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* $\text{dist}_G(s, t)$ *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

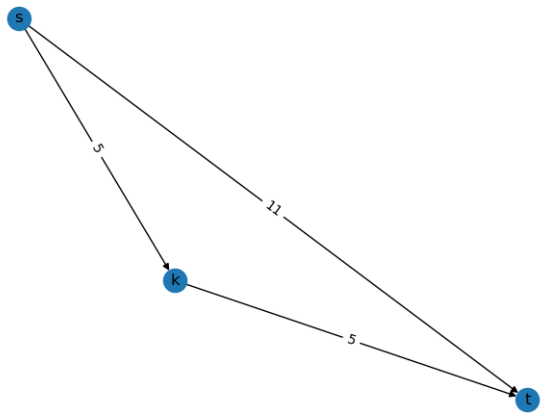
Matrix multiplication is an important concept that we will be making extensive use of to understand the technical components in this report. You are already familiar with the basic method of multiplying two matrices with the same inner dimension, where you multiply the n-th row of the first matrix with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

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

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

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 $A^3_{(s,t)}$ stores the shortest path between $s \to t$, $s \to k \to t$ for all graph vertices k, and $s \to k_0 \to k_1 \to t$ for all combinations of nodes k_0 and k_1 in the graph. We can

say that the adjacency matrix $A^n_{(s,t)}$ stores the shortest path from s to t using *up to* n intermediate vertices.

Why is the induction apparent? If we multiply adjacency matrix A^n by adjacency matrix A^m, we notice that we will be considering paths of the form $s \to k_0 \to \cdots \to k_{n-1} \to k_n \to k_{n+1} \to \cdots k_{n+m-1} \to t$. The total length of this path is $n+m$, which is also the power of the product as $A^n \times A^m = A^{n+m}$, and the number of intermediate nodes is $n+m-1$.

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 × 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 × 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^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:

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

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

It turns out this is entirely possible as long as our overridden + and × operations adhere to the rules of *algebraic rings*. A ring is an algebraic structure—that is, a set of elements together with two binary 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, $\forall a \in S : a \otimes 0_S = 0_S \otimes a = 0_S$.

Alternatively, we define F as an algebraic *field*, denoted as $(F, \oplus, \otimes)$ if it meets the below conditions:

1. $(F, \oplus)$ is an abelian group.
2. $(F \backslash \{0\}, \otimes)$ is also an abelian group, which implies the existence of a multiplicative inverse for every non-zero element.

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. Additionally, note that $\otimes$ does not have to be commutative.

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).

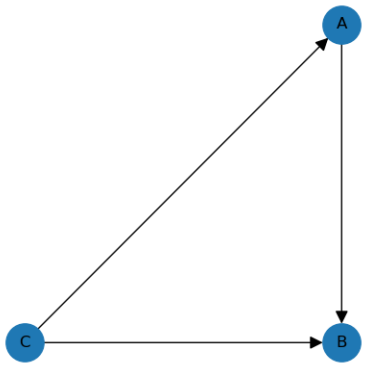
Our repeated squaring approach can take as little as $\mathcal{O}(n^3 \log(n))$ time if we are more efficient in how we implement *binary exponentiation* rather than using naive matrix multiplication for exponentiation. However, the repeated squaring method is still more inefficient than Floyd-Warshall's 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

Now, let's really start building the foundation to Jan van den Brand's `APSP` algorithm by taking about adjacency matrices with polynomial entries!

3.3.1 Unweighted Adjacency Matrices

First let's build some intuition for why we might want to build polynomial adjacency matrices. Going forward, we are specifically going to talk about *unweighted* adjacency matrices where $A_{(s,t)} = 1$ if there is a directed edge $s \to t$ and 0 otherwise. When we refer to simple unweighted adjacency matrices, it turns out we can use simple powers of A (ie. $A, A^2, A^3, \ldots, A^n$) and the traditional operations $+$ and $\times$ to achieve a similar effect to our min-plus repeated squaring algorithm that we explained in section 3.2. However, instead of tracking in the $A^n_{(s,t)}$ the shortest path between start node s and terminal node t using up to n intermediate nodes, in the $A^n_{(s,t)}$ position, we are instead tracking the number of paths that *exist* between s and t on *exactly* n nodes. Why exactly is this? Let's consider a simple example with the graph below:



We can build the unweighted adjacency matrix for the above graph:

$$A = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

Notice again how this first degree adjacency matrix only tells us if a *direct* edge exists between any start node and terminal node. Now, pay attention to the following calculations we will perform on two different paths for the sake of example: $C \to B$ and $B \to C$.

First, let's take $C \to B$. From the graph picture, we can see that there are two paths from $C \to B$. There is one direct path from C to B, which is why $A_{(C,B)} = 1$. Additionally, there is another path through intermediate node A, which corresponds to the path $C \to A \to B$. Let's see what happens when we calculate $A^2_{(C,B)}$, involving the dot product between the vector formed by row C and the vector formed by column B.

$$u = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \quad v = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Like we mentioned earlier in section 3.1, in this format, u represents the existence of a direct path that begins at node C and ends at every other node in the graph, while v represents the existence of a direct path that begins at every other node in the graph and ends at node B. We can think of each product in this dot product as if asking the question "Is there a *direct* path from C to some node k and then is there a *direct* path from that node k to B?" If the answer is yes to both parts of that question, this individual product amounts to 1. Below is the dot product calculation.

$$u \cdot v = (1 \cdot 1) + (1 \cdot 0) + (0 \cdot 1) = 1$$

You can see how the dot product considers every possible choice of intermediate node, and the result of the dot product should be the number of paths from C to B using exactly one intermediate node. Below is the completed product matrix A^2.

$$A^2 = \begin{array}{c} \\ A^2 \\ B^2 \\ C^2 \end{array} \begin{array}{c} \begin{array}{ccc} A^2 & B^2 & C^2 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

As you can see, if a path starting from s and ending at t exists with exactly one intermediate node, then that position will have a 1 in the matrix. By the same inductive reasoning we used in section 3.1, for an adjacency matrix of power

A^n, the $A^n_{(s,t)}$ entry will track the number of paths from s to t on exactly $n-1$ intermediate nodes.

It might be self-explanatory why paths that cannot be formed with any number of intermediate nodes, for example $B \to C$ will have $A^n_{(B,C)} = 0 \quad \forall n \in \mathbb{Z}^+$.

3.3.2 Matix Series Identity

Let's try something new now. What happens if we want to form the accumulated adjacency matrix A' where $A'_{(s,t)}$ tracks the total number of s-t paths that can take any number of intermediate nodes? A straightforward thing to do would be to consider the sum of matrices of powers counting from 0 to h, since each power represents the number of s-t paths on exactly $power - 1$ intermediate nodes.

$$A' = A^0 + A^1 + A^2 + A^3 + \cdots$$

Does this look familiar to you? You might notice that this sum has a similar format to the infinite power series. Note that $A^0 = I$, and the identity matrix adds 1's along the diagonal, which simply adds circular edges from each node to itself. This does not impact any calculations.

We know that a formal power series of the form $1 + x + x^2 + \cdots$ can be represented by the generating function

$$\frac{1}{1-x} = 1 + x + x^2 + \cdots$$

Interestingly, under the conditions of adjacency matrices representing directed acyclic graphs, or over a field with random weights, we can use a "generating function" to efficiently compute A' as well! In particular, we can use the below identity:

$$(I - A)^{-1} = A^0 + A^1 + A^2 + A^3 + \cdots$$

Now, by computing the matrix $A' = (I-A)^{-1}$, we have a new matrix we can use to check the existence of an s-t path. The $A'_{(s,t)}$ entry will be non-zero if a path exists from s to t on any number of intermediate nodes, or 0 if otherwise. To understand this identity rigorously, we can recall our abstract algebra concepts from section 3.2. Notably, our identity only works for `DAG`s and field matrices since we need a defined multiplicative inverse to be defined. In a field, every nonzero scalar has an inverse, and determinants live there, so you can talk about $\det(I - A) \neq 0$ and hence $(I - A)^{-1}$ exists. In an arbitrary ring (say $\mathbb{Z}$ or a semiring), you can't guarantee that. Additionally, we need the series $A^0 + A^1 + A^2 + A^3 + \cdots$ to eventually stop since an infinite sum of matrices does not have much meaning unless the tail vanishes. On a `DAG`, we know the adjacency matrix $A^n = 0$ since there are no paths of length n, as the longest path is of length $n - 1$. However, we will be covering how we can adapt some

of these restrictions to general commutative rings so that our algorithm can cover a broader range of matrices.

3.3.3 Polynomial Matrices as the Next Intuitive Step

1. Polynomial matrices make sense because we want to separate the number of n-degree paths using indeterminates $(x, x^2, x^3, \ldots)$
2. When we define polynomial matrices, we have to define them over a commutative wraparound ring $\mathbb{Z}\backslash\langle x^h \rangle$. This means we are no longer restricted to fields. Finally we tie this into *hinting* that we only need to update portions of the matrix, implying dynamic algorithm

3.4 Edge Updates

1. Keep most of the original document's part on edge updates. Looks good

3.5 Hitting Sets

1. Arjun will finish this part tomorrow morning

4 Solution

1. Almost exactly keep what we have already, just split between the two categories listed below

4.1 Algorithm

4.2 Time Complexity Analysis

5 Extension and Comparison

1. Yiqi is working on it.

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