UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

Task 1
Implementation of the software artifact field

Garrido Pozo, Enrique
Mora Herreros, Pablo
Ollero Jiménez, Adrián

Subject: Intelligent Systems

Group: 3ºA

Team: A2

Date: 10/10/17

INDEX

Previous considerations

We thought about the language we are going to use and we doubted about using Java, C or python. All in all, we have used Java because we know the use of classes and their relations better than other languages.

In order to print the new distributions, we thought to write the new position of the tractor and their new distributions with their adjacent points. However, we changed our idea and now, we print the new distributions using North, East, West and South.

To solve the problem, we considered using a recursive method implementing 3 nested fors or an iterative problem. Firstly, we used 3 fors, but it was so difficult to make the distributions. For that reason, we changed of idea and we made a recursive algorithm.

Finally, if the input which is introduced doesn't have the same format as problem's statement, a message of error will be executed and the program will stop. We assume that k, max, columns and rows will have 1 as minimum value.

Files

Action.java

This file is a class in which we store the coordinates of the next action. This class is in charge of all operations related to the treatment of the new movement. Apart from the *constructor* method, there is other essential method called *perform()*. This method consists on moving the sand before changing the tractor of a new position. After moving all the amount of sand and check if it can be moved to the north, east, west or south, the coordinates of the tractor will change.

```
public void perform(Field field) {

    if(sandMovement[1]!=0) {//Al norte
        field.setNumber(field.getXt()-1, field.getYt(), field.getNumber(field.getXt()-1, field.getYt())+sandMovement[1]);
        field.setNumber(field.getXt(), field.getYt(), field.getNumber(field.getXt(), field.getYt())-sandMovement[1]);
    }
    if(sandMovement[2]!=0) { //Al oeste
        field.setNumber(field.getXt(), field.getYt()-1, field.getNumber(field.getXt(), field.getYt()-1)+sandMovement[2]);
        field.setNumber(field.getXt(), field.getYt(), field.getNumber(field.getXt(), field.getYt())-sandMovement[2]);
    }
    if(sandMovement[3]!=0) { //Al este
        field.setNumber(field.getXt(), field.getYt()+1,field.getNumber(field.getXt(), field.getYt()+1)+ sandMovement[3]);
        field.setNumber(field.getXt(), field.getYt(), field.getNumber(field.getXt(), field.getYt())-sandMovement[3]);
    }
    if(sandMovement[4]!=0) { //Al sur
        field.setNumber(field.getXt()+1, field.getYt(), field.getNumber(field.getXt()+1, field.getYt())+sandMovement[4]);
        field.setNumber(field.getXt(), field.getYt(), field.getNumber(field.getXt(), field.getYt())-sandMovement[4]);
    }

    field.setXt(xt);
    field.setYt(yt);
}
```

perform method

Using the *toString()* method, we print the new position of the tractor and after, the sand we would move. In our program, the positions of the array represent: Current Position, North, West, East and South.

An example:

```
-----ACTIONS-----
Tractor: (0, 1) Sand: [8, 0, 0, 0, 0]
Tractor: (0, 1) Sand: [7, 1, 0, 0, 0]
Tractor: (0, 1) Sand: [7, 0, 1, 0, 0]
Tractor: (0, 1) Sand: [7, 0, 0, 1, 0]
Tractor: (0, 1) Sand: [7, 0, 0, 0, 1]
Tractor: (0, 1) Sand: [6, 2, 0, 0, 0]
```

Field.java

In this class, we store all the variables related to the information that the file gives to us. For that reason, we have introduced a method to read the file and to generate the solution. We thought about to introducing another class to read and write, but finally, we decided to do these methods in “Field class”. If these methods need to be changed in following steps, it would be done in following deliveries.

These are the variables we need to read from the file:

```
private int [][] field;
private int xt, yt, k, max, sizec, sizer;
```

All of them have their setter and getter to manage the data.

Method *readField()* is in charge of reading the file which is introduced as an input. If the format of the file is not the same as the example, the program gives you an error. Also, any error related with “maximum”, “k”, “rows”, “columns” is controlled.

This method calls another two methods called *readFirstLine()* and *generateValuesField()*. *readFirstLine()* is in charge of saving the variables of the first line from the file. The method *generateValuesField()* copies all the values in the matrix.

To finish, the *generateOuput()* has been implemented to store the current output in a file. We store in a variable called “nextMov” all the new variables that we store in the new distribution. We store in another variable called “matrix” to save the new distributions in the matrix.

mainClass.java

This class is the one in charge of executing the program. First of all, it shows all the attributes from the file. After, it looks for the adjacent points and distributes to all positions the current position. After that, we will applicate a new action in the coordinates and we store in the new matrix.

The most important method in this class is the *successor()*. This method calls other important methods such as *checkPositions(), printAdjacent(), moveSand(), removeMax(), createActions()* and *printActions()*. In future versions we will reduce the amount of methods. The main functionality of this program is to find all successors of the state.

```
private static ArrayList<Action> successor(Field field) {
    ArrayList <int[]> adjacentPositions, sandMovements;
    ArrayList <Action> actions =  new ArrayList<Action>();
    sandMovements= new ArrayList<int[]>();
    checkPositions(field);
    adjacentPositions=moveTractor(field);
    printAdjacent(adjacentPositions, field);
    moveSand(field, sandMovements);
    ////////////////////////////
    removeMax(sandMovements, field);
    ////////////////////////////
    createActions(adjacentPositions, sandMovements, actions);
    printActions(actions);
    return actions;
}
```

successor method

removeMax() is the method in charge of removing the distributions when the amount of sand is bigger than the maximum. When is the maximum, it stops there.

moveSand() is in charge of sharing the sand in all positions and *printAdjacent()* prints the positions in which you can move the sand in the coordinates.

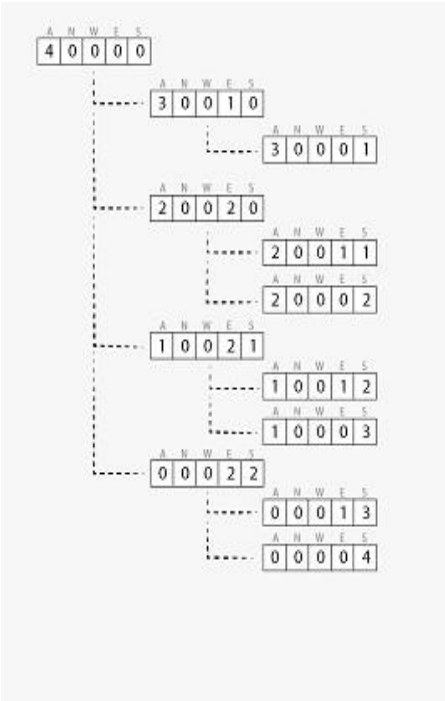
printActions() prints all the possible actions in which the tractor and sand can move.

checkPositions() checks if you can move to the north, the west, the east or the south.

Examples

We will do with this example:
0 0 5 8 2 2
4 6
3 8

Graphic:



```
-----ATTRIBUTES-----
Xt: 1
Yt: 1
K: 5
Max: 10
Size column: 3
Size Row: 3
-----MATRIX-----
5 2 5
5 9 3
4 1 10
-----ADJACENT POSITIONS-----
0 1----->2
1 0----->5
1 2----->3
2 1----->1

-----ACTIONS-----      A  N  W  E  S
Tractor: (0, 1) Sand: [0, 4, 0, 0, 0]
Tractor: (0, 1) Sand: [0, 3, 1, 0, 0]
Tractor: (0, 1) Sand: [0, 3, 0, 1, 0]
Tractor: (0, 1) Sand: [0, 3, 0, 0, 1]
Tractor: (0, 1) Sand: [0, 2, 2, 0, 0]
Tractor: (0, 1) Sand: [0, 2, 1, 1, 0]
Tractor: (0, 1) Sand: [0, 2, 1, 0, 1]
Tractor: (0, 1) Sand: [0, 2, 0, 2, 0]
Tractor: (0, 1) Sand: [0, 2, 0, 1, 1]
Tractor: (0, 1) Sand: [0, 2, 0, 0, 2]
Tractor: (0, 1) Sand: [0, 1, 3, 0, 0]
Tractor: (0, 1) Sand: [0, 1, 2, 1, 0]
Tractor: (0, 1) Sand: [0, 1, 2, 0, 1]
Tractor: (0, 1) Sand: [0, 1, 1, 2, 0]
Tractor: (0, 1) Sand: [0, 1, 1, 1, 1]
Tractor: (0, 1) Sand: [0, 1, 1, 0, 2]
Tractor: (0, 1) Sand: [0, 1, 0, 3, 0]
Tractor: (0, 1) Sand: [0, 1, 0, 2, 1]
Tractor: (0, 1) Sand: [0, 1, 0, 1, 2]
Tractor: (0, 1) Sand: [0, 1, 0, 0, 3]
Tractor: (0, 1) Sand: [0, 0, 4, 0, 0]
Tractor: (0, 1) Sand: [0, 0, 3, 1, 0]
Tractor: (0, 1) Sand: [0, 0, 3, 0, 1]
Tractor: (0, 1) Sand: [0, 0, 2, 2, 0]
Tractor: (0, 1) Sand: [0, 0, 2, 1, 1]
Tractor: (0, 1) Sand: [0, 0, 2, 0, 2]
Tractor: (0, 1) Sand: [0, 0, 1, 3, 0]
Tractor: (0, 1) Sand: [0, 0, 1, 2, 1]
Tractor: (0, 1) Sand: [0, 0, 1, 1, 2]
Tractor: (0, 1) Sand: [0, 0, 1, 0, 3]
Tractor: (0, 1) Sand: [0, 0, 0, 4, 0]
Tractor: (0, 1) Sand: [0, 0, 0, 3, 1]
Tractor: (0, 1) Sand: [0, 0, 0, 2, 2]
Tractor: (0, 1) Sand: [0, 0, 0, 1, 3]
Tractor: (0, 1) Sand: [0, 0, 0, 0, 4]
```

```
Tractor: (1, 0) Sand: [0, 4, 0, 0, 0]
Tractor: (1, 0) Sand: [0, 3, 1, 0, 0]
Tractor: (1, 0) Sand: [0, 3, 0, 1, 0]
Tractor: (1, 0) Sand: [0, 3, 0, 0, 1]
Tractor: (1, 0) Sand: [0, 2, 2, 0, 0]
Tractor: (1, 0) Sand: [0, 2, 1, 1, 0]
Tractor: (1, 0) Sand: [0, 2, 1, 0, 1]
Tractor: (1, 0) Sand: [0, 2, 0, 2, 0]
Tractor: (1, 0) Sand: [0, 2, 0, 1, 1]
Tractor: (1, 0) Sand: [0, 2, 0, 0, 2]
Tractor: (1, 0) Sand: [0, 1, 3, 0, 0]
Tractor: (1, 0) Sand: [0, 1, 2, 1, 0]
Tractor: (1, 0) Sand: [0, 1, 2, 0, 1]
Tractor: (1, 0) Sand: [0, 1, 1, 2, 0]
Tractor: (1, 0) Sand: [0, 1, 1, 1, 1]
Tractor: (1, 0) Sand: [0, 1, 1, 0, 2]
Tractor: (1, 0) Sand: [0, 1, 0, 3, 0]
Tractor: (1, 0) Sand: [0, 1, 0, 2, 1]
Tractor: (1, 0) Sand: [0, 1, 0, 1, 2]
Tractor: (1, 0) Sand: [0, 1, 0, 0, 3]
Tractor: (1, 0) Sand: [0, 0, 4, 0, 0]
Tractor: (1, 0) Sand: [0, 0, 3, 1, 0]
Tractor: (1, 0) Sand: [0, 0, 3, 0, 1]
Tractor: (1, 0) Sand: [0, 0, 2, 2, 0]
Tractor: (1, 0) Sand: [0, 0, 2, 1, 1]
Tractor: (1, 0) Sand: [0, 0, 2, 0, 2]
Tractor: (1, 0) Sand: [0, 0, 1, 3, 0]
Tractor: (1, 0) Sand: [0, 0, 1, 2, 1]
Tractor: (1, 0) Sand: [0, 0, 1, 1, 2]
Tractor: (1, 0) Sand: [0, 0, 1, 0, 3]
Tractor: (1, 0) Sand: [0, 0, 0, 4, 0]
Tractor: (1, 0) Sand: [0, 0, 0, 3, 1]
Tractor: (1, 0) Sand: [0, 0, 0, 2, 2]
Tractor: (1, 0) Sand: [0, 0, 0, 1, 3]
Tractor: (1, 0) Sand: [0, 0, 0, 0, 4]
Tractor: (1, 2) Sand: [0, 4, 0, 0, 0]
Tractor: (1, 2) Sand: [0, 3, 1, 0, 0]
Tractor: (1, 2) Sand: [0, 3, 0, 1, 0]
Tractor: (1, 2) Sand: [0, 3, 0, 0, 1]
Tractor: (1, 2) Sand: [0, 2, 2, 0, 0]
Tractor: (1, 2) Sand: [0, 2, 1, 1, 0]
Tractor: (1, 2) Sand: [0, 2, 1, 0, 1]
Tractor: (1, 2) Sand: [0, 2, 0, 2, 0]
Tractor: (1, 2) Sand: [0, 2, 0, 1, 1]
Tractor: (1, 2) Sand: [0, 2, 0, 0, 2]
Tractor: (1, 2) Sand: [0, 1, 3, 0, 0]
Tractor: (1, 2) Sand: [0, 1, 2, 1, 0]
Tractor: (1, 2) Sand: [0, 1, 2, 0, 1]
Tractor: (1, 2) Sand: [0, 1, 1, 2, 0]
Tractor: (1, 2) Sand: [0, 1, 1, 1, 1]
Tractor: (1, 2) Sand: [0, 1, 1, 0, 2]
Tractor: (1, 2) Sand: [0, 1, 0, 3, 0]
Tractor: (1, 2) Sand: [0, 1, 0, 2, 1]
Tractor: (1, 2) Sand: [0, 1, 0, 1, 2]
Tractor: (1, 2) Sand: [0, 1, 0, 0, 3]
```

```
Tractor: (1, 2) Sand: [0, 0, 4, 0, 0]
Tractor: (1, 2) Sand: [0, 0, 3, 1, 0]
Tractor: (1, 2) Sand: [0, 0, 3, 0, 1]
Tractor: (1, 2) Sand: [0, 0, 2, 2, 0]
Tractor: (1, 2) Sand: [0, 0, 2, 1, 1]
Tractor: (1, 2) Sand: [0, 0, 2, 0, 2]
Tractor: (1, 2) Sand: [0, 0, 1, 3, 0]
Tractor: (1, 2) Sand: [0, 0, 1, 2, 1]
Tractor: (1, 2) Sand: [0, 0, 1, 1, 2]
Tractor: (1, 2) Sand: [0, 0, 1, 0, 3]
Tractor: (1, 2) Sand: [0, 0, 0, 4, 0]
Tractor: (1, 2) Sand: [0, 0, 0, 3, 1]
Tractor: (1, 2) Sand: [0, 0, 0, 2, 2]
Tractor: (1, 2) Sand: [0, 0, 0, 1, 3]
Tractor: (1, 2) Sand: [0, 0, 0, 0, 4]
Tractor: (2, 1) Sand: [0, 4, 0, 0, 0]
Tractor: (2, 1) Sand: [0, 3, 1, 0, 0]
Tractor: (2, 1) Sand: [0, 3, 0, 1, 0]
Tractor: (2, 1) Sand: [0, 3, 0, 0, 1]
Tractor: (2, 1) Sand: [0, 2, 2, 0, 0]
Tractor: (2, 1) Sand: [0, 2, 1, 1, 0]
Tractor: (2, 1) Sand: [0, 2, 1, 0, 1]
Tractor: (2, 1) Sand: [0, 2, 0, 2, 0]
Tractor: (2, 1) Sand: [0, 2, 0, 1, 1]
Tractor: (2, 1) Sand: [0, 2, 0, 0, 2]
Tractor: (2, 1) Sand: [0, 1, 3, 0, 0]
Tractor: (2, 1) Sand: [0, 1, 2, 1, 0]
Tractor: (2, 1) Sand: [0, 1, 2, 0, 1]
Tractor: (2, 1) Sand: [0, 1, 1, 2, 0]
Tractor: (2, 1) Sand: [0, 1, 1, 1, 1]
Tractor: (2, 1) Sand: [0, 1, 1, 0, 2]
Tractor: (2, 1) Sand: [0, 1, 0, 3, 0]
Tractor: (2, 1) Sand: [0, 1, 0, 2, 1]
Tractor: (2, 1) Sand: [0, 1, 0, 1, 2]
Tractor: (2, 1) Sand: [0, 1, 0, 0, 3]
Tractor: (2, 1) Sand: [0, 0, 4, 0, 0]
Tractor: (2, 1) Sand: [0, 0, 3, 1, 0]
Tractor: (2, 1) Sand: [0, 0, 3, 0, 1]
Tractor: (2, 1) Sand: [0, 0, 2, 2, 0]
Tractor: (2, 1) Sand: [0, 0, 2, 1, 1]
Tractor: (2, 1) Sand: [0, 0, 2, 0, 2]
Tractor: (2, 1) Sand: [0, 0, 1, 3, 0]
Tractor: (2, 1) Sand: [0, 0, 1, 2, 1]
Tractor: (2, 1) Sand: [0, 0, 1, 1, 2]
Tractor: (2, 1) Sand: [0, 0, 1, 0, 3]
Tractor: (2, 1) Sand: [0, 0, 0, 4, 0]
Tractor: (2, 1) Sand: [0, 0, 0, 3, 1]
Tractor: (2, 1) Sand: [0, 0, 0, 2, 2]
Tractor: (2, 1) Sand: [0, 0, 0, 1, 3]
Tractor: (2, 1) Sand: [0, 0, 0, 0, 4]

--------PERMFORM AN ACTION (Example randomly chosen)------
 Performing the action: Tractor: (0, 1) Sand: [0, 2, 0, 1, 1]
 Xt: 0
 Yt: 1
 K: 5
 Max: 10
 Size column: 3
 Size Row: 3
 -----MATRIX-----
5 4 5
5 5 4
4 2 10
```

```
-----ATTRIBUTES-----
Xt: 1
Yt: 2
K: 3
Max: 5
Size column: 3
Size Row: 3
-----MATRIX-----
4 1 5
3 0 5
4 0 5
-----ADJACENT POSITIONS-----
0 2----->5
1 1----->0
2 2----->5

-----ACTIONS-----      A  N  W  E  S
Tractor: (0, 2) Sand: [0, 0, 2, 0, 0]
Tractor: (1, 1) Sand: [0, 0, 2, 0, 0]
Tractor: (2, 2) Sand: [0, 0, 2, 0, 0]

--------PERMFORM AN ACTION (Example randomly chosen)------
 Performing the action: Tractor: (0, 2) Sand: [0, 0, 2, 0, 0]
 Xt: 0
 Yt: 2
 K: 3
 Max: 5
 Size column: 3
 Size Row: 3
 -----MATRIX-----
4 1 5
3 2 3
4 0 5
-----ATTRIBUTES-----
Xt: 0
Yt: 0
K: 3
Max: 5
Size column: 3
Size Row: 3
-----MATRIX-----
2 5 5
5 3 2
2 2 1
-----ADJACENT POSITIONS-----
0 1----->5
1 0----->5

-----ACTIONS-----      A  N  W  E  S
Tractor: (0, 1) Sand: [0, 0, 0, 0, 0]
Tractor: (1, 0) Sand: [0, 0, 0, 0, 0]

--------PERMFORM AN ACTION (Example randomly chosen)------
 Performing the action: Tractor: (1, 0) Sand: [0, 0, 0, 0, 0]
 Xt: 1
 Yt: 0
 K: 3
 Max: 5
 Size column: 3
 Size Row: 3
 -----MATRIX-----
2 5 5
5 3 2
2 2 1
```

MILESTONE 2

New classes we have implemented

Frontier.java

We create the class frontier because we need it to manage both data structures. We use linked list and priority queues to make some tests because they are indexed and they allow to be sorted automatically (in their implementation). After that we have seen that queue is a better option than linked list because its time is the smallest in all tests (details at the end of the document).

In this class, the methods that are asked in the task have been implemented. We use getFrontier, setFrontier, createFrontier, insertNode, removeFirst and isEmpty. Most of these methods are implemented with functions that data structures include in their implementation.

Node.java

We create the class node where all attributes about the nodes are defined. Their attributes are state, cost, action, depth, value and parent.

- The state is of the type Field, and that is the state the node represents.
- The cost is an int which determines how expensive is to reach to that node
- The action is an Action, and represents the action that has modified the parent state to obtain the state in this node
- The depth is also an int and determines how deep the node is from the initial state
- The value, which is the one that will specify the order in which the nodes can be expanded using an informed algorithm
- The parent is a reference to another node, which is actually the parent of this node

Also, we implement the method toString to print all the values of the node. Finally, we implement the method compareTo, that we use explicitly in the queue and in the Linkedlist. This method compare one node with another node to order them by value in increasing order.

```
public int compareTo(Node other) {
    if(this.getValue()<other.getValue()){
        return -1;
    }
    if(this.getValue()>other.getValue()){
        return 1;
    }
    return 0;
}
```

Field.java

Is the same that in the previous milestone, but we have implemented another two new methods: isGoal, compareField.

The function isGoal is in charge of comparing the current state with the goal state returning true if so, and false otherwise.
The function compareField compares the field with the one passed as parameter. This is to control that a successor of a node is not the same as the parent of that node.

```java
/**********************************************************************************************
 * Method name: isGoal
 * Description: Compare both matrix to know if the matrix is goal or not
 * @param field -> one matrix to compare with the other
 * @return true -> if the field is goal it returns true
 **********************************************************************************************/
public static boolean isGoal(Field field) {
    int [][] aux = field.getField();
    for(int i =0; i< aux.length; i++)
        for(int j =0; j < aux[0].length; j++)
            if(aux[i][j] != field.getK()) return false;
    return true;
}
/**********************************************************************************************
 * Method name: compareField
 * Description: it compares the parent node with the other matrix to prove that the parent
 * @param parent -> The matrix that we have to prove that is the father
 * @return true -> if the matrix is the parent of the other child is true
 **********************************************************************************************/
public boolean compareField(Field parent) {
    for(int i = 0; i < sizer; i++)
        for(int j = 0; j < sizec; j++)
            if(parent.getField()[i][j] != field[i][j]) return false;
    return true;
}
```

Actions.java

About this class, is the same that the one in the previous milestone. We didn't add any new methods.

Successor.java

This class is our mainClass in the previous milestone. Now it has been changed to work as an object, so minor changes has been carried out. Every node uses a successor object to obtain all its successors.

mainClass.java

This class is actually the class that performs the execution of everything, from the creation of successors, to the tests of the data structures. It will not be included in the final delivery.

Times of execution: Data Structures

To make the tests we use two data structures: Linkedlist and priority queues.
We use that because they are indexed and they allow to be sorted automatically (which means that both of them have a function implemented that sort all elements stored in the structure).

The best time is in all cases is in the priority queue. We think that the best time is priority queue because it introduces the nodes in order with the method compareTo. On the other hand, the Linkedlist introduce the elements in the creation order and after that it is sorted when all elements are introduced.

Also, priority queues allow to store 35 million of nodes and the Linkedlist allow to store less than 30 million of nodes.
Our output:

```
Time of list: 661435ms
TIme of queue: 478985ms
Better? Priority Queue
```

OUTPUT

With this example:

```
1 2 3 5 3 3
 3 3 3
 3 2 5
 3 3 3
```

We obtain this output:
It's too large the output, but in this picture we show you the beginning of the solution.

```
├-------------------- LIST ------------------
------- Parent -----
Node:
3 3 3
3 2 5
3 3 3
(1, 2)
Cost: 0
Depth: 0
Value : 0
---- SUCCESSOR ----
Node:
3 3 5
3 2 3
3 3 3
(0, 2)
Cost: 1
Depth: 1
Value : 1
---- SUCCESSOR ----
Node:
3 3 3
3 3 3
3 3 4
(1, 1)
Cost: 1
Depth: 1
Value : 8
---- SUCCESSOR ----
Node:
3 3 3
3 2 3
3 3 5
(0, 2)
Cost: 1
```

(and so on… ordered by increasing value)