CMPSCI 187: Programming With Data Structures

Lecture #32: Searching Graphs
David Mix Barrington
28 November 2012

Searching Graphs

- Review Graphs and Implementations
- The `isPath` Problem
- Using a Stack: Depth-First Search
- Using a Queue: Breadth-First Search
- The Single-Source Shortest-Path Problem
- Finding Shortest Paths With a Priority Queue

Review Graphs and Implementations

- Recall that a **graph** is a set of **nodes** (also called **vertices**) and a set of **edges**. In an **undirected graph**, an edge is *between* two vertices without a direction. In a **directed graph**, an edge is *from* one vertex *to* another. We can also annotate edges with **weights** (often representing distance or cost of the edge) or other data.

- A **path** in a graph is a sequence of edges where the destination of each edge in the sequence is the source of the next edge. The weight of a path is the sum of the weights of its edges. A **simple path** is one that does not reuse any vertex. If any path exists from vertex u to vertex v, so does a simple path.

- DJW give a `WeightedGraphInterface` with methods to add vertices, add edges, mark vertices, and get a queue of the neighbors of any vertex. Last time we saw the `WeightedGraph<T>` class, whose objects are graphs with nodes from `T` and positive integer weights on the edges. It is an array-based implementation of the interface, using a 1-d array for the vertices and a 2-d array for the edges.

Implementing Graphs With Linked Lists

- The array-based implementation, called an **adjacency matrix**, is the simplest way to represent graphs on a computer. But on a graph with n vertices, it always takes O(n^2) memory locations, one for each entry in the edge array.

- Graphs in applications are usually **sparse**, meaning that they have relatively few neighbors for each vertex. A sparse graph of n vertices might have only O(n) edges, and its matrix would be mostly null edges. Another representation called an **adjacency list** saves space, and can save time if the algorithms are designed to use it efficiently.

- An adjacency list has an array of linked lists of vertices, one for each vertex. The list for vertex v has all of v's neighbors, so there is only one entry for each edge of a directed graph, or two for each edge of an undirected graph. We can implement the methods of the weighted graph interface fairly easily. For example, the queue of neighbors just takes its entries from the list for the given vertex.

The `isPath` Problem

- The simplest question about paths in a graph is whether one exists with a give source and destination. We'll define a boolean method `isPath` that takes two vertices as parameters and answers this question.

- Once we can answer this, we know how to approach more complicated questions, like whether the whole set of vertices is **connected** (there is a path from any vertex to any other) or how to list the vertices in the same **connected component** as a given vertex.

- We'll see three approaches to this problem. Each works in the same way -- it keeps a set of reachable vertices in some data structure. It advances its search by taking a vertex out of the structure, then putting any unseen neighbors it might have into the structure, until the goal node comes out of the structure or the structure becomes empty with no goal node found.

- What data structure we use determines the type of search.

Using a Stack: Depth-First Search

- It’s important that we mark vertices on their way into the stack, so we check only simple paths. This method only tests for a path but we could return the path too.

```
private static boolean isPath (WGI<String> graph, String
                               startVertex, String endVertex) {
   UnboundedStackInterface<String> stack = new LS<String>( );
   UnboundedQueueInterface<String> vertexQueue = new LUQ<String>( );
   boolean found = false; String vertex, item;
   graph.clearMarks( );
   stack.push(startVertex);
   do {vertex = stack.top( ); stack.pop( );
      if (vertex = endVertex) found = true;
      else if (!graph.isMarked(vertex)) {
         graph.markVertex(vertex);
         vertexQueue = graph.getToVertices(vertex);
         while (!vertexQueue.isEmpty( )) {
            item = vertexQueue.dequeue( );
            if (!graph.isMarked(item)) stack.push(item);}}
   while (!stack.isEmpty( ) && !found);
   return found;}
```

Using a Queue: Breadth-First Search

- The code here is almost identical with a queue replacing the stack.

```
private static boolean isPath2 (WGI<String> graph,
                         String startVertex, String endVertex) {
   UQI<String> queue = new LinkedUnbndQueue<String>( );
   UQI<String> vertexQueue = new LinkedUnbndQueue<String>( );
   boolean found; String vertex, element;
   graph.clearMarks( );
   queue.enqueue(startVertex);
   do {vertex = queue.dequeue( );
      if (vertex = endVertex) found = true;
      else if (!graph.isMarked(vertex)) {
         graph.markVertex(vertex);
         vertexQueue = graph.getToVertices(vertex);
         while (!vertexQueue.isEmpty( )) {
            element = vertexQueue.dequeue( );
            if (!graph.isMarked(element)) queue.enqueue(element);}}
   while (!queue.isEmpty( ) && !found);
   return found;}
```

Comparing DFS and BFS

- ~~Both isPath methods will examine most or all of the edges in the graph in the~~ worst case. If we use an array-based representation for the graph, with n vertices and e edges, we will spend O(n^2) time making the vertex queues, but only O(1) time per edge otherwise. The list-based representation will use only O(e) total time overall, which is faster if the graph is sparse.

- BFS has the advantage that it will always find a path with the fewest number of edges. This is because, as we saw in Project #4, it puts all vertices one step from the start onto the queue, then all vertices two steps away, then all vertices three steps away, and so on until it finds the goal. DFS might find a path that is not the shortest. But if the shortest path is long, BFS could spend a lot of time looking at all the shorter paths, while DFS might hit upon the correct path quickly.

- BFS will use more memory in many cases, as there are typically more than r nodes at distance r from the start.

- Obligatory XKCD reference: `http://xkcd.com/761`

The Single-Source Shortest-Paths Problem

- The path found by BFS has the smallest number of steps, but if the steps have different weights it might not be the **shortest path** in terms of total weight. A GPS system, for example, would want to return the best path according to some measure of cost, which might involve distance, travel time, or other factors.

- The **single-source shortest-path** problem is to take a start vertex x and determine the smallest-weight path from x to *each other reachable vertex* in the graph. It turns out that the simplest way to search for the best path from x to y also finds the best path from x to every vertex in the graph that is closer to x than y is.

- The algorithm is very easy -- just search as we did for DFS and BFS, but put the vertices into a **priority queue** where the *shortest distance from x* is the criterion for the best vertex to dequeue.

Finding Shortest Paths With a Priority Queue

- This method makes a priority queue of `Flight` objects, which contain a start vertex, an end vertex, and a total distance. The `compareTo` method of the `Flight` class treats objects with smaller distances as “larger”, so that they will be dequeued from the priority queue first.

- It takes a bit of reasoning to show this correct, which you’ll see in CMPSCI 250.

```
private static void shortestPaths(WGI<String> graph,
                                  String startVertex) {
   Flight flight, saveFlight;
   int minDistance, newDistance;
   PQInterface<Flight> pq = new Heap<Flight>(20);
   String vertex;
   UQI<String> vertexQueue = new LinkedUnbndQueue<String>( );
   graph.clearMarks( );
   saveFlight = new Flight(startVertex, startVertex, 0);
   pq.enqueue(saveFlight);
   // main loop on next slide
```

Main Loop of `shortestPaths` Method

- We print out the `Flight` object for the shortest path to each vertex as found.

```
do {flight = pq.dequeue( );
      if (!graph.isMarked(flight.getToVertex( ))) {
         graph.markVertex(flight.getToVertex( ));
         System.out.println(flight);
         flight.getFromVertex(flight.getToVertex( ));
         minDistance = flight.getDistance( );
         vertexQueue = graph.getToVertices(flight.getFromVertex());
         while (!vertexQueue.isEmpty( )) {
            vertex = vertexQueue.dequeue( );
            if (!graph.isMarked(vertex)) {
               newDistance = minDistance +
                  graph.weightIs(flight.getFromVertex( ), vertex);
               saveFlight = newFlight(flight.getFromVertex( ),
                                       vertex, newDistance);
               pq.enqueue(saveFlight);}}}}
while (!pq.isEmpty( ));}
```