Homework 9: Graph Search

Due: July 15, 2013 23:55 (11:55 PM)

Important

There are a few guidelines you must follow in this homework. If you fail to follow any of the following guidelines, you will recieve a **0** for the entire assignment.

1. All submitted code must compile under **JDK 7**. This includes unused code. Do not submit extra files that do not compile. Java is is backwards compatible so if it compiles under JDK 6, then it *should* compile under JDK 7.
2. Do not include any package declerations in your classes.
3. Do not include any *existing* class headers, constructors, or method signatures. It is fine to add extra methods and classes.
4. Do not import anything that would trivialize the assignment. E.g. you should not be importing `java.util.LinkedList` for a Linked List assignment.
5. You must submit your source code, the `.java` files. Do not submit the compiled `.class` files.

After you submit your files, please redownload them and make sure that they are indeed what you intended to submit. We will not be responsible if you submit the wrong files.

Assignment

In this assignment, you will be implementing three search algorithms: Breadth First Search (BFS), Depth First Search (DFS), and Dijkstra's Algorithm. These Algorithms are all very similar, and the only main difference is which data structure, which I will refer to as the "fringe" from now on, is used to store the next nodes for expansion. As you will soon see, the way in which you manage the fringe is extremely important and can result in wildly different paths. Just as in life, there are many methods and paths in which to reach a goal. Indeed, we will see that the journey is more important than the destination!

General Graph Search

Most (if not all) graph serach techniques follow the same principle. They all use some data structure (the fringe) to manage which nodes get explored and expanded next These algorithms also use a *visited set* in order to keep already visited nodes out of the fringe. The fringe is initialized with some start node, and the start node is added to the visited set. Then the algorithm goes into the main loop which does not stop execution until the fringe is completely empty. If the finge is ever empty, then we know that either the goal does not exist or it is unreachable from the start node. In either case, the method ends.

In the main loop, the algorithm removes the topmost element of the fringe and then examines its merits. If it is a goal node, then you return successfully out of the method. However, if it is not a goal node, then you must add it to the visited set and add all of its neighbours not already in the visited set back onto the fringe. Depending on the algorithm, you may have to calculate path costs. Once this step is over, then the algorithm loops yet again.

Here is some pseudocode for general graph search:

Algorithm 1 General Graph Search

```
procedure GENERALGRAPHSEARCH(Graph $G$, Vertex $start$, Vertex $end$)
    $fringe \leftarrow$ an instance of the fringe structure
    $set \leftarrow$ an empty set
    $node \leftarrow$ start
    $fringe.add(node)$
    $set.add(node)$
    while $fringe$ is not empty do
        $temp \leftarrow fringe.poll()$
        if $temp$ is $end$ then                          ▷ Check to see if we reached goal
            return success                               ▷ You can also return the path here as well
        else
            $set.add(temp)$
            for $neighbour$ in $G.getNeighbours(temp)$ do
                if $set$ does not contain $neighbour$ then
                    $fringe.add(neighbour)$              ▷ You may also calculate path costs, ect. here
                end if
            end for
        end if
    end while
    return failure
end procedure
```

Breadth First Search

Breadth First Search (or BFS) uses a queue as the fringe. Java has a multitude of classes that implement the queue interface, so you will not be short of choices. BFS works by exploring levels of the graph. It stars by exploring the immediate neigbours of the start vertex. In turn, the algorithm will explore the immediate nodes of all of those neighbour nodes. Since BFS explores the graph level by level, it is as a disadvantage if the goal vertex is deep within the graph. BFS will tend to return the shallowest path to the goal node. The time complexity of BFS is $O(|V| + |E|)$, where V is the vertex set and E is the edge set. In the absolute worst case, BFS will run in $O(|V|^2)$, since $max(|E|) = |V|^2$.

Depth First Search

Depth First Search (or DFS) uses a stack as the fringe. Java actually has a concrete stack data structure, so feel free to use that! DFS works by expanding as deeply as it can given the node it just popped off the fringe. It will then backtrack and explore the deepest vertex possible. DFS will never turn back unless it absolutely has to. This is a slight disadvantage when if the goal vertex is a shallow node. The time complexity of DFS is $O(|V| + |E|)$, where V is the vertex set and E is the edge set. In the absolute worst case, DFS will run in $O(|V|^2)$, since $max(|E|) = |V|^2$.

Dijkstra's Algorithm

Dijkstra's algorithm a greedy algorithm that finds the shortest path from one vertex to another. To do this, we must know the step cost of going between one vertex to another, as well as the total cost of the CURRENT path that was needed to reach the vertex we are examaning in order to establish some priority. Our general graph search algorithm changes ever so slightly. In order to calculate the priority of a certain vertex, V_T, being explored from another vertex V_S, use the following formula:

$$priorityValue = stepCost(V_S, V_T) + currentPathCost(V_S)$$

Implemenation Details

In this homework, you are to return the **PATH** that the algorithm returns for a graph. For Dijkstra's algorithm, you will also need to keep track of the current path cost to properly calculate the prority of a vertex. This all gets cumbersome to keep track of manually, so you are provided with a wrapper class that can keep track of the current path as well as the current path cost. You should be placing this wrapper class into your fringe data structures, and not the actual vertices themselves.

The "graphs" we provide you will be in the form of adjacency matrices. Adjacency matrices will be represented as two dimensional arrays, otherwise known as an "array of arrays". All of the vertex IDs will be integers ranging from 0 to $N - 1$, where N is the number of vertices in the graph. When you are finding the neighbours of some vertex, remember to scan that particular array from **LEFT** to **RIGHT** (ie. from index 0 to the end of the particular array). In otherwords, add the neighbours from least integer ID to greatest integer ID. The path you obtain will depend on the manner in which you place adjacencies into your fringe, so please be careful!

Deliverables

You must submit the following files.

1. `GraphSearches.java`