*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology
Professors Sivan Toledo and Alan Edelman

Tuesday, April 7th
Handout 10

# Problem Set 5

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

**Part A questions** are due ***WEDNESDAY*, April 22nd** at **11:59PM**.

**Part B questions** are due **Thursday, April 23rd** 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)

---

## Part A: Due *WEDNESDAY*, April 22nd

1. **(50 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) **(5 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) **(5 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) **(40 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 graph (represented as a list of Node objects and a list of undirected Edge objects), 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.

## Part B: Due Thursday, April 23rd

1. **(10 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.

   (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 $t$.

2. **(20 points)** Parameterized Graph

   This problem focuses on directed graphs $G = (V, E)$ in which the edges have weights that are functions of a parameter $x$. The weight of the edge $(u, v)$ is a function $A_{u,v}x + B_{u,v}$ where $A_{u,v}$ and $B_{u,v}$ are real numbers that are given for each edge as part of the input.

   At any particular value of $x$, each edge assumes a real weight. The goal of the problem is to find a value of $x$ for which there are no negative cycles in the graph or to determine that no such $x$ exists.

   (a) **(5 points)** Explain how to enhance the normal Bellman-Ford algorithm (for graphs with real-valued edges) so that it returns a negative-weight cycle if there is one.

(b) **(3 points)** Show that all the steps of the Bellman-Ford algorithm except for the conditional that checks whether $d[v] > d[u] + w(u, v)$ can be implemented for parametric-weighted edges if each $d[v]$ represents a function $A_v x + B_v$ rather than a real-valued variable. How would you represent $d$ in Python?

(c) **(3 points)** Show that in a run of Bellman-Ford on a graph with parametic-weighted edges (and with the representation of $d$ from the previous part of the problem), the condition $d[v] > d[u] + w(u, v)$ is equivalent to a condition that $x$ is greater or smaller than some real number or than $\pm\infty$.

(d) **(9 points)** Describe an algorithm to find a value of $x$ for which there are no negative cycles or to decide that there is no such $x$. Analyze the running time of the algorithm. Hints: run Bellman-Ford on the parametric-weighted edges so that all conditionals are resolved in a way that is consistent with the decision of the algorithm on a particular $x$ where there are no negative cycles. Keep track of a maximal interval $(x_{\min}, x_{\max})$ in which all the decisions of the algorithm so far are valid. Resolve conditionals in the parametric-weighted algorithm by running a normal Bellman-Ford at a particular $x$ in this interval; the outcome either finds that this $x$ leads to no negative cycles, or shrinks the interval, or indicates that there is no feasible $x$.

3. **(20 points)** Even-Length Paths

   An even-length path is a path traversing an even number of edges. Describe a modified version of Dijkstra's algorithm that finds the shortest even-length path in a graph $G = (V, E)$ from a given start vertex $s$ to all vertices $t \in V$. The graph has non-negative edge weights. Your solution should have the same asymptotic running time as Dijkstra's algorithm. (HINT: try solving the problem by constructing a graph $G'$ that is somehow related to $G$, running Dijkstra's algorithm on $G'$, and projecting the results back onto $G$.)