Program 2: All Pairs Shortest Paths

Fahad Alqahtani | Ishan Bhatt | James Skripchuk | Karan Shah
[fsalqaht|ivbhatt|jmskripc|krshah3]@ncsu.edu
Code(NCSU access): https://github.ncsu.edu/jmskripc/Program2

I. Introduction

In this project, we aim to compare the performance of two popular algorithms for solving the all-pairs shortest path problem:

- N iterations of the Dijkstra's algorithm (1 iteration per node)
- Floyd-Warshall Algorithm

We implement both these algorithms in C++ and analyze their performance on different graph structures using the values for runtime and number of comparisons. We make some hypotheses related to the performance of these algorithms and try to validate or invalidate them by designing experiments to test them. We explain how we executed the experiments and report our findings in this paper.

II. Implementation Description

A. General Flow - Keeping track of time & comparisons

We have a controller_main.cpp where the main() function is defined. This program takes three command-line arguments(CLAs): the algorithm-string ("d" for n-iterations of Dijkstra / "fw" for Floyd-Warshall); path to the input .gph file. If either of these CLAs are not provided, the program exists with an error.

The main program then uses a utility function (defined in utils.cpp) in order to read the .gph file and save it as an **Adjacency Matrix** in main memory. This adjacency matrix is then passed onto the functions that implement the shortest-path algorithms (defined in shortest_path_algos.cpp). This adjacency matrix is stored as a vector of vectors (vector class available in C++: Standard Template Library). Initially, this caused us to have extremely large run-times (especially for Floyd-Warshall) with the same number of comparisons because STL vectors are designed in a way to enable easy dynamic reallocation of memory and for that reason even though their random-access time-complexity is constant time, this constant for them is much higher than the constant for standard C++ 2-D arrays. We found out that using the -O3 flag to optimize with g++ compiler helped the compiler detect the absence of any dynamic allocation and optimize the final byte-code to provide better runtimes. Using the -O3 flag with g++ (for vectors of vectors) improved our runtimes by a factor of 20 (approx.)

Floyd-Warshall needs to be called only once but Dijkstra's algorithm must be run n-times with different start nodes. To keep Dijsktra reusable, we have implemented it in the standard-way (with one start node); The logic for calling it n-times with different start nodes and combining the results (shortest-distances, sum of time and sum of comparisons) is written in the main program. We are using pointers to receive time and number of comparisons from the algorithms. The results (on stdout) are

being printed from within the shortest_path_functions while the statistics (on stderr) are being printed by the main program.

Dijkstra also needs a minimum priority queue which was custom implemented by us via a heap. As part of our experiential procedure, we went past implementing a standard binary heap, and made it so that you could choose any arbitrary number to create a D-heap. This code is written in utils.cpp.
We provide **compile.sh** to compile the code; **run_apsp.sh** and **silent_run_apsp.sh** to run the code. Using these three scripts, we shield the user from the complexity in the code. Our code base was tested and is guaranteed to work on NCSU VCL (Ubuntu Base 18.04).

B. Floyd-Warshall

Floyd-Warshall is an all pair shortest path finding algorithm in a directed weighted graph. It can handle both positive and negative edge weights. It breaks the problem into subproblems and combines the answers of the subproblems to solve the big problem. Hence it follows the dynamic programming approach. The basic intuition is that if the shortest path from say vertices X to Z can be the path that is already found between X and Z or the sum of the shortest path from X to Y and the shortest path between Y and Z. Floyd Warshall is an n^3 algorithm guaranteed to find all pairs shortest paths. Floyd Warshall finds its applications in multiple computer science domains like routing in networks and optimal path finding etc.

In our implementation of the Floyd Warshall Algorithm, we have made the following assumptions and modifications to get better results:

- Infinity is INT_MAX, skip INFINITY _x summations to avoid overflow (still count them as comparisons)
- Less than n^3 comparisons are needed
- Vector vector implementation was initially very slow but we used -O3 optimization in g++ to reduce runtime by a factor of 20

C. n-Dijkstra

Dijsktra's is a greedy single source shortest path finding algorithm in a weighted graph. Unlike Floyd Warshall, Dijkstra's algorithm cannot handle negative weights. The intuition behind dijkstra's is that it works by marking one vertex at a time as it discovers a shortest path to that vertex. Initially we mark just the vertex since we know its (trivial) shortest path from itself. Consider the shortest path from that vertex to some vertex w. It traverses a sequence of vertices, with a final vertex v that then connects to w to finish the shortest path to w. The key observation is that the path up to v must be a shortest path to v. Because if it weren't, we could replace that path to v with a shorter path, which would also shorten the path to w, which would contradict the claim that we have a shortest path to w. The complexity of Dijkstra's is O(n + mlogn) where m is the number of edges and n is the number of nodes in the graph.

In our implementation of Dijkstra's, the following are some key points:

- Nodes are added to the heap as we discover them
- All results are from binary min heap unless specified otherwise
- We measure the runtime of each individual Dijkstra run (in milliseconds) and then add them to get the final time in seconds. So we don't lose accuracy

D. Min Heap (Priority Queue)

We implemented a D-ary Min Heap because, based on our proofs in the assignments, we wanted to experimentally verify the performance of the Dijsktra's algorithm for cases when we used a D-ary heap instead of a binary heap as a priority queue. We use an STL vector of STL pairs to store the nodes of the heap. To store the node-index we use an int and to store the distance to that node, we use long datatype. We use the long type to store the distance because we use INT_MAX for denoting infinite-distance or no-connection throughout our code.

For running an iteration of Dijkstra with V vertices, we allocate a heap with capacity to hold upto V nodes since we do not expect to ever have more than V nodes in the heap during a Dijkstra run on a graph with V vertices.

The D-ary heap has the following functionality (N is number of nodes in the heap):

- insertKey(node) is performed in O(log(N)/log(D)) time because that's the order of comparisons required to insert a new node at the bottom of the node and percolate it all the way up to the root (worst-case). Going up each level requires 1 comparison.
- extractMin() is performed in O(D * log(N)/log(D)) time because that's the order of comparisons required to remove the root; replace it with the last-inserted leaf-node and then percolate the leaf-node from the root all the way down to the leaves (worst-case). Going down each level requires D comparisons.
- decreaseKey(node, newKey) is also performed in O(log(N)/log(D)) time because that's the order of comparisons required to insert a new node at the bottom of the node and percolate it all the way up to the root (worst-case). Going up each level requires 1 comparison.
- We have a comparison_counter as the data-member of the DMinHeap class that helps us keep track of all relevant comparisons made by a particular instance of the DMinHeap.
- Other trivial functions like getters and setters are also implemented.

III. Experimental Design

A. Hypotheses

1. Runtime complexity:
   a) **Floyd-Warshall all-pairs-shortest path algorithm** when applied to a graph represented as an adjacency matrix has a runtime complexity of $O(n^3)$. Therefore, we believe that : if we try running the algorithm on graphs of different-sizes (keeping the edge density constant), we should be able to observe that both the number of comparisons and runtime show a strong correlation with the **n^3 - trend line.**
   b) Dijkstra's single source shortest paths algorithm when applied to a graph represented as an adjacency matrix has a runtime complexity of O(n+m*logn). For all-pairs-shortest-paths, we need to run an iteration of **Dijkstra starting from each node** in the graph making the runtime-complexity O(n*(n+m*logn)). Therefore, we believe that : if we try running the algorithm on graphs of different-sizes (keeping the edge density constant), we should be able to observe that both the number of comparisons and runtime show a strong correlation with the **n*(n+m*logn) - trend line.**
2. Varying Edge density in a graph (n-Dijkstra vs. FloydWarshall):
   a) We believe that if we keep the number of nodes in a Graph constant and keep on increasing the edge-density then n-Dijkstra's performance will take a hit.

b) However, Floyd-Warshall's performance in terms of number of comparisons and runtime should remain unaffected.

3. Dijkstra and D-ary Min Heaps:
   a) Implementing **Dijkstra's algorithm with D-ary MinHeaps** would result in n calls to the **extractMin** utility of the heap. Each call, in the worst-case would cost **d*log(n)/log(D) comparisons.** Therefore, we believe that the total comparisons made by the extractMin utility of the heap should show a strong correlation with the **D*n*log(n)/log(D) -trendline.**
   b) Implementing **Dijkstra's algorithm with D-ary MinHeaps** would result in m calls (worst-case) to the **deceaseKey** utility of the heap. Each call, in the worst-case would cost **log(n)/log(D) comparisons.** Therefore, we believe that the total comparisons made by the extractMin utility of the heap should show a strong correlation with the **m*log(n)/log(D) -trendline.**

B. Experimental Setup

1. Run-time complexity:
   a. Floyd-Warshall: To test this hypothesis, we use graphs ranging from **n = 500 to n = 1000**. The edge density is kept constant. Number of edges: **$m = n * \log_{10}(n)$**
   b. n-Dijkstra: To test this hypothesis, we use graphs ranging from **n = 2000 to n = 10000**. The edge density is kept constant. Number of edges: **$m = n * \log_{10}(n)$**
2. Varying Edge density in a graph (n-Dijkstra vs. FloydWarshall):
   a. n-Dijkstra: To test this hypothesis, we use graphs with **n = 5000**. We use edges in the range: **m= 5000 to 105000** with an interval of 20000.
   b. Floyd-Warshall: To test this hypothesis, we use graphs with **n = 1000**. We use edges in the range: **m= 5000 to 105000** with an interval of 20000.
3. Dijkstra and D-ary Min Heaps:
   a. We use graphs with 1024 nodes (n = 1024) and 10240 edges (m = 10240). We vary D from 2 to 18 in intervals of 4.

- All results were reported as a median of three runs.
- Experiments were run on a computer with a 2 GHz processor.
- Code was tested on NCSU VCL (Ubuntu 18.04 LTS Base) instance.
- Code was compiled with gcc version 7.5.0
- Graphs were generated using the MST repository on github

IV. Experimental Results & Analysis

1. Table 1.1 and Table 1.2 show the run-times of n-Dijkstra & FloydWarshall with different graph-sizes respectively.

Table 1.1: Runtimes for n-Dijkstra's algorithm

Vertices	Edges	Dijkstra	
		Runtime	Comparisons
100	200	0	971
500	1400	0	7914
1000	3000	0	18084
2000	6600	0.1	41451
2500	8500	0.1	54028
4000	12000	0.2	88674
6000	22668	0.4	148145
8000	31224	0.6	206021
10000	40000	0.9	265977

Table 1.2: Runtimes for Floyd Warshall

Vertices	Edges	Floyd Warshall	
		Runtime	Comparisons
500	1400	2.0	98640639
600	1667	4.2	173201909
700	1992	6.8	273350425
800	2322	8.9	406958135
900	2659	13.0	594501612
1000	3000	20.0	805688383

We plot these data on graphs to visualize them better. Graphs 1.1 through 1.4 show the runtimes and number_comparisons plotted on a graph.

Looking at the graphs 1.1 and 1.2, we notice that the actual observed values tend to stay under the projected trend-lines. One possible explanation for this observation is as follows:

- Trend-line denotes the worst-case runtime and number of comparisons
- For dijkstra, in the worst case, m calls to the decreaseKey operation are made. In other words, in the worst case, Dijkstra's algorithm finds a better path to some node by looking at each edge of the graph.
- In random examples, it is reasonable to believe that not every edge in the graph improves the distance to some node from the start-node.
- For this reason, assuming m to be the number of calls to decreaseKey is an over-approximation. Therefore, we observe the real observations tend to be lesser than the trend-line.

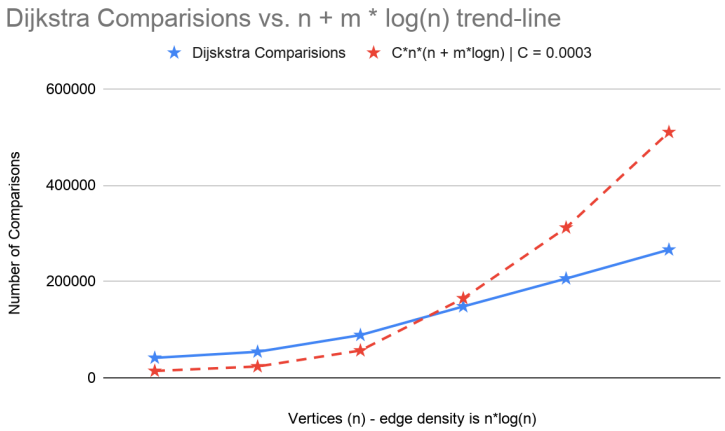

Figure 1.1: Dijkstra comparisons

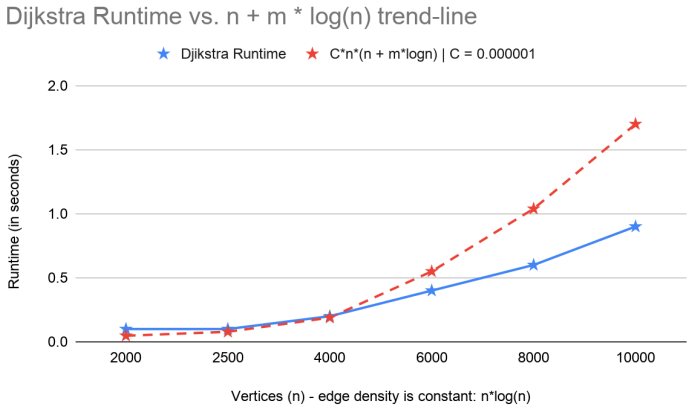

Figure 1.2: Dijkstra runtimes

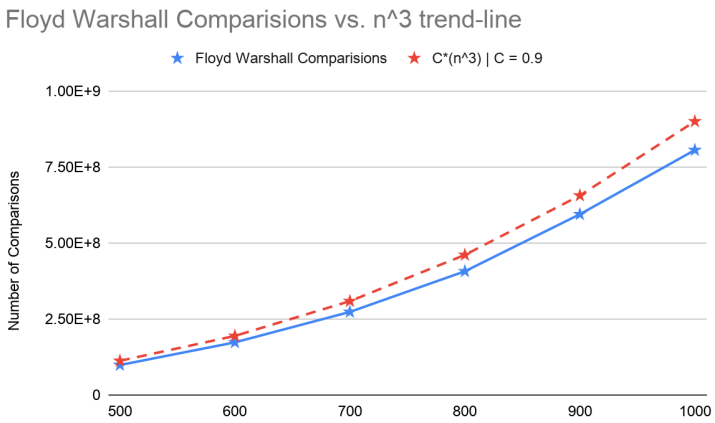

Figure 1.4: Floyd-Warshall comparisons

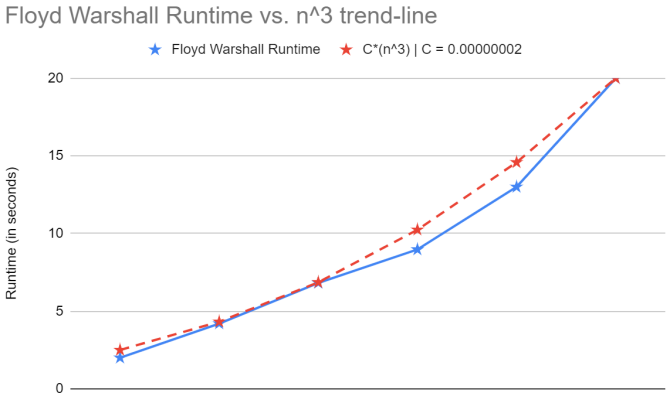

Figure 1.3: Floyd-Warshall runtimes

2. Varying Edge density in a graph (n-Dijkstra vs. FloydWarshall):

Table 2.1 and 2.2 shows the runtime and comparisons for graphs with fixed vertices. We increase the edges to range from sparsely connected graphs to strongly connected. The tables are for dijkstra's and floyd warshall respectively.

Table 2.1: Runtimes and comparisons for n-Dijkstra's algorithm

		Dijkstra	
Vertices	**Edges**	**Runtime**	**Comparisons**
5000	5000	1	82154
5000	25000	1.3	130693
5000	45000	1.7	156359
5000	65000	2	179663
5000	85000	2.3	202310
5000	105000	2.6	223896

Table 2.2: Runtimes and comparisons for Floyd Warshalls's algorithm

Vertices	Edges	Floyd Warshall	
		Runtime	Comparisons
1000	5000	3.9	872717329
1000	25000	3.8	972974903
1000	45000	3.9	986083829
1000	65000	3.8	990049042
1000	85000	3.9	992711928
1000	105000	3.8	994207557

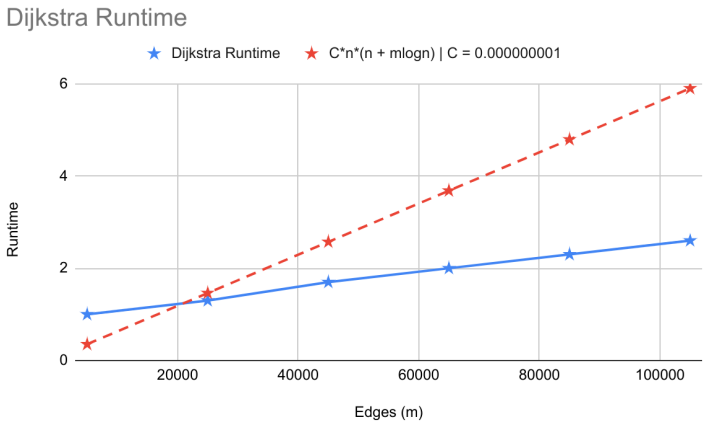

Figure 2.1: *Dijkstra runtime*

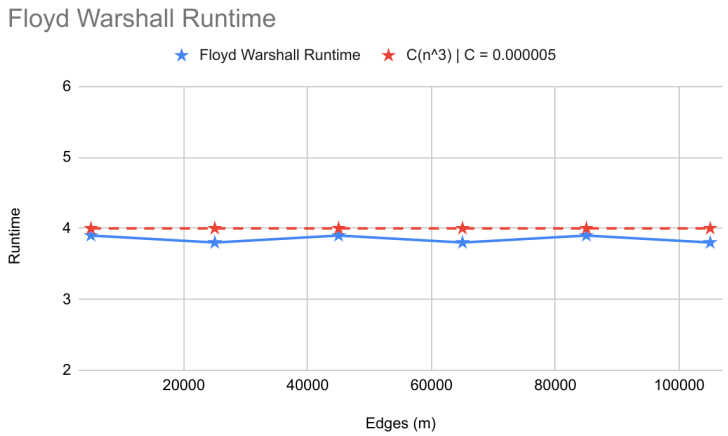

Figure 2.2: *Floyd warshall runtime*

From figures 2.1 and 2.2 above, we can observe that as the edges keep increasing, the runtime for n-Dijkstra slowly increases and stays below the projected trend lines from the upper bound of the time complexity of dijkstra. On the other hand, the runtime values for Floyd-Warshall remain fairly constant. This can be attributed to the fact that the complexity of Floyd-Warshall is dominated by the cube of the number of vertices it has and since they are constant, the runtimes don't change. The runtime of n-Dijkstra however, depends on the factor m*logn where m is the number of edges and hence increases with increasing connectivity in the graph.

3. Dijkstra and D-ary Min Heaps:
   Tables 3.1 and 3.2 show the number of comparisons made within a D-ary heap by a single run of Dijkstra on graphs with 1024 nodes and 10240 edges by varying D. Table 3.1 shows the comparisons made by extractMin operations while Table 3.2 shows comparisons made by the decreaseKey operation.

Table 3.1 Comparisons made by extractMin for a run of Dijkstra

Nodes	Edges	D	Comparisons made by extractMin
1024	10240	2	15207
1024	10240	3	15233
1024	10240	6	19942
1024	10240	10	26784
1024	10240	14	34053
1024	10240	18	39759

Table 3.2: Comparisons made by decreaseKey for a run of Dijkstra

Nodes	Edges	D	Comparisons made by decreaseKey
1024	10240	2	4872
1024	10240	3	3908
1024	10240	6	3262
1024	10240	10	3033
1024	10240	14	2926
1024	10240	18	2887

Figures 3.1 and 3.2 show the data presented in Tables 3.1 and 3.2 in a graphical format.

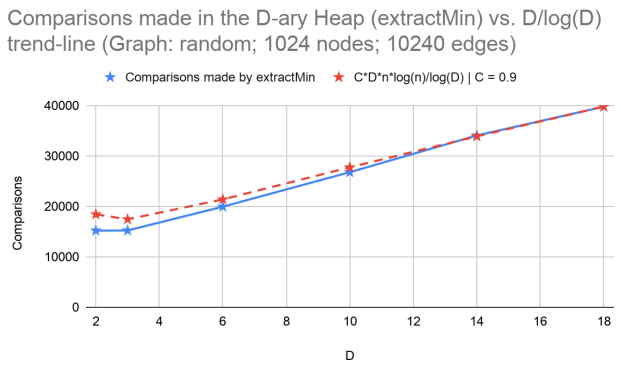

Figure 3.1: *Comparisons made by extractMin for a run of Dijkstra*

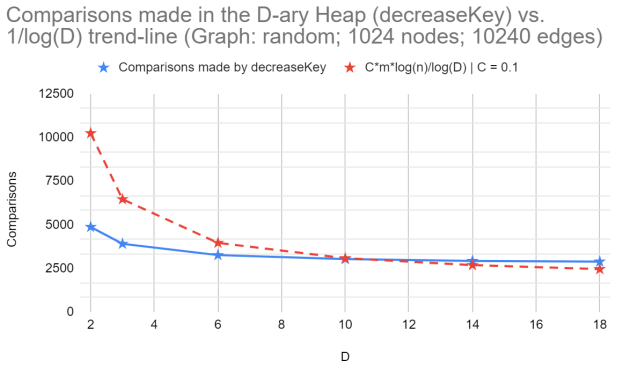

Figure 3.2: *Comparisons made by decreaseKey for a run of Dijkstra*

Here, we notice that the number of comparisons made by extractMin increases almost linearly along the trend-line D/log(D) while the number of comparisons made by decreaseKey reduces at logarithmic rate of 1/log(D). We notice that even though the m/n = 10 in this graph, the absolute minimum number of comparisons is found at D = 6. This behaviour can be attributed to the fact that the number of calls to decreaseKey is overestimated by m. In other words, extractMin is called C1*n times and decreaseKey is called C2*m times. However, C1 = 1 but C2 < 1. Therefore, we observe this discrepancy in real experiments. C2<1 since not all edge comparisons result in calling of a decreaseKey operation.

V. Conclusions and Future Work

In our Experimental Design, we made three hypotheses. Of these we could prove all three to be true. The conclusions are as follows:

1. Runtime -complexity:
   Experimental data for runs of FloydWarshall shows that both the number of comparisons and runtime show a strong correlation with the **n^3 - trend line.**
   Experimental data for runs of n-Dijkstra shows that both the number of comparisons and runtime show a strong correlation with the **n*(n+m*logn) - trend line.**
   This validates our hypothesis.
2. Varying Edge density in a graph (n-Dijkstra vs. FloydWarshall):
   The experimental data indicates that the runtime shows a correlation with the theoretical trend line and increases with increasing edges. While the floyd warshall's algorithm reports an almost constant runtime. Thus the hypothesis remains validated.
3. Dijkstra and D-ary Min Heaps:
   Experimental data shows that the total comparisons made by the extractMin utility of the heap should show a strong correlation with the **D*n*log(n)/log(D) -trendline.**
   Experimental data shows that the total comparisons made by the decreaseKey utility of the heap should show a strong correlation with the **m*log(n)/log(D) -trendline.**
   Thus the hypothesis is validated to be true.

Shortest Path finding is a vast field with many complex algorithms with various constraints. From the work that we did, we feel that the following could be good places to explore further:

1. Alternate Implementations of dijkstra's

Two alternate versions of Dijkstra (with PriorityQueue) can be implemented - One where all nodes are inserted into the PQ before-hand (with distance INFINITY); Other where we only insert nodes into the only when they are discovered. Since the second implementation has, on an average, a lesser number of nodes in the heap we expect that the second implementation will result in a significantly less number of heap-comparisons.

2. Experimenting with more unique graph structures

Most real world applications of shortest finding involve specific graphs types. Or graph types with unique structures. For example road networks in countries are usually graphs with a few very densely connected nodes representing metropolitan cities while most being loosely or sparsely connected representing smaller cities. It would be interesting to note the performance of these algorithms on such graphs.