CS 2112 : Object-Oriented Design and Data Structures – Honors
Fall 2014

Dijkstra's single-source shortest path algorithm

Topics:

The single-source shortest path problem

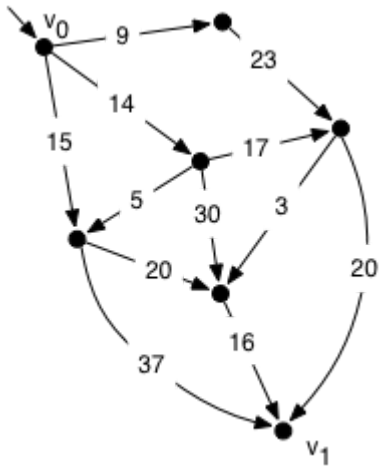
Dijkstra's algorithm

Proving correctness of the algorithm

Extensions: generalizing distance

Extensions: A*

The single-source shortest path problem



Suppose we have a weighted directed graph like the following, and we want to find the path between two vertices with the minimum total weight. Interpreting edge weights as distances, this is a **shortest-path problem**. There is a path with total weight 50, but it is not trivial to find it. We could find the shortest path by enumerating all possible permutations of the vertices and checking which one, but there are O(V!) such permutations.

Finding shortest paths in graphs is a very useful ability. In a sense, we have already seen an algorithm for finding shortest paths: breadth-first search finds shortest paths from a root node or nodes to all other nodes, where the length of a path is simply the number of edges on the path. In general it is useful to have edges with different "distances", so shortest-path problems are expressed in terms of weighted graphs, where the weights represent distances. (Weighted undirected graphs can be represented as weighted directed graph where each undirected edge is converted to a pair of edges in each direction.)

Both breadth-first search and depth-first search are instances of the abstract tricolor algorithm. The tricolor algorithm is nondeterministic: it allows certain things to be done at certain steps in the algorithm without requiring them to be done. BFS and DFS **resolve** the nondeterministic choices in different ways, and therefore visit nodes in different orders. In this lecture we will see a third instance of the tricolor algorithm, which solves the **single-source shortest path** problem.

We will be assuming that all edges have nonnegative weights. If edges have negative weights, it is possible to have cycles with net negative weight. In such a graph, minimum distance doesn't make much sense, because such a cycle can be traversed an arbitrary number of times. (The Bellman-Ford algorithm can be used in this case.)

We will be solving the problem of finding the shortest path from a given root node or set of root nodes. If used to find a path from multiple root nodes, the algorithm will not find the distances

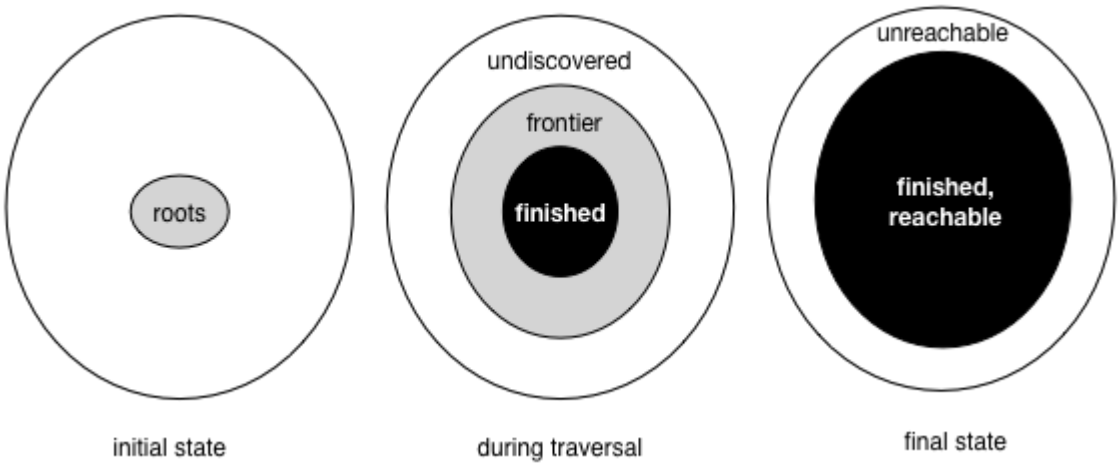
from all of the possible starting points. To answer this question, the single-source shortest path algorithm can be used repeatedly, once for each starting node. However, if the graph is dense, it is more efficient to use an algorithm for solving the all-pairs shortest-path problem. The Floyd-Warshall algorithm is the standard algorithm, and covered in CS 3110 or CS 4820.

Dijkstra's algorithm

The key insight behind Dijkstra's algorithm is to generalize breadth-first search, which both discovers and finishes nodes in order of path length from the root. When edges have arbitrary nonnegative weights, they can't be discovered in distance order, but they can be finished in distance order, as we will see.

Dijkstra's algorithm is an instance of the tricolor algorithm. Recall that a tricolor algorithm changes white nodes to gray and gray nodes to black while never allowing a black node to have an outgoing edge to a white node.

The progress of the algorithm is depicted by the following figure. Initially there are no black nodes and the roots are gray. As the algorithm progresses, reachable white nodes turn into gray nodes and these gray nodes eventually turn into black nodes. Eventually there are no gray nodes left and the algorithm is done.



For Dijkstra's algorithm, the three colors are represented as follows:

> **White** (undiscovered) nodes have their distance field set to ∞ because no path to them has been found yet. (`v.dist = ∞`)
>
> **Gray** (frontier) nodes have their distance field set to the total distance of some path from a root to the node. In addition, like BFS, the algorithm keeps track of frontier nodes in a queue. Unlike BFS, the queue is a **priority queue**, which we explain shortly. (`v.dist < ∞`, `v∈frontier`)
>
> **Black** (finished) nodes have their distance field set to the true minimum distance from the roots, and are not in the frontier priority queue. (`v.dist < ∞` , `v∉frontier`)

Dijkstra's algorithm starts with the roots in the priority queue at distance 0 (v.dist = 0), so they are gray. All other nodes are at distance ∞. The algorithm then works roughly like BFS, except that when node is popped from the priority queue, the node with the lowest current distance is

popped. When the algorithm completes, all reachable nodes are black and hold their true minimum distance.

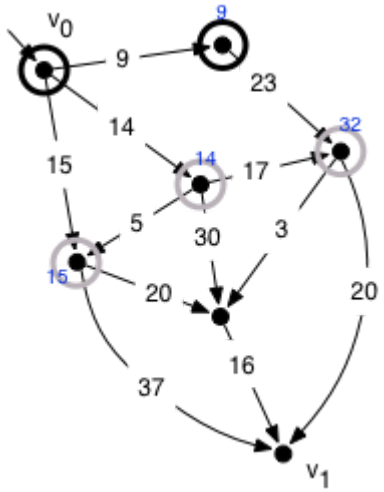
Here is the pseudo-code:

```
frontier = new PriorityQueue();
root.distance = 0;
fronter.push(root);
while (frontier not empty) {
    Vertex v = frontier.pop();
    foreach (Edge e == (v, Vertex v', int d)) {
        if (v'.dist == ∞) {
            v'.dist = v.dist + d;
            frontier.push(v');            // color v' gray
        } else {
            v'.dist = min(v'.dist, v.dist + d);
                // v' already gray, but update to shorter distance
            frontier.increase_priority(v');
        }
    }
    // color v black
}
```

Dijkstra's algorithm is an example of a **greedy algorithm**: an algorithm in which making simple local choices (like choosing the gray vertex with the lowest distance) lead to global optimal results (minimal distances computed to all vertices). Many other problems, such as finding a minimum spanning tree, can be solved by greedy algorithms. Other problems can't.

Example

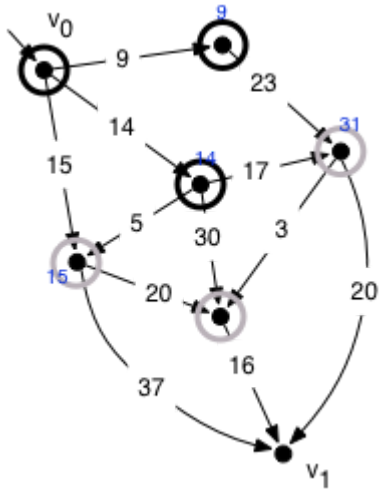
Suppose we run the algorithm on the graph above, with v_0 as the only root node. On the first loop interation we put the three successors of v_0 in the queue and update their distances accordingly. On the second iteration, we pop the node with the lowest distance (9) and add its own successor to the queue at distance 9+23=32. The new gray frontier consists of the three nodes shown in the figure below.



After the first iteration

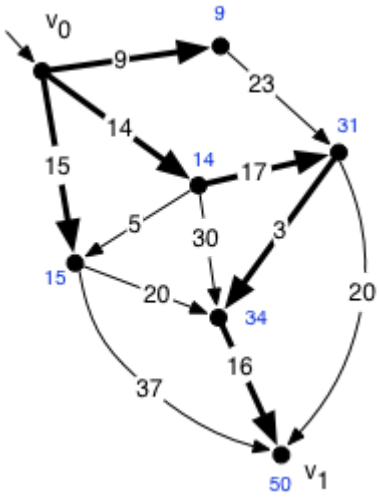
The next one to come off the queue is the one at distance 14. Inspecting its successors, we discover a new white vertex and set its distance to 44. We also discover a new path of length 31

to a node already on the fronter, so adjust its distance.



After the second iteration

At the end, the fronter is empty, all reachable nodes are black, and we have computed the minimum distance to each of them:



How do we recover the actual shortest paths? Observe that along any minimum-distance path containing an edge v→v' with length d, v'.dist = v.dist + d. If that were not true, there would have to be another shorter path. We can't have v'.dist > v.dist+d because there is clearly a path to v' with length v.dist+d. And we can't have v'.dist < v.dist+d, because that means we could make a shorter path to the final vertex by taking the better path to v' and concatenating it onto the tail of the current path starting at v'.

We can find the best paths, then, by working backward from each destination vertex v', at each step finding the predecessor satisfying this equation. We iterate the process until a root node is reached. The thick edges in the previous figure show the edges that are traversed during this process. They form a **spanning tree** that includes every vertex in the graph and shows a minimum-distance to each of them.

Performance of Dijkstra's algorithm

The algorithm does some work per vertex and some work per edge. The outer loop happens once for each vertex (as we will show), and uses the priority queue to get the minimum of up to V vertices. Getting the minimum vertex must take at least O(lg V) time. Otherwise we could use a priority queue to sort items faster than the lower bound of Ω(n lg n). In fact, one straightforward way to implement a priority queue is as a balanced binary tree. Its operations, including finding the minimum, are all O(lg V).

We also do work proportional to the number of edges; in the worst case every edge will cause the distance to its sink vertex to be adjusted. The priority queue needs to be informed of the new distance. This will take O(lg V) time if we use a balanced binary tree. So the total time taken by the algorithm is O(V lg V) + O(E lg V). Since the number of reachable vertices is O(E), this is O(E lg V).

It turns out there is a way to implement a priority queue as a data structure called a Fibonacci heap, which makes the operation of updating the distance O(1). The total time is then O(V lg V + E). However, Fibonacci heaps have poor constant factors so in practice they are not used. They only make sense for very large, dense graphs where E is not O(V).

Showing Dijkstra's algorithm works

To show this algorithm works, we will need a loop invariant.

Let us start by defining an **interior path** to a node v to be a path to v that starts from a root node and gets to v using only black vertices (except possibly for v itself).

The loop invariant for the outer loop is then the usual tricolor black–white invariant, plus the following:

1. For every finished vertex v and frontier vertex v', v.dist ≤ v'.dist.
2. For every discovered (gray or black) vertex, v.dist is the length of the shortest *interior* path from the roots to v.

Initialization. We can see that the invariant is true initially, because all the discovered vertices are roots at distance 0.

Postcondition. When the loop finishes, all paths are interior paths, so each distance must be the minimum.

Maintenance. We consider each of the two parts of the loop invariant in turn.

1. Each iteration of the loop first colors the minimum-distance gray node v black. This preserves (1) because distance(black) ≤ v ≤ distance(gray). Some new vertices may be discovered on each iteration, but they will all be on the frontier at a distance greater than v.dist too. So the first part of the invariant holds. We can see from this that vertices are moved from gray to black in order of increasing distance.
2. We need to show that we have the correct interior path distance to every discovered vertex. Coloring v black doesn't create new interior paths to it, but it might create new paths to other nodes.

   We don't have to worry about new interior paths to any nodes that are not successors to v, because any such interior path must have as its second-to-last step a different black node

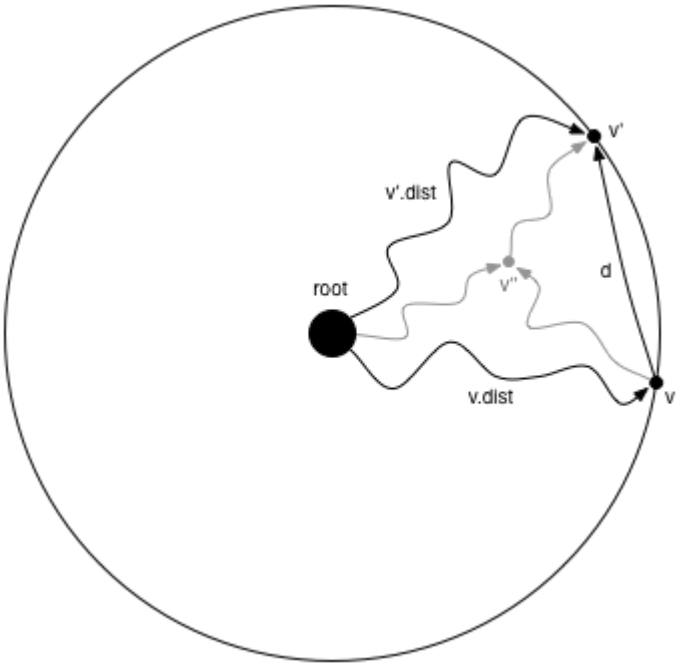
than v, one that is already at a closer distance than v. So going through v cannot produce a shorter interior path.

However, we still need to think about the interior paths to successors to v. Consider an arbitrary successor v' encountered in the inner loop. It can have one of three colors:

black. By part 1 of the invariant, any black successor v' must have v'.dist ≤ v.dist, so there can be no shorter interior path via v.

white. By the tricolor invariant, there is no preexisting interior path to v'. Therefore the new path via v with distance v.dist+d is the *only* interior path.

gray. If the node is already on the frontier, there is an existing interior path whose length is recorded in v'.dist. In addition, there is a new interior path to v' via v, with total distance v.dist. The code compares these two paths and picks the shorter one. Coloring v black can also create other new interior paths that don't go directly to v' via the new edge, but instead go via some other black node v". By invariant part (1), v".dist ≤ v.dist, so any such interior path must be at least as long as the preexisting interior path that goes from the roots directly to v" and then to v', and we know by invariant part (2) that *that* path is no shorter than v'.dist.



Generalizing shortest paths

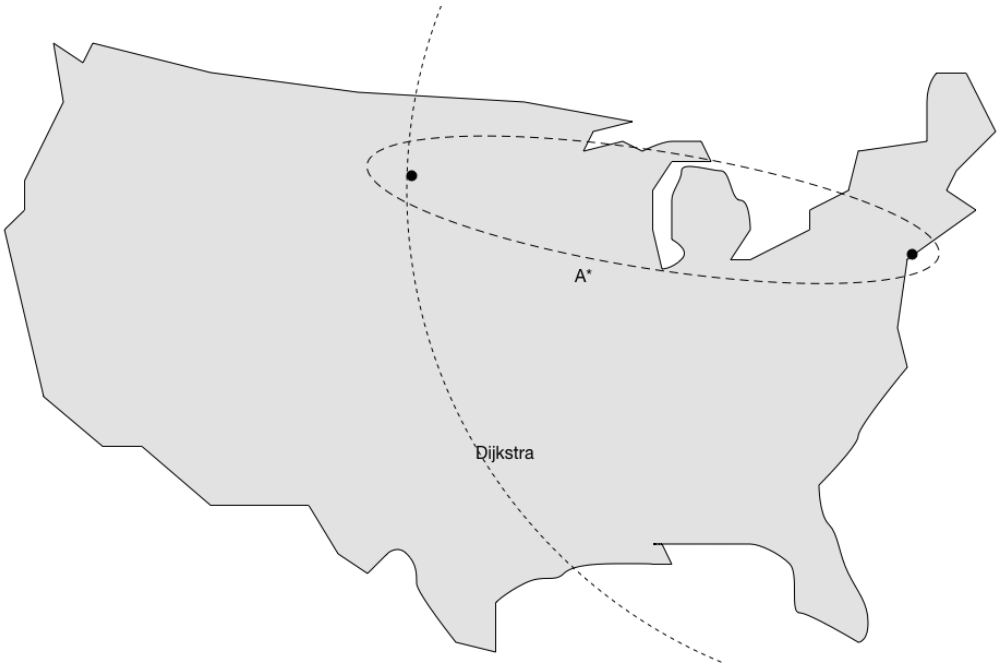
The weights in a shortest path problem need not represent distance, of course. They could represent time, for example. They could also represent probabilities. For example, suppose that we had a state machine that transitioned to new states with some given probability on each outgoing edge. The probability of taking a particular path through the graph would then be the *product* of the probabilities on edges along that path. We can then answer questions like “what is the most likely path that the system will follow in order to arrive at a given state,”, by solving a shortest path problem in a weighted graph in which the weights are the negative logarithms of the probabilities (since (-log a) + (-log b) = -(log ab)).

Heuristic search (A*)

Dijkstra's algorithm is a simple and efficient way to search graphs. However, in some applications it can search much more of the graph than necessary if we are only interested in the

shortest path to a particular vertex. One simple optimization is to stop the algorithm at the point when the destination vertex is popped off the priority queue. We know that v.dist is the minimal distance to v at that point because it is being colored black.

A second and more interesting optimization is to take advantage of more knowledge of distances in the graph. Dijkstra's algorithm searches outward in distance order from the source node and finds the best route to every node in the graph. In general we don't want to know every best route. For example, if we are trying to go from NYC to Mount Rushmore, we don't need to explore the streets of Miami, but that is what Dijkstra's algorithm will do:



A* search generalizes Dijkstra's algorithm to to take advantage of knowledge about the node we are trying to find. The idea is that for any given vertex, we may be able to estimate the remaining distance to the destination. We define a **heuristic function** h(v) that constructs this estimate. Then, the priority queue uses v.dist + h(v) as the priority instead of just v.dist. Depending on how accurate the heuristic function is, this change causes the search to focus on nodes that look like they are in the right direction.

Adding the heuristic to the node distance effectively changes the weights of edges. Given an edge from vertex v to v' with weight d, the change in priority along the edge is $d + h(v') - h(v)$. If h(v) is a perfect estimator of remaining distance, this quantity will be zero along the optimum path; in that case we can trivially "search" for the optimum path by simply following such zero-weight edges. More realistically, h(v) will be inaccurate. If it is inaccurate by up to an amount ε, we end up searching around the optimum path in a region whose size is determined by ε.

However, we have to be careful when defining h(v). If for an edge $v \rightarrow v'$ with distance d, the total distance including the heuristic function *decreases* along the edge, the conditions of Dijkstra's algorithm (nonnegative edge weights) are no longer met. In that case we have an **inadmissible heuristic** that could cause us to miss the optimal solution. A heuristic function is **admissible** if it never overestimates the true remaining distance. When searching with an admissible heuristic, each node is seen only once and the optimum path has been found immediately when the goal node is reached.

If using an inadmissible heuristic, negative-weight edges might cause us to find a new path to an already “finished” node, pushing it back on to the frontier queue. Even after finding the goal

node at some distance D, It is necessary to keep searching *above* priority level D to account for these negative-weight edges. Despite this issue, the reason we might want to use an inadmissible heuristic is that it can be more *accurate* than an admissible heuristic, and thus guide the search more effectively to the goal. In some cases inadmissible heuristics can pay off.

A* search is often used for single-player game search and various other optimization problems.

Notes by Andrew Myers, 11/15/12.