Coding + Theory Assignment - 4

Name: Krishnan R, Roll no: CS24BTECH11036

Theory Questions Answers

1) i) Algorithm: Bellman-Ford algorithm can be used

```
bellman-ford(G,s)
{
    for each v in V:
        d[v] ← +∞
        pred[v] ← NIL
    repeat{ converged = true;
    for each (u,v) in E:
        if (d[u] + w(u,v) < d[v]):
            d[v] ← d[u] + w(u,v)
            pred[v] ← u
            converged = false
    } until (converged)
    for each (u,v) in E:
        if (d[u] + w(u,v) < d[v]):
            x ← v
            for i = 1 to |V|:
                x ← pred[x]
            cycle ← empty list
            y ← x
            repeat
                append y to cycle
                y ← pred[y]
            until y == x
            ~~reverse(cy~~
            return cycle
    return "none"
}
```

ii) Theorem: The algo returns vertices of a directed negative weight cycle if it exists and "none" if no negative-weight cycle reachable from the chosen source.

Complexity: Time = $O(|V| \cdot |E|)$
Space = $O(|V|)$

iii) Proof:

The algo relaxes every edge $|V|-1$ times through $|V|-1$ iterations so that if no negative cycles exist in the graph, these iterations ensure that the distance b/w any two vertices u, v is ~~minimum~~ the shortest possible. However, if a negative weighted cycle exists, then there exists a shorter path between u, v through the negative weighted cycle. Hence, once we enter the if condition $d[u] + w(u,v) < d[v]$, we search for the negative weighted cycle in that path. We set x to v and set $x \leftarrow pred[x]$ ~~and~~ iteratively $|V|$ times ensuring that the cycle is encountered atleast once in between. Then we set $y \leftarrow x$; $y \leftarrow pred[y]$ till $y = x$ (initial value of y) indicating we got the cycle while appending it to the empty list cycle.

Time = $|V|-1$ relaxations $\times$ $|E|$ edges + 2^{nd} edge pass + $O(|V|)$ to walk through cycle = $O(|V||E|)$

Space = $O(|V|)$

2) i) Update_MST (G(V,E), MST T), non MST edge e = (u,v) weight decreased to $w'(e)$

```
For each x in V
    visited[x] ← false
    Parent [x] ← NIL
    parentEdgeWeight [x] ← 0
Create Q
Enqueue(Q, u)
visited [u] ← true
while Q is not empty {
    x ← Dequeue (Q)
    if (x = v): break
    for each neighbour y of x in T:
        if not visited [y]:
            visited [y] ← true
            Parent [y] ← x
            Parent EdgeWeight ← weight of edge (x, y)
            Enqueue (Q, y)
}
```

maxEdgeWeight ← −∞
f ← NIL
curr ← v
while (curr ≠ u):
    if parentEdgeWeight [curr] > maxEdgeWeight
        maxEdgeWeight ← parentEdgeWeight[curr]
        f ← edge (parent [curr], curr)
    curr ← parent[curr]
if $w'(e) <$ weight (f):
    $T' \leftarrow T - \{f\} \cup \{e\}$
else
    $T' \leftarrow T$
return T'
}

ii) Theorem: The algo produces a new MST of the modified graph after decreasing weight of e.

Complexity: Time = O(V)
Space = O(V)

iii) Proof: If we add e to T to form a cycle, then we have two cases a) Heaviest edge on path $p(f) > w'(e)$, then replace f, e to minimize weight of weight.

b) $w'(e) \geq w(f)$, adding e does not help and the original T is the MST.

Initialization, Enqueing, iteration over adjacency list, examining each of $|E|$ edges $= |V| - 1$.

Walk back, compare/update is also O(path length) ≤ O(|V|).

∴ Time = O(|V|)

Space = O(|V|) as all arrays/queues hold max |V| elements.

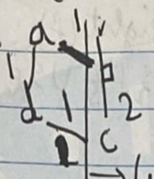
3) i) Statement: If for every cut of the graph, light edge crossing the edge is unique, then the graph has a unique MST. Give a proof and a counterexample for its converse.

ii) Theorem: If for every cut of a weighted undirected G, there is a unique light edge crossing that cut, then G has a MST.

Converse is false.

iii) Proof: Assume every cut has a unique light edge. Consider any MST algorithm that always picks up a light edge. Now, for every cut, this unique light edge must belong to every MST as if it doesn't, then across the cut, the MST would have chosen a heavier edge, which is wrong. Therefore, the MST must contain all these unique light edges and is hence unique.

Counter example: ~~b-a-d-c is the MST~~



→ Cut crosses a–b, c–d with same weight, MST is not unique

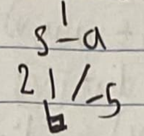
4) i)
```
MST_update(G(E,V), MST T, new edges({(V, u_i)} inserted) E_V = ) {
      if E_V is empty: return T
      best_u ← null
      best_w ← +∞
      for each u_i in ~~u_i~~ of E_V {        → deg(V) iterations
            w ← weight(V, u_i)
            if w < best_w:
                  best_w ← w
                  best_u ← u
      }
      ~~return best_u~~
      T' ← T ∪ {(V, best_u)}
      return T'
}
```
O(1) space due to 2 extra variables

ii) Theorem: The MST of the updated graph is the original MST T + min weight incident edge (V, u^*), $T' = T \cup \{(V, u^*)\}$

Complexity: ~~a) O(deg(V)) + O~~ Time $= O(\deg(V))$, Space $= O(1)$

iii) Proof: Adding a new vertex means we have to insert it into our existing MST T while making sure the result ~~result~~ is an MST. We must connect V with ~~the~~ T by exactly 1 edge as more edges leads to the formation of a cycle. The best way to attach V is using the min weight edge (V, u^*). If we use an edge (V, u) other than the min weight edge, then replacing (V, u) with (V, u^*) leads to a lighter spanning tree. Hence, MST must use (V, u^*).

5) i) Counter Example: Vertices {s,a,b} with edges: s-a (1)
s-b (2)
a-b (-5)



Running Dijikstra from S gives dist(s) = 0, dist(a) = 1, dist(b) = 2, subsequently dist(b) = 1+(-5) = -4. But, if we proceed differently dist(b) = 2, dist(a) = 2+(-5) = -3, which is wrong as the shortest dist b/w s,b is 1-5 = -4 not 2.

ii) Claim: Dijikstra's algo is correct only when all edge weights are non-negative. Neg weight edges may produce incorrect shortest path distances.

iii) Why proof breaks?
The Dijikstra's proof relies on the invariant: If a vertex u is extracted from the priority queue, dist[u] = shortest distance d(s,u). Now, the proof further assumes that since any path through a non extracted vertex has length atleast d(s,u) as edge weights are non-negative.
However, with negative edges, the invariant fails as a path through an extracted vertex be improved by ~~x~~ traversing edges with -ve weight from adjacent non-extracted vertices.
Hence, proof breaks.

6) i) Dijikstra(G(V,E), weights w(e) ∈ {0, 1, ..., W}, source S){

```
    For all v ∈ V: dist[v] ← +∞
    dist[S] ← 0
    Create an array of buckets B[0...V_W] each a queue (V_W = V*W)
    Insert s into B[0]
    current_index ← 0
    processed_count ← 0
    while processed_count < |V|:
        while current_index ≤ V_W and B[current_index] is empty:
            current_index ← current_index + 1
        if current_index > V_W: break
        u ← pop from B[current_index]
        if u is already finalized: continue
        finalize u; processed_count++;
```

```
        for each edge(u,v) in E:
            if (dist[u] + w(u,v) < dist[V]):
                old ← dist[V]
                dist[V] ← dist[u] + w(u,v)
                if dist[V] ≤ VW:
                    insert V into B[dist[V]]
                else: increase no of buckets to fit dist[V] fits
                      place V into bucket[dist[V]]
    return dist
}
```

ii) Theorem: Above algo computes SSSP in $O(VW+E)$ time.
Space complexity: $O(VW+V)$

iii) Proof: correctness: Buckets correspond to tentative distance values and we always process the smallest non-empty. Since, weights are non-negative integers, when we extract a vertex from the smallest non empty bucket its tentative distance is min among all not yet finalized vertices - same concept as Dijikstra. Hence, we cannot a ~~small~~ smaller distance for an extracted vertex.

Complexity: Each edge relaxation is done once time u is per ~~vertex~~ ~~relaxation~~ extracted. Total relaxations $\leq E$.
Finding next non empty buckets can cost scanning empty buckets, but index can only increase upto max dist processed, total scanning over full run is atmost $O(|V|W)$ as dist is in range $[0, W(|V|-1)]$. Scanning cost is $O(VW)$.
$\therefore$ Total time: $O(|V|W + |E|)$

7) i) Statement: HeapSort takes $\Omega(n \log n)$ time on arrays of n distinct elements even on the best input arrangement.

ii) Theorem: For distinct elements, HeapSort has best running time of $\Omega(n \log n)$.
Complexity ~~comment~~: Time $= \Omega(n \log n)$
Space $= O(1)$

iii) Proof: Heap-Sort is a comparison based sorting algo. n! possible permutations for n keys (distinct) and hence decision tree for sorting must have atleast n! leaves. ~~A~~ Binary decision Tree of height h has at most 2^h leaves so $2^h \geq n!$. Hence $h \geq \log_2(n!) \approx \Theta(n \log n)$ by Sterling's approximation, which applies in the best case as the no of permutations in the best case remains $n!$.

Heap sort performs $\Theta(n)$ work to build heap + $n-1$ extract max operations, each requiring $\Theta(\log n)$ work to re-heapify in the worst case, ~~& i~~ $\Omega(\log n)$ in the best case due to the Tree structure.

8) i) Consider a graph $G(V,E)$ be a directed graph (no self loops). Let B be the $|V| \times |E|$ incidence matrix with entries

$$b_{ij} = \begin{cases} -1, & \text{if edge } j \text{ leaves vertex } i \\ +1, & \text{if edge } j \text{ enters vertex } i \\ 0, & \text{otherwise} \end{cases}$$

We have to describe the entries of the matrix product BB^T.

ii) Theorem: $\underset{|V| \times |V|}{L} = BB^T$ is ~~the~~ a matrix such that

$$L_{ij} = \begin{cases} \text{no of edges incident on vertex } i \\ -(\text{no of edges connecting } i, j) \end{cases}$$

iii) Proof: $(BB^T)_{ij} = \sum_{e \in E} b_{i,e} \cdot b_{j,e}$

a) If $i = j$, for each incident edge e to i, we have $b_{i,e} \in \{\pm 1\}$, $b_{i,e}^2 = 1$ for each incident edge. Non incident edges contribute to 0. $\therefore (BB^T)_{ii} = \sum_{e \text{ incident to } i} 1 = \deg_{\text{undirected}}(i)$

= total no of incident edges (in-degree + out-degree)

b) If $i \neq j$, consider an edge from p to q

If $p = i, q = j$, $b_{i,e} = -1$, $b_{j,e} = +1$, contribution of $b_{i,e} b_{j,e} = -1$

If $p = j, q = i$, then contribution is also -1.

Otherwise, contribution is 0

Summing over all edges counts -1 for each edge connecting i, j. $\therefore (BB^T)_{ij} = -$no of edges b/w i, j.

9) i) Adjacency Matrix representation:

Let A be the $n \times n$ adjacency matrix of G (0/1 entries)

Algorithm:

1) Compute matrix A^2 where multiplication follows logical AND logic while addition follows logical OR logic.

$A^2_{uv} = V_w (A_{uw} \wedge A_{wv})$. Done by normal matrix multiplication but with logical ops.

2) Let $A^{'} = A$ OR A^2 (Boolean OR for each $A_{i,j}$ or $A^2_{i,j}$)

3) $A^{'}$ is the adjacency matrix of G^2.

Running time:

Naive boolean matrix multiplication gives $O(n^3)$

ii) Adjacency list representation:

G2(Adj [u] for all u ∈ V, G(V, E)) {

    For each u in V:
        ~~mat~~ array M [1 -- n] ← false
        for each v in adj [u]:
            M[v] ← true
        for each v in adj [u]:
            for each w in adj [v]:
                M[w] ← true
        Adj 2[u] ← {w | M[w] = true, w ≠ u}
        clear marks M entries used

Return G2 with adj matrix ~~A2~~ adj 2

Running Time!

For each vertex, work $\propto \sum_{v \in N(u)} (1 + deg(v))$ ~~+~~ neighbours $= v$ cost $|N(u)|$

iteration of each v cost $(\sum_{v \in N(u)} deg(v))$

Summing over all u, total time is $O(\sum_{u \in V} \sum_{v \in N(u)} deg(v))$

$= O(\sum_{v \in V} (deg(v))^2)$

Worst case cost $= O(|V| + \sum_v (deg(v))^2) = \theta(n^3)$ for dense graphs with $deg(v) = \theta(n)$, bounded by $O(E \cdot \Delta)$ ↳ Max degree

$= O(m^2)$ worst case in most cases