Universidad de Castilla-La Mancha

Escuela Superior de Informática

Intelligent Systems

3º Ingeniería Informática

Intelligent Systems Lab Project

Team A2
Adrián Ollero Jiménez
Enrique Garrido Pozo
Pablo Mora Herreros

Date:
November 29, 2017

Adrián Ollero Jiménez
Enrique Garrido Pozo
Pablo Mora Herreros

Contents

1 Milestone 1: Implementation of the software artifact field

1.1 Previous Considerations

We thought about the language we were 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 any 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 does not 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.

1.2 Files

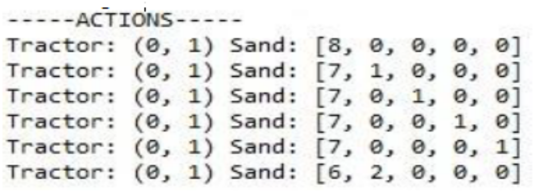
1.2.1 Action.java

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) {//To North
   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) { //To West
   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) { //To East
    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) { //To South
    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);
}//End perform
```

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]
```

1.2.2 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;
```

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.

1.2.3 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.

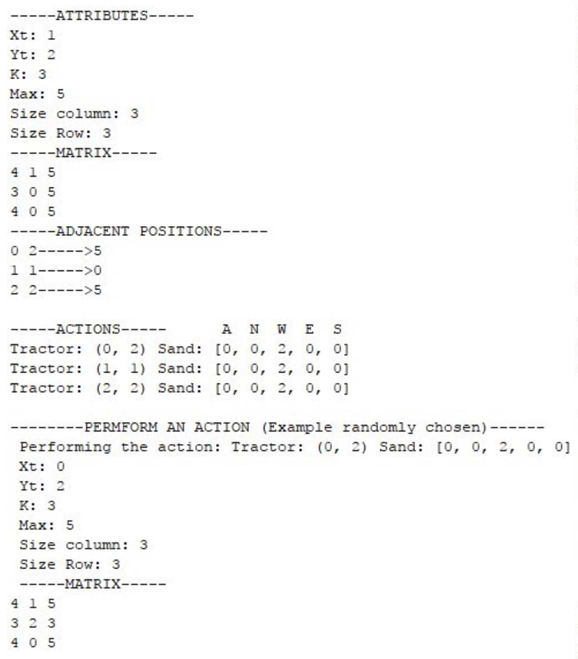
The most important method in this class is the *successor()*. This method calls other important methods such as *checkPositions()*, *printAdjacent()*, *moveSand()*, *removeMax()*, *createActions()* and *printActions()*. In future versions we will reduce the amount of methods. The main functionality of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

1.3 Example

This is an example of the input:

```
1 2 3 5 3 3
 4 1 5
 3 0 5
 4 0 5
```

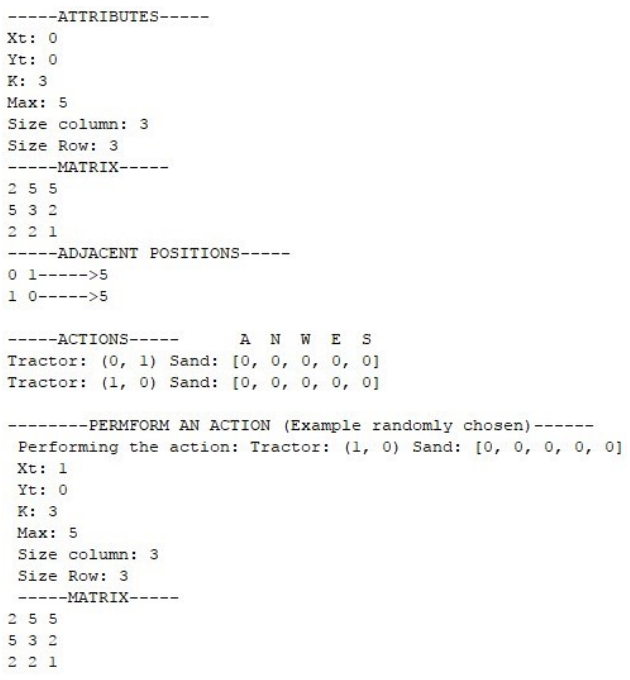
Here there is a screen-shot of the output of two examples:



```
-----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
```

2 Milestones 2: Definition of the problem and implementation of the frontier

2.1 New classes

2.1.1 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 (detailed at the end of section 2).

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.

2.1.2 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 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 uninformed algorithm
- The parent is a reference to another node, which is actually the parent of this node

Furthermore, 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;
    }
```

2.1.3 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, even this is not a requirement of this milestone.

```
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;
}
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;
}
```

2.1.4 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.

2.2 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 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.

Execution times of both structures:

```
----------- TIME -----------
Time of list: 510237ms
Time of queue: 311713ms
Better? Priority Queue
```

Complete output:

```
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 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
----
Node:
3 3 5
3 2 3
```

```
3 3 3
(2, 2)
Cost: 1
Depth: 1
Value : 6
[...]
---------------------- QUEUE -----------------
------- Parent -----
Node:
3 3 3
3 2 5
3 3 3
(1, 2)
Cost: 0
Depth: 0
Value : 0
----
Node:
3 3 5
3 2 3
3 3 3
(2, 2)
Cost: 1
Depth: 1
Value : 6
[...]
----------- TIME -----------
Time of list: 510237ms
Time of queue: 311713ms
Better? Priority Queue
```

3 Milestone 3: Basic version of the search algorithm

3.1 Classes for uninformed search

In other to implement the search algorithms some new classes has been created and some others has been modified.

3.2 Created classes

3.2.1 Interface.java

This is the class that now contains the *main* method. When the program is executed, the path of the setup file is asked to the user by the commands line and checked whether the file is found or not. Then, the desired strategy is specified as well as the maximum depth that the tree can reach. In case the selected strategy is *IDS (Iterative Deepening Search)*, the increment between iterations is needed.

After that, once all this information has been introduced, the algorithm starts the search algorithm defined in *UninformedSearch.java* file.

Once the algorithms has found the solution, it is written on a file called *"Output.txt"*.

3.2.2 UninformedSearch.java

It contains the search algorithm given in the statement of the milestone.

First, we can find the *search* algorithm, which is in charge of defining the solution list and starting the *boundedSearch* method, which result will be stored in the solution list.

```
public ArrayList<Node> search(Problem prob, Strategy strategy, int
                                      prof_max, int inc_prof)
{
  int currentProf = inc_prof;
  ArrayList<Node> solution = new ArrayList<Node>();
  while (currentProf <= prof_max){
        solution = boundedSearch(prob, strategy, 0);
        currentProf = prof_max + inc_prof;
        }
  return solution;
}
```

The *boundedSearch* method will look for a solution, obtaining a list with all the nodes that composes the final result. For that, the successors of the current node are computed and introduced on the frontier following the strategy selected, because the order criteria changes depending on the aproach.

At then end, once the solution has been found, the *createSolution()* method is called and traverses all the nodes from the goal to the initial, storing them on a list which is returned.

```
public static ArrayList<Node> boundedSearch(Problem prob,Strategy
    strategy,int currentProf){
  Frontier frontier = new Frontier();
  Node initial_node=new Node(prob.getInitState());
  Node current_node = null;
  frontier.createFrontier();
  frontier.insertNode(initial_node);
  boolean isSolution = false;
  while(!isSolution && !frontier.isEmpty()){
        current_node=frontier.removeFirst();
        if(current_node.getState().isGoal()){
          isSolution = true;
        }else {
          Successor successors =  new Successor();
          ArrayList<Node> suc = successors.successors(current_node,
              strategy);
          for(int i = 0; i < suc.size(); i++) {
            frontier.insertNode(suc.get(i));
          }
        }
  }
  if(isSolution)
        return createSolution(current_node);
  else return null;
}
```

3.2.3 Problem.java

In this class we could obtain the initial state and the goal state with the methods *getGoalState* and *getInitState*.

3.2.4 Strategy.java

This class inherits from the *Enum* class. We use it to make easier the way to call the type of strategy that the user wants to use.

```
public enum Strategy {
  BFS, DFS, DLS, IDS, UCS
}
```

3.3 Updated classes

3.3.1 Node.java

In this class we have modified one of the constructors. There are two of them, one for the initial node, which does not have a parent, and other for the child nodes, which uses a state and a actions that is applied to that state and calculates the value, for the insertion on the frontier, depending on the strategy chosen.

This second constructor has been modified from one on the previous milestone, where the value were a random number. Now, this value is defined as following:

```
public Node(Node parentNode, Action nextAction, Strategy strategy)
   {
  this.state =  new Field(parentNode.getState(), nextAction);
  this.action = nextAction;
  this.parent = parentNode;
  this.cost = parentNode.getCost()+action.getActionCost()+1;
  this.depth = parentNode.getDepth()+1;
  if(strategy == Strategy.BFS)
        this.value = depth;
        else if(strategy == Strategy.DFS || strategy == Strategy.
           DLS || strategy == Strategy.IDS)
        this.value = -depth;
        else if(strategy == Strategy.UCS)
          this.value = cost;
}
```

3.3.2 Action.java

We have added the new method called *getActionCost* that calculates the current cost of the action.

3.4 Deleted classes

3.4.1 Main.java

This class has been deleted and all its content has been moved to the *Interface* class because in this milestone we have created the menu where the user can interact.

3.5 Examples

Beginning with this *Setup.txt* file:

```
0 0 5 8 2 2
 3 7
 6 4
```

The program generates the following *Output.txt*:

```
1- Tractor: (0, 1) Sand: [0, 0, 0, 0, 0]
3 7
6 4

2- Tractor: (1, 1) Sand: [0, 0, 0, 0, 2]
3 5
6 6

3- Tractor: (1, 0) Sand: [0, 0, 1, 0, 0]
3 5
7 5

4- Tractor: (1, 1) Sand: [0, 2, 0, 0, 0]
5 5
5 5
```

This solution has been obtained using the BFS or UCS strategies because when using any depth-based strategies the program gets stuck due to the loops.

3.6 New documentation

We have decided to improve our documentation to get a professional view using LaTeX. It also allows us to add code and images easier and a clear format. This decision has been motivated by the lack of organization on a word document.

4 Milestone 4: A* and pruning

4.1 A* implementation

In order to include this new strategy the only thing that needs to be added is one more condition in the Node constructor where, if the strategy chosen is the new one, A*, then the value of the node is the cost plus the heuristic. So the condition inside the constructor is now the following:

```
if(strategy == Strategy.BFS)
  this.value = depth;
else if(strategy == Strategy.DFS || strategy == Strategy.DLS ||
    strategy == Strategy.IDS)
  this.value = -depth;
else if(strategy == Strategy.UCS)
  this.value = cost;
else if(strategy == Strategy.A_START)
  this.value = cost + state.h();
```

The heuristic is a function on the field class, where the number of incorrect cells is computed. We consider an incorrect cell as the one which has a different amount of sand than the desired one (k). This is calculated by traversing the whole field and counting the incorrect cells. This is the function:

```
public int h() {
  int h=0;
  for(int i=0; i<field.length; i++)
    for(int j=0; j<field[i].length; j++)
          if(field[i][j]!=k) h++;
  return h;
}
```

The rest of the algorithm is already implemented as we are using the search algorithm of the previous milestone.

4.2 Pruning

The implementation of the pruning has been done by using a structure called *visited*, which is actually a Hash table. We have decided to use this structure as it uses indexing to retrieve and organize the different elements stored on it. Then, when a new node is going to be added into the frontier, and the optimization is active (the user decided whether using or not optimization), it is checked to see if it is in the *visited* collection or not. If it is a new node it is added to the frontier and to the *visited* collection. If the node is already in *visited*, then the value is checked: if the new node has a lower value than the one stored in *visited* then it is changed, if not, the node is ignore.

The way of storing a node in *visited* is the following:

- Key (Node): the node is stored as an string. There is a function in Node that serialize the node in a string. The format is "@field@XY".
- Value (value or cost): depending on the strategy chosen the value stored is a different value. If the strategy is breath or depth search, then the value store is the cost of that node, while if the strategy is an informed search, then the value stored is the value of the node.

This is the function in charge of checking whether the node is in *visited* or not, and if it must be added or ignored:

```
private static boolean checkVisited(Node node, Strategy strategy) {
  String serial = node.serialize();
  if(!visited.containsKey(serial)) {
    visited.put(serial, node.getValueHash(strategy));
        return true;
  }else {
        if(visited.get(serial) > node.getValueHash(strategy)) {
      visited.remove(serial);
          visited.put(serial, node.getValueHash(strategy));
          return true;
  }else {
    /*Do nothing*/
    return false;}
  }
}
```

The function *getValueHash()* is in the node class and returns the corresponding value depending on the strategy chosen.

Firstly we tried to make a comparison between a Hash table and a dictionary, but when using the Dictionary class a warning showed saying that this implementation is obsolete and that we should use Hash tables instead, which actually extends the Dictionary class.

4.3 Other minor changes

There are some minor changes in the rest of classes aimed at approaching the inclusion of the pruning task and A* search. This changes are related to asking for the new strategy or the use of optimization when executing the interface.

4.4 Tests

Using the file given by the professor this are the numbers obtained for each of the strategies and both, using and not using the optimization feature.

Note: for all test we have selected that the maximum depth is 10 and the increment for IDS is 2.

4.4.1 Setup file

This is the initial field:

```
1 1 1 4 3 3
 3 3 2
 0 0 0
 0 1 0
```

4.4.2 Output example

This is just an example of the file obtained using, for example, BFS:

```
Strategy: BFS
n- (X, Y)[N, W, E, S]
1- (0, 1)[0, 0, 0, 0]
2- (0, 0)[0, 0, 2, 0]
3- (0, 1)[0, 0, 1, 1]
4- (0, 2)[0, 0, 0, 1]
5- (0, 1)[0, 3, 0, 0]
6- (1, 1)[0, 0, 0, 3]
7- (2, 1)[0, 0, 1, 2]
8- (2, 2)[0, 1, 1, 0]
Cost: 24
Time: 80.0ms
Depth: 8
Spatial Complexity: 4800 nodes generated
Optimization: Yes
```

4.4.3 No optimized

This are the numbers obtained when executing all strategies but choosing *No Optimization*.

	BFS	DFS	DLS	IDS	UCS	A*
Number of Actions	8	10	10	8	8	8
Cost	20	32	32	26	16	16
Time(ms)	2402	26	26	97	2862	54
Depth	8	10	10	8	8	8
Spatial Complexity (nodes)	1081971	7419	7419	57648	2421243	50884

4.4.4 Optimized

This are the numbers obtained when choosing the *Optimization* feature.

	BFS	DFS	DLS	IDS	UCS	A*
Number of Actions	8	10	10	8	8	8
Cost	24	30	30	26	16	16
Time(ms)	45	10	12	49	41	20
Depth	8	10	10	8	8	8
Spatial Complexity (nodes)	4800	642	642	6936	3919	1401

4.4.5 Conclusion of tests

As we can see the strategies works find as the result obtained in A* is the same than in UCS but the time spent in A* is lower that the one in UCS as well as the number of nodes. Also, the time and the number of nodes when using *Optimization* is lower for all the strategies than when not using it. With this, we can conclude saying that the search algorithms work fine.

4.5 Future implementation

Facing the future, where are implementing a graphical interface which is not ready yet, but we hope that it will be ready for the lab exams.