6.2 Types of Planning Systems

The proper representation of planning problems is an important issue and needs to be discussed at length. Planning problems involve, representing states, actions, and goals. An ideal language would be one which is expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient algorithms to operate over it. The different components may be represented in the following manner:

Representation of States For representation of states, planners decompose the world into logical conditions and then represent a state as a conjunction of predicate atoms (positive literals).

Representation of Goals A goal is a partially specified state and is represented as a conjunction of predicate atoms (ground literals).

Representation of Actions An action is specified in terms of preconditions (that must hold true before the action can be executed) and effects (that ensue when the action has been executed).

A number of formulations have been used so far that attempt to solve the planning problems, such as operator-based planning, case-based planning, logic-based planning, constraint-based planning, distributed planning, etc.

6.2.1 Operator-Based Planning

Before we attempt to solve a problem with the help of planning, we need to have a start state, a set of actions, a goal state, and a database of logical sentences about the start state. Once these are specified, the *planner* will try to generate a plan, which when executed by the *executor* in a state S (start state), satisfying the description of the start state, will result in the state G (goal state), satisfying the description of the final state. Here, actions are represented as *operators*. This approach, also known as the STRIPS approach (explained later), utilizes various *operator schemas* and *plan representations*. The major design issues and concepts are given as follows and have been explained in the following sections appropriately.

- **Operator schema** Add–delete–precondition lists, procedural vs declarative representations, etc.
- **Plan representations** Linear plans, non-linear plans, hierarchical plans, partial-order plans, conditional plans, etc.
- **Planning algorithms** Planning as search, world-space vs plan-space, partial-order planning, total-order planning, progression, goal-regression, etc.
- **Plan generation** Plan reformulation, repair, total-ordering, etc.

6.2.2 Planning Algorithms

Search techniques for planning involve searching through a search space. We now introduce the concept of *planning as a search* strategy. In search technique method, there are basically two approaches: searching a *world* (or *state*) space or searching a *plan* space. The concepts of world (or state) space and plan space may be defined as given below.

World-Space In world space, the *search space* constitutes a set of states of the world, and *action* is defined as transitions between states, and a *plan* is described as a path through the state space. In state space, it is easy to determine which sub goals are achieved and which actions are applicable; however, it is hard to represent concurrent actions.

Plan-Space In plan space, the *search space* is a set of plans (including partial plans). The start state is a null plan and transitions are plan operators. The order of the search is not the same as plan execution order. A shortcoming of the plan space is that it is hard to determine what is true in a plan. Both the approaches are discussed in detail as follows:

Searching a World Space

Each node in the state search graph denotes a *state* of the world, while arcs in the graph correspond to the execution of a specific action. The planning problem is to find a path from a given start state to the desired goal state in the search graph. For developing planning algorithms, one of the following two approaches may be used:

i. **Progression** This approach refers to the process of finding the goal state by searching through the states generated by actions that can be performed in the given state, starting from the start state. It is also referred to as the *forward chaining approach*. Here at a given state, an action (may be non-deterministic) is chosen whose preconditions are satisfied. The process continues until the goal state is reached.

ii. **Regression** In this approach, the search proceeds in the backward direction, that is, it starts with the goal state and moves towards the start state. This is done by finding actions whose effects satisfy one or more of the posted goals. Posting the preconditions of the chosen action as goals is called goal *regression*. It is also known as *backward chaining approach*. Here, we choose an action (which may be non-deterministic) that has an effect that matches an unachieved sub goal. Unachieved preconditions are added to the set of sub goals and this process is continued till the set of unachieved sub goals becomes empty. In regression, the focus is on achieving goals and is thus often more efficient.

It is to be noted that algorithms based on both the approaches are sound and complete. An algorithm is said to be *sound* if the plan generated succeeds in completing the desired job, and it is said to be *complete* if it guarantees to find a plan, if one exists. However, in most situations regression is found to be a better strategy.

Searching a Plan Space

Each node in plan space graph represents *partial plans*, while arcs denote *plan refinement operations*. One can search for either a plan with a totally-ordered sequence of actions or a plan with a partially-ordered set of actions. A partial-order plan has the following three components:

i. ***Set of actions*** As an example of a set of actions, we can consider some of the activities from our daily routine, such as *go-for-morning-walk*, *wake-up*, *take-bath*, *go-to-work*, *go-to sleep*, and so on.

ii. ***Set of ordering constraints*** In the actions mentioned above, the action *wake-up* is performed before *go-for-morning-walk*; therefore, we can represent it as [wake-up ← go-for-morning-walk]. Some of the partial ordering constraints in the set of actions given above may be written as follows:

wake-up	←	go-for-morning-walk
wake-up	←	take-bath
wake-up	←	go-to-sleep
wake-up	←	go-to-work
go-for-morning-walk	←	go-to-work
go-for-morning-walk	←	go-to-sleep
take-bath	←	go-to-sleep
go-to-work	←	go-to-sleep

While most activities will have to follow a certain order, some may not necessarily obey the constraints. For example, [take-bath ← go-to-work] may not have this strict ordering as one may go to work without taking bath or one may take bath after returning from work. The ordering [go-for-morning-walk ← go-to-work] has to be in constraint as specified as *go-for-morning-walk* activity cannot follow *go-to-work* activity.

iii. ***Set of causal links*** We observe that *awake* is a link from the action *wake-up* to the action *go-for-morning-walk* with an awakening state of the person between them represented as '*wake-up—awake—go-for-morning-walk*'. This denotes a *casual link*. When the action *wake-up* is added to the generated plan, the above causal link is also recorded along with the ordering constraint [wake-up ← go-for-morning-walk]. This is because the effect of the action *wake-up* is that the individual is awake; this is a precondition of the action *go-for-morning-walk*. The second action cannot occur until this precondition is satisfied. Causal links help in detecting inconsistencies whenever a partial plan is refined.

6.2.3 Case-Based Planning

In case-based planning, for a given case (consisting of start and goal states) of a new problem, the library of cases in case base is searched to find a similar problem, with similar start and goal states. The retrieved solution is then modified or tailored according to the new problem. Thus,

case-based planning helps in utilizing *specific* knowledge obtained by previous experience. This approach is based on human methodology of tackling a problem. Humans also obtain solutions to their problems by learning from their experience (Althoff & Aamodt, 19960) and solve new problem by finding a similar case handled in the past. So, in case-based planning, a new problem is matched against the cases stored in the case base (past experience) and one or more similar cases are *retrieved*. A solution suggested by the matched cases is then *reused* and tested for success. Unless the retrieved case is a close match, the solution will have to be *revised*, which produces a new case that can then be *retained* as a part of learning. An initial description of a problem defines a *new case*. This planning procedure is described as a cyclical process consisting of the following steps:

Retrieve The most similar cases are retrieved from the case base using various methods.

Reuse The cases are reused in an attempt to solve a new problem;

Revise The proposed solution of the retrieved case is revised and modified, if necessary, and

Retain The new solution is retained as a part of learning if it is very different from the retrieved case.

Case-based reasoning (CBR) systems will be discussed in detail in Chapter 11.

6.2.4 State-Space Linear Planning

In the process of linear planning, only one goal is solved at a time. It requires a simple search strategy that uses a stack of unachieved goals. The advantages and disadvantages of state-space linear planning are discussed as follows:

Advantages of Linear Planning

- Since the goals are solved one at a time, the search space is considerably reduced.
- Linear planning is especially advantageous if goals are (mainly) independent.
- Linear planning is sound.

Disadvantages of Linear Planning

- It may produce sub-optimal solutions (based on the number of operators in the plan).
- Linear planning is incomplete.
- In linear planning, the planning efficiency depends on ordering of goals.

6.2.5 State-Space Non-Linear Planning

In non-linear planning, the sub goals may be solved in any order and they may be interdependent. In this process, the basic idea is to use a goal set instead of a goal stack. All possible sub-goal orderings are included in the search space and the goal interactions are handled by interleaving. The advantages and disadvantages of state-space non-linear planning are discussed as follows:

Advantages of Non-Linear Planning

- Non-linear planning is sound and complete.
- It may be optimal with respect to plan length (depending on the search strategy employed).

Disadvantages of Non-Linear Planning

- Since all possible goal orderings may have to be considered, a larger search space is required in non-linear planning.
- Non-linear planning requires a more complex algorithm and a lot of book-keeping.

In the following section, we will discuss planning methods using a specific example of block world problem and explain the procedures of generating plans to solve goals using linear and non-linear methods.

6.3 Block World Problem: Description

A block world problem basically consists of handling blocks and generating a new pattern from a given pattern (Rich and Knight, 2003). This problem closely resembles a game of block arrangement and construction usually played by children. Our aim is to explain strategies that may lead to a plan (or steps), which may help a robot to solve such problems. For this, let us consider the following assumptions:

- All blocks are of the same size (square in shape).
- Blocks can be stacked on each other.
- There is a flat surface (table) on which blocks can be placed.
- There is a robot arm that can manipulate the blocks. The arm can hold only one block at a time.

In the block world problem, a state is described by a set of predicates, which represent the facts that are true in that state. For every action, we describe the changes that the action makes to the state description. In addition, some statements regarding the things which remain unchanged by applying actions are also to be specified. For example, if a robot picks up a block, the colour of the block will not change. This is the simplest possible approach and is described below. Table 6.1 provides descriptions of the operators or actions used for this problem.

Actions (Operations) Performed by Robot

To understand the concept of block world problem, let us use the following convention:

- Capital letters X, Y, Z, …, are used to denote variables
- Lowercase letters a, b, c, …. are used to represent specific blocks

The description of various operators or actions used in this problem is given in Table 6.1.

Table 6.1 Description of Operators or Actions Used in Block World Problem

Operators	Short Form	Description
UNSTACK(X, Y)	US(X, Y)	Pick up block X from block Y (current position is that X is on Y). The arm must be empty and top of X must be clear.
STACK(X, Y)	ST(X, Y)	Put block X on the top of block Y. The arm must be holding block X. Top of Y should be clear.
PICKUP(X)	PU(X)	Pick up block X from the table and hold it. Initially the arm must be empty and top of X should be clear.
PUTDOWN(X)	PD(X)	Put down block X on the table. The arm must be holding block X before putting down.

In block world problem, certain predicates are used to represent the problem states for performing the operations given in Table 6.1. These predicates are described in Table 6.2.

Table 6.2 Predicates Used in Block World Problem

Predicates	Description
ON(X, Y)	Block X is on the block Y
ONTABLE(X)	Block X is on the table
CLEAR(X)	Top of X is clear
HOLDING(X)	Robot arm is holding X
ARMEMPTY	Robot arm is empty

In the following sections, we will discuss logic-based planning, linear planning, and non-linear planning.

6.4 Logic-Based Planning

We will use the block world problem explained in the preceding section to explain the concept of logic-based planning. In this approach, we have to explicitly state all possible logical statements that are true in the block world problem. Some of these logical statements are described as follows:

- If the robot arm is holding an object, then arm is not empty
  $(\exists X)\,\text{HOLDING}(X) \rightarrow \sim \text{ARMEMPTY}$
- If the robot arm is empty, then arm is not holding anything
  $\text{ARMEMPTY} \rightarrow \sim (\exists X)\,\text{HOLDING}(X)$
- If X is on a table, then X is not on the top of any block
  $(\forall X)\,\text{ONTABLE}(X) \rightarrow \sim (\exists Y)\,\text{ON}(X, Y)$

- If X is on the top of a block, then X is not on the table
  $(\forall X)(\exists Y)\, ON(X, Y) \rightarrow \sim ONTABLE(X)$
- If there is no block on top of block X, then top of block X is clear
  $(\forall X)(\sim(\exists Y)\, ON(X, Y)) \rightarrow CLEAR(X)$
- If the top of block X is clear, then there is no block on the top of X
  $(\forall X)\, CLEAR(X) \rightarrow \sim(\exists Y)\, ON(Y, X)$

Further, the axioms reflecting the effect of the operations mentioned in Table 6.2 have to be described on a given state. Let us assume that a function named GEN generates a new state from a given state as a result of the application of some operator/action. For example, if the action OP is applied on a state S and a new state S1 is generated then S1 is written as S1= GEN(OP, S).

- The effect of UNSTACK(X, Y) in state S is described by the following axiom.
  $[CLEAR(X, S) \wedge ON(X, Y, S) \wedge ARMEMPTY(S)] \rightarrow$
  $[HOLDING(X, S1) \wedge CLEAR(Y, S1)]$

Here, S1 is a new state obtained after performing the UNSTACK operation in state S. If we execute UNSTACK(X, Y) in S, then we can prove that HOLDING(X, S1) $\wedge$ CLEAR (Y, S1) holds true. Here state is introduced as an argument of an operator.

Since the interpretation of the axioms showing effect of the following operations is self explanatory, we will omit these now onwards.

- The effect of STACK(X, Y) in state S is described as follows.
  $[HOLDING(X, S) \wedge CLEAR(Y, S)] \rightarrow$
  $[ON(X, Y, S1) \wedge CLEAR(X, S1) \wedge ARMEMPTY(S1)]$
- The effect of PU(X) in state S is described by the following axiom.
  $[CLEAR(X, S) \wedge ONTABLE(X, S) \wedge ARMEMPTY(S)] \rightarrow$
  $[HOLDING(X, S1)]$
- The effect of PD(X) in state S is described by as follows.
  $[HOLDING(X, S)] \rightarrow$
  $[ONTABLE(X, S1) \wedge CLEAR(X, S1) \wedge ARMEMPTY(S1)]$

It should be noted here that after any operation is carried out on a given state, we cannot comment about other situations in the new state S1. For example, after the operation UNSTACK(X, Y), we cannot make a statement regarding the current position of Y, that is, whether Y is still on the table or on another block. Similarly, we cannot say that properties such as colour or weight of any block in the new state are same as the previous state. Therefore, there might be many such properties which do not change with change in state. So, we have to provide a set of rules known as *frame axioms* for the properties that do not get affected in the new state with the application of each

operator. Table 6.3 lists a few such situations which will not get affected by using the UNSTA operator. We can define frame axioms for other operators in a similar manner.

Table 6.3 Frame Axioms for Operator UNSTACK

Current State S0	implies	State S1 = GEN(UNSTACK (X, Y), S0)
If block Z or Y is on a table in state S0, then UNSTACK(X, Y) in state S0 will not affect block Z or Y in the state S1		
ONTABLE(Z, S0)	→	ONTABLE(Z, S1)
ONTABLE(Y, S0)	→	ONTABLE(Y, S1)
If block Z is on block W or Y is on any block U, then UNSTACK(X, Y) in state S0 will not affect blocks Z and Y in the state S1		
ON(Z, W, S0)	→	ON(Z, W, S1)
ON (Y, U, S0)	→	ON(Y, U, S1)
If colour of blocks X, Y, or any block Z is same (say, red) in state S0 then it remains the same in state S1 after UNSTACK(X, Y) operation is performed		
COLOR(X, C, S0)	→	COLOR(X, C, S1)
COLOR(Y, C, S0)	→	COLOR(Y, C, S1)
COLOR(Z, C, S0)	→	COLOR(Z, C, S1)
ON relation is not affected by UNSTACK operator if the blocks involved in ON relation are different from those involved in UNSTACK operation		
ON(Z, W, S0) Λ NE (Z, X)	→	ON(Z, W, S1)

The advantage of this approach is that only a simple mechanism of resolution needs to be performed for all the operations that are required on the state descriptions. On the other hand, a disadvantage of this approach is that in case of complex problems, the number of axioms becomes very large as we have to enumerate all those properties which are not affected, separately for each operation. Further, if a new attribute is introduced into the problem, it becomes necessary to add a new axiom for each operator.

For handling the complex problem domain, we need a mechanism that does not require a large number of frame axioms to be specified explicitly. Such mechanism was used in early robot problem-solving system known as STRIPS (STanford Research Institute Problem Solver), which was developed by Fikes and Nilsson in 1971. Each operator in this approach is described by a list of new predicates that become true and a list of old predicates that become false after the operator is applied. These are called ADD and DEL (delete) lists, respectively. There is another list called PRE (preconditions), which is specified for each operator; these preconditions must be true before an operator is applied. Any predicate which is not included on either ADD or DEL list of an operator is assumed to remain unaffected by it. Frame axioms are specified implicitly in STRIPS; this greatly reduces the amount of information stored.

STRIPS-Style Operators

We observe that by making the frame axioms implicit, we have greatly reduced the amount of information that needs to be provided for each operator. Now, we only need to specify the required effect; the unaffected attributes are not included. Therefore, in this representation, if a new attribute is added, the operator lists do not get changed. Henceforth, we will use short forms of predicates as given in Table 6.4 for the sake of convenience.

Table 6.4 Short Forms of Predicates

Predicates	ON(X, Y)	ONTABLE(X)	CLEAR(X)	HOLDING(X)	ARMEMPTY
Short Form	O(X, Y)	T(X, Y)	C(X)	H(X)	AE

The three lists (PRE, DEL, and ADD) required for each operator are given in Table 6.5.

Table 6.5 Operator with PRE, DEL and ADD Lists

Operator	**PRE list**	**DEL list**	**ADD list**
ST(X, Y)	C(Y) ∧ H(X)	C(Y) ∧ H(X)	AE ∧ O(X, Y)
US(X, Y)	O(X,Y) ∧ C(X) ∧ AE	O(X, Y) ∧ AE	H(X) ∧ C(Y)
PU(X)	T(X) ∧ C(X) ∧ AE	T(X) ∧ AE	H(X)
PD(X)	H(X)	H(X)	T(X) ∧ AE

The state description is updated after each operation by deleting those predicates from the state that are present in the DEL list and adding those predicates to the state that are present on the ADD list of the operator applied. If an incorrect sequence is explored accidentally, then it is possible to return to the previous state so that a different path may be tried.

6.5 Linear Planning Using a Goal Stack

The goal stack method is one of the earliest methods that were used for solving compound goals, which may not interact with each other. This approach was used by STRIPS systems. In this system, the problem solver makes use of a single stack containing goals as well as operators that have been proposed to satisfy these goals. In goal stack method, individual sub goals are solved linearly and then, at the final stage, the conjoined sub goals are solved. Plans generated by this method contain complete sequence of operations for solving the first goal followed by complete sequence of operations for the next one, and so on. The problem solver uses database that describes the current state and the set of operators with PRE, ADD, and DEL lists.

6.5.1 Simple Planning using a Goal Stack

In this section, we will discuss the method of simple planning using a goal stack. Let us explain the method by using hypothetical goals. Consider the following goal that consists of sub goals G1, G2, ..., Gn.

$$GOAL = G1 \wedge G2 \wedge \ldots \wedge Gn$$

The sub goals G1, …, Gn are stacked (in any order) with a compound goal G1 Λ …Λ Gn at bottom of the stack. The status of stack is shown in Fig. 6.1.

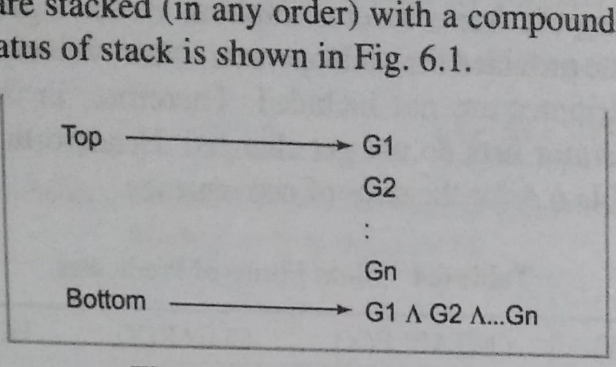

Figure 6.1 Status of Stack

The algorithm that is required to solve the goal is given below.

Algorithm 6.1 *GOAL_Stack*

```
{
• Push conjoined sub goals and individual sub goals onto Stack;
• flag = true;
While (Stack ≠ ϕ and flag = true) do
{ If top element of stack is an operator then
   {
   • POP it and add to PLAN_QUEUE of operations to be performed in a
     plan;
   • Generate new state from current state by using ADD and DEL
     lists of an operator
   }
   Else
   { If top of the stack is sub goal and is true in the current
     state then POP it
     Else
     {
        • Identify operator(s) that satisfies top sub goal of the stack
        • If (no operator exists) then set flag=false
          Else
          {
             • Choose one operator that satisfies the sub goal (use
               some heuristic);
             • POP sub goal and PUSH chosen operator along with its
               preconditions in the stack;
          }
     }
   }
• If (flag = false) then problem solver returns no plan else return
  the plan stored in PLAN_QUEUE for the problem;
}
}
```

6.5.2 Solving Block World problem using Goal Stack method

To illustrate the working of Algorithm 6.1, consider the following example, where the start and goal states of block world problem are shown in Fig. 6.2. Here a, b, c, and d are specific blocks.

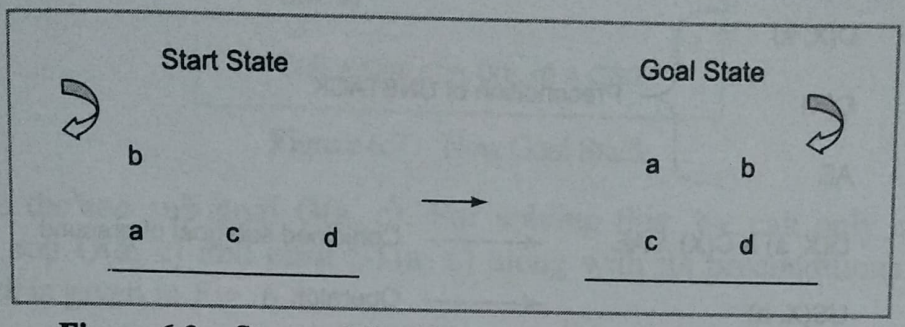

Figure 6.2 Start and Goal States of a Block World Problem

The logical representations of start and goal states may be written as

Start State: O(b, a) Λ T(a) ΛT(c) Λ T(d) Λ C(b) Λ C(c) Λ C(d) Λ AE
Goal State: O(a, c) Λ O(b, d) Λ T(c) Λ T(d) Λ C(a) Λ C(b) Λ AE
PLAN_QUEUE = ϕ

We notice that (T(c) Λ T(d) Λ C(b) Λ AE) is true in both start and goal, states. Hence, for the sake of convenience, we can represent it by CSG (conjoined sub goals present in both the states). We will work to solve sub goals O(a, c), O(b, d), and C(a) and while solving these sub goals, we will make sure that CSG remains true. We will first put these sub goals in some order in the stack. Let the initial status of the stack be as shown in Fig. 6.3.

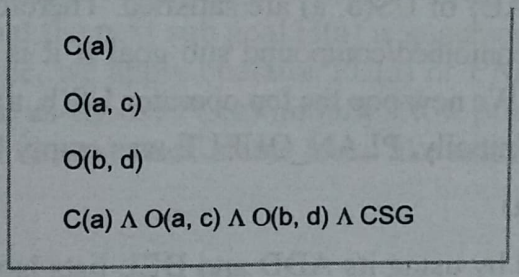

Figure 6.3 Initial Status of Goal Stack

Now, we need to identify an operator that can solve C(a). We notice that the operator US(X, a) can only be applied, where X gets bound to the actual block on top of 'a'. Here pop C(a) and-push US(X, a) in the goal stack. The status of the goal stack changes as shown in Fig. 6.4.

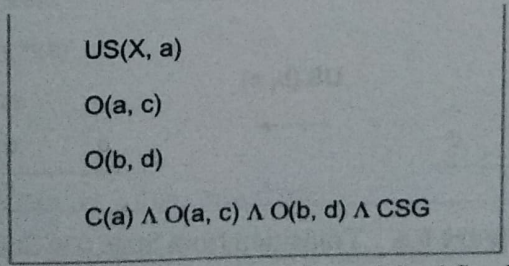

Figure 6.4 Changed Status of Goal Stack

Since stack operator US(X, a) can be applied only if its preconditions are true, therefore, we add its preconditions on top of the stack. The changed status of stack now looks as shown in Fig. 6.5.

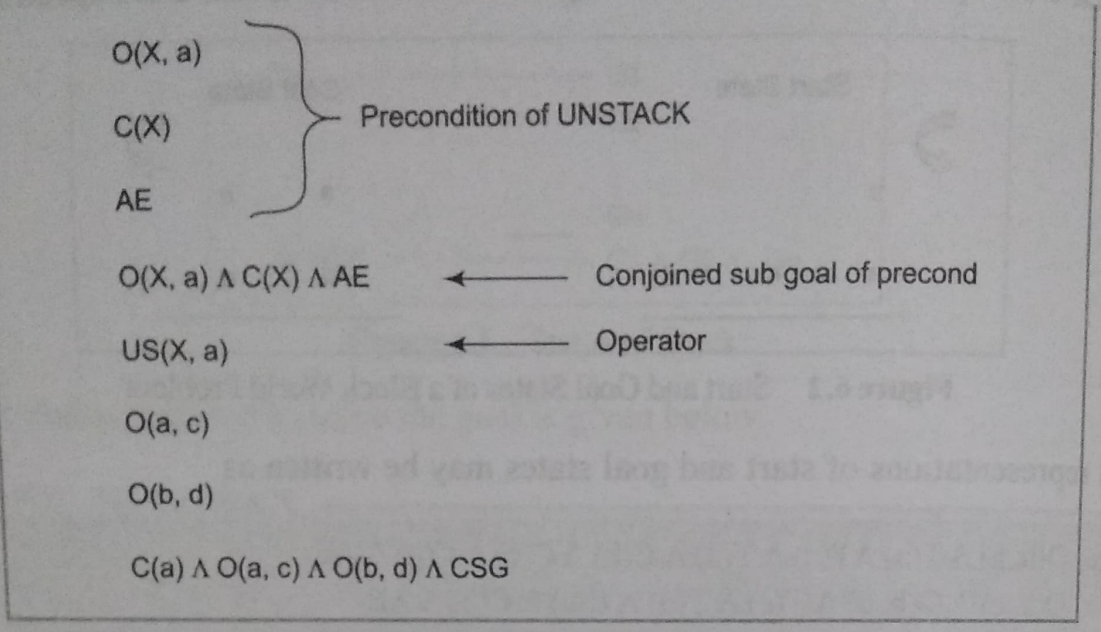

Figure 6.5 Status of Stack on Addition of Preconditions

The start state of the problem may be written as State 0

State 0 (Start state) $O(b, a) \wedge T(a) \wedge T(c) \wedge T(d) \wedge C(b) \wedge C(c) \wedge C(d) \wedge AE$

From State 0, we find that 'b' is on top of 'a', so the variable X is unified with block 'b'. Now, the preconditions {O(b, a), C(b), AE} of US(b, a) are satisfied. Therefore, the next step is to pop the preconditions along with its conjoined/compound sub goal if it is still true. In this case, we find compound sub goal to be true. We now pop the top operator US(b, a) and add it in a PLAN_QUEUE of the sequence of operators. Initially, PLAN_QUEUE was empty but now it contains US(b, a).

PLAN_QUEUE = US(b, a)

A new state State 1 produced by using its ADD and DEL lists is written as

State 1 $T(a) \wedge T(c) \wedge T(d) \wedge H(b) \wedge C(a) \wedge C(c) \wedge C(d)$

The transition from State 0 to State 1 is shown in Fig. 6.6.

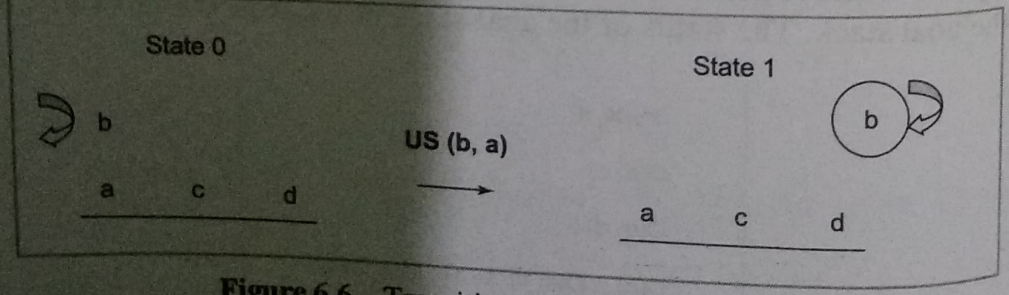

Figure 6.6 Transition from State 0 to State 1

The new goal stack is shown in Fig. 6.7.

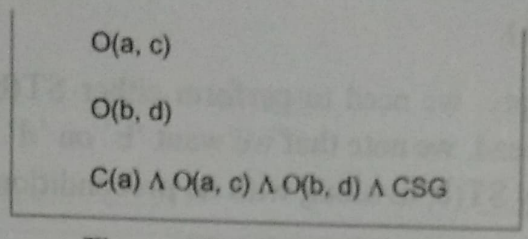

Figure 6.7 New Goal Stack

Now, let us solve the top sub goal O(a, c). For solving this, we can only apply the operator ST(a, c). So, we pop O(a, c) and push ST(a, c) along with its preconditions in the stack. The changed goal stack is given in Fig. 6.8.

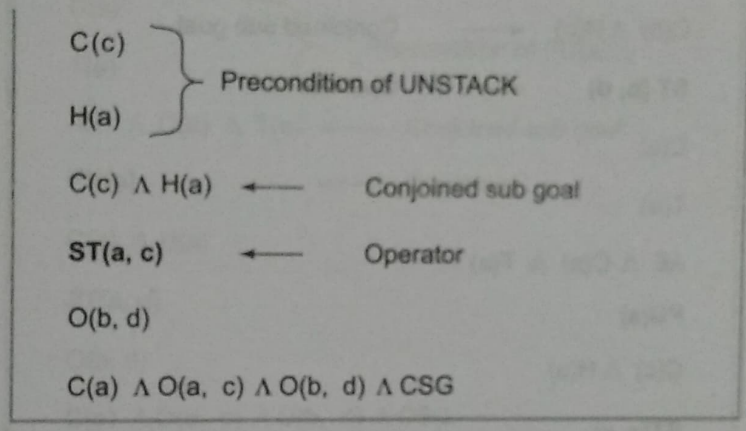

Figure 6.8 Changed Goal Stack

From *State 1*: {T(a) ∧T(c) ∧ T(d) ∧ H(b) ∧ C(a) ∧ C(c) ∧ C(d)}, we notice that C(c) is true, so we pop it. Then, we observe that the next sub goal H(a) is unachieved (not true), so we will solve this. For making H(a) to be true, we apply operator PU(a) or UN(a, X). In fact, any of the two operators can be applied but let us choose PU(a) initially. Now pop H(a) and push PU(a) with its preconditions to the stack. The current stack status looks like that shown in Fig. 6.9.

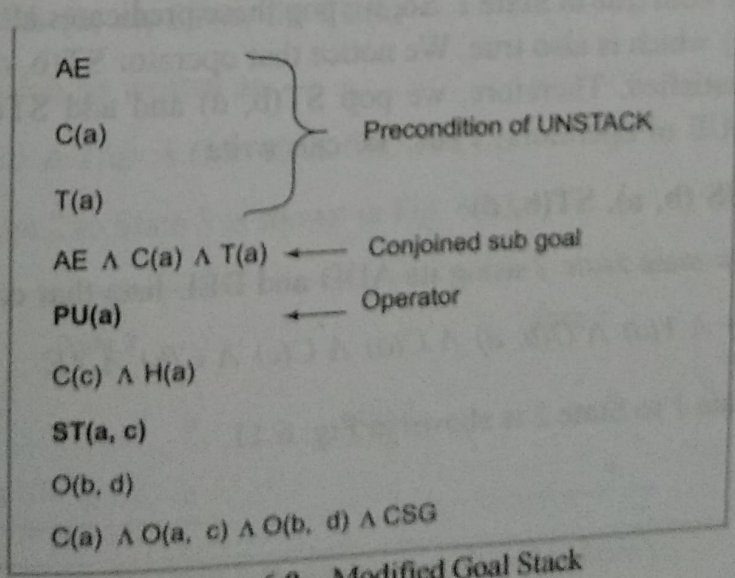

Figure 6.9 Modified Goal Stack

Now top sub goal AE above stack is to be solved. We notice that AE is not true as the a holding 'b' (as given in State 1).

In order to make the arm empty, we need to perform either ST(b, X) or PD(b). Let us ch ST(b, X). If we look a little ahead, we note that we want 'b' on 'd'. Therefore, we need to pu with 'd'. So, we replace AE by ST(b, d) along with its preconditions. The goal stack now ch to that shown in Fig. 6.10.

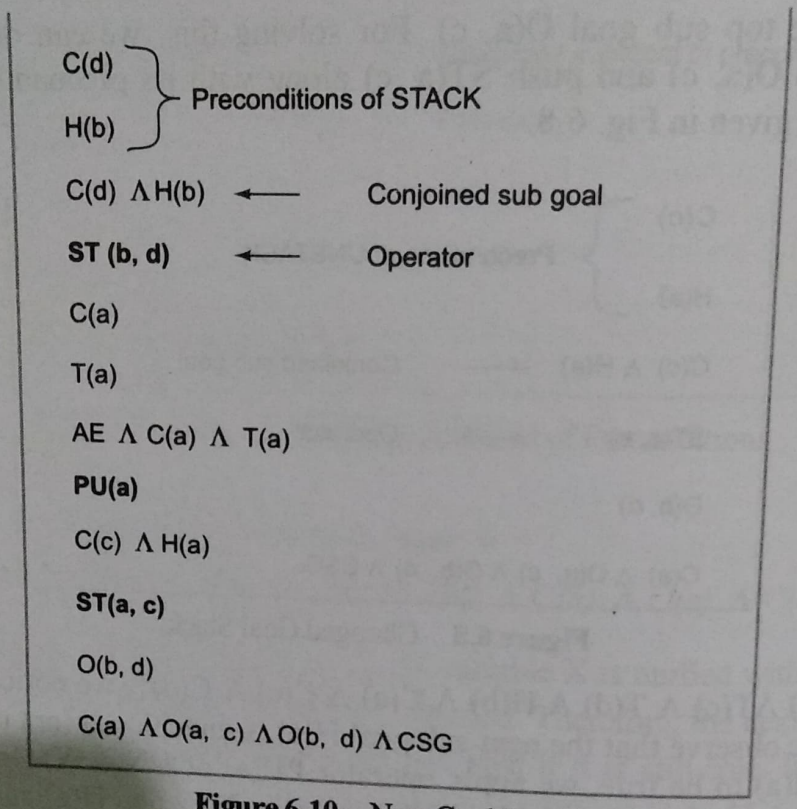

Figure 6.10 New Goal Stack

Now, C(d) and H(b) are both true in State 1. So, we pop these predicates along with the comp sub goal {C(d) Λ H(b)} which is also true. We notice that operator ST(b, d) can be applied its preconditions are satisfied. Therefore, we pop ST(b, d) and add ST(b, d) in the que sequence PLAN_QUEUE of operators. Thus, we can write

PLAN_QUEUE = US (b, a), ST(b, d)

Now we produce a new state *State 2* using its ADD and DEL lists that can be written as

State 2 *T(a) Λ T(c) Λ T(d) Λ O(b, d) Λ C(a) Λ C(c) Λ C(b) Λ AE*

The transition from State 1 to State 2 is shown in Fig. 6.11.

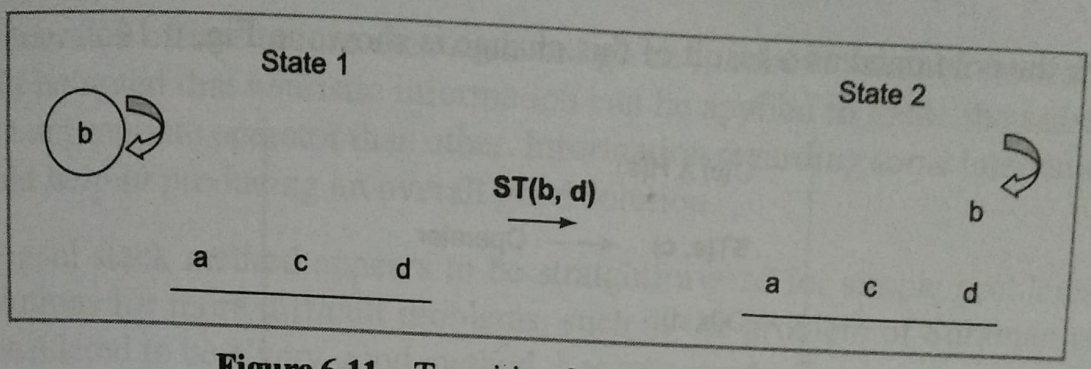

Figure 6.11 Transition from State 1 to State 2

The goal stack obtained as a result of this transition is shown in Fig. 6.12.

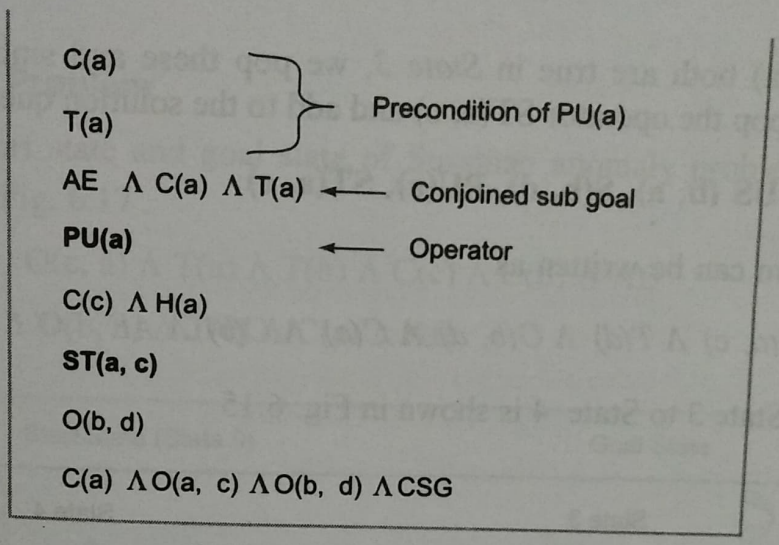

Figure 6.12 New Goal Stack

Now, AE, C(a) and T(a) are true (from State 2); hence, preconditions of PU(a) are satisfied. As a result, the operation PU(a) can be performed. Therefore, we pop it and add it in the queue of the sequence of operators and generate new state *State 3*. We can now write

PLAN_QUEUE = US (b, a), ST(b, d), PU(a)

State 3 of the problem can be written as

State 3: *T(c) Λ H(a) Λ T(d) Λ O(b, d) Λ C(c) Λ C(b)*

The transition from State 2 to State 3 is shown in Fig. 6.13.

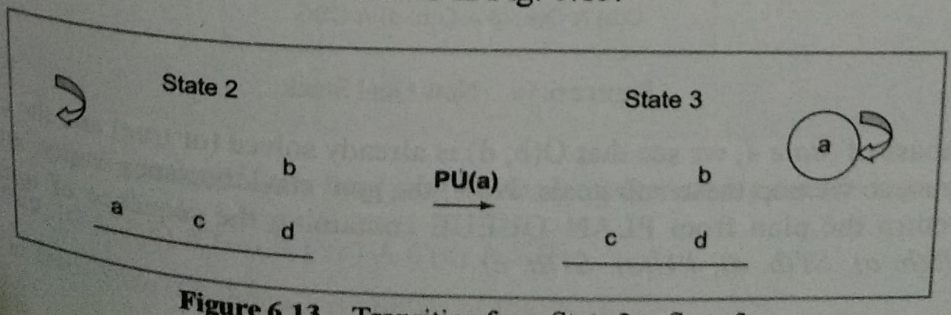

Figure 6.13 Transition from State 2 to State 3

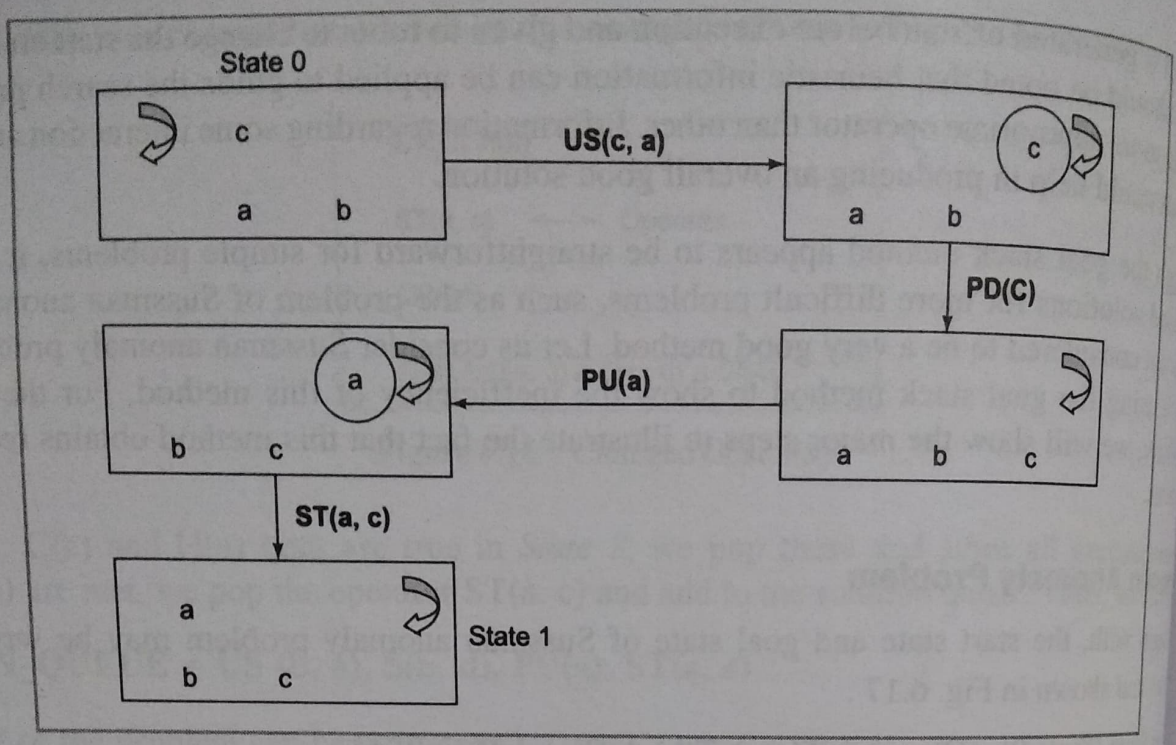

Figure 6.18 Transitions from State 0 to State 1

low, our aim is to satisfy the sub goal O(b, c). The sequence of operators **US(a, b), PD(a), P**
nd **ST(b, c)** is applied and *State 2* is generated. This state is represented as

State 2 O(b, c) Λ T(c) Λ T(a) Λ C(b) Λ C(a) Λ AE

The transition from State 1 to State 2 due to the application of the sequence of ope
mentioned above is given in Fig. 6.19.

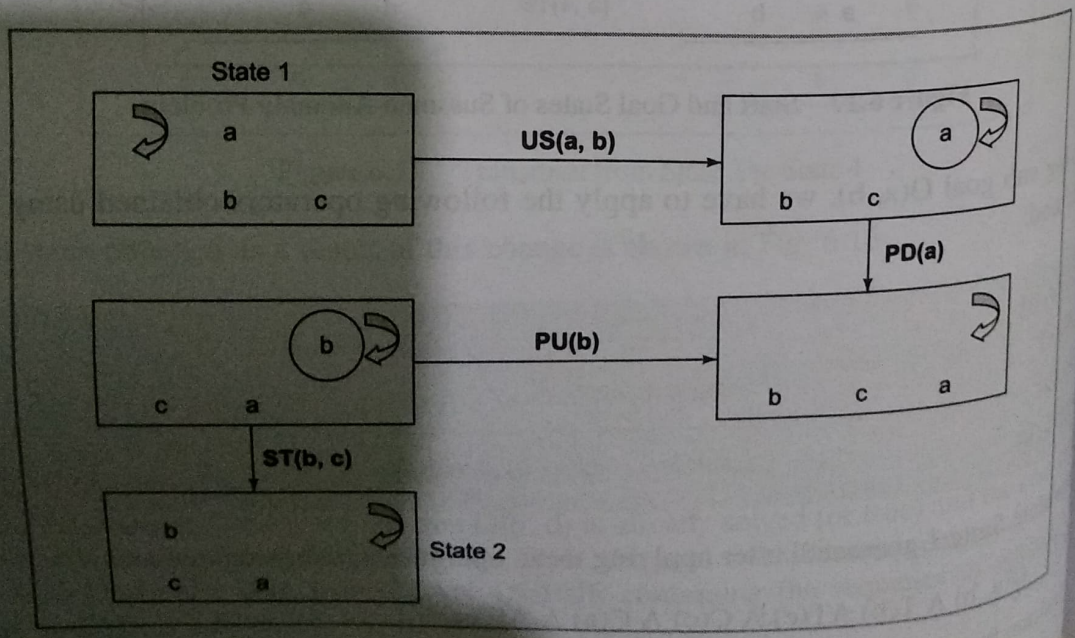

Figure 6.19 Transitions from State 1 to State 2

Finally, we need to
that while satisfying
O(a, b) again, we a
and it is found to be
plan. The transition

The complete plan thus ge
the following operations w

i. US(c, a)
ii. PD (c)
iii. PU(a)
iv. ST(a, b)
v. US(a, b)

Although this plan eventuall
because of the presence of a r
same blocks, one performed in
plan simply by repairing it. Rep
and undone immediately, such
plan, we notice that stacking a
complimentary sub goals, we ob

Finally, we need to satisfy the conjoined goal O(a, b) Λ O(b, c) Λ T(c) Λ C(a) Λ AE. We notice that while satisfying O(b, c), we have undone the already solved sub goal O(a, b). In order to solve O(a, b) again, we apply the operations **PU(a)** and **ST(a, b)**. The conjoined goal is checked again and it is found to be satisfied now. We obtain the goal state, and therefore, can now collect the total plan. The transition from State 2 to the goal state is shown in Fig. 6.20.

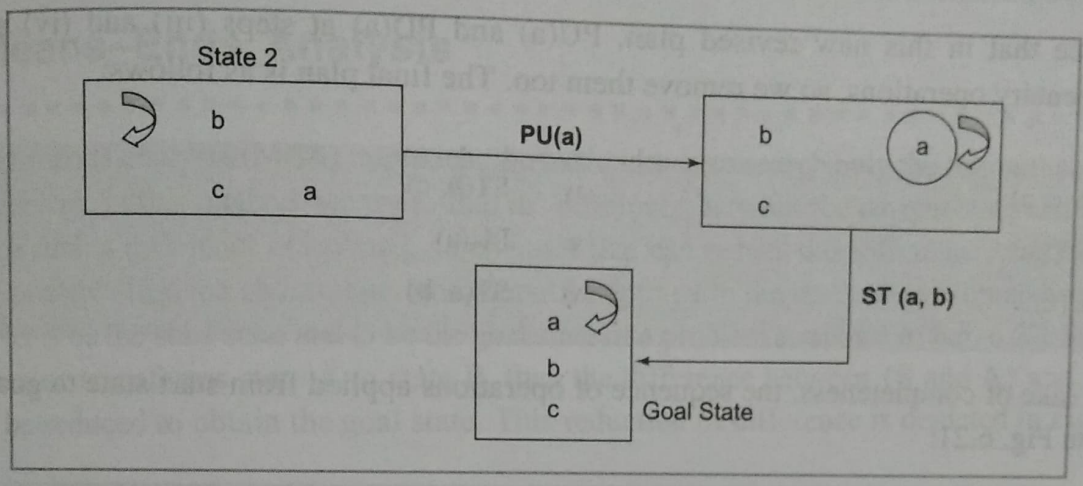

Figure 6.20 The Goal State from State 2

The complete plan thus generated is the sequence of all the sub plans generated above. Therefore, the following operations will be present in the solution sequence:

i. US(c, a)	vi. PD(a)
ii. PD (c)	vii. PU(b)
iii. PU(a)	viii. ST(b, c)
iv. ST(a, b)	ix. PU(a)
v. US(a, b)	x ST(a, b)

Although this plan eventually achieves the desired goal, it is not considered to be efficient because of the presence of a number of redundant steps, such as stacking and unstacking of the same blocks, one performed immediately after the other. We can get an efficient plan from this plan simply by repairing it. Repairing is done by looking at those steps where operations are done and undone immediately, such as ST(X, Y) and US(X, Y) or PU(X) and PD(X). In the above plan, we notice that stacking and unstacking are done at steps (iv) and (v). By removing these complimentary sub goals, we obtain the new plan as follows:

i. US(c, a)
ii. PD (c)
iii. PU(a)
iv. PD(a)
v. PU(b)
vi. ST(b, c)
vii. PU(a)
viii. ST(a, b)

We notice that in this new revised plan, PU(a) and PD(a) at steps (iii) and (iv) are complimentary operations, so we remove them too. The final plan is as follows:

i. US(c, a)
ii. PD (c)
iii. PU(b)
iv. ST(b, c)
v. PU(a)
vi. ST(a, b)

For the sake of completeness, the sequence of operations applied from start state to goal state shown in Fig. 6.21.

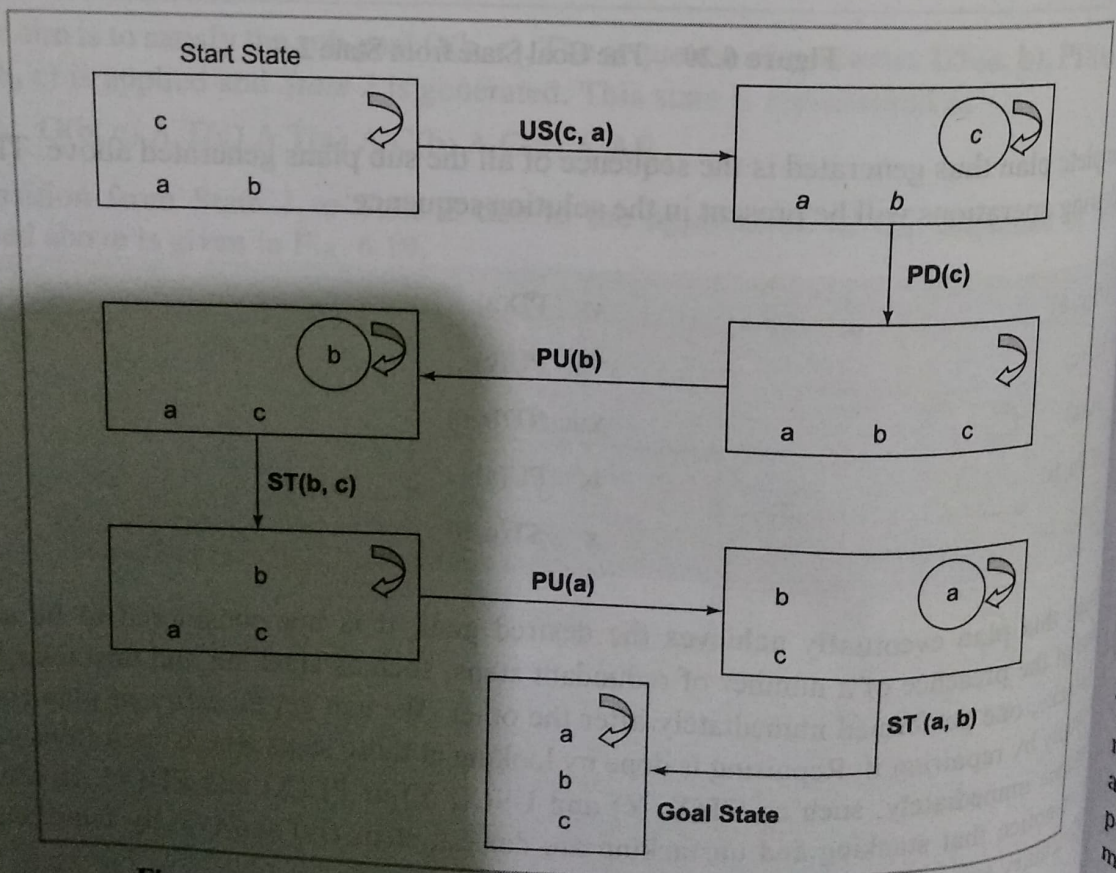

Figure 6.21 Complete Sequence of Operations for Sussman Anomaly

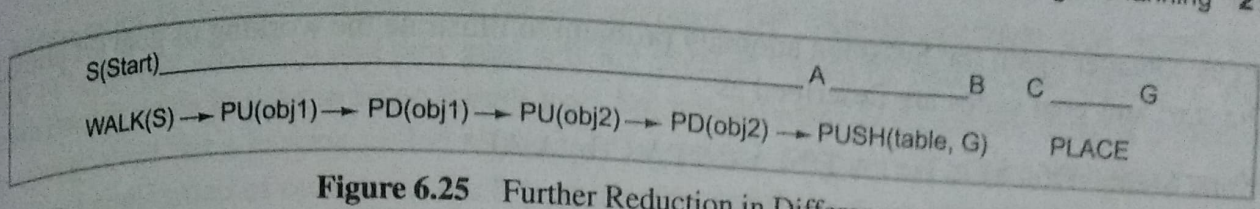

Figure 6.25 Further Reduction in Differences

The final difference between B and C can be reduced by using WALK to get the robot back to the objects, followed by PU and CARRY operations. Hence, the final plan is written as follows:

i. WALK(S_loc)
ii. PU(obj1)
iii. PD(obj1)
iv. PU(obj2)
v. PD(obj2)
vi. PUSH(table, G_loc)
vii. WALK(obj1_loc)
viii. PU(obj1)
ix. CARRY(obj1, G_loc)
x. PLACE(obj1, table)
xi. WALK(obj2_loc)
xii. PU(obj2)
xiii. CARRY(obj2, G_loc)
xiv. PLACE(obj2, table)

The order in which the differences are considered is critical. It is important that a significant difference is reduced before trying to reduce a less significant one otherwise we may end up wasting a great deal of effort.

This method is still not adequate for solving complex problems as working on one difference may interfere with the plan of reducing another. Therefore, we need to have methods which can handle complex problems by non-linear planning strategies. These strategies are discussed in detail in the following section.

6.7 Non-linear Planning Strategies

As we have already seen, complex problems such as Sussman anomaly problem cannot be efficiently handled by linear method described in subsection 6.5.2. To solve such problems directly, we have to consider non-linear way of generating plans. In this section, we will discuss two methods for solving complex problems: goal set method and constraint posting method.

6.7.1 Goal Set Method

In this method, a plan is generated by doing some work on one sub goal, then on another sub goal, and finally some more on the first sub goal. Such plans are called *non-linear plans* as they do not contain a linear sequence of complete sub plans (Rich & Knight, 2003). The abbreviations for the predicates remain the same as those used in linear planning with the goal stack method given in section 6.5.

Let us consider again the Sussman anomaly problem to illustrate the working of goal set m
The start and goal states are described as

Start State $O(c, a) \wedge T(a) \wedge T(b) \wedge C(c) \wedge C(b) \wedge AE$
Goal State $O(a, b) \wedge O(b, c) \wedge T(c) \wedge C(a) \wedge AE$

Figure 6.26 shows the start state and goal state of the problem.

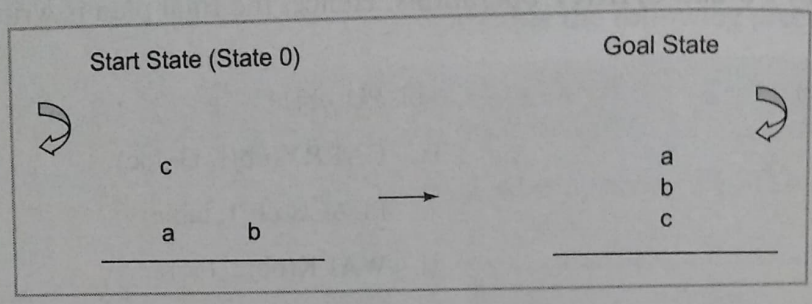

Figure 6.26 Start State and Goal State of Sussman Anomaly Problem

A good plan for solution of the above problem may ideally be written as follows:

- Start some work on sub goal O(a, b) by clearing 'a' using unstack operator that is, unstack 'c' f 'a' and then put 'c' on the table.
- Achieve O(b, c) by stacking 'b' on 'c'.
- Complete the goal O(a, b) by stacking 'a' on 'b'.

One way to find such a plan is to consider the collection of desired goals as a set. Here, a bac search procedure is used from the goal state to the start state, with no operators being a applied along the way. The idea is to look first for that operator which will be applied las final solution. It assumes that all sub goals except one have already been satisfied or solve need to find all the operators that might satisfy this final sub goal. Each of these operator have certain preconditions that must be met so that a new combined goal set is generated includes the preconditions as well as the original sub goals. Many of the paths can qui eliminated because of contradiction in the goal set. For example, if we have an arm empty as an arm holding both in the goal set, then evidently there is a contradiction in the set. Th is then pruned and not explored further.

Let us proceed to solve the Sussman anomaly problem. Consider that the last sub goa proved is either O(a, b) or O(b, c). Let us remove CLEAR and AE predicates from the go for the sake of simplicity. If we assume that O(b, c) is the last sub goal and it is solve application of some operator 'OP', then all its preconditions and sub goal O(a, b) must be

to be true prior to the application of the chosen operator 'OP'. All the operators that satisfy this final sub goal are identified and new goal sets are created. When an operator is applied, it may cause the sub goals in a set to be no longer true. So, non-selected goals are not directly copied to the new goal set but a process called *regression* is applied. Regression can be thought of as the backward application of operators. Each goal is regressed through an operator where, we try to determine which of them must be true before the operator is applied. For example, some of the predicates regressed under various operators are as follows:

- Reg(O(X, Y), ST(Z, X)) = O(X, Y)
- Reg(O(X, Y), ST(X, Y)) = *true*
- Reg(O(X, Y), PU(Z)) = O(X, Y)
- Reg(O(X, Y), PD(Z)) = O(X, Y)
- Reg(O(X, Y), US(X, Y)) = *false*
- Reg(O(X, Y), US(Z, W)) = O(X, Y)
- Reg(T(X), PU(X)) = *false*
- Reg(T(X), PD(X)) = *true*
- Reg(T(X), PU(Y)) = T(X)
- Reg(T(X), PD(Y)) = T(X)
- Reg(AE, PD(X)) = *true*
- Reg(AE, PU(X)) = *false*
- Reg(AE, ST(X, Y)) = *true*
- Reg(AE, US(X, Y)) = *false*

Interpretation of *Reg(O (X, Y), ST(Z, X) = O(X, Y)* is that the predicate O(*X*, *Y*) does not change when operator ST(Z, *X*) is applied on it. If O(*X*, Y) is regressed under US(*X*, Y) then O(*X*,Y) is no longer true and hence is false: Reg (O(*X*, *Y*), US(*X*, *Y*)) = false.

The backward search tree generated during the process of solving this problem is shown in Fig. 6.27. Note that many paths are pruned whenever a contradiction occurs in a goal set such as {H(a), AE} or if it contains a *false* value. For the sake of simplicity, pruned nodes are not shown. Contradiction might occur somewhere on these paths. The nodes are numbered in the order in which they are generated and are connected by arrow lines. The final plan generated for Sussman anomaly problem is written as

US(c, a) → PD(c) → PU(b) → ST(b, c) → PU(a) → ST(a, b)

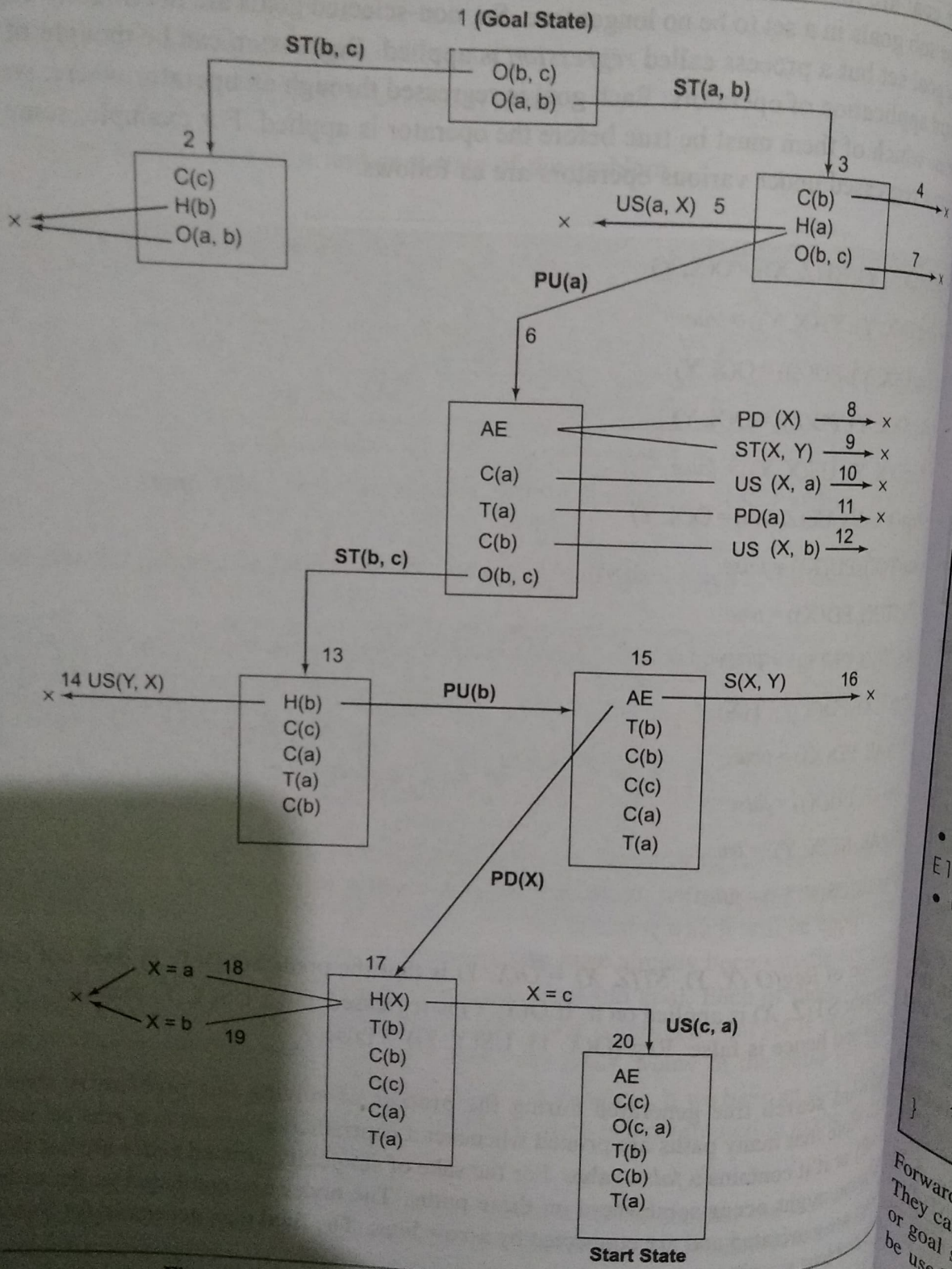

Figure 6.27 Backward Search Tree for Sussman Anomaly Problem

The algorithm that is used for this method is given below.

Algorithm 6.3 *Goal_set (Start_state, Goal_state)*

```
{
• STATE = START_state;
• GOALSET = GOAL_state;
• PLAN = [ ];  OPSTACK = [ ]; found = false;
• Until (All goals in GOALSET are satisfied OR found)
  {
   • Choose a goal G from the GOALSET;
   • For (each operator OP that satisfies G)
     {
      • Generate NEW_SET by adding preconditions of OP and copying all
        sub goals of GOALSET except G  after regressing these through
        operator OP;
      • If there is a contradiction in the NEW_SET then prune this
        path
        Else
        {
         • Push OP on the OPSTACK;
         • Set GOALSET = NEW_SET;
         • If (GOALSET = START_state) then found = true;
        };
     };
  };
• If (not found) then write("Not able to solve it")
Else {
• Until (OPSTACK ≠ ϕ)
     {
      • Pop operator OP from top of OPSTACK;
      • STATE = apply(OP, STATE);
      • PLAN = [PLAN; OP];
     };
  }
}
```

Forward and backward state-space searches are particular forms of *totally ordered* plan searches. They can explore strictly linear sequences of actions that are directly connected to the start state or goal state and cannot take advantage of problem decomposition. Other approaches that can be used to solve complex problems efficiently include those that work on several sub goals

independently, solve them with several sub plans, and then combine the sub plans. S... approaches have an advantage of flexibility of constructing the plan in any order. Th... approaches give planner the freedom to work on important decisions first, instead of working... steps in strict chronological order.

Before proceeding to discuss *constraint posting method*, the knowledge of *partial-order plan*... is required. This is being discussed as given below.

Partial-Order Planning

Any planner that places two actions into a plan without specifying which comes first is call... *partial-order planner*. In partial-order planning, actions dependent on each other are ordere... relation to themselves but not necessarily in relation to other independent actions. The solutio... represented as a *graph* of actions instead of a sequence of actions. The following two ma... operators may be defined here for the sake of simplicity. Macro-operator is one that ca... defined by a sequence of simple operators.

Macro Operator	Description	Sequence of Operators
MOT(X)	Move X onto table	US(X, _), PD(X)
MOVE(X, Y)	Move X onto Y	PU(X), ST(X, Y)

Consider the block world example given in Fig. 6.28 to illustrate the concept of partial-... planning method.

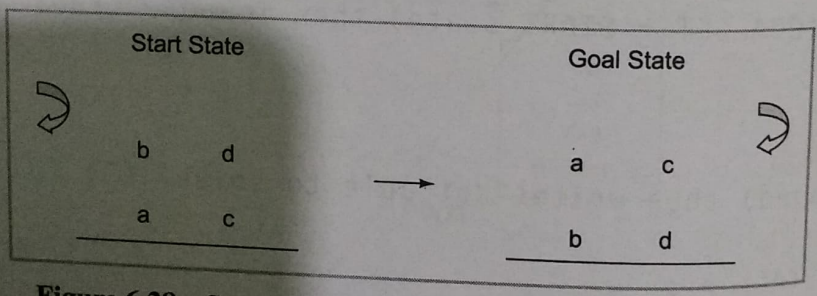

Figure 6.28 Start and Goal States of a Block World Problem

In the problem shown in Fig. 6.28, we notice that in goal state 'a' on top of 'b'; and to obt... configuration we have to first move 'b' onto the table from the start state, that is, MOT... then MOVE(a, b). Hence, we can apply partial ordering as MOT(b) ← MOVE(a, b) tha... true, that is, MOT(b) should come before MOVE(a, b) in the final plan. Similarly, to move... top of 'd', we first have to move 'd' onto the table and then move 'c' on top of 'd'. In th... partial ordering MOT(d) ← MOVE(c, d) is established.

A partial-order plan can also be represented in the form of a graph as shown in Fig... The directed edge from node MOT(d) to node MOVE(c, d) means that partial order MO... MOVE(c, d) holds true. It should be noted that the dummy actions START and FINISH ... beginning and end of the plan in the graph, respectively.

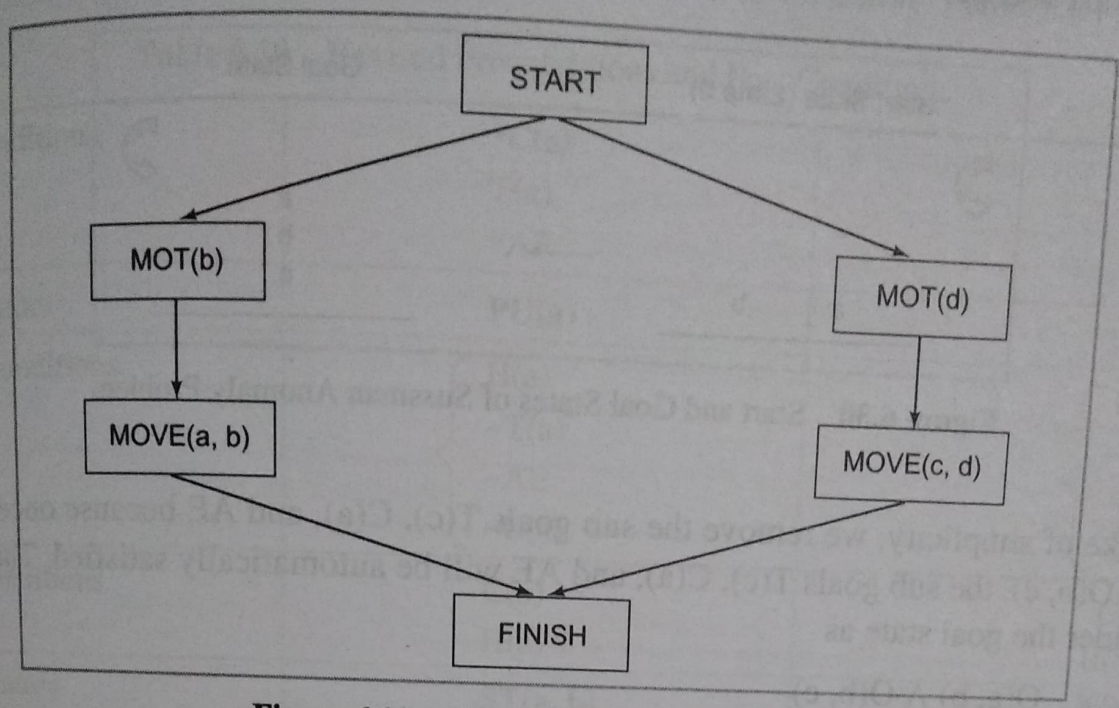

Figure 6.29 Graph for Partial Order Plans

This representation for partial plans enables the planner to consider a number of different total plans without bothering about ordering actions that are not dependent on each other. For example, six total plans can be generated as described in Table 6.8. Each of these is referred to as a *linearization* of the partial-order plan.

Table 6.8 Total Plans

Plan1	Plan2	Plan3	Plan4	Plan5	Plan6
MOT(b)	MOT(b)	MOT(b)	MOT(d)	MOT(d)	MOT(d)
MOVE(a, b)	MOT(d)	MOT(d)	MOVE(c, d)	MOT(b)	MOT(b)
MOT(d)	MOVE(a, b)	MOT(c, d)	MOT(b)	MOVE(c, d)	MOVE(a, b)
MOVE(c, d)	MOVE(c, d)	MOVE(a, b)	MOVE(a, b)	MOVE(a, b)	MOVE(c, d)

Partial-order planning can be implemented as a search in the space of partial-order plans.

6.7.2 Constraint Posting Method

The basic idea behind constraint posting method is also to generate a plan by incrementally hypothesizing operators, partial ordering between operators, and binding of variables within operators. At any given time in the planning process, a solution is a partially ordered sequence of operators with partially instantiated set of operators. Any number of total orders may be obtained from the partially ordered sequence of operators to generate the actual plan (Rich & Knight, 2003). Let us consider the Sussman anomaly problem again as shown in Fig. 6.30 and solve it by incrementally generating a non-linear plan. The start and goal states may be represented as

Start State O(c, a) Λ T(a) Λ T(b) Λ C(c) Λ C(b) Λ AE

Goal State O(a, b) Λ O(b, c) Λ T(c) Λ C(a) Λ AE