Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Erik Demaine and Srini Devadas

November 1, 2011
Problem Set 6

Problem Set 6

Both theory and programming questions are due on **Tuesday, November 15** 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.

We will provide the solutions to the problem set 10 hours after the problem set is due. You will have to read the solutions, and write a brief **grading explanation** to help your grader understand your write-up. You will need to submit the grading guide by **Tuesday, November 22, 11:59PM**. Your grade will be based on both your solutions and the grading explanation.

Problem 6-1. [30 points] **I Can Haz Moar Frendz?**

Alyssa P. Hacker is interning at RenBook (人书 / 人書 in Chinese), a burgeoning social network website. She needs to implement a new friend suggestion feature. For two friends u and v (friendship is undirected), the ***EdgeRank*** $ER(u, v)$ can be computed in constant time based on the interest u shows in v (for example, how frequently u views v's profile or comments on v's wall). Assume that EdgeRank is directional and asymmetric, and that its value falls in the range $(0, 1)$. A user u is ***connected*** to a user v if they are connected through some mutual friends, i.e., $u = u_0$ has a friend u_1, who has a friend u_2, ..., who has a friend $u_k = v$. The integer k is the ***vagueness*** of the connection. Define the ***strength*** of such a connection to be

$$S(p) = \prod_{i=1}^{k} ER(u_{i-1}, u_i).$$

For a given user s, Alyssa wants to have a way to rank potential friend suggestions according to the strength of the connections s has with them. In addition, the vagueness of those connections should not be more than k, a value Alyssa will decide later.

Help Alyssa by designing an algorithm that computes the **strength** of the strongest connection between a given user s and **every other user** v to whom s is connected with vagueness at most k, in $O(kE + V)$ time (i.e., for every pair of s and v for $v \in V \backslash \{s\}$, compute the strength of the strongest connection between them with vagueness at most k). Assume that the network has $|V|$ users and $|E|$ friend pairs. Analyze the running time of your algorithm. For partial credit, give a slower but correct algorithm. You can describe your algorithm in words, pseudocode or both.

Problem 6-2. [30 points] **RenBook Competitor**

Having experienced RenBook and its alternatives, you've come to the conclusion that the social networking websites currently out there lack focus. You've decided to create a social network that caters specifically to the algorithms-in-Python crowd. Excited about this idea, you've pitched to various investors, a few of whom have traded you a check in return for convertible debt. You're ready to go! Now you have to implement your idea.

The first step is to rent a machine on the web, a task which you've easily accomplished using your newly-acquired funding. The next step, however, is harder to complete: you must install a web server library and all of its dependencies. Each library that you wish to install can depend on a number of other libraries, which you will have to install first. Each of those libraries can in turn have its own dependencies.

You can try to install the libraries one-by-one and resolve the dependencies by hand, but you'd like to write a script that will do the work for you. Fortunately, having studied graphs in 6.006, you're confident that you will be able to accomplish this task. You will need to determine which libraries need to be installed and then generate the order in which the libraries will be installed so that there will be no dependency problems.

Examining the software library repository, you see that there are V total libraries, which together have a total of E dependencies. The repositories enforces the rule that dependencies cannot be cyclic. Libraries rarely all depend on one another, so you can safely assume that $E \ll V^2$.

(a) An ***installation order*** is an ordering of all the libraries such that each library's dependencies appear prior to it in the sequence. If we install each library in this sequence in order, we are guaranteed to avoid dependency problems. Describe in detail how to generate an installation order for the entire repository in $O(V + E)$ time.

We wish to install a web server library along with its dependencies. Suppose that some libraries are already installed on your system, and that only P libraries remain to be installed (you can determine whether a library has already been installed by performing a dictionary lookup in $O(1)$ time). Assume that the maximum number of dependencies for any given library is D.

(b) Give pseudocode for an algorithm that generates an installation order for the non-installed libraries that are needed for installing the web server library in $O(P + PD)$ time. Describe your algorithm. You may use any routine given in CLRS as a subroutine in your pseudocode, and you can use a textual description, a clarifying example, or a correctness proof for the description.

Problem 6-3. [30 points] **Rubik's Cube**

In this problem, you will develop algorithms for solving the $2 \times 2 \times 2$ Rubik's Cube, known as the *Pocket Cube*. Call a configuration of the cube "k levels from the solved position" if it can reach the solved configuration in exactly k twists, but cannot reach the solved configuration in any fewer twists.

The `rubik` directory in the problem set package contains the Rubik's Cube library and a graphical user interface to visualize your algorithm.

We will solve the Rubik's Cube puzzle by finding the shortest path between two configurations (the start and goal) using BFS.

A BFS that goes as deep as 14 levels (the diameter of the pocket cube) will take a few minutes (not to mention the memory needed). A few minutes is too slow for us: we want to solve the cube very quickly!

Instead, we will take advantage of a fact that we saw in lecture: the number of nodes at level 7 (half the diameter) is much smaller than half the total number of nodes.

With this in mind, we can instead do a two-way BFS, starting from each end at the same time, and meeting in the middle. At each step, expand one level from the start position, and one level from the end position, and then check to see whether any of the new nodes have been discovered in both searches. If there is such a node, we can read off parent pointers (in the correct order) to return the shortest path.

Write a function `shortest_path` in `solver.py` that takes two positions, and returns a list of moves that is a shortest path between the two positions.

Test your code using `test_solver.py`. Check that your code runs in less than 5 seconds.

Problem 6-4. [30 points] **From Berklee to Berkeley**

Jack Florey and his fellow hackers are planning to put a TARDIS[1] on Berkeley's most symbolic tower. However, the company responsible for transporting the construction material mistook the destination as Berklee College of Music. In order to save the extra cost of transportation back to Berkeley, Jack wants to help them to find the fastest route from Berklee to Berkeley. He downloaded the data from the National Highway Planning Network (NHPN)[2]. However, as he did not take 6.006 and does not know how to implement the Dijkstra's algoritm, he turns to you for help.

The partial code is in the `dijkstra` directory in the zip file for this problem set. It contains a `data` directory which 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` and `format.txt` have more precise descriptions of the data fields and their meanings). The Python module `nhpn.py` provided contains code to load the text files into `Node` and `Link` objects. Read `nhpn.py` to understand the format of the `Node` and `Link` objects.

In `dijkstra.py`, the `PathFinder` object contains a source node, a destination node, and a `Network` object which represents the highway network with a list of `Node` objects. For each `node`, `node.adj` contains a list of all nodes adjacent to `node`.

Your task is implementing the method

```
PathFinder.dijkstra(weight, nodes, source, destination)
```

using Dijkstra's algorithm. It is given a function `weight(node1, node2)` that returns the weight of the link between `node1` and `node2`, a list of all the `nodes`, a `source` node and a `destination` node in the network. The method should return a tuple of the shortest path from the `source` to the `destination` as a list of nodes, and the number of nodes visited during the execution of the algorithm. A `node` is visited if the shortest path of it from the `source` is computed. You should stop the search as soon as the shortest path to the `destination` is found.

Function `distance(node1, node2)` is a `weight` function used by the main program to call the `dijkstra` method. It returns the distance between two NHPN nodes. Nodes come with

[1] Doctor Who's Time And Relative Dimension In Space

[2] http://www.fhwa.dot.gov/planning/nhpn/

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.

The `NodeDistancePair` object wraps a given node and its distance which can be used as a key in the priority queue.

The Python module `priority_queue.py` contains a min-heap based implementation of priority queue. It is augmented with a map of keys to their indices in the heap, so that the `decrease_key(key)` method takes $O(1)$ time to lookup the key in the priority queue.

After implementing `PathFinder.dijkstra()`, you can run

```
python dijkstra.py < tests/0boston_berkeley.in
```

to see the shortest distance from Boston, MA to Berkeley, CA.

To visualize the result, you can run the following command.

```
TRACE=kml python dijkstra.py < tests/0boston_berkeley.in
```

On Windows, use the following command instead:

```
dijkstra_kml.bat < tests/0boston_berkeley.in
```

This will create two files, `path_flat.kml` and `path_curved.kml`. Both should be paths from Boston, MA to Berkeley, CA. `path_flat.kml` is created using the distance function described earlier, and `path_curved.kml` is created using a distance function that does not assume that the Earth is flat. `.kml` files can be viewed using Google Maps, by putting the file in a web-accessible location (such as your *Athena Public directory), going to http://maps.google.com and putting the URL (`http://...`) in the search box. Try asking Google Maps for driving directions from Berklee to Berkeley to get a sense of how similar their answer is. Two sample `.kml` files `path_flat_sol.kml` and `path_curved_sol.kml` are provided in the same directory and you can check your results against the samples.

You can use the following command to run all the tests on your Dijkstra's implementation:

```
python dijkstra_test.py
```

When your code passes all the tests and runs reasonably fast (the tests should complete in less than 40s when using CPython), upload your modified `dijkstra.py` to the course submission site. Our automated grading code will use our versions of `dijkstra_test.py`, `nhpn.py`, and `priority_queue.py`, so please do not modify these files.

*Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

6.006 Introduction to Algorithms
Fall 2011