CSCE 629

Assignment # 2

Fall 2017

Pentyala, Shiva Kumar UIN: 127003995

10-10-2017

Pre-requisite: As I'm assuming in this assignment solutions for Questions 1, 2 that decrease key operation in Binary heap takes O(log n) time assuming that we know the Index of the element to be modified. So I would like to elaborate on this. In order to identify the element in the heap, we need to maintain a pointer to that node externally, as heaps don't support a FIND operation. So the logic would be a pointer to a node that itself contains a pointer to the current node in the heap that contains the element; and the node that contains the element contains a pointer back to that indirection node. We are making heap element point to this external node, in order to propagate updates. So the node can be updated in O(log n) time. Using these external pointers into the structure, we can access the element whose key should be decreased in time O(1) time. Alternatively, we can store a hashmap inside our heap that maps the heap values to heap indexes. So when we are updating the heap, we will also update those index maps. For this to happen I would extend my usual heap-logic in Homework-1 just a bit as below:

```
Swap(i, j):
        map[value[i]] = j;
        map[value[j]] = i;
```

```
Insert(key, value):
     map.Add(value, heapSize) in the beginning;
```

```
ExtractMin:
       map.Remove(extractedValue) in the end;
```

```
DecreaseKey(value, newKey):
     index = map[value];
     keys[index] = newKey;
```

Of course, Heapify(index) is already present to restore min-heap-property.

Insert and ExtractMin call Swap log(n) times, when restoring heap property. We are adding O(1) overhead to Swap, so both operations remain O(log n). Our main operation, Decrease Key is O(log n): first we lookup index in a hashmap for O(1), then we are restoring heap property for O(log n) as we do in Insert/ExtractMin.

Inference: using values for index lookup will require that they are Unique. If not satisfied with this condition, we will have to add some unique id to our key-value pairs and maintain mapping between this unique id and heap index instead of value-index mapping.

But for Dijkstra it's not needed, as our values will be graph nodes and we don't want duplicate nodes in our Heap.

Please assume above explanation for Questions 1, 2 solutions when I assume that I know the index of the element to be removed in the heap.

Q-1) Given there are n line segments. So there must be 2n end points. For this problem, I assume that there are no vertical segments among the given segments. Using the below algorithm we can solve this problem in **O(n Log n)** time. Following are the detailed steps.

1) ADD all x coordinates of endpoints to Min HEAP. Maintain a flag to indicate whether a point is left end of its line or right end. Every Iteration we will select MINIMUM from the HEAP.

2) Suppose that we have a vertical line L. We sweep the plane from left to right. At every moment, the sweeping line status contains all segments intersecting the line L, sorted by the y-coordinates of their intersecting points with L. The sweeping line status is modified whenever one of the following two cases occurs:

> 1. The line L hits the left-end of a segment S. In this case, the segment S was not seen before and it may have intersections with other segments on the right side of the line L, so the segment S should be added to the sweeping line status. Check for intersection of its line segment with the segments just above and below it. If Intersection exists then just return those segments.
>
> 2. The line L hits the right-end of a segment S. In this case, the segment S cannot have any intersections with other segments on the right side of the line L, so the segment S can be deleted from the sweeping line status and check whether its two active neighbors (lines just above and below) intersect with each other. If Intersection exists then just return those segments.

3) return False as you have not found any intersections.

<u>Data structure selection:</u>

In general, the data structures should support efficient operations that are necessary for updating the structures while the line is sweeping the plane. It is easy to see that the sweeping line status of the line L will not be changed when it moves from left to right unless it hits either an endpoint of a segment or an intersection of two segments.

Therefore, the set of event points consists of the endpoints of the given segments and the intersection points of the segments. I use two data structures EVENT and STATUS to store the event points and the sweeping line status, respectively, such that the set operations MIN, INSERT, and DELETE can be performed efficiently.

So I would use Min-Heap for EVENT and a 2-3 tree for STATUS.

Algorithm SEGMENT-INTERSECTION // O(n Log n)

Given: n line segments l1, l2, . . ., ln //Notation S is used to represent any one of these segments
Output: intersecting segments/False based on Intersection exists or not.

{Implicitly, assumed a vertical line L to sweep the plane. At any moment, the segments intersecting L are stored in STATUS, sorted by the y-coordinates of their intersection points with L. The event points stored in EVENT }

```
 1. EVENT <- Build Min Heap with all x coordinates.                          // O(n)

 2. STATUS = ∅;                                                              // O(1)

3. While EVENT is NOT EMPTY do

3.1         p = MIN (EVENT); DELETE (EVENT, p);                              // O(log n)

3.2         IF p is a left-end of some segment S then                        // O(log n)
                INSERT(STATUS, S);
                let Si and Sj be the adjacent segments of S in STATUS;
                IF S intersects with Si then return (S, Si)
                IF S intersects Sj then return (S, Sj)

 3.3        ELSE p is a right-end of some segment S then                     // O(log n)
                 let Si and Sj be the two segments adjacent to S in STATUS;
                 IF Si and Sj intersect then return (Si, Sj);
                 DELETE(STATUS, S);


 4. return FALSE    //No Intersection                                        // O(1)
```

Time complexity Analysis:

Building a min heap take O(n) time and every extract MIN, DELETE operation takes O(log n) time as the heap needs to get reorganized. In the question it was mentioned to use Homework-1 results directly so I assumed we know the index of the element to be removed, else delete operation might have taken O(n) time.

Step 1 - Building heap with 2n endpoints of the segments, can be done in time O(n)

Step 2 takes constant time O(1).
In the while loop, each segment is inserted then deleted from the structure STATUS exactly once, and each event point is deleted from the structure EVENT exactly once. There are 2n event points. Operations INSERT, and DELETE can all be done in time O(log N) in 2-3 tree on a set of N elements, then processing each segment takes at most O(log n) time, and processing each event point takes at most O(log n) time. Therefore, in worst case the algorithm takes time of the order $O(n) + O(1) + 2n \times O(\log n) + 2n \times O(\log(n)) = O(n \log n)$.

Thus we conclude that the algorithm SEGMENT-INTERSECTION runs in time O(n log n).

Q-2) Dijkstra's Algorithm for SINGLE-SOURCE SHORTEST PATH problem.

Let us assume a positively weighted directed graph G = (V, E), we want to find a shortest path from a given source vertex s ∈ V to each vertex v ∈ V. We assume the weights w(u, v) ≥ 0 for each edge (u, v) ∈ E. Predecessor relation is shown by π[v]. For each vertex v ∈ V, d[v] gives the upper bound on the weight of a shortest path from source s to v. I considered δ(s, u) as shortest path from s to u. I assumed MIN-HEAP data structure (H) for storing vertices, keyed by their d values.

INITIALIZE-SINGLE-SOURCE (G,s)

```
1.  For each vertex v ∈ V[G] do
            d[v] ← INFINITY
            π[v] ← NIL
2.  d[s] ← 0
```

RELAX (u, v, w)

```
1.  IF d[v] > d[u] + w(u, v) then
        d[v] ← d[u] + w(u, v)
        π[v] ← u
```

DIJKSTRA (G, w, s)

```
1.  INITIALIZE-SINGLE-SOURCE (G,s)                  //O(V)
2.  S ← Ø                                           //O(1)
3.  H ← V[G]                                        // O(V)
4.  While H != EMPTY do
            u ← MIN(H)                              //O( log V)
            S ← S ∪ {u}                             //O(1)
5.          FOR each vertex v ∈ Adjacent [u] do
                    RELAX (u, v, w)                 //O(log V)
```

set S for storing vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u ∈ V − S with the minimum shortest path estimate, adds u to S, and relaxes all edges leaving u. In Dijkstra, each time a vertex u is added to set S, we will have d[u] = δ(s, u).

Time complexity Analysis:

The main operations involved are:

INITIALIZE-SINGLE-SOURCE – O(V)

BUILD HEAP – Time to build the binary min-heap is O(V)

EXTRACT-MIN – This operation is performed once per vertex, so in total V times. Each time this operation takes O(log V) as it needs to be reorganized. Hence in total it takes O(V log V).

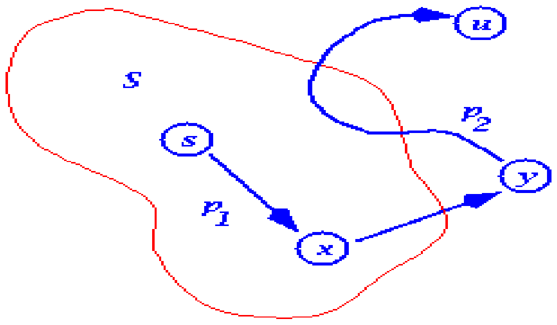
DECREASE-KEY – This takes place in RELAX method. Since the total number of edges in all the adjacency lists is E, in the worst case there might be a total of E iterations of this for loop, and thus a total of at most E DECREASE-KEY operations. Each time, this operation takes O(log V) time assuming we know the index of the element to be removed. Hence in total it takes O(E log V)

In conclusion, **The total running time is therefore O((V + E)log V).** Since Dijkstra works for graphs that are not connected or all vertices are reachable from the source, E ≥ V-1. **Hence running time can also be expressed as O(E log V)**. Alternatively, we can achieve running time of O(V log V + E) if we use Fibonacci heap, in which Decrease Key Operation can be done in O(1) time.

Formal Proof for Correctness of Dijkstra's algorithm:

I will prove this by contradiction.

Lemma 1: Dijkstra's algorithm, run on a positively weighted, directed graph G = (V, E) with a non-negative weight function w and source s, terminates with d[u] = δ(s, u) for all vertices u ∈ V.



Proof: We will show that the following loop invariant always holds:

Invariant: At the start of each iteration of the while loop, d[v] = δ(s, v) is the shortest path from s to v for each vertex v ∈ S.

Initialization: Initially, S is EMPTY SET, so the invariant is true.

Maintenance: Assume, for a contradiction, that there is a "u" in S such that d[u] is not the shortest path from s to u. Without loss of generality, assume "u" was the first vertex added to S, such that the loop invariant did not hold.

Fact:

1. u cannot be s, because d[s] = 0, and s is the first node added to S.
2. S is not Empty Set just before u was added to S, since s is in S.
3. There must be a path from s to u. If there were not, d[u] would be infinity i.e shortest path from s to u would be Infinity.

From 3, we know that there is some path from s to u, so there must be a shortest path from s to u, say p.

Consider the first vertex y along path p that is not in S. (Note that y might be u). Let x in S be the parent of y.

Claim: When u is added to S, d[y] is the shortest path from s to y

Proof of Claim: d[x] is the shortest path from s to x, since x is in S and u is the first vertex added to S such that d[u] is not the shortest path from s to u. We call the function RELAX on the edge (x,y) when x was added to S.
-> Either d[y] = d[x] + w(x , y) is the shortest path from s to y **OR**
-> There is a shorter path p' from s to y that does not go through x. In this case, consider the path p'' from s to u that is the same as p' from s to y and the same as p from y to u. Path p'' provides a contradiction, since p'' is shorter than p and we defined p to be the shortest path from s to u.

End of Claim: Thus, d[y] = d[x] + w(x , y) is the shortest path from s to y.

y occurs before u on the shortest path from s to u and all edge weights are non-negative -> shortest path from s to y ≤ shortest path from s to u.

Therefore d[y] = shortest path from s to y ≤ shortest path from s to u ≤ d[u] (since d[u] cannot be less that the shortest path from s to u). **–> (a)**

But u and y are both in V - S when u was chosen, so d[u] ≤ d[y] (since we extract the vertex with the minimum d value).

Therefore since d[y] <= d[u] and d[u] <= d[y], we have d[u] = d[y]. **–> (b)**

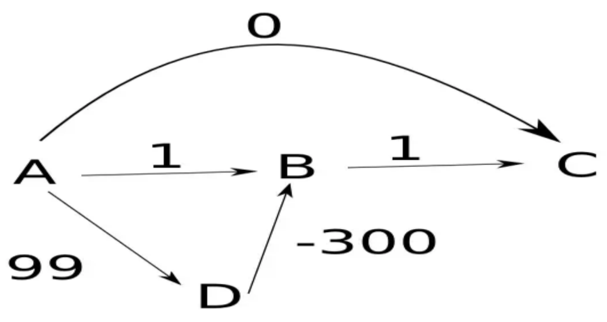
From **(a)** & **(b)**;
d[y] = the shortest path from s to y = shortest path from s to u = d[u]; where d[u] is the shortest path from s to u

This contradicts the fact that d[u] is not the shortest path from s to u.

Termination: At termination, Q = EMPTY SET Since Q = V - S, S = V. Therefore d[u] = shortest path from s to u for all vertices u ∈ V.

Q-3) **Instance for the SINGLE-SOURCE SHORTEST PATH with negative edges.**

Consider the following example:



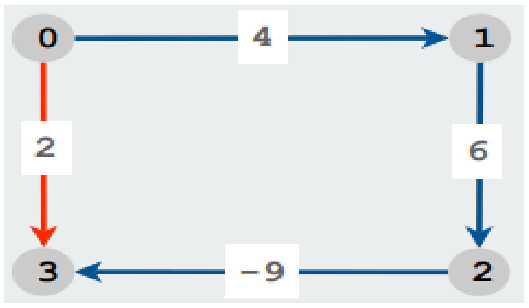
If A is our source and we run Dijkstra on this, we will get the minimum weight of path from A to C to be 0 which should instead be -200.

As discussed in Question-2, First we set s[v] as INFINITY for all vertices and as A is the source vertex we initialize d[s] as zero.

During 1st iteration of While loop, A will be added to set S and FOR loop makes C,B,D as the fringes with d[C]=0, d[D]=99, d[B]=1.

During second Iteration of While loop C would be picked as d[C] is minimum compared to other elements in the queue. Hence C will be added to set S with d value as 0. Thus C got added with 0 as the path length although there is another path A-> D-> B -> C with path cost of -200.

So Dijkstra algorithm seen in Question-2 produced an incorrect answer for this example.



In this example, If 0 is our source and we run Dijkstra on this, it selects vertex 3 immediately after 0 getting added to set S. Here shortest path from 0 to 3 is 0 -> 1-> 2 -> 3 with path cost of 1. But Dijkstra assumes path cost as 2 from 0->3. So Dijkstra algorithm seen in Question-2 produced an incorrect answer for this example.

Q-4) **Develop a linear-time algorithm that solves the Single-Source Shortest Path problem for graphs whose edge weights are positive integers bounded by 10.**

Since Breadth First Search (BFS) gives shortest path for unweighted graphs, I would like to transform given graph in to a graph with dummy nodes and just use regular BFS on it. The idea is if we have an edge with weight w between nodes A and B then remove it and add a path between A and B with w−1 nodes in between, where all the edges will have unit weight. Hence we can use regular BFS on our graph and it will reach the nodes in the same order as it would in the weighted version, without having to transform the algorithm.

In the given graph G1, edge weights are bounded by 10. So for each edge with weight w, we insert additional w-1 nodes in order to make its path length w and each edge to unit weight. We will repeat this step for all the edges in G1.

Assume our new Graph after transformation be G2. If V, E are the number of vertices and edges in G1, then G2 will have maximum of V+9E vertices and 10E edges. If G1 had a shortest path of weight W then there should be a shortest path of length W in G2. We can also say that shortest path of minimum weight in G1 is same as shortest path (obtained using BFS) of minimum length in G2.

BFS on an unweighted Graph with V vertices and E edges take O (V+E) time. Here G2 has at most V+9E vertices and 10E edges. On applying BFS on G2, we can find shorted path in time O(V+9E+10E) = O(V+19E) = O(V+E). From this path we can construct shortest path of the minimum weight in the previous graph G1 in constant time. So BFS gave us a linear-time algorithm that solves SINGLE-SOURCE SHORTEST PATH problem.

Q-5) **Given a positively weighted directed graph G and five vertices s, t, x, y, z, find a path from s to t that contains the vertices x, y, z such that the path is the shortest over all paths from s to t that contain x, y, z.**

We cannot apply Dijkstra directly for this problem due to the constraints involved in this problem. If Dijkstra's algorithm, is run on a weighted, directed graph G = (V, E) with a non-negative weight function w and source s, then when it terminates d[u] is the shortest path from s to u ∀ u ∈ V.

Algorithm: // d[s][x] is the shortest path of x from s. // s is the source and t is the destination.

```
1. Run Dijkstra with s as source and store d[s][x], d[s][y], d[s][z], d[s][t]      //O(m log n)
2. Run Dijkstra with x as source and store d[x][y], d[x][z], d[x][s], d[x][t]      //O(m log n)
3. Run Dijkstra with y as source and store d[y][x], d[y][z], d[y][s], d[y][t]      // O(m log n)
4. Run Dijkstra with z as source and store d[z][x], d[z][y], d[z][s], d[z][t]      // O(m log n)
5. Shortest <- INFINITY
6. For each permutation (a1,a2,a3) of the x, y, z nodes do                         // runs 3! = 6 times
           Shortest = min(shortest, d[s][a1]+d[a1][a2]+d[a2]d[a3]+d[a3][t])   // O(1)
7. return Shortest                                                                  //O(1)
```

Analysis of Algorithm:

Here the number of pass through vertices is limited to 3. This makes us to have an algorithm whose complexity order is the same as the standard Dijkstra. As vertices positions are fixed in the graph, it is sufficient to run Dijkstra 4 times and use those values every time to compute the statement inside for loop. To illustrate let us consider a possible permutation.

Here a1 = x ; a2 = z ; a3 = y ; so From 1, 2, 3, 4 steps of algorithm we know these d[s][x] ; d[x][z] ; d[z][y] ; d[y][t]. so we can calculate sum of these terms.

$$\begin{tikzcd}
s \arrow[r] & x \arrow[r] \arrow[d] & t \\
y \arrow[ur] & z \arrow[l] &
\end{tikzcd}$$

In the same way we compute the sum for all the possible permutations. Total there will be 3! Permutations = 6. They are (s,x,y,z,t) ; (s,x,z,y,t) ; (s,y,x,z,t) ; (s,y,z,x,t) ; (s,z,x,y,t); (s,z,y,x,t). Although in the question it was mentioned that paths may also contain repeated vertices and edges. But Dijkstra takes care of all those cases. For example, in the above illustration the path is s -> x -> z -> y -> x -> t. Although it seems like path contains multiple vertices, edges this comes under the permutation of (s,x,z,y,t). Here second visit through x just happens to be on shortest path from y->t and hence need not be checked explicitly.

Time Complexity:

Here as the number of pass through vertices are only 3, complexity of our algorithm will be same as that of the order of standard Dijkstra. If there are n vertices and m edges in our Graph then from Question-2 solution, running time of Dijkstra is O(m log n)

Step-1 -> O(m log n)

Step-2 -> O(m log n)

Step-3 -> O(m log n)

Step-4 -> O(m log n)

Step-6 -> executes 3! = 6 times -> O(1)

Step-7 -> O(1)

Hence total running time of this algorithm is O(m log n)

Thank you, Geethik