# The Floyd-Warshall and Dijkstra’s Algorithm Analyzed

The purpose of this program was to test the Floyd-Warshall and Dijkstra’s algorithms. Both algorithms were tasked with finding the shortest path from each node to all others in a complete, weighted, and directed graph. While setting up this experiment, a few assumptions were made:

* All edges have a weight between 1 and 10.
* There are no negative weights involved in any of the graphs.
* A weight of 0 is only possible when describing the distance from a node to itself.
* All nodes will have a path to every other node. This implies that for an N\*N matrix, there are N\*N connections.
* The distance from any node S🡪T, where S != T was not necessarily the same as the distance from T🡪S.

## Problem Description

The purpose of this experiment was to track and quantify the running times of both the Floyd Warshall and Dijkstra’s Algorithm, and develop a greater understanding of how they operate. While the amount of nodes in the matrix continues to increase, the running time difference between the two algorithms would hypothetically be negligible. This experiment will explore this theory and see if the theoretical advantages of the two algorithms hold true in a practical setting.

## Dijkstra’s Algorithm

Dijkstra’s algorithm functions by selecting the best solution local to each node in order to find the overall optimal solution. Its tendency to focus on immediate solutions for an individual source node at any given time allows the algorithm to act in a “greedy” manner. The algorithm itself only has a running time of O(n2), however since the algorithm only works with a single node at any given time, it must still cycle through all of the available nodes for the graph to be completed. This means that the overall run time for Dijkstra’s algorithm is O(n2 \* n) or simply O(n3).

The program for Dijkstra’s algorithm is based off code written by myself for the CSCE 2110 class, and borrows large sections from the previous assignments.

The program containing Dijkstra’s algorithm starts by reading in graph.txt created by our data creation program, and this data gives the program the size of the matrix as well as all of the data points making up the graph. This is handled using fstream commands.

New arrays were created to meet the needs of the program, including “finishedgraph[size][size]”, “distance[size]”, and “completed[size].”

As Dijkstra’s algorithm only works on a single node at a time, these arrays were needed to keep track of where the program was currently operating and what the next node would be. The for loop surrounding the majority of the function accounts for the other nodes being operated on, and allows for the results of each iteration to be stored in the final graph. The first iteration stores the results for the first node, while the second iteration represents the shortest paths to all other nodes from the second noted.

The “completed” array allows us to store a Boolean value for each node stating if it has been processed. Initially, all distances are set to infinity and all nodes are marked as false in the Boolean function to establish the fact that we haven’t visited them. This is done through the following code:

for (i = 0; i < size; i++)

{

distance[i] = INT\_MAX;

completed[i] = false;

}

distance[counter] = 0;

”Distance” stores the actual distances between the nodes. This array is eventually copied into the “finishedgraph” array with the final data points.

The actual processing of the algorithm all takes place within the while loop. The node is tested to see if it has been visited previously, and if not, it is again tested to see if the distance involved is shorter than the current shortest distance. This is accomplished with the following lines:

for (i = 0; i < size; i++)

{

if (!completed[i] && distance[i] < bestdistance)

{

a = i;

bestdistance = distance[i];

}

}

if (bestdistance == INT\_MAX)

{

break;

}

Following these procedures, a second for loop begins comparing the distance of the current node and the recorded distance between the stored node and the current node added with the stored node. Should the distance be larger, the distance is updated with that value.

for (b = 0; b < size; b++)

if (!completed[b] && data[a][b] != INF)

if (distance[b] > distance[a] + data[a][b])

distance[b] = distance[a] + data[a][b];

The node is then marked as completed and the loop continues until the break condition is met, stating that we’ve visited every node. Lastly, the data points are stored into the “finishedgraph” array.

## Floyd-Warshall Algorithm

Floyd-Warshall is considered a dynamic algorithm capable of computing the minimum cost of a weighted and directed matrix in O(n3) time. This algorithm can handle both negative and positive edge weights, however it cannot manage negative cycles.

The algorithm itself is fairly simple both in design and in practice, and is capable of comparing all possible paths through the graph between each pair of vertices until the optimal path is derived. Once the program has been passed the generated graph, it pulls out the key data identical to the way Dijkstra’s algorithm does. Once the data is parsed into a new matrix that can be edited without destroying the original, the algorithm begins to parse through it.

Three nested for loops allow each node to be addressed and compared in the matrix.

for (a = 0; a < size; a++)

for (b = 0; b < size; b++)

for (c = 0; c < size; c++)

if (distance[b][a] + distance[a][c] < distance[b][c])//

{

distance[b][c] = distance[b][a] + distance[a][c];

}

For every a cycled through the size of the graph, the entire matrix is addressed and updated if a shorter distance is discovered. Distance­ba + distanceac is compared against the distancebc, attempting to establish if there is a shorter path from b 🡪c through node a.

## Measuring Runtime

In order to accurately measure the runtime of the algorithms, the chrono library was used and two high\_resolution\_clock::timepoint variables were created per algorithm. The first of these variables is created after any arrays needed to be created, but immediately before any work was done on the actual algorithm. The second variable is created immediately after the final steps of either algorithm are completed. To get the run time of the algorithm, the first variable (time1) is subtracted from the second (time2). This data is altered to be read in nanoseconds in order to make it more viable for extracting data from the runs.

## Initial Data Generation

The initial data generation is handled by datacreation.cpp. The program initially asks you to enter the size of your matrix. You’re then asked if you would like all of the data points to be the same value or to have them randomly generated. You are also given the option to quit as a failsafe before generating the graph. For random numbers, srand and time.h are used to seed the random number generator, and numbers between one and ten are chosen. For both options, any node will have a distance of zero to itself.

The data is then stored via fstream in a way that’s easily accessible when passed to the other programs. The size is stored on top followed by a newline, and following that, the entire matrix is printed out. This standardization allows us to easily call this data into the other programs without worrying about where to look for key data points.

## Experiment Design

The experiment was conducted by running each program independently on graphs ranging from 50 nodes to 1000 and tracking their runtime. Each test was preformed three times per graph size and data type in order to minimize the chance out outliers and to provide more accurate data on the whole. The resulting averages were then graphed to aid visual assessment of the algorithms.

### All Nodes Filled With the Same Initial Value

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dijkstra's | Run 1 in Nanoseconds | Run 2 | Run 3 | Average Time in Nanoseconds |
| 50\*50 | 185399 | 223799 | 21700 | 208966 |
| 100\*100 | 977599 | 746400 | 907600 | 877199.7 |
| 150\*150 | 2443199 | 2445900 | 2577899 | 2488999 |
| 200\*200 | 6080499 | 615580 | 572800 | 5988700 |
| 250\*250 | 1114799 | 11036799 | 11105700 | 11085766 |
| 300\*300 | 19061400 | 19244500 | 19096099 | 19134000 |
| 400\*400 | 45073499 | 44751699 | 45133299 | 44986166 |
| 500\*500 | 87423399 | 87282800 | 87143499 | 87283233 |
| 600\*600 | 151109200 | 151040100 | 150507799 | 150885700 |
| 700\*700 | 237515600 | 239241999 | 239699900 | 238819166.3 |
| 800\*800 | 354826499 | 355444199 | 358562599 | 356277765.7 |
| 900\*900 | 45752439 | 505464599 | 462857499 | 475282166 |
| 1000\*10000 | 691592599 | 692104500 | 691848550 | 691848550 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Floyd-Warshall | Run 1 in Nanoseconds | Run 2 | Run 3 | Average Time in Nanoseconds |
| 50\*50 | 51800 | 59100 | 51599 | 54166.33 |
| 100\*100 | 393099 | 541800 | 354200 | 429699.7 |
| 150\*150 | 1307499 | 1180900 | 1446099 | 1311499 |
| 200\*200 | 3089199 | 3203099 | 3130200 | 3140833 |
| 250\*250 | 6068699 | 6013700 | 6074199 | 6052199 |
| 300\*300 | 10370799 | 10419399 | 9409700 | 100666633 |
| 400\*400 | 2486299 | 24576999 | 24611199 | 24668166 |
| 500\*500 | 48145500 | 47708000 | 48152600 | 48002033 |
| 600\*600 | 83316900 | 82887900 | 82948399 | 83051066 |
| 700\*700 | 118427400 | 131902499 | 131658199 | 127329366 |
| 800\*800 | 178985100 | 197737000 | 197395199 | 191372433 |
| 900\*900 | 254251599 | 281216199 | 273074700 | 269514166 |
| 1000\*10000 | 355046900 | 383044500 | 246030800 | 328040733 |

### All Nodes Filled With Random Initial Value

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dijkstra's | Run 1 in Nanoseconds | Run 2 | Run 3 | Average Time in Nanoseconds |
| 50\*50 | 147299 | 147299 | 147700 | 147432.7 |
| 100\*100 | 919599 | 1037700 | 1092699 | 1016666 |
| 150\*150 | 2883499 | 2984200 | 2861299 | 2909666 |
| 200\*200 | 6777700 | 6470299 | 6520299 | 6589433 |
| 250\*250 | 12409499 | 12572600 | 12437900 | 12473333 |
| 300\*300 | 20965400 | 21221999 | 21335299 | 21174233 |
| 400\*400 | 49055999 | 49097099 | 49116399 | 49089832 |
| 500\*500 | 94940599 | 94962100 | 95767599 | 95223433 |
| 600\*600 | 163869400 | 147118299 | 163755699 | 161581133 |
| 700\*700 | 260328200 | 260920299 | 261581999 | 260943499.3 |
| 800\*800 | 355633800 | 261581999 | 395383500 | 337533099.7 |
| 900\*900 | 558746399 | 395383500 | 557382199 | 503837366 |
| 1000\*1000 | 691592599 | 769548900 | 768960299 | 743367266 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Floyd-Warshall | Run 1 in Nanoseconds | Run 2 | Run 3 | Average Time in Nanoseconds |
| 50\*50 | 52700 | 58799 | 58600 | 56699.67 |
| 100\*100 | 430399 | 472399 | 479200 | 460666 |
| 150\*150 | 1403700 | 1420000 | 1467000 | 1430233 |
| 200\*200 | 3241000 | 3319899 | 3317900 | 3292933 |
| 250\*250 | 6357999 | 6294099 | 6399299 | 6350466 |
| 300\*300 | 10727200 | 10896599 | 11139899 | 10921233 |
| 400\*400 | 25890600 | 25258500 | 25513500 | 25554200 |
| 500\*500 | 47465899 | 49489800 | 4911499 | 48690233 |
| 600\*600 | 83945199 | 85356900 | 84731199 | 84677766 |
| 700\*700 | 120425899 | 133684800 | 134528500 | 129546399.7 |
| 800\*800 | 180710400 | 133684800 | 200629200 | 171674800 |
| 900\*900 | 282352900 | 200629200 | 261880999 | 248287700 |
| 1000\*10000 | 384840299 | 391066799 | 351066499 | 375657866 |

Analyzing this data and the resulting graphs, it becomes obvious that Dijkstra’s algorithm quickly becomes more demanding of resources when compared to Floyd-Warshall algorithms. With this in mind, I believe it’s safe to make the following assumptions regarding the two algorithms based on the results:

* Having the array filled with identical values for each node speeds up the algorithms. This is likely due to the fact that there are less calculations required if the data points all are equal, or presorted to some degree.
* Dijkstra’s algorithm requires an additional cycle from 0 to size to establish the bestdistance while dealing with the Boolean array. This adds to the overhead of the program and will only result in longer times
* Dijkstra’s algorithm also isn’t implemented to handle a min-priority queue.
* Since the graphs used in the experiment are complete graphs, Floyd-Warshall gains a distinct advantage since its better equipped to handle addressing a large volume of nodes that each require inspection.

## Conclusion

While either of the algorithms discussed are capable of computing the minimum cost matrix, there is a fairly obvious performance difference between the two. When the matrix is 1000\*1000 nodes filled with random initial values, the Floyd-Warshall algorithm is able complete its task roughly 50.53% of the time based on the averages found in testing. This margin is extremely likely to continue growing as the size of the arrays continues as well. Combined with the facts that Floyd-Warshall is easier to implement on the whole, it’s safe to state that the Floyd-Warshall algorithm should be considered before Dijkstra’s algorithm in most scenarios.