The shortest path problem

Consider the problem of finding the shortest path between nodes s and t in a graph (directed or undirected). We already know an algorithm that will solve it for unweighted graphs - BFS. Now, what if the egdes have weights? Consider the dist[] array that we used in BFS to store the current shortest known distance from the source to all other vertices. BFS can be thought of as repeatedly taking the closest known vertex, u, and applying the following procedure to all of its neighbours, v.

```
bool relax( int u, int v ) {
    if( dist[v] <= dist[u] + 1 ) return false;
    dist[v] = dist[u] + 1;
    return true;
}
```

The procedure relax() returns true if we can improve our current best known shortest path from s to v by using the edge (u, v). In that case, BFS also updates dist[v] and adds v to the back of the queue.

Imagine colouring all vertices white before running BFS. Then all the vertices on the queue can be considered gray, and all the vertices that have been processed and removed from the queue are black. We can prove that BFS works by demonstrating the following invariant: at the beginning of each iteration, dist[v] is equal to the shortest path distance from s to v for all black vertices, v. At the beginning, the invariant is true because we have no black vertices. During each iteration of BFS, we pick the closest known vertex, u, (one of them, if there are several) and execute relax(u, v) on all of its neighbours, v. Finally, we colour u black (pop it from the queue). Since u was the the closest vertex (to the source), any other path to u that we might discover during subsequent iterations must be longer than dist[u]. Hence, the invariant holds for u - the only new black vertex that we get during one iteration. Eventually, when BFS terminates, dist[v] will be set to the length of the shortest path for all black (visited) vertices, v. All other vertices will have dist[] set to infinity - "unreachable".

Dijkstra's algorithm

The reason why BFS does not work for weighted graphs is very simple - we can no longer guarantee that the vertex at the front of the queue is the vertex closest to s. It is certainly the closest in terms of the number of edges used to reach it, but not in terms of the sum of edge weights. But we can fix this easily. Instead of using a plain queue, we can use a priority queue in which vertices are sorted by their increasing dist[] value. Then at each iteration, we will pick the vertex, u, with smallest dist[u] value and call relax(u, v) on all of its neighbours, v. The only difference is that now we add the weight of the edge (u, v) to our distance instead of just adding 1.

```
bool relax( int u, int v ) {
    int newDist = dist[u] + weight[u][v];
    if( dist[v] <= newDist ) return false;
    dist[v] = newDist;
    return true;
}
```

The proof of correctness is exactly the same as for BFS - the same loop invariant holds. However, the algorithm only works as long as we do not have edges with negative weights. Otherwise, there is no guarantee that when we pick u as the closest vertex, dist[v] for some other vertex v will not become smaller than dist[u] at some time in the future.

There are several ways to implement Dijkstra's algorithm. The main challenge is maintaining a priority queue of vertices that provides 3 operations – inserting new vertices to the queue, removing the vertex with smallest dist[], and decreasing the dist[] value of some vertex during relaxation. We can use a **set** to represent the queue. This way, the implementation looks remarkably similar to BFS. In the following example, assume that graph[i][j] contains the weight of the edge (i, j).

Example 1: $O(n^2+(m+n)\log(n))$ **Dijkstra's**

```
int graph[128][128];            // -1 means "no edge"
int n;                          // number of vertices (at most 128)
int dist[128];

// Compares 2 vertices first by distance and then by vertex number
struct ltDist {
    bool operator()( int u, int v ) const {
        return make_pair( dist[u], u ) < make_pair( dist[v], v );
    }
}

void dijkstra( int s ) {
    for( int i = 0; i < n; i++ ) dist[i] = INT_MAX;
    dist[s] = 0;

    set< int, ltDist > q;
    q.insert( s );
    while( !q.empty() ) {
        int u = *q.begin();                 // like u = q.front()
        q.erase( q.begin() );               // like q.pop()

        for( int v = 0; v < n; v++ ) if( graph[u][v] != -1 ) {
            int newDist = dist[u] + graph[u][v];
            if( newDist < dist[v] )         // relaxation
            {
                if( q.count( v ) ) q.erase( v );
                dist[v] = newDist;
                q.insert( v );
            }
        }
    }
}
```

First, we define a comparator that compares vertices by their dist[] value. Note that we can't simply do "return dist[u] < dist[v];" because a set keeps only one copy of each unique element, and so using this simpler comparison would disallow vertices with the same dist[] value. Instead, we exploit the built-in lexicographic comparison for pairs.

The dijkstra() function takes a source vertex and fills in the dist[] array with shortest path distances from s. First, all distances are initialized to infinity, except for dist[s], which is set to 0. Then s is added to the queue and we proceed like in BFS: remove the first vertex, u, and scan all of its neighbours, v. Compute the new distance to v, and if it's better than our current known distance, update it. The order of the 3 lines inside the innermost 'if' statement is crucial. Note that the set q is sorted by dist[] values, so we can't simply change dist[v] to a new value - what if v is in q? This is why we first need to remove v from the set, then change dist[v] and after that add it.

The running time is n*log(n) for removing n vertices from the queue, plus m*log(n) for inserting into and updating the queue for each edge, plus n*n for running the 'for(v)' loop for each vertex u. We can avoid the quadratic cost by using an adjacency list, for a total of O((m+n)log(n)).

Another way to implement the priority queue is to scan the dist[] array every time to find the closest vertex, u.

Example 2: O(n^2) Dijkstra's

```
int graph[128][128], n;            // -1 means "no edge"
int dist[128];
bool done[128];

void dijkstra( int s ) {
    for( int i = 0; i < n; i++ ) {
        dist[i] = INT_MAX;
        done[i] = false;
    }
    dist[s] = 0;

    while( true ) {
        // find the vertex with the smallest dist[] value
        int u = -1, bestDist = INT_MAX;
        for( int i = 0; i < n; i++ ) if( !done[i] && dist[i] < bestDist ) {
            u = i;
            bestDist = dist[i];
        }
        if( bestDist == INT_MAX ) break;

        // relax neighbouring edges
        for( int v = 0; v < n; v++ ) if( !done[v] && graph[u][v] != -1 ) {
            if( dist[v] > dist[u] + graph[u][v] )
                dist[v] = dist[u] + graph[u][v];
        }

        done[u] = true;
    }
}
```

We have to introduce a new array, done[]. We could also call it "black[]" because it is true for those vertices that have left the queue. First, we initialize done[] to false and dist[] to infinity. Inside the main loop, we scan the dist[] array to find the vertex, u, with minimal dist[] value that is not black yet. If we can't find one, we break from the loop. Otherwise, we relax all of u's neighbouring edges.

This seemingly low-tech method is actually pretty clever in terms of running time. The main while() loop executes at most n times because at the end we always set done[u] to true for some u, and we can only do that n times before they are all true. Inside the loop, we do O(n) work in two simple loops. The total is $O(n^2)$, which is faster than the first implementation as long as the graph is fairly dense ($m > n^2/\log(n)$). This is if we do use an adjacency list in the first implementation; otherwise, the second one will almost always be faster).

Dijkstra's algorithm is very fast, but it suffers from its inability to deal with negative edge weights. Having negative edges in a graph may also introduce negative weight cycles that make us re-think the very definition of "shortest path". Fortunately, there is an algorithm that is more tolerant to having negative edges – the Bellman-Ford algorithm.

The Bellman-Ford algorithm

Dijkstra's algorithm is a generalization of the BFS algorithm – meaning that Dijkstra's is itself a **graph search** algorithm. A search algorithm can be thought of as starting at some source vertex in a graph, and "search" the graph by walking along the edges and marking the vertices. These search algorithms do not make use of the fact that we already know before-hand the entire structure of the graph. This explains why Dijkstra's algorithm cannot handle negative weights – it can only search from what we have seen so far, and does not expect new "discoveries" at some later stage would affect what we have already processed.

The Bellman-Ford algorithm is a Dynamic Programming algorithm that solves the shortest path problem. It looks at the structure of the graph, and iteratively generates a better solution from a previous one, until it reaches the best solution. Bellman-Ford can handle negative weights readily, because it uses the entire graph to improve a solution.

The idea is to start with a base case solution $\mathbf{S_0}$, a set containing the shortest distances from s to all vertices, using no edge at all. In the base case, d[s] = 0, and d[v] = ∞ for all other vertices v. We then proceed to **relax** every edge once, building the set $\mathbf{S_1}$. This new set is an improvement over $\mathbf{S_0}$, because it contains all the shortest distances using one edge – ie. d[v] is minimal in $\mathbf{S_1}$ if the shortest path from s to v uses one edge. Now, we repeat this process iteratively, building $\mathbf{S_2}$ from $\mathbf{S_1}$, then $\mathbf{S_3}$ from $\mathbf{S_2}$, and so on... Each set $\mathbf{S_k}$ contains all the shortest distances from s using k edges – ie. d[v] is minimal in $\mathbf{S_k}$ if the shortest path from s to v uses **at most k** edges.

Example 3: Bellman-Ford algorithm

```
vector< pair<int,int> > EdgeList;      // A list of directed edges (u,v)
int graph[128][128];                   // Gives the weight
int n, dist[128];

void bellman-ford( int s ) {
    // Initialize our solution to the BASE CASE S0
    for( int i = 0; i < n; i++ )
       dist[i] = INT_MAX;
    dist[s] = 0;

    for( int k = 0; k < n-1; k++ ) {   // n-1 iterations
       // Builds a better solution Sk+1 from Sk
       for( int j = 0; j < EdgeList.size(); j++ ) {   // Try for every edge
          int u = EdgeList[j].first, v = EdgeList[j].second;
          if( dist[u] < INT_MAX && dist[v] > dist[u] + graph[u][v] ) // relax
              dist[v] = dist[u] + graph[u][v];
       }
    }
    // ... Now we have the best solution after n-1 iterations
}
```

The algorithm above basically implements this idea. We start with a base case $\mathbf{S_0}$, and repeatedly relax every edge to generate $\mathbf{S_{k+1}}$ from $\mathbf{S_k}$. Note that in the relaxation step, we don't relax an edge if dist[u] is infinity, or otherwise we may get overflow in the addition (conceptually we never want to relax such an edge anyway). Also note that the order of using the edges can affect the intermediate sets $\mathbf{S_k}$, because we may first relax an edge (u,v), then relax another edge (v,w) in the same step, while choosing the reverse order of these two edges may not relax them both. However, we now show that $\mathbf{S_{n-1}}$ is unique, and contains the shortest distance possible from s to any vertex v.

Proposition 4: (Correctness of Bellman-Ford) Let $\mathbf{S_k}$ denote the set of distances from s such that d[v] is minimal in $\mathbf{S_k}$ if the shortest path from s to v uses **at most k** edges. Then the Bellman-Ford algorithm builds $\mathbf{S_0, S_1, ..., S_{n-1}}$ iteratively. Also, $\mathbf{S_{n-1}}$ is the **best** solution, and it is **unique**.

Proof. We have already establish that the Bellman-Ford algorithm generates $\mathbf{S_0, S_1, ..., S_{n-1}}$ iteratively in the above paragraphs. Now, assuming that negative weight cycles reachable from the source do not exist in the graph, $\mathbf{S_{n-1}}$ will contain the shortest possible distances from s to any other vertices. This is because any walk in the graph will go into a cycle if we use more than n-1 edges, and since negative cycles do not exist, we never want to use these positive weight cycles as part of a shortest path. And, because $\mathbf{S_{n-1}}$ contains the **best** distances, it is **unique**. QED.

So, the Bellman-Ford algorithm is correct, but does it always terminate? It does, as we only have two loops, one running n-1 iterations, and the other going through all edges. Hence, the algorithm **always terminates**, and has a run time of **O(n*m).**

While the Bellman-Ford algorithm can handle negative weight edges readily, the correctness of the algorithm breaks down when negative weight cycles exist that is reachable from s. However, the nature of the algorithm allows us to **detect** these negative weight cycles. The idea is that, if a negative weight cycle exist, then $\mathbf{S_{n-1}}$ will be the same as $\mathbf{S_n, S_{n+1}, S_{n+2}}$, ... If we run the iteration step more than n-1 times, we will not be changing the answer. On the other hand, if a negative weight cycle exist, then one of its edges must have negative weight, and any such edge can be relaxed further even after n-1 iterations, decreasing some of the distances.

Hence, to detect negative weight cycles, we just need to run the Bellman-Ford algorithm, and when it terminates, check whether we can relax any edges. If we can, then that edge is reachable from a negative weight cycle, and the cycle is also reachable from the source.

Example 5: Detecting negative weight cycles in a graph

```
vector< pair<int,int> > EdgeList;      // A list of directed edges (u,v)
int graph[128][128];                   // Gives the weight
int n, dist[128];

int main() {
    // ... Set up the graph
    bellman-ford( 0 );                 // Run bellman-ford on s=0

    // Check for negative weight cycles reachable from s
    for( int j = 0; j < EdgeList.size(); j++ ) {   // Try for every edge
       int u = EdgeList[j].first, v = EdgeList[j].second;
       if( dist[u] < INT_MAX && dist[v] > dist[u] + graph[u][v] ) // can relax
           cout << "Negative cycle reachable from s exists." << endl;
           return 1;
       }
    }
    cout << "No negative cycle detected, shortest distances found." << endl;
    return 0;
}
```

Bellman-Ford is slower than Dijkstra's, but with this added functionality of handling negative weights and detecting negative cycles easily, it can be more useful in some cases. In particular, in a directed acyclic graph (one with no cycles), we can use Bellman-Ford to find the **longest path** from s to any vertices v, by simply changing all the positive weights to negative, and vice versa. Note that finding the longest path in a general graph is NP-hard.