# Evaluation of Dijkstra's Algorithm

(Jack Hallissey)

This report contains the results of the evaluation of Dijkstra's algorithm and discussion of those results.

The code used consists entirely of my own code and code provided on Canvas for CS2515/CS2516.

The following Python files are submitted and are required to run the evaluations:

- apq.py: Contains the binary heap-based and unsorted list-based implementations of the Adaptable Priority Queue.
- graph.py: Contains the implementation of the Graph ADT and the implementations of Dijkstra's algorithm.
- evaluation.py: Contains the code to run the evaluations.

The following file is not needed to run the evaluations, but is included for completeness:

- stats.py: Computes various statistics relating to the grid graphs at different sizes, which are referenced in the report.

The evaluations can be run by running evaluation.py. Specific parts can be run by changing the function calls at the end of the file.

The output of running the code for questions 1 and 2 are included below. Results and discussion for questions 3-6 can be found thereafter.

## Question 1

14 to 5: 16
14 to 8: 13
14 to 6: 10
14 to 3: 6
14 to 11: 4
14 to 14: 0

## Question 2

Edge weights
(0, 0) to (1, 0): 1
(0, 0) to (0, 1): 1
(0, 1) to (1, 1): 1
(0, 1) to (0, 2): 1
(0, 2) to (1, 2): 2
(0, 2) to (0, 3): 1
(0, 3) to (1, 3): 1
(1, 0) to (2, 0): 2
(1, 0) to (1, 1): 2
(1, 1) to (2, 1): 1
(1, 1) to (1, 2): 2
(1, 2) to (2, 2): 1
(1, 2) to (1, 3): 1
(1, 3) to (2, 3): 2
(2, 0) to (3, 0): 2
(2, 0) to (2, 1): 1
(2, 1) to (3, 1): 2
(2, 1) to (2, 2): 2
(2, 2) to (3, 2): 2
(2, 2) to (2, 3): 1
(2, 3) to (3, 3): 1
(3, 0) to (3, 1): 2
(3, 1) to (3, 2): 1
(3, 2) to (3, 3): 1

Shortest path
(0, 0) to (3, 3): 7
(0, 0) to (3, 2): 6
(0, 0) to (3, 1): 5
(0, 0) to (2, 1): 3
(0, 0) to (1, 1): 2
(0, 0) to (0, 1): 1
(0, 0) to (0, 0): 0

## Question 3

| Graph Size | Average Path Cost | Average Runtime (seconds) |
|---|---|---|
| 10x10 | 18 | 0.000 |
| 50x50 | 345 | 0.014 |
| 100x100 | 1270 | 0.063 |
| 250x250 | 7552 | 0.503 |
| 500x500 | 29486 | 3.694 |
| 750x750 | 65711 | 10.831 |
| 1000x1000 | 116290 | 21.200 |

The runtime of the algorithm was evaluated on random graphs of increasing size. For each graph size, a random graph was generated and the shortest path was computed from [n//2][m//2] to [0][0]. This process was repeated 10 times for each size, and the average path cost and runtime were reported.

These tests use the original implementation of the algorithm, which uses a binary heap-based APQ and computes the shortest path from the source to all other nodes in the graph.

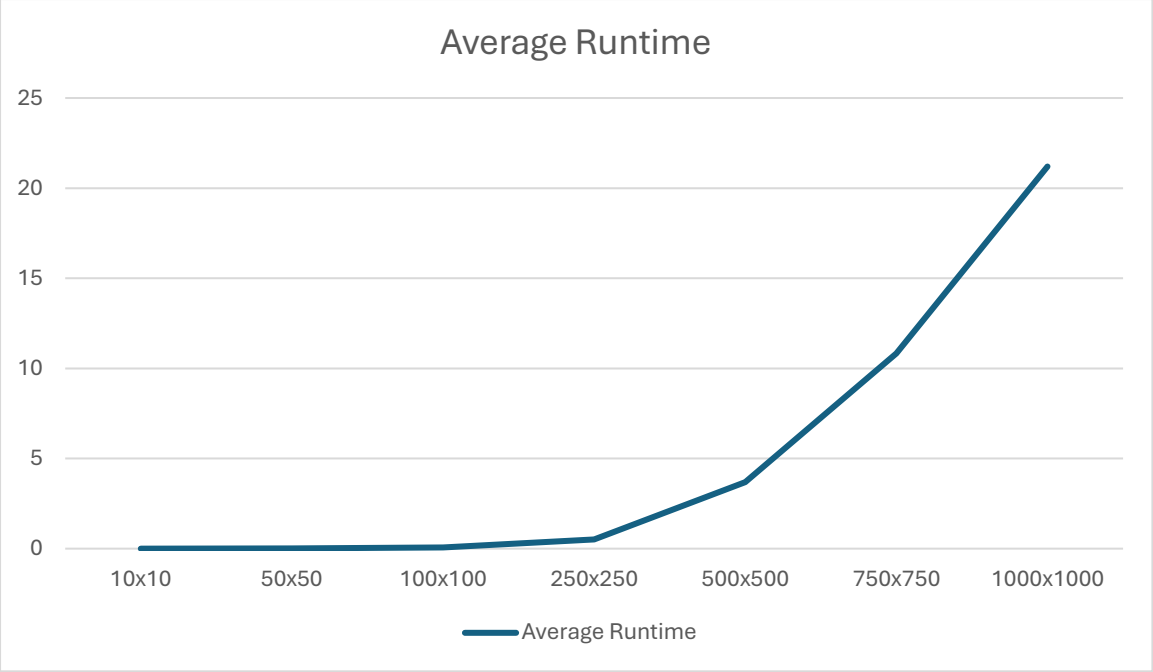
The worst case time complexity of Dijkstra's algorithm implemented using a heap APQ is O((n+m)log(n)), where *n* is the number of vertices and *m* is the number of edges.

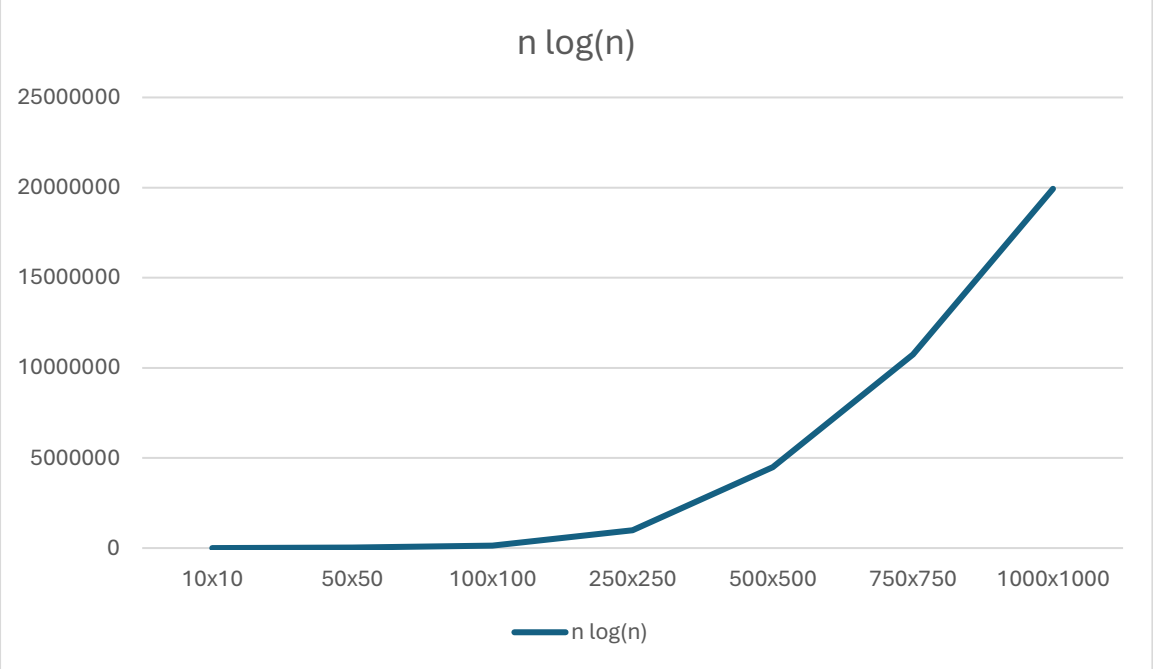
The grid graphs used in these evaluations are very sparse, as the number of edges is relatively low and each vertex has a small degree, as shown in the table below. On sparse graphs such as these, the worst case time complexity of the algorithm is O(n log(n)), as *m* is bound by some constant multiple of *n*; in this case, each graph has less than *2n* edges.

We can examine whether the runtime patterns are in line with this complexity by computing n log(n) for each graph size and plotting charts of both this expression and the average runtime of the algorithm for each graph size.

| Graph Size | Vertices | Edges | Average Degree | n*log(n) | Average Runtime |
|---|---|---|---|---|---|
| 10x10 | 100 | 180 | 3.600 | 664.386 | 0.000 |
| 50x50 | 2500 | 4900 | 3.920 | 28219.281 | 0.014 |
| 100x100 | 10000 | 19800 | 3.960 | 132877.124 | 0.063 |
| 250x250 | 62500 | 124500 | 3.984 | 995723.036 | 0.503 |
| 500x500 | 250000 | 499000 | 3.992 | 4482892.142 | 3.694 |
| 750x750 | 562500 | 1123500 | 3.995 | 10744590.134 | 10.831 |
| 1000x1000 | 1000000 | 1998000 | 3.996 | 19931568.569 | 21.200 |

The figures in this table, besides the runtimes, were computed by running stats.py.




The results suggest that, once *n* is high enough, the runtime is bound by some constant multiple of n log(n), i.e. that the worst case time complexity of the algorithm on these graphs is indeed O(n log(n)).

## Question 4

| Destination | Average Path Cost | Average Runtime (All Nodes) | Average Runtime (Specific Destination) |
|---|---|---|---|
| [275][275] | 3218 | 3.092 | 0.031 |
| [300][300] | 6163 | 3.294 | 0.198 |
| [350][350] | 12189 | 3.554 | 0.756 |
| [400][400] | 18098 | 3.306 | 1.839 |
| [450][450] | 23517 | 3.238 | 3.084 |
| [475][475] | 26472 | 3.264 | 3.292 |
| [499][499] | 29225 | 2.992 | 3.341 |

Two versions of the algorithm are compared:

- the original (all nodes) version, which computes the shortest path from the source to all other nodes in the graph, and
- the specific destination version, which breaks out of the loop and returns the closed dictionary if the node removed from the APQ is the destination.

Graphs of size 500x500 are used, and the node [250][250] is used as the source. The two versions of the algorithm are run repeatedly with destination nodes that are steadily further away from the source. For each destination, 10 random graphs are generated and both versions of the algorithm are run on each graph. The average path cost and the average runtime for each version are reported.

The all nodes version shows little variation in runtime as the destination gets further away from the source. The runtime of the specific destination version increases as the destination grows further away. It is far faster than the all nodes version for close destinations, but the benefit diminishes as the distance from the source to the destination increases.

This is to be expected, as the all nodes version computes the shortest path from [250][250] to all other nodes every time, regardless of the given destination. The specific destination version can stop once it has removed the destination from the APQ, meaning it does not have to traverse as much of the graph. When the destination is closer to the source, it will be removed by the algorithm sooner, and the time saved by not having to traverse the rest of the graph will be greater.

The all nodes version is in fact slightly faster on average than the specific destination version when the nodes [475][475] and [499][499] are used as the destinations, indicating that there is a cross over point between the two versions when the destination node is somewhere between [450][450] and [475][475]. This is likely because, in these cases, the destination is quite far away and, by the time it is removed, the algorithm has likely traversed almost all of the paths that it would traverse in the all nodes version, meaning the time saved by ignoring the remaining paths is very small or negligible.

It is also possible that the specific destination version is slowed down slightly by the small overhead of comparing the removed node with the destination being repeated many times, and in this case, it may outweigh the small benefit of stopping once the destination is removed.

## Question 5

| Graph Size | Average Path Cost | Average Runtime (Heap APQ) | Average Runtime (Unsorted List APQ) |
|---|---|---|---|
| 10x10 | 21 | 0.000 | 0.001 |
| 50x50 | 363 | 0.019 | 0.046 |
| 100x100 | 1279 | 0.104 | 0.403 |
| 250x250 | 7505 | 0.878 | 6.811 |
| 500x500 | 29773 | 3.802 | 54.387 |

This test compares the heap-based implementation of the algorithm, which uses an APQ implemented using a binary heap, with the list-based implementation, which uses an APQ based on an unsorted Python list. Both versions compute the shortest path from the source to all other nodes in the graph.

The runtimes of the two implementations were evaluated on random graphs of increasing size. For each graph size, 10 random graphs are generated and both implementations of the algorithm compute the shortest path from [n//2][m//2] to [0][0] in each graph. The average path cost and the average runtime for each implementation are reported.

The results show that the heap-based implementation is consistently faster than the list-based implementation. The difference between their runtimes is relatively insignificant for small graphs, but grows rapidly as the size of the graph increases.

This is because the grid graphs used in this evaluation are very sparse. Even for large grid graphs, the number of edges is relatively small, and the vertices have a very low degree.

| Graph Size | Vertices | Edges | Average Degree |
|---|---|---|---|
| 10x10 | 100 | 180 | 3.600 |
| 50x50 | 2500 | 4900 | 3.920 |
| 100x100 | 10000 | 19800 | 3.960 |
| 250x250 | 62500 | 124500 | 3.984 |
| 500x500 | 250000 | 499000 | 3.992 |

The figures in this table were computed by running stats.py.

In dense graphs, the number of neighbouring vertices to be added to the PQ when expanding from a vertex is high. A PQ implemented using an unsorted list is well suited to this, as it allows for addition in constant time, rather than the logarithmic time needed to add to a heap-based PQ. This does not apply in sparse graphs, as each vertex has a small number of neighbours, and the overhead of a logarithmic time PQ addition for each of the neighbours is not very significant.

The list-based implementation is much slower on these graphs because it requires a linear search to find and remove the vertex with the lowest cost. When the size of the PQ is nearly 500, this is a significant overhead. By contrast, the heap-based implementation of the PQ can find and remove the cheapest vertex much faster, in logarithmic time.

## Question 6

| Graph Size | Average Path Cost | Average Runtime (Original Version) | Average Runtime (Simpler PQ Version) |
|---|---|---|---|
| 10x10 | 19 | 0.000 | 0.001 |
| 50x50 | 345 | 0.014 | 0.023 |
| 100x100 | 1282 | 0.065 | 0.110 |
| 250x250 | 7557 | 0.652 | 1.091 |
| 500x500 | 29437 | 3.616 | 6.477 |
| 750x750 | 65793 | 9.393 | 16.120 |
| 1000x1000 | 115883 | 17.542 | 29.886 |

This test compares the original version of the algorithm, which allows for at most one entry for each vertex in the open PQ, and a version that uses a simpler PQ, which can contain multiple entries for each vertex. Both versions uses a binary heap-based APQ and compute the shortest path from the source to all other vertices in the graph.

The runtimes of the two versions of the algorithm were evaluated on random graphs of increasing size in the same manner as in questions 3 and 5. For each graph size, 10 random graphs are generated and both versions compute the shortest path from [n//2][m//2] to [0][0] in each graph. The average path cost and the average runtime for each version are reported.

The results show that the original version is consistently faster than the simpler PQ version. This is likely because the simpler PQ version can add several entries to the PQ for a single vertex. The algorithm adds every vertex it reaches to the PQ and continues running until all entries in the PQ have been removed. The number of entries that will be added to and removed from the PQ as the algorithm runs is likely to be significantly higher in the simpler PQ version than in the original version.

Using the heap-based PQ, each addition and removal operation is not constant time, so if the number of duplicate entries is high, there will be a considerable overhead.