

# **Project Documentation**

Beka Bekerí -  
Álvaro Guerrero del Pozo  
Fernando Vallejo Banegas  
Course 2017/2018

# Index

1. Introduction
2. Decisions taken
3. User manual
4. Testing
5. Code

# 1-Description:

We have to develop a program to distribute sand in a given terrain, represented as an array. The responsibility for doing so is for a truck, whose position will be represented as a tuple (X,Y), inside the field. The objective is to reach an amount of sand in every cell. The constraints are:

- The truck moves sand from the cell it is.
- The truck can only move sand to the adjacent cells (N, S, E, W)
- There is a maximum of sand in every cell.
- The truck must not move sand if the sand of the current cell is less than the objective.

At the end of the execution, the program generates a List of possible actions, and uses one of them randomly to generate a new state.

## 2 - Decisions taken

- We have used java because our understanding of the language is greater than in the case of other languages(like python, c, c++...)
- Each successor has a reference to its father, which in the case of the root (initial state) points to null. So, the successors are stored in a tree, although we do not use java implementation for trees.
- The frontier has been implemented as a PriorityQueue. It provides us the necessary tools for our problem: we can add nodes in increasing order of their value (which is random for now) and, more important, we can extract the peek of the frontier with a complexity of  $O(1)$ . Later we will provide some experiments regarding times and amount of nodes it can store.
- For simplicity, the actions are stored in a List. It provides the necessary tools, add, and get actions.
- The cost is equal to the sum of sand moved to all directions (N, E, S, W) + 1.

### Language specific problems:

We want to reduce the amount of memory used, so, we changed the integers that don't need to be bigger than 127 from integers (int = 32 bits) to bytes (byte = 8 bits). The problem was that we cannot add bytes, we need to use a casting for that. So, we questioned if the casting would delay the overall execution of the program, and therefore, our attempt of improvement was going to fail. So we did the next test:

```
8
9      byte num1 = 10;
10     byte num2 = 20;
11     byte result;
12
13     double t1= System.currentTimeMillis();
14     for (long i=0; i<1000000000; i++) {
15         result =(byte) (num1 + num2);
16     }
17     double t2 = System.currentTimeMillis();
18     System.out.println("Byte cast time: "+(t2-t1));
19
20     int numero1=10;
21     int numero2=20;
22     int resultado;
23
24     t1= System.currentTimeMillis();
25     for (long i=0; i<1000000000; i++) {
26         resultado= numero1+numero2;
27     }
28     t2 = System.currentTimeMillis();
29     System.out.println("Integer: "+(t2-t1));
30
31 }
```

<terminated> Tests [Java Application] C:\Program Files\Java\jre1.8.0\_144\bin\javaw.  
Byte cast time: 354.0  
Integer: 348.0

<terminated> Tests [Java Application] C:\Program Files\Java\jre1.8.0\_144\bin\  
Byte cast time: 347.0  
Integer: 352.0

With this result, that shows that casting won't have any negative effect, we can now use bytes, reducing the memory used by some of the variables of every state by ¼.

### 3 - User manual

The program starts asking the user what he/she wants to do. For now, only two options are available: read data from a file, and exit from the program and terminate execution (**Image 1**). If the first option is chosen, the user will be asked to provide the name of the file (**Image 2**). If the file is found, and no other errors happen\*(e.g the truck is out of field bounds), the program will start its functioning (**Image 3**). If the goal is achieved, the user will be asked if he/she wants to save the final result in a file, and the name.

Referring to files, remember that there are two options of introducing the name: either the file is (or is going to be created while writing) in the parent directory of /src, and therefore we can write just the name of the file (e.g example.txt); or the absolute path is introduced (e.g C:\Users\Myself\Documents\myfile.txt).

No errors are controlled while writing. The user will just receive a generic message. For this reason is recommended to use the first way of introducing the file name, <name>.<extension>, so we avoid permission problems. Also, take into account that if the file with the given name already exists, it will be replaced.

```
Please, choose an option:  
1-Read the data from a file.  
2-Exit.
```

**Image 1**

```
Please, choose an option:  
1-Read the data from a file.  
2-Exit.  
1  
Introduce the name of the file (don't forget the extension) :
```

**Image 2**

```
Please, choose an option:  
1-Read the data from a file.  
2-Exit.  
1  
Introduce the name of the file (don't forget the extension) :  
field.txt  
Execution started. Please wait while our program looks for a solution.
```

**Image 3**

*\*In the case of errors while reading the file, the user may receive a message giving further information about the problem.*

## 4 - Testing

To select a data structure to implement the frontier we had to make some testings with several kinds of data structure, always taking into account that those data structures should offer a way to order the states by a key value, which for this testings is set to random.

The data structures we have chosen to test their performance is:

1. LinkedList
2. PriorityQueue
3. SortedSet

The first thing we need to test is the speed of each data structure inserting states, to do that we can define an algorithm in which the computer tests inserting different quantities of states.

The quantities for which the data structures are going to be tested are:

```
Integer[] optionsS = {100, 1000, 10000, 100000, 200000, 500000, 1000000, 2000000, 5000000, 10000000, 20000000, 50000000, 100000000, 200000000};
Integer[] optionsP = {100, 1000, 10000, 100000, 200000, 500000, 1000000, 2000000, 5000000, 10000000, 20000000, 50000000};
Integer[] optionsL = {100, 1000, 5000};
```

And the algorithm to insert the states is:

```
for(int i = 0; i < optionsS.length; i++){
    t1 = System.currentTimeMillis();
    set = new TreeSet<State>();
    for(int j = 0; j < optionsS[i]; j++){
        State state = new State();
        set.add(state);
    }

    t2 = System.currentTimeMillis();
    System.out.println("The time to instert " + optionsS[i] + " states in a SortedSet has been: " + (t2 - t1) + ".");
}
```

These are the results we obtained:

```
The time to insert 100 states in a LinkedList has been: 5.
The time to insert 1000 states in a LinkedList has been: 434.
The time to insert 5000 states in a LinkedList has been: 63662.
The time to instert 100 states in a SortedSet has been: 0.
The time to instert 1000 states in a SortedSet has been: 1.
The time to instert 10000 states in a SortedSet has been: 2.
The time to instert 100000 states in a SortedSet has been: 9.
The time to instert 200000 states in a SortedSet has been: 12.
The time to instert 500000 states in a SortedSet has been: 18.
The time to instert 1000000 states in a SortedSet has been: 48.
The time to instert 2000000 states in a SortedSet has been: 59.
The time to instert 5000000 states in a SortedSet has been: 149.
The time to instert 10000000 states in a SortedSet has been: 302.
The time to instert 20000000 states in a SortedSet has been: 651.
The time to instert 50000000 states in a SortedSet has been: 1656.
The time to instert 100000000 states in a SortedSet has been: 3355.
The time to instert 1000000000 states in a SortedSet has been: 34137.
The time to instert 100 states in a PriorityQueue has been: 1.
The time to instert 1000 states in a PriorityQueue has been: 0.
The time to instert 10000 states in a PriorityQueue has been: 3.
The time to instert 100000 states in a PriorityQueue has been: 12.
The time to instert 200000 states in a PriorityQueue has been: 9.
The time to instert 500000 states in a PriorityQueue has been: 20.
The time to instert 1000000 states in a PriorityQueue has been: 46.
The time to instert 2000000 states in a PriorityQueue has been: 126.
The time to instert 5000000 states in a PriorityQueue has been: 1615.
The time to instert 10000000 states in a PriorityQueue has been: 3779.
The time to instert 20000000 states in a PriorityQueue has been: 7381.
The time to instert 50000000 states in a PriorityQueue has been: 26848.
```

By looking at this results we can see that by far the best option are the SortedSets but this is just inserting States, the problem is that if we need to use several operation on our data structure we need to take care of the cost of the other operations, such as retrieving data, etc.

If we take a look at those instructions the cost of those in our choosen data structures we can see that PriorityQueues are the best option from those 3 to represent our frontier.

Once that the priorityQueue has been selected we have to implement the method compareTo:

```
public int compareTo(State newState){
    int r = 0;

    if(newState.getValue() > this.value){
        r = -1;
    }
    else if(newState.getValue() < this.value){
        r = +1;
    }

    return r;
}
```

## 5 - Code

Here we explain the most “difficult” parts of the code.

### 1. GenerateActions:

```
for(int n = 0; n <= state.getPosition(state.getTractorX(), state.getTractorY()) - mean; n++){
    for(int e = 0; e <= state.getPosition(state.getTractorX(), state.getTractorY()) - mean; e++){
        for(int s = 0; s <= state.getPosition(state.getTractorX(), state.getTractorY()) - mean; s++){
            for(int w = 0; w <= state.getPosition(state.getTractorX(), state.getTractorY()) - mean; w++){

                auxA[0] = n; //NORTH MOVED SAND
                auxA[1] = e; //EAST MOVED SAND
                auxA[2] = s; //SOUTH MOVED SAND
                auxA[3] = w; //WEST MOVED SAND

                if(n+e+s+w == state.getPosition(state.getTractorX(), state.getTractorY()) - mean){

                    if(!((n > 0) && (state.getTractorX() == 0)) &&
                       !((e > 0) && (state.getTractorY() == state.getField()[0].length - 1)) &&
                       !((s > 0) && (state.getTractorX() == state.getField().length - 1)) &&
                       !((w > 0) && (state.getTractorY() == 0))){

                        if(isPossibleSand(auxA, state)) {
                            for(int mov = 0; mov < movementList.size(); mov++){
                                Movement move = new Movement(movementList.get(mov).getNewX(),
                                                                movementList.get(mov).getNewY());
                                Action auxAction = new Action(move, n, e, s, w);
                                actionList.add(auxAction);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

This method is made to obtain all the combinations of the actions with a 4 nested for loops that try all the possibilities from 0 to the maximum quantity of sand that can be moved from that positions which is (PositionQuantity - Mean)

Once that a combination is obtained we have to check if doing that is possible or not, to do that with the 2 if condition, we check that it doesnt move sand out of the bounds and if the total quantity moved is less than the total amount of sand that can be moved in that position.

Then just if the combination is right we create an object of the class Action, and introduce it into the actionList that we are going to return in this method.



## 2. GenerateMovements:

```
private static List<Movement> generateMovements(State state) {
    List<Movement> movementList = new ArrayList<Movement>();

    if (state.getTractorX()-1 >= 0 ) {
        Movement north = new Movement(state.getTractorX()-1, state.getTractorY());
        movementList.add(north);
    }
    if (!(state.getTractorY()+1 >= state.getField().length)) {
        Movement east = new Movement(state.getTractorX(), state.getTractorY()+1);
        movementList.add(east);
    }
    if (!(state.getTractorX()+1 >= state.getField().length)){
        Movement south = new Movement(state.getTractorX()+1, state.getTractorY());
        movementList.add(south);
    }
    if (state.getTractorY()-1 >= 0) {
        Movement west = new Movement(state.getTractorX(), state.getTractorY()-1);
        movementList.add(west);
    }
    return movementList;
}
```

This method is made to obtain a list of Movements, a class that represents the next movement of the tractor, so to do that we need the actual state, which contains the actual position of the tractor and with that information we can obtain the 4 possible movements of the tractor (North, East, South, West), once that those movements are created, we add to the list and that list is returned to be mixed with actions.

## 3. ApplyAction:

```
public static NodeState applyAction(NodeState state, Action action){
    NodeState newState = new NodeState();

    newState.setField(state.getField());
    newState.setTractorX(action.getNewMove().getNewX());
    newState.setTractorY(action.getNewMove().getNewY());
    newState.setFather(state);
    newState.setAppliedAction(action);
    newState.setCost(state.getCost() + action.getSandN() + action.getSandE() + action.getSandS() + action.getSandW() + 1);
    int centric = state.getPosition(state.getTractorX(), state.getTractorY());
    centric = centric - action.getSandN() - action.getSandE() - action.getSandS() - action.getSandW();
    newState.setPosition(state.getTractorX(), state.getTractorY(), centric);

    if((action.getSandN()>0) && (state.getTractorX() - 1 >= 0)){
        int northValue = state.getPosition(state.getTractorX() - 1, state.getTractorY());
        if ((northValue + action.getSandN())<= max) {
            northValue = northValue + action.getSandN();
            newState.setPosition(state.getTractorX() -1, state.getTractorY(), northValue);
        }
    }
}
```

(NOTE: this method is not complete, there are another 3 if clauses that are equal than the one in the screenshot but for the rest of the directions.)

This method is made to once that we have the whole possible actions to be applied to a state start applying them. To do that we first have to create a new State which is going to be the combination of the initial state and the action applied, Then we have to set the variables as the ones of the initial state and then change the values of the fields that have been modified, these are the steps:

1. Set the values of the tractor in the new state as the coordinates that come in the Action attribute movement.
2. Rest the sand that has been moved from the position of the tractor.
3. Set that new value to the same position but in the new state.
4. Finally, repeating this step to the four directions, find the sum of the value of the north and the sand that has been moved to the north, and setting that value in that position but in the new state.



Uploaded to: <https://github.com/BekaBekeri/Inteligentes>