Shortest Paths

The discussion in these notes captures the essence of Dijkstra’s algorithm discussed in textbook [pages: 652-657], but the representation of the solution slightly diverges from the traces shown in the textbook.

In the textbook, the Dijkstra’s algorithm is applied to a directed weighted input graph. Wherein our discussion in this notes, we are going to look at the Dijkstra’s algorithm applied to undirected weighted input graph.

In a weighted graph, the weights on the edges usually stand for something meaningful—distances between cities, bandwidths across network links, etc. It is often the case that we are interested in finding the shortest or least-expensive path from one vertex to another—the so-called shortest path problem. Note: we are now talking about “shortest” in terms of the sum of the weights of the edges along the path. Earlier in the chapter, when we talked about breadth-first search, we used “shortest” to mean “fewest edges.”

Once again we will assume we have a connected graph. It turns out that in order to find the shortest path from vertex $v$ to vertex $w$ we must also find the shortest paths from $v$ to every other vertex in the graph (reason—if we don’t know the distance from $v$ to some vertex $u$ then we can’t be certain that there is no shorter path to $w$ that goes through $u$). Therefore, the version of shortest paths we will study is sometimes called the single-source shortest paths problem, meaning we are finding distances from one vertex (called the “source”) to all other vertices. We’ll use the famous “Dijkstra’s algorithm.”

Dijkstra’s Algorithm: An Example

Consider the following directed graph:

We want to find shortest paths from vertex 0 to every other vertex in the graph. The algorithm proceeds as follows. For each vertex $v$, keep track of two things: The best-known shortest distance from 0 to $v$, denoted by $d(v)$, and the vertex just before $v$ along a path having this shortest distance,
denoted \( p(v) \). Initially, we know only that \( d(0) = 0 \); the distance to all other vertices is unknown: \( d(v) = \infty \), and there is no preceding vertex, so \( p(v) = ? \).

In table form, using the example above, we have:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d(0), p(0) )</td>
<td>0, 0</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
</tr>
<tr>
<td>1</td>
<td>( d(1), p(1) )</td>
<td>( d(2), p(2) )</td>
<td>( d(3), p(3) )</td>
<td>( d(4), p(4) )</td>
<td>( d(5), p(5) )</td>
<td></td>
</tr>
<tr>
<td>0, 0</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td></td>
</tr>
<tr>
<td>0, 0</td>
<td>9, 0</td>
<td>( \infty, ? )</td>
<td>20, 0</td>
<td>23, 0</td>
<td>( \infty, ? )</td>
<td></td>
</tr>
</tbody>
</table>

The algorithm now repeats the following step until all vertices have been marked:

find the unmarked vertex \( v \) with the smallest value of \( d(v) \)
mark \( v \)
for each \( w \) adjacent to \( v \):
    if \( w \) is not marked
        let \( x \) be the weight on the edge between \( v \) and \( w \)
        if \( d(v) + x \) is less than \( d(w) \)
            set \( d(w) = d(v) + x \), set \( p(w) = v \)

Observe what is happening: when we select a new vertex \( v \) to be marked, we check all the neighbors of \( v \) to see if there is a shorter path to them that goes through \( v \).

Let’s carry this out on the sample graph. Each row in the table, reading from top to bottom, stands for one iteration of the process. Marked vertices have their column entries shaded.

Initially, vertex 0 has the smallest value of \( d \), so we mark it and update distances to its neighbors:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d(0), p(0) )</td>
<td>0, 0</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
</tr>
<tr>
<td>1</td>
<td>( d(1), p(1) )</td>
<td>( d(2), p(2) )</td>
<td>( d(3), p(3) )</td>
<td>( d(4), p(4) )</td>
<td>( d(5), p(5) )</td>
<td></td>
</tr>
<tr>
<td>0, 0</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td></td>
</tr>
<tr>
<td>0, 0</td>
<td>9, 0</td>
<td>( \infty, ? )</td>
<td>20, 0</td>
<td>23, 0</td>
<td>( \infty, ? )</td>
<td></td>
</tr>
<tr>
<td>0, 0</td>
<td>9, 0</td>
<td>12, 1</td>
<td>20, 0</td>
<td>19, 1</td>
<td>( \infty, ? )</td>
<td></td>
</tr>
</tbody>
</table>

The unmarked vertex with the smallest value of \( d(v) \) is vertex 1, so we carry out the above step, marking 1 and inspecting its unmarked neighbors 2 and 4. In both cases we have a shorter path to those vertices going through 1:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d(0), p(0) )</td>
<td>0, 0</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
</tr>
<tr>
<td>1</td>
<td>( d(1), p(1) )</td>
<td>( d(2), p(2) )</td>
<td>( d(3), p(3) )</td>
<td>( d(4), p(4) )</td>
<td>( d(5), p(5) )</td>
<td></td>
</tr>
<tr>
<td>0, 0</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td>( \infty, ? )</td>
<td></td>
</tr>
<tr>
<td>0, 0</td>
<td>9, 0</td>
<td>( \infty, ? )</td>
<td>20, 0</td>
<td>23, 0</td>
<td>( \infty, ? )</td>
<td></td>
</tr>
<tr>
<td>0, 0</td>
<td>9, 0</td>
<td>12, 1</td>
<td>20, 0</td>
<td>19, 1</td>
<td>( \infty, ? )</td>
<td></td>
</tr>
</tbody>
</table>

Vertex 2 now has the smallest \( d(v) \) among the unmarked vertices, so the next step lets us update the distances to vertices 4 and 5:
Vertex 5 now has the smallest distance from 0, so we mark it and note a shorter distance to 4:

The minimum distance among the two remaining vertices is 17, to vertex 4:

We are essentially done! The last vertex is already marked with the shortest distance to vertex 0.

Study the graph to confirm that we really have identified the shortest paths.

Dijkstra’s algorithm allows us to identify which edges belong to the shortest paths by following the $p(v)$ values backwards from each vertex to 0. For instance, $p(3) = 4$, $p(4) = 5$, $p(5) = 2$, $p(2) = 1$, $p(1) = 0$; the shortest path from 0 to 3 is $0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3$.

### Implementing Dijkstra’s Algorithm

We did not look at the Java code for this algorithm. Not surprisingly, it involves a min-priority queue, since at each step we must locate the vertex whose distance from 0 is the minimum. However, we run into a problem:

In Dijkstra’s Algorithm, the distances to vertices are updated during the course of the algorithm. In other words, the “priorities” of the vertices in the priority queue change during the execution of the algorithm.
When we studied priority queues back in chapter 2 (in connection with Heap Sort), we required just two operations: insert a key into the priority queue and delete the key with the minimum value (or maximum, in the case of heap sort). We did not implement a method for changing the value of a key in the priority queue.

Consider the “min-heap” implementation of a priority queue. For purposes of illustration, suppose we use the values in row three of the table above. When we do a “delete-min” we remove the value 12 and do the usual min-heap restoration process: move the last leaf into the vacated root position and “sink” that by swapping with its smallest child until it’s smaller than its children (if any). The “∞” sinks down to a leaf. However, then Dijkstra’s algorithm alters the values of two of the priorities. The result is no longer a min-heap since 18 is greater than 16.

We have shown it for a relatively small heap, but imagine a graph where a vertex $v$ has many neighbors and all of their values get reduced. The min-heap property could be violated in many places once these updates are made.

We can solve this problem by performing a heap adjustment after each value is changed. The only change that can occur in Dijkstra’s algorithm is the reduction of a value, so we only need to worry about nodes higher up in the heap, which may no longer be less than the new, reduced value. A “swim” operation lets the modified value rise up in the heap until it is once again greater than or equal to its parent but less than or equal to its children.

Your book defined a new class of priority queue called an `IndexMinPQ` that permits an operation named `decreaseKey` to make a value in the priority queue smaller. If you look at the code on the textbook web site you’ll see the call to `swim`.

**Final Comments on Dijkstra’s Algorithm**

The algorithm itself is important to understand, even if you have not seen the Java code for it. It is one of the most famous algorithms in computer science!

We note that the priority queue contains vertices; in the worst case there could be $V-1$ vertices (assume that the first vertex 0 is joined to every other vertex). Each step involves deleting the
vertex with the minimum distance from the priority queue (log $V$ operations to do a `deleteMin`) and updating neighbors’ distances (log $V$ time needed to do a “decreaseKey” operation). In the worst case we might need to update every neighbor every time, i.e., we might need to do this $E$ times. The running time of the Dijkstra’s algorithm is therefore $O(E \log V)$.

**Graph Applications**

Graph applications are everywhere. Dijkstra’s Algorithm is one of the minimum spanning tree algorithms are used by routers in the Internet for deciding the shortest way to route packets from a source to a destination and to figure out optimal distribution trees for broadcast file-sharing.

They are at the heart of search engine algorithms, are the basis for database designs, and play a role in the design of operating systems software. If you have ever tried to install software and discovered that you needed a lot of additional software you hadn’t planned on installing, you’ve come across a dependency graph and found an application for topological sort. If you’ve ever heard someone talk about a “deadlock situation,” you’ve actually been listening to someone talk about cycles in graphs.

These are just computer science applications, but graphs are relevant to chemists, biologist, economists, . . . just about every discipline that involves any attempt to model the real world must deal with graphs. (There are even books about their use in literary analysis: Franco Moretti’s *Graphs, Maps, Trees: Abstract Models for a Literary History* is in Pelletier Library.)

**Class activity**

1. Given the graph below, apply Dijkstra’s algorithm with the starting vertex to be 1, show the shortest paths to all the other vertices in the graph. Use the table form representation to show your results at every iteration.

![Input graph](figure1.png)

Figure 1: Input graph
2. Given the graph below, apply Dijkstra’s algorithm with the starting vertex to be a, show the shortest paths to all the other vertices in the graph. Use the table form representation to show your results at every iteration.

Figure 2: Input graph

Post your solution file to slack, in order to receive the points for attendance and class participation today.