Practical Exercise 1: Planner

The coffee server

Planning and Approximate Reasoning

Universitat Rovira i Virgili

Master in Artificial Intelligence

1st Semester

Authors:
Javier Beltrán
Jorge Rodríguez

November 6, 2016

1 Introduction to the problem

For this first laboratory we were asked to consider the following scenario:

There is a squared building composed by 36 offices, which are located in a matrix of 6 rows and 6 columns. From each office it is possible to move (horizontally or vertically) to the adjacent offices. The building has some coffee machines in some offices that can make 1, 2 or 3 cups of coffee at one time. The people working at the offices may ask for coffee and a robot called "Clooney" is in charge of serving the coffees required. Each office may ask for 1, 2 or 3 coffees but not more. The petitions of coffee are done all at early morning (just when work starts) so that the robot can plan the service procedure. Each petition has to be served in a single service. The goal is to serve all the drinks to all the offices in an efficient way (minimizing the travel inside the building, in order not to disturb the people working).

We start with an initial state called e_i, which is compounded of a set of predicates. The initial state in particular has the following predicates: The initial position of the robot, the position of the coffee machines, and position of the coffee petitions. The final state e_f, on the other hand, is compounded of the following predicates: The final robot position and the petitions served. The way of achieving this final state is by going to the machines, taking the coffee and going to each petition to serve the coffee.

To solve this problem we implemented a general planner which will be used to solve the problem, for this we will use an algorithm called STRIPS. This algorithm is based on stacking the predicates needed to get to the final state and then finding the operators needed to get this predicates. The implementation has been done in Java, and implementation details will be explained in detail below.

As we mentioned before, to formalise the problem we need to define a set of predicates and operators, for each operator we have series of preconditions, adds and deletes which are sets of predicates. This is shown in the Table 1 with the arguments needed for each operator:

Table 1: Table with operators and its adds, deletes and preconditions

Opertors	Preconditions	Adds	Deletes

Make(o,n)	robot-location(o)
	robot-free
	machine(o,n)
		robot-loaded(n)	robot-free

Move(o1,o2)	robot-location(o1)
	steps(x)
		robot-location(o2)
		steps(x+distance(o1,o2))
			robot-location(o1)
			steps(x)

Serve(o,n)	robot-location(o)
	robot-loaded(n)
	petition(o,n)
		served(o)
		robot-free
			petition(o,n)
			robot-loaded

The predicates are easier to define, the following list shows the predicates considered for this problem:

- **Robot-location(o)**: the robot is in office o.
- **Robot-free**: the robot has no cup of coffee.
- **Robot-loaded(n)**: the robot has n cups of coffee.

- **Petition(o,n)**: office o wants n cups of coffee.
- **Served(o)**: office o has been served, no more coffee is needed in this office.
- **Machine(o,n)**: there is a coffee machine in office o that produces n cups of coffee each time (with n equal to 1, 2 or 3).
- **Steps(x)**: the total distance travelled by the robot (calculated with Manhattan distance).

There are some consideration thats can be taken so the problem is simpler:

- The robot only makes coffee for a single petition each time. That is, the robot cannot make 3 cups of coffee to serve two different offices. First, will make coffee of one office and serve it, and after will go to the same or another coffee machine to serve the second office.
- If a petition is of n cups, the robot will make coffee only with a unique machine of capacity n. That is, the robot cannot make coffee with 2 or more machines and accumulate the cups to serve a unique petition.

2 Analysis of the problem

Taking into account the considerations to simplify the problem, this problem can be seen as an optimization problem where we want to minimize the number of steps of the robot choosing the order in which it has to visit every petition after passing for a coffee machine with the same amount of coffee as the petition.

The search space has a total size of $|P!||M^{|P|}|$, where $|P|$ is the number of petitions and $|M|$ the number of possible machine, supposing that the robot always takes the optimal path from one point to another, so the number of possible paths is irrelevant,as the cost is always the same. To get to this result we just need to notice that we have two lists of possible orders, the petitions which can't be revisited, what means a factorial; and the machines which can be revisited what means a exponential number of combinations. As we can see the number of petitions is what increases more the complexity of the problem, interestingly the complexity rely on the number of petitions and machines, the size of the grid does not affect the complexity.

For the example we have a total of 375000 possible solutions, however as we have taken the consideration that to serve a petition we need to go to a machine with the same number of coffees, the total number decreases to 1920.

Solving the problem with an standard exploration algorithm is not an option, we must use a different one, for this exercise we will use the STRIPS algorithm. STRIPS has a linear cost, however is does not find the optimal solution, unless we use the correct heuristics. This heuristics we have defined are explained in the next section.

Another thing to take into account is that, in the definition of the problem the robot moves across a square grid consisting of 36 squares, the squares are called o1, o2, ..., o36. To make it easier to count the number of steps, we have considered a way to convert this to coordinates (x, y). For a 6×6 grid we have the following equation:

$$x = o - 1 \ (mod 6) \tag{2.1}$$
$$y = [\frac{o-1}{6}] \tag{2.2}$$

Where o is the value of the position, [ ] represents the integer part of the number and $mod6$ means the remainder of the division.

Since this is an example, the preconditions, operators and states are quite straightforward, however the way the steps are calculated could somehow be troublesome.

For the STRIPS algorithm we need to organize well the preconditions as new operators are needed; if we do not do so, we can get to situations like cycles or being unable to instantiate parameters. For example, if we use the operator $Make(o, n)$: if the precondition $robot-location(o)$ is the one in the top of the stack, the algorithm would not know the value of o, instead, if we stack $Machine(o, n)$ last, the value o is given by the number of coffees n needed. In the following section we will discuss the heuristics needed to avoid this kind of situations and to minimise the steps needed to serve the coffees.

3 Planning algorithm

The first thing we have to consider is the order we stack the predicates for the final state, as this is the way we will resolve the problem, the order determines directly the optimality. To solve this we have created a heuristic that orders the petitions based on the distance to the final position. This way we solve the problem trying to end the nearest as possible from the final position. This does not assure optimality but it gives a decent enough solution.

For the operators $Make$ and $Serve$ we have chosen a fixed order to stack the preconditions, since the order determines the result of the problem. For the $Make$ operator, first we stack $robot - location$, since we do not know the value for the o parameter; then we stack $Machine$ so we can get the location o based on the petition value n, notice that the location of the machine will be the nearest to the robot and the petition, in other words, the one that minimises, d:

$$d = distance(machine, robot) + distance(machine, petition)$$

Finally, $robot - free$ gets stacked, since this is the first thing needed before getting coffee, notice that the way we have defined this problem, the robot will always be free when it gets here.

For $Serve$ operator the order is more or less the same. First we stack $robot - location$, since we do not know the value of o; then $robot - loaded$, due to it having to carry coffee to serve it and we do not know the number of coffees needed; the last thing to stack, which will be the first thing to get unstacked, is the petition which will instantiate the values of o and n.

Summarizing the planner will stack $Served(o)$ then it will unstack it, to get to $Served(o)$ we need the operator $Serve(o, n)$ which has the preconditions: $robot - location(o)$, $loaded(n)$ and $petition(o, n)$. with petition it instantiates the parameters, the planner unstacks loaded, for which it needs the operator $Make(o', n)$ after stacking the preconditions: $robot - location(o')$, $machine(o', n)$ and $robot - free$. It instantiates o' so it can use the operator $Move(o_i, o')$. Then the planner has solved one petition, and the rest are exactly the same.

4 Implementation design

The architecture of the system is presented here, as well as the class diagram and package diagram in UML notation.

4.1 A generic approach of the linear planner

Given that the problem consists in building a planner for a concrete solution (the coffee server), we believe that the best approach is to design a generic planner, which can be used for modelling any linear planning problem by just extending it.

For achieving this, and taking into account that the program is built in Java, we followed an Object-Oriented approach. That is, every element of the algorithm has to be part of the model of the program. So the first task is to identify the main components of the STRIPS algorithm with a stack of goals:

- The **stack** itself, which is a data structure with a LIFO policy. It contains all kinds of elements that are part of the problem, like the predicates and the operators.
- The elements to be stacked, specifically:
  - The **predicates**. They define the state of the problem and have 0 or more parameters, which can be instantiated or not.
  - The **operators**. They define how a state changes, by removing some predicates and adding others. They also have parameters which can be instantiated or not.
  - A **set of predicates**. They have to be added at some steps to make sure that we did not lose any predicate in order to achieve another one. There is a special case: the final **state** of the problem, which also has to be added to the stack as a set of predicates.
- As we have seen before, **parameters** are needed. They are just a key-value item, but the value may have been instantiated or not. When a parameter gets a value, all the appearances of the parameter in the stack must be updated with the new value. For example, a parameter in our problem might be ”o”, and it may take values from ”o1” to ”o36”.

It can be seen that many different kinds of objects have to be stored in the same stack. Also, we need access to all the elements pushed to the stack, not only the one on the top, so we can instantiate all the appearances of a parameter. Our proposal is the following: we can use the *Stack* $< E >$ from *Java Collections*. It solves our two main concerns because of these reasons:

- It grants access to each element stored with $get(i)$, so we can instantiate all appearances of a parameter.
- It is a generic stack, so it can be used to store objects of any type. But this type has to be the same. We solved this by constructing a *marker interface* called *Stackable*. This means that every object that can be stacked must inherit from the *Stackable* Java interface.

Our architecture for the Stackable elements is the following:

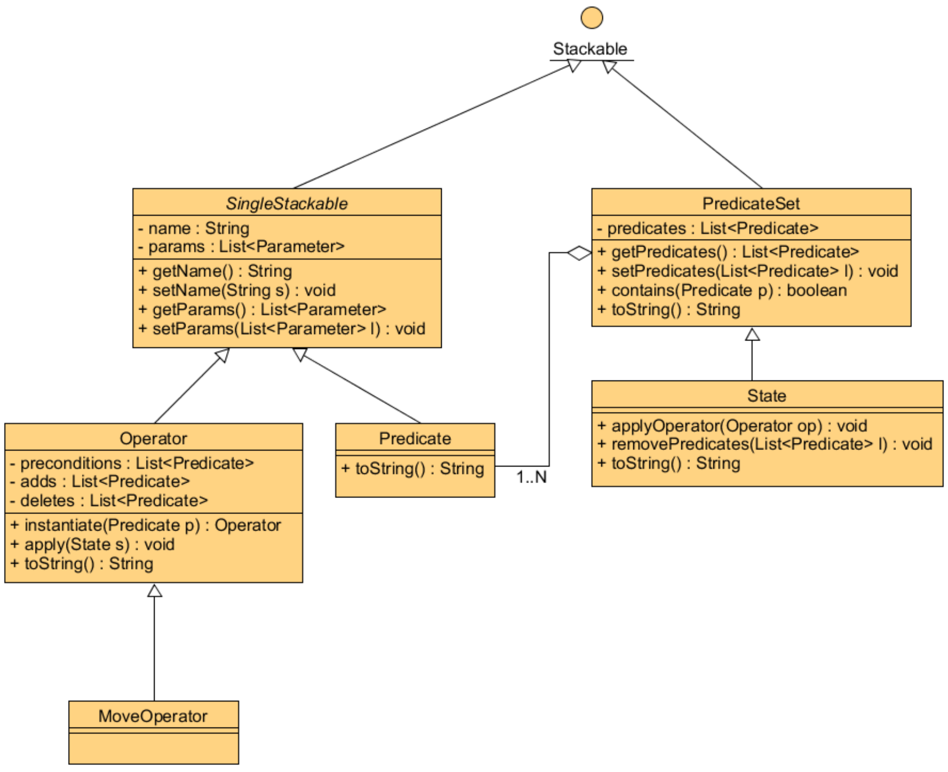

Stackable
SingleStackable
- name : String
- params : List<Parameter>
+ getName() : String
+ setName(String s) : void
+ getParams() : List<Parameter>
+ setParams(List<Parameter> l) : void
PredicateSet
- predicates : List<Predicate>
+ getPredicates() : List<Predicate>
+ setPredicates(List<Predicate> l) : void
+ contains(Predicate p) : boolean
+ toString() : String
State
+ applyOperator(Operator op) : void
+ removePredicates(List<Predicate> l) : void
+ toString() : String
Operator
- preconditions : List<Predicate>
- adds : List<Predicate>
- deletes : List<Predicate>
+ instantiate(Predicate p) : Operator
+ apply(State s) : void
+ toString() : String
Predicate
+ toString() : String
1..N
MoveOperator

Figure 1: Stackable elements architecture

The linear planning algorithm is executed in the **LinearPlanner** class. It needs some information about the problem to solve: the initial and final states, the available operators and the available predicates. It is also responsible for logging the execution of the algorithm so it is easier to debug. One can see that the Linear Planner is a complex object that requires multiple attributes and some constraints among them. For example, the predicates that compose the initial and final state should be consistent with the available predicates defined.

In order to make the task of creating a Linear Planner easier for the user, we added a *LinearPlannerBuilder*. It is a *Builder* pattern that helps in creating such a complex object. It also informs the user about errors or inconsistences in the declaration of the problem.

4.2 A specific application: The Coffee Server

If the problem to be solved is complex enough, just taking the generic planner and declaring the specific predicates, operators and states may not be enough. For example, some operators may have a special behaviour, that cannot be imitated by the generic Operator.

This is what happens in our Coffee Server. The $Move(o1, o2)$ operator, when applied, deletes the $Steps(x)$ predicates and adds $Steps(x + distance(o1, o2))$, where $distance(o1, o2)$ is the Manhattan distance between cells $o1$ and $o2$. We cannot use the generic Operator we built, because it just replaces some predicates for others, keeping the same parameters.

These problems can be solved by extending the generic Planner module, and adding some specific elements with a special behaviour. In our case, a MoveOperator has been created. It is able to instantiate the parameter of *Steps* properly, by increasing the path distance and saving it in the new predicate.

The heuristics used by the solver are also problem-specific, and should extend from the *Intelligence* interface. In our problem we define the heuristics explained in the previous section. They implement the methods *orderFinalState*() for stacking the petitions in the most efficient order, and *orderPreconditions*() for stacking the preconditions of every operator and preventing loops.

Here follows the class diagram of the program:

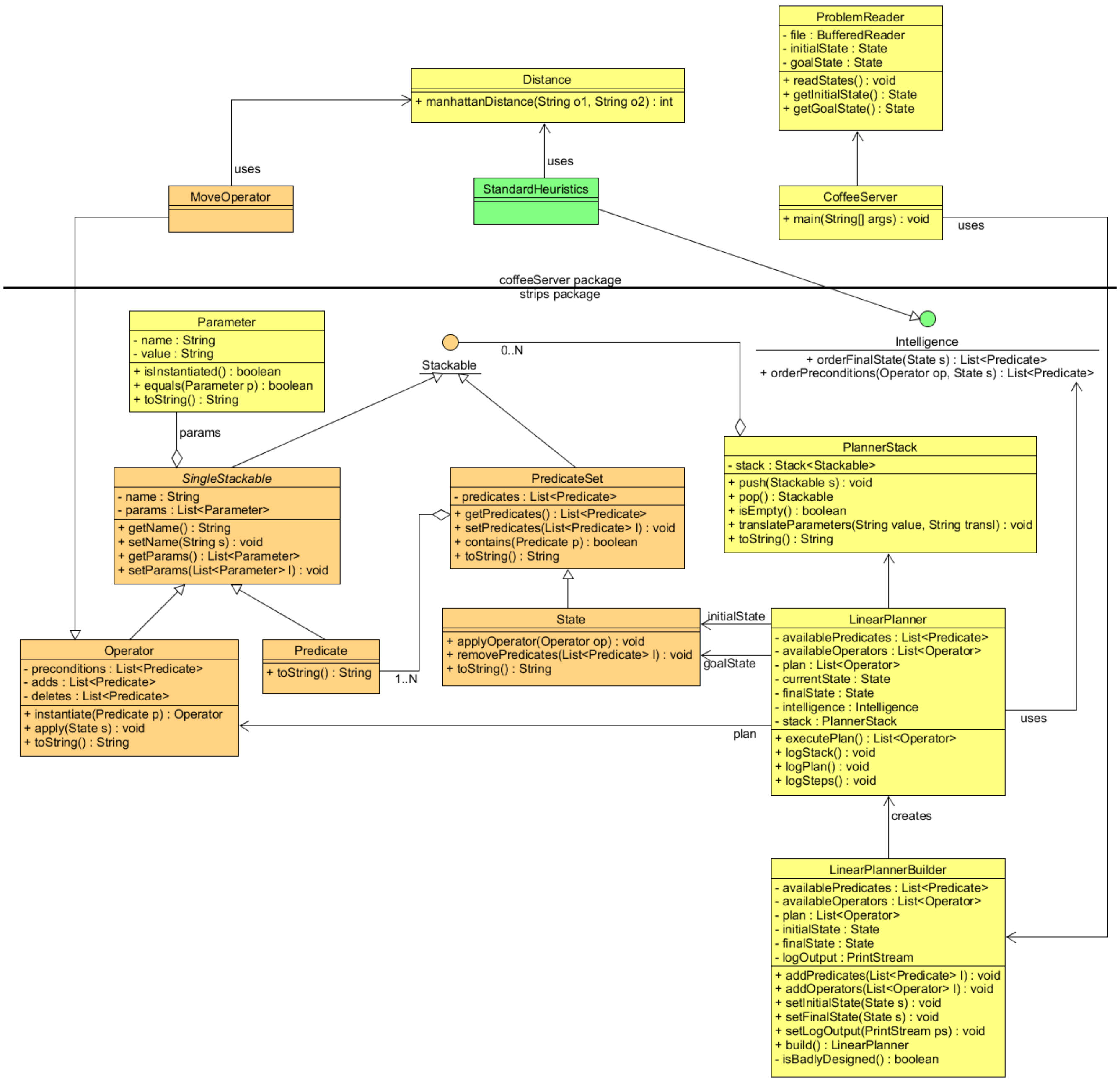

ProblemReader
- file : BufferedReader
- initialState : State
- goalState : State
+ readStates() : void
+ getInitialState() : State
+ getGoalState() : State
Distance
+ manhattanDistance(String o1, String o2) : int
uses
uses
MoveOperator
StandardHeuristics
CoffeeServer
+ main(String[] args) : void
uses
coffeeServer package
strips package
Parameter
- name : String
- value : String
+ isInstantiated() : boolean
+ equals(Parameter p) : boolean
+ toString() : String
params
Stackable
0..N
Intelligence
+ orderFinalState(State s) : List<Predicate>
+ orderPreconditions(Operator op, State s) : List<Predicate>
SingleStackable
- name : String
- params : List<Parameter>
+ getName() : String
+ setName(String s) : void
+ getParams() : List<Parameter>
+ setParams(List<Parameter> l) : void
PredicateSet
- predicates : List<Predicate>
+ getPredicates() : List<Predicate>
+ setPredicates(List<Predicate> l) : void
+ contains(Predicate p) : boolean
+ toString() : String
PlannerStack
- stack : Stack<Stackable>
+ push(Stackable s) : void
+ pop() : Stackable
+ isEmpty() : boolean
+ translateParameters(String value, String transl) : void
+ toString() : String
Operator
- preconditions : List<Predicate>
- adds : List<Predicate>
- deletes : List<Predicate>
+ instantiate(Predicate p) : Operator
+ apply(State s) : void
+ toString() : String
Predicate
+ toString() : String
1..N
State
+ applyOperator(Operator op) : void
+ removePredicates(List<Predicate> l) : void
+ toString() : String
initialState
goalState
LinearPlanner
- availablePredicates : List<Predicate>
- availableOperators : List<Operator>
- plan : List<Operator>
- currentState : State
- finalState : State
- intelligence : Intelligence
- stack : PlannerStack
+ executePlan() : List<Operator>
+ logStack() : void
+ logPlan() : void
+ logSteps() : void
plan
uses
creates
LinearPlannerBuilder
- availablePredicates : List<Predicate>
- availableOperators : List<Operator>
- plan : List<Operator>
- initialState : State
- finalState : State
- logOutput : PrintStream
+ addPredicates(List<Predicate> l) : void
+ addOperators(List<Operator> l) : void
+ setInitialState(State s) : void
+ setFinalState(State s) : void
+ setLogOutput(PrintStream ps) : void
+ build() : LinearPlanner
- isBadlyDesigned() : boolean

Figure 2: Class diagram of the whole project

Having explained the common and the specific part, the best way to organize the functionality of this program is to hold a STRIPS package, which contains the generic linear planner; and a coffeeServer package, with the specific part of this problem.

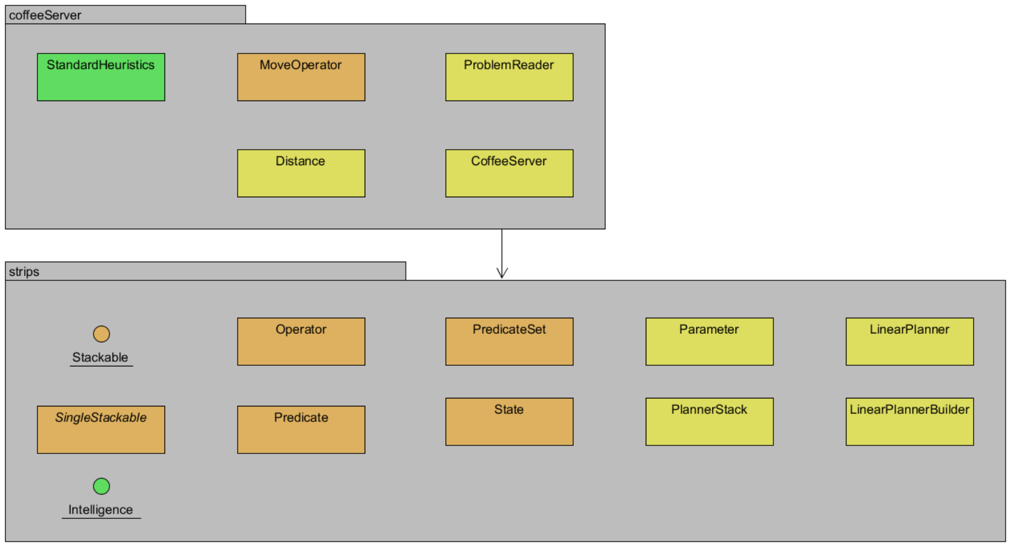


Figure 3: Package diagram of the project

5 Testing cases and results

Thanks to the *ProblemReader*, any CoffeeServer problem can be written and read from the program. One just needs to write the initial and final state, following the syntax indicated in the exercise. Specifically, we have created 5 new problems (and the example one). Their main goal is to test the efficiency of the method, in terms of optimality of the solution and execution time.

5.1 Optimality of the problem

The STRIPS linear planner does not achieve an optimal solution, but we can try to improve its results by using good heuristics. We explained our heuristics in previous sections, and here we present how they work. We will do this by comparing the results obtained for the example problem.

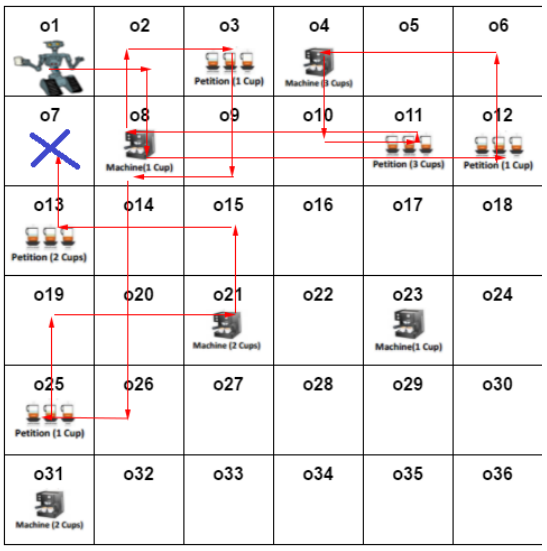


Figure 4: Example problem and the sub-optimal path found

The steps taken by the algorithm are 29. We know that this is sub-optimal, as it is higher than the number of steps indicated in the exercise definition.

Our conclusion is that our heuristics may be improved by taking into account not only the final position of the robot, but also the starting position, so as to organize the petition orders in a logical way for minimizing displacements.

5.2 Performance metrics

The number of steps taken to solve the problem is not the only measure we should try to minimize. Given that the algorithm iterates over the stack in order to apply operators and build a plan, there are multiple ways to get to a plan in a specific number of steps. And some of these plans are more efficient than others (they require a least amount of iterations of the algorithm). This is why we are also looking at the number of iterations to complete the problem. The number of iterations is proportional to the execution time, which is another measure we are trying to optimize.

Now we are presenting the 5 problems created for testing the planner. They have an increasing number of petitions, which is the predicate directly related to the complexity of the solution. The following image shows the initial state of every problem, as well a blue cross in the final position they must achieve. Please note that the definition of these problems in text files is found in the project, as well as their logs. For formatting reasons, you may find them in the Annex 1 of this document.

The performance results for these problems are presented in the following table, in the measures of number of steps, iterations and execution time in milliseconds.

Table 2: Table of performance results

Problem	**P1 (4)**	**P2 (6)**	**P3 (10)**	**P4 (20)**	**P5 (31)**
Steps	42	65	42	149	180
Iterations	82	120	196	386	595
Time (ms)	101	113	159	332	647

The number of steps depends on the disposition of the petitions themselves, so it is not a good measure for general performance of the algorithm. We will focus on the iterations and the execution time. The following graphics show their evolution.

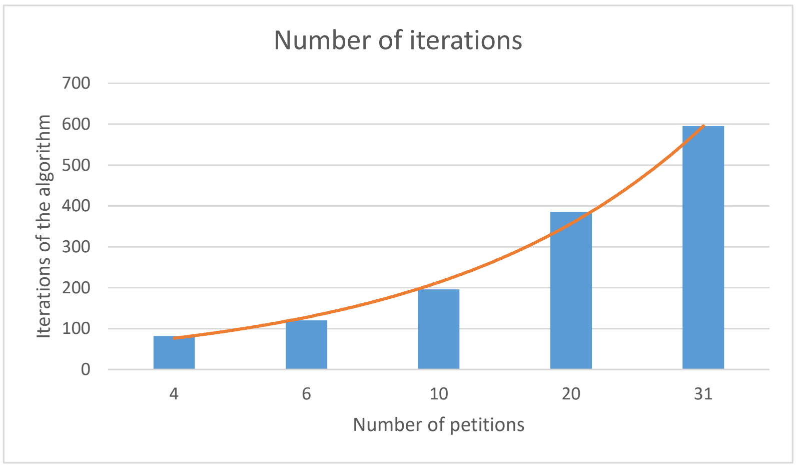


Figure 5: Evolution of the iterations as the petitions increase

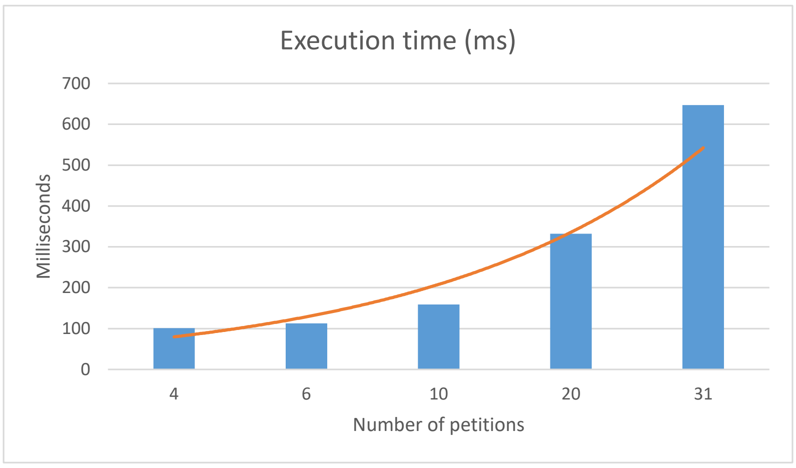


Figure 6: Evolution of the time as the petitions increase

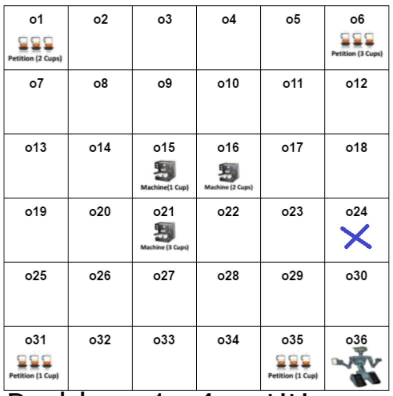
In both cases (which are profoundly related) the values follow an exponential trendline. This leads us to think that this method becomes impracticable for problems bigger than the ones for the coffee server, which are limited by the grid size.

6 Instructions to execute the program

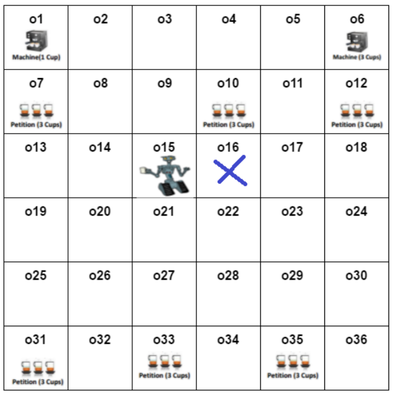
You can find attached to the delivery a *src* folder. It contains all the source files needed to execute the program, so you can just add this src folder to an Eclipse project if you want to reproduce it.

For just executing the program, you have a *jar* file in the delivery. It requires an argument to work, which is the path of the problem to solve. As the problems are in the same folder, you can just run *java -jar planner.jar problem1.txt*, if you want to run the problem1.txt. Then you will write the log into the log_problem1.txt.

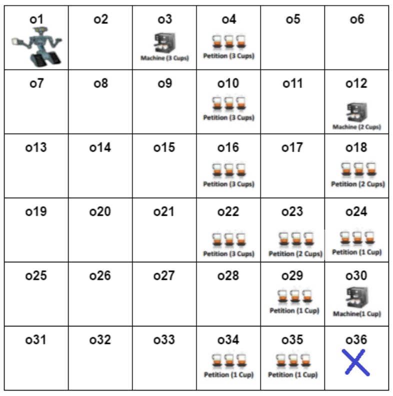
7 Annex 1: definition of the test problems



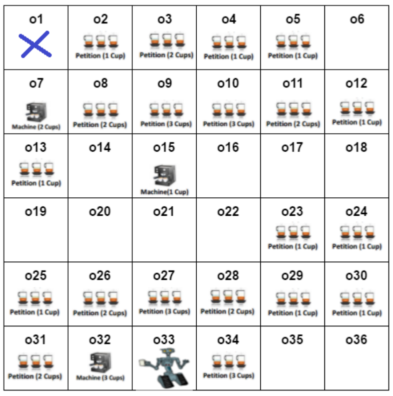
Problem 1: 4 petitions



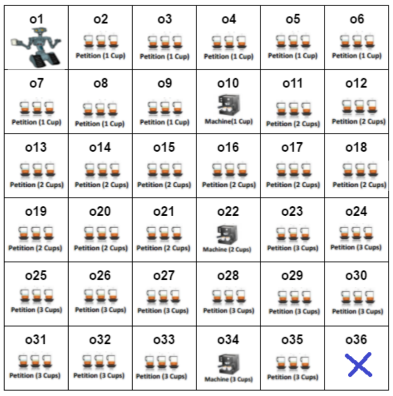
Problem 2: 6 petitions



Problem 3: 10 petitions



Problem 4: 20 petitions



Problem 5: 31 petitions

Figure 7: Definition of the 5 test problems