Practical Exercise 1: Planner

The coffee server

Planning and Approximate Reasoning

Universitat Rovira i Virgili

Master in Artificial Intelligence

1^{st} Semester

Authors:
Javier Beltrán
Jorge Rodríguez

November 5, 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 to not disturb the people working).

We start with a 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 machines position (no more petition appears). The way of achieving this final state is by going to the machines, take the coffee and them going to each petition to leave the coffee.

To solve this problem we will implement a general planner which later will be use 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 them finding the operators needed to get this predicates. The implementation of this has been done in Java, this will be explained in detail bellow.

As we mentioned before to formalise this 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 Table1 all this together 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 show 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 that 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

In this section we are going to analyse the problem. First of all we notice that to the given initial state we have to add robot-free, as to the goal state.

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 form 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 list 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.

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.

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).

3 Planning algorithm

4 Implementation design

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 $get(inti)$, 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:

(image of stackable elements)

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 in the right order. preventing loops.

(full class diagram)

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.

(package diagram)

5 Testing cases and results

6 Instructions to execute the program