Dijkstra’s Algorithm

Hello, and welcome to competitive programming. Today we are going to go over Dijkstra’s algorithm.

Objectives

The objective is to implement Dijkstra to solve a single-source shortest path problem. The version we will use here will work with unmodified C++ priority queues.

The Algorithm

You use Dijkstra’s algorithm when you have a weighted graph. It can be directed or undirected, though it’s more common to see it with a directed graph. If you use this with an undirected graph, you may need to keep track of the parents so you don’t loop back to a parent from a child node, but this can be safely omitted if you don’t have negative edges in your graph.

To start off, create a distance array and parent array like we did in the BST algorithm. But this time we also initialize a priority queue.

We will start off by pushing a pair into the queue: the source node `a` and its distance `0`.

(next)

In the main loop we dequeue `a` and check all of its edges. The operation we perform is called “relaxing”. We add the current node distance to the edge and see if that is smaller than the neighbor’s current distance. So in this case we are looking at `b` from `a`. `b` started with weight infinity, and 0 plus 2 is less than infinity, so we update the node `b`'s distance to 2, and put `b/2` into the priority queue.

(next)

Similarly, we look at the edge to `c` and relax it, and enqueue `c/4`.

(next)

Finally for `a`, we look at the edge to `d`, and relax it to `d/10`.

This accounts for all the edges incident to `a`, so we dequeue the next node `b`.

(next)

The first edge visited this time was the edge from `b` to `d`. `d`'s distance started as 10, but now it decreases to 7. We will now add `d/7` to the priority queue. Notice that the `d/10` entry is still in there! The way we handle it is this: when we dequeue a node, we check if the current weight in the graph is identical to what came out of the queue. If it is, we process the edges as normal. If the current weight is lower, then we know that this queue entry is obsolete, so we ignore it.

This is called lazy deleting.

In a more classic version of the algorithm we would update the item in the queue, but the C++ standard library priority queue lacks the ability to do this. Rather that reimplement priority queues, we use lazy deleting instead.

Now let's look at the edge from `b` to `e`.

(next)

This edge sets the distance to `e` to 8.

(next)

Dequeuing `c`, we set `f` to 11.

(next)

We now dequeue `d/7`. Looking at all the edges, none of them change anything, so nothing happens here.

(next)

We dequeue `e/8`, and update `g`.

There are three things left on the queue. The `d/10` entry will be ignored since the 10 in larger than the current weight. The `f` and `g` entries also will not change anything.

(next)

So here is the completed MST.

Implementation

Here is the implementation. As always, after studying this be sure you can write it yourself quickly.

We create a distance array `dist` and initialize it to infinities. From reading the problem statement you will be able to come up with a number large enough to be safe.

Next we create a priority queue over integer pairs. The second template argument says we want a vector of integer pairs for our container, and the third argument says we want to use pair comparison to determine priority.

We push the source node onto the queue.

In the main loop, we pop off the front. The first element of the pair is the distance, which we call `d`, and the second is the node index, which we call `u`.

The next line handles the lazy delete, checking if the distance has decreased since this element was added to the queue.

Finally, we visit all the neighbors, comparing the distance through this node to the distance previously recorded. If we find a shorter path, we update the neighbor and add it to the queue.

And that’s it for Dijkstra!

Next
Transcript — Divide and Conquer →