Dynamic Algebraic Algorithms Lecture Notes
CS8803 Fall'22

Lecturer: Jan van den Brand
TA: Mehrdad Ghadiri

2023/09/04

Contents

I Combinatorial Matrix Multiplication 7

1 Linear Least Squares Regression 9
- 1.1 Dynamic Least Squares . . . 10
  - 1.1.1 Initialization . . . 10
- 1.2 Exercises . . . 12
  - 1.2.1 Dynamic Determinant . . . 12
  - 1.2.2 Dynamic Weighted Least Squares . . . 12
- 1.3 Further Resources . . . 13

2 Dynamic All-Pairs-Reachability 15
- 2.1 Dynamic Path Counting for DAGs . . . 15
  - 2.1.1 Initialization . . . 17
- 2.2 An Algebraic Perspective . . . 18
- 2.3 Dynamic APR for General Graphs . . . 18
  - 2.3.1 Determinants and Cycle-Covers . . . 19
  - 2.3.2 Inverse and Reachability . . . 21
- 2.4 Exercise . . . 22
- 2.5 Further Resources . . . 23

3 Dynamic All-Pairs-Distances 25
- 3.1 Polynomial Matrices . . . 25
- 3.2 Dynamic Polynomial Matrix Inverse . . . 27
- 3.3 Hitting Sets . . . 30
- 3.4 Combining the Tools . . . 31
- 3.5 Exercises . . . 32
  - 3.5.1 Polynomial Matrix Inverse . . . 32
  - 3.5.2 Distances in Weighted Graphs . . . 32
  - 3.5.3 Dynamic Distances in Weighted Graphs . . . 33
- 3.6 Further Resources . . . 33

4 Solving Linear Programs in $nd^2 + n^{1.5}d$ time 35
- 4.1 Linear Programs . . . 35
  - 4.1.1 Example for Duality . . . 37
- 4.2 Framework for solving Linear Programs . . . 38

4.3 Primal-Dual Central Path Method 42
4.3.1 Initial Point 46
4.4 Improvements via Approximate Inverse 47

5 Solving Linear Programs in nd^2 time 53
5.1 Idea for a faster Algorithm 53
5.2 Robust Interior Point Method 54
5.2.1 Feasibility 55
5.2.2 Maintaining small Φ 56
5.3 Vector Maintenance 60
5.3.1 Heavy Hitter 62
5.3.2 Maintaining Approximate Vectors 64

II Fast Matrix Multiplication 71

6 Fast Matrix Multiplication 73
6.0.1 Strassen Matrix Multiplication 73
6.0.2 Rectangular Matrix Multiplication 74

7 Dynamic Matrix Inverse 75
7.1 Faster data structure for few updates 75
7.2 Worst-case update time 77
7.3 Rank and non-invertible matrices 78
7.4 Exercises 80
7.4.1 Matrix Data Structure, Faster Column Updates 80
7.4.2 Matrix Data Structure, Faster element Updates 80

8 Conditional Lower Bounds 83
8.1 Lower Bounds for combinatorial algorithms and data structures 83
8.2 OMv-Problem and Conjecture 85
8.2.1 Conditional Lower Bounds 87
8.3 Exercises 91
8.3.1 Reducing OuMv to OMv 91
8.3.2 Lower Bound for Row and Column Updates 91

III Approximation and Adaptivity 93

9 Sketching and Subspace Embeddings 95
9.1 Subspace embedding 95
9.2 Leverage Scores 98

10 Dynamic Approximate Least Squares **103**
10.0.1 Preliminaries . . . 104
10.1 Main Result . . . 105
10.2 Exercises . . . 108
10.2.1 Finding a large entry . . . 108
10.2.2 Counter example for regression . . . 109
10.2.3 Faster linear program solver via leverage scores . . . 109
10.2.4 Sketching for solving linear systems . . . 110

11 Approximate Distances **111**
11.1 Approximate Distances via Linear Algebra . . . 111
11.2 Exercise . . . 114

12 Handling Adaptive Adversaries via Random Noise **117**
12.1 Exercises . . . 121
12.1.1 Recursive Laplace Noise . . . 121
12.1.2 High Dimensional Laplace Noise . . . 121

Part I

Combinatorial Matrix Multiplication

Chapter 1

Linear Least Squares Regression

In this chapter, we consider the dynamic linear least squares regression problem. The least squares regression for a matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$ and vector $b \in \mathbb{R}^n$ $(n \geq d)$ can be stated as the following.

$$\min_{x \in \mathbb{R}^d} \|Ax - b\|_2$$

In the dynamic version of this problem, we are allowed to insert and delete rows to $\mathbf{A}$ and b. We want to construct a data structure that returns the new optimal solution x after each insertions/deletion.

Example Problem for Dynamic Regression Let us start with a quick example where a problem can be modelled by dynamic regression.

Suppose at time $t_i \in \mathbb{R}$, we receive the position information $p_i \in \mathbb{R}^2$ of an object (e.g. an airplane). Our goal is to predict the location of the object at other times. This can be achieved by equation $c(t) = s + tv$, where $s \in \mathbb{R}^2$ is the starting location, and $v \in \mathbb{R}^2$ is the velocity. Therefore our goal is to estimate/retrieve s and v from the position information we receive. We also want to update our estimate efficiently when we receive new information. The problem can be formulated as the following optimization problem.

$$\min_{s,v \in \mathbb{R}^2} \sum_i \|p_i - (s + t_i v)\|_2^2 = \left\| \mathbf{A} \begin{bmatrix} s_x \\ s_y \\ v_x \\ v_y \end{bmatrix} - b \right\|,$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & t_1 & 0 \\ 0 & 1 & 0 & t_1 \\ 1 & 0 & t_2 & 0 \\ 0 & 1 & 0 & t_2 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}, \text{ and } b = \begin{bmatrix} p_{1,x} \\ p_{1,y} \\ p_{2,x} \\ p_{2,y} \\ \vdots \end{bmatrix}.$$

Adding position information p_i for time t_i is equivalent to adding the rows

$$\begin{bmatrix} 1 & 0 & t_i & 0 \\ 0 & 1 & 0 & t_i \end{bmatrix}$$

to $\mathbf{A}$ and adding the rows

$$\begin{bmatrix} p_{i,x} \\ p_{i,y} \end{bmatrix}$$

to b. Similarly removing position information is equivalent to removing rows from $\mathbf{A}$ and b. Therefore we need to construct a data structure with the following operations:

- Initialization: Given $\mathbf{A} \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$, return $x = \arg\min \|Ax - b\|_2$.
- Add (a, b') where $a \in \mathbb{R}^d, b' \in \mathbb{R}$: Update

  $$\mathbf{A} \leftarrow \begin{bmatrix} \mathbf{A} \\ a^\top \end{bmatrix}, \text{ and } b \leftarrow \begin{bmatrix} b \\ b' \end{bmatrix},$$

  and return $x = \arg\min \|\mathbf{A}x - b\|_2$.
- Remove i: remove i'th row from $\mathbf{A}$ and b, and return $x = \arg\min \|\mathbf{A}x - b\|_2$.

1.1 Dynamic Least Squares

In this section we prove the following Theorem 1.1.1.

Theorem 1.1.1. *There exists a data structure with the following operations:*

- *Initialization: Given $\mathbf{A} \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$, return $x^\star = \arg\min \|Ax - b\|_2$ in $O(nd^2)$ time.*
- *Add (a, b') where $a \in \mathbb{R}^d, b' \in \mathbb{R}$: Update*

  $$\mathbf{A} \leftarrow \begin{bmatrix} \mathbf{A} \\ a^\top \end{bmatrix}, \text{ and } b \leftarrow \begin{bmatrix} b \\ b' \end{bmatrix},$$

  and return $x^\star = \arg\min \|\mathbf{A}x - b\|_2$ in $O(d^2)$ time.
- *Remove i: remove i'th row from $\mathbf{A}$ and b, and return $x^\star = \arg\min \|\mathbf{A}x - b\|_2$ in $O(d^2)$ time.*

1.1.1 Initialization

We assume $\mathbf{A}$ is full column rank and therefore $n \geq d$, and $\mathbf{A}^\top\mathbf{A}$ is full-rank and invertible. Initialization in Theorem 1.1.1 can be achieved by setting $x^\star = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b$.

Claim 1.1.2. $x^\star = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b$ *minimizes* $\|\mathbf{A}x - b\|_2$.

Proof. First note that $\|\mathbf{A}x - b\|_2$ is minimized, if $\mathbf{A}x - b$ is orthogonal to the hyperplane $\mathrm{Im}(\mathbf{A})$. This is because for any possible shift $v \in \mathbb{R}^d$ of x we have

$$\|\mathbf{A}(x+v) - b\|_2 = \|\mathbf{A}x - b\|_2 + 2(\mathbf{A}x - b)^\top \mathbf{A}v + \underbrace{\|\mathbf{A}v\|_2}_{\geq 0}.$$

So the distance can only decrease if $\langle \mathbf{A}x - b, \mathbf{A}v\rangle < 0$ but this is impossible when $\mathbf{A}x - b$ is orthogonal to the hyperplane $\mathrm{Im}(\mathbf{A})$ (i.e. the set of vectors $\{\mathbf{A}v \mid v \in \mathbb{R}^d\}$).

So we only need to show that $(\mathbf{A}x^\star - b) \perp \mathbf{A}v$, for all $v \in \mathbb{R}^d$. We have

$$\begin{aligned}
(\mathbf{A}x^\star - b)^\top \mathbf{A}v &= (\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b - b)^\top \mathbf{A}v \\
&= b^\top \mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{A}v - b^\top \mathbf{A}v \\
&= b^\top \mathbf{A}v - b^\top \mathbf{A}v = 0.
\end{aligned}$$

□

To analyse the running time note that $\mathbf{A}^\top\mathbf{A}$ can be computed in $O(nd^2)$ time. The inverse of $\mathbf{A}^\top\mathbf{A}$ can be computed in $O(d^3)$ time, and and multiplying $\mathbf{A}^\top$ by b can be done in $O(nd)$ time. Therefore the total running time is $O(nd^2)$ to compute $x^* = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}b$.

When we add (a, b') to $\mathbf{A}$ and b, the solution can be updated as follows:

$$x^\star = (\begin{bmatrix}\mathbf{A} \\ a^\top\end{bmatrix}^\top \begin{bmatrix}\mathbf{A} \\ a^\top\end{bmatrix})^{-1}(\begin{bmatrix}\mathbf{A} \\ a^\top\end{bmatrix}^\top \begin{bmatrix} b \\ b'\end{bmatrix}) = (\mathbf{A}^\top\mathbf{A} + aa^\top)^{-1}(\mathbf{A}^\top b + ab').$$

Having stored $\mathbf{A}^\top\mathbf{A}$, we can compute $\mathbf{A}^\top\mathbf{A} + aa^\top$ in $O(d^2)$ time and compute its inverse in time $O(d^3)$. Moreover having stored $\mathbf{A}^\top b$, we can compute $\mathbf{A}^\top b + ab'$ in $O(d)$ time and multiply it by $(\mathbf{A}^\top\mathbf{A} + aa^\top)^{-1}$ in $O(d^2)$ time. Therefore the add update can be done in $O(d^3)$ time. Removing a row can also be done in similar fashion and with the same running time. Using the following lemma, one can see that we do not need to recompute the inverse from scratch in each iteration and update the inverse matrix in $O(d^2)$ time. This lemma has several names in the literature, including, "Sherman-Morrison identity," "Woodbury identity," and "inversion lemma."

Lemma 1.1.3 (Sherman-Morrison 1950 [SM50], Woodbury 1950 [Woo50])**.** *Let $\mathbb{F}$ be a field, $\mathbf{M} \in \mathbb{F}^{n\times n}$, and $u, v \in \mathbb{F}^n$, such that $\mathbf{M}$ and $\mathbf{M} + uv^\top$ are invertible. Then*

$$(\mathbf{M} + uv^\top)^{-1} = \mathbf{M}^{-1} - \frac{\mathbf{M}^{-1}uv^\top\mathbf{M}^{-1}}{1 + v^\top\mathbf{M}^{-1}u}$$

The above lemma shows that the inverse matrix can be updated in $O(d^2)$ time and therefore the total running time of updated goes down to $O(d^2)$.

Theorem 1.1.4 (Theorem 1.1.1)**.** *For $\mathbf{A} \in \mathbb{R}^{n\times d}$ and $b \in \mathbb{R}^n$, a solution $x^\star \in \mathbb{R}^d$ to the dynamic least squares problem $\min_{x\in\mathbb{R}^d} \|\mathbf{A}x - b\|_2$, subject to insertion and deletion of rows can be maintained in $O(d^2)$ time per update. The initialization cost is $O(nd^2)$.*

Later in the course, we prove that $\Omega(d^2)$ time per update is necessary if we want to maintain and *exact* solution (Theorem 8.2.7). However, when allowing for approximation error it is possible to beat $O(n^2)$ time per update. One can maintain an approximate solution $x^\star$ such that $\|Ax^\star - b\|_2 \leq (1+\epsilon)\min_x \|Ax-b\|_2$ in $\widetilde{O}(d/\text{poly}(\epsilon))$ time per update. Both these results (the lower and upper bound) are due to Jiang, Peng and Weinstein (2022) [JPW22]. This will be discussed later, in Chapter 10.

1.2 Exercises

1.2.1 Dynamic Determinant

Let $\mathbb{F}$ be a field. Show there exists a data structure with the following operations:

- $\textsc{Initialize}(\mathbf{A} \in \mathbb{F}^{n\times n})$: Returns $\det(\mathbf{A})$ in $O(n^3)$ field operations.
- $\textsc{Update}(u, v \in \mathbb{F}^n)$: Sets $\mathbf{A} \leftarrow \mathbf{A} + uv^\top$ and return the new $\det(\mathbf{A})$ in $O(n^2)$ field operations.

You are allowed to assume that $\mathbf{A}$ is initially invertible and stays invertible throughout all updates. We can compute the determinant $\det(\mathbf{A})$ and inverse $\mathbf{A}^{-1}$ in $O(n^3)$ field operations. You are also allowed to use the following lemma without proof

Lemma 1.2.1. $\det(\mathbf{A} + uv^\top) = \det(\mathbf{A}) \cdot (1 + v^\top \mathbf{A}^{-1} u)$

1.2.2 Dynamic Weighted Least Squares

Remark. Theorem 1.1.1 supports both insertions and deletions of rows (measurements). Sometimes we do not want to completely remove old measurements and instead want to slowly reduce their impact on the solution over time. For this, consider the following variant of regression:

Let $\lambda \in (0,1)$, $\mathbf{A} \in \mathbb{R}^{n\times d}$, $b \in \mathbb{R}^n$ and consider the regression task $\min_{x\in\mathbb{R}^d} \sum_{i=1}^n \lambda^i (\mathbf{A}x - b)_i^2$. By $\lambda \in (0,1)$ the impact of the bottom coordinates of $(\mathbf{A}x - b)$ to this sum is exponentially decaying.

This weighted least-squares-regression can also be written as follows. Let $\mathbf{D}$ be the $n \times n$ diagonal matrix with $\mathbf{D}_{i,i} = \sqrt{\lambda^i}$. We minimize $\min_{x\in\mathbb{R}^d} \|\mathbf{D}(\mathbf{A}x - b)\|_2^2$.

Problem. Show there exists a data structure with the following operations:

- $\textsc{Initialize}(\mathbf{A} \in \mathbb{R}^{n\times d}, b \in \mathbb{R}^n, \lambda \in (0,1))$: Returns x minimizing $\sum_{i=1}^n \lambda^i (\mathbf{A}x - b)_i^2$ in $O(nd^2)$ time.
- $\textsc{Insert}(a \in \mathbb{R}^d, b' \in \mathbb{R})$: Inserts $a^\top$ and b' at the *top* of $\mathbf{A}$ and b (so $a^\top$ becomes the first row of $\mathbf{A}$ and b' becomes the first entry of b). Then returns x minimizing $\sum_{i=1}^n \lambda^i (\mathbf{A}x - b)_i^2$ in $O(d^2)$ time.

You are allowed to assume that $(\mathbf{A}^\top \mathbf{A})$ is invertible during initialization.

1.3 Further Resources

Dynamic least-squares-regression The same problem is also known as "*streaming least squares*" since the measurements come via some active data stream, see e.g. the lecture notes of ECE6250 by Mark Davenport. In other areas of computer science this is also referred to as "*recursive least squares*" and has connections to the "*Kalman Filter*" from signal processing.

Chapter 2

Dynamic All-Pairs-Reachability

In this chapter we discuss the dynamic reachability problems on graphs. Our goal is to devise a data structure with the following operations.

- Initialize on a directed graph $G = (V, E)$ with n vertices and m edges, and return an $n \times n$ matrix $\mathbf{M}$, where $\mathbf{M}_{u,v} = 1$ if v can be reached from u, and $\mathbf{M}_{u,v} = 0$, otherwise.
- Insert/delete an edge from G and maintain the reachability matrix $\mathbf{M}$ on the new graph.

An application of this problem is in compilers and garbage collectors where the goal is to find unreachable code and variables/data, respectively. In such applications the directed graph changes dynamically according to the running code.

The insertion/deletion of an edge can be maintained naively by running a breadth first search (BFS) from each vertex. This naive approach will have a running time of $O(nm) = O(n^3)$ for any insertion/deletion. In this chapter, our goal is to give an algorithm with $O(n^2)$ running time for each insertion/deletion. This has been elusive by graph theoretic techniques for a long time. King and Sagert (1999) brought a fresh combinatorial perspective to this dynamic reachability problem [KS02]. We will later show that their *combinatorial* algorithm can be interpreted as a special case of another *algebraic* algorithm that has been around since 1950 [SM50], but that connection hadn't been made until much later [San04].

2.1 Dynamic Path Counting for DAGs

We first consider the following path counting problem which also implies reachability.

- Initialize on a directed acyclic graph (DAG) $G = (V, E)$ with n vertices and m edges, and return an $n \times n$ matrix $\mathbf{P}$, where $\mathbf{P}_{s,t} = \#$ of paths from s to t. Note that the number of distinct paths is finite $< n^n$ because a DAG does not contain any cycles. So matrix $\mathbf{P}$ is well-defined. We later discuss in Section 2.1.1 how to efficiently compute this initial matrix $\mathbf{P}$.

- Insert/delete an edge from G and maintain the matrix $\mathbf{P}$ that represents the number of paths from each vertex to another. Since the graph is a DAG, the updates to $\mathbf{P}$ can be computed as follows:
  - Insert (u, v) edge: $\forall s, t \in V, \mathbf{P}_{s,t} \leftarrow \mathbf{P}_{s,t} + \mathbf{P}_{s,u} \cdot \mathbf{P}_{v,t}$. The equivalent matrix operation is $\mathbf{P} \leftarrow \mathbf{P} + \mathbf{P} e_u e_v^\top \mathbf{P}$.
  - Delete (u, v) edge: $\forall s, t \in V, \mathbf{P}_{s,t} \leftarrow \mathbf{P}_{s,t} - \mathbf{P}_{s,u} \cdot \mathbf{P}_{v,t}$. The equivalent matrix operation is $\mathbf{P} \leftarrow \mathbf{P} - \mathbf{P} e_u e_v^\top \mathbf{P}$.

Here the update to $\mathbf{P}_{s,t}$ correctly gives the new number of paths because after an insertion of edge (u, v) the following new st-paths are created: For any path $s \to u$ we can connect it with new edge (u, v) and any path $v \to t$ to obtain a new st-path. Thus we create $\mathbf{P}_{s,u} \cdot \mathbf{P}_{v,t}$ new paths by inserting edge (u, v). This leads to the update rule $\mathbf{P}_{s,t} \leftarrow \mathbf{P}_{s,t} + \mathbf{P}_{s,u} \cdot \mathbf{P}_{v,t}$. For edge deletions we simply replace $+$ (plus) by a $-$ (minus) as we remove the respective paths by deleting edge (u, v).

Note that for $s, t \in V$, if $\mathbf{P}_{s,t} \neq 0$, then s can reach t. The complexity of update for the above problem is $O(n^2)$ operations. However because the number of paths from s to t can be exponential in n (see Figure 2.1), we need $\Omega(n)$ bits to represent the numbers in $\mathbf{P}$. Therefore the total *time* complexity of an update is $\Omega(n^3)$.

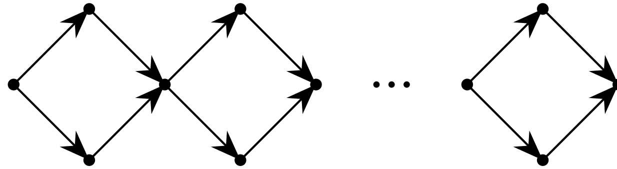


Figure 2.1: There is an exponential number of paths ($2^{n/3}$) from the left-most to the right-most vertex.

To decrease the time complexity that arises from the bit complexity of number of paths, we use the following “fingerprinting” technique.

Lemma 2.1.1. *There exists constant c such that the following holds. Let $k \leq 2^N$ and let p be a uniformly random prime number smaller than or equal to $c \cdot N \log N$. Then $\mathbb{P}[k \mod p = 0] \leq 1/2$.*

Proof. Any $k \leq 2^N$ can have at most N prime factors because 2 is the smallest prime. By the prime number theorem we can lower bound the number of primes of size at most $c \cdot N \log N$ by $\Omega(Nc)$. So the chance tat any one of these primes is a factor of k is at most

$$\frac{\#\text{prime factors of } k}{\#\text{pimes} \leq cN \log N} \leq \frac{N}{\Omega(Nc)} = O(1/c).$$

By picking large enough constant c this is at most $1/2$. □

We simply perform all our computations modulo p, i.e. we count the number of paths mod p. So for any $u, v \in V$ we have with probability at least $1/2$ that $\mathbf{P}_{u,v} \neq 0 \mod p$ whenever u can reach v. This is how we obtain Theorem 2.1.2.

Theorem 2.1.2. *The dynamic APR problem can be solved in $O(n^2 \log n)$ time per update (insertion or deletion) with high probability. More specifically, after each update, the output is correct with probability at least $1 - \frac{1}{n^{100}}$.*

Proof. We run $O(\log n)$ copies of the algorithm. Each copy maintains the matrix $\mathbf{P}$ (which counts the number of paths) modulo a uniformly random prime $p \leq c \cdot n \log n$. Here each copy picks its own independent random prime p. Note that if there is no path from s to t, then $\mathbf{P}_{s,t} = 0$. Alternatively, if there is a path from s to t, then for each copy of the algorithm we have $\mathbf{P}_{s,t} \neq 0$ with probability $\geq 1/2$ by Lemma 2.1.1. Therefore if there is a path from s to t, the probability that for *all* copies $\mathbf{P}_{s,t} = 0$ is bounded by at most $\leq (1/2)^{O(\log n)} \leq n^{-102}$. Taking the union bound over all n^2 pairs of vertices, the output is correct with probability at least $1 - n^{-100}$.

The prime numbers used in the algorithm are bounded by $O(n \log n)$, and we perform all arithmetic operations modulo p (i.e. over $\mathbb{Z}_p$). Thus each arithmetic operation takes only $O(1)$ time in the word-RAM model. Therefore the total running time of an update in the algorithm is $O(n^2 \log n)$ since we have $O(\log n)$ copies and computations for each copy takes $O(n^2)$ time. □

2.1.1 Initialization

We now turn our attention to initializing the matrix $\mathbf{P}$.

Lemma 2.1.3. *Let $G = (V, E)$ be a directed acyclic graph (DAG). The matrix $\mathbf{P}$ with $\mathbf{P}_{s,t} = \#$ of paths from s to t in G can be computed with $O(n^3)$ operations.*

Proof. Let $\mathbf{A}$ be the adjacency matrix of G, i.e., $\mathbf{A}_{u,v} = 1$ if $(u, v) \in E$, and $\mathbf{A}_{u,v} = 0$, otherwise. By induction over k, we can show that $(\mathbf{A}^k)_{s,t} = \#$ of paths from s to t in G with k edges. Therefore since G is a DAG, $\mathbf{A}^k = 0$ for $k \geq n$ since there are no paths with n edges in a DAG (otherwise there exists a cycle). Therefore $\mathbf{P} = \sum_{k=0}^{n} \mathbf{A}^k$. Moreover we have

$$\begin{aligned}\left(\sum_{k=0}^{n} \mathbf{A}^k\right) \cdot (\mathbf{I} - \mathbf{A}) &= \left(\sum_{k=0}^{n} \mathbf{A}^k\right) - \left(\sum_{k=0}^{n} \mathbf{A}^{k+1}\right) = \left(\sum_{k=0}^{n} \mathbf{A}^k\right) - \left(\sum_{k=1}^{n+1} \mathbf{A}^k\right) \\ &= \left(\sum_{k=0}^{n} \mathbf{A}^k\right) - \left(\sum_{k=1}^{n} \mathbf{A}^k\right) \\ &= \mathbf{A}^0 = \mathbf{I}.\end{aligned}$$

Therefore $\mathbf{P} = (\mathbf{I} - \mathbf{A})^{-1}$. Since we can find the inverse of an n-by-n matrix in $O(n^3)$ operations, $\mathbf{P}$ can be computed in $O(n^3)$ operations. □

Remark 2.1.4. *Algorithms for computing a matrix inverse usually perform divisions. Note that $\mathbb{Z}_p$ is a field so it has a well-defined division operation. Thus we can use a matrix inversion algorithm that internally uses division to compute the inverse of $(\mathbf{I} - \mathbf{A}) \in (\mathbb{Z}_p)^{n \times n}$. In particular, this yields the number of paths modulo p.*

2.2 An Algebraic Perspective

In the previous section we described how to maintain all-pairs-reachability on a dynamic graph by counting the number of paths. We gave a combinatorial argument for how the number of paths changes when inserting/deleting and edge (e.g. inserting (u, v) creates $\mathbf{P}_{s,u} \cdot \mathbf{P}_{v,t}$ many new st-paths).

With the identity $\mathbf{P} = (\mathbf{I} - \mathbf{A})^{-1}$ (Lemma 2.1.3), we can view the updates to the matrix $\mathbf{P}$ (as described at the start of Section 2.1) in light of the following algebraic identity that gives the inverse of a matrix that is perturbed by a rank one matrix.

Lemma 1.1.3 (Sherman-Morrison 1950 [SM50], Woodbury 1950 [Woo50]). *Let $\mathbb{F}$ be a field, $\mathbf{M} \in \mathbb{F}^{n\times n}$, and $u, v \in \mathbb{F}^n$, such that $\mathbf{M}$ and $\mathbf{M} + uv^\top$ are invertible. Then*

$$(\mathbf{M} + uv^\top)^{-1} = \mathbf{M}^{-1} - \frac{\mathbf{M}^{-1}uv^\top\mathbf{M}^{-1}}{1 + v^\top\mathbf{M}^{-1}u}$$

Consider adding the edge (u, v) to the graph. This is equivalent to updating the adjacency matrix as $\mathbf{A} + e_u e_v^\top$. Then the update to the matrix $(\mathbf{I} - \mathbf{A})^{-1}$ is as follows:

$$(\mathbf{I} - \mathbf{A} - e_u e_v^\top)^{-1} = (\mathbf{I} - \mathbf{A})^{-1} + \frac{(\mathbf{I} - \mathbf{A})^{-1}e_u e_v^\top(\mathbf{I} - \mathbf{A})^{-1}}{1 - e_v^\top(\mathbf{I} - \mathbf{A})^{-1}e_u}.$$

Therefore the update to $\mathbf{P}$ is given by

$$\mathbf{P} \leftarrow \mathbf{P} + \frac{\mathbf{P}e_u e_v^\top\mathbf{P}}{1 - \mathbf{P}_{v,u}} = \mathbf{P} + \mathbf{P}e_u e_v^\top\mathbf{P},$$

where the last equality holds because $\mathbf{P}_{v,u} = 0$ since otherwise adding (u, v) will create a cycle which would contradict the DAG assumption.

In summary, updating the inverse $(\mathbf{I} - \mathbf{A})^{-1}$ via the Sherman-Morrison identity is equivalent to updating the path-counting matrix $\mathbf{P}$ with the combinatorial idea (i.e. the new number of paths is given by $\mathbf{P}_{s,t} \leftarrow \mathbf{P}_{s,t} + \mathbf{P}_{s,u} \cdot \mathbf{P}_{v,t}$ for all $s, t \in V$).

2.3 Dynamic APR for General Graphs

The approach we presented in Sections 2.1 and 2.2 only works on DAGs. This is due to the fact that number of paths on general graphs is not well-defined. If we count the paths with cycles, then the number of paths could be infinite. If we count the paths without cycles, then the update $\mathbf{P} \leftarrow \mathbf{P} + \mathbf{P}e_u e_v^\top\mathbf{P}$ is not valid anymore since a path from s to u might have an edge in common with a path from v to t. In this case combining these two path results in a path with a cycle.

However, Sankowski (2004) [San04] showed that using uniformly random numbers in $\{0, \ldots, p - 1\}$ instead of the 1s in the adjacency matrix can solve the problem for general direction graphs [San04].

Theorem 2.3.1. *Let p be a prime and let $G = (V, E)$ be a directed graph. Add self-loops (u, u) to E for all $u \in V$. Then define the n-by-n matrix $M \in (\mathbb{Z}_p)^{n \times n}$ as follows*

$$\mathbf{M}_{u,v} = \begin{cases} \text{A uniformly chosen random number } \{0, \ldots, p-1\}, & \text{if } (u, v) \in E, \\ 0, & \text{else.} \end{cases}$$

Then $(M^{-1})_{s,t} = 0$ if there is no path from s to t, and if there is a path from s to t, then $(M^{-1})_{s,t} \neq 0$ with probability at least $1 - 2\frac{n}{p}$.

We can pick $p \approx n^{100}$ to achieve a high-probability result using the above theorem. Together with the Sherman-Morrison identity (Lemma 1.1.3) we directly obtain the following corollary.

Corollary 2.3.2. *There exists a data structure with the following operations:*

- INITIALIZE(G)*: Initializes on a directed n-node graph G in $\widetilde{O}(n^3)$ time.*
- INSERT/DELETE(u, v)*: Inserts (or deletes) an edge (u, v) in $\widetilde{O}(n^2)$ time and returns for all pairs $s, t \in V$ if s can reach t.*

The data structure is randomized and correct with high probability.

Note that the structure of non-zero entries of $\mathbf{M}$ in Theorem 2.3.1 is the same as $(\mathbf{I} - \mathbf{A})$ for adjacency matrix $\mathbf{A}$. Both have non-zero entries on the diagonal and a non-zero entry in position (u, v) if edge (u, v) exists. For DAGs we showed in the previous section that $(\mathbf{I} - \mathbf{A})^{-1}_{s,t} \neq 0$ iff s can reach t. Intuitively, Theorem 2.3.1 states that the same is true for general graphs if we replace the non-zero entries in $(\mathbf{I} - \mathbf{A})$ by random numbers and compute everything modulo some prime p.

Note that in Theorem 2.3.1 the randomness is over the non-zero entries, while the prime p is fixed, whereas previously the path counting argument used fixed non-zero entries (± 1) in $(\mathbf{I} - \mathbf{A})$ but a random prime p.

To prove Theorem 2.3.1, we need the following Lemma 2.3.3. The finite field version of this lemma was proven by Ore (1922) [Ore22], and the general field version was independently proven by Schwartz, Zippel, and DeMillo and Lipton [DL77, Zip79, Sch80].

Lemma 2.3.3 ([Ore22, DL77, Zip79, Sch80])**.** *Let $\mathbb{F}$ be a field and g be a polynomial of degree d over $\mathbb{F}$ with variables $x_1, \ldots, x_m$, i.e., $g \in \mathbb{F}[x_1, \ldots, x_m]$. Let $r_1, \ldots, r_m$ be uniformly random numbers picked from $\mathbb{F}$. Then*

$$\mathbb{P}\left[g(r_1, \ldots, r_m) = 0\right] \leq \frac{d}{|\mathbb{F}|}.$$

2.3.1 Determinants and Cycle-Covers

Before proving Theorem 2.3.1, we first need to show that matrix $\mathbf{M}$ is invertible with some good probability. Note that a matrix is invertible if and only if its determinant is non-zero. Therefore we first discuss the connection between a graph G and the determinant of its adjacency matrix $\mathbf{A}$.

Let $\mathbf{A}$ be the adjacency matrix of some graph $G = (V, E)$. In general, the determinant of a matrix can be written as

$$\det(\mathbf{A}) = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \prod_{i=1}^{n} \mathbf{A}_{i,\sigma(i)}. \tag{2.1}$$

Here S_n is the set of permutations on n elements. In other words, each $\sigma : \{1, ..., n\} \to \{1, ..., n\}$ is a reordering of n numbers. The function $\operatorname{sign} : S_n \to \{+1, -1\}$ is some function that just maps each permutation σ to plus or minus 1, and its exact definition can be ignored for now[1]. For now it suffices to observe that some terms of the sum are subtracted instead of added when $\operatorname{sign}(\sigma) = -1$.

Note that, since $\mathbf{A}$ is an adjacency matrix,

$$\prod_{i=1}^{n} \mathbf{A}_{i,\sigma(i)} = \begin{cases} 1, & \text{if } (i, \sigma(i)) \in E \text{ for all } i, \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

Further, by looking at the cycle decomposition[2] of σ, we have that this product is non-zero if and only if, the cycles in the cycle decomposition of σ are using edges of G. In particular, the product is non-zero if and only if σ describes a cycle cover[3] of G.

So we have $\det(\mathbf{A}) \neq 0$ if there exists exactly one cycle cover in G, since the determinant (2.1) is just a sum of (2.2)-terms. However, if there exists more than one cycle cover in G, then it could happen that the determinant is still 0. This is because $\operatorname{sign}(\sigma) \in \{+1, -1\}$ so two non-zero products (2.2) for different σ could cancel out in the sum (2.1).

The intuition now is that, if we simply replace all the non-zero entries in $\mathbf{A}$ by a random number, then each product (2.2) will be a random number (if σ describes a cycle cover of G), so it is unlikely that the cycle covers with $+1$ sign cancel the cycle covers with -1 sign in (2.1).

Formally, this is stated as the following.

Lemma 2.3.4. *Let $G = (V, E)$ be a directed graph, p be a prime, and $\mathbf{A} \in \mathbb{Z}_p$ where for each edge $(u, v) \in E$, entry $\mathbf{A}_{u,v}$ is an independent uniformly at random chosen number from $\{1, ..., p-1\}$. All other entries of $\mathbf{A}$ are 0.*

Then with probability at least $1 - n/p$, $\det(\mathbf{M}) \neq 0$ if and only if G has a cycle cover.

We already outlined why we would expect this lemma to be true. To formally prove it, we will use the Schwartz-Zippel Lemma 2.3.3. To formally prove Lemma 2.3.4 via Lemma 2.3.3, we must argue that $\det(\mathbf{A})$ is a non-zero polynomial.

Lemma 2.3.5. *Let $G = (V, E)$ be a directed graph with n vertices and m edges. For each edge $(u, v) \in E$ let $x_{(u,v)}$ be some symbol/variable. Define $\mathbf{A}_{u,v} = x_{(u,v)}$ for $(u, v) \in E$ and*

[1] The $\operatorname{sign}(\sigma) = (-1)^{N(\sigma)}$, where $N(\sigma)$ is the number of displaced pairs (also called inversions). For example if 2 is before 1 in the permutation, that counts as one inversion. A permutation with $\operatorname{sign}(\sigma) = +1$ is called an even permutation and it is called an odd permutation, otherwise.

[2] The cycle decomposition of a permutation is defined as the following. Consider a directed a graph with vertices in $\{1, \ldots, n\}$ such that node i is connected to node j, if σ maps i to j. One can observe that this graph consists of only a set of vertex disjoint cycles that cover all vertices. We call this graph the cycle decomposition of σ.

[3] A cycle cover of some graph G is a set of vertex disjoint cycles where each vertex is in exactly one cycle. The cycles must use edges that exist in the graph.

$\mathbf{A}_{u.v} = 0$ *otherwise. So* $\det(\mathbf{A}) =: p(x_{e_1}, ..., x_{e_m})$ *is a polynomial with* m *input variables* $x_{(u,v)}$ *for* $(u, v) \in E$.

Then $\det(\mathbf{A})$ *is a non-zero polynomial if and only if* G *has a cycle cover.*

Proof. We can write

$$p(x_{e_1}, ..., x_{e_m}) := \det(\mathbf{A}) = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \prod_{i=1}^{n} \mathbf{A}_{i,\sigma(i))}$$
$$= \sum_{\substack{\sigma \in S_n \\ \sigma \text{ is cycle cover in } G}} \operatorname{sign}(\sigma) \prod_{i=1}^{n} x_{i,\sigma(i)}$$

So if $p(x_{e_1}, ..., x_{e_m})$ is a non-zero polynomial then there must be a $\sigma \in S_n$ where $\prod_{i=1}^{n} \mathbf{A}_{i,\sigma(i))}$ is a non-zero polynomial. This σ satisfies $(i, \sigma(i)) \in E$ for all i, so this σ represents a cycle cover in G. So there exists a cycle cover in G.

Conversely, if there exists a cycle cover in G then we will argue why this implies p being a non-zero polynomial. We assign value $x_e = 1$ for all edges e used by the cycle cover and $x_{e'} = 0$ for all other edges. Then $\prod_{i=1}^{n} \mathbf{A}_{i,\sigma(i))} = 1$ if and only if σ represents that one cycle cover and thus $p(x_{e_1}, ..., x_{e_m}) = 1$ for that particular input. Since there exists an input for which the polynomial evaluates to non-zero, p is a non-zero polynomial. □

Lemma 2.3.4 now follows directly by applying Schwartz-Zippel Lemma 2.3.3 to the polynomial in Lemma 2.3.5, and observing that the polynomial is of degree n.

2.3.2 Inverse and Reachability

In **??** we saw that the determinant of an adjacency matrix of a graph G is related to the existence of cycle covers in G. We now want to use these insights to prove Theorem 2.3.1, i.e. we want to argue that $\mathbf{M}_{s,t}$ tells us about the existence of an st-path.

We first give some intuition again before writing the formal proof. Let $\mathbf{A}^{\cancel{ij}}$ be the matrix $\mathbf{A}$ where we replaced the ith row and jth column with all 0, except for entry (i, j) which is 1. So

$$\mathbf{A}^{\cancel{ij}}_{s,t} := \begin{cases} \mathbf{A}_{s,t} & \text{if } s \neq i \text{ and } t \neq j \\ 1 & \text{if } s = i \text{ and } t = j \\ 0 & \text{if } s = i \text{ and } t \neq j \\ 0 & \text{if } s \neq i \text{ and } t = j \end{cases}$$

Now let G be some graph where we added self-loops (v, v) to every vertex and let $\mathbf{A}$ be the adjacency matrix. Note that $\mathbf{A}^{\cancel{uv}}$ can be seen as another adjacency matrix of some graph G' that is obtained as follows: Start by $G' = G$, then delete all outgoing edges from u (because we set the uth row to 0 in $\mathbf{A}^{\cancel{uv}}$) and delete all incoming edges to v (because we set the vth column to 0 in $\mathbf{A}^{\cancel{uv}}$), and then insert edge (u, v) (because we set entry (u, v) to 1 in $\mathbf{A}^{\cancel{uv}}$). Now observe that any cycle cover in G' must use edge (u, v) because every vertex must be visited by some cycle and vertex u only has the outgoing edge (u, v). So any cycle cover in G' must contain a cycle $u \to v$ and then some path back to u. This path also exists in G. Conversely, if there exists a vu-path in G, then there exists a cycle cover in G' where we use

the cycle $u \to v$ and then some path back to u, and all remaining vertices are covered by the trivial cycles from the self-loops (v, v) we added initially. In summary, G has a vu-path if and only if G' has a cycle cover.

We can now prove Theorem 2.3.1.

Proof of Theorem 2.3.1. Given graph $G = (V, E)$ with self-loops (v, v) on each edge $v \in V$, let $x_{e_1}, ..., x_{e_m}$ be m variables, one for each $e_i \in E$. Let G' be the graph G after removing all incoming edges into s and all outgoing edges from t, except for the single edge (t, s) which is added to G'. We have that G' has a cycle cover if and only if G has an st-path.

Further, let $\mathbf{A}_{(u,v)} = x_{u,v}$ for all $(u, v) \in E$ and consider the polynomial $\det(\mathbf{A}^{\cancel{t,s}})$. This is a non-zero polynomial if and only if the graph G' has a cycle cover, i.e. graph G has an st-path.

By Cramer's rule, the inverse of a matrix can be written as $(\mathbf{A}^{-1})_{s,t} = \frac{\det(\mathbf{A}^{\cancel{t,s}})}{\det(\mathbf{A})}$, where the denominator is a non-zero polynomial by $\mathbf{G}$ having a cycle cover (every vertex v is covered by (v, v)) and the numerator is a non-zero polynomial if and only if there exists an st-path in G.

We now use Lemma 2.3.3 (Schwartz-Zippel lemma). By plugging independent uniformly at random numbers $\{0, ..., p-1\}$ into each $x_{(u,v)}$, we have with probability at least $1 - 2n/p$ that both $\det(\mathbf{A})$ and $\det(\mathbf{A}^{\cancel{t,s}})$ evaluate to non-zero for this random input, if there exists an st-path. If there is no st-path, then $\det(\mathbf{A}^{\cancel{t,s}})$ is a 0 polynomial and must evaluate to 0.

The matrix $\mathbf{M}$ as described in Theorem 2.3.1 is this matrix $\mathbf{A}$ for the random input, so with probability at least $1 - 2n/p$ we have $(\mathbf{M}^{-1})_{s,t} \neq 0$ if and only if there exists an st-path. □

2.4 Exercise

Show there exists a data structure with the following operations:

- INITIALIZE(G): Initializes on a directed n-node graph G in $\widetilde{O}(n^3)$ time.
- INSERT/DELETE(u, v): Inserts (or deletes) an edge (u, v) in $\widetilde{O}(nk)$ time, where k is the number of updates (insertions or deletion) we had so far.
- QUERY(s, t): For the two given vertices $s, t \in V$, in $\widetilde{O}(k)$ time, returns whether s can reach t. Here k is the number of updates (insertions or deletion) we had so far.

The $\widetilde{O}$ notation ignores polylog factors. E.g. $O(n \log^c n) = \widetilde{O}(n)$ for any constant c. Depending on how you prove the result, you might not have any polylog factors.

Hint: You might want to start by proving the following result first.

- INITIALIZE$(\mathbf{A} \in \mathbb{F}^{n \times n})$: Initializes on an invertible matrix $\mathbf{A}$ in $O(n^3)$ field operations.
- UPDATE$(i, j \in \{1, ..., n\}, f \in \mathbb{F})$: Sets $\mathbf{A} \leftarrow \mathbf{A} + e_i e_j^\top \cdot f$ in $\widetilde{O}(nk)$ operations, where k is the number of updates we had so far. We assume $\mathbf{A}$ stays invertible.
- QUERY$(s, t \in \{1, ..., n\})$: In $\widetilde{O}(k)$ operations, returns $(\mathbf{A}^{-1})_{s,t}$. Here k is the number of updates we had so far.

2.5 Further Resources

Finite Fields There was a request to provide some reading material on finite fields. These lecture notes by David Forney might be helpful.

Applied Algorithms There were questions on what tools are used in practice. I recommend the survey on experimental evaluation of dynamic graph algorithms by Hanauer, Henzinger and Schulz [HHS21].

Some experimental evaluation of (other) reachability algorithms: [FMNZ01], [KZ08]. These evaluations do not consider the matrix approach discussed in Chapter 2. They explicitly write "Regarding future work, it would be interesting to investigate the practicality of the algorithms in King and Sagert 1999 [KS02] and the recent one in Sankowski 2004 [San04]". Even though this was published ∼15 years ago, no one has done a proper experimental evaluation yet.

Chapter 3

Dynamic All-Pairs-Distances

In this chapter we want to construct a data structure that maintains all-pairs-distances. This data structure follows from ideas by Sankowski (2004, 2005) [San04, San05], though the result was not explicitly stated in these references. Abraham, Chechik and Forster[1] (2017) [ACK17] is the first reference stating that this result is possible using Sankowski's techniques.

Theorem 3.0.1. *There exists a data structure with the following operations:*

- INITIALIZE$(G = (V, E))$ *Initialize on an n-node graph and return all-pairs-distances in $\widetilde{O}(n^{3.5})$ time.*
- UPDATE$(v \in V, E' \subset (\{v\} \times V) \cup V \times \{v\})$ *Inserts edges E' to G or removes any such edge if it already exists in G. (All edges in E' are incident to the same vertex v.) Then return all-pairs-distances in $\widetilde{O}(n^{2.5})$ time.*

This type of update is often referred to as "node update" because it allows us to change all edges incident to some node v. In Chapter 2 we looked at "edge updates", i.e. updates that only add/remove a single edge.

Let us compare Theorem 3.0.1 to the trivial solution of just recomputing all-pairs-distances from scratch every time the graph changes, e.g. by running the Floyd-Warshall algorithm. This algorithm would need $O(n^3)$ time per update as opposed to our $\widetilde{O}(n^{2.5})$ time.

3.1 Polynomial Matrices

In Chapter 2, we argued that computing the inverse of some matrix allows us to retrieve information about the reachability in graphs (i.e. Theorem 2.3.1). We now want to extend this to retrieving information about the distances instead of just the reachability. For this purpose, we will use something referred to as "polynomial matrices", which are matrices that have polynomials as entries instead of numbers.

[1] Forster's last name was Krinninger at time of publication.

Polynomials modulo $\mathbf{x^h}$ First some quick basics about polynomials and modulo operations. Given a polynomial $p(x)$, computing modulo x^h for some integer h is equivalent to just truncating all high degree terms (i.e. degree h or higher). For example:

$$x^5 + 2x^3 - 2x^2 + 1 \equiv x^2 \cdot x^3 + 2x^3 - 3x^2 + 1 \equiv x^2 \cdot 0 + 2 \cdot 0 - 3x^2 + 1 \equiv -3x^2 + 1 \mod x^3$$

For some ring R we write $R[x]$ for the ring of polynomials with coefficients from R. For example $\mathbb{Z}[x]$ are polynomials with integer coefficients. We write $p \in R[x]/\langle x^h \rangle$ for the ring of polynomials where every operation is performed modulo x^h. In particular, this means we just truncate all high degree terms after each arithmetic operation. We will often use $\mathbb{Z}[x]/\langle x^h \rangle$, i.e. the ring of polynomials with integer coefficients where we always truncate terms of degree h or higher.

From a computational point of view this modulo operation is useful because adding or multiplying two degree h polynomials takes $O(h)$ and $O(h \log h)$ operations respectively. So by considering polynomials modulo x^h we always have an upper bound on the degree.

Polynomial Matrices When we think of matrices, we usually think of some objects with numbers inside them. However, one can also define matrices with polynomial entries. Matrix multiplication works just as we are used to (i.e."multiplying rows with columns") but now when we multiply two entries of a matrix we must multiply polynomials. Consider the following example:

$$\begin{bmatrix} x^2+1 & x & x^2 \\ x^2+x & x^2+1 & x \\ x & x^2 & 1 \end{bmatrix} \begin{bmatrix} 1 & -x & 0 \\ -x & 1 & -x \\ -x & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1-x^3 & -x^3 & 0 \\ -2x^3 & -x^3+1 & -x^3 \\ -x^3 & 0 & -x^3+1 \end{bmatrix} \tag{3.1}$$

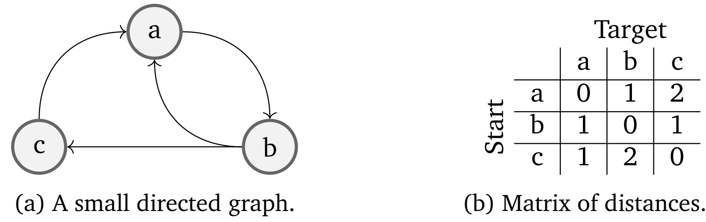
Here the top left entry of the right-hand matrix comes from $(x^2+1)\cdot 1 + x\cdot(-x) + x^2\cdot(-x) = 1 - x^3$. We write $\mathbf{M} \in (\mathbb{Z}[x])^{n\times n}$ to declare $\mathbf{M}$ an $n \times n$ matrix whose entries are polynomials with integer coefficients.

We can also extend the modulo arithmetic to matrices. Here we simply consider each entry of the matrix a polynomial and after each operation, we truncate all high degree terms that occur in the matrix. We write $\mathbf{M} \in (\mathbb{Z}[x]/\langle x^h \rangle)^{n\times n}$ to denote that the entries of $\mathbf{M}$ are polynomials with integer coefficients where we always truncate all terms of degree h or higher.

We note that some polynomial matrices do have an inverse thanks to our modulo arithmetic. For example, note that the right-hand matrix in (3.1) is just the identity when we perform arithmetic modulo x^3. So the two matrices on the left-hand side in (3.1) are the inverse matrix of each other if we consider them to be matrices from $(\mathbb{Z}[x]/\langle x^3 \rangle)^{n\times n}$.

Connections to graphs Consider the graph as in Figure 3.1 which also displays the distances between each pair of vertices. Let $\mathbf{A}$ be the incidence matrix of this graph and compute $(\mathbf{I} - x\mathbf{A})^{-1} \in (\mathbb{Z}[x]/\langle x^3 \rangle)^{n\times n}$. (Note that $(\mathbf{I} - x\mathbf{A})$ is one of the matrices in (3.1), and we just argued in the previous paragraph that the other matrix in (3.1) is the inverse.)

$$(\mathbf{I} - x\mathbf{A})^{-1} = \left(\begin{bmatrix} 1 & -x & 0 \\ -x & 1 & -x \\ -x & 0 & 1 \end{bmatrix}\right)^{-1} = \begin{bmatrix} x^2+1 & x & x^2 \\ x^2+x & x^2+1 & x \\ x & x^2 & 1 \end{bmatrix} = \begin{bmatrix} x^2+x^{\mathbf{0}} & x^{\mathbf{1}} & x^{\mathbf{2}} \\ x^2+x^{\mathbf{1}} & x^2+x^{\mathbf{0}} & x^{\mathbf{1}} \\ x^{\mathbf{1}} & x^{\mathbf{2}} & x^{\mathbf{0}} \end{bmatrix}$$



(a) A small directed graph.

Target / Start

	a	b	c
a	0	1	2
b	1	0	1
c	1	2	0

(b) Matrix of distances.

Figure 3.1: A small directed graph and the matrix of distances between its vertices.

Note that for each entry of this inverse, the smallest exponent we find in each entry corresponds exactly to the distance value in Figure 3.1. So by inverting the polynomial matrix $(\mathbf{I}-x\mathbf{A}) \in (\mathbb{Z}[x]/\langle x^3\rangle)^{n\times n}$ we can compute all-pairs-distances. The following Theorem 3.1.1 states that this property holds for all graphs.

Theorem 3.1.1. *Given directed graph $G = (V, E)$ let $\mathbf{A}$ be its incidence matrix. Then for $(\mathbf{I}-x\mathbf{A}) \in (\mathbb{Z}[x]/\langle x^h\rangle)^{n\times n}$ and any $s, t \in V$ entry $(\mathbf{I}-x\mathbf{A})^{-1}_{s,t} = \sum_{i=0}^{h-1} p_i x^i$ is some polynomial with $p_i = 0$ for $i < \operatorname{dist}(s,t)$ and $p_i \neq 0$ for $i = \operatorname{dist}(s,t)$.*

Proof. We have

$$(\mathbf{I} - x\mathbf{A})^{-1} = \sum_{i=0}^{h} x^i \mathbf{A}^i$$

which can be verified as follows

$$(\mathbf{I} - x\mathbf{A}) \cdot (\sum_{i=0}^{h} x^i \mathbf{A}^i) = \sum_{i=0}^{h} x^i \mathbf{A}^i - \sum_{i=1}^{h+1} x^i \mathbf{A}^i = x^0 \mathbf{A}^0 - x^{h+1}\mathbf{A}^{i+1} = \mathbf{I}$$

where in the last step we used $x^0\mathbf{A}^0 = \mathbf{I}$, and $x^{h+1}\mathbf{A}^{i+1} = 0$ because we perform all computations modulo x^h.

Since $\mathbf{A}$ is the adjacency matrix, $(\mathbf{A}^k)_{s,t} = \#st$-paths using k steps. So the smallest k for which $(\mathbf{A}^k)_{s,t} \neq 0$ is exactly the smallest number of steps required to reach t from s, i.e. the st-distance. □

Theorem 3.1.1 was first used by Sankowski [San05] (2005), though he used the adjoint of matrix $(\mathbf{I} - x\mathbf{A})$ instead of its inverse. This needed a more complicated proof but comes with the benefit that the adjoint of a polynomial matrix is always well-defined, even without modulo x^h. In comparison, our Theorem 3.1.1 uses the inverse instead of the adjoint. This allows for a shorter proof of Theorem 3.1.1 but we must perform all arithmetic modulo x^h otherwise the inverse of the polynomial matrix is not well-defined. To my knowledge, the variant using matrix inverse was first used in [BNS19].

3.2 Dynamic Polynomial Matrix Inverse

As outlined in the previous chapter (Theorem 3.1.1), we can compute all-pairs-distances (up to distance h) of some graph by computing the inverse of a polynomial matrix $(\mathbf{I} - x\mathbf{A})^{-1} \in$

$(\mathbb{Z}[x]/\langle x^h \rangle)^{n\times n}$. So one idea to obtaining the desired data structure from Theorem 3.0.1 would be to maintain this inverse using the Sherman-Morrison identity (Lemma 1.1.3).

$$(\mathbf{M} + uv^\top)^{-1} = \mathbf{M}^{-1} - \frac{\mathbf{M}^{-1}u\, v^\top \mathbf{M}^{-1}}{1 + v^\top \mathbf{M}^{-1} u} \tag{3.2}$$

For our use-case, we would just let $\mathbf{M} = \mathbf{I} - x\mathbf{A}$ for adjacency matrix $\mathbf{A}$. Then when an update occurs to graph G, we are changing one row and one column of $\mathbf{A}$ (because all inserted/deleted edges are incident to the same vertex). Changing one row can be done by letting vector $u = e_i$ (a standard unit vector that is 0 everywhere except for entry i which is 0) so $uv^\top$ is a matrix that is all 0 except for the ith row which is the vector $v^\top$. Likewise we can change an entire column by letting $v = e_i$. In summary, we can construct our distance data structure (Theorem 3.0.1) via the following data structure

Lemma 3.2.1. *For any ring R (e.g. $R = \mathbb{Z}$), there exists a data structure with the following operations:*

- INITIALIZE$(\mathbf{A} \in R^{n\times n}, h \in \mathbb{N})$ *Return* $(\mathbf{I} - x\mathbf{A})^{-1} \in (R[x]/\langle x^h \rangle)^{n\times n}$ *in* $O(hn^3)$ *operations.*
- UPDATE$(u, v \in R^n)$ *Set* $\mathbf{A} \leftarrow \mathbf{A} + uv^\top$*, then return* $(\mathbf{I} - x\mathbf{A})^{-1} \in (R[x]/\langle x^h \rangle)^{n\times n}$ *in* $\widetilde{O}(hn^2)$ *operations.*

Here the complexity is measured in arithmetic operations performed over R.

Note that the update operation here can change a row or a column. To change both a row and a column (e.g. to model the update to the graph as in Theorem 3.1.1), we must call the update function twice.

We remark that Lemma 3.2.1 can only maintain the distance up to h. We will later develop tools to handle distances larger than h in Section 3.3.

The high-level idea for proving Lemma 3.2.1 is to just use the Sherman-Morrison identity (3.2) to maintain the inverse. However, at first it is not clear that the Sherman-Morrison identity still holds when we consider matrices with polynomials as entries. This is because the identity (3.2) attempts a division, i.e. attempts to invert $(1 + v^\top \mathbf{M}^{-1} u)$. This value is a polynomial since $\mathbf{M}$ is a polynomial matrix. The set of polynomials form a ring, not a field, so it is not clear that such inverse polynomial even exists. So we must first show that such a polynomial can be inverted.

Lemma 3.2.2. *For any ring R (e.g. $R = \mathbb{Z}$), let $p(x) \in (R[x]/\langle x^h \rangle)$ be a polynomial with coefficients from R where all arithmetic is performed modulo x^h. If $p(x)$ is of the form $p(x) = 1 - x \cdot q(x)$ for some $q(x) \in (R[x]/\langle x^h \rangle)$, then $p(x)$ is invertible and the inverse can be computed in $\widetilde{O}(h)$ operations over R.*

Proof. The inverse is given by $p(x)^{-1} = (1 - x \cdot q(x))^{-1} = \sum_{i=0}^h x^i q(x)^i$, because

$$(1 - x \cdot q(x)) \cdot (\sum_{i=0}^h x^i q(x)^i) = \sum_{i=0}^h x^i q(x)^i - \sum_{i=1}^{h+1} x^i q(x)^i = x^0 q(x)^0 - x^{h+1} q(x)^{h+1} = 1$$

where the last step used that we perform all arithmetic modulo x^h, i.e. we truncate high degree terms.

We now argue that such inverse can be computed in $O(h \log^2 h)$ operations. For this, note that for any $k \in \mathbb{N}$ we have

$$\prod_{i=0}^{k} (1 + (x \cdot q(x))^{2^i}) = \sum_{i=0}^{2^{k+1}-1} (x \cdot q(x))^i$$

as can be proven by induction over k. So for $k = O(\log h)$ we can compute the inverse. Here $(x \cdot q(x))^{2^i}$ can be computed via repeated squaring, so in total we just need $O(\log h)$ products of polynomials. Multiplying two polynomials of degree h takes $O(h \log h)$ operations via Fast-Fourier transform. □

Using Lemma 3.2.2 we can now prove Lemma 3.2.1 via the Sherman-Morrison identity.

Proof of Lemma 3.2.1. During initialization we compute $(\mathbf{I} - x\mathbf{A})^{-1} = \sum_{i=0}^{h} x^i \mathbf{A}^i$. Since $\mathbf{A}$ is not polynomial, we can compute all the $\mathbf{A}^i$ for $i = 1, ..., h$ in $O(hn^3)$ operations.

We now maintain $(\mathbf{I} - x\mathbf{A})^{-1}$ via the Sherman-Morrison identity.

$$(\mathbf{I} - x(\mathbf{A} + uv^\top))^{-1} = (\mathbf{I} - x\mathbf{A})^{-1} - \frac{(\mathbf{I} - x\mathbf{A})^{-1} uxv^\top (\mathbf{I} - x\mathbf{A})^{-1}}{1 - x(v^\top (\mathbf{I} - x\mathbf{A})^{-1} u)}$$

Here each matrix-vector product takes $O(hn^2)$ operations as we have $n \times n$ matrices, each of degree h. The outer product takes $\widetilde{O}(hn^2)$ operations as we need to multiply n^2 polynomials, each of degree h. The inverse of $(1 - x(v^\top (\mathbf{I} - x\mathbf{A})^{-1} u))$ exists and can be computed in $\widetilde{O}(h)$ operations by Lemma 3.2.2. In total, we need $\widetilde{O}(n^2 h)$ operations. □

We now use Lemma 3.2.1 to obtain a dynamic algorithm for maintaining distances up to h in a graph G. We write $\mathrm{dist}^h(s,t)$ for the "h-bounded distance", that is, $\mathrm{dist}^h(s,t) = \mathrm{dist}(s,t)$ if the distance is at most h and $\mathrm{dist}^h(s,t) = \infty$ if the distance is larger than h. Using the polynomial matrix inverse, we can maintain h-bounded distances in a graph that undergoes edge insertions and deletions.

Corollary 3.2.3. *There exists a data structure with the following operations:*

- INITIALIZE$(G = (V,E), h)$ *Initialize on an n-node graph and return all-pairs-distances in $\widetilde{O}(hn^3)$ time.*
- UPDATE$(v \in V, E' \subset (\{v\} \times V) \cup V \times \{v\})$ *Inserts edges E' to G or removes any such edge if it already exists in G. (All edges in E' are incident to the same vertex v.) Then return h-bounded all-pairs-distances in $\widetilde{O}(hn^2)$ time.*

Proof. We simply maintain the inverse of $(\mathbf{I} - x\mathbf{A})^{-1} \in (\mathbb{Z}[x]/\langle x^{h+1} \rangle)^{n \times n}$ for adjacency matrix $\mathbf{A}$ using Lemma 3.2.1. By Theorem 3.1.1 the inverse tells us the distance if the distance is at most h.

Whenever an update occurs, we change a row or column of $\mathbf{A}$ and obtain the new inverse via Lemma 3.2.1 in $\widetilde{O}(hn^2)$ operations.

Technically, the coefficients used in the inverse could be very large numbers so performing one arithmetic operation might need a lot of time. We can assume each arithmetic operation takes only $O(1)$ time, by using $\mathbb{Z}_p$ instead of $\mathbb{Z}$ for a random prime p (i.e. we use the fingerprinting technique Lemma 2.1.1). Then with probability at most $1/2$ do we return an incorrect distance (which happens if $(\mathbf{A}^{\text{dist}(s,t)})_{s,t}$ is a multiple of p). By running $O(\log n)$ copies in parallel for independent random p, the result is correct with high probability. □

3.3 Hitting Sets

In the previous section, we showed that we can maintain h-bounded distances efficiently for small h. In general, a shortest path might be as long as $O(n)$, so Corollary 3.2.3 might need up to $\widetilde{O}(n^3)$ time to properly maintain the distance in the graph. This is not an improvement over trivially recomputing the distance from scratch via Floyd-Warshall algorithm.

We now show that it suffices to maintain the distance only up to some small h and that we can then extend the h-bounded distances to general distances. The larger h (i.e. the better our bounded distances reflect the true distance) the less time we need to extend the h-bounded distances to general distances.

Theorem 3.3.1. *Let $G = (V, E)$ be an n-node graph and $h \in \mathbb{N}$. Given h-bounded all-pairs-distances of G, we can compute the general all-pairs-distances in $\widetilde{O}(n^3/h)$ time.*

The idea for proving Theorem 3.3.1 is the following: Sample some $\widetilde{O}(n/h)$ random vertices $R \subset V$ and then consider some shortest st-path of length at least h. Then by R being random, we would expect that some of these random vertices are visited by the st-path. Further, one can show that it is unlikely to visit more than h consecutive vertices none of which are in R. That means the st-path can be split into segments $s \to r_1 \to r_2 \to ... \to t$ where each $r_i \in R$ and each segment is of length at most h. We know the length of each segment from the h-bounded distances. So to prove Theorem 3.3.1, we just need to combine the right segments. Before discussing how to do that, we first want to formally prove the claim that the st-path is split into segments of length at most h.

Lemma 3.3.2. *Let $G = (V, E)$ be an n-node graph and $h, p \in N$. Let $R \subset V$ be a random sample of size $\geq pn/h$. Then with probability at least $1 - n^2/2^{\Omega(p)}$ we have the following: For every $s, t \in V$ with $\text{dist}(s, t) \geq h$ there is a shortest st-path that can be split into segments $s \to r_1 \to r_2 \to ... \to t$ where each $r_i \in R$ and each segment has length at most h.*

By picking $p = \Theta(\log n)$ we get that sampling $\widetilde{O}(n/h)$ vertices is enough to have Lemma 3.3.2 hold with high probability.

Proof of Lemma 3.3.2. Let $s, t \in V$ and consider a shortest st-path. The probability that the shortest path does not visit any $r \in R \subset V$ within the first h steps can be bounded by

$$(1 - \frac{|R|}{n})^h = (1 - \frac{|R|}{n})^{(n/|R|)(|R|/n)h} \leq e^{(|R|/n)h} = e^{\Omega(p)} \leq 2^{\Omega(p)}$$

Via union bound over all possible pairs $s, t \in V$, we can say with probability at least $1 - n^2/2^{\Omega(p)}$ that for every $s, t \in V$ there is a shortest st-path that visits some $r \in R$ within the

first h steps. That in turn implies that we can segment any st-path with $\mathrm{dist}(s,t) \geq h$ into $s \to r_1 \to r_2 \to \ldots \to t$. This is because we just argued that we can split $s \to r_1 \to t$ where the first segment $s \to r_1$ is of length at most h, and now we just apply the same argument again recursively on the shortest $r_1 t$-path. This $r_1 t$-path can be split into $r_1 \to r_2 \to t$ with the first segment being of length at most h, and so on. □

Proof of Theorem 3.3.1. For two graphs G, H we write $\mathrm{dist}_G(s,t)$ and $\mathrm{dist}_H(s,t)$ for the distance in the respective graphs. Similarly, we write $\mathrm{dist}^h_G(s,t)$ for the h-bounded distance in G.

We first describe the algorithm for Theorem 3.3.1, then argue why it is correct.

- Sample a random $R \subset V$ of size $O(n/h \log n)$.
- We construct a new graph $H = (V, E')$ on the same vertex set as the original G but with different edges.
- For every $v \in V$, $r \in R$ we add an edge (v,r) and (r,v) to H' with weights $c_{v,r} = \mathrm{dist}^h_G(v,r)$ and $c_{r,v} = \mathrm{dist}^h_G(r,v)$ respectively.
- Run Dijkstra's Algorithm for each $v \in V$ on graph H to compute all-pairs-distances in H.
- Return for every $s,t \in V$ the value $\min\{\mathrm{dist}^h_G(s,t), \mathrm{dist}_H(s,t)\}$.

We claim the returned values are exactly the st-distance in G, i.e.,

$$\mathrm{dist}_G(s,t) = \min\{\mathrm{dist}^h_G(s,t), \mathrm{dist}_H(s,t)\}.$$

Note that $\mathrm{dist}_H(s,t) \geq \mathrm{dist}_G(s,t)$ because every shortest path in H uses edges that correspond to paths in G. So any path we construct find in H has a corresponding path in G.

If $\mathrm{dist}_G(s,t) \leq h$, then

$$\mathrm{dist}_G(s,t) = \mathrm{dist}^h_G(s,t) = \min\{\mathrm{dist}^h_G(s,t), \mathrm{dist}_H(s,t)\}$$

where the last equality comes from $\mathrm{dist}_H(s,t) \geq \mathrm{dist}_G(s,t) = \mathrm{dist}^h_G(s,t)$.

If $\mathrm{dist}_G(s,t) > h$, then there is a shortest st-path in G that can be segmented $s \to r_1 \to r_2 \to \ldots \to t$ for $r_i \in R$ and each segment is of length at most h (see Lemma 3.3.2). Thus each of these segments corresponds to an edge in H whose edge weight is exactly the length of the segment. So there exists an st-path in H of length $\mathrm{dist}_G(s,t)$, so $\mathrm{dist}_H(s,t) \leq \mathrm{dist}_G(s,t)$. Together with $\mathrm{dist}_G(s,t) \leq \mathrm{dist}_H(s,t)$ this implies $\mathrm{dist}_H(s,t) = \mathrm{dist}_G(s,t)$ so $\mathrm{dist}_G(s,t) = \min\{\mathrm{dist}^h_G(s,t), \mathrm{dist}_H(s,t)\}$.

Complexity Constructing graph H takes $O(|V| \cdot |R|) = O((n^2/h) \log n)$ time as we already know the h-bounded distances in G. Running n instances of Dijkstra's Algorithm takes time $O(n \cdot |E'|) = O((n^3/h) \log n)$.

□

3.4 Combining the Tools

We now prove Theorem 3.0.1 by combining Corollary 3.2.3 (which maintains h-bounded distances) with Section 3.3 (which extends the bounded distances to general distances).

Theorem 3.0.1. *There exists a data structure with the following operations:*

- INITIALIZE$(G = (V, E))$ *Initialize on an n-node graph and return all-pairs-distances in $\widetilde{O}(n^{3.5})$ time.*
- UPDATE$(v \in V, E' \subset (\{v\} \times V) \cup V \times \{v\})$ *Inserts edges E' to G or removes any such edge if it already exists in G. (All edges in E' are incident to the same vertex v.) Then return all-pairs-distances in $\widetilde{O}(n^{2.5})$ time.*

Proof. We run Corollary 3.2.3 to maintain h-bounded distances. This data structure initializes in $\widetilde{O}(hn^3)$ time and an update takes $\widetilde{O}(hn^2)$ time. After each update, we obtain the h-bounded all-pairs-distances. We then extend these distances to general distances via Theorem 3.3.1 in $\widetilde{O}(n^3/h)$ time. We return these distances. In summary, an update takes $\widetilde{O}(hn^2 + n^3/h)$ time, which is $\widetilde{O}(n^{2.5})$ when we set $h = \sqrt{n}$. □

3.5 Exercises

3.5.1 Polynomial Matrix Inverse

Let $\mathbb{F}$ be some field and let $\mathbf{A} \in (\mathbb{F}[x]/\langle x^h \rangle)^{n\times n}$ be of the form $\mathbf{A} = \mathbf{N} - x\mathbf{M}$ for $\mathbf{N} \in \mathbb{F}^{n\times n}$, $\det(\mathbf{N}) \neq 0$, $\mathbf{M} \in (\mathbb{F}[x]/\langle x^h \rangle)^{n\times n}$. We remark that here the entries of $\mathbf{M}$ can be of degree as large as $h-1$ and the entries of $\mathbf{N}$ are of degree 0. In particular, let $p(x) = \sum_{k=0}^{h-1} p_k x^k$ be the polynomial of some entry $\mathbf{A}_{i,j}$, then $p_0 = \mathbf{N}_{i,j}$ and $x\mathbf{M}_{i,j} = \sum_{k=1}^{h-1} p_k x^k$.

Problem 1 Show we can compute $\mathbf{A}^{-1} \in (\mathbb{F}[x]/\langle x^h \rangle)^{n\times n}$ in $\widetilde{O}(hn^3)$ operations.

Hint: It might help to first consider the case $\mathbf{N} = \mathbf{I}$, i.e. $\mathbf{A} = \mathbf{I} - x\mathbf{M}$. For inspiration, you might want to check the lecture notes how to efficiently invert a polynomial of the form $p(x) = 1 - q(x) \in \mathbb{Z}/\langle x^h \rangle$.

3.5.2 Distances in Weighted Graphs

Consider a directed weighted graph $G = (V, E, c)$ on n vertices where $c_{u,v} \in \mathbb{N}_{\geq 1}$ is the cost/weight of edge $(u, v) \in E$. Define $\mathbf{M} \in (\mathbb{Z}[x]/\langle x^h \rangle)^{n\times n}$ where $\mathbf{M}_{v,v} = 1$ for all $v \in V$ and $\mathbf{M}_{u,v} = -x^{c(u,v)}$ for all $(u, v) \in E$. All other entries of $\mathbf{M}$ are 0. We observe that for an unweighted graph (i.e. $c(u, v) = 1$ for all $(u, v) \in E$) and adjacency matrix $\mathbf{A}$ we have $\mathbf{M} = \mathbf{I} - x\mathbf{A}$.

Problem 2 For any $s, t \in V$ let $p(x) \in \mathbb{Z}[x]/\langle x^h \rangle$ be the polynomial in entry $\mathbf{M}^{-1}_{s,t}$. Show that $p(x) = \sum_{k=0}^{h-1} x^k \cdot \#$number of st-paths of length exactly k. In particular, the smallest k, for which x^k has a non-zero coefficient in $\mathbf{M}^{-1}_{s,t}$, is the st-distance (if the distance is less than h).

3.5.3 Dynamic Distances in Weighted Graphs

Construct a data structure with the following operations:

- INITIALIZE$(G = (V, E, c))$ We initialize on a directed graph with integer edge-weights in $\{1, ..., W\}$. Then return all-pairs-distances in $\widetilde{O}(\sqrt{W} n^{3.5})$ time.
- UPDATE$(v \in V, E' \subset (\{v\} \times V) \cup (V \times \{v\})$, edge weights $c'_e \in \{1, ..., W\}$ for all $e \in E'))$ We insert (or delete) the edges $e \in E'$ with edge weight c'_e (these edges are all incident to v). Then we return the new all-pairs-distances in $\widetilde{O}(\sqrt{W} n^{2.5})$ time.

For simplicity you may assume that adding and multiplying two elements from $\mathbb{Z}$ takes $O(1)$ time, even if they are large numbers. (This assumption could be removed via the Fingerprinting technique (Lemma 2.1.1), but that is not required for this exercise.) You are also allowed to use Problem 1 and Problem 2 even if you have not solved them.

3.6 Further Resources

Maintaining the Paths The algorithm from Theorem 3.0.1 maintains the distances between every pair of vertices in $\widetilde{O}(n^{2.5})$ time per update, but not the shortest paths. Section 4.2 in [ACK17] shows how to maintain the paths in the same $\widetilde{O}(n^{2.5})$ update time. That algorithm is not algebraic and purely graph theoretic.

Chapter 4

Solving Linear Programs in $nd^2 + n^{1.5}d$ time

Jan: I start with a very quick recap of linear programs in case some students haven't seen them before. Actual algorithms for solving linear programs start in Section 4.2.

4.1 Linear Programs

Consider the following problem: We own a bakery and currently have 6 units of flour and 6 unit of egg available to us. We have a recipe for "Bread A" which requires 2 units of flour and 1 unit of egg. This bread sells for \$3. We could also bake "Bread B" which needs 1 unit of flour and 2 units of egg, and sells for \$2. The task is to maximize our profit.

We can model the task via the following problem

$$\begin{aligned} \max 3a + 2b & \text{ subject to} \\ 2a + b \leq 6 & \text{ (we can use at most 6 flour)} \\ a + 2b \leq 6 & \text{ (we can use at most 6 eggs)} \\ a \geq 0, b \geq 0 & \text{ (we can not bake a negative number of bread)} \end{aligned} \tag{4.1}$$

where variables a and b are the number of "Bread A" and "Bread B" we bake.

The first intuitive idea would be to bake the largest possible number of "Bread A" since it has the highest price. We can bake 3 "Bread A" (as that needs $3 \cdot 2 = 6$ units of flour) which yields $\$3 \cdot 3 = \$9$ profit. However, it turns out that baking 2 "Bread A" and 2 "Bread B" is the better choice. This would need 6 units of flour and 6 unit of egg, and sells for $\$10$.

Geometric Interpretation The problem has the following geometric interpretation. We can interpret possible solutions (a, b) as points in a 2-dimensional plane. Each constraint can be interpreted as a "half-space", i.e. a line for which point (a, b) must be on a certain side of the line. For example, $a \geq 0$ means we only consider points (a, b) that are on the right half (see Figure 4.1). All these half-spaces together form a polytope (each side is one

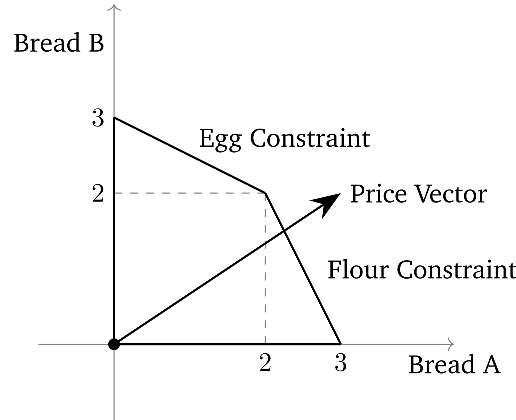


Figure 4.1: The geometric interpretation of the linear program associated with the bread problem (4.1).

of the constraints). The problem (4.1) can thus be modelled as finding the point inside the polytope that maximizes $(3,2)^\top(a,b)$, i.e. the point furthest in direction $(3,2)$.

Matrix formulation We can also write (4.1) as follows:

$$\max \begin{bmatrix} 3 \\ 2 \end{bmatrix}^\top \begin{bmatrix} a \\ b \end{bmatrix} \text{ subject to}$$
$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \leq \begin{bmatrix} 6 \\ 6 \\ 0 \\ 0 \end{bmatrix}$$

Note that here the inequality holds for each coordinate. In particular, we have a problem of the following form

$$\begin{aligned} &\max \vec{b}^\top \vec{y} \text{ subject to} \\ &\mathbf{A}\vec{y} \leq \vec{c} \end{aligned} \tag{4.2}$$

where $\mathbf{A}\vec{y} \leq \vec{c}$ is defined to mean $(\mathbf{A}\vec{y})_i \leq \vec{c}_i$ for all i. Problems of form (4.2) are called "linear programs". The example showed that it is a common problem when it comes to creating production plans. Many optimization problems can be written as linear programs: shortest paths, matching, optimal transport, transshipment, linear ℓ_1-regression, Markov decision processes to name just a few. In this class, we will discuss efficient algorithm for solving linear programs.

4.1.1 Example for Duality

In the previous section we motivated problems of the following form:

$$\begin{aligned} \max b^\top y & \text{ subject to} \\ \mathbf{A}y \leq c & \end{aligned} \tag{4.3}$$

This problem is closely related to the following optimization problem for the same matrix $\mathbf{A}$ and vectors b, c:

$$\begin{aligned} \min c^\top x & \text{ subject to} \\ \mathbf{A}^\top x = b & \\ x \geq 0 & \end{aligned} \tag{4.4}$$

This, too, is a linear program. We say these two problems are dual to each other (one problem is the dual of the other). To get some intuition why these two problem should be related (and are in-fact the same problem) consider the following example:

Shortest paths in form (4.3): Given a graph $G = (V, E, c)$ with edge weights $c_{(u,v)\in E}$ and two vertices $s, t \in V$, we want to compute the st-distance. The following is a physical experiment that reflects computing the st-distance: Assume the graph is given via a net, i.e. each edge (u, v) is a string/cord of length $c_{u,v}$ and a vertex is just the point where the strings are knotted together. If we take knot s and knot t and stretch them apart as far as possible, then eventually we can not stretch them any further because the strings are under tension (see **??** **Jan: todo figure**). The strings under tension are exactly the edges on the shortest paths.

Stretching the knots apart can be mathematically modelled as assigning each knot $v \in V$ a number y_v which is the place of the knot along some number line. Then stretching the knots can be modelled via:

$$\begin{aligned} \max y_t - y_s & \text{ (stretch } s \text{ and } t \text{ as far apart as possible)} \\ y_v - y_u \leq c_{u,v} \ \forall (u,v) \in E & \left(\begin{array}{c} \text{we can not stretch to vertices further apart than the length} \\ \text{of the string. Otherwise we would rip the string apart.} \end{array} \right) \end{aligned}$$

This can be brought into the matrix shape as follows: We consider $b \in \mathbb{R}^V$ with $b_v = 0$ for all $v \in V, v \neq s, t$, and $b_s = -1$, $b_t = 1$. The vector $c \in \mathbb{R}^E$ is just the vector representing the length of an edge. Matrix $\mathbf{A} \in \mathbb{R}^{E \times V}$ is a so called edge-vertex-incidence matrix, that is, $\mathbf{A}_{(u,v),u} = -1$, $\mathbf{A}_{(u,v),v} = 1$, for $(u, v \in E)$ and all other entries 0. Then above linear program is the same as:

$$\begin{aligned} \max b^\top y \\ \mathbf{A}y \leq c \end{aligned} \tag{4.5}$$

The dual problem would be

$$\begin{aligned} \max c^\top x \\ \mathbf{A}^\top x = b \\ x \geq 0 \end{aligned} \tag{4.6}$$

We now argue why these two problems are actually the same, i.e. they both compute the st-distance.

Shortest paths in form (4.6): For (4.5) we just argued that it is the interpretation of the st-distance being the furthest we can stretch knot (=vertex) s and t apart without ripping any strings (=edges) apart. For the dual problem (4.6) note that we can write it as follows:

$$\max \sum_{e \in E} c_e x_e$$
$$\underbrace{\sum_{(u,v)\in E} x_{(u,v)}}_{\text{flow incoming to } v} - \underbrace{\sum_{(v,u)\in E} x_{(v,u)}}_{\text{flow outgoing of } v} = b_v \ \forall v \in V x \geq 0$$

Here $x \in \mathbb{R}^E$ is defined on the edge set, so it can be seen as a flow. Further, we said $b_v = 0$ for all $v \neq s, t$ and $b_s = -1$, $b_t = 1$. So above linear program says that we have a flow that routes 1 unit of flow from s to t. Since there are no capacities on the edges (only costs) the flow of minimal cost is just the flow that goes along the shortest path (i.e. the path that allows us to go from s to t with the minimum cost).

In summary, an optimal solution $x \in \mathbb{R}^E$ to (4.6) is just the vector with $x_e = 1$ if e is used on the shortest path and $x_e = 0$ otherwise. On the other hand, an optimal solution $y \in \mathbb{R}^V$ to (4.5) would be $y_v = sv$-distance. So both problems compute the shortest paths, it's just that one considers the problem of computing the distance of vertices ($y \in \mathbb{R}^V$) while the other consider the problem of computing the path itself ($x \in \mathbb{R}^E$).

4.2 Framework for solving Linear Programs

In the previous section we introduced the concept of linear programs and their dual. We now want to prove some general properties of the dual and outline how it can be used to find the optimal solution.

For the following theorem, note that $\mathbf{A}y \leq c$ can also be written as $\mathbf{A}y + s = c$, $s \geq 0$. Here the vector $s = c - \mathbf{A}y$ is called the "slack vector" and it tells us how much space/slack is left for the inequality. E.g. if $s_i = 0$ then that means $(\mathbf{A}y)_i = c_i$ instead of just $(\mathbf{A}y)_i \leq c_i$.

Theorem 4.2.1 (Weak duality)**.** *Given any $x, s \in \mathbb{R}^n, y \in \mathbb{R}^d$ such that $\mathbf{A}^\top x = b, x \geq 0$, $\mathbf{A}y + s = c, s \geq 0$ we have $c^\top x \geq b^\top y$. In particular, $\min_{\mathbf{A}^\top x = b, x \geq 0} c^\top x \geq \max_{\mathbf{A}y \leq c} b^\top y$.*

In related literature, this property is called "weak duality". We will later argue that actually $\min_{\mathbf{A}^\top x = b, x \geq 0} c^\top x = \max_{\mathbf{A}y \leq c} b^\top y$ holds (equality "=" instead of inequality "≥"), which is referred to as "strong duality". Strong duality formalizes the claim from the previous section, where we said that a linear program and its dual are solving the same problem, just viewed from a different perspective.

Proof of Theorem 4.2.1.

$$0 \leq s^\top x = (c - \mathbf{A}y)^\top x = c^\top x - y^\top \mathbf{A}^\top x = c^\top x - y^\top b \tag{4.7}$$

The first inequality holds because $x \geq 0, s \geq 0$ so the inner product of the two only adds non-negative terms. In the next step we used $s = c - \mathbf{A}y$ and the last step follows from $\mathbf{A}^\top x = b$. This then gives us

$$y^\top b \leq c^\top x.$$

Note that this holds true for every x, y with $\mathbf{A}^\top x = b, x \geq 0$ and $\mathbf{A}y \leq c$. So in particular, it holds for the optimal solutions of the linear programs:

$$\min_{\mathbf{A}^\top x = b, x \geq 0} c^\top x \geq \max_{\mathbf{A}y \leq c} b^\top y.$$

□

Weak duality is also useful to improve solutions. Assume we have some solution $\mathbf{A}y \leq c$ (not necessarily optimal) and we want to know how close we are to being optimal. Then we can just find some $x \geq 0$ with $\mathbf{A}^\top x = b$ and compute $s^\top x$. More accurately, we can use the following lemma

Lemma 4.2.2. *Given any $x, s \in \mathbb{R}^n, y \in \mathbb{R}^d$ such that $\mathbf{A}^\top x = b, x \geq 0$, $\mathbf{A}y + s = c, s \geq 0$ we have*

$$b^\top y \geq (\min_{\mathbf{A}y \leq c} b^\top y) - s^\top x$$

and

$$c^\top x \leq (\min_{\mathbf{A}^\top x = b, x \geq 0} c^\top x) + s^\top x.$$

So the value $s^\top x$ gives us an upper bound on how far away our solutions are to being optimal.

Proof. In the proof of Theorem 4.2.1 we proved

$$s^\top x = c^\top x - y^\top b$$

which leads to

$$c^\top x = y^\top b + s^\top x \leq (\max_{\mathbf{A}y \leq c} y^\top b) + s^\top x \leq (\min_{\mathbf{A}^\top x = b, x \geq 0} y^\top b) + s^\top x.$$

The 2nd step comes from the optimal solution being larger than some y since it's a maximization problem. The last step uses Theorem 4.2.1. In the same way we can bound

$$y^\top b = c^\top x - s^\top x \geq (\min_{\mathbf{A}^\top x = b, x \geq 0} y^\top b) - s^\top x \geq (\min_{\mathbf{A}y \leq c} y^\top b) - s^\top x.$$

□

Besides giving a bound on how good a solution is, Lemma 4.2.2 also implies an algorithmic framework for solving linear programs.

Algorithmic Framework

- Find some $t \in \mathbb{R}_{>0}$ and $x \in \mathbb{R}^n_{>0}$ with $\mathbf{A}^\top x = b$, and some $y \in \mathbb{R}^d, s \in \mathbb{R}^n_{>0}$ with $\mathbf{A}y + s = c$, such that $x_i \cdot s_i \approx t$ for all $i = 1, ..., n$.
- while $t > \epsilon/n$
  - Set $t \leftarrow t \cdot (1 - \alpha)$
  - Set $x \leftarrow x + \delta_x$, $y \leftarrow y + \delta_y$, $s + \delta_s$ such that again $x \geq 0$, $\mathbf{A}^\top x = b$, $s \geq 0$, $\mathbf{A}y + s = c$, $x_i s_i \approx t\ \forall i$.
    That is, we move our solution x and y a bit such that $x_i s_i \approx t$ holds true again after we decreased t a bit.

At the end of this algorithm we have by Lemma 4.2.2 that

$$b^\top y \geq (\max_{\mathbf{A}y \leq c} b^\top y) - s^\top x = (\max_{\mathbf{A}y \leq c} b^\top y) - \sum_{i=1}^n s_i x_i \approx (\max_{\mathbf{A}y \leq c} b^\top y) - n \cdot t = (\max_{\mathbf{A}y \leq c} b^\top y) - \epsilon$$

because $x_i s_i \approx t$ and our algorithm stops when $t \leq \epsilon/n$. Thus at the end of the algorithm, we have a very accurate solution by picking ϵ small enough.

This framework is not yet a complete algorithm. There are several questions that must be answered first

1. How do we find the initial x, y, s, t that satisfy all the requirements? (That is, how to implement the very first step of the algorithm.)

2. How large can we choose α? Note that the number of iterations of the algorithm will be $O(1/\alpha\ \log(t^{start} n/\epsilon))$ so larger α will result in a faster algorithm. ($t^{start} \cdot (1-\alpha)^k \leq \epsilon/n$ for $k = O(1/\alpha\ \log(t^{start} n/\epsilon))$. Hence the number of iterations.)

3. How to pick and compute the movement of x, y, s, i.e. the vectors $\delta_x, \delta_y, \delta_s$?

4. How to formalize $x_i s_i \approx t$ for all i? What does it mean for these values to be close?

Depending on how these questions are answered, the algorithm will have a different complexity.

Before answering these questions, we want to give a geometric interpretation of what this algorithmic framework does. For this, consider the geometric interpretation of $\mathbf{A}y \leq c$. The set of y that satisfy the constraints $\mathbf{A}y \leq c$ forms a polytope. The optimal solution that maximizes $b^\top y$ is usually a corner of the polytope. Further, at the end of the algorithm, we have very tiny $x_i \cdot s_i\ \forall i$, so the solution y is almost optimal, i.e. close to the optimal corner. If we have some y with $x_i y_i = t$ for some large t, then it is far from optimal but still somewhere inside the polytope since it satisfies $\mathbf{A}y \leq c$. In particular, we can define some curve $c(t) = \{y \mid \mathbf{A}y = c, \exists x \geq 0, \mathbf{A}^\top x = b, x_i s_i = t \forall i\}$ (see Figure 4.2). That is, for any given $t > 0$, $c(t)$ is the point y that has some corresponding x where $x_i s_i = t$ for all i.

Note that during our algorithm we only have $x_i s_i \approx t$, so we are not exactly on this curve, but we are close by. What our algorithm does is to initially find some point close to this curve, and then repeatedly follow along the curve (by decreasing t) until it is very close to the optimal solution (when $t \leq \epsilon/n$).

This algorithmic framework is often referred to as “central path method” because our solution follows the curve (called the central path) from Figure 4.2. Central path methods

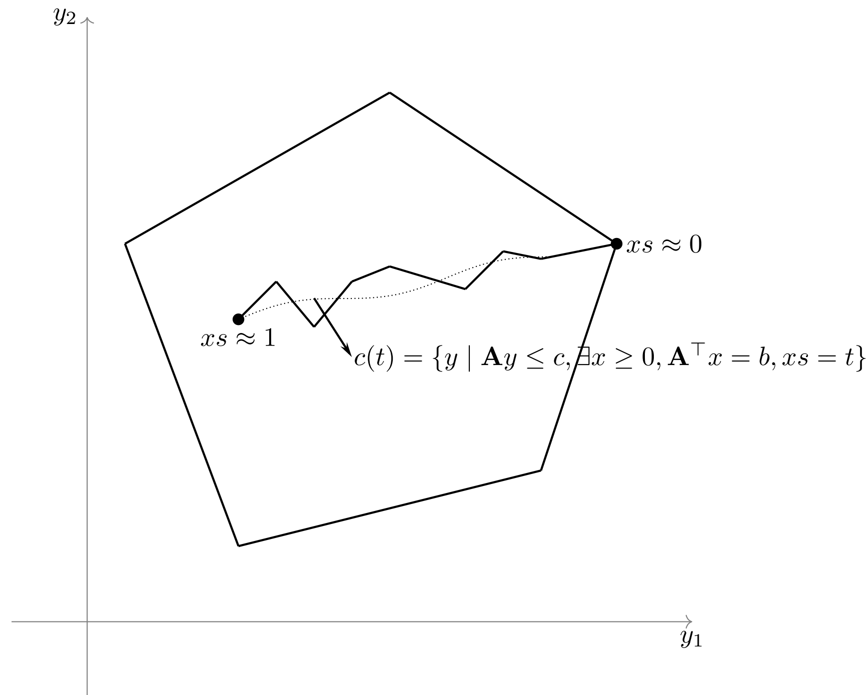


Figure 4.2: The geometric interpretation of the central path for linear programs. The central path is the dotted curve. Our algorithm finds a point close to this curve (i.e. $xs \approx t$ for some t) and then follows this curve towards the optimal solution. Following the curve is facilitated by maintaining $xs \approx t$ while t decreases over time.

are a subclass of algorithms called "interior point methods" because throughout the algorithm, we maintain a valid solution y, i.e. the point y is always in the *interior* of the polytope (Figure 4.2).

Questions During the lecture the question was raised why we need $x_i s_i \approx t$ for all i. Wouldn't $x_i s_i \leq t$ be enough to argue $x^\top s \leq \epsilon$ at the end of the algorithm?
Answer: Yes, generally it is possible to just use an inequality, but the algorithm will end up needing more iterations. For an intuition why, consider the following: If $x_i s_i \ll t$ is very small for some i, then x_i or s_i are very small. In that case we can not take very long steps, i.e. $x \leftarrow x + \delta_x$ and $s \leftarrow s + \delta_s$ need small δ_x, δ_s otherwise x or s could become negative (but we need $x \geq 0, s \geq 0$). Geometrically, what would happen is that we are not close to the curve in Figure 4.2, i.e. we might not be neatly in the center of the polytope but very close to some boundary. So by taking long steps we might step outside the polytope (which corresponds to $s_i < 0$ for some i).

4.3 Primal-Dual Central Path Method

Last lecture we defined the following algorithmic framework. This is often referred to as "central path method" because our solution x, y, s follow the curve (called the central path) from Figure 4.2. This framework fits into the class of "interior point methods" because throughout the algorithm, we maintain a valid solution y, i.e. the point y is always in the *interior* of the polytope (Figure 4.2).

Algorithmic Framework

- Find some $t \in \mathbb{R}_{>0}$ and $x \in \mathbb{R}^n_{>0}$ with $\mathbf{A}^\top x = b$, and some $y \in \mathbb{R}^d, s \in \mathbb{R}^n_{>0}$ with $\mathbf{A}y + s = c$, such that $x_i \cdot s_i \approx t$ for all $i = 1, ..., n$.
- while $t > \epsilon/n$
  - Set $t \leftarrow t \cdot (1 - \alpha)$
  - Set $x \leftarrow x + \delta_x$, $y \leftarrow y + \delta_y$, $s \leftarrow s + \delta_s$ such that again $x \geq 0$, $\mathbf{A}^\top x = b$, $s \geq 0$, $\mathbf{A}y + s = c$, $x_i s_i \approx t\ \forall i$.
    That is, we move our solution x and y a bit such that $x_i s_i \approx t$ holds true again after we decreased t a bit.

This framework is not yet a complete algorithm. There are several questions that must be answered first

1. How do we find the initial x, y, s, t that satisfy all the requirements? (That is, how to implement the very first step of the algorithm.)

2. How large can we choose α? Note that the number of iterations of the algorithm will be $O(1/\alpha\ \log(t^{start} n/\epsilon))$ so larger α will result in a faster algorithm. ($t^{start} \cdot (1-\alpha)^k \leq \epsilon/n$ for $k = O(1/\alpha\ \log(t^{start} n/\epsilon))$. Hence the number of iterations.)

3. How to pick and compute the movement of x, y, s, i.e. the vectors $\delta_x, \delta_y, \delta_s$?

4. How to formalize $x_i s_i \approx t$ for all i? What does it mean for these values to be close?

Depending on how these questions are answered, the algorithm will have a different complexity.

Notation Before answering above questions, let us quickly define some notation. For any two vectors u, v we define uv as the vector resulting from multiplying each entry, i.e. $(uv)_i = u_i v_i$. Note that for a diagonal matrix $\mathbf{U}$ with $\mathbf{U}_{i,i} = u_i$, we have $uv = \mathbf{U}v$.
We also define for a scalar t that $v + t$ is the vector where we add t to each entry of v, i.e. $(u + t)_i = u_i + t$.

How to formalize $x_i s_i \approx t$ for all i? Today, we will use the following definition for $x_i s_i \approx t$:

$$\sqrt{\sum_{i=1}^{n} \left(\frac{x_i s_i - t}{t} \right)^2} \leq \frac{1}{10}.$$

We will also write this in the following form (where we use that $(xs-t)_i = x_i s_i - t$ by the notation we defined at the start of this chapter.)

$$\|\frac{xs-t}{t}\|_2 \leq \frac{1}{10}$$

Note that $\|(xs-t)/t\|_\infty \leq \|(xs-t)/t\|_2 \leq \frac{1}{10}$ so we have

Lemma 4.3.1. *If $\|(xs-t)/t\|_2 \leq \epsilon$, then $(1-\epsilon)\, t \leq x_i s_i \leq (1+\epsilon)\, t$ for all i.*

How to pick α? Throughout the algorithm, we want that we always have $xs \approx t$, i.e. $\|(xs-t)/t\|_2$ should be small. We now analyze how much $\|(xs-t)/t\|_2$ changes when decreasing $t \leftarrow t \cdot (1-\alpha)$.

Lemma 4.3.2. *If $\|(xs-t)/t\|_2 \leq 1/10$ and $\alpha \leq 0.1$, then for $t' \leftarrow t \cdot (1-\alpha)$ we have $\|(xs-t')/t'\|_2 \leq \|(xs-t)/t\|_2 + 1.2\alpha\sqrt{n}$.*

Proof.

$$\begin{aligned}
\|\frac{xs-t'}{t'}\|_2 &= \frac{1}{1-\alpha}\|\frac{xs-t'}{t}\|_2 = \frac{1}{1-\alpha}\|\frac{xs-t+\alpha t}{t}\|_2 \leq \frac{1}{1-\alpha}(\|\frac{xs-t}{t}\|_2 + \sqrt{n}\alpha) \\
&= (1+1.1\alpha)\cdot(\|\frac{xs-t}{t}\|_2 + \sqrt{n}\alpha) \leq \|\frac{xs-t}{t}\|_2 + 1.2\sqrt{n}\alpha
\end{aligned}$$

□

This tells us that, if we start with $\|(xs-t)/t\|_2 \leq 1/10$ and pick α some value of size roughly $O(1/\sqrt{n})$, then we end up with $\|\frac{xs-t'}{t'}\|_2 \leq \frac{2}{10}$. So after decreasing t we still have $0.8\, t \leq xs \leq 1.2\, t$.

Next, the question is if by moving $x \leftarrow x + \delta_x$, $y \leftarrow y + \delta_y$, $s \leftarrow s + \delta_s$, we can improve the norm back to $\|\frac{xs-t}{t}\|_2 \leq \frac{1}{10}$. If we can do this, the our algorithm converges in just $O(\sqrt{n}\log(t^{start} n/\epsilon))$ iterations.

How to pick $\delta_x, \delta_s, \delta_y$? We perform updates of the form $x \leftarrow x+\delta_x$, $y \leftarrow y+\delta_y$, $s \leftarrow s+\delta_s$ and we want that these new solutions satisfy all the linear program constraints, so

$$\begin{aligned}
\mathbf{A}^\top(x+\delta_x) &= b, \\
\mathbf{A}(y+\delta_y) + (s+\delta_s) &= c, \\
x+\delta_x > 0, s+\delta_s &> 0
\end{aligned}$$

Since we already had $\mathbf{A}^\top x = b$ and $\mathbf{A}y + s = c$ before our update, these conditions are equivalent to the following:

$$\mathbf{A}^\top \delta_x = 0, \tag{4.8}$$

$$\mathbf{A}\delta_y = -\delta_s \tag{4.9}$$

$$-\delta_x/x < 1, -\delta_s/s < 1 \tag{4.10}$$

Further, we want these updates to reduce $\|(xs-t)/t\|_2$. Let's see how this value changes when moving x and s by some small δ_x, δ_s:

$$(x+\delta_x)(s+\delta_s)-t = xs-t+s\delta_x+x\delta_s+\delta_x\delta_s \approx xs-t+s\delta_x+x\delta_s.$$

For the last step, the idea is that, since we move x and s only by a small amount, the product of two small values becomes even smaller. So $\delta_x\delta_s$ is very close to 0. This now motivates the following requirement for the vectors δ_x, δ_s.

$$s\delta_x + x\delta_s = \gamma(t-xs) \tag{4.11}$$

for some small $\gamma \in \mathbb{R}_{>0}$. If this condition is true, then we would have $(x+\delta_x)(s+\delta_s)-t \approx (1-\gamma)(xs-t)$, so our new xs is much closer to t. Formally we have

Lemma 4.3.3. *If $x\delta_s + s\delta_x = \gamma(t-xs)$, and $\|\frac{xs-t}{t}\|_2 \le 2/10$ then*

$$\|\frac{(x+\delta_x)(s+\delta_s)-t}{t}\|_2 \le (1-\gamma)\cdot\|\frac{xs-t}{t}\|_2 + 1.2\cdot\|\mathbf{X}^{-1}\delta_x\|_2\|\mathbf{S}^{-1}\delta_s\|_2$$

So if δ_x and δ_s are small enough, then the norm decreases by roughly a $(1-\gamma)$ factor.

Proof.

$$\begin{aligned}\|\frac{(x+\delta_x)(s+\delta_s)-t}{t}\|_2 = \|\frac{xs-t}{t}+\frac{x\delta_s+s\delta_x}{t}+\frac{\delta_x\delta_s}{t}\|_2 = \|(1-\gamma)\frac{xs-t}{t}+\frac{\delta_x\delta_s}{t}\|_2 \\ \le (1-\gamma)\|\frac{xs-t}{t}\| + \|\frac{\delta_x\delta_s}{t}\|_2\end{aligned}$$

Here the last term can be bounded as follows

$$\|\frac{\delta_x\delta_s}{t}\|_2 = \|\frac{\delta_x}{x}\frac{\delta_s}{s}\frac{xs}{t}\|_2 \le \|\frac{\delta_x}{x}\|_2\|\frac{\delta_s}{s}\|_\infty\|\frac{xs}{t}\|_\infty \le \|\frac{\delta_x}{x}\|_2\|\frac{\delta_s}{s}\|_2\cdot 1.2$$

□

We now specify a construction for $\delta_x, \delta_y, \delta_s$ that satisfies requirements (4.8),(4.9),(4.11).

Lemma 4.3.4. *Given x, y, s, t define*

$$\begin{aligned}\delta_s &= \mathbf{A}\overbrace{(\mathbf{A}^\top\mathbf{X}\mathbf{S}^{-1}\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{S}^{-1}(t-xs)\gamma}^{=:-\delta_y} \\ \delta_x &= \mathbf{S}^{-1}(t-xs)\gamma - \mathbf{X}\mathbf{S}^{-1}\delta_s\end{aligned}$$

Then conditions (4.8),(4.9),(4.11) are all satisfied.

Proof. We have $\mathbf{A}\delta_y = -\delta_s$ (4.9) by definition of δ_y and δ_s.
For condition (4.11) we have

$$\mathbf{X}\delta_s + \mathbf{S}\delta_x = \mathbf{X}\delta_s + \mathbf{S}\mathbf{S}^{-1}(t-xs)\gamma - \mathbf{S}\mathbf{X}\mathbf{S}^{-1}\delta_s = \mathbf{X}\delta_s + (t-xs)\gamma - \mathbf{X}\delta_s = (t-xs)\gamma.$$

Lastly, for (4.8) we have

$$\begin{aligned}\mathbf{A}^\top \delta_x &= \mathbf{A}^\top \mathbf{S}^{-1}(t - xs)\gamma - \mathbf{AXS}^{-1}\delta_s \\ &= \mathbf{A}^\top \mathbf{S}^{-1}(t - xs)\gamma - \underbrace{\mathbf{A}^\top \mathbf{XS}^{-1}\mathbf{A}(\mathbf{A}^\top \mathbf{XS}^{-1}\mathbf{A})^{-1}}_{\text{cancel each other}}\mathbf{S}^{-1}(t - xs)\gamma \\ &= \mathbf{A}^\top \mathbf{S}^{-1}(t - xs)\gamma - \mathbf{A}^\top \mathbf{S}^{-1}(t - xs)\gamma = 0\end{aligned}$$

□

We are left with verifying (4.10). To prove it, we will need the following lemma.

Lemma 4.3.5 (Orthogonal Projection)**.** *For any $n \times d$ matrix $\mathbf{M}$ of rank d and n-dimensional vector v, we have*

$$\|v\|_2^2 = \|\mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top v\|_2^2 + \|(\mathbf{I} - \mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top)v\|_2^2$$

This implies $\|\mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top v\|_2^2 \leq \|v\|_2^2$ since a norm is always positive. (Geometrically, $\mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top v$ is an orthogonal projection of v onto the image of $\mathbf{M}$.)

Proof of Lemma 4.3.5.

$$\begin{aligned}\|v\|_2^2 = v^\top v &= v^\top \mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top v + v^\top(\mathbf{I} - \mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top)v \\ &= v^\top \mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top \mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top v \\ &+ v^\top(\mathbf{I} - \mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M})(\mathbf{I} - \mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top)v \\ &= \|\mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top v\|_2^2 + \|(\mathbf{I} - \mathbf{M}(\mathbf{M}^\top \mathbf{M})^{-1}\mathbf{M}^\top)v\|_2^2\end{aligned}$$

□

We use Lemma 4.3.5 to prove Lemma 4.3.6 which then implies (4.10).

Lemma 4.3.6. *For δ_x, δ_s as in Lemma 4.3.4 and for $\|(xs-t)/t\|_2 \leq 2/10$ we have $\|\mathbf{X}^{-1}\delta_x\|_2 \leq 4\gamma$, $\|\mathbf{S}^{-1}\delta_s\|_2 \leq 4\gamma$.*

Note that by $\|\mathbf{X}^{-1}\delta_x\|_\infty \leq \|\mathbf{X}^{-1}\delta_x\|_2$ and for small enough $\gamma < 1/4$, we have $\|\mathbf{X}^{-1}\delta_x\|_\infty < 1$ and thus (4.10) holds true. The same argument can be made for $\|\mathbf{S}^{-1}\delta_s\|_\infty < 1$.

Proof. Let's start with δ_s

$$\begin{aligned}\|\mathbf{S}^{-1}\delta_s\|_2 &= \|\mathbf{S}^{-1}\mathbf{A}(\mathbf{A}^\top \mathbf{XS}^{-1}\mathbf{A})^{-1}\mathbf{AS}^{-1}(t - xs)\gamma\|_2 \\ &= \|\mathbf{S}^{-1/2}\mathbf{X}^{-1/2}\mathbf{X}^{1/2}\mathbf{S}^{-1/2}\mathbf{A}(\mathbf{A}^\top \mathbf{XS}^{-1}\mathbf{A})^{-1}\mathbf{AX}^{1/2}\mathbf{S}^{-1/2}\mathbf{X}^{-1/2}\mathbf{S}^{-1/2}(t - xs)\gamma\|_2 \\ &\leq 1.2\frac{1}{\sqrt{t}}\| \underbrace{\mathbf{X}^{1/2}\mathbf{S}^{-1/2}\mathbf{A}(\mathbf{A}^\top \mathbf{XS}^{-1}\mathbf{A})^{-1}\mathbf{AX}^{1/2}\mathbf{S}^{-1/2}}_{\text{orthogonal projection matrix, Lemma 4.3.5 for } \mathbf{M}=\mathbf{X}^{1/2}\mathbf{S}^{1/2}\mathbf{A}} \mathbf{X}^{-1/2}\mathbf{S}^{-1/2}(t - xs)\gamma\|_2 \\ &\leq 1.2\frac{1}{\sqrt{t}}\|\mathbf{X}^{-1/2}\mathbf{S}^{-1/2}(t - xs)\gamma\|_2 = 1.2\|\sqrt{t}\mathbf{X}^{-1/2}\mathbf{S}^{-1/2}\frac{t - xs}{t}\gamma\|_2 \\ &\leq 1.5\gamma\|\frac{xs - t}{t}\|_2 \leq 2\gamma\end{aligned}$$

For δ_x we have

$$\|\mathbf{X}^{-1}\delta_x\|_2 = \|\mathbf{X}^{-1}\mathbf{S}^{-1}(xs-t)\gamma - \mathbf{S}^{-1}\delta_s\|_2 \le \gamma\|\mathbf{X}^{-1}\mathbf{S}^{-1}(xs-t)\|_2 + \|\mathbf{S}^{-1}\delta_s\|_2$$

where we just bounded $\|\mathbf{S}^{-1}\delta_s\|_2 \le 2\gamma$ already and $\|\mathbf{X}^{-1}\mathbf{S}^{-1}(xs-t)\|_2 \le 2\|(xs-t)/t\|_2$. $\square$

Maintaining $xs \approx t$ throughout all iterations: We now have all tools available to show that our algorithm always has $\|(xs-t)/t\|_2 \le 1/10$ at the end of each iteration. We pick the parameters as follows:
$\alpha = 16\gamma^2/\sqrt{n}$
$\gamma := 0.01/\sqrt{2 \cdot 1.2 \cdot 16} > 0.0016$

Then by Lemma 4.3.2 we have for $t' \leftarrow t(1-\alpha)$

$$\|(xs-t')/t'\|_2 \le \|(xs-t)/t\|_2 + 1.2\alpha\sqrt{n} \le \|(xs-t)/t\|_2 + 1.2 \cdot 16\gamma^2$$

While by Lemmas 4.3.3 and 4.3.6 we have $\|\frac{(x+\delta_s)(s+\delta_s)-t'}{t'}\|_2 \le (1-\gamma)\|\frac{xs-t'}{t'}\|_2 + 1.2 \cdot 16\gamma^2$
$\le (1-\gamma)\|\frac{xs-t}{t}\|_2 + 2(1.2 \cdot 16\gamma^2)$ (we replaced t' by t and get at most an extra $(1.2 \cdot 16\gamma^2)$ term as shown above)
$\le (1-\gamma)\|\frac{xs-t}{t}\|_2 + 0.01^2 = (1-\gamma)\|\frac{xs-t}{t}\|_2 + 0.0001$ by our choice of γ
$\le (1-\gamma)0.1 + 0.0001$ by assumption that at the start of each iteration we have $\|\frac{xs-t}{t}\|_2 \le 0.1$
$= 0.1 - \gamma 0.1 + 0.0001$
$\le 0.1 - 0.00016 + 0.0001 < 0.1$ by choice of γ.

So at the end of an iteration when changing both $t \leftarrow (1-\alpha)t$ and $x \leftarrow x + \delta_x, s \leftarrow s + \delta_s$ we again have $\|(xs-t)/t\|_2 \le 0.1$.

The final algorithm looks as follows:

- Find some $t \in \mathbb{R}_{>0}$ and $x \in \mathbb{R}^n_{>0}$ with $\mathbf{A}^\top x = b$, and some $y \in \mathbb{R}^d, s \in \mathbb{R}^n_{>0}$ with $\mathbf{A}y + s = c$, such that $\|(xs-t)/t\|_2 \le 1/10$.
- while $t > \epsilon/n$
  - Set $t \leftarrow t \cdot (1 - 16\gamma^2/\sqrt{n})$ for $\gamma = 0.01/\sqrt{2 \cdot 1.2 \cdot 16}$.
  - Set $x \leftarrow x + \delta_x$, $y \leftarrow y + \delta_y$, $s \leftarrow s + \delta_s$ where
    $\delta_y = -\gamma(\mathbf{A}^\top\mathbf{X}\mathbf{S}^{-1}\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{S}^{-1}(t - xs)$
    $\delta_s = -\mathbf{A}\delta_y$, $\delta_x = \gamma\mathbf{S}^{-1}(t-xs) - \mathbf{X}\mathbf{S}^{-1}\delta_s$.

Complexity Analysis: After $O(\sqrt{n}\log(t^{start}n/\epsilon)) = \widetilde{O}(\sqrt{n})$ iterations the algorithm stops. In each iteration we must compute $\delta_x, \delta_s, \delta_y$. Multiplying a vector with $\mathbf{A}$ or $\mathbf{A}^\top$ takes only $O(nd)$ time. Computing $(\mathbf{A}^\top\mathbf{X}\mathbf{S}^{-1}\mathbf{A})$ takes $O(nd^2)$ time. Inverting this matrix takes $O(d^3)$ time. In summary, one iteration can be performed in $O(nd^2)$ time, for a total of $\widetilde{O}(n^{1.5}d^2)$ time over all iterations. This complexity was first achieved by Renegar (1988) [Ren88].

4.3.1 Initial Point

Last week we analyzed an algorithm that solves a linear program in $O(\sqrt{n}\log 1/\epsilon)$ iterations. The algorithm assumes that we have some initial point $x, s \in \mathbb{R}^n_{\ge 0}$, $y \in \mathbb{R}^d$, $t \in \mathbb{R}_{>0}$ such that $\mathbf{A}^\top x = b$, $\mathbf{A}y + s = c$ and $\|(xs-t)/t\|_2 \le 1/10$.

Such an initial point can be constructed via the following reduction by [YTM94] (for a recent proof see Lemma A.6 in [CLS19]).

Lemma 4.3.7 ([YTM94, CLS19]). *Consider a linear program $\min_{\mathbf{A}^\top x=b,x\geq 0} c^\top x$ with n variables and d constraints. Assume that for any $x \geq 0$ with $\mathbf{A}^\top x = b$, we have that $\|x\|_\infty \leq R$. For any $0 < \delta \leq 1$, the modified linear program $\min_{\overline{A}^\top \overline{x}=\overline{b},\overline{x}\geq 0} \overline{c}^\top \overline{x}$ with*

$$\overline{A} = \begin{bmatrix} \mathbf{A} & \mathbf{1} \\ 0 & 1 \\ \frac{1}{R}b^\top - \mathbf{1}^\top \mathbf{A} & 0 \end{bmatrix}, \overline{b} = \begin{bmatrix} \frac{1}{R}b \\ n+1 \end{bmatrix}, \text{ and } \overline{c} = \begin{bmatrix} \delta/\|c\|_\infty \cdot c \\ 0 \\ 1 \end{bmatrix}$$

satisfies the following:

1. $\overline{x} = \begin{bmatrix} 1_n \\ 1 \\ 1 \end{bmatrix}$, $\overline{y} = \begin{bmatrix} 0_d \\ -1 \end{bmatrix}$ *and* $\overline{s} = \begin{bmatrix} 1_n + \frac{\delta}{\|c\|_\infty} \cdot c \\ 1 \\ 1 \end{bmatrix}$ *are satisfy the constraint of the modified linear program.*
2. *For any solution $(\overline{x}, \overline{y}, \overline{s})$ with $\sum_{i=1}^n \overline{x}_i \overline{s}_i \leq \delta^2$, consider the vector $\hat{x} = R \cdot \overline{x}_{1:n}$ ($\overline{x}_{1:n}$ is the first n coordinates of $\overline{x}$) is an approximate solution to the original linear program in the following sense*

$$\begin{aligned} c^\top \hat{x} &\leq \min_{\mathbf{A}^\top x=b,x\geq 0} c^\top x + \|c\|_\infty R \cdot \delta, \\ \|\mathbf{A}\hat{x} - b\|_1 &\leq 2\delta \cdot \left(R \sum_{i,j} |\mathbf{A}_{i,j}| + \|b\|_1 \right), \\ \hat{x} &\geq 0. \end{aligned}$$

So for small enough $\delta > 0$, we obtain an approximate solution of the linear program. The algorithm from last lecture (Section 4.3) runs in $O(\sqrt{n}\log(n/\delta))$ iterations and each iteration takes $O(nd^2)$ time, resulting in the following Theorem 4.3.8.

Theorem 4.3.8. *Given $\mathbf{A} \in \mathbb{R}^{n\times d}$ of rank d, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^d$, let R be a bound on $\|x\|_1$ for all $x \geq 0$ with $Ax = b$. Then for any $0 < \delta \leq 1$ we can compute $x \geq 0$ such that*

$$c^\top x \leq \min_{Ax=b,x\geq 0} c^\top x + \delta\|c\|_\infty R \qquad \text{and} \qquad \|Ax - b\|_1 \leq \delta \left(R \sum_{i,j} |A_{i,j}| + \|b\|_1 \right)$$

in time $O(n^{1.5}d^2 \log(n/\delta))$.

4.4 Improvements via Approximate Inverse

The linear program solver from Section 4.3 runs in $O(\sqrt{n}\log(n/\delta))$ iterations. In each iterations, we must compute

$$\begin{aligned} \delta_s &= \mathbf{A}(\mathbf{A}^\top \mathbf{X}\mathbf{S}^{-1}\mathbf{A})^{-1}\mathbf{A}^\top \mathbf{S}^{-1}(t - xs) \cdot \gamma, \\ \delta_x &= \mathbf{S}^{-1}(t - xs)\gamma - \mathbf{X}\mathbf{S}^{-1}\delta_s. \end{aligned}$$

The time complexity of computing these two vectors is dominated by the cost of computing $\mathbf{A}^\top \mathbf{X}\mathbf{S}^{-1}\mathbf{A}$ in $O(nd^2)$ time. This is done in each iteration so the total time is $O(n^{1.5}d^2 \log(n/\delta))$.

We now want to improve this complexity to $O((nd + n^{1.5}d)\log(n/\delta))$. This complexity was first achieved by Vaidya (1987) [Vai87].

The idea is to maintain the matrix $\mathbf{A}^\top \mathbf{X}\mathbf{S}^{-1}\mathbf{A}$ (i.e., create a data structure for computing this matrix) instead of recomputing it from scratch in each iteration. For this, consider what happens if some entry x_i changes a bit by adding some $\beta > 0$. Then the new matrix is given by

$$\mathbf{A}^\top(\mathbf{X} + e_i e_i^\top \beta)\mathbf{S}^{-1}\mathbf{A} = \mathbf{A}^\top \mathbf{X}\mathbf{S}^{-1}\mathbf{A} + \mathbf{A}^\top e_i e_i^\top \beta \mathbf{S}^{-1}\mathbf{A} = \mathbf{A}^\top \mathbf{X}\mathbf{S}^{-1}\mathbf{A} + a_i a_i^\top \cdot \beta/s_i$$

where a_i is the ith row of $\mathbf{A}$. So we can maintain the matrix in just $O(d^2)$ time per change to any x_i (and likewise per change to any s_i). The problem is that in general, every entry of x could change from one iteration to the next, so it still needs $O(nd^2)$ time per iteration to maintain this matrix.

Outline An idea for how to get around every entry of x changing, is to define some $\overline{x} \in \mathbb{R}^n$ such that

$$(1-\epsilon)x \leq \overline{x} \leq (1+\epsilon)x$$

for some small constant $\epsilon > 0$. Then we want to use the matrix $\mathbf{A}^\top \overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A}$ instead of $\mathbf{A}^\top \mathbf{X}\mathbf{S}^{-1}\mathbf{A}$. This should lead to a speed up, because when we update $x \leftarrow x + \delta_x$ and an entry x_i does not change a lot, the old $\overline{x}_i$ is still a good approximation of x_i. So we can reuse the old $\overline{x}_i$. Only when x_i changes too much do we need to change $\overline{x}_i$ and thus perform an update to maintain $\mathbf{A}^\top \overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A}$.

There are now two questions we must answer:

- Since changing an entry of $\overline{x}$ or $\overline{s}$ results in paying $O(d^2)$ time to maintain $\mathbf{A}^\top \overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A}$, we must bound how often these vectors change.
- Does the algorithm from last lecture (Section 4.3) still work when we replace $\mathbf{A}^\top \mathbf{X}\mathbf{S}^{-1}\mathbf{A}$ by $\mathbf{A}^\top \overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A}$?

Question 1: Bounding the number of changes.

Lemma 4.4.1. *After T iteration of our algorithm, in total only $O(T\sqrt{n})$ changes are performed to $\overline{x}$ and $\overline{s}$.*

Proof. Let $x^{(t)}, \overline{x}^{(t)}, \delta_x^{(t)}$ be the vectors $x, \overline{x}, \delta_x$ during the tth iteration of our algorithm. So we have

- $x^{(t+1)} = x^{(t)} + \delta_x^{(t)}$
- If $(1-\epsilon)x_i^{(t+1)} \leq \overline{x}_i^{(t)} \leq (1+\epsilon)x_i^{(t+1)}$, then $\overline{x}_i^{(t+1)} = \overline{x}_i^{(t)}$ because we can reuse the old value.
- Otherwise, $\overline{x}_i^{(t+1)} = x_i^{(t+1)}$.

for all $t = 1, ..., T$ and $i = 1, ..., n$.

To bound the number of times we change $\overline{x}$, consider the following for any i. Let's say we are currently in iteration t and we must update $\overline{x}_i^{(t)} = x_i^{(t)}$. Let $k < t$ be the last time we updated $\overline{x}_i^{(k)} = x_i^{(k)}$. Then we have

$$x_i^{(t)} > \overline{x}_i^{(t-1)}(1+\epsilon)$$

because otherwise we would not change $\overline{x}_i^{(t)}$. Since $x^{(t)}$ is the sum of δ_x values, we have

$$x_i^{(k)} + \sum_{\ell=k}^{t-1}(\delta_x^{(\ell)})_i = x_i^{(t)} > \overline{x}_i^{(t-1)}(1+\epsilon) = \overline{x}_i^{(k)}(1+\epsilon) = x_i^{(k)}(1+\epsilon).$$

Subtracting $x^{(k)}$ from both sides yields

$$\sum_{\ell=k}^{t-1}(\delta_x^{(\ell)})_i > \epsilon \cdot x_i^{(k)} = \epsilon \cdot \overline{x}_i^{(k)}$$

which can in turn be transformed to

$$\sum_{\ell=k}^{t-1} \frac{(\delta_x^{(\ell)})_i}{\overline{x}_i^{(\ell)} \cdot \epsilon} > 1$$

So whenever we perform an update, this sum is larger than 1. We can thus upper bound the total number of changes over all iterations and all entries by

$$\sum_{i=1}^{n}\sum_{t=1}^{T}\left|\frac{(\delta_x^{(\ell)})_i}{\overline{x}_i^{(\ell)} \cdot \epsilon}\right| = \sum_{t=1}^{T}\left\|\frac{\delta_x^{(\ell)}}{\overline{x}^{(\ell)}}\right\|_1 \cdot \frac{1}{\epsilon} \leq \sqrt{n} \cdot \sum_{t=1}^{T}\left\|\frac{\delta_x^{(\ell)}}{\overline{x}^{(\ell)}}\right\|_2 \cdot \frac{1}{\epsilon} \leq O(\sqrt{n}) \cdot \sum_{t=1}^{T}\left\|\frac{\delta_x^{(\ell)}}{x^{(\ell)}}\right\|_2 \cdot \frac{1}{\epsilon}$$

where the 2nd step comes from Cauchy-Schwarz and the last step uses that $\overline{x}$ is always a $(1 \pm \epsilon)$-approximation of x. By Lemma 4.3.6 we know $\|\delta_x^{(\ell)}/x^{(\ell)}\|_2 = O(1)$, so the total number of changes is bounded by $O(T\sqrt{n}/\epsilon) = O(T\sqrt{n})$ for constant $\epsilon > 0$. The same argument holds for the number of changes to $\overline{s}$. □

In summary, our algorithm needs only $O(n \log(n/\delta))$ changes to $\overline{x}$ and $\overline{s}$ as we have $T = O(\sqrt{n}\log(n/\delta))$ iterations. Since each change costs us $O(d^2)$ to maintain $(\mathbf{A}^\top\overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathbf{A})^{-1}$, the total cost is bounded by $O(nd^2 \log(n/\delta))$. To compute δ_x and δ_s we additionally need $O(nd)$ time per iteration for the product with $\mathbf{A}$ and $\mathbf{A}^\top$. The total time over all iterations is thus $O((nd^2 + n^{1.5}d)\log(n/\delta))$.

Question 2: Does the algorithm still work? In each iteration we will move x, y, s by the following vectors

$$\begin{aligned} \delta_s &= \mathbf{A}\overbrace{(\mathbf{A}^\top\overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}\overline{\mathbf{S}}^{-1}(t - xs) \cdot \gamma}^{=:-\delta_y} \\ \delta_x &= \overline{\mathbf{S}}^{-1}(t - xs) \cdot \gamma - \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\delta_s \end{aligned} \tag{4.12}$$

We now need to verify that all the lemmas from last lecture (Lemmas 4.3.3, 4.3.4 and 4.3.6 still hold for our slightly different choice of $\delta_x, \delta_y, \delta_s$.

Lemma 4.4.2 (Similar to Lemma 4.3.4)**.** *For δ_x, δ_s as defined in (4.12) we have*

$$\begin{aligned}\mathbf{A}^\top \delta_x &= 0\\ \mathbf{A}\delta_y &= -\delta_s\\ \overline{\mathbf{X}}\delta_s + \overline{\mathbf{S}}\delta_x &= \gamma \cdot (t - xs)\end{aligned}$$

This lemma implies that after moving x, y, s we still satisfy the constraints $\mathbf{A}^\top x = b$ and $\mathbf{A}y + s = c$ of the linear program.

Proof. Property $\mathbf{A}\delta_y = -\delta_s$ holds by definition. We further have

$$\begin{aligned}\mathbf{A}^\top \delta_x &= \mathbf{A}^\top \overline{\mathbf{S}}^{-1}(t - xs) \cdot \gamma - \mathbf{A}^\top \overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A}(\mathbf{A}^\top \overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}\overline{\mathbf{S}}^{-1}(t - xs) \cdot \gamma\\ &= \mathbf{A}^\top \overline{\mathbf{S}}^{-1}(t - xs) \cdot \gamma - \mathbf{A}^\top \overline{\mathbf{S}}^{-1}(t - xs) \cdot \gamma = 0\end{aligned}$$

and

$$\overline{\mathbf{X}}\delta_s + \overline{\mathbf{S}}\delta_x = \overline{\mathbf{X}}\delta_s + \overline{\mathbf{S}\mathbf{S}}^{-1}(t - xs) \cdot \gamma - \overline{\mathbf{S}\mathbf{X}\mathbf{S}}^{-1}\delta_s = \overline{\mathbf{X}}\delta_s + (t - xs) \cdot \gamma - \overline{\mathbf{X}}\delta_s = \gamma \cdot (t - xs)$$

□

Note that unlike (4.11), we now satisfy $\overline{x}\delta_s + \overline{s}\delta_x = \gamma \cdot (t - xs)$. Thus we must reprove Lemma 4.3.3 for this small variation.

Lemma 4.4.3 (Similar to Lemma 4.3.3)**.** *If $\overline{x}\delta_s + \overline{s}\delta_x = \gamma \cdot (t - xs)$, $\overline{x} = (1 \pm \epsilon)x, \overline{s} = (1 \pm \epsilon)s$ and $\|(xs - t)/t\|_2 \leq 1/10$ then*

$$\|\frac{(x + \delta_x)(s + \delta_s) - t}{t}\|_2 \leq (1 - \gamma)\|\frac{xs - t}{t}\|_2 + 2.2\epsilon(\|\mathbf{X}^{-1}\delta_x\|_2 + \|\mathbf{S}^{-1}\delta_s\|_2) + 1.2\|\mathbf{X}^{-1}\delta_x\|_2\|\mathbf{S}^{-1}\delta_s\|_2$$

So similar to the old algorithm, if $\|\mathbf{X}^{-1}\delta_x\|_2$, $\|\mathbf{S}^{-1}\delta_s\|_2$ are small enough, we move xs closer to t.

Proof.

$$\begin{aligned}\|\frac{(x + \delta_x)(s + \delta_s) - t}{t}\|_2 &= \|\frac{xs - t}{t} + \frac{x\delta_s + s\delta_x}{t} + \frac{\delta_x\delta_s}{t}\|_2\\ &= \|\frac{xs - t}{t} + \frac{\overline{x}\delta_s + \overline{s}\delta_x}{t} + \frac{(x - \overline{x})\delta_s + (s - \overline{s})\delta_x}{t} + \frac{\delta_x\delta_s}{t}\|_2\\ &\leq (1 - \gamma) \cdot \|\frac{xs - t}{t}\|_2 + \|\frac{(x - \overline{x})\delta_s}{t}\|_2 + \|\frac{(s - \overline{s})\delta_x}{t}\|_2 + \|\frac{\delta_x\delta_s}{t}\|_2\end{aligned}$$

Here the term $\|\frac{\delta_x\delta_s}{t}\|_2 \leq 1.2\gamma^2$ by the proof of Lemma 4.3.3. Further we can bound

$$\|\frac{(x - \overline{x})\delta_s}{t}\|_2 \leq 1.1\|\frac{(x - \overline{x})\delta_s}{xs}\|_2 = 1.1\|\frac{(x - \overline{x})}{x} \cdot \frac{\delta_s}{s}\|_2 \leq 1.1\epsilon\|\frac{\delta_s}{s}\|_2$$

and the same bound can be proven in the same way for $\|\frac{(s-\overline{s})\delta_x}{t}\|_2$. In summary,

$$\|\frac{(x+\delta_x)(s+\delta_s)-t}{t}\|_2 \leq (1-\gamma)\|\frac{xs-t}{t}\|_2 + 2.2\epsilon(\|\mathbf{X}^{-1}\delta_x\|_2 + \|\mathbf{S}^{-1}\delta_s\|_2) + 1.2\|\mathbf{X}^{-1}\delta_x\|_2\|\mathbf{S}^{-1}\delta_s\|_2$$

□

At last, we must reprove Lemma 4.3.6

Lemma 4.4.4 (Similar to Lemma 4.3.6)**.** *For $\|(xs-t)/t\|_2 \leq 2/10$, $\overline{x} = (1 \pm 0.1)x$, $\overline{s} = (1 \pm 0.1)s$, and δ_x, δ_s as in (4.12), we have $\|\mathbf{X}^{-1}\delta_x\|_2 \leq 4\gamma$, $\|\mathbf{S}^{-1}\delta_s\|_2 \leq 4\gamma$.*

Proof. The proof is essentially the same as in Lemma 4.3.6.

$$\begin{aligned}
\|\mathbf{S}^{-1}\delta_s\|_2 &= \|\mathbf{S}^{-1}\mathbf{A}(\mathbf{A}^\top\overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}\overline{\mathbf{S}}^{-1}(t-xs)\cdot\gamma\|_2 \\
&= \|\mathbf{S}^{-1}\overline{\mathbf{S}\mathbf{X}}^{-1/2}\overline{\mathbf{S}}^{-1/2}\overline{\mathbf{X}}^{1/2}\overline{\mathbf{S}}^{-1/2}\mathbf{A}(\mathbf{A}^\top\overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}\overline{\mathbf{S}}^{-1/2}\overline{\mathbf{X}}^{1/2}\overline{\mathbf{X}}^{-1/2}\overline{\mathbf{S}}^{-1/2}(t-xs)\cdot\gamma\|_2 \\
&\leq 1.1\cdot\|\overline{\mathbf{X}}^{-1/2}\overline{\mathbf{S}}^{-1/2}\overline{\mathbf{X}}^{1/2}\overline{\mathbf{S}}^{-1/2}\mathbf{A}(\mathbf{A}^\top\overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}\overline{\mathbf{S}}^{-1/2}\overline{\mathbf{X}}^{1/2}\overline{\mathbf{X}}^{-1/2}\overline{\mathbf{S}}^{-1/2}(t-xs)\cdot\gamma\|_2 \\
&\leq 1.3\cdot\|\mathbf{X}^{-1/2}\mathbf{S}^{-1/2}\overline{\mathbf{X}}^{1/2}\overline{\mathbf{S}}^{-1/2}\mathbf{A}(\mathbf{A}^\top\overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}\overline{\mathbf{S}}^{-1/2}\overline{\mathbf{X}}^{1/2}\overline{\mathbf{X}}^{-1/2}\overline{\mathbf{S}}^{-1/2}(t-xs)\cdot\gamma\|_2 \\
&\leq 1.5\frac{1}{\sqrt{t}}\cdot\|\overline{\mathbf{X}}^{1/2}\overline{\mathbf{S}}^{-1/2}\mathbf{A}(\mathbf{A}^\top\overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}\overline{\mathbf{S}}^{-1/2}\overline{\mathbf{X}}^{1/2}\overline{\mathbf{X}}^{-1/2}\overline{\mathbf{S}}^{-1/2}(t-xs)\cdot\gamma\|_2 \\
&\leq 1.5\frac{1}{\sqrt{t}}\cdot\|\overline{\mathbf{X}}^{-1/2}\overline{\mathbf{S}}^{-1/2}(t-xs)\cdot\gamma\|_2 \\
&\leq 1.7\frac{1}{\sqrt{t}}\cdot\|\mathbf{X}^{-1/2}\mathbf{S}^{-1/2}(t-xs)\cdot\gamma\|_2 \\
&\leq 1.9\cdot\gamma\cdot\|\frac{t-xs}{t}\|_2
\end{aligned}$$

Here we use that $xs \approx t$ and $\overline{x} \approx x$, $\overline{s} \approx s$. □

In summary, all the lemmas from the old algorithm still work if we replace x, s by approximate $\overline{x}, \overline{s}$ as in (4.12). In particular, the algorithm from Section 4.3 still works. We obtain Theorem 4.4.5, which is a faster version of Theorem 4.3.8.

Theorem 4.4.5. *Given $\mathbf{A} \in \mathbb{R}^{n\times d}$ of rank d, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^d$, let R be a bound on $\|x\|_1$ for all $x \geq 0$ with $Ax = b$. Then for any $0 < \delta \leq 1$ we can compute $x \geq 0$ such that*

$$c^\top x \leq \min_{Ax=b, x\geq 0} c^\top x + \delta\|c\|_\infty R \qquad \text{and} \qquad \|Ax-b\|_1 \leq \delta\left(R\sum_{i,j}|A_{i,j}| + \|b\|_1\right)$$

in time $O((nd^2 + n^{1.5}d)\log(n/\delta))$.

Chapter 5

Solving Linear Programs in nd^2 time

5.1 Idea for a faster Algorithm

In the previous lecture we improved the complexity of the linear program solver to $O((nd^2 + n^{1.5}d)\log(n/\delta))$ time. Note that we generally have $n \gg d$ so in most cases the term $n^{1.5}d$ will dominate. We now want to remove this term and further improve the complexity to $O(nd^2 \log(n/\delta))$ time.

For this, we first observe where this term comes from. In each iteration, our algorithm computes

$$\delta_s = \mathbf{A}(\mathbf{A}^\top \overline{\mathbf{X}}\overline{\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}^\top \overline{\mathbf{S}}^{-1}(t - xs).$$

Here the product $\mathbf{A}^\top \overline{\mathbf{S}}^{-1}(t-xs)$ takes $O(nd)$ time, as it is multiplying a $d \times n$ matrix by an n-dimensional vector. So after $O(\sqrt{n}\log(n/\delta))$ iterations, this is where the $O(n^{1.5}d\log(n/\delta))$ term in the overall complexity comes from. An intuitive idea for how to speed up this computation would be to define $\bar{t}$ as some approximate t with $(1-\epsilon)t \leq \bar{t} \leq (1+\epsilon)t$, similar to how $\overline{x}, \overline{s}$ are approximate versions of x, s. Then we can maintain $\mathbf{A}^\top \overline{\mathbf{S}}^{-1}(\bar{t} - \overline{xs})$ in $O(d)$ timer per changed entry in $\overline{x}$ or $\overline{s}$. Changing $\bar{t}$ would take $O(nd)$ time.

Note that maintaining $\mathbf{A}^\top \overline{\mathbf{S}}^{-1}(\bar{t} - \overline{xs})$ takes only $O(nd\log(n/\delta))$ total time over all iterations. This is because by Lemma 4.4.1 we only have $O(n\log(n\delta))$ changes to $\overline{x}$ and $\overline{s}$. Further, it takes $O(\sqrt{n}/\epsilon)$ iterations until t changes by a $(1-\epsilon)$-factor (since in each iteration we set $t \leftarrow (1 - O(1/\sqrt{n})) \cdot t$). So we change $\bar{t}$ only once every $O(\sqrt{n})$ iterations (for constant $\epsilon > 0$).

We will later argue why also the product with $\mathbf{A}$ on the far left in the definition of δ_s can be accelerated. For now, we focus on the idea of maintaining $\mathbf{A}^\top \overline{\mathbf{S}}^{-1}(\bar{t} - \overline{xs})$.

Problems with this approach Unfortunately, we can not simply replace x, s, t in $(t - xs)$ by $\overline{x}, \overline{s}, \bar{t}$. The algorithm will break. To see why, consider the case where $\overline{x} = (1+\epsilon)x$, $\overline{s} = s$, $\bar{t} = t$ and $x_i s_i = (1+\epsilon)t$ for all $i = 1, ..., n$. Then each of $\overline{x}, \overline{s}, \bar{t}$ is a valid approximation

of x, s, t. Further, $(\bar{t} - \overline{xs}) = 0$ so $\delta_s = \delta_s = 0$. However, $\|(xs - t)/t\|_2 = O(\sqrt{n}\epsilon)$. This is a problem, because the algorithm from Section 4.3 required $\|(xs - t)/t\|_2 \leq 1/10$. In each iteration, it moves x, s by δ_x, δ_s to decrease this norm. But now, even if this norm is much larger, we might not be able to reduce it. This is because in our example $\delta_x = \delta_s = 0$ so we are not moving our x, s at all.

One idea would be to use the norm $\|(xs - t)/t\|_\infty$ instead. This norm would only be of size $O(\epsilon)$ in above example. Unfortunately the proof of Lemma 4.3.3 breaks when replacing the $\|\cdot\|_2$-norm by the $\|\cdot\|_\infty$-norm, so we can not use it.[1]

The purpose of $\|(xs - t)/t\|_2$ was to measure how close the product xs is to t. We now want to use a slightly different method of measuring the closeness of $xs \approx t$.

Definition 5.1.1. For any $\lambda > 0$, define $\Phi_\lambda : \mathbb{R}^n \to \mathbb{R}$ via

$$\Phi_\lambda(v) = \frac{1}{2}\sum_{i=1}^{n} \exp(\lambda v_i) + \exp(-\lambda v_i) = \sum_{i=1}^{n} \cosh(\lambda v_i).$$

Note that this function Φ_λ is minimized for the all-zeros-vector. If any entry $v_i \neq 0$, then the exponential function will blow-up and $\Phi_\lambda(v)$ becomes large. So we can use $\Phi_\lambda((xs-t)/t)$ to measure how close $(xs - t)/t$ is to 0, or equivalently, how close xs is to t. Further, note that because of the exponential growth, the value of $\Phi_\lambda(v)$ will be dominated by the term $\exp(\lambda v_i)$ for which v_i is largest. So $\Phi_\lambda(v)$ behaves similarly to $\|v\|_\infty$ in that it's value will depend (almost) exclusively on the largest entry of v. More formally, we can prove the following:

Lemma 5.1.2. *For any $v \in \mathbb{R}^n$ we have $\|v\|_\infty \leq \ln(2\Phi_\lambda(v))/\lambda$.*

This lemma tells us that if $\Phi_\lambda((xs - t)/t) \leq \frac{1}{2}\exp(\lambda/10)$, then $0.9 \cdot t \leq xs \leq 1.1t$.

For the algorithm in Section 4.3 we used to prove $\|(xs - t)/t\|_2 \leq 1/10$ to argue $0.9 \cdot t \leq xs \leq 1.1t$. Now we want to modify this algorithm to promise $\Phi_\lambda((xs-t)/t) \leq 0.5\exp(\lambda/10)$ instead. We will later pick $\lambda = c\log(n)$ for some constant c, so we get $\|(xs-t)/t\|_\infty = O(1)$ when $\Phi_\lambda((xs - t)/t) \leq \mathrm{poly}(n)$. In particular for $\Phi_\lambda((xs - t)/t) \leq \frac{1}{2}\exp(\lambda/10)$ we get $0.9t \leq xs \leq 1.1t$ by Lemma 5.1.2.

5.2 Robust Interior Point Method

We recap the algorithmic framework from Section 4.3 for solving linear programs, together with the changes proposed in the previous section. This modified framework is often called "robust interior point method" [CLS19, LSZ19, **?**, JSWZ21] because it is robust against approximation errors. In particular, we can use some approximate $\overline{x}, \overline{s}$ instead of exact x, s.

- Find initial point $x, s \in \mathbb{R}^n_{\geq 0}$, $y \in \mathbb{R}^d$, $t \in \mathbb{R}_{>0}$ such that $\mathbf{A}^\top x = b$, $\mathbf{A}y + s = c$, and $\Phi_\lambda((xs - t)/t) \leq \frac{1}{2}\exp(\lambda/10)$.
- while $t > \delta/n$

[1] It's possible to fix this by increasing the number of iterations to $O(n\log(n/\delta))$, but that would mean we have a slower algorithm.

– decrease $t \leftarrow (1-\alpha) \cdot t$
– move $x \leftarrow x + \delta_x$, $s \leftarrow s + \delta_s$, $y \leftarrow y + \delta_y$, such that again $x \geq 0$, $\mathbf{A}^\top x = b$, $s \geq 0$, $\mathbf{A}y + s = c$, and $\Phi_\lambda((xs - t)/t) \leq \frac{1}{2}\exp(\lambda/10)$.

Here we define the movement $\delta_x, \delta_y, \delta_s$ to be

$$\delta_s = \mathbf{A}\overbrace{(\mathbf{A}^\top \overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}\overline{\mathbf{S}}^{-1} g}^{=:-\delta_y} \tag{5.1}$$
$$\delta_x = \overline{\mathbf{S}}^{-1} g - \overline{\mathbf{X}\mathbf{S}}^{-1}\delta_s$$
$$g = -\gamma \cdot t \cdot \frac{\nabla \Phi_\lambda(\overline{v})}{\|\nabla \Phi(\overline{v})\|_2}$$

for $\overline{x} \approx_\epsilon x, \overline{s} \approx_\epsilon s$, and $\overline{v}$ an approximate version of $v := (xs - t)/t$ with $\|\overline{v} - v\|_\infty \leq \epsilon$.

The intuition for $\delta_x, \delta_s, \delta_y$ as in (5.1) is as follows. If we were to let $g = \gamma(t - xs)$, then (5.1) would be exactly the same as (4.12). In our previous algorithm from **??**, we wanted to satisfy

$$\frac{(x+\delta_x)(s+\delta_s) - t}{t} \approx \frac{xs - t + g}{t} = (1-\gamma)\frac{xs - t}{t}$$

(see e.g. Lemma 4.4.3). Now the new algorithm instead picks g as in (5.1), so we have

$$\frac{(x+\delta_x)(s+\delta_s) - t}{t} \approx \frac{xs - t + g}{t} \approx \frac{xs - t}{t} - \gamma \frac{\nabla \Phi_\lambda((xs-t)/t)}{\|\nabla \Phi_\lambda((xs-t)/t)\|_2}.$$

In other words, we move $(xs - t)/t$ in direction $-\nabla \Phi_\lambda((xs - t)/t)$. That means we are performing a gradient descent that reduces the value of $\Phi_\lambda((xs - t)/t)$. So by picking $\delta_x, \delta_s \delta_y$ as in (5.1), we hope to be able to maintain $\Phi_\lambda((xs - t)/t) \leq 0.5\exp(\lambda/10)$ which then implies $0.9t \leq xs \leq 1.1t$ by Lemma 5.1.2.

To prove the algorithm works, and to analyze the number of iterations, we must analyze the following questions:

- For movement $\delta_x, \delta_s, \delta_y$, do the solutions x, y, s satisfy all linear program constraints?
- How large can we make α without blowing up $\Phi_\lambda((xs - t)/t)$ too much?
- How much can we decrease $\Phi_\lambda((xs - t)/t)$ when moving x, s by δ_x, δ_s?

Especially proving the last question requires a lot of work, because we do not perform a perfect gradient descent. We move the value $v := (xs - t)/t$ in direction $\nabla\Phi(\overline{v})$ instead of $\nabla\Phi(v)$. We must prove that even with the approximation of $\|\overline{v} - v\|_\infty \leq \epsilon$, this approximate gradient descent still works.

5.2.1 Feasibility

Let us first verify that our vectors x, y, s stay feasible (i.e. satisfy the linear program constraints).

Lemma 5.2.1. *For $\delta_x, \delta_y, \delta_s$ as in (5.1) we have $\mathbf{A}^\top \delta_x = 0$, $\delta_s = -\mathbf{A}\delta_y$.*

This implies that $\mathbf{A}^\top x = b$ and $\mathbf{A}y + s = c$ is satisfied after updating x and s.

Proof. Same proof as in Lemma 4.4.2. Just replace $\gamma(t - xs)$ by the vector g from (5.1). □

Lemma 5.2.2. *For $\delta_x, \delta_y, \delta_s$ as in (5.1) and $0.8t \leq xs \leq 1.2t$ we have $\|\mathbf{X}^{-1}\delta_x\|_2 \leq 4\gamma$ and $\|\mathbf{S}^{-1}\delta_s\|_2 \leq O(\gamma)$.*

For small enough constant $\gamma > 0$ this implies $x > 0$ and $s > 0$ after updating x and s.

Proof. Same proof as in Lemma 4.4.4. Just replace $\gamma(t - xs)$ by the vector g from (5.1). □

In summary, after moving $x \leftarrow x + \delta_x$, $y \leftarrow y + \delta_y$ and $s \leftarrow s + \delta_s$, we still satisfy all the constraints of the linear program.

5.2.2 Maintaining small Φ

How much to decrease t We decrease $t \leftarrow (1-\alpha) \cdot t$, but how large can we pick α so that $\Phi_\lambda((xs - t)/t)$ does not change too much?

The following lemma implies some $\alpha = O(1/\sqrt{n})$ works.

Lemma 5.2.3. *Let $t' \leftarrow t \cdot (1-\alpha)$, then $\Phi_\lambda((xs - t')/t') \leq (1 + \lambda\alpha) \cdot \Phi_\lambda((xs - t))$.*

This tells us that the maximum value for α is some $O(1/\lambda)$, because we have $\|(xs - t)/t\|_\infty \leq \ln(\Phi_\lambda((xs - t)/t))/\lambda$ (via Lemma 5.1.2) and want to guarantee $\|(xs - t)/t\|_\infty \leq 2/10$. We will later see though, the our update to x, y, s can decrease Φ_λ only by some $(1 - 1/\sqrt{n})$-factor. That means we have two options: Decrease t by $(1 - O(1/\lambda))$ and then repeatedly move x, y, s for $O(\sqrt{n})$ iterations, or alternatingly decrease t by some $O(1 - O(1/(\lambda\sqrt{n})))$ and then perform a single movement of x, y, s. Both variants would have the same complexity.

Proof. Let $v = (xs - t)/t$ and $v' = (xs - t')/t'$, then $\|v' - v\|_\infty \leq \alpha$ and

$$\Phi_\lambda(v') = \Phi_\lambda(v + (v' - v)) = \sum_{i=1}^n \exp(\lambda v_i) \exp(\lambda(v'_i - v_i)) + \exp(-\lambda v_i) \exp(-\lambda(v'_i - v_i))$$
$$\leq \exp(\lambda\|v' - v\|_\infty) \sum_{i=1}^n \exp(\lambda v_i) + \exp(-\lambda v_i) = \exp(\lambda\|v' - v\|_\infty) \cdot \Phi_\lambda(v).$$

Here we can bound for $\|v' - v\| < 0.5/\lambda$ that $\exp(\lambda\|v' - v\|_\infty) \leq 1 + 2\lambda\|v' - v\|_\infty = 1 + O(\lambda\alpha)$. □

How much do we decrease Φ? We now analyze how much the value of $\Phi((xs - t)/t)$ decreases when moving x, y, s as defined in (5.1). We will prove the following lemma:

Lemma 5.2.4. *For δ_x, δ_s as in (5.1), we have*

$$\Phi(\frac{(x + \delta_x)(s + \delta_s) - t}{t}) \leq \left(1 + \frac{\lambda}{2\sqrt{n}}(-\gamma + O(\epsilon\gamma + \gamma^2\lambda^2))\right) \Phi\left(\frac{xs - t}{t}\right) + O(\gamma\lambda\sqrt{n})$$

Note, this implies that for small enough $0 < \epsilon, \gamma = O(1/\lambda)$ that $\Phi((xs-t)/t) \le cn$ throughout all iterations for some constant c. So we pick $\lambda = 10\ln(cn)$ which then implies $\|(xs-t)/t\|_\infty \le \ln(\Phi((xs-t)/t))/\lambda \le 0.1$. Thus after proving Lemma 5.2.4, we are done with proving the correctness of our algorithm. The proof presented here is based on [**?**], but similar proofs have also been given in [CLS19, LSZ19, **?**, JSWZ21].

As previously outlined, we are essentially performing a gradient descent on the function Φ_λ. However, (5.1) uses a bunch of approximate vectors (i.e. $\overline{x} \approx_\epsilon x$, $\overline{s} \approx_\epsilon s$) so we are not actually moving in exactly the direction of the gradient. The following lemma bounds how far off we are from the gradient.

Lemma 5.2.5. *For δ_x, δ_s as in (5.1) and $0.8t \le xs \le t$ we have*

$$\|\frac{(x+\delta_x)(s+\delta_s)}{t} + \gamma\frac{\nabla\Phi(\overline{v}))}{\|\nabla\Phi(\overline{v})\|_2}\|_2 \le O(\epsilon\gamma + \gamma^2)$$

Proof.

$$\begin{aligned}
\frac{(x+\delta_x)(s+\delta_s)}{t} &= \frac{xs + x\delta_s + s\delta_x + \delta_x\delta_s}{t} \\
&= \frac{xs + \overline{x}\delta_s + \overline{s}\delta_x + (x-\overline{x})\delta_s + (s-\overline{s})\delta_x + \delta_x\delta_s}{t} \\
&= \frac{xs}{t} - \gamma\frac{\nabla\phi_\lambda(\overline{v})}{\|\nabla\Phi_\lambda(\overline{v})\|_2} + \frac{(x-\overline{x})\delta_s + (s-\overline{s})\delta_x + \delta_x\delta_s}{t}
\end{aligned}$$

Thus we just need to bound the norm of the right-most terms.

$$\|\frac{(x-\overline{x})\delta_s + (s-\overline{s})\delta_x + \delta_x\delta_s}{t}\|_2 \le \|\frac{(x-\overline{x})\delta_s}{t}\|_2 + \|\frac{(s-\overline{s})\delta_x}{t}\|_2 + \|\frac{\delta_x\delta_s}{t}\|_2$$

Here we can bound

$$\|\frac{(x-\overline{s})\delta_s}{t}\|_2 = O(1)\cdot\|\frac{(x-\overline{s})\delta_s}{xs}\|_2 = O(\epsilon)\cdot\|\frac{\delta_s}{s}\|_2 = O(\epsilon\gamma)$$

by $xs \approx t$ and Lemma 5.2.2. Further,

$$\|\frac{\delta_x\delta_s}{t}\|_2 = O(1)\cdot\|\frac{\delta_x\delta_s}{xs}\|_2 \le O(1)\cdot\|\frac{\delta_x}{x}\|_2\cdot\|\frac{\delta_s}{s}\|_2 = O(\gamma^2).$$

□

Next, observe that in (5.1) we do not pick exactly the gradient $\nabla\Phi_\lambda((xs-t)/t)$, but some approximate $\nabla\Phi_\lambda(\overline{v})$ for some $\|\overline{v} - ((xs-t)/t)\|_\infty \le \epsilon$. The following lemma bounds how large the error is from such an approximation.

Lemma 5.2.6. *For any $v, \delta \in \mathbb{R}^n$ with $\|\delta\|_\infty \le \epsilon/\lambda \le 1/(5\lambda)$ we have*

$$\|\nabla\Phi_\lambda(v)) - \nabla\Phi_\lambda(v+\delta)\|_2 \le \epsilon(\|\nabla\Phi_\lambda(v)\|_2 + \sqrt{n}\lambda).$$

Proof. Note that $\Phi_\lambda(v)_i = \sum_{i=1}^n \cosh(\lambda v_i)$, so $(\nabla\Phi_\lambda(v))_i = \lambda\sinh(\lambda v_i)$. Further, $\sinh(a+b) = \sinh(a)\cosh(b) + \cosh(a)\sinh(b)$ and $|\cosh(a) - \sinh(a)| \le 1$. Thus we can bound for $b \le \beta \le 1/5$

$$\begin{aligned}
|\sinh(a+b) - \sinh(a)| &= |\sinh(a)\cosh(b) + \cosh(a)\sinh(b) - \sinh(a)| \\
&=|(\cosh(b)-1)\cdot\sinh(a) + \cosh(a)\sinh(b)| \\
&\le |\cosh(b)-1|\cdot|\sinh(a)| + \cosh(a)\cdot|\sinh(b)| \\
&\le |\cosh(b)-1|\cdot|\sinh(a)| + (1+|\sinh(a)|)\cdot|\sinh(b)| \\
&= (|\cosh(b)-1| + |\sinh(b)|)\cdot|\sinh(a)| + |\sinh(b)| \\
&\le \beta|\sinh(a)| + \beta.
\end{aligned}$$

where the last step used $(|\cosh(b)-1| + |\sinh(b)| \le \beta$ for $\beta \le 1/5$. Thus we have

$$\|\nabla\Phi_\lambda(v+\delta) - \nabla\Phi(v)\|_2 \le \epsilon(\|\nabla\Phi_\lambda(v)\|_2 + \sqrt{n}\lambda).$$

□

Combining the previous two lemmas, we can bound the overall error between our movement of xs and the gradient that would reduce the function $\Phi_\lambda((xs-t)/t)$.

Corollary 5.2.7. *For* $\|\overline{v} - v\|_\infty \le \epsilon/\lambda$ *in* (5.1) *and* $0.8t \le xs \le 1.2t$ *we have*

$$\|\frac{(x+\delta_x)(s+\delta_s)}{t} - \gamma\frac{\nabla\Phi_\lambda((xs-t)/t)}{\nabla\|\Phi(\overline{v})\|_2}\|_2 \le \epsilon\gamma\cdot\frac{\|\nabla\Phi_\lambda((xs-t)/t)\|_2 + \sqrt{n}\lambda}{\nabla\|\Phi(\overline{v})\|_2} + O(\epsilon\gamma + \gamma^2)$$

We now prove Lemma 5.2.4.

Proof of Lemma 5.2.4. Let $v = (xs-t)/t$ then

$$\Phi(\frac{(x+\delta_x)(s+\delta_s)-t}{t}) = \Phi(v - \gamma\frac{\nabla\Phi(v)}{\|\nabla\Phi(\overline{v})\|_2} + w)$$

for some vector w with

$$\|w\|_2 \le \epsilon\gamma\cdot\frac{\|\nabla\Phi_\lambda((xs-t)/t)\|_2 + \sqrt{n}\lambda}{\nabla\|\Phi(\overline{v})\|_2} + O(\epsilon\gamma + \gamma^2).$$

By Taylor approximation we have for any $\delta \in \mathbb{R}^n$ with $\|\delta\|_\infty \le 1/\lambda$ that there exists some $v \le \zeta \le v + \delta$ with

$$\Phi(v+\delta) = \Phi(v) + \delta^\top\nabla\Phi(v) + \frac{1}{2}\delta^\top(\nabla^2\Phi(\zeta))\delta \le \Phi(v) + \delta^\top\nabla\Phi(v) + 4\delta^\top(\nabla^2\Phi(v))\delta$$

where the last step uses $\|\delta\|_\infty \le 1/\lambda$ so we have $0 \le 0.5\nabla^2\Phi(v+\delta) \le 4\nabla^2\Phi(v)$.

This then implies (by letting $\delta = \gamma\frac{\nabla\Phi(v)}{\|\nabla\Phi(\overline{v})\|_2} + w$ where $\|\delta\|_2 \le O(\gamma)$) that

$$\begin{aligned}
\Phi(\frac{(x+\delta_x)(s+\delta_s)-t}{t}) &\le \Phi(v) + (-\gamma\frac{\nabla\Phi(v)}{\|\nabla\Phi(\overline{v})\|_2} + w)^\top\nabla\Phi(v) + 4\gamma^2\|\frac{\nabla\Phi(v)}{\|\nabla\Phi(\overline{v})\|_2} + w\|^2_{\nabla^2\Phi(v)} \\
&= \Phi(v) - \gamma\frac{\|\nabla\Phi(v)\|_2^2}{\|\nabla\Phi(\overline{v})\|_2} + w^\top\nabla\Phi(v) + 4\gamma^2\|\delta\|^2_{\nabla^2\Phi(v)}
\end{aligned}$$

Here we can bound by $\|\overline{v} - v\| \leq \epsilon/\lambda$ that

$$\begin{aligned}\|\nabla\Phi(\overline{v})\|_2^2 &\leq \sum_{i=1}^{n}(\lambda(exp(\lambda\overline{v}_i) + \exp(-\lambda\overline{v}_i)))^2 \\ &\leq \sum_{i=1}^{n}(\lambda\exp(\epsilon)\cdot(exp(\lambda v_i) + \exp(-\lambda v_i)))^2 \\ &= \exp(\epsilon)\|\nabla\Phi(\overline{v})\|_2^2\end{aligned}$$

So $\|\nabla\Phi(\overline{v})\|_2 \leq \exp(\epsilon/2)\|\nabla\Phi(v)\|_2$ and thus

$$-\gamma\frac{\|\nabla\Phi(v)\|_2^2}{\|\nabla\Phi(\overline{v})\|_2} \geq -\gamma\exp(-\epsilon/2)\|\nabla\Phi(v)\|_2.$$

Next we bound $w^\top\nabla\Phi(v)$ via

$$\begin{aligned}w^\top\nabla\Phi(v) &\leq \|w\|_2\|\nabla\Phi(v)\|_2 \\ &\leq (\epsilon\gamma\cdot\frac{\|\nabla\Phi_\lambda(v)\|_2 + \sqrt{n}\lambda}{\|\nabla\Phi(\overline{v})\|_2} + O(\epsilon\gamma + \gamma^2))\cdot\|\nabla\Phi(v)\|_2 \\ &\leq (\epsilon\gamma\cdot(\exp(\epsilon/2) + \frac{\sqrt{n}\lambda}{\|\nabla\Phi(\overline{v})\|_2}) + O(\epsilon\gamma + \gamma^2))\cdot\|\nabla\Phi(v)\|_2 \\ &\leq O(\epsilon\gamma + \gamma^2)\cdot\|\nabla\Phi(v)\|_2 + O(\epsilon\gamma\sqrt{n}\lambda)\end{aligned}$$

And at last we bound

$$\begin{aligned}\|\delta\|_{\nabla^2\Phi(v)}^2 &= \sum_{i=1}^{n}(\delta_i)^2(\lambda^2\cosh(\lambda v_i)) \\ &\leq \|\delta\|_4^2\lambda^2\left(\sum_{i=1}^{n}\cosh(\lambda v_i)^2\right)^{1/2} \\ &\leq O(\gamma^2)\lambda^2\left(\sum_{i=1}^{n}1 + \sinh(\lambda v_i)^2\right)^{1/2} \\ &\leq O(\gamma^2)\lambda^2(\sqrt{n} + \|\nabla\Phi(v)\|_2)\end{aligned}$$

In summary, we can write

$$\begin{aligned}&\Phi(\frac{(x+\delta_x)(s+\delta_s) - t}{t}) \\ &\leq \Phi(v) - \gamma\frac{\|\nabla\Phi(v)\|_2^2}{\|\nabla\Phi(\overline{v})\|_2} + w^\top\nabla\Phi(v) + 4\|\delta\|_{\nabla^2\Phi(v)}^2 \\ &\leq \Phi(v) - \gamma\exp(-\epsilon/2)\|\nabla^2\Phi(v)\|_2 + O(\epsilon\gamma + \gamma^2)\cdot\|\nabla^2\Phi(v)\|_2 + O(\epsilon\gamma\sqrt{n}\lambda) \\ &\quad + O(\gamma^2\lambda^2)\cdot\|\nabla^2\Phi(v)\|_2 + O(\gamma^2\lambda^2\sqrt{n}) \\ &= \Phi(v) + (-\gamma\exp(-\epsilon/2) + O(\epsilon\gamma + \gamma^2\lambda^2))\cdot\|\nabla^2\Phi(v)\|_2 + O((\epsilon\gamma\lambda + \gamma^2\lambda^2)\sqrt{n})\end{aligned}$$

Here we can bound

$$\begin{aligned}
\|\nabla\Phi(v)\|_2 &= (\sum_{i=1}^n \lambda^2(\sinh(\lambda v_i))^2)^{1/2} \\
&= (\sum_{i=1}^n \lambda^2(\cosh(\lambda v_i)^2 - 1))^{1/2} \\
&\ge \frac{1}{\sqrt{n}} \sum_{i=1}^n \lambda\sqrt{\cosh(\lambda v_i)^2 - 1} \\
&\ge \frac{1}{\sqrt{n}} \sum_{i=1}^n \lambda(\cosh(\lambda v_i) - 1) \\
&= -\lambda\sqrt{n} + \frac{\lambda}{\sqrt{n}}\Phi(v)
\end{aligned}$$

So

$$\begin{aligned}
\Phi(\frac{(x+\delta_x)(s+\delta_s)-t}{t}) &\le \Phi(v) + (-\gamma\exp(-\epsilon/2) + O(\epsilon\gamma + \gamma^2\lambda^2)) \cdot (-\lambda\sqrt{n} + \frac{\lambda}{\sqrt{n}}\Phi(v)) \\
&\quad + O((\epsilon\gamma\lambda + \gamma^2\lambda^2)\sqrt{n}) \\
&\le (1 - \frac{\lambda}{\sqrt{n}}(\gamma\exp(-\epsilon/2) + O(\epsilon\gamma + \gamma^2\lambda^2)))\Phi(v) + O(\gamma\lambda\sqrt{n})
\end{aligned}$$

□

5.3 Vector Maintenance

In the previous section we showed that the linear program solver works, even if we replace all x, s by some approximate $\overline{x}, \overline{s}$. That is, in each iteration it suffices to compute some

$$\delta_s = -\mathbf{A}(\mathbf{A}^\top \overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}\mathbf{A}^\top \overline{\mathbf{S}}^{-1} \frac{t\gamma \cdot \nabla(\Phi(\overline{v}))}{\|\nabla\Phi(\overline{v})\|_2}.$$

Here the vector $\overline{\mathbf{S}}^{-1}\nabla(\Phi(\overline{v}))$ changes in only a single entry whenever we change an entry of $\overline{s}$ or $\overline{v}$. The vector $\overline{v}$ is such that $\|\overline{v} - (xs-t)/t\|_\infty \le \epsilon$, in other words, it suffices to pick $\overline{v} := ((\overline{xs} - \overline{t})/\overline{t})$ for some $\overline{x} \approx_{\epsilon/10} x, \overline{s} \approx_{\epsilon/10} s, \overline{t} \approx_{\epsilon/10} t$. The total number of changed entries in $\overline{\mathbf{S}}^{-1}\nabla(\Phi(\overline{v}))$ can thus be bounded by $\widetilde{O}(n\log(n/\delta))$ over all iterations of the algorithm (Lemma 4.4.1). So maintaining the product $\mathbf{A}^\top\overline{\mathbf{S}}^{-1}\frac{t\gamma\cdot\nabla(\Phi(\overline{v}))}{\|\nabla\Phi(\overline{v})\|_2}$ takes only $\widetilde{O}(nd\log(n/\delta))$ total time over all iterations combined. We already argued in Section 4.4 that maintaining $(\mathbf{A}^\top\overline{\mathbf{X}\mathbf{S}}^{-1}\mathbf{A})^{-1}$ can be done in $O(nd^2\log(n/\delta))$ total time. The last product with $-\mathbf{A}$ takes $O(nd)$ per iteration for a total cost of $O(n^{1.5}d\log(n/\delta))$ over all iterations. Today we want to outline how we can remove this last complexity term, so that the total time of our linear program solver becomes just $\widetilde{O}(nd^2\log(n/\delta))$.

Outline Let x^ℓ, s^ℓ be the value of x and s during the ℓ-th iteration of our algorithm and let $\overline{x}^\ell \approx_\epsilon x^\ell$, $\overline{s}^\ell \approx_\epsilon s^\ell$. Let g^ℓ be the vector $\nabla\Phi_\lambda(\overline{x}^\ell\overline{s}^\ell - \overline{t}^\ell/\overline{t}^\ell)$ from (5.1) during the ℓ-th iteration, and let $\beta^\ell = t\gamma/\|g^\ell\|_2$ be the respective normalization of the gradient. Based on (5.1), we have

$$s^t = s^0 + \sum_{\ell=1}^{t-1} \mathbf{A}(\mathbf{A}^\top \overline{\mathbf{X}}^\ell (\overline{\mathbf{S}}^\ell)^{-1}\mathbf{A})^{-1}\mathbf{A}^\top(\overline{\mathbf{S}}^\ell)^{-1} g^\ell \beta^\ell \tag{5.2}$$

$$x^t = x^0 + \sum_{\ell=1}^{t-1} (\overline{\mathbf{S}}^\ell)^{-1} g^\ell - \overline{\mathbf{X}}^\ell (\overline{\mathbf{S}}^\ell)^{-1}\mathbf{A}(\mathbf{A}^\top \overline{\mathbf{X}}^\ell (\overline{\mathbf{S}}^\ell)^{-1}\mathbf{A})^{-1}\mathbf{A}^\top(\overline{\mathbf{S}}^\ell)^{-1} g^\ell \beta^\ell$$

Note that x^t and s^t do not need to be computed as we only require $\overline{x}^t \approx_\epsilon x^t$, $\overline{s}^t \approx_\epsilon s^t$ to compute above expression. Only at the very end of the algorithm, at some iteration $T = \widetilde{O}(\sqrt{n})$, do we need to compute x^T in order to obtain the solution of the linear program. Thus, it suffice to represent x^t and s^t in some implicit form[2] and to only compute/maintain $\overline{x}^t$ and $\overline{s}^t$ explicitly[3] in each iteration.

Maintaining approx $\overline{x}$ and $\overline{s}$ The following is a data structure framework of [vdBLSS20] to maintain these explicit approximate vectors $\overline{x}^t$ and $\overline{s}^t$. In [vdBLSS20], maintaining $\overline{x}^t, \overline{s}^t$ explicitly is done via three data structure tasks:

(i) Maintain the vector $\mathbf{A}^\top(\overline{\mathbf{S}}^t)^{-1} g^t\beta^t$.

(ii) Maintain the inverse $(\mathbf{A}^\top\overline{\mathbf{X}}^t(\overline{\mathbf{S}}^t)^{-1}\mathbf{A})^{-1}$.

(iii) Maintain $\overline{s}^t \approx_\epsilon s^0 + \mathbf{A}\sum_{\ell=1}^{t-1} h^\ell$ and $\overline{x}^i \approx_\epsilon x^0 + (\sum_{\ell=1}^{t-1}(\overline{\mathbf{S}}^i)^{-1} g^\ell\beta^\ell) - \sum_{\ell=1}^{t-1}\overline{\mathbf{X}}^\ell(\overline{\mathbf{S}}^\ell)^{-1}\mathbf{A}h^\ell$ for some sequence of vectors $(h^\ell)_{\ell\geq 1}$.

Note that for $h^\ell = (\mathbf{A}^\top\overline{\mathbf{X}}^\ell(\overline{\mathbf{S}}^\ell)^{-1}\mathbf{A})^{-1}\mathbf{A}^\top(\overline{\mathbf{S}}^\ell)^{-1} g^\ell\beta^\ell$, the vectors $\overline{x}^t, \overline{s}^t$ are exactly the approximations of x^t, s^t in (5.2). Further, vector h^ℓ can be computed by using data structure (i) and (ii).

A data structure for (ii) which runs in $\widetilde{O}(nd^2)$ total time was already discussed in Section 4.4. Computing (i) can be done in $\widetilde{O}(nd)$ total time, because whenever an entry of $\overline{v}$ or $\overline{s}$ changes, only one entry of $\overline{\mathbf{S}}^{-1}g$ will change. In total there will be $\widetilde{O}(n)$ such entry changes (as argued in Section 4.4). When an entry changes, we only need to spend $O(d)$ time to maintain $\mathbf{A}^\top\overline{\mathbf{S}}^{-1}g$.

We are left with constructing a data structure for (iii).

Data structure for (iii) We outline the maintenance of $\overline{s}^t$ as that is the easier case compared to $\overline{x}$. Note that we can easily maintain s^t in an implicit from via

$$s^t = s^0 + \mathbf{A}\sum_{\ell=1}^{t-1} h^\ell, \tag{5.3}$$

[2] Maintaining some x^t implicitly means we do not write down the vector x^t in memory. Instead, we construct a data structure with some Query(i) operation that returns the ith entry x_i^t.

[3] "Explicit" means we write down the vector in memory.

by just maintaining the sum of h^ℓ. Here the vector h^ℓ is obtained via data structures (i) and (ii). Note that h^ℓ is just a d-dimensional vector, so maintaining the sum of these vectors only incurs an additional $O(d)$ time cost per iteration.

To maintain $\overline{s}^t$ explicitly, we follow the heavy hitter approach used in [vdBLSS20, vdBLN+20]. The task is to find a set of indices $J \subset [n]$ containing all j where s_j^t and s_j^{t-1} differ by some $\Omega(s_j^{t-1}\epsilon)$. The idea is that for those $j \in J$, our old $\overline{s}_j^{t-1} \approx_\epsilon s_j^{t-1}$ will no longer be a valid approximation of the new s_j^t, so those are the entries where we must update our approximation. Once we discovered these indices, we update $\overline{s}_j^t \leftarrow s_j^t$ for all $j \in J$, where any s_j^t can be computed in $O(d)$ time by (5.3). For $j \notin J$ we use the old value of $\overline{s}_j^{t-1}$ for $\overline{s}_j^t$.

Remark: this approach only detects large changes of s^t within a single iteration. We can also detect slower changes that occur over several iteration by checking every 2^k iterations if there are entries where s^t and s^{t-2^k} differ a lot. This is done for every $k = 0, ..., \log\sqrt{n}$ iterations. For simplicity, we for now focus only the case $k = 0$.

At last, I would like to point out that we can not even afford to write down $\overline{s}$ in each iteration. It is an n-dimensional vector and we have $\widetilde{O}(\sqrt{n})$ iterations, so just writing down this vector in each iteration would take $\widetilde{O}(n^{1.5})$ total time which is larger than nd when $n \gg d$. So instead, we write down $\overline{s}$ once at the start of the algorithm, and then just store the pointer to that location in memory. Then, throughout the algorithm, we only overwrite values of entries $\overline{s}_i$ in memory whenever we detect that these entries must be changes to maintain the valid approximation $\overline{s} \approx s$.

5.3.1 Heavy Hitter

As outlined above, we want to detect indices i where $|s_i^{\ell-1} - s_i^\ell| > \epsilon\overline{s}_i$, or equivalently, $|(\overline{\mathbf{S}}^{-1}\mathbf{A}h)_i| > \epsilon$. For this purpose, we want to construct the following data structure:

Theorem 5.3.1. *There exists a data structure with the following operations:*

- INIT($\mathbf{A} \in \mathbb{R}^{n\times d}, \mathbf{W} \in \mathbb{R}_{\geq 0}^{n\times n}$) *Initialize in $\widetilde{O}(nd)$ time (we assume $\mathbf{W}$ is a diagonal matrix).*
- UPDATE($i \in \{1, ..., n\}, v \in \mathbb{R}_{>0}$) *Set $\mathbf{W}_{i,i} \leftarrow v$ in $\widetilde{O}(d)$ time.*
- QUERY($h \in \mathbb{R}^h, \epsilon > 0$) *Return all indices i with $|(\mathbf{WA}h)_i| > \epsilon$ in $\widetilde{O}(d\|\mathbf{WA}h\|_2^2/\epsilon^2 + d)$ time.*

Note that there are up to $O(\|\mathbf{WA}h\|_2^2/\epsilon^2)$ indices i with $|(\mathbf{WA}h)_i| > \epsilon$ that we may have to return.

To construct Theorem 5.3.1, we will use the following lemma:

Lemma 5.3.2 (Johnson-Lindenstrauss). *There exists a random matrix $\mathbf{R} \in \mathbb{R}^{k\times n}$ for $k = O(\epsilon^{-2}\log n)$ such that the following holds: For any $v \in \mathbb{R}^n$ we have*

$$\mathbb{P}\left[(1-\epsilon)\|v\|_2^2 \leq \|\mathbf{R}v\|_2^2 \leq (1+\epsilon)\|v\|_2^2\right] \geq 1 - 1/poly(n).$$

In other words, w.h.p. we can approximate $\|v\|_2^2$ by $\|\mathbf{R}v\|_2^2$.

Proof sketch. Let each entry of $\mathbf{R}$ be randomly $+1, -1$. Then

$$\begin{aligned}\mathbb{E}[\|\mathbf{R}v\|_2^2] &= \mathbb{E}[\sum_{i=1}^{k}(\sum_{j=1}^{n}\mathbf{R}_{i,j}v_j)^2] \\ &= \mathbb{E}[\sum_{i=1}^{k}(\sum_{j=1}^{n}(\mathbf{R}_{i,j}v_j)^2) + \sum_{j\neq k}\mathbf{R}_{i,j}\mathbf{R}_{i,k}v_jv_k] \\ &= \sum_{i=1}^{k}(\sum_{j=1}^{n}\underbrace{\mathbb{E}[\mathbf{R}_{i,j}^2]}_{=1}v_j^2) + \sum_{j\neq k}\underbrace{\mathbb{E}[\mathbf{R}_{i,j}\mathbf{R}_{i,k}]}_{=0}v_jv_k \\ &= \sum_{i=1}^{k}(\sum_{j=1}^{n}v_j^2) = k\|v\|_2^2\end{aligned}$$

So by scaling $\mathbf{R}$ by $1/\sqrt{k}$ the expectation is exactly the norm $\|v\|_2^2$. For $k = O(\epsilon^{-2}\log n)$ we are w.h.p. close to the expectation by Chernoff-bound. □

Proof of Theorem 5.3.1. Assume we computed $\mathbf{RWA}$ during the preprocessing for $\mathbf{R}$ from Lemma 5.3.2. This is just a $k \times d$ matrix for $k = O(\log n)$ and then, for any vector h, we can quickly get a 1.1-approximation of $\|\mathbf{A}h\|_2^2$ in $\widetilde{O}(d)$ time by simply computing $\|\mathbf{RWA}h\|_2^2$. If this norm is small (let's say smaller than $\epsilon/2$, then we know there can be no index i with $|(\mathbf{WA}h)_i| > \epsilon$. If it is larger, however, then we might have such an index.

The idea is now to binary-search for this larger entry. Let $\mathbf{B}^{i,\ell}$ be the rows with index in $\{i2^\ell + 1, ..., i2^\ell + 2^\ell\}$ of $\mathbf{WA}$. So for example:
$\mathbf{B}^{i,0}$ is just the $(i-1)$-th row of $\mathbf{WA}$,
$\mathbf{B}^{0,\ell}$ for $\ell = \log n$ is the entire matrix $\mathbf{WA}$,
and $\mathbf{B}^{0,\ell-1}$ is the top half of $\mathbf{WA}$.

Init During initialization we compute $\mathbf{R}^{i,\ell}\mathbf{B}^{i,\ell}$ for all i, ℓ and independent random matrices $\mathbf{R}^{i,\ell}$ from Lemma 5.3.2. Here $\mathbf{R}^{i,\ell}\mathbf{B}^{i,\ell}$ has only $O(\log n)$ rows and computing all these matrices takes $\widetilde{O}(nd)$ time.

Query We start with $i = 0$, $\ell = \log n$ and compute $\|\mathbf{R}^{i,\ell}\mathbf{B}^{i,\ell}h\|_2$ in $\widetilde{O}(d)$ time. If the norm is larger than $\epsilon/2$, then there might be some index j with $|(\mathbf{B}^{i,\ell}h)_i| > \epsilon$. So we recurse by computing the norms $\|\mathbf{R}^{2i,\ell-1}\mathbf{B}^{2i,\ell-1}h\|_2$ and $\|\mathbf{R}^{2i+1,\ell-1}\mathbf{B}^{2i+1,\ell-1}h\|_2$. These are estimates of $\|\mathbf{B}^{2i,\ell-1}h\|_2$ and $\|\mathbf{B}^{2i+1,\ell-1}h\|_2$, i.e. the top and bottom half of $\mathbf{B}^{i,\ell}h$.

So after recursing often enough, we find all indices j where $\|\mathbf{B}^{j,0}h\|_2 = |(\mathbf{WA}h)_j| > \epsilon$.

The total time is bounded by

$$\sum_{\ell=0}^{\log n}\sum_{i=0}^{n/2^\ell}\widetilde{O}(d)\cdot\underbrace{\|\mathbf{B}^{i,\ell}h\|_2^2/\epsilon^2}_{\text{recursion happens if this} \geq 1} = \sum_{\ell=0}^{\log n}\widetilde{O}(d\cdot\|\mathbf{WA}h\|_2^2) = \widetilde{O}(d\|\mathbf{WA}h\|_2^2)$$

and an extra $+\widetilde{O}(d)$ because we compute at least one such matrix-vector product.

Update The query only works, assuming we have $\mathbf{R}^{i,\ell}\mathbf{B}^{i,\ell}$ precomputed. When an entry $\mathbf{W}_{j,j}$ changes, we must update these matrices. This takes only $\widetilde{O}(d)$ time, as for each ℓ a change to $\mathbf{W}_{j,j}$ affects $\mathbf{R}^{i,\ell}\mathbf{B}^{i,\ell}$ for only a single i. So we must update only $O(\log n)$ many $\mathbf{R}^{i,\ell}\mathbf{B}^{i,\ell}$. Since $\mathbf{R}^{i,\ell}$ has only $O(\log n)$ rows, updating one such matrix takes only $O(d \log n)$ time.

□

5.3.2 Maintaining Approximate Vectors

In this section we prove the data structure that maintains the iterates x, s and their approximations $\overline{x}, \overline{s}$ throughout the interior point method. The main result is the following Theorem 5.3.3.

Theorem 5.3.3. *There is a data structure with operations*

- $\textsc{Init}(\mathbf{A} \in \mathbb{R}^{n\times d}, x^{(0)}, s^{(0)}, g, \epsilon > 0)$ *Initialize in* $\widetilde{O}(nd)$ *time and set* $T = 0$*.*
- $\textsc{Update}(i, c)$ *Set* $g_i \leftarrow c$ *in* $\widetilde{O}(d)$ *time.*
- $\textsc{Add}(h \in \mathbb{R}^d, \beta \in \mathbb{R}_{>0}, I \subset \{1, ..., n\}, \delta_g \in \mathbb{R}^I)$ *Set* $T \leftarrow T + 1$ *and store* $h^{(T)} = h$*,* $\beta^{(T)} = \beta$*. Let* $g^{(T)} \leftarrow g^{(T-1)} + \delta_g$*. Then return*

$$\overline{x}^{(T)} \approx x^{(T)} =: x^{(0)} + \sum_{\ell=1}^{T} (\overline{\mathbf{S}}^{(\ell)})^{-1} \beta^{(\ell)} g^{(\ell)} - \overline{\mathbf{X}}^{(\ell)} (\overline{\mathbf{S}}^{(\ell)})^{-1} \mathbf{A} h^{(\ell)}$$

$$\overline{s}^{(T)} =: s^{(T)} \approx s^{(0)} + \mathbf{A} \sum_{\ell=1}^{T} h^{(\ell)}.$$

The vectors are returned in sparse representation, i.e. a pointer to $\overline{x}^{(T)}, \overline{s}^{(T)}$ *and a set* $I \subset [n]$ *where the entries differ compared to the previous output. For the first* $\sqrt{n}$ *calls, a call to* $\textsc{Add}$ *takes* $\widetilde{O}(d|I| + \sqrt{n}d)$ *amortized time if* $\|(s^{(\ell+1)} - s^{(\ell)})/s^{(\ell)}\|_2, \|(x^{(\ell+1)} - x^{(\ell)})/x^{(\ell)}\|_2 = O(1)$*.*

Here the vector h^ℓ is just $(\mathbf{A}^\top \overline{\mathbf{X}}^\ell (\overline{\mathbf{S}}^\ell)^{-1} \mathbf{A})^{-1} \overline{\mathbf{S}}^{-1} \beta^{(\ell)} g^{(\ell)}$ which is computed via (i) and (ii). Further, $g^{(\ell)} = \nabla\Phi((\overline{xs} - \overline{t})/\overline{t})$ and $\beta^{(\ell)} = \gamma t / \|\nabla\Phi((\overline{xs} - \overline{t})/\overline{t})\|_2$. So in summary, the vectors x, s and $\overline{x}, \overline{s}$ are exactly as required for our linear program solver.

The set of indices I (returned by $\textsc{Add}$) is required because those are the indices where we will have to recompute $(\nabla\Phi((\overline{xs} - \overline{t})/\overline{t}))_i$.

The proof of Theorem 5.3.3 is based on three other data structures. The first two data structures maintain x and s in implicit form, such that we can query any entry of these vectors efficiently. The third data structure detects which entries of x and s changed by some $(1 \pm O(\epsilon))$-factor from one iteration to the next. The idea is that if an entry x_i changed a lot for some i, then we set $\overline{x}_i \leftarrow x_i$. Since other entries x_j did not change by some $(1 \pm O(\epsilon))$-factor, the old value of $\overline{x}_j$ is still a valid approximation. This way we can maintain $\overline{x}$ in sublinear time, because we only change a few entries per iteration.

Note that an entry x_i might also change by some $(1 \pm \epsilon)$-factor over a longer time interval but only a little in each iteration. To maintain $\overline{x}$ with $\overline{x}_i \approx_\epsilon x_i$ for these slowly changing entries, we do the following: Every 2^ℓ iterations, we detect all entries i where x_i changed by

some $(1 \pm O(\epsilon/\log n))$-factor over the past 2^ℓ iterations. This is done for all $0 \le \ell \le \log \sqrt{n}$. If an entry changed sufficiently, we set $\overline{x}_i \leftarrow x_i$. This way we make sure that $\overline{x}_i$ is a $(1 \pm \epsilon)$-approximation of x_i, because x_i can never change too much without our data structure updating $\overline{x}_i$.

Implicit Maintenance

The following two Lemmas 5.3.4 and 5.3.5 allow us to maintain x and s implicitly, such that we can query any entry in $O(d)$ time.

Lemma 5.3.4 (Implicit Slack)**.** *There is a deterministic data structure with operations*

- INIT($\mathbf{A} \in \mathbb{R}^{n\times d}, s^{(0)} \in \mathbb{R}^n$) *Initialize in $O(nd)$ time and set $t = 0$.*
- UPDATE($h \in \mathbb{R}^d$) *Increase t and store $h^{(t)} = h$ in $O(d)$ time.*
- QUERY(i) *Return $s_i^{(t)} := (s^{(0)} + \sum_{\ell=1}^t \mathbf{A}h^{(t)})_i$ in $O(d)$ time.*

Proof. Maintain $v := \sum_{\ell=0}^t h^{(\ell)}$ in $O(d)$ time during each update. During a query return $(\mathbf{A}v)_i$ in $O(d)$ time. □

Lemma 5.3.5 (Implicit Primal)**.** *There is a deterministic data structure with operations*

- INIT($\mathbf{A} \in \mathbb{R}^{n\times d}, x^{(0)}, g^{(0)}, \overline{x}^{(0)}, \overline{s}^{(0)} \in \mathbb{R}^n$) *Initialize in $O(nd)$ time and set $t = 0$.*
- UPDATE($I \subset [n], \delta_g, \delta_{\overline{x}}, \delta_{\overline{s}} \in \mathbb{R}^I, h \in \mathbb{R}^d, \beta \in \mathbb{R}$) *Increase t and store $g^{(t)} = g^{(t-1)} + \delta_g$ (and likewise $\overline{x}^{(t)} = \overline{x}^{(t)} + \delta_{\overline{x}}$ and $\overline{s}^{(t)} = \overline{s}^{(t-1)} + \delta_{\overline{s}}$) and $h^{(t)} = h$, $\beta^{(t)} = \beta$ in $O(d|I|)$ time.*
- QUERY(i) *Return $x_i^{(t)} := (x^{(0)} + \sum_{\ell=1}^t \beta^{(t)} \cdot \overline{\mathbf{S}}^{-1} g^{(t)} - \overline{\mathbf{X}}^{(t)}(\overline{\mathbf{S}}^{(t)})^{-1}\mathbf{A}h^{(t)})_i$ in $O(d)$ time.*

Proof. We maintain a vector $v \in \mathbb{R}^n$, $z \in \mathbb{N}^n$ and vectors $(u^{(t)})_{t\ge 0} \in \mathbb{R}^d$ with the invariants

$$v_i = x_i^{(z_i)} \text{ for all } i \in [n], \quad u^{(t)} = \sum_{\ell=1}^t h^{(\ell)} \text{ for all } t, \quad b^{(t)} = \sum_{\ell=1}^t \beta^{(\ell)} \text{ for all } t.$$

During the initialization we set $v = x^{(0)}$, $b^{(0)} = 0$ and $\overline{x}, \overline{s}, \overline{w}, z, u$ to be zero vectors, thus the invariants hold true.

During an update we increase t and update v_i and z_i for all $i \in I$ such that the invariant holds:

$$\begin{aligned}
u^{(t)} &\leftarrow u^{(t-1)} + h \\
b^{(t)} &\leftarrow b^{(t-1)} + \beta \\
v_i &\leftarrow v_i + (b^{(t)} - b^{(z_i)}) g_i/\overline{s}_i + \beta(\delta_g)_i/(\overline{s}_i + \delta_{\overline{s}})_i \\
&\quad - \overline{x}_i/\overline{s}_i \mathbf{A}(u^{(t-1)} - u^{(z_i+1)}) - (\overline{x} + \delta_{\overline{x}})_i/(\overline{s} + \delta_{\overline{s}})_i \mathbf{A}h \\
z_i &\leftarrow t
\end{aligned}$$

Note that the invariants hold because $\overline{x}_i$, $\overline{s}_i$ and w_i stayed the same since the last time we updated v_i. At last, we set $\overline{x} \leftarrow \overline{x} + \delta_{\overline{x}}$, $\overline{s} \leftarrow \overline{s} + \delta_{\overline{s}}$, $g \leftarrow g + \delta_g$. Updating v_i this way takes $O(d)$ time and updating $u^{(t)}$ takes $O(d)$ time.

For a query we perform an empty update for $I = \{i\}$ and return v_i. □

Algorithm 1: Primal Dual Vector Maintenance.

```
1  parameters
2      t = 0, x̄ = x^(0), s̄ = s^(0).
3  procedure ADD(h ∈ ℝ^d, β ∈ ℝ_{>0}, I ⊂ {1,...n}, δ_g ∈ ℝ^n)
4      t ← t + 1, g ← g + δ_g
5      F^k ← F^k ∪ I for all k = 0, ..., log √n
6      Update implicit x^(t) and s^(t) via Lemmas 5.3.4 and 5.3.5.
7      for k ≤ log √n, if 2^k divides t do
8          Let w_i = 1/s̄_i for i ∉ F^k, w_i = 0 for i ∈ F^k ;  // i ∈ F^k are indices where
            x̄_i, s̄_i, g_i changed during the past 2^k iterations.
           // We now search for indices i ∉ F^k where we must update x̄_i or
              s̄_i.
9          I_k ← set of i with |(WA Σ_{ℓ=2^k+1}^t h^(ℓ))_i| > ε/10.
10         I_k ← I_k ∪ set of i ⊂ [n] \ F^k with |1/(s̄_i x̄_i) · g_i · Σ_{ℓ=t−2^k+1}^t β^(ℓ)| > ε/10.
11         J_k ← I_k ∪ F^k ;            // Set of indices that we want to update.
12         Set x̄_i and s̄_i to x_i^(t) and s_i^(t) for i ∈ J_k.
13         F^{k+1} ← F^{k+1} ∪ J_k, F^k ← ∅ ;      // Update set F^{k+1} so it contains
            indices where x̄_i and s̄_i have changed over past 2^k iterations.
14     return x̄, s̄
```

Proof of Theorem 5.3.3

We now have all tools to prove Theorem 5.3.3. As outlined before, the proof is based on detecting every 2^ℓ iterations (for all $0 \le \ell \le \log \sqrt{n}$) all entries i where x_i or s_i changed a lot over the past 2^ℓ iterations.

The algorithm is given by Algorithm 1. We first prove correctness, i.e. that $\overline{x}, \overline{s}$ returned by the tth call to update satisfies $\overline{x} \approx_\epsilon x^{(t)}$ and $\overline{s} \approx_\epsilon s^{(t)}$.

Lemma 5.3.6. *The output $\overline{x}, \overline{s} \in \mathbb{R}^n$ returned by the tth call to* ADD *satisfies $\overline{x} \approx x^{(t)}$, $\overline{s} \approx s^{(t)}$.*

Proof. We maintain $x^{(t)}$ and $s^{(t)}$ implicitly via Lemmas 5.3.4 and 5.3.5. We now argue that we set $\overline{s}_i \leftarrow s_i^{(t)}$ before $\overline{s}_i \not\approx_\epsilon s_i^{(t)}$ can occur (and likewise for $\overline{x}, x^{(t)}$). We start with the analysis of $\overline{s}$ and then cover $\overline{x}$ afterward.

Correctness of $\overline{s}$ Consider a loop of Line 7 for some k. Note that set I_k contains all $i \notin F^k$ where

$$|s_i^{(t)} - s_i^{(t-2^k)}| = |(\mathbf{A} \sum_{\ell=2^k+1}^{t} h^{(\ell)})_i| > \epsilon \overline{s}_i \epsilon/(10 \log n). \tag{5.4}$$

As $J_k = I_k \cup F^k$, the set J_k contains all $i \in [n]$ that satisfy (5.4). The algorithms sets $\overline{s}_i \leftarrow s_i^{(t)}$ for all $i \in J_k$, so for these indices i the approximation guarantee $\overline{s}_i \approx_\epsilon s_i^{(t)}$ holds.

Now consider $i \notin J_k$ and let $t' < t$ be the last time we set $\bar{s}_i \leftarrow s_i^{(t')}$. Then there is a sequence of length at most $\log(t'-t)$ many $t_1 < t_2 < ... < t_p$ where $t_j - t_{j-1}$ is a power of two and $t_1 = t'$, $t_p = t$. More accurately, these t_j are the time steps during which we previously had that $|s^{(t_j)} - s^{(t_{j-1})}| \le \epsilon \bar{s}_i \epsilon/(10 \log n)$ (as otherwise we would have set $\bar{s}_i \leftarrow s_i^{(t_j)}$). Thus by triangle inequality we have

$$|s_i^{(t)} - \bar{s}_i| = |s^{(t)} - s_i^{(t')}| < \log(t-t')\bar{s}_i \epsilon/(10 \log n).$$

For $t < \sqrt{n}$, this implies $\bar{s} \approx_\epsilon s^{(t)}$.

Correctness of $\bar{x}$ The correctness proof for $\bar{x}$ is almost the same to the previous proof for $\bar{s}$. We argue that an iteration of Line 7 for some k sets $\bar{x}_i \leftarrow x_i^{(t)}$ if $|x_i^{(t)} - x_i^{(t-2^k)}| > \epsilon \bar{x}_i/(5 \log n)$. By the same argument as before, this then implies $\bar{x}_i \approx_\epsilon x_i^{(t)}$ for $t \le \sqrt{n}$.

By definition of $x^{(t)}$ and $\beta^{(t)}$ we have for all i where $\bar{x}_i$ and $\bar{s}_i$ stayed the same during the past $2^k - 1$ iterations (i.e. all $i \notin F^k$) that

$$x_i^{(t)} - x_i^{(t-2^k)} = \frac{1}{\bar{x}_i} g_i \sum_{\ell=t-2^k+1}^{t} \beta^{(t)} - \bar{x}_i/\bar{s}_i (\mathbf{A} \sum_{\ell=t-2^k+1}^{t} h^{(\ell)})_i. \tag{5.5}$$

So indices $i \notin F^k$ with $|x_i^{(t)} - x_i^{(t-2^k)}| \ge \bar{x}_i \epsilon/(5 \log n)$ can be found by detecting indices $i \notin F^k$ with (a) $|(\mathbf{A} \sum_{\ell=t-2^k+1}^{t} h^{(\ell)})_i| > \bar{s}_i \epsilon/(10 \log n)$ or (b) $|g_i \sum_{\ell=t-2^k+1}^{t} \alpha^{(t)}| > \bar{x}_i \epsilon/(10 \log n)$. These are precisely the two checks performed in Lines 9 and 10.

□

Lemma 5.3.7. *Consider an execution of Line 7 in Algorithm 1 for some k. Let $I_k \subset [n]$ be the sets after line 9 and 10. Then $|I_k| = O(\frac{2^{2k}}{\epsilon^2})$.*

Proof. Note that $I_k \cap F^k = \varnothing$ and for $i \notin F^k$, the value of $\bar{x}_i$ and $\bar{s}_i$ stayed the same over the past $2^k - 1$ iterations and they were good approximations for $x^{(\ell)}$, $s^{(\ell)}$ for $t - 2^k \le \ell \le t-1$, so we can write

$$\begin{aligned}
|I_k| &= \sum_{i \notin F^k} \mathbf{1}_{i \in I_k} \\
&\le \sum_{i \notin F^k} \left(\frac{(\mathbf{WA} \sum_{\ell=t-2^k}^{t} h^{(\ell)})_i^2}{\epsilon^2} + \frac{(g_i)^2}{(\bar{x}_i \bar{s}_i)^2 \epsilon^2} \left(\sum_{\ell=t-2^k+1}^{t} \beta^{(\ell)} \right)^{-2} \right) \\
&= \sum_{i \notin F^k} \left(\frac{(s^{(t)} - s^{(t-2^k+1)})_i^2}{\bar{s}_i \epsilon^2} + \left(\frac{s_i^{(t)} - s_i^{(t-2^k+1)}}{\bar{s}_i \epsilon} + \frac{x_i^{(t)} - x_i^{(t-2^k+1)}}{\bar{x}_i \epsilon} \right)^2 \right) \\
&\le 2^k \sum_{\ell=t-2^k}^{t-1} (\|(s^{(\ell+1)} - s^{(\ell)})/s^{(\ell)}\|_2^2 + \|(x^{(\ell+1)} - x^{(\ell)})/x^{(\ell)}\|_2^2)/\epsilon^2 \\
&\le 2^{2k}/\epsilon^2.
\end{aligned}$$

Where for the third line we used the definition of x and s and (5.5). □

Lemma 5.3.8. *Assume we just performed the t-th call to* ADD. *Let k be the largest such that t is a multiple of 2^k. Let T be the total number of indices given over the past 2^k calls to* ADD. *Then most $O(2^{2k}/\epsilon^2)$ entries changed in $\overline{x}$ and $\overline{s}$ after the most recent call to* ADD.

Proof. After 2^k iterations we change the entries $\overline{x}_i$ and $\overline{s}_i$ for $i \in F^k$ and those i detected as large in Lines 9 and 10 (i.e. $i \in I_k$). For any such k, the number of entries in I_k is bounded by $O(2^{2k}/\epsilon^2)$ by Lemma 5.3.7. The indices in F^k are those that were in some $I_{k'}$ for $k' < k$, or indices that were given to ADD as input parameter over the past 2^k calls to ADD. So the size of F^k is at most $F^k = O(T + \sum_{k'<k} 2^{2k'}/\epsilon^2) = O(T + 2^{2k}/\epsilon^2)$. □

Lemma 5.3.9. *For the first $\sqrt{n}$ calls to* ADD, *the amortized time per call is bounded by* $\widetilde{O}\left(\frac{nd}{\epsilon^2} + |I|d\right)$.

Proof. The cost analysis consists of the following parts: (i) the cost of maintaining implicit x and s, (ii) the cost of updating $\overline{x}_i$ and $\overline{s}_i$ for some $i \in J_k$, (iii) the cost of finding large entries (Lines 9 and 10).

Our algorithm performs work every time an index of $\overline{x}$ or $\overline{s}$ changes, so we want to bound how often this happens. Further, throughout this proof, let T be the sum of sizes of set I given to all calls to ADD.

Number of changes to $\overline{x}$ and $\overline{s}$ By Lemma 5.3.8 the total number of changes to $\overline{x}$ and $\overline{s}$ is bounded by

$$\widetilde{O}(T + \sum_{k=1}^{\log \sqrt{n}} d \cdot 2^{2k} \cdot \sqrt{n}/2^k) = \widetilde{O}(nd).$$

Maintaining implicit x and s We maintain $x^{(t)}$ and $s^{(t)}$ implicitly via Lemmas 5.3.4 and 5.3.5. This takes $O(d)$ per iteration plus $O(d)$ per index $i \in [n]$ where $\overline{x}_i$ and $\overline{s}_i$ are changed and $O(d)$ per index i given as argument to ADD. So over all $\sqrt{n}$ iterations, the time is bounded by $\widetilde{O}(dT + nd)$.

Updating $\overline{x}$ and $\overline{s}$ Updating $\overline{x}_i$ and $\overline{s}_i$ requires to compute x_i and s_i which takes $O(d)$ time. So again, the total time is bounded by $\widetilde{O}(dT + nd)$.

Cost of finding large entries At last, we want to analyze the cost of Line 9 and Line 10. Line 10 can be implemented to take $O(\log n)$ time per returned index, by simply maintaining an ordered binary tree for the entries of the vector. This adds an additional $O(\log n)$ cost per update to $\overline{x}_i$, $\overline{s}_i$, g_i but is subsumed by the cost we computed in the previous paragraphs.

The cost of Line 9 is bounded by Theorem 5.3.1 to takes time

$$\begin{aligned}
&\widetilde{O}(d\|\mathbf{WA} \sum_{\ell=t-2^k+1}^{t} h^{(\ell)}\|_2^2/\epsilon^2) \\
&= \widetilde{O}(d2^k \sum_{\ell=t-2^k}^{t+1} \|(s^{(\ell+1)} - s^{(\ell)})/s^{(\ell)}\|_2^2/\epsilon^2) \\
&= O(d2^{2k}/\epsilon^2).
\end{aligned}$$

As this happens once every 2^k iterations, and have $k \leq \log\sqrt{n}$, we get total cost over $\sqrt{n}$ iterations of

$$\widetilde{O}(d2^{2k}\epsilon^{-1} \cdot \sqrt{n}/2^k) = \widetilde{O}(\sqrt{n}2^k d/\epsilon^2) = \widetilde{O}(nd/\epsilon^2).$$

Note that for Line 9 we must update the vector w. By Theorem 5.3.1, updating an entry i takes time $\widetilde{O}(d)$. We only need to update the entries where $\overline{s}_i$ changed or that were added/removed to/from F^k. This cost can again be bounded by $\widetilde{O}(dT + nd/\epsilon^2)$ in total over all iterations.

In summary, the total time over $\sqrt{n}$ iterations is

$$\widetilde{O}(dT + nd/\epsilon^2).$$

or just $\widetilde{O}(d + \sqrt{n}d)$ per call to ADD. □

Part II

Fast Matrix Multiplication

Chapter 6

Fast Matrix Multiplication

So far, we discussed many algorithms and data structures that rely on multiplying or inverting matrices. We always used that matrix multiplication and inversion of $n \times n$ matrices take $O(n^3)$ operations. However, it is actually possible to multiply matrices in fewer operations. Such algorithms are referred to as "fast matrix multiplication" and the current best upper bound is $O(n^{2.373})$ [AW21, Gal14].

Definition 6.0.1. We define ω as the "matrix multiplication exponent", that is, multiplying two $n \times n$ matrices takes $O(n^\omega)$ operations. The current best bound is 2.373 [AW21][1]. Many researchers believe $\omega = 2$ but so far no one could prove this.

The complexity bound $O(n^\omega)$ also holds for computing (i) inverse, (ii) rank, or (iii) determinant of an $n \times n$ matrix.

This is still an active area of research and it's not known what is the best possible complexity. The algorithms with $O(n^{2.373})$ complexity are rather complicated, so here we will only present a slower $O(n^{2.808})$ upper bound by Strassen [S+69].

Note that throughout, the complexity is measured in the number of arithmetic operations. So depending on the field being used, one might require more time. For instance, if one arithmetic operation needs T time, then the time complexity would be $O(n^\omega \cdot T)$.

Remark The algorithm community generally writes $O(n^\omega)$ for the complexity of multiplying two $n \times n$ matrices. However, the formal definition of ω is actually

$$\omega := \inf\{x \mid \text{two } n \times n \text{ matrices can be multiplied in } n^x \text{ operations.}\}.$$

So technically, multiplying two $n \times n$ matrices actually needs $n^{\omega+o(1)}$ operations, but for simplicity of notation the algorithm community uses $O(n^\omega)$ instead.

6.0.1 Strassen Matrix Multiplication

Here we quickly outline how to $n \times n$ matrices can be multiplied in $O(n^{2.808})$ operations.

[1] Since writing these lecture notes, a new bound $\omega \leq 2.372$ has been published on Arxiv [DWZ22].

Consider the product of two 2×2 matrices $\mathbf{A}$ and $\mathbf{B}$. Using the naive matrix multiplication algorithm of multiplying rows by columns, this would take 8 products and 4 additions:

$$\begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} & \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} & \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{bmatrix}$$

Strassen [S+69] observed, that one can actually perform the computation in just 7 multiplications, by computing

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}) \\
\mathbf{AB} &= \begin{bmatrix} \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 & \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{M}_2 + \mathbf{M}_4 & \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_5 + \mathbf{M}_6 \end{bmatrix}
\end{aligned}$$

This algorithm for multiplying 2×2 matrices can be extended to $n \times n$ matrices (let's assume for simplicity $n = 2^k$ for some k) by using this multiplication algorithm recursively.

For $n \times n$ matrices, $\mathbf{A}_{1,1}$ is the topleft $(n/2) \times (n/2)$ block of $\mathbf{A}$ (and likewise for the other parts of $\mathbf{A}$ and $\mathbf{B}$). We then use the same algorithm recursively when computing the products $\mathbf{M}_1, ..., \mathbf{M}_7$. The total time complexity is

$$T(n) := 7 \cdot T(n/2) + 18(n/2)^2 = O(n^{\log_2(7)}) \leq O(n^{2.808}).$$

Most improvements to fast matrix multiplication come from finding more efficient recursions. For example, Pan [Pan78] showed that a matrix product of 70×70 matrices can be computed using 143640 multiplications, leading to $O(n^{\log_{70}(143640)}) = O(n^{2.79512})$. The current best bounds are $O(n^{2.373})$ by Alman and V.Williams [AW21].

6.0.2 Rectangular Matrix Multiplication

So far we discussed multiplying two square $n \times n$ matrices. We now briefly discuss rectangular matrix multiplication. We will not present how these algorithms work but mention existing state-of-the-art results, as we will use them in later chapters in a black box way. For rectangular matrices, the matrix exponent ω is not just some constant, but actually a function:

Definition 6.0.2. We write $O(n^{\omega(a,b,c)})$ for the complexity of multiplying an $n^a \times n^b$ matrix with an $n^b \times n^c$ matrix.

Where for ease of notation we write just ω for $\omega(1,1,1) \leq 2.373$. It is known that $\omega(1,1,0.319) = 2$ and in general $\omega(a,b,c) \geq a + b + c - \min(a,b,c)$, and $\omega(a,b,c)$ is the same for any order of the arguments, e.g. $\omega(a,b,c) = \omega(b,c,a)$. Upper bounds on $\omega(1,1,k)$ were studied in [GU18] and in general $\omega(1,1,k) < \omega(1,1,1) \leq 2.373$ for $k < 1$ since smaller matrices are involved.

Chapter 7

Dynamic Matrix Inverse

In this chapter we will prove the following data structure:

Theorem 7.0.1 ([San04])**.** *There exists a data structure with the following operations*

- INIT($\mathbf{A} \in \mathbb{F}^{n\times n}$) *Initialize on a given invertible matrix* $\mathbf{A}$ *in* $O(n^{2.373})$ *operations.*
- UPDATE($i, j \in \{1, ..., n\}, c \in \mathbb{F}$) *Set* $\mathbf{A}_{i,j} \leftarrow c$ *in* $O(n^{1.529})$ *operations.*
- QUERY($i, j \in \{1, ..., n\}$) *Return* $(\mathbf{A}^{-1})_{i,j}$ *in* $O(n^{0.529})$ *operations.*

We assume that matrix $\mathbf{A}$ *stays invertible throughout all updates.*[1]

7.1 Faster data structure for few updates

We first show that the following variant of Theorem 7.0.1 exists which supports fast updates and queries if there are not too many updates in total.

Lemma 7.1.1. *There exists a data structure with the following operations*

- INIT($\mathbf{A} \in \mathbb{F}^{n\times n}$) *Initialize on a given invertible matrix* $\mathbf{A}$ *in* $O(n^{2.373})$ *operations.*
- UPDATE($i, j \in \{1, ..., n\}, f \in \mathbb{F}$) *Set* $\mathbf{A}_{i,j} \leftarrow \mathbf{A}_{i,j} + f$ *in* $O(nk)$ *operations where* k *is the number of updates so far.*
- QUERY($i, j \in \{1, ..., n\}$) *Return* $(\mathbf{A}^{-1})_{i,j}$ *in* $O(k)$ *operations where* k *is the number of updates so far.*

We assume that matrix $\mathbf{A}$ *stays invertible throughout all updates.*

Proof. The data structure stores the inverse in the following form: Let $\mathbf{A}'$ be the matrix $\mathbf{A}$ as given during initialization, then we store $\mathbf{A}'^{-1}$ in memory. After k updates, we also have some k vectors $u_1, ..., u_k, v_1, ..., v_k$ in memory. We claim that $\mathbf{A}^{-1} = \mathbf{A}'^{-1} + \sum_{i=1}^{k} u_i v_i^\top$. Before proving this claim, we define the operations of our data structure.

[1] We will prove in a later lecture that this assumption is not required.

- INITIALIZE($\mathbf{A} \in \mathbb{F}^{n\times n}$): Compute and store $\mathbf{A}^{-1}$ as $\mathbf{A}'^{-1}$.
- UPDATE($i, j \in \{1, ..., n\}, f \in \mathbb{F}$): If this is the kth update, we store

$$u_k = \mathbf{A}'^{-1} e_i + \sum_{\ell=1}^{k-1} u_\ell v_\ell^\top e_i$$

$$v_k = \left(\frac{-f \cdot (e_j^\top \mathbf{A}'^{-1} + \sum_{\ell=1}^{k-1} e_j^\top u_\ell v_\ell^\top)}{(1 - (\mathbf{A}'^{-1}_{j,i} + \sum_{\ell=1}^{k-1} (u_\ell)_j (v_\ell)_i) \cdot f)} \right)^\top.$$

- QUERY($s, t \in \{1, ..., n\}$): Return $\mathbf{A}'^{-1}_{s,t} + \sum_{\ell=1}^{k} (u_\ell)_s (v_\ell)_t$.

Assuming $\mathbf{A}^{-1} = \mathbf{A}'^{-1} + \sum_{\ell=1}^{k} u_\ell v_\ell^\top$, then QUERY correctly returns $(\mathbf{A}^{-1})_{s,t}$. This takes $O(k)$ operations because we have k products for which we take the sum.

The claim $\mathbf{A}^{-1} = \mathbf{A}'^{-1} + \sum_{i=1}^{k} u_i v_i^\top$ holds by induction. For $k = 0$, there is no update, and it follows from definition, i.e. only INITIALIZE was called so far, so $\mathbf{A}^{-1} = \mathbf{A}'^{-1}$.

For the induction step, let $\overline{\mathbf{A}}$ be the matrix $\mathbf{A}$ after k updates. We now perform the $(k+1)$st update. Then by Sherman-Morrison

$$\mathbf{A}^{-1} = (\overline{\mathbf{A}} + e_i e_j^\top \cdot f)^{-1} = \overline{\mathbf{A}}^{-1} - \frac{\overline{\mathbf{A}}^{-1} e_i f e_j^\top \overline{\mathbf{A}}^{-1}}{(1 - e_j^\top \overline{\mathbf{A}}^{-1} e_i \cdot f)}.$$

Here $u_{k+1} := \overline{\mathbf{A}}^{-1} e_i = (\mathbf{A}'^{-1} + \sum_{\ell=1}^{k-1} u_\ell v_\ell^\top) e_i$ and

$$v_{k+1} := \left(\frac{-f e_j^\top \overline{\mathbf{A}}^{-1}}{1 - e_j^\top \overline{\mathbf{A}}^{-1} e_i f} \right)^\top = \left(\frac{-f \cdot e_j^\top (\mathbf{A}'^{-1} + \sum_{\ell=1}^{k-1} u_\ell v_\ell^\top)}{(1 - e_j^\top (\mathbf{A}'^{-1} + \sum_{\ell=1}^{k-1} u_\ell v_\ell^\top) e_i \cdot f)} \right)^\top.$$

So indeed $\mathbf{A}^{-1} = \overline{\mathbf{A}}^{-1} + u_{k+1} v_{k+1}^\top = \mathbf{A}'^{-1} + \sum_{\ell=1}^{k+1} u_\ell v_\ell^\top$.

The time complexity of an update is $O(nk)$ because u_{k+1} and v_{k+1} are just one row or column of $\overline{\mathbf{A}}^{-1}$ (though v_{k+1} is also scaled by $-f/(1 - (\mathbf{A}'^{-1}_{j,i} + \sum_{\ell=1}^{k-1} (u_\ell)_j (v_\ell)_i) \cdot f))$. That means we must query $O(n)$ entries of $\overline{\mathbf{A}}^{-1}$. Note that computing $(\mathbf{A}'^{-1}_{j,i} + \sum_{\ell=1}^{k-1} (u_\ell)_j (v_\ell)_i)$ for the inverse also only takes $O(k)$ times which is dominated by $O(nk)$. We previously argued that QUERY returns an entry of the inverse and takes $O(k)$ operations. So we can just call this function $O(n)$ times to obtain u_{k+1} and v_{k+1} for a total cost of $O(nk)$ operations. □

Note that during initialization of Lemma 7.1.1 we compute the inverse of $\mathbf{A}$. This can be done in $O(n^{2.373})$ operations using fast matrix multiplication.

Note that we could run the data structure from Lemma 7.1.1 for some $T \geq 1$ many updates, and after T updates we restart the data structure, i.e. perform the initialization again. This would give an amortized update time of $O(nT + n^\omega/T)$, because each update takes at most $O(nT)$ time and once every T updates we pay $O(n^\omega) = O(n^{2.373})$ time to restart. Picking $T = n^{(\omega-1)/2}$ would lead to an average update time of $O(n^{(\omega+1)/2}) = O(n^{1.687})$. To prove Theorem 7.0.1 we are left with two tasks:

- Reduce the update time further to $O(n^{1.529})$ time.

- Turn the amortized update time into a worst-case update time.

For the first task, we will use fast rectangular matrix multiplication.

We show that we can restart the data structure in just $n^{\omega(1,1,\mu)}$ time after n^μ updates, which is less than the previously used restart cost of n^ω for $\mu < 1$. In particular, we get an amortized update time of $O(n^{1+\mu} + n^{\omega(1,1,\mu)-\mu})$ if we restart our data structure after $T = n^\mu$ updates.

Lemma 7.1.2. *When given* $\mathbf{A}$*,* $\mathbf{A}^{-1}$*, and a matrix* $\mathbf{A}'$ *that differs in* $\mathbf{A}$ *in at most* n^μ *entries for* $0 \leq \mu \leq 1$*, then we can compute* $\mathbf{A}'^{-1}$ *in* $O(n^{\omega(1,1,\mu)})$ *operations.*

We prove this lemma via the following extension of the Sherman-Morrison identity

Lemma 7.1.3 (Woodbury-identity [Woo50])**.**

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^\top)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^\top\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^\top\mathbf{A}^{-1}$$

Proof. This can be proven by multiplying with $(\mathbf{A} + \mathbf{U}\mathbf{V}^\top)$ and verifying that the product is just the identity matrix. □

Proof of Lemma 7.1.2. If $\mathbf{A}'$ and $\mathbf{A}$ differ in at most n^μ entries, then we can write $\mathbf{A}' = \mathbf{A} + \mathbf{U}\mathbf{V}^\top$ where $\mathbf{U}, \mathbf{V}$ are matrices of size $n \times n^\mu$ and each column has only one non-zero entry. Because of this structure of $\mathbf{U}$ and $\mathbf{V}$, the products $\mathbf{A}^{-1}\mathbf{U}$ and $\mathbf{V}^\top\mathbf{A}^{-1}$ are just n^μ rows and columns of $\mathbf{A}^{-1}$ respectively. Likewise, $\mathbf{V}^\top\mathbf{A}^{-1}\mathbf{U}$ is just a $n^\mu \times n^\mu$ submatrix of $\mathbf{A}^{-1}$.

This implies (i) we can compute $(\mathbf{I} + \mathbf{V}^\top\mathbf{A}^{-1}\mathbf{U})^{-1}$ in $O((n^\mu)^\omega) = O(n^{\omega(\mu,\mu,\mu)})$ operations. The product $(\mathbf{I} + \mathbf{V}^\top\mathbf{A}^{-1}\mathbf{U})^{-1}(\mathbf{V}^\top\mathbf{A}^{-1})$ then takes $O(n^{\omega(\mu,\mu,1)})$ operations. And lastly, we can compute

$$(\mathbf{A}^{-1}\mathbf{U})((\mathbf{I} + \mathbf{V}^\top\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^\top\mathbf{A}^{-1})$$

in $O(n^{\omega(1,\mu,1)}) = O(n^{\omega(1,1,\mu)})$ operations. Thus by Lemma 7.1.3 we can compute the inverse of $\mathbf{A}'^{-1}$ in $O(n^{\omega(1,1,\mu)})$ operations. □

Proof of Theorem 7.0.1, amortized update time. We now obtain Theorem 7.0.1 where the amortized (i.e. average) update time is $O(n^{1.529})$ time. We simply run Lemma 7.1.1 for $T = n^\mu$ updates. Then after T updates, we restart the data structure. For this we must compute the inverse which can be done in $O(n^{\omega(1,1,\mu)})$ operations as the matrix change in only $T = n^\mu$ entries. The amortized time per update is thus $O(n^{1+\mu} + n^{\omega(1,1,\mu)-\mu})$. Using the current best bounds on $\omega(1,1,\mu)$ by [GU18], we obtain $O(n^{1.529})$ update time for $\mu = 0.529$. A query takes $O(T) = O(n^\mu) = O(n^{0.529})$ time.[2] □

7.2 Worst-case update time

We now describe how to turn the amortized time complexity into a worst-case bound, that is, every update takes at most $O(n^{1.529})$ time.

[2] https://www.ocf.berkeley.edu/~vdbrand/complexity/?terms=1%2Bx%0Aomega(1%2C1%2Cx)-x%0A

Lemma 7.2.1. *Let $\mathcal{A}$ be a data structure with reset time $O(r(k))$, update time $O(u(k))$ and query time $O(q(k))$, where k is the number of past updates, since the last reset or initialization. For every $T \in \mathbb{N}$ there is a data structure $\mathcal{B}$ with worst-case update complexity $O(u(T) + r(T)/T)$ and query complexity $O(q(T))$.*

Proof. Assume we run a copy of $\mathcal{A}$ while we receive updates. The copy of $\mathcal{A}$ will have the following life-cycle:

1. For the first $T/3$ updates, we pass the updates directly into $\mathcal{A}$.
2. We now perform a reset. This reset takes at most $O(r(T))$ time. Instead of performing this reset all at once, we perform only a few steps of the reset operation every time we receive an update, such that it takes $T/3$ updates until the reset is completed. These $T/3$ updates are not passed to $\mathcal{A}$ as it is in the process of resetting, so we just put them in a queue.
3. Once the reset is completed, the data structure $\mathcal{A}$ is missing the past $T/3$ updates. For the next $T/3$ updates, we put each new update into the queue, but also remove two updates from the queue and process them in $\mathcal{A}$. After $T/3$ updates the queue is empty.
4. Go back to step 1.

Note that with each update we receive, we spend only $O(u(T) + r(T)/T)$ time. Further, while in phase 1, the data structure $\mathcal{A}$ can answer queries correctly. In phase 2 and 3 it can not answer queries correctly because not all updates have been processed yet. To fix this issue, we simply run 3 copies of this data structure in parallel, but with their life-cycle slightly offset. The first copy starts with phase 1. The 2nd copy starts in phase 2 (i.e. we reset it immediately), and the third copy starts in phase 3. This way, there is always one copy that can answer the queries while the time per update incrases only by a constant factor. □

For $r(n^\mu) = n^{\omega(1,1,\mu)}$, $u(n^\mu) = n^{1+\mu}$ this implies Theorem 7.0.1 can have worst-case update time $O(n^{1+\mu} + n^{\omega(1,1,\mu)-\mu})$.

7.3 Rank and non-invertible matrices

Previously, we have proven Theorem 7.0.1 under the assumption that the input matrix $\mathbf{A}$ stays invertible throughout all updates. We now extend the result to support non-invertible matrices. That is, when querying some entry of the inverse $\mathbf{A}^{-1}_{i,j}$ while the matrix is not invertible, the data structure returns an error message, but the data structure does not break/crash. It still supports updates and once the input matrix $\mathbf{A}$ is invertible again, queries to $\mathbf{A}^{-1}_{i,j}$ are correctly answered again.

This result is implies by the following lemma:

Lemma 7.3.1 ([San07])**.** *Given $\mathbf{A} \in \mathbb{F}^{n\times n}$, define*

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{X} & 0 \\ \mathbf{Y} & 0 & \mathbf{I} \\ 0 & \mathbf{I} & \mathbf{I}_k \end{bmatrix}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are $n \times n$ matrices and each entry is an independent uniformly at random sampled from $\mathbb{F}$, and $\mathbf{I}_k$ is a partial identity matrix, i.e. the first k diagonal entries being 1 and the remaining $n-k$ diagonal entries being 0.

Then with probability at least $1 - \frac{3n}{\|F\|}$ we have that $\det(\mathbf{M}) \neq 0$ if and only if rank$(\mathbf{A}) \geq n-k$.

Further, for full rank $\mathbf{A}$ and $k = 0$ we have

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & 0 & -\mathbf{A}^{-1}\mathbf{X} \\ 0 & 0 & \mathbf{I} \\ -\mathbf{A}^{-1}\mathbf{Y} & \mathbf{I} & -\mathbf{Y}\mathbf{A}^{-1}\mathbf{X} \end{bmatrix}$$

Note that we can run our matrix inverse data structure on matrix $\mathbf{M}$. Assume for simplicity that at the start, the matrix $\mathbf{A}$ is full rank and we pick $k = 0$. Then we can correctly answer queries to $\mathbf{A}^{-1}_{i,j}$ by returning $\mathbf{M}^{-1}_{i,j}$.

When changing an entry of $\mathbf{A}$, then the rank of $\mathbf{A}$ changes by at most 1, i.e. it increases by 1, decreases by 1, or stays the same. So whenever an entry of $\mathbf{A}$ is changed, we check if we can decrease k, keep using the same k, or must increase k. This way we always keep using the smallest possible k. This check is done using the property $\det(\mathbf{M} + uv^\top) = \det(\mathbf{M}) \cdot (1 - v^\top \mathbf{M}^{-1} u)$. So when changing an entry, we can quickly check if the determinant becomes 0.

Proof of Lemma 7.3.1. We start with the last claim as it is easy to verify:

$$\begin{bmatrix} \mathbf{A} & \mathbf{X} & 0 \\ \mathbf{Y} & 0 & \mathbf{I} \\ 0 & \mathbf{I} & \mathbf{I}_k \end{bmatrix} \cdot \begin{bmatrix} \mathbf{A}^{-1} & 0 & -\mathbf{A}^{-1}\mathbf{X} \\ 0 & 0 & \mathbf{I} \\ -\mathbf{A}^{-1}\mathbf{Y} & \mathbf{I} & -\mathbf{Y}\mathbf{A}^{-1}\mathbf{X} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & 0 & 0 \\ 0 & \mathbf{I} & 0 \\ 0 & 0 & \mathbf{I} \end{bmatrix}$$

So the matrix is indeed the inverse of $\mathbf{M}$.

Full-rankness of $\mathbf{M}$ We now prove the $\mathbf{M}$ is full rank when rank$(\mathbf{A}) \geq n-k$.

By expanding the determinant by minors we have $\det(\mathbf{M}) = \sum_i (-1)^{i+j} \mathbf{M}_{i,j} \det(\mathbf{M}^{\not{i,j}})$ for any j. Further, not that the last $n-k$ rows and columns of $\mathbf{M}$ are all 0 except for a single 1. Thus we have

$$\det(\mathbf{M}) = \pm \det \begin{bmatrix} \mathbf{A} & \mathbf{X}^L & 0 \\ \mathbf{Y}^T & 0 & \mathbf{I}_{k\times k} \\ 0 & \mathbf{I}_{k\times k} & \mathbf{I}_{k\times k} \end{bmatrix}$$

where $\mathbf{Y}^T$ are the k top rows of $\mathbf{Y}$, $\mathbf{X}^L$ are the left-most k columns of $\mathbf{X}$, and $\mathbf{I}_{k\times k}$ is the $k \times k$ identity matrix.

Adding a row onto another does not change the determinant, so we can transform above matrix to

$$\begin{bmatrix} \mathbf{A} & \mathbf{X}^L & 0 \\ \mathbf{Y}^T & -\mathbf{I}_{k\times k} & 0 \\ 0 & \mathbf{I}_{k\times k} & \mathbf{I}_{k\times k} \end{bmatrix}$$

and then use the expansion by minors again since the last k columns have only one non-zero entry. This way we obtain

$$\begin{bmatrix} \mathbf{A} & \mathbf{X}^L \\ \mathbf{Y}^T & -\mathbf{I}_{k\times k} \end{bmatrix}$$

Another set of row and column operations leads to

$$\begin{bmatrix} \mathbf{A} + \mathbf{Y}^T \mathbf{X}^L & 0 \\ 0 & -\mathbf{I}_{k \times k} \end{bmatrix}$$

Whp, the rank of this matrix is just $k + \min\{\text{rank}(\mathbf{A}) + k, n\}$ because $(\mathbf{Y}^T \mathbf{X}^L)$ is a random rank k matrix. Thus we the above matrix is full rank if and only if rank$(\mathbf{A}) \geq n - k$. □

7.4 Exercises

7.4.1 Matrix Data Structure, Faster Column Updates

Show that for any $0 \leq \mu \leq 1$ there exists a data structure with the following operations

- INITIALIZE($\mathbf{A} \in \mathbb{F}^{n \times n}$): Initialize on the given matrix in $O(n^\omega)$ operations.
- UPDATE($i \in \{1, ..., n\}, v \in \mathbb{F}$): Set the ith column of $\mathbf{A}$ to v in $O(n^{1+\mu} + n^{\omega(1,1,\mu)-\mu})$ operations.
- QUERY($i \in \{1, ..., n\}$): Return the ith row of $\mathbf{A}^{-1}$ in $O(n^{1+\mu})$ operations.

You are allowed to assume that $\mathbf{A}$ stays invertible throughout all updates, and it suffices to show an amortized (i.e., average) update time.

Hint: In the lecture (Theorem 10.0.1 and Lemma 10.1.1 of lecture notes) we proved this data structure for the case of entry updates. There we maintained the inverse in the implicit form

$$\mathbf{A}^{-1} = \mathbf{A}'^{-1} - \sum_{i=1}^{k} u_i v_i^\top$$

where $\mathbf{A}'$ is the matrix $\mathbf{A}$ during initialization, and $u_1, ..., u_k, v_1, ..., v_k \in \mathbb{R}^n$, and k is the number of updates so far. So instead of storing the n^2 entries of $\mathbf{A}^{-1}$ explicitly in memory, we stored this matrix *implicitly* by only storing the matrix $\mathbf{A}'^{-1}$ and the vectors $u_1, ..., u_k, v_1, ..., v_k$.

For column updates, computing the vectors u_i would be expensive. Instead try to store the vectors in some implicit form.

7.4.2 Matrix Data Structure, Faster element Updates

Show that for any $0 \leq \mu \leq 1$ there exists a data structure with the following operations:

- INITIALIZE($\mathbf{A} \in \mathbb{F}^{n \times n}$): Initialize on the given matrix in $O(n^\omega)$ operations.
- UPDATE($i, j \in \{1, ..., n\}, f \in \mathbb{F}$): Set $\mathbf{A}_{i,j} \leftarrow f$ in $O(n^{2\mu} + n^{\omega(1,1,\mu)-\mu})$ operations.
- QUERY($i, j \in \{1, ..., n\}$): Return $(\mathbf{A}^{-1})_{i,j}$ in $O(n^{2\mu})$ operations.

You are allowed to assume that $\mathbf{A}$ stays invertible throughout all updates, and it suffices to show an amortized (i.e., average) update time.

Hint: It might help to store the inverse $\mathbf{A}^{-1}$ implicitly in the form

$$\mathbf{A}^{-1} = \mathbf{A}'^{-1} - \mathbf{L}\mathbf{C}\mathbf{R}^\top$$

where $\mathbf{A}'$ is what matrix $\mathbf{A}$ looked like during initialization, $\mathbf{C}$ is a matrix of size at most $n^\mu \times n^\mu$, and $\mathbf{L}, \mathbf{R}$ are matrices of size at most $n \times n^\mu$. Then after each update to $\mathbf{A}$, update the matrices $\mathbf{L}, \mathbf{C}, \mathbf{R}$ so you again have $\mathbf{A}^{-1} = \mathbf{A}'^{-1} - \mathbf{L}\mathbf{C}\mathbf{R}^\top$.

Remark: Just to give some additional background why this data structure is interesting: For $\mu \approx 0.723$, this would be $O(n^{1.447})$ operations per update and query.[3] For comparison, in the lecture we constructed a data structure with slower $O(n^{1.529})$ operations per update but faster $O(n^{0.529})$ operations per query. So for applications with few queries, this new data structure is faster.

[3] https://www.ocf.berkeley.edu/~vdbrand/complexity/?terms=2mu%0Aomega(1%2C1%2Cmu)-mu

Chapter 8

Conditional Lower Bounds

8.1 Lower Bounds for combinatorial algorithms and data structures

We previously constructed a data structure Theorem 7.0.1 that beat $O(n^2)$ update time. This data structure internally uses fast matrix multiplication, which is slow in practice. This raises the question whether there might be a different way to beat $O(n^2)$ update time without using fast matrix multiplication. In this section we prove that this is not possible. Algorithms and data structures without fast matrix multiplication are often called "combinatorial". So we now want to prove a complexity lower bound for "combinatorial data structures".

Definition 8.1.1. The "boolean semi-ring" is defined as $\{0,1\}$ with the operations $0 \cdot 1 = 0, 1 \cdot 1 = 1, 0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 1$, i.e. arithmetic works as we are used to except that $1 + 1 = 1$. Formally, multiplication is "logical and", and addition is "logical or" when interpreting 0 as false and 1 as true.

Boolean matrix multipliation is the the product of two matrices from $\{0,1\}^{n\times n}$ where all operations are defined over the boolean semi-ring.

For intuition, boolean matrix multiplication can be interpreted as multiplying two regular matrices with non-negative entries, and then asking which entries of the result are non-zero. I.e. we just care about the zero vs. non-zero profile of the matrix but not the actual numbers involved. Example:

$$\begin{bmatrix} 0 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 10 & 0 \end{bmatrix} \qquad \text{regular product}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \qquad \text{boolean product}$$

Theorem 8.1.2 ([WW10])**.** *If there exists an algorithm that can detect a triangle in an n-node graph in $O(n^{3-\epsilon})$ time, then there exists an algorithm for computing a boolean matrix product in $O(n^{3-\epsilon/3}\log n)$ time.*

Note that we can find a triangle in a graph by computing just 2 matrix products. Given adjacency matrix $\mathbf{A}$, we simply compute $\mathbf{A}^3$ and check if a diagonal entry is non-zero. So if we can multiply matrices in less than $O(n^3)$ time, then we can find a triangle in less than $O(n^3)$ time.

The above theorem says that the reverse is also true: If we can detect a triangle in subqubic time, then we can multiply matrices in subqubic time. This was proven by Williams and Vassilevska Williams (2010) [WW10].

The reduction is somewhat surprising given that triangle detection gives only a single bit of information "is there a triangle, or not". Yet somehow we can blow up that single bit into n^2 many bits, since the result of a matrix product consists of n^2 entries.

To prove the above theorem, we start with the following lemma.

Lemma 8.1.3. *Given a triangle detection algorithm with complexity $O(T(n))$, two boolean matrices $\mathbf{A}, \mathbf{B}$, and a set $S \subset \{1, ..., n\} \times \{1, ..., n\}$. We can find all $(i, j) \in S$ with $(\mathbf{A} \cdot \mathbf{B})_{i,j} \neq 0$ in time $O(T(n) + (T(n) \cdot k \cdot \log n)$, where k is the number of such non-zero entries.*

Proof. We construct a graph with vertices $u_1, ..., u_n, v_1, ..., v_n, w_1, ..., w_n$. We add edges (u_i, v_j) for all i, j with $\mathbf{A}_{i,j} = 1$, and edges (v_i, w_j) for all i, j with $\mathbf{B}_{i,j} = 1$, and edges (w_i, u_j) for all $(i, j) \in S$.

There exists a triangle in this graph, if and only if there is some $(i, j) \in S$ with $(\mathbf{A} \cdot \mathbf{B})_{i,j} = 1$. This is because $(\mathbf{A} \cdot \mathbf{B})_{i,j} = \sum_k \mathbf{A}_{i,k} \mathbf{B}_{k,j}$ which is non-zero if and only if there is k with $\mathbf{A}_{i,k} = \mathbf{B}_{k,j} = 1$, i.e. we have a triangle $u_i v_k w_j$.

We can now find all the indices $(i, j) \in S$ with $(\mathbf{A} \cdot \mathbf{B})_{i,j} = 1$ via binary search. □

Proof of Theorem 8.1.2. Given $\mathbf{A}, \mathbf{B}$, we split the matrices into blocks of size $n^{1/3} \times n^{1/3}$. Let's call these blocks $\mathbf{A}^{i,j}, \mathbf{B}^{i,j}$ for $i, j = 1, ..., n^{2/3}$. Let $\mathbf{C} = \mathbf{AB}$ and perform the same split on $\mathbf{C}$, so we have

$$\mathbf{C}^{i,j} = \sum_{k=1}^{n^{2/3}} \mathbf{A}^{i,k} \mathbf{B}^{k,j}.$$

Observe that since over the boolean semi-ring $1 + 1 = 1$ (i.e. we only care about the non-zero entries) we do not need to compute $(\mathbf{A}^{i,k}\mathbf{B}^{k,j})_{i',j'}$ for $i'j,' \in \{1, ..., n^{1/3}\}$ if we know $(\mathbf{A}^{i,k}\mathbf{B}^{k',j})_{i',j'} \neq 0$ for some $k' \neq k$.

Thus we can compute all $\mathbf{C}^{i,j}$ via the following algorithm.

- For $i, j = 1, ..., n^{2/3}$:
  - //We now compute $C^{i,j}$
  - $S = \{1, ..., n^{1/3}\} \times \{1, ..., n^{1/3}\}$
  - //We now check for all $(i', j') \in S$ whether $C^{i,j}_{i',j'} = 1$.
  - For $k = 1, ..., n^{2/3}$
    - Find all non-zeros in S of $\mathbf{A}^{i,k}\mathbf{B}^{k,j}$ via Lemma 8.1.3.
    - For the discovered non-zero entries i', j', set $C^{i,j}_{i',j'} = 1$ and remove (i', j') from S.

Let $K_{i,j,k}$ be the number of non-zero we found in product $\mathbf{A}^{i,k}\mathbf{B}^{k,j}$. Since we only look for non-zeros in set S and remove each (i',j') if $(\mathbf{A}^{i,k}\mathbf{B}^{k,j})_{i',j'} \neq 0$, we have for all i,j that

$$\sum_{k=1}^{n^{2/3}} K_{i,j,k} = n^{2/3}.$$

Hence, by Lemma 8.1.3 the totaltime complexity of this is

$$O((n^{2/3})^2(\sum_k K_{i,j,k}T(n^{1/3})\log n)) = O(n^{4/3}(n^{2/3}n^{(3-\epsilon)/3})\log n) = O(n^{3-\epsilon/3}\log n).$$

□

Corollary 8.1.4. *If there is a data structure that can maintain st-reachability with $O(n^{3-\epsilon}$ initialization time and $O(n^{2-\epsilon})$ time for an edge insertion or deletion for some constant $\epsilon > 0$, then we can compute a boolean matrix product in $O(n^{3-\epsilon'})$ time for some constant $\epsilon' > 0$.*

Note that st-reachability can be trivially solved by just running BFS which takes at most $O(n^2)$ time. This corollary thus tells us that to get any non-trivial data structure, we must use fast matrix multiplication.

Proof. Given matrices $\mathbf{A}, \mathbf{B}$ we reduce this matrix product to triangle detection. Triangle detection can be reduced to st-reachability as follows. Given some graph $G = (V = \{1, ..., n\}, E)$ in which we want to detect a triangle, we create a graph

$$G' = (\{u_1, ..., u_n, v_1, ...v_n, w_1, ..., w_n, x_1, ..., x_n\}, E').$$

For every edge $(i,j) \in E$ we create edges $(u_i, v_j), (v_i, w_j), (w_i, x_j)$ in graph G'. We further add two isolated vertices s, t to G'. Note that a triangle exists in G, if and only if for some i there exists a path $u_i \to x_i$.

Now initialize the st-reachability data structure on G'. We insert edges $(s, u_1), (x_1, t)$, so if there now exists an st-path in G', then there is a triangle in G involving vertex 1. We then delete these two edges from G' again an insert $(s, u_2), (x_2, t)$ and so forth. After $O(n)$ deletion we checked for all i if there is a $u_i \to x_i$ path in G', i.e. if there exists any triangle in G.

So if the initialization of the data structure took less than $O(n^{3-\epsilon})$ time and each update took less than $O(n^{2-\epsilon})$ time for some constant $\epsilon > 0$, then we were able to detect a triangle (and thus compute a boolean product) in less than $O(n^{3-\epsilon'})$ time for some constant $\epsilon' > 0$. □

8.2 OMv-Problem and Conjecture

In previous lectures we learned that we can, for example, maintain reachability in a graph in $O(n^{1.529})$ time per update (edge insertion or deletion). A natural question is: what could be the best possible complexity for this problem we could ever hope for? How much can the update complexity be improved in future research? That is, we want to prove a lower bound on the update complexity.

In this section we will consider “conditional lower bounds”. A conditional lower bound holds under some assumption. A simple example for a conditional lower bound would be the following: Assuming $P \neq NP$, there is no polynomial time algorithm for the TSP problem.

Instead of P vs. NP, our lower bounds for data structures will be based on a different conjecture called “OMv-conjecture”, introduced by Henzinger, Forster, Nanongkai and Saranurak (2015) [HKNS15].

Definition 8.2.1 (OMv-Problem [HKNS15])**.** The OMv-problem consists of the following two phases

- We are given an $n \times n$ matrix $\mathbf{M} \in \{0, 1\}^{n \times n}$ and are allowed to preprocess it.
- After the preprocessing, we receive an online sequence of vectors $v_1, ..., v_n \in \{0, 1\}^n$. We only receive v_{i+1} after we returned/computed $\mathbf{M}v_i$.

The computation is performed over the boolean semiring.

Many people believe that this problem can not be solved in subqubic total time, i.e. there is no algorithm that runs in $O(n^{3-\epsilon})$ time (for constant $\epsilon > 0$) for both preprocessing and to compute the n matrix-vector products. This is called the OMv-conjecture

Conjecture 8.2.2. *The OMv-conjecture states that there is no algorithm for the OMv-problem that runs in $O(n^{3-\epsilon})$ total time.*

The OMv-problem and the conjecture are over the boolean semi-ring, but the problem was also studied when the input is over some field. Chakraborty et al.[CKL18] showed that any algorithm, that does not exploit the structure of the field, must have $\Omega(n^3)$ complexity. For example, the naive matrix-vector product where we multiply rows by columns works no matter what field the matrix or vector are defined over, so this would be an algorithm that does not exploit the structure of the field.

Preprocessing One might expect that the matrix-vector products need $\Omega(n^3)$ time, because there just isn’t enough useful information that can be extracted during $O(n^3)$-time preprocessing. One could ask if maybe the problem becomes easier if we allow for more than $O(n^3)$ preprocessing time. The following lemma shows this is not the case.

Lemma 8.2.3. *Assume we have an algorithm for the OMv-problem with $O(n^c)$ preprocessing time and $O(n^{3-\epsilon})$ time for computing the matrix-vector products. Here c, ϵ are constants.*

Then there exists an algorithm for the OMv-problem with $O(n^{3-\epsilon'})$ total time for both phases, for some constant $\epsilon' > 0$.

Proof. When we are given the matrix $\mathbf{M}$, we split it into blocks of size $n^k \times n^k$ for some $0 < k < 1$. We initialize the assumed algorithm on each block. This takes $O((n^k)^c \cdot (n^{1-k})^2) = O(n^{2+k(c-2)})$ time. So for $k \leq (1-\epsilon')/(c-2)$ this is subqubic time.

Next, we must multiply a matrix-vector product. Let $\mathbf{M}_{i,j}$ for $i, j \in \{1, ..., n^{1-k}\}$ be the $n^k \times n^k$-sized blocks of $\mathbf{M}$. Given a vector v, let v_j for $j \in \{1, ..., n^{1-k}\}$ be the same split of the vector v. Then we can compute $\mathbf{M}v$ by computing all products of form $\mathbf{M}_{i,j}v_j$.

For a total of n^k matrix-vector products, this will take $O((n^k)^{3-\epsilon} \cdot n^{2(1-k)}) = O(n^{2+k(1-\epsilon)})$ time. As we must compute in total n matrix-vector products, we get $O(n^{1-k} \cdot n^{2+k(1-\epsilon)}) = O(n^{3-k\epsilon})$ which is again subqubic.

□

There also exists the so called OuMv-problem, where instead of just some vector v_i for which we must compute $\mathbf{M}v_i$, we receive two vectors u_i, v_i and must compute $u_i^\top \mathbf{M} v_i$.

Definition 8.2.4. The OuMv-problem consists of the following two phases

- We are given an $n \times n$ matrix $\mathbf{M} \in \{0,1\}^{n \times n}$ and are allowed to preprocess it.
- After the preprocessing, we receive an online sequence of vectors $v_1, u_1, ..., v_n, u_n \in \{0,1\}^n$. We only receive u_{i+1}, v_{i+1} after we returned/computed $u_i^\top \mathbf{M} v_i$.

The computation is performed over the boolean semiring.

Even though here we must return only a sinlge value $u_i^\top \mathbf{M} v_i$ instead of a vector $\mathbf{M} v_i$, the OuMv-problem is not easier than the OMv-problem.

Lemma 8.2.5. *If there is an algorithm for the OuMv-problem with $O(n^c)$ preprocessing time and $O(n^{3-\epsilon})$ time for the vector-matrix-vector products, then there is an algorithm for the OMv-problem with $O(n^{c'})$ preprocessing time and $O(n^{3-\epsilon'})$ time for the matrix-vector products.*

In particular, the existence of such an algorithm for the OuMv-problem would contradict the OMv-conjecture.

Proof. Problem set 4. □

8.2.1 Conditional Lower Bounds

Theorem 8.2.6. *Assuming the OMv-conjecture, there exists no data structure with the following operations and complexities*

- INITIALIZE(G) *Preprocesses the given graph in polynomial time.*
- UPDATE(u, v) *Inserts (or deleted) the given edge in $O(n^{1-\epsilon})$ time for some constant $\epsilon > 0$.*
- QUERY(s, t) *Returns if s can reach t in $O(n^{2-\epsilon})$ time for some constant $\epsilon > 0$.*

Note that just running BFS would already take $O(n^2)$ time, so above theorem states that any non-trivial complexity for the queries must result in at least linear time for processing the updates.

Proof. We show that if such a data structure exists, then we could break the OMv-conjecture, i.e. we could solve the OMv-problem in subqubic time.

We show that the data structure could solve the OuMv-problem in polynomial preprocessing time and $O(n^{3-\epsilon})$ time to compute all n vector-matrix-vector products. Via the reduction from the previous section this woould imply we can solve the OMv-problem in less than $O(n^{3-\epsilon'})$ total time which would contradict the OMv-conjecture.

Preprocessing Given a matrix $\mathbf{M}$ we construct a bipartite graph with left vertices $\{1, ..., n\}$ and right vertices $\{1, ..., n\}$, and with edges (i, j) if $\mathbf{M}_{i,j} = 1$. We further insert two extra vertices s, t into the graph. This graph is given to the data structure durin initialization.

Vector-matrix-vector products Next, when we are given a vector u, we add edges (s, i) for $u_i = 1$. For the vector v we add edges (j, t) for $v_j = 1$.

Now there exists a path from s to t if and only if $u\mathbf{M}v \neq 0$, because

$$u^\top \mathbf{M}v = \sum_{i,j} u_i \mathbf{M}_{i,j} v_j \neq 0 \Leftrightarrow \exists i, j : u_i = \mathbf{M}_{i,j} = v_j = 1 \Leftrightarrow \exists i, j : (s, i), (i, j), (j, t) \in E$$

So we can answer $u^\top \mathbf{M}v$ and then delete the edges from s and t again.

This takes $O(n^{2-\epsilon})$ time per vector-matrix-vector product and thus $O(n^{3-\epsilon})$ time for all n such products, contradicting the OMv-conjecture. □

At the start of this course, we talked about a data structure that solves least squares regression. We are given a matrix $\mathbf{A} \in \mathbb{R}^{n\times d}$ and vector $b \in \mathbb{R}^n$ and must return the minimizer x of $\|\mathbf{A}x - b\|_2$. The data structure supports updates that allow inserting now rows into $\mathbf{A}$ and b. The data structure we constructed has update time $O(d^2)$ and we now want to prove that this is optimal.

Theorem 8.2.7 ([JPW22])**.** *Assuming the OMv-conjecture, there exists no data structure with the following operations and complexities*

- INITIALIZE$(\mathbf{A} \in \mathbb{R}^{n\times d}, b \in \mathbb{R}^n)$ *Preprocesses the matrix and vector in polynomial time.*
- UPDATE$(a \in \mathbb{R}^d, f \in \mathbb{R})$ *Appends $a^\top$ as new row into $\mathbf{A}$ and f as new entry into b. Then return the minimizer x of $\|\mathbf{A}x - b\|_2$ in $O(d^{2-\epsilon})$ time for some constant $\epsilon > 0$.*

Proof. We show that if such a data structure exists, then we could solve the OMv-problem in subqubic time.

Outline To solve the regression problem, we compute $x = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b$. If we now insert a new row, the new solution is given by $x' = (\mathbf{A}^\top\mathbf{A} + aa^\top)^{-1}(\mathbf{A}^\top b + af)$. If we assume for simplicity that the matrix $(\mathbf{A}^\top\mathbf{A} + aa^\top)^{-1} = (\mathbf{A}^\top\mathbf{A})^{-1}$, then we have $x' - x = (\mathbf{A}^\top\mathbf{A})^{-1}bf$. So we computed a matrix-vector product.

In our reduction we want to construct an instance of the regression problem where this matrix vector product can be sued to solve the OMv-problem.

Definitions Given a matrix $\mathbf{M} \in \{0, 1\}^{d\times d}$ for the OMv-problem, construct a matrix $\mathbf{H} \in \mathbb{R}^{2d\times 2d}$ as follows:

$$\mathbf{H} = \begin{bmatrix} 2\mathbf{I} & \frac{1}{d}\mathbf{M} \\ \frac{1}{d}\mathbf{M}^\top & 2\mathbf{I} \end{bmatrix}$$

Here we consider $\mathbf{M} \in \mathbb{R}^{d\times d}$, i.e. all computation is performed over the reals, not the boolean semiring.

Let v^t be the vector given during the t iteration of the OMv-problem, then we define for some scaling $s > 0$ (the exact value we will define later)

$$a^t := \begin{bmatrix} 0 \\ s \cdot v \end{bmatrix}$$

Note that

$$\mathbf{H}a^t := \begin{bmatrix} \frac{s}{d}\mathbf{M}v \\ 2s \cdot v \end{bmatrix}$$

so from the first d entries we can reconstruct the non-zero entries of $\mathbf{M}v$ which are required to solve the OMv-problem.

Initialization To initialize the data structure, we compute $\mathbf{A} \in \mathbb{R}^{2d \times 2d}$ with $(\mathbf{A}^\top \mathbf{A})^{-1} = \mathbf{H}$. This can be computed by inverting $\mathbf{H}$ and then constructing the SVD. Note that this requires all eigenvalues of $\mathbf{H}$ to be positive. Let $\lambda_i(\mathbf{H})$ be the ith smallest eigenvalue, then we have

$$1 \leq \lambda_1(\mathbf{H}) \leq \lambda_{2d}(\mathbf{H}) \leq 3.$$

This is because of the 2 on the diagonal of $\mathbf{H}$ and because of the scaling $1/d$ on matrix $\mathbf{M}$.

Update In the OMv-problem, we are given a vector v^t to multiply with $\mathbf{M}$. We construct the vector a^t as previously defined. The update to our data structure is such that we append this a^t as new row, and the new entry to b will just be the value 1.

Let $\mathbf{A}^t$ be the matrix $\mathbf{A}$ after appending t such vectors, likewise let b^t be the vector b after t updates. Let $\mathbf{H}^t = ((\mathbf{A}^t)^\top \mathbf{A})^{-1}$, and x^t be the tth output of the data structure. So $x^t = \mathbf{H}^t(\mathbf{A}^t)^\top b^t$.

We claim that

- $x^t - x^{t-1} \approx \mathbf{H}^t a^t$
- $\mathbf{H}^t a^t \approx \mathbf{H} a^t$

We are left with bounding how large these approximation errors are. If they are small enough, then we can reconstruct the non-zero entries of $\mathbf{H}a^t$ (and thus the product $\mathbf{M}v^t$) via $x^t - x^{t-1}$.

To prove this, we first need some additional bounds.

Bounds

$$\|a^t\|_2 \leq \frac{\sqrt{d}}{s} \tag{8.1}$$

For any vector $w \in \mathbb{R}^{2d}$ we have

$$w^\top (\mathbf{H}^t)^{-1} w = w^\top (\mathbf{H}^{-1} + \sum_{k=1}^{t} a^k (a^k)^\top) w \leq w^\top \mathbf{H}^{-1} w + \|w\|_2 \sum_{k=1}^{t} \|a^k\|_2^2 \leq \|w\|_2 + \|w\|_2 \cdot O(td/s^2)$$

Since $t \leq d$ we have $\lambda_1(\mathbf{H}^t) \leq 1/2$ for $s \leq O(1/d)$. Further, since we only add new outer products to $\mathbf{H}^{-1}$, we also have $\lambda_{2d}(\mathbf{H}^t) \leq \lambda_{2d}(\mathbf{H}) \leq 3$. In summary

$$1/2 \leq \lambda_1(\mathbf{H}) \leq \lambda_{2d}(\mathbf{H}) \leq 3 \tag{8.2}$$

Bounding the approximation

$$\begin{aligned}
x^t - x^{t-1} &= \mathbf{H}^t(\mathbf{A}^t)^\top b^t - \mathbf{H}^{t-1}(\mathbf{A}^{t-1})^\top b^{t-1} \\
&= ((\mathbf{H}^{t-1})^{-1} + a^t(a^t)^\top)^{-1}(\mathbf{A}^t)^\top b^t - \mathbf{H}^{t-1}(\mathbf{A}^{t-1})^\top b^{t-1} \\
&= (\mathbf{H}^{t-1} - \frac{\mathbf{H}^{t-1}a^t(a^t)^\top \mathbf{H}^{t-1}}{1+(a^t)^\top \mathbf{H}^{-1}a^t})(\mathbf{A}^t)^\top b^t - \mathbf{H}^{t-1}(\mathbf{A}^{t-1})^\top b^{t-1} \\
&= \mathbf{H}^{t-1}((\mathbf{A}^t)^\top b^t - (\mathbf{A}^{t-1})^\top b^{t-1}) - \frac{\mathbf{H}^{t-1}a^t(a^t)^\top \mathbf{H}^{t-1}}{1+(a^t)^\top \mathbf{H}^{-1}a^t}(\mathbf{A}^t)^\top b^t \\
&= \mathbf{H}^{t-1}a^t - \frac{\mathbf{H}^{t-1}a^t(a^t)^\top \mathbf{H}^{t-1}}{1+(a^t)^\top \mathbf{H}^{-1}a^t}(\mathbf{A}^t)^\top b^t
\end{aligned}$$

Thus we can bound $\|(x^t - x^{t-1}) - \mathbf{H}^t a^t\|_2$ as by the following

$$\begin{aligned}
\|\frac{\mathbf{H}^{t-1}a^t(a^t)^\top \mathbf{H}^{t-1}}{1+(a^t)^\top \mathbf{H}^{-1}a^t}(\mathbf{A}^t)^\top b^t\|_2 &\le \|\mathbf{H}^{t-1}a^t(a^t)^\top \mathbf{H}^{t-1}(\mathbf{A}^t)^\top b^t\|_2 \\
&\le \lambda_{2d}(\mathbf{H}^{t-1})^2 \cdot \|a^t\|_2^2 \|(\mathbf{A}^t)^\top b^t\|_2 \\
&\le O(d/s^2) \cdot \|(\mathbf{A}^t)^\top b^t\|_2 \\
&\le O(d/s^2) \cdot \sum_{k=1}^{t} \|a^t\|_2 \\
&\le O(td^{1.5}/s^3) = O(d^{2.5}/s^3)
\end{aligned}$$

Next, consider

$$\begin{aligned}
\|\mathbf{H}^t a^t - \mathbf{H}^{t-1}a^t\|_2 &= \|((\mathbf{H}^{t-1})^{-1} + a^t(a^t)^\top)^{-1}a^t - \mathbf{H}^{t-1}a^t\|_2 \\
&= \|(\mathbf{H}^{t-1} + \frac{\mathbf{H}^{t-1}a^t(a^t)^\top \mathbf{H}^{t-1}}{1+(a^t)^\top \mathbf{H}^{t-1}a^t})a^t - \mathbf{H}^{t-1}a^t\|_2 \\
&= \|\frac{\mathbf{H}^{t-1}a^t(a^t)^\top \mathbf{H}^{t-1}}{1+(a^t)^\top \mathbf{H}^{t-1}a^t}a^t\|_2 \\
&\le \|\mathbf{H}^{t-1}a^t(a^t)^\top \mathbf{H}^{t-1}a^t\|_2 \\
&\le \lambda_{2d}(\mathbf{H}^{t-1})^2 \|a^t\|_2^3 = O(d^{1.5}/s^3)
\end{aligned}$$

By triangle inequality we get

$$\|\mathbf{H}^t a^t - \mathbf{H}a^t\|_2 \le O(td^{1.5}/s^3) \le O(d^{2.5}/s^3).$$

In summary, we have

$$\|(x^t - x^{t-1}) - \mathbf{H}a^t\|_2 \le \|(x^t - x^{t-1}) - \mathbf{H}^t a^t\|_2 + \|mH^t a^t - \mathbf{H}a^t\|_2 = O(d^{2.5}/s^3).$$

So for all $1 \le i \le d$ we have

$$(x^t - x^{t-1})_i = (\mathbf{H}a^t)_i \pm O(d^{2.5}/s^3) = (1/(ds)) \cdot \mathbf{M}v \pm O(d^{2.5}/s^3).$$

So for s large enough, we can distinguish between zero and non-zero entries in $\mathbf{M}v$. □

8.3 Exercises

8.3.1 Reducing OuMv to OMv

In the lecture, we defined the OuMv-problem as follows:

- First, we are given a boolean $n \times n$ matrix $\mathbf{M} \in \{0,1\}^{n \times n}$ and are allowed to preprocess that matrix.
- After the preprocessing, we receive an online sequence of Boolean vectors $u_1, v_1, u_2, v_2, \ldots, u_n, v_n \in \{0,1\}^n$. We receive vectors u_{i+1}, v_{i+1} only after computing/returning $u_i^\top \mathbf{M} v_i \in \{0,1\}$.

The OMv-problem is similarly defined, but we receive only a sequence of vectors $v_1, ..., v_n$ and must return $\mathbf{M}v_i \in \{0,1\}^n$. For both the OMv- and OuMv-problem, all computation is performed over the Boolean semi-ring, hence $u_i^\top \mathbf{M} v_i \in \{0,1\}$ and $\mathbf{M}v_i \in \{0,1\}^n$.

Problem: Reduce the OMv-problem to the OuMv-problem. That is, show the following:
Assume there exists an algorithm for the OuMv-problem with preprocessing time $O(n^c)$ for some constant c, and $O(n^{3-\epsilon})$ total time for some constant $\epsilon > 0$ to compute all n vector-matrix-vector products (i.e., $u_i^\top \mathbf{M} v_i$).
Then show there exists an algorithm for the OMv-problem with preprocessing time $O(n^{c'})$ for some constant c', and $O(n^{3-\epsilon'})$ total time for some constant $\epsilon' > 0$ to compute all n matrix-vector products (i.e., $\mathbf{M}v_i$) — the constants c', ϵ' are allowed to be different from c, ϵ.

Hint: Split the matrix $\mathbf{M}$ into blocks of size $\sqrt{n} \times \sqrt{n}$. Adapt the proof of Theorem 12.1.2 and Lemma 12.1.3.

8.3.2 Lower Bound for Row and Column Updates

The OMv-conjecture implies that no algorithm exists for the OMv-problem with polynomial preprocessing time and $O(n^{3-\epsilon})$ time for computing all n matrix-vector products.

Problem: Show that, assuming the OMv-conjecture, there is no data structure with the following operations and complexities.

- INITIALIZE($\mathbf{M} \in \mathbb{F}^{n \times n}$): Preprocess matrix $\mathbf{M}$ in polynomial time.
- UPDATEROW($i \in \{1, ..., n\}, v \in \mathbb{F}^n$) and UPDATECOLUMN($i \in \{1, ..., n\}, v \in \mathbb{F}^n$): Change the ith row or column of $\mathbf{M}$ respectively to v in $O(n^{2-\epsilon})$ time for some constant $\epsilon > 0$.
- QUERY($i, j \in \{1, ..., n\}$): Return $(\mathbf{M}^{-1})_{i,j}$ in $O(n^{2-\epsilon})$ time for some constant $\epsilon > 0$.

You are allowed to use Problem 1's result, even if you have not solved it.

Part III

Approximation and Adaptivity

Chapter 9

Sketching and Subspace Embeddings

At the start of this course we covered least square regression, i.e. $\min_x \|\mathbf{A}x - b\|_2$. We said that this task can be solved by computing $x = (\mathbf{A}^\top \mathbf{A})^{-1}\mathbf{A}^\top b$ in $O(nd^2)$ time, or $O(nd^{\omega-1})$ time when using fast matrix multiplication.

In this chapter we will show that this task can be solved in $O(\text{nnz}(\mathbf{A}) + (d/\epsilon^2)^\omega)$ time (where $\text{nnz}(\mathbf{A})$ is the number of non-zeros in $\mathbf{A}$) if we allow for an approximation error, i.e. when our computed vector x satisfies $\|\mathbf{A}x - b\|_2 \le (1+\epsilon)\min_{x^*} \|\mathbf{A}x^* - b\|_2$. To prove this, we will use a technique called *Subspace embeddings*.

9.1 Subspace embedding

Definition 9.1.1. A matrix $\mathbf{S} \in \mathbb{R}^{k\times n}$ is a subspace embedding of $\mathbf{A} \in \mathbb{R}^{n\times d}$ if for all $v \in \mathbb{R}^d$

$$(1-\epsilon)\|\mathbf{A}v\|_2 \le \|\mathbf{SA}v\|_2 \le (1+\epsilon)\|\mathbf{A}v\|_2.$$

In this section we will prove the following lemma

Lemma 9.1.2. *Let $\mathbf{A} \in \mathbb{R}^{n\times d}$, $k = \widetilde{O}(d/\epsilon^2)$ and $\mathbf{S} \in \mathbb{R}^{k\times n}$ where each entry $\mathbf{S}_{i,j}$ is independently random $\pm 1/\sqrt{k}$. Then w.h.p $\mathbf{S}$ is a subspace embedding for $\mathbf{A}$, i.e. for all $v \in \mathbb{R}^d$ we have*

$$(1-\epsilon)\|\mathbf{A}v\|_2 \le \|\mathbf{SA}v\|_2 \le (1+\epsilon)\|\mathbf{A}v\|_2.$$

This allows us to approximately solve the least squares regression problem. To see this, consider

$$\mathbf{M} := \left[\begin{array}{c|c} \mathbf{A} & b \end{array}\right]$$

and let $\mathbf{S}$ be a subspace embedding for $\mathbf{M}$. Then for all $x \in \mathbb{R}^d$ we have

$$\|\mathbf{A}x - b\|_2 = \|\mathbf{M}\begin{bmatrix} x \\ -1 \end{bmatrix}\|_2 = (1\pm\epsilon)\|\mathbf{SM}\begin{bmatrix} x \\ -1 \end{bmatrix}\|_2 = (1\pm\epsilon)\|\mathbf{SA}x - \mathbf{S}b\|_2$$

In particular, if we let $x = \arg\min_x \|\mathbf{SA}x - \mathbf{S}b\|$, then

$$\|\mathbf{A}x - b\|_2 \leq (1+\epsilon) \min_{x^*} \|\mathbf{A}x^* - b\|_2$$

so we have an approximate solution. Since $\mathbf{SA}$ is just a $k \times d$ matrix, computing this approximate solution can be done much more efficiently.

Lemma 9.1.3. *Assume we already computed* $\mathbf{SA}$ *for some subspace embedding* $\mathbf{S} \in \mathbb{R}^{k \times n}$ *for matrix* $\mathbf{M}$ *and* $k = \widetilde{O}(d/\epsilon^2)$ *(e.g. as in Lemma 9.1.2). Then we can compute* $x = \arg\min_x \|\mathbf{SA}x - \mathbf{S}b\|$ *in* $O(k^\omega)$ *time.*

Proof. We compute $x = (\mathbf{A}^\top \mathbf{S}^\top \mathbf{SA})^{-1} \mathbf{A}^\top \mathbf{S}^\top \mathbf{S}b$ with can be done efficiently since $\mathbf{SA}$ is just of size $k = \widetilde{O}(d/\epsilon^2)$. □

Before we prove Lemma 9.1.2, we want to remind our-self of the following result (see Lemma 5.3.2), that is somewhat similar:

Lemma 9.1.4 (Johnson-Lindenstrauss). *Let* $v \in \mathbb{R}^n$ *and* $\mathbf{R} \in \mathbb{R}^{k' \times n}$ *for* $k = O(\epsilon^{-2} \log n)$ *with each entry* $\mathbf{R}_{i,j} = \pm 1/\sqrt{k'}$ *independent at random. Then w.h.p* $(1-\epsilon)\|v\|_2 \leq \|\mathbf{R}v\|_2 \leq (1+\epsilon)\|v\|_2$.

This lemma implies for any $w \in \mathbb{R}^d$ that w.h.p. $\|\mathbf{RA}w\|_2 = (1 \pm \epsilon)\|\mathbf{A}w\|_2$. This is similar to the subspace embedding Lemma 9.1.2 with one crucial difference. The random matrix $\mathbf{R}$ holds w.h.p for *one* vector w, whereas Lemma 9.1.2 holds w.h.p for *all* vectors $w \in \mathbb{R}^d$.

Thus an intuitive idea to prove Lemma 9.1.2 would be to simply use Lemma 9.1.4 and use union bound to bound the failure probability over all $w \in \mathbb{R}^d$. The problem is that there are infinitely elements in $\mathbb{R}^d$, so the union bound does not result in a finite failure probability. To prove Lemma 9.1.2, we will argue that the union bound over some $\exp(\widetilde{O}(d))$ vectors is enough. This is a common technique: quite often one can reduce union bound over $\mathbb{R}^d$ to a union bound over just $\exp(\widetilde{O}(d))$ many vectors.

Proof. Given $\mathbf{A}$ let $\mathbf{QR}$ be its QR-decomposition, i.e. the columns of $\mathbf{Q} \in \mathbb{R}^{n \times d}$ are orthogonal vectors with norm 1, and $\mathbf{R} \in \mathbb{R}^{d \times d}$ is an upper triangular matrix. For any $w \in \mathbb{R}^d$ let $h = \mathbf{R}w$, in which case we have $\|\mathbf{A}w\|_2 = \|\mathbf{QR}w\|_2 = \|\mathbf{Q}h\|_2 = \|h\|_2$ where the last step uses that $\mathbf{Q}$ has orthogonal columns of norm 1.

So to prove that $\mathbf{S}$ is a subspace embedding for $\mathbf{A}$, it suffice to prove that for all $h \in \mathbb{R}^d$ we have $\|\mathbf{SQ}h\| = (1 \pm \epsilon)\|\mathbf{Q}h\|_2$. We will do this via union bound, which first requires us to reduce the infinite number of $h \in \mathbb{R}^d$ to some finite subset.

Finite set of vectors For any $h \in \mathbb{R}^d$, we can assume without loss of generality that $\|h_i\|_2 = 1$ by simply scaling the vector. We can now define $\overline{h}$ via

$$\overline{h}_i = \lceil h_i \cdot \frac{\epsilon}{d} \rceil \cdot \frac{d}{\epsilon} \text{ for all } i$$

so $\overline{h}$ is just the vector h where we rounded each entry to the next largest multiple of ϵ/d. Note that $\|h - \overline{h}\|_2 \leq \epsilon$ and that by assumption $\|h_i\|_2 = 1$, so $\|\overline{h}\|_2 = (1 \pm \epsilon)\|h\|_2$. Let $S \subset \mathbb{R}^d$ be the set of all possible $\overline{h}$ vectors. We can bound $|S| \leq (2d/\epsilon)^d = \exp(\widetilde{O}(d))$ because there are only $2d/\epsilon$ options for each of the d entries.

Using the finite set to argue about all sets We will use union bound over this set S to argue that for all $\overline{h} \in S$ we have $\|\mathbf{SQ}\overline{h}\|_2 = (1 \pm \epsilon)\|\mathbf{Q}\overline{h}\|_2$. This will then imply that $\mathbf{S}$ is a valid subspace embedding for $\mathbf{A}$ as follows: For any h with $\|h\|_2 = 1$ and corresponding $\overline{h}$ we have

$$\begin{aligned}
\|\mathbf{SQ}h\|_2 &= \|\mathbf{SQ}\overline{h}\|_2 \pm \sum_{i=1}^{d} \|\mathbf{SQ}e_i\| \cdot |h_i - \overline{h}_i| \\
&= (1 \pm \epsilon)\|\mathbf{Q}\overline{h}\|_2 \pm (1 \pm \epsilon) \sum_{i=1}^{d} \|\mathbf{Q}e_i\| \cdot \epsilon/d \\
&= (1 \pm \epsilon)\|\overline{h}\|_2 \pm (1 \pm \epsilon) \sum_{i=1}^{d} \|e_i\| \cdot \epsilon/d \\
&= (1 \pm O(\epsilon))\|\overline{h}\|_2 \\
&= (1 \pm O(\epsilon))\|h\|_2 \\
&= (1 \pm O(\epsilon))\|\mathbf{Q}h\|_2
\end{aligned}$$

Here the first step uses triangle inequality. The 2nd step uses that $\|\mathbf{SQ}\overline{h}\|_2$ approximates $\|\mathbf{Q}\overline{h}\|_2$ for every possible $\overline{h} \in S$, which includes the standard unit vectors e_i. The 3rd and last step use that $\mathbf{Q}$ is has orthogonal columns with norm 1. The 4th and 5th step use that $\|\overline{h}\|_2 = (1 \pm \epsilon)\|h\|_2 = 1$.

So for any w where $h = \mathbf{R}w$ has $\|h\|_2 = 1$ let $\overline{h}$ be the respective rounding of h. Then

$$\|\mathbf{SA}v\|_2 = \|\mathbf{SQ}h\|_2 = (1 + O(\epsilon))\|\mathbf{Q}h\|_2 = (1 + O(\epsilon))\|\mathbf{A}v\|_2$$

By simply scaling the vector w, the result extends to all $w \in \mathbb{R}^d$ even when $\|h\|_2 \neq 1$.

Union Bound over the finite set Note that in expectation, the matrix $\mathbf{S}$ does return the correct norm. For any $v \in \mathbb{R}^n$ (e.g. $v = \mathbf{Q}\overline{h}$) we have

$$\begin{aligned}
\mathbb{E}[\|\mathbf{S}v\|_2] &= \mathbb{E}[\sum_{i=1}^{k} (\mathbf{S}v)_i^2] \\
&= \mathbb{E}[\sum_{i=1}^{k} (\sum_{j=1}^{n} \mathbf{S}_{i,j} v_j)^2] \\
&= \mathbb{E}[\sum_{i=1}^{k} (\sum_{j=1}^{n} \mathbf{S}_{i,j}^2 v_j^2 + \sum_{j \neq k} \mathbf{S}_{i,j}\mathbf{S}_{i,k} v_j v_k)] \\
&= \sum_{i=1}^{k} (\sum_{j=1}^{n} \mathbb{E}[\mathbf{S}_{i,j}^2] v_j^2 + \sum_{j \neq k} \mathbb{E}[\mathbf{S}_{i,j}] \cdot \mathbb{E}[\mathbf{S}_{i,k}] v_j v_k) \\
&= \sum_{i=1}^{k} (\sum_{j=1}^{n} \frac{1}{k} v_j^2) = \|v\|_2^2
\end{aligned}$$

We can argue via Chernoff-bound that this is also highly concentrated around this expectation. In particular, via Chernoff bound the failure probability (i.e. the norm not being a $(1\pm\epsilon)$-approximation) is bounded by $\exp(-\widetilde{O}(k\epsilon^2))$. As we want the failure probability to be small enough to allow for a union bound over $\exp(\widetilde{O}(d))$ vectors, we pick $k = \widetilde{O}(d/\epsilon^2)$. □

Note that computing $\mathbf{SA}$ to, e.g., solve the least squares regression problem, takes a lot of time. We have a dense matrix $\mathbf{S}$ of size $k \times n$ for $k = \widetilde{O}(d/\epsilon^2)$. So computing $\mathbf{SA}$ is not any faster than computing $\mathbf{A}^\top\mathbf{A}$. Thus we do not save anything when computing $x = (\mathbf{A}^\top\mathbf{S}^\top\mathbf{SA})^{-1}\mathbf{A}^\top\mathbf{S}^\top\mathbf{S}b$ vs $x = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b$. There is, however, a more efficient construction for $\mathbf{S}$, proven by Clarkson and Woodruff (2013) [CW13].

Lemma 9.1.5 ([CW13]). *Let $\mathbf{A} \in \mathbb{R}^{n\times d}$ and let $\mathbf{S} \in \mathbb{R}^{k\times n}$ for $k = \widetilde{O}(d/\epsilon^2)$ constructed as follows:*
For each $i \in \{1, ..., n\}$ set one random entry of the ith column of $\mathbf{S}$ to ± 1 at random.
Then w.h.p. $\mathbf{S}$ is a subspace embedding for $\mathbf{A}$.
Note that by sparsity of $\mathbf{S}$ we can compute $\mathbf{SS}$ in $O(nnz(\mathbf{A}))$ time.

We will not prove this here, but give an idea what additional property their proof uses. In our proof, we argued that $\mathbf{S}$ approximates the norm of some vector $v \in \mathbb{R}^n$ very well with failure probability $\exp(-\widetilde{O}(k\epsilon))$. Then we used union bound over $\exp(\widetilde{O}(d)$ such vectors. However, we don't have a general set of $\exp(\widetilde{O}(d))$ such vectors from $\mathbb{R}^n$. We have a set S of vectors $\overline{h}$ and only need to perform union bound over the $\exp(\widetilde{O}(d))$ vectors of form $\|\mathbf{Q}\overline{h}\|$. In other words, we do not have a *general set* of $\exp(\widetilde{O}(d))$ vectors, but a set of $\exp(\widetilde{O}(d))$ vectors *that all lie within the subspace spanned by* $\mathbf{Q}$. Such a set has additional structure that is exploited in the proof of [CW13]

9.2 Leverage Scores

In the previous lecture we defined and constructed subspace embeddings, i.e. random matrices $\mathbf{S}$ with the property that for all vector $v \in \mathbb{R}^d$ we can approximate $\|\mathbf{A}v\|_2 = (1 \pm \epsilon)\|\mathbf{SA}v\|_2^2$. The constructions we discussed so far worked w.h.p for any matrix $\mathbf{A}$. In this chapter, we discuss a construction of $\mathbf{S}$ that is tailored to the matrix $\mathbf{A}$ that we want to embed.

The subspace embedding discussed in this chapter has the additional property that it just samples the rows of $\mathbf{A}$. That is, $\mathbf{S}$ is a diagonal matrix where most entries will be 0 and only some $\widetilde{O}(d)$ entries are non-zero. When applied to the least-squares regression problem, this means that we reduce our n-point data set to a smaller $\widetilde{O}(d)$-point data set (i.e. we reduce n rows of $\mathbf{A}$ to only $\widetilde{O}(d)$ rows). In a sense, that means we figure out which $\widetilde{O}(d)$ data points of the n-points data set are important, and then restrict to those important points.

In a future lecture, we want to prove that dynamic least squares can be solved in $\widetilde{O}(d)$ time per new row inserted to $\mathbf{A}$. Our new subspace embedding helps us with that, because it implies that only few rows of $\mathbf{A}$ are important. If a new row inserted into $\mathbf{A}$ is not important, then we do not need to update our solution to the least-squares regression problem. Thus such a row insertion is essentially free. We will prove in a future lecture that this implies a good amortized complexity. The high-level idea being that not every inserted row can be important and thus on average we do not need to spend a lot of time to update the solution.

To formalize these ideas, we first must define some measure of "importance". The following definition can be seen as an importance-score for each row of $\mathbf{A}$ (e.g. each data point of the least-squares regression problem). In statistics, these scores is referred to as *leverage scores*.

Definition 9.2.1 (Leverage Scores)**.** For $\mathbf{A} \in \mathbb{R}^{n\times d}$ we define $\sigma(\mathbf{A}) \in \mathbb{R}^n$ as $\sigma(\mathbf{A})_i = (\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top)_{i,i}$.

To give some intuition why these leverage scores measure the important of a data point, consider the following example.

We take our data set, and move one point along the y-axis (todo, figure). If that data point is "important" (or "unique", "influential" etc.) then we would expect our solution to the least-squares regression problem to change a lot. On the other hand, is the point unimportant because it has, e.g., many other data points nearby, then the solution does not change a lot. This interpretation also gives some motivation for the term "*leverage score*" because important points behave like a lever moving the linear function in Figure **??** **Jan: todo, figure**.

The following lemma formalizes the observation from Figure **??** **Jan: todo**. The lemma shows that the quality of a solution changes exactly by $\sigma(\mathbf{A})_i$ when moving the ith point.

Lemma 9.2.2. *Let* $\mathbf{A} \in \mathbb{R}^{n\times d}$*,* $b, b' \in \mathbb{R}^n$*,* $\epsilon > 0$*,* $b' = b + e_i \cdot \epsilon$*. Let* $x' = \arg\min \|\mathbf{A}x - b'\|_2$*, then*

$$\|\mathbf{A}x' - b\|_2^2 = \sigma(\mathbf{A})_i \cdot \epsilon^2 + \min_x \|\mathbf{A}x - b\|_2.$$

Proof.

$$\begin{aligned}
\|\mathbf{A}x' - b\|_2^2 &= \|\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b' - b\|_2^2 \\
&= \|\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b - b + \mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top e_i\epsilon\|_2^2 \\
&= \|\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b - b\|_2^2 + \|\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top e_i\epsilon\|_2^2
\end{aligned}$$

Here the last step comes from the Pythagorean theorem and uses that the two vectors are orthogonal. To see this, note that $\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top e_i\epsilon$ is in the span of $\mathbf{A}$ and $(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b - b$ is orthogonal to the span of $\mathbf{A}$ since

$$\mathbf{A}^\top((\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b - b) = 0.$$

At last, note that

$$\|\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top b - b\|_2^2 = \min_x \|\mathbf{A}x - b\|_2^2$$

and

$$\begin{aligned}
\|\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top e_i\epsilon\|_2^2 &= \epsilon^2 \cdot e_i^\top \mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top \ \mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top e_i \\
&= \epsilon^2 \cdot e_i^\top \mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top e_i \\
&= \epsilon^2 \cdot (\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top)_{i,i} \\
&= \epsilon^2 \cdot \sigma(\mathbf{A})_i.
\end{aligned}$$

□

We now show, that just sampling each row proportional to the leverage score result w.h.p in a subspace embedding.

Theorem 9.2.3. *There exists constant $c > 0$ such that for any $\mathbf{A} \in \mathbb{R}^{n\times d}$ and $p_i \geq \min\{1, c \cdot \epsilon^{-2}\sigma(\mathbf{A})_i \log n\}$ the following holds:*

Let $\mathbf{S} \in \mathbb{R}^{n\times n}$ be a random matrix with $\mathbf{S}_{i,i} = 1/\sqrt{p_i}$ independently with probability p_i and $\mathbf{S}_{i,i} = 0$ otherwise. Then w.h.p. for all $v \in \mathbb{R}^d$ we have $\|\mathbf{SA}v\|_2^2 = (1 \pm \epsilon)\|\mathbf{SA}\|_2^2$, i.e. $\mathbf{S}$ is a subspace embedding for $\mathbf{A}$. Further, $\mathbf{S}$ has at most $\widetilde{O}(d/\epsilon^2)$ non-zero entries.

To prove this, we will use the following Matrix-Chernoff bound. It is the generalization of the Chernoff bound to the sum of random matrices.

Lemma 9.2.4 (Matrix-Chernoff)**.** *Finite sequence $X_1, ..., X_T$ of independent random matrices with $0 \prec \mathbf{X}_k$ and $\lambda_{\max}(\mathbf{X}_k) \leq K$ for all $1 \leq k \leq T$. $\mu_{\max} := \lambda_{\max}(\sum_k \mathbb{E}\mathbf{X}_k)$, $\mu_{\min} := \lambda_{\min}(\sum_k \mathbb{E}\mathbf{X}_k)$.*

$$\mathbb{P}\left[\lambda_{\max} \sum_k \mathbf{X}_k \geq (1+\epsilon)\mu_{\max}\right] \leq d \cdot \exp(-\epsilon^2 \mu_{\max}/(3K))$$

$$\mathbb{P}\left[\lambda_{\min} \sum_k \mathbf{X}_k \leq (1-\epsilon)\mu_{\min}\right] \leq d \cdot \exp(-\epsilon^2 \mu_{\min}/(3K))$$

Proof of Theorem 9.2.3. Note that the Matrix-Chernoff bound only bounds the largest and smallest eigenvalue, but Theorem 9.2.3 is supposed to hold for all vectors, not just the eigenvectors of the largest and smallest eigenvalue.

To prove the result for all vectors, we shift the problem a bit so that the largest and smallest eigenvalue are both 1. This way, all vectors are eigenvectors of the same eigenvalue, so the Matrix-Chernoff bound applies to all vectors.

Shift Let $\mathbf{M} := \mathbf{A}^\top\mathbf{A}$, then $\mathbf{S}$ is a valid subspace embedding for $\mathbf{A}$, if for all $v \in \mathbb{R}^d$

$$v^\top \mathbf{I} v = v^\top \mathbf{M}^{-1/2}\mathbf{A}^\top\mathbf{A}\mathbf{M}^{-1/2} v = (1 \pm \epsilon) v^\top \mathbf{M}^{-1/2}\mathbf{A}^\top\mathbf{S}^2\mathbf{A}\mathbf{M}^{-1/2} v$$

Note that here $\mathbf{M}^{-1/2}\mathbf{A}^\top\mathbf{S}^2\mathbf{A}\mathbf{M}^{-1/2}$ can be seen as the sum of random matrices $\mathbf{X}_k$ where $\mathbf{X}_k$ is defined as

$$\mathbf{X}_k = \begin{cases} \frac{1}{p_k}(\mathbf{M}^{-1/2}\mathbf{A}^\top e_k)(e_k^\top \mathbf{A}\mathbf{M}^{-1/2}) & \text{with probability } p_k \\ 0 & \text{otherwise} \end{cases}.$$

Note that this is just sampling the outer product of the kth row of $\mathbf{A}\mathbf{M}^{-1/2}$ with itself. We now want to use Chernoff-bound on this sum of random $\mathbf{X}_k$. For this we first want to bound $\lambda_{\max}(\mathbf{X}_k) \leq K := \Theta(\epsilon^2/\log n)$. This holds because for all $v \in \mathbb{R}^d$

$$\begin{aligned} p_k^{-1} v^\top (\mathbf{M}^{-1/2}\mathbf{A}^\top e_k)(e_k^\top \mathbf{A}\mathbf{M}^{-1/2}) v \leq p_k^{-1}\|\mathbf{M}^{-1/2}\mathbf{A}^\top e_k\|_2^2 \|v\|_2^2 = p_k^{-1}(\mathbf{A}\mathbf{M}^{-1}\mathbf{A}^\top)_{k,k}\|v\|_2^2 \\ = p_k^{-1}\sigma(\mathbf{A})_k \|v\|_2^2 \leq \|v\|_2^2 \end{aligned}$$

where we used that $p_k \geq \sigma(\mathbf{A})_k \cdot \Theta(\epsilon^{-2}\log n)$. Further, we have

$$\mu_{\max} := \lambda_{\max}(\sum_k \mathbb{E}\mathbf{X}_k) = \lambda_{\max}(\mathbf{M}^{-1/2}\mathbf{A}^\top\mathbf{A}\mathbf{M}^{-1/2}) = \lambda_{\max}(\mathbf{I}) = 1$$

and the same for $\mu_{\min} := \lambda_{\min}(\sum_k \mathbb{E}\mathbf{X}_k) = 1$. By Chernoff we can thus write

$$\begin{aligned}\mathbb{P}\left[\lambda_{\max}\sum_k \mathbf{X}_k \geq (1+\epsilon)\mu_{\max}\right] &\leq d\cdot\exp(-\epsilon^2\mu_{\max}/(3R)) \\ &= d\cdot\exp(-\epsilon^2 O(\epsilon^{-2}/\log n)) \\ &= 1/\text{poly}(n)\end{aligned}$$

and the same for $\mathbb{P}[\lambda_{\min}\sum_k \mathbf{X}_k \leq (1-\epsilon)\mu_{\min}]$. Thus we have w.h.p. that for all $v \in \mathbb{R}^d$

$$v^\top\mathbf{M}^{-1/2}\mathbf{A}^\top\mathbf{S}^2\mathbf{A}\mathbf{M}^{-1/2} = (1\pm\epsilon)v^\top\mathbf{I}v$$

which implies that w.h.p for all $v \in \mathbb{R}^d$

$$v^\top\mathbf{A}^\top\mathbf{S}^2\mathbf{A} = (1\pm\epsilon)v^\top\mathbf{M}v = (1\pm\epsilon)v^\top\mathbf{A}^\top\mathbf{A}v.$$

□

The following lemma implies that w.h.p the leverage score sampling from Theorem 9.2.3 samples at most $\widetilde{O}(d)$ rows of $\mathbf{A}$.

Lemma 9.2.5. *For any full-rank $\mathbf{A}$ in$\mathbb{R}^{n\times d}$ we have $\sum_{i=1}^n \sigma(\mathbf{A})_i = d$.*

Proof.

$$\begin{aligned}\sum_{i=1}^n \sigma(\mathbf{A})_i &= \sum_{i=1}^n (\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top)_{i,i} = \text{tr}(\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top) \\ &= \text{tr}(\mathbf{A}^\top\mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}) = \text{tr}(\mathbf{I}_{d\times d}) = d\end{aligned}$$

Here we used the cyclic property of the trace, i.e. $\text{tr}(\mathbf{MN}) = \text{tr}(\mathbf{NM})$. □

Graph Application, Example The property that our leverage score sampling yields a subspace embedding that is just a submatrix of the original $\mathbf{A}$ also has some uses outside of matrices.

Given a (possibly weighted) graph $G = (V, E, c)$ with edge weights $(c_e)_{e\in E}$, we call the subgraph $H = (V, E', c')$ (with possibly different edge weights $c'_e \neq c_e$) a cut sparsifier if the following holds: For every cut $S\dot{\cup}T = V$, the size of the cut between S and T is the same in G and H up to some $(1+\epsilon)$-factor. That is

$$\sum_{\{u,v\}\in E, u\in S, v\in T} c_{u,v} = (1\pm\epsilon)\cdot \sum_{\{u,v\}\in E', u\in S, v\in T} c'_{u,v}$$

Corollary 9.2.6. *Let $G = (V, E, c)$ be a graph and let $\mathbf{A} \in \mathbb{R}^{E \times V}$ be a weighted incidence matrix with $\mathbf{A}_{\{u,v\},u} = \pm\sqrt{c_{u,v}}$ so that $\mathbf{A}^\top \mathbf{A}$ is the weighted Laplacian of G. Let H be the graph obtained by sampling each edge $e \in E$ with probability $p_e = \min\{1, O(\epsilon^{-2}\sigma(\mathbf{A})_i \log n)\}$ and setting its edge weight to c_e/p_e if the edge is included in H.*

Then w.h.p H is a $(1+\epsilon)$-approximate cut-sparsifier of G with $\widetilde{O}(n/\epsilon^2)$ many edges.

Proof. A cut $S \dot\cup T = V$ can be identified with the vector $w \in \mathbb{R}^V$ with $w_v = 1$ for $v \in S$ and $w_v = 0$ otherwise. Then the number of edges being cut is given by

$$\sum_{\{u,v\} \in E, u \in S, v \in T} c_{u,v} = \sum_{e \in E} (\mathbf{A}w)_e^2 = \|\mathbf{A}w\|_2^2$$

Since the sampling procedure resulting in H is exactly the leverage score sampling for sparsifying the matrix $\mathbf{A}$, the graph H approximates every cut.

Remark: Technically the incidence matrix is not full rank, i.e. $\mathbf{A}^\top \mathbf{A}$ is not invertible. One can extend the notion of leverage scores and sampling thereof to non-invertible matrices by using the Penrose-pseudoinverse. □

This proof relies on $\mathbf{SA}$ being an incidence matrix when $\mathbf{A}$ is an incidence matrix. Since leverage score sampling yields a submatrix, the subspace embedding we constructed in this section is also an incidence matrix. In comparison, this property was not given by the subspace embeddings from the previous chapter. There each row of $\mathbf{SA}$ is some linear combination of different rows of $\mathbf{A}$ which breaks the incidence matrix structure.

Chapter 10

Dynamic Approximate Least Squares

In Theorem 8.2.7 we showed that, unless the OMv-conjecture is wrong, no data structure can maintain exact dynamic least squares in less than $O(d^2)$ time per update. Since the lower bound holds only for the exact case, we now consider the approximate case. We show that when allowing for $(1+\epsilon)$-approximation, one can solve dynamic regression in $\widetilde{O}(d)$ amortized time per insertion.

Theorem 10.0.1 ([JPW22]). *There exists a data structure with the following operations.*

1. *Init($A^{(0)} \in \mathbb{R}^{d\times d}, b^{(0)} \in \mathbb{R}^d$)*
2. *Insert($a \in \mathbb{R}^d, \beta \in \mathbb{R}$): it sets*

$$A^{(t+1)} \leftarrow \begin{bmatrix} A^{(t)} \\ a^\top \end{bmatrix}, \qquad b^{(t+1)} \leftarrow \begin{bmatrix} b^{(t)} \\ \beta \end{bmatrix},$$

and returns x with

$$\left\| A^{(t+1)}x - b^{(t+1)} \right\|_2 \leq (1+\epsilon)\cdot \min_{x^*} \left\| A^{(t+1)}x^* - b^{(t+1)} \right\|_2$$

Moreover the total running time of the data structure is $\widetilde{O}(nnz(A^{(T)}) + \frac{d^3}{\epsilon^2}\log(TD/L))$ in expectation, where T is the number of insertions, L is a lower bound on the smallest singular value of the initial matrix and $D/2$ is an upper bound on the ℓ_2 norm of any row of the matrix $A^{(T)}$, and absolute value of any entry of the vector $b^{(T)}$.

We start by outlining the idea of the data structure. Remember that in Section 9.1 (Lemma 9.1.3) we explained how to efficiently solve approximate regression in the static setting using subspace embeddings. One such subspace embedding was obtained via leverage scores (Theorem 9.2.3, Section 9.2), i.e. a sparse diagonal matrix S was constructed with the property that $\|SAv\| \approx \|Av\|$ for all vectors v. Then the approximate regression could be solved (Section 9.1) by computing $(A^\top S^2 A)^{-1} A^\top S b$. Now consider the dynamic

setting: we add a new row to A which means size of diagonal matrix S increases by 1. Let A', S' be the new larger matrices. We will show that S' is a valid subspace embedding if we just reuse the old diagonal entries of S and only sample the new diagonal entry of S'. If that new diagonal entry of S' is 0, then there is no need to compute anything since $A^\top S^2 A = A'^\top S'^2 A'$ and $A^\top S b = A'^\top S' b'$. Only if the new diagonal entry of S' is non-zero do we need to update the solution. We will show that getting a new non-zero entry to S' does not happen to often (on average only once every $O(d/\epsilon^2)$ updates), which then leads to the small amortized update time of Theorem 10.0.1.

10.0.1 Preliminaries

Here we list a few definitions and observations that will be useful for formally proving the data structure outlined in the previous subsection.

Definition 10.0.2 (Spectral approximation)**.** Let M, N be positive semi-definite matrices. Then we say M is a ϵ-spectral approximation of N, and denote it by $M \approx_\epsilon N$, if for all vectors v, $(1-\epsilon)v^\top N v \leq v^\top M v \leq (1+\epsilon)v^\top N v$. One can show that if $M \approx_\epsilon N$, then $N^{-1} \approx_\epsilon M^{-1}$.

The notion of spectral approximation is closely related to subspace embeddings.

Observation 10.0.3. *Let S be a subspace embedding (Definition 9.1.1) of A with $(1-\epsilon)\left\|Av\right\|_2 \leq \left\|SAv\right\|_2 \leq (1+\epsilon)\left\|Av\right\|_2$. Then $A^T S^T S A \approx_\epsilon A^T A$.*

Subspace embeddings (and spectral approximations) allow us to approximate the leverage scores (Definition 9.2.1)

Lemma 10.0.4. *Let S be a subspace embedding for A. Then $(1-\epsilon)\sigma(A)_i \leq (A(A^\top S^\top S A)A)_{ii} \leq (1+\epsilon)\sigma(A)_i$, where $\sigma(A)_i := (A(A^\top A)A^\top)_{ii}$.*

Proof. We have

$$(A(A^\top S^\top S A)A)_{ii} = (e_i^\top A)(A^\top S^\top S A)(A^\top e_i).$$

Therefore by assumption,

$$(1-\epsilon)(e_i^\top A)(A^\top A)(A^\top e_i) \leq (A(A^\top S^\top S A)A)_{ii} \leq (1+\epsilon)(e_i^\top A)(A^\top A)(A^\top e_i).$$

Finally note that $(e_i^\top A)(A^\top A)(A^\top e_i) = (A(A^\top A)A^\top)_{ii} = \sigma(A)_i$. □

We can write the leverage scores as the following norm.

Lemma 10.0.5. $\left\|A(A^\top A)^{-1} A e_i\right\|_2^2 = \sigma(A)_i$.

Proof. We have

$$\begin{aligned}
\left\|A(A^\top A)^{-1} A^\top e_i\right\|_2^2 &= e_i^T A(A^\top A)^{-1} A^\top A(A^\top A)^{-1} A^\top e_i \\
&= e_i^\top A(A^\top A)^{-1} A^\top e_i \\
&= \sigma(A)_i.
\end{aligned}$$

□

To approximate a norm, we do not need a full subspace embedding. Just a JL-matrix (Lemma 9.1.4) does already suffice. So we can efficiently compute approximate leverage scores as follows:

Corollary 10.0.6 (Johnson-Lindenstrauss projection for leverage score computation)**.** *Let R be a random k-by-n matrix where each entry is uniformly picked from the set $\{-\frac{1}{\sqrt{k}}, +\frac{1}{\sqrt{k}}\}$ and $k = O(\frac{1}{\epsilon^2}\log(n))$. Then*

$$(1-\epsilon)\cdot\sigma(A)_i \leq \left\|RA(A^\top A)^{-1}A^\top e_i\right\|_2^2 \leq (1+\epsilon)\cdot\sigma(A)_i.$$

Moreover given $A(A^\top A)^{-1}A^\top$, $RA(A^\top A)^{-1}A^\top$ can be computed with $\widetilde{O}(\frac{1}{\epsilon^2})$ matrix-vector products.

Proof. The result easily follows from noting that a random Johnson-Lindenstrauss projection preserves the norm approximately for all vectors with high probability, i.e., $(1-\epsilon)\|v\| \leq \|Rv\|_2 \leq (1+\epsilon)\|v\|$, for any v with a probability of at least $1 - 1/\text{poly(n)}$. □

Lemma 10.0.7. *Given a matrix A, we can compute $\widetilde{\sigma}(A) \in \mathbb{R}^n$ in $\widetilde{O}(nnz(A) + \frac{d^\omega}{\epsilon^2})$ time, where for all $i \in [n]$,*

$$(1-\epsilon)\sigma(A)_i \leq \widetilde{\sigma}(A)_i \leq (1+\epsilon)\sigma(A)_i$$

Proof. Compute SA in $\widetilde{O}(\text{nnz}(A))$ time for **S** as in Lemma 9.1.5. Compute $(A^\top S^\top SA)^{-1}$ in $\widetilde{O}(\frac{d^\omega}{\epsilon^2})$ time. Compute $RSA(A^\top S^\top SA)^{-1}$ in $\widetilde{O}(\text{nnz}(A) + d^2)$ time. For all $i \in [n]$, compute $\left\|RSA(A^\top S^\top SA)^{-1}A^\top e_i\right\|_2^2$ in total $\widetilde{O}(n)$ time. Note that

$$(1-\epsilon)\sigma(A)_i \leq \left\|RSA(A^\top S^\top SA)^{-1}A^\top e_i\right\|_2^2 \leq (1+\epsilon)\sigma(A)_i$$

□

10.1 Main Result

We denote the matrix and the vector after t insertions with $A^{(t)}$ and $b^{(t)}$, respectively. Moreover, we denote the leverage score sampling matrix defined as the following with $S^{(t)}$,

$$S_{ii}^{(t)} = \begin{cases} \frac{1}{\sqrt{p_i^{(t)}}} & \text{with probability } p_i^{(t)}, \\ 0 & \text{otherwise.} \end{cases}$$

The data structure of Theorem 10.0.1 works as Algorithm 2. The next lemma give bounds on the running time of operations of this data structure.

Lemma 10.1.1. *In Algorithm 2, Step 4 of the algorithms runs in time $\widetilde{O}(nnz(a))$. If $S_{t+1,t+1}^{(t+1)} \neq 0$, Step 5 takes $\widetilde{O}(d^2)$ time, and Step 6 of the algorithm takes $\widetilde{O}(d^2)$ time.*

Algorithm 2: Sampling algorithm after $t+1$ insertions

1 **parameters**

2 $1 > \epsilon > 0$, c,$t = 0$, $A^{(0)} \in \mathbb{R}^{d\times d}$, $b^{(0)} \in \mathbb{R}^d$, $M^{(0)} = R^{(0)} A^{(0)} ((A^{(0)})^\top A^{(0)})^{-1}$, where R is a k-by-n matrix where each entry is uniformly picked from the set $\{-\frac{1}{\sqrt{k}}, +\frac{1}{\sqrt{k}}\}$, $k = O(\log(n))$ so that norms with R is preserved within $1 \pm \frac{1}{10}$ factor, n is the total number of rows after all insertions, and $R^{(0)}$ is the matrix obtained from R by taking the first d columns. $N^{(0)} = ((A^{(0)})^\top A^{(0)})^{-1}$ and set $S^{(0)}$ to the d-by-d identity matrix. $x^{(0)} = ((A^{(0)})^\top A^{(0)})^{-1} (A^{(0)})^\top b^{(0)}$.

3 **procedure** INSERTION($a \in \mathbb{R}^d, \beta \in \mathbb{R}_0$)

4 Given the new row a of $A^{(t+1)}$ and new entry β of $b^{(t+1)}$ (i.e., $A^{(t+1)}_{t+1} = a$ and $b^{(t+1)}_{t+1} = \beta$), set $S^{(t+1)}$ to be a diagonal matrix where its $t \times t$ leading principle submatrix is equal to $S^{(t)}$ and $S^{(t+1)}_{t+1,t+1} = \frac{1}{\sqrt{p}}$ with probability

$$p = \min\{1, (c \cdot \epsilon^{-2} \log n) \cdot \frac{100}{81} \left\| M^{(t)} a \right\|_2^2\},$$

and set $S^{(t+1)}_{t+1,t+1} = 0$, otherwise.

5 If $S^{(t+1)}_{t+1,t+1} \neq 0$, compute a new inverse $N^{(t+1)} = ((A^{(t+1)})^\top (S^{(t+1)})^2 A^{(t+1)})^{-1}$, and a new sketch as

$$M^{(t+1)} = R^{(t+1)} S^{(t+1)} A^{(t+1)} ((A^{(t+1)})^\top (S^{(t+1)})^2 A^{(t+1)})^{-1},$$

where $R^{(t+1)}$ is the matrix obtained from R by taking the first $t+1$ columns. If $S^{(t+1)}_{t+1,t+1} = 0$, set $M^{(t+1)} = M^{(t)}$.

6 If $S^{(t+1)}_{t+1,t+1} \neq 0$, compute the new solution vector as

$$x^{(t+1)} = ((A^{(t+1)})^\top (S^{(t+1)})^2 A^{(t+1)})^{-1} (A^{(t+1)})^\top (S^{(t+1)})^2 b^{(t+1)},$$

and if $S^{(t+1)}_{t+1,t+1} = 0$, set $x^{(t+1)} = x^{(t)}$.

Proof. Note that if $S^{(t+1)}_{t+1,t+1} \neq 0$, then by Sherman-Morrison identity Lemma 1.1.3, we have,

$$\begin{aligned}
M^{(t+1)} &= R^{(t+1)} S^{(t+1)} A^{(t+1)} ((A^{(t+1)})^\top (S^{(t+1)})^2 A^{(t+1)})^{-1} \\
&= (R^{(t)} S^{(t)} A^{(t)} + c \cdot r a^\top)(N^{(t)} - \frac{c^2 \cdot N^{(t)} a a^\top N^{(t)}}{1 + c^2 \cdot a^\top N^{(t)} a}) \\
&= M^{(t)} + c \cdot r a^\top N^{(t)} - \frac{c^2 \cdot M^{(t)} a a^\top N^{(t)}}{1 + c^2 \cdot a^\top N^{(t)} a} - \frac{c^3 \cdot r a^\top N^{(t)} a a^\top N^{(t)}}{1 + c^2 \cdot a^\top N^{(t)} a} \\
&= M^{(t)} + c \cdot r a^\top N^{(t)} - \frac{c^2 \cdot M^{(t)} a a^\top N^{(t)}}{1 + c^2 \cdot a^\top N^{(t)} a} - \frac{(c^3 a^\top N^{(t)} a) \cdot r a^\top N^{(t)}}{1 + c^2 \cdot a^\top N^{(t)} a}
\end{aligned}$$

where $N^{(t)} = ((A^{(t)})^\top (S^{(t)})^2 A^{(t)})^{-1}$, and r is the last column of $R^{(t+1)}$. Since $N^{(t)}$ is a d-by-d matrix, we can compute $a^\top N^{(t)} a$ in $O(d^2)$ time. Moreover, we have access to $M^{(t)}$.

For the rest of the terms above, we first compute $a^\top N^{(t)}$ (in $O(d^2)$ time), which is a d-vector, and then we multiply that with r, which can be done in $\widetilde{O}(d)$ time. Moreover by Sherman-Morrison identity Lemma 1.1.3, we can compute

$$N^{(t+1)} = N^{(t)} - \frac{c^2 \cdot N^{(t)} a a^\top N^{(t)}}{1 + c^2 \cdot a^\top N^{(t)} a},$$

by first computing $a^\top N^{(t)}$. Now we have

$$\begin{aligned}
x^{(t+1)} &= ((A^{(t+1)})^\top (S^{(t+1)})^2 A^{(t+1)})^{-1} (A^{(t+1)})^\top (S^{(t+1)})^2 b^{(t+1)} \\
&= (N^{(t)} - \frac{c^2 \cdot N^{(t)} a a^\top N^{(t)}}{1 + c^2 \cdot a^\top N^{(t)} a})((A^{(t)})^\top (S^{(t)})^2 b^{(t)} + \beta c^2 \cdot a) \\
&= x^{(t)} + \beta c^2 \cdot N^{(t)} a - \frac{c^2 \cdot N^{(t)} a a^\top x^{(t)}}{1 + c^2 \cdot a^\top N^{(t)} a} - \frac{\beta c^4 a^\top N^{(t)} a \cdot N^{(t)} a}{1 + c^2 \cdot a^\top N^{(t)} a} \\
&= x^{(t)} + \beta c^2 \cdot N^{(t)} a - \frac{c^2 a^\top x^{(t)} \cdot N^{(t)} a}{1 + c^2 \cdot a^\top N^{(t)} a} - \frac{\beta c^4 a^\top N^{(t)} a \cdot N^{(t)} a}{1 + c^2 \cdot a^\top N^{(t)} a}.
\end{aligned}$$

We can compute $a^\top x$ and $a^\top N^{(t)} a$ in $O(d)$ and $O(d^2)$ time, respectively. Then, the rest of the computation can be carried out in $O(d^2)$ time by first computing $N^{(t)} a$. Since $M^{(t)}$ has $O(\log n)$ rows, we can compute $M^{(t)} a$ and its norm in $\widetilde{O}(\text{nnz}(a))$ time. □

The next lemma gives a bound on the sum of leverage score estimations we use in Algorithm 2.

Lemma 10.1.2. *Let T be the number of insertions, L be a lower bound on the smallest singular value of the initial matrix $A^{(0)}$ and $D/2$ be an upper bound on the ℓ_2 norm of any row of the matrix $A^{(T)}$, and absolute value of any entry of the vector $b^{(T)}$. Then*

$$\sum_{t=1}^{T} \sigma^{A^{(t-1)}}(A^{(t)})_{d+t} \le O(d \log(TD/L)),$$

where

$$\sigma^{A^{(t-1)}}(A^{(t)})_{d+t} = \min\{1, (a^{(t)})^\top ((A^{(t-1)})^\top A^{(t-1)})^{-1} a^{(t)}\}$$

Proof. By Lemma 1.2.1, we have

$$\begin{aligned}
\det((A^{(t)})^\top A^{(t)}) &= \det((A^{(t-1)})^\top A^{(t-1)}) \cdot (1 + (a^{(t-1)})^\top ((A^{(t-1)})^\top A^{(t-1)})^{-1} a^{(t-1)}) \\
&= \det((A^{(t-1)})^\top A^{(t-1)}) \cdot (1 + \sigma^{A^{(t-1)}}(A^{(t)})_{d+t}) \\
&\ge \det((A^{(t-1)})^\top A^{(t-1)}) \cdot \exp(\frac{\sigma^{A^{(t-1)}}(A^{(t)})_{d+t}}{2}),
\end{aligned}$$

where the inequality holds since $\sigma^{A^{(t-1)}}(A^{(t)})_{d+t} \le 1$. Therefore

$$\begin{aligned}
\det((A^{(T)})^\top A^{(T)}) &\ge \det((A^{(0)})^\top A^{(0)}) \cdot \exp(\sum_{t=1}^{T} \frac{\sigma^{A^{(t-1)}}(A^{(t)})_{d+t}}{2}) \\
&\ge L^{2d} \cdot \exp(\sum_{t=1}^{T} \frac{\sigma^{A^{(t-1)}}(A^{(t)})_{d+t}}{2}).
\end{aligned}$$

Moreover by the AM-GM inequality (on the eigenvalues),

$$\det((A^{(T)})^\top A^{(T)})^{1/d} \le \frac{1}{d}\mathrm{tr}((A^{(T)})^\top A^{(T)}) = \frac{1}{d}\left\|A^{(T)}\right\|_F^2 \le \frac{T+d}{d}\cdot\frac{D^2}{4}.$$

Combining the above and taking logarithm, we have

$$2d\log(L) + \sum_{t=1}^{T}\frac{\sigma^{A^{(t-1)}}(A^{(t)})_{d+t}}{2} \le d\log(\frac{T+d}{d}\cdot\frac{D^2}{4}).$$

Therefore

$$\sum_{t=1}^{T}\sigma^{A^{(t-1)}}(A^{(t)})_{d+t} = O(d\log(TD/L)).$$

□

Proof of Theorem 10.0.1. The initial solution for $A^{(0)}$ and $b^{(0)}$ is trivially correct and the running time is $O(d^\omega)$ as presented in Algorithm 2. Since for $t' > t$, $(A^{(t')})^\top A^{(t')} \succeq (A^{(t)})^\top A^{(t)}$, $((A^{(t')})^\top A^{(t')})^{-1} \preceq ((A^{(t)})^\top A^{(t)})^{-1}$. Therefore, the leverage score estimations are always larger than the actual leverage scores for the subsequent matrices. Thus, the sampling from these estimations gives the guarantees of leverage score sampling for all matrices. Therefore, by Theorem 9.2.3, $S^{(t)}A^{(t)}$ is a subspace embedding of $A^{(t)}$. Let

$$\widetilde{x} = \arg\min\left\|S^{(t)}A^{(t)}x - S^{(t)}b^{(t)}\right\|_2, \text{ and } \quad x^* = \arg\min\left\|A^{(t)}x - b^{(t)}\right\|_2$$

Then since $S^{(t)}A^{(t)}$ is a subspace embedding of $A^{(t)}$, we have

$$\left\|S^{(t)}A^{(t)}\widetilde{x} - S^{(t)}b^{(t)}\right\|_2 \le \left\|S^{(t)}A^{(t)}x^* - S^{(t)}b^{(t)}\right\|_2 \le (1+\epsilon)\left\|A^{(t)}x^* - b^{(t)}\right\|_2$$

To bound the expected running time of the algorithm, we note that by Lemma 10.1.2 and definition of the probabilities of sampling in Algorithm 2, in expectation, we only sample $\widetilde{O}(\frac{d}{\epsilon^2}\log(TD/L))$ rows in expectation. By Lemma 10.1.1, any time a row is sampled, we pay a cost of $\widetilde{O}(d^2)$, and we pay a cost of $\widetilde{O}(\mathrm{nnz}(a))$ irrespective of the outcome of the sampling. Therefore, the total cost over the whole course of the algorithm is $\widetilde{O}(\mathrm{nnz}(A^{(T)}) + \frac{d^3}{\epsilon^2}\log(TD/L))$ in expectation. □

10.2 Exercises

10.2.1 Finding a large entry

For a vector $v \in \mathbb{R}^n$ we define $v_{\text{tail}} \in \mathbb{R}^n$ as the vector v with the largest entry (in absolute value) set to 0. So $\|v_{\text{tail}}\|_2^2 = \|v\|_2^2 - \|v\|_\infty^2$.

We now want to construct a procedure that allows us to find this large entry.

Problem: Prove the following claim.

We can construct random matrix $\mathbf{R} \in \mathbb{R}^{k \times n}$ for $k = \widetilde{O}(1/\epsilon^2)$ such that the following holds:
We are given $y = \mathbf{R}v \in \mathbb{R}^k$ (but we are not given the vector v) for some $v \in \mathbb{R}^n$. Then w.h.p. we can find the index i where $|v_i| > (1+\epsilon)\|v_{\text{tail}}\|_2$. We may return an incorrect index if no such i exists.

Hint: Use JL-matrices to estimate the following norm of vectors. Let w be a sub-vector of v (that is, we obtain w by setting some entries of v to 0). How large is $\|w\|_2^2$ if w contains the large entry (that we must return) vs. how large is the norm if it does not contain the large entry? You can use this to test if w contains the large entry. How many different w do you need to reconstruct the position of the large entry?

10.2.2 Counter example for regression

Consider the dynamic least squares problem, that is, we start on a matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$ and vector b and then repeatedly receive new rows to be inserted into $\mathbf{A}$ and b. After each new row, we must return an x with $\|\mathbf{A}x - b\| \leq (1+\epsilon) \min_{x^*} \|\mathbf{A}x^* - b\|_2$.

In the lecture we discussed a data structure that takes $\widetilde{O}(\text{nnz}(\mathbf{A})/\epsilon^2 + (d/\epsilon)^\omega)$ total time over all insertions. This data structure used the assumption that the new row $a \in \mathbb{R}^d$ and entry $\beta \in \mathbb{R}$ inserted to $\mathbf{A}$ and b have bounded norm, i.e. there is some $D > 0$ such that all inserted rows satisfy $\|(a, \beta)\|_2 \leq D^2$. Under this assumption, we showed in the lecture that the data structure must change its approximate solution x at most $\widetilde{O}(d)$ times over all row insertions.

In this exercise, we want to show that guaranteeing only $\widetilde{O}(d)$ changes to x is not possible without an assumption like $\|(a, \beta)\|_2 \leq D$. We want to show that without this assumption, the solution x may have to change after every new insertion. This is true for every possible data structure, not just the one from the lecture.

Problem: Show there is an initial $\mathbf{A}^{(0)} \in \mathbb{R}^{d \times d}, b^{(0)} \in \mathbb{R}^d$ and an (infinite) sequence of insertions $a^{(t)} \in \mathbb{R}^n, \beta^{(t)} \in \mathbb{R}$, $t \geq 1$ with the following property: Any data structure that maintains $x^{(t)}$ with $\|\mathbf{A}^{(t)}x^{(t)} - b^{(t)}\|_2 \leq (1+\epsilon) \min_{x^*} \|\mathbf{A}^{(t)}x^* - b^{(t)}\|_2$ must return a new $x^{(t)}$ after the new insertion. That means, if $x^{(t-1)}$ is the value returned before the tth insertion, then $\|\mathbf{A}^{(t)}x^{(t-1)} - b^{(t)}\|_2 > (1+\epsilon) \min_{x^*} \|A^{(t)}x^* - b^{(t)}\|_2$ for some constant $\epsilon > 0$.

10.2.3 Faster linear program solver via leverage scores

On problem set 3, we showed that we can solve linear programs in $\widetilde{O}(\sqrt{\|\tau\|_1})$ iterations for any $\tau \in \mathbb{R}^n$ with $\tau_i \geq 1$ for all i. Because of the lower bound on τ, we have $\|\tau\|_1 \geq n$ so with this approach we can not obtain less than $\widetilde{O}(\sqrt{n})$ iterations.

To get a faster algorithm, one would want to pick vector τ that can have $\tau_i > 0$ without being lower bounded by 1. The lower bound on τ was required to guarantee $\|\mathbf{S}^{-1}\delta_s\|_\infty \leq \|\mathbf{S}^{-1}\delta_s\|_\tau \leq 1$ so that $s + \delta_s > 0$ stays positive.

In this exercise, we will show that instead of $\tau \geq 1$, the condition $\tau \geq \sigma(\mathbf{X}^{1/2}\mathbf{S}^{-1/2}\mathbf{A})$ (where σ are the leverage scores) does already suffice. Since the sum of the leverage scores is d, Lee and Sidford showed in 2014 that this leads to an $\widetilde{O}(\sqrt{d})$ iteration algorithm.

Problem: Let $\mathbf{A} \in \mathbb{R}^{n\times d}$, $x, s \in \mathbb{R}^n_{>0}$, $\|\frac{xs-t\tau}{t\tau}\|_\infty < 0.1$, $\|\frac{xs-t\tau}{t\tau}\|_\tau < 0.1$ for $\tau \in \mathbb{R}^n$ and $\tau_i \geq \sigma(\mathbf{X}^{1/2}\mathbf{S}^{-1/2}\mathbf{A})_i$ for all i. (Here the norm $\|v\|_\tau$ was defined as $\sqrt{\sum_i \tau_i \cdot v_i^2}$.) For $\gamma \in \mathbb{R}_{\geq 0}$ let

$$\delta_s = \mathbf{A}(\mathbf{A}^\top\mathbf{X}\mathbf{S}^{-1}\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{S}^{-1}(t\tau - xs)\gamma.$$

Prove that $\|\mathbf{S}^{-1}\delta_s\|_\infty \leq O(\gamma)$.

Hint:

$$\begin{aligned}|(\mathbf{S}^{-1}\delta_s)_i| &= |e_i^\top\mathbf{S}^{-1}\mathbf{A}(\mathbf{A}^\top\mathbf{X}\mathbf{S}^{-1}\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{S}^{-1}(t\tau - xs)\gamma| \\ &\leq \|(\mathbf{A}^\top\mathbf{X}\mathbf{S}^{-1}\mathbf{A})^{-1/2}\mathbf{A}^\top\mathbf{S}^{-1}e_i\|_2 \cdot \|(\mathbf{A}^\top\mathbf{X}\mathbf{S}^{-1}\mathbf{A})^{-1/2}\mathbf{A}^\top\mathbf{S}^{-1}(t\tau - xs)\gamma\|_2\end{aligned}$$

by Cauchy-Schwarz inequality.

10.2.4 Sketching for solving linear systems

In the lecture we used subspace embeddings $\mathbf{S} \in \mathbb{R}^{k\times n}$ for $\mathbf{A} \in \mathbb{R}^{n\times d}$ with $k = \widetilde{O}(d/\epsilon^2)$ to obtain a spectral approximation $\mathbf{A}^\top\mathbf{S}^\top\mathbf{S}\mathbf{A} \approx_\epsilon \mathbf{A}^\top\mathbf{A}$ (so for all $v \in \mathbb{R}^d$ we have $v^\top\mathbf{A}^\top\mathbf{S}^\top\mathbf{S}\mathbf{A}v = (1\pm\epsilon)v^\top\mathbf{A}^\top\mathbf{A}v$).

This allows to solve the linear system $(\mathbf{A}^\top\mathbf{A})x = b$ approximately by letting $x = (\mathbf{A}^\top\mathbf{S}^\top\mathbf{S}\mathbf{A})^{-1}b$. One can show that this solution satisfies

$$\|x - x^*\|_{\mathbf{A}^\top\mathbf{A}} \leq \epsilon\|x^*\|_{\mathbf{A}^\top\mathbf{A}}$$

where $x^* = (\mathbf{A}^\top\mathbf{A})^{-1}b$ is the exact solution and the norm is defined as $\|v\|_{\mathbf{A}^\top\mathbf{A}} := \sqrt{v^\top(\mathbf{A}^\top\mathbf{A})v}$ for any $v \in \mathbb{R}^d$.

The complexity of this has a poly$(1/\epsilon)$ dependence to compute $\mathbf{A}^\top\mathbf{S}^\top\mathbf{S}\mathbf{A}$. We now want to argue that one can solve the linear system approximately with just a $\log(1/\epsilon)$ complexity dependence.

Problem: Given a matrix $\mathbf{M}$ such that $\mathbf{M} \approx_{0.1} \mathbf{A}^\top\mathbf{A}$, let $x^{(0)} = \mathbf{M}^{-1}b$. Then define recursively

$$x^{(k+1)} = x^{(k)} - \mathbf{M}^{-1}(\mathbf{A}^\top\mathbf{A}x^{(k)} - b).$$

Show that
for $k = O(\log 1/\epsilon)$ we have $\|x^{(k)} - x^*\|_{\mathbf{A}^\top\mathbf{A}} \leq \epsilon\|x^*\|_{\mathbf{A}^\top\mathbf{A}}$.

Hint: Show there is a constant $0 < c < 1$ with $\left\|x^{(k)} - x^*\right\|_{\mathbf{M}} \leq c\left\|x^{(k-1)} - x^*\right\|_{\mathbf{M}}$.

Chapter 11

Approximate Distances

In previous chapters we discussed various approximation techniques for linear algebra. In this chapter, we want to look at approximate techniques for distances in graphs. We will prove the following data structure for maintaining single source distances. The following is a slightly slower (but simpler) version of a data structure by v.d.Brand and Nanongkai (2019) [**?**].

Theorem 11.0.1. *For any $0 \leq \mu, s \leq 1$, there exists a data structure with the following operations:*

- INITIALIZE*($G = (V, E), \epsilon > 0$) in $O(n^{\omega+s})$ time.*
- INSERT/DELETE(u, v) *Insert/delete edge (u, v) in time $O(n^s(n^{1+\mu}+n^{\omega(1,1,\mu)-\mu})+n^{\omega(1-s,s,1)}/\epsilon)$.*
- QUERY$(v \in V)$ *Return a $(1+\epsilon)$-approximation of the single-source distances rooted at v in $O(n^{1+\mu+s}+n^{2-s})$ time.*

This is $O(n^{1.765}/\epsilon)$ update time and $O(n^{1.765})$ query time for $\mu \approx 0.529$ and $s \approx 0.236$.[1]

11.1 Approximate Distances via Linear Algebra

Earlier this semester we discussed that we can compute distances in graphs by computing the inverse of a polynomial matrix. In particular, given graph G and its $n \times n$ adjacency matrix $\mathbf{A}$, the inverse $(\mathbf{I} - X \cdot \mathbf{A})^{-1} \in (\mathbb{Z}[X]/\langle X^h \rangle)^{n \times n}$ encodes the h-bounded distance (i.e. distance up to h). Using the following notation (Definition 11.1.1), we had that the smallest $0 \leq k \leq d$ with $((\mathbf{I} - X \cdot \mathbf{A})^{-1}_{s,t})^{[k]} \neq 0$ is exactly the st-distance in G.

Definition 11.1.1. For a vector $v \in (\mathbb{Z}[X]/\langle X^d \rangle)^n$ (or matrix $\mathbf{M} \in (\mathbb{Z}[X]/\langle X^d \rangle)^{n \times n}$) we write $v^{[k]} \in \mathbb{Z}^n$ for the vector consisting of he coefficients of X^k, i.e. $v = \sum_{k=0}^{d-1} v^{[k]} X^k$. Likewise, $\mathbf{M} = \sum_{k=0}^{d-1} \mathbf{M}^{[k]} X^k$ where $\mathbf{M}^{[k]} \in \mathbb{Z}^{n \times n}$ are the coefficients of X^k.

[1] https://www.ocf.berkeley.edu/~vdbrand/complexity/?terms=omega(1%2C1%2Cmu)-mu%2Bs%0A1%2Bmu%2Bs%0Aomega(1-s%2Cs%2C1)

We can extend this to approximate distances as follows. Take graph G and add self-loops to every vertex. Then if there exists an st-path of length k, then there also exists a path of length k' for all $k' \geq k$. This is because we can simply use the self-loop to repeatedly go from t to t.

Thus we can find a $(1+\epsilon)$-approximation of the st-distance by searching for the smallest k of the form $k = \lfloor (1+\epsilon)^i \rfloor$ for some $i \in \mathbb{Z}$, where $((\mathbf{I} - X \cdot \mathbf{A})^{-1}_{s,t})^{[k]} \neq 0$.

In particular, this means to compute $(1+\epsilon)$-approximate distances, we only need to compute $((\mathbf{I} - X \cdot \mathbf{A})^{-1})^{[k]}$ for $O(\log_{(1+\epsilon)}(n)) = O(\epsilon^{-1} \log n)$ many different k. For comparison, computing the exact distance requires to use $k = 1, ..., n$, i.e. n different values for k.

This motivates the following algebraic data structure, which corresponds to computing $(1+\epsilon)$-approximate distances (up to h) for a set of pairs $S \times T$.

Theorem 11.1.2. *For any $0 \leq \mu \leq 1$, $\epsilon > 0$, there exists a data structure with the following operations:*

- INITIALIZE*($\mathbf{A} \in \mathbb{Z}^{n \times n}$, $S, T \subset \{1, ..., n\}$, $h \in \{1, ..., n\}$) in $O(h \cdot n^\omega)$ time.*
- UPDATE$(i, j \in \{1, ..., n\}, f \in \mathbb{Z})$ *Set $\mathbf{A}_{i,j} \leftarrow f$. Then return the submatrix $((\mathbf{I} - X \cdot \mathbf{A})^{-1}_{S,T})^{[k]}$ for $k = \lfloor (1+\epsilon)^0 \rfloor, \lfloor (1+\epsilon)^1 \rfloor, ..., n^s$. The complexity is $O(n^s(n^{1+\mu} + n^{\omega(1,1,\mu)-mu}) + n^{\omega(x,s,y)}/\epsilon)$ where $n^x = |S|, n^y = |T|, n^s = h$.*
- QUERY$(i, j \in \{1, ..., n\})$ *Return $(\mathbf{I} - X\mathbf{A})^{-1}_{i,j}$ (i.e. all coefficients for all X^k, $k = 1, ..., h$) in $O(n^{\mu+s})$ time for $n^s = h$.*

Proof. In a previous lecture we explained that we can maintain the inverse of some matrix $\mathbf{M}$ in $O(h(n^{1+\mu} + n^{\omega(1,1,\mu)-\mu}))$ time per update. Here an update changes one entry of $\mathbf{M}$ and we can query any entry $\mathbf{M}^{-1}_{i,j}$ in $O(n^\mu)$ time.

We use this data structure to maintain $(\mathbf{I} - X\mathbf{A})^{-1} \in (\mathbb{Z}/\langle X^h \rangle)^{n \times n}$ supporting entry updates to $\mathbf{A}$ in $\widetilde{O}(h(n^{1+\mu} + n^{\omega(1,1,\mu)-\mu}))$ time per update. The extra $\widetilde{O}(h)$ factor comes from the fact that we now consider polynomials of degree up to h.

This data structure maintains the inverse of the form

$$\mathbf{M}^{-1} = \mathbf{M}'^{-1} = \sum_{i=1}^{\ell} u_i v_i^\top$$

where $\mathbf{M}'$ is the matrix $\mathbf{M}$ during initialization, ℓ is the number of updates so far, and after each update we add one new outer product to the sum above.

We are left with describing how to compute $(\mathbf{M}^{-1}_{S,T})^{[k]}$ for the $O(\epsilon^{-1} s \log n)$ many different k. Assume we already know these values from the previous update, then all we need to compute is (during the ℓ-th update)

$$(\mathbf{M}^{-1}_{S,T})^{[k]} \leftarrow (\mathbf{M}^{-1}_{S,T})^{[k]} + (u_\ell v_\ell^\top)^{[k]}$$

This can be computed in $O(n^{\omega(x,d,y)})$ time where $n^x = |S|, n^y = |T|, n^s = h$ via Lemma 11.1.3. As we repeat that for $O(\epsilon^{-1} \log n^s) = O(s\epsilon^{-1} \log n)$ different k, we get an extra $\widetilde{O}(n^{\omega(x,d,y)}/\epsilon)$ per update. □

Lemma 11.1.3. *For $u \in (\mathbb{Z}[X]/\langle X^{n^d}\rangle)^{n^x}$, $v \in (\mathbb{Z}[X]/\langle X^{n^d}\rangle)^{n^y}$ and $z := uv^\top$, we can compute $z^{[k]}$ for any one k in time $O(n^{\omega(x,d,y)})$.*

Proof.

$$(uv^\top)^{[k]} = \sum_{i=0}^{k} u^{[i]}(v^\top)^{[k-i]} = \left[u^{[0]}|u^{[1]}|...|u^{[k]}\right] \begin{bmatrix} (v^{[k]})^\top \\ (v^{[k-1]})^\top \\ \vdots \\ (v^{[0]})^\top \end{bmatrix}$$

Where the last expression is a matrix product of an $n^x \times n^d$ and $n^d \times n^y$ matrix, so we can compute it in $O(n^{\omega(x,d,y)})$ time. □

Theorem 11.1.2 can be used to obtain approximate distances up to h. The following Lemma 11.1.4 is a variation of Lemma 3.3.2 and theorem 3.3.1 to extend these h-bounded distances to general distances without an upper bound.

Lemma 11.1.4. *Let $1 \leq h \leq n$, $G = (V, E)$, and $R \subset V$ be a uniformly at random samples subset of size $\widetilde{O}(n/h)$.*

If we are given $(1+\epsilon)$-approximate h-bounded distances for the pairs $(\{s\} \cup R) \times V$ for some $s \in V$, then we can compute $(1+\epsilon)$-approximate single source distances for source s in time $\widetilde{O}(n^2/h)$.

Proof. Similar to Lemma 3.3.2 and theorem 3.3.1. Construct a graph $H = (V, E')$ with edges $E' = \{s \cup R\} \times V$ and the edge weight being the respective $(1+\epsilon)$-approximate h-bounded distance. Then run Dijkstra's algorithm from s in H and return for each pair $\{s\} \times V$ the minimum of the distance in H and the $(1+\epsilon)$-approximate h-bounded distance that we were given.

This is a good distance estimate because any path in H corresponds to a path in G of the same distance (up to $(1+\epsilon)$-factor) because each edge in H corresponds to a path in G. Conversely, any path that uses at most h steps is split into segments of length at most h by Lemma 3.3.2. So each of these segments has a corresponding edge in G, so we can find the path in H.

Only sv-paths in G that are too short (less than length h) might not exist in H. But for such pairs s, v we already know their (approximate) distance from the provided h-bounded distances.

The complexity is $\widetilde{O}(n^2/h)$ as that is the number of edges in H and Dijkstra's algorithm runs in $\widetilde{O}(|E'|)$ time. □

Lemma 11.1.5. *Let $1 \leq h \leq n$, $G = (V, E)$, and $R \subset V$ be a uniformly at random samples subset of size $\widetilde{O}(n/h)$.*

If we are given $(1+\epsilon)$-approximate h-bounded distances for the pairs $(\{s\} \cup R) \times (\{t\} \cup R)$ for some $s, t \in V$, then we can compute $(1+\epsilon)$-approximate st-distance in time $\widetilde{O}(n^2/h^2)$.

Proof. Same as Lemma 11.1.4 but instead of edges from R to V we just have edges from R to $R \cup \{t\}$. □

Together with Theorem 11.1.2 we now directly obtain Theorems 11.0.1 and 11.1.6.

Theorem 11.1.6. *For any $0 \le \mu, s \le 1$ there exists a data structure with the following operations:*

- INITIALIZE$(G = (V, E), \epsilon > 0)$ *in* $O(n^{\omega+s})$ *time.*
- INSERT/DELETE(u, v) *Insert/delete edge* (u, v) *in time* $O(n^s(n^{1+\mu}+n^{\omega(1,1,\mu)-\mu})+n^{\omega(1-s,s,1-s)}/\epsilon)$.
- QUERY(s, t) *Return a* $(1+\epsilon)$*-approximation of the* st*-distance in* $O(n^{1+\mu} + n^{2-2s})$ *time.*

This is $\widetilde{O}(n^{1.686}/\epsilon)$ update time and $\widetilde{O}(n^{1.685})$ query time for $\mu \approx 0.529$ and $s \approx 0.157$.[2]

Proof. We prove both Theorems 11.0.1 and 11.1.6

Theorem 11.0.1: We maintain the $(1+\epsilon)$-approx distances by adding self-loops to every vertex in G. Then maintain $(\mathbf{I} - X \cdot \mathbf{A})^{-1}$ via Theorem 11.1.2 for $S = R$ (the random set from Lemma 11.1.4) and $T = V$. This way get $((\mathbf{I} - X \cdot \mathbf{A})^{-1}_{R,V})^{[k]}$ for the different powers of $k = \lfloor (1+\epsilon)^i \rfloor \le h$. For any $u \in R, v \in V$, the smallest such k where $((\mathbf{I} - X \cdot \mathbf{A})^{-1}_{u,v})^{[k]} \neq 0$ is at most a $(1+\epsilon)$-factor larger than the uv-distance. So we obtain $(1+\epsilon)$-approximate distance estimates for the pairs $R \times V$ up to distance h.

This takes $\widetilde{O}(n^{1+\mu+s} + n^{\omega(1,1,\mu)-\mu+1} + n^{\omega(1-s,s,1)}/\epsilon)$ time per update by Theorem 11.1.2, as $h = n^s$, $|S| = \widetilde{O}(n/h) = \widetilde{O}(n^{1-s})$ and $|T| = n$.

Query Via Lemma 11.1.4 we can now compute the single source distances from any s by computing the h-bounded distances from s to all $v \in V$ in $\widetilde{O}(hn^{1+\mu}) = \widetilde{O}(n^{1+s+\mu})$ time, and then using Lemma 11.1.4 in $\widetilde{O}(n^{2-s})$ time.

Theorem 11.1.6: The data structure is the same, but by Lemma 11.1.5 we only need the distances for $(\{s\} \cup R) \times (\{t\} \cup R)$. The distances between $R \times R$ can be computed in $\widetilde{O}(n^{1+\mu+s} + n^{\omega(1,1,\mu)-\mu} + n^{\omega(1-s,s,1-s)}/\epsilon)$ time per update by $S = T = R$ and $|R| == \widetilde{O}(n/h) = \widetilde{O}(n^{1-s})$ for $h = n^s$.

To apply Lemma 11.1.5 to answer a query we then only need to compute the distances from s to each vertex in R, and from R to t in $\widetilde{O}(|R|hn^{\mu}) = \widetilde{O}(n^{1+\mu})$ time. Lemma 11.1.5 needs an additional $\widetilde{O}(n^2/h^2) = \widetilde{O}(n^{2-2s})$ time. □

11.2 Exercise

The diameter of a graph G is defined as the largest distance, i.e.

$$\text{diam}(G) = \max_{s,t \in V} \text{dist}(s, t)$$

[2] https://www.ocf.berkeley.edu/~vdbrand/complexity/?terms=omega(1%2C1%2Cmu)-mu%2Bs%0A1%2Bmu%2Bs%0Aomega(1-s%2Cs%2C1-s)

Problem: Construct a data structure with the following operations and complexity.

Theorem 11.2.1. *For any* $0 \le \mu \le 1, 0 \le s \le 1$ *,* $\epsilon > 0$ *there exists a data structure with the following operations:*

- INITIALIZE*(*$G = (V, E), \epsilon > 0$*) in* $\widetilde{O}(n^{\omega+s})$ *time.*
- INSERT/DELETE(u, v)*: Inserts/deletes the edge* (u, v) *in time* $\widetilde{O}(n^s(n^{1+\mu} + n^{\omega(1,1,\mu)-\mu}) + \frac{n^{\omega(1,s,1)}}{\epsilon} + n^{3(1-s)})$*. Then returns a* $(1 + \epsilon)$*-approximation of the diameter.*

The graph is unweighted and directed.

Chapter 12

Handling Adaptive Adversaries via Random Noise

We discussed several randomized algorithms and data structures. These randomized techniques work if we assume the input to be independent of the random choices. For algorithms this independence is usually given, but for data structures we might encounter issues.

For example, the user might perform updates that change the input. So if the previous outputs of the data structure depend on the randomness, then now the input depends on the randomness as well.

Here we want to discuss a technique to solve this issue, i.e. make sure the output of the data structure can not depend on the random choices performed by the data structure.

Random noise We will hide any information about the random choices of the data structure by adding random noise to the output. For example, an approximate data structure might leak information about the internal random choices via the approximation error (e.g. whether the returned approximation is slightly larger or smaller than the correct exact result). On an intuitive level, adding noise will hide any information about the internal random choices because the user does not know if the approximation error comes from the random noise or from the internal randomness of the data structure.

We will use Laplace noise, i.e. add a random Laplace distributed random variable to the output of the data structure.

Definition 12.0.1. We say $X \in \mathbb{R}$ is Laplace distributed (notation $X \sim L(\mu, b)$) if the density function of the distribution is

$$f(x|\mu, b) = \frac{1}{2b} \exp(-\frac{|x-\mu|}{b})$$

so $\mathbb{P}_{X\in[\ell,r]} [=] \int_\ell^r f(x|\mu,b)\ dx$.

Note that μ is the expectation an b is a parameter that controls the variance (the variance is $2b^2$). One can generate a sample a random $X \sim L(\mu, b)$ as follows:

Sample two independent and uniform $y_1, y_2 \in [0, 1]$, then set $x = \mu + b \cdot \ln(y_1/y_2)$.

Lemma 12.0.2. *For $X \sim L(\mu, b)$ we have w.h.p. $|X - \mu| < O(b \log n)$.*

Lemma 12.0.3.

$$\exp(-\frac{|mu - \mu'|}{b}) \leq \frac{f(x|\mu, b)}{f(x|\mu', b)} \leq \exp(\frac{|\mu - \mu'|}{b})$$

So if the noise parameter $b = \Omega(|\mu - \mu'|)$ then the fraction $f(x|\mu, b)/f(x|\mu', b)$ is very close to 1, i.e. the two distribution $L(\mu, b)$ and $L(\mu', b)$ are almost the same.

Further, the parameter b allows for a trade-off: Large b means we add a lot of noise (i.e. the results of the data structure become less accurate) but the adversary can learn less about the data structure's internal randomness.

Theorem 12.0.4. *Assume we can compute μ in time T and μ' in time T'. Assume further we have some bound $D \geq |\mu - \mu'|$ and α, β such that $D/b \leq \alpha \leq 1$.*

Then we can sample $X \sim L(\mu, b)$ in expected time $O(T' + \alpha T)$.

Note that the naive way to sample from $L(\mu, b)$ would be to compute μ in time T and then add some Laplace noise to it. Theorem 12.1.2 says that (for small α) we can sample from $L(\mu, b)$ without computing μ (most of the time).

Proof. We define the following function with name SIMULATE(μ, μ', α), because it simulates sampling from $L(\mu, b)$ without needing to compute μ (most of the time). The steps are as follows:

1. With probability $p = \exp(-\alpha)$
   - sample and return $y \sim L(\mu', b)$.
2. else
   - Sample and return $z \in \mathbb{R}$ where the density function of the distribution is

$$g(x) := \frac{f(x|\mu, b) - \exp(-\alpha) \cdot f(x|\mu', b)}{1 - \exp(-\alpha)}$$

Correctness First, let us verify that the distribution of z is well defined, i.e. that g is a valid density function. We have $g(x) \geq 0$ for all x because by Lemma 12.0.3

$$f(x|\mu, b)/f(x|\mu', b) \geq \exp(-|\mu - \mu'|/b) \geq \exp(-\alpha)$$

Further we have

$$\int_{-\infty}^{\infty} \frac{f(x|\mu, b) - \exp(-\alpha) \cdot f(x|\mu', b)}{1 - \exp(-\alpha)} \, dx = \frac{\int_{-\infty}^{\infty} f(x|\mu, b) \, dx - \exp(-\alpha) \int_{-\infty}^{\infty} f(x|\mu', b) \, dx}{1 - \exp(-\alpha)}$$
$$= \frac{1 - \exp(-\alpha)}{1 - \exp(-\alpha)} = 1.$$

Thus g is a valid density function for a random distribution.

Next, let us verify that the output of $\text{SIMULATE}(\mu, \mu', \alpha)$ indeed has the distribution of $L(\mu, b)$. The density function of the output is

$$
\begin{aligned}
&\underbrace{\exp(-\alpha) \cdot f(x|\mu', b)}_{\text{Branch 1}} \;+\; \underbrace{(1-\exp(-\alpha)) \cdot g(x)}_{\text{Branch 2}} \\
=\;& \exp(-\alpha) \cdot f(x|\mu', b) + (1-\exp(-\alpha)) \frac{f(x|\mu, b) - \exp(-\alpha) \cdot f(x|\mu', b)}{1-\exp(-\alpha)} \\
=\;& f(x|\mu, b)
\end{aligned}
$$

So the output has indeed distribution $L(\mu, b)$.

How to sample z? We verified that the output is correct, assuming z has the right distribution (i.e. density function g), but how can we actually sample something with this density function?

We will sample z via the following procedure

- while true:
  1. Sample $z \sim L(\mu, b)$.
  2. Return z with probability $p = 1 - \exp(-\alpha)\frac{f(x|\mu', b)}{f(x|\mu, b)}$, otherwise discard z and go to the next loop.

The density function will be proportional to

$$
\underbrace{f(x|\mu, b)}_{\text{Step 1}} \cdot \underbrace{(1-\exp(-\alpha)\frac{f(x|\mu', b)}{f(x|\mu, b)}}_{\text{Step 2}} = f(x|\mu, b) - \exp(-\alpha) f(x|\mu', b) \propto g(x)
$$

So it has the correct distribution.

Complexity To bound the complexity of SIMULATE, we first bound the complexity of sampling z. Note that the procedure to z has an infinite loop. In each iteration we break the loop with some probability, let's say q, so the expected number of iterations would be $1/q$ (since the number of loops will be geometric distributed). Let us compute this q. We have

$$
\begin{aligned}
q &= \int_{-\infty}^{\infty} \underbrace{f(x|\mu, b)}_{\text{Step 1}} \cdot \underbrace{(1-\exp(-\alpha)\frac{f(x|\mu', b)}{f(x|\mu, b)}}_{\text{Step 2}} \; dx \\
&= \int_{-\infty}^{\infty} f(x|\mu, b)\; dx - \exp(-\alpha) \int_{-\infty}^{\infty} f(x|\mu', b)\; dx \\
&= 1 - \exp(-\alpha).
\end{aligned}
$$

So the expected number of iterations will be $1/(1-\exp(-\alpha))$. However, this assumes that we must sample z in the first place. The probability of needing to sample z (i.e. going into

branch 2 of SIMULATE) is $1 - \exp(-\alpha)$. So the expected number of iterations to sample z, assuming we call SIMULATE is just

$$\frac{1 - \exp(-\alpha)}{1 - \exp(-\alpha)} = 1.$$

So the time complexity of SIMULATE is just $O(T' + (1 - \exp(-\alpha)T)$ because no matter if we are in branch 1 or 2, we must always compute μ' in time T'. But only in branch 2 do we need to compute μ in time T. Note that we can bound $1 - \exp(-\alpha) = O(\alpha)$ because for $0 \leq \alpha \leq 1$ we have $\exp(\alpha) \leq 1 + 3\alpha$. So the expected time complexity of SIMULATE is $O(T' + \alpha T)$. □

Example In a previous lecture we construct a data structure that could maintain the st-distance up to multiplicative $1+\epsilon$ error in $O(n^{1.686}/\epsilon)$ time per update (see Theorem 11.1.6).

Theorem 12.0.5 (Restatement of Theorem 11.1.6). *There exists a data structure with the following operations:*

- INITIALIZE*($G = (V, E), \epsilon > 0$)*
- INSERT/DELETE(u, v) *Insert/delete edge (u, v) in time $O(n^{1.686}/\epsilon)$.*
- QUERY(s, t) *Return a $(1 + \epsilon)$-approximation of the st-distance in $O(n^{1.686})$ time.*

This data structure requires the oblivious adversary assumption. We now extend it to an adaptive adversary via Theorem 12.1.2. We use the SIMULATE method to return after each update a random value $\text{dist}(s, t) \cdot 2^x$ where $x \sim L(0, \delta)$, i.e. the exact st-distance with some noise multiplied to it. Note that, since the adversary receives a random variable whose distribution depends only on $\text{dist}(s, t)$ and parameter δ, they have no idea what the approximate value was that our data structure returned. So they have no way to perform updates that in some way depend on the internal randomness of our data structure. So now the data structure works against the adaptive adversary.

Let d be the exact st-distance and $\widetilde{d}$ be the $(1 + \epsilon)$-approximate st-distance returned by the data structure. We define the following values:

- $\mu = \log(d)$, $\mu' = \log(\widetilde{d})$
- $T = O(n^2)$, $T' = O(n^{1.686}/\epsilon)$.
- $|\mu - \mu'| = |\log(d) - \log(\widetilde{d})| \leq \log(1 + \epsilon) \leq \epsilon =: D$.
- $\alpha = 1/n^x$ for some value x we specify later.
- $b = \delta/(c \log n)$ for some $\delta > 0$ which will be the accuracy of our output at the end, and some large constant c.
- $\epsilon = \delta/(cn^x \log n)$ the accuracy with which we run our data structure.

So now SIMULATE samples a value with distribution $L(\mu, b)$. Note that by exponentiating this value, we get w.h.p.

$$2^{\text{SIMULATE}(\mu,\mu',\alpha)} = 2^{d \pm O(b \log n)} = \text{dist}(s,t) \cdot 2^{\pm O(b \log n)} = \text{dist}(s,t) \cdot (1 \pm \delta)$$

where we used that a Laplace variable $L(\mu, b)$ is at most $O(b \log n)$ distance from μ with high probability. In summary, we get a $(1 \pm \delta)$-approximation of the st-distance. The expected time complexity is

$$O(T' + \alpha T) = O(n^{1.686}/\epsilon + \alpha n^2) = \widetilde{O}(n^{1.686+x}/\delta + n^{2-x})$$

which for the right choice of x is

$$\widetilde{O}(n^{(1.686+2)/2}/\delta) = \widetilde{O}(n^{(1.843+2)/2}/\delta)$$

So we can now maintain the st-distance against an adaptive adversary in expected $\widetilde{O}(n^{(1.843+2)/2}/\delta)$ time per update and query.

12.1 Exercises

12.1.1 Recursive Laplace Noise

In the lecture we discussed how we can sample from a Laplace distribution $L(\mu, b)$ such that it often suffices to compute only some $\mu' \approx \mu$ instead. We showed that we only need to compute μ with some probability $O(\alpha)$. This probability depends on the accuracy of $\mu' \approx \mu$. The more accurate μ', the smaller we can make α. Since computing a very accurate estimate μ' might be inefficient, we here want to prove a generalization that requires the accurate estimate less frequently.

The following theorem states that if we have several estimate μ_i of μ_0 of varying accuracy, then we need the more accurate estimates less frequently.

Problem: Prove the following Theorem.

Theorem 12.1.1. *Assume we can compute $\mu_i \in \mathbb{R}$ in time T_i for $i = 0, ..., k$. Assume further we have some bound D such that $D \cdot 2^i \geq |\mu_i - \mu_{i+1}|$ for all $i = 0, ..., k-1$, and α, b are such that $D/b \leq \alpha \leq 2/2^k$.*

Then we can sample $X \sim L(\mu_0, b)$ in expected time $O(T_k + \alpha \sum_{i=0}^{k-1} 2^i T_i)$.

Hint: Theorem 18.0.4 in the lecture notes is the special case for $k = 1$.
Use recursion.

12.1.2 High Dimensional Laplace Noise

In the lecture we discussed how to hide approximation errors of data structures by adding Laplace noise. We assumed that the output of the data structure μ' and the exactly correct value μ are some real numbers. We here want to extend the result to the case where we might compute a vector instead of a real number.

Problem: Prove the following Theorem.

Theorem 12.1.2. *Assume we can compute $\mu \in \mathbb{R}^n$ in time T and $\mu' \in \mathbb{R}^n$ in time T'. Assume further we have some bound $D \geq \|\mu - \mu'\|_1$ and α, b such that $D/b \leq \alpha \leq 1$.*

Then we can sample in expected time $O(T' + \alpha T)$ a random $X \in \mathbb{R}^n$ such that each X_i is independently $X_i \sim L(\mu_i, b)$ distributed for $i = 1, ..., n$.

Hint: Theorem 18.0.4 in the lecture notes is the special case for $n = 1$.

Bibliography

[ACK17] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *SODA*, pages 440–452. SIAM, 2017.

[AW21] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.

[BNS19] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *FOCS*, pages 456–480. IEEE Computer Society, 2019.

[CKL18] Diptarka Chakraborty, Lior Kamma, and Kasper Green Larsen. Tight cell probe bounds for succinct boolean matrix-vector multiplication. In *STOC*, pages 1297–1306. ACM, 2018.

[CLS19] Michael B. Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. In *STOC*, pages 938–942. ACM, 2019.

[CW13] Kenneth L Clarkson and David P Woodruff. Low rank approximation and regression in input sparsity time. In *STOC*. ACM, 2013.

[DL77] Richard A DeMillo and Richard J Lipton. A probabilistic remark on algebraic program testing. Technical report, GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1977.

[DWZ22] Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing. *arXiv preprint arXiv:2210.10173*, 2022.

[FMNZ01] Daniele Frigioni, Tobias Miller, Umberto Nanni, and Christos D. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM J. Exp. Algorithmics*, 6:9, 2001.

[Gal14] François Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303. ACM, 2014.

[GU18] François Le Gall and Florent Urrutia. Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor. In *Proceedings of*

the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 1029–1046. SIAM, 2018.

[HHS21] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms. *CoRR*, abs/2102.11169, 2021.

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30. ACM, 2015.

[JPW22] Shunhua Jiang, Binghui Peng, and Omri Weinstein. Dynamic least-squares regression. *CoRR*, abs/2201.00228, 2022.

[JSWZ21] Shunhua Jiang, Zhao Song, Omri Weinstein, and Hengjie Zhang. A faster algorithm for solving general lps. In *STOC*, pages 823–832. ACM, 2021.

[KS02] Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences*, 65(1):150–167, 2002.

[KZ08] Ioannis Krommidas and Christos D. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *ACM J. Exp. Algorithmics*, 12:1.6:1–1.6:22, 2008.

[LSZ19] Yin Tat Lee, Zhao Song, and Qiuyi Zhang. Solving empirical risk minimization in the current matrix multiplication time. In *COLT*, volume 99 of *Proceedings of Machine Learning Research*, pages 2140–2157. PMLR, 2019.

[Ore22] Øystein Ore. Über höhere kongruenzen. *Norsk Mat. Forenings Skrifter*, 1(7):15, 1922.

[Pan78] Victor Y. Pan. Strassen's algorithm is not optimal: Trililnear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *FOCS*, pages 166–176. IEEE Computer Society, 1978.

[Ren88] James Renegar. A polynomial-time algorithm, based on newton's method, for linear programming. *Math. Program.*, 40(1-3):59–93, 1988.

[S+69] Volker Strassen et al. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[San04] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *FOCS*, pages 509–517. IEEE Computer Society, 2004.

[San05] Piotr Sankowski. Subquadratic algorithm for dynamic shortest distances. In *COCOON*, volume 3595 of *Lecture Notes in Computer Science*, pages 461–470. Springer, 2005.

[San07] Piotr Sankowski. Faster dynamic matchings and vertex connectivity. In *SODA*, pages 118–126. SIAM, 2007.

[Sch80] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[SM50] Jack Sherman and Winifred J Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127, 1950.

[Vai87] Pravin M. Vaidya. An algorithm for linear programming which requires o(((m+n)nˆ2 + (m+n)ˆ1.5 n)l) arithmetic operations. In *STOC*, pages 29–38. ACM, 1987.

[vdBLN+20] Jan van den Brand, Yin-Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 919–930. IEEE, 2020.

[vdBLSS20] Jan van den Brand, Yin Tat Lee, Aaron Sidford, and Zhao Song. Solving tall dense linear programs in nearly linear time. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 775–788, 2020.

[Woo50] Max A Woodbury. *Inverting modified matrices*. Statistical Research Group, 1950.

[WW10] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS*, pages 645–654. IEEE Computer Society, 2010.

[YTM94] Yinyu Ye, Michael J. Todd, and Shinji Mizuno. An o($\sqrt{}$nl)-iteration homogeneous and self-dual linear programming algorithm. *Math. Oper. Res.*, 19(1):53–67, 1994.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.