CHAPTER 2

Unit II

Planning and Constraint Satisfaction

Syllabus Topics

Problem Decomposition : Goal Trees, Rule Based Systems, Rule Based Expert Systems.

Planning : STRIPS, Forward and Backward State Space Planning, Goal Stack Planning, Plan Space Planning, A Unified Framework For Planning.

Constraint Satisfaction : N-Queens, Constraint Propagation, Scene Labeling, Higher order and Directional Consistencies, Backtracking and Look ahead Strategies.

Syllabus Topic : Problem Decomposition

2.1 Problem Decomposition

DEFINITION

Problem decomposition *is the problem solving strategy of breaking a problem up into a set of subproblems, solving each of the subproblems, and then composing a solution to the original problem from the solutions to the subproblems.*

- **Example :** The Necklace Problem
- A necklace is constructed from 20 blue beads and 14 red beads. Each blue bead costs $0.22 while the red beads cost $0.33 each. The cost of the thread is $0.14 per inch, and 24 inches are required. What is the total cost of materials for the necklace ?
- The necklace problem provides a very pure application of the problem decomposition strategy. We can decompose the main problem into three subproblems.
  1. Determine the cost of the blue beads
  2. Determine the cost of the red beads
  3. Determine the cost of the thread
- We might then decompose each of these three problems into two distinct subproblems, each of whose solutions is "trivial".
  (a) Determine the cost of one blue bead, and
  (b) Determine the number of blue beads.
  (c) Determine the cost of one red bead, and
  (d) Determine the number of red beads.
  (e) Determine the unit cost of the thread, and
  (f) Determine the length of the thread.

Syllabus Topic : Goal Trees

2.2 Goal Trees

- Goal tree is nothing but a tree comprising of node representing the goals or sub goals and links representing the actions in order to achieve that goal. In a problem decomposition space, the nodes represent problems to be solved or goals to be achieved, and the edges represent the decomposition of the problem into sub problems.
- Goal tree can be best illustrated by the example of the Towers of Hanoi problem.

- The root node, labelled "3AC" represents the original problem of transferring all 3 disks from peg A to peg C. The goal can be decomposed into three sub goals : 2AB, 1AC, 2BC. In order to achieve the goal, all 3 sub goals must be achieved.

Syllabus Topic : Rule Based Expert Systems

2.2.1 Rule Based Expert Systems

- Let's first understand the components of expert systems.
- Following are the two principle components of every expert system.

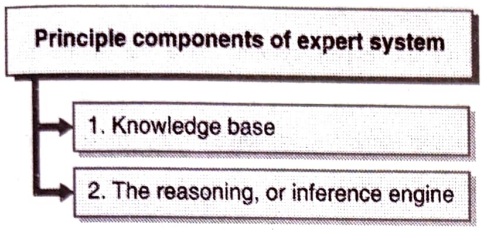


Fig. 2.2.1 : Principle components of expert system

- Every expert system is developed for a specific task domain. It is the area where human intelligence is required. Task refers to some goal oriented problem solving activity and Domain refers to the area within which the task is being performed.
- Consider the expert system architecture in Fig. 2.2.2.

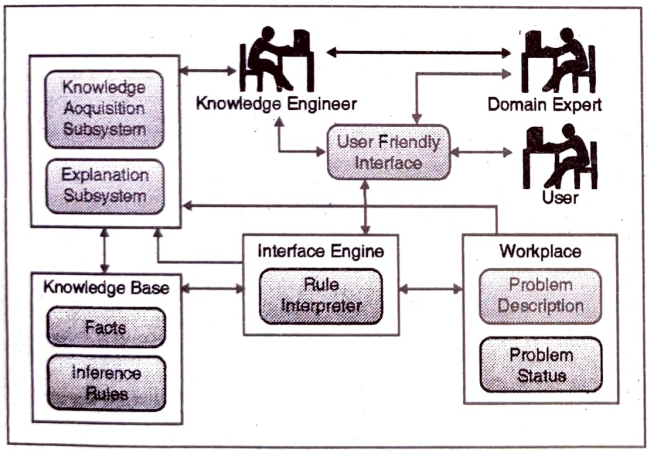


Fig. 2.2.2 : Components of Expert System

- If we consider inference engine as the brain of the expert systems, then knowledge base is the heart. As the heart is more powerful, the brain can function faster and efficient way. Hence the success of any expert system is more or less depends on the quality of knowledgebase it works on.

➜ **1. Knowledge base**

There are two types of knowledge expert systems can have about the task domain.

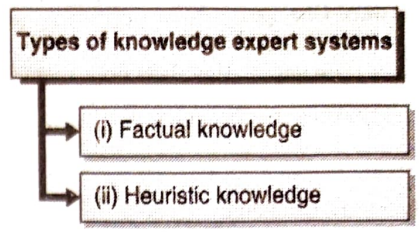


Fig. 2.2.3 : Types of knowledge expert systems

➜ **(i) Factual knowledge**

It is the knowledge which is accepted widely as a standard knowledge. It is available in text books, research journals and internet. It is generally accepted and verified by domain experts or researchers of that particular field.

➜ **(ii) Heuristic knowledge**

It is experiential, judgmental and may not be approved or acknowledged publically. This type of knowledge is rarely discussed and is largely individualistic. It doesn't have standards for evaluation of its correctness. It is the knowledge of good practice, good judgment and probable reasoning in the domain. It is the knowledge that is based on the "art of good guessing." It is very subjective to the practitioner's knowledge, and experience in the respective problem domain.

The knowledge base an expert uses is based on his learning from various sources over a time period. Hence, the knowledge store of an expert increases with number of years of experience in the given field, which allows him to interpret the information in his databases to advantage in diagnosis, analysis and design.

➜ **2. Inference engine**

The inference engine has a problem solving module which organizes and controls the steps required to solve the problem. A common but powerful inference engine involves chaining of IF…THEN rules to form a line of reasoning.

There are two types of chaining practices to solve a problem.

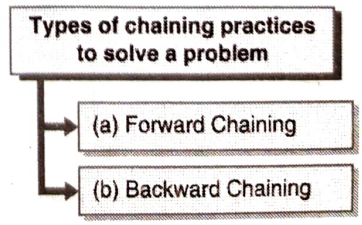


Fig. 2.2.4 : Types of chaining practices to solve a problem

➜ **(a) Forward Chaining**

- This type of reasoning strategy starts from a set of conditions and moves toward some conclusion, also called as data driven approach.

➜ **(b) Backward Chaining**

- Backward chaining is a goal driven approach. In this type of reasoning, the conclusion is known and the path to the conclusion needs to be found out. For example, a goal state is given, but the path to that state from start state is not known, then backward reasoning is used.

- Inference engine is nothing but these methods implemented as program modules. Inference engine manipulates and uses knowledge in the knowledge base to generate a line of reasoning.

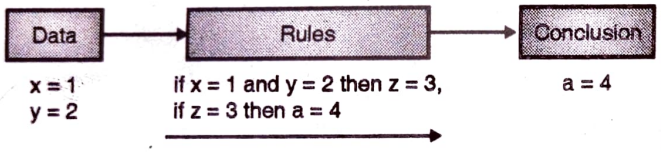


Fig. 2.2.5 : Forward Chaining

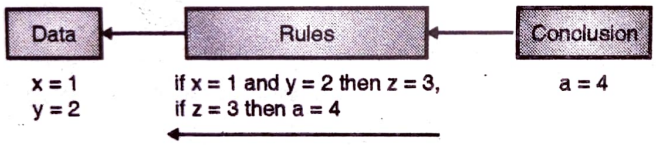


Fig. 2.2.6 : Backward Chaining

- Knowledge is most of the times incomplete and uncertain. There are various ways to deal with uncertainty in knowledge. One of the simplest methods is to associate a weight or a confidence factor with each rule.

- The set of methods used to represent uncertain knowledge in combination with uncertain data in the reasoning process is called reasoning with uncertainty. "Fuzzy Logic" is an important area of artificial intelligence which provides methods for reasoning with uncertainty and the systems that use it are called as "fuzzy systems".

☞ **Explanation subsystem**

- Most of the times, human experts can guess or use their gut feeling to approximate or estimate the results. As an expert system try to mimic human thinking, it uses uncertain or heuristic knowledge as we humans do.

- As a result, its credibility of the system is often in question, as is the case with humans. As an answer to a problem itself is questionable, one urges to know the rationale or the reasoning behind the answer.

- If the rationale seems to be acceptable, generally the answer is considered to be correct. So is the case with expert systems!!

- Most of the expert systems have the ability to answer questions of the form : "Why is the answer X?" Inference engine can generate explanations by tracing the line of reasoning used by it.

☞ **Knowledge engineer**

- Building an expert system is also known as "knowledge engineering" and its practitioners are called knowledge engineers.

- The primary job of knowledge engineer is to make sure that the expert system has all the knowledge needed to solve a problem. He does a vital task of gathering knowledge from domain experts.

- Knowledge engineer needs to learn how the domain expert reasons with their knowledge by interviewing them. Then he translates his knowledge into programs using which he designs the interface engine.

- There might be some uncertain knowledge involved in the knowledge base; knowledge engineer needs to decide how to integrate this with the available knowledge base.

- Lastly, he needs to decide upon what kind of explanations will be required by the user, and according to that he designs the inference levels.

Syllabus Topic : Planning

2.3 Planning

DEFINITION

Planning in Artificial Intelligent can be defined as a problem that needs decision making by intelligent systems to accomplish the given target.

- The intelligent system can be a robot or a computer program.

- Take example of a driver who has to pick up and drop people from one place to another. Say he has to pick up two people

from two different places then he has to follow some sequence, he cannot pick both passengers at same time.

- There is one more definition of planning which says that, **Planning** is an activity where agent has to come up with a sequence of actions to accomplish target.
- Now, let us see what information is available while formulating a planning problem and what results are expected.
- We have information about the initial status of the agent, goal conditions of agent and set of actions an agent can take.
- Aim of an agent is to find the proper sequence of actions which will lead from starting state to goal state and produce an efficient solution.

2.3.1 Simple Planning Agent

- Take example of an agent which can be a coffee maker, a printer and a mailing system, also assume that there are 3 people who have access to this agent.

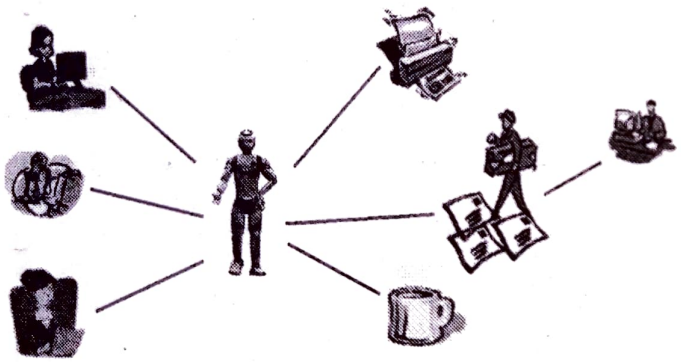


Fig. 2.3.1 : Example of a Planning Problem

- Suppose at same time if all 3 users of an agent give a command to execute 3 different tasks of coffee making, printing and sending a mail.
- Then as per definition of planning, agent has to decide the sequence of these actions.
- Fig. 2.3.2 depicts a general diagrammatic representation of a planning agent that interacts with environment with its sensors and effectors/actuators.
- When a task comes to this agent it has to decide the sequence of actions to be taken and then accordingly execute these actions.

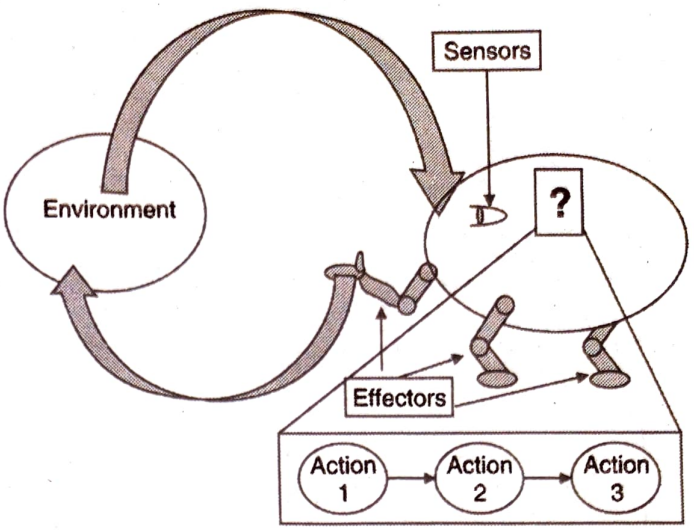


Fig. 2.3.2 : Planning agent

Syllabus Topic : STRIPS

2.4 STRIPS

- Language should be expressive enough to explain a wide variety of problems and restrictive enough to allow efficient algorithms to operate on it.
- Planning languages are known as action languages.

☞ **Stanford Research Institute Problem Solver (STRIPS)**

- Richard Fikes and Nils Nilsson developed an automated planner called STRIPS (Stanford Research Institute Problem Solver) in 1971.
- Later on this name was given to a formal planning language. STRIPS is foundation for most of the languages in order to express automated planning problem instances in current use.

☞ **Action Description Language (ADL)**

ADL is an advancement of STRIPS. Pednault proposed ADL in 1987.

☞ **Comparison between STRIPS and ADL**

Sr. No.	STRIPS language	ADL
1.	Only allows positive literals in the states, **For example :** A valid sentence in STRIPS is expressed as ⇒ Intelligent^ Beautiful.	Can support both positive and negative literals. **For example :** Same sentence is expressed as ⇒ ¬Stupid ∧ ¬Ugly

Sr. No.	STRIPS language	ADL
2.	Makes use of closed-world assumption (i.e. Unmentioned literals are false)	Makes use of Open World Assumption (i.e. unmentioned literals are unknown)
3.	We only can find ground literals in goals. **For example :** Intelligent ∧ Beautiful.	We can find quantified variables in goals. **For example :** ∃x At (P1, x) ∧ At(P2, x) is the goal of having P1 and P2 in the same place in the example of the blocks.
4.	Goals are conjunctions **For example :** Intelligent ∧ Beautiful	Goals may involve conjunctions and disjunctions **For example :** Intelligent ∧ (Beautiful ∨ Rich).
5.	Effects are conjunctions.	Conditional effects are allowed : when P:E means E is an effect only if P is satisfied.
6.	Does not support equality.	Equality predicate (x = y) is built in.
7.	Does not have support for types.	Supported for types **For example :** The variable p : Person

2.4.1 Example of Block World Puzzle

Q. 2.4.1 Design planning agent to solve block world problem. Assume suitable initial state and final state for the problem. (Refer section 2.4.1) **(8 Marks)**

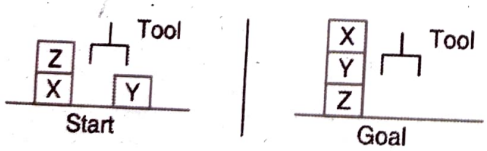


Fig. 2.4.1

Standard sequence of actions

1. Grab Z and Pickup Z
2. Then Place Z on the table
3. Grab Y and Pickup Y
4. Then Stack Y on Z
5. Grab X and Pickup X
6. Stack X on Y

- Elementary problem is that framing problem in AI is concerned with the question of what piece of knowledge or information is pertinent to the situation.
- To solve this problem we have tom make an Elementary Assumption which is a Closed world assumption. (i.e. If something is not asserted in the knowledge base then it is assumed to be false, this is also called as "Negation by failure")
- Standard sequence of actions can be given as for the block world problem :

on(Y, table)	on(Z, table)
on(X, table)	on(Y, Z)
on(Z, X)	on(X, Y)
hand empty	hand empty
clear(Z)	clear(X)
clear(Y)	

Fig. 2.4.2

- We can write 4 main rules for the block world problem as follows :

	Rule	Precondition and Deletion List	Add List
Rule 1	pickup(X)	hand empty, on(X, table), clear(X)	holding(X)
Rule 2	putdown(X)	holding(X)	hand empty, on(X, table), clear(X)
Rule 3	stack(X,Y)	holding(X), clear(Y)	on(X,Y), clear(X)
Rule 4	unstack(X,Y)	on(X,Y), clear(X)	holding(X), clear(Y)

Based on the above rules, plan for the block world problem : Start → goal can be specified as follows :

1. unstack(Z,X)
2. putdown(Z)
3. pickup(Y)
4. stack(Y,Z)
5. pickup(X)
6. stack(X,Y)

Execution of this plan can be done by making use of a data structure called "Triangular Table".

1	on (C, A) clear (C) hand empty	unstuck (C, A)				
2		holding (C)	putdown (C)			
3	on (B, table)		hand empty	pickup (B)		
4			clear (C)	Holding(B)	stack (B,C)	
5	on (A, table)	clear (A)			Hand empty	pickup (A)
6					clear (B)	holding (A)	stack (A, B)
7			on (C, table)		on (B, C)		on (A, B) clear (A)
	0	1	2	3	4	5	6

Fig. 2.4.3

- In a triangular table there are N + 1 rows and columns. It can be seen from the Fig. 2.4.4 that rows have 1 → n + 1 condition and for columns 0 → n condition is followed. The first column of the triangular table indicates the starting state and the last row of the triangular table indicates the goal state.
- With the help of triangular table a tree is formed as shown below to achieve the goal state :

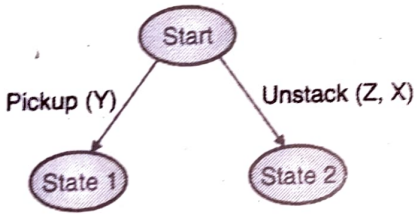


Fig. 2.4.4

- An agent (in this case robotic arm) can have some amount of fault tolerance. Fig. 2.4.5 shows one such example.

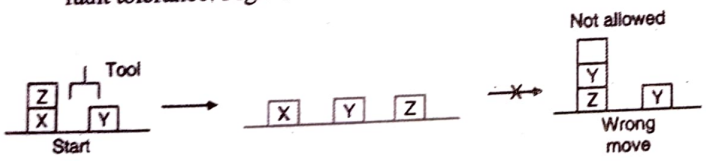


Fig. 2.4.5

2.4.2 Example of the Spare Tire Problem

Q. 2.4.2 Explain planning problem for spare tire problem. (Refer section 2.4.2) **(8 Marks)**

- Consider the problem of changing a flat tire. More precisely, the goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is a very abstract one, with no sticky lug nuts or other complications.
- There are just four actions - removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.
- The ADL description of the problem is shown. Notice that it is purely propositional. It goes beyond STRIPS in that it uses a negated precondition, ¬At(Flat, Axle), for the PutOn(Spare, Axle) action. This could be avoided by using Clear (Axle) instead, as we will see in the next example.

☞ **Solution using STRIPS**

Init(At(Flat, Axle) ∧ At(Spare, Trunk))

Goal(At(Spare, Axle)) Action(Remove(Spare, Trunk),

PRECOND : At(Spare, Trunk)

EFFECT : ¬ At(Spare, Trunk) ∧ At(Spare, Ground))

Action(Remove(Flat, Axle),

PRECOND : At(Flat, Axle)

EFFECT : ¬ At(Flat, Axle) ∧ At(Flat, Ground))

Action(PutOn(Spare, Axle),

PRECOND : At(Spare, Ground) ∧ ¬ At(Flat, Axle)

EFFECT : ¬ At(Spare, Ground) ∧ At(Spare, Axle))

Action(LeaveOvernight)

PRECOND

EFFECT : ¬ At(Spare, Ground) ∧ ¬ At(Spare, Axle) ∧ ¬ At(Spare, Trunk) ∧ ¬ At(Flat, Ground) ∧ ¬ At(Flat, Axle))

Syllabus Topic : Forward and Backward State Space Planning

2.5 Forward and Backward State Space Planning

2.5.1 Progression Planners

Q. 2.5.1 Explain progression planners with example. (Refer section 2.5.1) **(7 Marks)**

- **"Forward state-space search"** is also called as **"progression planner".** It is a deterministic planning technique, as we plan sequence of actions starting from the initial state in order to attain the goal.
- With forward state space searching method we start with the initial state and go to final goal state. While doing this we need to consider the probable effects of the actions taken at every state.
- Thus the prerequisite for this type of planning is to have initial world state information, details of the available actions of the agent, and description of the goal state.
- Remember that details of the available actions include preconditions and effects of that action.
- See Fig. 2.5.1 it gives a state-space graph of progression planner for a simple example where Flight 1 is at location A and Flight 2 is also at location A. These Flights are moving from location A to location B. In 1st case only Flight 1 moves from location A to location B, so the resulting state shows that after performing that action Flight 1 is at location B whereas Flight 2 is at its original location A. Similarly In 2nd case only Flight 2 moves from location A to location B and the resulting state shows that after performing that action Flight 2 is at location B while Flight 1 is at its original location A.
- It can be observed from the Fig. 2.5.1 that, rectangles show state of the flights (i.e. their current location), and lines give the corresponding actions from one state to another (i.e. move from one location to other location).
- Note that the lines coming out of every state matches to all of the permissible actions which can be accepted if the agent is in that state.

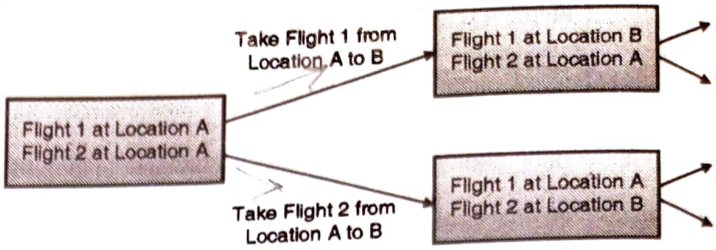


Fig. 2.5.1 : State-space graph of a Progression planners

Progression planner algorithm

1. Formulize the state space search problem :
   - Initial state is the first state of the planning problem which has a set of positive, the literals which don't appear are considered as false.
   - If preconditions are satisfied then the actions are favoured i.e. if the preconditions are satisfied then positive effect literals are added for that action else the negative effect literals are deleted for that action.
   - Perform goal testing by checking if the state will satisfy the goal.
   - Lastly keep the step cost for each action as 1.
2. Consider example of A* algorithm. A complete graph search is considered as a complete planning algorithm. Functions are not used.
3. Progression planner algorithm is supposed to be inefficient because of the irrelevant action problem and requirement of good heuristics for efficient search.

2.5.2 Regression Planners

Q. 2.5.2 Explain regression planning. (Refer section 2.5.2) **(6 Marks)**

Q. 2.5.3 Explain regression planners with example. (Refer section 2.5.2) **(8 Marks)**

- **"Backward state-space search"** is also called as **"regression planner"** from the name of this method you can make out that the processing will start from the finishing state and then you will go backwards to the initial state.
- So basically we try to backtrack the scenario and find out the best possibility, in-order to achieve the goal to achieve this we have to see what might have been correct action at previous state.

- In forward state space search we used to need information about the successors of the current state now, for backward state-space search we will need information about the predecessors of the current state.
- Here the problem is that there can be many possible goal states which are equally acceptable. That is why this approach is not considered as a practical approach when there are large numbers of states which satisfy the goal.
- Let us see flight example, here you can see that the goal state is Flight 1 is at location B and Flight 2 is also at location B. We can see in Fig. 2.5.2 that if this state is checked backwards we have two acceptable states in one state only Flight 2 is at location B, but Flight 1 is at location A and similarly in 2nd possible state Flight 1 is already at location B, but Flight 2 is at location A.
- As we search backwards from goal state to initial state, we have to deal with partial information about the state, since we do not yet know what actions will get us to goal. This method is complex because we have to achieve a conjunction of goals.

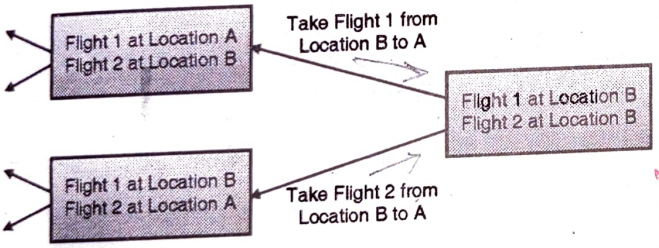


Fig. 2.5.2 : State-space graph of a regression planners

- In this Fig. 2.5.2 rectangles are goals that must be achieved and lines shows the corresponding actions.

Regression algorithm

1. Firstly predecessors should be determined :
   - To do this we need to find out which states will lead to the goal state after applying some actions on it.
   - We take conjunction of all such states and choose one action to achieve the goal state.
   - If we say that "X" action is relevant action for first conjunct then, only if pre-conditions are satisfied it works.
   - Previous state is checked to see if the sub-goals are achieved.
2. Actions must be consistent it should not undo preferred literals. If there are positive effects of actions which appear in goal then they are deleted. Otherwise Each precondition literal of action is added, except it already appears.
3. Main advantage of this method is only relevant actions are taken into consideration. Compared to forward search, backward search method has much lower branching factor.

2.5.3 Heuristics for State-Space Search

- Progression or Regression is not very efficient with complex problems. They need good heuristic to achieve better efficiency. Best solution is NP Hard (NP stands for Non-deterministic and Polynomial-time).
- There are two ways to make state space search efficient :
  1. **Use linear method :** Add the steps which build on their immediate successors or predecessors.
  2. **Use partial planning method :** As per the requirement at execution time ordering constraints are imposed on agent.

Syllabus Topic : Goal Stack Planning

2.6 Goal Stack Planning

Basic Idea to handle interactive compound goals uses goal stacks, Here the stack contains :

- Goals,
- Operators - ADD, DELETE and PREREQUISITE lists
- A database maintaining the current situation for each operator used.

Consider the following where wish to proceed from the *start* to *goal* state.

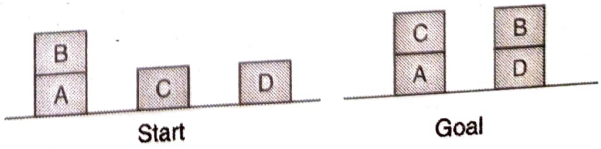


Fig. 2.6.1 : Goal Stack Planning Example

- We can describe the *start* state :

  ON(B, A) ∧ ONTABLE(A) ∧ ONTABLE(C)

  ∧ ONTABLE(D) ∧ ARMEMPTY

  And *goal* state :

  ON(C, A) ∧ ON(B,D) ∧ ONTABLE(A) ∧ ONTABLE(D)

- Initially the goal stack is the goal state.
- We then split the problem into four subproblems
- Two are solved as they already are true in the initial state - ONTABLE(A), ONTABLE(D).
- With the other two - there are two ways to proceed :

ON(C,A)

ON(B,D)

ON(C,A) ∧ ON(B,D)

∧ ONTABLE(A) ∧ ONTABLE(D)

ON(B,D)

ON(C,A)

ON(C,A) ON(B,D)

∧ ONTABLE(A) ∧ ONTABLE(D)

The method is to,

- Investigate the first node on the stack i.e. the top goal.
- If a sequence of operators is found that satisfies this goal it is removed and the next goal is attempted.
- This continues until the goal state is empty.

Consider alternative 1 above further :

- The first goal ON(C,A) is not true and the only operator that would make it true is STACK (C,A) which replaces ON(C,A) giving :

B<>STACK (C,A)

ON(B,D)

ON(C,A) ∧ ON(B,D)

∧ ONTABLE(A) ∧ ONTABLE(D)

- STACK has prerequisites that must be met which means that block A is clear and the arm is holding block C.
- So we must do :

B<>CLEAR(A)

HOLDING(C)

CLEAR(A) ∧ HOLDING(C)

- Now top goal is false and can only be made true by unstacking B. This leads to :

B<>ON(B,A)

CLEAR(B)

ARMEMPTY

ON(B,A) ∧ CLEAR(B)

∧ ARMEMPTY

UNSTACK(B,A)

HOLDING(C)

CLEAR(A) ∧ HOLDING(C)

- Now the first goal is true, the second is universally true, and the arm is empty. Thus all top three goals are true means that we can apply the operator UNSTACK(B,A) as all prerequisites are met. This gives us the first node in database.

ONTABLE(A) ∧ ONTABLE(C) ∧ ONTABLE(D)

∧ HOLDING(C) ∧ CLEAR(A)

- Note as a future reference of the use of UNSTACK(B,A) that HOLDING(B) is true as well as CLEAR(A)
- The goal stack becomes

HOLDING(C)

CLEAR(A) ∧ HOLDING(C)

STACK (C,A)

ON(B,D)

ON(C,A) ∧ ON(B,D) ∧ ONTABLE(A)

^ ONTABLE(D)

- There are two ways we can achieve HOLDING(C) by using the operators PICKUP(C) or UNSTACK(C,x) where x is an unspecified block.
- This leads to two alternative paths :

ON(C, x)

CLEAR(C)

ARMEMPTY

ON(C, x) ∧ CLEAR(C) ∧ ARMEMPTY

UNSTACK(C, x)

CLEAR(A) ∧ HOLDING(C)

STACK (C, A)

ON(B, D)

ON(C,A) ∧ ON(B, D) ∧ ONTABLE(A) ∧ ONTABLE(D)

ONTABLE(C)

CLEAR(C)

ARMEMPTY

ONTABLE(C) ∧ CLEAR(C) ∧ ARMEMPTY

PICKUP(C)

CLEAR(A) ∧ HOLDING(C)

STACK (C,A)

ON(B,D)

ON(C,A) ∧ ON(B,D) ∧ ONTABLE(A) ∧ ONTABLE(D)

- In this first route we can see three references to some block, x and these must refer to the same block, although in the search it is conceivable several blocks will become temporarily attached. Hence the binding of variables to blocks must be recorded. Investigating further we need to satisfy the first goal and this requires stacking C on some block which is clear.

CLEAR(x)

HOLDING(C)

CLEAR(x) ∧ HOLDING(C)

STACK (C, x)

CLEAR(C)

ARMEMPTY

- We now notice that one of the goals created is HOLDING(C) which was the goal we were trying to achieve by applying UNSTACK(C, *some block*) in this case and PICKUP(C) in the other approach. So it would appear that we have added new goals and not made progress and in terms of the A* algorithm it seems best to try the other approach.
- So looking at the second approach :
  - We can see that the first goal is achieved block C is on the table.
  - The second goal is also achieved block C is clear.
  - Remember that HOLDING(B) is still true which means that the arm is not empty. This can be achieved by placing B on the table or planting it on block D if it is clear.
  - Look ahead could be used here to compare the ADD lists of the competing operators with the goals in the goal stack and there is a match with ON(B,D) which is satisfied by STACK (B,D). This also binds some block to block D.
  - Applying STACK (B,D) generates extra goals CLEAR(D) and HOLDING(B)

- The new goal stack becomes;

CLEAR(*D*)

HOLDING(*B*)

CLEAR(*D*) ∧ HOLDING(*B*)

STACK (*B*, *D*)

ONTABLE(*C*) ∧ CLEAR(*C*) ∧ ARMEMPTY

PICKUP(*C*)

- At this point the top goal is true and the next and thus the combined goal leading to the application of STACK (*B*,*D*), which means that the world model becomes

ONTABLE(*A*) ∧ ONTABLE(*C*) ∧ ONTABLE(*D*) ∧ ON(*B*,*D*) ∧ ARMEMPTY

- This means that we can perform PICKUP(*C*) and then STACK (*C*,*A*)
- Now coming to the goal ON(*B*,*D*) we realise that this has already been achieved and checking the final goal we derive the following plan

UNSTACK(*B*,*A*)

STACK (*B*,*D*)

PICKUP(*C*)

STACK (*C*,*A*)

- This method produces a plan using good Artificial Intelligence techniques such as heuristics to find matching goals and the A* algorithm to detect unpromising paths which can be discarded.

Syllabus Topic : Plan Space Planning

2.7 Plan Space Planning

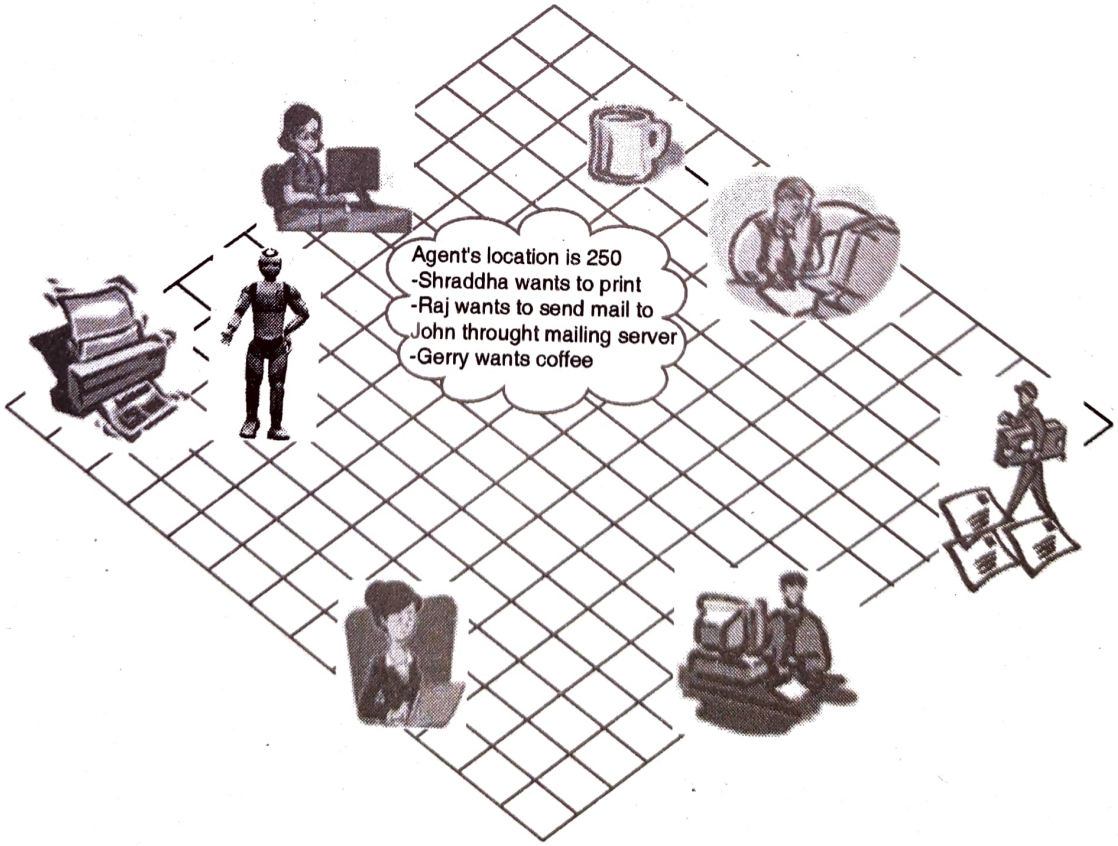


Fig. 2.7.1 : Office agent example with finite state space

- We have seen example of an agent that can perform three tasks of printing, sending a mail and making coffee namely lets' call this agent as office agent.
- When this office agent gets order from three people at a same time to perform these three different tasks then, let us see how planning with state space search problem will look if we have a finite space.
- You can understand from Fig. 2.7.1 that the office agent is at location 250 on the state space grid. When he gets a task he has to decide which task can be performed more efficiently in lesser time.
- If it finds some input and output locations nearer on state space grid for example in case of printing task then the probability of performing that task will increase.
- But to do this it should be aware of it's own current location, the locations of people who are assigning tasks and the locations of the required devices.
- State space search is unfavourable for solving real-world problems because, it requires complete description of every searched state, also search should be carried out locally.
- There can be two ways of representations for a state :

1. Complete world description
2. Path from an initial state

1. Complete world description

- Description is available in terms of an assignment of a value to each previous suggestion.
- Or we can say that description is available as a suggestion that defines the state.
- Drawback of this types is that it requires a large amount of space.

Path from an initial state

- As per the name, path from an initial state gives the sequence of actions which are used to reach a state from an initial state.
- In this case, what holds in a state can be deduced from the axiom which specifies the effects of an actions.
- Drawback of these types is that it does not explicitly specify "What holds in every state". Because of this it can be difficult to determine whether two states are same.

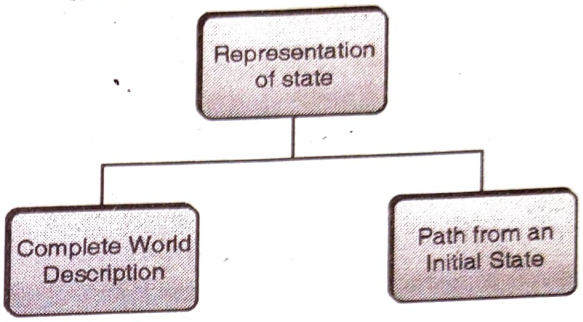


Fig. 2.7.2 : Representation of states

2.7.1 Example of State Space Search

Q. 2.7.1 Explain Water Jug problem with State Space Search Method. (Refer section 2.7.1) **(8 Marks)**

Let us take an example of a water jug problem :

- We have two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water how can you get exact 2 gallons of water into the 4-gallon jug?
- The state space for this problem can be described as of ordered pairs of integers (x, y), such that x = 0,1,2,3 or 4, representing the number of gallons of water in the 4-gallon jug and y = 0,1,2 or 3, representing the quantity of water in the 3-gallon jug. The start state is (0, 0). The goal state is (2, n) for any value of n, since the problem does not specify how many gallons need to be in the 3- gallon jug.
- The operators to be used to solve the problem can be described as shown bellow. They are represented as rules whose left side are matched against the current state and whose right sides describe the new state that results from applying the rule.

Ruleset

1.	(x,y)	→	(4,y) fill the 4- gallon jug
	If x<4		
2.	(x,y)	→	(x,3) fill the 3-gallon jug
	If x<3		
3.	(x,y)	→	(x-d,y) pour some water out of the 4- gallon jug
	If x>0		
4.	(x,y)	→	(x-d,y) pour some water out of the 3- gallon jug
	If y>0		
5.	(x,y)	→	(0,y) empty the 4- gallon jug on the ground
	If x>0		
6.	(x,y)	→	(x,0) empty the 3- gallon jug on the ground
	If y>0		
7.	(x,y)	→	(4,y-(4-x)) pour water from the 3- gallon jug into the 4-gallon
	If x+y>=4 and y>0 jug until the 4-galoon jug is full		
8.	(x,y)	→	(x-(3-y),3)) pour water from the 4- gallon jug into the 3-gallon
	If x+y>=3 and x>0 jug until the 3-gallon jug is full		
9.	(x,y)	→	(x+y,0) pour all the water from the 3 -gallon jug into
	If x+y<=4 and y>0 the 3-gallon jug		
10.	(x,y)	→	(0,x+y) pour all the water from the 4 -gallon jug into
	If x+y<=3 and x>0 the 3-gallon jug		
11.	(0,2)	→	(2,0) pour the 2-gallon from the 3 -gallon jug into the 4-gallon jug
12.	(2,y)	→	(0,x) empty the 2 gallon in the 4 gallon on the ground

Production for the water jug problem

Gallons in the 4- gallon Jug	Gallons in the 3- gallon	Rule Applied
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5 or 12
2	0	9 or 11

One solution to the water jug problem.

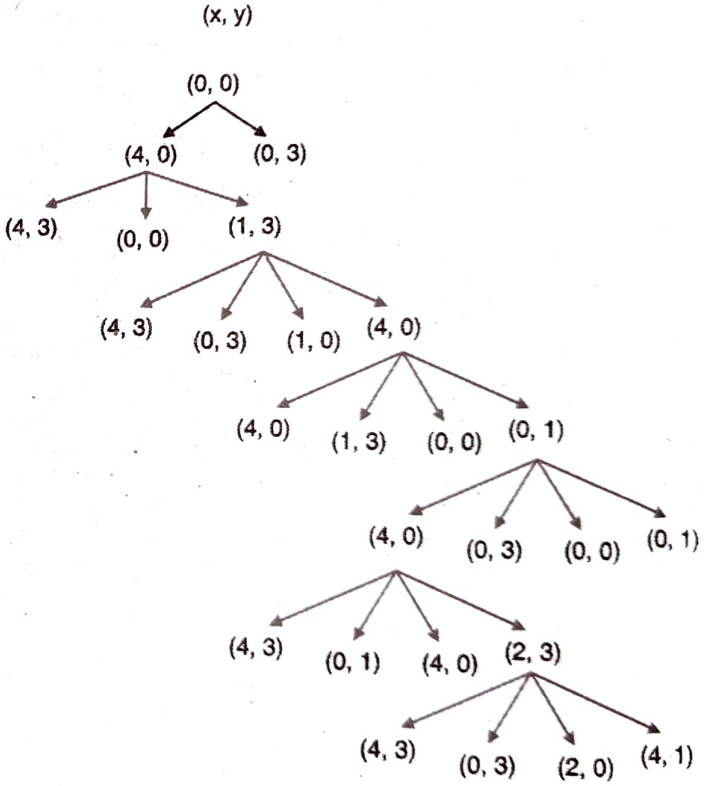


Fig. 2.7.3

2.7.2 Classification of Planning with State Space Search

Q. 2.7.2 How planning strategies are classified? (Refer section 2.7.2) **(7 Marks)**

- As the name suggests state space search planning techniques is based on the spatial searching.
- Planning with state space search can be done by both forward and backward state-space search techniques.

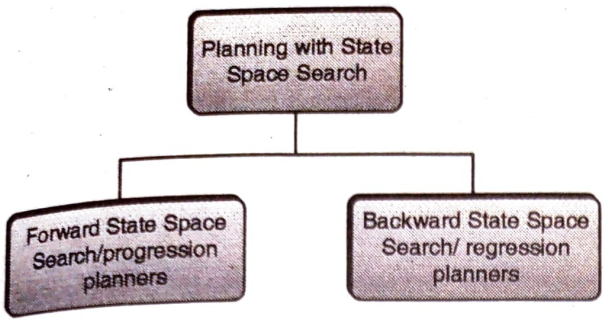


Fig. 2.7.4 : Classification of Planning with State Space Search

Syllabus Topic : A Unified Framework for Planning

2.8 A Unified Framework for Planning

- In typical planning problems, we have set of preconditions which must be satisfied before we start with actual actions and there are effects of the action. These pre conditions and effects can be represented as operators. For real world planning problems these operators represent some complex control problem on their own. Take example of block world problem, where a robotic arm is used to pick and place the blocks. In this case number of contingencies can occur like, motor failure or obstruction by an obstacle. This will potentially affect the outcome and applicability of the action. Hence to incorporate all the possibilities of the real world scenarios operators must be complex in nature.
- However, the complex operators will make the plan untraceable, which the planner will never be interested in. So there arises a conflict. By simplifying the operators, the reliability itself will be challenged and by using complex operators, we may not be able to solve the planning problem.
- We have to maintain both simplicity and reliability in the planning operators. In order to solve this matter, if we can identify those details of the world dynamics likely to become relevant within our distribution of problems and structure our operators accordingly, we may achieve the goal to certain extent.
- Say in case of block world problem, possibility of the arm getting stuck by an obstacle is high only when it is placed in crowded environment; which is mostly not likely. So while designing the planner, we may skip this detail.
- Fig. 2.8.1 depicts the scenario of classical planning proceeds.

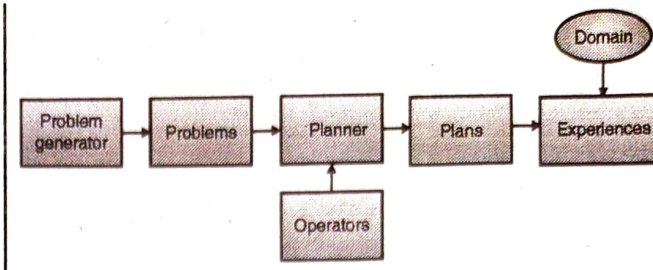


Fig. 2.8.1 : Classical planning framework

- The **problem generator** produces planning problems according to some fixed but unknown probabilities. The **planner** takes in the problems, along with a library of operators, and outputs plans, which, when applied to the domain, result in some set of experiences. In this framework, the operators are designed by a domain expert and are immutable.
- As the domain expert designs the operators, they are kept simple in order to keep the planning phase manageable. Hence there is less specificity of the operators. Many unimportant details might get covered while losing some critical specifications.
- For example, operators like motor failure in case of robot are is introduced which is very less likely as all the robot are well maintained. While, an important aspect such as the presence of obstacles in the arm's path might get ignored; which is very critical if our tasks frequently take place in a crowded environment.
- In order to avoid such situations, the planning process is supplemented by a new operator design module, detailed in Fig. 2.8.2. Instead of planning with an a priori fixed set of operators, specified by a domain expert, this module takes advantage of the observed world experiences to tailor operator definitions to the particular distribution of planning problems. The domain expert specifies only a general body of domain knowledge. The knowledge can be of varying specificities, and need not be consistent.
- Three submodules make up the operator design module.

1. **Explanation submodule :** It is used to associate observed world dynamics with a consistent causal model, calling on explanations from the knowledge base.

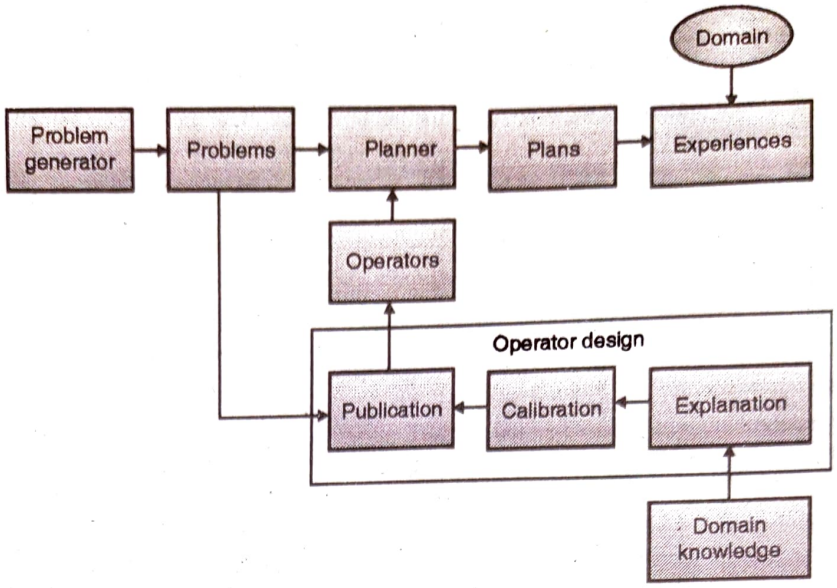


Fig. 2.8.2 : Addition of Operator Design Module

2. **Calibration mechanism** : It associates precise numerical functions to the qualitative causal structure.
3. **Publication module :** Assesses the capabilities of our operators against the distribution of planning problems, and produces a minimal sufficient set of operator definitions for use by the planner.

Example : In case of block world problem, we can have operator as below,

- Op(ACTION : Unstack(b, x),
- PRECOND : On(b, x) and Clear(b),
- EFFECT : On(b, Table) and Clear(x) and !On(b, x))

Syllabus Topic : Constraint Satisfaction

2.9 Constraint Satisfaction

Constraint

Constraint is a logical relation among variables. Constraints arise in most areas of human endeavor. They are a natural medium for people to express problems in many fields.

Examples of constraints

- The sum of three angles of a triangle is 180 degrees,
- The sum of the currents flowing into a node must equal zero.

Constraint satisfaction

- The Constraint satisfaction is a process of finding a solution to a set of constraints.
- Constraints articulate allowed values for variables, and finding solution is evaluation of these variables that satisfies all constraints.

Constraint Satisfaction Problems (CSPs)

- In standard search problems, we are given with state space, heuristic function and goal state. In this case the state is represented in the form of any data structure that supports successor function, heuristic function, and goal test.
- Constraint Satisfaction Problems are special type of search in which, a state is defined by variables X_i with values from domain D_i and goal test is a set of constraints specifying allowable combinations of values for subsets of variables. Many real problems in AI can be modeled as Constraint Satisfaction Problems. CSP allows useful general purpose algorithms with more power than standard search algorithms CSPs are solved through search.

2.9.1 Examples of CSPs

Q. 2.9.1 Give examples of real time CSPs. (Refer section 2.9.1) **(6 Marks)**

Some poplar puzzles like, map coloring problem, Latin Square, Eight Queens, and Sudoku are stated below.

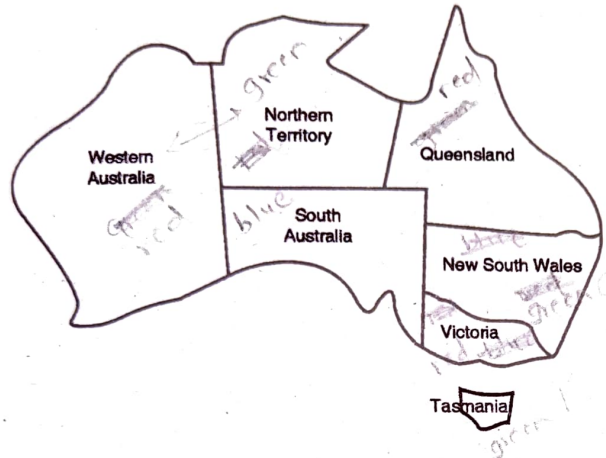
1. **Map coloring problem**

In this example, Variables are WA, NT, Q, NSW, V, SA, T

Domains D_i = {red, green, blue}

Constraints : adjacent regions must have different colors

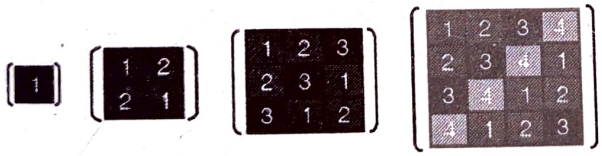
e.g., WA ≠ NT, or (WA,NT) in {(red, green),(red, blue), (green, red), (green, blue),(blue, red),(blue, green)}



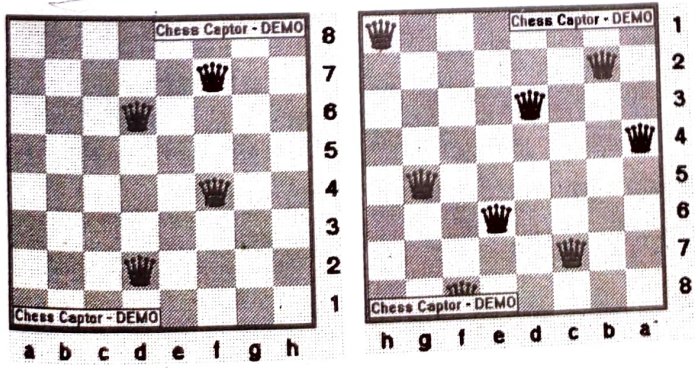
Solutions are complete and consistent assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

2. **Latin Square Problem :** How can one fill an n × n table with n different symbols such that each symbol occurs exactly once in each row and each column ?

Solutions : The Latin squares for n = 1, 2, 3 and 4 are :



3. **N-Queens Puzzle Problem :** How can one put 8 queens on a (8 × 8) chess board such that no queen can attack any other queen ?i.e. no two queens shall be placed in the same row or same column or diagonal to each other.



Attacking positions **Non-attacking position**

Solutions : The puzzle has 92 distinct solutions. If rotations and reflections of the board are counted as one, the puzzle has 12 unique solutions.

- This problem, first posed in a German chess magazine almost 150 years ago, requires that 8 queens be placed on an 8 × 8 chessboard in such a way that no two queens attack each other, i.e. no two queens occupy the same row, column, or diagonal. This problem can be formally represented as a CSP in several different ways, and as we shall see, the choice of formal representation has a significant impact on the computational complexity of the problem.
- Suppose, for example, that we choose to represent the problem by numbering the queens 1 through 8 and then defining a set z of 64 variables, one for each square of the chessboard, where each variable has domain} = {0,1,2,..., 8} and a variable z having a value of i means that queen i occupies the corresponding square if $1 \le i \le 8$ or that the square is unoccupied if i = 0. This formulation of the problem yields a search space of cardinality $9^{64} \approx 1.18 \times 10^{61}$, making it impractical from a computational perspective.
- Consider instead a representation with a set z of 8 variables $Q_1, Q_2, \ldots, Q_8$ representing the 8 rows of the board, with the domain of each variable Q_i being the set $D_i = \{1, 2, \ldots, 8\}$; in this case the value of the variable Q_i represents the column in the i throw in which a queen is placed. The constraints can then be specified as logical expressions of the form $i_6 = j \Rightarrow Q_i\ 6 = Q_j$ (i.e., no two queens are in the same column) and $i_6 = j \Rightarrow | Q_i - Q_j | 6 = | i - j |$. (i.e., no two queens are in the same diagonal). Note that we have implicitly incorporated one of the problem constraints (that no two queens occupy the same row) as well as our reasoning about the consequences of this constraint (it implies that each row must have precisely one queen in it) into our representation. Note too that the cardinality of the search space is now reduced to $8^8 \approx 1.68 \times 10^7$ states.
- This example clearly illustrates the need to carefully select a representation for a CSP from among the many formally valid possibilities, bearing in mind the computational complexity which each one entails.

☞ **Real-world CSPs**

1. Assignment problems : E.g., who teaches what class.
2. Timetabling problems : E.g., which class is offered when and where?
3. Transportation scheduling : Factory scheduling.

2.9.2 Varieties of CSPs

Q. 2.9.2 How CSPs can be classified ? Give example. (Refer section 2.9.2) **(7 Marks)**

Basis on various combinations of different types of variables and domains, we have varieties of CSPs.

They are as follows :

1. Discrete variables and finite domains : n variables, domain size $d \rightarrow O(d^n)$ complete assignments.

   e.g., Boolean CSP

2. Discrete variables and infinite domains : integers, strings, etc. range of values

   e.g., job scheduling, variables are start/end for each job need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

3. Continuous variables : E.g., start/end times for Hubble Space Telescope observations

4. Linear constraints solvable in polynomial time by linear programming.

2.9.3 Varieties of Constraints

1. Unary constraints involve a single variable.

   e.g., SA ≠ green

2. Binary constraints involve pairs of variables.

   e.g., SA ≠ WA

3. Higher-order constraints involve 3 or more variables.

   e.g., crypt arithmetic column constraints

2.9.4 Backtracking in CSPs

– *How to solve CSPs? Is there any standard method? Let's start with the straightforward approach, then fix it. States are defined by the values assigned so far*

1. **Initial state :** The empty assignment { }
2. **Successor function :** Assign a value to an unassigned variable that does not conflict with current assignment; fail if no legal assignments.
3. **Goal test :** The current assignment is complete.

– Variable assignments are commutative}, i.e.,[WA = red then NT = green] same as [NT = green then WA = red]. Only need to consider assignments to a single variable at each node. Depth-first search for CSPs with single-variable assignments is called backtracking search. Backtracking search is the basic uninformed algorithm for CSPs. Using this technique one can solve n-queens for $n \approx 25$.

☞ **Backtracking algorithm**

```
function BACKTRACKING-SEARCH(csp) returns a
solution, or failure
        return RECURSIVE-BACKTRACKING({}, csp)
function
RECURSIVE-BACKTRACKING(assignment, csp)returns a
solution, or failure
        if assignment is complete then return assignment
        var←SELECT-UNASSIGNED-
VARIABLE(Vaiables[csp], assignment,csp)
        for each value in ORDER-DOMAIN-
        VALUES(var, assignment,csp)do
            if value is consistent with assignment
              according to Constraints[csp] then
    add{var = value} to assignment
    result ← RECURSIVE-BACKTRACKING
    (assignment, csp)
                if result ≠ failure then return result
                remove {var = value} from assignment
            return failure
```

2.9.5 Improving Backtracking Efficiency

Q. 2.9.3 How to improve efficiency of backtracking in CSP? (Refer section 2.9.5) **(6 Marks)**

Following are the general-purpose methods that can give huge gains in speed by enforcing minimum number of backtracks.

While assigning values are choosing variable for evaluation following questions are inevitable. By answering these questions we can design an efficient strategy in order to improve efficiency of backtracking process.

1. Which variable should be assigned next?

   The answer to this question leads to two simple strategies to select the next variable for assignment, called "Most Constrained Variable" and "Most Constraining Variable".

2. In what order should its values be tried?

   This question designs a strategy called "Least Constraining Value", which helps us in choosing a value among all possible values from the domain.

3. Can we detect inevitable failure early?

   This is the key question which has a significant impact on the speed of generating the solution. These are the strategies which can foresee the failure if the current path of assignment is followed. Thereby, guide us whether we are on the right track. There are three strategies under this section namely, "Forward checking", "Constraint Propagation", and "Arc Consistency".

2.9.6 Backtracking and Look ahead Strategies

- In general these are called as Look ahead and backtracking strategies. If we have approximate inference, using these techniques one can foresee impact of next move and hence can select next value to be assigned or next variable to be chosen for assignment.
- Using backtracking and look ahead strategies we can efficiently compute the remaining legal values given current assignment.
- While using these techniques we have to assign values and need to propagate constraints and it may incur extra cost for assigning values, but usually there are very less chances of worst case performance.
- It can cause a good trade-off between cost and performance.

Following is a generalised look ahead algorithm.

Algorithm

```
Require : A constraint network R
Ensure : A solution or notification that the network is
         inconsistent
i ← 1
DO
D'i ← Di
while 1 <= i <= n do
xi ← SelectValueX
if xi is null then
i ← i - 1
Reset D'K for each k > i to its value before i was last
instantiated
else
i ← i + 1
end if
end while
if i is 0 then
return inconsistent
else
return instantiated valuse for {x1; _ _ _ ; xn}
end if
```

Let's study all these strategies one by one.

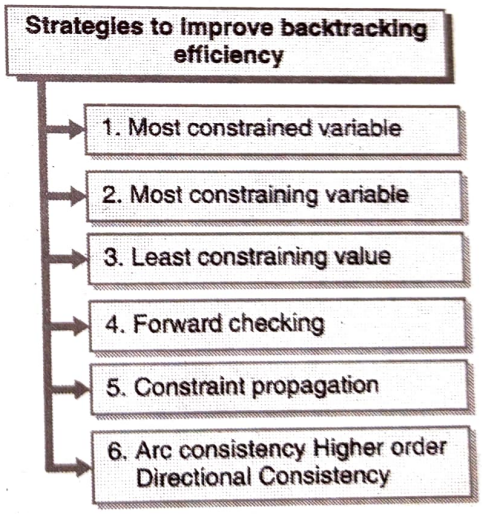
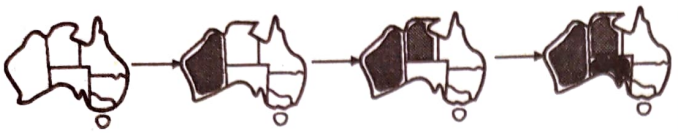


Fig. 2.9.1 : Strategies to improve backtracking efficiency

➜ **1. Most constrained variable**

- It choose the variable with the fewest legal values. By assigning this variable first, we can get a fair idea of other variable assignment.
- Also most of the constraints get satisfied by the assignment, hence chances of getting on wrong track are lowered down.

- This is also called as minimum remaining values (MRV) heuristic or Forced first heuristic.

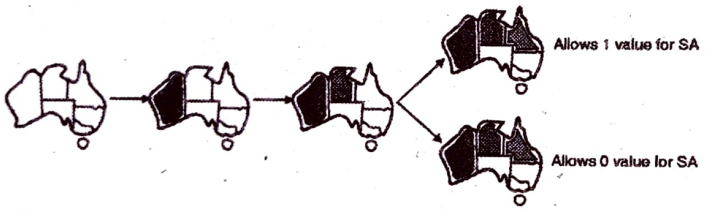


➜ 2. Most constraining variable

- This is the tie-breaker among most constrained variables. In Most constraining variable strategy we first choose the variable with the most constraints on remaining variables.
- Example, the region surrounded by maximum number of regions. Once we assign color to his part of the graph, it as good as we have decided the colors for all the surrounded parts.

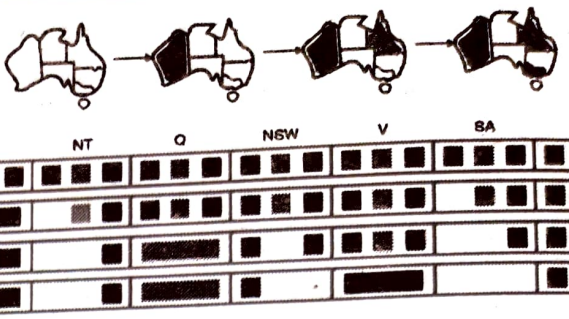
➜ 3. Least constraining value

- Given a variable, choose the least constraining value that is the one that rules out the fewest values for the remaining variables.



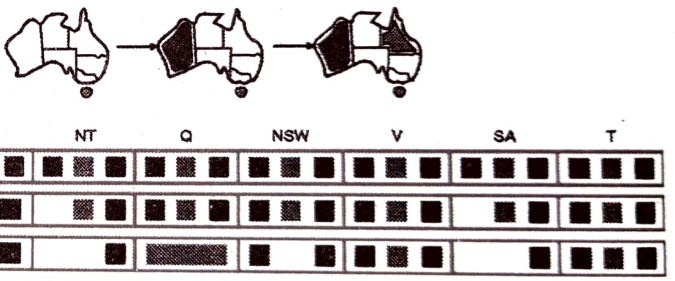
➜ 4. Forward checking

- It keeps track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.
- As shown in the graph, as we go on assigning colors to different regions, we notice that SA is not left with any valid color to assign. Hence the assignment will terminate and the process backtracks to previous state.
- It is the most limited form of constraint propagation. Forward checking propagates the effect of a selected value to future variables separately. In this process, if domains of one of future variables becomes empty, it will try next value for current variable.



➜ 5. Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures.
- In constraint propagation strategy, as we assign color to one of the part of the graph, the other parts are evaluated for the valid assignments. Hence we can detect the conflicts early.
- As shown the following example, as the region Q is getting assigned Green color, both NT and SA are left with blue color; but they both can't have same color as they are adjacent regions. Hence the wrong assignment of green to Q region is detected a step early.



NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally.

➜ 6. Arc consistency / Higher order Directional Consistency

- In this technique, arcs are drawn from one variable to other if they have values satisfying the constraints. The aim is to make each arc consistent while propagating the constraints.
- The arc P → Q is consistent if and only if for every value p of 'P' there is some allowed value q. If 'P' loses a value, neighbors of 'P' need to be verified for permitted values and all arcs are again checked for consistency.
- Hence failures can be detected even early as compared to forward checking. We check arc consistency after every assignment.

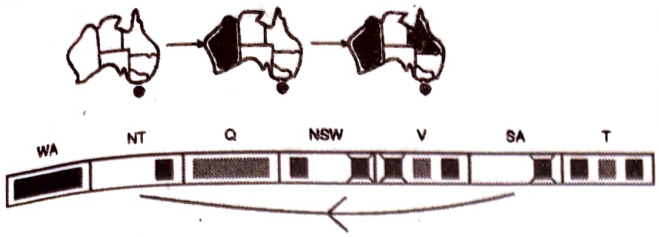


Fig. 2.9.2 : Arc Consistency

- In Fig. 2.9.2 as WA and Q are assigned red and green colors respectively, only blue is left for NT and if it is assigned then there is no consistent value left for SA.

Ex. 2.9.1 : Show the solution for map coloring using constraint satisfaction for following states of India : Maharashtra, Madhya Pradesh, Andhra Pradesh, Karnataka, Goa and Gujarat. What is the optimal number of colors required for coloring?

Soln. :

No. of regions = 6 (Guj, MP, MH, AP, KAR, Goa)

Constraints = Guj≠ MP ≠ MAH

MAH ≠ GOA ≠ KAR

KAR ≠ AP ≠ MAH

GUJ	MP	MAH	GOA
KAR	AP		
Red	Red	Red	Red
Red	Red	Blue	Blue
Blue	Green	Green	Yellow

Hence, the optimal number of colours required to colour this map is '4'.

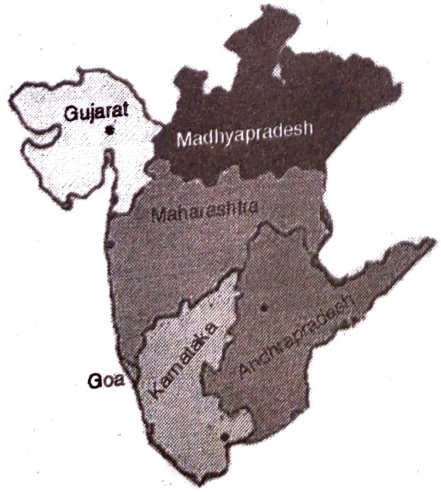


Fig. P. 2.9.1

Syllabus Topic : Scene Labeling

2.10 Scene Labeling Problem

- This is a subproblem of the vision problem, which seeks to deduce facts about the physical world from visual representations of it. In the scene labeling problem we are given a 2-D line drawing of a 3-D scene and must label the components of the drawing in such a way as to make explicit the 3-D structure the drawing represents.
- This problem has been studied extensively since the early 1970s; we will consider a simplified version which was studied by David Waltz as part of his Ph.D. thesis at M.I.T. during this period. The assumptions we make to achieve this simplification are :

1. Every vertex is trihedral (formed by the junction of three planar surfaces; implicitly, we assume that all surfaces are planar).
2. Surfaces do not contain 0-width cracks (lines formed by the junction of coplanar surfaces).
3. No shadows (lines which represent projections of edges onto surfaces along the direction of light sources) are allowed.
4. The scene is viewed from a general viewing position (roughly, for some sufficiently small $Q > 0$ we can move our viewing point in an arbitrary direction by a distance of Q without any topologically significant change in the resulting 2-D representation).

- It actually turns out, as was revealed by later work on the problem, that assumption 3 is not really a simplification, since the information which can be deduced from the presence and positions of shadows in some sense outweighs the potential for confusion with actual edges they bring with them, but we will follow Waltz' path here and let the assumption stand.

- Although a detailed analysis of the problem is beyond the scope of this lecture, one of the fundamental results of this analysis is that a relatively simple taxonomy of 4 classes of vertices and 4 classes of lines suffices to cover all physical possibilities, modulo the assumptions we have made.
- While we will not give formal definitions of the classes here, they may be intuitively described as follows :

Four Classes of Vertices

1. **L-Junction Vertices :** Vertices at which only 2 visible lines meet.
2. **T-Junction Vertices :** Vertices at which one line segment ends in the middle of another.
3. **Fork Vertices :** Vertices at which three lines meet, with no angles greater than or equal to 180°between any pair of them.
4. **Arrow Vertices :** Vertices at which three lines meet, with an angle greater than 180°between one pair of them.

Four Classes of Lines

1. **Convex Lines :** Lines formed by the junction of two visible surfaces for which a line segment connecting the two surfaces would be inside the solids they bound.
2. **Concave Lines :** Lines formed by the junction of two visible surfaces for which a line segment connecting the two surfaces would be outside the solids they bound.
3. **Right/Up Arrow Lines :** Lines formed by the junction of one visible and one obscured surface, with the visible surface lying below/to the right of the line.
4. **Left/Down Arrow Lines :** Lines formed by the junction of one visible and one obscured surface, with the visible surface lying above/to the left of the line.

- Depending on the viewing location, these 4 classes of vertices and 4 classes of lines can be combined to give the eighteen types of vertices shown in the Fig. 2.10.1.

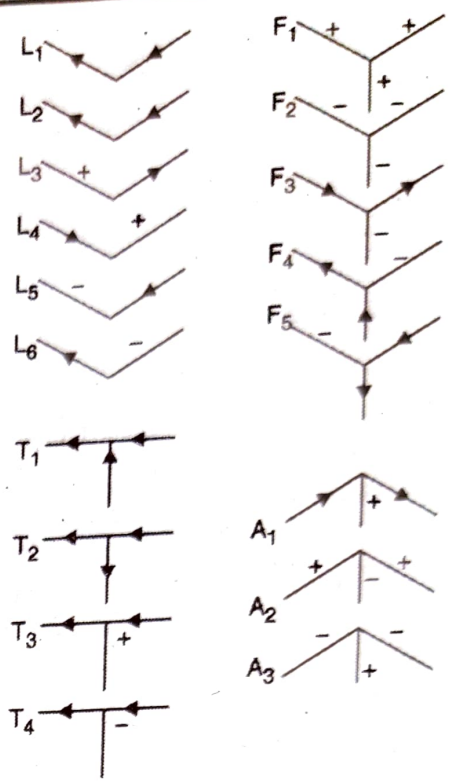


Fig. 2.10.1

Representative examples of each of these classes may be seen in the following drawing :

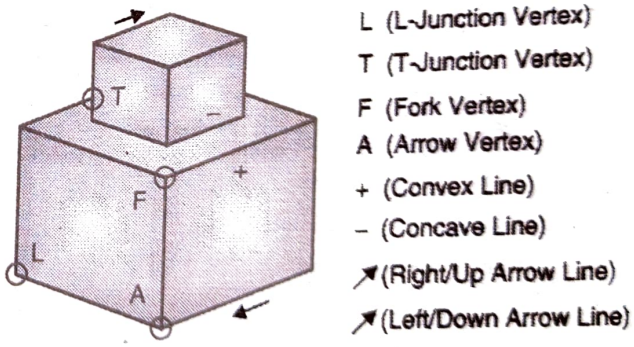


Fig. 2.10.2

- As we shall see, the classification of vertices and lines in this way allows us to pose the scene labeling problem as a CSP, with vertices as variables to be labeled with values from the list of classes above; since each class of vertex can only have certain classes of lines meet to form it, the restrictions imposed by the classification of the two vertices at the endpoints of each line must be compatible, which then provides us with the constraints.

The general algorithm for scene-labeling is as follows :

1. Identify the boundary edges (optional but desirable).

2. Number the vertices (optionally, start with a boundary vertex) in some order.
3. Visit a vertex v (in the order of its numbering) and attempt to label it :
   (a) Attach to v all vertex labels compatible with its type.
   (b) Eliminate any labels that are not consistent with local constraints.
   (c) For each vertex A that was visited in (b),
      i. Eliminate every label of A that has no consistent label in v
      ii. If any label of A was eliminated, continue propagating the constraints until no more labels are eliminated.

There are three possible outcomes of this algorithm :

1. Every line has only one label left,
2. One or more lines have multiple labels left, or
3. One or more lines have no labels left.

- If we are left with multiple labels on some lines, the scene can be interpreted in more than one way.
- An exhaustive search through the remaining possible combinations will identify the valid interpretations. This takes considerably less effort than an exhaustive search the entire search space. On the other hand, if some lines do not have any labels left, the scene violates one or more of the assumptions made earlier.
- For instance, **vertex J** in the Fig. 2.10.3 violates the assumption that all vertices are trihedral. The algorithm will end with the lines at **vertex J** having no labels.

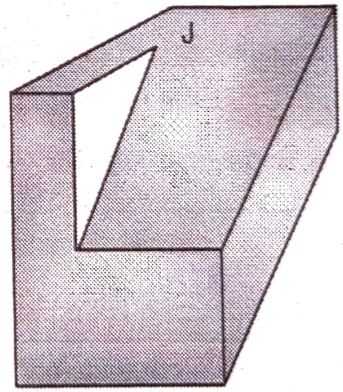


Fig. 2.10.3

❑❑❑