Com S 228
Fall 2010
Programming Assignment 5
Due at 11:59 pm Friday, December 10
200 Points

Note: there is no "late deadline" for this assignment. All submissions must be received by Friday night at the end of dead week!

This assignment is to be done on your own. If you need help, see me or one of the TAs. Please make sure you understand the "Academic dishonesty" section of the syllabus.

1. Introduction

Graph search algorithms provide wonderful opportunities to apply everything we have learned about data structures and their performance implications. We will be implementing several graph search algorithms. There is a viewer application provided for you that will help to visualize and test the different algorithms. (To start the viewer, run the Main class in the viewer package. It won't do anything, however, until you finish item 1 below.)

Summary of tasks

1. Create a class ListGraphImpl that implements the ListGraph interface using adjacency lists.

   This is not hard; there are some implementation suggestions below. Once this is done you will be able to create graphs in the viewer, and you can use the Step and Run buttons for BFS.

2. Implement the depth-first search algorithm, Dijkstra's algorithm, and the A* algorithm. You do this by designing and creating one or more classes that implement the GraphAnimation interface.

   The design of these classes is up to you, although you will probably want to use the sample class BFSFinder as a starting point.

3. Modify the PathFinderFactoryImpl class, and ensure that appropriately initialized instances of your class or classes from Task 3 are returned by the createBFSFinder() and createDFSFinder() methods.

At this point you'll be able to watch your algorithms run using the GraphAnimationViewer.

4. Implement a class for bidirectional path-finding. Modify the PathFinderFactoryImpl appropriately.

Further details can be found in Section 3, "More task details," below.

You will also need to verify that your code works correctly for large implicit graphs. You will be provided with an implementation of a graph for the *n*-puzzle on which you can try your A* algorithm. (The simplest version, the 8-puzzle, has 181440 nodes. The more familiar 15-puzzle has over 10 trillion nodes.) The n-puzzle code will be provided separately. You do not have to do anything with this code except try it.

Performance requirements

Your implementations in (2) must be efficient in use of time and space, that is, as close as possible to the theoretical bounds of $O(n + m)$ space, $O(m)$ time for BFS/DFS, and $O((n + m) \log n)$ time for Dijkstra's and the A* algorithm. (Here n is the number of vertices and m is the number of edges.) The key is to use appropriate data structures from the Java libraries. In particular, you will most likely use HashSet for the closed set, HashMap for recording distances and predecessors, and PriorityQueue for the open set used in Dijkstra's and A*. We are assuming that basic operations on HashMap and HashSet are constant-time.

Getting started

You will need the project file cs228hw5.zip. The project includes the Graph, GraphAnimation, BidirectionalGraphAnimation and PathFinderFactory interfaces. Source code is provided for a sample BFS implementation (see BFSFinder), and source code is provided for the GraphAnimationViewer demonstrated in class (which you should not need to read or modify). You can start the viewer using the Main class but none of the controls will work until you implement ListGraphImpl.

A good strategy would be to start by implementing ListGraphImpl, which is relatively straightforward. Then you can run the BFSFinder in the viewer. Then try modifying it so that it performs a depth-first search instead of a breadth-first search. Then update the PathFinderFactoryImpl so that the createDFSFinder method returns an appropriate instance; at this point, the DFS selection in the viewer's drop-down menu should work. Next you'll want to be sure you understand Dijkstra's algorithm. Try working out some examples by hand. Then implement it, and update the PathFinderFactoryImpl accordingly. Try the A* variation. Then look at opportunities for code reuse and decide how to organize your code.

You can implement the bidirectional search class at any point, since it should depend only on the GraphAnimation interface.

Viewer options

You can start the viewer using the Main class in the viewer package. It won't work until you implement ListGraphImpl. Assuming you have done so, try the following in the viewer:

1. Click the "Create graph" button
2. Number of points 20
3. Max edge length 50
4. Density 30
5. UNcheck "Use goal node"
6. Click OK
7. Select BFS from the drop down box
8. Use the Step button to execute the algorithm one step at a time.

Nodes added to the open set are shown in red (along with their predecessor edges), and nodes added to the closed set are shown in black. There is a light blue circle drawn around the last node added to the closed set. To start over with the same graph, click Reset. To animate the execution of the algorithm, use the Run button instead of the Step button.

If a goal node is selected, the viewer should highlight the path from the start to the goal.

The graph is generated randomly according to some user-selectable parameters: the number of vertices, the maximum edge length within the 100 by 100 grid, and the "density" of edges. This number is essentially the proportion of all possible edges, i.e., the probability that a given edge will be included in the graph. If a density of 100% is selected, you'll get a complete graph. The graph is not guaranteed to be connected.

If, for testing purposes, you want to reproduce an identical graph later on, click "Create graph", select the radio button for "Enter seed", and copy the seed to a text file along with the parameters you used. If you run the viewer again with the same parameters and the same seed, it will generate the same graph.

2. Search algorithm overview

The breadth-first (BFS) and depth-first (DFS) search or traversal algorithms have the following form. The "closed set" consists of vertices that do not need to be processed any further, and the "open set" consists of vertices which are neighbors of those in the closed set and which still need to be processed. For BFS the open set is processed in FIFO order, and for DFS the open set is processed in LIFO order. (DFS has many variations; see the last section for some further remarks on this.)

```
add start vertex to open set
while open set is not empty
    remove a vertex u from open set
    add u to closed set
    if u is the goal vertex
        done = true
        break
    for each neighbor v of u
        if v is not in open set nor in closed set (*)
            add v to open set
            record u as the predecessor of v
done = true
```

Figure 1: Basic pseudocode for graph search and traversal algorithms

The idea of a recording the "predecessor" is that any traversal of a graph implicitly finds a unique path to each node. This is useful information in itself (because in many algorithms the point is to find an optimal path) and is also useful for understanding and visualizing the progress of the algorithm. For example, given the graph below in Figure 2, the diagram at the right shows an edge only for the predecessors recorded by the BFS algorithm (using A as the start node). Since the result is always a tree, the resulting edges are sometimes called "tree edges".

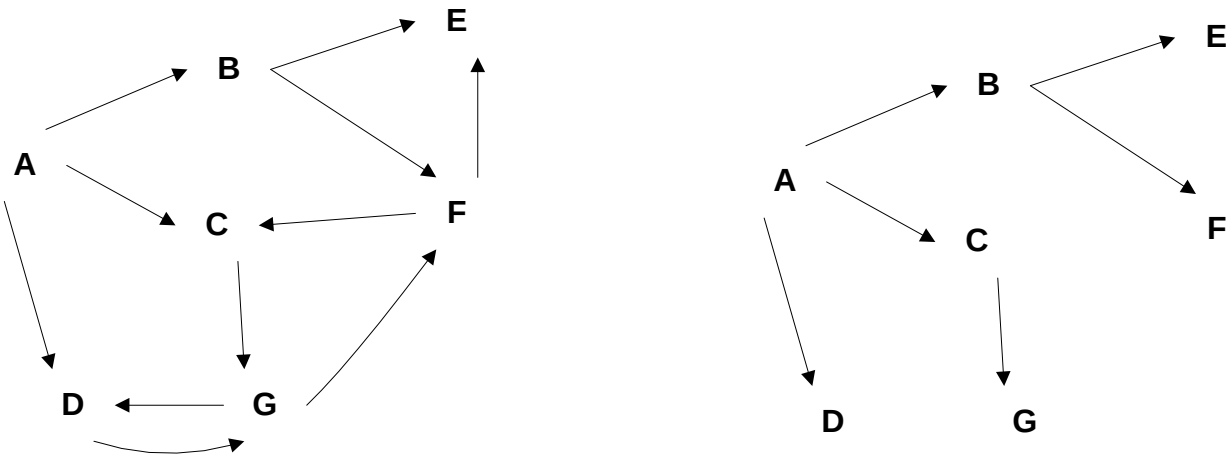


Figure 2 - a sample graph, and the tree edges represented by the predecessors

One thing to point out is that with a normal implementation of a queue or stack, it is an O(n) operation to check whether a the structure contains a given vertex v, as at (*) in the pseudocode above. However, since we are keeping track of predecessors anyway, an efficient way to detect whether a vertex is in the open set or closed set is simply by checking whether it has a predecessor (and is not the start vertex). So in practice, (*) would be implemented as follows:

```
if v is not the start vertex and v does not have a predecessor
```

In the case of Dijkstra's algorithm and the closely related A* algorithm, the algorithm structure is similar to that shown in the box above, but the elements in the open set are prioritized according to a "score". For Dijkstra's algorithm, the score for a vertex v is always equal to dist(v), where dist(v) is the length of a shortest (minimum-weight) path from start to v that contains only vertices in the closed set. For the A* algorithm, the score consists of dist(v) plus the value of h(v), which is an "estimate" of the length of a shortest path from v to the goal. (See the discussion of the PointGraph class for more explanation about the "h" function.) For Dijkstra's algorithm, a value of zero is always used for h(v). See the pseudocode below. (We assume that a comparison such as alt < dist(v) is logically true if dist(v) is undefined.)

Pseudocode for Dijkstra's algorithm and A* algorithm

```
add start vertex to open set
let dist(s) = 0
dist(v) is undefined for all v other than s
pred(v) is undefined for all v
while open set is not empty
    remove a vertex u from open set that has minimum score
    if u is not in closed set  // checks that entry for u is not redundant
        add u to closed set
        if u is the goal vertex
            done = true
            break
        for each neighbor v of u
            if v is not in closed set
                let alt = dist(u) + weight of edge (u, v)
                if v is not in open set or alt < dist(v)
                    // could be a duplicate entry
                    add v to open set with score alt + h(v)
                    let dist(v) = alt
                    let pred(v) = u
done = true
```

Figure 3: Pseudocode for Dijkstra's algorithm and A* algorithm

3. GraphAnimation overview

The idea of the GraphAnimation interface is to implement graph search algorithms in a manner that makes it easy for an animation tool (such as our GraphAnimationViewer) to execute an algorithm one step at a time and periodically query the state of the internal data. The key method is

```
public E step()
```

which executes one step of a search algorithm and returns the element that was added to the closed set, if any. One "step" of the algorithm consists of a single execution of the while loop

body. In our implementations, the viewer runs the outer "while" loop, and each step will be initiated from the viewer. Essentially the viewer runs a loop of the form

```
while not done()
    step()
```

We vary the algorithm by implementing the step() method. See the sample BFSFinder code for an example.

To see how this style of implementation might be used in general (that is, outside the viewer), see the example below. The code shows how you might use the GraphAnimation interface to execute a breadth-first search on a PointGraph. This is essentially what is going on inside the viewer. (Note this also illustrates the use of the PathFinderFactory interface, described in detail below.)

Example: using the GraphAnimation interface to run a breadth-first search

```
PathFinderFactory factory = new PathFinderFactoryImpl();
PointGraph g = new PointGraph();
Point start = new Point(0,0);
Point v = new Point(100, 100);
g.addVertex(start);
g.addVertex(v);
// ... add other vertices and edges ...

// creation performs algorithm initialization
GraphAnimation<Point> bfsSearcher =
    factory.createBFSFinder(g, start, null);

// main algorithm loop
while (!bfsSearcher.done())
{
  Point vertex = bfsSearcher.step();
  if (vertex != null)
  {
    System.out.println(vertex.toString() + " added to closed set");
  }
}

// can query state during and after completion
List<Point> path = bfsSearcher.getPath(v);
if (path.size() == 0)
{
  System.out.println(v.toString() + " is not reachable");
}
else
{
  System.out.println("Path: " + path.toString());
}
```

3. More task details

Please refer also to the javadoc for the Graph, GraphAnimator, BidirectionalGraphAnimator, and PathFinderFactory interfaces, located in the api package.

Task 1: Creating the class ListGraph

The Graph interface is a very simple and general abstraction of the idea of a graph as a set of relationships between objects. It has just two methods that are required for graph search algorithms, a method getNeighbors(u) that returns an Iterator over the vertices adjacent to a given vertex, and a method called "h" that is used by the A* algorithm. (In your ListGraph implementation, h(u,v) should always return zero.) The interface also specifies a static inner class Edge used to represent adjacent vertices. That is, an edge from u to v is described by the adjacent vertex v and its weight.

ListGraphImpl should be a concrete implementation of ListGraph based on adjacency lists. There is a skeleton in the package edu.iastate.cs228.hw5. You can use any appropriate data structures from java.util.*, so this should not be too hard. One simple implementation is to start with:

```
private Map<E, Set<Edge>> edgeSets = new HashMap<E, Set<Edge>>();
```

That is, each vertex of type E is associated with a Set of edges representing the adjacent vertices.

Task 2: Implementations of the GraphAnimation interface for DFS, Dijkstra's algorithm and A*

It is up to you to design the class or classes that contain your algorithm implementations. The four algorithms are similar in basic structure, and all implementations of GraphAnimation have certain routine methods in common (e.g. returning an iterator over the closed set). Code reuse encouraged, but you are not absolutely required to eliminate all duplicate code. *The most important things are clarity and correctness.* You are welcome to reuse the BFSFinder code as you see fit.

One possibility is to start with an abstract class containing common code, and then implement two subclasses, one for the BFS/DFS implementations (which are really almost identical) and one for Dijkstra and A* (which are also almost identical).

Another possibility is to start with one class that has an abstract method to represent the "strategy" for handling the open set and updating distances and predecessors – then create four concrete classes, each encapsulating the details of a particular algorithm.

It is also acceptable to implement all four algorithms in one class, as long as your code is clear and readable and you have eliminated most of the obviously duplicated code.

Task 3: Implementation of the PathFinderFactoryImpl class

The viewer will use the classes you write to provide a visual display of the execution of each algorithm on a PointGraph. Most of the code in the viewer is written to the GraphAnimation and BidirectionalGraphAnimation interfaces and is not dependent on any particular implementation. At some point, however, the viewer needs to be able to construct concrete instances of the classes you write that implement the various search algorithms. But since the author of the viewer does not know what class or classes you will write, or what constructors they will have, you might well wonder how it is possible for the viewer to instantiate them.

A partial solution to this problem is to use a *factory*. A factory is a class whose only purpose is to produce a set of objects satisfying a particular interface. The viewer doesn't need to know anything about the concrete types you use, as long as you can provide a factory object that will produce the appropriate objects on demand.

The PathFinderFactory interface describes four methods, each of which produces an object with particular behavior. In all cases the objects produced implement the GraphAnimation interface. For example, when you have selected "BFS" from the drop-down list in the viewer, and you click "Step" or "Run", the viewer will invoke the statement

```
graphAnimator = factory.createBFSFinder(graph, start, goal);
```

to create the appropriate GraphAnimation instance. See the PathFinderFactory javadoc for more detail.

Task 4: Implementation of a bidirectional path finder

Some applications of graph search algorithms have to visit every node in the graph. For example, the first phase in garbage collection is to find and mark all objects that can be reached from outside the memory heap. Other applications are just interested in finding a path, or a shortest path, from the start node to a particular "goal" node. Using Google Maps to find driving directions is a common example. When a specific goal node is known, a bidirectional search can significantly reduce the number of edges that have to be examined in finding a path from the start node to the goal node.

The idea of a bidirectional search is simple: simultaneously run two search algorithms, one that searches "forward", starting at the start node, and one that searches in "reverse", starting from the goal node. For simplicity, we assume that you will always strictly alternate: one step of the

forward algorithm, then one step of the reverse algorithm, and so on. Each time you add a node to the closed set of one algorithm, check whether it is already in the closed set of the other algorithm. If so, then you can construct a path from the start to the goal by putting together the two paths to the common node. (As a technical aside, note this strategy will not always produce a *shortest* path unless a slightly more sophisticated termination condition is added, a complication that we will not attempt to address in this assignment.)

Based on the description above, you can implement a general bidirectional path finder that implements the interface BidirectionalGraphAnimation. It should encapsulate two instances of an existing search algorithm (that is, two GraphAnimation instances), one for the forward search and one for the reverse search. The same implementation should work for any existing search algorithm; that is, you should probably have a constructor that takes two existing instances of GraphAnimation as its arguments.

Implementation of the bidirectional search class should be relatively simple because most of the existing methods of GraphAnimation, such as getPredecessor() or openSet(), are simply delegated to the forward search instance, and the additional methods of BidirectionalGraphAnimation, such as getPredecessorReverse() or openSetReverse(), are delegated to the reverse search instance. The main difference is in the step() method. The step() method for BidirectionalGraphAnimation should *alternate* calls to the step() methods of the forward search and reverse search algorithms, and should detect completion. The getCompletePath() method should return the complete path from the start node to the goal node as indicated in the javadoc.

4. Note on the PointGraph class and the “h” function

The class edu.iastate.cs228.hw5.viewer.PointGraph is a ListGraph of java.awt.Point objects that is used by the viewer, so the viewer will not be able to do anything at all until you implement ListGraphImpl. The viewer will create edges whose weight is based on the straight-line distance between points in the x-y plane (in order to use integer weights, the distance is rounded and multiplied by 100). The only actual work done is overriding the h function. In fact, here is the entire code for PointGraph:

```
import java.awt.Point;
public class PointGraph extends ListGraph<Point>
{
  @Override
  public int h(Point p, Point q)
  {
    double distance = p.distance(q);
    return (int) Math.round(distance * 100);
  }
}
```

The idea of the function "h" is that it provides a hint, or estimate, to the A* algorithm about the length of the shortest path between two vertices. In practice, the second vertex is always the one designated as the "goal". One easy way to estimate the length of a path between points u and v in the x-y plane is with the straight-line distance. (For technical reasons, h should always provide an *underestimate* of (a value less than or equal to) the actual shortest path length. Note that the straight-line distance has this property.)

5. Special documentation requirement

The javadoc for your factory implementation *must* include a brief overview of your design choices for the class or classes for the GraphAnimation implementations.

6. Turning In

Place all the Java source files in a zip archive (do not use tar or rar). Remember to save the directory structure in the zip file. Do **not** turn in your class files.

7. Additional note on depth-first search (*optional reading*)

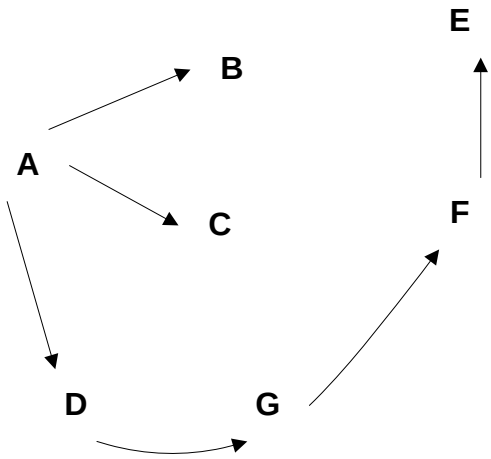
A depth-first traversal of a graph can be performed recursively as we did for trees. However, for a tree there is a unique path to each node from the root, but for an arbitrary graph, we have to keep track of nodes already visited since each node may be reached by multiple paths. With recursion, we don't need an open set, since the call stack implicitly provides the ability to backtrack to unprocessed nodes. A typical recursive implementation has the form:

```
void dfs(u)
{
  add u to closed set
  for each neighbor v of u
    if v is not in closed set
      record u as predecessor of v
      dfs(v)
}
```

The ordering of nodes (that is, the order in which nodes are added to the closed set) is different for the recursive algorithm above than for the iterative version that we are using in this assignment. For example consider the graph of Figure 2. (Assume that both algorithms encounter the same ordering for the neighbors when executing "`for each neighbor v of u`", that is, iterating over the neighbors of A yields B, C, and D, and iterating over the neighbors of B yields E and then F). Then the iterative algorithm from Figure 1, using A as the start node, gives the ordering:

A D G F E C B

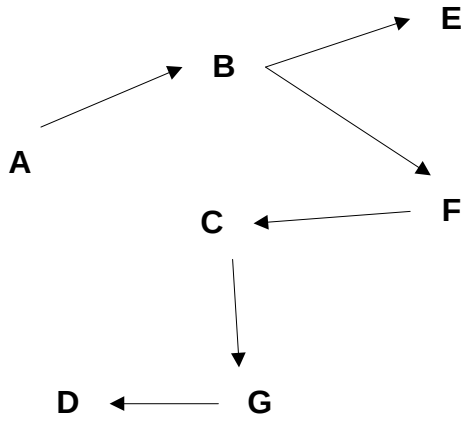
where the tree edges are shown at right.



The recursive version would give

A B E F C G D

assuming that the initial call is dfs(A) . The tree edges would be as shown at right.



It can be argued that the recursive version is a “real” depth-first traversal while the first version is a kind of hybrid, since the immediate neighbors of a node are always saved on a stack for later. The homework uses the first version, however.