CS61B Summer 2006
Instructor: Erin Korber
Lecture 25,26: 9,10 Aug.

Graphs - Shortest Path Algorithms

The *shortest path problem* is the problem of finding a path between two vertices of a graph such that the total weight of the edges on the path is minimized. These algorithms can also be applied to an unweighted graph to find the path of minimum length by simply treating it as a weighted graph with all edge weights equal (e.g. all set to 1).

We will be looking at three algorithms, each of which solves a different variation of this problem. For simplicity, we will be assuming that all of the graphs that we are dealing with are connected (if undirected) or strongly connected (if directed).

1 Single-source shortest path - Dijkstra's Alg.

The *single-source shortest path problem* is the problem of finding the shortest path to all other vertices (or to 1 particular destination vertex) in the graph from a given origin vertex. As we saw previously, on an unweighted graph, we can solve this problem easily using breadth-first search. Dijkstra's algorithm is a more sophisticated search that accounts for the edge weights as it traverses the graph (still visiting each node only once). To do this, we'll need to maintain a "distance" variable for each node that tracks the distance of the node from the origin along the shortest path found to it so far, in addition to a "previous" pointer to keep track of the path (which we used previously for BFS). The algorithm:

- For each vertex v in G, set v.distance = infinity and v.previous = null
- Set origin.distance = 0
- Insert all the vertices into a priority queue Q by distance
- while Q is not empty:
  - let u = Q.removeMin
  - for each edge e = (u,v) out from u
    - if u.distance + e.weight < v.distance
    - set v.distance to u.distance + e.weight
    - set v.previous to u

So we start by assuming that each node is infinitely far away, then move to the next closest node and update our estimates from that point.

You will notice that the priorities of the vertices are changing as the algorithm runs, so the simple implementations of priority queues that we were using previously will not be sufficient. There are a few ways to deal with this problem. One is to simply use an unsorted list as the priority queue, and search each time for the minimum priority to remove it. This has the advantage of being simple to implement, since we do not need to do anything to a vertex when its priority changes, but has the disadvantage that removeMin now takes $\Theta(n)$ time, instead of log(n). Another solution is to use a heap-based priority queue as before, but add an "changePriority" method to bubble a node up the tree when its distance changes. Note that bubbling up will be sufficient, since distances will only ever get smaller, not larger.

The running time of Dijkstra's algorithm will depend on the choice of priority queue implementation. Note that removeMin will occur exactly $|V|$ times, and changePriority (which would be called every time a distance is updated) will occur at most $|E|$ times (since for each vertex in the graph, it could be called for each edge out of that vertex).

For the unsorted list priority queue, removeMin takes $\Theta(|V|)$ time, and changePriority takes $\Theta(1)$ time (since we don't really need to do anything). So the total running time is in $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$.

For a binary heap-based priority queue, removeMin takes $\Theta(\log |V|)$ time as usual, and changePriority takes $\Theta(\log |V|)$ also. So the total time is in $\Theta(|V| \log |V| + |E| \log |V|) = \Theta((|V| + |E| \log |V|))$.

You may notice that Dijkstra's algorithm won't work in any situation where an edge may have a *negative* weight. (Can you see why?) We'll need a different algorithm to solve that problem.

2 Negative-weight single-source shortest path - / Bellman-Ford Algorithm

The Bellman-Ford algorithm can solve the single-source shortest path problem in the presence of negative edge weights. It is usually only used in this situation because it is not as fast as Dijkstra's algorithm.

Note that if the graph has a negative-weight cycle (a cycle for which the total sum of the edges on it is negative), then the shortest path is not well-defined, since we can make any path shorter by going around and around the negative- weight cycle. The algorithm will check for this and return an error if this is the case.

The algorithm:

- For each vertex v in G, set v.distance = infinity and v.previous = null
- Set origin.distance = 0
- For i = 0 to $|V| - 1$
  - For each edge e = (u,v) in G
    - if u.distance + e.weight < v.distance
    - set v.distance to u.distance + e.weight
    - set v.previous to u
- For each edge e = (u,v) in G
  - if u.distance + e.weight < v.distance
  - Error: graph contains a negative-weight cycle

As you can see from the loop structure, this algorithm runs in $\Theta(|V||E|)$ time.

2.1 Proof of correctness

Note: you are not responsible for this content, but you might be interested, so it's given here:

We can show by induction that this algorithm is correct. We will prove that after i iterations of the outer for loop:

- if u.distance is not infinity, then it is equal to the length of some path from the origin to u
- if there is a path from origin to u with at most i edges, then u.distance is at most the length of the shortest such path from origin to u.

Proof: Base case (i = 0): Before any iterations occur, origin.distance = 0 and u.distance = infinity for all other u in G, which is correct since there is no path from the origin to any other vertex with 0 edges.

Inductive case (assume that the statement holds for all values less than i and show that it holds for i). We first prove the first part. Consider a moment when a vertex v's distance is updated to u.distance + e.weight. By inductive hypothesis, u.distance is the length of some path from origin to u. Then u.distance + e.weight is the length of the path from origin to v that follows the path from source to u and then goes to v.

For the second part, consider the shortest path from origin to u with at most i edges. Let v be the last vertex before u on this path. Then, the part of the path from origin to v is the shortest path between source to v with i-1 edges. By the inductive hypothesis, v.distance after i-1 cycles is at most the length of this path. Therefore, e.weight + v.distance is at most the length of the path from origin to u. In the ith cycle, u.distance gets compared with e.weight + v.distance, and is set equal to it if that was smaller. Therefore, after i cycles, u.distance is at most the length of the shortest path from the origin to u that uses at most i edges.

When i equals the number of vertices in the graph, each path will be the shortest path overall, unless there are negative-weight cycles. If a negative-weight cycle exists, then given any path, a shorter one exists, so there is no shortest path. Otherwise, the shortest path will not include any cycles (because going around a non-negative weight cycle would not make the path shorter), so each shortest path visits each vertex at most once, and its number of edges is less than the number of vertices in the graph. Therefore, each path stored in the graph by this point is the shortest path.

3 All-pairs shortest path - Floyd-Warshall algorithm

If we want to find the shortest path between each pair of nodes in a graph, we could simply run Dijkstra'a algorithm or the Bellman-Ford algorithm for each node. Another simpler option (especially simple if your graph is represented as an adjacency matrix) is the Floyd-Warshall algorithm. Suppose that the graph is given as an adjacency matrix, with the vertices numbered from 1 to N (where N = $|V|$). We will use a 3-dimensional array A, where `A[k][i][j]` is the length of the shortest path from i to j that uses only the vertices 1... k.

The algorithm:

```
for i = 1 to N
   for j = 1 to N
      if there is an edge e = (i,j) in E
         dist[0][i][j] = e.weight
      else
         dist[0][i][j] = infinity

for k = 1 to N
   for i = 1 to N
      for j = 1 to N
         dist[k][i][j] = min(dist[k-1][i][j],
                   dist[k-1][i][k] + dist[k-1][k][j])
```

As you can see from the loop structure, this algorithm runs in $\Theta(|V|^3)$ time. It also has a very small constant factor (since very little work is done in the body of the inner loop).