Shortest Path Algorithms

Albert Gural

January 27, 2012

1 Introduction

Many USACO problems are based off of graph theory (sometimes disguised as a different type of problem), and many involve finding a shortest path from some start node to some goal node. Usually it's not too hard to tell when a problem requires a shortest path algorithm. The hard part is deciding which shortest path algorithm you should use. What follows are the four shortest path algorithms that will most often be useful in USACO. I've provided some actual C++ code snipets, found on our SCT Wiki: `http://tjsct.wikidot.com/shortest-path`, followed by a quick summary. If you want to implement any of these solutions, make sure to check out the wiki, since the following code examples are not complete. Note: In the Big-O time complexities, V is the number of vertices and E is the number of edges.

2 Floyd-Warshall: $O(V^3)$

```
for(int k = 0; k < V; k++)
    for(int i = 0; i < V; i++)
        for(int j = 0; j < V; j++)
            // Edge Relaxation
            if(dist[i][k] + dist[k][j] < dist[i][j])
               dist[i][j] = dist[i][k] + dist[k][j];
```

3 Dijkstra (no heap): $O(E + V^2)$

```
cur = 1; priolist[1] = 0;
while(cur!=V) {
    long long tempmin = INTMAX;
    visited[cur] = 1;
    for(int i = 0; i < adjlist[cur].size(); i++) {
        // Edge Relaxation
        priolist[adjlist[cur][i].second] <?=\
        priolist[cur] + adjlist[cur][i].first;
    }
    // Pick the next node by searching for the one with min distance
    for(int i = 1; i <= V; i++) {
        if(visited[i]==0 && priolist[i] < tempmin) {
            cur=i;
            tempmin = priolist[i];
        }
    }
}
```

4 Dijkstra (heap): $O((E+V)\log(V))$

Note: In this implementation, `priolist` is a `vector`, but it could be replaced with a `priority_queue`.

```
priolist.push_back(pair<long long, int>(0LL,1));
while(!priolist.empty()) {
    int cur = priolist[0].second;
    int curdist = priolist[0].first;

    //pop_heap takes the head of the heap and puts it on the
    //end of the vector, then we take it off.
    //also we use greater to make a min heap (default heap is max)
    pop_heap(priolist.begin(),priolist.end(),\
             greater< pair<long long, int> >());
    priolist.pop_back();

    if(cur == V) {
        fout << curdist << endl;
        break;
    }
    if(visited[cur] != 0) {
        continue;
    }
    visited[cur]=1;
    for(int i = 0; i < adjlist[cur].size(); i++) {
        if(visited[adjlist[cur][i].second] == 0) {
            //first we put the desired item on the back of
            //the vector for heaping
            priolist.push_back(pair<long long, int>(curdist\
            + adjlist[cur][i].first, adjlist[cur][i].second));
            //then we do the heaping
            push_heap(priolist.begin(), priolist.end(),\
            greater< pair<long long, int> >());
        }
    }
}
```

5 Bellman-Ford: $O(EV)$

```
for (int i = 0; i < MAXV; i++)
    bestd[i] = INFTY;
bestd[0] = 0;
// Bellman-Ford Algorithm
for (int i = 1; i < v; i++)
    for (int j = 0; j < el.size(); j++)
        bestd[el[j][1]] = min(bestd[el[j][0]] + el[j][2], bestd[el[j][1]]);
// Negative-cycle detection
for (int i = 0; i < e; i++) {
    if (bestd[el[i][1]] > bestd[el[i][0]] + el[i][2]) {
        fout << "NONE" << endl;
        break;
    }
}
```

6 Cheat Sheet

The following cheat sheet is a basic guide of when to use which shortest path algorithm (suggested one is bolded). When considering which algorithm to use, make sure to note the following:

- Whether you are given the start node or whether you will be queried a random start/end node.
- Whether the graph is sparse ($E \propto V$) or dense ($E \propto V^2$).
- Whether the graph doesn't or does have negative edge weights ($W \geq 0$ or $W \in \mathbb{R}$, respectively).

		Given Start Node		Any Start/End	
Sparse ($E \propto V$)	$W \geq 0$	**Heap Dijkstra**	$O(V \log(V))$	**Heap Dijkstra**	$O(V^2 \log(V))$
	$W \in \mathbb{R}$	**Bellman-Ford**	$O(V^2)$	**Floyd-Warshall** or Bellman-Ford	$O(V^3)$
Dense ($E \propto V^2$)	$W \geq 0$	**No-Heap Dijkstra**	$O(V^2)$	**Floyd-Warshall** or No-Heap Dijkstra	$O(V^3)$
	$W \in \mathbb{R}$	**Floyd-Warshall** or Bellman-Ford	$O(V^3)$	**Floyd-Warshall**	$O(V^3)$

7 Final Notes

In general, this cheat sheet should effectively find the most efficient algorithm for a given shortest path problem, at least as far as USACO is concerned. In all cases with $W \geq 0$, you *could* use a Fibonacci Heap Dijkstra algorithm (Big-O: $O(E + V \log(V))$). For USACO, the increased coding effort is not worth the gains in efficiency. Another note is that both Bellman-Ford and Floyd-Warshall can deal with negative edge weights. The only issue is whether there are negative cycles. These can be tested for in both algorithms using the same routine - try one more minimization pass and see if any edges relax. If any edges relax, then there is a negative cycle.