GEORGIA TECH SPRING 2025 CS 3511: HONORS ALGORITHMS

Final Project

Authors: *Arjun Sheshadri, Vilohith Gokarakonda, Yiqi Sun, Tyler Huang*
Date: Wednesday 4/16/25

1 Introduction

Suppose if you have a graph $G = (V, E)$ with a given adjacency matrix, which represents where any two nodes are connected to each other. Suppose you wanted to find the shortest path to one node, using all nodes as a starting point.

Shortest distances between all pairs of nodes in a graph is important for its real world applications in communication networks, social graphs, and more. Edges in such graphs can be added and removed at any time, so being able to efficiently maintain all pairs distances is crucial. Using the Floyd-Warshall Algorithm, we are able to accomplish this in $\mathcal{O}(n^3)$ time. However, this becomes inefficient for larger or frequently changing graphs.

This report explores a *dynamic algorithm* described in Jan van den Brand's notes for maintaining All-Pairs-Shortest-Paths (`APSP`) in directed graphs with or without edge weights. Instead of recalculating shortest distances from scratch every update, it efficiently maintains a data structure where all distances can be updated in $\tilde{\mathcal{O}}(n^{2.5})$ time when an edge to a single vertex is added or removed. Specifically, it uses concepts from dynamic algebraic algorithms, involving polynomial matrix inverses to represent path information, and extending them to full distances with random sampling and Dijkstra's algorithm.

In this report, we present the problem statement, and the technical background relevant to the paper, including ring algebra, polynomial matrices, and how edges are updated. We also cover the steps to the solution, and connect this approach to the original groundbreaking paper by Sankowski, which can do updates in $\tilde{\mathcal{O}}(n^{1.932})$ randomized time and queries in $\tilde{\mathcal{O}}(n^{1.288})$ randomized time. Specifically, we compare their mathematical foundations, update operations, applications, and time complexities.

2 Problem Statement

Given a graph $G(V, E)$, we want to develop a data structure that can maintain `APSP` dynamically with an initial overhead $\tilde{\mathcal{O}}(n^{3.5})$ time and supports queries and updates in $\tilde{\mathcal{O}}(n^{2.5})$ time. Since this algorithm is optimized for maintaining `APSP` for changing graphs (hence requires a dynamic algorithm), having a time complexity of $\tilde{\mathcal{O}}(n^{2.5})$ is more efficient compared to the naive approach which takes $\mathcal{O}(n^3)$ time for queries and updates.

Specifically, we will be working towards proving the following theorem:

Theorem 1.0.0: *There exists a data structure that supports the following operations*:

1. Initialize$(G = (V, E))$ *Initialize an n-node graph and return* `APSP` *in $\tilde{\mathcal{O}}(n^{3.5})$ time.*
2. Update$(v,\ E^+,\ E^-)$ *Given a vertex v and two sets of edges $E^+ \subseteq (\{v\} \times V \cup V \times \{v\})$ to insert and E^- to delete (all incident to v), update G and return the new* `APSP` *in $\tilde{\mathcal{O}}(n^{2.5})$ time.*

Additionally, we will be extending this theorem using Sankowski's paper:

Theorem 1.0.1: *There exists a data structure that supports the following operations*:

1. Initialize$(G = (V, E))$ *Preprocess in $\tilde{\mathcal{O}}(n^3)$ time.*
2. Update(e) *Insert or delete a single edge $e \in V \times V$ in $\mathcal{O}(n^{1.932})$ randomized time.*
3. Query(s, t) *Return the current distance $\operatorname{dist}_G(s, t)$ in $\mathcal{O}(n^{1.288})$ randomized time.*

All operations succeed with high probability over the random choices.

We will be covering general overviews of topics that help toward understanding the intuition behind these theorems in the next section.

3 Technical Background

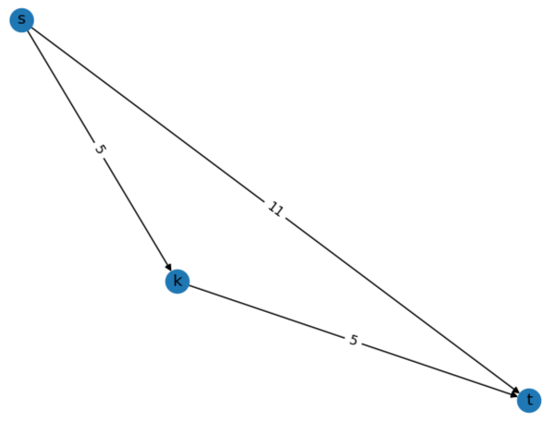
3.1 Matrix Multiplication Recap and Naive APSP

Matrix multiplication is an important concept that we will be making extensive use of to understand the technical components in this report. You are already familiar with the basic method of multiplying two matrices with the same inner dimension, where you multiply the n-th row of the first matrix with the n-th column of the second matrix. Below is a visual representation:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a \cdot e + b \cdot g & a \cdot f + b \cdot h \\ c \cdot e + d \cdot g & c \cdot f + d \cdot h \end{bmatrix}$$

Notice how in each (i, j) entry of the product matrix, we compute the *dot product* of the vector that forms the i-th row of the first matrix and the j-th column of the second matrix. The *dot product* is simply an operation on the entries of two vectors involving the typical $+$ and $\times$ operations.

Let's take a quick detour from matrix multiplication to talk about the well-known Floyd-Warshall algorithm for computing `APSP`. The key realization to understanding this algorithm is that between some start node s and some terminal node t, the shortest path from $s \to t$ is the shorter path we can get from directly traveling from $s \to t$ vs. the path we get by traveling from s to some intermediate node k and then from k to t. For a visualization, consider the trivial example below:



In the above graphic, the direct path from $s \to t$ has a cost of 11, while the path from $s \to k$ has a cost of 5 and the path from $k \to t$ has a cost of 5 as well. We can see that if we take $\min\{\text{dist}_{s,t} = 11, \text{dist}_{s,k} + \text{dist}_{k,t} = 10\}$, we get that by considering the intermediate node k, we arrive at a cheaper path from s to t.

How does the above logic tie into matrix multiplication? Well, it might be intuitive to think of graphs in terms of adjacency matrices and the core Floyd-Warshall computation as a dot product between some vector u that represents the shortest known paths from the starting node s to all possible intermediate nodes k, and another vector v that represents the shortest paths from all possible intermediate nodes k to the terminal node t. Let's see an example of this analogy on the simple graph presented earlier:

Consider the adjacency matrix A for the graph presented earlier:

$$A = \begin{array}{c} \\ s \\ k \\ t \end{array} \begin{array}{c} \begin{array}{ccc} s & k & t \end{array} \\ \begin{pmatrix} 0 & 5 & 11 \\ \infty & 0 & 5 \\ \infty & \infty & 0 \end{pmatrix} \end{array}$$

Notice that in A, the (i, j) entry represents the cost of the *direct* path from node i to node j. The rows and columns are labeled with their appropriate nodes on the borders.

For the sake of computation, let's take the row s and the column t of the matrix as our u and v vectors. That is, let

$$u = \begin{bmatrix} 0 & 5 & 11 \end{bmatrix} \qquad v = \begin{bmatrix} 11 \\ 5 \\ 0 \end{bmatrix}$$

Just for clarity, recall that u represents the shortest known path between the starting node s and every other node k in the graph. For column s of u, the shortest path cost is 0 since we are already at s. For column k of u, the path cost is 5 since the edge $s \to k$ on the graph has a cost of 5. For column t of u, the path cost is 11 since the edge $s \to t$ has a cost of 11. In general, we see that each *column* in u represents the cost of the edge $s \to$ *column in u*. Following a similar logic, notice that in the vector v, each *row* represents the cost of the edge *row in v* $\to t$. Both *column in u* and *row in v* represent all the nodes in the graph.

Let's replace the numbers in A with symbols, and see what happens when we take the dot product of u and v:

$$A = \begin{array}{c|ccc} & s & k & t \\ \hline s & \text{dist}_{s\to s} & \text{dist}_{s\to k} & \text{dist}_{s\to t} \\ k & \text{dist}_{k\to s} & \text{dist}_{k\to k} & \text{dist}_{k\to t} \\ t & \text{dist}_{t\to s} & \text{dist}_{t\to k} & \text{dist}_{t\to t} \end{array}$$

$$u = \begin{bmatrix} \text{dist}_{s\to s} & \text{dist}_{s\to k} & \text{dist}_{s\to t} \end{bmatrix} \quad v = \begin{bmatrix} \text{dist}_{s\to t} \\ \text{dist}_{k\to t} \\ \text{dist}_{t\to t} \end{bmatrix}$$

Let's see what happens when we take the dot product of u and v, which will be the (s, t) entry of the matrix A^2:

$$\begin{aligned} A^2_{(s,t)} &= \text{dist}_{s\to s} \times \text{dist}_{s\to t} && (1)\\ &+ \text{dist}_{s\to k} \times \text{dist}_{k\to t} && (2)\\ &+ \text{dist}_{s\to t} \times \text{dist}_{t\to t} && (3) \end{aligned}$$

Based on the dot product formula, it seems that in the (s, t) entry of the matrix A^2, we are able to calculate some information involving all paths from node s to node t involving one intermediate step through all the other nodes in the graph. Particularly, the product on line (1) gives us information regarding the path $s \to s \to t$, the product on line (2) gives us information regarding the path $s \to k \to t$, and the product on line (3) gives us information regarding the path $s \to t \to t$. You may notice the paths $s \to s \to t$ and $s \to t \to t$ are simply the same as the path $s \to t$, a direct path from s to t with no intermediate nodes. It may become apparent to you that by induction, the matrix $(A^2)^2_{(s,t)}$ stores the shortest path between $s \to t$, $s \to k \to t$ for all graph vertices k, and $s \to l \to m \to t$ for all combinations of nodes l and m in the graph. We can say that the adjacency matrix $(A^2)^n_{(s,t)}$ stores the shortest path from s to t using *up to* n intermediate vertices.

As shown in the above analogy, we are essentially able to store the result of a dot product operation between the path lengths of $s \to t$ and $s \to k \to t$ for every intermediate node k in the graph. As we explained earlier during our discussion of Floyd-Warshall's algorithm, we specifically need to compute the minimum cost path between all the aforementioned products from the dot product calculation. In the next section, we will show how we can override the standard $+$ and $\times$ operations such that simply computing the squares of adjacency matrices allows us to solve `APSP`, and we will also explain how other properties of environments with overridden $+$ and $\times$ operations may help us solve `APSP` more efficiently.

3.2 Abstract Algebra Overview and Ring Operations

In this section, we cover the basics of the field of abstract algebra as relevant in solving the `APSP` problem. As we showed in section 3.1, on an adjacency matrix of power $A^{2\times n}$, the (s,t) entry encodes information regarding the dot product between s-t paths involving up to n intermediate nodes. When we explained our theory regarding Floyd-Warshall's algorithm, we mentioned that we somehow need to store the minimum cost path between all of the path costs $(s \to t, s \to k_0 \to t, \ldots, s \to k_0 \to k_1 \to \cdots \to k_{n-1} \to t)$ that make up the individual products in the dot product. We may now ask ourselves, when computing a dot product in the form:

$$\begin{aligned}A^2_{(s,t)} &= \mathrm{dist}_{s\to s} \times \mathrm{dist}_{s\to t}\\ &+\ \mathrm{dist}_{s\to k} \times \mathrm{dist}_{k\to t}\\ &+\ \mathrm{dist}_{s\to t} \times \mathrm{dist}_{t\to t}\end{aligned}$$

Is it possible to override the $+$ and $\times$ operations in order to suit our desired min and $+$ operations?

It turns out this is entirely possible as long as our overridden $+$ and $\times$ operations adhere to the rules of *algebraic rings.* A ring is an algebraic structure—that is, a set of elements together with two operations $\oplus$ and $\otimes$—that satisfies certain axioms:

S is an algebraic *semiring* defined as $(S, \oplus, \otimes)$. The below conditions must be met:

1. $(S, \oplus)$ is an Abelian group. In other words, it satisfies the commutative property that $\forall a, b \in S : a \oplus b = b \oplus a$ and it satisfies the identity that $\forall a \in S : a \oplus 0_S = a$, where 0_S refers to the zero element in S.
2. $(S, \otimes)$ is a monoid. In other words, it satisfies the identity $\forall a \in S : a \otimes I = a$, where I refers to the identity element, which can typically be thought of as 1.

3. The $\otimes$ operation must distribute over the $\oplus$ operation. In other words, $\forall a, b, c \in S : a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$.
4. 0_S is an annihilator in $\otimes$. In other words, $\forall a \in S : a \otimes 0_S = 0_S \otimes a = 0_S$.

Note, the difference between a *ring* and a *semiring* is that a semiring does not need to support the additive inverse. Meaning, there does not necessarily have to be a "negative" corresponding to each element

So how does this tie into our `APSP` problem?

We can define our dot product over the semiring $(S, \oplus, \otimes)$ where:

$$\begin{aligned} &a, b \in \mathbb{R} \\ &a \oplus b = \min\{a, b\} \\ &a \otimes b = a + b \qquad \text{Where } + \text{ is the typical addition} \end{aligned}$$

It is straightforward why we can do this, since $(S, \min, +)$ satisfies all of the axioms. You can convince yourself of this fact. It is important to note that S does not have an additive inverse since there is no possible $a, b \in S : \min\{a, b\} = \infty$ unless $a = \infty$ and $b = \infty$.

Now, notice how the dot product defined at the end of section 3.1 changes. Under our ring S, we get:

$$\begin{aligned} A^2_{(s,t)} = \;&\text{dist}_{s\to s} \otimes \text{dist}_{s\to t} \\ \oplus \;&\text{dist}_{s\to k} \otimes \text{dist}_{k\to t} \\ \oplus \;&\text{dist}_{s\to t} \otimes \text{dist}_{t\to t} \end{aligned} \quad = \quad \min\begin{Bmatrix} \text{dist}_{s\to s} + \text{dist}_{s\to t}, \\ \text{dist}_{s\to k} + \text{dist}_{k\to t}, \\ \text{dist}_{s\to t} + \text{dist}_{t\to t} \end{Bmatrix}$$

This exactly what we wanted to compute when we discussed Floyd Warshall's algorithm! As you can see, using algebraic rings, we have come up with a system where simple repeated squaring of the adjacency matrix A over the ring R effectively solves `APSP`. We notice that after squaring the adjacency matrix $n-1$ times, the result contains the answer to `APSP`. Since matrix multiplication takes on the order of $\mathcal{O}(v^3)$ time, where v is the number of vertices in the graph, and since we must effectively square the adjacency matrix v times, we can see how a min-plus squaring algorithm attains a time complexity of $\mathcal{O}(v^4)$ (using naive matrix multiplication).

Our repeated squaring approach can take as little as $\mathcal{O}(n^3 \log(n))$ time if we are more efficient in how we implement *binary exponentiation* rather than using naive matrix multiplication for exponentiation. However, the repeated squaring method is still more inefficient than Floyd-Warshall's algorithm, which takes only $\mathcal{O}(n^3)$ time. It turns out that both the min-plus repeated squaring approach and Floyd-Warshall's algorithm are performing the same operations to compute `APSP`, however Floyd-Warshall's algorithm attains more efficiency using *dynamic programming* and "simulating" the min-plus matrix multiplications in place. This min-plus repeated squaring approach also lies at the heart of the algorithm mentioned in Jan van den Brand's paper, but before we can conceptualize it, we are going to use our knowledge or algebraic rings to talk about polynomial matrices.

3.3 Polynomial Matrices

3.4 Edge Updates

3.5 Hitting Sets

4 Solution

4.1 Algorithm

4.2 Time Complexity Analysis

5 Extension and Comparison

5.1 Broader Algebraic Structures

5.2 General Update Query

5.3 Multiple Matrix Functions

5.4 Better Complexity Bounds

References

1. Jan van den Brand's Notes
2. Sankowski's Paper