CP implementations

David del Val

December 23, 2020

Contents

Chapter 1

Graphs

1.1 Dijkstra

Shortest path from `orig` node to `dest` (or to every node) in a graph that does not contain negative edges. It chooses the best path greedily in each iteration and, therefore, it only works on graphs without negative weights.

```
20 int dijkstra(int orig, int dest, vector<vector<pii>> &graph) {
21     vi dists(graph.size(), INT_MAX / 10);
22     priority_queue<pii, vector<pii>, greater<pii>> q;
23
24     q.push({0, orig});
25     dists[orig] = 0;
26     while (!q.empty()) {
27         auto a = q.top();
28         q.pop();
29         if (a.se == dest) { // If we already reached the destination
30             return a.fi;
31         }
32         if (dists[a.se] != a.fi) { // If we have found a better option
33             continue;
34         }
35         for (auto b : graph[a.se]) { // Else iterate through the edges
36             if (dists[b.fi] > dists[a.se] + b.se) {
37                 dists[b.fi] = dists[a.se] + b.se;
38                 q.push({dists[b.fi], b.fi});
39             }
40         }
41     }
42     return dists[dest];
43 }
```

Running time: $\mathcal{O}(V + E \log(E))$
(V = vertices, E = edges)

Observations

- If we ignore the check in line 30, we can return the distances vector, which will contain the shortest distance from `dist` to every other node.
- If we are doing some kind of pruning it is imperative that we prune as many branches as possible in the main loop. That is to say, we should introduce as many `if` statements in line 36 to make sure that we run the `for` loop as few times as possible.

  An example of this approach is problem `UVA-11635`. In that problem, we add a lot of branches to the queue (we may run the `for` loop twice in some nodes) but we prune them in the main loop. Thus the running time is still acceptable.

1.2 Bellman Ford

Shortest parth from `orig` to every other node. It is slower than Dijkstra but it works on graphs with negative weights.

This algorithm works by trying to relax every edge $V-1$ times. If there are no negative cycles, after $V-1$ iterations, we must have found the minimum distance to every node. Therefore if after these iterations, we run another iteration and the distance to a node decreases, we must have a negative cycle.

```
20 vi bellmanFord(int orig, vector<pair<pii, int>> edges, int n) {
21     vi dist(n, INT_MAX / 10);
22     dist[orig] = 0;
23     for (int i = 0; i < n - 1; ++i) {
24         for (auto e : edges) {
25             dist[e.fi.se] = min(dist[e.fi.se], dist[e.fi.fi] + e.se);
26         }
27     }
28     // dist[i] contains the shortest path from 0 to i
29
30     //We can now check for a negative cycle
31     bool negativeCycle = false;
32     for (auto e : edges) {
33         if (dist[e.fi.se] > min(dist[e.fi.se], dist[e.fi.fi] + e.se)) {
34             negativeCycle = true;
35         }
36     }
37     return dist;
38 }
```

Running time: $\mathcal{O}(VE)$
(V = vertices, E = edges)

Observations

- If we keep track of the distance that decrease when we check for a negative cycle, we will get at least one node of every negative cycle present in the graph.

  We can use this, for instance, to check if we can reach a node with a cost smaller than a given bound. If it is connected to a node in a negative cycle, it's distance will be as small as we want it to be (by looping in the cycle).

  This can be seen at play in `UVA-10557`

- If we modify slightly the main loop, iteration i will be the result of considering paths of at most $i + 1$ edges:

```
for (int i = 0; i < n - 1; ++i) {
        for (auto e : edges) {
            dists2[e.fi.se] = min(dists2[e.fi.se], dists[e.fi.fi] + e.se);
        }
    dists = dists2;
}
```

  This can be seen at play in `UVA-11280`