Aashish Yadavally
Spring 2021

Homework 2

Question 1

Let us consider $W.|V| + 2$ bins with labels $0, 1, 2...W.|V|, \infty$ with the source node s in bin 0 and all the other nodes in bin ∞.

Thus, the distance from source node for each of these vertices would be $d(s) = 0$ for the source node, and $d(v) = \infty$ for all the other nodes initially.

Each of these bins are now checked sequentially until the first non-empty bucket is found.

All nodes in the first non-empty bucket are scanned, and the one with minimum distance label are identified.

This node is deleted from the bin, $d.$ is updated for its neighbors (i.e. edges from it to its neighbors are relaxed), and finally it is added to S.

This process is repeated until all the vertices are deleted and no non-empty bin exists.

Time taken to reach the end of all the buckets is $O(W.|V|)$. Since a total of $|E|$ edges are relaxed, the total running time of this algorithm would be $O(W.|V| + |E|)$

Question 2

Let the edge weights in the Dijkstra's algorithm represent the capacity of the edge.

In the bottleneck problem, we want to find a maximum path with minimum capacity.

For a vertex v, $flow(v)$ can be defined as the capacity of the maximum path from s to v.

Thus, for a vertex x having an edge from v,

$$flow(x) = max[min[flow(v), capacity(v, x)]]$$

To begin with, $flow(s) = \infty$ and that of all other vertices is 0.

Using the above notations and updating the $flow$ like in Dijkstra's algorithm, the bottleneck problem can be solved.

The following invariant is maintained by this modified algorithm with respect to vertex x in that the algorithm computes an estimate of the flow of x from the source such that if $flow[x] = capacity(x)$ at any stage of the algorithm, then $flow[x] = capacity(x)$ for the rest of the algorithm.

This can be proved with the fact that $flow(x)$ can never be less than $capacity(v, x)$ based on how the update is designed above for the RELAX-EDGE procedure.

Thus, the algorithm is correct.

Aashish Yadavally
Spring 2021

Homework 2

Question 3

Let $L^{(k)} = (l_{ij}^{(k)}$ be the length of the shortest path from i to j with at most k edges.

In such a case, matrix-multiplication based all-pairs-shortest-path algorithm will get negative values on the diagonal of the matrix $L^{(n)}$ if a negative-weight cycle exists.

Thus, by detecting the presence/absence of negative values on the diagonal of $L^{(n)}$ matrix, it can be deduced whether the graph contains a negative-weight cycle.

Question 4

Let $a_1, a_2, ...a_n$ be a sequence of n integers. Each of the labels in the binary tree T, $v_1, v_2, ...v_n$ from left to right correspond to the sequence of integers.

Let D_i be the length of path from v_i to the root node of T. Cost of the tree T is given by:

$$cost(T) = \sum_{i=1}^{n} a_i.D_i$$

Base case: The root node in the tree is at a depth 0. Thus, $cost(T)$ containing only the root node is equal to 0.

Now, let us consider a subtree T' rooted at vertex v_i, with the integer a_i. The cost of this subtree, $cost(T')$ can be computed only based on the cost of its children.

Minimum cost of this subtree can be computed as follows:

$$cost(parent) = min_i\{cost_i(left\_child) + a_i * (D_i - D_{parent})\}$$
$$+min_j\{cost_j(right\_child) + a_j * (D_j - D_{parent})\}$$

This algorithm can be recursively used to compute the minimum cost of each of the nodes. Since all the possible nodes are explored as children of a particular node, all possible combinations of binary trees are explored exhaustively. Thus, the algorithm is complete.

In the construction of the binary tree, we perform $O(n)$ merges and each merge must compare $O(n^2)$ pairs. Thus, the time complexity of the above algorithm is $O(n^3)$. The $cost(.)$ matrix has a space complexity of $O(n^2)$ and the space complexity of the growing tree is $O(n)$. Thus, the space complexity of the algorithm is $O(n^2)$.

Question 5

```
FIBONACCI(n)
    Let F be an array of size n + 1
    F[0] = 1
    F[1] = 1
```

```
    for i = 2 to n
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

The subproblem graph has $n+1$ vertices $v_0, v_1, ...v_n$ such that v_0, v_1 have no leaving edges and the rest of the vertices has two leaving edges. The edges in this subproblem graph correspond to the relationship between larger subproblems to smaller subproblems, i.e. computation of F_i is related to computation of F_{i-1} and F_{i-2}. Thus, there are $2n-2$ edges in all.

Question 6

Let $C(m)$ be the minimum number of coins required to create the total m using the given positive denominations $d_1, d_2, ..d_n$. $C(m)$ can be calculated based on $C(t)$ for $1 \leq t < m$ as follows:

$$C(m) = min_{i \in [1,n]} C(m - d_i) + 1$$

Base Case: $C(0) = 0$, $C(t) = \infty$ when $t < 0$.

The base case $C(0)$ is true, because by using 0 coins, 0 total can be obtained.

For $d \in d_1, d_2, ..d_n$, given that total $m - d$ can be produced by using x coins, m can be produced by using $x + 1$ coins.

Thus, $C(m) \leq 1 + C(m - d)$.

As we wish to minimize $C(m)$, it becomes equivalent to the subtask of minimizing $C(m-d)$. This can be further extended similarly. Thus by considering all values of d, we will be able to generate all possibilities of producing the total m from the given denominations. In case it cannot be generated by any one such combination, m would be ∞, representing that there is no valid combination. Thus, the algorithm is exhaustive, indicating its correctness.

```
m: Total
D: Array of denominations
MIN-COUNT(m, D)
    if m is greater than 0
        min_d <- inf
        for d in D
            Cd <- MIN-COUNT(m - d, D) + 1
            if Cd < min_d
                min_d <- Cd
        return min_d
    else if m is equal to 0
        return 0
```

```
    else if m is less than 0
        return inf
```

It takes $O(n)$ time to construct each $C(m)$ for any m. There are $O(m)$ values of m to consider. Therefore, the running time of the algorithm would be $O(mn)$.

Question 7

```
COUNT-PATHS(G)
    TOPOLOGICAL-SORT(G)
    REVERSE-ORDER(G)
    for vertex u in G (topologically sorted and in reverse-order)
        for vertex v in Adj[u]
            v.path-count = u.path-count + v.path-count + 1
    total <- 0
    for vertex u in G:
        total <- total + u.path-count
    return total
```

In the above algorithm, topological sorting takes $O(|V| + |E|)$ time. Calculation of all path counts takes $O(|E|)$ time. Computing total number of paths takes $O(|V|)$ time. Thus, the running time for the algorithm is $O(|V| + |E|)$.

Question 8

Algorithm:

For any $0 \leq i \leq k$, let $\delta_r(u, i)$ be the length of the shortest path with i changes for a red edge coming to u.

Let $\delta_b(u, i)$ be the length of the shortest path with i changes for a blue edge coming to u.

$$\delta_r(u, i) = min_{v \in N_r^-(u)} min(c(v, u) + \delta_r(v, i), c(v, u) + \delta_b(v, i-1)) \quad \text{-(i)}$$
$$\delta_b(u, i) = min_{v \in N_b^-(u)} min(c(v, u) + \delta_b(v, i), c(v, u) + \delta_r(v, i-1)) \quad \text{-(ii)}$$

where $N_r^-(u)$ is a set of all vertices v which have red edges to u and $N_b^-(u)$ is the set of all vertices v which have blue edges to u.

Consider a vertex u. For all the vertices v which have a red edge from v to u, the possible alternatives are that either a red edge or a blue edge comes to v. Thus, in (i), a minimum of both of these alternatives are considered, wherein computing $\delta_r(v, i)$ and $\delta_b(v, i-1)$ would

be the sub-problems respectively. Similarly, for all the vertices v which have a blue edge from v to u, the possible alternatives are that either a red edge or a blue edge comes to v. Thus, computing $\delta_b(v, i)$ and $\delta_r(v, i-1)$ would be the sub-problems respectively. It can easily be seen that this is exhaustive, and all the possible edge-color combinations between all the vertices are considered. Thus, the path cost so obtained would be minimum, implying the correctness of the algorithm. Each of the recursive components run in linear time. Thus, the running time complexity of the algorithm is $O(|V| + |E|)$.