Recitation 13

Dijkstra's Algorithm

Dijkstra is possibly the most commonly used weighted shortest paths algorithm; it is asymptotically faster than Bellman-Ford, but only applies to graphs containing non-negative edge weights, which appear often in many applications. The algorithm is fairly intuitive, though its implementation can be more complicated than that of other shortest path algorithms. Think of a weighted graph as a network of pipes, each with non-negative length (weight). Then turn on a water faucet at a source vertex s. Assuming the water flowing from the faucet traverses each pipe at the same rate, the water will reach each pipe intersection vertex in the order of their shortest distance from the source. Dijkstra's algorithm discretizes this continuous process by repeatedly relaxing edges from a vertex whose minimum weight path estimate is smallest among vertices whose out-going edges have not yet been relaxed. In order to efficiently find the smallest minimum weight path estimate, Dijkstra's algorithm is often presented in terms of a minimum priority queue data structure. Dijkstra's running time then depends on how efficiently the priority queue can perform its supported operations. Below is Python code for Dijkstra's algorithm in terms of priority queue operations.

```
1  def dijkstra(Adj, w, s):
2      d = [float('inf') for _ in Adj]          # shortest path estimates d(s, v)
3      parent = [None for _ in Adj]             # initialize parent pointers
4      d[s], parent[s] = 0, s                   # initialize source
5      Q = PriorityQueue()                      # initialize empty priority queue
6      for v in range(len(Adj)):                # loop through vertices
7          Q.insert(v, d[v])                    # insert vertex-estimate pair
8      for _ in range(len(Adj)):                # main loop
9          u = Q.extract_min()                  # extract vertex with min estimate
10         for v in Adj[u]:                     # loop through out-going edges
11             relax(Adj, w, d, parent, u, v)   # relax!
12             Q.decrease_key(v, d[v])          # update key of vertex
13     return d, parent
```

This algorithm follows the same structure as the general relaxation framework. Lines 2-4 initialize shortest path weight estimates and parent pointers. Lines 5-7 initialize a priority queue with all vertices from the graph. Lines 8-12 comprise the main loop. Each time the loop is executed, line 9 removes a vertex from the queue, so the queue will be empty at the end of the loop. The vertex u processed in some iteration of the loop is a vertex from the queue whose shortest path weight estimate is smallest, from among all vertices not yet removed from the queue. Then, lines 10-11 relax the out-going edges from u as usual. However, since relaxation may reduce the shortest path weight estimate $d(s, v)$, vertex v's key in the queue must be updated (if it still exists in the queue); line 12 accomplishes this update.

Why does Dijkstra's algorithm compute shortest paths for a graph with non-negative edge weights? The key observation is that shortest path weight estimate of vertex u equals its actual shortest path weight $d(s,u) = \delta(s,u)$ when u is removed from the priority queue. Then by the upper-bound property, $d(s,u) = \delta(s,u)$ will still hold at termination of the algorithm. A proof of correctness is described in the lecture notes, and will not be repeated here. Instead, we will focus on analyzing running time for Dijkstra implemented using different priority queues.

Exercise: Construct a weighted graph with non-negative edge weights, and apply Dijkstra's algorithm to find shortest paths. Specifically list the key-value pairs stored in the priority queue after each iteration of the main loop, and highlight edges corresponding to constructed parent pointers.

Priority Queues

An important aspect of Dijkstra's algorithm is the use of a priority queue. The priority queue interface used here differs slightly from our presentation of priority queues earlier in the term. Here, a priority queue maintains a set of key-value pairs, where vertex v is a value and $d(s,v)$ is its key. Aside from empty initialization, the priority queue supports three operations: `insert(val, key)` adds a key-value pair to the queue, `extract_min()` removes and returns a value from the queue whose key is minimum, and `decrease_key(val, new_key)` which reduces the key of a given value stored in the queue to the provided `new_key`. The running time of Dijkstra depends on the running times of these operations. Specifically, if T_i, T_e, and T_d are the respective running times for inserting a key-value pair, extracting a value with minimum key, and decreasing the key of a value, the running time of Dijkstra will be:

$$T_{Dijkstra} = O(|V| \cdot T_i + |V| \cdot T_e + |E| \cdot T_d).$$

There are many different ways to implement a priority queue, achieving different running times for each operation. Probably the simplest implementation is to store all the vertices and their current shortest path estimate in a dictionary. A hash table of size $O(|V|)$ can support expected constant time $O(1)$ insertion and decrease-key operations, though to find and extract the vertex with minimum key takes linear time $O(|V|)$. If the vertices are indices into the vertex set with a linear range, then we can alternatively use a direct access array, leading to worst case $O(1)$ time insertion and decrease-key, while remaining linear $O(|V|)$ to find and extract the vertex with minimum key. In either case, the running time for Dijkstra simplifies to:

$$T_{Dict} = O(|V|^2 + |E|).$$

This is actually quite good! If the graph is dense, $|E| = \Omega(|V|^2)$, this implementation is linear in the size of the input! Below is a Python implementation of Dijkstra using a direct access array to implement the priority queue.

```
class PriorityQueue:                        # Hash Table Implementation
    def __init__(self):                     # stores keys with unique labels
        self.A = {}

    def insert(self, label, key):           # insert labeled key
        self.A[label] = key

    def extract_min(self):                  # return a label with minimum key
        min_label = None
        for label in self.A:
            if (min_label is None) or (self.A[label] < self.A[min_label].key):
                min_label = label
        del self.A[min_label]
        return min_label

    def decrease_key(self, label, key):     # decrease key of a given label
        if (label in self.A) and (key < self.A[label]):
            self.A[label] = key
```

If the graph is sparse, $|E| = O(|V|)$, we can speed things up with more sophisticated priority queue implementations. We've seen that a binary min heap can implement insertion and extract-min in $O(\log n)$ time. However, decreasing the key of a value stored in a priority queue requires finding the value in the heap in order to change its key, which naively could take linear time. However, this difficulty is easily addressed: each vertex can maintain a pointer to its stored location within the heap, or the heap can maintain a mapping from values (vertices) to locations within the heap (you were asked to do this in Problem Set 5). Either solution can support finding a given value in the heap in constant time. Then, after decreasing the value's key, one can restore the min heap property in logarithmic time by re-heapifying the tree. Since a binary heap can support each of the three operations in $O(\log |V|)$ time, the running time of Dijkstra will be:

$$T_{Heap} = O((|V| + |E|) \log |V|).$$

For sparse graphs, that's $O(|V| \log |V|)$! For graphs in between sparse and dense, there is an even more sophisticated priority queue implementation using a data structure called a **Fibonacci Heap**, which supports amortized $O(1)$ time insertion and decrease-key operations, along with $O(\log n)$ minimum extraction. Thus using a Fibonacci Heap to implement the Dijkstra priority queue leads to the following worst-case running time:

$$T_{FibHeap} = O(|V| \log |V| + |E|).$$

We won't be talking much about Fibonacci Heaps in this class, but they're theoretically useful for speeding up Dijkstra on graphs that have a number of edges asymptotically in between linear and quadratic in the number of graph vertices. You may quote the Fibonacci Heap running time bound whenever you need to argue the running time of Dijkstra when solving theory questions.

```
class Item:
    def __init__(self, label, key):
        self.label, self.key = label, key

class PriorityQueue:                        # Binary Heap Implementation
    def __init__(self):                     # stores keys with unique labels
        self.A = []
        self.label2idx = {}

    def min_heapify_up(self, c):
        if c == 0: return
        p = (c - 1) // 2
        if self.A[p].key > self.A[c].key:
            self.A[c], self.A[p] = self.A[p], self.A[c]
            self.label2idx[self.A[c].label] = c
            self.label2idx[self.A[p].label] = p
            self.min_heapify_up(p)

    def min_heapify_down(self, p):
        if p >= len(self.A): return
        l = 2 * p + 1
        r = 2 * p + 2
        if l >= len(self.A): l = p
        if r >= len(self.A): r = p
        c = l if self.A[r].key > self.A[l].key else r
        if self.A[p].key > self.A[c].key:
            self.A[c], self.A[p] = self.A[p], self.A[c]
            self.label2idx[self.A[c].label] = c
            self.label2idx[self.A[p].label] = p
            self.min_heapify_down(c)

    def insert(self, label, key):           # insert labeled key
        self.A.append(Item(label, key))
        idx = len(self.A) - 1
        self.label2idx[self.A[idx].label] = idx
        self.min_heapify_up(idx)

    def extract_min(self):                  # remove a label with minimum key
        self.A[0], self.A[-1] = self.A[-1], self.A[0]
        self.label2idx[self.A[0].label] = 0
        del self.label2idx[self.A[-1].label]
        min_label = self.A.pop().label
        self.min_heapify_down(0)
        return min_label

    def decrease_key(self, label, key):     # decrease key of a given label
        if label in self.label2idx:
            idx = self.label2idx[label]
            if key < self.A[idx].key:
                self.A[idx].key = key
                self.min_heapify_up(idx)
```

Fibonacci Heaps are not actually used very often in practice as it is more complex to implement, and results in larger constant factor overhead than the other two implementations described above. When the number of edges in the graph is known to be at most linear (e.g., planar or bounded degree graphs) or at least quadratic (e.g. complete graphs) in the number of vertices, then using a binary heap or dictionary respectively will perform as well asymptotically as a Fibonacci Heap.

We've made a JavaScript Dijkstra visualizer which you can find here:
`https://codepen.io/mit6006/pen/BqgXWM`

Exercise: CIA officer Mary Cathison needs to drive to meet with an informant across an unwelcome city. Some roads in the city are equipped with government surveillance cameras, and Mary will be detained if cameras from more than one road observe her car on the way to her informant. Mary has a map describing the length of each road and the locations and ranges of surveillance cameras. Help Mary find the shortest drive to reach her informant, being seen by at most one surveillance camera along the way.

Solution: Construct a graph having two vertices $(v, 0)$ and $(v, 1)$ for every road intersection v within the city. Vertex (v, i) represents arriving at intersection v having already been spotted by exactly i camera(s). For each road from intersection u to v: add two directed edges from $(u, 0)$ to $(v, 0)$ and from $(u, 1)$ to $(v, 1)$ if traveling on the road will not be visible by a camera; and add one directed edge from $(u, 0)$ to $(v, 1)$ if traveling on the road will be visible. If s is Mary's start location and t is the location of the informant, any path from $(s, 0)$ to $(t, 0)$ or $(t, 1)$ in the constructed graph will be a path visible by at most one camera. Let n be the number of road intersections and m be the number of roads in the network. Assuming lengths of roads are positive, use Dijkstra's algorithm to find the shortest such path in $O(m + n \log n)$ time using a Fibonacci Heap for Dijkstra's priority queue. Alternatively, since the road network is likely planar and/or bounded degree, it may be safe to assume that $m = O(n)$, so a binary heap could be used instead to find a shortest path in $O(n \log n)$ time.