**CS 344: Design and Analysis of Computer Algorithms** **Rutgers: Spring 2021**

# Homework #3

March 25, 2021

*Name: Letao Zhang* *Extension: No*

## Homework Policy

- If you leave a question completely blank, you will receive 25% of the grade for that question. This however does not apply to the extra credit questions.
- You are allowed to discuss the homework problems with other students in the class. **But you must write your solutions independently.** You may also consult all the materials used in this course (video recordings, notes, textbook, etc.) while writing your solution, but no other resources are allowed.
- Do not forget to write down your name and whether or not you are using one of your two extensions. Submit your homework on Canvas.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "use DFS (or BFS) to find all vertices reachable from a given vertex $s$ in a graph $G$ in $O(n+m)$ time". You are **strongly encouraged to use graph reductions** instead of designing an algorithm from scratch whenever possible.
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question).
- The "Challenge yourself" and "Fun with algorithms" are both extra credit. These problems are significantly more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.

---

**Problem 1.** Recall the job scheduling problem from the lectures: we have a collection of $n$ processing jobs and the length of job $i$, i.e., the time to process job $i$, is given by $L[i]$. This time, you are given a number $M$ and you are told that you should finish all your processing jobs between time 0 and $M$; any job not fully processed in this window then should be paid a penalty that is the same across all the jobs. The goal is to find a schedule of the jobs that minimizes the penalty you have to pay, i.e., it minimizes the number of jobs not fully processed in the given window.

Design a greedy algorithm that given the array $L[1:n]$ of job lengths and integer $M$, finds the scheduling that minimizes the penalty in $O(n \log n)$ time. **(25 points)**

**Solution.** *Algorithm.* We are using greedy algorithm to solve this problem. Firstly we sort the intervals in increasing (non-decreasing) order based on their starting point and let A[1 : n] denote the intervals in the sorted order. The logic is simple, after sorting array, we start from the beginning to traverse the array. While traversing, using for loop to let integer M subtracts processing jobs from given total jobs that M = M - arr[i]. If $M < 0$ means we have finished the specified jobs so that break the loop. We will then pay a penalty of M - i for the remaining jobs.

*Proof of Correctness.* We analyze the algorithm using an exchange argument. Let G = g1, . . . , gk be the output of the greedy algorithm and let $O = o_1, o_2, ..., o_W$ be any optimal solution to the problem; we sort the items picked by the optimal solution also in the non-increasing order of their values. We are now going

to start exchanging O into G without decreasing the value of the solution. The way to do this is to find the first place where the solution G and O differ from each other.

More formally, we let j be the index such that $g_1 = o_1, g_2 = o_2, ..., g_j - 1 = o_j - 1$but $g_j \neq o_j$. In other words, we can write $O = g_1, ..., g_j - 1, o_j, o_j + 1, ..., o_W$. Let us now examine what happens if we exchange oj with gj in O to obtain a solution O', $O' = g_1, ..., g_j, o_j + 1, ..., o_W$. We would like to argue that the value we obtain by picking the items in O' is at least as large as the value we obtain by picking the items in O. To prove this, we only need to show that $v_g j \geq v_o j$ because all other items are the same in O and O'.

Consider the choice of d before we pick $g_j + 1$ in the greedy. By construction, all intervals with end point before d are already covered in both G and O and no interval with endpoint after d is covered yet. Greedy picks the interval $g_j + 1$ with the largest endpoint among such intervals. Moreover $o_j$. Start needs to also be at most equal to d as otherwise O will not cover the gap between d and $o_j$ .start (remember that union of all intervals is [mini ai, maxj bj ]). As such, $g_j.end \geq o_j$ .end and hence if O could cover $[d, o_j.end]$ and cover the entire line, thus has the same union as union of all intervals. This means that O' is also another optimal solution for the problem. We are now done since we can keep exchanging all elements of O with G one by one and obtain that G is also another optimal solution.

*Runtime Analysis.* Finally, we should analyze the runtime of the algorithm. We can use any fast sorting algorithm, say, merge sort, to sort the items in the first step. This takes O(n log n) time. The second step of subtracting can be done in O(n) time. Therefore, the total runtime is $O(n \log n)$.

---

**Simple bonus credit:** Can you design an algorithm that instead runs in $O(n + M)$ time? **(+5 points)**

**Solution.** We can use DFS algorithm which runs in $O(n + M)$ time.

---

**Problem 2.** You are stuck in some city $s$ in a far far away land and you know that your only way out is to reach another city $t$. This land consists of a collection of $c$ *cities* and $p$ *ports* (both $s$ and $t$ are cities). The cities in the land are connected by one-way *roads* to other cities and ports and you can travel these roads as many times as you like. In addition, there are one-way *shipping routes* between certain ports. However, unlike the roads, you need a ticket to use these shipping routes and you only have 3 tickets; so effectively you can use at most 3 shipping routes in your journey.

Assume the map of this land is given to you as a graph with $n = c + p$ vertices corresponding to the cities and ports and $m$ directed edges showing one-way roads and shipping routes. Design an algorithm that in $O(n + m)$ time outputs whether or not it is possible for you to go from city $s$ to city $t$ in this land following the rules above, i.e., by using any number of roads but at most 3 shipping routes. **(25 points)**

**Solution.** *Algorithm.* We simply use Breadth Search Problem(BFS) to solve this problem. 1. Initialize an array mark[1 : n] with 'FALSE'. 2. Create a queue data structure Q and insert s to Q. Determine the cities, ports, roads and shipping routes. 3. While Q is not empty: (a) Let v be the first vertex of Q and dequeue (remove) this vertex from Q. (b) Visit all adjacent cities and ports considering whether it is been visited before. (c) If mark[v] = 'TRUE'; continue from the beginning of the while-loop. (d) Otherwise, let mark[v]

= 'TRUE', and for $u \in N(v)$ : insert u to the end of Q. (e) If the shipping routes are less than equal to 3 , then push it into the Queue Q. 4. At the end, we return all vertices v with mark[v] = 'TRUE' as the answer.

*Proof of Correctness.* We need to prove that every vertex connected to s will be output and no other vertex is also output by this algorithm. The second part is straightforward because we are only visiting city s, neighbors of s, their neighbors and so on and so forth, so for any vertex marked, there is a path from s to that vertex in the graph. The other part is also easy: consider any unmarked vertex u. We prove that u is not in the connected component of s. Since u is unmarked, none of the neighbors v of u can be marked: otherwise, when we visited v for the first time, we would have marked u also at some point later We can then continue like this to all neighbors of v and say that all neighbors of v needs to be unmarked also. We expand this until we find all of the vertices that are connected to u and we know that they are all unmarked. But this means that s was not any of those vertices since s is actually marked; hence u cannot be part of the connected component of s since there is no path from city s to city t.

*Runtime Analysis.* For each vertex v, we run the entire body of the while-loop at most once (which takes $O(|N(v)|)$ time) and after that it is marked and we only dequeue the vertex in O(1) time (this time can be charged to the time spent by vertex u that inserted v for the second (or third, etc.) time to the queue Q instead of accounting for it again by vertex v). As such, the total runtime of the algorithm is at most $O(n+m)$.

---

**Problem 3.** We are given a directed acyclic graph (DAG) $G = (V, E)$ and two vertices $s$ and $t$ in this DAG. Design a dynamic programming algorithm that in $O(n + m)$ time, decides if the *number* of different paths from $s$ to $t$ is an *odd* number or an *even* one. You can assume that the number of paths from a vertex to itself (i.e., when $s = t$) is one and thus the correct answer in this case is *odd*. **(25 points)**

**Solution.** *Algorithm.* Similar to any other dynamic programming algorithm, we start with a recursive formula:

*Specification:* For every vertex $v \in V$ :

• Num(u): denote the he number of different paths from s to t is an odd number or even starting from s and ending in v. By convention, we define Num(u) = 1.

The output answer is the array that contains Num(u) for every $v \in V$ .

*Recursive formula:* For every vertex $v \in V$ :

$$Num(u) = \begin{cases} 1, & if v = s \\ \sum_{(u,v \in E)} Path(v), & otherwise \end{cases}$$

We can calculate the number of paths by slightly modifying the DFS algorithm.

(a) Initialize an array mark[1 : n] with 'FALSE'. Run the following recursive algorithm on s, i.e., return DFS(s).

(b) If start is end vertex then return 1 path.

(c) Else, if number of paths to t from u is not set yet, then number of paths from u to t is just the sum of paths from its children to t. Then DFS(s,t) would return the number of paths.

(d) DFS(v): 1. If mark[v] = 'TRUE' terminate. 2. Otherwise, set mark[v] ='TRUE' and for every neighbor $u \in N(v)$.

At the end, we return all vertices v with mark[v] = 'TRUE' as the answer, i.e., as vertices that are in the same connected component of s.

*Proof of Correctness.* The correctness follows directly from the correctness of DFS. We need to prove that every vertex connected to s will be output and no other vertex is also output by this algorithm. The second part is straightforward because we are only visiting s, neighbors of s, their neighbors and so on and so forth, so for any vertex marked, there is a path from s to that vertex in the graph.The other part is also easy: consider any unmarked vertex u. We prove that u is not in the connected component of s. Since u is unmarked, none of the neighbors v of u can be marked: otherwise, when we visited v for the first time, we would have marked u also at some point later (because we run DFS(u) for that vertex when visiting v, i.e., in DFS(v)). We can then continue like this to all neighbors of v and say that all neighbors of v needs to be unmarked also. We expand this until we find all of the vertices that are connected to u and we know that they are all unmarked. But this means that s was not any of those vertices since s is actually marked; hence u cannot be part of the connected component of s since there is no path from s to u.

*Runtime Analysis.* The runtime is same as the runtime for DFS that we visit each vertex v at most once (after that it is marked and we do not spend more time in visiting the vertex (think of memoization)) and it takes O(—N(v)—) time to process this vertex. Hence, the total runtime is at most $c\sum_( v \in V)|Num(v)|$ for some constant c: since the total degree of vertices is proportional to the number of edges, the runtime of this algorithm is $O(n+m)$.

---

**Problem 4.** You are given a 3D-matrix $A[1:n][1:n][1:n]$ with entries in $\{0,1\}$. You start from position $A[1][1][1]$ in this matrix and you can only move as follows:

- if you are at position $A[i][j][k] = 0$, then you can only move to either

$$A[i+1][j][k] \quad \text{or} \quad A[i][j+1][k];$$

- if you are at position $A[i][j][k] = 1$, then you can only move to either

$$A[i][j+1][k] \quad \text{or} \quad A[i][j][k+1];$$

In either of the cases, you cannot make a move that takes you outside the boundary of the matrix. The goal is to find a longest sequence of valid moves in this matrix starting from $A[1][1][1]$.

Design an $O(n^3)$ time algorithm that outputs the length of the longest sequence of moves. **(25 points)**

**Solution.** *Algorithm.* To solve this problem, we are using Dynamic Programming to store the result of outputs the length of the longest sequence of moves. Firstly, we traverse all the indexes and start from position $A[1][1][1]$ in this matrix. We are using three for loops of i, j and k to illustrate that for (int i=2; $i \leq n+1$; i++) ; for (int j=2; $j \leq n+1$; j++) ; for (int k=2; $k \leq n+1$; k++). We then have two

options that if $(A[i-1][j-1][k-1] == 0)$, $mat[i][j][k] = MAX(mat[i-1][j][k], mat[i][j-1][k]) + 1$; else $mat[i][j][k] = MAX(mat[i][j-1][k], mat[i][j][k-1]) + 1$.

*Proof of Correctness.* We need to argue that the answer to the length of the longest sequence of moves. We do so by showing that there is a one-to-one correspondence between paths leaving s in G and increasing subsequences in A. Consider any path s, v1, . . . , vk; by definition of edges we know A[v1] ¡ A[v2] ¡ . . . ¡ A[vk] and v1 ¡ . . . ¡ vk, so this is indeed an increasing subsequence in A. We are checking if moving along the row gives maximum output or moving along the column. Whichever path gives the maximum output, we store it in the mat and add 1 for reaching that point. The same exact proof also shows that any increasing subsequence maps to a unique path in G. Hence, the longest increasing subsequence of A corresponds to the longest path starting from s and vice versa, proving the correctness.

*Runtime Analysis.* We output the length of the longest sequence of moves using 3 loops, which makes the runtime of this algorithm be $O(n^3)$.

---

**Challenge Yourself.** A *bridge* in an undirected connected graph $G = (V, E)$ is any edge $e$ such that removing $e$ from $G$ makes the graph disconnected. Design an $O(n + m)$ time algorithm for finding *all* bridges of a given graph with $n$ vertices and $m$ edges. **(+10 points)**

**Solution.** We are solving this problem based on depth first search and has $O(n + m)$ complexity.

(a) Initialize an array mark[1 : n] with 'FALSE'. Run the following recursive algorithm on s, i.e., return DFS(s). (b) If start is end vertex then return 1 path. (c) Else, if number of paths to t from u is not set yet, then number of paths from u to t is just the sum of paths from its children to t. Then DFS(s,t) would return the number of paths. (d) DFS(v): 1. If mark[v] = 'TRUE' terminate. 2. Otherwise, set mark[v] ='TRUE' and for every neighbor $u \in N(v)$. At the end, we return all vertices v with mark[v] = 'TRUE' as the answer, i.e., as vertices that are in the same connected component of s.

---

**Fun with Algorithms.** We are given an undirected connected graph $G = (V, E)$ and vertices $s$ and $t$. Initially, there is a robot at position $s$ and we want to move this robot to position $t$ by moving it along the edges of the graph; at any time step, we can move the robot to one of the neighboring vertices and the robot will reach that vertex in the next time step.

However, we have a problem: at every time step, a subset of vertices of this graph undergo maintenance and if the robot is on one of these vertices at this time step, it will be destroyed (!). Luckily, we are given the schedule of the maintenance for the next $T$ time steps in an array $M[1 : T]$, where each $M[i]$ is a linked-list of the vertices that undergo maintenance at time step $i$.

Design an algorithm that finds a route for the robot to go from $s$ to $t$ in at most $T$ seconds so that at no time $i$, the robot is on one of the maintained vertices, or output that this is not possible. The runtime of your algorithm should ideally be $O((n + m) \cdot T)$ but you will receive partial credit for worse runtime also.

**(+10 points)**

**Solution.** We are solving this problem based on breadth first search. 1. Initialize an array mark[1 : n] with 'FALSE'. 2. Create a queue data structure Q and insert s to Q. Determine the cities, ports, roads and shipping routes. 3. While Q is not empty: (a) Let v be the first vertex of Q and dequeue (remove)

this vertex from Q. (b) Visit all adjacent cities and ports considering whether it is been visited before. (c) If mark[v] = 'TRUE'; continue from the beginning of the while-loop. (d) Otherwise, let mark[v] = 'TRUE', and for $u \in N(v)$ : insert u to the end of Q. (e) If the shipping routes are less than equal to 3 , then push it into the Queue Q. 4. At the end, we return all vertices v with mark[v] = 'TRUE' as the answer.

---