Graphs and Graph Algorithms

Chapter 18[1] works through a series of increasingly sophisticated representations of a graph, which is the most generic of all linked data structures. As with all of our linked structures, the primary primitive is the **node** type, which carries all of the information about a particular entity—an airport, a street corner, a person within a larger social network—in the domain of interest. As opposed to the other linked structures we've discussed, the links within graphs are modeled more elaborately—by a complementary **arc** type, which tracks the pair of **node** endpoints and the cost one incurs by crossing it.

Here are the relevant data types, which very much imitate those used in Chapter 18.

```
struct node {
   string name;
   int x, y;
   Set<struct arc *> arcs;
};

struct arc {
   double cost;
   node *from, *to;
};

struct graph {
   Set<node *> nodes;
   Set<arc *> arcs;
   Map<string, node *> index;
};
```

Each node in the graph knows its name (e.g. **"SFO"**, **"Memorial Church"**, **"CS106X"**, etc.). For our purposes, we'll require each node track its location within a graphics window, since we'll be displaying them. And each node knows all of the ways one can travel away from it. That's what's tracked by a node's set of arcs.

Each arc knows how expensive it is, it knows what nodes it connects, and it even knows its directionality (e.g. it might encode that you can drive down a one-way street, or that you must take **"CS106A"** before you take **"CS106B"**).

The graph keeps track of all of its nodes and all its arcs. In particular, the **arcs** set within the graph is the union of all of **arcs** sets embedded within the nodes. We store redundant copies of the nodes and the edges, because some graph algorithms are best framed in terms of nodes (but need to know about the arcs that connect them), and others are oriented around the arcs.

[1] We're skipping Chapter 17, as it speaks about balancing strategies used to make sure the binary search trees are optimally structured so that our **Set**s run as quickly as possible.

Chapter 18 includes a series of introductory algorithms to traverse graphs, but they're all fairly straightforward in comparison to an algorithm—one credited to Edsgar W. Dijkstra—that identifies the least expensive way to travel from one node to a second. The algorithm is similar to the breadth-first search algorithm you used for the word ladder portion of Assignment 2, though it's careful to recognize that individual arcs cost different amounts. That means we need to rely on a priority queue to keep track of all partial paths extending from the starting point.

```
/**
 * Function: findShortestPath
 * --------------------------
 * Follows Dijkstra's algorithm to determine the shortest path
 * connecting start and finish in the surrounding graph.  We assume
 * that start and finish are different nodes.
 */
static Vector<arc *> findShortestPath(node *start, node *finish) {
   PriorityQueue<Vector<arc *>> pq;
   Map<node *, double> shortestPathLengths;
   shortestPathLengths[start] = 0.0;

   for (arc *a: start->arcs) {
      Vector<arc *> path;
      path += a;
      pq.enqueue(path, computePathCost(path));
   }

   while (!pq.isEmpty()) {
      Vector<arc *> best = pq.dequeue();
      const node *intermediate = best[best.size() - 1]->to;
      if (shortestPathLengths.containsKey(intermediate)) continue;
      if (intermediate == finish) return best;
      double cost = computePathCost(best);
      shortestPathLengths[intermediate] = cost;

      for (arc *a: intermediate->arcs) {
         if (shortestPathLengths.containsKey(a->to)) continue;
         best += a;
         pq.enqueue(best, computePathCost(best));
         best.remove(best.size() - 1);
      }
   }

   // got here? then you depleted the priority queue without finding a path
   return Vector<arc *>();
}
```

Here are some key questions about the algorithm you should be able to answer:

- Why do we need to wait for a full path from start to finish to be **`dequeue`**d from the priority queue? Can't we just identify it as a full path before we **`enqueue`** it and save some time?
- Why doesn't the final **`for`** loop include a check for cycles? Restated, why don't we confirm that **`a`** isn't already in **`best`** before we execute **`best += a`**?
- How and why does Dijskstra's Algorithm break down if negative weights are permitted?

Another canonical algorithm—one you've already seen while generating random mazes—is one that finds the minimal spanning tree in an undirected graph. A **spanning tree** is a cycle-free subset of a graph with the perk that you can still travel from any one node to another (provided you can in the original graph). A **minimal spanning tree** is a spanning tree where the sum of the arcs is as small as possible.

The following presents an implementation of Kruskal's minimal spanning tree algorithm:

```
/**
 * Function: generateMinimumSpanningTrees
 * --------------------------------------
 * Implements Kruskal's algorithm to construct a minimal
 * spanning tree within the supplied graph.
 */
static Set<arc *> generateMinimumSpanningTrees(graph& g) {

   Vector<graph> forests(g.nodes.size());
   Map<node *, int> memberships;
   int i = 0;
   for (node *n: g.nodes) {
      forests[i].nodes += n; // forest is a single node with no edges
      memberships[n] = i++;
   }

   PriorityQueue<arc *> pq;
   for (arc *a: g.arcs) {
      pq.enqueue(a, a->cost); // queue edges up, lightest to heaviest
   }

   Set<arc *> mst;
   while (!pq.isEmpty()) {
      arc *lightest = pq.dequeue();
      if (memberships[lightest->from] == memberships[lightest->to]) {
         cout << "Skipping edge (cost: " << lightest->cost << ") that "
              << "would introduce a cycle." << endl;
      } else {
         cout << "Adding edge (cost: " << lightest->cost << ") to merge "
              << "two forests." << endl;
         mst.add(lightest);
         int from = memberships[lightest->from];
         int to = memberships[lightest->to];
         forests[from].nodes += forests[to].nodes;
         forests[from].arcs += forests[to].arcs;
         forests[from].arcs += lightest;
         for (node *n: forests[to].nodes) {
            memberships[n] = from;
         }
      }
   }

   return mst;
}
```

The first section creates a collection of n **`forests`**, where n is the number of nodes in the graph. Each forest consists of a single node isolated from all others (much as all chambers were initially isolated from one another just as the maze-generator executable was starting

out). The **memberships** map is a convenience data structure that allows us to quickly determine which forest each node is part of; each node maps to the index of the graph within **forests** that it's a part of.

The second section builds a priority queue of arcs, where all arcs in the graph are inserted and effectively sorted so that low cost arcs are extracted before high cost ones. Kruskal's algorithm considers the graph's arcs in that order.

The third section continually extracts arcs and determines if its **to** and **from** nodes reside in different forests. If not, the arc is discarded, since including it would introduce a cycle into the ever-growing MST, and trees are by definition forbidden from having cycles. If the two nodes bridged by the arc are in different forests, we add the arc to the set of arcs we ultimately return as the MST, and then take care to transfer all trees in the second forest to the first (leaving the second forest empty). When the while loop ends, the singleton forests have been merged into the smallest number of forests possible. Each of these forests is a minimal spanning tree.