Problem Set 5

All parts are due Tuesday, April 29 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 5-1. [15 points] **Hobbits' Gold**

Your company, Mathematically Intricate Technologies, is branching out into the video game business, since they have not been very successful at selling mathematical educational software or statistical software for the movie industry. Their first product will be a blockbuster game called "Hobbits' Gold". This involves guiding your hobbit character through a maze which is a connected DAG, with designating starting point s and destination (target) t that is reachable from s. We assume that t is the only "sink" in the graph, that is, the only node without any outgoing edges.

Since the maze is a DAG and t is the only sink, just traversing any edges starting from s would get your hobbit to t. However, there is a complication. Each edge may have a pile of gold coins on it, which your hobbit may load into his/her rucksack, or may have a toll gate requiring deposit of some number of gold coins, or both. He/she starts out with an empty rucksack.

By playing the game numerous times, you manage to collect complete information about the DAG, including the sizes of all the piles and the tolls for all the gates.

(a) [7 points]

Assume that the gates accept IOUs as well as coins, so your hobbit can have a negative net number of coins at any point during the traversal, or even at the end. Design an algorithm (use very clear English or pseudocode, as usual) that outputs a path from s to t that your hobbit can traverse to wind up at the destination, and that gives him/her the largest possible net number of gold coins at the end (this may be positive or negative). Your algorithm should run in $O(|V| + |E|)$ time. Include an analysis.

(b) [8 points]

Now suppose that the gates do not accept IOUs, so your hobbit must always maintain a nonnegative number of coins in his/her rucksack. Design another algorithm that outputs a path from s to t that your hobbit can traverse to wind up at the destination if that is possible, and that gives him/her the largest possible number of gold coins at

the end. If this is not possible (because your hobbit cannot collect enough gold coins to get through a gate), your algorithm should say that. Your algorithm should run in $O(|V| + |E|)$ time. Include an analysis.

Problem 5-2. [25 points] **Dijkstra Properties and Proofs**

Here we will develop a slightly different proof for the Dijkstra algorithm from the one in CLRS, and consider some other Dijkstra-related issues.

Notation: S is the subset of V consisting of nodes that have already been processed, and $Q = V - S$. Recall that $\delta(s, v)$ is the minimum length of a path from s to v in the graph. At any point in the algorithm between iterations of the loop, define $\delta_S(s, v)$ to be the minimum length of a path from s to v in which all nodes, except possibly the final node v, are in S. (As a special case, $\delta_S(s, s) = 0$.) If there is no such path, then define $\delta_S(s, v)$ to be ∞.

Correctness (safety, that is) for Dijkstra's algorithm follows immediately from the following two-part invariant, which is true after any number of iterations of the loop:

1. If $u \in S$ then $d[u] = \delta(s, u)$.
   That is, all nodes in S record their correct distances.
2. If u is any node in V, then $d[u] = \delta_S(s, u)$.
   That is, every node in the entire graph records its best distance along a path through S.

Part 1 of the invariant in the final state immediately implies that all the outputs are correct.

In Parts (a)-(d) of this problem, you will prove that this two-part invariant holds, using induction on the number of loop iterations. (Part (e) asks a different, but related question.) Your proof of the inductive step should use just the fact that the invariant holds before the iteration plus the behavior of the algorithm during the current iteration. It should not need to deal with what happened before this iteration.

We will help you along. In proving any part of this problem, you are allowed to use the statements of the previous parts. For example, you can assume the statements of parts (a), (b), and (c) when proving part (d).

(a) [3 points]

For the base of the induction, prove that both parts of the invariant hold after 0 loop iterations.

Now we move on to the inductive step. For Parts (b)-(d), consider a particular iteration, in which some particular node w is moved from Q to S. Assume that the invariant (both parts) holds just before the iteration. This is the inductive hypothesis.

(b) [6 points]

Let w be the node that is chosen in an iteration. Prove that $\delta_S(s, w) = \delta(s, w)$ just before the iteration. That is, there is a best path from s to w that passes through only nodes in S.

(c) [4 points]

Now we continue to assume that the invariant (both parts) holds just before the iteration, and we may also use the result of Part (b) above, which we just proved. Prove that Part 1 of the invariant holds after the iteration.

(d) [5 points]

Now finish proving the inductive step by proving that Part 2 of the invariant holds after the iteration. As before, you are allowed to assume that the invariant holds just before the iteration. You may also use the results of Parts (b) and (c) above. Notice that Part (c) means that you can use Part 1 of the invariant for the state of the algorithm *after* this iteration.

Hint: You might want to break this down into three cases: u is in S just before the iteration, $u = w$, and u is still in Q after the iteration. The interesting case is the last one.

(e) [6 points]

Is the following true or false? After any number of iterations of the loop in Dijkstra's algorithm, if $u \in S$ and $v \in Q$, then $\delta(u) \leq \delta(v)$.
Give a proof or a counterexample.

Problem 5-3. [20 points] **Hobbits' Gold II**

The initial release of Hobbits' Gold was so successful that MIT is eager to produce a more elaborate successor game. The game is very similar to Hobbits' Gold, but now the maze is an arbitrary directed graph in which the destination t is reachable from the source s, not necessarily a DAG. It still has piles of gold and toll gates on the edges, as before, and the goal is still to get your hobbit from s to t with the greatest possible number of gold coins.

(a) [7 points]

As in Problem 5-1, Part (a), assume that the gates accept IOUs as well as coins, so your hobbit can have a negative net number of coins at any point during the traversal, or even at the end. Design an algorithm that outputs a path from s to t that your hobbit can traverse and that gives him/her the largest possible net number of gold coins at the end (this may be positive or negative). If there is no bound on how many gold coins your hobbit can collect, then your algorithm should just output "unlimited-gold", without actually producing any path. Analyze your algorithm.

(b) [7 points]

Now, as in Problem 5-1, Part (b), suppose that the gates do not accept IOUs, so your hobbit must always maintain a nonnegative number of coins in his/her rucksack. Design another algorithm that outputs a path from s to t that your hobbit can traverse to wind up at the destination if that is possible, and that gives him/her the largest possible number of gold coins at the end. If this is not possible (because your hobbit always

gets stuck at a gate), then your algorithm should say that. Also, if there is no bound on how many gold coins your hobbit can collect, then your algorithm should just output "unlimited-gold". Analyze your algorithm.

(c) [6 points]

But hobbits would not go around cycles collecting gold coins forever, because that is pointless. Rather they would do this just long enough to be certain that, when they reach t, they will have "enough" coins.

So let's revisit the case where IOUs are allowed, as in Part (a). This time, you are given not just a weighted graph, but also a cut-off value $enough$ (a positive integer) representing what hobbits regard as "enough" gold coins. Solve the same problem as in Part (a), except that, if there is no bound on how many coins your hobbit could collect, instead of just outputting "unlimited-gold", your algorithm should output some (not necessarily simple) path on which a hobbit could collect at least $enough$ coins. Analyze your algorithm.

Part B

Problem 5-4. [40 points] **Dijkstra Implementation and Applications**

In this problem, you will implement Dijkstra's shortest paths algorithm using a min-priority queue, which is in turn implemented by a min-heap. This should allow you to achieve time complexity $O((V + E) \log V)$. You will test this out on three simple applications.

You should add your code to the file `solution_template.py`. Once you write some code you can test by just running `python solution_template.py` in the command line. This should run a test suite for all methods required. We will provide you with the class `PriorityQ`, which is an implementation of a priority queue and has the following interface:

- `PriorityQ()`: The constructor of the class. Creates an empty priority queue.
- `insert(element, priority)`: Given an element $element$ and a priority $priority$ this method inserts the element with the given priority in the priority queue. If the element is already in the priority queue nothing happens.
- `decrease_priority(element, new_priority)`: Given an element $element$, this method decreases the priority of the element to $new\_priority$. If $new\_priority$ is not lower than the previous priority nothing happens. If the element is not in the priority queue, the program returns an error.
- `extract_min()`: Extracts the element with the minimum priority and returns the element. If the priority queue is empty when this method is called, the method returns an error.

Note that PriorityQ does not have methods for retrieving the priority of a certain element. You should do your own bookkeeping to be able to maintain the priority of a certain element.

(a) [20 points]

Implement Dijkstra's algorithm. The input to the algorithm will be a weighted directed graph with positive real-valued weights. The digraph will be presented in adjacency list format, and the output should be a list of all the vertices in the graph, in order of the lengths of their shortest paths. For each vertex u, you should include in your output both the final distance estimate $d[u]$ and the final parent decision $\pi[u]$.

Specifically, write your implementation for Dijkstra's algorithm in the method `dijkstra(N, adjacency_list, source)`. The inputs for the algorithm are N, the number of nodes in the graph, $adjacency\_list$ which is the adjacency list of the graph, and $source$, which is the source node. Assume that the nodes of the graph correspond to the integers $0, \ldots, N-1$. At position i, $adjacency\_list[i]$ contains a list of all the neighbors of node i. Each element in $adjacency\_list[i]$ is a tuple $(j, weight)$ representing one edge from node i to node j of weight $weight$. All weights are positive reals.

You should return a list res of size N. Each element $res[i]$ should be a tuple (d_i, pi_i), where d_i represents the length of the shortest path from $source$ to node i, and pi_i represents the predecessor of node i in the shortest path from $source$ to i. The predecessor of the source should be set to -1.

(b) [7 points]

Next door to the new and larger offices of MIT Technologies is the production plant of a popular manufacturer of Greek-style yogurt, Natural Whey. Natural Whey's manufacturing process generates a great deal of toxic waste as a byproduct. Since they can no longer dump this waste on neighboring land, they must now truck it every day to a toxic waste disposal area located a few miles away from their plant. Naturally, they want to minimize the distance traveled by their truck.

They ask for your help. It's only fair, since you are now occupying their previous dump site. Natural Whey provides for you a description of all of the roads and streets between their plant and the dump. The input is in the form of a directed graph giving landmarks and their GPS locations, plus a list of directed roads connecting the landmarks. The roads are approximately straight.

Use your algorithmic experience to produce a shortest route from the Natural Whey manufacturing plant to the toxic waste dump.

Specifically, the area of interest is represented by a cartesian coordinate system with N landmarks each located at specific coordinates (x_i, y_i). The plant and the toxic waste dump are located at two different landmarks. In addition there are M unidirectional roads between the landmarks.

Complete the code in method `shortest_path(N, coords, roads, plant_index, dump_index)` in order to return a shortest path between Natural Whey's plant and the toxic waste dump. The inputs of the method are N the number of landmarks,

$coords$ a list of size N representing the coordinates of the landmarks ($coords[i]$ is a tuple of the form (x_i, y_i)), $roads$ a list where each element is a tuple (a, b) representing a road from landmark a to b, $plant\_index$ representing the index of the landmark where Natural Whey's plant is located and $dump\_index$ representing the index of the landmark where the toxic waste dump is located.

The method should return a list $path$ representing a path of shortest length from the plant to the dump. Each element in $path$ should be an index of a landmark such that there is a road between any two adjacent landmarks and the first and last landmarks should be the plant and dump respectively.

(c) [6 points]

Natural Whey seems to have a slight problem in that the dump they initially identified does not want to handle all their toxic waste. So they need to identify more dumps. They also have another problem: it seems that a small amount of toxic waste spills out of their truck along the way, so they really want to minimize the distance traveled.

Naturally, they ask you for more help. This time, with the same input, they would like you to find the k closest toxic waste dumps on their map. They have, helpfully, indicated for you which waypoints contain toxic waste dumps; all you need to do is to find the k closest ones.

Complete the code in `closest(N, coords, roads, plant_index, dump_indices, k)`. The inputs to this method are the same as the ones in part b, except for $dump\_indices$ which is now a list representing the indices of all the dumps and k.

The method should return a list with the indices of the closest k dumps in **ascending order** of indices. Note that the number of dumps in $dump\_indices$ is at least k.

(d) [7 points]

After some more study, Natural Whey's environmental engineer has discovered that the spillage on each truck on each road segment is proportional to the amount of waste on the truck when it enters that segment, with a constant of proportionality that may be different for each segment. That is, every edge e has a "retention factor" r_e (this depends on the number of potholes in the road and perhaps other considerations), where $0 < r_e < 1$. If a truck with a load consisting of x pounds of waste enters the segment, then when it emerges from the segment, it retains exactly $r_e x$ pounds of waste.

Suppose we are considering a single, fixed dump. Use your algorithmic experience to determine a path from the Natural Whey plant to the dump for which the truck will end up with the largest amount of remaining waste when it reaches the dump.

Complete the code in method `least_spillage(N, adjacency_list, plant_index, dump_index)`. The inputs for the method are N, the number of vertices in the graph, $adjacency\_list$ representing the adjacency list of the graph—this is similar to part a, but now the weight of an edge is the retention factor of an edge, which will be a real number between 0 and 1, $plant\_index$ representing the index of the plant and $dump\_index$ representing the index of the waste dump.

The method must return the path that results in the least spillage; use the same output format as in part b.