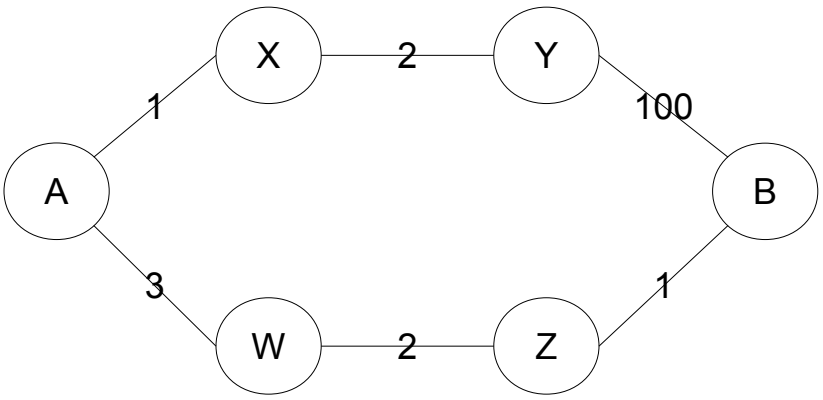
Lecture 40

Single source shortest path

We next consider the problem of finding the shortest path between edges in a weighted graph. Our discussion will focus on an undirected graph, but the analog discussion for directed graphs is equally relevant.

Finding the shortest path poses problems not seen in the Minimum Spanning Tree problem. Consider the following example:



If we are trying to find the shortest path from `A` to `B`, then we see that locally optimal solution at each step (similar to our approach in the Prim-Jarnik algorithm) does not work. The best path out of `A` is clearly `A-X` with weight 1. However, examining the graph for a few moments reveals that the shortest path from `A` to `B` is actually `A-W-Z-B`.

Now, consider that the edge `Y-B` has weight -100, rather than 100. We can't really say what the shortest path is. We can improve any path to `B` by from `B` to `Y`, then back to `B`. This technique will reduce the distance of our path by 200 each time we follow it, and it can conceivable be done as many times as we like.

Dijkstra's algorithm

The first algorithm we will look at is Dijkstra's algorithm. If we assume that there are no negative weight edges in the graph, then it will not only calculate the shortest path from `A` to `B`, but also the shortest path from `A` to all other vertices in the graph.

A pseudocode version of the algorithm appears below.

```
map<int,int> shortest_paths(int start,
        const map<int,list<pair<int,int> > > & graph) {

    map<int,int> parents;
    priorityqueue62 frontier;

    parents[start]=start;
    frontier.push(start, 0);

    while (!frontier.is_empty()) {
        int v = frontier.top_key();
        int p = frontier.top_priority();
        frontier.pop();

        for (the neighbors (n,w) of v) {
            if (n == parents[v])
                ; // do nothing
            else if (n is not in the frontier and has not been visited) {
                parents[n] = v;
                frontier.push(n, p + w);
            } else if (p + w < frontier.get_priority(n)) {
                parents[n] = v;
                frontier.reduce_priority(n, p + w);
            }
        }
    } // end while

    return parents;
}
```

If we consider the problem we had in out first example again, we see that the edge `A-X` was not on the shortest path from `A` to `B`. However, if we assume there were no negative weight edges, it **was** on the shortest path from `A` to `X`. (In fact, it was the entirety of the shortest path). This is the basic idea behind Dijkstra's algorithm. We begin at the start vertex, and at each step we choose another vertex for whom we know the shortest path. Thus, we calculate the shortest path from the start to all vertices, and just use the one we care about.

If we examine the algorithm above, we see that it works as follows. We add the start vertex to a priority queue, where the priority is the distance from the start node to that node. Since we are adding the start node itself first, and we don't have negative weight edges, we know that the distance from the start node to the start node is 0.

At each step, we take the lowest priority vertex off the priority queue. Then, for each of it's neighbors, we check to see if there is a cheaper path to that neighbor through the vertex we just removed.

If we have not visited that vertex before, then we have just found a path to it, which must be the cheapest one we know of so far, so we add the vertex to the priority queue. For it's distance, we use the distance to the vertex we just removed, plus the edge from the vertex to the neighbor. This is guaranteed to be a path.

If the neighbor is already in the priority queue, we must compare the distance of the current path to it (represented by it's current priority) to the distance of the path we just found (represented by the path to the vertex plus the weight of the edge from the vertex to the neighbor). If the new path is better, we update the distance and parent information.

Notice that once a vertex is selected off of priority queue, it will not be possible to find a shorter path to it (assuming no negative weight edges, of course). We won't prove this fact here, but you should be able to

work through the algorithm and convince yourself that it's true.

Run time analysis

How much work are we doing each time we push a vertex onto the frontier (priority queue), or pop a vertex off of the frontier? We are keeping vertices on the frontier, so $\log(|V|)$ work for each push, and each pop. Over the course of the algorithm, we push each vertex onto the frontier once, and pop it off once. That is, we push it on when we encounter it for the first time, and pop it off when we know that we are done with it. Thus, the total work of the algorithm just for pushing and popping the vertices on and off the priority queue is $|V| \log(|V|)$.

Now, how much work are we doing each time we call `reduce_priority`? Again, each call changes a priority value in the heap, so it can, in the worst case, result in $\log(|V|)$ work. How many times do we call `reduce_priority` over the course of the whole algorithm? We may, in the worst case, call it for every edge in the graph. That is, every edge may result in a better path to it's incident vertex. Thus, the total work contributed to the algorithm by calls to `reduce_priority` is $|E| \log(|V|)$.

If we assume that these two things drive the run time of the algorithm, then the total work is $|V| \log(|V|) = |E| \log(|V|) = (|V| + |E|) \log(|V|)$.

If the graph is especially sparse, then $|E|$ approaches $|V|$, and the algorithm is $|V| \log(|V|)$. If the graph is very dense, then the algorithm is $|V|^2 \log(|V|)$

Clearly, we can improve this if we can improve the performance of our heap. It turns out that we can, but it is beyond the scope of this class.