CIS 675 Fall 2018 Midterm 2 Practice Problems Solutions

Note: Some of these solutions are only *sketches*: I want you to practice writing out the full solutions!

1. Suppose that you are given a directed, weighted graph G with both positive and negative edges. However, all of the negative edges are *leaving* the source vertex s- all other edges are positive (and some edges leaving the source vertex s may also be positive). Additionally, there are no negative cycles. If you run Dijkstra's Algorithm on this graph, will you get the correct answers? Why or why not?

Solution Sketch: Yes, we will get the correct answers. The reason Dijkstra's can't handle negative edges in general is due to the order of processing nodes- if it adds a node to the known region (i.e., receives that node from the delete-min operation), but later finds a shorter distance to that node, it never goes back and 'fixes' the distance to neighbors of that node (i.e., it won't 're-process' that node). In the graph described in the problem, think about what order the neighbors of s are processed- is it possible we'll ever find a shorter path to a specific neighbor? Why or why not?

2. Suppose that you have a directed graph, but instead of having weights on the *edges*, it has positive weights on the *nodes* (you can think of these weights as representing, e.g., tolls to enter a city). How would you modify Dijkstra's Algorithm to find shortest paths in this case?

Solution Sketch: This is a very simple change to Dijkstra's Algorithm- the only change we need to make to the pseudocode is to the dist calculations. What is the new distance to a node in this case?

3. There is a set of n employees $E_1, ..., E_n$, and a set of tasks $T_1, ..., T_n$. Each employee must be assigned to one task, and every task must be assigned to an employee. Each task T_i has a difficulty level d_i, and each employee has a skill level s_i. The goal is to match employees to tasks such that the average difference between each employee's skill level and his or her task's difficulty level is minimized.

In other words, if employee E_i is matched to task $T_{f(i)}$, we wish to minimize

$$\frac{1}{n}\sum_{i=1}^{n}|s_i - d_{f(i)}|$$

.

Part a: Consider the algorithm that begins with with employee E_1, and assigns that employee the task with the difficulty level that is closest to E_1's skill level. Remove E_1 and the selected task from the pool, and assign employee E_2 the task with difficulty level closest to E_2's skill level. Repeat until all employees and tasks are matched. What is the running time of this algorithm? Give a counterexample showing when this method fails to find the optimal solution.

Part b: Describe an $O(n \log n)$ algorithm to find the optimal solution.

Solution: Part a: This algorithm will not work. For example, suppose the difficulty levels of the tasks are $d_1 = 3, d_2 = 5$, and the employees have skill levels $s_1 = 3.1, s_2 = 2.8$. Then this algorithm assigns employee 1 to task 1, and employee 2 to task 2. This has a total cost of 2.3. A better solution would be to assign task 1 to employee 2, and task 2 to employee 1, which has a cost of 2.1.

Part b: Sort the employees by skill level, and sort the tasks by difficulty level. Then assign the least skilled worker to the easiest task, the second-least skilled worker to the second-easiest task, etc.

4. Suppose you are organizing a party, and are deciding who to invite. The rule you are following is that everyone at the party must know at least k other people, and must be a stranger to at least k other people. As input, you are given a list of who knows whom (if a pair of people is not on the list, that means that they are strangers to each other). Design an algorithm to identify a maximal set of people to invite.

Solution Sketch: This can be solved by using a greedy algorithm, but instead of building the solution one piece at a time, remove pieces from the solution one at a time. In the first step, remove everyone who doesn't meet the requirements. This affects whether other people meet the requirements,

so in the second step, remove them. Proceed until everyone remaining meets the requirements.

5. You are scheduling a set of jobs on a CPU. Only one job can run on the CPU at a time. Each job j_i requires some amount of time t_i, and has a deadline d_i, which is the time that the job must be finished by (assume that the current time is 0). Your goal is to schedule jobs on the CPU so that every job finishes by its deadline.

For example, there may be 3 jobs j_1, j_2, j_3 that have deadlines of $10, 2$, and 3, and take $5, 1$, and 2 seconds each to run. In the above example, you could schedule job j_2 at time 0, so it finishes at time 1. Job j_3 could then start at time 1, and finish at time 3. Then job j_1 could start at time 3 and finish at time 8. In this way, all jobs finish by their deadlines. There are other solutions as well.

Design a greedy algorithm to determine what time each job should start. The output of your algorithm should be an array $[s_1, s_2, ...s_n]$, where s_i represents the starting time for job j_i. If it is not possible for every job to finish before its deadline, then your algorithm should output this.

Solution Sketch: The optimal solution is to schedule the job with the earliest deadline first. In other words, sort the jobs by deadline (earliest to latest), and schedule jobs in that order. Each job should be scheduled to start as soon as the previous job finishes. If some job finishes after its deadline, then the algorithm will output that it is not possible to successfully schedule each job. Otherwise, output the start times of each job.

6. In class, we solved the maximum contiguous subsequence problem. Here, you must solve the maximum *non-contiguous* subsequence problem, as follows: Suppose you are given a sequence $S = \{a_1, a_2, ..., a_n\}$ of values. You want to select values from this sequence that give you the greatest sum, subject to the rule that you cannot pick two elements that are adjacent to one another in the sequence (so, for example, if you pick a_3, you cannot pick a_2 or a_4.) Your algorithm should return the sum of the maximum non-contiguous subsequence, but you do not need to return the indices in the sequence.

Solution Sketch: Part (a): The nodes represent the array indices $1...n$.

Draw an edge from i to j if $j > i + 1$. **Part (b):** Define $M(j)$ to be the sum of the maximum non-contiguous subsequence ending at j. Then $M(j) = \max\{\max\{M(i) + S[j] : j < i - 1\}, S[j]\}$. For $j = 1, 2$, $M(j) = S[j]$. Fill in these $M(j)$ values beginning with 1 and working up. Then finally, scan over all M values and see which is the largest.

7. Suppose you are trying to cross a river. You begin on the left side of the river and are trying to get to the right side. There is a set of n stones $R_1, ..., R_n$ that you can step on to help you cross. These stones are in a straight line, and in order of their index (that is, stone R_1 is closest to the side you are on, and stone R_2 comes next, etc.). Each stone R_i is a distance d_i from the left side of the river. Assume all distances d_i are whole numbers. Each stone R_i has a height h_i. Assume that your starting location on the left side of the river is at distance 0 and height 0. The right side of the river is your destination, and is at location M and has height 0.

When crossing, you must obey the following rules: (1) You can only jump to a stone that is no more than 10 units of distance from your current location. (2) You can only jump to a stone no more than 2 units of height, above or below, from your current location (e.g., if you are at height 6, you can jump to a stone between heights 4 and 8). (3) You cannot jump to a stone behind your current location.

Additionally, there is a *success probability* associated with jumping from one stone to another, which measures the probability of successfully making the jump. When jumping from a stone R_i to another stone R_j, this probability is calculated as $\frac{1}{d_j - d_i}$. Thus, the farther apart the stones are, the lower chance of success. If one jump fails to succeed, the whole river crossing fails to succeed.

Design a dynamic programming algorithm to determine how to maximize your probability of making a successful crossing. You do not need to report the actual stones used: just the total probability. Remember the rules of probability: if event A has probability p and event B has probability q, the probability that both A and B occur is given by pq.

Solution Sketch: Sort the stones by d_i, and let each node represent a stone. Add a node at the beginning to represent the left bank, and a node

at the end to represent the right bank. Draw an edge from node i to node j if the distance between d_i and d_j is less than or equal to 10, and if they are within 2 units of height of each other. Then initialize $P(0) = 1$. Set $T(j) = max(\frac{1}{d_j - d_i} T(i)$, where the max is over all nodes i such that (i, j) is in the DAG. Then return T(last node).

8. Prove that the edit distance between two strings is the same as the edit distance between the reverse of the two strings.

Solution Sketch: Think about what edit distance means, and the operations/costs allowed when transforming the two strings. Are any of these sensitive to whether you look at the strings in their original order or in reverse?

9. There are n students in the CS program, and each of them must take the Algorithms course. There are m sections of Algorithms offered: $A_1, ..., A_m$. They begin at times $T_1, ..., T_m$. Each student can sign up for at most one section. Because students also have other courses to take, each student can only attend some subset of the Algorithms sections. This varies by student: some may only be free during one or two sections, while others might be able to attend all of them. In addition, each section A_i has a cap C_i on the total enrollment, so you cannot put more than C_i students in that section. Your goal is to assign students to sections so that as many students as possible are enrolled in a section of Algorithms (note that it might be impossible to enroll *everybody* into a section: this is fine).

Design an algorithm to solve this problem by using network flows. Be precise about the structure of the graph. It is sufficient to explain how to create the graph, and then say to apply the existing network flow algorithm to the graph.

Solution Sketch: Create a bipartite graph in which the students are on the left and the sections are on the right. Draw an edge from a student to a section if the student can attend that section. Each of these edges should have capacity 1. Create a source node. Draw an edge from the source node to each student, with capacity 1. Create a sink/target node. Draw an edge from each section to the sink node, with capacity equal to the maximum

capacity of that section (C_i). Find the maximum flow on this graph. Look at which edges between student-nodes and section-nodes have 1 unit of flow along them; these edges correspond to the assignment of students into sections.

10. Suppose that in some country, there is a set of cities $C_1, ..., C_n$. The water for these cities all comes from the same lake. Some pairs of cities are connected by water pipes (for example, C_1 might be connected to C_2, but not necessarily C_3). In addition, the lake is connected by pipes to some of the cities (but not necessarily all). You are given a list of the pairs of cities that are connected to each other, and a list of cities that are connected to the lake.

The pipes are of different sizes, and pipe P_j can carry w_j units of water per day. The pipes are directed: for instance, a pipe that connects C_1 to C_2 can carry water from C_1 to C_2, but not from C_2 to C_1 (though there may be a separate pipe that carries water from C_2 to C_1).

The cities have different populations, so require different amounts of water per day. Assume that city C_i requires d_i amount of water per day. Also assume that due to limited rainfall, the lake can only release K units of water per day.

Design a network flow-based algorithm to determine whether it is possible, given the current system of pipes and the limitation on how much water the lake can release every day, for each city to get as much water as it requires.

Solution: This is a network flow problem with multiple sinks. To solve it, create a graph where the nodes are the cities and the lake, and directed edges with capacities represent the amount of flow that can be sent between two cities, or the lake to a city. These capacities are determined by the w_j values (pipe capacities). Then create a 'super-sink' node. Add a directed edge from each city to the 'super-sink', with capacity equal to the demand from that city. Also create a new source node, with a single edge going from the source node to the lake node. This edge has capacity K. Run the network flow algorithm. If it is able to send $d_1 + ... + d_n$ units of flow to the super-sink node, then all demands can be satisfied.