kazirhut.com

Planning

Chapter 7

An agent interacts with the world via perception and actions (Figure 7.1). Perception involves sensing the world and assessing the situation. In sophisticated systems, it may involve creating some internal representation of the world, which we can call a mental model. Actions are what the agent does in the domain. Planning involves reasoning about actions that the agent intends to carry out. "*Planning is the reasoning side of acting*" (Ghallab et al., 2004). This reasoning involves the representation of the world that the agent has, as also the representation of its own actions.

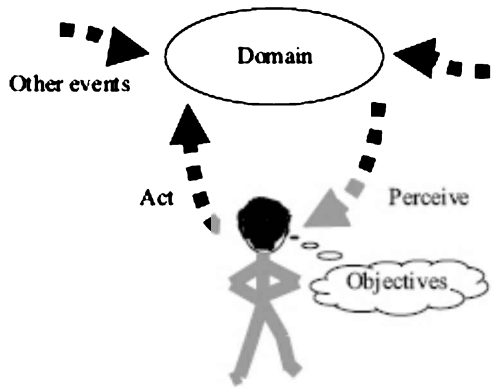

Figure 7.1 A planning agent can perceive the world, and produces actions designed to achieve its objectives. In a static domain, the agent is the only one who acts. A dynamic domain can be modelled by other agencies that can change the world.

The objectives of planning are to achieve some desired situation. The objectives could be certain properties of the final state after all actions have been completed, or even along the way in the sequence of states. These could be strict requirements or hard constraints where the objectives *have to* be achieved completely for success. The objectives could also be soft constraints, or *preferences*, to be achieved to the extent possible.

The planning problem can be described or posed at varying levels of detail. As the detail increases, so does the complexity of the computation. In the simplest planning problems, the domain is static, the agent has complete information of the domain (perception is perfect), actions are instantaneous and their effects are deterministic. The agent knows the world completely, and it can take all facts into account while planning. It is the only one changing the world, and therefore does not have to worry about the world changing while it is planning or executing a plan. The fact that actions are

kazirhut.com

kazirhut.com

instantaneous implies that there is no notion of time, but only of sequencing of actions. The effects of actions are deterministic, and therefore the agent knows what the world will be like after each action. Each of these constraints can be relaxed to define richer planning domains, but in which planning is a computationally harder task.

The algorithms being developed in the planning community are all based on explicit representations of actions in a domain. This is a little different from our earlier view of search which was more abstract and tended to assume a move generating function. Representing actions explicitly allows us to explore richer domains, for example when actions have durations. It also allows us to investigate approaches to planning different from graph search.

In *classical planning*, reasoning about actions involves simulation of changes in the mental model that the agent has. The agent peers into the future to examine the different possibilities and select the desirable course of action. This is called *projection* into the future. The planner searches through the possible combination of actions to find the *plan* that will work.

An alternative approach is to exploit the agent's experience, looking into the past. If the agent has memory, it can reuse a plan that worked earlier, or reuse the problem solving experience to find a solution. The agent can fish out (retrieve) a plan or planning cues from its memory. Such an approach is known as *memory based planning*. It involves representation and reasoning over the events in the past. Human beings typically use both methods. If a problem is familiar then a known solution is used from memory. The idea is to not reinvent the wheel every time. However, when a problem is a new one, the first principles projection based methods have to be used. An efficient agent, for example an intelligent human being, will keep learning from experience, as and when new problems are encountered and solved. We will explore memory and knowledge based techniques later in the book (see Chapter 15). For the moment, we focus on how to find plans from first principles, by projecting the effects of candidate actions into the future.

In order to explore planning algorithms in a domain independent form, a standardized notation for describing the world and also the actions has been developed. This is known as the *Planning Domain Description Language* (PDDL) (McDermott, 1998). PDDL allows one to define the language in terms of which the world will be described, as well as the actions that the planner is capable of, expressed in terms of schemas. The language is based on First Order Logic notation (see Chapter 12). It also allows one to state a specific planning problem by describing the current state and the objectives or goals using the constructs defined in the schema. The planning algorithms can then be written in a domain independent form. The domain and problem descriptions[1] will become inputs to the domain independent algorithms.

The complexity of planning will depend upon the expressiveness of the PDDL and the constructs defined. Starting with the simplest planning domain and problems, a series of more expressive PDDL languages have been defined (Fox and Long, 2003; Edelkamp, 2004; Gerevini and Long, 2005). In each new language, the world and the actions can be described in a richer fashion, taking more and more aspects from the real world problem. However, even the simplest of planning domains—known as *STRIPS*

kazirhut.com

kazirhut.com

domains described below—pose problems that are computationally hard to solve (Gupta and Nau, 1992). This is not surprising, given that they are basically viewed as search problems in the classical approaches. Nevertheless, there has been considerable interest in planning, and we shall look at some of the vast gamut of techniques that have been developed in the area.

7.1 The *STRIPS* Domain

Planning refers to the choice of moves an agent makes before making the moves in the domain. It involves synthesizing a set of actions, often arranged in a linear sequence, but sometimes also in a partial order. A partial ordering could mean that any linearization is a valid plan. In some domains it could also mean that some actions can be done in parallel. When the action sequence is executed in the domain, it has the effect of transforming the world from a given state into a new state, which if the plan works, is a desired or goal state.

The search algorithms we have seen in the preceding chapters can also be used for planning. The algorithms assume that the state description is available in a compact form, and that generated moves produce new modified states. In practice, the state will be described by its components, and a database describes the world in a given state. Consequently, we also amend our notion of a move. Instead of viewing moves as state transitions, we will view them as *operators* that transform a part of a state (see also Chapter 6). Each operator looks for a pattern in the state and, when applied, makes some changes in the state. In doing so, it does implement the state transition function. But the same operators may be applicable over many states, leading to a more compact representation. The operators are close to the idea of production rules, being applicable when the associated preconditions hold. An *action* (or move in the state transition view) is an instance generated from an operator.

The first program to use this formalism was *STRIPS* (Fikes, 1971)[2]. The program was designed to plan in the domain of blocks world. In this domain, a set of labelled blocks are in some configuration to start with. A block can be on another block, or a block can be on the table, and the table has unlimited capacity. The objective of the planner is to rearrange the blocks into a desired (goal) configuration. The output of the planner is the set of moves that brings about this desired rearrangement. Figure 7.2 illustrates the domain with an example problem.

kazirhut.com

kazirhut.com

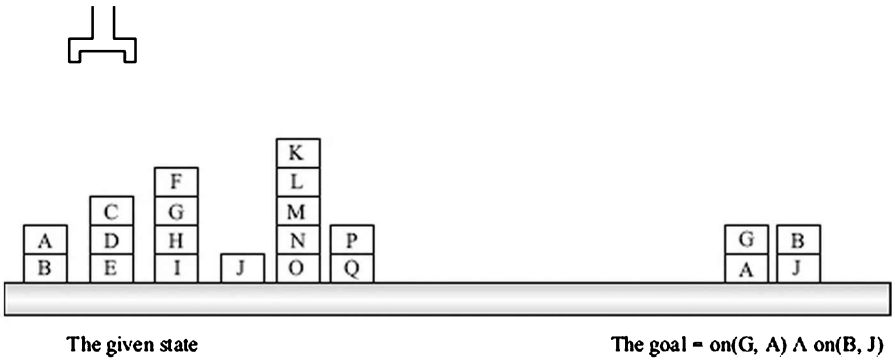

Figure 7.2 The blocks world domain. Observe that the goal state is incompletely specified. The given state has to be specified completely.

The world, in the blocks world domain, is described by a set of statements that conform to the following predicate schema:

on(*X*, *Y*)	:	block *X* is on block *Y*
ontable(*X*)	:	block *X* is on the table
clear(*X*)	:	no block is on block *X*
holding(*X*)	:	the robot arm is holding *X*
armempty	:	the robot arm is not holding anything

One can make the following observations about the description. There are no metrics involved. A block can have only one block on it, and there is no notion of size, weight or location co-ordinates. We assume an arbitrarily large table with no space constraints. For all purposes, the blocks are all of the same size and shape. They can be handled identically, and only one can rest on another. The one-armed robot can hold one block. The description is essentially qualitative in nature and does not involve numbers in any form.

The representation borrows from logic, specifically first order logic. Unlike in classical logic, however, predicates in planning such as *on*(*C, D*) can be true at one time instance, and false at another. Such predicates are also known as *fluents*. It was natural that the first approaches to planning too borrowed from logic, and viewed planning as an exercise in theorem proving (Green, 1970).

Classical reasoning with first order logic does not allow for assertions to be withdrawn. Once an assertion is true, it remains true. There is no notion of time and change. For example, the Pythagoras theorem is true, independently of time. This is fine in the world of mathematics. In a world where an agent is trying to bring about change, one has to have a mechanism for representing situations. For example, in the start state in the illustration above *on*(*C, D*) is true. If I pick up (or *unstack*) *C* and place it on *J* then *on*(*C, J*) will become true. What shall one do with the assertion *on*(*C, D*), now that it is no longer true? One way to handle the problem is to introduce time as a parameter, and have every predicate time stamped. Then *on*(*C, D,* t_0) would mean that block *C* is on block *A* at time t_0. Then one can have another fact *on*(*C, J,* t_2) that states that *C* is on *J* at time t_2. At time t_1, the arm is holding *J*, represented by *holding*(*J,* t_1). One would also have to

kazirhut.com

kazirhut.com

describe goals using variables for time. In Figure 7.2 for example, one would have to describe the goal as "there is some time *t* at which *on*(*G, A, t*) and *on*(*B, J, t*) are simultaneously true."

However, this creates an enormous book-keeping problem of carrying forward facts as time moves on. For example, *ontable*(*B*) and all other unchanging facts will need to be asserted afresh as being true at each time step. How do we know that if a fact like *ontable*(B, t_1) is true then *ontable*(B, t_2) will be true as well? This is the well known *Frame Problem* introduced by John McCarthy (McCarthy, 1969)[3]. This problem can be solved by adding the so called "frame axioms", which explicitly specify that all conditions not affected by actions are not changed while executing that action. Thus, the frame axioms need to be applied at each stage to keep track of what is true as time marches on. We will look at frame axioms in Chapter 10.

The ingenious solution introduced in *STRIPS* was to make *all change explicit*. Thus, one was saved the bother of asserting facts all over again, each time an action was added to the plan. This is the approach a cartoon animator would take making cartoon films. She would take (copy) the previous frame, delete a few strokes, and add a few others to create the next frame. Whatever was not changed remained the same. *STRIPS* operators follow this strategy, and are made up of the following three components:

P: Precondition List The assertions needed to be true for the operator to be applicable.

A: Add List The assertions that became true as a consequence of the operator being applied.

D: Delete List The assertions that are no longer true as a consequence of the operator being applied.

The set of assertions in the database at any time describe the world as it is. When an instance of an operator is applied to a database of assertions, some assertions may be deleted, while others may be added. Fikes introduced the following operators for the blocks world domain.

PICKUP (*X*)
  P: ontable (*X*) ∧ clear (*X*) ∧ armempty
  A: holding (*X*)
  D: ontable (*X*) ∧ armempty

PUTDOWN (*X*)
  P: holding (*X*)
  A: ontable (*X*) ∧ armempty
  D: holding (*X*)

UNSTACK (*X, Y*)
  P: on (*X, Y*) ∧ clear (*X*) ∧ armempty
  A: holding (*X*) ∧ clear (*Y*)
  D: on (*X, Y*) ∧ armempty

STACK (*X, Y*)

kazirhut.com

kazirhut.com

P: holding (X) ∧ clear (Y)
A: on (X, Y) ∧ clear (X) ∧ armempty
D: holding (X) ∧ clear (Y)

Notice that when one picks up a block X, either with *STACK* or *PICKUP*, the operators do not delete the assertion *clear(X)*. When we put it down, the assertion will already exist in the database. This will not cause a problem with the reasoning, and in fact save some computation, and it is anyway not clear what *clear(X)* would mean when it is being held.

An encoding of the blocks world domain in PDDL is given in the accompanying Box 7.1. Observe that the notation used there combines the *add list* and the *delete list* into a common set of *effects.* The facts from the delete list are preceded by a "not", to depict that the fact is not true in the resulting state. This is consistent with the PDDL standard that evolved later. Also shown is the coding of a small planning problem.

Another important observation is that the goal description may not describe the final state completely. In the above illustration in Figure 7.2, nothing is said about the location of blocks *A* and *J.* Likewise, it is not stated what if anything is on *B* and *J.* There can be many states in which the given goal conditions will be true. As we will see, this also implies that backward reasoning will have to deal with incomplete state descriptions.

Definition

We say that a state S *satisfies* a goal G iff $G \subset S$.

Box 7.1: The Blocks World Domain in PDDL

The blocks world domain may be described using a typed version of PDDL (PDDL3.0) which is given below. This also introduces the notion of variable types. Also note that add list and the delete list from the *STRIPS* notation is combined into an effects list.

```
(define (domain Blocks)    /* Blocks is a domain dealing with block types */
     (:requirements :typing)
     (:types block)

     (:predicates  (on ?x - block ?y - block)    /*The predicates used for
                   (onTable ?x - block)        describing the situation */
                   (holding ?x - block)
                   (clear ?x - block)
                   (armempty))
```

kazirhut.com

kazirhut.com

```
 /* Given below are the four kinds of actions defined in the Blocks
domain */
     (:action pickup
            :parameters (?x - block)
            :precondition (and (onTable ?x) (armempty) (clear ?x))
            :effect (and (not (armempty)) (holding ?x) (not (onTable ?x))))

     (:action putdown
            :parameters (?x - block)
            :precondition (and (holding ?x))
            :effect (and (not (holding ?x)) (armempty) (onTable ?x)))

     (:action stack
            :parameters (?x - block ?y - block)
            :precondition (and (holding ?x) (clear ?y))
            :effect (and (not (holding ?x)) (armempty) (on ?x ?y) (not
             (clear ?y))))

(:action unstack
            :parameters (?x - block ?y - block)
            :precondition (and (on ?x ?y) (clear ?x) (armempty))
            :effect (and (not (on ?x ?y)) (holding ?x) (clear ?y) (not
             (armempty))))
)
```

We now look at some planning algorithms in the *STRIPS* domain. The search algorithms developed in the preceding chapters can easily be adapted to the task of planning. We do so in the next section. After that, we look at other ways of searching for a plan. We look at the possibility of starting the search from the goal description. This is known as backward reasoning/search. We then describe a way of combining the good features of both, forward and backward search, into an algorithm called Goal Stack Planning. We also look at an approach that seeks to fill in the actions constituting the plan in a nonlinear order, filling in actions as and when we spot their requirement. Such an approach is also described by terms like *Plan Space Planning* and *Partial Order Planning*. Following this, we look at some flavours of hierarchical planning that have been developed. In subsequent chapters, we introduce the concepts of domain independent heuristics and look at their applicability in the planning algorithms discussed here.

7.2 Forward State Space Planning

Given a state, and given a set of operators, one can determine the actions (or moves) that can be applied in the state to generate successor states. This corresponds to the implementation of the *moveGen* function described in Chapter 2. Forward State Space Planning (FSSP) then refers to the search algorithms that start with the given state as the start state, generate the set of successor states, and search through them generating more successors till they find a state that satisfies the goal conditions.

Box 7.2: A Blocks World Problem in PDDL

kazirhut.com

Figure 7.3 below depicts a planning problem from the blocks world domain. A description of the planning problem in PDDL follows. Observe that the domain field refers to the domain description named Blocks.

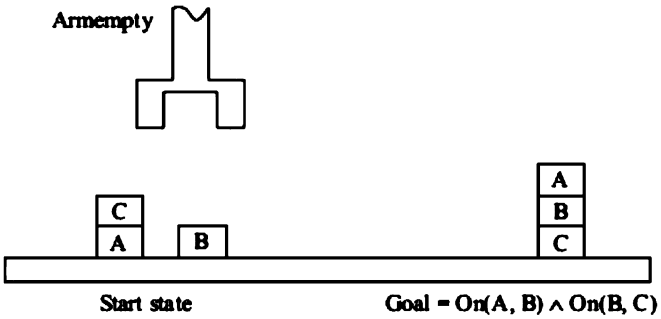

Figure 7.3 A three block planning problem.

The description in PDDL.

```
(define (problem blocksProblem1)
        (:domain Blocks)
        (:objects
                A - block
                B - block
                C - block )
        (:init
                (onTable A)
                (onTable B)
                (on C A)
                (clear B)
                (clear C)
                (armempty))

        (:goal (and (on A B) (on B C)))
)
```

Let *effects(a)* denote all the effects of action *a.* Let $effects^{+}(a)$ denote the set of positive effects of action *a.* This is the same as the add list in the *STRIPS* notation. Corresponding to the delete list, we have a similar set denoted by $effects^{-}(a)$. Then given a state *S* in which the action *a* is applicable, the state *S'* after action *a* is applied is given by,

$$S' \leftarrow \{S - effects^{-}(a)\} \cup effects^{+}(a)$$

That is, from the sets of facts representing *S*, we delete the facts that are

not true after the action, and add the facts that become true, resulting in the new state *S'.* We say that the state has *progressed* through the action *a.* Let us define a function *progress(A, S*) that returns the successor state when action *A* is applied to state *S.*

Given the *progress*(*A, S*) function, the search algorithms in Chapter 2 can easily be used. They will, however, suffer from the same drawbacks. The main one being that the search space generated will be huge, and that the search algorithm will have no sense of direction. The given state in Figure 7.2 is described by the following collection of facts:

{*AE*, ontable(*B*), on(*A*, *B*), clear(*A*), ontable(*E*), on(*D*, *E*), on(*C*, *D*), clear(*C*), ontable(*I*), on(*H*, *I*), on(*G*, *H*), on(*F*, *G*), clear(*F*), ontable(*J*), clear(*J*), ontable(*O*), on(*N*, *O*), on(*M*, *N*), on(*L*, *M*), on(*K*, *L*), clear(*K*), ontable(*Q*), on(*P*, *Q*), clear(*P*)}.

In the given state, one instance of the *Pickup* operator is applicable, generating the action *Pickup*(*J*). Several instances of the *Unstack* operator are generated. Each of the actions does a small change in the world, and many actions are applicable in turn in the changed world. Figure 7.4 illustrates the fact that the search space generated by FSSP is huge. Looking at the full problem, one can notice that it includes actions that have nothing to do with achieving the goal.

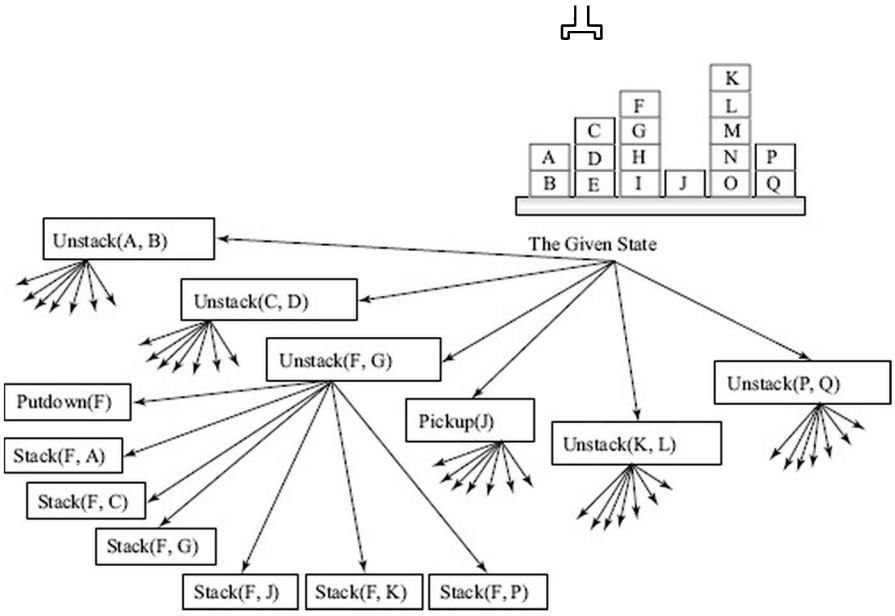

Figure 7.4 The search space generated by FSSP is huge.

7.3 Backwards State Space Planning

Like the uninformed search algorithms seen earlier, FSSP is oblivious of the goal, and simply explores the set of all future states possible in some

predetermined order. One way to take the goal state into account is to start searching backwards from the goal. That would make the reasoning purposeful and deliberative, focused towards the goal, or teleological. This is known as *Backward State Space Planning* (*BSSP*). Backward reasoning or goal-directed reasoning has been proposed as being fundamental to intelligent behaviour, and we have already visited it while studying goal trees in Chapter 6. It is also the foundation of reasoning with logic, and the backbone of the logic programming language Prolog (Kowalski, 1974; Colmerauer, 1992). We will explore logic and reasoning later in Chapter 12. At this moment, let us focus on the possibility of backward state space planning.

One feature strongly in support of such an approach is the fact that goal states are often incompletely specified. That is, one often expresses only what is desired in the final state, rather than a complete description of the final state. For example, at some point of time, one might have the goal of satisfying hunger. Reasoning in a backward fashion, one will only focus on actions that will result in eating something, without worrying about anything else. Similarly, the goal in the planning problem of Figure 7.2 is that block *G* should be on block *A*, and block *B* on block *J*. Can we focus only on the moves that will achieve these conditions? To do this, we have to first define the notion of regression. This, in some sense, is the opposite of progression. It allows us to move back from a set of goal clauses to a set of subgoal clauses. For example, if in a regressed state the robot were holding block *G*, and block *A* was clear, and block *B* was on block *J*, then the *stack*(*G, A*) action would achieve the goal state of Figure 7.2.

Given a goal *g*, we say that an action *a* is *relevant* to *g* iff[4]

$$(g \cap effects(a) \neq \varphi) \gg (g \cap effects^{-}(a) = \varphi)$$

Let *preconditions*(*a*) denote the preconditions of action *a*. Given a goal *g*, and a relevant action *a*, one can regress to the goal *g′* as follows.

$$g' \leftarrow \{g - effects(a)\} \cup preconditions(a)$$

That is, from the set of goal facts, remove the effects of the action, and add the preconditions of the action. Let us define a function *regress*(*A*, *G*) that returns the regressed goal over action *A* when applied to goal *G*. The regressed goal represents the minimum set of facts that must be true in a state in which the action can be applied and which would result in a goal state.

However, the regression process is not sound, in the sense that the resulting set of predicates may not necessarily represent a state. This can happen because some actions that are relevant according to our definition may not be feasible in practice. This may result in an invalid plan. Figure 7.5 depicts some of the problems that may crop up in backward search. The algorithm begins with the given goal description *on*(*G, A*) ∧ *on*(*B, J*) for the problem depicted in Figure 7.2. Observe that if a goal predicate is true in the current state then one does not need to look for a relevant action. Two actions are relevant here, *stack*(*G, A*) and *stack*(*B, J*). The former regresses to a goal *holding*(*G*) ∧ *clear*(*A*) ∧ *on*(*B, J*), from which four operators are

relevant, as shown in the figure.

The first, *pickup*(*G*), would need block *G* to be on the table. But since the algorithm is working with incomplete descriptions, it does not know whether this will be so. If block *G* were on some block then the way to end up *holding*(*G*) would be to unstack it from some block, but which one? One way would be to instantiate an instance of *unstack* for each possible block in the problem. One can see that this will soon lead to an explosion in the search tree. The preferred method is to keep it as a variable, and instantiate it later. Some action to achieve *clear*(*A*) would be needed if block *A* were not already clear. And again, the algorithm does not know what should be unstacked from *A* to achieve *clear*(*A*). Furthermore, in the action *unstack*(*?X, A*) required for *clear*(*A*), the variable *?X* can be bound only to *G*. This is because *unstack*(*?X, A*) also results in *holding*(*?X*), and the goal set requires *holding*(*G*) to be true. However, this connection between the requirements of the two predicates *holding*(*G*) and *clear*(*A*) requires some more sophisticated reasoning. Perhaps some kind of consistency check can force the binding of *?X* to *G*.

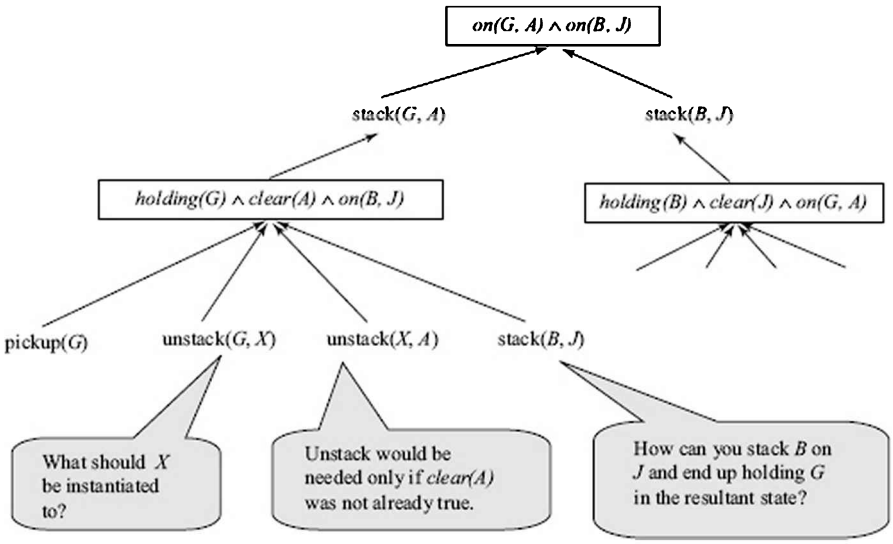

Figure 7.5 BSSP does not have a concrete state description to work with!

The last action *stack*(*B, J*) causes a more serious problem. It is meant to achieve the fact *on*(*B, J*). But one can notice that the robot arm could not be holding block *G* (asserted in the given goal) and block *B* (required for the stack action) at the same time. This shows that the algorithm *could be* reasoning with collections of facts that are not states. Figure 7.6 illustrates how a sequence of actions that is not a plan could be found by unhindered backward search. We consider a simple problem where in the given state three blocks, *A, B* and *C*, are on the table, and the goal is to stack *A* on *B*, and *B* on *C*. The figure shows a possible trace that backward search, using the regression process, might result in. The reader is encouraged to verify that at each stage of backward search the action chosen is relevant.

kazirhut.com

The reason why progression over states with planning operators works while regression does not, is that the operators are not symmetrical. They are designed to represent actions. An action converts a given state into a successor state. The applicability of the action is predicated only on the given state. If the given state satisfies the preconditions of the actions, it can be applied. Operators (or actions) have an in-built arrow of time, and the FSSP algorithm is aligned with it. Actions have preconditions on the given state which determine their applicability. The preconditions are necessary and sufficient for applicability. Once an action is applied to a given state, it produces a new state by adding some facts and deleting some facts. The point is that after progressing over an action, we necessarily land up in a state. One can say that the progression operation is *closed over the state space*.

Regression is done over sets of goal predicates. A goal represents facts that one wants true in some future state[5]. The relevance of an action to a goal simply means that *it looks like* that the action could achieve some part of the goal. But it does not mean that the action may be *applicable* in the preceding state. For example, for the goal in the above problem (Figure 7.6), the action *stack*(*B, C*) satisfies our definition of being relevant. But it obviously cannot be applied as a last action in a plan, because one can only move one block at a time, and block *A* needs to be on block *B*. The backward search algorithm being discussed could easily have found another invalid plan with *stack*(*B, C*) as the last action. While the goal description presumably applies to a state, one can observe that the regression operator is not closed over the state space. One may start with a (subset of *a*) state but end up with a set of predicates that is not the subset of any state. For example, the description "*holding*(*A*) $\wedge$ *clear*(*B*) $\wedge$ *holding*(*B*) $\wedge$ *clear*(*C*)" in the figure cannot describe a state, because our robot has only one arm and cannot hold two blocks.

kazirhut.com

kazirhut.com

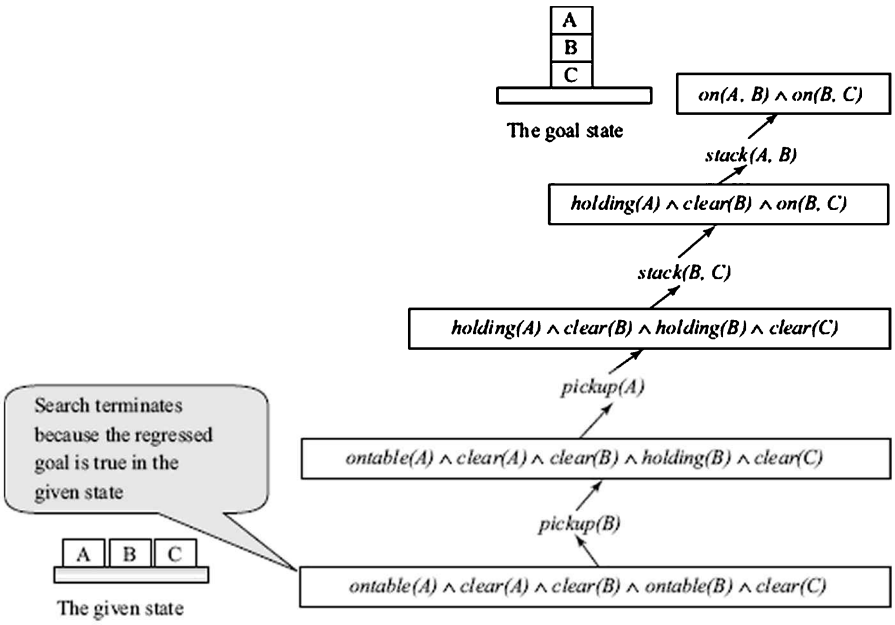

Figure 7.6 The plan *pickup(B)*, *pickup(A), stack(B, C), stack(A, B)* is not a valid plan.

Since we can have action sequences that are not feasible plans, let us define the notion of a valid plan.

Definition

A planning problem *P* is defined as a triple (*S*, *G*, *O*) where,

— *S* is set of facts completely describing the given or start state,
— *G* is a set of facts required to be true in a goal state, and
— *O* is the set of operators.

The above defines the simplest of planning problems. The simplest notion of a plan is a sequence of actions, each of which is an instance of some operator in *O*.

Definition

A sequence of actions $\Pi = (a_1, a_2, \ldots, a_n)$ is a *valid plan* for a problem *P*, if a_1 is applicable in the state *S*, progressing to state S_1, a_2 is applicable in the resulting state S_1, and so on, yielding the state S_n after the action a_n, and $G \subseteq S_n$.

A simple linear time procedure may be written to verify that a plan $\Pi = (a_1, a_2, \ldots, a_n)$ is a valid plan, by successively progressing through the sequence of actions and checking whether the final state contains the goal facts *G*.

kazirhut.com

kazirhut.com

```
ValidPlan(plan, state, goal)
  if Empty(plan)
      then if Satisfies(state, goal)
              then return TRUE
              else return FALSE
      else state ← Progress(Head(plan), state)
return ValidPlan(Tail(plan), state, goal)
```

Figure 7.7 Algorithm to verify a plan. Function Satisfies checks whether the state satisfies the goal (that is $G \subseteq S$). Function *Progress* progresses the state over an action.

Given the function to check the validity of a plan, we can write a backward state space planning algorithm that will apply this check before termination.

In the algorithm described below, we also *assume* a function *Consistent*(*G*) that takes a goal *G* and checks whether *G* can be part of a state. Observe that this check does not have to be perfect. We use the check to filter out some inconsistent goals. The more inconsistent goals we can filter out, the faster our search algorithm will perform. But for the sake of correctness, the check should not filter out any consistent goals. The trivial correct case is when *Consistent*(*Goal*) always returns *true*. In this case, it does not filter out any goals, which means the algorithm may waste time over an infeasible sequence of actions. Some work on checking for consistency has been described in (Fox and Long, 1998; Kumashi and Khemani, 2002).

In the algorithm below, the search node is a pair made up of the goal to be satisfied, and the current plan. We use two functions, *Goal*(*node*) and *Plan*(*node*), to extract the goal component and the plan component from a given search node. They return respectively the first member and the second member of the list representing the node. The plan grows in a backward fashion, with the last action being found first. The planning algorithm takes as input the given state, the goal description, and a set of actions that are all possible ground instances of the set of operators for the given domain. One can adapt the algorithm to work with the set of operators itself. Working with instances is simpler for a finite domain. Also, not shown in the algorithm, is the role of a CLOSED list to check for looping. This is left as an exercise.

7.4 Goal Stack Planning

The main problem with BSSP is that goal regression is not sound and the algorithm may be handling goal descriptions that are not consistent with any state. The advantage of BSSP over FSSP is that because it focuses on the goal, the search space it generates is smaller due to the branching factor being lower. FSSP, on the other hand, is sound but has a large branching factor, since it looks at modifications to the complete state.

One of the earliest planners *STRIPS* (Fikes, 1971) attempted to combine the focused search of BSSP with the soundness of FSSP. It considered actions by reasoning in a backward manner, but committed itself to actions only in a forward manner. The algorithm is also called *Goal Stack Planning* (GSP) (Rich and Knight, 1990) because it pushes subgoals and actions into a stack, and picks an action only when all its preconditions are satisfied. That

kazirhut.com

way, it works with state descriptions that are always consistent for growing plans, and goal descriptions for growing the search tree. The algorithm is however incomplete, in the sense that it could terminate without finding a plan. It is possible that the algorithm could commit to a wrong action at some time, reaching a state from which a plan cannot be found. For domains where the actions are reversible, for example the blocks world domain, the algorithm can be made complete by putting in an extra check, somewhat similar to the one in BSSP. However, the reasons for introducing the check are different, as described below.

```
BSSP(givenState, givenGoal, actions)
  goalPlan ←(givenGoal, ())    /* start with the empty plan */
  open ←(goalPlan)             /* note the extra set of brackets */
  while not Empty(open)
    do  n ← Head(open)         /*initially n = (GivenGoal, ()) */
      goal ← Goal(n)           /* extracts goal from search node n */
      plan ← Plan(n)           /* extracts plan from search node n */
      open ← Tail(open)        /* retain other search nodes */
      if not Satisfies(givenState, goal)
        then                  /* search further */
           R ← set of relevant actions for goal    /*may be empty */
           for each action a ∈ R
             do  p ← Cons(a, plan)
                 g ← Regress(a, goal)
                  if Consistent(g)    /* do not add if not consistent */
                      then
                        gp ← List(p, g)
                        open ← Cons(gp, open)    /* Depth First */
      else if ValidPlan(plan, givenState, givenGoal)
             then return plan
return "No plan found"
```

Figure 7.8 Backward State Space Search regresses over goals. It validates a plan before returning it. It also assumes a function *Consistent* to prune away inconsistent subgoals.

GSP breaks up a set of goal predicates into individual *subgoals* and attempts to solve them *individually one after another*. This approach is also sometimes called *linear planning*. This refers to the fact that the subgoals are attempted and solved in a linear order[6]. When we implement BSSP with depth first strategy, we are in fact doing something similar. As we will see, this does not always work. In some domains, a subgoal solved earlier may get disrupted by later actions. We saw this depicted in Figure 3.15 for the Rubik's cube. Most human solvers of the cube attempt it as a sequence of subgoals, but end up disrupting them on the way to the final solution. In domains where the goal state is reachable from all states, one can tackle this problem of *subgoal interaction* by putting in a check to verify that all subgoals have indeed been solved before termination. This check is necessary to take care of subgoal interaction, which is different from the check in BSSP, where it was to determine the validity of the plan constructed.

Goal Stack Planning works by pushing the goal description onto a stack. It pushes both the conjunct, as well as each of the individual goal predicates separately. The algorithm pops the element on the top of the stack. If it is a (goal) predicate that is true in the current state then nothing is done and the next element is popped from the stack. If it is a goal predicate that is not true in the current state, a relevant action is pushed onto the stack, followed by

kazirhut.com

the preconditions—first the conjunction, and then the individual preconditions. The precondition on the top of the stack becomes the next subgoal to be addressed recursively.

In practice, one may have to choose from a set of relevant actions that is pushed onto the stack. In the algorithm described below, this is shown as a nondeterministic choice by a *choose* operator. In practice, during implementation, this should be a choice point for a backtracking search algorithm which will try out different actions.

If all the preconditions of an action are true in a given state, they will get popped away, and the action will be on the top of the stack. In this case, the action is popped and added to the partial plan being grown in a forward manner. The current state progresses over the chosen action, and planning resumes by looking at the top of the stack. If the stack becomes empty, all the goal predicates have been achieved, and a plan is returned. If at any stage the planner cannot find an action to achieve a goal predicate, the algorithm reports failure.

We assume a function *PushSet*(*Set, Stack*) that takes a set of goal predicates and (a) pushes the conjunction of the predicates onto the Stack, and (b) also the individual goal predicates in some order. This order may be determined heuristically if possible. For example, in the blocks world domain, *holding*(*Block*) should be attempted last because it blocks up the robot arm. If it is solved or achieved first then it could be undone while achieving some other subgoal, since the plan is for a one-armed robot. The algorithm for *GSP* is shown in Figure 7.9. We assume, as is the case with many modern planners, that all instances of the planning operators that match the domain are available, is a set called *actions*.

```
GSP(givenState, givenGoal, actions)
state ← givenState
plan ← ()                          /* start with the empty plan */
stack ← emptyStack                 /* start with the empty stack */
PushSet(givenGoal, stack)
while not Empty(stack)
  do x ← Pop(stack)
  if x ∈ ← actions
    then plan ← (plan • x)         /* operator • extends plan with x */
         state ← Progress(x, state)
     else if x is a conjunct of goal predicates C
         then  solvedFlag ← TRUE
               for each g ∈ C
                 do if g ⊆ ← state
                      then solvedFlag ←FALSE
               if solvedFlag = FALSE
                 then PushSet(C, stack)
     else if x ∉ givenState        /* x is an unsatisfied goal predicate */
         then  CHOOSE action a that achieves x
               if no such action exists
                 then return FAILURE
               Push(a, stack)
               PushSet(Preconditions(a), stack)
return plan
```

Figure 7.9 Goal Stack Planning uses a stack to keep pending goals and actions. It picks actions only when it is applicable in the given state, and progresses forward.

kazirhut.com

kazirhut.com

Let us look at the trace of algorithm *GSP* on a small problem depicted in Figure 7.6. Let the initial state be:

> *S* = {onTable(*A*), onTable(*B*), onTable(*C*), clear(*A*), clear(*B*), clear(*C*), armempty}

Let the final state be a tower of *A*, *B* and *C* expressed by {on(*A*, *B*), on(*B*, *C*)}. Assume that we execute the algorithm *GSP* and keep a watch on the *push* and the *pop* actions, the plan being assembled, and the current state *S* as and when it changes. Remember, that *PushSet* is a function that pushes the set onto the stack, first the conjunct of predicates, and then each one individually. Let us assume that the trace shows us the object being pushed or popped, the plan whenever it is augmented, and the state after the most recent action is popped. The trace of the algorithm will then look like the following:

```
S = {onTable(A), onTable(B), onTable(C), clear(A), clear(B), clear(C),
armempty}
Plan = ( )
  push → on(A, B) ∧ on(B, C)
  push →on(A, B)      /* choice: tackle on(B, C) first and then on(A, B) */
  push →on(B, C)
  pop    ← on(B, C)
  push →stack(B, C) /* an action to achieve on(B, C) */
  push → clear(C) ∧ holding(B)    /* preconditions of stack(B, C) */
  push → holding(B)
  push → clear(C)
  pop    ← clear(C)            /* true in the given (start) state */
  pop    ← holding(B)
  push → pickup(B)             /* this is a choice point */
  push → onTable(B) ∧ armempty ∧ clear(B)
  push → armempty
  push → onTable(B)
  push → clear(B)
  pop    ← clear(B)
  pop    ← onTable(B)
  pop    ← armempty
  pop    ← onTable(B) ∧ armempty ∧ clear(B)
  pop    ← pickup(B)
S = {onTable(A), onTable(C), clear(A), clear(B), clear(C), holding(B)}
Plan = (pickup(B))
  pop    ← clear(C) ∧ holding(B)
  pop    ← stack(B, C)
S = {onTable(A), onTable(C), clear(A), clear(B), on(B, C), armempty}
Plan = (pickup(B), stack(B, C))
  pop    ← on(A, B))
  push → stack(A, B)
  push → clear(B) ∧ holding(A)
  push → holding(A)
  push → clear(B)
  pop    ← clear(B)
  pop    ← holding(A)
  push → pickup(A)          /* this is a choice point */
  push → onTable(A) ∧ armempty ∧ clear(A)
 push → armempty
```

kazirhut.com

```
  push → clear(A)
  push → onTable(A)
  pop    ← onTable(A)
  pop    ← clear(A)
  pop    ← armempty
  pop    ← onTable(A) ∧ armempty ∧ clear(A)
  pop    ← pickup(A)
S = {onTable(C), clear(A), clear(B), on(B, C), holding(A)}
Plan = (pickup(B), stack(B, C), pickup(A))
  pop    ← clear(B) ∧ holding(A)
  pop    ← stack(A, B)
S = {onTable(C), clear(A), on(B, C), on(A, B), armempty}
Plan = (pickup(B), stack(B, C), pickup(A), stack(A, B))
  pop    ← on(A, B) ∧ on(B, C)
Stackempty
Return Plan = (pickup(B), stack(B, C), pickup(A), stack(A, B))
```

The reader would have noticed that the algorithm has made some choices in the above execution. Choice points occur when there is more than one way of achieving a goal predicate. For example, the above trace has to find actions to achieve the goal predicate *holding*(*B*). This can be achieved either by picking up block *B* from the table, or by unstacking it from some other block. Which block to unstack it from, would also be a choice point. Likewise, if the robot is holding a block, and the goal predicate being addressed is *armempty*, it has to decide where to put that block. One approach would be to introduce backtracking here. Another would be to do some secondary reasoning to resolve this choice. For example, one could look at the state to see which action would be appropriate. In the example trace, this would tell us that since block *B* is on the table, one can assume that nothing will disturb it till the time comes for *holding*(*B*) to be made true, the correct action is *pickup*(*B*). The choice can also be resolved by looking at the goal set. If, for example, the robot is holding a block (say) *M*, and the goal has a predicate *on*(*M, N*) then a good way to achieve *armempty* would be to use the action *stack*(*M, N*).

A different kind of choice is concerned with the order of attempting subgoals. By pushing the subgoals onto the stack one by one, the algorithm is serializing them. The order in which you attempt subgoals is obviously going to affect performance. In the given trace, the algorithm chose to solve for *on*(*B, C*) first. This turned out to be a good choice, and after stacking *B* on *C*, the algorithm stacked *A* on *B*. What if the order had been reversed? The algorithm would have first stacked *A* on *B* to reach the state,

> *S* = {onTable(*B*), onTable(*C*), clear(*A*), clear(*C*), on(*A*, *B*), armempty}.

Then to achieve *on*(*B, C*) it would need to stack *B* on *C*, for which it would need *holding*(*B*), for which it would have to *pickup*(*B*), for which it would have to achieve *clear*(*B*), for which it would have to *unstack*(*A, B*) and put it somewhere. It would have thus *undone the subgoal on*(*A, B*) *achieved earlier*. We have addressed this problem by adding the full goal conjunct in

kazirhut.com

the stack. After achieving *on*(*B, C*) the algorithm will reach the state *S* = {onTable(*A*), onTable(*C*), clear(*A*), clear(*B*), on(*B*, *C*), armempty} and again look at the conjunct *on*(*A, B*)∧*on*(*B, C*). At this point *on*(*B, C*) is true but not *on*(*A, B*). Now it will pickup *A* and stack it on *B* to find the plan (pickup(*A*), stack(*A*, *B*), unstack(*A*, *B*) putdown(*A*), pickup(*B*), stack(*B*, *C*), pickup(*A*), stack(*A*, *B*)). It did solve the problem, but found a suboptimal plan.

Another place that ordering is imposed is in the order in which the preconditions are pushed onto the stack. For the action *pickup*(*?X*) the preconditions are *onTable*(*?X*), *clear*(*?X*) and *armempty*. It makes sense to push a*rmempty* first, and hence tackle it later. If *armempty* is addressed first then the second precondition will need to disrupt it because the robot arm has to be used anyway. If for example *clear*(*?X*) is done last then that would be achieved by unstacking something from *?X*, which would mean *holding*(*?X*) would be *true* and *armempty* would have got disrupted. Likewise, the preconditions of *stack*(*?X, ?Y*) are *holding*(*?X*) and *clear*(*?Y*). For a similar reason *holding*(*?X*) should be done later, and hence pushed first.

While in the above problem it was possible to find a subgoal ordering to solve them linearly, this may not always be the case. A seemingly similar problem (see Figure 7.3) known as the Sussman's Anomaly, demonstrates that even in simple problems in the blocks world, the subgoals may not be serializable (Sussman, 1975). That is, there is no ordering of subgoals where when each of the subgoals is solved individually, the goal is solved as a consequence. The progress of the *GSP* on Sussman's Anomaly is illustrated in Figure 7.10. We can observe that neither serialization of the two predicates will result in success. In both cases, the goal predicate achieved later undoes the one solved earlier. We have tackled this problem by adding an extra check for the entire conjunct. The algorithm will, therefore, realize that the goal has not been achieved and attempt the problem again, from the new state it is in. From here, it will find a solution in both cases. However, with this extra check we have created another problem. If the goal is inconsistent then the algorithm may never terminate. For example, if the goal were to be *on*(*A*, *B*) ∧ *on*(*B*, *C*) ∧ *on*(*C*, *A*), the algorithm will go into an infinite loop.

kazirhut.com

kazirhut.com

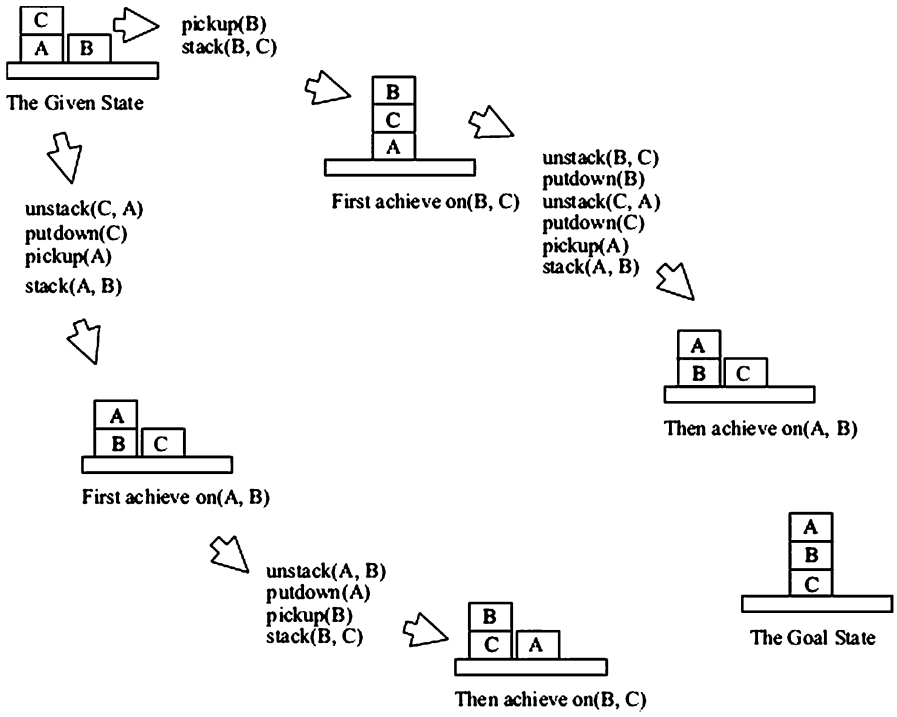

Figure 7.10 Neither order in Sussman's Anomaly solves the problem.

In the blocks world, the extra check works. This is because in this domain, any state is reachable from any other state. If this were not the case then the *GSP* algorithm would have run into problems. Consider, for example, planning in the culinary domain, where the actions are mostly irreversible. A planner, for example, may need to do some actions in a particular order. If it commits to a wrong set of actions then it would need to be able to go back to the choice point and choose another action. Thus for completeness, backtracking in the backward phase would be necessary.

We can think of the *GSP* algorithm as doing depth first search on an *AND*/*OR* goal tree. Pushing into the stack is like generating the tree. A goal predicate is *solved* if it occurs in the current state. An action node (*AND* node) is solved if all its children (preconditions of the action) are solved. An *OR* node represents a choice of actions for achieving a goal predicate. We will revisit this algorithm in the chapter on reasoning with logic, as backward chaining, the foundation of the programming language Prolog (see Chapter 12). The theorem proving algorithm searches for supporting statements in the backward direction, but commits to inferences in the forward direction. A recursive version (Figure 7.11) of the *GSP* algorithm adapted from (Ghallab et al., 2004) illustrates the dual nature of *GSP* search, and its relation to an *AND*/*OR* tree. It uses a function *PlanProgress*(*P, S*), shown in Figure 7.12, that progresses a given state *S* over a plan *P* by successively progressing over its actions.

kazirhut.com