CS1027: Foundations of Computer Science II
Assignment 2: Path Planning
Due: February 27, 11:55pm.

Purpose

To gain experience with

- The solution of problems through the use of stacks
- The design of algorithms in pseudocode and their implementation in Java.
- Handling exceptions

1. Introduction

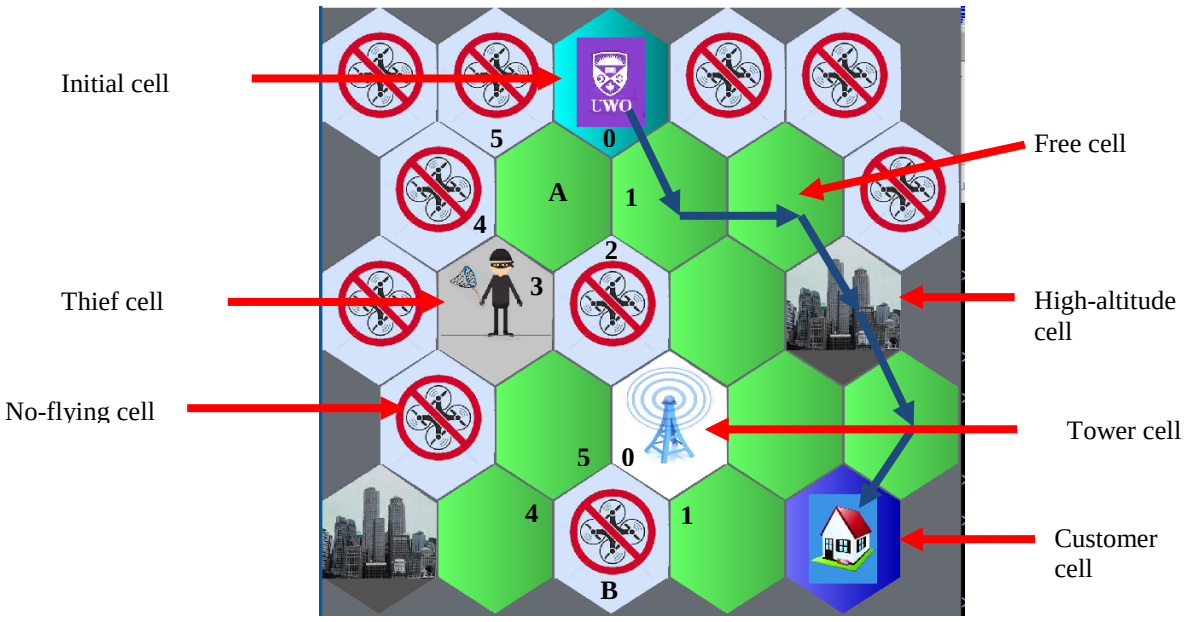
The UWO store has decided to use drones (unmanned flying vehicles) to deliver merchandise to its customers. For this assignment, you will design and implement a program in Java to compute a path that a drone can follow from the UWO store to a customer's house. You are given a map of the city, which is divided into hexagonal cells to simplify the task of computing the path. You are also given the initial map cell, where the UWO store is located, and the map cell where the customer's house is situated. The following figure shows an example of a map divided into cells.

There are 6 types of map cells:

- *Free cells*. The drone can safely fly across these cells.
- *High-altitude cells*. These are map cells where buildings are located. The drone can fly across these cells, but it must increase its flying altitude, which consumes additional power from its battery, reducing its flying time.
- *Thief's cells*. These map cells are locations where thieves lurk. Thieves in these places will try to bring down a drone to steal the merchandise it carries. A drone can go across these cells, but it needs to move very fast, and this also consumes additional power from its battery.
- *Tower cells*. These are map cells where cellular towers with large antennae transmit electromagnetic signals that interfere with the navigation system of the drone. The drone must avoid these cells **and any cells adjacent** to a tower cell.
- *No-flying cells*. These cells contain skyscrapers that impede the flight of a drone, and so a drone cannot fly across these cells.
- *Customer cell*. This is the map cell where the costumer's house is located. This is the final destination for the drone. The customer cell is a free cell.

Initially the drone is positioned in the free map cell containing the UWO store. As the drone flies toward the customer cell, it might have several choices as to what path to follow; when given the choice between going to a free, a high-altitude, or a thief's cell, the drone will prefer the free one over the other two, and it will prefer a high-altitude cell over a thief's cell.

The following figure shows an example of a map. The starting cell is marked with a purple UWO label. There is one tower cell, a thief's cell and two high-altitude cells. The customer cell is located near the bottom right. A path from the starting cell to the destination is marked. Note that the free cells adjacent to the tower cannot be part of the solution, so the drone has to go over a high-altitude cell.



Each map cell has up to 6 neighboring cells indexed from 0 to 5. Given a cell, the neighboring cell located in its upper right corner has index 0 and the remaining neighboring cells are indexed in clockwise order. For example, in the above figure the neighboring cells of cell A are indexed from 0 to 5 as shown. Note that some cells have fewer than 6 neighbors and the indices of these neighbors might not be consecutive numbers. For example, cell B in the figure has 4 neighbors indexed 0, 1, 4, and 5.

2. Classes to Implement

A description of the classes that you need to implement in this assignment is given below. You can implement more classes, if you want. You **cannot** use any static instance variables. You **cannot** use java's provided *Stack* class or any of the other java classes from the java library that implements collections. The data structure that you must use for this assignment is an array, as described in Section 2.1.

2.1 MyStack.java

This class implements a stack using an array. The header of this class must be this:

public class MyStack<T> implements MyStackADT<T>

You can download *MyStackADT.java* from the course's website. This class will have the following three private instance variables:

- *private T[] arrayStack*. This array will store the data items of the stack.
- *private int numItems*. This variable stores the number of data items in the stack.
- *private int maximumCapacity*. The value of this variable indicates the maximum size of the array that stores the items of the stack.

This class needs to provide the following public methods.

- *public MyStack()*. Creates an empty stack. The default initial capacity of the array used to store the items of the stack is 10. The default maximum size for the array storing the items of the stack is 1000.

- *public MyStack(int initialCapacity, int maxCap).* Creates an empty stack using an array of length equal to the value of the first parameter. The maximum size for the array representing the stack is given by the second parameter.
- *public void push (T dataItem) throws OverflowException*. Adds *dataItem* to the top of the stack. If the array storing the data items is full, you will increase its capacity as follows:
    - If the capacity of the array is smaller than 60, then the capacity of the array will be increased by a factor of 3.
    - Otherwise, the capacity of the array will increase by 100. So if, for example, the size of the array is 60 and the array is full, when a new item is added the size of the array increases to 160.
    - If the size of the array is increased to more than the value of *maximumCapacity* (i.e. *maximumCapacity* + 1 or larger) an *OverflowException* exception will be thrown. When creating an object of the class *OverflowException*, a *String* message needs to be passed as parameter to the constructor. We set an upper bound on the capacity of the stack to prevent your program from exhausting the memory of the computer if a bug causes it to enter in an infinite loop.
- *public T pop() throws EmptyStackException*. Removes and returns the data item at the top of the stack. An *EmptyStackException* is thrown if the stack is empty. When creating an *EmptyStackException* a *String* message must also be passed as parameter.
- *public T peek() throwns EmptyStackException*. Returns the data item at the top of the stack **without** removing it. An *EmptyStackException* is thrown if the stack is empty.
- *public boolean isEmpty()*. Returns true if the stack is empty and it returns false otherwise.
- *public int size().* Returns the number of data items in the stack.
- *public String toString().* Returns a String representation of the stack. Read the lecture notes to learn how to implement this method.

You can use implement other methods in this class, but they must be declared as private.

2.2 ComputePath.java

This class will have an instance variable

Map cityMap;

This variable will reference the object representing the city map where the drone will be flying. This variable must be initialized in the constructor for the class, described below. You must implement the following methods in this class:

- *public ComputePath (String filename).* This is the constructor for the class. It receives as input the name of the file containing the description of the city map, and the initial and destination map cells. In this method you must create an object of the class *Map* (described in Section 5) passing as parameter the given input file; this will display the map on the screen. Some sample input files are also provided in the course's website. Read them if you want to know the format of the input files.
- *public static void main (String[] args).* This method will first create an object of the class *ComputePath* using the above constructor. The parameter for the constructor will be the first command line argument of the program, i.e. *args[0]* (see Section 4); the command line argument is the name of the input file. This method then will try to find a path from the initial cell to the destination according to the restrictions specified above. The algorithm that looks for a path from

the initial cell to the destination **must use a stack**. Suggestions on how to look for this path are given in the next section. The code provided to you will show the path selected by the algorithm as it tries to reach the destination, so you can visually verify how the program works (read Section 3 and Section).

- *private boolean interference(MapCell cell).* The parameter is the cell where the drone currently is. Class *MapCell*, described in Section 5, represents the map's cells. This method returns *true* if any of the adjacent cells to the current one is a tower cell, and it will return *false* otherwise. Read Section 5 to learn how to get the cells that are adjacent to the current one.
- *private MapCell nextCell(MapCell cell).* The parameter is the cell where the drone currently is. This method returns the best cell for the drone to move to from the current one. If several unmarked cells (details about marked and unmarked cells are given in Sections 3 and 5) are adjacent to the current one, then this method must return one of them in the following order:
    - a free cell, if one exists, or the customer cell. If there are several free cells, then the first one (the one with smallest index) is returned. Read the description of the class *MapCell* below to learn how neighboring cells are indexed.
    - a high-altitude cell, if no adjacent free cells exist
    - a thief's cell, if there is no neighboring free or high-altitude cell.

  If there are no unmarked cells adjacent to the current one this method returns `null`.

Your program must catch any exceptions that might be thrown. For each exception thrown an appropriate message must be printed. The message must explain what caused the exception to be thrown.

You can write more methods in this class, but they must be declared as private.

3. Algorithm for Computing a Path

Below is a description in pseudocode of an algorithm for looking for a path from the starting cell to the destination. Make sure you understand the algorithm before you implement it.

- Create an empty stack.
- Get the starting cell using the methods of the supplied class *Map* (description of this class below).
- Push the starting cell into the stack and mark the cell as *inStack*. You will use methods of the class *MapCell* to mark a cell.
- ***While*** the stack is not empty *and* the destination has not been found perform the following steps:
    - Peek at the top of the stack to get the current cell.
    - If the current cell is the destination, then the algorithm exits the loop.
    - If any of the neighbouring cells to the current one has a tower, then pop the top cell from the stack and mark the popped cell as *outOfStack*.
    - Otherwise, find the best unmarked neighbouring cell (use method *nextCell* from class *ComputePath* to do this). If the cell exists, push it into the stack and then mark it as i*nStack*; otherwise, there are no unmarked neighbouring cells, so pop the top cell from the stack and mark it as *outOfStack*.

Your program must print a message indicating whether the destination was reached or not. If a path was found the algorithm must also print the number of cells in the path from the initial cell to the destination. Notice that your algorithm does not need to find the shortest path from the starting cell to the destination.

4. Command Line Arguments

Your program **must** read the name of the input map file from the command line. You can run the program with the following command:

```
java ComputePath name_of_map_file
```

where `name_of_map_file` is the name of the file containing the city map. You can use the following code to verify that the program was invoked with the correct number of arguments:

```
public class ComputePath {
        public static void main (String[] args) {
                try{
                        if(args.length < 1)
                                throw new IllegalArgumentException(
                                "Provide the name of the file with the input map");
                        String mapFileName = args[0];
                        ...
```

To get Eclipse to supply a command line argument to your program open the "Run -> Run Configurations..." menu item. Make it sure that the "Java Application->ComputePath" is the active selection on the left-hand side. Select the "Arguments" tab. Enter the filename for the map in the "Program arguments" text box.

5. Classes Provided

You can download from the course's webpage several java classes that allow your program to display the map on the screen. You are encouraged to study the given code so you learn how it works. Below is a description of some of these classes. Full documentation for the code provided is in the course's website.

5.1 Class *Map*.java

This class represents the map of the city over which the drone will fly. The methods that you might use from this class are the following:

- *public Map (String inputFile) throws InvalidMapException, FileNotFoundException, IOException.* This method reads the input file and displays the map on the screen. An *InvalidMapException* is thrown if the *inputFile* has the wrong format.
- *public MapCell getStart().* Returns a *MapCell* object representing the cell where the UWO store is located.

5.2 Class *MapCell*.java

This class represents the cells of the map. Objects of this class are created inside the class *Map* when its constructor reads the map file. The methods that you might use form this class are the following:

- *public MapCell neighbourCell (int i) throws InvalidNeighbourIndexException*. As explained above, each cell of the map has up to six neighbouring cells, indexed from 0 to 5. This method returns either a *MapCell* object representing the *i*-th neighbor of the current cell or *null* if such a neighbor does not exist. Remember that if a cell has fewer than 6 neighbouring cells, these neighbours do not necessarily need to appear at consecutive index values. So, it might be, for example, that *this.getNeighbour*(0) and *this.getNeighbour*(3*)* are null, but *this.getNeighbour*(*i*) for all other values of *i* are not null.
  An *InvalidNeighbourIndexException* exception must be thrown if the value of the parameter *i* is negative or larger than 5.

- *public boolean* methods: *isTower*(), *isFree*(), *isNoFlying*(), *isThief*(), *isHighAltitude*(), *isInitial(), isCustomer*(), return true if *this MapCell* object represents a cell of type tower, free, no-flying, thief, high-altitude, the initial cell where the UWO sore is, or the destination cell where the customer house is, respectively.
- *public boolean isMarked*() returns true if *this MapCell* object represents a cell that has been marked as *inStack* or *outOfStack*.
- *public void markInStack*() marks *this MapCell* object as *inStack*.
- *public void markOutStack*() marks *this MapCell* object as *outOfStack*.

6. Image Files and Sample Input Files Provided

You are given several image files that are used by the provided java code to display the different kinds of map cells on the screen. You are also given several input map files that you can use to test your program. In Eclipse put all these files inside your project file in the same folder where the src folder is. ***Do not*** put them inside the src folder as Eclipse will not find them there. If your program does not display the map correctly on your monitor, you might need to move these files to another folder, depending on how your installation of Eclipse has been configured.

7. Extra Credit (5% Bonus)

Note that the algorithm described above is guaranteed to find a way to reach the destination if there is a way to do so by following the specifications of the assignment. However, the algorithm might not find a path from the initial cell to the destination that goes only through free cells, even if such a path exists. To get the extra bonus marks you need to design and implement an algorithm that either

- finds a way to go from the initial cell to the destination by going only through free cells if such a path exists, or if this path does not exist, then
- it finds a path from the initial cell to the destination that goes only through free and high-altitude cells if such a path exists, or if this path does not exist then
- it finds a path from the initial cell to the destination that goes through free, high-altitude, or thief's cells, if such a path exists.

None of the above paths can go through a tower cell or a cell that is adjacent to a tower.

8. Submission

Submit all your .java files to OWL. If you implement the extra credit you must submit only one version of the ComputePath.java class. If you submit two versions of this class the TA's will mark only one of them. **Do not** put the code inline in the textbox. **Do not** submit your *.class* files. If you do this and do not submit your *.java* files you will receive a mark of zero. **Do not** submit a compressed file with your java classes (.zip, .rar, .gzip, …).

9. Non-functional Specifications

1. **Assignments are to be done individually and must be your own work.** Software will be used to detect cheating.
2. Include comments in your code in javadoc format. Add javadoc comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add javadoc comments to the methods and instance variables. Read information about javadoc in the second lab for this course.
3. Add comments to explain the meaning of potentially confusing and/or major parts of your code.
4. Use Java coding conventions and good programming techniques. **Read the notes about comments, coding conventions and good programming techniques in the first assignment**.

10. What You Will Be Marked On

1. Functional specifications:
    - Does the program behave according to specifications? Does it run with the test input files provided? Are your classes created properly? Are you using appropriate data structures? Is the output according to specifications?
2. Non-functional specifications: as described above