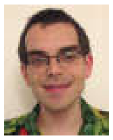
Dijkstra's Algorithm For Shortest Paths



By Alex Allain

Graphs are useful for representing many problems in computer science and in the real world. Applications of graph representations range from the seemingly simple, finding out whether a node is reachable from another node, to the extremely complex, such as finding a route that visits each node and minimizes the total time (the "travelling salesman" problem). A common, but solvable problem is that of problem of simple path finding. Generally, the task is determining the shortest path from a given node to any other node on the graph.

This might be useful for problems such as designing a MapQuest-like program, or allowing computer players in a game to navigate quickly from place to place. Each vertex would be a location of interest, and each edge would be a path or road between those locations--most likely, a directed edge. (For the purposes of giving driving directions, it might make sense to also include each intersection as a separate vertex of the graph.)

One of the most common algorithm algorithms for solving this problem is Dijkstra's algorithm, which solves the problem of finding shortest paths from a particular source node to any other node where no edge has a negative weight (i.e., you don't go "back in time" when traversing that edge). Although negative weights may sometimes be necessary, for many applications, they are impossible and the possibility of negative edge weights can be ignored.

Dijkstra's Algorithm

Dijkstra's algorithm works on the principle that the shortest possible path from the source has to come from one of the shortest paths already discovered. A way to think about this is the "explorer" model--starting from the source, we can send out explorers each travelling at a constant speed and crossing each edge in time proportional to the weight of the edge being traversed. Whenever an explorer reaches a vertex, it checks to see if it was the first visitor to that vertex: if so, it marks down the path it took to get to that vertex. This explorer must have taken the shortest path possible to reach the vertex. Then it sends out explorers along each edge connecting the vertex to its neighbors.

It is useful for each vertex of the graph to store a "prev" pointer that stores the vertice from which the "explorer" came from. This is the vertex that directly precedes the current vertex on the path from the source to the current vertex.

The pseudocode for Dijkstra's algorithm is fairly simple and reveals a bit more about what extra information needs to be maintained. Vertices will be numbered starting from 0 to simplify the pseudocode.

```
Given a graph, G, with edges E of the form (v1, v2) and vertices V, and a
source vertex, s

dist : array of distances from the source to each vertex
prev : array of pointers to preceding vertices
i    : loop index
F    : list of finished vertices
U    : list or heap unfinished vertices

/* Initialization: set every distance to INFINITY until we discover a path */
for i = 0 to |V| - 1
    dist[i] = INFINITY
    prev[i] = NULL
end

/* The distance from the source to the source is defined to be zero */
dist[s] = 0

/* This loop corresponds to sending out the explorers walking the paths, where
 * the step of picking "the vertex, v, with the shortest path to s" corresponds
 * to an explorer arriving at an unexplored vertex */

while(F is missing a vertex)
   pick the vertex, v, in U with the shortest path to s
   add v to F
   for each edge of v, (v1, v2)
        /* The next step is sometimes given the confusing name "relaxation"
        if(dist[v1] + length(v1, v2) < dist[v2])
            dist[v2] = dist[v1] + length(v1, v2)
            prev[v2] = v1
            possibly update U, depending on implementation
        end if
    end for
end while
```

There are a few things to keep in mind in this pseudocode. First, every distance is initialized to INFINITY, except the distance from the source. This suggests that a real implementation needs an actual notion of infinity -- perhaps INT_MAX, defined in limits.h, would do. Notice also that the concept of a pointer to a vertex and the use of numbers to refer to vertices isn't clearly distinguished. In an implementation, the use of NULL for the prev pointers might not be appropriate (if vertices are numbered, using -1 would be more reasonable).

Another factor to notice is that the definition of U, which stands for Unfinished, is ambiguous. This is because there are several ways of implementing Dijkstra's algorithm, none of which is strictly better in all cases. To understand the choices, it's important to know how to think about computational efficiency in a graph context.

To talk about computational efficiency on graphs, there are two variables of interest -- the number of edges, |E|, and the number of vertices, |V|. Keep in mind that the number of edges cannot exceed |V|^2 (and only if every vertex is connected to every other vertex in a directed graph). Algorithms that perform in "linear time" on graphs run in O(|E| + |V|) time.

One option is that U is simply a list of vertices that have not yet been visited. In this case, the algorithm will need to traverse U every time it picks the next vertex. This is an O(|V|) operation repeated |V| times, for a total run time of O(|V|^2).

Another option is to use a heap to keep track of which node should come next as one property of heaps is that they always have the next element at the top (either the minimum or the maximum). Removal or update operations on heaps require O(log|V|) time. Since it's possible that in the above algorithm each edge may cause a vertex's position in the heap to change, a heap may require O(|E|log|V|) time. Overall, Dijkstra's algorithm takes O(|E|log|V|) time in this case, as all other terms (such as O(|V|log|V|) for finding the nearest vertex and updating the heap and the O(|V|) initialization) are dominated (assuming there are at least |V| edges in the graph.

Other Approaches to The Problem of Shortest Paths

There are a variety of algorithms for solving the "single-source shortest path" problem--finding the shortest path from a single vertex to all the other possible vertices. Some algorithms only work in special cases, but are faster, relying on the properties of the special case. For instance, the algorithm we're interested in looking at, Dijkstra's algorithm, only works if none of the edges on the graph have negative weights -- the "time" it takes to traverse the edge is somehow less than 0. (When might this come up in practice?)

Algorithms that can handle even the case of negative edge weights do exist, and are interesting in their own right. They are, however, significantly less efficient computationally.

The algorithm that handles negative weighted edges runs in O(|E| * |V|) time, which is significantly slower -- for so-called "dense" graphs with many edges, it can approach a cubic time of O(|V|*|E|) as |E| is O(|V|^2).

On the other hand, Dijkstra's algorithm can be implemented in O(|V|^2) or O(|E|*log(|V|)) time depending on how the next node to consider is chosen. Importantly, as |E| increases, it approaches |V|^2, so choosing the right programming representation requires knowing something about the potential inputs (or choosing between two representations on the fly).

Limitations

Once thing we haven't looked at is the problem of finding shortest paths that must go through certain points. This is a hard problem and is reducible to the Travelling Salesman problem--what this means in practice is that it can take a very long time to solve the problem even for very small inputs.

Related articles

Using Graphs in Computer Science

Using Heaps

Algorithmic Efficiency and Big-O Notation