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

Homework #3 Solutions

April 1, 2021

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. A complete solution consists of the algorithm, proof of correctness, and runtime analysis.

Algorithm:

1. Sort the job in increasing order of their length using merge sort.
2. Output this ordering as the schedule.

Proof of Correctness: An obvious observation is that minimizing the penalty is equivalent to maximizing the number jobs that are fully processed. So we focus on the second task instead. We now prove the correctness of the algorithm using an exchange argument (there are multiple ways to do an exchange argument for this problem; we simply pick one of them here).

Let $G = \{g_1, g_2, \ldots, g_k\}$ denote the indices of the jobs fully processed by the greedy algorithm, sorted in increasing (non-decreasing) order of their length. Let $O = \{o_1, o_2, \ldots, o_\ell\}$ denote the indices of the jobs fully processed in *some optimal* solution, again sorted in increasing (non-decreasing) order of their length (note that we cannot assume $k = \ell$; this is in fact precisely what we want to prove in the first place).

We now show how to exchange O to G. Let j be the first index where O and G differ, i.e., $g_1 = o_1, \ldots, g_{j-1} = o_{j-1}$ but $g_j \neq o_j$. Since both G and O are sorted in increasing order of their job-lengths, and by definition of the greedy, we know that $L[g_j] \leq L[o_j]$. We replace the order of processing jobs g_j and o_j in O to get a new solution. Now if we denote $O' = \{g_1, \ldots, g_j, o_{j+1}, \ldots, o_\ell\}$ in this new solution; all these jobs will be fully processed because $L[g_j] \leq L[o_j]$ and thus we have

$$\sum_{i=1}^{\ell} L[o'_i] \leq \sum_{i=1}^{\ell} L[o_i] \leq M.$$

Since size of O' is equal to size of O, it is also optimal.

We can thus continue doing the exchange with index $j+1$ and so on until we entirely exchanged O to G, and thus proving that G is also optimal.

Runtime analysis: The algorithm needs $O(n \log n)$ time to sort the input using merge sort.

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

Solution. The algorithm is as before with a simple modification. We first go over the array L and remove any job with $L[i] > M$ as these jobs cannot be processed at all in the time constraint and we should pay a penalty for them anyway. Then, the resulting array consists of jobs with length in $\{1, \ldots, M\}$. We run the previous algorithm but instead of sorting L using merge sort, we sort it using counting sort.

The correctness is exactly as before since the jobs ignored always incur a penalty.

The runtime is now $O(n + M)$ by the runtime of counting sort.

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. A complete solution consists of the algorithm, proof of correctness, and runtime analysis. For the algorithm, we give a graph reduction.

Algorithm (Reduction):

1. Create a graph G with $4n$ vertices and $\leq 4m$ edges as follows:
   - Create four copies of vertices V in the map, named V_1, V_2, V_3, V_4. For any vertex $v \in V$, we use $v_i \in V_i$ to denote the copy of v in V_i for $i \in \{1, 2, 3, 4\}$.
   - For any road (u, v) in the map, add four edges (u_i, v_i), for $i \in \{1, 2, 3, 4\}$, to G.
   - For any shipping route (w, z) in the map, add three edges (w_i, z_{i+1}), for $i \in \{1, 2, 3\}$, to G.
2. Run any graph search algorithm starting from s_1 in G. If the set of vertices reachable from s in G consists of any of the copies of t, i.e., t_1, t_2, t_3, or t_4, output it is possible to reach t from s in the map and otherwise output it is not possible.

Proof of Correctness: As any reduction, the proof consists of two parts:

- If there is a way to reach t from s in the map, then at least one of $\{t_1, t_2, t_3, t_4\}$ is reachable from s_1 in G.

  Proof: Consider the path $P = s, u^1, u^2, \ldots, u^k, t$ in the map. Suppose there are exactly $\ell \leq 3$ shipping routes in this path and they corresponds to edges $(u^{i_1}, u^{i_1+1}), \ldots, (u^{i_\ell}, u^{i_\ell+1})$ in P.

  Then, in G, we can take the path $s_1, u_1^1, \ldots, u_1^{i_1}$ to reach $u_1^{i_1}$ in V_1; then use the edge $(u_1^{i_1}, u_2^{i_1+1})$ to reach vertex $u_2^{i_1+1} \in V_2$; and then continue again until there is another shipping route which now will take us to V_3 and so on. As such, we will eventually end up at t_ℓ and since $\ell \leq 3$, we are done.

- If at least one of $\{t_1, t_2, t_3, t_4\}$ is reachable from s_1 in G, then there is a way to reach t from s in the map.

  Proof: Suppose t_ℓ for $\ell \leq 4$ is reachable from s_1 in G and consider the s_1-t_ℓ path P in G. This path can use at most 3 shipping route edges as each shipping route edge take us from some V_i to V_{i+1} but it can use any number of road edges as they keep us in the same V_i. Thus, if we translate back the edges of this path to the original roads and shipping routes in the map, the corresponding sequence consists of arbitrary number of roads but at most 3 shipping routes; so there is a valid way of reaching t from s in the map following this sequence.

Runtime analysis: Creating the graph takes $O(n+m)$ time as each vertex and edge of the original map is translated to $O(1)$ vertices and edges in G. Solving graph search can be done using DFS/BFS in $O(n+m)$ time also.

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

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

- $Path(v)$: denote the number of paths starting from vertex v in G mod 2, i.e., it is 1 if the number of paths is odd, and 0 if it is even.

The output answer is *odd* if $Path(s) = 1$ and otherwise it is *even*.

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

$$Path(v) = \begin{cases} 1 & \text{if } N(v) = \emptyset \\ (\sum_{u \in N(v)} Path(u)) \mod 2 & \text{otherwise} \end{cases}.$$

Proof of correctness:

- The base case for $Path(v) = 1$ for any v with $N(v) = \emptyset$ is true because the only path starting from v is the path consists of a single vertex v.
- Consider any vertex $v \in V$ which not a sink (i.e., $N(v) \neq \emptyset$). Any path starting from v, goes to one of the neighbors u of v in $N(v)$ and then continues from there; so if there $Path(u)$ paths starting from u, there will be so many paths starting from v with u as their second vertex; this means in total the number of paths starting from v will be $Path(u_1) + Path(u_2) + \ldots$ for $u_1, u_2, \ldots \in N(v)$.

Algorithm: We can turn this recursive formula into a dynamic programming algorithm using memoization exactly as in previous dynamic programming algorithms—we omit the details.

Runtime: 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. A complete solution consists of the algorithm, proof of correctness, and runtime analysis. For the algorithm, we give a graph reduction.

Algorithm (Reduction):

1. Construct a directed graph $G = (V, E)$ from the matrix A as follows:
   - For each cell (i, j, k) in the matrix, make a vertex $v_{i,j,k}$.
   - Add a directed edge from $v_{i,j,k}$ to $v_{i',j',k'}$ if and only if it is possible to move from cell (i, j, k) to cell (i', j', k') according to the rules given by matrix A.
2. Compute the longest path starting from $v_{1,1,1}$ in G using the algorithm for longest path in a DAG (in lecture 8). Output the length of this path.

Proof of correctness: To show that the algorithm is correct we need to show that G is a DAG and the length of the longest path in G starting from $v_{1,1,1}$ is the same as the longest sequence of moves to go from cell $(1, 1, 1)$ to some other cell.

- **Part one:** G is a DAG.

  To show that G is a DAG, we can simply give a topological ordering of its vertices (recall that a directed graph is a DAG if and only if it admits a topological ordering). The ordering is as follows:

$$v_{1,1,1}, v_{1,1,2}, \ldots, v_{1,1,n}, v_{1,2,1}, \ldots, v_{1,2,n}, \ldots, v_{1,n,n}, v_{2,1,1}, \ldots, v_{2,n,n}, \ldots, v_{n,n,n}.$$

  More formally, the ordering starts with $v_{1,1,1}$, and then increase the third coordinate one by one until it reaches $v_{1,1,n}$; then it rest the third coordinate and increase the second one, and continues by increasing the third coordinate as before. Once, we have $v_{1,n,n}$, we reset coordinates two and three, increase the first coordinate to $v_{2,1,1}$, and continue like this until we reach $v_{n,n,n}$.

  The reason this is a topological ordering of G is because any edge in G would either the first or second coordinate (it if corresponds to a 0 in A), or second or third coordinate (if it corresponds to a 1 in A); so every edge goes from left to right in this ordering. This proves that G is a DAG.

- For every sequence of moves $(i_1, j_1, k_1), (i_2, j_2, k_2), \ldots (i_t, j_t, k_t)$ in the matrix, there is a corresponding path $v_{i_1,j_1,k_1}, v_{i_2,j_2,k_2}, \ldots v_{i_t,j_t,k_t}$ in G and vice versa. This means that length of the longest path in G from $v_{1,1,1}$ is the same as the longest sequence of moves to go from cell $(1, 1, 1)$ to any other cell.

Thus we have shown that our algorithm will always give the correct answer.

Runtime analysis: G has n^3 nodes and each node $v_{i,j,k}$ has at most two edges going out of it according to the rules (to $v_{i+1,j,k}$ or $v_{i,j+1,k}$ or $v_{i,j,k+1}$) thus the number of edges is $O(n^3)$. Thus construction of G takes $O(n^3)$ time. The algorithm for longest path in a DAG runs in time linear in the number of nodes and edges. Thus this algorithm on G should take time $O(n^3)$. Both steps take time $O(n^3)$ thus the runtime is $O(n^3)$.
