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.

2.2.1 Simple Implementation

Note: This is a simple implementation, but incorrect. This was a challenge we faced to use extended functionality as mentioned in the research paper.

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.

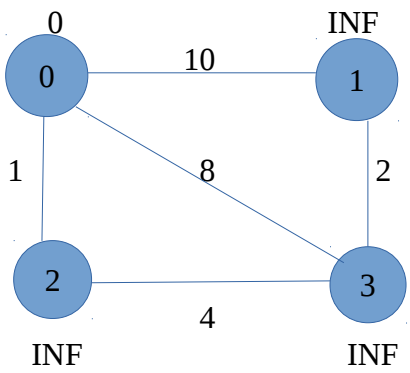


Figure 1a

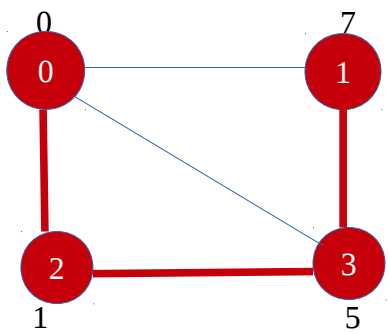


Figure 1b

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.

2.2.2 Correct Implementation (Extending the simple implementation)

Another major problem with the previous logic 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 in ***Algorithm 3***:

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***:

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

Algorithm 4

Compare ***Algorithm 4*** with ***Algorithm 1*** to see how the idea discussed earlier is incorporated.

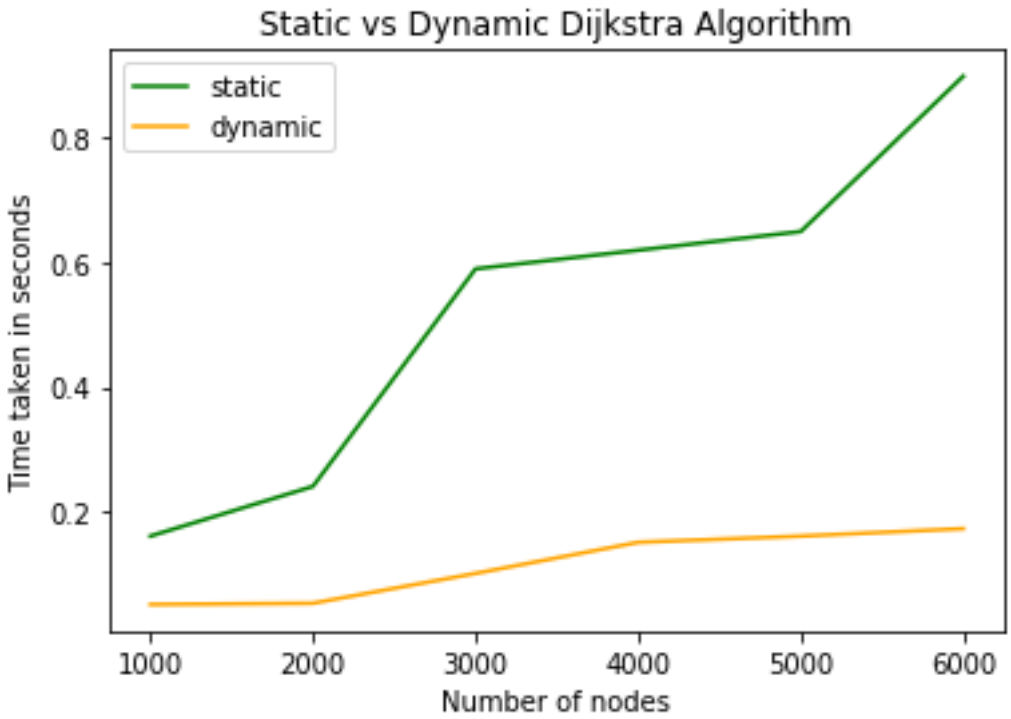
2.3 Time Complexity

In the case of **static dijkstra's algorithm**, it is quite evident that the complexity is ***O(E log V)*** where **E** is the number of edges and **V** is the number of vertices. However, in the case of it's **dynamic algorithm** the complexity in the worst case goes up to ***O(V'E log V)*** where ***V'*** is the number of vertices to update**.** The extra multiplication of **V'** is for the saving state of the graph. However bad it may sound, it's not. In a scenario where edges may change it's weight (or disconnect) or vertices gets added or removed, the static algorithm will have to learn it from the scratch every time. Thus the time taken is always high independent of the updates. Whereas in the

dynamic implementation, the time taken in recomputing is completely dependent on the updates to the graphs, more specifically how far in to the past does it have to go to recompute. Thus, in the practical scenario dynamic graphs perform lot better. The analysis is discussed in the next section.

2.4 Analysis

We have compared static and dynamic algorithm by giving random graphs as input. In every input, the graph changes randomly, such as edge insertion, edge deletion, cost change, vertex insertion and vertex deletion. Below is the result of our analysis:



2.5 Conclusion

In the above implementation we have shown how dynamic graphs are much faster given a random graph with random connectivity. This can be improved using better data structures such as AVL trees or Red Black Trees. Fibonacci Heaps or Van Em De Boas trees can also help in boosting the performance further to extract nodes with least distance.

2.6 References

1. https://www.sciencedirect.com/science/article/pii/S1319157817303828 (Main Implementation)

2. http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382017000300487

3. Depth First Search (DFS)

Static Depth First Search will compute the graph G(V,E) by visiting each node and each edge once.This makes it an O(m+n) algorithm where m is number of edges and n is number of vertices.We address the problem of maintaining DFS when the graph is undergoing updates G + U. (Insertion and deletion of vertices and edges).

Notations

• T (x) : The subtree of T rooted at vertex x.
• path(x, y) : Path from the vertex x to the vertex y in T .
• T ∗ : The DFS tree rooted at r computed by our algorithm for the graph G + U .
• par(w) : Parent of w in T ∗ .

A subtree T' is said to be hanging from a path p if the root r' of T' is a child of some vertex on the path p and r 0 does not belong to the path p.

3.1 Basis of Dynamic Algorithm for DFS on Undirected Graph

We will preprocess the undirected graph into a data structure ,let us call it S. Once done we will query it for each update in U.Below are the queries that can be answered using S in O(log ^{3} n) time.

1. **Query(w, x, y)** : Among all the edges from w that are incident on path(x, y) in G + U , return an edge that is incident nearest to x on path(x, y).

2. **Query(T (w), x, y)** : Among all the edges from T (w) that are incident on path(x, y) in G + U , return an edge that is incident nearest to x on path(x, y).

Hence instead of starting from scratch we can use previous computed DFS(unaffected) and above Query along with Rerooting Algo(we will see later) to maintain DFS.

3.1.1 Properties of DFS Tree

Property 2.1: *Given any rooted spanning tree of graph G a necessary and sufficient condition for any rooted spanning tree to be a DFS tree is that every non-tree edge is a back edge.A non-tree edge is Back- Edge if one of its endpoints is an ancestor of the other in the tree.*

Property 2.2: (*Components Property) Let T∗ be the partially grown DFS tree(white portion of Graph in Figure 1) and v be the vertex currently being visited. Let C (C_1 and C_2 in Figure 1) be any connected component in the subgraph formed by the unvisited vertices. Suppose two edges e and* ***e'*** *from C are incident respectively on v and some ancestor (not necessarily proper) w of v in T∗ . Then it is sufficient to consider only lowest edge* ***e*** *as part of the DFS traversal, i.e., the edge* ***e'*** *need not be scanned. (Refer to Figure 1).*

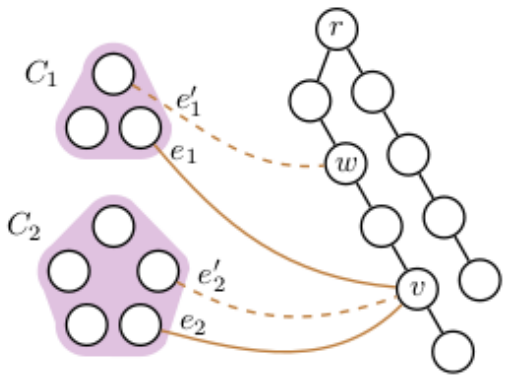


Figure 1: Edges e'_1 as well as e'_2 can be ignored during the DFS traversal.

We will make use of above stated properties in our implementation of Dynamic Updates in Graph.

3.2 Handling an Update

Consider the failure of a single edge (b, f) (refer to Figure 2 (i)). Exploiting the flexibility of DFS traversal, we can assume a stage in the DFS traversal of G - {(b, f)} where the partial DFS tree T∗(rooted at node r) is T - {T (f)} (white portion) and vertex b is currently being visited. Thus, the unvisited graph is a single connected component containing the vertices of T (f).Now to rebuild DFS, according to the components property we need to process only the lowest edge from T (f) to path (b, r) in partially grown Tree T*.
For Example from set {(l,a) ,(k, b)} ∈ G(V,E) in Figure 2 (ii)).We select latter as per Property 2.2.

Our Query from data structure S **Query(T (f), b, r)** should be provide us exactly this edge in **O(log^{3}n)** time.

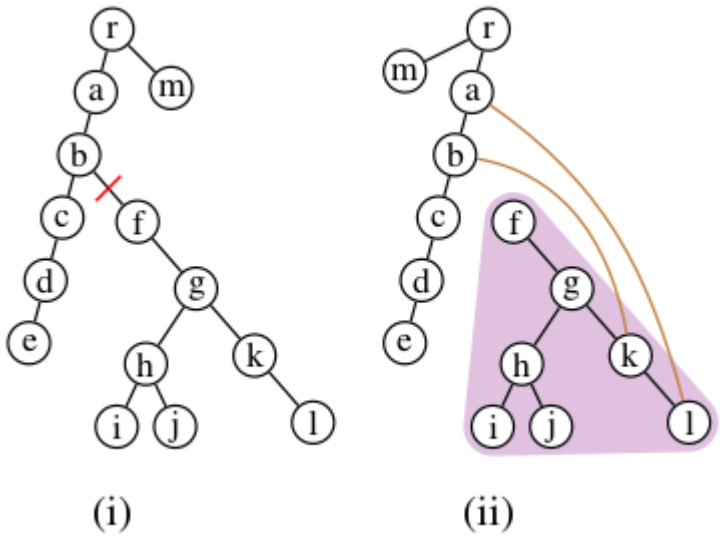


Figure 2.i (b f) Edge is deleted 2(ii) Yellow Edges {(l,a) ,(k, b)} ∈ G(V,E)

The resulting DFS tree of this subgraph would now be rooted at k (Figure 3). Rebuilding the DFS tree after the failure of edge (b, f) thus reduces to finding the lowest edge from T (f) to path(e, r), and then rerooting a subtree T (f) of T at the new root k.

We now describe how this rerooting can be performed in Õ(n) time in the following section.

3.2.1 Rerooting a Tree

Let G'(V',E') be the subgraph whose vertices were included in T(f) in original DFS.Now as per our Figure 2(iii) we need to reroot the subtree from T(f) to T(k) which we can do with help of rerooting algorithm.

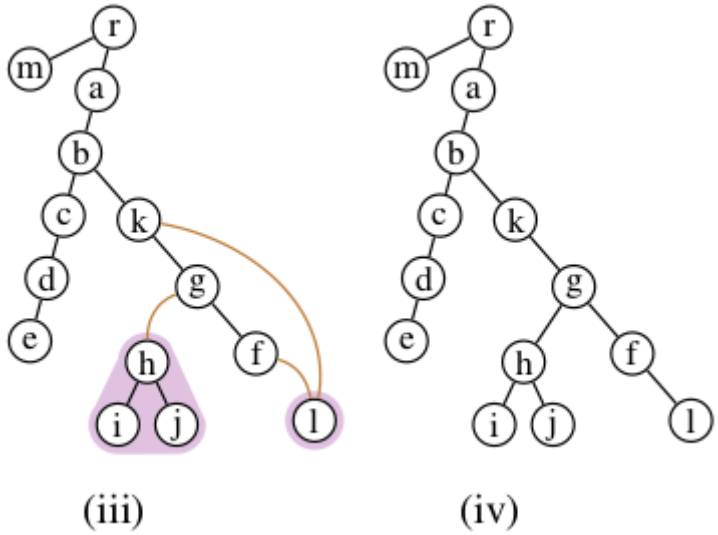


Figure 2(iii) G(V',E') where {k,g,f,l,h,i,j} ∈ V' and T(k) is the rerooted subtree

Below is algorithm that simulates the DFS Traversal inside the subgraph G'(V',E') such that T* (partially grown Tree) has entered from node k.

```
Procedure Reroot(T (f), k ): Reroots the subtree T (f) of T to be rooted at the vertex k ∈ T (f).
foreach (x, y) on path(k, f ) do
    par(x) ← y;                                  /* x = par(y) in original tree T (f ).
        foreach child c of y not on path(f , k ) do
            (u, v) ← Query(T (c), f , y) ;   /* where u ∈ path(r 0 , r 0 ) and v ∈ T (c).
            if (u, v) is non-null then
                Reroot(T (c), v);
                par(v) ← u;
            end
        end
end
```

Query(T (c), f , y) returns lowest edge from T(c) (child subtree of y in path of f to k) to path f ,k. As we have seen before this query will be answered in O(log ^{3} N) from the Data structure S.

Procedure to reroot subtree T' of DFS tree T will take time as no. of edges in subtree T'.And each time Query(T (c), f , y) will take O(log^3 n) in worse case when subtree has many branches.Hence Time complxity will be O(T'log^3 N).

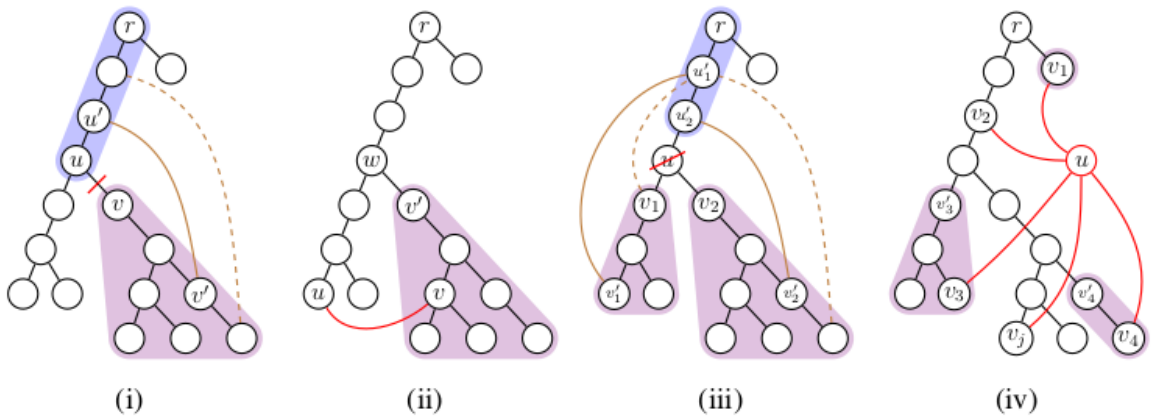


Figure 4: Updating the DFS tree after a single update: (i) deletion of an edge, (ii) insertion of an edge, (iii) deletion of a vertex, and (iv) insertion of a vertex. The reduction algorithm reroots the marked subtrees (shown in violet) and hangs it from the inserted edge (in case of insertion) or the lowest edge (in case of deletion) on the marked path (shown in blue) from the marked subtree.

3.4 Data Structure (S)

We now describe construction of the data structure S.

3.4.1 Stepwise Overview of Construction

1. Perform a DFS to calculate and store the size of subtree formed by the vertex in the DFS tree.

2. Perform a preorder traversal of tree T with the following restriction: Upon visiting a vertex v ∈ T , the child of v that is visited first is the one storing the largest subtree. Let L(Figure 4) be the list of vertices ordered by this traversal.This algorithm is called Heavy-Light Decomposition.(Please refer to reference [2] for details on Heavy Light Decomosition)

3. Build a segment tree S whose leaf nodes from left to right represent the vertices in list L.

4. Augment each node z of T_B with a binary search tree E(z), storing all the edges (u, v) ∈ E where u is a leaf node in the subtree rooted at z in S . These edges are sorted according to the position of the second endpoint in L.(position of second vertex inside sequence L)

3.4.2 Implementation details of above stated steps(Challenges)

- **Step 1& 2.** Basic Idea for usage of HLD is to divide the tree into vertex disjoint chains in such a way that to move from any node in the tree to the root node , we will change at most log N chains.We use this concept as subproblem while quering lowest edge from subtree to a path.

- **Step 3.** Once we get sequence of nodes from HLD traversal we can use the same to form the BaseArray in construction of Segment Tree which is essentially our data structure S.

- **Step 4**. We have used array of std::multiset<std::pair<int,int>,Cmp> to represent data structure S, as well as for BaseArray(which is used fr construction of S) as each cell in the array stores edges from graph (u, v) ∈ E in Binary search Tree .

- **Step 4.** In Base Array Index of array corresponds to position of vertices from sequence obtained HLD Traversal (which corresponds to List L).

Challenge in decision of Data structure :

Constraint1. We have to store keys as pairs (u , v) such that either of them can be repeated as graph can have multiple edges emanating from same vertex.

Constraint 2. Algorithm demands data should be stored in form of BST.

C++ provides inbuilt container Multiset which stores values itself as keys and multiple elements can have equivatlent values.Also its implemented as R-B Tree. We have our custom build comparator to store edges in form of pair to satisfy sorting criteria based on position of second enpoint in L.

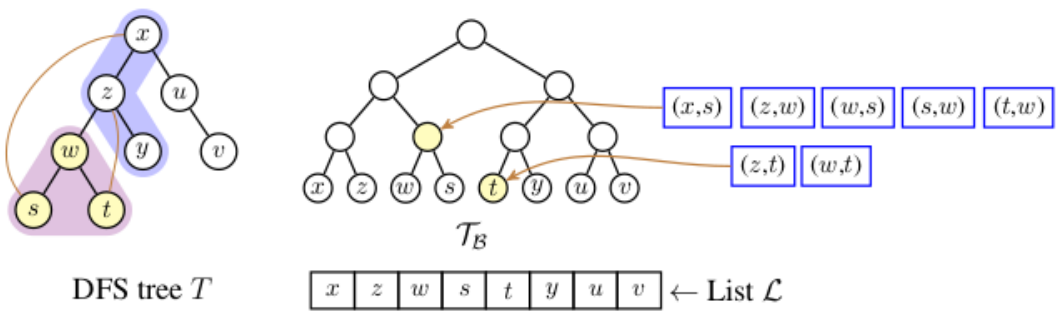


Figure 5: (i) The highest edge from subtree $T(w)$ on $path(x, y)$ is edge (x, s) and the lowest edges are edge (z, w) and (z, t). (ii) The vertices of $T(w)$ are represented as union of two subtrees in segment tree $\mathcal{T}_{\mathcal{B}}$.

3.4.3 Purpose of Data Structure

The construction of S described above ensures the following properties which are helpful in answering a query Query(T (w), x, y) (see Figure 5)

- T (w) is present as an interval of vertices in L (by step 1). Moreover, this interval can be expressed as a union of O(log n) disjoint subtrees in S (by Step 2). Let these subtrees be $S(z_1), \ldots, S(z_q)$.

- It follows from the heavy-light decomposition used in step 1 that path path(x, y) can be divided into O(log n) subpaths $path(x_1, y_1), \ldots, path(x_l, y_l)$ such that each subpath $path(x_i, y_i)$ is an interval in L.

- Let query Q(z, x, y) return the edge on path(x, y) from the vertices in the subtree T B (z), that is closest to vertex x. Then it follows from step 3 that any query $Q(z_j, x_i, y_i)$ can be answered by a single query on BST $E(z_j)$ in O(log n) time.

To answer Query(T (w), x, y), we thus find the edge closest to x among all the edges reported by the queries $\{Q(z_j, x_i, y_i) \mid 1 \le j \le q \text{ and } 1 \le i \le q\}$. Thus, Query(T (w), x, y) can be answered in $O(\log^3 n)$.The space required by D is O(m log n) as each edge is stored at O(log n) levels in S .

The queries Query(T(w),x,y), Query(w,x,y) on T can be answered in $O(\log^3 n)$ worst case time using a data structure S of size O(m log n),which can be built in O(m log n)

3.5 Pseudocode

Below is pseudocode for Preprocessing of Graph which takes O(m log n) where m is edges and n is number of vertices

```
1.    Scan Input stream of edges into adjacency listof graph G(V,E)

2.    call DFS ( )              /* We set up subtree size and parent of each node */
3.    call HLD ( )              /*Heavy-Light Decomposition to fetch sequence to
                                      create BaseArray*/
4.    call make_tree( )         /* Create Segment Tree Data structure S */

5.    pre_LCA( )                /*preprocess the parent of node at each level which
                                saves time when querying for LCA between 2 nodes
                                say x,y*/

6.    scan Input stream for Update requests
```

After preprocessing is complete we compute DFS case by case based on nature of updates.

3.6 Steps for DFS on (Edge and Vertex) updates

1.**Deletion of an edge (u, v):**

a. In case (u, v) is a back edge in T , simply delete it from the graph. Otherwise,

b. Let u= par(v) in T . The algorithm finds the lowest edge (u', v') on the path(u, r) from T (v), where v' ∈ T (v).

c. The subtree T (v) is then rerooted at its new root v' and hanged from u' using (u' , v') in the final tree T * .

2. **Insertion of an edge (u, v):**

a. In case (u, v) is a back edge, simply insert it in the graph. Otherwise,

b. let w be the LCA(Lowest Common Ancestor) of u and v in T and v' be the child of w such that v ∈ T (v').

c. The subtree T (v') is then rerooted at its new root v and hanged from u using (u, v) in the final tree T* .

3. **Deletion of a vertex u**:

a. Let v_1 , ..., v_c be the children of u in T . For each subtree T (v_i), the algorithm finds the lowest edge (u_i' , v_i') on the path(par(u), r) from T (v_i), where v_i' ∈ T (v i).

b. Each subtree T (v_i) is then rerooted at its new root v_i' and hanged from u_i ' using (u_i ' ,v_i') in the final tree T* .

4. I**nsertion of a vertex u:**

a. Let v_1 , ..., v_c be the neighbors of u in the graph. Make u a child of some v_j in T ∗ . For each v_i , such that v_i ∈ path(v_j , r), let T (v_i 0) be the subtree hanging from path(v_j , r) such that v_i ∈ T (v_i').

b. Each subtree T (v_i ') is then rerooted at its new root v_i and hanged from u using (u, v_i) in the final Tree T*.

3.7 Conclusion

In conclusion reroot function will print the subtree which has been affected due to Update.We can see that in our test cases where the nodes mentioned after 'REROOT' are part of the subtree that needs to be recalculated. So this reduces DFS traversal from scratch to the DFS in subtree and thus time complexity is reduced.

3.8 References

1. https://www.cse.iitk.ac.in/users/sbaswana/Papers-published/soda-2016.pdf(Main Implementation)

2. https://blog.anudeep2011.com/heavy-light-decomposition/ (Heavy Light Decomposition)

Project Link:
https://github.com/devloper13/DynamicGraphAlgorithms