Introduction to Algorithms: 6.006
Massachusetts Institute of Technology November 2, 2010
Professors Konstantinos Daskalakis and Patrick Jaillet Handout 5

Problem Set 5

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

Part A questions are due **Tuesday, November 16th** at **11:59PM**.

Part B questions are due **Friday, November 19th** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be a valid Python file, together with one PDF file containing your solutions to the two theoretical questions in Part B.

Templates for writing up solutions in LaTeX are 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.

Part A: Due Tuesday, November 16th

1. (**15 points**) No Odd Cycles

   Given an undirected graph $G = (V, E)$, design an algorithm that decides whether or not G contains a cycle of odd length. Prove its correctness. Your algorithm should run in time $O(|V| + |E|)$.

2. (**15 points**) Maximum Bandwidth Path

   Suppose we have a telephone network N with n nodes, m links (undirected edges) between them, and a distinguished source node s. Each link has an associated *bandwidth* which is a non-negative integer that indicates the maximum number of requests that link can handle. In a path of links, the link in the path with the smallest bandwidth constrains the number of requests that can be passed along the path. Hence, we define the bandwidth of a path P consisting of links $\ell_1, \ldots, \ell_k$ to be $\min(\text{bandwidth}(\ell_1), \ldots, \text{bandwidth}(\ell_k))$. The *maximum bandwidth path* to a node v is the path from s to v with the maximum bandwidth.

   Design an algorithm that computes the maximum bandwidth path to every node v in the network, and prove its correctness. Your algorithm should have the same asymptotic running time as Dijkstra's.

3. (**20 points**) Shortest Roundtrip Path

   Suppose G is a strongly connected directed graph, meaning that for every two vertices u and v in G, there is a directed path from u to v and a directed path from v to u. Each edge has a weight, not necessarily non-negative. Let s be a distinguished node in G. For every vertex v in the graph, we want to calculate a path from s to v and a path from v to s such that the sum of the weights of edges on the two paths is minimized. Give an efficient algorithm to perform this task, and show correctness. If your algorithm encounters a negative-weight cycle, it should output `false`. Otherwise, it should output the pair of paths from s to v and from v to s that minimizes the roundtrip cost.

Part B: Due Friday, November 19th

1. **(50 points)** Speeding up 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.

Your goal in this problem is to implement and test several techniques for speeding-up Dijkstra's algorithm in order to compute shortest paths between various pairs of locations.

Implementation of Dijkstra's algorithm is already provided. Function

```
dijkstra(nodes, edges, weight, source)
```

is given a graph with non-negative edges (represented as a list of Node objects and a list of undirected Edge objects), a function `weight(node1, node2)` that returns the weight of any edge between `node1` and `node2`, and a source node `source`. This function updates the `node.visited` field for all nodes, which indicates whether a shortest path to `node` has been found, as well as `node.distance` and `node.parent` for visited nodes, which are the length of the shortest path from the source node to `node` and the previous node on that path respectively. The function returns the number of nodes visited during the execution of the algorithm.

The links you are given do not include weights, so instead we use the geographical distance between their endpoints. Function `distance(node1, node2)` returns the distance between two NHPN nodes. Nodes come with latitude and longitude (in millionths of degrees). For simplicity, we treat these as (x, y) coordinates on a flat surface, where the distance between two points can be calculated using the Pythagorean Theorem.

Dijkstra's algorithm uses a priority queue, but this priority queue has one subtle requirement. 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 have written an augmented heap object, `heap_id`, with the following extra features:

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

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 the appropriate test functions. We have provided several test functions that test each part separately or perform comparison tests of several methods. You should follow the instructions for each part of the problem, perform appropriate tests and draw conclusions. Please submit the modified `dijkstra.py` file with the code and `dijkstra.pdf` file with proofs and short answers. Keep them short.

(a) **(3 points)** Examine the code provided in `nhpn.py`, `heap.py` and `dijkstra.py` to learn the structure of the Node and Link classes and the implementation of Dijkstra's algorithm. Run `test_a()`. Is there a significant difference in the execution time for different pairs of nodes? Explain your observation.

(b) **(7 points)** One way to speed up Dijkstra's algorithm is to terminate the algorithm early once a shortest path to the destination has been found.

Implement function

```
dijkstra_early_stop(nodes, edges, weight, source, dest)
```

that performs this optimization. As with the function `dijsktra()`, this function should update the `node.visited`, `node.distance` and `node.parent` fields, and return the number of nodes visited during its execution. Run `test_b()`. What characterizes pairs of nodes for which there is a significant speed-up using this optimized version of Dijkstra's algorithm?

HINT: Reuse the implementation of Dijkstra's algorithm provided, making the required changes to allow for early termination.

(c) **(20 points)** We will apply the potentials method with a landmark node to obtain a faster shortest path algorithm. For a given landmark node l, we denote the potential of a node u with respect to a destination node t by $\lambda_t^l(u)$. The potential is defined as $\lambda_t^l(u) = \delta(u, l) - \delta(t, l)$ if there exists a path from u to t through l, and $\lambda_t^l(u) = C$

where C is some fixed constant if no such path exists. (Here, $\delta(u, v)$ denotes the length of a shortest path from u to v).

i. Prove that this potential function is feasible, i.e. the modified weight of every edge is non-negative.

ii. Implement function

`compute_landmark_distances(nodes, edges, weight, landmark)`

that computes shortest paths from all nodes to the given landmark node `landmark`. For each node `node`, `node.land_distance` should be set to the value of the shortest path distance from `node` to `landmark` or to some constant C if no such path exists (e.g., $C = 10^9$). Why is it more useful to precompute distances to the landmark node than precomputing the potentials themselves?

iii. Implement function

`dijkstra_with_potentials(nodes, edges, weight, source, dest)`

that performs Dijkstra's algorithm using edge weights modified according to the potentials method (i.e. $w'(u, v) = w(u, v) - \lambda_t^l(u) + \lambda_t^l(v)$ for a landmark l and destination t) and terminates as soon as a shortest path to the destination node `dest` has been found. This function assumes that `node.land_distance` is already set to the proper value (no need to call `compute_landmark_distances()` from it). As before, this function should update `node.visited`, `node.distance` and `node.parent` fields, and return the number of nodes visited during its execution. Run `test_d()`. In which scenarios is the speed-up most significant (compare to both Dijkstra's algorithm and Dijkstra's algorithm with early termination)?

Hint: Reuse the implementation of Dijkstra's algorithm provided, making the necessary changes.

(d) **(20 points)** We will now describe a potentials method where multiple landmarks are used. For a given set of landmarks L, the potential of a node u with respect to a destination node t is $\lambda_t^L(u) = max_{l \in L} \lambda_t^l(u)$, where $\lambda_t^l(u)$ is defined as before.

i. Prove that this potential function is also valid, i.e. the modified weight of every edge is non-negative.

ii. How does the potential function $\lambda_t^L(u)$ compare to a potential function $\lambda_t^l(u)$ for any single landmark vertex $l \in L$ in terms of the number of visited nodes when used in Dijkstra's algorithm with early termination? Explain which is better and why.

iii. Implement function

```
compute_multi_landmark_distances(nodes, edges, weight,
                                 landmarks)
```

that computes shortest paths from all nodes to all given landmark nodes in `landmarks`. For each node `node`, `node.land_distances` should be set to a list of values, such that `node.land_distances[i]` is the shortest path length from `node` to `landmarks[i]` or some constant C if no such path exists (e.g., $C = 10^9$).

iv. Implement function

```
dijkstra_with_max_potentials(nodes, edges, weight,
                              source, dest)
```

that performs Dijkstra's algorithm using edge weights modified according to the potentials method with multiple landmarks (i.e. $w'(u,v) = w(u,v) - \lambda_t^L(u) + \lambda_t^L(v)$ for a set of landmarks L and destination t), and terminates as soon as a shortest path to the destination node `dest` is found. This function assumes that `node.land_distances` is already set to the proper list of values. As before, this function should update `node.visited`, `node.distance` and `node.parent` fields, and return the number of nodes visited during its execution. Run `test_f()`. Does the performance of your algorithm match your assertions from part ii.?

(e) **(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 that 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.