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 structre 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

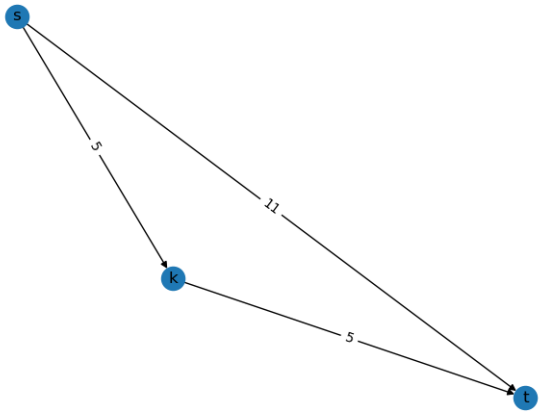
Matrix multiplication is an important concept that we will be making extensive use of to understand the technical components in this report. You are already familiar with the basic method of multiplying two matrices with the same inner dimension, where you multiply the n-th row of the first matrix with 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 travelling from $s \to t$ vs. the path we get by travelling 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 intutive to think of graphs in terms of adjacency matricies 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.

For those already familiar with computing the shortest path between some start node s and some terminal node t on a directed and optionally weighted graph, it might be intuitive to also represent the calculation in terms of a dot product. Suppose some vector v corresponds to the to the best known path costs from s to every other node t in the graph, and the index of an element in v corresponds to the node t that that element represents. An example of v may be the following

$$v = \begin{bmatrix} 1 & 2 & 5 & 7 & 0 & 10 \end{bmatrix}$$

3.2 Abstract Algebra Overview and Ring Operations

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