Problem Set 3

CS 331H

Due Tuesday, March 21

1. You have n power plants which, being fancy modern renewable energy designs, do not work all the time (they depend on sun, wind, tides, etc). Each power plant runs from a start time s_i to a finish time f_i, and costs c_i to run (you either run it the whole time or no time). You would like to find a set of power plants to run such that you have power over the entire interval $[0, T]$. What is the minimum cost achievable?

   You may suppose that the s_i and f_i are integers between 0 and $T = O(n)$.

   (a) Observe that the answer corresponds to the shortest path on an appropriate graph, which can be solved in $O(n \log n)$ time using Dijkstra's algorithm. [You may have already done this on a previous problem set.]

   (b) Now suppose that you can sell off extra power if you have more than one power plant running at a time. At each time step $t \in [T]$, each power plant beyond the first that you run gives you value $v_t \geq 0$, decreasing your costs. Show how to extend the part (a) graph to handle this case.

   Now suppose that the cost c_i to run a power plant is larger than the value of the electricity it produces, $\sum_{t=s_i}^{f_i} v_t$. Show that Dijkstra's algorithm will find the solution in $O(n \log n)$ time, by constructing an appropriate potential function so the edge costs become nonnegative.

   (c) [Optional] Now solve the previous part without assuming that the cost to run a power plant is larger than the value of the electricity it produces.

2. You are given a three dimensional object. On the horizontal plane it is an $n \times n$ square, and on the vertical axis each square (x, y) is a square pillar rising to height $h_{x,y} \geq 1$. Adjacent pillars, even ones sharing corners, are fused together.

You submerse this object into a bucket of water, then carefully lift it out. Water will then drain off the sides, but it cannot drain through pillars. How many units of water will be captured in the object? Give an $O(n^2 \log n)$ algorithm.

As an example, in the following grid 2 units will be captured, all in the center tile:

0	5	9
7	3	6
7	5	2

Hint: You may use from class that a variant of Dijkstra's algorithm can solve the *minimax path* problem. The shortest path problem is to find paths minimizing total length $\sum_{e \in P} c(e)$; the minimax path problem is to find paths minimizing the *maximum* length $\max_{e \in P} c(e)$.

3. In this problem, we're going to examine the results of using A^* with different potential functions. Download the code from the course website. It uses Python and requires numpy (version 1.8 or higher) and matplotlib. The program creates a bunch of nodes placed in a square, and links each node to nearby ones, with ℓ_2 distance as the distance. It then picks an s and t and computes the shortest s-t path using A^* and various potential functions. If you run the program, you should see a result like Figure 1, and it should print out the computed distance.

   Play around with different potential functions and different size graphs, getting familiar with the program. Then:

   (a) If you run the program several times, one of the printed distances will often not match the other two. Why is that?

   (b) How do the number of nodes/edges visited compare for the various potentials?

   (c) Modify the code so that s and t lie in opposite corners of the square, and see how the results behave.

   (d) Now, implement the ALT algorithm. The ALT algorithm stands for "A^*, landmarks, and triangle inequality". It's a way of preprocessing the graph to construct a potential function such that fresh queries—new s and t pairs—can be solved very efficiently.

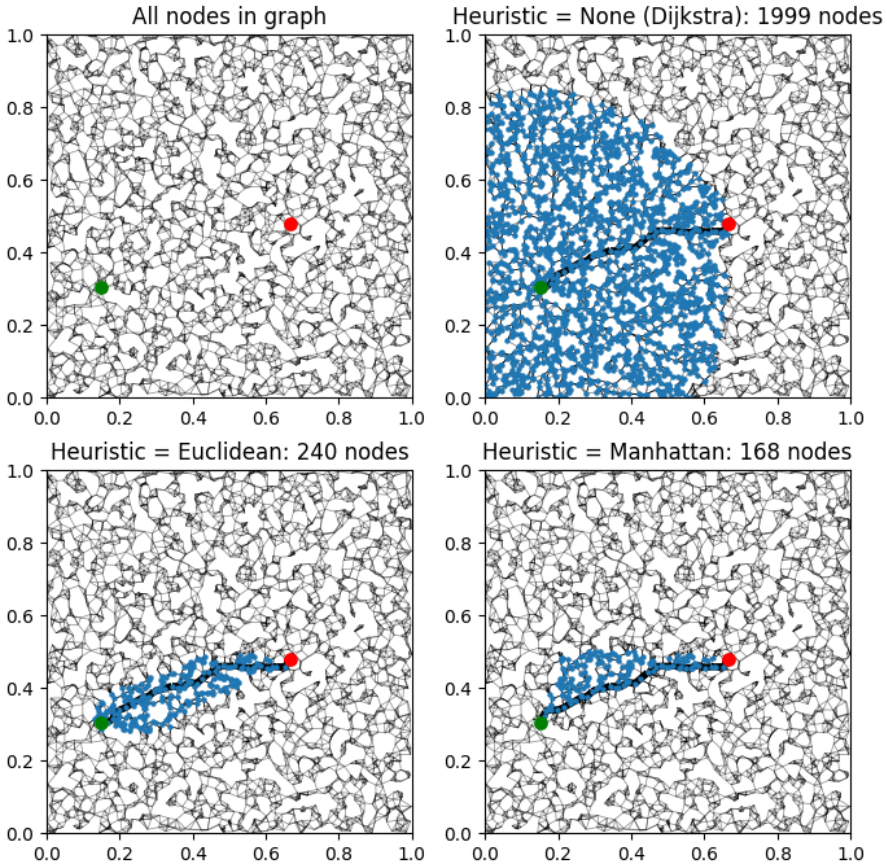

Figure 1: The top left gives the problem instance: shortest paths from the green dot to the red dot. The other plots give the result of running A^* with different potentials. The thick path is the shortest path found; the nodes are the nodes visited; the blue nodes are visited, and the edges in the shortest path are darker.

The idea is to choose a small number nodes $l_1, \ldots, l_L$ to be "landmarks" (say, 10-20). We preprocess the graph using full Dijkstra to compute $\text{dist}(u, l_i)$ for all vertices u and landmarks l_i.

On a new query s, t, we observe by the triangle inequality that

$$\text{dist}(u, t) \geq \text{dist}(u, l_i) - \text{dist}(t, l_i)$$

and

$$\text{dist}(u, t) \geq \text{dist}(l_i, t) - \text{dist}(l_i, u)$$

for all landmarks l_i. Hence, if we set

$$\phi(u) := \max_i |\text{dist}(u, l_i) - \text{dist}(t, l_i)|$$

we get an admissible heuristic (i.e., $\phi(u) \leq d(u, t)$ for all u).

i. Show that ϕ is in fact consistent (i.e. $\phi(u) - \phi(v) \leq \text{dist}(u, v)$ for all u, v).

ii. Implement the ALT heuristic. With 20 landmarks, it should usually visit less than 1/3 as many nodes as the ℓ_2 heuristic. (You can either choose random landmarks or try to pick them more carefully.)
Print out the resulting graph, comparing the nodes picked by ℓ_2 and ALT. Also include the number of nodes visited by each in several sample runs.