CSCE 629: Analysis of Algorithms
Home Work - 2

Shabarish Kumar Rajendra Prasad
UIN - 228000166

September 2019

1 Question 1

Since the heap is a complete binary tree data structure, with only some of its right-most element missing, we shall consider that the heap is a stored in an array $H(n)$, where n is the number of elements in the heap H. Having such a data-structure makes it easier to find the children or parent of any node. The children of any node i can be found in $H(2i)$ and $H(2i+1)$, whereas the parent of i can be found in $H(i/2)$.

If we design this heap to be a min-heap, then it should satisfy the conditions $H(i) \leq H(2i)$ and $H(i) \leq H(2i+1)$ for all $1 \leq i \leq n/2$. This implies that the value of every parent node is not greater than its children, which means the minimum of all the values is stored in the root. The algorithm assumes that the root of the heap is at $H(1)$. The algorithms are given below:

Algorithm: Min(H)

```
return H[1]
```

Algorithm: Insert(H, a)

```
n = n+1
H[n] = a
i = n

while (H[i/2] > H[i]) and (i >= 2):
    swap (H[i], H[i/2])
    i = i/2
```

Algorithm: Delete(H, i)

```
H[i] = H[n]
n = n-1

if (i >= 2) and (H[i/2] > H[i]):
    while (H[i/2] > H[i]) and (i >= 2):
        swap (H[i], H[i/2])
        i = i/2
else
    while (2i <= n) and (H[i] > H[2i] or H[i] > H[2i+1]):
        if (2i == n) or (H[2i] < H[2i+1]):
            swap (H[i], H[2i]); i = 2i;
        else:
            swap (H[i], H[2i+1]); i = 2i+1;
```

Here the operation Min(H) takes time $O(1)$, whereas Insert(H, a) and Delete(H, i) takes time $O(\log n)$ since the time taken for these two operations are bounded by the height of the tree.

2 Question 2

Given a fully connected graph $G = \{V, E\}$, where a V is the set of vertices and E is the set of edges, a start vertex s and set of weights for the edges w, the Dijkstra's algorithm for a single-source shortest path problem is given below:

Dijkstra's shortest path algorithm $(G = \{V, E\}, s, w)$

```
for each v in V:
    status[v] = unseen

status[s] = intree
H = Null

for each e(s,v) in E:
    status[v] = fringe
    dad[v] = s
    dist[v] = w(s,v)
    Insert(H,v)

while (H != Null):
    a = Min(H)
    status[a] = intree
    Delete(H,a)

    for each e(a,v) in E:

        if (status[v] == unseen):
            status[v] = fringe
            dad[v] = a
            dist[v] = dist[a] + w(a,v)
            Insert(H,v)

        else if (status[v] == fringe) and (dist[v] > dist[a]+w(a,v)):
            Delete(H,v)
            dad[v] = a
            dist[v] = dist[a] + w(a,v)
            Insert(H,v)

return dad, dist
```

Time Complexity

We have seen that the heap operations Insert(H, a) and Delete(H, i) take time $O(\log n)$ and the operation Min(H) takes $O(1)$. Consider that n is the number of vertices and m is the number of edges, then the first for loop executes in $O(n)$. The Insert(H, v) statement inside the second for loop executes in $O(\log n)$ and the loop can execute upto m times. So the time taken for this loop to execute will be $O(m \log n)$. The Delete(H, a) statement inside the while loop executes in $O(\log n)$ and the while loop may execute upto n times since it is executed once for each of the vertex. Thus the time taken for the first 3 statements in the while loop overall will be $O(n \log n)$. The for loop inside while loop may execute for upto m times over all the iterations of the while loop, since it is executed once for each edge in the graph. The loop also have Insert and Delete statement calls and hence they add $O(\log n)$ complexity. Thus the overall complexity of this for loop is $O(m \log n)$

Hence the overall time complexity of this algorithm is $O(n+m\log n+n\log n+m\log n)$ or simply the time complexity of the program is $O((m+n)\log n)$.

Proof of correctness

Let s and v be two vertices in the graph and $\delta(s,v)$ be the shortest distance between the two vertices. Say, we have a larger distance dist[v] $>$ $\delta(s,v)$ gets selected as *intree* first. Let F be the path from s to v, then before removing v from H, there should be at least one other vertex which is a fringe or unseen in F, otherwise dist[v] $= \delta(s,v)$. Let y be the first vertex in F, such that status[y] $\neq$ *intree* and let x be the last vertex that appears along the path of y in the path F. Then, if status[x] = *intree*, and dist[x] $= \delta(s,x)$. Then it follows that status[y] = fringe and y is in H. Thus, dist[y] $\leq$ dist[x] + $w(x,y)$. As x and y are on the path F and all edges are non-negative, we have dist[x] + $w(x,y) \geq \delta(s,v)$. Now if v is the node that is selected from H and not y, then dist[v] $\leq$ dist[x] + $w(x,y) \leq delta(s,v)$ which is a contradiction. Therefore dist[v] $= \delta(s,v)$ for all vertices.

3 Question 3

Since the weight of each edge is bounded by 10, we shall form a new graph of equally weighted edges. For example, for edge in E, if $w(s,v)$ is the weight of an edge between the vertices s and v, in the new graph we shall form a path $\{s \mapsto t_1 \mapsto t_2 \mapsto ....t_{w(s,v)-1} \mapsto v\}$ where each edge in the new graph is of unit weight. We shall use breadth-first search on the new graph and the time complexity of that algorithm will be linear on the number of edges. The algorithm is given below:

Algorithm: Linear time path$(G = \{V,E\}, w, s, v)$

```
G1{V1,E1} = new graph
V1 = V

for each e(s,v) in E:
    append w(s,v)-1 vertices to V1
    append the edges e(s,t[1]), e(t[1],t[2]), ..., e(t[w(s,v)-1],v) to E1

return bfs(G1,s,v)

function bfs(G,s,t):
    for each v in V:
        visited[v] = false

    Queue Q = new Queue
    enqueue(Q,s)
    dist[s] = 0

    while (Q != Null):
        a = dequeue(Q)
        visited[a] = true

        for each e(a,v) in E:
            if visited[v] == false:
                enqueue(Q,v)
                dad[v] = a
                dist[v] = dist[a]+1

    return dad, dist
```

Time Complexity

Let n be the number of vertices and m be the number of edges in the original graph. The taken for the first for loop will be $O(m)$ and the time complexity of bfs would be proportional to the number of edges on the new graph. Since the weight of edges in the original graph is bounded by 10, the number of new vertices in the new graph should be at most 9 for each edge. Thus the total number of vertices in the new graph is $n + 9m$ and the corresponding number of edges should be at most $10m$. Considering the time complexity of the bfs, the while loop may execute up to the total number of vertices which is and the for loop may execute up to the total number of edges. Thus the complexity of while loop is $O(n + m)$ and that of the for loop is $O(m)$. We can conclude that the time complexity of the entire algorithm is $O(n + m)$ which is linear.

4 Question 4

The goal is to start from s and reach t while visiting x, y and z on the way. But the problem does not imply any restrictions on the order that the middle nodes are to be visited. Hence the path can be any one of the following six ways:

- $\{s \mapsto x \mapsto y \mapsto z \mapsto t\}$
- $\{s \mapsto y \mapsto z \mapsto x \mapsto t\}$
- $\{s \mapsto z \mapsto x \mapsto y \mapsto t\}$
- $\{s \mapsto y \mapsto x \mapsto z \mapsto t\}$
- $\{s \mapsto x \mapsto z \mapsto y \mapsto t\}$
- $\{s \mapsto z \mapsto y \mapsto x \mapsto t\}$

Thus we can calculate the cost of all the six paths and use the least expensive path. To calculate the costs of these path we can use Dijkstra's algorithm multiple times as:

1. Run Dijkstra's algorithm with source at s to find the distances $(s \mapsto x), (s \mapsto y)$ and $(s \mapsto z)$
2. Run Dijkstra's algorithm with source at x to find the distances $(x \mapsto y), (x \mapsto z)$ and $(x \mapsto t)$
3. Run Dijkstra's algorithm with source at y to find the distances $(y \mapsto x), (y \mapsto z)$ and $(y \mapsto t)$
4. Run Dijkstra's algorithm with source at z to find the distances $(z \mapsto y), (z \mapsto x)$ and $(x \mapsto t)$

With all these values, we can calculate the costs of all the six paths. Once we have calculated all the path costs, we shall use the least costly path.

Time Complexity

Since we are just calling Dijkstra's algorithm four times, the time complexity of this program is the same as the time complexity of the Dijkstra's algorithm, which is of the order $O(m \log n)$.