Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — April 7, 2016
Instructors: Nancy Lynch, Aleksander Madry, and Bruce Tidor — Problem Set 5

Problem Set 5

All parts are due Tuesday, April 26 at 11:59PM. Note the unusual due date. You have some extra days, because of Quiz 2. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions.

Part A

Problem 5-1. [25 points] **Running to Lose Weight** Donald is trying to lose weight by running every day from MIT to Newton Center. The routes he chooses vary from day to day. This problem is designed to help him plan his route to maximize his weight loss.

In every case, all of Donald's routes are modeled as a directed graph $G = (V, E)$, where the vertices V correspond to a particular set of intermediate points and the edges E are directed road segments. The graph has a source node s, which represents MIT, and a target node t, which represents Newton Center.

Thanks to his Fitness Tracker, Donald knows exactly how many calories he expends when he runs on each segment. Unfortunately, some segments pass by fast food restaurants and Donald, being weak-willed, cannot resist stopping at each restaurant he passes and getting a small snack.

Specifically, every directed road segment $e \in E$ has two associated calorie measures: $loss(e)$, a positive integer representing the number of calories Donald expends by running segment e, and $gain(e)$, a positive integer representing the number of calories Donald ingests by snacking on segment e.

(a) [5 points] First suppose that G is a directed acyclic graph. Design an algorithm (using very clear English or pseudocode) that outputs a path in G that Donald can use to run from MIT to Newton Center, and that results in the largest possible net calorie loss (that is, loss minus gain—the result may be positive or negative). Your algorithm should run in $O(V + E)$ time. Include an analysis.

(b) [6 points] Donald decides it might be a good idea to impose some constraints on where he runs. Namely, he will limit his routes to just those for which, at no point during the run (that is, after no prefix of the path) does he have a net positive weight gain. (He is still weak-willed, so if he traverses a segment with restaurants, he will still snack.)

Describe another algorithm (using clear English and pseudocode) that outputs a path that Donald can use to run from MIT to Newton Center, with the new constraint, and that results in the largest possible net calorie loss among the possible paths. If this is not possible, your algorithm should say that. Again, your algorithm should run in $O(V + E)$ time. Include an analysis.

(c) [7 points] Donald realizes that his idea from part (b) wasn't that great (see Fig. 1 for a counterexample), since it sometimes rules out the maximum-calorie-loss paths. So his next idea is to expand his set of possible routes to include some less-direct routes, including routes that revisit some locations and re-traverse some segments. Now the routes are modeled by a graph G that is an arbitrary (not necessarily acyclic) directed graph in which the destination t is reachable from the source s.

Design an algorithm that outputs a path in G that Donald can use to run from MIT to Newton Center, and that results in the largest possible net calorie loss. If there is no bound on how many calories Donald might expend, then your algorithm should just output "unbounded weight loss", without producing any path. Analyze your algorithm.

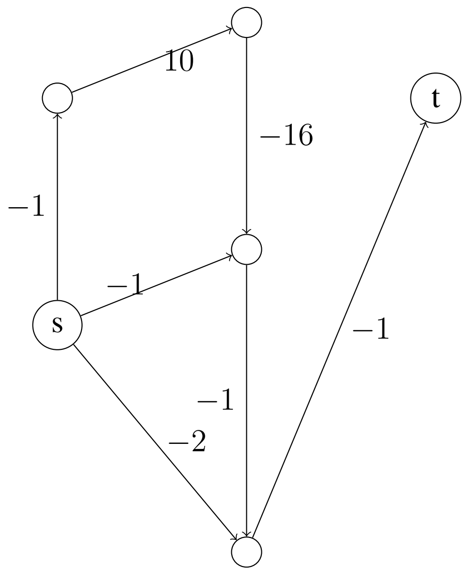

Figure 1: Example of a case when Donald's strategy from (b) fails. The optimal path of greatest weight loss contains the edge with weight 10, but using the strategy from (b), Donald would prune this path because the second edge on the path results in a net positive weight gain.

(d) [7 points] Donald really likes the idea that he could lose an unbounded amount of weight, but he does not actually want to run in cycles forever. Rather, he would like to run just long enough to be sure that, when he reaches Newton Center, he will have expended "enough" calories.

So, now you are given not just a weighted directed graph G, but also a cut-off value $enough$ (a positive integer) representing Donald's net calorie loss goal for one day. Solve the same problem as in part (d), except that, in case there is no bound on the possible weight loss, instead of just outputting "unbounded weight loss", your algorithm should output some (not necessarily simple) path on which Donald could run to achieve $enough$ weight loss. Analyze your algorithm.

Problem 5-2. [25 points] **Properties of Dijkstra's Algorithm**

Dijkstra's algorithm has many interesting properties. This problem will explore some of them.

Assumptions and notation: We assume that $G = (V, E, weight)$ is a weighted directed graph, and $s \in V$ is a designated source node. We write $\delta(u)$ for the minimum distance from s to u in the graph (measured in terms of the sum of edge weights).

At any point in an execution of the algorithm between iterations of the main loop, we define S to be the subset of V consisting of the nodes that have already been processed, and $Q = V - S$. For any node u, $d[u]$ is the currently-recorded distance from s. We also define $\delta_S(u)$ to be the minimum length of a path from s to u in which all nodes, except possibly the final node u, are in S. (As a special case, $\delta_S(s) = 0$.) If there is no such path, then $\delta_S(u) = \infty$.

The first property is:
Property 1: After any number of iterations of the main loop:
If u is any node in V, then $d[u] \geq \delta(u)$.
That is, no node ever acquires an estimate that is smaller than its real distance from s.

(a) [5 points] Explain carefully why Property 1 is true.
Hint: You might want to use induction on the number of steps in which a node is moved from Q to S.

The next two properties are:
Property 2: After any number of iterations of the main loop:
If $u \in S$ then $d[u] = \delta(u)$.
That is, all nodes in S record their correct distances.

Property 3: After any number of iterations of the main loop:
If u is *any* node, then $d[u] = \delta_S(u)$.
That is, every node in the entire graph records its best distance along a path that goes entirely through S, except possibly for u itself.

We consider them together. It's easy to see that they hold initially. The key thing to show is that they are "preserved" by any iteration of the main loop. (This is essentially induction, again.) So, assume that Properties 2 and 3 hold at some point during the execution, and then a new node w is selected from Q and added to S.

(b) [8 points] Explain carefully why Property 2 must hold after this step.
Hint: Show first that, for the w that is chosen at this step, $d[w] = \delta(w)$.

(c) [7 points] Explain carefully why Property 3 must hold after this step.

The final property is:
Property 4: After any number of iterations of the main loop:
If $u \in S$ and $v \in Q$, then $\delta(u) \leq \delta(v)$.
That is, all the nodes in S have distances from s that are less than or equal to those for the nodes in Q.

(d) [5 points] Explain carefully why Property 4 is true.

Part B

Problem 5-3. [50 points] **Dijkstra and Some Variations**

In this problem, you will implement Dijkstra's shortest paths algorithm, which finds shortest paths in a weighted directed graph from a particular source node s to all nodes. Your implementation should use a min-priority queue, which is implemented by a min-heap. This should allow you to achieve time complexity $O((V + E) \log V)$ for a complete execution of Dijkstra's algorithm. You will then adapt the implementation to solve the *Single-Pair Shortest Path (SPSP)* problem, that is, to determine the shortest pat from a particular source node s to a particular target node t. Then, you will implement two heuristic improvements to try to solve the SPSP problem more efficiently, namely, bidirectional search and A*-search.

We provide a priority queue implementation, which you should use. You should need the PriorityQueue.add(item, priority) method, which adds an item with the indicated priority. This can also be used to change an item's priority. All of the other methods should be fairly straightforward, and you should look at them all before you begin programming.

(a) [15 points] **Dijkstra's algorithm:**

Implement Dijkstra's algorithm. The input to your algorithm should be a weighted directed graph $G = (V, E, weight)$ with positive real-valued weights, together with a designated source node $s \in V$. The digraph will be presented in adjacency list format. 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 list both the final distance estimate $d[u]$ and the final parent decision $\pi[u]$

Specifically, you should write a function that takes as input the number of nodes n, a dictionary of edges mapping every vertex u to a list of tuples $(v, weight)$, representing an edge from u to v with weight $weight$, and a source node s. The nodes will be represented by the integers from 0 to $n - 1$, inclusive. The function should output a list of tuples in the form $(v, d[v], u)$, where v is a node that can be reached from s, $d[v]$ is the length of the shortest path from s to v, and u is the predecessor of v in that shortest path. This list should be sorted by $d[v]$ values, so the first element will always be $(s, 0, None)$.

The test cases will all be digraphs that represent (directed) road and street segments in a metropolitan area, where the costs represent the time required to traverse the segment by car. We do not guarantee that the costs are geometric, i.e. the costs need not satisfy the triangle inequality.

(b) [5 points] **SPSP Algorithm 1** Implement an algorithm that solves the Single-Pair Shortest Path problem by making small modifications to your Dijkstra implementation from Part (a). Now the input should be a weighted directed graph together with designated source node s and designated target node t. The output should be a tuple of two elements. The first element should be a list of the vertices in a shortest path from s to t, in order. The second element should be the final length of the path. Try to design your algorithm so that, if t is close to s, the time cost is correspondingly small. For example, if t is only one short edge away from s, then the algorithm should terminate quickly rather than loop through the entire graph.

(c) [12 points] **SPSP Algorithm 2: Bidirectional Search** Implement another algorithm that solves the Single-Pair Shortest Path problem, with the same inputs and outputs as in part (b). This time, use bidirectional search from s and t, keeping the distances balanced. That is, in searching from both s and t, always choose a node u with the smallest $d[u]$ value among the nodes in both queues. The output should be a tuple of two elements. The first element should be a list of the vertices in a shortest path from s to t, in order. The second element should be the final length of the path.

(d) [12 points] **SPSP Algorithm 3: A* Search** Implement yet another algorithm that solves the Single-Pair Shortest Path problem, this time using A* search. Now we assume that each vertex of the graph has an associated location in 2-dimensional Euclidean space. We define the weight of each edge to be its ordinary Euclidean length. (That can be considered a special case of the "traversal time" measure used in parts (a)-(c), if we consider time to be proportional to length.)

For the potential function λ used in your A* search, use simply the Euclidean distance to the target t. Note that, with this potential function, the modified weights are guaranteed to be nonnegative, by the Triangle Inequality. That is, this potential function is *feasible*.

Specifically, write a function that takes as input a length n list of locations (x, y), a dictionary of edges mapping a node u to a list of nodes v (with no weights!), a source node s, and a target node t. We still use the integers from 0 to $n - 1$ to represent the nodes, but now each node corresponds to an index in the list of locations. Note that we do not include the edge weights in the edge dictionary this time, since the edge weights are determined by the Euclidean distances between the node locations.

The output should be a tuple of two elements. The first element should be a list of the vertices in a shortest path from s to t, in order. The second element should be the final length of the path.

(e) [6 points] **Counting Nodes** In parts (b)-(d), you implemented three Single-Pair Shortest Paths algorithms; here you will compare their performance, in terms of the numbers of priority queue operations they perform.

Specifically, use PriorityQueue.num_ actions, which counts the number of times you call PriorityQueue.add or PriorityQueue.pop for a given priority queue. Augment each of your implementations from parts (b), (c), and (d) to print the needed counts. For parts (b) and (d), this is just the value of PriorityQueue.num_ actions for the single queue, but for the bidirectional search in part (c), it is the sum of the values of PriorityQueue.num_ actions for both queues.

Run the tests on cases 26, 27, and 28. These test cases run Dijkstra, bidirectional search, and A* search, respectively, on a 100x100 grid where the source is in the center and the target is in a corner. Report the number of actions that were used for each case and explain briefly why these numbers differ from each other.

Make sure to delete your print statements before submitting your code!!