Caleb Howard
S5155126
2801ICT

K Shortest Paths Solver C++

Overview

The k-shortest path problem is a deviation of the shortest path problem in which we must find not only the shortest path in a given graph but also the next k-1 shortest paths. The variance of the problem that this document concerns is the acyclic k-shortest paths problem.
This document will include an explanation of the solver, a description of how it works, results, an innovation discussion and a performance analysis.

Solution

The solver exploits the existing Dijkstra's algorithm, a single source shortest path algorithm which finds the shortest distances to all nodes in a graph from a single source.

Firstly to get the distances to all nodes from our source node we run Dijkstra's algorithm on our adjacency matrix. Once we have a vector of these values we can exploit Dijkstra's algorithm by reversing all directions in our graph, this can be achieved by transposing our adjacency matrix. Once we have our reversed directions we run Dijkstra's algorithm with the destination node as the source node, this will give a vector of the shortest distances from all nodes to the destination node.

Now that we have our two shortest-distance vectors, one for shortest distance to all nodes from source, and another for shortest distance from all nodes to destination, we can sum the values of these two vectors. The produced vector will be for each node, the shortest path distance it takes to get to that node from source plus the shortest path distance it takes to get from that node to the destination.
The resultant shortest-distance vector will contain many duplicates, these are simply from nodes that all lie on the same shortest path; we remove these duplicates (path distances) by feeded the vector into a set. The set will also be responsible for sorting our shortest-path-distances.
We print the k first values in our set to get our k-shortest-paths.

The limitations of this implementations are as follows:

- We must calculate the shortest path for every node in the graph to get our k shortest distances. This means that for small k-values the algorithm produces slightly higher run times then algorithms such as Yen's algorithm, however, the growth of runtime is not exponential and we receive significantly fast runtimes for large values of k.
- If two different paths have the same distance then one path will be ignored from the final result, i.e. it will not be listed as the succeeding shortest distance.

Innovation

Although the Dijstra’s algorithm is a commonly known single source shortest path algorithm, I have yet to see it applied this way. By reversing the directions of a graph and using the destination as source we are able to implement a single destination shortest path algorithm. Using this implementation of Dijstra’s algorithm allows us to calculate for each node the shortest path to that node from the source and the shortest path from that node to the destination. This is useful as summing these values will show us which nodes lie on which shortest paths (as seen in figure 1). By removing duplicates, as duplicates generally mean that the nodes are on the same path (note that this is not always the case as listed in the limitations above), we can obtain a vector of shortest-paths.

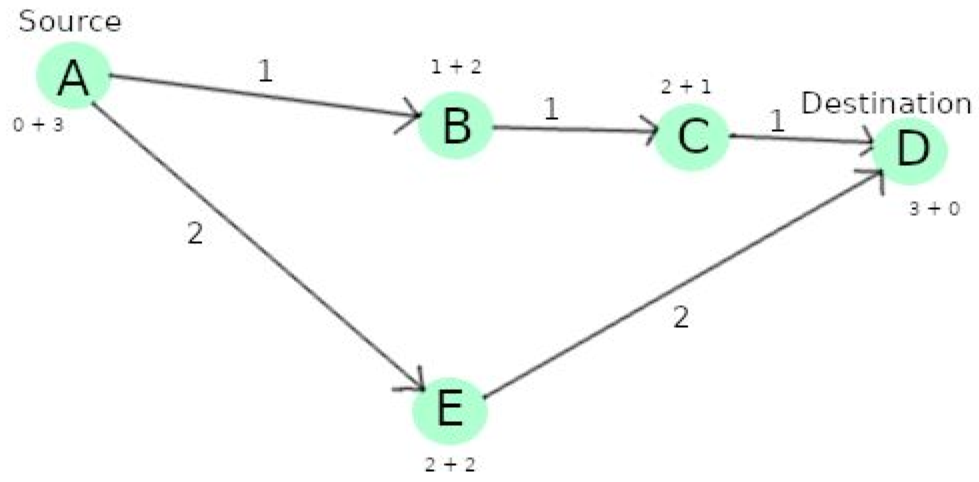

Figure 1: Each node represents: shortest distance from source to node + shortest distance from node to destination.
Nodes B and C both equal 3, therefore as defined by our algorithm, lie on the same path.
It is important to note that if node E had a shortest path result equal to 3, for example 1+2, then it would not be included as the second shortest path.

Pseudocode

```
BEGIN main_program
    // src: source node
    // dst: destination node
    // k: k-shortest-paths value
    // matrix: adjacency matrix representing the digraph
    // matrixT: transposed adjacency matrix (reversed directions)
    // Load matrix load returns a tuple of the required data
    // from the input file
    src, dst, k, matrix, matrixT ⇦ load_matrix(“input_data.txt”)

    k_shortest_paths(matrix, src, dst, k)
 END

BEGIN k_shortest_paths
    // Get the distance from src to all other nodes
    srcDist ⇦dijkstra(matrix, src)

    // If the k value is one we can simply print the shortest
    // path to the destination node from source
    IF k == 1:
        print: k=1 shortest path is srcDist[dst]
        RETURN

    // Get the distances from all nodes to the destination
    dstDist ⇦dijkstra(matrixT, dst)

    // Add the corresponding vector values
    For i ⇦ 0 to length(dstDist) - 1 do:
        srcDist[i] = srcDist[i] + dstDist[i]

    // Load the values into a set to remove duplicates
    // and sort the values
    distances = set(srcDist)

    // Print k-shortest paths
    For i ⇦ 0 to k-1 do:
        print: k=i+1 shortest path is distances[i]
END
```

```
BEGIN dijkstra
    // Vector of distances all set to infinity
    dists ⇦ vector of size length(matrix) where each value is DOUBLE_MAX

    // boolean vector, initially all false, if node i is included in the shortest path or
    // the shortest distance from src to i is processed then
    // p[i] = true else = false
    p ⇦ vector of size length(matrix) where each value is false

    // Distance from source node to itself is 0
    dist[src] = 0

    // Pick the minimum distance vertex from the set of vertices not
    // yet processed
    min_v ⇦ min_distance(dist, p);

    // Mark the node as processed
    p[min_v] = 1;

    // Update the distance value of the nodes adjacent to the processed node
    For  v ⇦ 0 to length(dist) -1 do:
        // Update dist[v] only if it is not in p, there is an edge from
        // min_v to v, and total weight of path from src to v through m is
        // smaller than the current value of dist[v]
        IF not p[v] and matrix[min_v][v] and dist[min_v] is not DOUBLE_MAX
           and dist[min_v] + matrix[min_v][v] < dist[v] do:
           dist[v] ⇦ dist[min_v] + matrix[min_v][v];

    // Return the list of distances from src to all nodes
       Return dist
```

```
BEGIN min_distance
    // function to find the minimum distance value in dist
    // that is not included in p

    min ⇦ DOUBLE_MAX
    minIndex ⇦ 0

    For v ⇦ 0 to length(dist)-1 do:
        If not p[v] and dist[v] < min:
            min ⇦ dist[v]
            minIndex = v

    Return minIndex
END
```

Results

Input (k value)	Output (each row is additive to the k-vector)	CPU Time (seconds)
1	1035.62	0.336805
2	1036.72	0.446804
3	1036.95	0.496830
4	1037.23	0.512539
5	1037.83	0.434214
6	1038.46	0.422320
7	1038.55	0.500101
8	1038.57	0.499870
9	1038.58	0.410028
10	1038.82	0.461893

It can be inferred from the results table above that the algorithm produced the correct output for the given input data and did so efficiently. A more in depth analysis is done in the section below.

Note: These results are from the new input file that Peter uploaded. These inputs are the same except duplicate lines were removed. This explains the difference in results from the assessment sheet. If you wish to run the old file and check it against the assessment sheet results, I have left that file in the program directory as input_data_old.txt.
Simply run the program with the program argument “input_data_old.txt”.

Performance Analysis

The performance analysis will be done on each function starting from the core functions and working up to the main solver function. These complexities will be combined to give the final Time and Space complexity of the entire program.
Note that Dijkstra's algorithm is already a commonly known complexity, therefore an analysis will not be conducted for the dijkstra function.

Let V = number of vertices.
Let E = the number of edges.
Let k = the k-shortest-paths value.

load_matrix()
Time Complexity:
The load matrix method loads the data from each line of the input file into the associated data structure. Assume that reading a single string from the data file costs constant time of 1, as loading data into a 2D vector matrix costs O(1) time. The first and last lines both contain two strings each and the lines in between, of which there are E lines, contain three strings each. Putting this together we get a time complexity of 3E + 4, we take the dominant term and write it as O(E).

Space Complexity:
The load_matrix() method creates two 2D vector matrices of sizes V by V. Therefore we get a space complexity of $O(V^2)$.

min_distance()
Time Complexity:
The main operation in this function is a loop that runs for size V, other operations nested in the loop are basic operations with constant time complexities of 1. Therefore, the time complexity of min_distance() is O(V).

Space Complexity:
The min_distance() function uses references to the vector matrix and p vector, therefore it uses constant space of O(1).

dijkstra()
Time Complexity:
$O(E\log_2 V)$

Space Complexity:
Dijkstra's algorithm uses two vectors of size V to store data. Therefore we get O(V) space complexity.

k_shortest_paths()
Time Complexity:
The first part of this function runs Dijkstra for the source node which is $O(E\log_2 V)$ as calculated above. If k equals 1 the function will print the shortest distance and terminate, we will therefore have $O(E\log_2 V)$ time complexity.
Assuming worst case k > 1, the function will not terminate and we will run Dijkstra for the destination node on the transposed matrix with $O(E\log_2 V)$ time complexity. Now we cycle through the two calculated vectors linearly with O(V) time complexity, summing the corresponding values. Next we feed the resultant vector into a set. An std::set is based on a Red-black tree data structure, which has a single element insertion complexity of $\log_2 V$. Therefore, inserting V elements into the set gives us $V\log_2 V$ time complexity for that operation. Finally we iterate through the set k times to print the k-shortest-paths. Putting the complexities together we get, $2E\log_2 V + V\log_2 V + k$. This gives us a final big O time complexity of $O(V\log_2 V + E\log_2 V)$.

Space Complexity:
To calculate the return value this method stores two distance vectors, each of size V, and a set with worst case size complexity of V. This gives us a final space complexity of 3V which translates to O(V) in big O complexity.

Performance conclusion:
Now that we have calculated the complexity for each function we take the most dominant part that is proportional to the input.

Time Complexities:
$O(V\log_2 V + E\log_2 V + V + k)$
By taking the dominant terms that are not proportion to each other we get:
Final Time Complexity: $O(V\log_2 V + E\log_2 V + k)$

Final Space Complexity:
$V^2 + 2V$
By taking the dominant term we get $O(V^2)$.