CS161 Homework 5 **Due:** 20 May 2016
Submit on Gradescope **Handed out:** 13 May 2016

Instructions: Please answer the following questions to the best of your ability. If you are asked to design an algorithm, please describe it clearly and concisely, prove its correctness, and analyze its running time. If you are asked to show your work, please include relevant calculations for deriving your answer. If you are asked to explain your answer, give a short ($\sim$ 1 sentence) intuitive description of your answer. If you are asked to prove a result, please write a complete proof at the level of detail and rigor expected in prior CS Theory classes (i.e. 103). When writing proofs, please strive for clarity and brevity (in that order). Cite any sources you reference.

1 (14 points) Poe Selling Fish

Poe the Penguin is setting up a business to sell fish to his fellow penguin friends. He predicts the quantity of sales to expect over the next n days. Let d_i denote the number of sales he expects on day i. Poe procures fish from Barr the Bear in batches, and Barr requires a fixed fee of K each time Poe requests a batch (regardless of the number of fish Poe requests; also Poe doesn't need to pay for the fish, just the fee). We will assume that Poe gets a batch of fish (if any) in the morning, and all sales happen in the afternoon. Any fish that are not sold are stored until the next day. Poe can store at most S fish, and it costs C to store a single fish for a day. Poe starts out with no fish on day 1.

In this problem we will design a dynamic programming algorithm that decides how to place requests so that Poe satisfies all the demands d_i, of his penguin friends and minimizes his costs. (Assume that S, C, K, d_i's are all positive integers.)

(a) (2 points) Let the subproblem represent the optimal way to satisfy demands for days 1 to i with a total of s fish left over after day i. Let $M[i, s]$ denote the cost of an optimum solution to this subproblem (but not including the storage fee of the unsold fish on day i, if any). What values of i and s should we plug in for the problem we want to solve? How many subproblems do we have in total?

Solution:

$M[n, 0]$ is what we want (so $i = n$ and $s = 0$). Note that $M[n, s]$ for any $s \neq 0$ is not the answer because Poe will have to pay a storage fee forever (as there will be no sales after day n).

The number of subproblems is $n \times (S + 1)$ as $i \in [1, n]$ and $s \in [0, S]$. Depending on what base case(s) you choose, this could be larger/smaller, but so long as it is $O(nS)$, it's OK.

(b) (2 points) Let us try to solve a subproblem $M[i, s]$ using the values of the previous subproblems $M[i-1, z]$ where $z \in [0, S]$. $M[i - 1, z]$ is the total cost incurred by the end of day $i - 1$, and Poe will have to pay zC for storage (if $z = 0$, this will be 0). On day i, recall that Poe needs to sell d_i fish and have s fish remaining after the sale. Let us consider two cases as follows.

(i) If $s + d_i > S$, then Poe must request a batch on day i (regardless of what z is because $z \leq S$). In this case, we can solve $M[i, s]$ as follows:

$$M[i, s] = \min_{z \in [0,S]} (M[i - 1, z] + zC + K)$$

(ii) If $s + d_i \leq S$, then depending on what z is, Poe may not need to order fish (to avoid the fee):

$$M[i, s] = \min \left(\min_{z \in [0, s+d_i-1]} (M[i - 1, z] + zC + K), (M[i - 1, s + d_i] + (s + d_i)C) \right)$$

What should the base cases be? If we use this to solve the problem, what would be the runtime of this algorithm?

Solution:

$M[0,0] = 0$ and $M[0,s] = \infty$ (for $s > 0$) should be the base cases as Poe begins with no fish on day 1 (meaning, no fish left after day 0). Alternatively, one can define $M[1,s]$ as the base cases; in this case you should specify that $M[1,s] = K$ for all $s \in [0,S]$ (note that $d_1 > 0$ as stated in the problem description so Poe must order some fish).

The size of the table (or the number of subproblems to be solved) is $O(nS)$; filling in each $M[i,s]$ can be done in time $O(S)$ as in both cases above it iterates over a range of z whose size is $O(S)$. The overall runtime is $O(nS^2)$.

(c) (7 points) It turns out that we can optimize the algorithm presented above, and get a faster algorithm. In both cases shown above, we can show that $z = 0$ achieves the minimum (where min is computed over a range of z). Prove this.

Solution:

For simplicity assume $M[i,s] = \infty$ for all i if $s > S$ (since Poe cannot store more than S fish anyway). With this simplification, we can merge the two cases from previous part (regardless whether $s + d_i > S$ or not):

$$M[i,s] = \min\left(\min_{z\in[0,s+d_i-1]}(M[i-1,z] + zC + K), (M[i-1,s+d_i] + (s+d_i)C)\right)$$

(Note that this simplification does not require us solving more sub-problems, as we can just check this condition by an if-statement on z and s.)

We now prove by induction on i that $M[i,s] \geq M[i,s-1]$ for all $s > 0$.

Base case: When $i = 0$, as we defined earlier $M[0,0] = 0$ and $M[0,s] = \infty$ for all $s > 0$, so $M[i,s] \geq M[i,s-1]$ for all $s > 0$.

Inductive step:

By inductive hypothesis, we have $\min_{z\in[0,s+d_i-1]}(M[i-1,z]+zC+K) = M[i-1,0]+K$, which simplifies the above expression to:

$$M[i,s] = \min\left(M[i-1,0] + K, M[i-1,s+d_i] + (s+d_i)C\right)$$
$$M[i,s-1] = \min\left(M[i-1,0] + K, M[i-1,s-1+d_i] + (s-1+d_i)C\right)$$

By inductive hypothesis we know $M[i-1,s+d_i] + (s+d_i)C \geq M[i-1,s-1+d_i] + (s-1+d_i)C$, so we conclude $M[i,s] \geq M[i,s-1]$ for all $s > 0$.

(d) (3 points) Let us define $M[i,s] = \infty$ if ($i > 0$ and $s > S$) or ($i = 0$ and $s > 0$) and define $M[0,0] = 0$. Fill in the blanks below using Part (b) and (c). If we use this to solve the problem, What is the runtime of the new algorithm?

$$M[i,s] = \min\left(\boxed{}, \boxed{}\right).$$

Solution:

$$M[i,s] = \min\left(M[i-1,0] + K, M[i-1,s+d_i] + (s+d_i)C\right).$$

The number of subproblems did not change, but now it requires $O(1)$ time to fill in each $M[i,s]$. The overall runtime is $O(nS)$.

2 (14 points) Fun with Walks

Consider a weighted, directed graph G with n vertices and m edges that have integer weights. A graph walk is a sequence of not-necessarily-distinct vertices $v_1, v_2, \ldots, v_k$ such that each pair of consecutive vertices v_i, v_{i+1} are connected by an edge. This is similar to a path, except a walk can have repeated vertices and edges. The length of a walk in a weighted graph is the sum of the weights of the edges in the walk. Let s, t be given vertices in the graph, and L be a positive integer. We are interested counting the number of walks from s to t of length exactly L.

a) (5 points) Assume all the edge weights are positive.

Describe an algorithm that computes the number of graph walks from s to t of length exactly L in $O((n+m)L)$ time. Prove the correctness and analyze the running time.

Solution:

If we let $x_{i,l}$ be the number of walks of length l from s to i, then $x_{i,l} = \sum_{j:(j,i)\in E, w(j,i)\leq l} x_{j,l-w(j,i)}$. We can solve this recurrence using dynamic programming as follows.

```
1: x[i][l] = 0 for i = 1,...n; l = 0,...,L
2: x[s][0] = 1
3: for l = 1,...,L do
4:     for i = 1,...n do
5:         for j : (j,i) ∈ E, w_{j,i} <= l do
6:             x[i][l] += x[j][l - w_{j,i}]
7:         end for
8:     end for
9: end for
10: return x[t][L]
```

The loop in line 4 iterates over each vertex's outgoing edges, and hence takes $O(n+m)$ time, and executes L times, so the total time complexity is $O((n+m)L)$.

b) (9 points) Now assume all the edge weights are non-negative (but they can be 0), but there are no cycles consisting entirely of zero-weight edges. That is, for any cycle in the graph, at least one edge has a positive weight.

Describe an algorithm that computes the number of graph walks from s to t of length exactly L in $O((n+m)L)$ time. Prove correctness and analyze running time.

Solution: The algorithm is identical to Part (a) except with one important difference: since edges can have length 0, if there is an edge of length 0 from u to v, then we have to be careful to process u before v when iterating over the vertices on line 4. This ensures that the $x_{u,l}$ term in RHS of the expression $x_{v,l} = \sum_{j:(j,v)\in E, w(j,v)\leq l} x_{j,l-w(j,v)}$ is computed before we compute $x_{v,l}$. To do this, we consider the subgraph of G consisting only of edges with weight 0. This is a directed acyclic graph, so we can perform a topological sort on this graph, and iterate over the vertices in topological order. The cost of the topological sort is only $O(n+m)$, so the total time complexity is still $O((n+m)L)$.

3 (12 points) Shortest and Longest Paths

Consider a (directed) grid network $G = (V, E)$ in which each node is labeled as $v_{i,j}$ where $1 \leq i \leq r$ and $1 \leq j \leq c$, and there is an edge from $v_{i,j}$ to $v_{i,j+1}$ if and only if $j < c$ and an edge from $v_{i,j}$ to $v_{i+1,j}$ if and only if $i < r$. (That is, the edges go from left to right and top-down to adjacent grid points.) For each edge e in the grid network let w_e denote its weight. We will assume that w_e's are integers which can be positive or negative.

For example, Figure 1 shows a grid network when $r = 4$ and $c = 5$ (weights are not shown).

We will use this definition of grid networks in the sub-problems of this question.

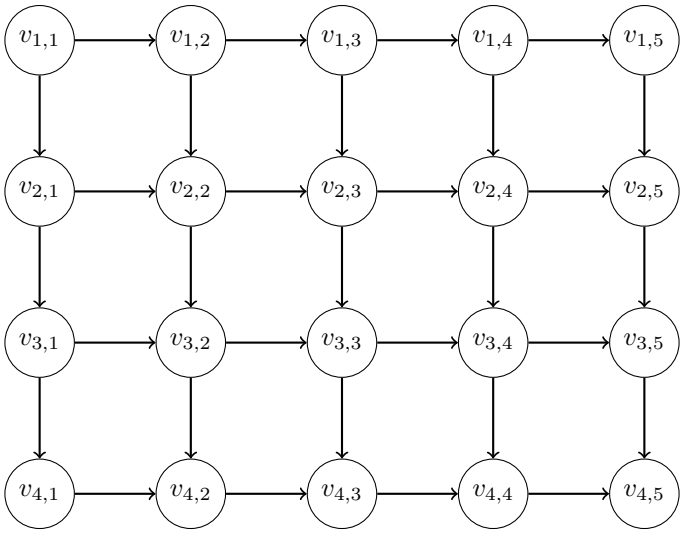

Figure 1: Example: Directed Grid Network of Size 4×5.

(a) (6 points) We want to compute the distance from $v_{1,1}$ to $v_{r,c}$.

Explain how we can use Dijkstra's algorithm to solve this problem. (Recall that edge weights can also be negative.) Prove its correctness, and analyze its running time. Assume that Dijkstra's algorithm is implemented with Fibonacci Heaps.

Solution:

Any path from $v_{1,1}$ to $v_{r,c}$ contains exactly $(r-1)+(c-1)$ edges. To see why, an edge coming out of $v_{i,j}$ increases the index of i by 1 or the index of j by 1 (and those are the only edges), and therefore any path from $v_{1,1}$ to $v_{r,c}$ must contain $(r-1)+(c-1)$ edges.

We can use the Negative-Dijkstra algorithm from previous homework (which re-weighs the weights of edges to make them non-negative) to compute the shortest path. The runtime of the algorithm is $O(m + n \log n)$ (due to Negative-Dijkstra); in this problem $n = m = O(rc)$, and we have $O(rc \log(rc))$.

To see why this works: Suppose that P is the shortest path found by the Negative-Dijkstra, and for contradiction assume that P' is a shortest path with smaller distance than P in the original grid network. That is, $d(P') < d(P)$ in the original grid network where $d(\cdot)$ is the distance of the given path. If the Negative-Dijkstra subtracted w^* from every edge (where w^* is the minimum edge weight), then we know that in the modified graph the new distance of P' is $d(P') - w^*((r-1)+(c-1))$ and the new distance of P is $d(P) - w^*((r-1)+(c-1))$. Since the new graph has no negative edge weight and due to Dijkstra's correctness, we know that $d(P) - w^*((r-1)+(c-1)) \leq d(P') - w^*((r-1)+(c-1))$, which is a contradiction since $d(P') < d(P)$. Hence the path found by the Negative-Disjktra is a shortest path in the original graph.

(b) (6 points) We want to compute the longest path among all paths in this grid network.

Design an algorithm that runs in $O(rc)$ time and finds the longest path. Prove its correctness and analyze its running time. (Recall that edge weights can be negative.)

Solution:

Let $d[i,j]$ denote the distance of a longest path that ends at $v_{i,j}$. To compute $d[i,j]$, we first find a topological ordering of the vertices in the grid network which runs in time $O(rc)$ as there are $O(rc)$ edges (notice that the grid network is a DAG, and thus a topological ordering exists). To compute $d[i,j]$: $d[i,j] = \max(0, d[i-1,j] + w(v_{i-1,j}, v_{i,j}), d[i,j-1] + w(v_{i,j-1}))$ where $d[0,j]$ and $d[i,0]$ are assumed to be $-\infty$ (consider them as base cases). We need to compute $O(rc)$ entries of $d[]$, and filling in each one can be done in $O(1)$ time. We then return $\max_{i,j} d[i,j]$ as the distance of the longest path in the network. To find an actual path, when we compute $d[i,j]$, we record its predecessor $\pi(i,j)$ depending

on where the max-value comes from (if it is equal to 0, then $\pi(i,j) = \text{NIL}$). If $d[i^*, j^*]$ is the maximum among all $d[]$, we can construct the path by following the predecessors until we hit NIL (similar to what we did for Dijkstra's algorithm).

To prove correctness, we prove it by induction. Assume that $d[]$ has been computed correctly for all vertices that appear before $v_{i,j}$ in the topological ordering we use (the base case being $d[1,1] = 0$, which is correct). In particular, $d[i-1,j]$ is the longest path distance ending at $v_{i-1,j}$ (if $i - 1 \geq 1$) and $d[i, j-1]$ is the longest path distance ending at $v_{i,j-1}$ (if $j - 1 \geq 1$). Since any path ending at $v_{i,j}$ must be either a path of no edges, a path containing $(v_{i-1,j}, v_{i,j})$ as the last edge, or a path containing $(v_{i,j-1}, v_{i,j})$ as the last edge, correctness follows immediately as we compute $d[i,j] = \max(0, d[i-1,j] + w(v_{i-1,j}, v_{i,j}), d[i,j-1] + w(v_{i,j-1}))$.

4 (12 points) Poe Travels with Clones of Barr

Poe the Penguin is a mad scientist, and recently invented a machine that produces clones of Barr the Bear from South Pole dark matter. Poe has produced n clones of Barr the Bear (let us label them as $1, 2, \ldots, n$). Poe wishes to travel the universe with *exactly k clones* as his spaceship has $k+1$ seats (one for himself and k for the clones – Poe does not want to have empty seats!). The machine has a minor flaw in that some clones, if they are separated, would explode, and put an end to the universe. Luckily Poe knows which pairs of clones need to be kept together; let $E = \{(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)\}$ be a set of pairs of clones where each pair (i, j) indicates that clone i and clone j should not be separated; that is, Poe must either take both i and j with him on his spaceship or take neither of them. Can you help Poe find out whether it is possible to choose exactly k clones and take them with him, or Poe should just stay home while regretting his latest invention?

Design an algorithm that takes (n, k, E) as an input, and determines in $O(m + nk)$ runtime which k clones Poe should take with him (or output "S.T.A.Y." if Poe should not travel). Prove correctness of your algorithm and analyze its running time. (You may assume that $k \leq n$ because Poe only has n clones and thus cannot travel with exactly k clones if $k > n$.)

Solution: Formulate this as a graph problem where clones are vertices and each pair in E is an edge. Let there be x connected components (CCs) in this graph whose sizes are $c_1, c_2, \ldots, c_x$ (note that $x \in [1, n]$ and $\sum_{i \in [1,x]} c_i = n$). We first run DFS or BFS to find all connected components (mainly, to know of their sizes) in time $O(n + m)$. It is clear that all clones in a connected component (CC) must be taken together or left together (otherwise, by definition of connectivity, there will be at least one pair of clones that would explode).

Now define $D[i,j]$ such that $D[i,j] = 1$ if there exists a subset of CCs from $\{c_1, c_2, \ldots, c_i\}$ such that the sum of their sizes is equal to j, and $D[i,j] = 0$ otherwise. For simplicity assume $D[i,j] = 0$ if $j < 0$ or $j > k$ (which can be taken care of by an if-statement instead of declaring a DP table) and also assume $D[i,0] = 1$ (as we can always choose no clones). Then $D[x,k]$ contains the answer we seek, and the DP table we need is of size $O(nk)$ as $x = O(n)$.

For base cases: Let $D[0,0] = 1$ and $D[0,j] = 0$ for all $j \in [1,k]$. Then we compute $D[i,j] = \max(D[i-1,j], D[i-1,j-c_i])$ where $D[i-1,j'] = 0$ if $j' < 0$ (this can be handled by an if-statement, for instance); equivalently, instead of max, one can use logical-or of the two. We compute $D[i,j]$ from $j = 1$ to k in increasing order before we compute $D[i+1,\cdot]$, and so on.

To prove correctness: We prove by induction on i. Base case is clear when $i = 0$ for all j. Let us prove that $D[i,j]$ is computed correctly for all j when $D[i-1,j']$'s are correctly computed for all j'. If $D[i,j] = 0$ then we know that $D[i-1,j] = 0$ and $D[i-1,j-c_i] = 0$. By inductive hypothesis, this means that no subset of $\{c_1, c_2, \ldots, c_{i-1}\}$ has the sum of sizes equal to j or $j - c_i$. Hence if there exists a subset of $\{c_1, c_2, \ldots, c_i\}$ whose sum of sizes is equal to j, call the subset S, then $c_i \in S$ leads to a contradiction (as $S \setminus \{c_i\}$ would have sum of sizes equal to $j - c_i$) and $c_i \notin S$ leads to a contradiction (as S would have sum of sizes equal to j). Otherwise when $D[i,j] = 1$ then we know that $D[i-1,j] = 1$ or $D[i-1,j-c_i] = 1$ or both. If $D[i-1,j] = 1$, then the subset of $\{c_1, c_2, \ldots, c_{i-1}\}$ whose sum of sizes is equal to j is also a subset of $\{c_1, \ldots, c_i\}$, and thus $D[i,j] = 1$ is correct. If $D[i-1,j-c_i] = 1$, then we can add c_i to the subset of $\{c_1, c_2, \ldots, c_{i-1}\}$ whose sum of sizes is equal to $j - c_i$ to obtain a subset whose sum of sizes is equal to j.

Since we wish to find a subset of CCs whose sum is equal to k, we return "S.T.A.Y" if and only if $D[x, k] = 0$. Otherwise, we can construct a subset as follows: When computing $D[i, j]$, we remember which one of $D[i-1, j]$ and $D[i-1, j-c_i]$ is equal to 1 (if both are, then either one is fine); this is the 'predecessor' equivalent. We begin with the empty set S at $D[x, k]$. As we trace back from $D[i, j]$ to $D[i-1, j]$ we do nothing to S; if we trace back from $D[i, j]$ to $D[i-1, j-c_i]$, then we add all clones in the i-th component to S. We stop tracing back when we reach $i = 0$ or $j = 0$, and output S. Since i always decreases this process terminates (and j sometimes decreases, but never goes below 0).