BFS for SSSP

Hello, and welcome to competitive programming. Today we are going to talk about single source shortest path.

Objectives

A single source shortest path means we have one node we consider a root or a source, and we want to find the shortest path from that source to all the other nodes. There are several algorithms that handle this, depending on the circumstances.

Your objective for this video is to implement the BFS shortest path algorithm for unweighted graphs.

The Algorithm

If our graph is unweighted, we can use an augmented breadth first search to generate a shortest path spanning tree. You will want to keep an array of distances and an array of parents. If you only need the distance and not the path itself, you can skip the parent array.

To start, you can initialize the distance array to all infinity. Minus one actually serves as a pretty good infinity here, since in this situation we can't get negative distances.

Then we put the root `a` into a queue and initialize it's distance to 0.

(next)

We now go though the BFS steps. We dequeue `a` , and add nodes `b` , `c` , and `d` to the queue. At the same time we set their distances to 1 and their parent to `a` .

(next)

Next we dequeue `b` and add `e` to the queue. `e` has distance 2 and parent `b` .

(next)

Next we dequeue `c` and add `f` to the queue. `f` has distance 2 and parent `c` .

(next)

Next we dequeue `d` . All it's neighbors have been visited already, so nothing happens.

(next)

Finally, we dequeue `e` . This sets `g` 's distance to 3 and its parent to `e` .

Notice how we have a nice spanning tree, with all the edges on a shortest path from the root to a node.

Now lets look at the implementation.

Implementation

This is very similar to the BFS code you've seen before, we just add a couple lines.

First we create a distance vector, and initializes source `s` distance to 0.

Next we create a queue and enqueues `s` .

Then we create an integer vector for the parents.

Now we enter the main loop.

Line 4 pops the head of queue into `u` .

We then loop over `u` ’s neighbors.

For each neighbor, we set `v` to be the entry in the adjacency vector.

If the distance array has infinity for that node, it means we haven’t visited it yet, so we update its distance, update the parent, and push the neighbor into the queue.

That’s it for this algorithm. If the edges happen to be weighted, this algorithm is not going to work. You’ll have to use Dijkstra’s algorithm or Bellman Ford instead.

Next
Transcript — The Bitmask Technique →

Copyright © 2019 - Mattox Beckman