06-30175
Data Structures & Algorithms
Spring Semester 2010-2020

The University of Birmingham
School of Computer Science
© Alan P. Sexton 2010-2020

Assignment 02

1 Introduction

This assignment will be marked out of 10 and constitutes 10% of your final module mark.
Deadline for submission is:

14:00 Friday 14th February

With apologies to J.R.Tolkien, the dwarf mine under the mountain of Moria (c.f. `https://en.wikipedia.org/wiki/Moria_(Middle-earth)`) has been discovered and your job is to program a drone to explore this underground world and map out the maze formed by the chambers and connections between them.

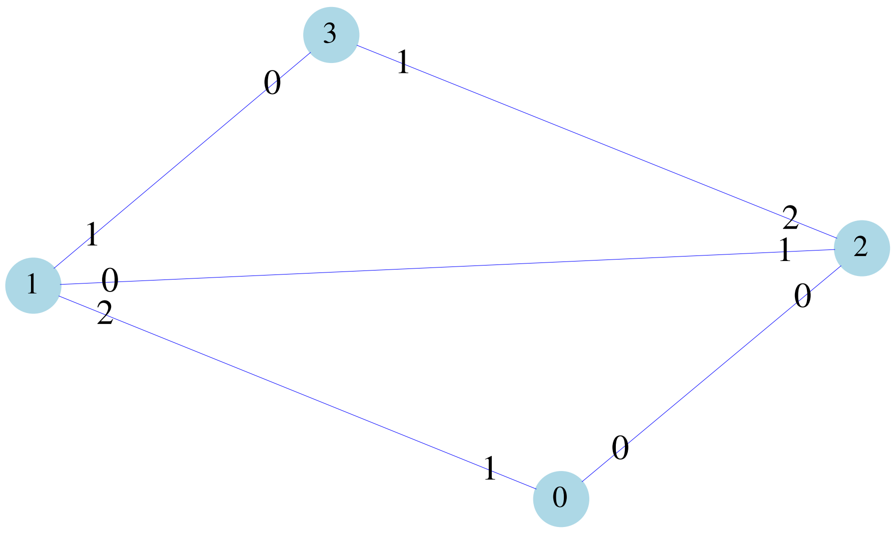

Figure 1: A small sample maze

There are an unknown number of chambers in the maze. Every chamber has a number of doorways (the doors themselves are no longer intact so there is no barrier to traversing the passageways or "wildlife" to deal with during the search). For simplicity, each chamber is numbered, starting at 0, and each doorway is numbered in each chamber, ranging from 0 to the number of doorways in the chamber minus 1.

Figure 1 indicates a sample maze (Moria itself has **many** more chambers). The numbers in the circles are the chamber numbers and the numbers on the lines are the doorway numbers. Thus doorway 1 of chamber 0 connects to doorway 2 of chamber 1, etc.

The drone always starts in chamber 0 and, after exploring the whole maze, should finish in chamber 0 as well.

Your drone has access to a sensor system (a Maze class) that it can interrogate to identify

- which chamber it is currently in,
- how many doorways are visible in the current chamber and,
- which chamber it then enters and through which doorway, when the drone proceeds through a doorway and down the passage to the next chamber.

The sensors cannot tell you about any chamber that the drone is not currently in.

In summary, for this assignment you have to write the code to accomplish a number of tasks:

a. Implement the code for a method to decide on which doorway to enter at each step during the search so that every chamber is explored and the drone ends up back in the start chamber (chamber 0).

b. Keep a record of every doorway that the drone passes through, either in leaving or in entering a chamber, to make it available at any point to the drone's user (i.e. the caller).

c. In case of low battery, the drone must be able to calculate a direct route back (a sequence of doorways to take) to the start chamber from whatever chamber it is currently in. This route should avoid taking detours, loops or entering dead-ends that it then needs to backtrack out of.

2 Marking

- Any submission that passes all the tests in `DroneTest.java`, although on a range of different mazes other than just the one current used in that file, will get full marks.
- Further tests may be added during marking to distinguish between the different ways that a submission that is not passing all the tests is failing, so that partial marks can be awarded.
- There will be 1 mark if your submission compiles correctly and can be run and passes the single test that already passes in the initial assignment files.
- If your submission is not structured correctly or does not compile, or crashes or enters an infinite loop before any tests pass, then no marks will be earned.

3 Plagiarism

Plagiarism will not be tolerated: it is unfair to the other students and prevents you from learning, and would give you a mark you don't deserve, and qualifications you don't deserve, hence being unfair to society as a whole too. Please behave well and reward the module lecturer and TA's by submitting your own, hard work.

All submissions will be checked for copying against other submissions and against sources on the web and elsewhere. When the module lecturer decides that there is evidence of plagiarism, the case will be forwarded to the Senior Tutor, who will look at the evidence and apply penalties and warnings, or, if necessary, forward this to the University.

- University regulations on plagiarism: `https://intranet.birmingham.ac.uk/as/studentservices/conduct/plagiarism/index.aspx`
- University Library sources on plagiarism: `https://intranet.birmingham.ac.uk/as/libraryservices/library/referencing/index.aspx`

4 Student Welfare

If you miss or are going to miss a deadline, or are getting behind the learning material, for reasons outside of your control, please contact the Wellfare (`https://canvas.bham.ac.uk/courses/28575`. Occasionally students represent the University in sports, programming competitions, etc., and if this is going to affect your ability to comply with a deadline, Wellfare should be informed. It is the Welfare Team, not the teaching team, who can award extensions and cancellations, and devise other measures to cope with adverse situations. It is important to contact welfare as soon as possible when such situations arise.

5 Background to this Assignment

While the first assignment was all about the internals of implementing pointers and lists, this assignment is about using some of the available data structures in the Java standard library (the collection classes) so that you can avoid having to implement your own basic data structures but instead use the various standard lists, maps, arrays, queues and stacks, and their associated implemented algorithms, to solve your programming problems.

There is not a great deal of code that needs to be written, but the logic for some of it can be a bit involved, so please start on the assignment early and be diligent with your unit tests and study the identified classes in the JavaDoc.

The collection classes you will need to use are:

- `Deque<>`: A *Double Ended Queue* interface with the most commonly used implementations `ArrayDeque<>` and `LinkedList<>`

  You use the a `Deque<>` in Java as the standard way to implement both a *Stack* and a *Queue* as well as a *Deque*
- `Set<>`: A *Set* interface with the most commonly used implementations `HashSet<>` and `TreeSet<>`

  This implements the set abstraction, where we can add and remove elements and easily check whether an element is currently in the set. Unlike a list, even if we add multiple elements with the same value to a set, that value will only appear in the set once.

- `Map<>`: A *Map* interface with most commonly used implementations `HashMap<>` and `TreeMap<>`

  `Map<>` has 2 type parameters. For example, a `Map<String, Integer>` provides a look up table that associates String *keys* with Integer *values* so that you can ask for the Integer that you have previously associated with a String: think of a phone book where you can look up phone numbers by the name of the person you want to call.

  While you do not need to use `Map<>` collections in the code you write for this exercise, the `Maze` class uses them to store, and easily look up, the connections between chambers

- `Collections` and `Arrays`: (Note that the "s" on the end of these class names are important!). These are library classes that provide many static utility functions that can make your life easier. You should browse through their methods so that you know what is available and thus avoid having to write methods that are already implemented there.

6 Setup and Specification

You have been provided with a Java project folder that contains sources and tests for this assignment: see the `dsa_assignment2` folder in the `dsa_2029` GIT repository (don't forget to do a "`git pull`" inside your copy of the repository to update it). You have been provided with the following files and packages:

```
dsa_assignment2/src/main/java/dsa_assignment2/
    Drone.java
    DroneInterface.java
    Maze.java
    Portal.java
dsa_assignment2/src/test/java/dsa_assignment2/
    DroneTest.java
```

If you set the top level `dsa_assignment2` as an Eclipse project, with `src/main/java` and `src/test/java` as the source folders on your build path, and add the user libraries for hamcrest-all (or hamcrest-core), JUnit and Log4J to your project, you should be able to compile and run the tests (all except one of which should fail!)

Your task is to add missing code to the methods in `Drone.java` to make all the tests pass.

- The precise information about the behaviour required of the missing code is detailed in the Javadoc comments in the `Drone.java` and `DroneInterface.java` files
- You should modify the `Drone.getStudentId()` and `Drone.getStudentName()` in `Drone.java` to return your student id and name.
- You should **NOT** add any further class variables to `Drone.java`, or change the types of the existing class variables nor modify any of the other files in the `main` part of the src tree. You can modify the `DroneTest.java` file or add any other test files as you please. You should not modify the package structure or add any new classes that `Drone.java` depends upon to run. Your final submission will be purely the `Drone.java` file, so any modifications outside that file will not be considered and the `Drone.java` that you submit must work with the other files as they stand.
- You should not use any print statements in any files or or output to standard out or standard error streams: if you want to have some debug output, use the logging calls for Log4j. The logging system will be switched to disable log output when your submission is run for marking purposes.
- For marking purposes, your code will be compiled and executed against a test set in a secure sandbox environment. Any attempts to break the security of the sandbox will cause the execution of your program to fail. This includes infinite loops in your code.
- When you have completed your code changes to your satisfaction, submit your `Drone.java` file

7 Generating JavaDoc

You may find it helpful to generate the JavaDoc HTML files so that you can easily browse and navigate through the JavaDoc information for the assignment classes as well as the related library classes. Once they are generated you will be able to view the JavaDoc for your project classes, methods and variables in a web browser just like you can for the standard library classes (hover over an identifier and press `<Shift-F2>` on Linux).

To do this, in eclipse, when you have no compilation errors in the project:

1. `Project|Generate Javadoc...`
2. Select the `src/main/java` "types" to generate Javadoc for

3. Select "`Private`", and set the destination to be a directory called "`javadoc`" in the top level of your project directory (beside `bin` and `src`). It is important to use that name instead of the default "`doc`", because your `.gitignore` file causes GIT to never store the `javadoc` directory (as it should not), but it will store a `doc` directory.
4. Click on `Next>`
5. Select all the `Basic Options`, all the `Document these tags` and click the `Select All` button on the right.
6. Click on `Next>`
7. Click on `Finish`

It may ask if you want to update the Javadoc location for your project with the chosen destination folder. If so, click on `Yes`. In such a case, the generation may fail without giving any warning. If you run the Javadoc generation again, it will work the second time.

8 Support Classes

8.1 Immutable Objects

The concept of *immutable objects* is important in the design of Java data structures. An immutable object is an object of a class that does not allow the value of the object to be changed after it has been constructed. Examples from the Java standard library are the `String` class and the `Integer` class. Objects of a class are made immutable by ensuring:

- that any *instance fields* (sometimes called *"class fields"* or *"class variables"* are declared to be private,
- that there are no mutators (setters) for the instance fields,
- that the code of the class itself never modifies the class fields after construction, and
- that we do not *"leak"* references to modifiable class fields outside the class. For example, if we have a private array of ints as an instance field, and we have an accessor (getter) method that returns this array to an object of another class, then this other object can modify the contents of this array and our supposedly immutable object will find that the array contents that were supposed to be unchangeable have now been changed without its knowledge.

  To avoid this problem we must ensure that we never return a reference to a non-immutable instance field from any method of our immutable class. We can safely return immutable values or references (e.g. String, Integer, etc.), but if we need to return an inherently mutable object, such as an array, or Collection class object, we must not return a direct reference to it but instead, either provide controlled getters for the information in the mutable structure, or we return a copy of the object so that if the copy is changed, the original version is not affected.

  For a collection class object which contains immutable objects, you can call the `toArray()` method to return an array which is a copy of the contained objects. For an array you can call the `Arrays.copyOf(..)` method to get a new copy of the array.

8.2 Portal

Our maze has chambers and each chamber has a number of doors. The passageways between chambers connect a particular door in one chamber to a particular door in another. This pairing of door number with the number of the chamber that the door is in is such a basic element in this program that the `Portal` class has been written to capture it.

It is an immutable object, thus it is safe to pass around references to these objects even if they are in other data structures.

Portal has proper `equals()` and `hashCode()` methods (generated by Eclipse: `Source|Generate hashCode() and equals()...`). These are necessary to be able to put objects of this type into various collection classes, such as `Set<>`, `Map<>`, etc.

8.3 Maze

The `Maze` class is designed to represent the properties of an underground maze as seen by a drone. The drone can only ever see the chamber it is in (i.e. identify the chamber and the doors in that chamber), and it can go through a doorway to find the chamber connected to it. The drone cannot see or query the maze about anything beyond that. Any other information it needs it has to build up by exploring the maze. As the drone wanders through the maze, the maze maintains the information about what chamber it is in so that it can respond appropriately to the drone's queries.

`Maze`'s `traverse(...)` method handles moving to a different chamber by going through a particular door (the parameter) in the current chamber. The `Portal` object through which the drone enters the new chamber is returned.

The drone can always ask what chamber it is currently in and how many doors the current chamber has. The drone cannot just jump, or teleport, to any chamber in the maze, it has to find its ways there.

There are a number of constructors for `Maze`. With one exception they all generate a random maze with some number of chambers, some average number of doors per chamber and, optionally, a seed number for the random number generator.

If you use the same seed number when you generate a maze with the same number of chambers and same average number of doors per chamber, then you will get the same maze generated. This can be useful for debugging purposes. If you do not set the seed, then a different maze will be generated each time.

The maze will never have connections that loop back from one chamber to the same chamber. If you set the average number of doors too high it will stop adding connections, and hence doors, when every chamber is directly connected to every other chamber.

The exceptional constructor is a copy constructor. This makes a new maze which is a copy of the parameter maze. Since the new maze shares the data structures of the old maze (except for the `currentChamber` instance field), this is a very quick and cheap operation. The purpose of this copy constructor is to make testing easier. You can see it in action in the `DroneTest` class.

There is a simple `toString()` method that returns a printable `String` version of the maze connections, and a `toDotFormat()` method, that generates a version of the same information that can be used to generate a graphical map of the maze. See the JavaDoc of the method for more information and see the `DroneTest` class for using them in log messages.

The only complicated part of this class is in generating a random maze. It is not necessary to understand this to do the assignment, but you might find it worthwhile to study for its examples of using `Set<>` and `Map<>`. Basically it works in two halves. The first half generates a random tree to connect all the chambers together, by starting with chamber 0 in the tree and, until all the chambers are added to the tree, choosing a random chamber that has not yet been added and connecting it to a random chamber in the tree. In each case the next available door number is used for the connection at each end. The second half just adds random chamber connections from a set of possible connections (excluding connections between the same chambers or connections that already have been made.

8.4 DroneInterface and DroneTest

These two classes require little discussion. *DroneInterface* is there to ensure that the `Drone` class has the correct method declarations.

`DroneTest` is a standard JUnit test class. You can change the baseMaze constructor to try different mazes with different sizes.

8.5 Drone

This is where you have to write your code. You only need to fill in your student id number and your name in the `getStudentID()` and `getStudentName()` and otherwise write the code for 3 methods:

1. `searchStep()`: This executes a **SINGLE** step in searching through the maze and corresponds to the drone choosing a door to go through, traversing through the door and ending up in the connected chamber. The Portal by which the drone enters the connected chamber should be returned. If the drone has completed the search and is back in chamber 0, then any further calls to this method should return null. See section 9 for a discussion on the search algorithm.

   Note that this executes a **SINGLE** step only, i.e. it chooses a doorway in the current chamber to go through and goes through it by calling the `traverse()` method from the `maze` object **ONCE AND ONLY ONCE**. To search the entire maze, this `searchStep()` method would have to be called many times: once for every door the drone goes through.

2. `getVisitOrder()`: This simply returns an array of Portal objects recording the path that the drone has taken through the maze so far. Note that both the Portal taken to exit a chamber as well as the Portal taken to enter the next one should be recorded. Thus the first Portal in the array should be the Portal through which the drone left chamber 0 at the start of its search, while the last element in the array should be Portal by which the drone has entered the chamber that it is currently in.

3. `findPathBack()`: This returns an array of Portal objects that show the route that the drone should take to get back to chamber 0 with no diversions on the way. Here the list of Portals should contain only those ones to exit the sequence of chambers that the drone will pass through on its way back to chamber 0. Thus the first element will be the Portal to exit the current chamber, the second element be the one to exit the chamber that you are in after exiting the first chamber, etc. The final Portal should be the one to exit the chamber that returns the drone to chamber 0. If your current chamber is already chamber 0, then the `findPathBack()` method should return an empty list (NOT null!). Note that there may be many different paths back. You are NOT required to find the shortest one and the tests used for marking will only check that your path does actually work to get the drone back to chamber 0 and that it does not have any loops or dead-end backtracking.

Section 11 has the log output of an execution of a solution to this problem that shows the current chamber, the visit order list and the path back after every search step. If you work through that you will see precisely what is required.

9 Search Algorithm

The simplest suitable search algorithm to implement, and the one you are advised to use, is called *Depth First Search*. Here the idea is, at every step, to select, if possible, a doorway out of the current chamber that you have not used before and go through that doorway. If there is no such unused door, you have to backtrack out of the current chamber. To figure out how to do that, when you find an unused door to go through to enter the next chamber, you push the Portal that you enter that next chamber through onto the stack. When you have exhausted all the doors in that chamber, you pop off the Portal from the stack and follow that door to backtrack. You are finished the search when you try to pop a backtrack Portal off the stack but the stack is empty. At this point you should be back in chamber 0.

You should use:

a. the `visitStack` instance field for the backtrack stack,

b. the `visited` instance field to keep track of which Portals you have passed through (either entering or exiting a chamber). When you are not back tracking, and you are trying to choose a new, unexplored, portal to go through to leave this chamber, you should avoid choosing any door in this set.

c. the `visitQueue` instance field to keep track of the sequence of Portals you have passed through.

   This will be what both the `getVisitOrder()` and the `findPathBack()` methods will use in order to construct their return values.

10 Tips and Strategy

The tests are designed to support incremental development of your solution. Target a particular test to get working at each stage and only go on to target a different test when the current test is fully working. I suggest you tackle the tests in the following order and try to resist the temptation to add code to handle a later test before you have completed the current test you are working on, because this ends up making your code messier and harder to change as you are developing:

1. `testDroneConstruction()`: should already work!
2. `testCheckStudentIdentification()`: a particularly easy one!
3. `testDroneOneStep()`: Just checks that if you take one step you move through one passage and that you correctly report the Portal you enter.
4. `testDroneSearch()`: This is the full search. It is a little involved but should not be too difficult
5. `testDroneFinalVisitOrder()`: This should be very easy once you have the previous tests passing
6. `testDroneAllVisitOrders()`: Again very easy
7. `testDroneSimpleBackPath()`: Easy again, as you do not, for this test, have to have removed loops or dead-ends that you need to backtrack out of to get home
8. `testDroneBackPathNoLoops()`: More complicated for extra marks. You do not need to find the shortest path home, but you do need to find a path home that does not enter any chamber a second time.

Note that if you have `getVisitOrder()` working, then it is pretty easy to get a basic version of `findPathBack()` working, where a basic version does not remove the loops, or dead-end backtracking. This basic version would pass `testDroneSimpleBackPath()` but not `testDroneBackPathNoLoops()`.

11 Sample Output

```
dsa_assignment2.DroneTest.logMazeAndVisitsAndBackPath(DroneTest.java:38): 0    DEBUG: Maze in toString format:
1:0 -- 2:1
0:0 -- 2:0
1:1 -- 3:0
2:2 -- 3:1
0:1 -- 1:2

dsa_assignment2.DroneTest.logMazeAndVisitsAndBackPath(DroneTest.java:39): 2    DEBUG: Maze in dot format:
// Use the graphvis package to generate an image. Put this output in x.gv and run:
// neato -Tpng -ox.png x.gv
// to generate the x.png with the image of the connections. Alternatively, use
// "fdp" or "circo" with the same arguments to get a different layout
strict graph G {
  size="6,6!" dpi=100 splines=true ratio=fill margin=0
  edge[penwidth=0.2 color=blue forcelabels=true labelangle=0]
  node[shape=circle margin=0.02 fixedsize=false
       width=0.25 style=filled fontsize=12 color=lightblue]
  1 -- 2 [taillabel=0 headlabel=1]
  0 -- 2 [taillabel=0 headlabel=0]
  1 -- 3 [taillabel=1 headlabel=0]
  2 -- 3 [taillabel=2 headlabel=1]
  0 -- 1 [taillabel=1 headlabel=2]
}

dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:57): 3    TRACE: Drone search steps log Begin:
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 3    TRACE: Current chamber: 2
```

```
Visit order: [0:0, 2:0]
Path back:   [2:0]
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 3    TRACE: Current chamber: 1
Visit order: [0:0, 2:0, 2:1, 1:0]
Path back:   [1:0, 2:0]
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 3    TRACE: Current chamber: 3
Visit order: [0:0, 2:0, 2:1, 1:0, 1:1, 3:0]
Path back:   [3:0, 1:0, 2:0]
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 3    TRACE: Current chamber: 2
Visit order: [0:0, 2:0, 2:1, 1:0, 1:1, 3:0, 3:1, 2:2]
Path back:   [2:0]
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 3    TRACE: Current chamber: 3
Visit order: [0:0, 2:0, 2:1, 1:0, 1:1, 3:0, 3:1, 2:2, 2:2, 3:1]
Path back:   [3:0, 1:0, 2:0]
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 3    TRACE: Current chamber: 1
Visit order: [0:0, 2:0, 2:1, 1:0, 1:1, 3:0, 3:1, 2:2, 2:2, 3:1, 3:0, 1:1]
Path back:   [1:0, 2:0]
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 3    TRACE: Current chamber: 0
Visit order: [0:0, 2:0, 2:1, 1:0, 1:1, 3:0, 3:1, 2:2, 2:2, 3:1, 3:0, 1:1, 1:2, 0:1]
Path back:   []
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 3    TRACE: Current chamber: 1
Visit order: [0:0, 2:0, 2:1, 1:0, 1:1, 3:0, 3:1, 2:2, 2:2, 3:1, 3:0, 1:1, 1:2, 0:1, 0:1, 1:2]
Path back:   [1:2]
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 4    TRACE: Current chamber: 2
Visit order: [0:0, 2:0, 2:1, 1:0, 1:1, 3:0, 3:1, 2:2, 2:2, 3:1, 3:0, 1:1, 1:2, 0:1, 0:1, 1:2, 1:0, 2:1]
Path back:   [2:1, 1:2]
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:63): 4    TRACE: Current chamber: 0
Visit order: [0:0, 2:0, 2:1, 1:0, 1:1, 3:0, 3:1, 2:2, 2:2, 3:1, 3:0, 1:1, 1:2, 0:1, 0:1, 1:2, 1:0, 2:1, 2:0, 0:0]
Path back:   []
dsa_assignment2.DroneTest.logDroneVisitsAndBackPath(DroneTest.java:67): 4    TRACE: Drone search steps log End
```