Dynamic Graph Algorithms: A dynamic implementation of Shortest Path Algorithm and DFS in an undirected graph

Yash Mandilwar - 2018201087
Rishabh Chandra - 2018201074

---

1. Introduction

Shortest Path Algorithm and Depth First Search are two of the most commonly used algorithms in the practical world. **Shortest Path Algorithms (SPA)** are more widely used in computer networks (for faster packet delivery), road networks (to find an efficient route) and many similiar routing scenarios. By these examples you can guess how these algorithmsare used in apps like Google Maps, Routing algorithms like LSR. **Depth First Search (DFS)** is used in problems such as reachability of a node and finding strongly connected components among others. Efficient algorithms (static) for both SPA and DFS are known but they are not useful in a dynamically changing environment, where the nodes and edges may appear/disappeaar or the cost of the edges changes. The static algorithms for both would recompute the entire graph for every change, which is very inefficient. Thus we need to dynamize these static algorithms in such a way that it does not recompute the entire Graph thus making the process faster. No proper solution is known to dynamize a static graph algorithm but many attempts have been made in the past few decades. In this project, we have taken one of the best research papers available for these problems and implemented the same. Section 2 and 3 discusses the dynamic implementation in detail.

2. Shortest Path Algorithm (SPA)

Dynamic single source shortest path problem is a type of dynamic shortest path problem which gives shortest paths from a source vertex to all the other vertices of a graph in dynamic scenario. Almost all of the previous approaches for the solution of dynamic single source shortest path problems are based on identifying all the vertices which may be affected by the given changes in the graph and then updating the shortest paths accordingly. Another way to solve this problem is to make the Dijkstra algorithm dynamic. The Static Dijkstra algorithm is an iterative algorithm which is used to find the shortest path from a specific vertex of the graph called as source vertex to all the other vertices of the graph. The working of the algorithm can be mathematically represented by:

```
if d[u] + w(u, v) < d[v]
        then d[v] = d[u] + w(u, v)
```

where d[u] represents the estimated distance of vertex u from the source vertex and w(u,v) represents the weight of edge (u,v). In section

2.1 Static Implementation of Dijkstra's Algorithm

In this section we briefly discuss how the Dijkstra's static algorithm works so as to create a premise for understanding of the dynamic implementation. In both the static and the dynamic implementation we have used binary heaps to extract the node with the minimum reachable distance from the source.

```
1  function Dijkstra(Graph, source):
2      dist[source] ← 0                        // Initialization
3
4      create vertex set Q
5
6      for each vertex v in Graph:
7          if v ≠ source
8              dist[v] ← INFINITY              // Unknown distance from source to v
9          prev[v] ← UNDEFINED                 // Predecessor of v
10
11         Q.add_with_priority(v, dist[v])
12
13
14     while Q is not empty:                   // The main loop
15         u ← Q.extract_min()                 // Remove and return best vertex
16         for each neighbor v of u:           // only v that are still in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]
19                 dist[v] ← alt
20                 prev[v] ← u
21                 Q.decrease_priority(v, alt)
22
23     return dist, prev
```

Algorithm 1

- Line 2-9 deals with preprocessing data structures by setting distance of every node initially to inifinty (except the source node which iss set to 0).

- Line 11-21 is the actual implementation. The source node is first added to the priority queue (Line 11). Since it is the only node in the queue at the moment, it is also the node with the least distance and hence it is extracted (line 15).

- After fetching the minimum distance node, the next step is to update all it's neighbouring nodes with their distances (given the condition is true) from the source which happens from Line 16 to Line 20. And finally, all it's neighbouting nodes are also pushed into the priority queue (if a node already exists in the queue, it simply updates it with it's new optimal weight at line 21)

This proccess continues till the priority queue is empty i.e all the nodes have been updated with it's optimal distance from the source.

2.2 Dynamizing Dijkstra's Algorithm

In the previous section we discussed how static Dijkstra's Algorithm work. In this section we explain the changes to be made in the above algorithm to dynmaize it. Before we jump to the algorithm itself, it is important that we introduce the reader to the extra components that dynamizes the above algorithm.

Logically, what we mean by dynamizing is that we are trying to avoiding recalculating the shortest path from scratch every time we make a change in the graph, be it edge or vertex. We also know that a change in edge cost or deletion may result in change in shortest path to other vertices. Thus it is important that we propagate the changes from the the vertex **V** (inclined on the edge where the changes were made) to those vertices which were processed after **V**. So the steps are:

1. Detect changes in edge/vertices
2. Make changes in the adjacency list
3. Start Dijkstra again from the vertex **V** (directly inclined to the edge where the changes were made) and propagate the changes throughout.

Algorithm 2

This logic, though simple and intuitive may not always work for undirected graphs as there may always be the case where change in cost of one the edges also effects the distance (from source) of a vertex that was processed before **V** and by the above algorithm we completely ignore them because we were aiming to propogate changes in distances only to those nodes which were processed after **V.** Here's what we mean:

Initially we have the graph in **Figure 1a**. Blue circles represent unvisited nodes. The numbers on the edge represents edge weight and the numbers above the node represents distance from source (INF represents infinity). Number in the nodes represents node number. We consider 0 as the source. **Figure 1b** represents graph after traversal (red edges represent edges included in the shortest path) and **Table 1** shows the order in which the vertices were included in our shortest path set.

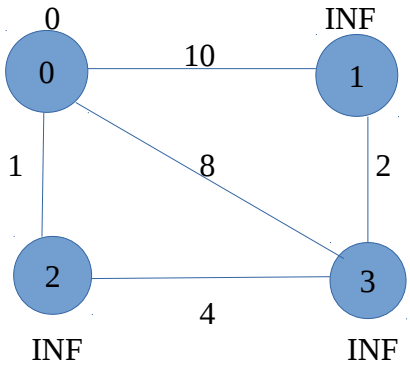

Figure 1a

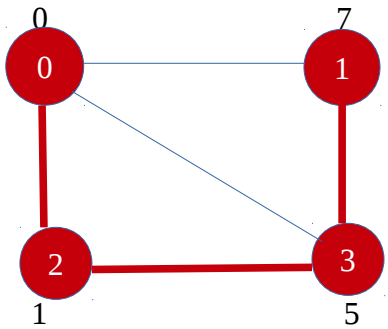

Figure 1b

Time (T)	1	2	3	4
Nodes	0	2	3	1

Table 1

Now, imagine if the edge weight between nodes 2 and 3 changes to 8. According to the steps above we must start dijkstra's algorithm from the node attached to the edge that was just changed. Since it is an undirected graph we have a choice of chosing between nodes 2 and 3. So we check which among them was first processed. Looking at table in **Table 1**, we see that 2 was processed the earliest. Thus we start Dijkstra's Algorithm from node 2 and propogate the changes till all the nodes after node 2 are processed. Notice how we avoid computing from scratch, i.e node 0.

You may notice that when the edge between node 2 and node 3 is changed to 8 the shortest path to node 3 now updates to 9. Here lies the problem. Since node 0 will not be processed again, the algorithm will never discover the edge from node 0 to node 3 which is 8. Thus, the edge between node 0 and node 3 should have been the optimal distance. This happens because node 0 is already 'visited' and thus the node 3 does not check it's distance to node 0.

Another major problem with this is that when we start the Dijkstra's Algorithm from some node **V** (node 2 in the above example), we would expect to have nodes 1 and 3 in our priority queue to treat queue (because when node 0 initially was popped from the priority queue, it added it's neigbours in the queue). This is essential as we would want to reduce this problem of dynamizing to a sub-problem of a static implementation. Thus, it means we somehow have to go back in time as if we knew what changes in a graph is going to occur. **And hence we add an extra step to our above algorithm where we store the states of the heap at every time before a node is popped from the priority queue along with which nodes were visited at that time, and what was the distance of each node at that time**. The structure is discussed below:

```
save_state{

    priority_queue[]
    dist[];
    time_order[];
}
```

where priority_queue[] stores the priority queue at some time T, dist[] stores the distance to each node at some time T and time_order[] stores ***Table 1*** at some time T. And for every time T we maintain an **array of save_state** which enables us to go back in time.

And thus ***Algorithm 2*** can now be modified as shown below:

---

1. Detect changes in edge/vertices
2. Make changes in the adjacency list
3. Restore state at which vertex **V** (directly inclined to the edge where the changes were made) was going to be popped.
4. Start Dijkstra again from the vertex **V** and propagate the changes throughout.

Algorithm 3

Using ideas from ***Algorithm 3*** in ***Algorithm 1*** our final algorithm now is as shown in ***Algorithm 4***:

```
1  function Dijkstra(Graph, source,isInitital):
     If (isInitial==1) {        //isInitial is 1 when Dijkstra is run for the first time
2       dist[source] ← 0                        // Initialization
3        time_order[source] = 1;
4       create vertex set Q
5
6       for each vertex v in Graph:
7           if v ≠ source
8               dist[v] ← INFINITY               // Unknown distance from source to v
9           pred[v] ← UNDEFINED                   // Predecessor of v
10
11       Q.add_with_priority(v, dist[v])
         prev = source
     }
     Else{
         restore_state(source)     //restores state where source will be popped
     }
14    while Q is not empty:                     // The main loop
15        u ← Q.extract_min()                   // Remove and return best vertex
16        for each neighbor v of u:             // only v that are still in Q
17            alt ← dist[u] + length(u, v)
18            if alt < dist[v]
19                dist[v] ← alt
20                pred[v] ← u
21                Q.decrease_priority(v, alt)
          time_order[u] = time_order[prev] + 1
          prev = u
          save_state(Q,dist,prev)          //saves current state
22
23    return dist, prev
```

Algorithm 4