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

Homework #3

March 24, 2021

Name: Ryan Coslove *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.

We have array L[1:n]. Sort the array in increasing (non-decreasing) order.

Pick the items with smallest values whose sum is $\leq$ M, i.e. the first W items in the sorted scheduling order π . All remaining elements are added to a counter which will equal the penalty.

Proof of Correctness

Firstly, we can assume that $W < n$; if not, i.e., when $W \geq n$, the greedy algorithm correctly picks every item in the array π (as they all fit together) and so its solutions is clearly correct.

Let $G = \{g_1, g_2, ..., g_W\}$ be the greedy solution. Let $O = \{o_1, o_2, ..., o_W\}$ be any optimal solution to the problem. We will use exchange argument. We exchange o_j with g_j in O to obtain a solution $O', O' = \{g_1, ..., g_j, o_j + 1, ..., o_W\}$

- For all items $i(not\exists)\{g_1, .., g_{j-1}\}, v_{gj} \leq v_i$: we are looking for the minimum value among the items not picked yet
- $o_j(not\exists)\{g_1, ..., g_{j-1}\}$ because $\{g_1, ..., g_{j-1}\} - \{o_1, ..., o_{j-1}\}$ (side note: the command for not exists was not working, sorry)
- By combining the two above we have $v_{gi} \leq v_{oj}$

Now we have proof that our optimized solution is less than the greedy solution. The value of O' is no larger than the value of O. By the end we have entirely exchanged O with G and the value of items in G is at most as large as the ones in O, making G the optimal solution. By choosing the smallest values in the sorted array, we were able to fit the most amount of elements possible that fit into M, thus minimizing the size of the penalty.

Runtime

Using a fast sort algorithm in the beginning like merge sort will take $O(n \log n)$ time and the second step iterating through the list for sum up to value M can be done in $O(n)$ time, making the total time $O(n \log n)$

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.

We will adapt from the breadth-first-search algorithm.

- Initialize an array mark[1:n] with 'FALSE'
- Create a queue data structure Q and insert s into Q.
- While Q is not empty:

1. Let v be the first index of Q and dequeue (remove) this vertex from Q
2. If mark[v] - 'TRUE'; continue from the beginning of the while loop
3. If current v and v+1 are ports, update ticket by +1. If tickets used ≤ 3, can move to next step (4).
4. Otherwise, let mark[v] - 'TRUE' and for $u \exists N(v)$: insert u to the end of Q.

- At the end, return all vertices v with mark[v] - 'TRUE' as the answer.

Proof of Correctness

The proof is close to the proven correctness of a DFS in lecture. By changing the data structure Q from a queue to a stack, the resulting algorithm is identical DFS. The strategy is exactly the same as DFS-TREE by storing the edges that are going from a vertex v to its unmarked neighbor u that we are going to do a DFS over and calling v the parent of u. Then by following the parent of each vertex u repeatedly until we reach s, we obtain an s-u path in G (after reversing the order of this sequence). (This is from lecture)

Runtime

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. As such, the total runtime of the algorithm is at most $O(n) + c * \sum_{v \exists V} |N(v)| = O(n + m)$.

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.

Let f(u) be the number of ways once can trade from node u to destination vertex. We want to use topological sorting to order the vertices $u_1, ..., u_n$ such that any edge can only be directed from a vertex with a lower index to a vertex with a higher index. We are looking to find outgoing edges with in-degree value 0 that lead to value t

- Let D[1:t] be an array initilized to 0 and O be an empty linked-list for the output ordering. Set a second array C[1:n] that will contain all existing topoligcal sorts.
- Go over all vertices v and for any $u \exists N(v)$, increase D[u] by one. (At the end of this step, D[v] denotes the in-degree of v for all $v \exists V$).
- Insert every vertex v with $D[v] = 0$ into a queue Q.
- While Q is not empty:
  1. Let v be the first vertex of Q and dequeue (remove) this vertex from Q.
  2. Add v to the end of the linked-list O.
  3. For $u \exists N(v)$: Reduce D[u] by one. If D[u] = 0 insert u to Q
- If $|O| < t$, output as not a DAG path from s to t; otherwise output O as a topological ordering path from s to t, place this sorted linked list into C[], and increase count by +1.

- Run the while loop again, check if sorted path exists in C. If not, add it to C. Once C[i] = C[i+1], aka no more distinct topological sorts, check count.
- If count % 2 = 0, number of paths is "even". Else, number of paths is "odd".

Proof of Correctness

Suppose first that $|O| = n$: we prove O is a topological ordering from s to t in G in this case:

- Consider any edge (u, v) in G. For the vertex v to be added to O, it should first join the queue Q; forthis to happen, we should have D[v] = 0 at some point.
- For vertex v to have $D[v] = 0$, we should have 'removed' edge (u, v) first: in other words, as long as u is not dequeued, and so we reduce D[v] by one, $D[v] > 0$ and so v will not joinQ. (Here, we are also using the fact that at the beginning D[v] was equal to in-degree of v and we only reduce D[v] by one whenever we add one of the in-neighbors of v to Q).
- This implies that u should have joined O before v, thus appearing before v in the ordering. This implies that if $|O| = n$ (and hence all vertices appear in the ordering), O would be a topological ordering of s to t in G.
- Also, we are checking for duplicate sorted paths as we iterate. If we get a duplicate path, we have found the number of available paths from s to t. Keeping count of each path, we know if the number of paths is even or odd.

Runtime

The first and third line before the for-loop takes $O(n)$ and the second line takes $O(n + m)$ (we go over each edge only once). Also, the while-loop takesat most $O(n + m)$ time because we consider each vertex at most once in the while-loop and when considering each vertex, we go over its out-degree; since sum of the out-degrees is $O(m)$, we obtain the $O(n + m)$ bound.

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.

We can find the longest path using a DAG and applying reduction, turning the matrix into a DAG. To prove it's a DAG, use topological sorting. If the edges are not going from right to left then its a proven DAG.

So first, turn into DAG G = (V,E)

Second, for every vertex $v \exists V$:

dp(v): denote the length of the longest path starting from s and ending in v. By convention, we define dp(s) = 0 and dp(v) = 'undefined' if v is not reachable from start. The output answer is the array that contains dp(v) for every $v \exists V$.

dp(i) = {1

{max $\{dp[i-1][j][k], dp[i][j-1][k]) + 1$

$\{dp[i][j-1][k], dp[i][j][k-1]) + 1$ (sorry for formatting, but the {dp's belong to the max's piecewise function)

Running three nested for loops of A[i][j][k] with the above dp(i) function, we write the cells corresponding to the path output

Proof of Correctness

We have a DAG and we know the longest path algorithm is correct, now the last step is to find the longest path. The longest path of the DAG will eqaul the sequence of valid moves in the matrix. To prove, we pick a sequence of valid moves such as:

(1, 1, 1) = $(i_1, j_1, k_1) -> (i_2, j_2, k_2) ... (i_i, j_i, k_i)$

In this sequence there's always an edge between two vertices so any sequence has valid moves in the DAG. The DAG and the matrix are essentially the same and if there is a one to one mapping ot bijection between paths of the valid moves then the longest path corresponds to the longest move. Because we search for the longest path of the DAG which is equal to the matrix, we can also find the longest sequence of moves, which is the correct answer.

Runtime

We know that step 2 has a runtime of $O(|V| + |E|)$. The number of vertices and the number of edges are n^3 vertices and n^3 edges in the DAG, so the total size of the graph is $O(n^3)$. Total runtime is then $O(n^3)$

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)