CPS 149s: Graph Problems

Patrick Reynolds and Joseph Tate

Fall, 1999

Introduction

Graphs are often the best way to represent networks, maps, and paths, which show up in a small but significant fraction of programming contest problems. Having good graph representations and algorithms handy can make otherwise difficult problems relatively easy.

Any problem that asks for a path, especially a shortest path, or a list of connectable entities is a good candidate for a solution that makes use of a graph. Some examples and counter-examples follow in later sections.

Graph Representations

Graphs are generally stored either as adjacency lists or as adjacency matrices. A better discussion of these representations is in CLR Chapter 23, but here's a summary.

Adjacency lists refer to a list of lists: one large list contains one smaller list for each node, indicating the nodes that are reachable from that node. So:

```
vector<list<int> > adjacency;
adjacency[2].push_back(3);    // path from 2 to 3 exists
```

Adjacency matrices refer to a large matrix where a one in (row, col) indicates the existence of a path from V_{row} to V_{col}. For example:

```
int adjacency[50][50];
adjacency[4][9] = 1;          // path from 4 to 9 exists
```

Adjacency matrices sometimes store weights instead: a number indicates the distance between two adjacent nodes, or ∞ indicates that no path exists. E.g.,

```
int adjacency[50][50];
adjacency[4][9] = 4;          // path from 4 to 9 exists, costs 4 units
adjacency[5][6] = INFINITY;   // no path exists from 5 to 6
```

Note that adjacency lists can indicate directed or undirected graphs; adjacency matrices inherently represent directed graphs. `adjacency[x][y]` need not equal `adjacency[y][x]`.

Graph Algorithms

Three classes of graph algorithms are most relevant for programming contest problems: depth-first search, shortest path calculation, and connectivity calculation. You should already know DFS by now, but the Firetruck problem (below) and Getting in Line (homework) serve as examples, just in case. Shortest path calculation is the most common and is used in Jill's Bike (below) and Fire! (homework). Connectivity calculation is useful but less common; an example is provided in Calling Circles (homework).

Shortest path calculation is done by means of graph search (for unweighted graphs) or Dijkstra's algorithm (for weighted graphs). See any algorithms textbook for details. Jill's Bike uses Dijkstra's algorithm.

Connectivity calculation is best done by filling in unweighted adjacency matrices using the following axiom:

```
adjacency[x][y] && adjacency[y][z] => adjacency[x][z]
```

Loop through the matrix applying that axiom until it cannot be applied any further. Then `adjacency[x][y]` indicates that a (possibly indirect) path exists from V_x to V_y.

Examples

Jill's Bike

This problem presents a simple map of one-way streets. Ignoring the elevation information, we get an unweighted, directed graph with up to 400 vertices. The elevation information has a simple effect: any path with a climb of more than 10 meters is discarded from the graph. Once you have constructed this graph (adjacency lists are probably best), performing a simple graph search yields the shortest valid path from source to destination. Since both source and destination are variable, the graph search must be performed independently for each proposed path.

Firetruck

The input data for this problem is a set of undirected edges for a graph with an unknown (but < 21) number of nodes. An adjacency list probably works best here, too; be sure to do both `adjacency[a].push_back(b)` and `adjacency[b].push_back(a)` because the graph is undirected (i.e., reading A B from the input means that A→B and B→A are both valid paths). Once your graph is constructed, perform a DFS to find all paths from 1 to the desired destination. Standard DFS techniques apply: keep a global adjacency list, mark each node as visited to prevent loops, unmark when backtracking, and keep global variables to hold each path and the total number of paths.

Counterexamples

Bee Breeding

This problem is a poor candidate for graph algorithms (despite the goal of "find the length of the shortest path") because the number of nodes is potentially huge and because adjacency is relatively difficult to calculate.

A better approach is to find any valid path and then optimize it by cancelling out steps that go in opposing directions. My technique for finding a valid path consists of finding a path from A to the start of A's ring, from B to the start of B's ring (backwards), and from the start of A's ring to the start of B's ring. A ring is all cells that are a given distance from cell 1. See the solution for details.

The Unsinkable Ship?

This problem is a poor candidate for graph algorithms (again, despite the desire to find a valid path) because the connectivity changes as the passenger flees the ship: what was formerly a valid edge between two nodes may cease to be valid if either of the nodes floods. None of the elementary graph algorithms here is appropriate for dynamic graphs.

A better approach is to figure out in advance when each square of the ship will flood and then perform a DFS, checking at each step whether or not the available lifeboats have been exhausted and whether or not the current square is flooded and thus not a valid path. Again, see the solution for details.

References

- Representations of graphs: CLR 23.1 (p. 465)
- Breadth-first graph search: CLR 23.2 (p. 470)
- Dijkstra's algorithm: CLR 25.2 (p. 527)

CLR is *Introduction to Algorithms* by Cormen, Leiserson, and Rivest. Borrow a copy from a professor or a graduate student if you need one. Page numbers refer to the first edition. The second edition (by Cormen, Leiserson, Rivest, and Stein) also covers graph algorithms.