K-Shortest Path Problem

Lucas Geurtjens | s5132841 | 26/05/2019

Table of Contents

[Problem Overview 2](#_Toc9795780)

[Algorithm Design 2](#_Toc9795781)

[Algorithm Description 2](#_Toc9795782)

[Dijkstra’s Algorithm 2](#_Toc9795783)

[Alternate K – 1 Paths Algorithm 2](#_Toc9795784)

[Innovation 2](#_Toc9795785)

[Algorithm Pseudo-Code 3](#_Toc9795786)

[Dijkstra’s Algorithm 3](#_Toc9795787)

[Alternate K – 1 Paths Algorithm 3](#_Toc9795788)

[Results and Algorithm Analysis 4](#_Toc9795789)

[Testcase Results 4](#_Toc9795790)

[Time Complexity 4](#_Toc9795791)

[References 5](#_Toc9795792)

[Appendix 5](#_Toc9795793)

# Problem Overview

The developed algorithm aims to solve the k-shortest path problem. The k-shortest path problem involves finding the k paths shortest paths to travel from a chosen starting node to a goal node within a directed graph. In the developed algorithm, the list of k shortest paths included 1 optimal route found using Dijkstra’s shortest path algorithm and k – 1 approximate routes, found using an altered version of Yen’s algorithm[2]. From this, for a given input set of connected vertices and their edge weights, the algorithm is able to return k number of shortest distance paths.

# Algorithm Design

## Algorithm Description

### Dijkstra’s Algorithm

Firstly, the program utilised an implementation of Dijkstra’s algorithm[1] to find the optimally shortest distance from the start to goal nodes. Here, the assumption of non-negative edge weights was made, to avoid issues with the use of this model. The algorithm attempts to loop over nodes of the graph until it has reached its goal node. For each node that is inspected, the distance cost to get to that node is tracked. From this, we look at the nodes connected edges. Here, the node with the lowest distance cost is chosen, and relaxation is applied if the distance is superseded. After this, the node with the minimum distance out of all unexplored nodes is chosen to be used as the next inspected node. The cycle is repeated until the shortest path to the goal node is found. From this, the algorithm calculates the path that was taken to find the goal node by tracing backwards from each edge pair. After this, the algorithm is then able to return both the optimal cost and path to the goal node.

### Alternate K – 1 Paths Algorithm

In addition, the program utilized an algorithm, as inspired by Yen’s algorithm[2], for calculating the k-1 approximate shortest paths. Here, the algorithm employs the optimal shortest path as previously calculated by Dijkstra’s algorithm. From this, the algorithm iterates though each of the optimal node edges found in the path and, one-by one towards the goal, removes them by setting their edge cost to a value of infinity (such that it won’t be selected). This is calculated k amount of times and is sorted in ascending order when complete. If there is overflow and not enough optimal edges exist, the algorithm pads the results with the last element in the list (which will be the largest distance).

## Innovation

Innovation was made on the referenced, Yen’s algorithm, to create a list of shortest paths. Firstly, the concepts of applying Dijkstra’s algorithm to find the first shortest path and removing certain edges when finding the approximate shortest paths was retained. The main differences can be observed in how the algorithm finds the k – 1 approximate shortest paths. Here, the algorithm, as mentioned in the ‘Alternate K – 1 Paths’ algorithm description, finds its paths by setting the edge weights of optimally found nodes to an infinite value (also done in Yen’s algorithm), therefore skipping that node and requiring the algorithm to find a different path. This is done one-by-one towards the goal node, and only ever alters one optimal edge at a given time. This was thought to increase runtime of the algorithm by reducing the number of unique paths that can be taken as the algorithm only alters route nodes that were used in the optimal path. Likewise, with only 1 edge changed, it was thought the algorithm would have a higher chance to find a way to resume the already discovered optimal path, hence, expecting to create lower variance of non-optimal node edges used and thus lower distance cost.

## Algorithm Pseudo-Code

### Dijkstra’s Algorithm

# Dijkstra's Shortest Path  
# Input: A graph containing edges and weights, a starting node, a goal node  
# Output: Optimal shortest path tour and the distance cost of reaching the goal node  
  
function dijkstra(graph, start, goal)  
distance\_costs <- blank dictionary initialised with the starting node  
current\_node <- start  
explored <- blank set  
  
while current\_node is not goal  
 explored <- append current\_node  
 children\_paths <- append current node's edges  
 current\_weight <- distance cost of current node  
  
 for child\_node in children\_paths  
 new\_weight = current\_node.weight + child\_node.weight  
  
 if child\_node is not in distance\_costs dictionary  
 distance\_costs <- append the child node edge  
  
 else  
 minimum\_weight <- distance cost of current child  
  
 if new\_weight is less than minimum\_weight  
 distance\_cost <- relax cost of the child node to new found weight  
  
 next\_nodes\_to\_check <- dictionary filled with nodes that have not been explored.  
  
 if next\_nodes\_to\_check is empty  
 return  
  
 else  
 current\_node <- minimum distance cost node in the next\_nodes\_to\_check dictionary  
  
 optimal\_path <- blank list  
 optimal\_cost <- distance cost of goal  
  
 while current\_node is not the start node parent  
 optimal\_path <- append current\_node  
 current\_node <- child\_node of current\_node  
  
 reverse(optimal\_path) # Assumes a function for reversing the list  
  
 return optimal\_path, optimal\_cost # Assumes can return multiple items (like in Python)

### Alternate K – 1 Paths Algorithm

# Alternate K - 1 Shortest Path's  
# Input: Shortest path from the Dijkstra algorithm, the node graph and k value  
# Output: K no. of approximate shortest distance costs to goal node  
  
function alternative\_shortest\_paths(optimal\_path, graph, k)  
  
k\_shortest\_paths <- blank list  
optimal\_nodes\_used <- int size of elements in optimal path  
  
for node in optimal\_nodes\_used - 1  
 if k == 0  
 break  
  
 else  
 node\_pair <- get first pair of nodes from optimal path e.g. node i and node i++  
 graph\_copy <- copy of the original graph  
 graph\_copy.weights.node\_pair <- infinity or very large integer  
 k\_path <- dijkstra(graph\_copy, optimal\_path's start, optimal\_path's goal)  
 k--  
  
 sort(k\_shortest\_paths) # Assuming a function for sorting the list in ascending order  
  
 if k not 0  
 while k is not 0  
 k\_shortest\_paths <- append the last element in k\_shortest\_paths for padding  
 k--  
  
 return k\_shortest\_paths

# Results and Algorithm Analysis

## Testcase Results

|  |  |  |
| --- | --- | --- |
| Input (k) | Output | Time Taken |
| 2 | 1038.57, 1043.028 | 22.69 |
| 3 | 1038.57, 1043.028, 1043.028 | 42.45 |
| 5 | 1038.57, 1043.028, 1043.028, 1043.028, 1043.028 | 55.72 |
| 7 | 1038.57, 1043.028, 1043.028, 1043.028, 1043.028, 1043.028, 1043.028 | 79.11 |
| 10 | 1038.57, 1043.028, 1043.028, 1043.028, 1043.028, 1043.028, 1043.028, 1043.028, 1043.028, 1043.028 | 113.37 |

Table 1: Testcase Results with varying k values

As seen in *Table 1*, the algorithm was able to find shortest path results the input as seen in *Appendix Figure B*. Firstly, as compared to known results as seen in *Appendix Figure C*, it appeared the algorithm correctly found the optimal path of distance 1038.57. Alternatively, it appeared that each k – 1 approximate path distances was equal. This indicates either an unexpected error in the code or the possibility that because the data set is so large, there are equal alternative edges which bring the distance to the same value. It was found that in a smaller data set, as seen in *Appendix Figure A*, the algorithm did not repeat approximate distances and behaved correctly. Finally, it was observed that the time taken to solve for the output was fairly substantial as compared to the known results speeds. This indicates a less efficient implementation.

## Time Complexity

In order to find the time complexity of the algorithm, the algorithms various declarations and functions were inspected. Firstly, there was a Dijkstra algorithm implementation that followed a worst-case time complexity of O(E log N)[3] where E = number of edges and N = number of nodes. Furthermore, there was an algorithm for finding the approximate k – 1 shortest paths which called the Dijkstra’s algorithm a worst case of K times. In addition, it performed a single timsort upon a size K list which had a worst-case time complexity of O(K log K)[4] . Finally, the algorithm had various low-cost calls and declarations which could be simplified as a constant 1.

Therefore:

T(n) = K ⋅ (E log N) + K log K + 1

= K ⋅ E log N + K log K + 1

≤ Const K ⋅ E log N

≤ O(K ⋅ E log N)

Hence, it is observed that the algorithm followed a time complexity of O(K ⋅ E log N) which appeared to fit the characteristics of the testcase results.

# References

[1] Ben Alex Keen, 2017, *Dijkstra's Shortest Path and graph structure implementation heavy influence. URL: http://benalexkeen.com/implementing-djikstras-shortest-path-algorithm-with-python/*

[2] Jin Y. Yen, 1971, Yen’s K shortest path algorithm loose inspiration. *URL: https://en.wikipedia.org/wiki/Yen%27s\_algorithm*

[3] Wikipedia, 2019, *Dijkstra’s Algorithm.* *URL:* *https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm*

[4] Awdesh, 2018, *Timsort: The Fastest sorting algorithm for real-world problems. URL: https://dev.to/s\_awdesh/timsort-fastest-sorting-algorithm-for-real-world-problems--2jhd*

# Appendix

*Figure A: Input & Results of smaller testcase*

6 9  
C D 3  
C E 2  
D F 4  
E D 1  
E F 2  
E G 3  
F G 2  
F H 1  
G H 2  
C H 3

Found Path Costs in 0.00 (secs):

[5.0, 7.0, 8.0]

*Figure B: Sample of testcase used*

11825 28524  
0 1 9.868366945875247  
1 0 9.868366945875247  
2 0 13.22475463876114  
0 2 13.22475463876114  
3 4 23.43284736824765  
4 3 23.43284736824765  
5 6 1.6381453043463388  
6 5 1.6381453043463388  
7 8 49.093299303324045  
8 7 49.093299303324045  
...  
11823 11817 3.8468667731967576  
11817 11823 3.8468667731967576  
11823 11824 1.299408290979295  
11824 11823 1.299408290979295  
7685 8714 3

*Figure C: Known result of testcase used (as seen in Figure B)*

