Graph Theory, Continued

Senior Computer Team
Jeff Chen, Jacob Steinhardt

November 2, 2006

1 Shortest Path

Previously, we covered different ways to traverse graphs. The following traversal algorithms are specialized for finding the shortest paths between vertices on the graph. Dijkstra's algorithm, which we went over last time, is a common shortest path tool, so let's review it quickly:

1.1 Dijkstra's Algorithm

Given an array of weights $w[i][j]$ =weight of edge from vertex i to vertex j, a boolean array visited[], and a dist[] array being the distance from the source vertex to every other vertex, we will find the shortest path between the source node and all other nodes. The runtime for a simple implementation is $O(V^2)$. We first start off at the source node, check its neighbors and update their dist[]s. Then we find the closest vertex to the source and repeat the process there, getting its neighbors and updating their costs. The pseudocode is as follows:

```
void dijkstra()
{
    set all dist to infinity;
    set all visited to 0;
    dist[source]=0;//source vertex to source vertex=0
    while(not all visited)
    {
        find unvisited vertex i with shortest path to source;
        visited[i]=1;
        for(each neighbor j of vertex i)
            if(dist[i]+w[i][j]<dist[j])
                dist[j]=dist[i]+w[i][j];
    }
}
```

Now we have found the shortest path from the source vertex to every other vertex.

1.2 Bellman-Ford Algorithm

The Bellman-Ford algorithm is an alternative to Dijkstras. It generally runs a bit slower, at $O(V * E)$, but will work on negative edge graphs, in addition to being easier to code than Dijkstra's algorithm.

```
Bellman-Ford(a graph with V vertices and E edges)
{
   bestdist = 0 for source, INF for all others
   for 1 to V-1
      for edges i=1 to E
         if bestdist[startpt[i]] + weight[i] < bestdist[endpt[i]]
            bestdist[endpt[i]]= bestdist[startpt[i]] + weight[i]

         //do this step only if the graph is undirected
         if bestdist[endpt[i]] + weight[i] < bestdist[startpt[i]]
            bestdist[startpt[i]]= bestdist[endpt[i]] + weight[i]
}
```

1.3 Floyd-Warshall Algorithm

So far we have two algorithms to find shortest path from one vertex to everything else. What if we need to find the path from every vertex to every other vertex? We need to use the almighty Floyd-Warshall.

The Floyd-Warshall operates at a whopping $O(V^3)$ worst-case, but is deliciously simple to code. It's useful, but *make sure you don't time out!* Sometimes there are less brute-forcey things you can do. This time we will have a dist[i][j] for vertex i to vertex j, and again, $w[i][j]$.

```
void floydwarshall()
{
//let's get all single-edge paths down
for(all dist)
dist[i][j]=weight[i][j];
for(k, all vertices)
for(i, all vertices)
for(j, all vertices)
if(dist[i][k]+dist[k][j]<dist[i][j])
dist[i][j]=dist[i][k]+dist[k][j];
}
```

2 Minimum Spanning Trees

A spanning tree if a graph is any connected sub-graph (if you recall, connected basically means we can use this tree to get from any vertex to any other vertex). A *minimal* spanning tree, or MST, is a spanning tree that has the lowest sum of the weights of its edges. There are two main algorithms for finding MSTs: Prim's algorithm and Kruskal's algorithm; both function on a greedy principle. As Prim's algorithm is easier to code, that is preferred. In sparse graphs, however, Kruskal's algorithm may provide a better alternative.

2.1 Prim's Algorithm

2.2 Kruskal's algorithm

Kruskal's idea is that we first sort the connection edges in increasing costs. Then we go through them in that order and see if using an edge is good or not. So the Greedy here is that if an edge does not create a cycle and the edge costs less than other edges, then use it. And it works in $O(ElogE)$ time. However, coding this algorithm is somewhat difficult.

```
Kruskal(a graph with V vertics and E edges)
{
   sort edges in increasing cost

   for each edge (1->V)
      if this edge does not create cycle
         then use this edge
}
```