

COMPUTING ALGORITHMS 2801ICT

Assignment 3 – K Shortest Paths



May 26, 2019

Sam grumley

S5048240

# Algorithm Design

## Overview

n this problem, you are given 2 integers (N, M), N is the number of vertices, M is the number of edges. You will also be given , , where and represents an edge from a vertex to a vertex and represents the weight of that edge. Finally, you will be given two vertices S and D, where S denotes the source vertex (Southport) and D denotes the destination vertex (Brisbane CBD) and the value of K.

## Algorithm Description

Key points to expand how the algorithm works:

### Queue lookup:

Using sets greatly increases the time taken to verify if the node is already in the queue

### Using the first search to find the other searches

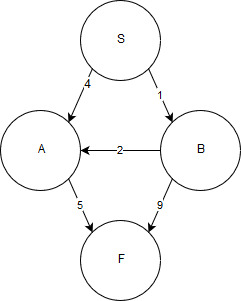
Since Dijkstra finds the most optimal path. We can work with the fact that the first solution is optimal. To avoid recalculating data we can return the list of explored nodes along with the cost, optimal path and a list of changed costs for each node. Using the explored list, we can make permutations to the optimal path.

### Check if the search was close to finding a different route:

To get the best information from the graph possible the read in function was altered to read in each line and store it in a list before any processing. This gives the opportunity to see what the start and goal nodes are before the edges and nodes are transformed into a dictionary. Any node that connects to the goal node is recorded in a list. Pairing the list with the explored list returned from Dijkstra’s search we can then check if any nodes close to the goal node were explored. This can allow for multiple paths to be found without running Dijkstra’s again.

### Using the recorded data to permute the original path:

Since we know the first path is optimal and that the other paths do not need to be optimal we can assume that there are very similar paths that are valid K solutions. This requires at least some of the nodes in the optimal path to have been updated in Dijkstra’s search.



In the graph above we would have an optimal path of S->B->A->F. From noting that node A will change from (A, 4, S) to (A, 3, B) we can simply revert that change in the explored list and run from the goal node creating a new list that would look like S->A->F. when looking back through the list A would be updated to say that it came from S instead of B. Given a graph that has many updates you can produce many permutations. To best carry out this process the list of changes is sorted by the difference in the update of cost (e.g. 4-3 in the graph above). This allows the algorithm to check the lowest change costs which will in turn give us the shortest cost paths. S->B->A->F would have a cost of 8 therefore, S->A->F would be the cost of S->B->A->F + the difference in cost update (8+1)

### Securing K paths in unique graphs:

Since the above two solutions have requirements on how Dijkstra will search it is possible in some cases that these requirements will not be met a result in less than K paths. To guarantee that there will be K paths a variation of Yen’s algorithm will run at the end to make up for missing paths. The variation will come by the nature of only iterating through the solution once and taking the top k - len(k[]) solutions. This is by far the slowest method of results but will typically only be required on small graphs.

Like Yens algorithm we can find alternative sub graphs as we remove connections in the original graph. Unlike Yen’s algorithm that is aimed at optimality we can speed up the process significantly by removing nodes from the goal node rather than the first node. This is somewhat of a heuristic considering most optimal paths will alter towards the end since we have found the most optimal path through most of the graph.

## Pseudo Code

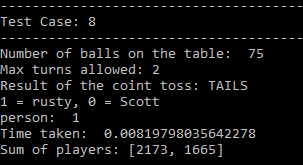
# Results and Analysis

## Results

Results of all tests (average time over 10 runs):

|  |  |  |
| --- | --- | --- |
| Input | Average time | Correct Solution |
| 1 | 2.88E-05 | Y |
| 2 | 1.18E-04 | Y |
| 3 | 4.78E-05 | Y |
| 4 | 0.000731 | Y |
| 5 | 2.48E-03 | Y |
| 6 | 3.61E-04 | Y |
| 7 | 7.24E-04 | Y |
| 8 | 6.38E-03 | Y |
| 9 | 2.69E-04 | Y |
| 10 | 1.89E-03 | Y |

Example output of a testcase:



## Performance Analysis

### Upheap from leaves vs Downheap every node

Two algorithms were considered for this task. Upheaping from the leaf nodes and downheaping from everything except for the leaf nodes.

|  |  |  |  |
| --- | --- | --- | --- |
| Downheap | Upheap | Difference in time | Difference of output(upheap) |
| 2.88E-05 | 2.63E-05 | -2.51E-06 | 0 |
| 1.18E-04 | 0.000111 | -6.38E-06 | 0 |
| 4.78E-05 | 4.93E-05 | 1.42E-06 | 0 |
| 0.000731 | 0.000732 | 7.39E-07 | 0 |
| 2.48E-03 | 4.05E-03 | 1.57E-03 | 133, -133 |
| 3.61E-04 | 0.000343 | -1.76E-05 | 0 |
| 7.24E-04 | 0.00069 | -3.35E-05 | 47,-47 |
| 6.38E-03 | 0.006395 | 1.67E-05 | 0 |
| 2.69E-04 | 0.000273 | 4.76E-06 | 0 |
| 1.89E-03 | 0.001914 | 2.68E-05 | 0 |

## Complexity

### Time:

While (values are still on the table) = n

Heapify(n/2)\*log n

For (max turns) = k

checkMax(n+n+(n\*log n))

checkMax( = n + n + (n\*insert into heapQueue) since we are only looking for the most significant function we can focus on the insert loop (n log n)

Worst Case Rusty has every turn resulting in check max called every iteration, where all values have the same summed value. This situation would lead to adding every element back in to the priority queue except for 1 resulting in:

### Space:

Space complexity will occur at the creation of the priority queue which takes place in the dataStructure function. sumVal will \* n while the insert function will b n\*2 values (sumval and other val) making it:

2n

### Improvements:

Since this is the function checkMax() function is the cause of the time complexity, any improvements should be focused at this function. A suggestion would be to check how many turns Rusty has within that round and return the number of choices to match while also returning how many turns to skip. Although it may not reduce the worst time it would increase the average run time significantly.