Shortest Path Problem (Graph Algorithms)

2801ICT – Computing Algorithms (A1: Task 2)

Tiam Adnum

2023

Contents

[2 Problem Formulation 2](#_Toc136724355)

[2.1 Problem Overview 2](#_Toc136724356)

[2.2 Input Specifications 2](#_Toc136724357)

[2.3 Output Specifications 2](#_Toc136724358)

[3 Algorithmic Design 2](#_Toc136724359)

[3.1 Algorithm Description 2](#_Toc136724360)

[3.1.1 Dijkstra Algorithm 2](#_Toc136724361)

[3.1.2 Yens Algorithm 3](#_Toc136724362)

[3.2 Algorithm Pseudocode 4](#_Toc136724363)

[3.2.1 Dijkstra Algorithm 4](#_Toc136724364)

[3.2.2 Yens Algorithm 5](#_Toc136724365)

[4 Algorithm Analysis 5](#_Toc136724366)

[4.1 Testcase 5](#_Toc136724367)

[4.2 Performance Analysis 6](#_Toc136724368)

[4.2.1 Dijkstra Algorithm Performance 6](#_Toc136724369)

[4.2.2 Yens Algorithm Performance 6](#_Toc136724370)

[4.2.3 Overall time complexity 6](#_Toc136724371)

# Problem Formulation

## Problem Overview

Melanie’s father wants to travel from Southport to Brisbane CBD and wants Melanie to recommend the exactly K-shortest loopless paths. Melanie’s father insists that the first recommended path must be the shortest path between Southport and Brisbane CBD. While the rest of the K – 1 paths are approximated shortest paths which Melanie’s father will use as backup paths.

This is an application of the shortest path problem, in where the goal is to find the shortest path between two vertices in a graph. BFS cannot be used to solve this problem as the weights of the graph can be any value. Two of the most popular algorithms, Dijkstra’s and Bellman Ford’s, are unable to solve this problem so another algorithm must be used.

## Input Specifications

The input must be given in a file named “finalInput.txt”. The first line of input will contain 2 integers (N, M). Where N is the number of vertices and M is the number of edges. The following lines of input will contain 3 variables ai, bi, wi where ai and bi is the edge between vertices ai and bi while wi is the weight of that edge.

## Output Specifications

The output must print out the length of the K-shortest loopless paths separated by commas. These paths must include the shortest possible path from the Southport to Brisbane CDB followed by the rest of the k-1 shortest paths.

# Algorithmic Design

## Algorithm Description

### Dijkstra Algorithm

Uses a famous algorithm called Dijkstra’s Algorithm to find the lowest cost path between two points on a graph. It does this by iterating through a priority queue choosing the lowest cost vertices in the queue and iteratively updating the costs and child vertices as it moves through the graph.

* Takes 3 parameters: graph, source, and destination. Graph is the graph class that holds the structure of the graph. Source is the starting point from where we want to travel from. Destination is the end point, where we want to travel to.
* Initialize a dictionary that will hold the minimum cost to travel to each vertex from the source vertex. Initialize all vertex costs in dictionary to infinity.
* Initialize a dictionary that will hold the child\_vertex of each vertex in the graph. This will be used to backtrack the shortest path from the destination to the source. Initialize all vertex children in dictionary to None.
* Initialize cost of source vertex to 0.
* Initialize priority queue to hold vertices on the graph and add the source vertex with corresponding cost to the queue.
* While queue is not empty:
  + Pop the next vertex with the lowest cost out of the queue as current cost and current\_vertex.
  + If current\_vertex = destination, then end loop as we have found our end point.
  + If current\_cost > costs[current\_vertex], then shortest path to vertex has already been found, so we can go to next iteration in while loop.
  + Iterate through each neighbour of current\_vertex.
    - Calculate alternate cost (alt) for each neighbour by getting the cost of the edge from current\_vertex to neighbour and adding it to the current\_cost.
      * If alt < costs[neighbour]:
        + Update costs[neighbour] to alt.
        + Set child\_vertex[neighbour] to current\_vertex.
        + Add alt cost and neighbour vertex to queue.
  + Check if path to destination exists. If not return path=None, and cost=Infinity.
  + Create path from source to destination using the child\_vertex dictionary.
  + Return the shortest path from source to destination and corresponding cost.

### Yens Algorithm

Implements an algorithm called Yen’s algorithm which is a popular graph algorithm that is used to find a specified number of paths (k) between two points with the smallest path cost. It does this by using Dijkstra’s algorithm to find the shortest path, then removes edges from the graph and creates alternate paths.

* Takes 4 parameters: graph, source, destination, and K. Graph is the graph class that holds the structure of the graph. Source is the starting point from where we want to travel from. Destination is the end point, where we want to travel to. K is the number of shortest paths we want to create from source to destination.
* Uses the dijkstra\_algorithm function to calculate the shortest distance between the source and destination and stores the outputs in shortest\_path and shortest\_cost.
* Add shortest path to paths list with corresponding cost.
* Iterate k from 1 to K. This loop will find k+1 shortest paths since the shortest path was found using Dijkstra’s algorithm.
  + Iterates through each vertex in the shortest path:
    - Creates a spur from vertex (from existing vertex on shortest path).
    - Creates a root path (the path of vertices that lead up to the spur vertex).
    - Deletes all edges that are apart of the previous k-1 paths, to ensure that no vertices are repeated in new paths.
    - Uses the dijkstra\_algorithm function to calculate the shortest distance between the spur vertex and destination and stores output in spur\_path and spur\_cost.
    - If spur\_path exists:
      * Combine root\_path and spur\_path to make new path.
      * Calculate cost of new path.
      * Add new path and corresponding cost to list of possible paths.
  + Add the lowest cost path in possible paths to the paths list.
* Return list of paths.

## Algorithm Pseudocode

### Dijkstra Algorithm

function dijkstra\_algorithm(graph, source, destination):  
 for each vertex in graph.vertices:  
 cost[vertex] = INFINITY  
 child[vertex] = None  
 cost[source] = 0  
 add source to Q  
   
 while Q is not empty:  
 current\_vertex = vertex in Q with min cost[vertex]  
 remove current\_vertex from Q  
   
 for each neighbour of current\_vertex:  
 cost = cost[current\_vertex] + graph.edge\_weight[(current\_vertex, neighbour)]  
 if cost < cost[neighbour]:  
 cost[neighbour] = cost  
 child[neighbour] = current\_vertex  
 add neighbour to Q  
   
 if child[destination] is None:  
 return path=None, cost=INFINITY  
   
 path = []  
 vertex = destination  
 while vertex is not source:  
 add vertex to path  
 vertex = child[vertex]  
   
 add source to path  
 reverse path  
   
 return path, cost[destination]

### Yens Algorithm

function yen\_algorithm(graph, source, destination, K):  
 shortest\_path = dijkstra\_algorithm(graph, source, destination)  
 if shortest\_path is None:  
 return []  
 paths = [dijkstra\_algorithm(graph, source, destination)]  
 possible\_paths = []  
   
 for k from *1* to K  
 for i from *0* to len(paths[k-*1*] - *1*)  
 spur\_vertex = paths[k-*1*][i]  
 root\_path = paths[k-*1*].vertices(*0* to i)  
   
 for path in paths:  
 if root\_path = path.vertices(*0* to i)  
 remove path.edge(i, i+*1*) from graph  
   
 spur\_path = dijkstra\_algorithm(graph, spur\_vertex, destination)  
   
 total\_path = root\_path + spur\_path  
 if total\_path not in possible\_paths:  
 add total\_path to possible\_paths  
   
 restore edges to graph  
   
 if possible\_paths is empty:  
 break  
   
 sort possible\_paths  
   
 add possible\_paths[*0*] to paths  
 possible\_paths = []  
   
 return paths

# Algorithm Analysis

## Testcase

Testcase for Python Implementation using Yen’s Algorithm:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input (k)** | **Expected Output** | **Actual Output** | **Expected CPU time (milliseconds)** | **Actual CPU time (milliseconds)** |
| 1 | 1038.57, 1042.38 | 1038.577, 1042.384 | 1286 | 1043 |
| 3 | 1038.57, 1042.38, 1043.02 | 1038.577, 1042.384, 1043.029 | 2007 | 2543 |
| 5 | 1038.57, 1042.38, 1043.02, 1043.29, 1044.05 | 1038.577, 1042.384, 1043.029, 1043.291, 1044.058 | 7767 | 6351 |
| 7 | 1038.57, 1042.38, 1043.02, 1043.29, 1044.05, 1044.12, 1044.63 | 1038.577, 1042.384, 1043.029, 1043.291, 1044.638, 1044.900, 1045.044 | 10897 | 10323 |
| 10 | 1038.57, 1042.38, 1043.02, 1043.29, 1044.05, 1044.12, 1044.63, 1044.90, 1045.73, 1045.97 | 1038.577, 1042.384, 1043.029, 1043.291, 1044.638, 1044.900, 1045.044, 1045.306, 1045.648, 1049.455 | 20345 | 15207 |

Program always outputted the correct shortest path. Program had a slight amount of variation with k+1 approximate shortest paths, however this is to be expected as they are meant to be approximated. Algorithms seemed to be slightly more efficient overall than the ones used on the assessment task sheet, only performing slightly slower on the k=3 testcase.

## Performance Analysis

### Dijkstra Algorithm Performance

v = number of vertices in graph, e = number of edges in graph.

* Initiation of costs and child\_vertex: **O(v)**
* Outer while loop iterates until heap is empty, worse case: **O(v)**
  + Getting min value from heap, inside while loop: **O(log v)**
  + Iterating through neighbours of current vertex: **O(e)**
    - Adding distance and neighbour to heap: **O(log v)**
* Making path from source to destination: **O(v)**

Overall time complexity: **O((v + e) log v)**

### Yens Algorithm Performance

k = number of paths, v = number of vertices in graph, e = number of edges in graph.

* Outer loop iterates from 1 to K: **O(K)**
  + Second loop iterates through all vertices: **O(v)**
    - Third loop iterates through K paths to delete: **O(K)**
    - dijkstra\_algorithm function is used: **O((v + e) log v)**

Overall time complexity: **O(K \* v \* (v + e) log v)**

### Overall time complexity

Since the time complexity of the yen’s algorithm is greater, the overall time complexity for the program is **O(K \* v \* (v + e) log v).**