and A have the same initial memory state.

At any given point in time, A has k things in its cache, so we'll request the one page that is not in the cache, and we keep doing this in a loop. Here, if there are m requests, there will be m page faults.

Note that although we described the access sequence adaptively, we can imagine that we've simulated A before and just gave the full sequence upfront; since A behaves deterministically, it will always evict the same items for each request.

For OPT, we always evict the page that is requested farthest in the future. Whenever OPT has a page fault, it means that we've requested the one item that is not in the cache. OPT then evicts the item requested further in the future, so all $k-1$ other items still in the cache will be requested again before we request the item we just evicted. This means that once OPT faults, it will never faults again until another k requests; in total it faults at most $\lceil \frac{m}{k} \rceil$ times. □

Theorem 15.2

LRU is k-competitive.

Proof. To show this, we'll show that another algorithm "1-bit LRU" is k-competitive. It turns out that LRU is a special case of 1-bit LRU, so the k-competitiveness of LRU follows from the k-competitiveness of 1-bit LRU.

In 1-bit LRU, we first unmark all pages. When a page x is requested, we do the following:

- If x is not in the cache:
  - If all pages are marked, then unmark all pages

    - Evict an arbitrary unmarked page
    - Insert x into the cache
  - Mark x (after this, x will always be in the cache and it will be marked)

Here, notice that we have a choice of which page to evict if multiple are unmarked. In particular, all of the unmarked pages will be less recently accessed than all of the marked pages. In LRU, we choose the least recently accessed page, which will definitely be unmarked.

(As a side note, we can simply choose a *random* unmarked page to get a randomized algorithm that is $\log(k)$-competitive; we'll show this too later.)

To show that 1-bit LRU is k-competitive, we divide time into phases. We start a new phase every time we unmark all pages. During one phase, 1-bit LRU will always have at most k page faults, since we mark a page for every fault (we also mark on hits, so it could be less).

Whenever we transition to a new phase, we must have requested $k+1$ pages (all pages in the cache had been requested, and now we're requesting another), so OPT must have had at least one fault.

This gives us the k-competitiveness of 1-bit LRU and thus LRU. □

15.2 Randomized Paging

There are three types of adversaries to consider:

- **Omniscient adversary**: This adversary can see the future, including the outputs to all calls to `rand()` (i.e. an omniscient adversary knows the result of all randomness).
- **Adaptive adversary**: This adversary cannot see the future, but knows the random decisions made in the past, and can see the contents of your cache.
- **Oblivious adversary**: This adversary knows the code, but does not know any future outputs of randomness, and does not necessarily know what is in the cache at this point (if the algorithm is randomized).

Randomized paging algorithms also can't beat k-competitiveness with an omniscient or adaptive adversary, by the same proof as before—we only needed to know the contents of the cache. In contrast, we'll see that there is an algorithm that is $2H_k$-competitive with an oblivious adversary, where H_k is the kth harmonic number; $H_k = \sum_{i=1}^{k} \frac{1}{k} \approx \ln k$.

It is known that H_k-competitiveness is possible (due to McGeoch and Slader in 1991 [MS91]), and that we cannot do any better than H_k.

We'll show today that $2H_k$-competitiveness is possible, and we'll also show that we cannot do any better than H_k, but we won't go through the actual H_k-competitive algorithm.

Before we do, let us formally define what we mean by competitive ratios for randomized algorithms.

Definition 15.3: Competitive Ratio of Randomized Algorithms

The competitive ratio of a randomized algorithm A is C if for all sequences σ,

$$\mathbb{E}[\text{cost}(A(\sigma))] \leq C \cdot \text{OPT}(\sigma) + O(1).$$

In particular, we're looking at expected cost compared to OPT.

Theorem 15.4

Randomized marking (MARK) is $2H_k$-competitive.

In particular, the randomized marking algorithm is the 1-bit LRU algorithm but we simply choose a random unmarked page to evict.

Proof. Just like with 1-bit LRU, we divide time into phases. We'll mark some pages as clean and some pages as stale. A page is clean if it was not requested in the last phase or so far in the current phase; a page is stale if it was requested in the last phase but not yet in the current phase.

We'll also define D_i as the number of pages in OPT's cache at the beginning of phase i, which are not in MARK's cache. (Note that since the beginning of phase $i+1$ is the same as the end of phase i, D_{i+1} denotes the same quantity, but at the end of phase i.)

We'll also define L_i as the number of clean requests in phase i.

First, we will show that

$$\text{OPT} \geq \frac{1}{2}\sum_i (L_i + D_{i+1} - D_i) \tag{15.1}$$

and then we'll show that

$$\mathbb{E}[\text{cost}(\text{MARK})] \leq H_k \sum_i L_i \tag{15.2}$$

Notice that the summation in Eq. (15.1) telescopes; we have

$$\text{OPT} \geq \frac{1}{2}\sum_i (L_i + D_{i+1} - D_i) = \frac{1}{2}\sum_i L_i - \underbrace{D_{init}}_{0} + \underbrace{D_{final}}_{\geq 0} \geq \frac{1}{2}\sum_i L_i,$$

so the ratio between MARK and OPT is $2H_k$.

To show Eq. (15.1), we'll show that

$$\text{cost of OPT in phase } i \geq \max(L_i - D_i, D_{i+1}).$$

In particular, since the maximum is at least the average, the cost of OPT in phase i is at least $\frac{1}{2}(L_i - D_i + D_{i+1})$, and we can sum over all i to get the inequality.

Suppose we've just entered phase i. The number of clean requests in phase i in L_i, and MARK will fault on all of these requests (these requests were for pages that were not requested in the last phase, and were not requested in the current phase so far).

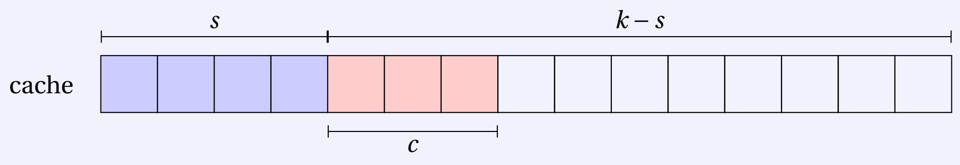
However, OPT may not fault on all of these pages; it could have different items in its cache. Since OPT could have at most D_i items in its cache that are not in MARK's cache, it could potentially have D_i fewer faults, giving a minimum of $L_i - D_i$ faults.

At the same time, we can look at the end of phase i (i.e. D_{i+1}). MARK's cache has all the items that were requested in the current phase; in each phase, there were k distinct pages that were requested, and these pages must also have been serviced by OPT. However, OPT *also* had D_{i+1} pages that are different from MARK, so it must have brought in at least $k + D_{i+1}$ pages into its cache, meaning it had at least D_{i+1} faults (since it has a cache of size k).

Together, OPT must have had at least $\max(L_i - D_i, D_{i+1})$ faults in phase i, which shows the desired inequality.

To show Eq. (15.2), let us look at the expected cost of MARK in phase i. There are L_i clean requests, and all of these are page faults. There are also some stale requests during this phase; let us look at these stale requests in order by time.

Suppose there were c clean requests and s stale requests so far in this phase.



At this point in time, suppose we're servicing the next stale request. We will incur a cost of 1 for this request only if we chose one of the stale pages evicted when we brought in a clean page earlier in this phase. (Otherwise, the stale request must be for one of the existing pages in the cache, and we'd have a cache hit).

Out of the $k-s$ stale pages left to be accessed, c of them got clobbered by a clean page request, so we have a probability $\frac{c}{k-s}$ that this new stale request causes a page fault.

Looking at all the possible values of s to sum over, the first stale request has $s=0$, and the last stale request has $s=k-L_i-1$, since we have at most $k-L_i$ stale requests. (Additionally, c depends on the current time and varies throughout the summation, so we'll index by s.)

This gives an expected cost of

$$\begin{aligned}\mathbb{E}\big[\text{cost}(\textsc{Mark})\text{ at phase } i\big] &\le L_i + \sum_{s=0}^{k-L_i-1}\frac{c_s}{k-s}\\ &\le L_i + L_i\sum_{s=0}^{k-L_i-1}\frac{1}{k-s}\\ &= L_i + L_i\sum_{j=L_i+1}^{k}\frac{1}{j}\\ &= L_i + L_i(H_k - \underbrace{H_{L_i}}_{\ge 1})\\ &\le L_i + L_iH_k - L_i = L_iH_k\end{aligned}$$

Summing over all phases now, we've shown (15.2). □

Theorem 15.5

It is impossible to beat H_k-competitiveness.

Proof. Suppose that at every point in time, suppose we request a uniformly random page amongst the pages $\{1,\dots,k+1\}$.

For any algorithm A, we have $\mathbb{E}[\text{cost}(A(\sigma))] = \frac{m}{k+1}$, since each request has $\frac{1}{k+1}$ probability of not being in the cache.

For OPT, whenever we have a page fault, we evict the page that is accessed furthest in the future; the expected number of accesses before it's accessed again is just a coupon collector problem. In particular, this expected value is the expected number of tries before we've accessed every single other page at least once. Here, it turns out that it takes an expected kH_k tries, so OPT has a page fault once every kH_k accesses. With m accesses, it has an expected $\frac{m}{kH_k}$ page faults.

This means that the competitive ratio must be at least H_k. □

3/9/2023

Lecture 16

k-Server Problem, LP Duality, Online Primal/Dual

16.1 k-server problem

The k-server problem generalizes many online problems. Here, we have an n-point metric space with k servers (for $k < n$). We receive points online, and we must move some server to that point if one is not already there, with cost equal to the distance moved.

Paging is a special case k-server; our metric space consists of the pages on the machine, k is the size of the cache, and the distance function is 1 for all pairs of points.

Another special case is weighted paging. With normal paging, we pay a cost of 1 if the page is not in the cache, and pay 0 if it is in the cache. With weighted paging, requesting a page is not always of uniform cost; each page p has an associated cost $c(p)$ if it is not in the cache.

Here, the distance function is $d(x, y) = \frac{1}{2}\big(c(x) + c(y)\big)$. Here, when we move a server to a point x (to retrieve the page from disk), we pay $\frac{1}{2}c(x)$, and when we evict the page, we end up paying the other $\frac{1}{2}c(x)$. There are edge cases where at the end of the sequence of accesses, we've only paid half the cost for some pages, but this can be resolved with a proper reduction, which we will not go into here.

There is an $O(\log k)$-competitive algorithm for weighted paging, originally showed by Bansal, Buchbinder, and Naor in FOCS 2007 [BBN07].

There is also a deterministic $(2k-1)$-competitive algorithm for k-server, called the "work function" algorithm, shown by Koutsoupias and Papadimitriou [KP95], and it was conjectured that a $O(\log k)$-competitive randomized algorithm is possible. However, Bubeck, Coester, Rabani in 2022 [BCR22] (currently under review) showed that no algorithm can do better than $\Omega(\log^2 k)$. Earlier, Bubeck et al. in STOC 2018 [BCL+18] showed that there is an $O(\log^2 k \log n)$-competitive algorithm, though it is conjectured that one can remove the $\log n$ dependency.

16.2 LPs and LP Duality

Before we go into online primal-dual, we'll first recap linear programming. Given as input a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, a vector $\vec{b} \in \mathbb{R}^m$, and a vector $\vec{c} \in \mathbb{R}^n$, the goal is to find a vector $\vec{x} \in \mathbb{R}^n$ to solve the *primal LP*:

$$\begin{aligned} \min_{\vec{x}} \quad & \vec{c}^T \vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} \geq \vec{b}, \\ & \vec{x} \geq \vec{0} \end{aligned}$$

(Here, the nonnegativity constraint on $\vec{x}$ will be useful in writing the dual, but it can be omitted and gives the same LP.)

We'd like to have a lower bound on the values of $\vec{c}^T\vec{x}$ we can possibly have. To find one, suppose we take the inequality $\mathbf{A}\vec{x} \geq \vec{b}$; this is essentially specifying m different inequalities on the elements of the two resulting vectors. If we multiply each inequality by y_i on both sides, where $y_i \geq 0$, the inequalities would still hold. If we add all these resulting inequalities up, the statement would still be true.

Doing this transforms the inequality into $\vec{y}^T\mathbf{A}\vec{x} \geq \vec{b}^T\vec{y}$, or equivalently $(\mathbf{A}^T\vec{y})^T\vec{x} \geq \vec{b}^T\vec{y}$, for some $\vec{y} \geq 0$.

Notice that if we can show that $\mathbf{A}^T\vec{y} = \vec{c}$, then we now have a lower bound on $\vec{c}^T\vec{x}$—namely, $\vec{b}^T\vec{y}$. It turns out that if we have $\mathbf{A}^T\vec{y} \leq \vec{c}$, since $\vec{x}$ is nonnegative, then $\vec{b}^T\vec{y}$ is still a lower bound on $\vec{c}^T\vec{x}$. This is what motivates the *dual LP*:

$$\begin{aligned} \max_{\vec{y}} \quad & \vec{b}^T \vec{y} \\ \text{s.t.} \quad & \mathbf{A}^T\vec{y} \leq \vec{c}, \\ & \vec{y} \geq \vec{0} \end{aligned}$$

Here, the dual is essentially wanting to get the best lower bound possible; if we can find a $\vec{y}$ such that $\mathbf{A}^T\vec{y} \le \vec{c}$, then $\vec{b}^T\vec{y}$ is a lower bound for $\vec{c}^T\vec{x}$.

Theorem 16.1: Weak Duality

For all $\vec{x}$ and $\vec{y}$ feasible for the primal and dual respectively, we have

$$\text{cost}_{primal}(\vec{x}) \ge \text{cost}_{dual}(\vec{y}).$$

In particular,

$$\text{OPT(primal)} \ge \text{OPT(dual)}.$$

Weak duality can be shown by construction of the dual; there is a stronger duality, called *strong duality*, which we won't prove today (but we will later).

Theorem 16.2: Strong Duality

If the primal is bounded and feasible, then the dual is also bounded and feasible, and moreover,

$$\text{OPT(primal)} = \text{OPT(dual)}.$$

16.3 Online Primal-Dual

Online primal/dual algorithms were introduced by Buchbinder and Naor in 2009 [BN09].

Here, we have a vector $\vec{x} \in \mathbb{R}^n$ starting off at $\vec{0}$, and we see linear constraints (A_i, b_i) come online (from the primal). Each constraint must be satisfied at each timestep, and $\vec{x}$ is only allowed to increase monotonically.

On the problem set, you'll have to prove the following theorem:

Theorem 16.3: Approximate Complementary Slackness

Suppose $\vec{x}$ and $\vec{y}$ are feasible solutions to the primal and dual, respectively. Also suppose the following:

$$\begin{aligned} x_i > 0 &\implies \frac{c_i}{\alpha} \le (\mathbf{A}^T\vec{y})_i \le c_i \\ y_i > 0 &\implies \beta b_i \ge (\mathbf{A}\vec{x})_i \ge b_i \end{aligned}$$

(The right-most inequalities are automatic from the primal/dual constraints.)

Then, $\vec{c}^T\vec{x} \le \alpha\beta\vec{b}^T\vec{y}$.

The importance of this theorem is that it allows us to approximate the solution to the original LP.

By strong duality, we know that for any feasible $\vec{x}$ and $\vec{y}$,

$$\vec{b}^T\vec{y} \le \text{OPT} \le \vec{c}^T\vec{x} \le \alpha\beta\vec{b}^T\vec{y} \le \alpha\beta \cdot \text{OPT}.$$

In particular, we have that $\vec{c}^T\vec{x} \le \alpha\beta \cdot \text{OPT}$; that is, if we maintain primal/dual solutions that satisfy approximate complementary slackness, then the $\vec{x}$ we've found gives a cost not too far off from OPT.

16.3.1 Ski Rental (with primal/dual)

We'll use the ski rental problem again as practice with online primal/dual algorithms. In the ski rental problem, we have the following optimization problem on day n:

$$\begin{aligned}\min_{x,\vec{z}} \quad & B\cdot x + \sum_{i=1}^{n} z_i \\ \text{s.t.} \quad & (\forall i)\, x + z_i \geq 1, \\ & x, \vec{z} \geq 0\end{aligned}$$

Here, B is the cost of buying skis, x is a 0-1 variable for whether we've bought the skis, and z_i denotes whether we rented on day i. The constraint $x + z_i \geq 1$ ensures that we either buy the skis or rented skis on day i.

If we wrote this in standard form, we'd have the matrices and vectors

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 \\ 1 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 1 & 0 & \cdots & 0 & 1 \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} x \\ z_1 \\ \vdots \\ z_n \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \qquad \vec{c} = \begin{bmatrix} B \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

The dual is then

$$\begin{aligned}\max_{\vec{y}} \quad & \sum_{i=1}^{n} y_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} y_i \leq B, \\ & (\forall i)\, y_i \leq 1, \\ & \vec{y} \geq \vec{0}\end{aligned}$$

For the algorithm, we'll maintain both primal and dual feasible solutions x, z_i's, and y_i's (initially all zero). When we see a new constraint, it's like we see a new variable y_i (as each y_i corresponds to a constraint). We'll gradually increase y_i until one of the constraints y_i is involved in is tight.

Here, there are two constraints that y_i can be involved in; either y_i hits 1, or the sum over all y_i hits B. Recall that the constraints in the dual correspond to variables in the primal; when we hit a tight constraint, then we set the corresponding primal variable to 1. In this case, if the $\sum_{i=1}^{n} y_i \leq B$ constraint is tight, then we set $x = 1$, and if $y_i \leq 1$ becomes tight, then we set $z_i = 1$.

Now, suppose we look at the approximate complementary slackness.

For the primal, recall that the variables are x and z_i's; let's look at each separately.

- If $x > 0$, it must be 1; we set it to be 1 only if the corresponding constraint in the dual is tight. That is, we must have had $\sum_{i=1}^{n} y_i = B$. This means that $x > 0 \implies \sum_{i=1}^{n} y_i = B$, or $\alpha = 1$.
- If $z_i > 0$, it must similarly be 1; we set it to be 1 only if the corresponding constraint in the dual is tight. That is, we must have had $y_i = 1$. This means that $z_i > 0 \implies y_i = 1$, or $\alpha = 1$.

In either case, we have a lower bound given by $\alpha = 1$.

For the dual, each variable y_i corresponds to a constraint in the primal, namely $x + z_i \geq 1$. A trivial upper bound for $x + z_i$ is just 2, since both could be 1. This means that we have $\beta = 2$.

Putting this together, approximate complementary slackness allows us to conclude that any solutions $\vec{x}$ and $\vec{y}$ throughout this algorithm must have a cost $\leq \alpha\beta \cdot \text{OPT} = 2 \cdot \text{OPT}$.

Here, technically OPT is the optimal solution to the primal LP we described, which is not the same as the optimal solution to the ski rental problem—there are nonlinear integrality constraints that we have not included. However, adding constraints to a minimization problem can only make the cost go up, so the optimal solution to the real ski rental problem is always at least the optimal solution to the primal LP we have, and the 2-competitiveness still holds.

This algorithm turns out to be exactly the same as the one we arrived at a couple lectures ago, just through a primal/dual lens.

16.3.2 Ski Rental (randomized)

We'll now show that actually with randomization, a competitive ratio of approximately $\frac{e}{e-1}$ is possible.

Here, we'll modify our algorithm slightly to maintain fractional solutions to the problem, and we'll fix the fractional solutions afterward with randomization.

In particular, when we see a new day, if x is already at least 1, we do nothing; otherwise, we set

$$\begin{aligned} y_i &\leftarrow 1 \\ z_i &\leftarrow 1 - x \\ x &\leftarrow \left(1 + \frac{1}{B}\right)x + \frac{1}{cB} \end{aligned}$$

where c is to be determined.

We want to first check that the primal and dual solutions are both feasible, and we also want to look at the ratio between the primal and dual cost. In particular, we already know that $\vec{\boldsymbol{b}}^T\vec{\boldsymbol{y}} \le \text{OPT} \le \vec{\boldsymbol{c}}^T\vec{\boldsymbol{x}}$, but if we can also show that $\vec{\boldsymbol{c}}^T\vec{\boldsymbol{x}} \le \gamma \cdot \vec{\boldsymbol{b}}^T\vec{\boldsymbol{y}}$, then since $\vec{\boldsymbol{b}}^T\vec{\boldsymbol{y}} \le \text{OPT}$, we also have $\vec{\boldsymbol{c}}^T\vec{\boldsymbol{x}} \le \gamma \cdot \text{OPT}$.

Put another way, we want to upper bound $\frac{\vec{\boldsymbol{c}}^T\vec{\boldsymbol{x}}}{\vec{\boldsymbol{b}}^T\vec{\boldsymbol{y}}} \le \gamma$, which will show that we have a γ-approximately optimal solution.

First, we'll check that the solutions are feasible. Looking at the primal, we're always increasing the value of x, and x starts out at 0, so it's always nonnegative. Similarly, we always set b to be $1 - x$, and x never exceeds 1, so b is also nonnegative. The $x + z_i \ge 1$ constraint is also satisfied, since we first set $z_i = 1 - x$, and then make x even bigger, so the sum $x + z_i$ will always be at least 1. This satisfies all of the primal constraints, so the primal solution is indeed feasible.

Looking at the dual, we always set $y_i = 1$ if $x < 1$ or $y_i = 0$ if $x \ge 1$, so it's always at most 1. However, we need to ensure that $\sum_{i=1}^n y_i \le B$. That is, we need to ensure that $x \ge 1$ after B days in order to not have the sum exceed 1.

For the ratio between solutions, let's look at the ratio of the changes $\frac{\Delta\vec{\boldsymbol{c}}^T\vec{\boldsymbol{x}}}{\Delta\vec{\boldsymbol{b}}^T\vec{\boldsymbol{y}}} = \frac{\Delta\text{primal}}{\Delta\text{dual}}$ for each day. Notice that the final costs in the primal and dual is the sum of these changes respectively, so if we can show that the ratio of the changes for each day is bounded by γ, then the ratio of the sum of changes will also be bounded by γ.

If x is already 1, then the change is zero, so we're done. Otherwise, if x isn't already 1, then the change in the dual cost is 1, since we set another y_i to be 1. The change in the primal cost is

$$\begin{aligned} B \cdot \Delta x + z_i &= B \cdot \Delta x + (1 - x) \\ &= B\Bigg(\underbrace{\left(1 + \frac{1}{B}\right)x + \frac{1}{cB}}_{\text{final}} - \underbrace{x}_{\text{initial}}\Bigg) + (1 - x) \\ &= Bx + x + \frac{1}{c} - Bx + 1 - x \\ &= 1 + \frac{1}{c} \end{aligned}$$

This means that the ratio

$$\frac{\Delta\text{primal}}{\Delta\text{dual}} \le 1 + \frac{1}{c}.$$

The only thing left is to show that $x = 1$ after at most B days, which will allow us to determine c.

Suppose we set $r = \frac{1}{cB}$ and $q = 1 + \frac{1}{B}$; let's look at how x evolves. At the beginning, $x = 0$, and each day we set $x \leftarrow qx + r$. In particular, we go

$$x = 0 \to r \to qr + r \to q^2 r + qr + r \to \cdots \to r \sum_{i=0}^{k-1} q^i$$

after k days. We want to ensure that when $k = B$, this value is 1. This is a geometric series, and we have

$$\begin{aligned} 1 = r \sum_{i=0}^{B-1} q^i &= r \cdot \frac{q^B - 1}{q - 1} \\ &= \frac{1}{cB} \cdot \frac{\left(1 + \frac{1}{B}\right)^B - 1}{\frac{1}{B}} \\ &= \frac{1}{c}\left(\left(1 + \frac{1}{B}\right)^B - 1\right) \\ c &= \underbrace{\left(1 + \frac{1}{B}\right)^B}_{\approx e} - 1 \end{aligned}$$

Setting c to this value gives a ratio of approximately $\frac{e}{e-1}$.

The only missing part is to deal with the fractional solutions. If we look at the evolution of x over time, we add some x_i at every day, working our way from 0 to 1. Suppose that we pick some $u \sim \text{Uniform}[0, 1]$, and we buy the skis on the day that x exceeds u. (That is, suppose our integral output is $\hat{x}$ and $\hat{z}_i$; if x exceeds u, then we set $\hat{x} = 1$ and leave $\hat{z}_i = 0$, and otherwise we keep $\hat{x} = 0$ and set $\hat{z}_i = 1$.)

The expected cost is then

$$\mathbb{E}[\text{cost}] = B \cdot \mathbb{P}(\hat{x} = 1) + \sum_{i=1}^{n} \mathbb{P}(\hat{z}_i = 1),$$

since $\hat{x}$ and $\hat{z}_i$ are essentially indicator RVs. The probability that $\hat{x} = 1$ is exactly x, and the probability that $z_i = 1$ is exactly $z_i = 1 - x$ (since in this case $x = 0$ and we did not exceed u).

This means that the expected cost is actually the cost of the primal, and we've shown that we have an $\frac{e}{e-1}$-approximate solution to the primal LP, so this randomized algorithm gives an $\frac{e}{e-1}$-approximate solution to the ski rental problem.

3/14/2023

Lecture 17

Online Primal/Dual (Set Cover), Approximation Algorithms

17.1 Online Set Cover

In weighted set cover, we have sets $S_1, S_2, \dots, S_m \subseteq [n]$, and each set S has a weight c_S. The goal is to find a subcollection A of S_i such that $\bigcup_{i \in A} S_i = [n]$, and that $\sum_{i \in A} c_{S_i}$ is minimized.

In online set cover, we start with the names of the sets, without knowledge of what elements each set contains. We then see the elements of the universe one by one, along with which sets it is part of. At this point, we must make an irrevocable decision on whether to include a new set, ensuring that every element so far is covered.

Similar to the ski rental problem, set cover is an integral problem, so we'll look at fractional relaxations. The primal is

$$\begin{aligned} \min_{\vec{x}} \quad & \sum_S c_S \cdot x_S \\ \text{s.t.} \quad & \sum_{S \ni i} x_S \geq S_i \quad \forall i \in [n], \\ & \vec{x} \geq 0 \end{aligned}$$

The dual turns out to be

$$\begin{aligned} \min_{\vec{y}} \quad & \sum_{i=1}^{n} y_i \\ \text{s.t.} \quad & \sum_{i \in S} y_i \le c_S \forall S, \\ & \vec{y} \ge 0 \end{aligned}$$

Today, we'll see a randomized online algorithm for (non-weighted) set cover with expected competitive ratio $O(\log n \log m)$. This is due to Alon, Awerbuch, Azar, Buchbinder, and Naor in 2009 [AAA+09]. They also showed that a competitive ratio better than $\tilde{O}(\log n \log m)$ is impossible.

We'll look at two analyses of the same algorithm; the first will only look at the primal, and the second will be a primal/dual analysis.

The following algorithm is for the unweighted case of set cover (i.e. $c_S = 1$ for all S); it's an online fractional algorithm with competitive ratio $O(\log m)$, but making it integral will introduce a $O(\log n)$ penalty.

Initially, we set $x = (\frac{1}{m}, \frac{1}{m}, \dots, \frac{1}{m})$. Upon seeing $i \in S_{j_1}, \dots, S_{j_k}$, we will evolve all of these S's simultaneously in continuous time according to the differential equation

$$\frac{\mathrm{d}}{\mathrm{d}t} x_S(t) = x_S(t),$$

until the ith constraint is satisfied. This evolution essentially multiplies each x_S by e^t, for the smallest value of t that satisfies the ith constraint (i.e. the t such that $e^t \sum_{S \ni i} x_S \ge 1$, or $t = -\ln\left(\sum_{S \ni i} x_S\right)$, but we won't think of it in this way for the analysis).

17.1.1 Primal Analysis

We'll do the analysis in two parts. First, we'll show that the total time we are in "evolution mode" is $\le \mathrm{OPT} \cdot \ln(m)$. Next, we'll show that $\frac{\mathrm{d}}{\mathrm{d}t}\text{objective(LP)} \le 1$ when we evolve.

Initially, the objective is 1. We'll show that the rate in which the objective increases is at most 1 at all time. In particular, if we evolve for a total of t time, the objective increases by at most t. This means that we increase the objective by at most $\mathrm{OPT} \cdot \ln m$, so our final objective value is at most $\mathrm{OPT} \cdot \ln m + 1$.

We'll prove the second point first. Suppose we're evolving because we have an item i in sets $S_{j_1}, \dots, S_{j_k}$ that is not yet covered. We then have

$$\frac{\mathrm{d}}{\mathrm{d}t}\text{objective(primal)} = \frac{\mathrm{d}}{\mathrm{d}t} \sum_{r=1}^{k} x_{S_{j_r}}(t) = \sum_{r=1}^{k} x_{S_{j_r}}(t)$$

since we know $\frac{\mathrm{d}}{\mathrm{d}t} x_S(t) = x_S(t)$. Further, since we are evolving, we know that this sum is less than 1 (it is exactly the ith objective), as desired.

Looking at the first point, we want to show that the total time we are in "evolution mode" is at most $\mathrm{OPT} \cdot \ln(m)$.

When we evolve, we're evolving all of the sets that contain i. One of these sets must be taken by OPT. If we look at any arbitrary set S, since x_S starts out as $\frac{1}{m}$, it can only participate in evolutions for $\ln m$ time. This is because every time we evolve, we multiply x_S by e^t; when $t = \ln m$, we multiply x_S by a total of m, getting us a value of ≥ 1, so any item involved in S is already covered.

This means that the total evolution time is at most $\mathrm{OPT} \cdot \ln m$, since we are always evolving some element of OPT for at most $\ln m$ time.

17.1.2 Primal/Dual Analysis

Here, we'll evolve a dual solution simultaneously with the primal solution. In particular, when we are trying to cover i, we will evolve y_i (corresponding to the primal constraint we're trying to satisfy) according to the differential equation $\frac{\mathrm{d}}{\mathrm{d}t} y_i(t) = 1$. That is, after t timesteps, we set $y_i \leftarrow y_i + t$.

In particular, this means that $\frac{\mathrm{d}}{\mathrm{d}t}$objective(dual) = 1, since we're only changing one variable in the sum.

Something here seems a little bit off; we know by weak duality that for all feasible $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$, $\text{cost}_d(\vec{\boldsymbol{y}}) \le \text{OPT} \le \text{cost}_p(\vec{\boldsymbol{x}})$. However, here, at the very end we know that $\text{cost}_p(\vec{\boldsymbol{x}}) < \text{cost}_d(\vec{\boldsymbol{y}})$, since the primal objective increases at a rate strictly less than 1. This would actually show that we've found an algorithm strictly better than OPT:

$$\text{cost}_d(\vec{\boldsymbol{y}}) \le \text{OPT} \le \text{cost}_p(\vec{\boldsymbol{x}}) < \text{cost}_d(\vec{\boldsymbol{y}}) \le \text{OPT}.$$

The issue is that this $\vec{\boldsymbol{y}}$ is not actually feasible for the dual. The claim is that $\frac{y}{\ln m}$ is dual-feasible; if this is the case, then we have

$$\frac{1}{\ln m}\text{cost}_d(\vec{\boldsymbol{y}}) \le \text{OPT} \le \text{cost}_p(\vec{\boldsymbol{x}}) < \text{cost}_d(\vec{\boldsymbol{y}}) \cdot \ln m \le \text{OPT} \cdot \ln m.$$

To see why $\frac{1}{\ln m}\vec{\boldsymbol{y}}$ is dual-feasible, we will show that for all S, $\sum_{i \in S} \le \ln m$.

We know that we increase $\sum_{i \in S} y_i$ when we see such an uncovered i online. The total time spent evolving y_i's for all $i \in S$ is at most $\ln m$, since x_S never evolves for more than $\ln m$ time.

17.1.3 Integral Solutions

To make this solution integral, at the very beginning of the algorithm, for each S, we pick $\alpha_1, \dots, \alpha_r \in [0,1]$ independently uniformly at random (in total, we have mr random numbers). We take the set S at the time in which x_S grows larger than the smallest α_i.

Here, r is a parameter we choose, which will turn out to be approximately $\ln n$.

Suppose we had $r = 1$; this means that the expected cost is the expected sum of the indicators for whether we picked each set S. By linearity, this is equal to the sum of the probabilities that $x_S \le \alpha$, which is just x_S. In particular, this means that the cost is exactly the cost of the fractional solution.

However, the issue with this is that the final solution may not be feasible; we may not take some sets required to cover all elements. If we set r to be large enough, this probability becomes very small (it won't cover all elements with probability 1 though).

Here, we first claim that

$$\mathbb{E}\left[\text{cost(integral solution)}\right] \le r \cdot \text{cost}_p(x) \le r \cdot \text{OPT} \cdot \ln m.$$

The expected cost of the integral solution is equal to

$$\mathbb{E}\left[\text{cost(integral solution)}\right] = \mathbb{E}\left[\sum_S \mathbf{1}\{\text{took set } S\}\right] = \sum_S \mathbb{P}(\text{took set } S) \le \sum_S \sum_{i=1}^{r} \mathbb{P}(x_S \ge \alpha_i) = \sum_S r \cdot x_S = r \cdot \text{cost}_p(x),$$

by the union bound.

Next, we'll show that with high probability, the integral solution is feasible. For each $i \in [n]$, we have

$$\mathbb{P}(i \text{ is not covered}) = \prod_{S \ni i} (1 - x_S)^r,$$

since every x_S involved in i must not have exceeded any α_i's. We know that $(1 - x_S)^r \le e^{-rx_S}$, so we have

$$\mathbb{P}(i \text{ is not covered}) \le \prod_{S \ni i} e^{-rx_S} \le e^{-r\sum_{S \ni i} x_S} \le e^{-r},$$

since $\sum_{S \ni i} x_S \le 1$.

This means that if we chose $r = \ln n + 100$, this probability is bounded by $e^{-100}\frac{1}{n}$, which by union bound the probability that any element is uncovered is e^{-100}, which is very small.

17.2 Approximation Algorithms

Now, let us consider weighted set cover. This algorithm is due to Chvatal in 1979 [Chv79].

The algorithm proceeds as follows. While there exists some uncovered element, take the set S which minimizes c_S divided by the number of newly covered elements.

This is a greedy algorithm that is the direct generalization of the unweighted greedy algorithm.

When we take set S into the solution, we set $x_S \leftarrow 1$; for each i that is newly covered, we set

$$y_i \leftarrow \frac{c_S}{\text{number of newly covered elements}}.$$

Here, the cost of the primal and the cost of the dual increase at the same rate, so $\text{cost}_p(x) = \text{cost}_d(y)$. However, the issue is that $\vec{y}$ may not be dual-feasible.

The claim is that y/H_n is dual-feasible. (As a side-note, this is also known as "dual-fitting", where we take an infeasible solution and scale it by a constant to make it feasible.)

We need to ensure that for all S, $\sum_{i\in S} y_i \le c_S \cdot H_n$. Suppose we enumerate the items in S in the order of when they were covered according to the greedy algorithm; here, suppose the order is $e_1, e_2, \dots, e_k \in [n]$.

Suppose we're about to cover e_i. We already took $e_1, \dots, e_{i-1}$ and are covered by sets we already took. The fact that e_i is not covered means that we have not taken S yet, but could have. If we *had* taken S, we would have given i a price of $\frac{c_S}{k-i+1}$. The greedy price for e_i must have been at most this value, since we minimize this fraction. This implies that the sum

$$\sum_{i\in S} y_i \le c_S \cdot \left(\frac{1}{k} + \frac{1}{k-1} + \cdots + 1\right) = c_S \cdot H_k.$$

Since $k \le n$, this is at most $c_S \cdot H_n$.

The same argument as before gives a chain of inequalities that shows that this algorithm has cost at most $\text{OPT} \cdot H_n$.

3/16/2023

Lecture 18

Approximation Algorithms, Improving Approximation Ratios, PTAS, FPTAS

18.1 Unweighted Vertex Cover

Given an undirected graph $G = (V, E)$, we want to choose a subset S of the vertices that cover all of the edges E. We want to minimize the size of S. This can be thought of as a special case of set cover, where the sets are the edges incident to a given vertex, and the universe is the set of all edges.

A greedy solution is to start S as empty, and while there exists an edge $e = (u, v)$ that is not covered (i.e. neither u nor v were chosen), we take $S \leftarrow S \cup \{u, v\}$. Here, $|S| \le 2 \cdot \text{OPT}$.

We can also give a primal/dual understanding of why this is a 2-approximation.

The primal LP relaxation is

$$\begin{aligned} \min_{\vec{x}} \quad & \sum_{v\in V} x_v \\ \text{s.t.} \quad & x_u + x_v \ge 1 \quad \forall e = (u, v) \in E, \\ & \vec{x} \ge \vec{0} \end{aligned}$$

Here, (with the integrality constraint) $x_v = 1$ if we take v into the vertex cover, and 0 otherwise.

The dual LP is

$$\begin{aligned} \max_{\vec{\boldsymbol{y}}} \quad & \sum_{e\in E} y_e \\ \text{s.t.} \quad & \sum_{e=(v,\cdot)\in E} y_e \le 1 \quad \forall v \in V, \\ & \vec{\boldsymbol{y}} \ge \vec{\boldsymbol{0}} \end{aligned}$$

As a side note, there is another algorithm and analysis that only involves the primal; we solve the primal LP (there are solvers to do this), and simply round to the nearest integer; if $x_u + x_v \ge 1$, at least one of x_u or x_v must be at least 0.5, so this rounded solution is still feasible. The rounding at most doubles each x_v, so we at most double our cost, giving us the 2-approximation.

For the primal/dual algorithm, suppose we initialize $\vec{\boldsymbol{x}} = \vec{\boldsymbol{0}}$ and $\vec{\boldsymbol{y}} = \vec{\boldsymbol{0}}$. Whenever $e = (u, v)$ causes us to add to S, we set $x_u = x_v = y_e = 1$.

This algorithm implies that cost(primal) = 2 cost(dual), since we always set two primal variables to 1 when we set one dual variable to 1. By weak duality, we have that for all feasible $\vec{\boldsymbol{x}}$ and $\vec{\boldsymbol{y}}$,

$$\text{cost}_d(\vec{\boldsymbol{y}}) \le \text{OPT} \le \text{cost}_p(\vec{\boldsymbol{x}}) = 2\,\text{cost}_d(\vec{\boldsymbol{y}}) \le 2\cdot\text{OPT} \implies \text{cost}_p(\vec{\boldsymbol{x}}) \le 2\cdot\text{OPT}.$$

This solution is primal-feasible by construction; we only stop when all primal constraints are satisfied. This solution is also dual-feasible, since it is impossible to take two or more edges touching a vertex v (whenever we take an edge touching v, we always take v, so we will never take any other edge touching v).

18.2 Barriers to Improving Approximation Ratios

Strong duality says that for optimal solutions $\vec{\boldsymbol{x}}$ and $\vec{\boldsymbol{y}}$ for the primal and dual, $\text{cost}_d(\vec{\boldsymbol{y}}) = \text{cost}_p(\vec{\boldsymbol{x}})$. However, this optimal solution is for the *LP*, not for the original problem, which would usually have integrality constraints.

In general, we have that OPT(LP) ≤ OPT(IP) (IP for integral program), but if this inequality is strict, then no matter how well we solve the LP, we cannot eliminate this strict inequality.

Definition 18.1: Integrality Gap

The *integrality gap* is the worst case over all problem instances of

$$\frac{\text{OPT(IP)}}{\text{OPT(LP)}}.$$

18.2.1 Vertex Cover

Suppose we look at vertex cover again. Suppose we have the complete graph $G = K_n$. The optimal solution is to take $n-1$ vertices (the edges adjacent to the last vertex are all already covered). However, for the LP, the optimal solution has cost $\frac{n}{2}$, where we take $\vec{\boldsymbol{x}} = (\frac{1}{2}, \frac{1}{2}, \dots, \frac{1}{2})$. Here, this means that the integrality gap is approximately 2.

This integrality gap is also the worst case, since the solution we gave prior is already a 2-approximation of the IP (since our solution is integral).

18.2.2 Set Cover

We can also show an integrality gap for set cover. Suppose we have the universe $\mathcal{U} = \mathbb{F}_2^q \setminus \vec{\boldsymbol{0}}$; that is, we have q-dimensional vectors whose entries are in $\mathbb{F}_2$ (i.e. mod 2). The size of the universe is then $2^q - 1$. For each $\vec{\boldsymbol{v}} \in \mathbb{F}_2^q$, we define

$$S_v = \{\vec{\boldsymbol{\alpha}} \in \mathcal{U} \mid \langle \vec{\boldsymbol{\alpha}}, \vec{\boldsymbol{u}} \rangle = 1\}.$$

Here, we have $m = 2^q$ sets (though the empty set where $\vec{\boldsymbol{v}} = \vec{\boldsymbol{0}}$ doesn't actually help).

For any element $\vec{\boldsymbol{\alpha}} \in \mathcal{U}$, the number of v such that $S_v \ni \vec{\boldsymbol{\alpha}}$ is exactly $\frac{m}{2}$. There are many ways to show this; one way is through a probabilistic lens. For a uniformly random $\vec{\boldsymbol{v}} \in \mathbb{F}_2^q$, the probability that $\langle \vec{\boldsymbol{\alpha}}, \vec{\boldsymbol{v}} \rangle = 1$ is exactly $\frac{1}{2}$. This is because we have some vector $\vec{\boldsymbol{\alpha}} = (0, 0, \cdots, 1, \cdots)$, where the first 1 occurs at position j^*. The dot product is of the form $v_{j^*} + \sum(\cdots)$; conditioning on the latter summation, v_{j^*} flips the dot product with probability $\frac{1}{2}$, so the overall probability is $\frac{1}{2}$.

Here, this means that the OPT(LP) ≤ 2, since we can set $\vec{\boldsymbol{x}} = (\frac{2}{m}, \frac{2}{m}, \dots, \frac{2}{m})$. We claim that OPT(IP) $\ge q$.

Suppose for the sake of contradiction that $S_{v_1}, \dots, S_{v_{q-1}}$ is feasible. This means that these sets cover all elements of the universe. In particular, we know that

$$\bigcap_{i=1}^{q-1} \overline{S_{v_i}} = \varnothing.$$

This means that we have $q - 1$ constraints on α, giving a 1-dimensional subspace.

Since $q \approx \log_2 n$, this gives an integrality gap of $\frac{1}{2} \log_2(n)$.

18.3 Polynomial Time Approximation Schemes

Definition 18.2: PTAS

A PTAS is a *polynomial time approximation scheme*; it is a family of algorithms indexed by ε such that:

- A_ε gets $\le (1+\varepsilon)$OPT for any input
- The runtime of A_ε is $\le O(n^{f(\frac{1}{\varepsilon})})$.

Definition 18.3: FPTAS

A FPTAS is a *fully polynomial time approximation scheme*; it's the same as a PTAS, but now the runtime of A_ε is $\le \text{poly}(\frac{n}{\varepsilon})$.

Not all problems have a PTAS or an FPTAS; in particular, vertex cover and set cover do not have FPTAS's, since they are integral problems. If there existed an FPTAS, there we can just set $\varepsilon = \frac{1}{3n}$, and since OPT $\le n$, the algorithm must just solve the problem in polynomial time.

18.3.1 Knapsack

In the knapsack problem, we have an input of n items, each with a value and a weight; we have values $\{v_1, \dots, v_n\}$ and weights $\{w_1, \dots, w_n\}$ for items $1, \dots, n$ respectively. We also have a knapsack that can carry up to W pounds in weight.

The goal is to pack as much total value as possible where the items we take weigh at most W in total.

The LP is

$$\begin{aligned} \max_{\vec{\boldsymbol{x}}} \quad & \sum_{i=1}^n x_i v_i \\ \text{s.t.} \quad & \sum_{i=1}^n x_i w_i \le W, \\ & \vec{\boldsymbol{x}} \le \vec{\mathbf{1}}, \\ & \vec{\boldsymbol{x}} \ge \vec{\mathbf{0}}, \\ & (\vec{\boldsymbol{x}} \in \mathbb{Z}) \end{aligned}$$

There are several exact algorithms to solve knapsack:

- Dynamic programming: $O(nW)$ time.

  We define $f(i,w)$ as the maximum value obtainable with a knapsack of capacity w, only taking items amongst the first i items $\{1,\dots,i\}$. We want $f(n,W)$, and we can define the recurrence

$$f(i,w) = \begin{cases} -\infty & w < 0 \\ 0 & i = 0 \\ \max(f(i-1,w), v_i + f(i-1, w-w_i)) & \text{otherwise} \end{cases}$$

  We only ever take constant time for any pair of inputs, and there are nW possible pairs of inputs, so this takes $O(nW)$ time.

- Dynamic programming (again): $O(nV)$, where $V = \sum_{i=1}^n v_i$

  We define $f(i,v)$ to be the minimum weight required to take items with total value exactly v. We want the maximum value of v such that $f(n,V) \le W$. This function also satisfies a recurrence relation, which we can compute via DP in $O(nV)$ time.

These DP runtimes are pseudopolynomial, so it is not quite polynomial.

With the LP relaxation, there is a simple greedy algorithm that solves the LP. In particular, we sort the items by v_i/w_i, and take the most items that fit, in this order.

This greedy algorithm doesn't do well for integral knapsack; one example is with items $(v_1 = 1+\delta, w_1 = 1)$ and $(v_2 = W, w_2 = W)$. The greedy algorithm will take (v_1, w_1), which will prevent us from taking (v_2, w_2). This algorithm is off by a factor of approximately W.

A better algorithm is to (1) run greedy, and (2) take only the most valuable item, and take the better of these two options. This actually gives at least $\frac{1}{2}$OPT (it's a 2-approximation), which we'll show next time.

3/21/2023

Lecture 19

PTAS, FPTAS, FPRAS

19.1 Knapsack Approximations

Last time, we claimed that the following algorithm was a 2-approximation of knapsack: (1) run the greedy algorithm for knapsack, (2) take only the item with the maximum value. We then take the better of (1) or (2); this is always $\ge \frac{1}{2}$OPT.

We'll first prove a lemma:

Lemma 19.1

Let k be the first item (in the sorted order) that greedy did not take; that is, greedy takes items $1,\dots,k-1$, but not the kth item. In particular, we have

$$\frac{v_1}{w_1} \ge \frac{v_2}{w_2} \ge \frac{v_3}{w_3} \ge \cdots \ge \frac{v_k}{w_k}.$$

Then, $\sum_{i=1}^k v_i > \text{OPT(ILP)}$.

Proof. Recall that the LP relaxation of knapsack is

$$\begin{aligned} \max_{\vec{x}} \quad & \sum_{i=1}^n x_i v_i \\ \text{s.t.} \quad & \sum_{i=1}^n x_i w_i \le W, \end{aligned}$$

$$0 \le x_i \le 1 \quad \forall i$$

We know that OPT(ILP) ≤ OPT(LP), and that greedy is optimal for the fractional version of the problem.

We know that greedy can take the first $k-1$ items, but may be able to take some fractional part of the kth items. This means that OPt(LP) < $\sum_{i=1}^{k} v_i$, giving our desired result. □

Using the lemma, we know that

$$\underbrace{\sum_{i=1}^{k-1} v_i}_{\le \text{greedy}} + \underbrace{v_k}_{\le v_{max}} > \text{OPT}.$$

Since the sum of the greedy solution and v_{max} is strictly greater than OPT, then at least one of these two solutions is $> \frac{1}{2}$OPT, which shows that this is a 2-approximation.

19.1.1 Knapsack PTAS

Recall that in a PTAS, we want $\ge (1-\varepsilon)$OPT in time $n^{f(\frac{1}{\varepsilon})}$.

Observe that the number of items of value $> \varepsilon$OPT that the optimal solution takes is $\le \lfloor \frac{1}{\varepsilon} \rfloor$ (otherwise, these items would add up to more than OPT).

The idea is to guess the set S of such items (of value $> \varepsilon$OPT) that OPT takes (i.e. we try every possible set S). For any particular guess S, we want to pack the remaining items well, i.e. using greedy.

In particular, we should never take any items with value $> \varepsilon$OPT that we did not guess, but we don't know what εOPT is. However, there are at most εOPT different thresholds that it could be, so we can just try them all.

From the lemma we proved prior, greedy on the items $\le \varepsilon$OPT loses $v_k \le \varepsilon$OPT, so our final set of items is $\ge (1-\varepsilon)$OPT.

The final runtime is then (# guesses for S) · (time per S). We spend poly(n) time per S. The number of guesses for S is at most $\binom{n+\frac{1}{\varepsilon}}{\frac{1}{\varepsilon}}$, since we can add $\frac{1}{\varepsilon}$ dummy items, and the number of real items we chose must be most $\frac{1}{\varepsilon}$. Since $\binom{n}{k} \le \left(\frac{en}{k}\right)^k$, this is at most $(e(\varepsilon n+1))^{\frac{1}{\varepsilon}}$.

19.1.2 Knapsack FPTAS

Recall that in an FPTAS, we want a poly$\left(\frac{n}{\varepsilon}\right)$ runtime. Today, we'll see $O\left(\frac{n^3}{\varepsilon}\right)$, but there are some better results known.

There is an algorithm with runtime $\tilde{O}(n + \frac{1}{\varepsilon^4})$ due to Lawler in 1979 [Law79], and the best known now is $\tilde{O}(n + \frac{1}{\varepsilon^{2.2}})$, due to Deng, Jin, and Mao in SODA 2023 [DJM23].

A starting point we'll use is the exact DP algorithm with $O(nV)$ runtime, where $V := \sum_{i=1}^{n} v_i$.

Next, for all i, we define

$$v_i' := \frac{n}{\varepsilon} \cdot \left\lfloor \frac{v_i}{v_{max}} \right\rfloor.$$

Then, we return the optimal set found by the DP algorithm on the values given by v'.

The runtime is $O(nV')$, where $V' := \sum_{i=1}^{n} v_i'$. We know that each

$$v_i' \le \left\lfloor \frac{n}{\varepsilon} \cdot \frac{v_i}{v_{max}} \right\rfloor \le \frac{n}{\varepsilon}.$$

Summing over all i, we have $V' \le \frac{n^2}{\varepsilon}$, giving a final runtime of $O\left(\frac{n^3}{\varepsilon}\right)$.

Analyzing this FPTAS, suppose the optimal set for v_i is A, and the optimal set for v_i' is B. The DP algorithm will find B, and we want to argue that the value of B is a good approximation of the value of A.

For ease, let us define $\alpha \coloneqq \frac{n}{\varepsilon v_{max}}$, so that we essentially scaled each value by α and took the floor when converting to v'_i.

We know that $\alpha v_i - 1 \le v'_i \le \alpha v_i$, since we took the floor of αv_i to get v'_i. Rearranging, we know that $\frac{1}{\alpha} v'_i \le v_i \le \frac{1}{\alpha}(v'_i + 1)$.

We know that $\mathrm{val}'(B) \ge \mathrm{val}'(A)$, where val' is the value you get when using the v'_i values, since B is the optimal set using v'_i. This means that $\frac{1}{\alpha}\mathrm{val}'(B) \le \frac{1}{\alpha}\mathrm{val}'(A)$.

Since $\frac{1}{\alpha} v'_i \le v_i$, we know that $\mathrm{val}(B) \ge \frac{1}{\alpha}\mathrm{val}'(B)$. Further, $\mathrm{val}'(A) \le \alpha\,\mathrm{val}(A) - |A|$, since $\alpha v_i - 1 \le v'_i$. Putting this together, we have

$$\begin{aligned}
\mathrm{val}(B) \ge \frac{1}{\alpha}(\alpha\,\mathrm{val}(A) - |A|) &= \mathrm{val}(A) - \frac{\varepsilon v_{max}}{n}|A| \\
&\ge \mathrm{OPT} - \varepsilon v_{max} && \left(\mathrm{val}(A) = \mathrm{OPT}, \frac{|A|}{n} \le v_{max}\right) \\
&\ge (1-\varepsilon)\mathrm{OPT} && (v_{max} \le \mathrm{OPT})
\end{aligned}$$

19.2 FPRAS

Definition 19.2: FPRAS

An FPRAS is a *fully polynomial (time) randomized approximation scheme*. It's essentially just a randomized FPTAS.

An example FPRAS is counting the number of solutions to an input DNF formula.

Recall that a conjunctive normal form (CNF) is of the form $C_1 \wedge C_2 \wedge \cdots \wedge C_m$ where each clause C_i is the OR of some literals. A disjunctive normal form (DNF) is of the form $C_1 \vee C_2 \vee \cdots \vee C_m$, where each clause C_i is the AND of some literals.

We know that given a CNF formula, it's NP-complete to check whether it is satisfiable (this is the definition of SAT). However, it's very easy to determine whether a DNF is satisfiable (we just take a clause that is non-empty and doesn't contain a literal and its negation, and set the variables as it tells us).

The goal here is that given a DNF formula φ, we want to count the number of x's such that $\varphi(x) = \text{True}$.

Theorem 19.3

DNF counting is NP-hard.

Proof. We'll show that SAT reduces to DNF-counting.

Given a SAT instance, i.e. a CNF formula $\varphi = C_1 \wedge C_2 \wedge \cdots C_m$, we want to determine whether there is a satisfying assignment to φ. If we look at $\overline{\varphi}$, we have

$$\overline{\varphi} = \overline{C_1} \vee \overline{C_2} \vee \cdots \vee \overline{C_m}.$$

Since each clause used to be an OR of literals, $\overline{C_i}$ is now an AND of literals, so $\overline{\varphi}$ is a DNF.

Suppose that we run DNF-counting to count the number of solutions to $\overline{\varphi}$. Suppose this gives T solutions; since there are at most 2^n possible assignments, we know that φ has $2^n - T$ solutions. □

Since DNF-counting is hard, we'll just hope to get an approximation of the number of solutions in polynomial time.

The new goal is to output $\tilde{T}$ such that

$$\mathbb{P}\left(\left|T - \tilde{T}\right| > \varepsilon T\right) < \delta.$$

That is, $\tilde{T} = (1 \pm \varepsilon)T$ with high probability.

A first idea is to define $p := \frac{T}{2^n}$, the fraction of assignments that are actually solutions. We want to estimate p, i.e. we want to know $\tilde{p} = (1 \pm \varepsilon)p$, so we can return $2^n \tilde{p}$. The typical way to approximate this proportion is to take a random sample of assignments, and compute the proportion of assignments that are satisfying. Empirically, we can get an estimate by taking the average of many of these samples.

The issue here is that the variance of this method is bad. For example, if $\varphi = x_1 \wedge x_2 \wedge \cdots \wedge x_n$, then the samples will almost never be satisfying, and we'll get an estimate of 0. In particular, if p is small, it takes on average $\frac{1}{p}$ samples before we even see any satisfying assignment, so we'd need to take exponentially many samples, which we cannot do.

A better idea is to change the game to ensure p is not too small. This is due to Karp, Luby, Madaras in 1989 [KLM89].

Suppose B is the set of satisfying assignments, and B' is the set of all pairs (i, x) such that x satisfies C_i. If we let S_i to be the set of x that satisfy C_i, then $B = \bigcup S_i$ and $B' = \biguplus S_i$, i.e. the disjoint union of the S_i's, taking each set into B' multiple times if it occurs multiple times among the S_i's.

The new goal is to estimate $\frac{|B|}{|B'|}$, which we'll define as p'. We know that $\left|B'\right| \leq m \cdot |B|$, since each solution for B satisfies at most m of the clauses (there are only M clauses), so each solution appears in B at most m times. This means that $p' \geq \frac{1}{m}$, so we can sample B' to get an estimate of p'.

The algorithm is then to pick a random $(i, x) \in B'$, check if $(i, x) \in B$, and repeat this ℓ times. We then return $\frac{1}{\ell} \cdot (\text{\# samples in } B)$. There are a few things missing: how do we check $(i, x) \in B$? How do we sample a random element of B'?

An observation is that B is in bijective correspondence with a subset of B'. In particular, B is a bijection from $\{(i, x) : C_i \text{ is the first clause that } x \text{ satisfies}\}$. In this latter set, each satisfying assignment is counted exactly once. This means that we can check whether $(i, x) \in B$ by simply checking all clauses $1, \ldots, i-1$ to ensure that x does not satisfy any of the previous clauses.

How do we uniformly sample the elements of B'? First, we want to decide which clause to satisfy, and we'll pick a random satisfying assignment for the clause. It's easy to pick a random satisfying assignment for the clause, since there's only one way to assign the variables inside the clause, and we can pick the values of all other variables not in the clause uniformly randomly.

To pick a random clause to give a uniform sample of satisfying assignments, we can notice that $|S_i| = 2^{n-(\text{\# literals in } C_i)}$, so we can pick i with probability $\frac{|S_i|}{\sum_{j=1}^m |S_j|}$.

Now, let us analyze this algorithm. Suppose we define $X_1, \ldots, X_\ell$ as follows:

$$X_j = \begin{cases} 1 & \text{if } j\text{th sample} \in B \\ 0 & \text{otherwise} \end{cases}.$$

Let us also define $X = \sum_{j=1}^{\ell} X_j$, so that $\mathbb{E}[X] = p' \cdot \ell$.

By the Chernoff bound, we know that

$$\mathbb{P}\left(\left|X - \mu\right| > \varepsilon\mu\right) \leq 2e^{-\varepsilon^2\mu/3} = 2e^{-\varepsilon^2 p' \ell/3} \leq 2e^{-\varepsilon^2 \ell/(3m)},$$

since we know that $p' \geq \frac{1}{m}$. If we choose

$$\ell = \left\lceil \frac{3m}{\varepsilon^2} \ln\left(\frac{2}{\delta}\right) \right\rceil,$$

then the probability is at most δ, as desired.

3/23/2023

Lecture 20

SDP, Approximations in Streaming

20.1 SDP Relaxation: Max Cut

In the max-cut problem, we're given an undirected graph $G = (V, E)$, and the goal is to find a cut $(S, V \setminus S)$ with the most number of edges crossing the cut. That is, we want to find a maximal $|E(S, V \setminus S)|$. (The weighted version would look at the sum of weights of edges crossing the cut.)

This problem is NP-hard, and the decision version is NP-complete, so we'll try to give a good approximation algorithm.

One simple approximation algorithm is just to pick a random partition; for each vertex, with probability $\frac{1}{2}$ put it in S and otherwise put it in $V \setminus S$. The expected number of edges that are cut is then

$$\mathbb{E}[\#\text{ edges cut}] = \mathbb{E}\left[\sum_{e \in E} \mathbf{1}\{e \text{ cut}\}\right] = \sum_{e} \mathbb{P}(e \text{ cut}) = \frac{m}{2},$$

since each edge has a probability $\frac{1}{2}$ of being cut. Notice that OPT $\le m$, so this is an $\frac{1}{2}$-approximation.

Another simple algorithm is greedy. Here, we start with an arbitrary partition, say $S = \{1\}$, with $V \setminus S = \{2, \dots, n\}$. While there exists a v such that moving v to the other partition increases the cut size, we do this swap.

Since the number of edges crossing the cut is at most m, and each iteration of the loop increases the cut size by at least 1, this will take polynomial time. It can be shown that this gets $\ge \frac{m}{2}$ edges cut.

Can we do better? For a while, this was the best known approximation algorithms. The algorithm that did better used an *SDP Relaxation.*

20.1.1 Semidefinite Programming (SDP)

Semidefinite programming is a generalization of linear programming. These optimization problems take the form of

$$\begin{aligned} \min_{\mathbf{X}} \quad & \operatorname{tr}(\mathbf{C}^T\mathbf{X}) \\ \text{s.t.} \quad & \operatorname{tr}(\mathbf{A}_i^T\mathbf{X}) = b_i \quad \forall i, \\ & \mathbf{X} \succeq 0 \end{aligned}$$

Here, we say that $\mathbf{X} \succeq 0$ if $\mathbf{X}$ is PSD. Here, $\mathbf{X}$ is a matrix of variables and each $\mathbf{A}_i$ are matrices.

As a recap, we have $\operatorname{tr}(\mathbf{M}) := \sum_i M_{ii}$, and we have $\operatorname{tr}(\mathbf{A}^T\mathbf{B}) = \sum_{i,j} A_{ij}B_{ij}$.

The first objective function and the first constraint are both linear constraints, but the last constraint is different; instead of saying that some vector $\vec{x}$ is nonnegative, we're now saying that $\mathbf{X}$ is PSD.

Further, an LP is just a special case of an SDP; we can just take all of the matrices to be diagonal matrices. This turns all the traces into dot products, and the PSD constraint turns into a nonnegativity constraint.

Recall that a matrix $\mathbf{A}$ is PSD if any of the following hold (they're all equivalent)

- $\forall \vec{y}$, $\vec{y}^T\mathbf{A}\vec{y} \ge 0$
- All eigenvalues of $\mathbf{A}$ are ≥ 0
- There exists a $\mathbf{B}$ such that $\mathbf{A} = \mathbf{B}^T\mathbf{B}$.

20.1.2 Max Cut as an SDP

A first observation is that we can formulate max-cut as a quadratic program:

$$\begin{aligned} \max_{\mathbf{X}} \quad & \sum_{\substack{e=(u,v)\\ e\in E}} \frac{1-x_u x_v}{2} \\ \text{s.t.} \quad & x_v^2 = 1 \quad \forall v \in V \end{aligned}$$

Here, we'll say that $x_v = 1$ if it's on one side of the cut, and $x_v = -1$ if it's on the other side of the cut. Notice that if the two vertices adjacent to an edge are on the same side, then $x_u x_v = 1$, and it does not contribute to the cost. Otherwise, if the two vertices are on opposite sides, then $x_u x_v = -1$, and it contributes 1 to the cut.

This is a quadratic program (it has quadratic terms in the objective and constraints), and it's NP-hard to solve. With ILPs, we relaxed them to LPs, and similarly here, We'll relax this quadratic program to an SDP.

Before we do so, we'll look at *vector programming*, and why SDPs are equivalent to vector programming.

An observation to relax the QP is that a matrix is PSD when $\mathbf{A} = \mathbf{B}^T\mathbf{B}$ for some matrix $\mathbf{B}$, we can see that each entry of $\mathbf{A}$ is just a dot product between two columns of $\mathbf{B}$.

Viewing each variable as a vector now, we have the SDP

$$\begin{aligned} \max_{\mathbf{B}} \quad & \sum_{\substack{e=(u,v)\\ e\in E}} \frac{1-\left\langle \vec{\boldsymbol{b}}_u, \vec{\boldsymbol{b}}_v \right\rangle}{2} \\ \text{s.t.} \quad & \left\| \vec{\boldsymbol{b}}_v \right\|_2^2 = 1 \quad \forall v \in V \end{aligned}$$

In particular, notice that if we look at the PSD matrix $\mathbf{X} = \mathbf{B}^T\mathbf{B}$, the dot product $\left\langle \vec{\boldsymbol{b}}_u, \vec{\boldsymbol{b}}_v \right\rangle$ is just the (u, v)th entry of $\mathbf{X}$, and the ℓ_2 norm $\left\| \vec{\boldsymbol{b}}_v \right\|_2^2$ is just the (v, v)th entry of $\mathbf{X}$. This means that this is an SDP, with constraints on the entries of a PSD matrix.

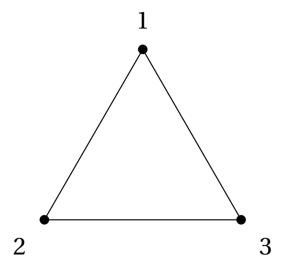
Here, also note that every solution to the QP is still a solution to the SDP, if we view each vector as a 1-dimensional vector.

This SDP relaxation of max-cut is due to Goemans and Williamson in 1995 [GW95].

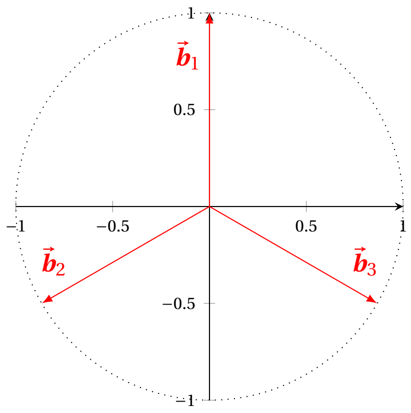
There are two things left to consider. Firstly, whether there is an integrality gap, as this will worsen our approximation, and secondly, how we get from a vector $\vec{\boldsymbol{b}}_u$ to an integer $x_u \in \{-1, 1\}$.

20.1.3 Max-Cut Integrality Gap

Consider the triangle graph G. Here, the max-cut is 2, since no matter what partitions we choose, we have 2 edges crossing the cut.



For the SDP, suppose we take the following unit vectors:



The dot products between these unit vectors is equivalent to $\cos(\theta)$, where θ is the angle between the two vectors. For these vectors, the angle is $\frac{2\pi}{3}$, giving $\cos\left(\frac{2\pi}{3}\right) = -\frac{1}{2}$, so the contribution of each edge to the cost is $\frac{3}{4}$, for a total cost of $\frac{9}{4} = 2.25$.

This means that there is indeed an integrality gap (it turns out that there are actually situations where the integrality gap is worse, but we won't prove the exact gap in this lecture).

20.1.4 Max-Cut Rounding

The GW-algorithm (hyperplane rounding) partitions vectors based on $\operatorname{sign}\left(\left\langle \vec{b}_u, \vec{g} \right\rangle\right)$ for a random vector $\vec{g} \sim \mathcal{N}(0, \mathbf{I})$, and this gives us our cut of the vertices.

Now, let us look at the approximation ratio for this rounding algorithm. Consider $\frac{\mathbb{E}[\text{size of cut}]}{\mathrm{OPT(QP)}}$. Ideally, we want this ratio to be as close to 1 as possible, but we want to lower bound this ratio.

We know that $\mathrm{OPT(QP)} \le \mathrm{OPT(SDP)}$, so this fraction is at most $\frac{\mathbb{E}[\text{size of cut}]}{\mathrm{OPT(SDP)}}$. Using indicator variables, this is equal to

$$\frac{\mathbb{E}[\text{size of cut}]}{\mathrm{OPT(SDP)}} = \frac{\sum_{e \in E} \mathbb{P}(e \text{ cut})}{\sum_{\substack{e \in E \\ e=(u,v)}} \frac{1 - \left\langle \vec{b}_u, \vec{b}_v \right\rangle}{2}}.$$

Suppose we focus on the plane containing $\vec{b}_u$ and $\vec{b}_v$, for some specific vertices u and v.

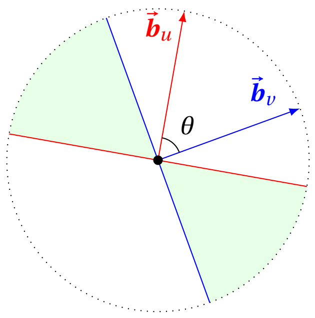


Figure 20.1: Plane containing $\vec{b}_u$ and $\vec{b}_v$, along with the region in which $\vec{g}$ would cut the edge between u and v.

We put a vector in one partition if $\left\langle \vec{b}_u, \vec{g} \right\rangle < 0$, and we put it in the other partition if $\left\langle \vec{b}_u, \vec{g} \right\rangle > 0$. This directly corresponds to whether $\cos(\theta) < 0$ or whether $\cos(\theta) > 0$. That is, whether $\vec{g}$ lies above or below the line perpendicular to the vector $\vec{b}_u$ (and similarly for $\vec{b}_v$).

The regions in green in Fig. 20.1 are the regions for which $\vec{g}$ would put u and v on two different partitions. These regions has a total angle of 2θ radians (each section covers θ radians), so the probability of $\vec{g}$ landing in these regions is $\frac{2\theta}{2\pi} = \frac{\theta}{\pi}$, and as such the probability that $e = (u, v)$ is cut is $\frac{\theta}{\pi}$.

This gives a ratio of

$$\frac{\mathbb{E}[\text{size of cut}]}{\text{OPT(SDP)}} = \frac{\sum_{e\in E} \frac{\theta(u,v)}{\pi}}{\sum_{\substack{e\in E\\ e=(u,v)}} \frac{1-\cos(\theta(u,v))}{2}} = \frac{2}{\pi} \cdot \frac{\sum_{\substack{e=(u,v)\\ e\in E}} \theta(u,v)}{\sum_{\substack{e=(u,v)\\ e\in E}} 1-\cos(\theta(u,v))}.$$

Observe that if $\frac{\alpha_i}{\beta_i} \geq z$ for all i, then $\frac{\sum \alpha_i}{\sum \beta_i} \geq z$. This means that the approximation ratio is at least

$$\gamma_{GW} = \inf_{\theta\in[0,2\pi)} \frac{2}{\pi} \cdot \frac{\theta}{1-\cos\theta}.$$

This is approximately 0.87856, which is better than $\frac{1}{2}$.

It is known that the integrality gap of the SDP relaxation of max-cut can be made arbitrarily close to γ_{GW}. That is, there is a way to construct a graph with integrality gap $\gamma_{GW} - \varepsilon$ for any ε.

It's also known (assuming the unique games conjecture) that we cannot beat this approximation ratio for max-cut [KKM+07].

20.2 Streaming Algorithms

In streaming algorithms, we're working under the assumption that we have very low memory. Typically, we'd store an entire data structure to allow us to perform queries or operations quickly. In streaming algorithms, we don't even remember most of the data.

In particular, streaming algorithms involve dynamic data structures with sublinear memory.

For example, suppose we want to maintain a database of numbers, supporting a query of the sum of all numbers in the database. We don't actually need to remember all of the numbers in the database, and we can just keep a counter containing the sum of numbers so far.

Some more streaming algorithms include:

- Count the number of distinct integers in a set (given insertions and queries, and no deletions)

  If all integers are between 1 and n, the trivial solution is to maintain a bit-vector of length n, and querying sums the 1's in the bit vector.

  With randomized approximations, we can give an $(1 \pm \varepsilon)$-approximation with probability $\geq 1 - \delta$ with $O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta} + \log n)$ bits [KNW10]. That is, given ε and δ are constant, this reduces the space to $O(\log n)$ bits.

- Counting (given increments of the counter)

  With n increments, counting should take $\log n$ memory, since it takes $\log n$ bits to write down the number.

  With randomized approximations, we can give an $(1 \pm \varepsilon)$-approximation with probability $\geq 1 - \delta$ with $O(\log\log n + \log \frac{1}{\varepsilon} + \log\log \frac{1}{\delta})$ bits [NY22]. That is, given ε and δ are constant, this reduces the space to $O(\log\log n)$ bits.

20.2.1 Count Distinct (idealized)

One (idealized) streaming algorithm to count distinct integers is as follows. We first pick a random $h : [n] \to [0, 1]$. Here, this is idealized, since we can't actually store perfectly continuous real numbers, and the memory it takes to store the hash table is $O(n)$ (but we can use a 2-wise independent family of hash functions to fix this).

We then initialize $Z = 1$. To insert a number i, we set $Z = \min(Z, h(i))$.

For a query, suppose t is the number of distinct elements (we don't know this number). If $t = 1$, then we expect $Z = 0.5$, since the hash is uniform in $[0.1]$. If $t = 2$, then we should expect $Z = \frac{1}{3}$, since we're taking the minimum of two uniformly random reals. In general, we expect $Z = \frac{1}{t+1}$.

This means that we should return $\frac{1}{z} - 1$ as our estimate of t.

We claim that $\mathbb{E}[Z] = \frac{1}{t+1}$. (This is an order statistic of uniform random variables.)

We have

$$\mathbb{E}[Z] = \int_0^1 \mathbb{P}(Z > x)\,\mathrm{d}x = \int_0^1 \mathbb{P}(h(1) > x)^t\,\mathrm{d}x = \int_0^1 (1-x)^t\,\mathrm{d}x = \frac{1}{t+1}.$$

The variance $\mathrm{Var}(Z)$ can also be computed as $\mathrm{Var}(Z) = \mathbb{E}\left[Z^2\right] - \mathbb{E}[Z]^2$, where $\mathbb{E}\left[Z^2\right] = \frac{2}{(t+1)(t+2)}$, so the variance turns out to be $\Theta\left(\frac{1}{(t+1)^2}\right)$.

We'd like to say that with high probability, Z is close to $\frac{1}{t+1}$. If we knew that $Z = (1\pm\varepsilon)\cdot\frac{1}{t+1}$, then $\frac{1}{z} - 1 = \frac{1}{1\pm\varepsilon}\cdot(t+1) - 1$, which boils down to approximately $(1 \pm O(\varepsilon))\,t$.

We can use Chebyshev's inequality to bound

$$\mathbb{P}\left(\left|Z-\mu\right| \geq \varepsilon\mu\right) < \frac{\mathrm{Var}(Z)}{\varepsilon^2\mu^2}.$$

We have that the mean $\mu \approx \frac{1}{t}$ and $\mathrm{Var}(Z) \approx \frac{1}{t^2}$, but most of the time the RHS is greater than 1, which is quite useless.

So, suppose we modify our algorithm slightly to store k different independent hash functions $h_1, \dots, h_k$, and store independent $Z_1, \dots, Z_k$ for each one of the hash functions. That is, each time we have a new number i, we take $Z_1 \leftarrow \min(Z_1, h_1(i))$, $Z_2 \leftarrow \min(Z_2, h_2(i))$, …, $Z_k \leftarrow \min(Z_k, h_k(i))$.

We then define $\tilde{Z} = \frac{1}{k}\sum_{i=1}^k Z_i$. Here, the expectation is still $\frac{1}{t+1}$, but the variance is now $\frac{1}{k}\mathrm{Var}(Z)$.

If we set $k = \frac{C}{\varepsilon^2}$, then Chebyshev's gives a failure probability

$$\mathbb{P}\left(\left|\tilde{Z}-\mu\right| \geq \varepsilon\mu\right) < \frac{\mathrm{Var}(Z)}{k\varepsilon^2\mu^2} = \frac{\frac{1}{t^2}}{\left(\frac{C}{\varepsilon^2}\right)\varepsilon^2\left(\frac{1}{t}\right)^2} = \frac{1}{C},$$

If we want failure probability δ, then we can set $k = \frac{C}{\varepsilon^2}\cdot\frac{1}{\delta}$, but this causes the memory to scale linearly with δ.

To resolve this, we can just run this algorithm T times, and by Chernoff, half of these copies fail with probability exponentially small in T. As such, setting $T = C\ln\frac{1}{\delta}$ gives a probability exponentially small in δ. Using the median of these copies, this gives $\frac{1}{\varepsilon^2}\ln\frac{1}{\delta}$ space.

20.2.2 Deterministic and Exact Count Distinct

Is a deterministic and exact algorithm possible in $o(n)$ space?

We claim that if there exists a space S algorithm A, then there exists an injection from $\{0,1\}^n \to \{0,1\}^S$. In particular, this means that we'd be able to compress bitstrings of length n as bitstrings of length S, which implies that $n \leq S$.

We'll define the injection f as follows: given $x \in \{0,1\}^n$, then $f(x)$ is the memory of A run on the indices of bits where x is 1. That is, we take the indices where x is 1, and feed these numbers as inputs to A.

To show that this is actually an injection, we'll give an algorithm to find the inverse of some memory contents. We can feed numbers from 1 to n into A, starting with the memory contents given. This means that we're starting from a point where all the indices where x is 1 have already been fed into A to count.

If the index we now give A changes the result of `query()`, then it must not have seen that index before, and we know the bit at the index must be 0. Otherwise, if it does not change the result of `query()`, then A must have seen this number already, and we know the bit at the index must be 1. In this fashion, we can fully reconstruct the original bitstring, showing that f is indeed injective.

4/4/2023

Lecture 21

Linear Programming: Simplex Method

21.1 Simplex Method

Recall that LPs are defined as

$$\begin{aligned} \min_{\vec{x}} \quad & \vec{c}^T \vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} \leq \vec{b}, \\ & \vec{x} \geq \vec{0} \end{aligned}$$

The simplex method (due to Dantzig in 1947) assumes that the LP is written in standard form:

$$\begin{aligned} \min_{\vec{x}} \quad & \vec{c}^T \vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} = \vec{b}, \\ & \vec{x} \geq \vec{0} \end{aligned}$$

Here, $\mathbf{A} \in \mathbb{R}^{m \times n}$ where $n \geq m$. We can always convert any LP into standard form:

- We can turn inequality constraints into equality constraints with *slack variables.*

  For the inequality $\langle a_i, x \rangle \leq b_i$, we can introduce the slack variable s_i to give the constraint $\langle a_i, x \rangle + s_i = b_i$ and $s_i \geq 0$.

- If we allow x_i to be negative, we can always write $x_i = x_i^+ - x_i^-$, where $x_i^+, x_i^- \geq 0$.

We'll also assume at some point that the rows of $\mathbf{A}$ are linearly independent. This isn't hugely crucial, but it'll make the discussion today simpler. There are algorithms to find linearly dependent rows, in which case we can just remove it—either this linearly dependent row is redundant, or it is contradictory.

An LP will always describe a polytope—we'll see that OPT is always achieved at a vertex of the polytope. This means that we can start at a vertex, and greedily move to an adjacent vertex that decreases the cost.

Note that we aren't given the vertices, so we'll have to figure out where vertices are, and how we can move to adjacent vertices efficiently.

In pseudocode, the simplex method proceeds as follows:

- Find a starting vertex $\vec{x}_0$
- While $\vec{x}_i$ is not optimal, greedily move to a better neighboring vertex $\vec{x}_{i+1}$
- Halt, and return $\vec{x}_T$.

Note that the first step here is just as hard as solving LPs; there's an efficient reduction from optimizing LPs to finding a feasible point in the LP.

To do so, we can try to determine whether OPT $\leq \alpha$; this gives the LP (with zero cost)

$$\begin{aligned} \min \quad & \vec{0}^T \vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} \leq \vec{b}, \\ & \vec{c}^T \vec{x} \leq \alpha, \\ & \vec{x} \geq \vec{0} \end{aligned}$$

If we can find a feasible point in this LP, then we know that OPT $\leq \alpha$, so we can just binary search to any arbitrary precision.

21.1.1 Finding a starting vertex

Before we address this point, let us give some definitions:

Definition 21.1: LP Feasibility, Boundedness, Vertex

Let P be the set of all $\vec{\boldsymbol{x}}$ satisfying all constraints:

$$P = \{\vec{\boldsymbol{x}} \mid \mathbf{A}\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}, \vec{\boldsymbol{x}} \geq \vec{\boldsymbol{0}}\}.$$

We say that $\vec{\boldsymbol{x}}$ *is feasible* if $\vec{\boldsymbol{x}} \in P$, and that the *LP is feasible* if $P \neq \varnothing$. We say that the LP is *bounded* if $\mathrm{OPT} > -\infty$.

We call $x \in P$ a vertex if $x + y \in P \wedge x - y \in P \implies y = 0$. That is, there is no direction such that we can move in that direction and the opposite direction such that we stay feasible.

To find the starting vertex, we want to find

$$\begin{aligned}
\min_{\vec{\boldsymbol{x}}, t} \quad & t \\
\text{s.t.} \quad & \mathbf{A}\vec{\boldsymbol{x}} = (1-t)\vec{\boldsymbol{b}}, \\
& \vec{\boldsymbol{x}}, t \geq 0, \\
& t \geq 1
\end{aligned}$$

(Note that we can turn the $t \leq 1$ constraint into standard form by adding a slack $s_t \geq 0$ with the modified constraint $t + s_t = 1$.)

If the original LP is feasible, then the best t we can hope for is $t = 0$. This means that the OPT for this LP is $t = 0$ if and only if the original LP is feasible. We'll then use the simplex method to solve this new LP.

However, this means we need a starting vertex in *this* LP to run simplex. However, this LP has a pretty easy starting vertex: $\vec{\boldsymbol{x}} = \vec{\boldsymbol{0}}$ and $t = 1$.

This is definitely a feasible solution, and this is also a vertex. We can't make $\vec{\boldsymbol{x}}$ negative, so y cannot have support on $\vec{\boldsymbol{x}}$ (either the direction or the reverse direction will make $\vec{\boldsymbol{x}}$ negative). Similarly, it cannot have support on t, since it'll always make $t > 1$ in some direction or its reverse.

21.1.2 OPT is at a vertex

We'll now show that if an LP is bounded and feasible, then for all $\vec{\boldsymbol{x}} \in P$, there exists a vertex $\vec{\boldsymbol{x}}' \in P$ such that $\vec{\boldsymbol{c}}^T\vec{\boldsymbol{x}}' \leq \vec{\boldsymbol{c}}^T\vec{\boldsymbol{x}}$. That is, there always exists a vertex $\vec{\boldsymbol{x}}'$ that does at least as well as $\vec{\boldsymbol{x}}$.

Suppose $\vec{\boldsymbol{x}}$ is not a vertex (otherwise, we're done with $\vec{\boldsymbol{x}}' = \vec{\boldsymbol{x}}$). By definition, this implies that there exists a $\vec{\boldsymbol{y}}$ such that

$$\begin{aligned}
\mathbf{A}(\vec{\boldsymbol{x}} + \vec{\boldsymbol{y}}) &= \vec{\boldsymbol{b}} \\
\mathbf{A}(\vec{\boldsymbol{x}} - \vec{\boldsymbol{y}}) &= \vec{\boldsymbol{b}} \\
\vec{\boldsymbol{x}} + \vec{\boldsymbol{y}}, \vec{\boldsymbol{x}} - \vec{\boldsymbol{y}} &\geq 0
\end{aligned}$$

Subtracting the first two equalities, we have $\mathbf{A}\vec{\boldsymbol{y}} = \vec{\boldsymbol{0}}$. WLOG, suppose that $\vec{\boldsymbol{c}}^T\vec{\boldsymbol{y}} \geq 0$ (otherwise we can just flip the sign of $\vec{\boldsymbol{y}}$). If $\vec{\boldsymbol{c}}^T\vec{\boldsymbol{y}} = 0$, then WLOG there exists a j such that $y_j < 0$; $\vec{\boldsymbol{y}}$ is not the zero vector, so there is a coordinate that is negative for either $\vec{\boldsymbol{y}}$ or $-\vec{\boldsymbol{y}}$.

This means that we have two cases

1. $\vec{\boldsymbol{c}}^T\vec{\boldsymbol{y}} = 0$, so there exists a j such that $y_j < 0$.

   Note that $x_i = 0$ implies that $y_i = 0$. That is, the support of $\vec{\boldsymbol{y}}$ is subset of the support of $\vec{\boldsymbol{x}}$. This is because if $x_i = 0$, then either adding or subtracting from this coordinate will make it negative, which is not allowed.

Consider $\vec{x} + t\vec{y}$, for $t \geq 0$. This is also a feasible solution, so let us start at $t = 0$ and gradually increase t until some coordinate in $\vec{y}$ that was negative now becomes zero.

This must happen, since the fact that $y_j < 0$ means that $x_j > 0$ (it cannot be 0, as explained prior), and we're gradually making it more and more negative until it becomes 0.

Here, it turns out that

$$t^* = \min_{i:y_i<0} \left| \frac{x_i}{y_i} \right|.$$

We'd then change $\vec{x}$ to be $\vec{x} + t^* \vec{y}$.

2. $\vec{c}^T \vec{y} < 0$, so for all j, $y_j \geq 0$. That is, $\vec{y} \geq 0$. (WLOG we can choose either $\vec{y}$ or $-\vec{y}$ that makes $\vec{c}^T \vec{y} < 0$, as mentioned earlier.)

   Note that $\vec{x} + t\vec{y} \in P$ for all $t \geq 0$. This is because we're always adding a nonnegative amount to $\vec{x}$. However, this means that OPT $= -\infty$, since this will make the cost smaller and smaller.

   This means that case 2 is not possible if the LP is bounded.

Every time we enter case 1, some coordinate of $\vec{x}$ becomes 0. This means that at some point, we'll run out of coordinates to make 0, and we'll halt. This means that we'll always halt at some vertex at the very end of this process.

In other words, this is an iterative procedure that transforms $\vec{x}$ while decreasing the cost, until we reach a vertex.

Definition 21.2: Basis

Given a vertex $\vec{x} \in P$, we define the *basis* $B_{\vec{x}}$ of $\vec{x}$ as

$$B_{\vec{x}} = \{j \in [n] : x_j > 0\}.$$

That is, the basis of $\vec{x}$ is the support of $\vec{x}$.

Lemma 21.3

We now claim that $\vec{x} \in P$ is a vertex if and only if the columns of $A_{B_{\vec{x}}}$ are linearly independent. Here, $\mathbf{A}_{B_{\vec{x}}}$ is the matrix formed by the columns of $\mathbf{A}$ restricted to the columns indexed by $B_{\vec{x}}$.

Proof. Note that we have

$$\begin{aligned} \mathbf{A}\vec{x} &= \sum_{i=1}^{n} x_i a_i \\ &= \sum_{i \in B_x} x_i A_i \\ &= \mathbf{A}_{B_{\vec{x}}} \end{aligned}$$

So $\mathbf{A}\vec{x} = \mathbf{A}_{B_{\vec{x}}} \vec{x} = \vec{b}$. This means that $\vec{x}_{B_{\vec{x}}} = \mathbf{A}_{B_{\vec{x}}}^{-1} \vec{b}$.

($\Longrightarrow$) Suppose $\vec{x} \in P$ is not a vertex. This means that the there exists a $\vec{y} \neq 0$ such that $\mathbf{A}(\vec{x} + \vec{y}) = \vec{b}$ and $\mathbf{A}(\vec{x} - \vec{y}) = \vec{b}$.

This is equivalent to saying that $\mathbf{A}\vec{y} = \mathbf{A}_{B_{\vec{y}}} \vec{y}' = \vec{0}$. Since $B_{\vec{y}} \subseteq B_{\vec{x}}$, then $A_{B_{\vec{x}}} \vec{y}'' = 0$. This means that the columns of $\mathbf{A}_{B_{\vec{x}}}$ are linearly dependent.

Here, we're just restricting $\vec{y}$ to the coordinates indexed by $B_{\vec{y}}$ or $B_{\vec{x}}$ to form $\vec{y}'$ and $\vec{y}''$ respectively, to make the dimensions work out.

($\Longleftarrow$) In the other direction, Suppose the columns of $\mathbf{A}_{B_{\vec{x}}}$ are linearly dependent. This means that there exists a $\vec{y} \neq 0$ such that $\mathbf{A}_{B_{\vec{x}}} \vec{y} = 0$. This means that there exists a $\vec{y}' \in \mathbb{R}^n$ where the support of y is also contained in $B_{\vec{x}}$.

If we zero-pad $\mathbf{A}_{B_{\vec{x}}}$ to give $\mathbf{A}$, we now have $\mathbf{A}\vec{y}' = \vec{0}$. This means that there exists a t such that $\vec{x} + t\vec{y}' \in P$ and $\vec{x} - t\vec{y}' \in P$, so $\vec{x}$ is not a vertex. □

21.1.3 Moving to neighboring vertices

For any vertex $\vec{x}$ such that $|B_{\vec{x}}| < m$, we'll artificially add more linearly independent columns from $\mathbf{A}$ until $|B_{\vec{x}}| = m$.

We'll now determine how to move to a neighboring basis of a vertex. We can rewrite the LP as

$$\begin{aligned} \min \quad & \vec{c}_B^T \vec{x}_B + \vec{c}_N^T \vec{x}_N \\ \text{s.t.} \quad & \mathbf{A}_B \vec{x}_B + \mathbf{A}_N \vec{x}_N = \vec{b}, \\ & \vec{x}_B, \vec{x}_N \geq 0 \end{aligned}$$

All we've done here is partition the coordinates into ones in B and ones in $N = [n] \setminus B$.

If we rearrange the first constraint, we get

$$\vec{x}_B = \mathbf{A}_B^{-1}\vec{b} - \mathbf{A}_B^{-1}\mathbf{A}_N \vec{x}_N.$$

This means that the cost is now

$$\vec{c}_B^T\left(\mathbf{A}_B^{-1}\vec{b} - \mathbf{A}_B^{-1}\mathbf{A}_N\vec{x}_N\right) + \vec{c}_N^T\vec{x}_N.$$

Note that $\vec{c}_B^T \mathbf{A}_B^{-1}\vec{b}$ is a constant that does not depend on $\vec{x}$, so we only need to optimize for

$$\underbrace{\left(\vec{c}_N - \mathbf{A}_N^T\left(\mathbf{A}_B^{-1}\right)^T \vec{c}_B\right)}_{\vec{\tilde{c}}_N}{}^T \vec{x}_N.$$

This means that there exists a better neighboring basis (a neighboring vertex) if there exists a j such that $(\vec{\tilde{c}}_N)_j < 0$.

If there is an entry in $\vec{\tilde{c}}_N$ that is negative, we can just increase the corresponding entry in $\vec{x}_N$ to decrease the cost. Because we defined $\vec{x}_B$ in terms of $\vec{x}_N$, increasing an entry in $\vec{x}_N$ may decrease some entries in $\vec{x}_B$. When such an entry in $\vec{x}_B$ becomes 0, it gets kicked out of the basis, and we've just moved to a new vertex.

The only issue here is that we've added to $B_{\vec{x}}$ until it has size m, so some entries in $\vec{x}_B$ may actually be 0. This means that it is possible for us to not make any progress at all, otherwise $\vec{x}$ will become negative. To resolve this, we can swap out the offending entry in $\vec{x}_B$, but we need to be careful in how we do this.

We won't cover how to do this, but it involves "pivot rules", such as Bland's Rule in 1977 [Bla77]. We don't have any pivot rules that prevent exponential time, but there are pivot rules that guarantee termination.

4/6/2023

Lecture 22

Linear Programming: Strong Duality, Complementary Slackness, Ellipsoid Method

Recall that in the simplex method, we'd like to solve the LP

$$\begin{aligned} \min_{\vec{x}} \quad & \vec{c}^T\vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} = \vec{b}, \\ & \vec{x} \geq \vec{0} \end{aligned}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $n \geq m$.

Each vertex is determined by a basis $B \subseteq [n]$, where $|B| = m$; a vertex v_B is defined as $\mathbf{A}_B^{-1}\vec{\boldsymbol{b}}$. Given a basis B, we can then rewrite the LP as

$$\begin{aligned} \min \quad & \vec{\boldsymbol{c}}_B^T\vec{\boldsymbol{x}}_B + \vec{\boldsymbol{c}}_N^T\vec{\boldsymbol{x}}_N \\ \text{s.t.} \quad & \mathbf{A}_B\vec{\boldsymbol{x}}_B + \mathbf{A}_N\vec{\boldsymbol{x}}_N = \vec{\boldsymbol{b}}, \\ & \vec{\boldsymbol{x}}_B, \vec{\boldsymbol{x}}_N \geq 0 \end{aligned}$$

If we rearrange the first constraint, we get

$$\vec{\boldsymbol{x}}_B = \mathbf{A}_B^{-1}\vec{\boldsymbol{b}} - \mathbf{A}_B^{-1}\mathbf{A}_N\vec{\boldsymbol{x}}_N,$$

so we only need to optimize for

$$\underbrace{\left(\vec{\boldsymbol{c}}_N - \mathbf{A}_N^T\left(\mathbf{A}_B^{-1}\right)^T\vec{\boldsymbol{c}}_B\right)}_{\vec{\tilde{\boldsymbol{c}}}_N}{}^T\vec{\boldsymbol{x}}_N.$$

Here, $\vec{\tilde{\boldsymbol{c}}}_N$ is called the "reduced cost vector".

Simplex halts when $\vec{\tilde{\boldsymbol{c}}}_N \geq 0$, and otherwise we gradually increase the coordinate x_j that corresponds to $(\vec{\tilde{\boldsymbol{c}}}_N)_j < 0$ until we can't anymore. We then repeat the process with the new values of these vectors.

We'll see today that this halting condition of $\vec{\tilde{\boldsymbol{c}}}_N \geq 0$ will allow us to construct an optimal dual solution, which also proves strong duality.

22.1 Strong Duality

Strong duality states that if the primal LP is bounded and feasible, then the dual LP is bounded and feasible, and moreover OPT(primal) = OPT(dual).

Suppose the simplex algorithm achieved $\vec{\tilde{\boldsymbol{c}}}_N \geq 0$ for some vertex with the corresponding basis B.

We know that $\vec{\boldsymbol{x}}_B = \mathbf{A}_B^{-1}\vec{\boldsymbol{b}}$, and we want $\vec{\boldsymbol{b}}^T\vec{\boldsymbol{y}} = \vec{\boldsymbol{c}}_B^T\vec{\boldsymbol{x}}_B$ for a dual feasible $\vec{\boldsymbol{y}}$.

We know that $\vec{\boldsymbol{c}}_B^T\vec{\boldsymbol{x}}_B = \vec{\boldsymbol{c}}_B^T\mathbf{A}_B^{-1}\vec{\boldsymbol{b}}$, and we can rewrite the LHS as $y, >^T \vec{\boldsymbol{b}}$, which means that $\vec{\boldsymbol{y}}^T = \vec{\boldsymbol{c}}_B^T\mathbf{A}_B^{-1}$, or $\vec{\boldsymbol{y}} = (\mathbf{A}_B^{-1})^T\vec{\boldsymbol{c}}_B$.

Now the question is whether this is feasible for the dual LP. Here, the dual is

$$\begin{aligned} \max_{\vec{\boldsymbol{y}}} \quad & \vec{\boldsymbol{b}}^T\vec{\boldsymbol{y}} \\ \text{s.t.} \quad & \mathbf{A}^T\vec{\boldsymbol{y}} \leq \vec{\boldsymbol{c}} \end{aligned}$$

This means that we only need to check that $\mathbf{A}^T\vec{\boldsymbol{y}} \leq \vec{\boldsymbol{c}}$.

Suppose we look at the slack vector $\vec{\boldsymbol{s}} := \vec{\boldsymbol{c}} - \mathbf{A}^T\vec{\boldsymbol{y}}$. Here, $\vec{\boldsymbol{y}}$ is dual variable when the slack vector is positive.

There are n entries in $\vec{\boldsymbol{s}}$, so let us look at the entries corresponding to B and those that don't.

For entries corresponding to B, we have

$$\vec{\boldsymbol{s}}_B = \vec{\boldsymbol{c}}_B - \mathbf{A}_B^T\vec{\boldsymbol{y}} = \vec{\boldsymbol{c}}_B - \mathbf{A}_B^T\left(\mathbf{A}_B^{-1}\right)^T\vec{\boldsymbol{c}}_B = \vec{\boldsymbol{c}}_B - \underbrace{\mathbf{A}_B^T\left(\mathbf{A}_B^T\right)^{-1}}_{\mathbf{I}}\vec{\boldsymbol{c}}_B = \vec{\boldsymbol{c}}_B - \vec{\boldsymbol{c}}_B = 0.$$

For entries corresponding to $N = [n] \setminus B$, we have

$$\vec{\boldsymbol{s}}_N = \vec{\boldsymbol{c}}_N - \mathbf{A}_N^T\left(\mathbf{A}_B^{-1}\right)^T\vec{\boldsymbol{c}}_B = \vec{\tilde{\boldsymbol{c}}}_N \geq 0.$$

22.2 Simplex Runtime

Given a polytope $P \subseteq \mathbb{R}^n$ defined as the intersection of m half-spaces, we can define a graph G_P where the vertices of G_P are the vertices of the polytope, and the edges are adjacent vertices of the polytope.

In principle, any of the vertices could be OPT; this means that the graph must have low diameter (i.e. we must be able to get from the starting vertex to the OPT vertex in polynomial time).

The Hirsch conjecture from 1957 says that the diameter of G_p is at most $m - n$. Santos in 2011 showed that the Hirsch conjecture is false; there exists a polytope in 32 dimensions ($n = 32$) that is the intersection of 86 halfspaces ($m = 86$) with diameter at least 44.

This means that we're quite far to showing whether there is a polynomial time implementation of simplex; the only kinds of lower bounds we've shown so far are linear, when we actually want a superpolynomial lower bound to prove that simplex cannot be run in polynomial time.

However, despite these shortcomings, simplex is used very frequently in practice. Spielman and Teng in 2004 introduced smoothed analysis, wanting to prove that the hard examples used to show exponential time cases for simplex were very carefully constructed, and unnatural. They proved that there exists a pivoting rule such that for all inputs $I = (\mathbf{A}, \vec{\boldsymbol{b}}, \vec{\boldsymbol{c}})$, the expected runtime of simplex on perturb(I) is at most polynomial.

Here, we're taking the expectation over the perturbation σ; we add a bunch of iid Gaussians with very small variance to each entry of $\mathbf{A}$, $\vec{\boldsymbol{b}}$, and $\vec{\boldsymbol{c}}$.

This is somewhat surprising, as it says that even for the hard examples given, if we perturb the inputs very slightly, we'd actually get polynomial runtime.

22.3 Complementary Slackness

Suppose that the primal and dual are both bounded and feasible with optimal solutions $\vec{\boldsymbol{x}}$ and $\vec{\boldsymbol{y}}$. Here, we're assuming that the primal is in standard form. We define the slack variables again as $\vec{\boldsymbol{s}} := \vec{\boldsymbol{c}} - \mathbf{A}^T \vec{\boldsymbol{y}}$.

Then, for all $i = 1, \dots, n$, we must have $x_i > 0 \implies s_i = 0$ and $s_i > 0 \implies x_i = 0$.

We know that $\vec{\boldsymbol{c}}^T \vec{\boldsymbol{x}} - \vec{\boldsymbol{b}}^T \vec{\boldsymbol{y}} = 0$, and we also know that

$$\begin{aligned}
\sum_{i=1}^n x_i s_i &= \sum_{i=1}^n x_i \left(c_i - \left(\mathbf{A}^T \vec{\boldsymbol{y}}\right)_i\right) \\
&= \vec{\boldsymbol{c}}^T \vec{\boldsymbol{x}} - \vec{\boldsymbol{x}}^T \mathbf{A}^T \vec{\boldsymbol{y}} \\
&= \vec{\boldsymbol{c}}^T \vec{\boldsymbol{x}} - (\mathbf{A}\vec{\boldsymbol{x}})^T \vec{\boldsymbol{y}} \\
&= \vec{\boldsymbol{c}}^T \vec{\boldsymbol{x}} - \vec{\boldsymbol{b}}^T \vec{\boldsymbol{y}}
\end{aligned}$$

This means that $\sum_{i=1}^n x_i s_i = 0$; in particular, it is impossible that both x_i and s_i are nonzero, otherwise the sum would be nonzero. This proves commentary slackness.

22.4 Ellipsoid Algorithm

The ellipsoid algorithm is due to Khachiyan in 1979 [Kha79; Kha80], and is the first polynomial time algorithm for LPs. The ellipsoid algorithm is polynomial in n, m, and the bit complexity L (i.e. all numbers are at most L bits).

However, ellipsoid only solves the feasibility problem for LPs; we only give $\mathbf{A}$ and $\vec{\boldsymbol{b}}$, and it gives us an $\vec{\boldsymbol{x}}$ that is feasible.

This is thus a weakly polynomial time algorithm, as it depends on the bit complexity; it's an open question of whether there exists a strongly polynomial time algorithm for solving LPs, i.e. an algorithm that has runtime polynomial in only n and m.

It also turns out that there are some LPs with an exponential number of constraints that it can solve in polynomial time; this can be especially useful if the LP formulation is not the natural formulation of a given problem.

Firstly, let's define what an ellipsoid is. Given a vector $\vec{\boldsymbol{\alpha}} \in \mathbb{R}^n$ and a symmetric PSD matrix $\mathbf{A} = \mathbf{B}^T\mathbf{B}$, an ellipsoid is the set

$$\begin{aligned} E &= \{\vec{\boldsymbol{x}} \mid (\vec{\boldsymbol{x}} - \vec{\boldsymbol{\alpha}})^T \mathbf{A}(\vec{\boldsymbol{x}} - \vec{\boldsymbol{\alpha}} \leq 1\} \\ &= \{\vec{\boldsymbol{x}} \mid \|\mathbf{B}(\vec{\boldsymbol{x}} - \vec{\boldsymbol{\alpha}})\|_2^2 \leq 1\} \end{aligned}$$

In the ellipsoid algorithm, we want to find some $\vec{\boldsymbol{x}} \in P := \{\vec{\boldsymbol{x}} \mid \mathbf{A}\vec{\boldsymbol{x}} \leq \vec{\boldsymbol{b}}\}$.

Here, P is a polytope; it'll first start with an ellipsoid that definitely contains P. To get this starting ellipsoid, we use the fact that we're working with bounded precision numbers. In particular, vertices are given by linear systems, and linear systems use formulas involving these bounded precision numbers. This means that we can bound the size of the numbers in any given vertex, and we can choose our starting ellipsoid to be a very large sphere that definitely contains all of the vertices.

Suppose we're given a subroutine that determines whether a given point is feasible, and gives us a separating hyperplane if the point is infeasible. The ellipsoid algorithm takes the center of the ellipsoid (i.e. $\vec{\boldsymbol{\alpha}}$), and queries whether the center is feasible.

If the point is feasible, then we're done. Otherwise, we look at the intersection of the starting ellipsoid and the halfspace, and find the minimum volume ellipsoid that contains this intersecting region. There is a closed formula to compute this ellipsoid, so we compute this new ellipsoid and repeat the process. That is, we query again the center of this new ellipsoid, and compute a new ellipsoid if the center is infeasible.

The ellipsoid algorithm is thus:

- Let E_0 be the initial ellipsoid; it's an ellipsoid centered at $\vec{\mathbf{0}}$ with a large radius.
- For $j = 1, \ldots, k$:
  - Let $\vec{\boldsymbol{x}}_{j-1}$ be the center of E_{j-1}.
  - If $\vec{\boldsymbol{x}}_j$ is feasible, then return it
  - Otherwise, $\vec{\boldsymbol{x}}_j$ violates some halfspace H, so let E_j be the minimum volume ellipsoid containing $E_{j-1} \cap H$.
- Return "infeasible"

We'll choose k to ensure that this algorithm always terminates, and always produces a correct answer.

We claim that

$$\frac{\operatorname{vol}(E_{k+1})}{\operatorname{vol}(E_k)} \leq e^{-\frac{1}{2(n+1)}}.$$

That is, the volume of each successive ellipsoid decreases by some factor. With some $k \leq \operatorname{poly}(nL)$, we can make the volume after k steps much smaller than $\exp(-nL)$.

We'll also transform the polytope to make its volume larger than this quantity, which also takes care of edge cases where the original polytope has zero volume (ex. we have equality constraints). In particular, we define

$$P' = \left\{(\vec{\boldsymbol{x}}, z) \mid \mathbf{A}\vec{\boldsymbol{x}} \leq \vec{\boldsymbol{b}} + z \cdot \vec{\mathbf{1}}, -2^L \leq x_i \leq 2^L, -2^L \leq z \leq 2^L\right\}.$$

22.4.1 Optimizing LPs using Ellipsoid

We've already seen one way to go from an algorithm that solves feasibility to an algorithm that optimizes the LP. There's a different approach, which is to utilize strong duality, and create a new LP

$$\begin{aligned} \min \quad & 1 \\ \text{s.t.} \quad & \mathbf{A}\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}, \\ & \mathbf{A}^T\vec{\boldsymbol{y}} \leq \vec{\boldsymbol{c}}, \\ & \vec{\boldsymbol{x}} \geq 0, \\ & \vec{\boldsymbol{c}}^T\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}^T\vec{\boldsymbol{y}} \end{aligned}$$

This LP essentially solves both the dual and primal at the same time, so the only feasible point is OPT.

4/11/2023

Lecture 23

Interior Point Method, Newton's Method, Gradient Descent

Today, we'll be focusing on the interior point method, but we'll be skipping over some of the specifics, as it is more of a continuous optimization method—different from what we've seen so far in this course.

The interior point method takes in an LP of the form

$$\begin{aligned} \min \quad & \vec{c}^T \vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} \geq \vec{b} \end{aligned}$$

The basic idea is to find functions in terms of $\lambda \in [0, \infty)$, defined with

$$f_\lambda(x) := \lambda \vec{c}^T \vec{x} + p(s(\vec{x}))$$

where $\vec{s}(x) = \mathbf{A}\vec{x} - \vec{b}$ is the slack vector, and p is a function such that $p(\vec{z}) \to \infty$ if any $z_i \to 0$.

We'll start with λ being very small, and we'll find an optimal solution for $f_\lambda(\vec{x})$. If λ is small, then the cost function doesn't really matter, and most of the optimization is for $p(s(\vec{x}))$.

Here, we call p the "barrier function". For us today, we have

$$p(s(\vec{x})) := -\sum_{i=1}^{m} \ln(s(\vec{x})_i).$$

(Note that as $s(\vec{x})_i \to 0$, then $-\ln(s(\vec{x})_i) \to \infty$.)

The gradient and hessian of f_λ is as follows:

$$\begin{aligned} \nabla f_\lambda(\vec{x}) &= \lambda \vec{c} - \mathbf{A}^T \mathbf{S}_{\vec{x}}^{-1} \vec{\mathbf{1}} \\ \nabla^2 f_\lambda(\vec{x}) &= \mathbf{A}^T \mathbf{S}_{\vec{x}}^{-2} \mathbf{A} \end{aligned}$$

Here, $\mathbf{S}_{\vec{x}}$ is a diagonal matrix, where the ith diagonal entry is the ith entry of $s(\vec{x})$.

Conceptually, we start deep in the center of the polytope for the LP (the "analytic center", defined as the minimizer of $p(s(\vec{x}))$, i.e. the minimizer of f_λ for $\lambda = 0$), and we gradually increase λ, moving the point through the interior of the polytope corresponding to optimal points for each f_λ.

As $\lambda \to \infty$, the $p(s(\vec{x}))$ component matters less and less, and the cost function $\vec{c}^T \vec{x}$ matters more and more. This traces a path (called the "central path") from the analytic center to the optimal vertex of the polytope. However, since we'll be increasing λ in discrete steps, we'll never actually be *on* the central path, but we'll try to stay close to it.

On a high level, we'll start with an initial point, and apply Newton's method on larger and larger values of λ. In particular, we'll define an $\vec{x}$ to be "central" if it is the minimizer for f_λ; that is, $\left\|\nabla f_\lambda(\vec{x})\right\| = 0$. We'll say that an $\vec{x}$ is "awesome" if it is very very close to the minimizer; say $\left\|\nabla f_\lambda(\vec{x})\right\| \leq 0.0001$. We'll also say that an $\vec{x}$ is "okay" if it is not very very close, but close enough to the minimizer; say $\left\|\nabla f_\lambda(\vec{x})\right\| \leq 0.1$.

We have the following:

1. Set λ_0 to be very small, i.e. $\lambda_0 = \exp(-cL)$ where L is the precision of the problem
2. Get $\vec{x}(\lambda_0)$ which is "awesome" for λ_0
3. For $i = 1, \ldots, k$:
    - Set $\lambda_i \leftarrow (1+\gamma)\lambda_{i-1}$

- Run $O(1)$ Newton's method iterations on $\tilde{x}(\lambda_{i-1})$ for f_{λ_i} to obtain $\tilde{\vec{x}}(\lambda_i)$ that is "awesome" for λ_i.

Intuitively, if we have an "awesome" $\vec{x}$, then increasing λ by a very small amount should intuitively still make $\vec{x}$ "okay" for this new value of λ. Newton's method can then be used to improve this solution, getting us an "awesome" $\vec{x}$ again; we'd then repeat the process.

There are a couple questions to answer: How do we get a starting point $\tilde{\vec{x}}(\lambda_0)$? How big does k need to be? How small does γ need to be? Why is $\tilde{\vec{x}}(\lambda_i)$ "awesome" for λ_i from Newton's method?

23.1 Starting Point

To find an $\tilde{\vec{x}}(\lambda_0)$ which is "awesome" for λ_0, we first pick N to be really bit (say $\exp(cL)$). We'd then have a new LP

$$\begin{aligned} \min \quad & \vec{c}^T\vec{x} + N\vec{z} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} + \vec{z}\cdot\vec{\mathbf{1}} \geq \vec{b}, \\ & 0 \leq \vec{z} \leq 2^{L+1}, \\ & -2^{L+1}\vec{\mathbf{1}} \leq \vec{x} \leq 2^{L+1}\vec{\mathbf{1}} \end{aligned}$$

Intuitively, since N is very big, we don't want to put much value in $\vec{z}$, since it's being multiplied by a very large number.

This new LP has a very easy interior point $\vec{z} = \left\|\vec{b}\right\|_\infty$, $\vec{x} = \vec{\mathbf{0}}$.

Note that if $\mathbf{A}$ is PSD, then we define $\|\vec{x}\|_{\mathbf{A}} = \sqrt{\vec{x}^T\mathbf{A}\vec{x}}$.

We'll define "centrality" for λ as

$$\delta_\lambda(\vec{x}) := \left\|\nabla f_\lambda(\vec{x})\right\|_{(\nabla^2 f_\lambda(\vec{x}))^{-1}}.$$

With this definition, we now have concrete measures of "awesome" and "okay" from the big pictures previously. We'll say $\vec{x}$ is "awesome" if $\delta_\lambda(\vec{x}) \leq \frac{1}{100}$, and we'll say $\vec{x}$ is "okay" if $\delta_\lambda(\vec{x}) \leq \frac{1}{3}$.

Suppose $\lambda = 1$. Here,

$$\nabla f_\lambda(\vec{x}) = \vec{c} - \mathbf{A}^T\mathbf{S}^{-1}_{(\vec{x}_0,\vec{z}_0)}\vec{\mathbf{1}}.$$

This gradient isn't zero, but it *is* zero if we let the cost vector be

$$\vec{c}' = \mathbf{A}^T\mathbf{S}^{-1}_{(\vec{x}_0,\vec{z}_0)}\vec{\mathbf{1}}.$$

With this new cost vector used in f_λ from now on, at this point, we'll essentially run the Newtons' method loop backwards, decreasing λ slowly and using Newton's method to make our $\vec{x}_i$'s better for smaller values of λ.

Eventually, we'll make λ small enough, on the order of $\lambda_0 = \exp(-cL)$, and we'd have found an "awesome" $\tilde{\vec{x}}(\lambda_0)$ with this modified cost function. However, since λ_0 is so small, the cost function doesn't actually matter—we can just swap out $\vec{c}'$ with the original $\vec{c}$, and $\tilde{\vec{x}}(\lambda_0)$ will still be "awesome" for the original cost function.

Once we have the $\tilde{\vec{x}}(\lambda_0)$, we'll then continue the forward direction with the original $\vec{c}$ as normal.

23.2 Number of iterations

We'll now talk about how large k should be. It turns out that k should be around $O(L\sqrt{m})$, and that γ should be around $O(\frac{1}{\sqrt{m}})$. We'll show that if $\gamma = O(\frac{1}{\sqrt{m}})$, then $k = O(L\sqrt{m})$ is sufficient to give us enough precision.

Let $\vec{x}(\lambda) := \operatorname{argmin}_{\vec{x}} f_\lambda(\vec{x})$, and let $\vec{x}^*$ be the optimizer for the LP.

We have

$$\begin{aligned}
0 &= \langle \vec{\mathbf{0}}, \vec{x}(\lambda) - \vec{x}^* \rangle \\
&= \left\langle \lambda \vec{c} - \mathbf{A}^T \mathbf{S}_{\vec{x}(\lambda)}^{-1} \vec{\mathbf{1}}, \vec{x}(\lambda) - \vec{x}^* \right\rangle \\
\lambda \vec{c}^T \big(\vec{x}(\lambda) - \vec{x}^*\big) &= \vec{\mathbf{1}} \mathbf{S}_{\vec{x}(\lambda)}^{-1} \mathbf{A}\big(\vec{x}(\lambda) - \vec{x}^*\big) \\
&= \vec{\mathbf{1}}^T \mathbf{S}_{\vec{x}(\lambda)}^{-1} \big(s(\vec{x}(\lambda)) - s(\vec{x}^* 0)\big) \\
&= \sum_{i=1}^{n} \frac{s(\vec{x}(\lambda))_i - s(\vec{x}^*)_i}{s(\vec{x}(\lambda))_i} \\
&\le \sum_{i=1}^{n} \frac{s(\vec{x}(\lambda))_i}{s(\vec{x}(\lambda))_i} \\
&= m \\
\vec{c}^T \big(\vec{x}(\lambda) - \vec{x}^*\big) &\le \frac{m}{\lambda}
\end{aligned}$$

If we want to be ε away from optimizing the LP (since $\vec{c}^T(\vec{x}(\lambda) - \vec{x}^*)$ is exactly how far away we are from the optimal solution to the LP), then we want the RHS to be at most ε, i.e. we want $\frac{m}{\lambda} < \varepsilon$.

Since it's possible to round to the actual optimal vertex when $\varepsilon < \exp(-cL)$. This means that we're done when $\frac{m}{\lambda} < \varepsilon < \exp(-cL)$, or when $\lambda > m\exp(cL)$.

This means that we need $(1+\gamma)^k \lambda_0 > m\exp(cL)$, or that $(1+\gamma)^k > m\exp(2cL)$. This means that

$$k \ge \frac{1}{\gamma} \ln\big(m\exp(2cL)\big) = O(\sqrt{m} + \big(L + \log m\big)),$$

if $\gamma = O(\frac{1}{\sqrt{m}})$.

23.3 Continuous Optimization

We'll take a little detour into continuous optimization, namely Newton's method and gradient descent.

23.3.1 First-order Methods

Given oracle access to $f : \mathbb{R}^n \to \mathbb{R}$ and the gradient ∇f, and if we're promised that for all $\vec{x}$, $\alpha \mathbf{I} \preceq \nabla^2 f(\vec{x}) \preceq \beta \mathbf{I}$, our goal is to minimize $f(\vec{x})$.

For notation, we say that $\mathbf{A} \preceq \mathbf{B}$ if $\mathbf{B} - \mathbf{A}$ is PSD, or that for all $\vec{z}$, $\vec{z}^T \mathbf{A} \vec{z} \le \vec{z}^T \mathbf{B} \vec{z}$. (This is called a Loewner ordering.)

The basic idea is to start with some iterate $\vec{x}_0 \in \mathbb{R}^n$, and gradually move from $\vec{x}_k$ to $\vec{x}_{k+1}$ to make f smaller. Gradient descent is motivated by Taylor's theorem, which says that

$$f(\vec{x}_{k+1}) = f(\vec{x}_k) + \langle \nabla f(\vec{x}_k), \vec{x}_{k+1} - \vec{x}_k \rangle + \int_0^1 \int_0^t \langle \vec{x}_{k+1} - \vec{x}_k, \nabla^2 f(\vec{x}_\alpha)(\vec{x}_{k+1} - \vec{x}_k) \rangle \, \mathrm{d}x \, \mathrm{d}t.$$

Here, we define $\vec{x}_\alpha := \vec{x}_k + \alpha(\vec{x}_{k+1} - \vec{x}_k)$, i.e. the line between $\vec{x}_{k+1}$ and $\vec{x}_k$. The integrals are the exact error of the linear approximation.

At the end of the day, we can prove that the error is at most $\frac{\beta}{2} \|\vec{x}_{k+1} - \vec{x}_k\|_2^2$, meaning

$$f(\vec{x}_{k+1}) \le f(\vec{x}_k) + \langle \nabla f(\vec{x}_k), \vec{x}_{k+1}, \vec{x}_k \rangle + \frac{\beta}{2} \|\vec{x}_{k+1} - \vec{x}_k\|_2^2.$$

This means that we can choose $\vec{x}_{k+1}$ that optimizes the RHS. In doing so, we'd have an iteration formula of the form

$$\vec{x}_{k+1} = \vec{x}_k - \frac{1}{\beta} \nabla f(\vec{x}_k).$$

With this value of $\vec{x}_{k+1}$, we'd find that

$$f(\vec{x}_{k+1}) \le f(\vec{x}_k) - \frac{1}{2\beta}\left\|\nabla f(\vec{x}_k)\right\|_2^2.$$

We can do a similar analysis with some additional manipulation that the norm of the gradient $\left\|\nabla f(\vec{x}_k)\right\|_2^2$ is large—this means that we're actually making progress, decreasing the value of f by some large quantity.

In particular, we can show that

$$f(\vec{x}_{k+1}) - f(\vec{x}^*) \le \left(1 - \frac{\alpha}{\beta}\right)\left(f(\vec{x}_k) - f(\vec{x}^*)\right).$$

That is, at each iteration, we've improved our gap to optimality. This means that we can halve the gap to optimality in $O(\frac{\beta}{\alpha})$ oracle calls.

Under the same setup, this isn't actually the best rate of convergence. Only $O(\sqrt{\frac{\beta}{\alpha}})$ oracle calls is both achievable and optimal—this is called "accelerated gradient descent", due to Nesterov in the 1980s [Nes83].

One note here: we can't use gradient descent for the interior point method, since our hessian $\nabla^2 f_\lambda(\vec{x}) = \mathbf{A}^T\mathbf{S}_{\vec{x}}^{-2}\mathbf{A}$ is unbounded—as we get closer to the optimal point, the slacks in $\mathbf{S}_{\vec{x}}$ go to zero. Since we're dividing by these slacks, this makes the eigenvalues of the Hessian blow up.

23.3.2 Newton's Method

Given oracle access to f, ∇f, and $\nabla^2 f$, our goal is to minimize $f(\vec{x})$. Here, we're working under the assumption that the Hessian doesn't change too quickly on the line from $\vec{x}_k$ to $\vec{x}_{k+1}$. That is, for all k, with $\vec{x}_\alpha := \vec{x}_k + \alpha(\vec{x}_{k+1} - \vec{x}_k)$, we assume that

$$(1-\varepsilon)\nabla^2 f(\vec{x}_k) \preceq \nabla^2 f(\vec{x}_\alpha) \preceq (1+\varepsilon)\nabla^2 f(\vec{x}_k).$$

Note that

$$(1-\varepsilon)\mathbf{A} \preceq B \preceq (1+\varepsilon)\mathbf{A} \iff -\varepsilon\mathbf{I} \preceq \mathbf{A}^{-\frac{1}{2}}(\mathbf{B}-\mathbf{A})\mathbf{A}^{-\frac{1}{2}} \preceq \varepsilon\mathbf{I}.$$

The heart of Newton's method is that

$$f(\vec{x}_{k+1}) \approx f(\vec{x}_k) + \left\langle \nabla f(\vec{x}_k), \vec{x}_{k+1} - \vec{x}_k\right\rangle + \frac{1}{2}\left\langle \vec{x}_{k+1} - \vec{x}_k, \nabla^2 f(\vec{x}_k)(\vec{x}_{k+1} - \vec{x}_k)\right\rangle.$$

There is some error here, but if we optimize this approximation, then we get

$$\vec{x}_{k+1} = \vec{x}_k - \left(\nabla^2 f(\vec{x}_k)\right)^{-1}\nabla f(\vec{x}_k).$$

Lemma 23.1

If f is twice differentiable and our assumptions for Newton's method hold, then

$$\left\|\nabla f(\vec{x}_{k+1}\right\|_{(\nabla^2 f(\vec{x}_{k+1}))^{-1}} \le \frac{\varepsilon}{1-\varepsilon}\left\|\nabla f(\vec{x}_k\right\|_{(\nabla^2 f(\vec{x}_k))^{-1}}.$$

Note that these norms are essentially our centrality function $\delta_\lambda(\vec{x})$.

It turns out that the value of γ comes from looking at how much we can increase γ while still satisfying the assumptions of Newton's method; this gives the value of $\gamma = O(\frac{1}{\sqrt{m}})$ mentioned prior.

4/13/2023

Lecture 24

Multicalibration and Fairness in Prediction

Guest lecture: Michael Kim

These days we're seeing more of using algorithms and machine learning to make predictions about people.

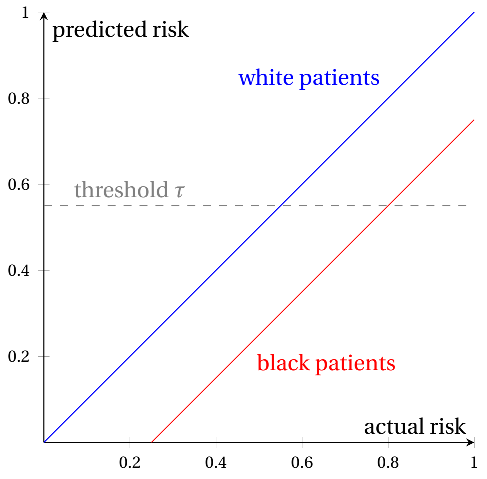
For example, suppose we're wanting to make an algorithm that takes in an individual's health record, and passes it into a program called the "predictor", which will output a value in $[0, 1]$, interpreted as a predicted probability of some health outcome (ex. probability of heart attack in the next 10 years).

The hope is that such predictors can be helpful and guide clinicians to provide better care for patients. As such, we'd like the predictor to have good accuracy, but we'd also like to analyze the accuracy across various problems and groups.

Obermeyer et. al. in 2019 [OPV+19] looked at various predictors.

We have a threshold of predicted probability, and if the risk is above the threshold, the patient qualifies for advanced care, and otherwise the patient does not qualify.

The ideal predictor would match up exactly with the actual probability. When looking at different demographic groups, it turns out that for white patients, the curve matched up pretty closely to the ideal curve, but for black patients, the curve was consistently right-shifted.



This means that black patients needed to be considerably more sick in order to get advanced care. In particular, the predictor was not very accurate in predicting risk for all patients. This is the kind of inaccuracy and fairness that we'll be looking at today.

24.1 Setup

We have a domain $\mathcal{X} \subseteq \{0,1\}^d$ of boolean features. An individual $x \in \mathcal{X}$ represents an individual.

We also have an outcome space $\mathcal{Y} \in \{0,1\}$ (ex. yes/no to having a heart attack).

The distribution $\mathcal{D}$ is defined on $\mathcal{X} \times \mathcal{Y}$. We'll assume that we know this distribution.

The predictor is a function $p \colon \mathcal{X} \to [0,1]$ outputting a probability. The optimal predictor is denoted as

$$p^*(x) = \mathbb{P}_{\mathcal{D}}(y = 1 \mid x = x).$$

That is, the true probability that the outcome is true, given all of the information about the individual. We'll assume that this relationship is very complicated, and we don't know the exact distribution $p^*(x)$.

In supervised learning, the first thing we do is fix a hypothesis class $\mathcal{H} \subseteq \{h \colon \mathcal{X} \to [0,1]\}$ (ex. depth 3 decision trees, neural networks, etc.). We'll usually take this set as a bounded set of functions—that is, we won't look at all functions.

The traditional goal is then to find

$$\operatorname*{argmin}_{h \in \mathcal{H}} \mathbb{E}\left[(h(x) - y)^2\right].$$

That is, we want the hypothesis function h that minimizes the expected square loss (we could choose some other loss function as well).

This is the standard approach, but the concern is that we're picking a minimizer across *all* individuals. In particular, if we can't perfectly fit p^* (which is almost always the case), we don't care where these errors lie; usually this could concentrate errors in certain demographic groups, and smaller populations would be marginalized by this process.

24.2 Calibration

To address this issue, we'll look at a new goal, called calibration.

We say that a predictor $p\colon \mathcal{X} \to [0,1]$ is *calibrated* if for all $v \in [0,1]$,

$$\mathbb{P}_{\mathcal{D}}(y = 1 \mid p(x) = v) \approx v.$$

Intuitively, calibration encapsulates the idea that predictors mean what they say. That is, if our predictor predicts a probability of v, then the proportion of their predictions that say v, and actually has an outcome of 1, is v.

We say that a predictor is α-calibrated if for all $v \in [0,1]$,

$$\left|\mathbb{E}\left[(y - v) \cdot \mathbf{1}\{p(x) = v\}\right]\right| \le \alpha.$$

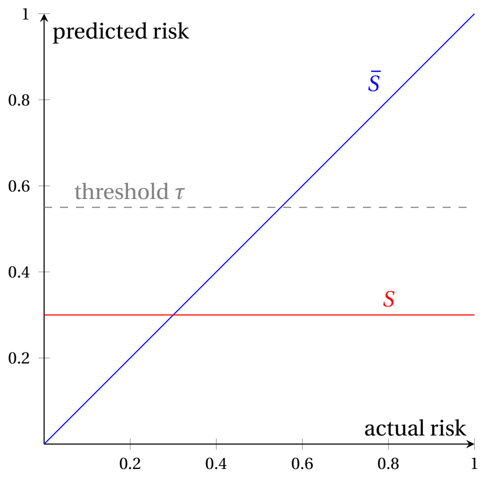
In our earlier example, the predictor is very poorly calibrated for one demographic group. That is, for a group S, we have $\mathbb{E}\left[y \mid p(x) = v \wedge x \in S\right] \gg v$.

The question we need to ask is what calibration is good for, and whether calibration is enough to ensure accuracy.

Consider the following predictor:

$$p^\dagger(x) = \begin{cases} p^*(x) & x \notin S \\ \mu_S & x \in S \end{cases}$$

where $\mu_S \coloneqq \mathbb{E}\left[y \mid x \in S\right]$.



This is called *algorithmic stereotyping*, where the algorithm completely overlooks any variation in the subgroup.

However, we claim that $p^\dagger$ is 0-calibrated over S.

Suppose we look at $\mathbb{E}\left[y \cdot \mathbf{1}\{p^\dagger(x) = \mu_S, x \in S\}\right]$. Since $p^\dagger$ assigns the same value (namely, μ_S) to all members in S, the indicator can be simplified:

$$\begin{aligned}
\mathbb{E}\left[y \cdot \mathbf{1}\{p^\dagger(x) = \mu_S, x \in S\}\right] &= \mathbb{E}\left[y \cdot \mathbf{1}\{x \in S\}\right] \\
&= \mathbb{E}\left[y \mid x \in S\right] \cdot \mathbb{P}(x \in S) \\
&= \mu_S \cdot \mathbb{P}(x \in S)
\end{aligned}$$

This suggests that calibration is a good idea, but is not sufficient on its own; we can get away with making gross generic predictions across the entire group.

In particular, we're missing structure in how we're performing the calibration. If we can identify some structure in these groups, then perhaps calibrating over these structured groups can be beneficial.

24.3 Multicalibration

At the individual level, calibration gives p^*, which is out of reach. Group calibration, as we saw, is too weak. We'll look at something in the middle, which we'll call *multicalibration* [HKR+18].

Suppose we fix a collection of subpopulations $\mathcal{C} \subseteq \{0,1\}^{\mathcal{X}}$. A predictor is $(\mathcal{C}, \alpha)$-multicalibrated if for all $S \in \mathcal{C}$ and all $v \in \operatorname{supp}(\tilde{p})$,

$$\left|\mathbb{E}\left[(y - \tilde{p}(x)) \cdot \mathbf{1}\{\tilde{p}(x) = v, x \in S\}\right]\right| \leq \alpha.$$

Here, we want to take $\mathcal{C}$ to be as rich as possible. If we took $\mathcal{C}$ to be singletons, then we'd recover p^* exactly (which is infeasible), and we saw that if $\mathcal{C}$ is too broad, then we allow for gross generalizations.

As such, we'd like to determine a good $\mathcal{C}$ to use. We'll look at $\mathcal{C}$ which are "computationally identifiable".

In particular, set membership must be efficient; it should be efficient to determine whether $x \in S$ for all $S \in \mathcal{C}$. We must also enforce that auditing is possible; that is, given a predictor p, we must be able to answer whether $\exists S \in C$ that violates multicalibration (and give a witness).

A trivial auditor would go through every single set $S \in \mathcal{C}$ and check individually in linear time. It turns out that we can achieve sublinear time with more sophisticated learning.

24.4 Multicalibration Boosting

We'll now look at an algorithm that constructs a $(\mathcal{C}, \alpha)$-multicalibrated predictor. This is usually called multicalibration boosting, and sometimes called the HKRR algorithm as well.

This algorithm is an iterative algorithm inspired by boosting and gradient descent. We take a predictor, and check whether it is multicalibrated. If it is, then we're done, and otherwise we'll nudge the predictor in such a way that makes it more multicalibrated.

- Initialize $p_0(x) = \frac{1}{2}$ for all $x \in \mathcal{X}$
- Repeat for $t = 0, 1, \ldots$:
  - If there exists an $S \in \mathcal{C}$ and $v \in [0,1]$ such that

    $$\left|\mathbb{E}\left[(y - p_t(x)) \cdot \mathbf{1}\{p_t(x) = v, x \in S\}\right]\right| > \alpha,$$

    then set

    $$p_{t+1}(x) \leftarrow p_t(x) - \eta_t \cdot \mathbf{1}\{p_t(x) = v, x \in S\}.$$

  - Otherwise, return

For correctness, we claim that if MCBOOST terminates, then $\tilde{p}$ is (C, α)-multicalibrated. This follows immediately by the termination condition, since we only halt when there is no violation of multicalibration.

Suppose the algorithm runs for T iterations; how complex can $\tilde{p}$ be? Suppose for all $S \in \mathcal{C}$, set membership is computable in time s. Then, we claim that $\tilde{p}$ is computable in time $O(s + Ts)$.

We'll show that MCBOOST terminates in $T \leq O(\frac{1}{\alpha^2})$ iterations.

We define a potential function $\Phi(p) := \mathbb{E}\left[(p(x) - p^*(x))^2\right]$. This captures the distance from p to the optimal p^*.

It turns out that $\Phi(p_0) = \frac{1}{4} = O(1)$. We'll show that $\Phi(p_t) - \Phi(p_{t+1}) \geq \Omega(\alpha^2)$. This means that we're making some amount of progress at each iteration, and since we only have a constant amount of progress left, then we will always terminate in $O(\frac{1}{\alpha^2})$ steps.

We have

$$\begin{aligned}
\Phi(p_t) - \Phi(p_{t+1}) &= \Phi(p_t) - \mathbb{E}\big[((p_t(x) - y) - \eta_t \cdot \mathbf{1}\{p_t(x) = v, x \in S\})^2\big] \\
&= \Phi(p_t) - \Phi(p_t) - \eta_t^2 \mathbb{E}\big[\mathbf{1}\{p_t(x) = v, x \in S\}\big] + 2\eta_t \cdot \mathbb{E}\big[(p_t(x) - y) \cdot \mathbf{1}\{p_t(x) = v, x \in S\}\big]
\end{aligned}$$

If we choose η_t to be $\pm\alpha$ to move p_t in the correct direction, then the last term is $2\alpha^2$. Further, $\eta_t^2 = \alpha^2$, so the entire quantity is at least $2\alpha^2 - \alpha^2 = \alpha^2$.

It turns out that the auditing process is equivalent to weak agnostic learning.

References

[AAA+09] Noga Alon, Baruch Awerbuch, Yossi Azar, Niv Buchbinder, and Joseph (Seffi) Naor. “The Online Set Cover Problem”. In: *SIAM Journal on Computing* 39.2 (Jan. 2009), pp. 361–370. ISSN: 0097-5397. DOI: 10.1137/060661946. (Visited on 03/14/2023).

[ABK+99] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. “Balanced Allocations”. In: *SIAM Journal on Computing* 29.1 (Jan. 1999), pp. 180–200. ISSN: 0097-5397. DOI: 10.1137/S0097539795288490.

[BBN07] Nikhil Bansal, Niv Buchbinder, and Joseph Naor. “A Primal-Dual Randomized Algorithm for Weighted Paging”. In: *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS’07)*. 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS’07). Oct. 2007, pp. 507–517. DOI: 10.1109/FOCS.2007.43.

[BCL+18] Sébastien Bubeck, Michael B. Cohen, Yin Tat Lee, James R. Lee, and Aleksander Mądry. “K-Server via Multiscale Entropic Regularization”. In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2018. New York, NY, USA: Association for Computing Machinery, June 20, 2018, pp. 3–16. ISBN: 978-1-4503-5559-9. DOI: 10.1145/3188745.3188798.

[BCR22] Sébastien Bubeck, Christian Coester, and Yuval Rabani. *The Randomized k-Server Conjecture Is False!* Nov. 10, 2022. DOI: 10.48550/arXiv.2211.05753. arXiv: arXiv:2211.05753. preprint.

[Bel58] Richard Bellman. “On a routing problem”. en. In: *Quarterly of Applied Mathematics* 16.1 (1958), pp. 87–90. ISSN: 0033-569X, 1552-4485. DOI: 10.1090/qam/102435.

[Bla77] Robert G. Bland. “New Finite Pivoting Rules for the Simplex Method”. In: *Mathematics of Operations Research* 2.2 (1977), pp. 103–107. ISSN: 0364-765X. JSTOR: 3689647. URL: https://www.jstor.org/stable/3689647 (visited on 04/13/2023).

[Blo70] Burton H. Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692.

[BLT12] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. “Strict fibonacci heaps”. In: *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. STOC ’12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 1177–1184. ISBN: 978-1-4503-1245-5. DOI: 10.1145/2213977.2214082.

[BM85] Jon L. Bentley and Catherine C. McGeoch. “Amortized analyses of self-organizing sequential search heuristics”. In: *Communications of the ACM* 28.4 (Apr. 1985), pp. 404–411. ISSN: 0001-0782. DOI: 10.1145/3341.3349.

[BN09] Niv Buchbinder and Joseph (Seffi) Naor. “The Design of Competitive Online Algorithms via a Primal –Dual Approach”. In: *Foundations and Trends® in Theoretical Computer Science* 3.2–3 (May 14, 2009), pp. 93–263. ISSN: 1551-305X, 1551-3068. DOI: 10.1561/0400000024.

[BNW22] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. “Negative-Weight Single-Source Shortest Paths in Near-linear Time”. In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. Oct. 2022, pp. 600–611. DOI: 10.1109/FOCS54457.2022.00063.

[Bro96] Gerth Stølting Brodal. “Worst-case efficient priority queues”. In: *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*. SODA ’96. USA: Society for Industrial and Applied Mathematics, Jan. 1996, pp. 52–58. ISBN: 978-0-89871-366-4. DOI: 10.5555/313852.313883.

[Chv79] V. Chvatal. “A Greedy Heuristic for the Set-Covering Problem”. In: *Mathematics of Operations Research* 4.3 (1979), pp. 233–235. ISSN: 0364-765X. JSTOR: 3689577. URL: https://www.jstor.org/stable/3689577 (visited on 03/14/2023).

[CKL+22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. "Maximum Flow and Minimum-Cost Flow in Almost-Linear Time". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. Oct. 2022, pp. 612–623. DOI: 10.1109/FOCS54457.2022.00064.

[DHI+07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pǎtraşcu. "Dynamic Optimality—Almost". In: *SIAM Journal on Computing* 37.1 (Jan. 2007), pp. 240–251. ISSN: 0097-5397. DOI: 10.1137/S0097539705447347.

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". en. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0945-3245. DOI: 10.1007/BF01386390.

[DJM23] Mingyang Deng, Ce Jin, and Xiao Mao. "Approximating Knapsack and Partition via Dense Subset Sums". In: *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Proceedings. Society for Industrial and Applied Mathematics, Jan. 2023, pp. 2961–2979. DOI: 10.1137/1.9781611977554.ch113. (Visited on 03/21/2023).

[EK72] Jack Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *Journal of the ACM* 19.2 (Apr. 1972), pp. 248–264. ISSN: 0004-5411. DOI: 10.1145/321694.321699.

[Emd75] P. van Emde Boas. "Preserving order in a forest in less than logarithmic time". In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. Oct. 1975, pp. 75–84. DOI: 10.1109/SFCS.1975.26.

[FF56] L. R. Ford and D. R. Fulkerson. "Maximal Flow Through a Network". en. In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. ISSN: 0008-414X, 1496-4279. DOI: 10.4153/CJM-1956-045-5.

[For56] L. R. Ford. *Network Flow Theory*. en. Santa Monica, CA: RAND Corporation, Jan. 1956. URL: https://www.rand.org/pubs/papers/P923.html.

[FT87] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: 10.1145/28869.28874.

[FW90] M. L. Fredman and D. E. Willard. "BLASTING through the information theoretic barrier with FUSION TREES". In: *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*. STOC '90. New York, NY, USA: Association for Computing Machinery, Apr. 1990, pp. 1–7. ISBN: 978-0-89791-361-4. DOI: 10.1145/100216.100217.

[Gol95] Andrew V. Goldberg. "Scaling Algorithms for the Shortest Paths Problem". In: *SIAM Journal on Computing* 24.3 (June 1995), pp. 494–504. ISSN: 0097-5397. DOI: 10.1137/S0097539792231179.

[GR98] Andrew V. Goldberg and Satish Rao. "Beyond the flow decomposition barrier". In: *Journal of the ACM* 45.5 (Sept. 1998), pp. 783–797. ISSN: 0004-5411. DOI: 10.1145/290179.290181.

[GW95] Michel X. Goemans and David P. Williamson. "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming". In: *Journal of the ACM* 42.6 (Nov. 1, 1995), pp. 1115–1145. ISSN: 0004-5411. DOI: 10.1145/227683.227684. (Visited on 03/24/2023).

[Han02] Yijie Han. "Deterministic sorting in O(nlog log n) time and linear space". In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. STOC '02. New York, NY, USA: Association for Computing Machinery, May 2002, pp. 602–608. ISBN: 978-1-58113-495-7. DOI: 10.1145/509907.509993.

[HKR+18] Ursula Hebert-Johnson, Michael Kim, Omer Reingold, and Guy Rothblum. "Multicalibration: Calibration for the (Computationally-Identifiable) Masses". In: *Proceedings of the 35th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, July 3, 2018, pp. 1939–1948. URL: https://proceedings.mlr.press/v80/hebert-johnson18a.html (visited on 04/13/2023).

[HT02] Yijie Han and M. Thorup. “Integer sorting in O(n/spl radic/(log log n)) expected time and linear space”. In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* Nov. 2002, pp. 135–144. DOI: 10.1109/SFCS.2002.1181890.

[Kha79] L. G. Khachiyan. “A polynomial algorithm in linear programming”. In: *Doklady Akademii Nauk SSSR* 244 (1979), pp. 1093–1096. ISSN: 0002-3264.

[Kha80] L. G. Khachiyan. “Polynomial Algorithms in Linear Programming”. In: *USSR Computational Mathematics and Mathematical Physics* 20.1 (Jan. 1, 1980), pp. 53–72. ISSN: 0041-5553. DOI: 10.1016/0041-5553(80)90061-0. (Visited on 04/13/2023).

[KKM+07] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O’Donnell. “Optimal Inapproximability Results for MAX-CUT and Other 2-Variable CSPs?” In: *SIAM Journal on Computing* 37.1 (Jan. 2007), pp. 319–357. ISSN: 0097-5397. DOI: 10.1137/S0097539705447372. (Visited on 03/24/2023).

[KLM89] Richard M Karp, Michael Luby, and Neal Madras. “Monte-Carlo Approximation Algorithms for Enumeration Problems”. In: *Journal of Algorithms* 10.3 (Sept. 1, 1989), pp. 429–448. ISSN: 0196-6774. DOI: 10.1016/0196-6774(89)90038-2. (Visited on 03/21/2023).

[Knu63] Don Knuth. *Notes on “open” addressing*. en. July 1963. URL: https://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf.

[KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. “An Optimal Algorithm for the Distinct Elements Problem”. In: *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’10. New York, NY, USA: Association for Computing Machinery, June 6, 2010, pp. 41–52. ISBN: 978-1-4503-0033-9. DOI: 10.1145/1807085.1807094. (Visited on 03/24/2023).

[KP95] Elias Koutsoupias and Christos H. Papadimitriou. “On the K-Server Conjecture”. In: *Journal of the ACM* 42.5 (Sept. 1, 1995), pp. 971–983. ISSN: 0004-5411. DOI: 10.1145/210118.210128.

[Law79] Eugene L. Lawler. “Fast Approximation Algorithms for Knapsack Problems”. In: *Mathematics of Operations Research* 4.4 (1979), pp. 339–356. ISSN: 0364-765X. JSTOR: 3689221. URL: https://www.jstor.org/stable/3689221 (visited on 03/21/2023).

[LP13] Shachar Lovett and Ely Porat. “A Space Lower Bound for Dynamic Approximate Membership Data Structures”. In: *SIAM Journal on Computing* 42.6 (Jan. 2013), pp. 2182–2196. ISSN: 0097-5397. DOI: 10.1137/120867044.

[LS14] Yin Tat Lee and Aaron Sidford. “Path Finding Methods for Linear Programming: Solving Linear Programs in $\tilde{O}(\sqrt{\text{rank}})$ Iterations and Faster Algorithms for Maximum Flow”. In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. Oct. 2014, pp. 424–433. DOI: 10.1109/FOCS.2014.52.

[MRS01] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. “The Power of Two Random Choices: A Survey of Techniques and Results”. en. In: *Handbook of Randomized Computing*. Ed. by Sanguthevar Rajasekaran, Panos M. Pardalos, John H. Reif, and José Rolim. Vol. 9. Combinatorial Optimization. Boston, MA: Springer US, 2001, pp. 255–312. ISBN: 978-1-4613-4886-3. DOI: 10.1007/978-1-4615-0013-1_9.

[MS91] Lyle A. McGeoch and Daniel D. Sleator. “A Strongly Competitive Randomized Paging Algorithm”. In: *Algorithmica* 6.1 (June 1, 1991), pp. 816–825. ISSN: 1432-0541. DOI: 10.1007/BF01759073.

[Nes83] Yu. E. Nesterov. “A Method of Solving a Convex Programming Problem with Convergence Rate (0(1/k^2))”. In: *Sov. Math., Dokl.* 27 (1983), pp. 372–376. ISSN: 0197-6788.

[NY22] Jelani Nelson and Huacheng Yu. “Optimal Bounds for Approximate Counting”. In: *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS ’22. New York, NY, USA: Association for Computing Machinery, June 13, 2022, pp. 119–127. ISBN: 978-1-4503-9260-0. DOI: 10.1145/3517804.3526225. (Visited on 03/24/2023).

[OPV+19] Ziad Obermeyer, Brian Powers, Christine Vogeli, and Sendhil Mullainathan. “Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations”. In: *Science* 366.6464 (Oct. 25, 2019), pp. 447–453. DOI: 10.1126/science.aax2342. (Visited on 04/13/2023).

[Orl13] James B. Orlin. “Max flows in $O(nm)$ time, or better”. In: *Proceedings of the forty-fifth annual ACM symposium on Theory of Computing*. STOC ’13. New York, NY, USA: Association for Computing Machinery, June 2013, pp. 765–774. ISBN: 978-1-4503-2029-0. DOI: 10.1145/2488608.2488705.

[PPR09] Anna Pagh, Rasmus Pagh, and Milan Ružić. “Linear Probing with Constant Independence”. In: *SIAM Journal on Computing* 39.3 (Sept. 2009), pp. 1107–1120. ISSN: 0097-5397. DOI: 10.1137/070702278.

[PR01] Rasmus Pagh and Flemming Friche Rodler. “Cuckoo Hashing”. en. In: *Algorithms — ESA 2001*. Ed. by Friedhelm Meyer auf der Heide. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 121–133. ISBN: 978-3-540-44676-7. DOI: 10.1007/3-540-44676-1_10.

[PT06] Mihai Pătraşcu and Mikkel Thorup. “Time-space trade-offs for predecessor search”. In: *Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing*. STOC ’06. New York, NY, USA: Association for Computing Machinery, May 2006, pp. 232–240. ISBN: 978-1-59593-134-4. DOI: 10.1145/1132516.1132551.

[PT07] Mihai Pătraşcu and Mikkel Thorup. “Randomization does not help searching predecessors”. In: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. SODA ’07. USA: Society for Industrial and Applied Mathematics, Jan. 2007, pp. 555–564. ISBN: 978-0-89871-624-5.

[PT15] Mihai Pătraşcu and Mikkel Thorup. “On the k-Independence Required by Linear Probing and Minwise Independence”. In: *ACM Transactions on Algorithms* 12.1 (Nov. 2015), 8:1–8:27. ISSN: 1549-6325. DOI: 10.1145/2716317.

[Ram96] Rajeev Raman. “Priority queues: Small, monotone and trans-dichotomous”. en. In: *Algorithms — ESA ’96*. Ed. by Josep Diaz and Maria Serna. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 121–137. ISBN: 978-3-540-70667-0. DOI: 10.1007/3-540-61680-2_51.

[ST85a] Daniel D. Sleator and Robert E. Tarjan. “Amortized efficiency of list update and paging rules”. In: *Communications of the ACM* 28.2 (Feb. 1985), pp. 202–208. ISSN: 0001-0782. DOI: 10.1145/2786.2793.

[ST85b] Daniel Dominic Sleator and Robert Endre Tarjan. “Self-adjusting binary search trees”. In: *Journal of the ACM* 32.3 (July 1985), pp. 652–686. ISSN: 0004-5411. DOI: 10.1145/3828.3835.

[Vöc03] Berthold Vöcking. “How asymmetry helps load balancing”. In: *Journal of the ACM* 50.4 (July 2003), pp. 568–589. ISSN: 0004-5411. DOI: 10.1145/792538.792546.

[Vui78] Jean Vuillemin. “A data structure for manipulating priority queues”. In: *Communications of the ACM* 21.4 (Apr. 1978), pp. 309–315. ISSN: 0001-0782. DOI: 10.1145/359460.359478.

[Wil64] J. W. J. Williams. “Algorithm 232 Heapsort”. In: *Communications of the ACM* 7.6 (June 1964), pp. 347–349. ISSN: 0001-0782. DOI: 10.1145/512274.512284.

[Wil83] Dan E. Willard. “Log-logarithmic worst-case range queries are possible in space $O(N)$”. en. In: *Information Processing Letters* 17.2 (Aug. 1983), pp. 81–84. ISSN: 0020-0190. DOI: 10.1016/0020-0190(83)90075-3.