Practical Exercise 2: Planner

Abhishek Saurabh *[1] and Juanjo Rubio †[2]

[1]Course Name: Planning and Absolute Reasoning (PAR)
[2]Course Supervisor: Dr. Aida Valls & Dr. Antonio Moreno
[3]Master's in Artificial Intelligence, Polytechnic University of Catalonia (UPC), University of Barcelona (UB) and University of Rovira Virgili (URV)

31-January-2016

1 Introduction

The goal in this practical exercise was to design and implement the *robot-box-office-cleaning* problem using a *linear planner with a stack of goals* that can discover how to go efficiently from an initial state of the world to a final state.

The java implementation of this problem was based on STRIPS problem description language. This report describes the design and implementation of the said system. This document is organized in the following manner: Section 2 provides a description of the problem. Section 3 describes the manner in which the developed system was given an input through a text file following which Section 4 has a description on methodology including the heuristic applied. Section 5 and 6 deal with the results & discussions and limitations of this implementation respectively.

2 Problem description

The problem requires an automated cleaning robot to clean offices, which are 9 in number located in a 3*3 matrix in a square building. The offices can be either clean or dirty. Robot can move from one office to the other only along the row or along the column. Besides, there are one through eight boxes which could be present in the office, with no more than one box in one office. An office can be cleaned only if it is empty. Robot can move a box from an office to the adjacent office. In the goal state, all the rooms has to be clean, with positions of the robot and the boxes clearly stated.

3 Input file description

The initial configuration is fed into the system through a text file. The said text file has the following fields, in the order mentioned below:

1. Boxes: Comma separated box names from At most A through H i.e. at most 8 boxes.

*abhishek.saurabh@est.fib.upc.edu
†juanjo.rubio@est.fib.upc.edu

2. Offices: Comma separated office name identifiers *o1* through *o9*.
3. Initial state: Semi-colon separated predicates viz. *Dirty*, *Clean*, *Robot-location*, *Box-location* and *Empty*.
4. Goal state: Semi-colon separated predicates *Robot-location*, *Box-location* and *Empty*. Predicate *Clean* is optional.

In addition to the sample initial configuration which had been provided along with the assignment, the implementation had to be tested with at four other configurations, with some clean and dirty offices and a box in some of the rooms.

All the five sets of initial configuration can be found in Appendix A and C with the latter having a diagrammatic representation of the initial and final configurations.

4 Methodology

The java implementation of this problem was based on STRIPS problem description language. STRIPS planner, which is a simple linear solver, is also known as the total order solver as it assumes that all steps can be planned one after the other with a strict ordering. It works backwards starting from the goal state and searching for actions that satisfy the preconditions of the goal state. As an when the program finds an acceptable solution, it gets added to the plan and the newly added action becomes the new goal state. If an action is added that conflicts with previously satisfied goal, the latter is reintroduced and re-satisfied. During this process, the algorithm keeps working backwards, stacking goals on top of each other until either all the goals are satisfied and it arrives at the final plan or it exhausts all possible plans and finally gives up.

Following two subsections details out the implementation and the heuristics applied. Please note that the instructions to execute the code has been provided in appendix D.

4.1 Implementation

Classes have been created for all the elements involved in this planner. Both predicates and operators are described following an Object Oriented Programming (OOP) approach, allowing us to group them as *StackElements* in order to build the stack of the planner. On the other hand, all the physical elements of the scene are also modeled as objects in order to represent the scene in a natural way and for having the option of accessing its attributes easily.

Appendix B includes a simplified version of the class diagrams depicting the implementation of the project. For the sake of brevity of this document, only the diagrams of the classes which depict the modeling of the stack of the goals have been shown here. Some other classes like *PARser*, *BoardParameters* etc, which are not a part of the goal stacking process have been excluded.

Regarding the linear planner with stack of goals, a part of the Planner class, including only the main steps, is expressed in the following snippet of code:

```
while(!stack.isEmpty()){
        // Pop each element from the main stack at each iteration.
        currentStackElement = goalStack.popFromStack();
        if (currentStackElement instanceof Predicate){
                // If predicate is already accomplished, remove it from the
                    stack.
                if(((Predicate) currentStackElement).checkPredicate()){
                        // Get rid of the predicate.
```

```
                        currentStackElement = null;
                }
                else{
                        // If the predicate is unsatisfied, find operator that
                            accomplishes it.
                        OperatorFinder operatorFinder = new OperatorFinder();
                        Operator operator = operatorFinder.findOperator(
                            boardParameters,((Predicate)currentStackElement));
                        List<Predicate> preconditions = operator.
                            listPreconditions();
                        /* ... */
                }
        }else if (currentStackElement instanceof Operator){
                        /* ... */
                        ((Operator) currentStackElement).add();
                        plan.add(currentStackElement);
                        /* ... */
        }
}
```

The linear planner is following the steps of the STRIPS algorithm. At each iteration, the top element of the stack is evaluated. If the given element is a predicate, first it is checked in order to proceed to next iteration if the predicate is satisfied, if not, it is used to find an operator that has the predicate in its add effects. This last case is solved in the *OperatorFinder* class, where the predicate is checked with the add effects given by the *addEfects* method in each operator described above. When an operator that fulfills the predicate is found, it is returned to the planner and added to the stack as well as all the preconditions of the new operator. This new operator and preconditions are then a new sub-goal of the linear planner. Finally, if top of the stack is an operator and the preconditions regarding that particular sub-goal are satisfied, the operator is applied and added to the final plan.

As mentioned before, the *OperatorFinder* class is in charge of finding an operator that can fulfill a given unsatisfied predicate. Each operator has a *getAddEfects* that returns the add efect given an argument passed. This process can be seen in the following snippet of code extracted from the *OperatorFinder* class for checking an unsatisfied *CleanOperator*:

```
    // Check CLEAN PREDICATES
        } else if (predicate instanceof CleanPredicate) {
            // Check clean office operator
            CleanOfficeOperator cleanOfficeOperatorChecker = new
                CleanOfficeOperator();
            // Obtain list of add efects for CleanOfficeOperator with the
                office from unsatisfied predicate
            List<Predicate> predicates = cleanOfficeOperatorChecker.
                getAddEffects(((CleanPredicate) predicate).getOffice());
            for (Predicate operatorAddEffect : predicates) {
                if (predicate.getClass().equals(operatorAddEffect.getClass()))
                    {
                    if (((CleanPredicate) predicate).getOffice() ==
                            ((CleanPredicate) operatorAddEffect).getOffice())
                                {
                        // If the precondition is accomplished by the
                            CleanOfficeOperator, create it and return it.
                        CleanOfficeOperator cleanOfficeOperator = new
                            CleanOfficeOperator(((CleanPredicate) predicate).
                            getOffice(),
                            boardParameters.getRobot());
                        return cleanOfficeOperator;
                    }
                }
            }
```

4.2 Heuristic applied

One of the instances when heuristic was applied to the problem was while creating an operator for pushing a box. The first move to an adjacent cell which the robot takes while pushing a box from its current position, is taken by taking into account the final location of the box in question. This action is performed by the *getBestAdjacentToPushTo* method in the *OperatorFinder* class. From the adjacent cell, again distance from the goal office is calculated using *manhattan block distance*. The aforesaid process has been implemented in the *getBestAdjacentToPushTo* as shown in the snippet below:

```
/ ... /
for (int i=0;i<adjacents.size();i++){
            rowOrigin = getXcoordinates()[adjacents.get(i)-1];
            columnOrigin = getYcoordinates()[adjacents.get(i)-1];
            distance = Math.abs(rowOrigin-rowDestination) + Math.abs(
                columnOrigin-columnDestination);
            distances.add(distance);
            officeIndices.add(adjacents.get(i));
        }
        Integer minDistance = Collections.min(distances);
        bestAdjacentToPush =  officeIndices.get(distances.indexOf(Collections.
            min(distances)))-1;
/ ... /
```

The aforesaid heuristic can be loosely approximated as A* star search algorithm for which the evaluation function is defined as:

$$f(n) = h(n) + g(n)$$

where, $h(n)$ is the cost to reach the nearest goal node from the node n, $g(n)$ is the cost to reach node n. The evaluation function $f(n)$ is defined as the estimated cost of the cheapest solution through n.

Drawing a parallel with this algorithm we can say that $g(n) = 1 unit manhattan block distance(constant)$, in the direction which would lead to the goal node the fastest. So for example, if the robot has three possible paths to move towards an office, it first moves to the adjacent cell which can lead to the goal office the fastest. While $h(n)$ is equal to the *manhattan block distance* from this node to the goal node.

Another heuristic which has been applied taking into account the domain knowledge is while defining the order of the stack. Considering that in a linear planner the elements of the stack need to be checked linearly, the order of the final goal stack is one of the ways to improve the performance. The following order of the stack was arrived at after applying different variants of the solution.

- Clean predicates
- Box-location predicates
- Empty predicates
- Robot-location predicate

The rationale behind ordering the predicates in this fashion is based on the fact that both *Clean predicates* and *Box-location predicates* would eventually move boxes, thereby making the process of checking *Empty predicates* earlier redundant. In one of our earlier attempts, we were checking *Empty predicates* before the *Box-location predicates* and the planner still succeeded. But many unneeded moves were performed in order to satisfy the *Empty predicates*. For decreasing the number of operators, the *Clean predicates* have been sorted in ascending order with respect to the initial position of the robot. Once sorted, they are pushed to the stack, making the robot start the cleaning process in the closest offices when applying the linear planner.

5 Results and Discussion

Considering that only two approaches to improve the linear planner have been applied regarding the movement of the robot and the adjacent cell to push a box to, the plan in many cases is sub-optimal. The order of the box-location predicates from the stack is not implemented in a particular order, which leads to solve the first box-location added from the settings text file, not taking into account the robot location after finishing the cleaning predicates. One way of improving this behavior would be ordering the box-location predicates taking into account the last cleaned office given by the order explained in heuristics section.

Next in this section we will analyze and evaluate the different testing cases we attached with the project. The configurations are provided in the settings text files present in the code folder. Configurations have also been shown in appendix C of this report.

Plan corresponding to settings2 file
[CLEAN-OFFICEo4, MOVEo4,o1, CLEAN-OFFICEo1, MOVEo1,o2, MOVEo2,o5, CLEAN-OFFICEo5, MOVEo5,o4, MOVEo4,o7, CLEAN-OFFICEo7, MOVEo7,o4, MOVEo4,o5, MOVEo5,o6, CLEAN-OFFICEo6, MOVEo6,o3, PUSHA,o3,o2, MOVEo2,o3, CLEAN-OFFICEo3, MOVEo3,o2, PUSHA,o2,o3, MOVEo3,o2, MOVEo2,o5]

This case is a very simple case in terms of complexity of the scene. The case was used to test the ability of the robot to clean the closest offices to its initial location, reducing the steps for cleaning compared with the case of not ordering the clean predicates when adding them to the stack. The other purpose of this scenario was to test the behaviour of the robot when cleaning an office with a box which in the goal state has the same location as in the initial. As we can see, the behaviour was correct, pushing the box for cleaning o3 and putting it when the box-location(A,o3) is evaluated in the stack. In terms of efficiency, the steps taken in the plan are good.

Plan corresponding to settings3 file
[MOVEo3,o2, CLEAN-OFFICEo2, MOVEo2,o3, MOVEo3,o6, CLEAN-OFFICEo6, MOVEo6,o5, MOVEo5,o4, CLEAN-OFFICEo4, MOVEo4,o5, MOVEo5,o8, PUSHB,o8,o5, MOVEo5,o8, CLEAN-OFFICEo8, MOVEo8,o5, MOVEo5,o2, MOVEo2,o1, PUSHA,o1,o2, MOVEo2,o5, PUSHB,o5,o6, MOVEo6,o3, MOVEo3,o2, PUSHA,o2,o5, MOVEo5,o6, PUSHB,o6,o3, MOVEo3,o2, MOVEo2,o1, MOVEo1,o4, MOVEo4,o7]

This case confirms that our order of stacking the clean predicates works because the robot starts cleaning the closest offices with respect to the initial position. We can see how the robot starts with o2 and o6 instead of an unordered predicates which would follow o4 after o2. On the other hand, this settings shows how pushing B when cleaning it pushes the box towards the final position of the box (o3). Finally, we observe how ordering the box-location predicates when populating the stack at the beginning would reduce the number of steps since when cleaning the last office o8, instead of pushing box B, the planner finds the box-location(A,5) first.

Plan corresponding to settings4 file
[CLEAN-OFFICEo6, MOVEo6,o5, PUSHB,o5,o2, MOVEo2,o5, CLEAN-OFFICEo5, MOVEo5,o6, MOVEo6,o9, PUSHC,o9,o6, MOVEo6,o9, CLEAN-OFFICEo9, MOVEo9,o6, MOVEo6,o5, MOVEo5,o4, CLEAN-OFFICEo4, MOVEo4,o5, MOVEo5,o8, CLEAN-OFFICEo8, MOVEo8,o5, MOVEo5,o2, MOVEo2,o1, PUSHA,o1,o4, PUSHA,o4,o5, PUSHA,o5,o8, MOVEo8,o5, MOVEo5,o2, PUSHB,o2,o3, MOVEo3,o6, PUSHC,o6,o5, PUSHC,o5,o2, MOVEo2,o5]

This case shows one case where when pushing box A from o1 to its final destination, one of the candidate adjacent cell to push to is blocked by another box (B) and instead of moving box B to push A to o1, it decides to push A to o4. When selecting the best adjacent to push

to, if two paths have the same minimum cost and one of them is not empty, the empty path is chosen when building the PUSH operator. This same case is faced again when pushing C from o6 to o3, since o3 is blocked by B, the alternative path is selected pushing to o5. Regarding the rest of the analysis we face the same box-location order when first pushing to the stack, since a sub-optimal path is chosen to move the boxes to its final location. In this case the complexity has increased since we added a third box.

Plan corresponding to settings5 file
[CLEAN-OFFICEo4, MOVEo4,o5, PUSHA,o5,o4, MOVEo4,o5, CLEAN-OFFICEo5, MOVEo5,o4, MOVEo4,o7, CLEAN-OFFICEo7, MOVEo7,o4, MOVEo4,o1, MOVEo1,o2, CLEAN-OFFICEo2, MOVEo2,o3, PUSHE,o3,o2, MOVEo2,o3, CLEAN-OFFICEo3, MOVEo3,o2, MOVEo2,o5, MOVEo5,o8, PUSHC,o8,o5, MOVEo5,o6, MOVEo6,o9, PUSHF,o9,o8, MOVEo8,o9, CLEAN-OFFICEo9, MOVEo9,o6, MOVEo6,o5, MOVEo5,o4, PUSHA,o4,o7, MOVEo7,o4, MOVEo4,o5, MOVEo5,o6, PUSHB,o6,o3, MOVEo3,o2, PUSHE,o2,o1, MOVEo1,o2, MOVEo2,o3, PUSHB,o3,o2, MOVEo2,o1, PUSHE,o1,o4, MOVEo4,o1, MOVEo1,o2, PUSHB,o2,o1, MOVEo1,o2, MOVEo2,o3, MOVEo3,o6]

In terms of complexity, this new scenario is more difficult to solve compared to the rest of the cases since 5 boxes have been included. As we can see, one case of blocked box arises when trying to clean o9 since box F is blocked by B in o6 and C in o8. We manage to push C to o5 before pushing F to o8 in order to clean o9. Another interesting case appears when pushing B to its final location o1 since E it is on its way, since our goal at this point is related to the box-location(B,o1), box E is an obstacle and it's pushed at each step in order to be able to push B to o1. This shows inefficiency since at each step, pushing the obstacle box in the same direction of B's direction makes the problem recurrent. Even though this case is not efficiently solved, the planner manages to build a valid plan. As stated, the number of actions is not optimal mainly because of the recurrent problem of a box blocking the shortest path of another box.

6 Limitations

The linear planner works as expected while testing but we found one case that caused our implementation to fail. This case is found when pushing one box to its final destination and another box is found in the particular case where the obstacle box final location is in the opposite direction, i.e., when we have to swap positions of boxes, our implementation fails. Since our heuristics tries to push a box to its closest adjacent considering the final box-location, if there is only one minimum distance path and that path is blocked by a box which its final location is in the opposite direction, our planner goes into an infinite loop. This issues arise when increasing the number of boxes since many paths get blocked by other boxes. Notwithstanding the description of the case in which the planner fails, the behaviour of the planner is as per the implementation and hence expected.

References

[1] Astar. http://www.redblobgames.com/pathfinding/grids/graphs.html.

Appendix A

Sample initial configuration
Boxes=A,B
Offices=o1,o2,o3,o4,o5,o6,o7,o8,o9
InitialState=Dirty(o1);Clean(o2);Clean(o3);Clean(o4);Dirty(05);Clean(o6);Clean(07);
Dirty(o8);Dirty(o9);Robot-location(o4);Box-location(A,o5);Box-location(B,o6);
Empty(o1);Empty(o2);Empty(o3);Empty(o4);Empty(o7);Empty(o8);
GoalState= Robot-location(o9); Box-location(A,o8); Box-location(B,o6);Empty(o1);Empty(o2);Empty(o3); Empty(o4);Empty(o5);Empty(o7);

Test initial configuration 1: Settings2
Boxes=A
Offices=o1,o2,o3,o4,o5,o6,o7,o8,o9
InitialState=Dirty(o1);Clean(o2);Dirty(o3); Dirty(o4);Dirty(o5);Dirty(o6);
Dirty(o7);Clean(o8);Clean(o9); Robot-location(o4);Box-location(A,o3);
Empty(o1);Empty(o2);Empty(o4);Empty(o5);Empty(o6);Empty(o7);Empty(o8);Empty(o9);
GoalState=Robot-location(o5); Box-location(A,o3);Empty(o1);Empty(o2);Empty(o4);Empty(o5);Empty(o6);Empt

Test initial configuration 2: Setting3
Boxes=A,B
Offices=o1,o2,o3,o4,o5,o6,o7,o8,o9
InitialState=Clean(o1);Dirty(o2);Clean(o3); Dirty(o4);Clean(o5);Dirty(o6);
Clean(o7);Dirty(o8);Clean(o9); Robot-location(o3);Box-location(A,o1);Box-location(B,o8);Empty(o2);Empty(o3);Empty(o4);Empty(o5);Empty(o6);Empty(o7);Empty(o9);
GoalState=Robot-location(o7); Box-location(A,o5); Box-location(B,o3);Empty(o1);Empty(o2);Empty(o4);Empty(o9);Empty(o6);Empty(o7);Empty(o8);

Test initial configuration 3: Setting4
Boxes=A,B,C
Offices=o1,o2,o3,o4,o5,o6,o7,o8,o9
InitialState=Clean(o1);Clean(o2);Clean(o3); Dirty(o4);Dirty(o5);Dirty(o6);
Clean(o7);Dirty(o8);Dirty(o9); Robot-location(o6);Box-location(A,o1);Box-location(B,o5);Box-location(C,o9);Empty(o2);Empty(o3);Empty(o4);Empty(o6);Empty(o7);Empty(o8);
GoalState=Robot-location(o5); Box-location(A,o8); Box-location(B,o3);Box-location(C,o2);Empty(o1);Empty(o4);Empty(o5);Empty(o6);Empty(o9);

Test initial configuration 4: Setting5
Boxes=A,B,C,E,F
Offices=o1,o2,o3,o4,o5,o6,o7,o8,o9
InitialState=Clean(o1);Dirty(o2);Dirty(o3);Dirty(o4);Dirty(o5);Clean(o6);Dirty(o7);Clean(o8);Dirty(o9);
Robot-location(o4);Box-location(A,o5);Box-location(B,o6);Box-location(C,o8);Box-location(E,o3);Box-location(F,o9);Empty(o1);Empty(o2);Empty(o4);Empty(o7);
GoalState=Robot-location(o6);Box-location(A,o7);Box-location(B,o1);Box-location(C,o5);Box-location(E,o4);Box-location(F,o8);Empty(o2);Empty(o3);Empty(o6);Empty(o9)

Appendix B

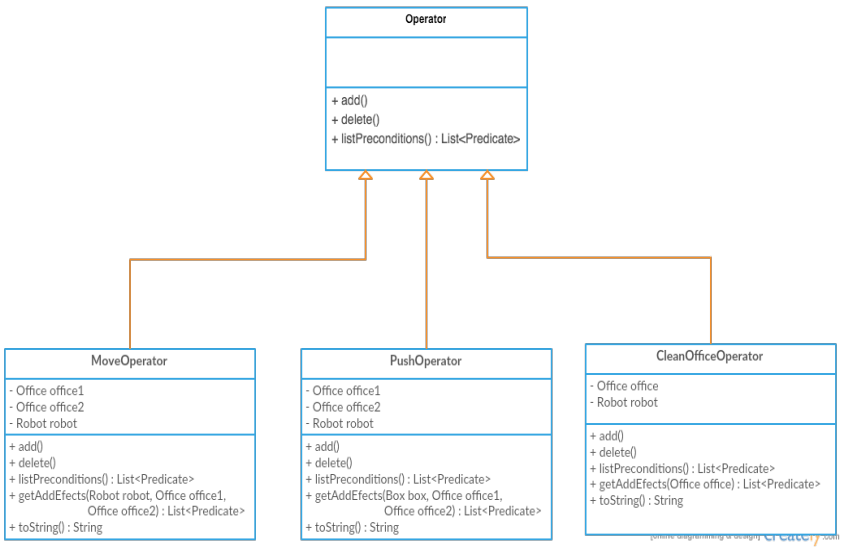

Figure 1: Class diagram for Operators

.

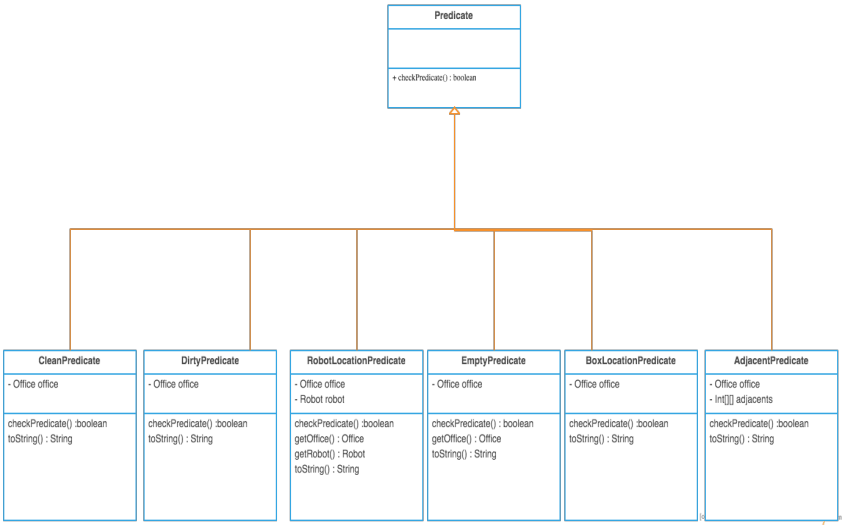

Figure 2: Class diagram for Predicates

.

Appendix C

Labels: D: Dirty; C: Clean; R: Robot; E: Empty, Box- A through F

Setting 2:
Initial configuration

D	C	D,BoxA
D, R	D	D
D	C	C

Goal configuration

C,E	C,E	BoxA
C,E	R,E	C,E
C,E	C,E	C,E

Setting 3:
Initial Configuration

C, BoxA	D	C, R
D	C	D
C	D,BoxB	C

Goal configuration

E	E	BoxB
E	BoxA	E
E	R	E

Setting 4:
Initial Configuration

C,BoxA	C	C
D	D,Box5	D,R
C	D	D,BoxC

Goal configuration

E	BoxC	BoxB
E	R,E	E
E	BoxA	E

Setting 5:
Initial Configuration

C	D	D,BoxE
D,R	D,BoxA	C,BoxB
D	C,BoxC	D,Box9

Goal configuration

BoxB	E	E
BoxE	BoxC	R,E
BoxA	BoxF	E

Appendix D

READ ME

1. The code provided is an eclipse project.
2. Open the file *RobotPlanner.java*. This file has the main function.
3. Change the name of the setting file in line 13 Please note that the extension .txt is not to be added. Code takes care of this extension. The custom files provided are settings2.txt, settings3.txt, settings4.txt and settings5.txt.
4. Please note that any other configuration other than those provided with the code, should have the same structure as provided in the files. Boxes, Offices, InitialState and GoalState in separate lines.
5. The code provided creates a text output saved as *inputfilename_ out.txt*. In order to have the output at the console, please comment line 29 in the *RobotPlanner.java*.