A COMPARISON BETWEEN LABEL-SETTING AND LABEL-CORRECTING ALGORITHMS FOR COMPUTING ONE-TO-ONE SHORTEST PATHS

A project report submitted in partial fulfilment of the requirements of the course-

FOUNDATIONS OF ARTIFICIAL INTELLIGENCE - 19AI602

Submitted by:

Golkonda Ramananjali Mounica
(AM.EN.P2ARI20028)

Narendra Singh Bisht
(AM.EN.P2ARI20043)

Urvi Sharma
(AM.EN.P2ARI20022)

Under the supervision of:

Dr. Georg Gutjahr

SCHOOL OF ENGINEERING,

AMRITA VISHWA VIDYAPEETHAM,

AMRITAPURI - 690525

September-December (2020)

DECLARATION

We the members of Group- 1 Golkonda Ramananjali Mounica (AM.EN.P2ARI20028), Narendra Singh Bisht (AM.EN.P2ARI20043), and Urvi Sharma (AM.EN.P2ARI20022) hereby declare that this project entitled "A comparison between Label Setting and Label Correcting Algorithms" is a record of the original work done by us, and that this is the bonafide project report for the course 19AI602, and that this work has not formed the basis for any degree/ diploma/associationship/fellowship or similar awards to any candidate in any university to the best of our knowledge.

ACKNOWLEDGEMENT

First, we offer our sincere gratitude to the God almighty whose abundant grace and mercy enable the successful completion of my minor thesis. We would like to express our profound gratitude to all the people who had inspired and motivated us to do the work.

We would also like to express our sincere thanks to our professor Dr. Georg Gutjahr, for his guidance, support, encouragement, and patience throughout the entire period of our study as well as the write-up of this report.

We wish to express my sincere thanks to our Head of the Department for giving us the support and encouragement that was necessary for the completion of this minor project.

We would also like to express our gratitude to all the faculty members and staff who have helped in our minor thesis project in one way or other.

We would also like to thank our Beloved Chancellor, Sri (Dr.) Mata Amritanandamayi Devi for providing us with guidance and a friendly environment to work.

Lastly, we would like to thank our family and friends who have constantly supported us throughout the pandemic situation with our online classes and project work.

ABSTRACT

Label Setting algorithms in general are considered more efficient compared to Label correcting algorithms for finding shortest paths. In the year 1998 F. *Benjamin Zhan* and *Charles E. Noon* did a study of several different types of shortest path algorithms and implemented them on 10 large real-road networks for finding one to one shortest path. The results of their study claimed that in certain circumstances the label correcting algorithms perform better than label setting algorithms. The selected label-setting algorithm is Dijkstra's algorithm implemented with approximate buckets (DIKBA) and the selected label-correcting algorithm is Pallottino's graph growth algorithm implemented with two queues (TWO-Q).

In this project our aim is to understand and implement the two labelling algorithms.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

With the fast-moving life, development of technology and advancement of Geographic Information Systems (GIS) technology and the availability of high-quality network data, network and transportation shortest path applications are getting importance. Shortest path algorithms have many applications. Mapping software like Google or Apple maps makes use of shortest path algorithms. They are also important for road network, operations, and logistics research. Shortest path algorithms are also very important for computer networks, like the Internet. The computation of shortest paths between different source and destination nodes on a network is a central and computationally-intensive task in many transportation and network analysis problems.

Shortest path problems are commonly solved using iterative labeling algorithms, of which two major types are label-setting and label-correcting. Label-setting algorithms, such as Dijkstra's algorithm, have a fixed number of iterations, but the amount of time required by each iteration depends on the data structures used. In contrast, label-correcting algorithms, such as the Bellman Ford-Moore algorithm, generally take a constant amount of time per iteration, but vary in the number of iterations needed to complete the shortest path computation. Some label-correcting algorithms have polynomial bounds on the worst-case behavior; others do not. In either case, the best label-setting algorithms have better worst-case theoretical behavior than that of any label correcting algorithms. In practice, however, results of the paper by Zhan and Noon indicates the label-correcting algorithms often have the same or better performance than the label-setting algorithms.

1.2 Label-Setting Algorithms

Label-setting algorithms remove from the list, the node with the smallest distance label during each iteration of the algorithm. These algorithms greedily determine the shortest path from the source to every other node. Because the arc costs are non-negative, once a node is removed from the list, it will never re-enter. Thus, the algorithm ends after n iterations, when the list is empty.

The time complexity of the algorithm depends on how the list is stored and/or how the minimum label is found. The basic algorithm is attributed to Dijkstra. In this algorithm, finding this label requires O (n) time for a total of O (n^2) time for n iterations. Regardless of the data structures or search techniques, all m arcs are examined once. Hence the total worst case time complexity of the algorithm is O (n^2 + m) = O (n^2).

1.3 Label-Correcting Algorithms

Label-correcting algorithms do not permanently set distance labels; all labels are not considered correct until the algorithm ends. During each iteration, the node removed from the list does not necessarily have the minimum label. Therefore, a removed node may re-enter the list at a later time. Since the labels are not "set" as in the label-setting algorithms, the distance labels are not guaranteed to be correct after n steps. Distance labels are all correct when for every edge (i; j), dl[j] <= dl[i] + c_{ij}. This inequality is the optimality condition.

Therefore, as long as there are edges for which this inequality does not hold, the distance labels are not correct and the algorithm continues.

In terms of worst-case performance, the complexity bounds depend on the method for removing and adding nodes to the list. If the nodes are selected in a first-in first-out (FIFO) order from the list, the selection order is the same as breadth-first search of the network. This algorithm is equivalent to that of Bellman-Ford-Moore, which is attributed with the first polynomial bounded label-correcting algorithm, which has a worst-case behavior of O(nm).

1.4 One to One Shortest Path

Depending on the nature of the problem, it is sometimes necessary to compute shortest paths from a source node to every other node (one-to-all) or from every node to every other node (all-to-all) on a network. Sometimes, it is only necessary to compute shortest paths from a source node to a destination node (one-to-one) or from a source node to some destination nodes (one-to-some) on a network.

1.5 Study conducted by Zhan and Noon

Empirical evaluation of shortest path algorithms has been an extensively researched topic in the literature of operations research and management science. Among these evaluations, the most recent comprehensive evaluations are the ones conducted by Cherkassky et al., and by Zhan and Noon.

In both studies, the algorithms were categorized into two groups: label-setting and label-correcting. Both groups of algorithms are iterative and both employ the labeling method in computing one-to-all shortest paths (discussed in Section 2). The two groups of algorithms differ, however, in the ways in which they update the estimate (i.e., upper bound) of the shortest path distance associated with each node at each iteration and in the ways in which they converge to the final optimal one-to-all shortest paths.

In label-setting algorithms, the final optimal shortest path distance from the source node to the destination node is determined once the destination node is scanned and permanently labeled. Hence, if it is only necessary to compute a one-to-one shortest path, then a label-setting algorithm can be terminated as soon as the destination node is scanned, and there is no need to exhaust all nodes on the entire network.

In contrast, a label-correcting algorithm treats the shortest path distance estimates of all nodes as temporary and converges to the final one-to-all optimal shortest path distances at its final step when the shortest paths from the source node to all other nodes are determined. This characteristic of label-correcting algorithms implies there is no difference in the computational time used for computing a one-to-one shortest path and one-to-all shortest paths.

If a label-setting code and a label-correcting code had equivalent performance in computing one-to-all shortest paths, then clearly the label-setting code would be preferred in computing a one-to-one shortest path. Such equivalence, however, was not observed in the computational study of Zhan and Noon (1998). In that study, the best-performing label setting algorithm was observed to be over 50% slower than the best-performing label-correcting algorithm for computing one-to-all shortest paths on real road networks. It was concluded that Dijkstra's algorithm implemented with approximate buckets (DIKBA) is the best-performing algorithm in the group of label-setting algorithms, and Pallottino's graph growth algorithm implemented with two queues (TWO-Q) is the best-performing algorithm from the group of label-correcting algorithms.

CHAPTER 2

OVERVIEW OF THE ALGORITHMS

For this study the selected label-setting algorithm is Dijkstra's algorithm implemented with approximate buckets (DIKBA) and the selected label-correcting algorithm is Pallottino's graph growth algorithm implemented with two queues (TWO-Q).

2.1 Dijkstra Algorithm

One algorithm for finding the shortest path from a starting node to a target node in a weighted graph is Dijkstra's algorithm.

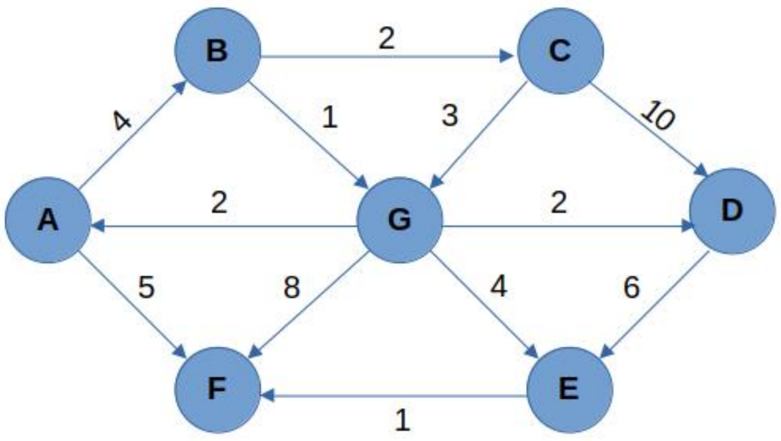


Figure 2.1.1 Graph

- Dijkstra's algorithm, published in 1959 and named after its creator Dutch computer scientist Edsger Dijkstra, can be applied on a weighted graph.
- The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.
- The graph can either be directed or undirected.
- One stipulation to using the algorithm is that the graph needs to have a nonnegative weight on every edge.
- Dijkstra's shortest path algorithm runs in O (E log V) time when implemented with adjacency list representation and O (V^2) when implemented with adjacency matrix representation.

2.1.1 Original Implementation

In the original Dijkstra algorithm, a labeled node with the minimum distance label is selected for scanning. A result of this selection criteria is that, once a node is scanned, it becomes permanently labeled and is never again selected for scanning.

The set of labeled nodes is managed as an unordered list, which is clearly a bottleneck since all labeled nodes must be checked during an iteration in order to select the node with the minimum distance label.

A natural enhancement of the original Dijkstra algorithm is to maintain the labeled nodes in a data structure such that the nodes are either exactly or approximately sorted according to distance label.

One such data structure is the bucket.

2.1.2 Dijkstra Algorithm with approximate buckets

In the approximate bucket implementation of the Dijkstra algorithm (DIKBA), a bucket "i" contains those labeled nodes whose distance labels are within the range of

$[i * \beta, (i + 1) * \beta - 1]$,

where β is a chosen constant.

The bucket structure is considered approximate since the values of the distance labels in a bucket are not exactly the same, but are within a certain range.

Nodes within each bucket are maintained in a FIFO ordered list.

Algorithm DIKBA requires a maximum of $(C/\beta) + 1$ buckets and has worst case complexity of $O(m\beta + n(\beta + C/\beta))$, where C is the length of the longest arc.

- When β is chosen as 1, the buckets are numbered with a range from 0, 1, 2, …, nC, where n is the number of nodes in the graph and C is the largest length of an arc, so nC is the upper bound of any temporary labelled node’s distance.

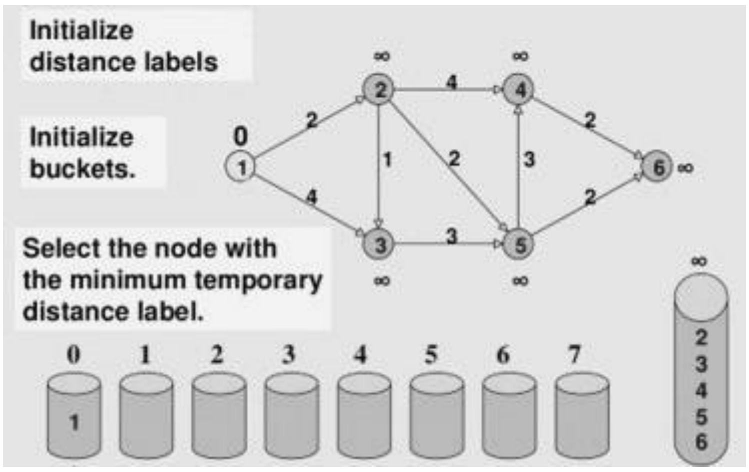

Initialize distance labels
Initialize buckets.
Select the node with the minimum temporary distance label.
0
∞
2
4
1
2
3
3
2
2
4
0 1 2 3 4 5 6 7
1
2 3 4 5 6

Figure 2.1.2 DIKBA

Image Source: www.ocw.mit.edu

- Each bucket stores all temporary labelled nodes with a distance that equals to the bucket's number.
- This enables us to scan the buckets in increasing order from 0 and up until we find the first non-empty bucket when we select the next node to scan.
- All temporary labelled nodes that are updated during a scan are moved to the bucket that corresponds to its distance.
- Note that we don't need to begin from bucket 0 when we continue with the next selection of node to scan, we can continue from where we found the last one since there cannot be any nodes with lower distance than that.

The total time of distance updates in this algorithm is O (m) and the number of scans for node selection is O (nC), so the running time is O (m + nC). This is not a very good upper bound if C is large and could easily become worse than the original implementation of Dijkstra's algorithm, which runs in O (n^2).
In many practical cases C is modest in size, however, and the running time in reality is much better than its worst-case scenario.

2.2 Pallotino's graph growth algorithm

2.2.1 A Brief History - Deque Algorithm

A list which combines the properties of both the queue and the stack is the deque, or double-ended queue. A deque is a list in which addition and deletion are possible at either end. A deque is used in the well-known D'Esopo-Pape algorithm, formally called as deque. In the deque Q used in deque additions are made at both ends, while deletions are made at the head. This deque can be interpreted as a queue Q' and a stack Q" connected in series, in such a way the tail of the stack points to the head of the queue

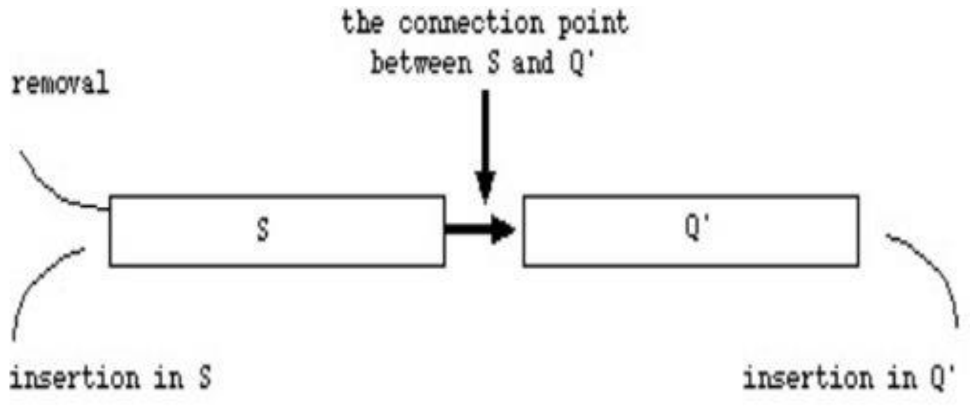


Figure 2.2.1 Dequeue Structure

Image source: Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures: JGIDA vol.1, no.1, pp. 69-82

- The first time a node is to be inserted into Q, it is added to Q' at the tail. This corresponds to a breadth-first search strategy.
- When later on, the same node, after being removed from Q, again becomes a candidate for insertion into Q, it is added to Q" at the head: this means that the node is now processed according to a depth-first search strategy.
- The nodes are removed from the head of Q", if Q" is empty then the first element of Q' is removed.

The rationale for using this rather peculiar type of list is that, every time a label d(u) is updated (decreased), except the first time, it is worth trying to decrease the labels of the successors of u in the current tree as well: this is the aim of the depth-first search phase.

The computational complexity is O (n*2^n), which is huge for both sparse and dense networks.

2.2.2 Two Queue Algorithm

The graph growth algorithm implemented with two queues (TQQ) was introduced by Pallottino in 1984. In TWO-Q, the labeled nodes are stored in one of two ordered lists (or queues), a node becomes labeled if its distance label is changed during a scanning operation.

In the Implementation TQQ, nodes are partitioned into two sets:

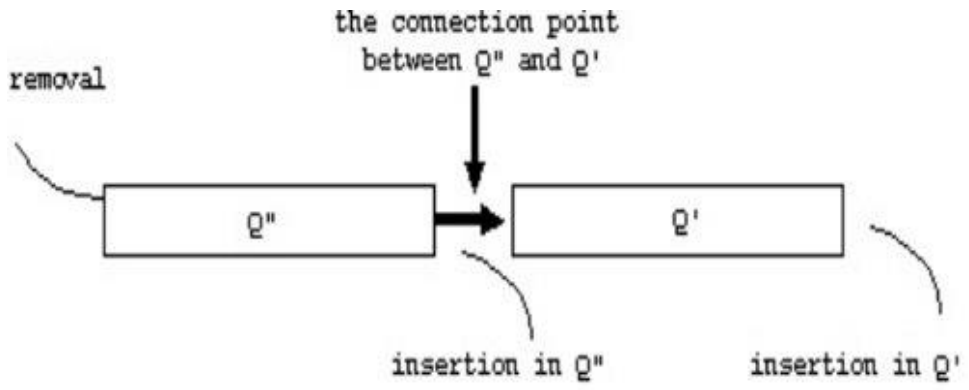


Figure 2.2.2 Two Queue Structure

Image source: Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures: JGIDA vol.1, no.1, pp. 69-82

- The unreached nodes which have never entered Q, i.e., nodes whose distance labels are still infinite. These nodes are inserted into Q'
- Labeled nodes, i.e., the nodes that have passed through Q at least once, and the nodes whose current distance labels have already been used. These nodes are inserted into Q"

The nodes are not considered permanently labeled until the entire set of labeled nodes is empty. Computational time complexity for two-queue algorithm is O (m. n^2), which is reduced in comparison to that of deque algorithm. Two Q, in most of the experimentations by Zhan and Noon study has always proved to be almost as good as L-deque, without the risk of bad behavior in pathological cases, it can be strongly recommended to risk-averse users.

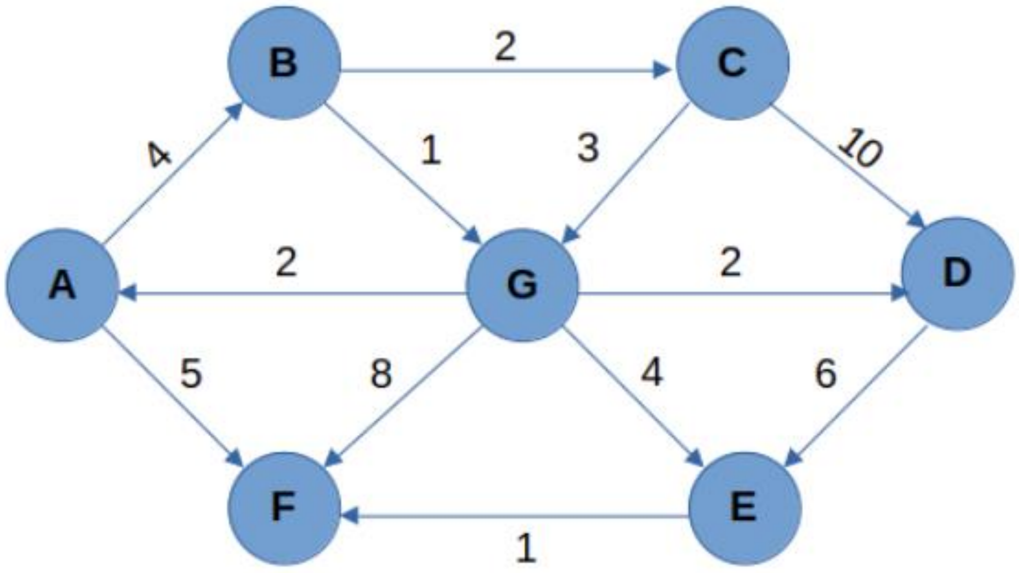
IMPLEMENTATION

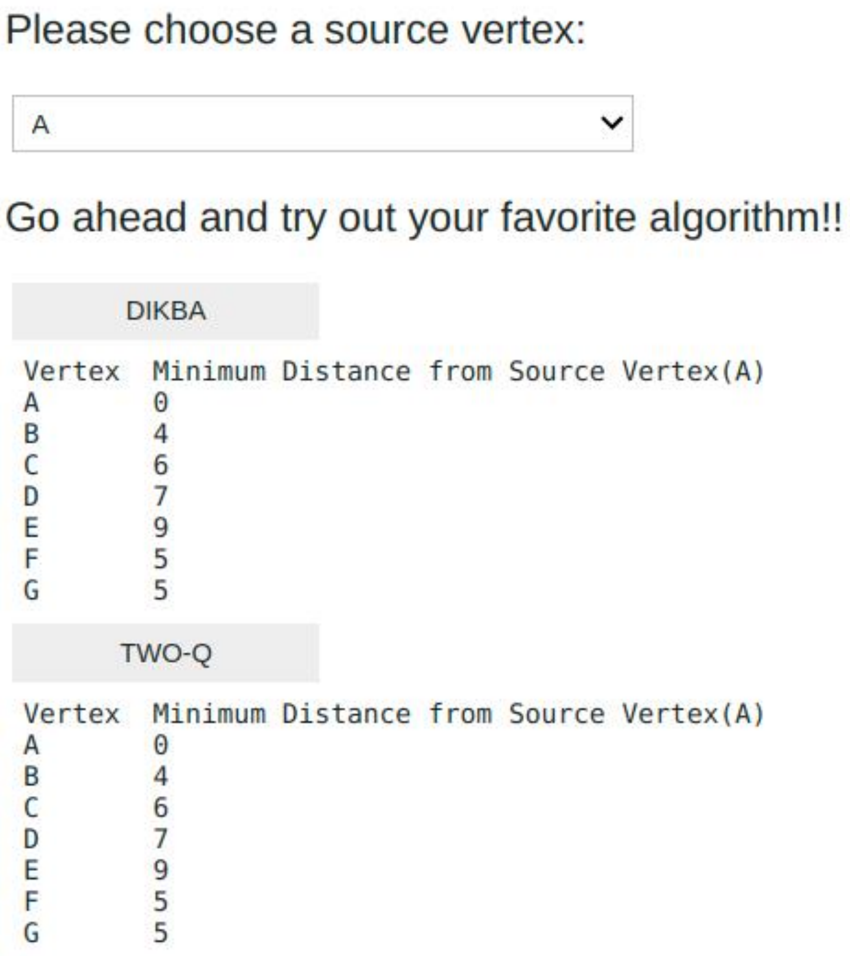
The implementation involved three classes, Graph, Two_Queue and Search.

- *Graph:* It implements an init method to initialize the adjacency matrix and the number of vertices in the graph.
- *Two_Queue*: It implements all the basic operations of Queue data structure. These methods are created explicitly for both the queues Q1 and Q2.
- *Search:* It implements the main logic of the two algorithms, defined as two individual methods.

```
def dijkstra_using_buckets(self, src):
   source_vertex = src
# Example: "A" -> 0
   source_vertex_index = ord(src) - 65
   min_dist_src_vtx = \
   [sys.maxsize] * self.network.no_of_vtx
   min_dist_src_vtx[source_vertex_index] = 0
   shortest_path_tree_status = \
      [False] *self.network.no_of_vtx
 # Maximum distance between any two node can be at max w(V – 1)
 # w is maximum edge weight
 # we can have at max V-1 edges between two vertices
   buckets = \
 [[] for bucket in range(max(map(max,self.network.graph))*(self.network.no_of_vtx - 1))]

 for _ in range(self.network.no_of_vtx):
      for vertex_index, vertex_distance_from_source in enumerate(min_dist_src_vtx):
                if ((shortest_path_tree_status[vertex_index] == False)
                    and (vertex_distance_from_source < sys.maxsize)
                    and vertex_index not in buckets[vertex_distance_from_source]):
                    buckets[vertex_distance_from_source].append(vertex_index)

  # From the set of vertices not yet processed,
  # Pick the vertex having the minimum distance from the source node
  # NOTE: curr_min_dist_vtx_idx is always equal to source_vertex_index in the first
  iteration
            for bucket in buckets:
                if len(bucket) != 0:
                    curr_min_dist_vtx_idx = bucket.pop(0)
                    break
  # Update the status minimum distance vertex
  # in the shortest path tree

            shortest_path_tree_status[curr_min_dist_vtx_idx] = True

  # Update distance values of the adjacent vertices of the picked vertex
  # only if the current distance is greater than new distance
  # and the vertex in not in the shortest path tree

            for v in range(self.network.no_of_vtx):
                if (self.network.graph[curr_min_dist_vtx_idx][v] > 0
                    and shortest_path_tree_status[v] == False
                    and min_dist_src_vtx[v] > (
                    min_dist_src_vtx[curr_min_dist_vtx_idx]
                    + self.network.graph[curr_min_dist_vtx_idx][v])
                   ):
                    min_dist_src_vtx[v] = (
                    min_dist_src_vtx[curr_min_dist_vtx_idx]
                    + self.network.graph[curr_min_dist_vtx_idx][v]
                    )
        self.print_solution(source_vertex, min_dist_src_vtx)


 def two_queues(self,src):
        source_vertex = src

        # Example: "A" -> 0
        source_vertex_index = ord(src) - 65

        min_dist_src_vtx = [sys.maxsize] * self.network.no_of_vtx
        min_dist_src_vtx[source_vertex_index] = 0

        Q = Two_Queue()
        Q.insert_Q2(source_vertex_index)


        while not Q.isEmpty():
            if not Q.isEmpty_Q2():
                temp_vtx = Q.Q2[0]
                Q.remove_Q2()
            else:
                temp_vtx = Q.Q1[0]
                Q.remove_Q1()


            for neighb in range(self.network.no_of_vtx):

                if self.network.graph[temp_vtx]\ [neighb] > 0 and
                min_dist_src_vtx[neighb] == sys.maxsize:
```

```
                        Q.insert_Q1(neighb)
                        min_dist_src_vtx[neighb] = \
                        min_dist_src_vtx[temp_vtx] + self.network.graph[temp_vtx][neighb]

                    elif self.network.graph[temp_vtx]\[neighb] > 0 and
(min_dist_src_vtx[temp_vtx] + self.network.graph[temp_vtx]\[neighb] <
min_dist_src_vtx[neighb]):

                        Q.insert_Q2(neighb)
                        min_dist_src_vtx[neighb] = \
                        min_dist_src_vtx[temp_vtx] + self.network.graph[temp_vtx][neighb]
                        self.print_solution(source_vertex, min_dist_src_vtx)



adjacency_matrix = [[0, 4, 0, 0, 0, 5, 0],
                    [0, 0, 2, 0, 0, 0, 1],
                    [0, 0, 0, 10, 0, 0, 3],
                    [0, 0, 0, 0, 6, 0, 0],
                    [0, 0, 0, 0, 0, 1, 0],
                    [0, 0, 0, 0, 0, 0, 0],
                    [2, 0, 0, 2, 4, 8, 0],
                   ];
network = Graph(adjacency_matrix)
```




Please choose a source vertex:

A

Go ahead and try out your favorite algorithm!!

DIKBA

```
Vertex  Minimum Distance from Source Vertex(A)
A       0
B       4
C       6
D       7
E       9
F       5
G       5
```

TWO-Q

```
Vertex  Minimum Distance from Source Vertex(A)
A       0
B       4
C       6
D       7
E       9
F       5
G       5
```

Figure 3.1 Output 😜

CHAPTER 4

CONCLUSION

In this project, we learned about the different types of shortest paths (one-to-all, one-to-one, etc.), and the key differences between Label Setting and Label Correcting Algorithms. Also, about the best performing Label Setting Algorithm i.e., Dijkstra Algorithm (implemented with approximate buckets) and the best Label Correcting Algorithm i.e., Pallottino’s Graph Growth Algorithm (implemented with TWO Queues).

Additionally, after implementing the above two algorithms, we have developed a better understanding about possible optimizations that can be achieved in labeling algorithms by using different data structures.

In the studies conducted by Zhan and Noon, it was recommended that:

- For sparse networks, it's better to use Label Correcting Algorithms, while Label Setting is more optimal for dense Networks.
- When there is no prior knowledge about the network-based shortest path distance between a source node and a destination node relative to the longest shortest path distance on a shortest path tree rooted at the source node, use TWO-Q to compute one-to-one (some) shortest paths on a network.
- DIKBA may be chosen if it is known that the shortest path distance from a source node to a destination node is within 40% of the longest shortest path distance on the shortest path tree rooted at the source node. The choice of TWO-Q is perhaps, most appropriate in situations where it is necessary to compute a large number of relatively long routes.

Given the variety of applications and the relative closeness in performance, it is perhaps best to test both types of algorithms before making a selection.

REFERENCES

- Zhan, F. B., and Noon, C. E. (1998) Shortest Path Algorithms: An Evaluation Using Real Road Networks, Transportation Science
- Zhan, F. B., and Noon, C. E. (2000) A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths
- Geek-for-Geeks