CSCI 104L Lecture 9: Graph Search

Update Heap

```
void MinHeap::UpdatePriority(int node, int priority) {
  int location = map[node];
  if (a[location] < priority) {
    a[location] = priority;
    trickleDown(location);
  } else {
    a[location] = priority;
    bubbleUp(location);
  }
}
class MinHeap {
  public:
    void UpdatePriority(int node, int priority);
  private:
    int *a; // stores the priorities
    int *map; // stores the locations of each node
};
```

Dijkstra's Algorithm

```
int d[n]; //distances from the start node u
int p[n]; //predecessors
int c[n][n]; //edge costs
void Dijkstra (int u) {
  PriorityQueue<int> pq();  //How should we implement this?
  d[u] = 0;
  pq.add(u, d[u]);
  while(!pq.isEmpty()) {
    int v = pq.peek();
    pq.remove();
    for all nodes outgoing edges (v,w) from v {
      if (w hasn't been visited || d[v] + c[v][w] < d[w]) {
        d[w] = d[v] + c[v][w];
        p[w] = v;
        if (this is w's first visit) {
          pq.add(w, d[w]);
        }
        else pq.update(w, d[w]);
      }
    }
  }
}
```

Question 1. How many add/remove/update calls are needed?

Question 2. What is the runtime of Dijkstra's using an unsorted array as an implementation?

Question 3. What is the runtime of Dijkstra's using a sorted array?

Question 4. What is the runtime of Dijkstra's using a MinHeap?

Note that in Dijkstra's Algorithm, we only ever bubbleUp.

Question 5. Is there any type of graph where you'd want to use an unsorted array instead of a heap?

A* Search

A* search is a heuristic search. That means that it uses "rules of thumb" to often improve the runtime. It never runs worse than Dijkstra, and always finds the correct solution, but it requires more information (which you may not have readily available).

- A* modifies Dijkstra's so that we always next explore the node with smallest d[v]+h[v], where h[v] is our estimate of how far v is from the destination.
- This only works if our heuristic never over-estimates.
- Our heuristic, when it is wrong, must underestimate. We want it to be as accurate as possible however.

The two extremes:

- h(v) = 0.
- h(v) = actual distance.

A simple heuristic for A* is Manhattan Distance. This works well for the 15-tile puzzle, and Pac-man. There are better heuristics, but Manhattan Distance is good and simple enough for you to code up yourself.

Use A* to solve the following 5-tile puzzle, using Manhattan distance as your heuristic.

Starting state:

5	2	
1	4	3

Goal State:

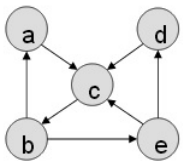
1	2	3
4	5	

Search space size = 6! = 720

PageRank

The key insight that resulted in the dominance of the Google Search Engine was to think of the world wide web as a graph.

- If page A links to page B, this is an endorsement from A of B.
- Not all links are created equal. If A links to B, to determine how good B is requires us to first know how good A is. Sounds circular, but there is a nice solution to the problem.
- Number of links is important. If A and B are equally good pages, but A has one outgoing link to C, and B has 10,000 outgoing links, one of them to D, we would say that the link from A to C confers more support (since it was chosen more carefully).



rA = 0.5*rB
rB = rC
rC = rA+rD+0.5*rE
rD = 0.5*rE
rE = 0.5*rB
rA+rB+rC+rD+rE=1

To solve this system of equations requires lots of linear algebra, and the actual PageRank has a few extra optimizations. There is a simpler way to explain the solution without linear algebra:

- You have a web-surfer who starts at a page chosen uniformly at random, and chooses an outgoing link uniformly at random.
- If the surfer gets stuck (no outgoing links) they move to a page chosen uniformly at random. The surfer can get stuck in a less obvious trap (two pages linking only to each other), so there is usually a small probability (around 15%) that at each step the surfer moves to a page chosen uniformly at random.
- We can iteratively calculate the probability the surfer is at any given page at any given step. At the first step, each page is equally likely. We use those values to calculate the probability the surfer is at any given page on the second step, and repeat.
- Typically this process will stop at some small number of iterations (such as 20), but in theory you would want to find the limit as the number of iterations approaches infinity.

This system is called PageRank, and started as a research project by two grad students at Stanford. It was so much better than the competition, that it quickly became a serious product that made them filthy stinkin' rich.