University of Toronto
CSC148—Introduction to Computer Science, St. George Campus, Fall 2007

Assignment 3: Graph Search

Due: 10:10 a.m. (at the **beginning** *of class), Monday, Nov. 19*
This assignment is worth 10 percent of your grade in this course.

Introduction

Directed graphs are an abstract data structure which generalize linked lists. In particular, within a directed graph each vertex can have any number of out-going edges (see Figure 1 below). In this assignment you will study reachability and shortest path computations within directed graphs.

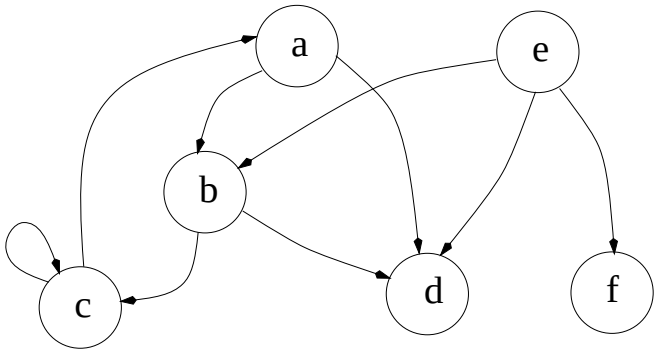

Figure 1: A simple directed graph having vertices labeled "a" through "f". Circles represent the vertices of the graph, each of which are labelled with their own data items (the characters "a" through "f"). Solid arrows denote edges in the graph. Note that vertices "e" and "f" are not reachable from vertex "a". But there are many possible routes to take from "a" to "d", including the scenic route of "a" to "b" to "c", then to "c", and next to "a", then "b", and finally "d". A shortest, or minimal path, from "a" to "d" is to take the single edge from "a" to "d".

Definitions

A (directed) graph $G = (V, E)$ consists of a set of vertices V and edges E. Here E is the set of pairs (v_i, v_j) with $v_i, v_j \in V$, where (v_i, v_j) denotes an edge from v_i to v_j. If (v_i, v_j) is an edge in the graph, then we say v_j is a (forward) *neighbour* of v_i and, in reverse, v_i is a *previous neighbour* of v_j.

A *path* in G is a sequence of vertices $\{v_1, v_2, ..., v_n\}$ such that there is an edge from v_i to v_{i+1} for each $i \in \{1, 2, \ldots, n-1\}$.

The *length* of a path is the number of nodes in it.

A vertex w is *reachable* from a vertex v if and only if there is a path from v to w. In particular, with this definition, any vertex is reachable from itself by a path of length 1.

Let P be a path in G from v to w. Then P is a *shortest path* from v to w if and only if, for any other path Q in G from v to w, the length of P is less than or equal to the length of Q.

```
MoveOracle mo = new GraphOracle(); // Creates a description of the graph in Fig. 1
mo.getMoves("a")  // Note the double quotes, the arg is a String.
  // Returns "[b, d]" the two neighbours of "a" in Fig. 1
mo.getMoves("c")
  // Returns "[c, a]" the two neighbours of "c" in Fig. 1
Handle h = new GraphHandle("d");
h.matches("d");  // Returns true
h.matches("a");  // Returns false
Reachable r = new Reachable(mo); // We consider the graph defined by mo.
r.setStart("a"); // Set the vertex to start at.
r.setGoal(h); // Sets the goal state(s) to be those for which
              // h.matches( vertex's data ) is true.
// Your goal is to revise Reachable so that it searches for
// a path from the start vertex to some goal vertex.  For example:
r.reachable()  // Will return true to indicate "d" is reachable
               // from "a" (but currently throws an exception).
r.minimalPath()  // Will return the path "[a d]"
// Now seek a path from e to a
r.setStart("e"); // Set the vertex to start at.
h = new GraphHandle("a");
r.setGoal(h); // Set the goal handle.
r.reachable()  // Will return true to indicate "a" is reachable
               // from "e" (but currently throws an exception).
r.minimalPath()  // Will return the path "[e b c a]"
```

Figure 2: Interacting with the handout code in the Graph.pjt project. Each line followed by the comment "// Will..." describes what your solution will do, not what the handout code does.

Getting Started

In the handout code we have provided Java class and interface definitions for much of what you will need to solve reachability and shortest path search problems. Moreover, in the following we walk you through the major remaining steps involved in solving these problems. As in the previous assignment we have some vital rules about which code you can change. You must obey these or you will get a mark of zero.

Create a new directory, say `A3`, and download the starter code `A3handout.zip` from the CSC 148 web site. Unzip this starter code.

You should now be able to load and compile the following files:

- **Interfaces:** `Handle.java, MoveOracle.java, ReachableSet.java`
- **Classes:** `ArrayListReachableSet.java, GraphHandle.java, GraphOracle.java, MultiNode.java, Reachable.java`

There is no main method, but you can use DrJava's interaction pane as described in Figure 2.

Vital Rules. The only files in this handout code that you can change are `ArrayListReachableSet.java`, `Reachable.java`, and `Board.java`. Moreover, you can only change certain parts of these files. Search for "YOU" to see the places you are allowed to change. Do not change any other parts of these files besides the

```
BFS(v, h)   //  v is the source vertex where the search starts
            //  and h is the handle used to recognize a goal vertex.
    // Initialize the search.
    // Start with an empty reachable set and an empty  queue
    ReachableSet reachSet = new ArrayListReachableSet()
    Queue q = new FleXQueue()

    //Initially the source vertex is at distance 0, and has no known
    //previous neighbours.  We denote this by (v,0, null).
    put (v,0,null) into the reachable set.
    if v is a goal state, i.e. h.matches(v.getData()) is true,
      return // We have found a goal state.
    end if
    put (v,0,null) into the q.

    // Perform the search.
    while ( q is not empty ) do
        Set (u,d,p) to be the item returned by q.dequeue()
        for each (forward) neighbour w of u do
            if ( (w,*,*) is not in the reachable set for any *'s) do
                put (w,d+1,u) into the reachable set
                put (w,d+1,u) into q
            end if
        end for
    end while
```

Figure 3: The breadth first search (or BFS) algorithm for determining reachability and shortest distances within a graph.

ones described by these comments. You can add private helper methods to `Reachable.java`, but not any instance or static variables, nor any non-private methods. Do not use recursion.

Reachable Set

Step 1. A partial implementation of the interface `ReachableSet` is provided in `ArrayListReachableSet.java`. Complete the methods `get(MultiNode m)` and `find(Handle h)` in this class according to their method comments.

Vital Rules. See above.

Breadth First Search

Step 2. Your second programming task is complete the method `breadthFirstSearch()` in `Reachable.java`. A sketch of the algorithm is given in Figure 3. The algorithm uses `MultiNode` objects which we summarize here with triples of the form (see `MultiNode.java` in the handout code):

```
(vertex, distance, previous)
```

where distance is the distance of the vertex from the source, and previous is a partial list of previous neighbours of this vertex. You can decide how many previous neighbours to keep in this list (it will effect how easy it is for you to find the minimal path in Step 3 below).

In short, BFS works as follows. As long as the queue in this BFS algorithm is not empty, we get the next `MultiNode` off of the queue and check whether each of its vertex's neighbours are in the reachable set. If they aren't, we add them to the reachable set, along with a record of their distance from the source, and with a link referring back from this neighbour to the previous vertex. We also add these same, newly reached neighbours to the end of the queue for later processing.

If this queue becomes empty, we claim that all vertices that are reachable from v are in the reachable set. Moreover, the distances recorded in the `MultiNodes` in the reachable set are the *minimal* distances from the source to these vertices. Do you believe us? Why? Can you prove it? (Again, these are thought questions; don't hand in answers.)

For the queue needed by BFS, use a `FleXQueue` (the implementation is provided in the `adt` directory).

Vital Rules. Same as above.

Test your completed `breadthFirstSearch()` in DrJava's interaction pane by following the examples in Fig. 2. The method r.reachable() should now work, returning true or false according to whether a goal vertex is reachable from the start vertex. (But it does not yet give you the minimal path.) This need not be a thorough test (you do not need to hand it in).

Shortest Paths

Step 3. As defined above, the length of a path is just the number of nodes in it. Given that a vertex `w` is reachable from `v`, we would like to know a path from `v` to `w` which has the minimum possible length.

Complete the method `minimalPath()` in `Reachable.java` so that it fulfills the contract in the method comment. After you have run reachable(), the instance variable `reachableSet` in class `Reachable` contains all the information you need to determine a minimal path.

Vital Rules. Same as above.

Test your completed `Reachable` class with the example in Fig. 2. This need not be a thorough test (you do not need to hand it in).

Big-Oh

Step 4. Given a graph with n nodes and $O(n^2)$ edges, what is the worst case runtime of the BFS algorithm you implemented in Step 2 above? In your written report, you should explain in detail the steps in your derivation of this runtime order. Follow a similar style argument to the one we used in the lecture notes, working from the inside of the algorithm out.

The analysis of the BFS algorithm is a little tricky in that you need to figure out how many times the while loop (see Fig. 3) is executed. **Hint:** It is easiest if you first consider how many times (at most) any single graph node will appear as the variable `u` in an item popped off of the queue. Similarly, how many times (at most) is any single forward edge considered in the subsequent for loop? The answers to both these questions

allows you to get a simple expression for the worst case big-oh runtime of this algorithm.

Also, assume each individual `FleXQueue` method call (e.g., `q.enqueue(o)`, `q.dequeue()`, etc.) runs in $O(1)$ time. This is not exactly true, due to the need to change the capacity of the queue from time to time. But we saw in two previous labs that `FlexQueue` requires only $O(n)$ time to enqueue n items and dequeue them all again. So, **on average**, it only takes $O(1)$ time to enqueue or dequeue a single item. So this assumption is true in the average case.

You can make a similar assumption about the `ArrayList` used in `ArrayListReachableSet`. Namely, the `add(o)` method takes $O(1)$ time (on average). However, note that the `contains(o)` method will have a worst case runtime equal to the size of the `ArrayList`.

In your report identify the single line in Fig. 3 in which the algorithm will spend (by far) the most time when n becomes large. Explain.

Finally, do a similar worst case runtime analysis for your `minimalPath()` algorithm, assuming that `reachable()` has already been run to set up the reachable set.

A-Mazing Fun

Step 5. In DrJava (or whatever you use), open:

- **Interfaces:** `Handle.java, MoveOracle.java, ReachableSet.java`
- **Classes:** `ArrayListReachableSet.java, MazeHandle.java, MazeOracle.java, MultiNode.java, Reachable.java`

Here `ArrayListReachableSet.java` and `Reachable.java` are your files completed in the previous steps. The other files are in the handout code. Nothing else needs to be changed besides what you have already done in the previous steps. (The files `*Gui.java` involve elements of Java that are outside of the scope of this course. You are **not** required to understand the code in these two files.)

The file `MazeGui.java` contains the main method. Compile the files. There may be a couple of warnings. You can ignore them (we would need to tinker with your BFS code to avoid the warning, and I promise the code will run fine despite the warning).

When the main method in `MazeGui` is run, it pops up a window (it may be buried under others). Click File->Open Maze File. and use the file browser to select the file named `level0.maz` included with the handout code. When the maze game board appears, you can use the arrow keys on your keyboard to move the penguin around. The goal of the game is to move the penguin to "pick up" all of the diamonds. Note the bricks with yellow arrows on them can be moved by pushing them into empty spaces using the penguin. Also, you can only push movable bricks, and cannot pull them. Use the arrow keys to pick up all the diamonds for the `level0.maz` and `level1.maz` maze files.

Your breadth first graph search method can also be used to find a solution to some of the simple levels, including level1.maz. To do this, the handout code considers a directed graph where each vertex in the graph is a unique configuration of the maze game board. The board is divided into squares the size of the penguin, called `cells`. The configuration of the board is specified by what entity is in each cell of the board, namely a wall, penguin, open space, diamond, or a movable brick. The edges from any vertex in the graph correspond to the allowed moves of the penguin for the state corresponding to that vertex. So there are at most four edges leaving any vertex of this graph. The start vertex is specified by the particular board configuration,

but the goal state is specified by a handle which simply checks if there are no more diamonds on the board.

To get your breadth first search and minimal path code to find a solution for the maze click Mode->Solve. For level0.maz, after a brief wait, it should show the goal state is reached, along with the yellow text "Done! (33)". Here the number 33 is the size of the reachable set constructed before finding a goal node.

Finally, click Mode->Show to show the path computed by your minimal path method. Use the buttons labelled "Next" and "Prev" (near the top-right corner of the window) to step forward and backwards through your solution.

You can try some other levels. But, beware, some require very large reachable sets. These can take a long time to execute and can eventually use up all the memory on the heap.

For this step you are to analyze the size of the reachable sets for the mazes defined by level3a.maz, level3b.maz and level3c.maz. In each case, derive a mathematical expression for the number of distinct vertices that are reachable from the start state. For example, we provide an appropriate answer for level3.maz below (you do not need to repeat this):

> **Sample solution for level3.maz:** The board is 7×7, and the Diamond is hidden by two movable walls in the bottom right corner (which actually can't be moved). The remaining movable brick can be pushed into any of the (7^2-3) remaining cells. Finally, for each of these placements of the movable brick there are (7^2-4) possible locations for the player (i.e. any remaining open space on the board, besides the four spaces taken up by the 3 bricks and the diamond). Therefore the total number or reachable states is $(7^2-3)(7^2-4)$ which is 2070. This agrees with the size of the reachable list returned by the MazeGui program.

Provide a similar analysis for each of the remaining level3?.maz files.

A Faster Algorithm

Step 5. We observed in step 4 above that the reachable set can get rather large. We search the reachable set using the method `get(MultiNode)` that you wrote in `ArrayListReachableSet.java`. This method just cycles through the items in the set, testing each of them for equality, stopping when either an equal item is found in the set or the whole set is searched.

Here we wish to speed up this search by using a tree data structure. We are going to use a self-balancing tree, called a red-black tree, which is implemented in the `java.util.TreeMap` class. Read the Java API for this class.

This class will allow us to store a set of n `(Board, MultiNode)` pairs. Here the first element of the pair is called the "key" and the second element is called the "value". Given a `TreeMap` instance `t`, we can add a key-value pair using `t.put(b, mb)`. Later on we can use the `t.containsKey(b)` method to check to see if any pair with a key equal to `b` is in the map. Also, we can recover the value associated with this key using the `t.get(b)` method. Moreover, since the underlying data structure is a self-balancing binary search tree, **all** these methods run in $O(\log(n))$ time, where n is the total number of pairs stored in the map.

One catch in getting this to work is that in order to achieve this performance `TreeMap` must be able to compare different keys for their ordering. That is, given two different boards `a` and `b`, then `a.compareTo(b)` should return +1 (or -1), indicating that `b` comes after `a` in the ordering (or before `a`, respectively).

So your first task is to implement the `compareTo` method in `Board.java` according to the specification provided by the `java.lang.Comparable` interface. Given a suitable `compareTo` method, the statements in

```
import graph.*;
import maze.*;
import java.util.TreeMap;
int[][] m = new int[2][2];
m[0][0] = Board.PLAYER;
Board a = new Board(m);
m[0][1] = Board.DIAMOND;
Board b = new Board(m);
a.compareTo(b)  // Should be 1 or -1
b.compareTo(a) == - a.compareTo(b)  // Should be true
TreeMap<Board, MultiNode> t = new TreeMap<Board, MultiNode>();
// Put a and its associated MultiNode in the TreeMap
MultiNode ma = new MultiNode(a);
t.put(a, ma);
// Put b and its associated MultiNode in the TreeMap
MultiNode mb = new MultiNode(b);
t.put(b, mb);
t.size()  // returns 2... since 2 pairs have been entered.
int [][] p = new int[2][2];
p[0][0] = Board.PLAYER;
p[0][1] = Board.DIAMOND;
Board c = new Board(p);
c.equals(a)   // Should return false
c.equals(b)   // Should return true
c == b        // false
t.containsKey(c)  // Should return true
t.get(c) == mb    // Should return true
```

Figure 4: The use of `TreeMap` to store a set of `MultiNode`s indexed by their associated `Board` objects.

Fig. 4 should then work in DrJava's interaction pane.

Write a new class called `ReachableTreeMap`, where `ReachableTreeMap` implements the same `ReachableSet` interface that `ArrayListReachableSet` implements. However, instead of storing the reachable set in an `ArrayList`, use a `TreeMap` instead.

Then in the Maze project, replace `ArrayListReachableSet.java` with `ReachableTreeMap.java`. Finally, edit `Reachable`'s constructor to build a new `ReachableTreeMap` instead of an `ArrayListReachableSet`. Recompile and run. Do you observe a significant increase in the program's speed (especially on problems with larger reachable lists)? Explain.

What to Hand In

Submit `Board.java`, `Reachable.java`, `ArrayListReachableSet.java` and `ReachableTreeMap.java` electronically. Also, print these four completed files, and staple the printed pages for each file together. Include these print outs with your written answers to the questions above (see steps 4, 5, and 6 above). Write your name and student number on each separate set of papers. Put all these stapled sets into an envelope, and clearly mark the envelope with "CSC148, Assignment 3" along with your name and student number.