Implementing (LIFO) stacks and (FIFO) queues with priority queues

As we saw on an old theory assignment, we can use priority queues to implement queues and stacks. We do this by choosing appropriate priority values for the elements we insert into the priority queue such that they’ll be removed in the order we want them to be to satisfy the LIFO property of a stack or FIFO property of a queue.

Specifically, to get a priority queue to behave like a LIFO stack, we insert elements in decreasing priority order, so that the thing most recently inserted into the priority queue has lowest priority and will be removed in a delmin operation.

Similarly, to get a priority queue to behave like a FIFO queue, we insert elements in increasing priority order, so that the thing least recently inserted will have lowest priority and be removed in a delmin operation.

Implementing breadth-first search and depth-first search with priority queues

Think back to peglab (homework 5 programming part). During a recitation, we discussed depth-first search and breadth-first search. Recall that in depth-first search we explore the search tree (or, more generally, graph) as deeply as possible first and that in breadth-first search we explore each level of the search tree (again, or graph) before exploring the next.

Note that doing depth-first or breadth-first search on a graph requires us to choose some starting point for the search. We can then define “levels” of the graph in terms of that vertex so that our definitions of depth-first and breadth-first search make sense.

Let’s work through some examples of breadth-first and depth-first search, using mazes similar to those we used in lecture. (Remember for the examples that we can implement depth-first search by pushing elements onto a stack and breadth-first search by enqueuing elements into a queue.)

However, we can implement stacks and queues with priority queues. So, if we choose priorities correctly, we can use priority queues to implement BFS and DFS.

We don’t have to assign priorities in only those two ways, though. We can use any function we want to implement graph search, with the same general procedure:

1. Pick a vertex to start at. Refer to this as $s$ (for the source).
2. Add all edges coming out of $s$ to a priority queue, assigning priorities as appropriate.
3. while the priority queue is nonempty:
   (a) Remove the minimum priority edge from the priority queue. Call this edge $e$.
   (b) If the vertex of $e$ that is farther from $s$ (call this vertex $v$) is not visited:
       i. Mark $v$ as visited
       ii. Add all edges leaving $v$ to the priority queue, assigning priorities as appropriate.

With this algorithm, we can easily implement breadth-first and depth-first traversal of a graph: simply use the idea from the homework of assigning priorities in increasing or decreasing order.
We can also assign priorities in different, more interesting ways that let us accomplish different tasks. Before we talk about these different ways of assigning priorities, it’s useful to discuss the idea of edge weights for a graph.

### Edge weights

It’s often useful in computer science to give weights to the edges of a graph. These weights often represent the cost of going down the edge. (Graphs with edge weights are often referred to as weighted graphs.)

For example, if we have a graph representing a set of cities and roads between them (where cities are vertices and roads are edges), the weights on edges could be the length of the road, the amount of toll needed to drive on the road, or the typical amount of time traveling down that road takes.

Here’s an example of a weighted graph, from [http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html](http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html). The numbers next to each edge of the graph are the weights associated with each edge.

![Example of a weighted graph](http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html)

### Dijkstra’s algorithm and the single-source shortest path problem

Now that we have defined edge weights, we can talk about some interesting problems we might want to solve with them. One such problem is called the single-source shortest path problem. The single-source shortest path problem asks us to fix a source vertex $s$ and calculate the minimum cost of going from that vertex to each other vertex in the graph.

Note that this isn’t as simple as picking an edge that goes from $s$ to another vertex if it exists. Consider, in the graph above, the shortest path from vertex 5 to vertex 4. It goes through vertex 2, having weight 9. If we just go directly from vertex 5 to vertex 4, that path has weight 58. Clearly, $9 < 58$, so sometimes it makes more sense to go through multiple edges to get the shortest path.

The basic idea behind Dijkstra’s algorithm is to perform a traversal of the input graph, but for removal from the priority queue, remove the vertex that has the path with lowest discovered weight to $s$. It’s critical to note here that this shortest discovered path can and will change as we explore more of the graph, so we need to update our priority queue interface to support updating the priority of an element.

Dijkstra’s algorithm works as follows (we’re given a graph $G$ and a starting vertex $s$):

1. Set the cost of going to $s$ to 0 and the cost of going to every other vertex to $\infty$.
2. Let $v$ be equal to $s$.
3. For each unvisited neighbor $n$ of $v$:...
(a) if the cost of going to \( v \) and then directly to \( n \) is smaller than the currently known minimum cost of getting to \( n \), update the cost of getting to \( n \) to be the cost of getting to \( v \) plus the cost of getting from there to \( n \).

4. Mark \( v \) as visited. (The distance associated with it is now final and minimal.)

5. If there are no more unvisited vertices, we’re done and can return.

6. Let \( v \) be equal to the next-smallest tentative distance and go to step 3.

While there are some differences between this algorithm and BFS/DFS (the most important ones being the capability to update priorities and the fact that we don’t immediately mark a vertex as visited when we see it), it follows the same general structure of “add things to be explored in the graph to a priority queue and remove them to search the graph.”

Note: There’s an important assumption we’re making here. Namely, that the edges have non-negative weight. Why does this algorithm not work if the edges have negative weights? (Hint: What does Dijkstra’s algorithm do if there is a cycle in the graph that has a negative total cost? What should the minimum cost there be? [Note that the algorithm does not always work if there are any negative weights, even if there is no cycle. The Bellman-Ford algorithm, which is slower, works for graphs that might have negative weights.])