SHORTEST PATH: DIJKSTRA'S ALGORITHM

TONY DE LA NUEZ

October 28, 2016

Abstract

Dijkstra's algorithm is a greedy algorithm used to calculate the shortest path from a single source to any give node on a weighted, directed graph G = (V, E) where V are vertex and E are edges. Yes, vertexes is a word. A **greedy algorithm** [1] is an algorithm that follows the pattern of making the locally optimal choice at each stage with the end goal being a global optimum.In the context of Dijkstra's algorithm, this means that at every node (or vertex) the algorithm looks for the best route (edge) possible. *In the graph used for Dijkstra's algorithm, all edge weights are nonnegative*. Greedy algorithms work by recursively making a set of objects from the smallest parts of the algorithm. As we know from recursion, larger recursive problems are solved by using the solution to smaller versions of the same problem. When we use greedy algorithms, the advantage lies in the ease of solution of the smaller instances of the problem.

CONTENTS

1 THE ALGORITHM

When computing the shortest path, Dijkstra's algorithm maintains a set S of vertexes whose final shortest-path weights from the source s have already been calculated. In our implementation in Java, we continually updated the map to reflect our "visited" nodes. The algorithm then continues to pick unvisited nodes (call them u with the minimum shortest-path estimate, adds u to the visited list S, and relaxes all the edges that leave u. The following animation may help visualize the process:
`https://upload.wikimedia.org/wikipedia/commons/5/57/Dijkstra_Animation.gif`

[1] `https://en.wikipedia.org/wiki/Greedy_algorithm`

1.1 *Pseudocode from Cormen*

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

2 TIME COMPLEXITY

Dijkstra's algorithm maintains the min-priority queue we implemented by calling INSERT, EXTRACT-MIN, and DECREASE-KEY (in RELAX).
INSERT and EXTRACT-MIN are called once per vertex.
Each vertex added to set S once, each edge in the adjacency list Ajd[u] observed in **for** once.

Number of Edges: $\mid E \mid$ meaning at most $\mid E \mid$ calls to DECREASE-KEY.
Number of Vertices: $\mid V \mid$
Now, runtime depends on implementation.
Using a simple list-based priority queue, each INSERT and DECREASE-KEY operation takes O(1) time and each EXTRACT-MIN operation takes O(V) time.
List-based priority queue implementation run time:

$$O(V^2 + E) = O(V^2) \tag{1}$$

If E = o(V^2/lgV) we can use a min-heap.
Each EXTRACT-MIN operation then takes O(lg V). Time to build min heap is O(V). Each DECREASE-KEY operation takes O(lg V) time and there are still only $\mid E \mid$ of those operations.

Priority Queue with Binary Queue Min-heap implementation run time:

$$O((V + E)lgV) \tag{2}$$

which is

$$O(ElgV) \tag{3}$$

if all vertexes reachable from source.

Priority Queue with Fibonacci Heap implementation run time:

$$O(VlgV + E) \tag{4}$$

3 ALTERING THE INPUT GRAPH

First thing to note: you're allowed to have negative graphs in Dijkstra's algorithm, but you will no longer be guaranteed to find the optimal path. The path you find will be *locally optimal* rather than *globally optimal*. The only time you can guarantee that Dijkstra's algorithm will return the globally optimal path is when all edge weights are nonnegative.

For simplicity's sake, think of the input graph as defined in the abstract. This is a graph G = (V,E). We have vertexes, and edges.

I'll now try to address Dr. Cytron's list of questions on Piazza about the input graph:

1. "changes to the input graph (what happens if a bridge breaks on a road, for example, what if there is traffic on a street, how can those

things be factored into the algorithm to provide shortest (least-time) paths)?"

2. "why negative weights would cause problems"

Answering question 1:
We made a very, very simplistic implementation of Dijkstra's Algorithm. In our implementation, we basically think of the weight of an edge as just the distance (say, in miles) between the vertexes that the edge connects. In a more clever implementation of this algorithm, we would use many different things to calculate the weight of each edge. Basically we have different resources to keep in mind and we give them each a priority. A few resources to think of:

- Cost (tolls)
- Scenery (who doesn't love a pretty view)
- Speed limits
- Freeway
- Distance (miles)
- Traffic
- Broken bridge, etc.

When factoring these into our weight equation, we'd simply assign a priority value to each of them and include a way to calculate them into our weight. For example, let's say you care about three things in order of rising significance: scenery, traffic, and distance.

You'd then want to assign priority values to each and make sure that the larger your distance becomes, the higher the weight becomes. But not only this, an increase in distance must yield a higher increase in the weight than an increase in your scenery value.

It would look something like this:

- Distance (miles) = d
- Traffic (1 - 100), 100 being worst traffic = t
- Scenery (1 - 100), 100 being worst possible view = s
- weight = w

$$w = 10d + 7t + 2s \tag{5}$$

We notice in equation 1 that a one mile increase in distance is 5 times as significant as a 1 point increase in the quality of the scenery. This way, we've properly added priority values to make sure that in order to pick a scenic route over a short one or one with less traffic, it better be like driving along the California coast[2].

A broken bridge could be simulated by adding in an infinite value to the weight, seeing as you can't get past a broken bridge (therefore making it an impassable route).

[2] http://beta.costtodrive.com/wp-content/uploads/2013/06/Pacific-Coast-Hwy-cover.jpg

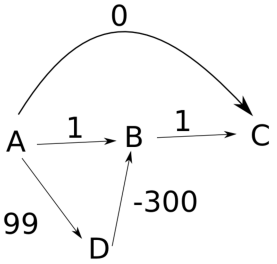
3.1 *Negative Weights Example*

I found this example on StackOverflow, linked here:
`http://stackoverflow.com/questions/6799172/negative-weights-using-dijkstras-algorithm/6799344#6799344`
The explanation:
"Assume the edges are directed from left to right as in your example,your algorithm will work as follows:

1. First, you set d(A) to zero and the other distances to infinity.
2. You then expand out node A, setting d(B) to 1, d(C) to zero, and d(D) to 99.
3. Next, you expand out C, with no net changes.
4. You then expand out B, which has no effect.
5. Finally, you expand D, which changes d(B) to -201.

Notice that at the end of this, though, that d(C) is still 0, even though the shortest path to C has length -200. Your algorithm thus fails to accurately compute distances in some cases. Moreover, even if you were to store back pointers saying how to get from each node to the start node A, you'd end taking the wrong path back from C to A."

Fundamentally, because the algorithm is greedy and looks for local minimum, the negative values can end up tricking it. We're only processing the nodes once, so we don't recompute the path upon finding the negative value. We also see that the edge A -¿ C is not needed for the explanation of the negative values, as it would still break. We cannot send changes back through after "visiting" a node.



A very important note the poster made in the comments:

"This is how Dijkstra's algorithm works. The algorithm does not explore paths, but instead works by processing nodes. Each node is processed exactly once, so as soon as we process the B node and get that its distance is 1, we will never revisit the node B or attempt to update its distance." -templatetypedef

4 PROOF OF CORRECTNESS

Rather than retyping the correctness proof I suggest looking at page 659 of the textbook, or if you don't have the book follow this link:
`http://www.math.ucsd.edu/~fan/teach/202/notes/04greed.pdf`
The proof starts on slide 48 (looking at the bottom corners). It's not as involved as the one in the book but the ideas are explained well. But seriously, get the book. It's extremely helpful.

5 REFERENCES

- `http://www.math.ucsd.edu/~fan/teach/202/notes/04greed.pdf`
- `http://stackoverflow.com/questions/6799172/negative-weights-using-dijkstras-algorithm/6799344#6799344`

- https://en.wikipedia.org/wiki/Greedy_algorithm
- https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm