Appendix B

Automated Planning in Brief

This is a brief, summarized guide on Automated Planning, based on how it was implemented in this thesis. It is based on Malik Ghallab's implementation in the book *Automated Planning: Theory and Practice* [25], but has been modified to fit this thesis. Where necessary, code examples will be used. This explanation will be based on the Dock Worker Robot simulation of the same book. This is a version of Hierarchical Task Network planning, called Simple Task Network.

B.1 How To Represent Your Planner Model

The first consideration when deciding to use an Automated Planning system is how you will represent knowledge in your code. This project is based on the Classical Representation, which is derived from first-order logic. This was chosen because of it's implementation in the Artificial Intelligence course in DT228 Year 4, so most readers should be familiar with it. For those who aren't, there is a brief introduction to all important in the next few sections.

B.1.1 Propositional Logic

Propositional logic is a way of formally defining the first order language for your system. It has sentences, or *propositions*, that define how objects in the system are related, and these propositions also have *semantic data* that indicate whether a query based on a rule is true or not. It also has *theories* that can be used to derive other, new statements from existing ones.

For example, one possible proposition may be $wet \rightarrow raining$, which is read "'If it it wet, it implies that it is raining"'. The semantic data of this would indicate whether the proposition is true or not, so if it is both wet and it is raining, we can say that proposition is *true*. Beyond that, we can also apply theories to the sentences such as **modus ponens**, which will result in a new proposition if it applies.

Propositions are written as a lowercase p. The sum of all the propositions in the system are the *set* of propositions, indicated by an uppercase P. The first element of the set P is designated p_0.

We can also use *operators* on propositions;

$\neg$: the negative of a term; if we write $\neg wet$ we are saying that it is not wet.

$\wedge$: the AND operator; if we write $wet \wedge raining$ we are saying that it is wet and it is raining.

$\vee$: the OR operator; if we write $wet \vee raining$ we are saying it is either wet, or it is raining.

There are many other elements to Propositional Logic not mentioned here; they are out of scope for the needs of creating a similar planner to this project.

B.1.2 First Order Logic

Although propositional logic is simple and easy to understand, it can be difficult to express exactly how you want to model your system with it. To remedy this, First Order logic can be used to define your system in a way that captures it's complexity.

The smallest object in first order logic is a *term*. A term is simply a building block for creating a first order sentence. A term could be "truck", "blue", "John", or whatever you decide. The sentence, or *atom* as it is known in first order logic, is a combination of your terms, and some statement about them, called a *predicate*. For instance, in the atom $inRoom(Jane, DiningRoom)$, $inRoom()$ is our predicate, and $(Jane, DiningRoom)$ are our terms. We would read this atom as *Jane is in the room called DiningRoom*. This statement may be true or false; it is up to your system to figure out the truth of the statement. For instance, if it were a game we were modelling, we would perform a check to see which room Jane is in now; from there, we can then say whether the atom is true or false. We can indicate the truth value using the $\neg$ operator from propositional logic.

Again, for the sake of brevity, the remaining elements of First Order logic have been excluded as they are not needed for this project.

B.2 Setting Up Your Planning Domain

The first order of business of creating an Automated Planner is to figure out how to define a *state*. A state is a set of *atoms* that define how the objects in your model relate to each other. For this project, I had atoms that declare locations, status, if the agent can see the player, and others. For the purposes of this guide, though, we will use the Dock Worker Robot simulation defined by Ghallab et al as see in Figure B.1.

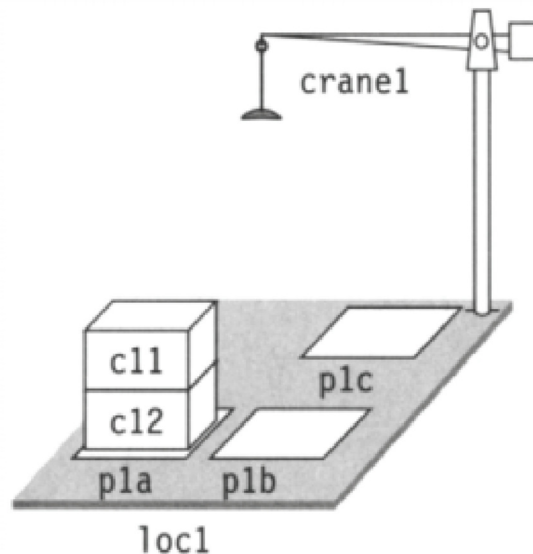

Figure B.1: The Dock Worker Robot simulation

Using atoms, we can define the state s_0, or the current state, as such:

$s_0 = attached(p1a, loc1), attached(p1b, loc1), attached(p1c, loc1), in(p1a, c12), in(p1a, c11), top(c11, p1a)$

Then, every time something in the world changes (say we pick up a container with the crane), we update the s_0 to reflect the changes. This can be done by having an update routine that simple interrogates each object in your system (in this case, the crane), finds out it's current status, and then writes a new atom to reflect the changes. In my thesis I used strings to store the atoms, and had a State object with a list of Atoms that reflected my state. On updating the state, I simply erased the current set of atoms to flush out any old ones, and interrogates all the objects to create the new atoms. This could be somewhere where an improvement can be made - maybe keep the atoms that haven't changed.

B.2.1 Operators

Next, you need to define the primitive operations that can be performed in your system. This is more design than anything, as you need to be able to model the preconditions

that have to be true in order for your operator to be used, and model the effects that will happen after it is executed. For instance, in th DWR domain, the `take` operator is :$take(crane, location, containerA, containerB, pile)$. This is read as *take containerA off of containerB in pile, using crane in location.*

Our preconditions are the set of atoms that must be true to perform this operation, so in order to take a container, we have to ensure the crane isn't already holding a container ($empty(crane)$), the crane actually belongs to that location ($belong(crane, location)$) and the pile is in that location too ($attached(pile, location)$), and the containerA is actually the top container in the pile ($top(containerA, pile)$).

If all these preconditions are fulfilled, we can then say that the operation is *applicable* in this state. Our effects, then, allow us to "see into the future", and tell exactly how the system will end up when the operation is performed. For the take operation, it means that the crane is no longer empty but holding containerA ($\neg empty(crane), holding(crane, containerA)$) and containerB is now at the top of the pile ($top(containerB, pile)$).

In this thesis, the Operator class held a list of terms of the objects it was concerned with, a list of atoms representing the preconditions that needed to be true, and a list of effects so we can update the world state after it is performed. The Operator class had functions that would take in a list of terms and was predefined how to update it's preconditions and effects depending on it's name. When the project checked to see if an operation was applicable, I would compare the atoms for each operator against the current state; the operator would only be applicable if all it's preconditions were true. If they were, I simply called the function `take` and passed into it the terms from the Operator, and then update the world state again to reflect the changes. This solution isn't very elegant, but it worked fine.

B.2.2 Tasks and Methods

At this stage, we actually have a planner, and can implement logic to call the operators in whatever order we want in order to fulfill a plan. The problem, though, is that we have to explicitly define each and every primitive function call in order - we can't just say "move this pile over there", but must detail out all the `take` and `put` operations manually.

Wouldn't it be more fun if we just told it to move them, and it figured out the best way itself?

That's where *tasks* and *methods* come in. A *task* is something we want to do, whether is simple like a take operator, or complex like moving a stack of containers but keeping

the ordering, and a *method* is the blueprint for our planner to figure out the best way to complete a task. Tasks and Methods are related, because each method has it's corresponding task that it *decomposes*, or breaks down into smaller tasks.

Tasks have terms like operators. They can be primitive, which means the task has the same name as an Operator. This means that, in order to complete that task, we simple call the operator function to perform it. However, we can also have tasks that are complex. Complex tasks have lists of subtasks to perform in order to complete them. One thing to be aware of; your subtasks can be complex too! All this means is that each subtask has to be broken down too.

Methods are our instructions for performing a task. Think of them as the recipe, and a task as the cooking. Methods store the name of the task that the decompose (say pancakes). They also have preconditions that work the same was as operators' preconditions do, which would be making sure we have enough eggs and flour. The most important part of a method is it's subtask. These would be the actual instructions, like "crack 2 eggs" and "heat the pan" and so forth. So, in order to complete the task "make pancakes", we have to check we have all the ingredients, then follow the instructions.

In our Dock Worker Robot simulation, say we got tired of having to give 2 instructions (a take and a put) to move a container from one pile to another. We could create a task called *moveTopContainer*, which would do them both. MoveTopContainer isn't an operator call, so it's *complex*. MoveTopContainer therefore would have a method related to it that would describe how to achieve this task, by breaking it down into primitive operators; let's call our method *takeAndPut*.

TakeAndPut will have it's list of preconditions that need to be true in order to perform the task, so we check to see if the crane is empty again etc. Once all our preconditions are true, we can say the task is *applicable* in the current state, and can start to break it down using the method's subtasks. These can be either primitive or complex; if primitive, we perform function calls, passing in the terms of the task where appropriate. If complex, we then have to lookup the method to perform that subtask.

It's possible that a subtask may be the same as the task itself. Say we want to move a stack from one pile to another. To do this, we can create a task called `moveStack`, and a method called `recursiveMove` that describes how we perform it (the naming will soon become apparent). In our method, we have the subtasks that need to be performed in order to move the stack - we need to move the top container of the pile again and again until there are none left. In this case, our first subtask for `moveStack` would be `moveTopContainer`. Next, we can simply call the `moveStack` task again, but on the remainder of the pile! This mean we move the top of the current pile, and then move

the top from the remainder, and again, until the whole pile is gone. If this is hard to imagine, Figure B.2 is a network diagram of how the tasks are performed, and is read left to right.

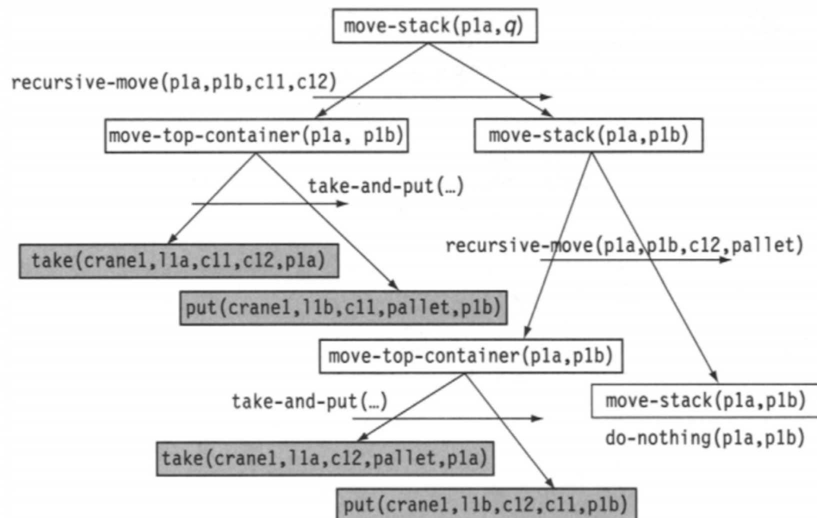

Figure B.2: The moveStack operation

Yep, network - although it didn't seem like it, we were building a tree of tasks to perform, and each layer in the tree is a subtask that has to be performed. What you should end up with is a list of primitive operator calls at the leaf nodes of your tree - from left to right ordering, these are now the actions that need to be performed to complete the task. Excellent!

This is an example of a total-order forward search, and has some really cool benefits and side effects:

- Our planner only considers actions whose preconditions are satisfied in the current state, and we only consider actions that will lead us to the goal. We're basically searching the tree from the front and back at the same time.
- We generate the actions in the same order in which they'll be executed, which means that every time we plan how to accomplish a task, we've already worked out everything that needs to take place beforehand.

And that's it! I hope from this brief guide that the fundamentals of automated are clear, and you can build your own system!