Dijkstra's Algorithm

Ryan Jian, Yongkoo Kang

November 1, 2013

1 Introduction

Recall that a graph is defined as a set of vertices connected by a set of edges. The shortest path problem asks for the shortest path between two different vertices and is a basic problem in graph theory that shows up frequently in Silver. The problem has a number of variations that can be solved by various different algorithms.

2 Single Source Shortest Path

This variation asks for the shortest path from one vertex to all the other vertices in a graph. For the case of an unweighted graph, a breadth-first search will do the job. However, this breaks down for weighted graphs because the path with the least number of edges is not always the the path with the least total weight. As such, we have to turn to other algorithms to solve the problem.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm works by maintaining a set of distances estimating the length of the shortest path to all vertices and a set of visited vertices, greedily choosing the current shortest path and trying each of the different edges. At the start, the array of distances is initialized to infinity, except for the starting vertex which is initialized to 0. We then set the current vertex to the starting vertex. Afterwards, we mark the current vertex as visited and iterate through all the neighbors of the current vertex and *relax* the edge. That is, we add the length of the edge to the current distance. If this sum is lower than the distance to the neighboring vertex then we have it update the value in the set of distances. Next, we set the vertex with the minimum distance in the set of all distances that has not been visited to be the current vertex, and repeat this process until every vertex has been reached. Here's some pseudocode:

```
for each vertex v:
    dist[v] = infinity
    visited[v] = false
dist[start] = 0
while there are unvisited vertices:
    current = unvisited vertex with smallest distance in dist[]
    visited[current] = true
    for each neighbor v of current:
        if dist[current] + v.length < dist[v]:
            dist[v] = dist[current] + v.length
```

The time complexity of this algorithm depends on how fast the operations implementing the following two lines.

```
current = unvisited vertex with smallest distance in dist[]
dist[v] = dist[current] + v.length
```

The first operation is executed V times for each iteration of the while loop. The second is executed at most E times because each edge is processed at least once the algorithm terminates

If we have dist[] be just an array then the first operation is a linear search taking $O(V)$ time and the second operation is $O(1)$ time. This makes our final time complexity $O(E + V^2) = O(V^2)$.

However, if we make dist a priority queue implemented as a binary heap, then the first operation takes $O(\log V)$ and the second also takes $O(\log V)$, giving us a complexity of $O((E + V)\log V)$. Using a Fibonacci heap improves this to $O((E + V)\log V)$.

NOTE: The second operation implemented in a priority queue is called decrease-key, which changes a value in the priority queue without inserting or deleting anything. This is how priority queue Dijkstra was originally implemented. However, most of the priority queues bundled in a programming language only have push, pop, and find-min, and not the decrease-key operation. As a result we slightly modify the algorithm to use only these operations by having the priority queue hold all the distances we try.

```
for each vertex v:
    visited[v] = false
PQ.insert([0, start]) // a pair consisting of the current distance and the last vertex in the path
while PQ is not empty:
    d = PQ.findmin().distance
    current = PQ.findmin().v
    PQ.pop()
    if visited[current]:
        continue
    dist[current] = d // record the shortest distance
    visited[current] = true
    for each neighbor v of current:
        if not visited[v]:
            PQ.insert([d + v.length, v])
```

While this does use more memory than our original decrease-key approach, the time complexity is still the same. In fact, since decrease-key is a fairly complicated operation in priority queues, this modification ends up being faster most of the time.

2.1.1 Single Destination

If we're just looking for the shortest path between one pair of vertices, we can save a little bit of time and just end the loop when current = target.

3 Problems

1. (USACO Training Pages, butter) Farmer John owns a collection of pastures with weighted edges between some pairs of locations. Each pasture is inhabited by a cow, and the cows wish to all congregate at one of the pastures. Find the pasture at which the cows should meet in order to minimize combined travel distance

2. (USACO FEB12, relocate) FJ is moving! He is trying to find the best place to build a new farm so as to minimize his daily travel time.

   The region to which FJ plans to move has N towns ($1 \leq N \leq 10,000$). There are M bi-directional roads ($1 \leq M \leq 50,000$) connecting certain pairs of towns. All towns are reachable from each-other via some combination of roads. FJ needs your help selecting the best town as the home for his new farm.

There are markets in K of the towns ($1 \le K \le 5$) that FJ wants to visit every day. In particular, every day he plans to leave his new farm, visit the K towns with markets, and then return to his farm. FJ can visit the markets in any order he wishes. When selecting a town in which to build his new farm, FJ wants to choose only from the $N - K$ towns that do not have markets, since housing prices are lower in those towns.

Please help FJ compute the minimum distance he will need to travel during his daily schedule, if he builds his farm in an optimal location and chooses his travel schedule to the markets as smartly as possible.

3. (MIT 6.006 Quiz 2, How I met your midterm) Ted and Marshall are taking a roadtrip from Somerville to Vancouver (that's in Canada). Because it's a 52-hour drive, Ted and Marshall decide to switch off driving at each rest stop they visit; however, because Ted has a better sense of direction than Marshall, he should be driving both when they depart and when they arrive (to navigate the city streets). Given a route map represented as a weighted undirected graph $G = (V; E; w)$ with positive edge weights, where vertices represent rest stops and edges represent routes between rest stops, devise an efficient algorithm to find a route (if possible) of minimum distance between Somerville and Vancouver such that Ted and Marshall alternate edges and Ted drives the first and last edge.

4. (IOI '99, Traffic Lights) In the city of Dingilville the traffic is arranged in an unusual way. There are junctions and roads connecting the junctions. There is at most one road between any two different junctions. There is no road connecting a junction to itself. Travel time for a road is the same for both directions. At every junction there is a single traffic light that is either blue or purple at any moment. The color of each light alternates periodically: blue for certain duration and then purple for another duration. Traffic is permitted to travel down the road between any two junctions, if and only if the lights at both junctions are the same color at the moment of departing from one junction for the other. If a vehicle arrives at a junction just at the moment the lights switch it must consider the new colors of lights. Vehicles are allowed to wait at the junctions. You are given the city map which shows

   - the travel times for all roads (integers),
   - the durations of the two colors at each junction (integers)
   - the initial color of the light and the remaining time (integer) for this color to change at each junction.

   Your task is to find a path which takes the minimum time from a given source junction to a given destination junction for a vehicle when the traffic starts. In case more than one such path exists you are required to report only one of them.