CS1027 Assignment 3

Due: March 22, 11:55pm.
Weight: 9%

Purpose

To gain experience with

- The solution of problems through the use of priority queues
- The design of algorithms in pseudocode and their implementation in Java.

1. Task

In this assignment you will design and implement a program in Java to find a path for the drone from the last assignment that it can use to go from the UWO store to the customer's house. However, this time your program is required to find a shortest possible path, if one exists. Your program **must** use a priority queue, as described below. To make things simpler for you, for this assignment you will not need to consider high altitude or thief's cells (except for the bonus part as described below).

For this assignment, there will be four types of cells (these are the same as in Assignment 2):

- Free cells. The drone can safely fly across one of these cells to move to an adjacent one.
- Tower cells. Each one of these cells contains a cellular tower. The drone cannot go through these cells or through any cell adjacent to a tower cell.
- No-flying cells. The drone cannot enter them.
- Customer cell. This is the destination. The customer cell is a free cell.

Initially the drone is positioned in the free cell where the UWO store is located. The following figure shows an example of a map in which the shortest path from the initial cell to the customer cell goes through cells 1, 2, 5, 7, 9, and 11 and has length 6. Note that there might be other paths of the same length, like 1, 2, 5, 6, 9, 11; your algorithm just needs to find one of the paths of shortest length. There might also be other, longer paths, like 1, 2, 3, 5, 7, 9, 11 which your algorithm will not select. Paths that go through tower cells or through cells adjacent to tower cells must not be selected by your algorithm.

2. Classes to Implement

2.1 DLPriorityQueue

A priority queue is an ADT that stores a collection of data items in which each data item has a priority. For this assignment the priorities of the data items are going to be values of type *double*. The operations provided by this ADT are the following: *enqueue*, *dequeue*, *getSmallest*, *changePriority*, *isEmpty*, *numItems*, and *toString*. The operations are described below. You are provided with a Java interface for the priority queue ADT called *PriorityQueueADT.java*. Your class *DLPriorityQueue.java* must implement all the methods in the *PriorityQueueADT.java* interface and it must store the data items of the priority queue in a **doubly linked list**. The header for this class will then be

public class DLPriorityQueue<T> implements PriorityQueueADT<T>

This class will have three private instance variables:

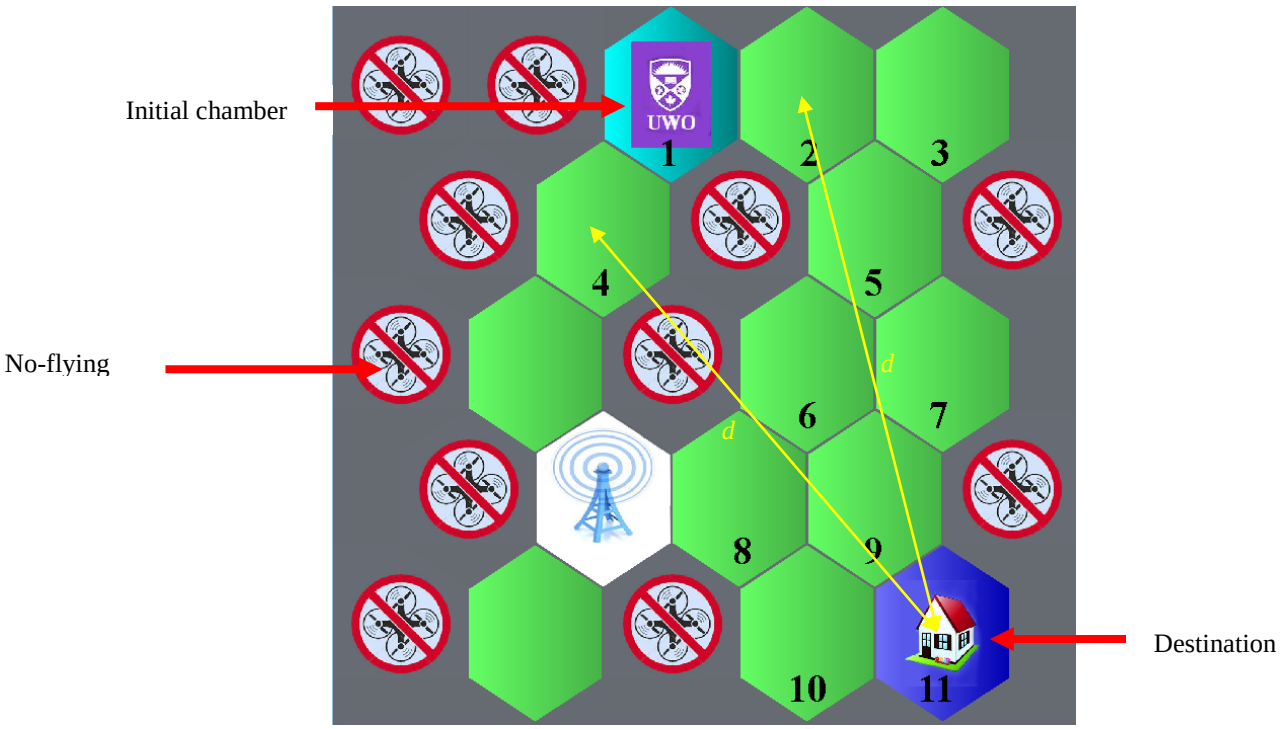

Figure 1.

- *private PriorityNode<T> front*. This is a reference to the first node of the doubly linked list. Class *PriorityNode* is described in the next section.
- *private PriorityNode<T> rear*. This is a reference to the last node of the doubly linked list.
- *private int count*. The value of this variable is the number of data items in the priority queue.

This class needs to implement the following methods.

- *public DLPriorityQueue()*. Creates an empty priority queue.
- *public void enqueue (T dataItem, double priority)*. Adds to the priority queue the given *dataItem* with its associated priority. The new data item must be added to the rear of the doubly linked list.
- *public T dequeue() throws EmptyPriorityQueueException*. Removes and returns the data item at the front of the priority queue.
- *public double getPriority(T dataItem) throws InvalidDataItemException*. Returns the priority of the specified *dataItem*. An *InvalidDataItemException* in thrown if the given *dataItem* is not in the priority queue.

  Note that to check whether *dataItem* is in the priority queue you need to scan the linked list using linear search. To check if the data item stored in a node of the linked list is the same as *dataItem* you must use the equals method, not the "==" operator.
- *public void changePriority (T dataItem, double newPriority) throws InvalidDataItemException*. Changes the priority of the given element to the new value. An *InvalidDataItemException* is thrown if the given *dataItem* is not in the priority queue. Read note in *getPriority* method.
- *public T getSmallest() throws EmptyPriorityQueueException*. Removes and returns the data item in the priority queue with smallest priority. If several items in the priority queue have the

same smallest priority, any one of them is returned. An *EmptyPriorityQueueException* is thrown if the priority queue is empty.

Note that to find the data item with smallest priority the entire linked list needs to be scanned using linear search.

- *public boolean isEmpty()*. Returns *true* if the priority queue is empty and it returns *false* otherwise.
- *public int numItems().* Returns the number of data items in the priority queue.
- *String toString().* Returns a *String* representation of the priority queue. This method will just invoke the *toString* method from each data item in the priority queue and concatenate these strings.

2.2 PriorityNode

This class represents the nodes of the doubly linked list used to implement the priority queue. This class will be declared as follows:

public class PriorityNode<T>

This class will have four instance variables:

- *private T dataItem*. A reference to the data item stored in this node.
- *private PriorityNode<T> next*. A reference to the next node in the liked list.
- *private PriorityNode<T> previous*. A reference to the previous node in the linked list.
- *private double priority*. This is the priority of the data item stored in this node.

You need to implement the following methods in this class:

- *public PriorityNode (T data, double prio).* Creates a node storing the given data and priority.
- *public PriorityNode().* Creates an empty node, with null data and priority zero.
- Setter and getter methods: *getPriority, getDataItem, getNext, getPrevious, setDataItem, setNext, setPrevious, setPriority.*

2.3 FindShortestPath

In this class you will implement an algorithm to compute a shortest path from the UWO store cell to the customer cell. A description of this algorithm is given below in Section 3. This class will contain the main method:

public static void main (String[] *args)*

This method will first create an object of the class *Map* passing *args*[0] as the parameter to the constructor; *args*[0] should store the name of the file containing the city map. As in the previous assignment, when your program creates an object of the class *Map*, the city map will be displayed on the screen. Description of the class *Map* is given below.

Your program must print the number of cells in the path from the initial cell to the customer cell. For example, for the map Figure 1 the algorithm must print a message indicating that the shortest path has length 6. If there is no path from the initial cell to the destination, your program must print an appropriate message. If you want, you might add other private methods and/or instance variables to any of the required classes.

Important. Your program must catch any exceptions that might be thrown. Any invocation to a method that throws an exception must be inside a try-catch block. For each exception thrown an

appropriate message must be printed. The message must explain what caused the exception to be thrown instead of just a generic message saying that an exception was thrown**.**

3. Algorithm for Finding a Shortest Path

The algorithm starts at the cell with the UWO store and as it traverses the map, it will keep in the priority queue the map cells that it might visit next. Each map cell *C* has a priority equal to the distance from *C* to the UWO store plus the Euclidean or straight-line distance from *C* to the customer cell. This priority is an estimation of the length of the shortest path from the UWO store to the customer's house that passes through *C* and it will be used by the algorithm to try to find the required shortest path as explained below.

As the algorithm looks for the path to the customer cell it will keep track of the distance from the current cell *C* to the UWO store. However, the algorithm will not yet know the distance from *C* to the customer cell as the algorithm has not yet reached the customer. This is the reason why the algorithm uses the Euclidean distance from *C* to the customer cell to estimate the length of the path from *C* to the destination. The algorithm explores these possible paths in increasing order of length until it finds the shortest one to the customer's place.

A description of the algorithm is given below:

- First, create an empty priority queue.
- Get the initial cell where the UWO store is by using method *getUWOstore* from class *Map*. Each map cell is represented with an object of class *MapCell*, described below.
- Add the initial cell to the priority queue with a priority of zero. Mark this cell as *enqueued* (use method *markEnqueued* from class *MapCell*).
- *W**hile*** the priority queue is not empty, ***and*** the customer cell has not been reached, perform the following steps:

    - Remove the cell *S* with smallest priority from the priority queue and mark it as *dequeued* (use method *markDequeued* from class *MapCell*).
    - If *S* is the customer cell, then the algorithm exits the while loop.
    - If *S* has a cellular tower in it or if any of the neighbouring cells has a tower, then *S* cannot be part of the solution; go back to the beginning of the ***while*** loop to try a different path.

      Otherwise, consider each one of the neighbouring cells of *S* that is not *null*, is not of type *no-flying*, and has not been marked as *dequeued*. For each one of these neighbouring cells *C* perform the following steps:

        - Set *D* = 1 + distance from *S* to the initial cell.
        - If the distance between *C* and the initial cell is larger than *D* then:
            - set the distance of *C* to the initial cell to *D* (to do this use methods *getDistanceToStart* and *setDistanceToStart* from class *MapCell*).
            - Set *S* as the predecessor of *C* in the path to the UWO store (use method *setPredecessor* from class *MapCell*)*;* this is necessary to allow the algorithm to reconstruct the path from the initial cell to the destination once the destination has been reached.
        - Set *P* = distance from *C* to the initial cell + Euclidean distance from *C* to the destination cell. The Euclidean distance to the destination can be computed using

method *euclideanDistToDest* from the *MapCell* class. The distance to the initial cell is obtained by invoking method *getDistanceToStart* from the *MapCell* class.

- If *C* is marked as *enqueued* and *P* is less than the priority of *C* then use the *changePriority* method from class *DLPriorityQueue* to update the priority of *C* to *P*. You can use method *getPriority* from class *DLPriorityQueue* to get the priority of *C*.
- If *C* is not marked as *enqueued*, then add it to the priority queue with priority equal to its distance to the initial cell plus the Euclidean distance to the destination. Then mark *C* as *enqueued*.

Note. Recall that you are expected to use meaningful names for your variables when implementing this algorithm. So, do not use *C, P* and *S* as names for your variables; use more meaningful names. You can use some of the methods that you implemented for assignment 2, if you want.

Consider, for example the map given in Figure 1. The drone starts at cell 1 and it sets the distance from this cell to the initial cell to 0. Then it examines neighboring cells 2 and 4. For each of these cells the distance to the initial cell is set to 1 and the predecessor of each one of these cells is set to cell 1. Since the Euclidean distance between two cells is the length of the straight line connecting the centers of the cells, the drone will assign the same priority, 1 + *d*, to cells 2 and 4, where *d* is the Euclidean distance between cells 2 and 11 or between cells 4 and 11 (which is the same; see the figure).

Next the drone examines the cell with the smallest priority, say 2 (as cells 2 and 4 have the same priority). From here it will examine the neighboring cells to cell 2, namely 3 and 5. The distances from 3 and 5 to the initial cell are set to 2; cell 2 is set as the predecessor of cells 3 and 5. The priority of cell 5 will be set to a lower value than the priority of cell 3 as the Euclidean distance from 5 to the destination is smaller than the Euclidean distance from cell 3 to the destination. The drone keeps examining cells, avoiding those containing a tower and those adjacent to a tower, until it reaches the destination or it determines that the destination cannot be reached. We will post a demo showing how the algorithm works.

4. Classes Provided

You are given several java classes. Some of these classes are the same as those for the previous assignment, but some of them have been modified. Read carefully the descriptions of the classes. Full documentation for the code provided is in the course's website.

- **Class *Map***. This class represents the city map. The methods that you might use from this class are the following:
  - *public Map (String inputFile) throws InvalidMapException, IOException, FileNotFoundException.* Reads the input map file and displays the map on the screen. An *InvalidMapException* is thrown when *inputFile* does not contain a valid map. Look at the sample input files to learn their format.
  - *public MapCell getUWOstore().* Returns a *MapCell* object representing the initial cell.
  - *public int numCells().* Returns the number of cells in this map. You might need this method only if you decide to implement the optional part for the bonus marks, as explained below.
- **Class *MapCell***. This class represents a map cell. Objects of this class are created inside class *Map* when the map file is read. The methods that you might use from this class are the following:

- *public MapCell getNeighbour (int i) throws InvalidNeighbourIndexException*. Each map cell has up to six neighbouring cells, indexed from 0 to 5. For each value for the index *i*, from 0 to 5, the method might return either a *MapCell* object representing a cell or *null*. Note 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 is thrown if the value of the parameter *i* is negative or larger than 5.
- *public boolean* methods: *isTower*(), *isFree*(), *isNoFlying*(), *isCustomer*(), return true if ***this*** *MapCell* object represents a cell of type tower, free, no-flying, or customer, respectively. There is also method *isThief*() if you want to implement the bonus part.
- *public boolean isMarkedEnqueued*() returns true if ***this*** *MapCell* object represents a cell that has been marked as *enqueued*.
- *public boolean isMarkedDequeued*() returns true if ***this*** *MapCell* object represents a cell that has been marked as *dequeued*.
- *public void markEnqueued*() marks ***this*** *MapCell* object as *enqueued*.
- *public void markDequeued*() marks ***this*** *MapCell* object as *dequeued*.
- *public int euclideanDistToDest(Map city).* Returns the Euclidean distance from the cell represented by ***this*** *MapCell* object to the destination. The parameter is the *Map* object containing ***this*** *MapCell* object.
- *public int getDistanceToStart*(). Returns the distance from the cell represented by ***this*** *MapCell* object to the initial cell.
- *public void setDistanceToStart(int dist)*. Sets the distance from the cell represented by ***this*** *MapCell* object to the initial cell to the specified value.
- *public void setPredecessor(MapCell pred).* Sets the predecessor of ***this*** *MapCell* object to the specified value.
- *public Boolean equals(MapCell otherCell).* Returns true if *otherCell* points to ***this*** *MapCell* object; otherwise it returns false.

- **Exception Classes:** *EmptyPriorityQueueException, InvalidNeighbourIndexException, InvalidDataItemException, InvalidMapException.*

5. Image Files and Sample Input Files Provided

You are given several image files that are used by the provided java code to display the map 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 directory where the default package and the JRE System Library are. **Do not** put them in 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.

6. Extra Credit (5% Bonus Marks)

For the extra credit your algorithm must find a shortest path from the initial cell to the destination when the map can also contain thief cells:

- If there is a way to go from the initial cell to the destination that uses only free cells, then your algorithm must find a shortest path from the initial cell to the destination that goes only through free cells.
- Otherwise, your algorithm must find a path form the initial cell to the destination that uses the **least** number of thief's cells. If the minimum number of thief's cells in such a path is, say, *k* then the path computed by your algorithm must use the smallest number of free cells among all paths from the initial cell to the destination that use *k* thief's cells.

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

For example, for the map shown Figure 2, every path to the destination needs to go through at least one thief cell. There are several paths that use one thief's cell; two of these paths have 5 free cells and the others have either 6, 7, or 8 free cells. Hence, the algorithm must select a path with one thief's cell and 5 free cells, for example the path that goes through cells 1, 2, 3, 4, 5, and 6. Note that the path 1, 7, 8, 5, 6 cannot be selected by your algorithm even though it has only 5 cells because it includes 2 thief's cells.

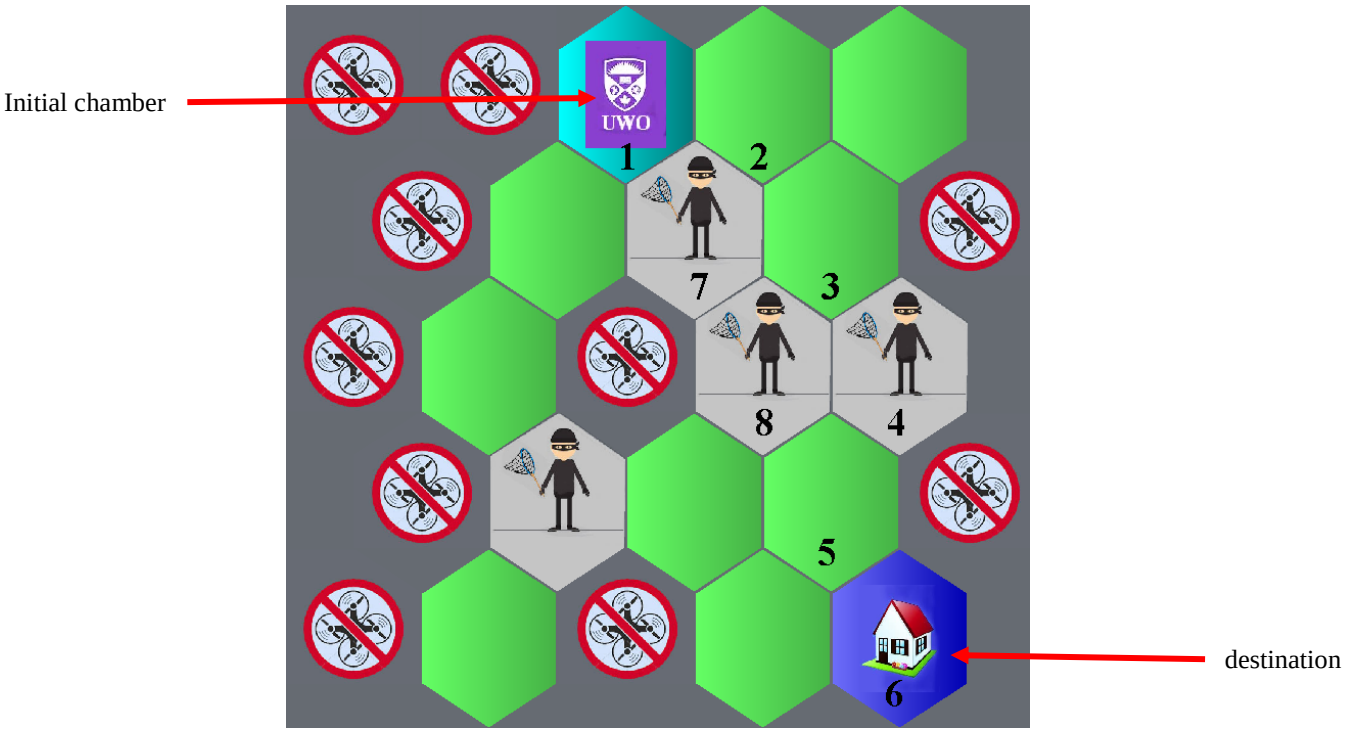

Figure 2

7. 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 giving a brief description of the class. Add javadoc comments to 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.

Submit all your .java files to OWL. **DO NOT** put the code inline in the textbox. Do not submit a compressed file (.zip, .rar, .gzip, …) with your code. Do not submit your *.class* files. If you do this, and do not attach your *.java* files, you will receive a mark of zero!

8. 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
3. Assignment submission: via OWL assignment submission.