

Laboratory assignment

Subtask III



LABORATORY GROUP: A1 – 07

COMPONENTS:

AMANDA SÁNCHEZ GARCÍA

FERNANDO VELASCO ALBA

GITHUB REPOSITORY: A1-07

15/11/2017

#### INDEX

### sUBTASK i

### eXPLANATION OF THE PROBLEM

### IMPLEMENTATION OF THE PROBLEM

### PROGRAMMING LANGUAGE CHOSEN

### CLASSES IMPLEMENTED

### MAIN METHODS

### TESTING

### SUBTASK ii

### CLASSES Introduced

### METHODS introduced

### DATA STRUCTURES USED

### SubTask III

### methods introduced

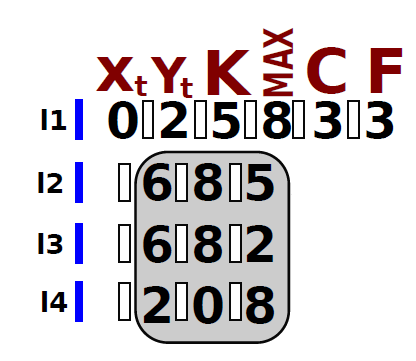
#### SUBTASK i

### eXPLANATION OF THE PROBLEM

The main goal of this laboratory assignment consists of defining, designing and developing an agent program to find the sequence of actions to be taken by a tractor to ensure that all the sand in a field is evenly distributed on the ground. So, all the boxes will have an equal amount of sand K.

The first things to be done are:

* Implement an internal representation of the field.
* Create a field.
* Reading and writing a field from/to a file.
* Generate all possible actions from a field with the tractor in the (Xy, Yt) box.
* Get a new field after applying an action to a given one.

It is necessary to take into account the format of the file where the provided information is going to be:

When we have all this implemented, we need to create two data structures: the node and the frontier of the tree.

In the node, we need:

* Access to the parent
* State
* Cost
* Action
* Depth
* Value (a random value)

We need an ordered list to create the frontier of the tree, where the nodes are sorted from lower to higher value of “Value”.

With all this, we need to define the state space of the problem, the initial state and a goal function.

I.2 OUR IMPLEMENTATION OF THE PROBLEM

1.2.1 PROGRAMMING LANGUAGE CHOSEN

The programming language that we are going to use is Java. We have decided to use it because it is the programming language that we know best, also Java is a very complete language so we will have available all the data structures we are going to need.

### 1.2.2 CLASSES IMPLEMENTED

### The field is going to be represented as a bidimensional array which boundaries are defined through the file. The file defines the position (x,y) of the tractor in the field, the desired quantity of sand in each square (k), the maximum sand that can be placed in a square, and the number of columns and rows of the field.

### We have implemented six classes:

### **State.** Class where the current state is defined. That is, the field is defined with its number of rows and columns, and the maximum and allowed sand in each square. It is also defined the position of the tractor (X, Y) and the current sand.

### In this class are found the methods to move the tractor and sand along the field, and generate the successors of the tree.

### **FileHandler.** This class is used to manage the file, that is, to have the possibility to read it and write on it. It checks that the file is read correctly, taking into account the format, throwing the corresponding errors.

### **InputExceptions.** Class to check that the format of the file is the correct one (checking that there are only positive integers, the correct use of blank spaces or other error).

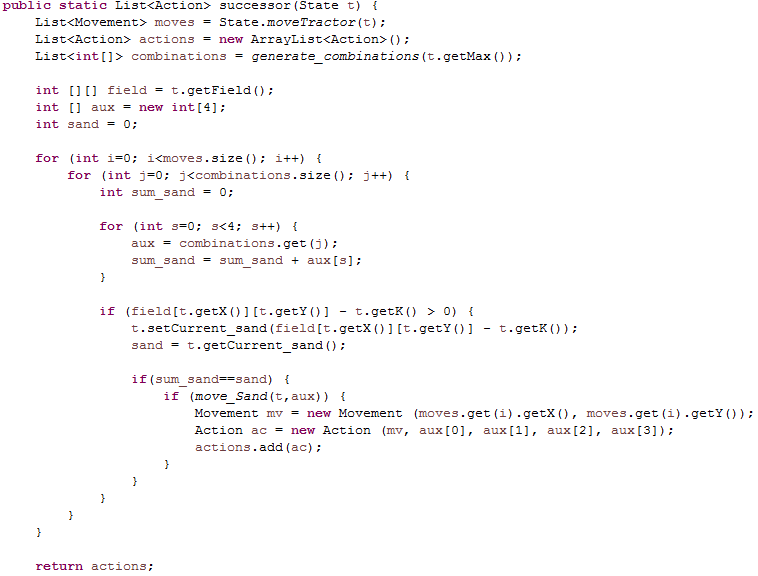
### **Action.** Class where an action is defined. The main attributes of this class are the next movement of the tractor and the quantity of sand to be redistributed in each possible move (north, east, west and south).

### **Movement.** Where every movement of the tractor is stored.

### **Main.** Main class of the program. Here the field is created and the file is read, and invokes methods of other classes.

1.2.3 MAIN METHODS

* succesor



The method *successor* returns all the possible actions that can be executed from a given position. In order to generate them, we have to consider the possible combinations of sand, this is, the amount of sand that can be distributed to north, south, east and west.

Once we have these amounts, we have to generate all the possible actions for all the possible movements, which have been determined previously.

The sand we have to take is the current sand in a given position minus the desired amount of sand in each position.

If this quantity is bigger than zero, then we update the field, and, if it is possible to move sand to all the adjacent positions, we create a new Action.

1.2.4 testing

Now, we are going to prove that the actions are generated correctly with two different examples.

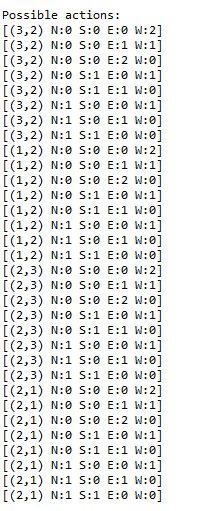
In the first one, the initial position is (2,2) and the field is:

8 8 8 8

3 7 7 3

0 0 7 0

0 8 7 6

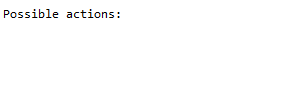


In the second example, the initial position is (0,2) and the field is:

6 8 5

6 8 2

2 0 8



As the amount of sand in this position is equal to the desired amount of sand (K), there are no possible actions.

#### SUBTASK II

#### For this assignment, we have to include the class *Node*, which must contain all the elements needed to work with the tree search. These elements are: access to the parent, a state, the cost, the depth and the value.

#### We have to include a class *Problem* too. In this class, we are going to find the state space, the initial state and a method to check if a state is the solution we are looking for.

#### In addition, we have to create the *Frontier.* In order to select the most suitable data structure, we are going to compare two structures. The elements in the frontier will be sorted from lower to higher *value* of the node*.* Initially, this value will be generated randomly.

#### 

#### ClASSES INTRODUCED

### **Node.** Here every node of the tree is defined. Every node has cost, depth, value and action, and they have associated a state and a reference to the parent.

### **Problem.** Main class of the program. Here the field is created and the file is read, and invokes methods of other classes. Also, it creates the queue of the tree and apply every action.

### METHODS INTRODUCED

* frontier queue

### https://i.gyazo.com/4111fac11d1ff0a9eedc9addd67ef06c.png

This method is located in the Problem class. A structure called frontier is created to be used in the tree. The nodes are placed in the frontier in ascending order of their value (which is randomly generated). In this method it is also computed the time needed to insert every node in the frontier (maximum, minimum and average time).

There is another method similar to this one but implements the frontier as a list instead of a queue.

### IS goal

### https://i.gyazo.com/45f01e280450c699ec79721c3ff54b3e.png

Method that also belongs to the Problem class. Checks whether the node is a goal state or not (true or false).

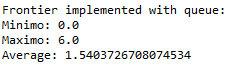
### https://i.gyazo.com/fa786c9ca348cc29496c4f4f4cd69b31.pngapply action

Another method from the Problem class, in which every action of the state space is applied. That is, moving the quantity of sand allowed (looking at the restrictions of the problem) to the North, South, East or West.

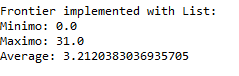
### DATA STRUCTURES USED

The data structures we have used to implement the frontier are a *Priority Queue* and a *List.*

The times for inserting all the nodes into the priority queue are:



The time for inserting the same nodes into the List are:



#### SUBTASK iii

#### In this subtask, we have to program a basic version of the search algorithm with the following strategies:

#### Breadth-First Search

#### Depth-First Search, Depth-Limited Search and Iterative Deepening Search

#### Uniform Cost

#### A method *Cost* must be defined in the class State, which is going to define the cost of every node. The method should have an action as an argument and it have to return the cost. In order to compute the cost for the node, we have to take into account that the cost is going to be considered as the total amount of sand moved in the action plus 1.

#### METHODS INTRODUCED

### BOUNDED sEARCH

### https://i.gyazo.com/9661ff93e031764e749130200b83a50d.png

#### The method BoundedSearch is the implementation of the search algorithm. This method is going to be used for any of the strategies mention before, as the only difference between strategies is where the node is inserted in the frontier. In this case, the position in where the node is inserted is not relevant because the frontier is sorted depending on the node value.

#### The method basically generates the successors for the node which is in the head of the frontier while the frontier is not empty and there is no solution.

#### If a solution is found, we return a list of nodes generated with the method CreateSolution.

#### https://i.gyazo.com/65c3dc60fa9f4ba402973bbaab3c236b.png

### CREATE node list

### https://i.gyazo.com/9c6816074de2799a9003ad1302c2c18f.png

### The method CreateNodeList creates a list containing the nodes created after applying each action contained in the list of actions. The method ApplyAction returns a new state which is included in the new node. The cost of the node depends on the amount of sand moved in the action. It is computed in the method *Cost*, located in the class State. The depth of the new node is the depth of the node introduced to the method plus 1 and the parent is the node introduced.

### The value of the node is going to depend on the strategy chosen. The method SelectValueNode is going to assign the correct value to the node.

### https://i.gyazo.com/455efeea9b6b019bfd0c4137986f647f.png

### 