Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

K Shortest Path Implementation

by

RadhaKrishna Nagubadi

LIU-IDA/LITH-EX-A--13/041--SE

2013-06-27

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

K Shortest Path Implementation

by

RadhaKrishna Nagubadi

LIU-IDA/LITH-EX-A--13/041--SE

2013-06-27

Supervisor: Fang Wei-Kleiner
Examiner: Patrick Lambrix

Abstract

The problem of computing K shortest loopless paths, or ranking of the K shortest loopless paths between a pair of given vertices in a network is a well-studied generalization of shortest path problem. The K shortest paths problem determines not only one shortest path but the K best shortest paths from *s* to *t* in an increasing order of weight of the paths.

Yen's algorithm is known to be the efficient and widely used algorithm for determining K shortest loopless paths. Here, we introduce a new algorithm by modifying the Yen's algorithm in the following way: instead of removing the vertices and the edges from the graph, we store them in two different sets. Then we modified the Dijkstra's algorithm by taking these two sets into consideration. Thus the algorithm applies glass box methodology by using the modified Dijkstra's algorithm for our dedicated purpose. Thus the efficiency is improved. The computational results conducted over different datasets, shows the proposed algorithm has better performance results.

Acknowledgements

First and foremost I would like to thank my supervisor Fang Wei for giving me the opportunity to work under her supervision and guiding me throughout this period with her valuable time and suggestions.

I would like to thank my examiner Patrick Lambrix for his valuable suggestions.

Finally, I would like to thank my loving family and friends, who are there with me all the time by giving moral support and encouragement in all phases of time.

Table of Contents

List of Figures and Tables

Chapter 1

1. Introduction

1.1 Problem Statement

The problem of shortest path in a graph determines the path with shortest distance between given any two vertices namely, source and destination vertices. The shortest path problem is a well-known graph problem studied in the graph theory. There are several applications which include network routing, transportation, robot motion planning, word ladder puzzles, etc.

However, in most of the complex network applications, it is important to have more than just the computation of a single shortest path problem, since the graph usually represents a mathematical model which lacks appropriate data. Thus the best solution to the mathematical problem is sometimes not the same as the best solution for the real world problem. Hence, in addition to the best solution, the computation of more alternative paths between given two vertices would be useful to define the best solution according to the information not included.

Additionally in a transportation network, if the shortest path between two vertices is busy, then in order to travel between these vertices the next best shortest path is required, which is the second shortest path. And if the second path is also busy then the third shortest path is required and so on. Thus the computation of alternative paths between any two vertices, beforehand would be beneficial. The problem of computing more alternative paths than the best shortest path solution is said to be the *K shortest path problem*.

The K shortest paths problem is an extension of the shortest path problem, which determines not only one shortest path but the K best shortest paths from s to t in an increasing order of weight of the paths. The problem of computing K shortest paths, or ranking of the K shortest paths between a pair of given vertices in a network is also a well-studied generalization of the shortest path problem. Hoffman and Pavley [1] proposed an algorithm for solving it in 1959. The problem determines, for any given $K \geq 1$, the first shortest path, the second shortest path . . . until the K^{th} shortest path between given pair of vertices. Given a graph *G* with non-negative edge weights, a positive integer K, and the two vertices *s* and *t*, the algorithm computes the K shortest paths from *s* to *t* in increasing order of weight of the paths.

Since then, several articles have been published on determining K shortest paths with different algorithms. One can find the bibliography of K shortest paths, with several articles available online at: http://liinwww.ira.uka.de/bibliography/Theory/k-path.html.

The K shortest paths problem is classified into two types. The first one is allowed to have cycles (or loops) in paths from *s* to *t*, where the path contains repeated vertices. The best known algorithm for this type of problem is proposed by David Eppstein [2], with $O(m + n \log n + K)$ run time complexity in worst-case analysis. The algorithm by Eppstein, determines an implicit representation of paths, where each path can be output in $O(n)$ additional time.

In the second type of problem it is not allowed to have repeated vertices in paths, which were called loop less paths or simple paths. The problem of computing the K shortest loopless paths is proved to be harder, when compared with computing of the paths that have loops. The best known algorithm for this type of problem till date is proposed by Yen in 1971 [3] and the run time complexity is O (K n (m + n log n)) in worst-case time. Several improvements have been proposed to Yen's algorithm, which results with the same worst-case bound as of Yen and even some algorithms are exponential even though their practical performance is good. Martins and Pascoal implementation [4] is the one that has same worst-case bound as of Yen's algorithm, whereas its performance in the computational experiments shown to be better than Yen's.

1.2 Goal of Thesis

The main objective of this thesis is to find the K shortest paths between any two given source and destination vertices in a graph efficiently, where $K \geq 1$. The K shortest paths in a graph determines the shortest path, second shortest path, third shortest path and so on until the K^{th} shortest path between given source and destination vertices.

1.3 Algorithm

Here, we introduce a new algorithm to determine the K shortest loopless paths between two vertices in a graph. Our algorithm is based on the modified implementation of Yen's algorithm. The algorithm uses a different approach, instead of removing the vertices and the edges from the graph, we store these vertices and edges in two different sets. Then we modified the Dijkstra's algorithm by taking these two sets into consideration and apply the modified Dijkstra's algorithm whenever required to determine the shortest path. Even though, the worst-case analysis of the proposed algorithm differs to the Yen's algorithm, with complexity of O (K n (m log n)), the proposed one gives a better performance in the computational results. The comparison is made to estimate their behaviors in average-case.

1.4 Overview

This section provides an overview of the thesis organization and a review of this report.

- **Chapter 2**
  This chapter explains about the Yen's algorithm. It covers two approaches of Yen's algorithm namely, straight-forward approach of Yen's algorithm and Martins and Pascoal Implementation of Yen's algorithm.
- **Chapter 3**
  This chapter explains about the algorithm proposed in this thesis work. It covers the working method of Best Path Algorithm which determines the shortest path and the K Shortest Path Algorithm which determines the K shortest paths.
- **Chapter 4**
  This chapter explains a brief overview of the data structures used in the implementation process of Best Path Algorithm and K Shortest Path Algorithm. Moreover, this chapter provides the worst-case analysis of the algorithm presented and the computational results made.
- **Chapter 5**
  This chapter discusses about the thesis conclusion and future work.

Chapter 2

2. Yen's Algorithm

This chapter describes the working of two algorithms. The straight-forward implementation of Yen's algorithm and the new implementation of Yen's algorithm for computing the K shortest paths for any given two vertices in a graph. Although the worst case time complexity of both the approaches are same, it is shown in [4], the new implementation approach has better running time than the straight forward approach of Yen's algorithm.

Yen's algorithm determines K shortest loopless paths for given source and destination vertices in a graph with non-negative edge weight. The algorithm proposed that, for determining the K^{th} shortest path, the first K-1 shortest paths has to be determined previously. Thus the first shortest path has to be determined before determining any other shortest paths. And this can be done by any shortest path algorithm, whereas Dijkstra's algorithm is used by Yen's. Each vertex *v* in the first shortest path will be analyzed, and a set of candidate paths will be generated by deviating at *v* in the first shortest path. The next shortest path would be determined from this set of candidate paths. With the newly determined shortest path, the process continues to compute the next shortest path until the K^{th} shortest path has been determined.

2.1 Straight-Forward Implementation

The straight-forward approach was proposed by Yen [3]. In order to compute the shortest path P^k, where k<K, the shortest paths P^1, $P^2 \ldots P^{k-1}$ has to be determined previously. Thus each vertex in the path P^{k-1} has to be analyzed. The shortest paths that were already known (i.e., the paths P^1, $P^2 \ldots P^{k-1}$), will be stored in a list (list A) and the candidate paths for the next shortest path will be stored in another list (list B). Let the path P^{k-1} be $<s, v_1^{k-1}, v_2^{k-1} \ldots v_l^{k-1}, t>$. If a vertex v_i^{k-1} is being analyzed, then the sub path from *s* to v_i^{k-1} is said to be *root path* and the path which has to be computed from v_i^{k-1} to *t* is said to be *spur path*. When a *spur path* is determined, then it is joined with the *root path* to form a complete path from *s* to *t* and the vertex v_i^{k-1} is said to be the deviation node of the newly computed path which could be a candidate for the next shortest path. Thus when a path deviates at some vertex *v*, then *v* is the deviation node of that path. In figure 2.1 (b), the path P^3 is deviated at vertex 4 in path P^2 and hence 4 is the deviation node of path P^3. Let us consider the graph *G* in Figure 2.1 (a) shown below. Let vertices 1 and 0 be the source and destination vertices and the shortest path from 1 to 0 is determined from Dijkstra's algorithm as <1, 2, 3, 0>. Let this path be P^1.

Now, all the vertices in path P^1, except the destination vertex, from the set of nodes {1, 2, 3} has to be analyzed. Since the deviation node for the first shortest path is source vertex, 1 will be analyzed first. Now, the Dijkstra's algorithm is applied with source and destination vertices as 1 and 0, but the weight of the edge (1, 2) will be considered as infinity. Thus the Dijkstra's algorithm generates a new shortest path from 1 to 0 as <1, 4, 3, 0> and the deviation node of this new path is 1 since it is deviated to new edge (1, 4) instead of edge (1, 2). Add the path <1, 4, 3, 0> to list B, where all the candidates for the next shortest path are stored. Then restore the weight of the edge (1, 2) to its original value.

The next vertex to be analyzed is vertex 2. Now the shortest path from 2 to 0 has to be determined by setting the edge (2, 3) weight to infinity. The path <1, 2> is the root path and the path to be determined from 2 to 0 is the spur path. In order to determine the loopless spur path, the vertices in the root path has to be

temporarily removed from the graph. There is no possible path from 2 to 0 and thus the Dijkstra's algorithm returns null path. Restore the edge (2, 3) and the removed vertices in the root path. The next vertex to be analyzed is vertex 3. Set the weight of the edge (3, 0) to infinity and the root path is <1, 2, 3>. Remove the vertices in the root path from the graph and determine the shortest path from 3 to 0. Thus the spur path <3, 4, 5, 0> is found and added to the root path to form the total path as candidate which will be added to list B. The deviation node of this newly found path is 3.

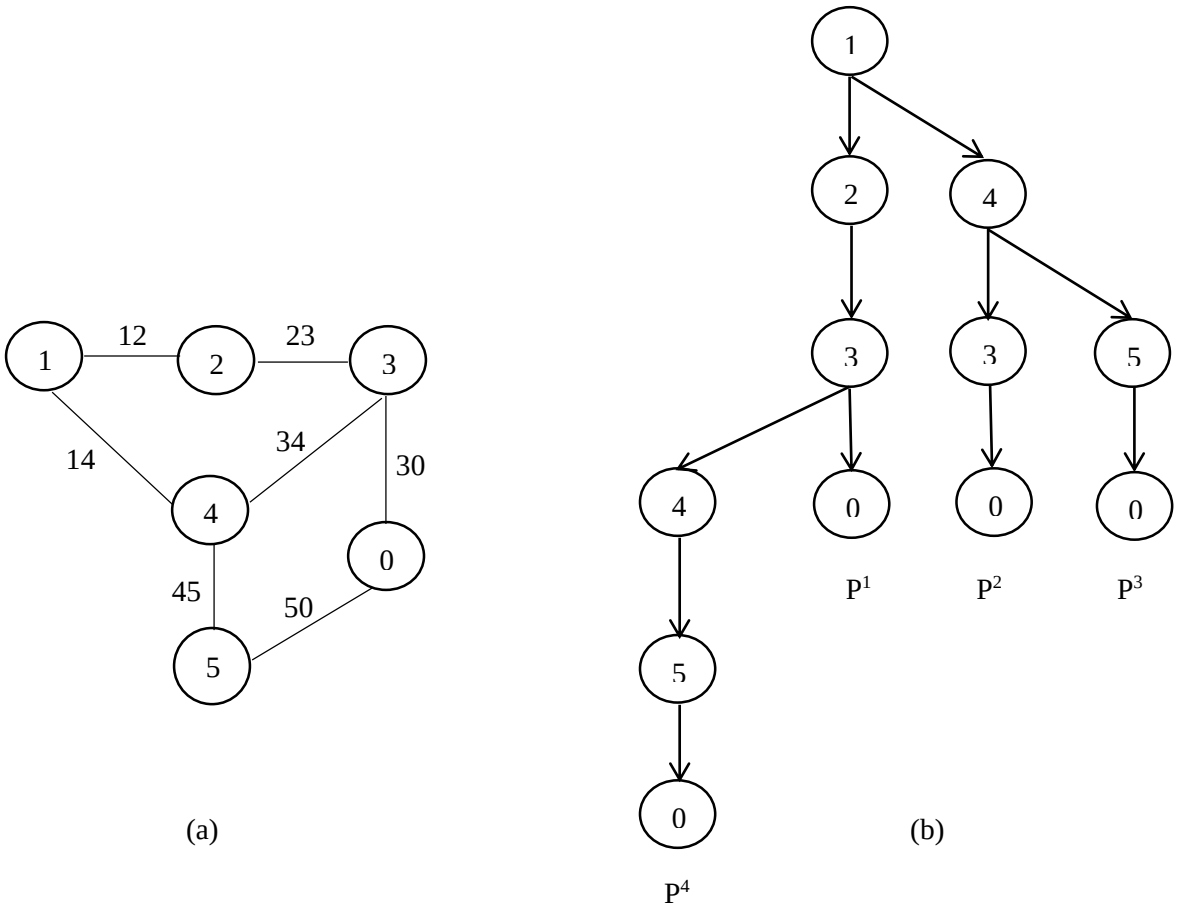


Figure 2.1: (a) The graph G = (V, E) and (b) Set of shortest paths from 1 to 0 in G as deviation paths.

All vertices in the path P^1 are analyzed and we choose the path with shortest distance from the list B as the next shortest path and move it into list A. The shortest path P^2 is <1, 4, 3, 0>. Similarly, all vertices in the path P^2 has to be analyzed as above and a new candidate path <1, 4, 5, 0> is found and added into list A. Now, the path with shortest distance in the list B is moved into list A as P^3 be <1, 4, 5, 0>. When P^3 is analyzed no new candidate paths are added into list B. Thus the only path in list B is moved into list A as P^4 <1, 2, 3, 4, 5, 0> with deviation node 3. Thus the vertices to be analyzed in P^4 are 3, 4 and 5. Here when analyzing the vertex 3, the root path is <1, 2, 3>. If any path in list A has the sub path same as the current root path then the weight of the edge from 3 to its immediate neighbor in that path has to be set to infinity.

Here, P^1 has same sub path, thus the weight of the edge (3, 0) has to be set to infinity along with the edge (3, 4) in path P^4. Then the Dijkstra's algorithm is used to determine the spur path from 3 to 0.

2.1.1 Algorithm

1. Determine the shortest path P^1 from s to t in a graph G by using the Dijkstra's shortest path algorithm.
2. Assume that k-1 (where k = 2, 3 . . . K) shortest paths are already determined and stored in list A and candidate paths for next shortest path are stored in list B.
3. In order to determine the shortest path P^k, get the shortest path P^{k-1} and let the path be $<s, v_1^{k-1}, v_2^{k-1} \ldots v_l^{k-1}, t>$ and the set of vertices to be analyzed is $DS = \{s, v_1^{k-1}, v_2^{k-1} \ldots v_l^{k-1}\}$.
4. For each vertex v in DS do
    1. If there exists a path P^j in list A that has the path $<s, v_1^{k-1}, v_2^{k-1} \ldots v>$ as the sub path. Then set the weight of the edge from v to its immediate neighbor to infinity for P^j.
    2. Set the sub path $< s, v_1^{k-1}, v_2^{k-1} \ldots v >$ in P^{k-1} as the root path, R^k. Set the path to be determined from v to t is as the spur path, S^k. Remove the vertices in the R^k from the graph. so that they are not repeated in spur path.
    3. Compute the shortest path from v to t by using the Dijkstra's algorithm.
    4. If a path is found and returned by Dijkstra's algorithm, then add both R^k and S^k to form a candidate path, for next shortest path. Add this path into list B and continue.
5. Choose the path from list B with shortest distance as P^k and move it into list A.
6. Go to step 3 and continue until K shortest paths have been determined.

Algorithm 1: Yen's Algorithm [3]

2.2 Pascoal and Martins (PM) Implementation

Pascoal and Martins (PM) implementation is the modification of Yen's algorithm which was proposed by E. Martins and M. Pascoal [4]. Unlike the straight-forward approach proposed by Yen, the new approach analyzes the vertices in the path P^{k-1} in reverse order from the last vertex to the deviation node except the destination vertex. Additionally, a *shortest tree* T_v rooted at destination vertex v will be computed from the remaining graph in which the shortest distances from every vertex to v is labeled in the tree. Thus the spur paths will be determined by using the shortest tree instead of the shortest path algorithm.

Let us consider the graph shown in Figure 2.1 (a) again and the shortest path from 1 to 0 is determined by using the Dijkstra's algorithm as P^1 be <1, 2, 3, 0>. Now, all the vertices and edges in the path P^1 are removed temporarily from the graph except the vertex 0 which is the destination vertex. Let us denote the modified graph as *G1*. With the help of Dijkstra's algorithm, the shortest tree rooted at vertex 0 is determined. The shortest distance from each vertex in *G1* to 0 is labeled as shown in Figure 2.2 (a). Then the first vertex to be analyzed is vertex 3 which is not in *G1*. So, first vertex 3 is restored into *G1* and the shortest distance from vertex 3 to vertex 0 has to be computed by looking at the adjacent vertices of 3 in *G1*. If any of the adjacent vertex of 3 is in *G1*, then the distance from 3 to 0 can be determined.

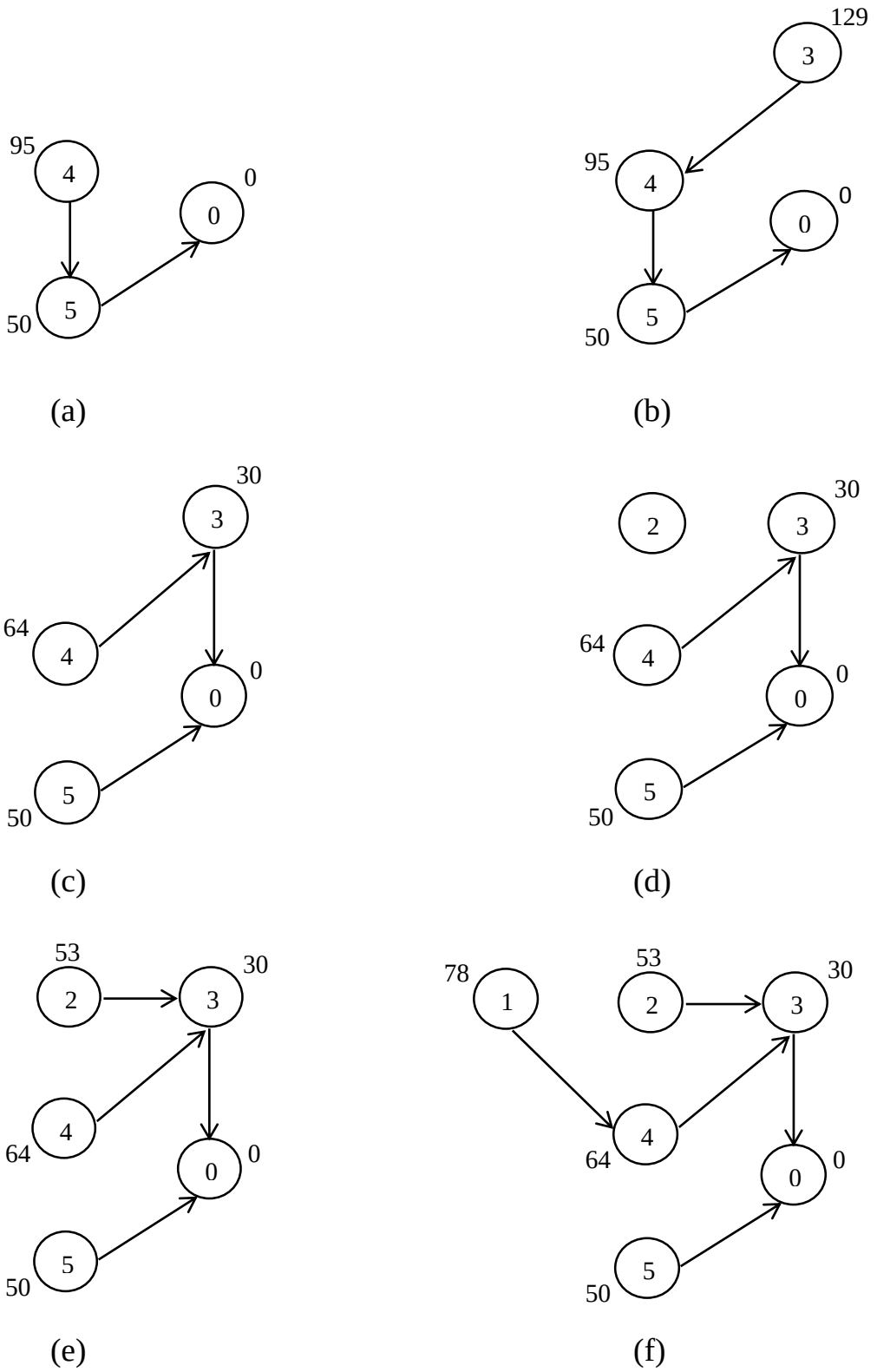


Figure 2.2: iterations of shortest tree when analyzing vertices in a path

The adjacent vertices of 3 are 2, 4 and 0. But since the vertex 2 is not in *G1* and vertex 0 cannot be reached directly as the edge (3, 0) is removed from the graph, vertex 4 is the only option to be considered and the shortest distance from 3 to 0 is determined and labeled as shown in Figure 2.2 (b) and the shortest path from 3 to 0 can be obtained from the tree. Now, add this sub path to the root path <1, 2, 3> and insert the entire path <1, 2, 3, 4, 5, 0> into list B.

Before the next vertex 2 to be analyzed, the edge (3, 0) has to be restored into the graph *G1* and the label of vertex 3 in the tree has to be updated if possible. If the label of vertex 3 is improved, then the labels of vertices adjacent to vertex 3 could be improved. After the edge (3, 0) is restored into the graph, the updated tree is shown in Figure 2.2 (c). Note that the label of vertex 4 is updated due to the insertion of vertex 3. Then the vertex 2 is restored into the graph but the label of vertex 2 is not determined as seen in Figure 2.2 (d) as the adjacent vertex 1 is not in the graph and the edge (2, 3) is also not in graph. Next the edge (2, 3) has to be restored into *G1*. Figure 2.2 (e) represents the tree, after the edge (2, 3) has been restored. Then vertex 1 is restored and the label is updated as represented in the Figure 2.2 (f). As the shortest distance from vertex 1 is determined, the shortest sub path from 1 to 0 is determined and could be joined with its root path and add into list B. Now, all the vertices in the path P^1 has analyzed and the path with the shortest distance stored in list B is the next shortest path and moved into list A. Thus the path P^2 be <1, 4, 3, 0> and all the remaining removed vertices and the edges were restored into the graph.

In order to determine the next shortest path, the path P^2 has to be analyzed now. Thus all vertices and edges in the path P^2 have to be removed and the shortest tree rooted at 0 has to be computed again. Consider the vertex to be analyzed in path P^2 be the deviation node, vertex 1 in this case. Then along with edge (1, 4), all the edges that were removed when P^2 was determined have to be again removed from the graph. Thus the edge (1, 2) has to be removed again and continue the same procedure as above until the required number of shortest paths have been determined. Figure 2.1 (b) represents the shortest paths from 1 to 0 in a graph discussed previously.

2.2.1 Algorithm

In the following we introduce the PM implementation. The algorithm differs from Yen's algorithm by introducing the new data structure shortest tree T_v rooted at vertex v. The tree, T_v represents the tree of shortest paths from every vertex to v and path from vertex v_i to v will be denoted by T_v (v_i). The weight of the path will be represented by π (v_i) which was labeled beside the vertex as shown in the above Figure 2.2. T_v has to be determined from the Dijkstra's algorithm.

When a vertex v_i has to be restored into the graph, the label of v_i has to be determined by using the *forward star form method*. Forward star form is the method of determining the labels of a vertex in the shortest tree, when a vertex is restored back into the graph. For all the vertices v_j adjacent to v_i in the graph, add the edge weight (v_i, v_j) with the π (v_j) and the shortest among them be the label of vertex v_i. When an edge (v_i, v_{i+1}) has to be restored into the graph, the vertices which were adjacent to v_i could be improved and this can be updated by the *reverse star form method*. Reverse star from is the method of updating the labels of a vertex in the tree, when an edge is restored into the graph. If label of a vertex v_j is improved then the label of its adjacent vertex could be improved. Thus only after restoring the vertex v_i and the edge (v_i, v_{i+1}), the next vertex v_{i-1} has to be analyzed.

1. Determine the shortest path P^1 from u to v by using a shortest path algorithm (preferably Dijkstra's algorithm).
2. Consider that k-1 (where k = 2, 3 . . . K) shortest paths are already determined and stored in a list A and candidate paths for next shortest path are stored in list B.
3. To determine the shortest path P^k, all the vertices and edge except the destination vertex in the path P^{k-1} have to be removed temporarily from the graph and let the path be $<u, v_1^{k-1}, v_2^{k-1} \ldots v_l^{k-1}, v>$ and the deviation node be v_i^{k-1} where $i < l$. Also remove the edges that are removed from the graph when P^{k-1} was generated have to be, once again, removed from the graph when deviation node is being analyzed.
4. A shortest tree, T_v rooted at v has to be determined from the remaining graph.
5. For each vertex in the path P^{k-1} from deviation node to v_l^{k-1} in reverse order do
    1. Restore the vertex v_i^{k-1} into the graph and compute the label of vertex v_i^{k-1} in the tree T_v by using the forward star form method.
    2. If the label of v_i^{k-1} determined, then the shortest path from v_i^{k-1} to v could be obtained from the shortest tree by T_v (v_i^{k-1}).
    3. If the label of v_i^{k-1} not determined, there wouldn't be any path from v_i^{k-1} to v and then the edge (v_i^{k-1}, v_{i+1}^{k-1}) has to be restored into the graph in order to analyze the next vertex.
    4. Update the labels of vertices in the tree by using the reverse star form method.
6. Restore all the remaining removed vertices and edges into the graph and continue from step 3 until the K shortest paths have been determined.

Algorithm 2: PM's Algorithm [4]

Procedure 1: update label of v_i using forward star form

1. For all the neighbors v_j of v_i in $G1$
    1. If (π (v_i) > π (v_j) + weight (v_i, v_j))
    2. Then π (v_i) = π (v_j) + weight (v_i, v_j)
2. Then continue with step 1

Procedure 2: update labels of neighbors of v_i using reverse star form

1. Add v_i into list.
2. Repeat until the list is empty.
3. pop up the first element of list as v_i
4. For all the neighbors v_j of v_i do
    1. If (π (v_i) > π (v_j) + weight (v_i, v_j))
    2. Then update the label π (v_i) = π (v_j) + weight (v_i, v_j)
    3. Add v_j into the list
5. If list is empty stop

Algorithm 3: Forward and reverse star form [4]

2.3 Comparison

The PM implementation uses the shortest tree T_v for determining the spur paths instead of applying the shortest path algorithm every time when a new vertex is being analyzed. Note that to generate the shortest tree T_v one needs to call the Dijkstra's algorithm only once. This way, the efficiency of the algorithm is improved. On the other hand, by updating the label of the newly restored vertex and edges, one has to loop through all the vertices in T_v, in order to compute the shortest path in the process of forward star form. Moreover, the reverse star form method at step 5 (4) in Algorithm 2 involves a recursive function which could be time consuming.

Chapter 3

3. The Algorithm

In both the approaches that were discussed in the previous chapter, when a vertex v_i^k is being analyzed in a path P^k <s, v_1^k, v_2^k . . . v_l^k, t>, then the vertices in the sub path of s to v_{i-1}^k and the edge (v_i^k, v_{i+1}^k) will be temporarily removed from the graph. Then the shortest loopless path from v_i^k to t will be determined and then the removed vertices and edges have to be restored into the graph. When a vertex v_i^k is analyzed, the path from s to v_{i-1}^k is said to be root path and the vertices in the root path will be temporarily removed from the graph in order to compute the shortest loopless path.

In this chapter, we introduce a new algorithm by improving Yen's algorithm in the following way: Instead of removing vertices and edges from the graph, we stored the root path in one set and the neighbors of v_i^k whose edges have to be set to infinity into another set. Then we modified Dijkstra's algorithm by taking these two sets into the consideration. We named this algorithm as the Best Path Algorithm (BP Algorithm) (details see Section 3.1). The shortest loopless path will be computed by the BP Algorithm.

Intuitively speaking, Yen's algorithm and PM's implementation apply the black box approach by calling Dijkstra's algorithm each time the shortest distance needs to be computed. While our algorithm applies the glass box methodology by modifying the Dijkstra's algorithm for our dedicated purpose. This way, the efficiency can be improved.

This chapter first describes the working method of the BP Algorithm, which is the modified version of Dijkstra's algorithm. Then we continue with determining the remaining K-1 best shortest paths by using the BP Algorithm.

3.1 Best Path Algorithm

The Best Path Algorithm modifies the Dijkstra's algorithm by taking two additional data structures into consideration, namely root path and *E*, where root path consists of set of vertices which should not be considered while computing the k^{th} path and E consists of set of vertices that are the neighbors of given source vertex whose weight of the edges has to be set to infinity.

The BP Algorithm determines the best shortest loopless path between any given single source and destination vertices in a graph *G* (*V, E*). Let d[*v*] be the distance from *s* to *v*. We introduce a new data structure *Q* consisting a set of (key, value) pairs which are sorted according to the key. Here d[*v*] is stored as a key and the path from *s* to *v* is stored as the value.

The algorithm starts at source (*s*) vertex and traverses the graph in a breadth first search (BFS) manner and determines the path from *s* to every vertex *v* in *V*. For each adjacent vertex *w* of *v*, the path from *s* to *w* is stored in the container *Q* and vertex *w* is said to be *visited*. And then chooses the path with shortest weight from *Q* and the last vertex in the path is said to be determined. If a vertex *w* is visited before, then a path from *s* to *w* is already stored in *Q* and hence it checks for the weight of the path in *Q* and the weight of the newly found path and keeps the best path in the container. Thus there would be one path for each vertex *w* from *s* in *Q*. The path will be removed from *Q* when vertex *w* is determined and if the determined vertex is

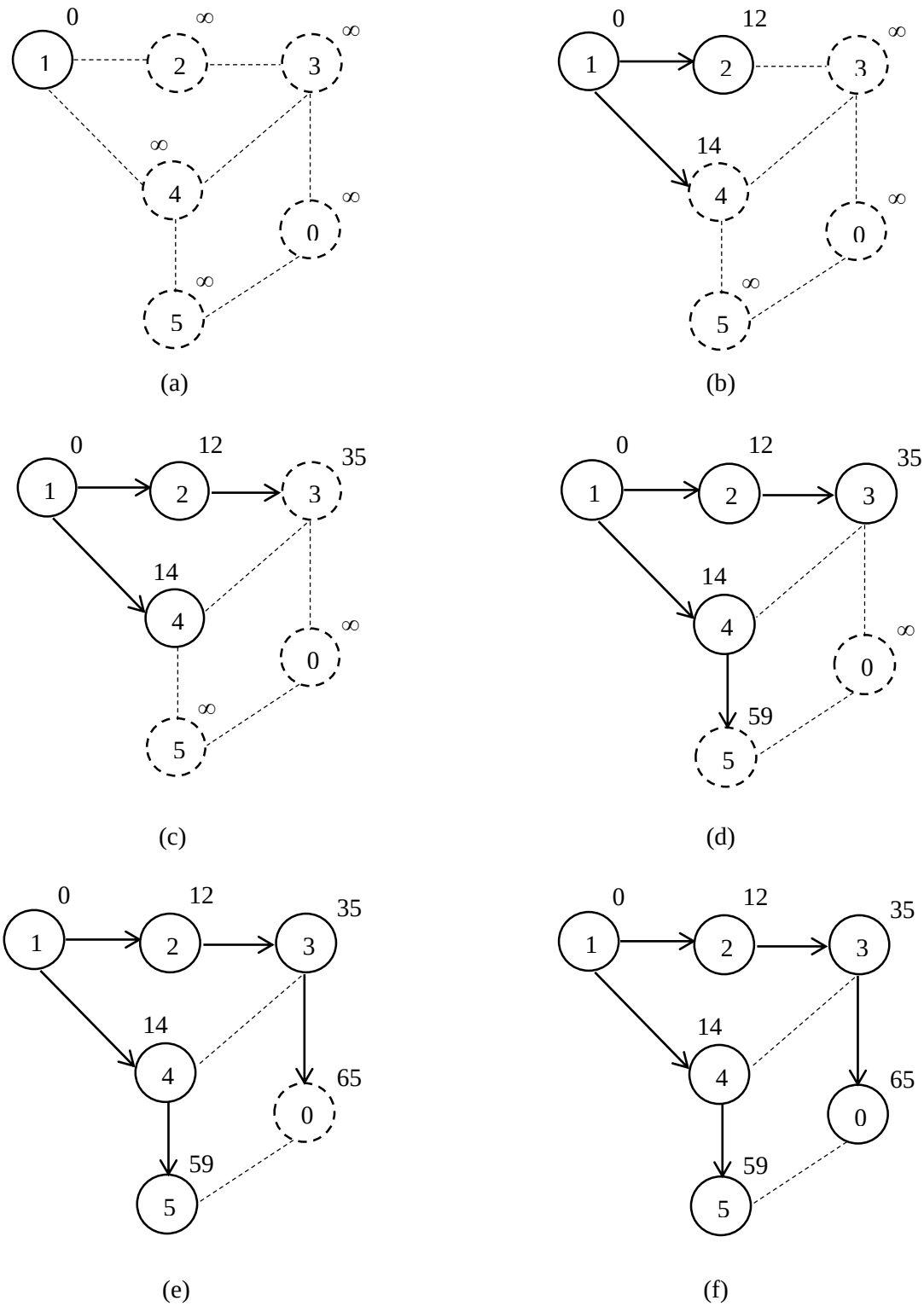


Figure 3.1: steps of computing the shortest loopless path from 1 to 0.

destination then the shortest loopless path is found and the algorithm returns it. A vertex *w* is said to be determined only if the path from *s* to *w* is the shortest.

Let us consider the graph in Figure 2.1 with source (1) and destination (0) vertices, the BP Algorithm starts searching for adjacent vertices from vertex 1. Let us assume the input parameters *root path* and *E* are both empty sets. Vertices 2 and 4 are adjacent to vertex 1 and the path <1, 2> with d [2] = 12 and path <1, 4> with d [4] = 14 is inserted into *Q* and mark the vertices 2 and 4 as visited. Then the path with lowest distance 12 is removed from *Q* and now current vertex is 2 which will be marked as determined and current path holds <1, 2>. Now, search for adjacent vertices from 2 which in this case is only vertex 3. Note that, in order to compute shortest loopless paths, the vertex which is already in the path <1, 2> would not be considered. Thus vertex 1 is ignored here and the adjacent vertex 3 is added to the current path and inserted into the container *Q* with path <1, 2, 3> and d [3] = 35. Vertex 3 is marked as visited.

Now, the current path becomes <1, 4> which is lowest in the *Q* and current vertex becomes 4. Vertex 3 and 5 are adjacent vertices to current vertex 4. Now, the temporary distance until 3 with path <1, 4, 3> is 48 and the adjacent vertex 3 is already visited but not determined. So, if 48 is less than d [3] = 35 then replace <1, 4, 3> instead of <1, 2, 3> in *Q*. Otherwise, continue with the next adjacent vertex, path <1, 4, 5> and d [5] = 59 is inserted into *Q* by marking 5 as visited. Now, the lowest distance in *Q* is removed and the current path holds <1, 2, 3> and current vertex becomes 3 which has adjacent vertex 0. Here adjacent vertex 4 is determined and vertex 2 is in the current path and hence both are ignored. Thus the path <1, 2, 3, 0> with d [0] = 65 is inserted into *Q*.

Then the path <1, 4, 5> is shortest and removed from the container and current path holds <1, 4, 5> and current vertex is 5. The path <1, 4, 5, 0> with distance 109 is not less than 65. Thus the next path removed from *Q* is <1, 2, 3, 0> where current vertex 0 is the destination vertex and hence the shortest path is determined from the best path algorithm. Figure 3.1 illustrates the steps of the best path algorithm diagrammatically where the distance from source to a vertex *v*, d[*v*] is labeled beside the vertex. And the path from source to *v* can be represented by the series of arrow lines as shown. Thus the shortest path from 1 to 0 can be represented by Figure 3.1 (f).

3.1.1 Pseudo Code

In order to maintain the paths from source vertex *s* to each vertex *v* in the graph and the paths respective distance d[*v*], the algorithm is using the multimap (discussed in detail in section 4.1.1) data structure with d[*v*] as key and the path from *s* to *v* as value whereas Dijkstra's implementation is based on a priority queue implemented by a Fibonacci heap. From the above example, path <1, 2, 3, 0> is value of the container *Q* and d[0] is key. Each path is stored as a vector of integers. If a vertex *v* is visited for first time then visited[*v*] is set to true and if the distance of the vertex from *s* to *v* is the shortest, then determined[*v*] is set to true. Initially both visited[*v*] and determined[*v*] will be set to false for all vertices *v* in the graph.

The algorithm starts by initializing d[*v*] as infinity, *v* could be any vertex from the graph *G* (*V, E*), where d[*v*] be the shortest distance from source vertex to *v* and all vertices were marked unvisited. Then start with the source vertex as current vertex by setting its distance d[*s*] to zero and marking source vertex as determined. Then searches for the adjacent vertices from current vertex and let *w* be the vertex adjacent to source vertex. Then line 22 in the below pseudo code computes the temporary distance from *s* to *w* by adding weight of the edge (*s, w*) to d[*w*]. If the temporary distance is less than d[*w*], then set d[*w*] equal to temporary distance as in line 23 or in 27.

Simultaneously, from line 28 the shortest path from the source vertex to vertex *w* will be inserted into list A along with its distance d[*w*] when a vertex is visited. This process repeats until the loop from line 17 to

```
Function BP (s, t, root path, E)
Input: s: source vertex, t: destination vertex
Input: root path: set of vertices not to be considered from s to t
Input: E: set of vertices not be considered as neighbors of s
Output: P_t, the shortest path from s to t
//P_v: path from s to v, if determined[v] = true then P_v is the shortest path from s to v.
//d[v]: distance of path P_v
//Q: set of pairs (Key: d[v] & value: P_v), path is from s to the current vertex v
1.  for each vertex v in graph
2.      d[v] = infinity;                          // d[v] is the distance from source to vertex v
3.      visited[v] = false;                       // visited[v] : true if a vertex v is visited before
4.      determined[v] = false;                    // determined[v] : true if  d[v] is shortest from s to v
5.  end for
6.  add s to path P_s                             // path is a vector of type integer
7.  d[s]  = 0;
8.  add d[s] and P_s to Q
9.  while (Q not empty)
10.     (P_v, d[v]) = pop up of first element of Q
11.     v = last vertex in P_v
12.     determined[v] = true;
13.     if (v = t)
14.         return P_v;
15.     else
16.         for each adjacent vertex w of v
17.             if determined[w] = false and w not in P_v and root path
18.                 if w is not in E
19.                     new_path = P_v;
20.                     add w to new_path;
21.                     tempdist = d[v] + weight(v, w);
22.                     if tempdist < d[w] and visited[w] = true
23.                         d[w] = tempdist;
24.                         P_w = new_path;
25.                     else
26.                         d[w] = tempdist;
27.                         insert path P_w into Q
28.                     end if
29.                 end if
30.             end if
31.         end for
32.         if (v == s)
33.             clear list E
34.             insert max value in E
35.         end if
36.     end if
37. end while
```

Algorithm 4: BP algorithm

line 32 executes and then get the path with shortest distance in list A and set the last vertex in the path as determined and make it as current vertex as shown above from line 10 to 12. Then remove that path from the candidate path list. If a vertex is determined then it would not be considered again even though if that vertex is adjacent for any current vertex. And if the current vertex is the destination vertex, then the shortest path is found and the algorithm returns the shortest path from *s* to *t*. Otherwise, search for the adjacent vertices from the current vertex and continue this process until the destination vertex is determined or the candidate list is empty. Thus the best path algorithm determines the shortest loopless path from *s* to *t*. The condition in line 18, makes sure that the paths determined will be loopless paths. And the condition in line 19, checks whether the edge (*v*, *w*) to be allowed or ignored while computing the k^{th} shortest path. Thus the best path algorithm discussed here will be applied while computing the k^{th} shortest path.

3.2 K Shortest Path Algorithm

The best shortest loopless path P^1 from *s* to *t* is determined from the BP Algorithm as described in previous section. Now, the K Shortest Path Algorithm (KSP Algorithm) determines the remaining K-1 shortest loopless paths by using P^1. The KSP Algorithm modifies the Yen's algorithm such that the modification of graph is not required each time when a vertex has to be analyzed. Instead of modifying the graph by removing vertices and edges, the algorithm additionally gives two sets as parameters while applying the BP Algorithm. The two sets are *root path* and *E* as described previously. Rest of the work will be done by the BP Algorithm.

In order to compute the shortest path P^j, where $j <$ K, the vertices in the shortest path P^{j-1} have to be analyzed. The shortest paths P^1, P^2 . . . P^{j-1} are stored in list A which holds the final K shortest paths and candidates for next shortest path will be stored in list B. Analyzing each vertex *v* in path P^{j-1} results in computing the shortest path P_{vt} without the edge (*v*, *w*), where *w* is the first vertex after *v* in path P^{j-1} and also without the vertices in the sub path P_{sv}. Here P_{sv} is the root path and P_{vt} to be determined is the spur path. Moreover, if any path *P* in list A has P_{sv} as the sub path then the edge (*v*, *w*) is also not to be considered while computing the spur path.

Let us consider the graph in Figure 2.1 again. The shortest path from 1 to 0 is determined by the BP Algorithm and hence P^1 is <1, 2, 3, 0>. Now, the KSP Algorithm starts with analyzing the path P^1. The vertices from deviation node of P^1 until 3 (i.e., 1, 2 and 3) has to be analyzed. The first vertex to be analyzed is vertex 1. The root path will be empty and the set E consists of vertex 2. The BP Algorithm is applied with source and destination vertices as 1 and 0 and the two sets root path and E. Figure 3.2 (a) depicts how the BP Algorithm considers this when determining the shortest path from 1 to 0. Since the vertex 2 is in set E, the BP Algorithm would not consider the edge (1, 2) and deviates through vertex 4 to reach vertex 0.

Thus the BP Algorithm returns the shortest path <1, 4, 3, 0> which is the spur path and the root path is empty this time. Add both the root path and the spur path together and add the path to B which could be one of the candidates for next shortest path P^2. Then clear the set E and the next vertex to be analyzed is vertex 2. Now, the root path holds <1>. The set E consists vertex 3 and with source (2) and destination (0) vertices the BP Algorithm is applied. But it did not return any path, since there are no adjacent vertices found from vertex 2 in the graph. Vertex 1 is in the root path and vertex 3 is in the set E. Then vertex 3 has to be analyzed and the root path holds <1, 2> and the set E has 0 in it. The path from 3 to 0 is returned by the BP Algorithm as <3, 4, 5, 0> which would be represented from Figure 3.2 (b). Now, combine the root path and the spur path together and add it into B.

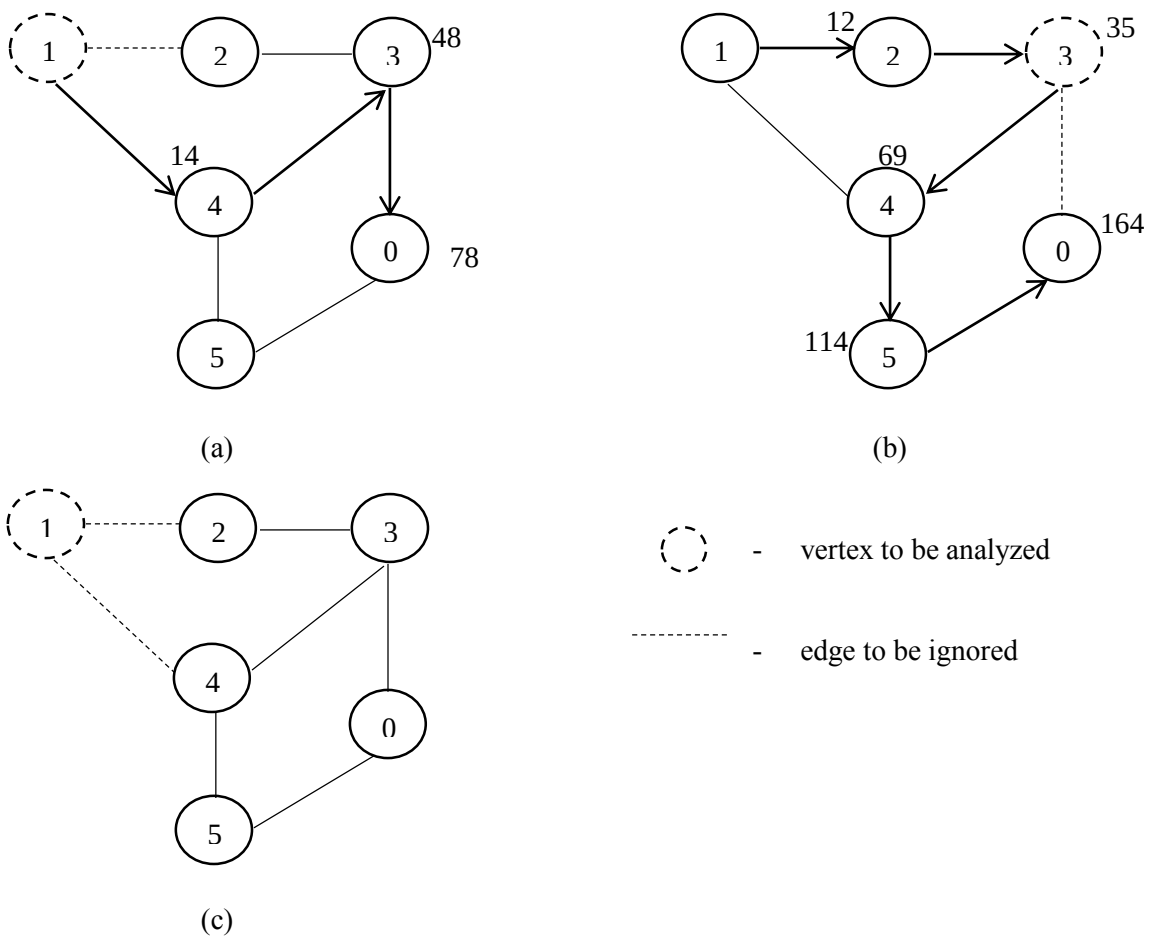


Figure 3.2: best path algorithm path finding to determine k^{th} shortest path

Where (a) & (b) when analyzing vertices in path P^1, (c) when analyzing a vertex in path P^2.

All the vertices in the path P^1 are analyzed. Now, add P^1 into A. Currently the candidate paths that are stored in B are <1, 4, 3, 0> and <1, 2, 3, 4, 5, 0>. The shortest path among them is <1, 4, 3, 0> which is the next shortest path P^2. Remove P^2 from B. The vertices analyzed in P^2 are {1, 4, 3}. When vertex 1 is analyzed, other than the edge (1, 4), also the edge (1, 2) has to be ignored and hence the set E consists of 2 and 4. Note the Figure 3.2 (c) as vertex 1 cannot reach vertex 0. Since P^1 has root path as sub path of P^2 the edge (1, 4) has to be ignored and hence added into set E. So, the algorithm returns null.

Thus all vertices in P^2 are analyzed and a new candidate path <1, 4, 5, 0> that deviates from vertex 4 is found and added into B. Add the path P^2 into A. Then P^3 is the path with shortest weight from B which is the first element in B. P^3 is <1, 4, 5, 0>. No new candidate paths are found after all vertices in P^3 are analyzed and the next shortest path P^4 is <1, 2, 3, 4, 5, 0>. The path P^4 will be analyzed from d (P^4) = 3 until 5 (i.e., 3, 4 and 5 will be analyzed, vertex 1 and 2 are not required to analyze). No new candidate paths are found and B is empty. The KSP Algorithm is stopped after finding four shortest paths even though K (let K = 10) is not reached. If K = 2 then the algorithm stops after the shortest path P^2 found.

3.2.1 Pseudo Code

Let us consider the path P^{j-1} be $<s, v_1^{j-1}, v_2^{j-1} \ldots v_l^{j-1}, t>$. For computing the shortest path P^j, the vertices from d (P^{j-1}) until v_l^{j-1} in the path P^{j-1} have to be analyzed. Let the vertex to be analyzed is v_i^{j-1}, then the vertices in path P^{j-1} from s until v_i^{j-1} is the root path. Now, the vertex v_i^{j-1} is considered as source vertex and a shortest path from v_i^{j-1} to t has to be computed by applying the BP Algorithm, is the spur path. Let d[P^k] be the weight of the k^{th} shortest path. In order to make sure that the same path P^{j-1} would not be determined again, the immediate neighbor of the vertex to be analyzed should not be considered by the BP Algorithm. Hence, the edge (v_i^{j-1}, v_{i+1}^{j-1}) should be removed temporarily from the graph. Instead of removing the edge from the graph, add the neighbor vertex v_{i+1}^{j-1} into E. Moreover, if any of the paths stored in A, has root path as the sub path, then the neighbor vertex of v_i^{j-1} in that path is also added into E as in line 11 to 15 in the algorithm 5.

The lists A and B are implemented by using multimap data structure (see Section 4.1.1) with weight of path as key and the path itself as value. Since the path with shortest distance is required each time when line 6 executes, it is easier with multimap as the elements are always sorted in an order based on its key. Thus the path with shortest distance is at beginning of the multimap container and can access it easily. List E in line 10 is implemented by using the set of vertices which are not considered as neighbors of source vertex which has to be set to infinity. The total path in line 19 is the new candidate path obtained by adding the root path and the spur path.

Moreover, in order to compute the loopless spur path, the vertices in the root path will not be allowed while computing the spur path. Thus the root path is passed as a parameter into the BP Algorithm. If the spur path is not found, then continue with the next vertex to be analyzed. If a spur path is found, add it into B (line 19 and line 20). Once all the vertices in the path P^{j-1} are analyzed, then the path with lowest distance in B is set as the next shortest path P^j and the vertex at which the path P^j is deviated from path P^{j-1} is the deviation node of P^j. Then remove the path from B and insert it into A. Now, the candidates from path P^j will be generated and added into B. And then the shortest path P^{j+1} can be obtained from B.

When the BP Algorithm is computing the shortest loopless path, and if the edge in ignored edges list appears to add in the path, it will skip that edge and look for another possible edges. Thus the BP Algorithm will determine spur path if any path available between source and destination vertices. After that, the list of ignored edges will be cleared and start analyzing next vertex. The algorithm continues until the required number of K shortest paths are determined or there is no path available to analyze in the candidate path list.

```
K shortest path (s, t, K)
Input:
s - source vertex
t - destination vertex
K - number of shortest paths required
Output: P^1, P^2 . . . P^K
 0. Initialize root path, A, B and E as empty set
 1. P^1 = BP(s, t, root path, E);
 2. if(P^1 found)
 3.       add P^1 into B
 4. end if
 5. while B not empty and k<K
 6.    (P^k, d[P^k]) = pop up the first element of B
 7.    Add (P^k, d[P^k]) into A
 8.    for each vertex v in P^k
 9.          root path = all vertices from s to v in P^k
10.          Add w into E                    // where w is the next vertex of v in path P^k
11.          for each path P in A
12.                if (root path is a sub path of P)
13.                      add the neighbor of v in P into E
14.                end if
15.          end for
16.          if (number of neighbors of v > 1)
17.                spur path = BP (v, t, root path, E);
18.                if (spur path != NULL)
19.                     total path = root path + spur path
20.                     add total path into B
21.                end if
22.          end if
23.          clear list E
24.    end for
25.    print path P^k
26. end while
```

Algorithm 5: KSP Algorithm

3.2.2 Correctness proof of the algorithm

Lemma 1: Triangle inequality

If $\delta(s,t)$ is the weight of the shortest path between s and t, then $\delta(s,t) \leq \delta(s,v) + \delta(v,t)$.

Theorem 1: Let $G = (V,E)$ be a graph and $s,t \in V$. If P^j be the j^{th} shortest path from s to t, then weight of the path P^j is always less than or equal to weight of the path $P^j_{s_i}$. $P^j_{s_i}$ is the path from s to t, which deviates at vertex $s_i \in P^j$ to vertex $v \notin P^j$.

Proof:

Let us assume that the j^{th} shortest path P^j is $\langle s, s_1, s_2 \ldots s_l, t\rangle$. Now, consider a vertex $s_i \in \{s, s_1, s_2 \ldots s_l\}$ in P^j, the deviation path from s_i to vertex $v \notin P^j$ and finally reaches t can be expressed as

$$P^j_{s_i} = \min_{(s_i,v)\in E}(\langle s, s_i\rangle + e(s_i,v) + \langle v,t\rangle) \text{ where } \begin{cases} v \notin P^j \\ \langle s, s_i\rangle \text{ is a sub path of } P^j \end{cases}$$

Now from Lemma 1, we know that $\delta(s_i,t) \leq \delta(s_i,v) + \delta(v,t)$. Hence, proved that weight of the path P^j is always less than or equal to weight of the path $P^j_{s_i}$.

Theorem 2: Algorithm 5 is correct.

Proof:

By using the theorem 1, we can prove that the path that is picked from list B in algorithm 5 is the correct shortest path and there would not be any path shorter than that. we can prove it by induction method. If P^j is the j^{th} shortest path and P^{j+1} be the $(j+1)^{th}$ shortest path, then there would not be any path P from s to t such that $P^j \leq P \leq P^{j+1}$.

Basis: Let P^1 be the shortest path from s to t in G (from line 1 in Algorithm 5) and list B holds set of paths that deviates at each vertex $s_i \in P^1 - t$. From theorem 1, it is obvious that weight of the path P^1 is always less than or equal to weight of each path in B. Also one can observe in theorem 1 that the path $P^1_{s_i}$ is obtained as *min* value for all neighbors of s_i. Therefore, the second shortest path P^2 is

$$P^2 = \min_{\forall s_i \in P^1}\left(P^1_{s_i}\right)$$

Induction: Assume that the theorem holds until j^{th} shortest path P^j from s to t and list B contains the set of deviation paths that deviate at vertex $s_i \in \{P^1, P^2 \ldots P^j\}$. Note that $P^1 \leq P^2 \ldots \leq P^j$. From Theorem 1, it is obvious that $P^j \leq P^j_{s_i}$. Therefore, the weight of P^j is less than or equal to weight of any path P in list B. Now, P^{j+1} can be obtained from list B (line 6 in Algorithm 5) as

$$P^{j+1} = \min_{\forall s_i \in \{P^1, P^2 \ldots P^j\}}\left(P^1_{s_i}, P^2_{s_i} \ldots P^j_{s_i}\right)$$

Since, new paths will be generated from P^{j+1} (as shown from line 8 to 24 in Algorithm 5), we know that weight of P^{j+1} is always less than or equal to weight of deviation paths from P^{j+1} (by using Theorem 1). Thus, P^{j+1} is the $(j+1)^{th}$ shortest path from s to t.

Hence proved that there would not be any path P such that $P^{j} \leq P \leq P^{j+1}$. And we can say that the algorithm presented here, will always provide correct results.

Chapter 4

4. Implementation and Evaluation Results

4.1 Implementation

The algorithms described in this report are implemented using the C++ language. The standard template library (STL) is a C++ library of container classes, algorithms and iterators. The STL provides several algorithms and data structures which would reduce the complexity of programs. This section describes the various types of data structures used in the implementation and their importance.

4.1.1 Multimap

Map is an associative container which stores elements in association between a key value and a mapped value. Each key in the map is unique, but a mapped value can be associated with more than one key values. Moreover, the elements in the container can arrange in a specific order. In order to store the multiple keys with same values, a *multimap* is used. In a multimap, more than one element can be associated with the same key value. Internally there is a comparison object which can be used to sort the elements based on its key values.

The multimap container is used to store the generated paths. Here the paths are stored in container as the mapped value and their associative weights as the key value. Multimap allows to have same key values and the ordering of the keys are important because the algorithm always consider the path with minimum length. Thus the ordered multimap container is chosen to fulfill these requirements. Multimap template has various member functions and the functions used here are described as follows:

4.1.1.1 Initializing a Multimap

Constructor member function constructs a multimap container object and initializes its contents. This can be done in three ways: initialize an empty container, construct a container by specifying a range of elements and construct a copy of another multimap container.

```
(1) multimap<key type, value type> variable1;
(2) multimap<key type, value type> variable2 (variable1.begin(), variable1.end());
(3) multimap<key type, value type> variable3 (variable2);
```

4.1.1.2 Insert Elements into Container

Insert member function inserts new elements into the container, which increases the container size by one with each insertion. An element can be inserted in different ways into the container. An element can insert at certain position by specifying the position within the container. Also a range of elements can insert into the container by specifying a range. Here paths are inserted into the container with their respective path weight as key value and the path as mapped value.

```
(1) variable1.insert (value_type& value);                  // insert value to container
(2) variable1.insert (iterator position, value_type& value);    //
(3) variable1.insert [iterator first, iterator last);
```

4.1.1.3 Finding Element in a Container

The *find* function will search the container for an element with key equal to k and returns an iterator pointing to that element. If the key is not found in the container, then it returns an iterator to the end of the container. But, this function returns iterator to the first element even though there were more than one element with same key value. In order to get all the elements with same key value from the container equal range member function can do that.

```
variable1.find (key_type& key);
```

If the container has more than one element with same key value then the find function returns to the first element with certain key in the container. The member function *equal_range* returns a range containing all the elements with the same key value from the container with logarithmic complexity. In order to update d [v] and its associative path in the shortest path algorithm the equal_range function is used.

```
Pair<iterator, iterator> equal_range (key_type key);
```

4.1.2 Set

Set is a container which stores unique elements, and the elements in a set container will be sorted internally using the key comparison function. The elements in a set are unique of type key. Searching an element, removing and inserting elements from the container can take logarithmic time complexity. Because of its effectiveness in inserting and searching the elements set containers are used in this implementation. In order to store the edges to be ignored while computing the spur paths, set container is used.

4.1.2.1 Initializing Set Container

Set container can be initialized by using the *constructor* member function of any defined data type and can be initialized as empty container or a container with elements by specifying a range of elements from another container or a copy of all elements from existing container.

```
Set <key type> variable1;                    // empty container

Set <key type> variable2 (iterator position1, iterator position2);   // container with range from
                                                                      position1 to position2

Set <key type> variable3 (variable2);                    // copy all contents from variable2
```

4.1.2.2 Searching Elements

In order to search the container for an element *find* operation is used. This searches for an element in the container and returns an iterator pointing to the position of element if found. Otherwise it returns an iterator pointing to the end of the container which indicates that the element is not found in the container.

```
Variable1.find (value);
```

4.1.2.3 Inserting Elements

Inserting an element modifies the container as the elements in a set are ordered. Inserts the element only if the element is not in the container as elements in a set are unique. Inserting an element takes logarithmic time in size of the container.

```
Variable1.insert (value);

Variable1.insert (iterator position1, iterator position2);
```

4.1.3 Vector

Vector is a sequence container which can be represented as a dynamic array of variables or objects. Vectors can store multiple elements of a defined data type. Vectors are very much efficient to access the elements when compared with other available sequence containers. Elements will be added and removed from end of the container. The elements in a vector can be accessed directly by its index. The indexing of elements in a container starts at zero. Thus the first element in a container is at the position zero and can be accessed by using the [] operator as in arrays. Because of its efficient accessing of elements in a container and adding and removing from end of the container, vectors are used in this implementation. Vector also has several predefined member functions provided by STL template classes. Vector of vectors will give a two dimensional array. Let us see briefly some of the member functions that are used in this implementation.

4.1.3.1 Initializing a Vector

The member function *constructor* constructs a vector container of any defined data type. This could be done in different ways: can construct an empty container, can construct a container by specifying a range of elements from another container and can construct a copy of all elements from another container.

```
Vector<type> variable1;

Vector<type> variable2 (variable1);

Vector<type> variable3 (variable2.begin, variable2.end);
```

4.1.3.2 Accessing Elements of a Vector

In order to access the element in a position *n* in the container, the [] operator member function will return a reference to that element in position *n*. Similarly, *at* can be used to access element in position *n*. The element at the end of a vector can be accessed directly by the member function *back* which returns reference to the last element of the container.

```
variable1 [n];

variable1.at (n);

variable1.back ();          // returns the content of the last element
```

4.1.3.3 Adding and Removing Elements

With the help of *push_back* member function, an element can be added at the end of the vector container. And can remove the last added element from the vector by using the member function *pop_back*. Adding a new element effectively increases the size of the container by one and removing decreases the size by one. Both the functions are effective and of constant time complexity.

```
Variable1.push_back (value);     // adds value to end of the vector
Variable1.pop_back ();           // removes the last element from the vector
```

Following member functions represents all the three template classes discussed above.

4.1.3.4 Begin and End

The function *begin* returns an iterator pointing to the first element of the container. If the container is empty, then it returns an iterator pointing to end of the container. Then the function *end* returns an iterator pointing to the last element of the container. The functions begin and end could be used to iterate through the container from the first element to the last element.

```
variable1.begin ();
variable1.end ();
```

4.1.3.5 Erase and Clear

The *erase* function erase the elements from the container and reduce the container size by the number of elements erased. This function is able to erase an element at certain position, erase specific element and can erase given range of elements. Whereas with function *clear* removes all the elements from the container and the size of the container becomes zero. This function doesn't require any parameters.

```
(1) variable1.erase (key_type& k);                  // erase element k from container
(2) variable1.erase (iterator position);            // erase element at position from container
(3) variable1.erase [iterator first, iterator last); // erase elements from range first to last
```

```
variable1.clear ();
```

4.1.3.6 Size and Empty

If the container is *empty*, that is the size of the container is zero then this function returns true. And returns false, if the container is not empty or its size greater than zero. Thus the function is used to test whether the container is empty or not. *Size* returns the number of elements in the container. It is often required to know the size of the container and thus the function size was used.

```
!variable1.empty ();      // if variable1 not empty do
```

```
variable1.size ();
```

4.2 Evaluation Results

This section describes the evaluation of K shortest path algorithm and the experimental results will be shown where the run time performance of PM's implementation (named NIYA) and the current implementation, KSP Algorithm (named RIYA) is compared. The algorithm is evaluated on two different datasets and the tests are done on an Intel(R) Core(TM) i5 CPU, 2.67 GHz and 4 GB of RAM memory. All the algorithms are implemented in C++ language with the Standard Template Library (STL) and run on Ubuntu Operating system.

4.2.1 Analysis of Time Complexity

The worst case time complexity of an algorithm determines the maximum running time of the algorithm based on the input size in a worst case scenario. Thus the worst case time complexity of the KSP Algorithm implemented in this thesis work has to be estimated based on the given input. The input for the K shortest path finding is based on the number vertices, *n*, number of edges, *m* and the number of K shortest paths required, K. In order to determine the K number of shortest paths the algorithm always applies the shortest path algorithm and thus the running time of shortest path algorithm is an important aspect.

The shortest path algorithm described in section 3.1.1 uses the multimap STL container to store the shortest paths from source to vertex *v*, which is visited but not yet determined. This container holds for each vertex *u* in V, the shortest path from *s* to *u* and the vertex *u* is not yet determined. The while loop at line 9 continues until the container A becomes empty or breaks when the shortest path from given source to destination vertex is determined. The container will be empty when every vertex in the graph is said to be determined. Thus the while loops runs *O* (*n*) times as there are n vertices in the graph. Each time when while loop is started, the shortest path from the container A has to be removed as shown in line 10 and this could be done in constant time, as the elements are sorted in an order in a multimap and the path with minimum length will always be at beginning of the container.

Then for each adjacent vertex *w* of *v* from line 17 to line 32, a shortest path from *s* to *w* could be computed and has to be inserted into the container A with its associative weight of the path. This loops takes *O* (*m*) times. The *Insert* operation in line 28 has logarithmic in size of the container as the elements will be sorted in an order and thus each insertion takes *O* (*log n*) since the container has at most *n* elements in it. And if the vertex *w* is already visited before, then the path has to be replaced with new shortest path from *s* to *w* which passes through vertex *v*, in line 25. The *Equal_range* operation finds the path from the multimap container and the appropriate element has to be replaced. This operation requires *O* (*log n*) time. Since both the operations *Insert* and *Equal_range* are required to perform at most once for each edge, the algorithm requires a running time of *O* (*m log n*) in worst case scenario for computing a shortest path.

Then consider the algorithm in section 3.2.1, the while condition at line 5 runs at most of K times to determine K shortest paths. And the for loop inside while condition from line 8 to line 24 runs as long as the size of the path - 1. Thus for loop requires to run maximum of *n* times and the shortest path algorithm will be applied inside this for loop at line 17, is of *O* (*m log n*) complexity. Since the shortest path algorithm is applied every time, hence the complexity of the algorithm in worst-case time is *O* (K *n* (*m log n*)).

4.2.2 Performance Analysis

The practical run time performance of PM's implementation, NIYA and the k^{th} SP Algorithm, RIYA is compared. The tests are run on Ubuntu 12.04 LTS operating system and running time for determining K shortest paths have been computed. The source and destination vertices are generated randomly by using

rand function and the K shortest loopless paths have been determined for those vertices. The tests are done for K = 1 to 1000 loopless paths have been determined in a graph of 4475 and 1133 vertices. And the densities of 1 and 10, whereas density of a graph could be determined by d = *m*/*n*.

The graph in figure 4.1 is plotted with K value in x-axis varying from 1 to 1000, for a random source and destination vertices. The graph clearly indicates that the current implementation can determine any number of shortest paths with in shorter time than the PM implementation. Almost the k^{th} SP Algorithm determines the K shortest paths in half time when compared with the other implementation. However, the graph is plotted for one random source and destination vertices. In order to have better idea on how the run time behavior would be for different random sets of source and destination vertices, the graph in figure 4.2 is plotted. This graph shows the run time performance of both the implementations, where different sets of source and destination vertices are considered.

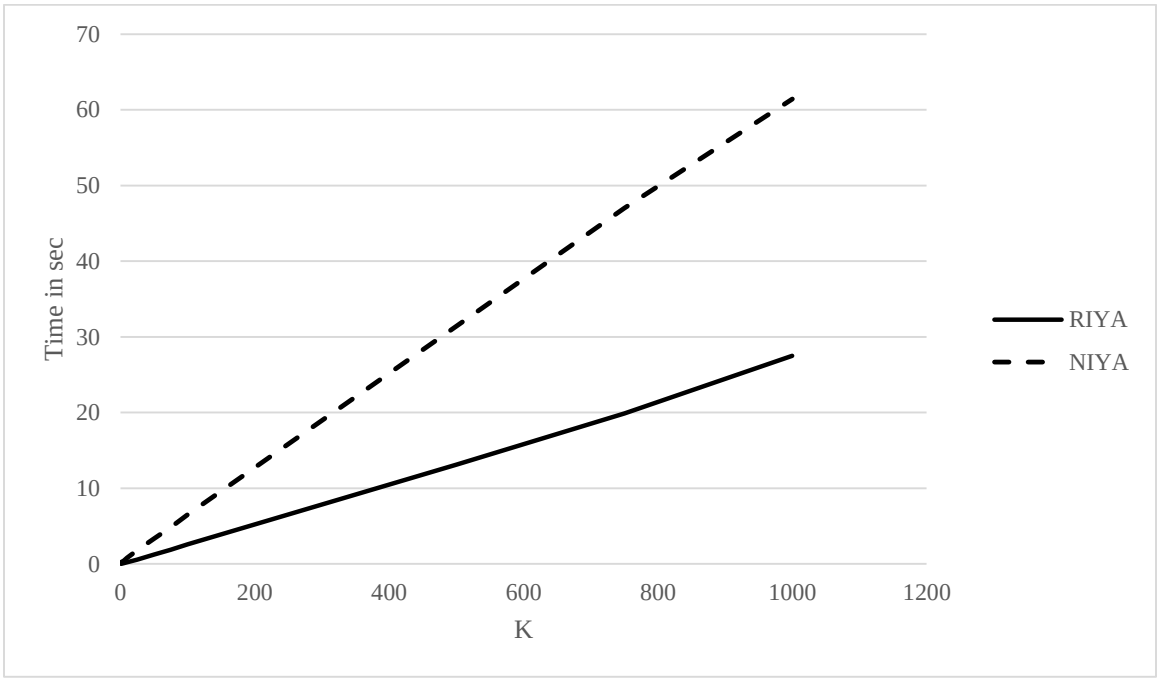


Figure 4.1: Graph with 4475 vertices and 4652 edges, K varies from 1 until 1000

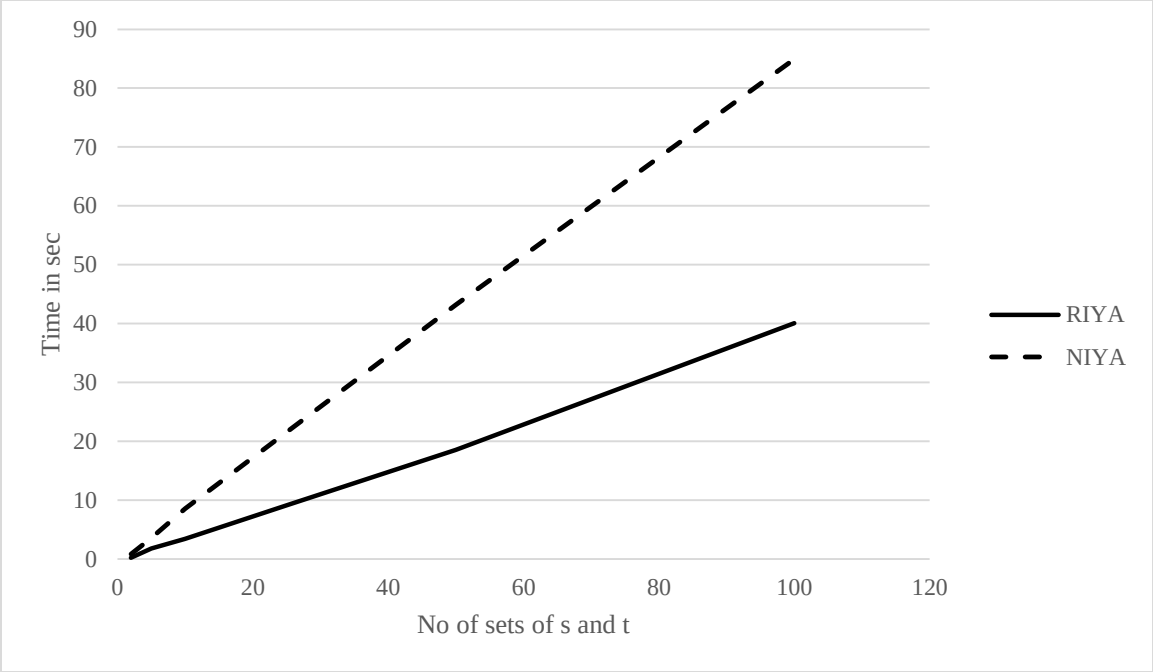
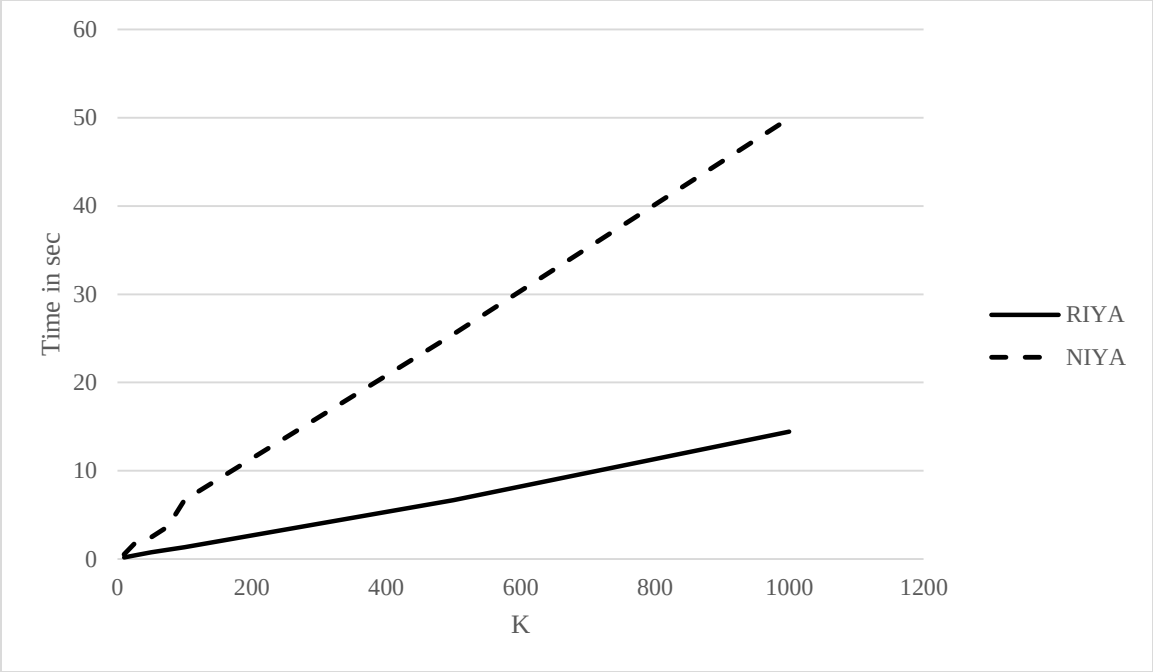


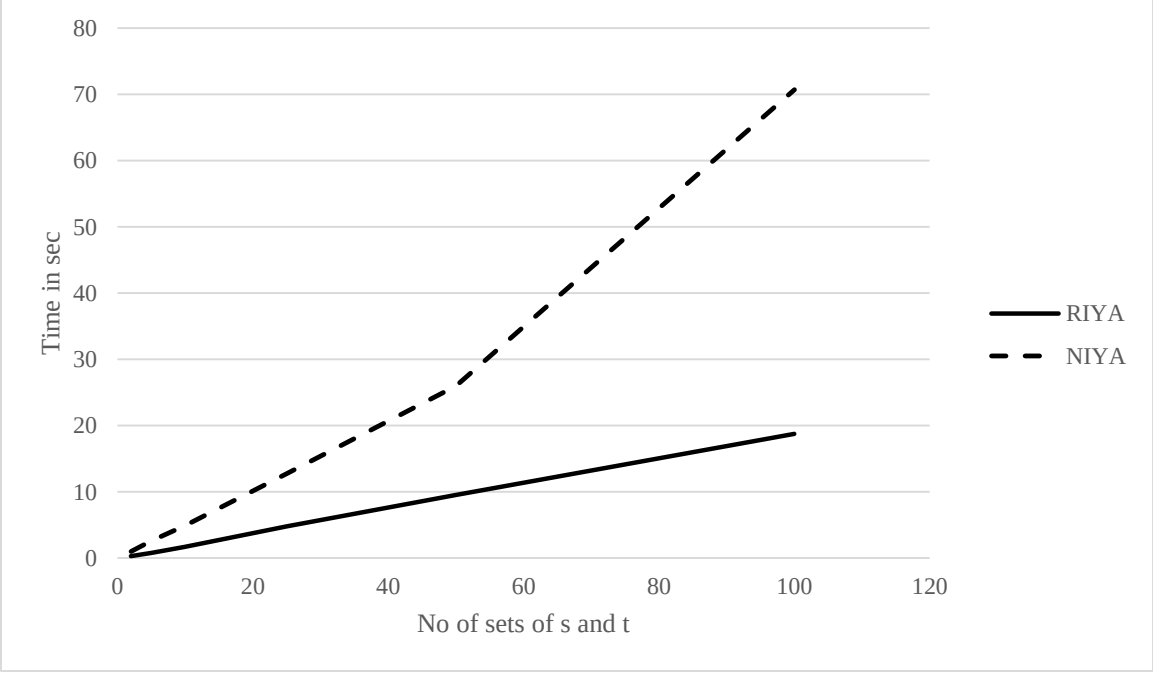
Figure 4.2: Graph with 4475 nodes and 4652 edges, varying source and destination vertices and constant K =10.

The number of source and destination vertices are varying in the sense that in the above figure 4.2, the running times are computed for 2 different sets of source and destination vertices with always K as 10. That is the algorithm first computes 10 shortest paths for a random set of source and destination and then computes 10 shortest paths for another random set of source and destination vertices. As the density of the graph (d = m/n) is one for the above plotted graphs, a graph with density as 10 is also considered to analyze the run time performance of both the implementations below in figure 4.3 (a) and (b). Even though the density of graph in figure 4.3 is more, the runtime performance results to be same as before. Both the graphs in figure 4.1 and 4.3 (a) clearly suggests that the current implementation, RIYA completely outperforms the PM implementation, NIYA.

And the table 4.1 shows the CPU running times of both the implementations on random graphs. Table shows the running times for K number of shortest paths and different number of sets of source and destination vertices. All the running times are measured and taken the average time into consideration. And the times are rounded to three decimal numbers.



(a)



(b)

Figure 4.3: Graph with 1133 vertices and 10902 edges.

(a) K varies from 1 until 1000 and (b) s and t varies from 1 to 100, where s and t are source and destination vertices.

K	Graph (V, E) with n = 4475 and m= 4652 density = 1		Graph (V, E) with n = 1133 and m = 10902 density = 10	
	RIYA	NIYA	RIYA	NIYA
K = 10	0.224	0.813	0.189	0.526
K = 100	2.61	6.523	1.348	6.709
K = 500	13.104	31.394	6.674	25.46
K = 1000	27.514	61.432	14.43	49.945
No. of Sets of s and t				
10	3.462	8.595	1.714	4.901
50	18.546	43.209	9.518	25.953
100	40.053	84.945	18.756	70.691

Table 4-1: Running times in seconds rounded to 3 decimal for random networks

Chapter 5

5. Conclusion and Future Work

5.1 Conclusion

In this thesis work, we introduced a new KSP Algorithm for determining the K shortest paths between two vertices in a graph. The KSP Algorithm is an improvement based on the straight-forward approach of Yen's algorithm. Instead of modifying the graph by removing vertices and edges, the KSP Algorithm uses a different approach and computes the K shortest paths. Moreover, in order to compute the best shortest path between two vertices, the KSP Algorithm uses the BP Algorithm. The Dijkstra's algorithm is modified as BP Algorithm such that the modification of graph is not required. Through the computational experiments over different datasets, we shown that the KSP Algorithm performs better than the PM implementation and the efficiency can be improved.

5.2 Future Work

In the case of future work, the worst-case analysis of the algorithm can be reduced by implementing the BP Algorithm with Fibonacci heap data structure. Moreover, the PM implementation can be implemented without modifying the graph when a vertex is being analyzed, as described in the KSP algorithm presented in this report.

References

[1] H. Walter and P. Richard, "A method for the solution of the Nth best path," *J. Assoc. Comput. Mach.*, vol. 6, pp. 506--514, 1959.

[2] D. Eppstein, "Finding the k shortest paths," *SIAM J. Comput*, vol. 28, no. 2, pp. 652-673, 1998.

[3] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, vol. 17, pp. 712--716, 1971.

[4] M. Pascoal and E. Martins, "A new implementation of Yen's ranking loopless paths algorithm," *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, p. 13, 2003.