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

Homework #3

March 21, 2021

Name: Yinfeng Cong *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. Solution to Problem 1 goes here.
Algorithm:
1. Apply merge sort on the array L[1:n] in the increasing order.
2. set i from 0 to n, and set temp = 0
If $L[i] > M$, output penalty = n
else
temp = temp + L[i]
if temp $\leq$ M, continue
else if $temp > M$, output penalty = n - i + 1
Proof of correctness:
Let G = g1,g2,....gw be the greedy solution, $g1 < g2.... < gw$

Let O = o1,o2....ow be an optimal solution of the problem, and $o1 < o2 < ...ow$
We assume that g1 = o1, g2 = o2....until g_j != o_j
we set O' = g1,g2,...g_{j-1}, g_j,o_{j+1}....
the elements in O' and O are nearly the same except for g_j
We want to prove that O' is also an optimal solution, so we need to prove $g_j < o_j$
1. For all items i that does not belong to g1 ... g_{j-1} , $g_j <= i$, this is because g_j is the smallest number among the items not picked yet
2. o_j does not belong to g1 ... g_{j-1} , this is because g1 ... g_{j-1} = o1 ... o_{j-1} by definition of the index j and o_j clearly does not belong to the set
3. by combining 1 and 2 above, we have $g_j < o_j$
This implies that O' is another optimal solution. Then we continue the previous exchange, this time G and O', and switch the item j+ 1, and continue like this until we switch all the items originally in Oto become the items in G
At last, we have the value of items in G is at least as small as the values in O, which means that G is also an optimal solution. Hence the correctness is proved
Runtime analysis:
Runtime of merge sort is nlogn, and runtime of the loop is n. So total runtime = O(nlogn) + O(n) = O(nlogn)

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

Solution. Solution to Problem 1 bonus credit goes here.

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. Solution to Problem 2 goes here.
Algorithm:
Create an stack S [1:n], and create an array of size [1:n] called A
Run the following recursive algorithm on v
When the stack S is not empty,
Define Search(v):
Put v into the stack S and find the neighbor of v, set count =0
—If v is a port, count ++
If count > 3
pop the element in S and put it into the output
Search (the v that is on the top of the stack)
make count = 0 again

If count <= 3
we move to the next neighbor u, and use Search(u)
if No neighbors then we pop the element in S and put them in the output, search (the top of the stack)
—If v is a city
we move to the next neighbor u, and use Search(u)
if No neighbors then we pop the element in S and put them in the output, search (the top of the stack)
Proof of correctness:
We have known the correctness of stack. We need to prove the correctness of our algorithm Search (v)
The difference of this question from DFS in the lecture is that we need to care about the number of ports we passed
we need to make sure that we passed 3 or less than 3 ports to get to our destination
So for any beginning point u, and destination v. we passed several ports and cities.
we just find the neighbor of the u step by step until we arrived at v. The only thing we need to care about is that we need to make sure that the passed ports are smaller or equal to 3
So we have the condition if port > 3, we need to search again from the beginning point u until we arrived the destination with less or equal to 3 times.
And then we output the vertices we have passed as the answer
This is the same as DFS, and we have proved the correctness of DFS in the lecture.
Runtime:
We visit all city at most once, and we can at most visit the ports for 3 times. and it takes O(—N(v)—) time to process this vertex.
since total degree of vertices is proprtional to the number of edges, the runtime of thsi algorithm is O(m+n)

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. Solution to Problem 3 goes here.
Algorithm:
part1:
First we build an algorithm to find all the possible paths from u to v.
Set the beginning point as u and destination as v, and build a stack S [1:n], set count = 0
Search (v):
set u into the stack, and find the neighbor of u (we call it i)
Then, we find the neighbor of i called o, we put i into the stack
We do this recursively
until we find the destination v
—If we find the destination v
we put v into the stack, and we pop the element in S, count = count +1
we come back to the parent of v(we call it p), and run Search (p) until we get v again (but we mark v as true this time, so p can't just arrive v)
If we did not find a path by search(p), we find the parent of p and do the step above again recursively until we find v again, and then we pop the stack, and we make count = count +1 again. Again, we run Search(parent of v) again.
—or we arrive at the end of the DAG, and we didn't find v. Then we use Search(v) again
part 2:
Then we use dynamic programming to help us solve this problem
Specification:

path(v): denote the number of paths from u to v is even or odd, and we use what we get in part 1 — count to help us solve it.
Recursive formula:
path(v) =
undefined if u = v
even if there is no path from u to v
otherwise,
odd if count % 2 = 1
even if count % 2 = 0
Proof of correctness:
For part 1,
we have known the correctness of stack, and the Search(v) algorithm is in fact very similar to the DFS algorithm (which was mentioned in the lecture)
For the reason that it is a DAG, so when we try finding v, it won't run without stopping.
We assume that we have found a path from u to v (By finding neighbors again and again)
If we want to find other paths, we need to make sure that we won't count the path we have found.
So we go back to the parent of v, we assume it as i, to find if there is another path from it to v
except for i-¿ v, we may find i -¿ o-¿ v
Then,we got another path for this problem, after we searching all the neighbors of i, we go back to its parents recursively until we come back to u, and find that there is no other neighbors of u we have never visited
For part 2,
basecase: there is no path from u to v
0 % 2 = 0, so the formula is proved
induction:
we assume that count = n, and count % 2 = 1
Then for (count + 1), (count + 1) %2 = 0
and it obeys the rule of even and odd
Runtime analysis:
Runtime of this algorithm would then be O(n+m) because there are n subproblems and each subproblem v takes time proportional to in-degree of v; since the total sum of in-degrees is equal to m, the total runtime 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. Solution to Problem 4 goes here.
Algorithm:
First, we need to utilize DFS to find A[i][j][k]

Specification:
Long[i][j][k]: denote the longest move of sequence starting from A[1][1][1] to A[i][j][k]. By convention, we define A[1][1][1] = 0, and A[i][j][k] as 'undefined' if it can't be reached.
Recursive formula:
Long[i][j][k] =
0 if A[1][1][1]
undefined if A[i][j][k] can't be reached
Otherwise,
max$\{A[i-1][j][k], A[i][j-1][k], A[i][j][k-1]\}$+1
Proof of correctness:
Basecase:
we have set A[1][1][1] = 0, so it is true
Then, the longest move of sequence from [1][1][1] to [i][j][k] one of the in-neighbors of [i][j][k], say [i][j][m] then take the edge (m,k) to reach k. Then the move of sequence from s to [i][j][m] should be the longest path before.
Hence, length of that path should be Long [i][j][m]. By taking maximum over all in-neighbors of k, we will set Long[i][j][k] = Long[i][j][m] +1 which is correct
Runtime analysis:
For the reason that this is a 3d array, so it has n^3 subproblems and each problem takes time proportional to in-degree of v; since the total sum of in-degrees is equal to m, the total runtime is $O(n^3 + m) = O(n^3)$.

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)**

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)