Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — April 10, 2008
Professors Srini Devadas and Erik Demaine — Handout 10

Problem Set 5

This problem set is due **Thursday April 24** at **11:59PM**.
Solutions should be turned in through the course website in PDF form using LaTeX or scanned handwritten solutions.
A template for writing up solutions in LaTeX is available on the course website.
Remember, your goal is to communicate. Full credit will be given only to the 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.

Exercises are for extra practice and should not be turned in.
Exercises:

- CLRS 24.1-1 (page 591)
- CLRS 24.3-2 (page 600)
- CLRS 24.3-4 (page 600)
- CLRS 24.5-8 (page 614)
- CLRS 24.3-6 (page 600)

---

1. **(15 points)** True or False.

   Decide whether these statements are **True** or **False**. You must briefly justify all your answers to receive full credit.

   (a) **(5 points)** If some edge weights are negative, the shortest paths from s can be obtained by adding a constant C to every edge weight, large enough to make all edge weights nonnegative, and running Dijkstra's algorithm.

   Solution: False.

   $$w(s,v) = -3$$
   $$w(v,t) = 2$$
   $$w(s,t) = 1$$

In the above graph, the shortest path from s to t is through edge (s, v) and (v, t). After adding a constant $C = 3$, we get a new graph:

$$w(s, v) = 0$$
$$w(v, t) = 5$$
$$w(s, t) = 4$$

The shortest path from s to t is now through edge (s, t), By running Dijkstra, we can not necessarily find the shortest path of the original graph, because more weight will be added to paths with more edges.

(b) **(5 points)** Let P be a shortest path from some vertex s to some other vertex t. If the weight of each edge in the graph is squared, P remains a shortest path from s to t.

Solution: False.

$$w(s, v) = 1$$
$$w(v, t) = 1$$
$$w(s, t) = 1.5$$

In the above graph, the shortest path from s to t is through edge (s, t), but after squaring:

$$w(s, v) = 1$$
$$w(v, t) = 1$$
$$w(s, t) = 2.25$$

The shortest path from s to t is through edge (s, v) and (v, t).

(c) **(5 points)** A *longest simple path* from s to t is defined to be a path from s to t that does not contain cycles, and has the largest possible weight.

Given a directed graph G with nonnegative edge weights and two nodes s and t, the following algorithm can be used to either find a longest simple path from s to t, or determine that a cycle is reachable from s:

- Negate all the edge weights.
- Run Bellman-Ford on the new graph.
- If Bellman-Ford finds a shortest path from s to t, return that as the longest simple path.
- Otherwise, declare that a cycle is reachable from s.

Assume t is reachable from s.

Solution: In the case of no cycle:

G has all nonnegative weights edges, after negating, the longest simple path which has the most positive weights now has the most negative weights, thus it is the shortest path in the new graph, and Bellman-Ford Algorithm can be used to find it.

In the case of cycle: if there exists cycles, must be zeros-weight cycle, which we ignore, and positive-weights cycles. After negating, positive-weights cycles reachable from s become negative-weights cycle reachable from s, which is detectable by Bellman-Ford Algorithm.

2. **(15 points)** Critical Edges.

   You are given a graph $G = (V, E)$ a weight function $w : E \to \Re$, and a source vertex s. Assume $w(e) \geq 0$ for all $e \in E$.

   We say that an edge e is $u$pwards critical if by increasing $w(e)$ by any $\epsilon > 0$ we increase the shortest path distance from s to some vertex $v \in V$.

   We say that an edge e is downwards critical if by decreasing $w(e)$ by any $\epsilon > 0$ we decrease the shortest path distance from s to some vertex $v \in V$ (however, by definition, if $w(e) = 0$ then e is not downwards critical, because we can't decrease its weight below 0).

   (a) **(5 points)** Claim: an edge (u, v) is downwards critical if and only if there is a shortest path from s to v that ends at (u, v), and $w(u, v) > 0$. Prove the claim above.

   Solution: First, note that if (u, v) is on any shortest path, then because subpaths of shortest paths are shortest paths, (u, v) will also be on a shortest path to v.

   Second, we prove that (u, v) is downwards critical implies (u, v) is on the shortest path from s to v.

   Proof by contradiction: Assume (u, v) is downwards critical, but it is not on the shortest path from s to v. Then $\delta[s, v] < \delta[s, u] + w(u, v)$, so let $\epsilon = (\delta[s, u] + w(u, v) - \delta[s, v])/2$ is positive. If we decrease $w(u, v)$ by ϵ, we'll only be changing the cost of the paths to v going through (u, v), so the cost of the minimum path will stay the same. By the choice of ϵ, the best path going through (u, v) will still cost more than the minimum path. So the minimum path cost doesn't change when $w(u, v)$ is decreased by ϵ. Contradiction.

   Third, we prove that (u, v) is on the shortest path from s to v implies (u, v) is downwards critical.

   If (u, v) is on a shortest path to v, then decreasing its weight by any $\epsilon > 0$ decreases the cost of that path. We know that no other path through v had a lower cost than $w(u, v)$, so the path containing (u, v) is still the shortest path to v. So by decreasing the weight of (u, v), the weight of the shortest path to v is decreased, which means (u, v) is downwards critical.

(b) **(5 points)** Make a claim similar to the one above, but for upwards critical edges, and prove it.

Solution:

Claim: (u, v) is upwards critical if and only if all the shortest paths from s to v end at (u, v).

First, we again start by noting that if (u, v) is on all shortest paths to any particular node, then it must also be on all shortest paths to v.

Second, we prove that if (u, v) is upwards critical then all the shortest paths from s to v end at (u, v).

If (u, v) is upwards critical, then increasing $w(u, v)$ by $\epsilon > 0$ must increase the cost of all shortest paths from s to v, otherwise the minimum cost to get from s to v would stay the same. Increasing $w(u, v)$ only impacts the paths containing (u, v), therefore (u, v) must be contained on all shortest paths to v. Since all the edges have positive weights, (u, v) must be the last edge on any the shortest path from s to v.

Third, we prove that if all the shortest paths from s to v end at (u, v) then (u, v) is upwards critical.

If all the shortest paths from s to v include (u, v), then increasing $w(u, v)$ by any $\epsilon > 0$ increases the cost of all these paths. Therefore, the minimum cost to get from s to v is increased, so (u, v) is upwards critical.

(c) **(5 points)** Using the claims from the previous two parts, give an $O(E \log V)$ time algorithm that finds all downwards critical edges and all upwards critical edges in G.

Solution:

Run Dijkstra using binary heaps as a priority queue (binary trees or Fibonacci heaps are also acceptable data structures here). Save the results in $d[v]$ and $\pi[v]$.

Iterate through all edges, and report an edge (u, v) as downwards critical if $d[u] + w(u, v) = d[v]$. This is correct because the edges satisfying the condition must belong to the shortest paths from s to v. While doing this, compute $dc[v]$ = the number of downwards critical edges coming into v.

Iterate through all the vertices, and report $(\pi[v], v)$ as upwards critical if $dc[v] = 1$. This is correct because the check implies that $(\pi[v], v)$ is the only edge coming into v that is on a shortest path from s to v.

Running time analysis: all vertices are reachable from v, so it must be that $V = O(E)$. Then the running time of Dijkstra is $O((V + E)logV) = O(ElogV)$. Reporting downwards critical edges takes $O(E)$, because we do $O(1)$ work per iteration over all the edges. Reporting upwards critical edges takes $O(V)$, because we do $O(1)$ work per iteration over all the vertices. So the total running time is $O(ElogV + E + V) = O(ElogV)$

3. **(30 points)** Implementing Dijkstra.

   The Howe & Ser Moving Company is transporting the Caltech Cannon from Caltech's campus to MIT's and wants to do so most efficiently. Fortunately, you have at your disposal the National Highway Planning Network (NHPN), packaged for you in `ps5_dijkstra.zip`. You can learn more about the NHPN at `http://www.fhwa.dot.gov/planning/nhpn/`

   This data includes node and link text files from the NHPN. Open `nhpn.nod` and `nhpn.lnk` in a text editor to get a sense of how the data is stored (`datadict.txt` has a more precise description of the data fields and their meanings). To save you the trouble of parsing these structures from a file, we have provided you with a Python module `nhpn.py` containing code to load the text files into Node and Link objects. Read `nhpn.py` to understand the format of the Node and Link objects you will be given.

   Additionally, we have provided some tools to help you visualize the output from your algorithms. You can use the `Visualizer` class to produce a KML (Google Earth) file. To view such a file on Google Maps, place it in a web-accessible location, such as your Athena `Public` directory, and then search for its URL on Google Maps.

   For this problem, you will modify the file `dijkstra.py`. As you solve each part of the problem, check your work by running `test_dijkstra.py`. As usual, remember to comment your code, including docstrings at the top of each function.

   (a) **(3 points)** Write a short function `node_by_name(nodes, city, state)` to return a node from the given city/state. Note that some nodes have a description which isn't solely the city name, e.g. `CAMBRIDGE NW` or `NORTH CAMBRIDGE`, either of which we would like to match a query where `city=='CAMBRIDGE'`. Given a choice of more than one node, choose the first node that appears in the data.

   (b) **(3 points)** The links you are given do not include weights, so instead we will use the geographical positions of the edge's nodes.

   Write a function `distance(node1, node2)` to return the distance between two NHPN nodes. Nodes come with latitude and longitude (in millionths of degrees). For simplicity, treat these instead as (x, y) coordinates on a flat surface, where the distance between two points can be easily calculated using the Pythagorean Theorem.

   Hint: You may find the `math.hypot` function useful.

   (c) **(24 points)** Implement Dijkstra's algorithm to find the shortest path between two vertices in a graph with non-negative edge weights.

   Your function `shortest_path(nodes, edges, weight, s, t)` will be given a list of Node objects, a list of Edge objects (undirected), a function `weight(node1, node2)` which returns the weight of any edge between `node1` and

`node2`, a source Node s and a destination Node t. Your function should return a list of `Node` objects representing a path from s to t.

Dijkstra's algorithm uses a priority queue, but this priority queue has one subtle requirement not met by the `heap.py` implementation seen earlier in class. Dijkstra's algorithm calls `decrease_key`, but `decrease_key` requires the index of an item in the heap, and Dijkstra's algorithm would have no way of knowing the current index corresponding to a particular Node. To solve this problem, the course staff has written an augmented heap object, `heap_id`, with the following extra features:

- `insert(key)` returns a unique ID.
- A new method, `decrease_key_using_id(ID, key)` takes an `ID` instead of an index.
- A new method, `extract_min_with_id()` extracts the minimum element and returns a pair `(key, ID)`

You may `import heap_id`, without submitting the separate file.

Hint: The format in which you are given the data (a list of nodes, and a list of edges), is not what you want to use for Dijkstra's algorithm. Start by preprocessing the data into a more useful graph representation. Don't forget that the edges you are given are undirected.

(d) **(Optional)** Included in `nhpn.py` is a method to convert a list of nodes to a `.kml` file. `.kml` files can be viewed using Google Maps, by putting the file in a web-accessible location (like your Athena Public directory), going to `http://maps.google.com` and putting the URL in the search box.

Run `visualize_path.py`. This will create two files, `path_flat.kml` and `path_curved.kml`. Both should be paths from Pasadena CA to Cambridge MA. `path_flat.kml` was created using the distance function you wrote in part (b), and `path_curved.kml` was created using a distance function that does not assume the Earth is flat. Can you explain the differences? Also, try asking Google Maps for driving directions from Caltech to MIT to get a sense of how similar their answer is.