LABORATORY ASSIGNMENT

FINAL DOCUMENTATION

LABORATORY GROUP: A1 – 07
COMPONENTS:
AMANDA SÁNCHEZ GARCÍA
FERNANDO VELASCO ALBA
GITHUB REPOSITORY: A1-07
13/12/2017

INDEX

1. SUBTASK 1

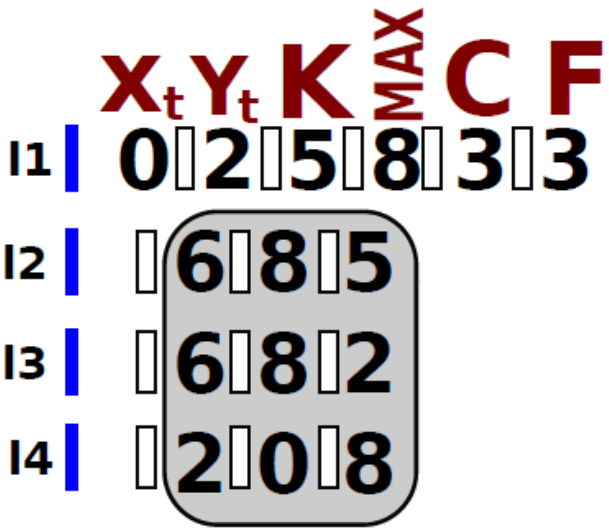
1.1 EXPLANATION OF THE PROBLEM

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 (X_y, Y_t) 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

```
public static List<Action> successor(State t) {
    List<Movement> moves = State.moveTractor(t);
    List<Action> actions = new ArrayList<Action>();
    List<int[]> combinations = generate_combinations(t.getMax());

    int [][] field = t.getField();
    int [] aux = new int[4];
    int sand = 0;

    for (int i=0; i<moves.size(); i++) {
        for (int j=0; j<combinations.size(); j++) {
            int sum_sand = 0;

            for (int s=0; s<4; s++) {
                aux = combinations.get(j);
                sum_sand = sum_sand + aux[s];
            }

            if (field[t.getX()][t.getY()] - t.getK() > 0) {
                t.setCurrent_sand(field[t.getX()][t.getY()] - t.getK());
                sand = t.getCurrent_sand();

                if(sum_sand==sand) {
                    if (move_Sand(t,aux)) {
                        Movement mv = new Movement (moves.get(i).getX(), moves.get(i).getY());
                        Action ac = new Action (mv, aux[0], aux[1], aux[2], aux[3]);
                        actions.add(ac);
                    }
                }
            }
        }
    }

    return actions;
```

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
```

```
Possible actions:
[(3,2) N:0 S:0 E:0 W:2]
[(3,2) N:0 S:0 E:1 W:1]
[(3,2) N:0 S:0 E:2 W:0]
[(3,2) N:0 S:1 E:0 W:1]
[(3,2) N:0 S:1 E:1 W:0]
[(3,2) N:1 S:0 E:0 W:1]
[(3,2) N:1 S:0 E:1 W:0]
[(3,2) N:1 S:1 E:0 W:0]
[(1,2) N:0 S:0 E:0 W:2]
[(1,2) N:0 S:0 E:1 W:1]
[(1,2) N:0 S:0 E:2 W:0]
[(1,2) N:0 S:1 E:0 W:1]
[(1,2) N:0 S:1 E:1 W:0]
[(1,2) N:1 S:0 E:0 W:1]
[(1,2) N:1 S:0 E:1 W:0]
[(1,2) N:1 S:1 E:0 W:0]
[(2,3) N:0 S:0 E:0 W:2]
[(2,3) N:0 S:0 E:1 W:1]
[(2,3) N:0 S:0 E:2 W:0]
[(2,3) N:0 S:1 E:0 W:1]
[(2,3) N:0 S:1 E:1 W:0]
[(2,3) N:1 S:0 E:0 W:1]
[(2,3) N:1 S:0 E:1 W:0]
[(2,3) N:1 S:1 E:0 W:0]
[(2,1) N:0 S:0 E:0 W:2]
[(2,1) N:0 S:0 E:1 W:1]
[(2,1) N:0 S:0 E:2 W:0]
[(2,1) N:0 S:1 E:0 W:1]
[(2,1) N:0 S:1 E:1 W:0]
[(2,1) N:1 S:0 E:0 W:1]
[(2,1) N:1 S:0 E:1 W:0]
[(2,1) N:1 S:1 E:0 W:0]
```

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

```
6 8 5
6 8 2
2 0 8
```

```
Possible actions:
```

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

2. 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.

2.1 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.

2.2 METHODS INTRODUCED

- FRONTIER QUEUE

```
public static void frontierQueue (State t) {
    List<Action> actions = new ArrayList<Action>();
    PriorityQueue<Node> frontier = new PriorityQueue<Node>();

    Node initialState = new Node();
    initialState.setState(t);
    frontier.offer(initialState);

    double max = 0;
    double min = 10000000;
    double average=0;
    double time1 = 0;
    double time2;
    double t_time = 0;

    List<Double> times = new ArrayList<Double>();

    while(!frontier.isEmpty() && !isGoal(frontier.peek().getState())) {
        time1 = System.currentTimeMillis();
        actions = State.successor(frontier.peek().getState());
        State newState = new State();
        State currentState = frontier.poll().getState();

        for (int i=0; i<actions.size(); i++) {
            newState = applyAction(currentState, actions.get(i));
            newState.setN_cols(currentState.getN_cols());
            newState.setN_rows(currentState.getN_rows());
            Node aux = new Node();
            aux.setState(newState);
            frontier.offer(aux);
            time2 = System.currentTimeMillis();
            t_time= time2-time1;
            System.out.println(t_time);
            times.add(t_time);
```

```
                if(t_time < min) {
                    min = t_time;
                } else if (t_time > max) {
                    max = t_time;
                }
            }
        }
        System.out.println("Minimum: " + min);
        System.out.println("Maximum: " + max);

        for(int i=0;i<times.size(); i++) {
            average += times.get(i);
        }
        System.out.println("Average: " + average/times.size());

    }
```

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

```
public static boolean isGoal (State st) {
    int [][] field = st.getField();

    for (int i=0; i<field.length; i++) {
        for (int j=0; j<field.length; j++) {
            if(field[i][j] != st.getK()) {
                return false;
            }
        }
    }
    return true;
}
```

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

- APPLY ACTION

```
public static State applyAction (State st, Action ac) {

    int [][] newField = st.getField();
    int pos = newField[st.getX()][st.getY()]; //Amount of sand in the current position
    int movedSand = ac.getSand_e() + ac.getSand_n() + ac.getSand_s() + ac.getSand_w(); //Totally amount of sand to move
    int newPos = pos - movedSand;
    newField[st.getX()][st.getY()] = newPos; //Remaining sand in the current position

    /* If there is sand to move to the North, East, West or South && if it is possible to move to the North, East, West or South*/
    if(ac.getSand_n() > 0 && st.getX()-1 > 0) {
        newField[st.getX()-1][st.getY()] += ac.getSand_n(); //Update the amount of sand of the field
    }
    if(ac.getSand_s() > 0 && st.getX()+1 < newField.length) {
        newField[st.getX()+1][st.getY()] += ac.getSand_s();
    }
    if(ac.getSand_w() > 0 && st.getY()-1 > 0) {
        newField[st.getX()][st.getY()-1] += ac.getSand_w();
    }
    if(ac.getSand_e() > 0 && st.getY()+1 < newField.length) {
        newField[st.getX()][st.getY()+1] += ac.getSand_e();
    }

    State newState = State.copyState(st, ac, newField);

    return newState;
}
```

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.

2.3 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:

```
Frontier implemented with queue:
Minimo: 191161.0 nanoseconds
Maximo: 1.077517E7 nanoseconds
Average: 1723686.60087241 nanoseconds
```

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

```
Frontier implemented with List:
Minimo: 198859.0 nanoseconds
Maximo: 1.3448007E7 nanoseconds
Average: 2061711.9353846153 nanoseconds
```

3. 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.

3.1 METHODS INTRODUCED

- BOUNDED SEARCH

```
public static List<Node> boundedSearch(State st, String strategy, int max_depth) {
    PriorityQueue<Node> frontier = new PriorityQueue<Node>();
    Node initial_node = new Node(st, 0, 0, null); //Initial node with the initial state, cost and depth are 0 and there's no reference to a parent
    initial_node.selectValueNode(strategy);
    frontier.add(initial_node);
    Node current_node = new Node();

    boolean solution = false;


    while(!frontier.isEmpty() && solution==false) {
        current_node = frontier.poll(); //Removes the node which is in the head of the queue

        if(isGoal(current_node.getState())) {
            solution = true;
        } else {
            List<Action> actions = State.successor(current_node.getState()); //Generates all the possible actions
            List<Node> nodes = createNodeList(actions, current_node, max_depth, strategy, current_node.getState()); //Creates a node list where each node has a new state,
                                                                                                                     //generated after applying the corresponding action

            for(int i=0; i<nodes.size(); i++) {
                frontier.add(nodes.get(i));
            }
        }
    }

    if(solution) {
        return createSolution(current_node);
    } else {
        System.out.println("No solution");
        return null;

    }
}
```

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.

```
private static List<Node> createSolution(Node current_node) {
    List <Node> solution = new ArrayList<Node>();

    while (current_node != null) {
        solution.add(current_node);
        current_node=current_node.getFather();
    }

    return solution;
}
```

- CREATE NODE LIST

```
public static List<Node> createNodeList (List<Action> actions, Node cn, int depth, String strategy, State st) {
    List <Node> nodes = new ArrayList<Node>();
    int new_depth = cn.getDepth() +1;
    if(cn.getDepth() + 1 <= depth) {
        for(int i=0; i<actions.size(); i++) {
            State c_state = applyAction(st, actions.get(i));
            Node aux = new Node(c_state, State.cost(actions.get(i)), new_depth, cn);
            aux.selectValueNode(strategy);
            nodes.add(aux);
        }
    }

    return nodes;
}
```

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.

```
public void selectValueNode(String strategy) {
    if(strategy == "BFS") {
        value = depth;
    } else if(strategy == "DFS" || strategy == "DLS" || strategy == "IDS") {
        value = -depth;
    } else if (strategy == "UCS") {
        value = cost;
    }
}
```

3.2 TESTING

```
Enter the name of the .txt
field2

535
584
555

Possible movements:
[2,1]
[0,1]
[1,2]
[1,0]


Please, enter the strategy
BFS, DFS, DLS, ILS or UCS
BFS
Now, enter the depth desired.
5
Action [sand_n=2, sand_s=0, sand_w=0, sand_e=1]
555
555
555

The total cost is 4 and the total depth is 1
```

The initial field for this testing is:

5 3 5
5 8 4
5 5 5

The amount of sand desired is 5 and the maximum sand is 8. So, we know that, with only one action, the solution is going to be found.

The total cost is 4, as it is the total amount of sand moved in each action plus 1.

4. SUBTASK IV

In this subtask, we have to add the strategy A* using as heuristic for a concrete state:

h(state) = the number of boxes without k elements of sand.

So, we have to implement a method to compute the heuristic of the strategy.

We also need a list of visited states where there is the lowest rating of the state node.

In order to keep the list of visited states we are using a *hashtable,* having as key an encrypted state and as value the associated value to each state.

If a state has been previously visited, we have to check its value. If the value is better, we update the value of the hashtable associated to this state. As the state has been visited, it's no needed to insert it into the frontier. In the other hand, is the value is worse, we simply ignore it.

If the state has not been visited, then we insert it to the frontier.

This is also known as pruning or optimization.

4.1 METHODS INTRODUCED

There is a modification in the method SelectedValue in order to choose an A* strategy, taking into account the cost and the heuristic.

```
public void selectValueNode(String strategy) {
    if(strategy.equals("BFS")) {
        this.value = this.depth;
    } else if(strategy.equals("DFS") || strategy.equals("DLS") || strategy.equals("IDS")) {
        this.value = this.depth * -1;
    } else if (strategy.equals("UCS")) {
        this.value = this.cost;
    } else if (strategy.equals("A*")) {
        this.value = cost + heuristic(this.state);
    }
}
```

- HEURISTIC

```
public int heuristic(State st) {
    int h=0;
    int [][] field = st.getField();


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

In this method, the heuristic of the A* strategy is calculated. Taking into account the field, the method checks if in the square there isn't the desired amount of sand, and it increases the value of the variable, which is going to be the heuristic.

- CHECK VISITED

```
public static void checkVisited(List<Node> nodes, String strategy) {
    for (int i = 0; i < nodes.size(); i++) {
        if (visited.containsKey(encryptState(nodes.get(i).getState()))) {
            if (strategy.equals("BFS") || strategy.equals("DFS") || strategy.equals("DLS") || strategy.equals("IDS")) {
                if (nodes.get(i).getCost() < visited.get(encryptState(nodes.get(i).getState()))) {
                    visited.replace((encryptState(nodes.get(i).getState())), nodes.get(i).getCost());
                } else {
                    nodes.remove(i);
                }
            } else {
                if (nodes.get(i).getValue() < visited.get(encryptState(nodes.get(i).getState()))) {
                    visited.replace((encryptState(nodes.get(i).getState())), nodes.get(i).getValue());
                } else {
                    nodes.remove(i);
                }
            }
        } else {
            if (strategy.equals("BFS") || strategy.equals("DFS") || strategy.equals("DLS") || strategy.equals("IDS")) {
                visited.put(encryptState(nodes.get(i).getState()), nodes.get(i).getCost());
            } else {
                visited.put(encryptState(nodes.get(i).getState()), nodes.get(i).getValue());
            }
        }
    }
}
```

The method checkVisited is used to look for the visited states of the tree.

It checks if a state is contained in the hashtable. If the state is already in the list of visited states, then the value associated to a state is checked. If the value is better than the one in the hashtable, then it is replaced. If not, the node is removed from the list of nodes which is going to be inserted in the frontier (List <Node> nodes).

If the state has not been visited yet, then it is inserted into the hashtable.

We have to take into account that, if the strategy used is *BFS, DFS, DLS* or *IDS* the value associated is going to be the cost of the action taken into this node. Otherwise, the value is going to be the value assigned previously.

4.2 TESTING

Data for the A* and UCS testing:

- Initial position of the robot: X=1 and Y=1
- Desired sand for each box (K) = 5
- Maximum sand in each box = 8
- Initial Field:

  736
  284
  654

TESTING WITH A*

```
Initial Field:
736
284
654

Action [sand_n=1, sand_s=1, sand_w=0, sand_e=1] Next move: [2,1] Cost of Action: 4
Total cost 4
746
255
664

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=1] Next move: [2,0] Cost of Action: 2
Total cost 6
746
255
655

Action [sand_n=1, sand_s=0, sand_w=0, sand_e=0] Next move: [1,0] Cost of Action: 2
Total cost 8
746
355
555
```

```
Action [sand_n=0, sand_s=0, sand_w=0, sand_e=0] Next move: [0,0] Cost of Action: 1
Total cost 9
746
355
555

Action [sand_n=0, sand_s=2, sand_w=0, sand_e=0] Next move: [0,1] Cost of Action: 3
Total cost 12
546
555
555

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=0] Next move: [0,2] Cost of Action: 1
Total cost 13
546
555
555


Action [sand_n=0, sand_s=0, sand_w=1, sand_e=0] Next move: [1,2] Cost of Action: 2
Total cost 15
555
555
555
```

THE TOTAL COST IS 15 AND THE TOTAL DEPTH IS 7

- TESTING WITH UCS

```
Initial Field:
736
284
654

Action [sand_n=0, sand_s=1, sand_w=1, sand_e=1] Next move: [2,1] Cost of Action: 4
Total cost 4
736
355
664

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=1] Next move: [2,0] Cost of Action: 2
Total cost 6
736
355
655
```

```
Action [sand_n=1, sand_s=0, sand_w=0, sand_e=0] Next move: [1,0] Cost of Action: 2
Total cost 8
736
455
555

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=0] Next move: [0,0] Cost of Action: 1
Total cost 9
736
455
555

Action [sand_n=0, sand_s=1, sand_w=0, sand_e=1] Next move: [0,1] Cost of Action: 3
Total cost 12
546
555
555

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=0] Next move: [0,2] Cost of Action: 1
Total cost 13
546
555
555

Action [sand_n=0, sand_s=0, sand_w=1, sand_e=0] Next move: [0,1] Cost of Action: 2
Total cost 15
555
555
555
```

THE TOTAL COST IS 15 AND THE TOTAL DEPTH IS 7

OBSERVATIONS:

We can see that the solution for A* and UCS is the same.

Data for the testing of BFS:

- Initial position of the robot: X=1 and Y=1
- Desired sand for each box (K) = 1
- Maximum sand in each box = 4
- Initial Field:

  332
  000
  010

TESTING WITH BFS

```
Initial Field:
332
000
010

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=0] Next move: [0,1] Cost of Action: 1
Total cost 1
332
000
010

Action [sand_n=0, sand_s=1, sand_w=0, sand_e=1] Next move: [0,2] Cost of Action: 3
Total cost 4
313
010
010

Action [sand_n=0, sand_s=0, sand_w=2, sand_e=0] Next move: [0,1] Cost of Action: 3
Total cost 7
331
010
010

Action [sand_n=0, sand_s=1, sand_w=1, sand_e=0] Next move: [0,0] Cost of Action: 3
Total cost 10
411
020
010

Action [sand_n=0, sand_s=1, sand_w=0, sand_e=2] Next move: [0,1] Cost of Action: 4
Total cost 14
131
120
010
```

```
Action [sand_n=0, sand_s=2, sand_w=0, sand_e=0] Next move: [1,1] Cost of Action: 3
Total cost 17
111
140
010

Action [sand_n=0, sand_s=2, sand_w=0, sand_e=1] Next move: [2,1] Cost of Action: 4
Total cost 21
111
111
030

Action [sand_n=0, sand_s=0, sand_w=1, sand_e=1] Next move: [2,0] Cost of Action: 3
Total cost 24
111
111
111

The total cost is 24 and the total depth is 8
Spatial complexity: 1080158. Time complexity: 5.736 seconds.
```

OBSERVATIONS:

This is the execution of the problem provided in Moodle. It's said that the depth in the solution provided and in our solution, must be the same.

As we can see, we get the same depth that in the solution provided.

- TESTING PRUNING

```
Initial Field:
332
000
010

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=0] Next move: [0,1] Cost of Action: 1
Total cost 1
332
000
010

Action [sand_n=0, sand_s=1, sand_w=0, sand_e=1] Next move: [0,2] Cost of Action: 3
Total cost 4
313
010
010
```

```
Action [sand_n=0, sand_s=2, sand_w=0, sand_e=0] Next move: [1,2] Cost of Action: 3
Total cost 7
311
012
010

Action [sand_n=0, sand_s=1, sand_w=0, sand_e=0] Next move: [0,2] Cost of Action: 2
Total cost 9
311
011
011

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=0] Next move: [0,1] Cost of Action: 1
Total cost 10
311
011
011

Action [sand_n=0, sand_s=0, sand_w=0, sand_e=0] Next move: [0,0] Cost of Action: 1
Total cost 11
311
011
011

Action [sand_n=0, sand_s=2, sand_w=0, sand_e=0] Next move: [1,0] Cost of Action: 3
Total cost 14
111
211
011

Action [sand_n=0, sand_s=1, sand_w=0, sand_e=0] Next move: [0,0] Cost of Action: 2
Total cost 16
111
111
111

The total cost is 16 and the total depth is 8
Spatial complexity: 62749. Time complexity: 1.422 seconds.
```

OBSERVATIONS:

We have run the same example than the one used for the testing of BFS.

We can see that the depth of the solution is the same although the cost is reduced.

In order to prove that the optimization or pruning is working, we have to check the spatial complexity as well as the time complexity. In the non optimized version of the solution, the number of nodes explored is 1080158 and the time spent for the execution is almost 6 seconds.

In contrast, in the optimized version, the spatial complexity is reduced to 62749 and the time complexity is 1.42 seconds.

We can clearly see that the optimization works correctly.