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

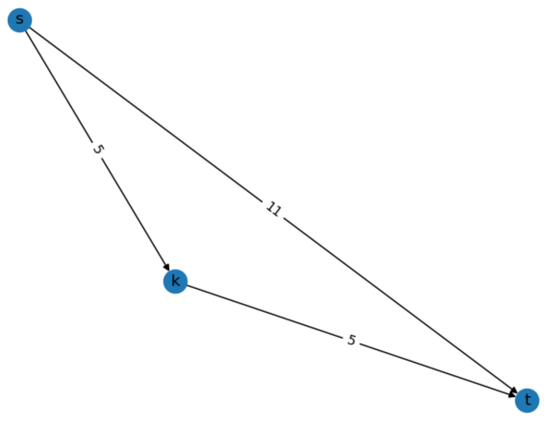
3.1 Matrix Multiplication Recap and Naive APSP

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:

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

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 $\times$ 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^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 $\times$ 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 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:

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

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:

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

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).

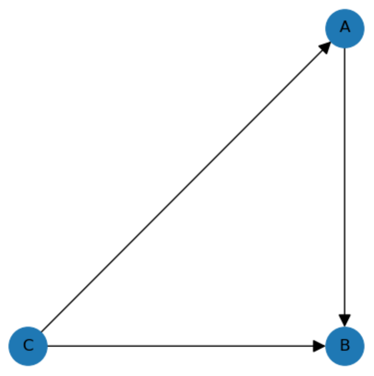
Our repeated squaring approach can take as little as $\mathcal{O}(n^3 \log(n))$ time if we are more efficient in how we implement *binary exponentiation* rather than using naive matrix multiplication for exponentiation. However, the repeated squaring method is still more inefficient than Floyd-Warshall's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

So where do polynomial matrices come into the picture? As discussed previously, our accumulated adjacency matrix A' can track the number of s-t paths on any number of intermediate nodes. However, just the total number of s-t paths does not help us solve `APSP` since we need some mechanism to determine the *shortest* s-t path. An intelligent mechanism to do just this would be to make use of *indeterminates* where we can collect the number of s-t paths taking n intermediate nodes as the coefficient of the x^n term of a polynomial.

Let's formalize this theory. Instead of the geometric series we saw earlier, we can instead represent A' using a formal power series; we'll call this new matrix A'':

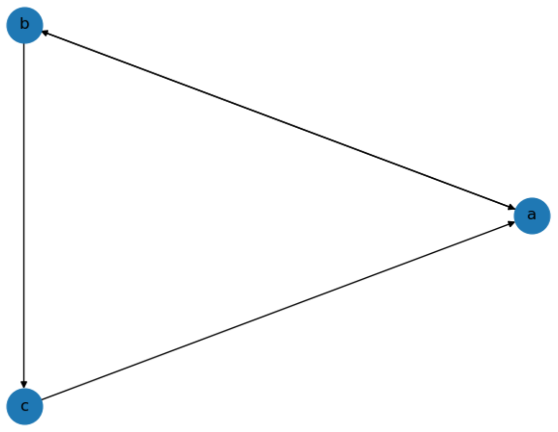
$$A'' = A^0x^0 + A^1x^1 + \cdots + A^nx^n$$

We can use a similar identity to the one we came up with for A' to represent A''.

$$A'' = (I - xA)^{-1}$$

We will show why this identity works in subsection 3.3.4, but for now, let's see a simple example taking it for granted.

Consider the following graph:



With the adjacency matrix below:

$$A = \begin{array}{c|ccc} & a & b & c \\ \hline a & 0 & 1 & 0 \\ b & 1 & 0 & 1 \\ c & 1 & 0 & 0 \end{array}$$

We compute A'' as $(I - xA)^{-1}$:

$$A'' = \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - x \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \right)^{-1} = \left(\begin{bmatrix} 1 & -x & 0 \\ -x & 1 & -x \\ -x & 0 & 1 \end{bmatrix} \right)^{-1}$$

A'' can be simplified to:

$$A'' = \begin{bmatrix} x^2+1 & x & x^2 \\ x^2+x & x^2+1 & x \\ x & x^2 & 1 \end{bmatrix}$$

We will cover why the above matrix corresponds to the inverse of $(I - xA)$ in the section 3.3.4, but for now, notice how the *coefficient* of the x^n term of each entry $A''_{(s,t)}$ indicate the number of s-t paths of length n. It may occur to you that if we can find an algorithm to efficiently compute A'' for a given graph, we can solve `APSP` by simply looking at the lowest degree term of each $A''_{(s,t)}$ entry with a non-zero coefficient.

The example above shows us what we can expect from the matrix A'', but before we can take the results for granted, we must discuss the details of polynomial matrices make mathematical sense.

3.3.4 How are Polynomial Matrices Defined?

Before we get ahead of ourselves, let's understand how exactly we can even define matrices over polynomials and how we can expect them to have inverses. To understand this properly, we will be referring to the abstract algebra concepts we defined in section 3.2.

We traditionally think of matrices as algebraic structures that contain numbers, but in reality we just make the assumption that the matrices we encounter are defined over a *ring* R that contains the real numbers. To pass from "numeric" matrices to polynomial matrices, we must consider defining matrices over the ring $R[x]$, which replaces each scalar in R with a polynomial in the indeterminate x.

Let's be a little more precise in what we mean. For our matrix A'', we notice that any entry $A''_{(s,t)}$ is of the form

$$A''_{(s,t)} = \alpha_0 x^0 + \alpha_1 x^1 + \cdots$$

where $\alpha_0, \alpha_1, \ldots$ are *integers* in $\mathbb{Z}$. We can specifically define $R[x]$ as a ring that suits this purpose. In particular:

1. $R[x]$ is defined over the set of polynomial with integer coefficients, which is represented as $\mathbb{Z}[x]$
2. $(R[x], \oplus)$ is defined as typical polynomial addition where we can combine like terms
3. $(R[x], \otimes)$ is defined as typical polynomial multiplication, which involves convolving the coefficients of each polynomial

While this definition of $R[x]$ allows us to define polynomial matrices, we still cannot assume that a polynomial matrix has an inverse, which we use in our A'' identity. Recall earlier where we specifically defined the identity for A' to only work over *fields*, where every non-zero scalar was defined to have a multiplicative inverse and we also assumed that we only deal with `DAG`s to get rid of the infinite matrix sum. While we cannot make the same assumptions for a general commutative ring like $R[x]$, we can show that under the below listed conditions, the matrix $A'' = (I - xA)^{-1}$ always entries that have a multiplicative inverse and are finite series.

1. To make sure that all the entries of A'' are finite, we can define all polynomial arithmetic to consider only polynomial modulo x^h. Specifically, we will redefine $R[x]$ as $\mathbb{Z}[x]/\langle x^h \rangle$. When we talk about a polynomial modulo x^h, we simply mean that we truncate all of the polynomial terms that have a degree greater than h.

   A size effect of considering only $\mathbb{Z}[x]/\langle x^h \rangle$ is that we won't be able use A'' to track paths of length greater than h. We will discuss this more in section 3.5 and section 4

2. In the ring $R[x] = \mathbb{Z}[x]/\langle x^h \rangle$, the polynomial $p(x)$ is a unit (i.e. invertible) if and only if its constant term is a unit in $\mathbb{Z}$. In other words,

$$p(x) = \alpha_0 x^0 + \alpha_1 x^1 + \cdots + \alpha_{h-1} x^{h-1}$$

   and we notice that there cannot be a $q(x)$ such that $p(x) \cdot q(x) \equiv 1 \mod x^h$ if α_0 is not invertible in $\mathbb{Z}$. Since we defined $(A'')^{-1} = (I - xA)$, entries along the diagonal of $(I - xA)$ must have a non-zero constant term. We may notice in this matrix that the identity matrix guarantees constant terms of 1 along the diagonal, so we can confidently say that $(I - xA)$ always has an inverse in $\mathbb{Z}[x]/\langle x^h \rangle$. It can further be proven that a matrix of the form $(I - xA)$ has an inverse in *any* ring, but we'll spare you the detailed proof.

Due to the conditions we enforce above, we can guarantee that A'' exists. If you recall from section 3.3.3, we magically computed the inverse of an example $(I - xA)$ matrix. Using our knowledge of the ring $\mathbb{Z}[x]/\langle x^h \rangle$, we can show the work for computing the inverse below:

$$(I - xA) = \begin{bmatrix} 1 & -x & 0 \\ -x & 1 & -x \\ -x & 0 & 1 \end{bmatrix} \qquad (I - xA)^{-1} = \begin{bmatrix} x^2 + 1 & x & x^2 \\ x^2 + x & x^2 + 1 & x \\ x & x^2 & 1 \end{bmatrix}$$

Let's compute $(I - xA)^{-1}(I - xA)$ and show that it equals the identity matrix:

$$\begin{aligned}(I - xA)^{-1}(I - xA) &= \begin{bmatrix} 1 - x^3 & -x^3 & 0 \\ -2x^3 & -x^3 + 1 & -x^3 \\ -x^3 & 0 & -x^3 + 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \bmod x^h\end{aligned}$$

If we take $h = 3$, we truncate all terms of each polynomial of degree h or greater. Therefore, you can notice how $(I - xA)^{-1}(I - xA)$ is actually the identity matrix.

It might occur to you that the simplest way to compute A'' would be to simply take the sum of the first h powers of A. To compute every next power of A, $A^{i+1} = A^i \times A$, so we will require a total of h matrix multiplications that take $\mathcal{O}(n^3)$ time each naively. Therefore, we can say to initialize the data structure that holds A'', we require $\mathcal{O}(hn^3)$ time. Doesn't this seem rather inefficient compared to just using the Floyd-Warshall algorithm in $\mathcal{O}(n^3)$ time? Well, the benefit of using this data structure is that we can more efficiently support computing `APSP` on graphs that *change* after initialization. This is the essence of dynamic algorithms, as we tradeoff potentially heavier initialization costs for lighter costs to compute changes. We will explore how exactly we achieve better update efficiency in the next section.

Now that we have covered the fundamentals of polynomial matrices, in the next sections, we will talk about intelligent ways to modify them dynamically, allowing us to conclude how we might arrive at Jan van den Brand's algorithm.

3.4 Edge Updates

As we mentioned at the end of the previous section, the real benefit of using a polynomial matrix as our adjacency matrix in terms of time efficiency is that we can more effectively track *changes* in `APSP` for *changing* graphs as opposed to a simple re-running of the Floyd-Warshall algorithm. Let's explore exactly how we can be more efficient:

We want to accomplish a time complexity that is better than $\mathcal{O}(n^3)$, which is the Floyd-Warshall time complexity, when we update our graph G by either adding or removing nodes. When representing a graph G as an adjacency matrix A, we realize that we can add or remove a direct edge by modifying the entry in $A_{(s,t)}$.

One way we can mathematically represent this type of modification is through the expression:

$$A + u \cdot v^\top$$

where one of u or v is an elementary column vector e_i, which is a vector of all 0's and a 1 in the i-th position. It is important to note that u and v are both *column* vectors (i.e they have the dimension of $n \times 1$, hence $n \times 1$ vector multiplied by a $1 \times n$ vector is an $n \times n$ matrix). You can see how if we set $u = e_i$, the *row* vector $v^\top$ replaces the i-th row of A. Setting $v = e_i$ replaces the i-th column of A with the *column* vector u. We call these types of updates as *rank-one* updates.

Why is this notation particularly useful? We can use an identity called the Sherman-Morrison Identity to efficiently calculate A'' using rank-one updates. Below is the Sherman-Morrison Identity:

$$\begin{aligned}(M + uv^\top)^{-1} &= M^{-1} - \frac{M^{-1}uv^\top M^{-1}}{1 + v^\top M^{-1}u} \\ &= M^{-1}u(1 + v^\top M^{-1}u)^{-1}v^\top M^{-1}\end{aligned}$$

The derivation of this identity is not pivotal to understanding Jan van den Brand's algorithm, however, it is useful to understand why we can use this identity even though it requires a division which may not always be defined on a ring. In reality, we can invert $1 + v^\top M^{-1}u$ because the M matrix is really just our $(1 - xA)$ matrix, which we provided an argument for being invertible in section 3.3.4. Let's extend the Sherman-Morrison Identity to our context to make this more obvious:

We want to compute $A''' = (I - x(A + uv^\top))^{-1}$. Note that $M = (A'')^{-1} = (I - xA)$.

$$\begin{aligned} A''' &= (I - x(A + u\,v^\top))^{-1} \\ &= (I - xA - x\,u\,v^\top)^{-1} \\ &= (M - x\,u\,v^\top)^{-1} \end{aligned}$$

Let $\quad U = x\,u, \quad V^\top = -\,v^\top,$

$$A''' = (M + U\,V^\top)^{-1}$$

(by Sherman–Morrison)

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

$$\begin{aligned} M^{-1}U &= x\,M^{-1}u, \quad V^\top M^{-1} = -\,v^\top M^{-1}, \\ V^\top M^{-1}U &= (-\,v^\top)\,(M^{-1})\,(x\,u) = -\,x\,(v^\top M^{-1}u), \\ I + V^\top M^{-1}U &= I - x\,(v^\top M^{-1}u), \end{aligned}$$

$$\begin{aligned} \therefore \quad A''' &= M^{-1} - \bigl(x\,M^{-1}u\bigr)\bigl(I - x\,(v^\top M^{-1}u)\bigr)^{-1}\bigl(-\,v^\top M^{-1}\bigr) \\ &= M^{-1} + x\,M^{-1}u\bigl(I - x\,(v^\top M^{-1}u)\bigr)^{-1}v^\top M^{-1} \end{aligned}$$

We can see that the computation for A''' depends on four operations:

1. $M^{-1}U$, an $n \times n$ polynomial matrix times an n vector. For each of the n^2 entries of M, you multiply two h-degree polynomials, which takes $\mathcal{O}(h \log h)$ time using the Fast Fourier Transform (`FFT`). This multiplication totally takes $\mathcal{O}(hn^2)$ (omitting a negligible factor of $\log h$).
2. $V^\top M^{-1}$, similar to above, also takes $\mathcal{O}(hn^2)$.
3. $(M^{-1}U)(V^\top M^{-1})$, the outer product of two length-n polynomial vectors: n^2 polynomial multiplications (since $n \times 1$ multiplied by $1 \times n$) at $\mathcal{O}(h \log h)$ each, for $\mathcal{O}(hn^2)$.
4. Inversion of the scalar polynomial $1 + V^\top M^{-1}U$ (degree $< h$): by repeated squaring, (similar to how we calculated the inverse of A'') and `FFT`-based multiplications takes $\mathcal{O}(h \log^2 h) \approx \mathcal{O}(h)$ (Since we have $\mathcal{O}(\log h)$ squares and a cost of $\mathcal{O}(h \log h)$ per square).

Based on the above, we can see that computing A''' using the Sherman-Morrison Identity takes $\mathcal{O}(hn^2 + h \log^2 h) = \mathcal{O}(hn^2)$ time, which beats $\mathcal{O}(n^3)$ from Dijkstra's depending on choice of h.

It is important to observe that the polynomial-series coefficients can grow very large, making exact arithmetic expensive. To avoid this, we compute all coefficients modulo a large prime p, working over the finite field $\mathbb{Z}_p$. Of course, if the true number of s-t paths is divisible by p, then the corresponding coefficient becomes zero modulo p, potentially masking reachability.

However, by the Schwartz-Zippel lemma, any nonzero polynomial of degree at most h can vanish modulo p with probability at most

$$\frac{\deg(f)}{p} \leq \frac{h}{p}.$$

In particular, choosing $p > 2h$ guarantees a failure probability of at most $\frac{1}{2}$. Moreover, if we repeat the construction independently for $O(\log n)$ different random primes $p_1, \ldots, p_k$, then by a union bound the chance that any one of the n^2 entries is falsely zero in *all* runs becomes negligible (e.g., n^{-c} for any constant c). Hence, with high probability, no genuine reachability is lost.

3.5 Hitting Sets

In section 3.4, we covered how we can manipulate polynomial matrices to track `APSP` for path lengths up to length h in $\tilde{\mathcal{O}}(hn^2)$ time per node update. One limitation of tracking the `APSP` using A'' from section 3.3.3 is that we only track shortest s-t paths of length h, since we assumed that all polynomial arithmetic would be limited by modulus x^h on the ring $\mathbb{Z}[x]/\langle x^h \rangle$. However, a shortest s-t path may have a length that is greater than h, so how would we recover such paths without defaulting to Floyd-Warshall's algorithm that takes $\mathcal{O}(n^3)$ time?

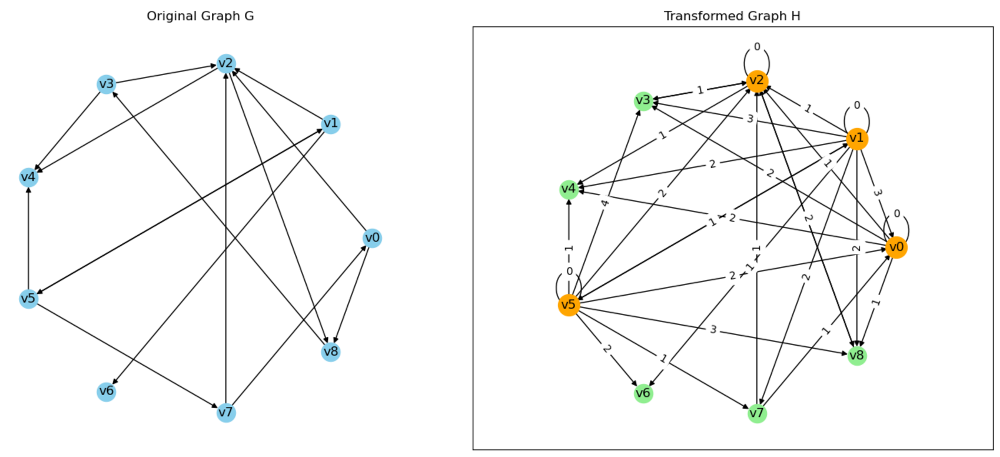
To track the shortest paths of an arbitrary length n, we would need to find a way to deal with "chunks" of the path that are only of length h. A straightforward way to do this is to somehow subdivide the nodes of the graph G into a new graph H of $\frac{n}{h}$ nodes, where each node somehow is capable of representing a $\frac{n}{h}$ section of nodes from G. It turns out that we can subdivide the original graph G by randomly sampling some $\frac{n}{h}$ nodes, with a high probability that shortest paths pass through the randomly sampled $\frac{n}{h}$ nodes of H.

Let's formalize this approach:

> We define the reduced graph $H(V, E')$ as a graph with the following properties using only the h-bounded distances from A'':
>
> 1. V is the same vertex set as in $G(V, E)$
> 2. E' can be defined in the following manner:
>   Define $R \subseteq V$ such that $|R| = \tilde{\mathcal{O}}(\frac{n}{h})$. We pick R from V uniformly at random.
>
>   For *every* $v \in V$ and *every* $r \in R$, we add *only* the following edges into E': (1), we add the directed edge $v \to r$ with a weight of $\text{dist}_{h-bounded}(v, r)$ and (2), we add the edge $r \to v$ with a weight of $\text{dist}_{h-bounded}(r, v)$. In simpler terms, we're building a "hub-and-spoke" mini-graph: for every node v and every landmark $r \in R$, we're adding exactly two directed spokes—one $v \to r$ and the other $r \to v$—and we label each with the true distance in G.

For the sake of being explicit, notice that H has $|V| \times |R| \times \mathcal{O}(\frac{n}{h}) = 2 \times n \times \mathcal{O}(\frac{n}{h}) = \mathcal{O}(\frac{n^2}{h})$ edges.

Below is a visualization for your convenience:



With this structure for H, we can be confident that *every* shortest path of a length greater than or equal to h hits some $r \in R$ within each consecutive block of h edges. In other words, every long s-t path in G can be "stitched together" in H by hopping from one landmark node $r \in R$ to the next in steps guaranteed to be at most h edges, so that the total cost in H exactly matches the cost of the s-t path in G.

Why is this? Since we know $|R| = \tilde{\mathcal{O}}(\dfrac{n}{h})$, we can restate this as $|R| = \frac{p \cdot n}{h}$ for some $p \in \mathbb{N}$.

For a fixed (s,t) whose shortest path has a length that is greater than or equal to h, we can say the probability that none of the first h vertices lie in R is:

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

Recall, $(1 - \frac{x}{k})^k \leq e^{-x}$, and $e^{-p} = (2^{\ln e})^{-p} = (2^{\frac{1}{\ln 2}})^{-p}$, and $\Omega(p) = \frac{p}{\ln 2} \therefore e^{-p} = 2^{-\Omega(p)}$

There are at most n^2 choices for pairs of (s,t), therefore, the probability that *some* pair fails to "hit" R is $n^2 \cdot 2^{\Omega(p)}$, which implies that the probability that every pair "hits" R is $1 - \frac{n^2}{2^{-\Omega(p)}}$.

Supposing that $\text{dist}_G(s,t) \geq h$, we know the probability that we "hit" a single node in R within h nodes is $1 - \frac{n^2}{2^{-\Omega(p)}}$. Once we "hit" the first landmark node r_0, if the remaining length of the s-t path is greater than or equal to h, we again face the probability of $1 - \frac{n^2}{2^{-\Omega(p)}}$ to "hit" another landmark node in R within an additional h nodes Therefore, we can reliably write any s-t path as a dependency chain as follows: $s \to r_0 \to r_1 \to \cdots \to r_{i \leq \frac{n}{h}} \to t$.

Above, we saw how we can convert our original graph G with n nodes into a reduced graph H with $\frac{n}{h}$ landmark nodes which are strongly connected to all non-landmark nodes. Just to be precise, remember that while H also has n nodes like G, our landmark nodes are in the set $R \subseteq V$ and $|R| = \tilde{\mathcal{O}}(\frac{n}{h})$. Now, let's discuss precisely how we can recover s-t paths of length greater than or equal to h using our reduced graph H.

We know that $\text{dist}_G(s,t) = \min\{\text{dist}_{h-bounded}(s,t), \text{dist}_H(s,t)\}$, since either there is a smaller than h-length s-t path already present in A'' or we need to use our reduced graph H to compute the distance. To compute the s-t path distance on H, we can simply use Dijkstra's algorithm to solve `APSP` for H. This has a time complexity of $\mathcal{O}(|V| \times |E'|) = \mathcal{O}(n \times \frac{n^2}{h}) = \mathcal{O}(\frac{n^3}{h})$.

In section 3.4, we saw how we can achieve a time complexity of only $\mathcal{O}(hn^2)$ to update our A'' data structure. When we update our reduced graph H, we only need to update the edges from the new node v to each $r \in R$ and from each $r \in R$ to v. This takes $\mathcal{O}(|R|) = \mathcal{O}(\frac{n}{h})$ time. Since we need to follow this up with an additional run of Dijkstra's algorithm, we can see the total update cost for H is $\mathcal{O}(|V| \times |R|) = \mathcal{O}(n \times \frac{n}{h}) = \mathcal{O}(\frac{n^2}{h})$ time.

4 Solution

Using the technical background from section 3, we have everything we need to go through the dynamic `APSP` algorithm from Jan van den Brand's notes.

Let's walk through the algorithm start to finish! Let's start with maintaining the h bounded shortest path distances. These are the distances between node pairs where h is the maximum length of the shortest path. We use the inverse of the polynomial matrix, $A'' = (I - xA)^{-1}$, over a truncated ring $R[x] = \mathbb{Z}[x]/\langle x^h \rangle$ to do this, where A is the adjacency matrix of the graph, and the minimum degree of x in an entry $A''_{(s,t)}$ represents the shortest distance between nodes s and t. The Sherman-Morrison identity is then used to modify the inverse matrix, A'', for updates that affect edges incident to a single vertex.

The next part involves extending the h-bound pair distances to cover distances of arbitrary length n. We do this using hitting sets, where we randomly pick $\frac{n}{h}$ nodes from the original graph. This is based on the idea that any long path is highly likely to pass through at least one of the randomly sampled nodes. We then construct a reduced graph H, where each node is connected to the sampled nodes using the short distances from before up to length h.

Finally, we run Dijkstra's algorithm for each node v on the auxiliary graph to compute the shortest path between v and every landmark node r. We store the computed `APSP` on H.

In the end, we have two pieces of information to work with. We have the h-bounded distances in A'', represented by $\text{dist}_{h-bounded}(s,t)$, which are exact for the paths of length $\leq h$. We also have the reduced graph distances stored in H, represented by $\text{dist}_H(s,t)$, which are accurate for longer paths greater than h. For each node pair (s,t), we return $\min(\text{dist}_{h-bounded}(s,t), \text{dist}_H(s,t))$ as $\text{dist}_G(s,t)$.

To analyze the time complexity of this algorithm, we can analyze the time complexities related to the two data structures we maintain: A'' and H:

1. We need to initialize our data structure from **theorem 1.0.0**, which is essentially our matrix $A'' = (I - xA)^{-1}$ from section 3.3.3, combined with our reduced graph H mentioned in section 3.5. As we mentioned at the end of section 3.3.4, it takes $\mathcal{O}(hn^3)$ time to initialize A'', and taking $h = \sqrt{n}$ as mentioned in section 3.4, we can accomplish this in $\mathcal{O}(n^{3.5})$ time. Additionally, as mentioned at the end of section 3.5, it takes $\mathcal{O}(\frac{n^3}{h})$ time to initialize H, and taking $h = \sqrt{n}$, we find it takes $\mathcal{O}(n^{2.5})$ time. In total, initialization takes $\mathcal{O}(n^{3.5} + n^{2.5}) = \mathcal{O}(n^{3.5})$ time, as desired from the theorem.

2. We need to update our data structure mentioned in the previous part. As mentioned in section 3.4, we can accomplish adding and removing an edge from A'' using a *rank-one* update in $\mathcal{O}(hn^2)$, which taking $h = \sqrt{n}$, is $\mathcal{O}(n^{2.5})$ time. In order to update our reduced graph H, we take $\mathcal{O}(\frac{n^2}{h})$ time as mentioned at the end of section 3.5. Again, taking $h = \sqrt{n}$, we get an update time of $\mathcal{O}(n^{1.5})$. In total, we have an update time complexity of $\mathcal{O}(n^{2.5} + n^{1.5}) = \mathcal{O}(n^{2.5})$, as desired from the theorem.

Looking at the time complexities, it is obvious that the Floyd-Warshall algorithm is more efficient in initialization costs, as it takes $\mathcal{O}(n^3)$ to initialize compared to the $\mathcal{O}(n^{3.5})$ that Jan van den Brand's algorithm takes. However, Jan van den Brand's algorithm is more efficient when dealing with updating edges in the graph G, as we can effectively update `APSP` in $\mathcal{O}(n^{2.5})$ time whereas the Floyd-Warshall algorithm would have to be re-run, taking $\mathcal{O}(n^3)$ time again. We can see how Jan van den Brand's algorithm prioritizes dynamic update costs while the Floyd-Warshall algorithm prioritizes static cost.

5 Extension and Comparison

5.1 Broader Algebraic Structures

5.2 General Update Query

References

1. Jan van den Brand's Notes

2. Sankowski's Paper