CSCI 104L Lecture 20 : Heaps and A*

Here are some of the functions within the PriorityQueue class:

```cpp
T PriorityQueue::peek() const {
    return a[0];
}
void PriorityQueue::add(const T& data) {
    a[size] = data;
    bubbleUp(size);
    size++;
}
void PriorityQueue::bubbleUp(int pos) {
    if (pos > 0 && a[pos] > a[(pos-1)/2]) {
        a.swap(pos, (pos-1)/2);
        bubbleUp((pos-1)/2);
    }
}
void PriorityQueue::remove() {
    a.swap(0,size-1);
    size--;    
    trickleDown(0);
}
void PriorityQueue::trickleDown(int pos) {
    int child = 2*pos+1;
    if (child < size) {
        if (child+1 < size && a[child] < a[child+1]) child++;
        if (a[child] > a[pos]) {
            a.swap(child, pos);
            trickleDown(child);
        }
    }
}
```

**Question 1.** What is the runtime for each of the following PriorityQueue operations:

- Peek
- Add
- Remove

```cpp
void HeapSort(int *a, int size) {
    Heap h;
    for (int i = 0; i < size; i++) {
        h.add(a[i]);
    }
    for (int i = 0; i < size; i++) {
        a[i] = h.peek();
        h.remove();
    }
}
```
Question 2. What’s the runtime of Heapsort?

Question 3. Is it stable?

XKCD #835: Not only is that terrible in general, but you just KNOW Billy’s going to open the root present first, and then everyone will have to wait while the heap is rebuilt.
Dijkstra’s Algorithm

```java
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);
                } else pq.update(w, p[w]);
            }
        }
    }
}
```

**Question 4.** How many add/remove/update calls are needed?

**Question 5.** What is the runtime of Dijkstra’s using an unsorted array as an implementation for that?

**Question 6.** What is the runtime of Dijkstra’s using a sorted array for that?

**Question 7.** What is the runtime of Dijkstra’s using a MinHeap/Priority Queue for that structure?

To implement the UpdatePriority function for a heap, you must maintain a map from node names to heap indices. Since nodes are typically named 0 through n – 1, we’ll just make an int array, where the index is the node name, and the value is the location where that node is currently stored in the heap.

Every time you change the location of a node in the heap, through add, remove, trickleDown, or bubbleUp, you must also update the value of the node in the map.

```java
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
};

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

**Question 8.** 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 16-tile puzzle, and Pac-man. There are better heuristics, but Manhattan Distance is good and simple enough for you to code up yourself.