

Graph Travesals: Using Libraries, Building Libraries

- **Code reviews are to be done by 30 March 2011**
- **Problem 1 is due on Wednesday 30 March 2011**
- **Problem 2 is due on Wednesday 6 April 2011 but you need to get a lot done this week and have to check in your partial work.**

Practice Problems: The Java Collections Framework

Practice problems help you get started, if some of the lab and lecture material is not clear. You are not required to do these problems, but make sure you understand how you would solve them. Solving them on paper is a great preparation for the exams.

Read through the documentation for Java Collections Framework library. Find how you can use the stacks and queues and the priority queue defined there. Write a simple program that will test these algorithms.

Pair Programming Assignment

Code Reviews

Reminder only

This part of the assignment will act as a multiplier for the grade for the last week's assignment. If you fail to do this part, **the grade for the entire assignment** will be **zero**. If you fulfill this part of the assignment, the grade for this assignment will be computed in the usual way, based on the remaining three problems.

To get a passing grade for this part, you must meet with your TA during the next two weeks (**prior to 30 March 2011**) and go with him over the code in some of your previous homeworks, especially the *Chicken Game*. Please,

look at the wiki, where the TAs are posting their availability, email you TA to schedule an appontment, or talk to your TA during the lab. The TA assignments are posted on the wiki with the pair assignments.

10.1 William Shakespeare

10.1.1 The Application

Have you ever wondered about the size of Shakespeare's vocabulary? For this assignment you will write a program that reads its input from a text file and lists the words that occur most frequently, together with a count of how many different words occur in the file. If this program were to run on a file that contains all of Shakespeare's works, it would tell you the approximate size of his vocabulary, and how often he uses the most common words.

Macbeth, for example, contains about 4542 distinct words, and the word "*king*" occurs 202 times.

Macbeth, for example, contains about 3201 distinct words, and the word "macbeth" occurs 288 times.

10.1.2 The Problem

Start by downloading the file `Assignment10.zip` and making an Eclipse project named `Assignment-10-1` (with a `src` directory) to contain the files. Run the project, to make sure you have all pieces in place. The `Examples` class uses the `tester` package as we have done before.

You are given the file `Macbeth.txt` that contains the entire text of *Macbeth* and a file `StringIterator.java` that contains code that generates `Words` from a file (e.g., `Macbeth.txt`) one at a time. Save the file `Macbeth.txt` in the `Eclipse` project directory (where you find the subdirectories `src` and `bin`). The `Examples` class includes a code that invokes the processing of the complete text of the play *Macbeth*.
Note: Here you will use the imperative Iterator interface that is a part of Java Standard Library. Make sure to look up the documentation for this interface and understand how it works.

We've given you skeletons of the classes involved... finish the implementations by completing the following tasks:

1. Design the class `Word` that represents one word of Shakespeare's vocabulary together with its *frequency counter*. The constructor takes only the `String` (for example the word "king") and starts the *counter* at 1 (one).

   Two `Word` instances are equal to each other if they represent the same `String`, regardless of their frequency counters. That means that you have to override the `equals()` and `hashCode()` methods.

2. Implement a `toString` method for `Word` that returns the word `String` and its frequency, and an `increment()` method that increments the `Word`s frequency.

3. Design a class `WordsByFreq` that implements the `Comparator` interface, so that the words can be sorted *by frequencies*. (Be careful!) When you are done, place this class definition as the last part of the class definition of the class `Word`. This is called an **inner class**.

   Note: In this program there will be two ways of comparing the instances of the Word class - by the String that it represents and by the counter for the word that this instance represents.

4. Design the class `WordCounter` that keeps track of all the words we have seen so far. It should include the following methods:

```
// Record the Word objects generated by the given Iterator
//   and update the number of ocurrences
void countWords(Iterator<Word> it) { ... }

// How many different Words has this WordCounter recorded?
int words(){ ... }

// Prints the n most common words and their frequencies.
void printWords(int n) { ... }
```

   Here are additional details:

5. `countWords` consumes a `Word` iterator that generates the words and builds the collection of the appropriate `Word` instances, with the correct frequencies. This collection is then used by the next two method to show the results of our text analysis.

6. `words` produces the number of different words that have been counted.

7. `printWords` consumes an integer `n` and prints the top `n` words with the highest frequencies (using the `toString` method defined in the class `Word`).

Note: The given code expects that you implement the classes as given, with the same names and methods. It will then check whether your program works correctly. That does not mean you do not need to design tests.

Testing of the Shakespeare Project

Of course, you need to test all methods as you are designing them. Design the tests in two stages:

1. For the class `Word` and the the class `WordCounter` use a technique similar to what was done in the past assignments, i.e. design a class `Examples` with the necessary sample data and all tests. *We've astarted you off... just keep going*.

2. Convert all tests into `JUnit` tests. Hand in both versions.

Documentation

The projects should contain `Javadoc` documentation that should produce the documentation pages **without any warnings**. You do not need to submit the documentation pages to the repository.

10.2 Graph Algorithms: BFS, DFS, Shortest Path

This problem is a continuation and refinement of the **Graph Algorithms** problem from Lab 10. Save your files in a `Assignment-10-2/src` directory in your repository.

The problem description

Your program needs to represent a graph with nodes that represent capitals of the 48 US states. Each node has a name — the name of the state. For each node, record the information about the capital of that state and a list of the names of its neighbors. We assume that any time there is a conncetion between two states, it is a bi-directional connection (i.e. if you can get from here to there, you can also get from there to here).

We have already provided for you the class `USMap` that represents the map of the 48 US contiguous states with all the information you will need, including a method that computes distances between two states. (*This computation is grossly simplified and inaccurate, but suffices for our problem*)

Ultimately, the user will select the names of the origin and destination cities and which algorithm should be used to ocmpute the route, and your program will display the route in some user-readable way.

The assignment description will guide you through the process and should be used partially as a tutorial on graph traversal algorithms.

Using Libraries

Throughout the project you are encouraged to leverage as much as possible from the existing Java libraries. The designer should focus on the design of interfaces between tasks, between components, wrapper and adopter class that allow you to use an existing library class in a customized setting.

Classes for the Graph Traversal Algorithms: provided

The goal of this exercise is to use the *Java* libraries to do the work for us. We want to compute a path from one city to another, in a graph that represents the 48 contiguous US states. Start a new project *GraphAlgorithms*. You will be able to reuse some of what you have done before for the problems that refered to the US cities, but we are starting anew with more effective use of the Java libraries and a better organization of the data.

Start by downloading the file `Lab10-Graphs.zip` and making an Eclipse project named `Assignment-10-2` to contain the files. Run the project, to make sure you have all pieces in place. Run the tests. To help you focus on the interesting parts, we have given you the following classes:

- `City` that represents a capital of a state. It includes the location given as latitude and longitude, as well as methods that compute the location of the city on a `Scene` of size $400x400$.
- `State` that represent a state. Its fields are the name of the state (the two letter abbreviation, the capital `City` and an `ArrayList` of the names of the neighboring states.
- `USMap` that represents the whole graph - the 48 US capitals and the connections to the neighboring states. This class already has the code that will initialize it with the necessary data.

Classes for the Graph Traversal Algorithms: to design

1. **Reviewing existing code**

Start by looking at the representation of the graph of the US. It represents the graph of states as a `HashMap<String, State>`, that makes it very easy to find a state and its neighbors.

Note: (**this is not important**) The method `makeStates` uses a different technique for initializing an entire `ArrayList` to the given list of data. You do not need to understand how it is done. At some later time you may want to trace through *JavaDoc*s to understand how this is accomplished,

2. **Representing edges of the graph**

   In looking for a path from one city to another we keep track of the visited States. For each state we visit we also remember the state we came from and the distance we have traveled so far.

   Design a class `FromTo` that will represent this information: the name of the two states: the *origin* and the *destination*, as well as the distance between them in our graph. Because all information about the capitals of all states is already recorded in the class `USMap`, you only need to record the names of the states. However, the `distance` field is necessary, because when designing the shortest path algorithm we will include the distance we have traveled from the origin, not the distance between the two states represented.

3. **Representing a path in the graph**

   We now start defining the classes we will need to implement the *Graph Traversal Algorithms*. We need to keep track of the `USMap`, the *path* to the visited states, and a *To-Do-List* of states to visit. We start with the visited states:

   Define the class `Path` that keeps track of the visited states using a `HashMap`. Use the visited state's name as the `Key` and the instance of your `FromTo` class as the `Value`. So, for example, we may have the following information about states and traveling between them:

   ```
   MA - visited first: came from "", distance 0
   NY - we came from MA, distance 130
   NH - we came from MA, distance 60
   VT - we came from NH, distance 60 + 70
   NJ - we came from NY, distance 130 + 100
   PA - we came from NJ, distance 130 + 100 + 90
   ```

   Make sure you include the above example in your tests. (The distances you get may be different from the ones we gave you — the

given classes implement the computation of distances and your program should use it.

The class `Path` should have a constructor that consumes the `String` that is the name of the *origin* of our journey and adds the first item to its record of visited states. This first `FromTo` object should have the origin set to the empty `String`, the distance set to 0, and the destination to be the given *origin*.

4. In the class `Path` design the method `pathTo` that produces an `ArrayList` of `FromTo`-s we need to go through to get to the given `City`. So, for the above example, we would expect the following results:

```
pathTo(MA) --> [MA distance 0]
pathTo(NY) --> [MA distance 0;
                  NY distance 130]
pathTo(PA) --> [MA distance 0;
                 NY distance 0 + 130;
                  NJ distance 0 + 130 + 100;
                   PA distance 0 + 130 + 100 + 90]
```

5. In the class `Path` design the method `contains` that determines whether the state given as `String` is a destination in this `Path`.

6. In the class `Path` design the method `directionsFromTo` that consumes the state of origin and our desired destination (as two `String`s) and produces the travel directions as a `String`. For example,

```
directionsFromTo("MA", "MA") produces:
   "from MA go to traveling a total of 0 miles"

directionsFromTo("MA", "PA") produces:
   "from MA go to traveling a total of 0 miles
    from MA go to NY traveling a total of 130 miles
    from NY go to NJ traveling a total of 230 miles
    from NJ go to PA traveling a total of 320 miles"
```

Note: Do not worry about making this special. Later you may change your code to specify the direction of the travel, the distance to tne next city as well as the cumulative distance, but design the basic solution first and move on. Also, your formatting of the result is up to you, as long as it is helpful and readable by the user.

7. **Representing a list of edges to consider next**

   We now want to keep track of the neighbors of the states we plan to visit soon (the `ToDo` checklist). So, for example, if we visit `MA`, we will add to the `ToDo` checklist all of its neighboring states. However, there

are some restrictions. We do not add a neighbor to the checklist if it is already in the `Path`.

The interface `ToDo` describes the desired behavior:

```
interface ToDo{
   /** Add a new edge to this ToDo
    * @param edge the edge that we should add
    * @param path the path that has been already traveled
    */
   public void add(FromTo edge, Path path);

   /**
    * remove a state from the ToDo checklist
    * throw an exception if the checklist is empty
    * @return next state to be visited
    */
   public FromTo remove();

   /**
    * is this ToDo list empty?
    * @return true if there are no more states to visit
    */
   public boolean isEmpty();

   /**
    * Does this ToDo list contain a link to the given state?
    * @param state the given state
    * @return true if the given state is a destination
    */
   public boolean contains(String state);
}
```

8. Define the class `ToDoStack` that keeps track of the neighbors to visit soon that uses the `Java Stack` class to implement the `ToDo` interface as a `stack`.

   Note that here we do not add to the *stack* any edges leading to a node that is already destination in one of the edges in this *stack*.

   Note: We will reduce the credit for this part if you do not use the Java library classes for this part.

9. Define the class `ToDoQueue` that keeps track of the neighbors to visit soon that uses the `Java LinkedList` class to implement the `ToDo` interface as a `queue`.

   Note that here we do not add to the *queue* any edges leading to a node that is already destination in one of the edges in this *queue*.

   Note: We will reduce the credit for this part if you do not use the Java library classes for this part.

10. **Designing graph algorithms**

    Define the class `GraphAlgorithms` that implements the graph algorithms for a specific graph. The beginning of the definition is given as follows:

```
/**
 * To represent an implementation of the three classical
 * graph traversal algorithms:
 * Breadth-First Search, Depth-First Search
 * and Shortest Path.
 */
class GraphAlgorithms{
   /** the data that describes this graph (a US map) */
   USMap graphData;

   /** The path: the list of edges leading to the visited nodes */
   Path path;

   /** The list of the edges leading from the visited nodes
    * to their neighbors - candidates for next step in search */
   ToDo todolist;

   /**
    * Initialize the graph that will not change
    */
   GraphAlgorithms(USMap graphData){
      this.graphData = graphData;
   }

   /* ... */
}
```

    Add the method `findRoute` that will initialize the remaining data from the given *origin* of the travel, the *destination* of the travel, and the selected algorithm (one of `"BFS"`, `"DFS"`, `"SPS"`) compute the route using the selected algorithm and produce an `ArrayList` of `FromTo` data that represents the desired route.

    Note: At the beginning you may put a stub here that only prints out what was the selected origin, destination, and algorithm, but initalizes the path and the 'todo' list.

The ground work you have done here provides all the parts you need for implementing three different graph traversal algortihms *Breadth-First Search*, *Depth-First Search*, and *Shortest Path Search*. You should make sure you finish this part by next Wednesday, March 30th.

Algorithms

Your model should implement three graph traversal algorithms:

- Depth-First Search: uses a *Stack* to record the *ToDo* information
- Breadth-First Search: uses a *Queue* to record the *ToDo* information
- Shortest Path Search: uses a *Priority Queue* to record the *ToDo* information

The detailed description of the algorithm appears in a separate document. You will encounter a significant penalty for repeating the code - one algorithm implementation should run all three variants, distinguishing between them by selecting the appropriate implementation of a common interface for dealing with the *ToDo* information.

10.3 User interactions: To be done next week

The file `GraphAlgoView.java` provides the code that creates a GUI allowing the user to select one of the three algorithms, the origin and the destination for the path.

Read the code, or at least the documentation and find the three places where you need to add the code that will invoke your implementation of the three algorithms.

You need to add code to your program that will show the user the path you have computed.

Before the user selects the algorithm to use and the origin and the destination for the path, she must be able to view a representation of the graph for which the computation is to be done.

This can be a graphical display, a text that lists the nodes and the edges (with their weights), or a graphical display of the text that lists the nodes and the edges.

Once the path has been computed, the user should be able to see the resulting path.

This may be a graphical display, or just a text listing the nodes along the path.

Here is a list of possible enhancements:

- Highlight the path is a different color in the graphics display.
- Display the steps in the search by highlighting in a different color the *visited* nodes, the *fringe* nodes (those currently in the queue or the stack), the *origin*, the *target*, and the *unseen* nodes. Animate the process using either the timer, or a user advance triggered by a key press.

- Animate the reconstruction of the path by traversing from the found target back to the previous node, all the way up to the origin.

10.4 Abstractions

The code you have written should work for any graph that is represented by node labes given as `String`s and some class that represents the node data. There may be some methods required to be present in the various classes we need to implement this algorithm, but we should be able to parametrize all of them over the type of data that represents each node of this graph.

The last part — with more details available next week — will be to generalize your code so you can test it with graphs we provide.