UNIT - II

Problem Solving by Search-II and Propositional Logic
Adversarial Search: Games, Optimal Decisions in Games, Alpha–Beta Pruning, Imperfect Real-Time Decisions.
Constraint Satisfaction Problems: Defining Constraint Satisfaction Problems, Constraint Propagation, Backtracking Search for CSPs, Local Search for CSPs, The Structure of Problems.
Propositional Logic: Knowledge-Based Agents, The Wumpus World, Logic, Propositional Logic, Propositional Theorem Proving: Inference and proofs, Proof by resolution, Horn clauses and definite clauses, Forward and backward chaining, Effective Propositional Model Checking, Agents Based on Propositional Logic.

Adversarial Search in Artificial Intelligence

AI Adversarial search: Adversarial search is a game-playing technique where the agents are surrounded by a competitive environment. A conflicting goal is given to the agents (multiagent). These agents compete with one another and try to defeat one another in order to win the game. Such conflicting goals give rise to the adversarial search. Here, game-playing means discussing those games where human intelligence and logic factor is used, excluding other factors such as luck factor. Tic-tac-toe, chess, checkers, etc., are such type of games where no luck factor works, only mind works.
Mathematically, this search is based on the concept of 'Game Theory.' *According to game theory, a game is played between two players. To complete the game, one has to win the game and the other looses automatically.'*

We are opponents- I win, you loose.

Techniques required to get the best optimal solution
There is always a need to choose those algorithms which provide the best optimal solution in a limited time. So, we use the following techniques which could fulfill our requirements:

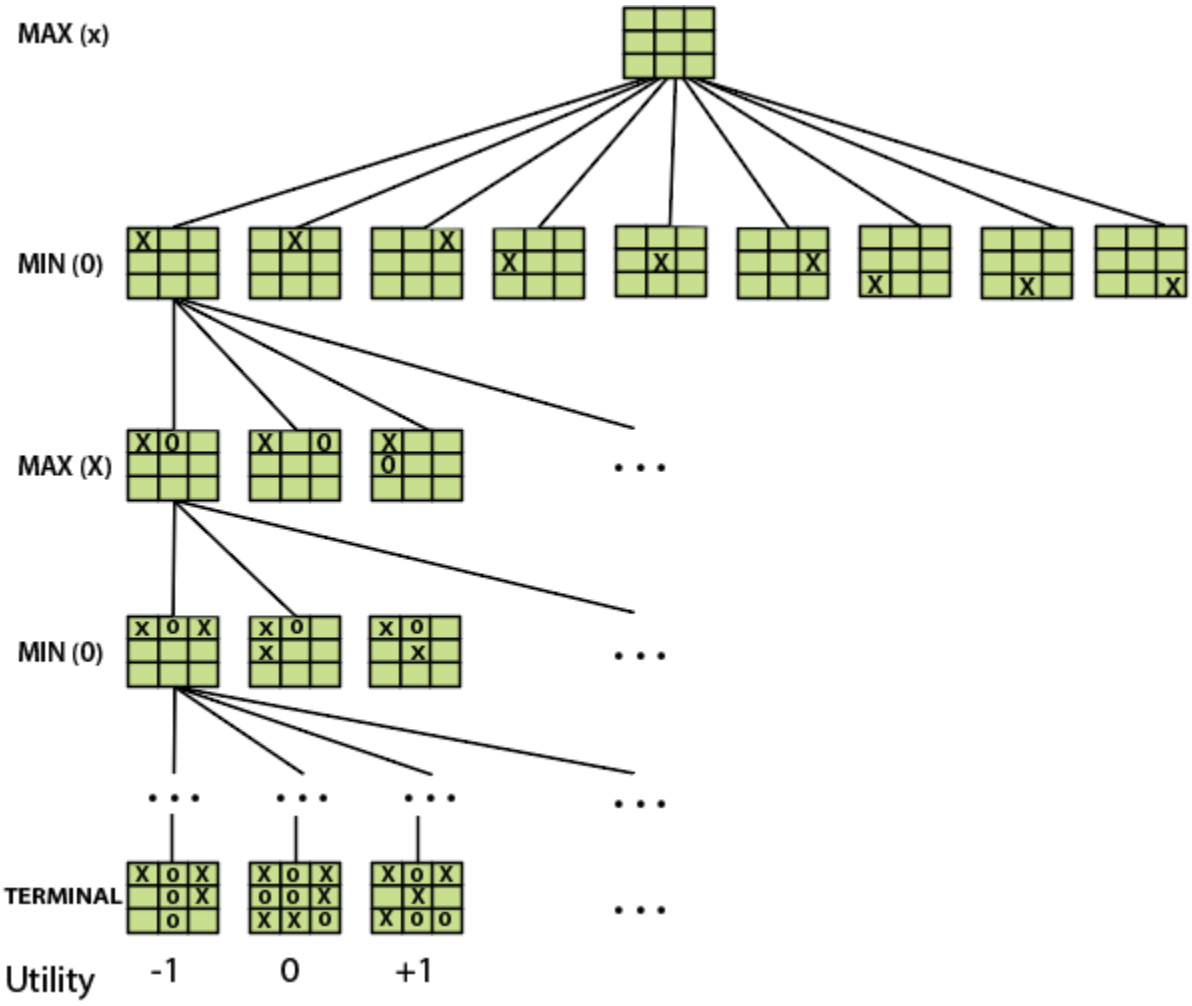
- Pruning: A technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.
- Heuristic Evaluation Function: It allows to approximate the cost value at each level of the search tree, before reaching the goal node.

Elements of Game Playing search

To play a game, we use a game tree to know all the possible choices and to pick the best one out. There are following elements of a game-playing:

- S0: It is the initial state from where a game begins.
- PLAYER (s): It defines which player is having the current turn to make a move in the state.
- ACTIONS (s): It defines the set of legal moves to be used in a state.
- RESULT (s, a): It is a transition model which defines the result of a move.
- TERMINAL-TEST (s): It defines that the game has ended and returns true.
- UTILITY (s,p): It defines the final value with which the game has ended. This function is also known as Objective function or Payoff function. The price which the winner will get i.e.
- (-1): If the PLAYER loses.
- (+1): If the PLAYER wins.
- (0): If there is a draw between the PLAYERS.

For example, in chess, tic-tac-toe, we have two or three possible outcomes. Either to win, to lose, or to draw the match with values +1,-1 or 0.
Let's understand the working of the elements with the help of a game tree designed for tic-tac-toe. Here, the *node represents the game state and edges represent the moves taken by the players.*



A game-tree for tic-tac-toe

- INITIAL STATE (S0): The top node in the game-tree represents the initial state in the tree and shows all the possible choice to pick out one.
- PLAYER (s): There are two players, MAX and MIN. MAX begins the game by picking one best move and place X in the empty square box.
- ACTIONS (s): Both the players can make moves in the empty boxes chance by chance.
- RESULT (s, a): The moves made by MIN and MAX will decide the outcome of the game.
- TERMINAL-TEST(s): When all the empty boxes will be filled, it will be the terminating state of the game.
- UTILITY: At the end, we will get to know who wins: MAX or MIN, and accordingly, the price will be given to them.

Types of algorithms in Adversarial search

In a normal search, we follow a sequence of actions to reach the goal or to finish the game optimally. But in an adversarial search, the result depends on the players which will decide the result of the game. It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.
There are following types of adversarial search:

- Minmax Algorithm
- Alpha-beta Pruning.

Minimax Strategy

In artificial intelligence, minimax is a decision-making strategy under game theory, which is used to minimize the losing chances in a game and to maximize the winning chances. This strategy is also known as 'Minmax,' 'MM,' or 'Saddle point.' Basically, it is a two-player game strategy where *if one wins, the other loose the game*. This strategy simulates those games that we play in our day-to-day life. Like, if two persons are playing chess, the result will be in favor of one player and will unfavor the other one. The person who will make his bes*t try,efforts as well as cleverness, will surely win.*
We can easily understand this strategy via game tree- where the *nodes represent the states of the game and edges represent the moves made by the players in the game*. Players will be two namely:

- MIN: Decrease the chances of MAX to win the game.
- MAX: Increases his chances of winning the game.

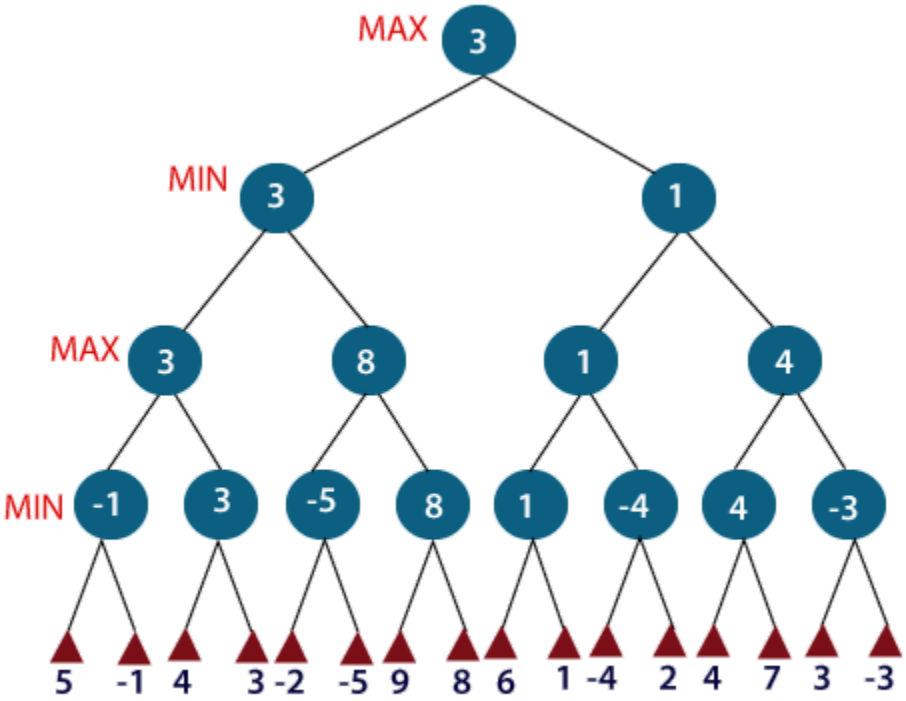
They both play the game alternatively, i.e., turn by turn and following the above strategy, i.e., if one wins, the other will definitely lose it. Both players look at one another as competitors and will try to defeat one-another, giving their best.
In minimax strategy, the result of the game or the utility value is generated by a heuristic function by propagating from the initial node to the root node. It follows the backtracking technique and backtracks to find

the best choice. MAX will choose that path which will increase its utility value and MIN will choose the opposite path which could help it to minimize MAX's utility value.

MINIMAX Algorithm

MINIMAX algorithm is a backtracking algorithm where it backtracks to pick the best move out of several choices. MINIMAX strategy follows the DFS (Depth-first search) concept. Here, we have two players MIN and MAX, and the game is played alternatively between them, i.e., when MAX made a move, then the next turn is of MIN. It means the move made by MAX is fixed and, he cannot change it. The same concept is followed in DFS strategy, i.e., we follow the same path and cannot change in the middle. That's why in MINIMAX algorithm, instead of BFS, we follow DFS.

- Keep on generating the game tree/ search tree till a limit d.
- Compute the move using a heuristic function.
- Propagate the values from the leaf node till the current position following the minimax strategy.
- Make the best move from the choices.



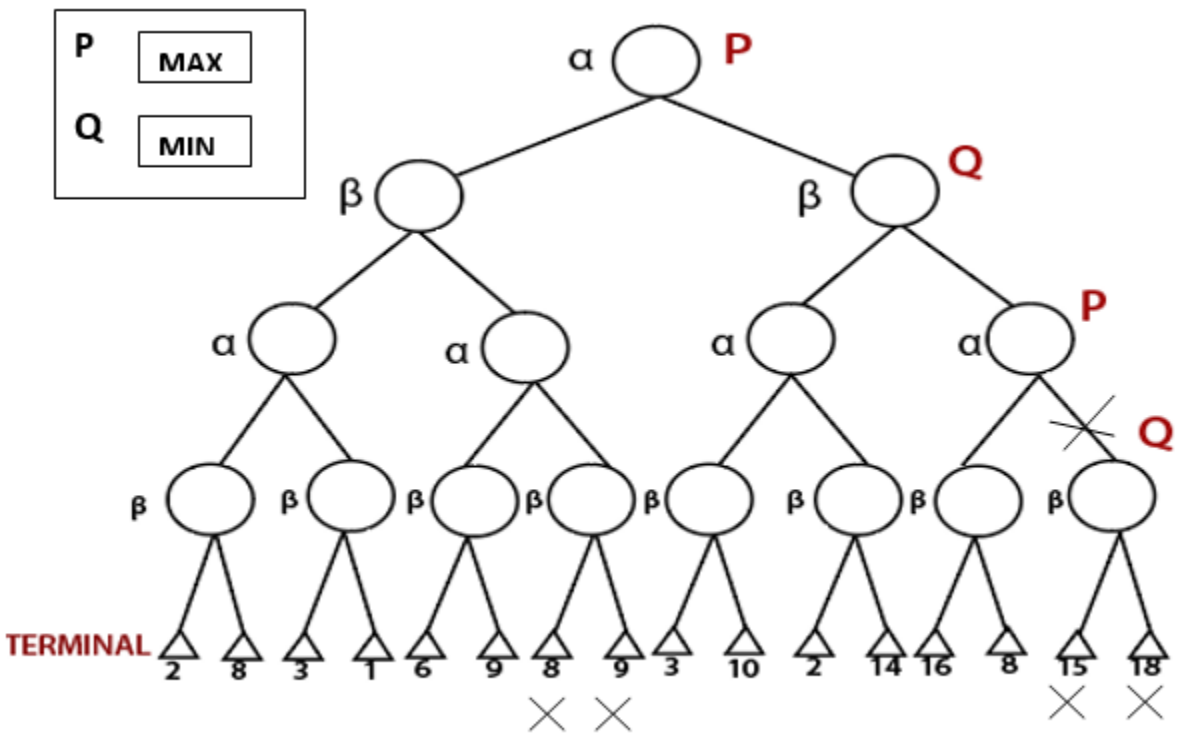
For example, in the above figure, the two players MAX and MIN are there. MAX starts the game by choosing one path and propagating all the nodes of that path. Now, MAX will backtrack to the initial node and choose the best path where his utility value will be the maximum. After this, its MIN chance. MIN will also propagate through a path and again will backtrack, but MIN will choose the path which could minimize MAX winning chances or the utility value.
So, if the level is minimizing, the node will accept the minimum value from the successor nodes. If the level is maximizing, the node will accept the maximum value from the successor.
Note: The time complexity of MINIMAX algorithm is O(bd) where b is the branching factor and d is the depth of the search tree.

Alpha-beta Pruning | Artificial Intelligence

Alpha-beta pruning is an advance version of MINIMAX algorithm. The drawback of minimax strategy is that it explores each node in the tree deeply to provide the best path among all the paths. This increases its time complexity. But as we know, the performance measure is the first consideration for any optimal algorithm. Therefore, alpha-beta pruning reduces this drawback of minimax strategy by less exploring the nodes of the search tree.
The method used in alpha-beta pruning is that it cutoff the search by exploring less number of nodes. It makes the same moves as a minimax algorithm does, but it prunes the unwanted branches using the pruning technique (discussed in adversarial search). Alpha-beta pruning works on two threshold values, i.e., ? (alpha) and ? (beta).

- ?: It is the best highest value, a MAX player can have. It is the lower bound, which represents negative infinity value.
- ?: It is the best lowest value, a MIN player can have. It is the upper bound which represents positive infinity.

So, each MAX node has ?-value, which never decreases, and each MIN node has ?-value, which never increases.
Note: Alpha-beta pruning technique can be applied to trees of any depth, and it is possible to prune the entire subtrees easily.
Working of Alpha-beta Pruning
Consider the below example of a game tree where P and Q are two players. The game will be played alternatively, i.e., chance by chance. Let, P be the player who will try to win the game by maximizing its winning chances. Q is the player who will try to minimize P's winning chances. Here, ? will represent the maximum value of the nodes, which will be the value for P as well. ? will represent the minimum value of the nodes, which will be the value of Q.



Alpha-beta pruning

- Any one player will start the game. Following the DFS order, the player will choose one path and will reach to its depth, i.e., where he will find the TERMINAL value.
- If the game is started by player P, he will choose the maximum value in order to increase its winning chances with maximum utility value.
- If the game is started by player Q, he will choose the minimum value in order to decrease the winning chances of A with the best possible minimum utility value.
- Both will play the game alternatively.
- The game will be started from the last level of the game tree, and the value will be chosen accordingly.
- Like in the below figure, the game is started by player Q. He will pick the leftmost value of the TERMINAL and fix it for beta (?). Now, the next TERMINAL value will be compared with the ?-value. If the value will be smaller than or equal to the ?-value, replace it with the current ?-value otherwise no need to replace the value.
- After completing one part, move the achieved ?-value to its upper node and fix it for the other threshold value, i.e., ?.
- Now, its P turn, he will pick the best maximum value. P will move to explore the next part only after comparing the values with the current ?-value. If the value is equal or greater than the current ?-value, then only it will be replaced otherwise we will prune the values.
- The steps will be repeated unless the result is not obtained.

- So, number of pruned nodes in the above example are four and MAX wins the game with the maximum UTILITY value, i.e.,3

The rule which will be followed is: "Explore nodes if necessary otherwise prune the unnecessary nodes."
Note: It is obvious that the result will have the same UTILITY value that we may get from the MINIMAX strategy.

Imperfect real-time decisions

Because moves must be made in a reasonable amount of time, usually it is not feasible to consider the whole game tree (even with alpha-beta), so programs should cut the search off at some point earlier and apply a heuristic evaluation function to states in the search, effectively turning nonterminal nodes into terminal leaves.
i.e. Alter minimax or alpha-beta in 2 ways:
1) replace the utility function by a heuristic evaluation function E_{VAL}, which estimates the position's utility.
2) replace the terminal test by a cutoff test that decides when to apply E_{VAL}.

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

Evaluation function
An evaluation function returns an estimate of the expected utility of the game from a given position.
How do we design good evaluation functions?
1) The evaluation function should order the terminal states in the same way as the true utility function.
2) The computation must not take too long.
3) For nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

Features of the state: Most evaluation functions work by calculating various features of the state, e.g. in chess, number of white pawns, black pawns, white queens, black queens, etc.
Categories: The features, taken together, define various categories (a.k.a. equivalence classes) of states, the states in each category have the same values for all the features.
Any given category will contain some states that lead to win, draws or losses, the evaluation function can return a single value that reflects the proportion of states with each outcome.
Two ways to design a evaluation function:
a. Expected value (requires too many categories and hence too much experience to estimate) e.g.
E.g. 72% of the states encountered in the two-pawns vs. one-pawn category lead to a win(utility e.g.!); 20% to a loss(0) and 8% to a draw(1/2). Then a reasonable evaluation for states in the category is the expected value: (0.72*1)+(0.20*0)+(0.08*1/2)=0/76.
As with terminal states, the evaluation function need not return actual expected values as long as the ordering of the states is the same. b. weighted linear function (most evaluation functions use that.) We can compute separate numerical contributions from each feature and then combine them to find the total value.

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

Each w_i is a weight and each f_i is a feature of the position. For chess, the f_i could be the numbers of each kind of piece on the board (i.e. feature), and w_i could be the values of the pieces (1 for pawn, 3 for bishop, etc.).
Adding up the values of features in fact involves a strong assumption (that the contribution of each feature is independent of the values of the other features), thus current programs for games also use nonlinear combinations of features.

Cutting off search

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

To modify ALPHA-BETA-SEARCH:

1) Replace the two lines that mention TERMINAL-TEST with
If CUTOFF-TEST(state, depth) then return EVAL(state)
2) Arrange for some bookkeeping so that the current depth is incremented on each recursive call.
The most straightforward approach: set a fixed depth limit so that CUTOFF-TEST(state, depth) returns true for all depth greater than some fixed depth d.
A more robust approach: apply iterative deepening.

quiescence search: The evaluation function should be applied only to positions that are quiescent (i.e. unlikely to exhibit wild swing in value in the near future). Nonquiescent positions can be expanded further until quiescent positions are reached, this extra search is called a quiescence search.
Horizon effect: arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by delaying tactics.
Singular extension: One strategy to mitigate the horizon effect, a move that is “clearly better” than all other moves in a given position. Once discovered anywhere in the tree in the course of a search, the singular move is remembered. When the search reaches the normal depth limit, the algorithm checks to see if the singular extension is a legal move; if it is, the algorithm allows the move to be considered.

Forward pruning
Forward pruning: Some moves at a given node are pruned immediately without further consideration.
PROBCUT (probabilistic) algorithm: A forward-pruning version of alpha-beta search that uses statistics gained from prior experience to lessen the chance that the best move will be pruned.
Alpha-beta search prunes any node that is probably outside the current(α, β) window, PROBCUT also prunes nodes that are probably outside the window. It computes this probability by doing a shallow search to compute

the backed-up value v of a node and then using past experience to estimate how likely it is that a score of v at depth d in the tree would be outside (α, β).

Search versus lookup
Many game programs precompute tables of best moves in the opening and endgame so that they can look up a move rather than search.
For the opening (and early moves), the program use table lookup, relying on the expertise of human and statistic from a database of past games;
After about ten moves, end up in a rarely seen position, the program switch from table lookup to search;
Near the end of the game there are again fewer possible positions, and more chances to do lookup.
A computer can completely solve the endgame by producing a policy, which is a mapping from every possible state to the best move in that state. Then we can just look up the best move rather than recompute it anew.

Constraint Satisfaction Problems in Artificial Intelligence

We have seen so many techniques like Local search, Adversarial search to solve different problems. The objective of every problem-solving technique is one, i.e., to find a solution to reach the goal. Although, in adversarial search and local search, there were no constraints on the agents while solving the problems and reaching to its solutions.
In this section, we will discuss another type of problem-solving technique known as Constraint satisfaction technique. By the name, it is understood that constraint satisfaction means *solving a problem under certain constraints or rules.*
Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. This type of technique leads to a deeper understanding of the problem structure as well as its complexity.
Constraint satisfaction depends on three components, namely:

- **X:** It is a set of variables.
- **D:** It is a set of domains where the variables reside. There is a specific domain for each variable.
- **C:** It is a set of constraints which are followed by a set of variables.

In constraint satisfaction, domains are the spaces where the variables reside, following the problem specific constraints. These are the three main elements of a constraint satisfaction technique. The constraint value consists of a pair of **{scope, rel}**. The **scope** is a tuple of variables which participate in the constraint and **rel** is a relation which includes a list of values which the variables can take to satisfy the constraints of the problem.
Solving Constraint Satisfaction Problems
The requirements to solve a constraint satisfaction problem (CSP) is:

- A state-space
- The notion of the solution.

A state in state-space is defined by assigning values to some or all variables such as
{X1=v1, X2=v2, and so on…}.
An assignment of values to a variable can be done in three ways:

- **Consistent or Legal Assignment:** An assignment which does not violate any constraint or rule is called Consistent or legal assignment.
- **Complete Assignment:** An assignment where every variable is assigned with a value, and the solution to the CSP remains consistent. Such assignment is known as Complete assignment.
- **Partial Assignment:** An assignment which assigns values to some of the variables only. Such types of assignments are called Partial assignments.

Types of Domains in CSP
There are following two types of domains which are used by the variables :

- **Discrete Domain:** It is an infinite domain which can have one state for multiple variables. **For example,** a start state can be allocated infinite times for each variable.
- **Finite Domain:** It is a finite domain which can have continuous states describing one domain for one specific variable. It is also called a continuous domain.

Constraint Types in CSP
With respect to the variables, basically there are following types of constraints:

- **Unary Constraints:** It is the simplest type of constraints that restricts the value of a single variable.
- **Binary Constraints:** It is the constraint type which relates two variables. A value **x2** will contain a value which lies between **x1** and **x3**.
- **Global Constraints:** It is the constraint type which involves an arbitrary number of variables.

Some special types of solution algorithms are used to solve the following types of constraints:

- **Linear Constraints:** These types of constraints are commonly used in linear programming where each variable containing an integer value exists in linear form only.
- **Non-linear Constraints:** These types of constraints are used in nonlinear programming where each variable (an integer value) exists in a non-linear form.

Note: A special constraint which works in the real-world is known as **Preference constraint.**
Constraint Propagation
In local state-spaces, the choice is only one, i.e., to search for a solution. But in CSP, we have two choices either:

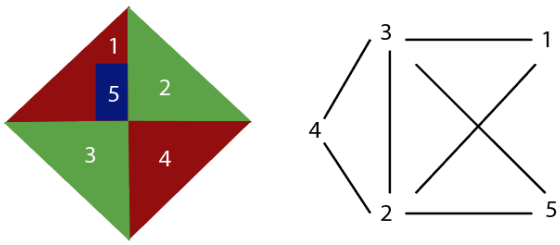
- We can search for a solution or
- We can perform a special type of inference called **constraint propagation**.

Constraint propagation is a special type of inference which helps in reducing the legal number of values for the variables. The idea behind constraint propagation is **local consistency.**
In local consistency, variables are treated as **nodes**, and each binary constraint is treated as an **arc** in the given problem. **There are following local consistencies which are discussed below:**

- **Node Consistency:** A single variable is said to be node consistent if all the values in the variable's domain satisfy the unary constraints on the variables.
- **Arc Consistency:** A variable is arc consistent if every value in its domain satisfies the binary constraints of the variables.
- **Path Consistency:** When the evaluation of a set of two variables with respect to a third variable can be extended over another variable, satisfying all the binary constraints. It is similar to arc consistency.
- **k-consistency:** This type of consistency is used to define the notion of stronger forms of propagation. Here, we examine the k-consistency of the variables.

CSP Problems
Constraint satisfaction includes those problems which contain some constraints while solving the problem. CSP includes the following problems:

- **Graph Coloring:** The problem where the constraint is that no adjacent sides can have the same color.



Graph Coloring

- **Sudoku Playing:** The gameplay where the constraint is that no number from 0-9 can be repeated in the same row or column.

SUDOKU

4							5	9
2	6		5				3	
				9	2			
		2		6			1	
		3	8	1	9	7		
	7			3		5		
			3	4				
	3				6		2	7
5	9							6

Puzzle

4	1	7	6	8	3	2	5	9
2	6	9	5	7	1	8	3	4
3	8	5	4	9	2	6	7	1
8	4	2	7	6	5	9	1	3
6	5	3	8	1	9	7	4	2
9	7	1	2	3	4	5	6	8
7	2	6	3	4	8	1	9	5
1	3	8	9	5	6	4	2	7
5	9	4	1	2	7	3	8	6

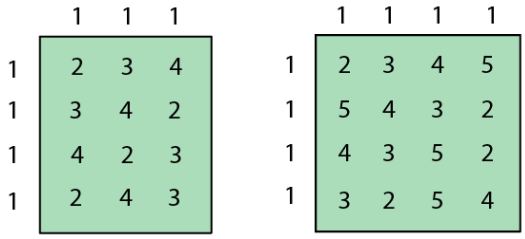
Solution

- **n-queen problem:** In n-queen problem, the constraint is that no queen should be placed either diagonally, in the same row or column.

Note: The n-queen problem is already discussed in Problem-solving in the AI section.

- **Crossword:** In crossword problems, the constraint is that there should be the correct formation of the words, and it should be meaningful.

- **Latin square Problem:** In this game, the task is to search the pattern which is occurring several times in the game. They may be shuffled but will contain the same digits.



Latin Squence Problem

- **Cryptarithmetic Problem:** This problem has one most important constraint that is, we cannot assign a different digit to the same character. All digits should contain a unique alphabet.

Backtracking search for CSPs

Backtracking search, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search. Commutativity: CSPs are all commutative. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. Backtracking search: A depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.

There is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test. BACKTRACKING-SEARCH keeps only a single representation of a state and alters that representation rather than creating a new one.

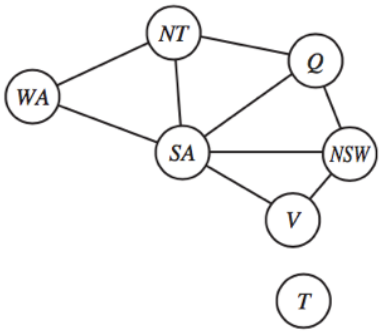
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

To solve CSPs efficiently without domain-specific knowledge, address following questions:
1)function SELECT-UNASSIGNED-VARIABLE: which variable should be assigned next?
function ORDER-DOMAIN-VALUES: in what order should its values be tried?
2)function INFERENCE: what inferences should be performed at each step in the search?
3)When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

1. Variable and value ordering



SELECT-UNASSIGNED-VARIABLE
Variable selection—fail-first
Minimum-remaining-values (MRV) heuristic: The idea of choosing the variable with the fewest "legal" value. A.k.a. "most constrained variable" or "fail-first" heuristic, it picks a variable that is most likely to cause a failure soon thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.
E.g. After the assignment for WA=red and NT=green, there is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q.
[Powerful guide]
Degree heuristic: The degree heuristic attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. [useful tie-breaker]
e.g. SA is the variable with highest degree 5; the other variables have degree 2 or 3; T has degree 0.

ORDER-DOMAIN-VALUES
Value selection—fail-last
If we are trying to find all the solutions to a problem (not just the first one), then the ordering does not matter.
Least-constraining-value heuristic: prefers the value that rules out the fewest choice for the neighboring variables in the constraint graph. (Try to leave the maximum flexibility for subsequent variable assignments.)
e.g. We have generated the partial assignment with WA=red and NT=green and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA, therefore prefers red to blue.

The minimum-remaining-values and degree heuristic are domain-independent methods for deciding which variable to choose next in a backtracking search. The least-constraining-value heuristic helps in deciding which value to try first for a given variable.

2. Interleaving search and inference

INFERENCE forward checking: [One of the simplest forms of inference.] Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.
There is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	(R)	B	(G)	R B	R G B	B	R G B
After *V=blue*	(R)	B	(G)	R	(B)		R G B

Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$ is assigned, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$ is assigned, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

Advantage: For many problems the search will be more effective if we combine the MRV heuristic with forward checking.
Disadvantage: Forward checking only makes the current variable arc-consistent, but doesn't look ahead and make all the other variables arc-consistent.
MAC (Maintaining Arc Consistency) algorithm: [More powerful than forward checking, detect this inconsistency.] After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs(X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately.

3. Intelligent backtracking

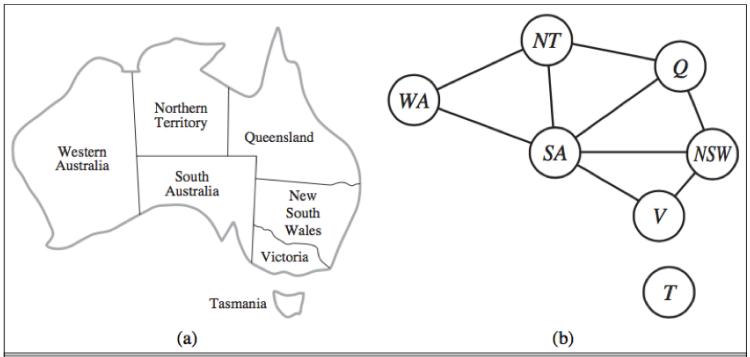


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

chronological backtracking: The BACKTRACKING-SEARCH in Fig 6.5. When a branch of the search fails, back up to the preceding variable and try a different value for it. (The most recent decision point is revisited.)
E.g. Suppose we have generated the partial assignment {Q=red, NSW=green, V=blue, T=red}.
When we try the next variable SA, we see every value violates a constraint. We back up to T and try a new color, it cannot resolve the problem. Intelligent backtracking: Backtrack to a variable that was responsible for making one of the possible values of the next variable (e.g. SA) impossible. Conflict set for a variable: A set of assignments that are in conflict with some value for that variable. (e.g. The set {Q=red, NSW=green, V=blue} is the conflict set for SA.) backjumping method: Backtracks to the most recent assignment in the conflict set. (e.g. backjumping would jump over T and try a new value for V.)

Forward checking can supply the conflict set with no extra work. Whenever forward checking based on an assignment X=x deletes a value from Y's domain, add X=x to Y's conflict set; If the last value is deleted from Y's domain, the assignment in the conflict set of Y is added to the conflict set of X. In fact,every branch pruned by backjumping is also pruned by forward checking. Hence simple backjumping is redundant in a forward-checking search or in a search that uses stronger consistency checking (such as MAC).

Conflict-directed backjumping:

E.g. consider the partial assignment which is proved to be inconsistent: {WA=red, NSW=red}.
We try T=red next and then assign NT, Q, V, SA, no assignment can work for these last 4 variables.
Eventually we run out of value to try at NT, but simple backjumping cannot work because NT doesn't have a complete conflict set of preceding variables that caused it to fail.
The set {WA, NSW} is a deeper notion of the conflict set for NT, causing NT together with any subsequent variables to have no consistent solution. So the algorithm should backtrack to NSW and skip over T.
A backjumping algorithm that uses conflict sets defined in this way is called conflict-directed backjumping.
How to Compute:

When a variable's domain becomes empty, the "terminal" failure occurs; that variable has a standard conflict set. Let X_j be the current variable, let conf(X_j) be its conflict set. If every possible value for X_j fails, backjump to the most recent variable X_i in conf(X_j), and set conf(X_i) ← conf(X_i) ∪ conf(X_j) – {X_i}.

The conflict set for a variable means, there is no solution from that variable onward, given the preceding assignment to the conflict set.
E.g. assign WA, NSW, T, NT, Q, V, SA.
SA fails, and its conflict set is {WA, NT, Q}. (standard conflict set)
Backjump to Q, its conflict set is {NT, NSW} ∪ {WA,NT,Q}-{Q} = {WA, NT, NSW}.
Backtrack to NT, its conflict set is {WA} ∪ {WA,NT,NSW}-{NT} = {WA, NSW}.
Hence the algorithm backjump to NSW. (over T)

After backjumping from a contradiction, how to avoid running into the same problem again:
Constraint learning: The idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a no-good. We then record the no-good, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods.

Backtracking occurs when no legal assignment can be found for a variable. Conflict-directed backjumping backtracks directly to the source of the problem.

Local search for CSPs

Local search algorithms for CSPs use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time. The min-conflicts heuristic: In choosing a new value for a variable, select the value that results in the minimum number of conflicts with other variables.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp.VARIABLES
    value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Local search techniques in Section 4.1 can be used in local search for CSPs. The landscape of a CSP under the mini-conflicts heuristic usually has a series of plateaus. Simulated annealing and Plateau search (i.e. allowing sideways moves to another state with the same score) can help local search find its way off the plateau. This wandering on the plateau can be directed with tabu search: keeping a small list of recently visited states and forbidding the algorithm to return to those States.

Constraint weighting: a technique that can help concentrate the search on the important constraints. Each constraint is given a numeric weight W_i, initially all 1. At each step, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints. The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.

Local search can be used in an online setting when the problem changes, this is particularly important in scheduling problems.

The structure of problem

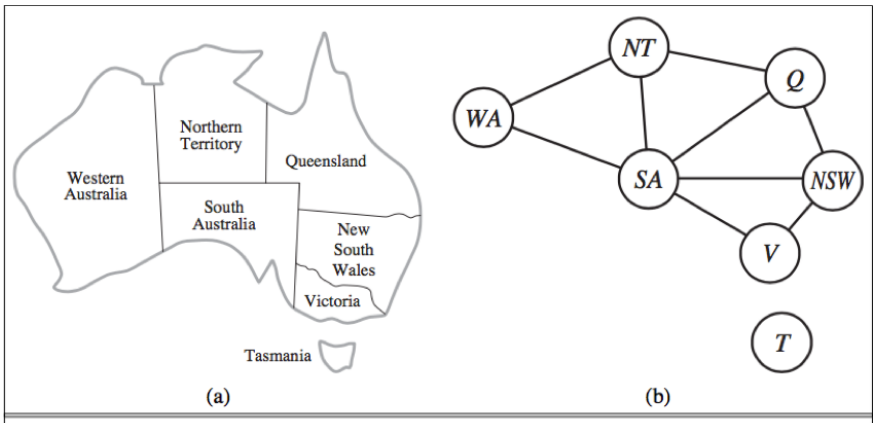


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

1. The structure of constraint graph

The structure of the problem as represented by the constraint graph can be used to find solution quickly. e.g. The problem can be decomposed into 2 independent subproblems: Coloring T and coloring the mainland.
Tree: A constraint graph is a tree when any two varyiable are connected by only one path. Directed arc consistency (DAC): A CSP is defined to be directed arc-consistent under an ordering of variables $X_1, X_2, \dots, X_n$ if and only if every X_i is arc-consistent with each X_j for j>i. By using DAC, any tree-structured CSP can be solved in time linear in the number of variables.

How to solve a tree-structure CSP: Pick any variable to be the root of the tree; Choose an ordering of the variable such that each variable appears after its parent in the tree. (topological sort) Any tree with n nodes has n-1 arcs, so we can make this graph directed arc-consistent in O(n) steps, each of which must compare up to d possible domain values for 2 variables, for a total time of $O(nd^2)$. Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value. Since each link from a parent to its child is arc consistent, we won't have to backtrack, and can move linearly through the variables.

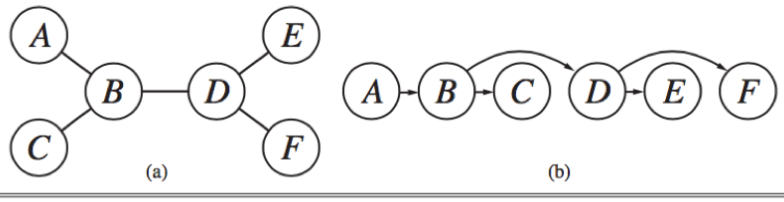


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with *A* as the root. This is known as a **topological sort** of the variables.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C

  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent then return failure
  for i = 1 to n do
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value then return failure
  return assignment
```

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

There are 2 primary ways to reduce more general constraint graphs to trees:

1. Based on removing nodes;

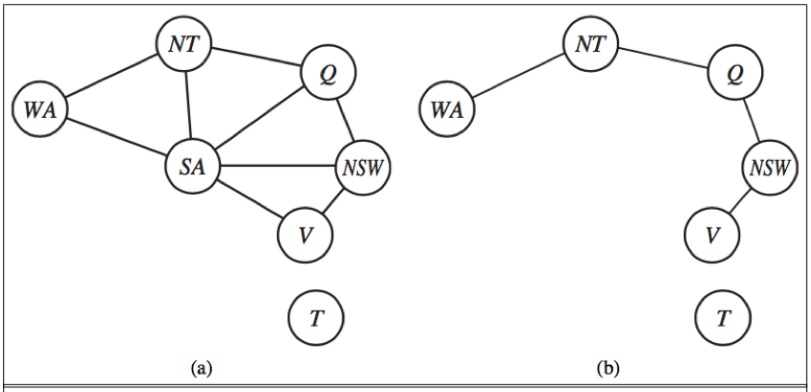


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of *SA*.

e.g. We can delete SA from the graph by fixing a value for SA and deleting from the domains of other variables any values that are inconsistent with the value chosen for SA.

The general algorithm:
Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a cycle cutset.
For each possible assignment to the variables in S that satisfies all constraints on S,
(a) remove from the domain of the remaining variables any values that are inconsistent with the assignment for S, and
(b) If the remaining CSP has a solution, return it together with the assignment for S.
Time complexity: $O(d^c \cdot (n-c)d^2)$, c is the size of the cycle cut set.

Cutset conditioning: The overall algorithmic approach of efficient approximation algorithms to find the smallest cycle cutset.

2. Based on collapsing nodes together

Tree decomposition: construct a tree decomposition of the constraint graph into a set of connected subproblems, each subproblem is solved independently, and the resulting solutions are then combined.

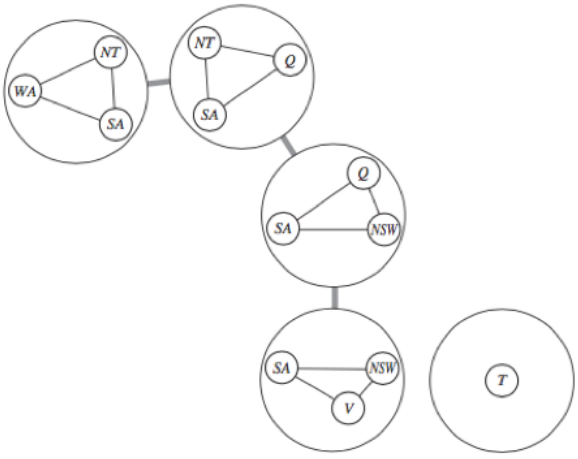


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

A tree decomposition must satisfy 3 requirements:
·Every variable in the original problem appears in at least one of the subproblems.
·If 2 variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
·If a variable appears in 2 subproblems in the tree, it must appear in every subproblem along the path connecting those those subproblems.

We solve each subproblem independently. If any one has no solution, the entire problem has no solution. If we can solve all the subproblems, then construct a global solution as follows: First, view each subproblem as a "mega-variable" whose domain is the set of all solutions for the subproblem. Then, solve the constraints connecting the subproblems using the efficient algorithm for trees.

A given constraint graph admits many tree decomposition; In choosing a decomposition, the aim is to make the subproblems as small as possible.

Tree width:

The tree width of a tree decomposition of a graph is one less than the size of the largest subproblems. The tree width of the graph itself is the minimum tree width among all its tree decompositions. Time complexity: $O(nd^{w+1})$, w is the tree width of the graph.

The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. Cutset conditioning can reduce a general CSP to a tree-structured one and is quite efficient if a small cutset can be found. Tree decomposition techniques transform the CSP into a tree of subproblems and are efficient if the tree width of the constraint graph is small.

The structure in the values of variables

By introducing a symmetry-breaking constraint, we can break the value symmetry and reduce the search space by a factor of n!.
E.g. Consider the map-coloring problems with n colors, for every consistent solution, there is actually a set of n! solutions formed by permuting the color names.(value symmetry)
On the Australia map, WA, NT and SA must all have different colors, so there are 3!=6 ways to assign.
We can impose an arbitrary ordering constraint NT<SA<WA that requires the 3 values to be in alphabetical order. This constraint ensures that only one of the n! solutions is possible: {NT=blue, SA=green, WA=red}. (symmetry-breaking constraint)

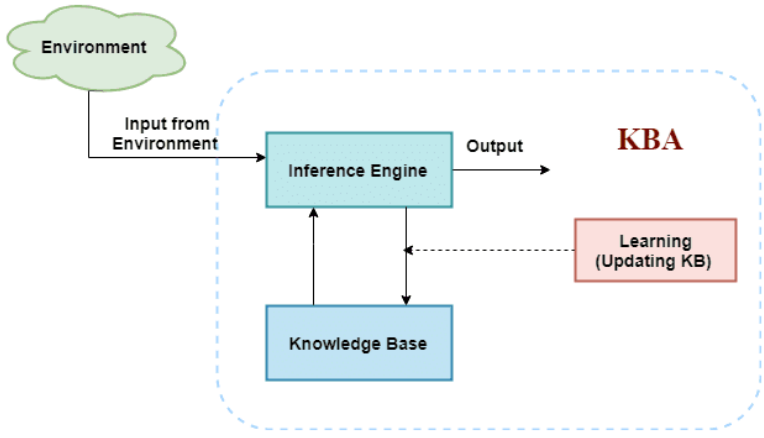
Knowledge-Based Agent in Artificial intelligence

- For efficient decision-making and reasoning, an intelligent agent need knowledge about the real world.
- Knowledge-based agents are capable of maintaining an internal state of knowledge, reasoning over that knowledge, updating their knowledge following observations, and taking actions. These agents can use some type of formal representation to represent the world and act intelligently.
- Knowledge-based agents are composed of two main parts:
    - Knowledge-base and
    - Inference system

The following must be able to be done by a knowledge-based agent:

- Agents should be able to represent states, actions, and other things.
- A representative New perceptions should be able to be incorporated.
- An agent's internal representation of the world can be updated.
- An agent can infer the world's intrinsic representation.
- An agent can deduce the best course of action.

The architecture of knowledge-based agent:



Architecture of knowledge-based agent

A generic architecture for a knowledge-based agent is depicted in the diagram above. By observing the environment, the knowledge-based agent (KBA) receives input from it. The input is taken by the agent's inference engine, which also communicates with KB to make decisions based on the knowledge store in KB. KBA's learning component keeps the KB up to date by learning new information.

Knowledge base: A knowledge-based agent's knowledge base, often known as KB, is a critical component. It's a group of sentences ('sentence' is a technical term that isn't the same as'sentence' in English). These sentences are written in what is known as a knowledge representation language. The KBA Knowledge Base contains information about the world.

Why use a knowledge base?

For an agent to learn from experiences and take action based on the knowledge, a knowledge base is required.

Inference system

Inference is the process of creating new sentences from existing ones. We can add a new sentence to the knowledge base using the inference mechanism. A proposition about the world is a sentence. The inference system uses logical rules to deduce new information from the KB.

The inference system generates new facts for an agent to update the knowledge base. An inference system is based on two rules, which are as follows:

- Forward chaining
- Backward chaining

The Wumpus World in Artificial intelligence

The Wumpus world is a basic world example that demonstrates the value of a knowledge-based agent and how knowledge representation is represented. It was inspired by Gregory Yob's 1973 video game Hunt the Wumpus.

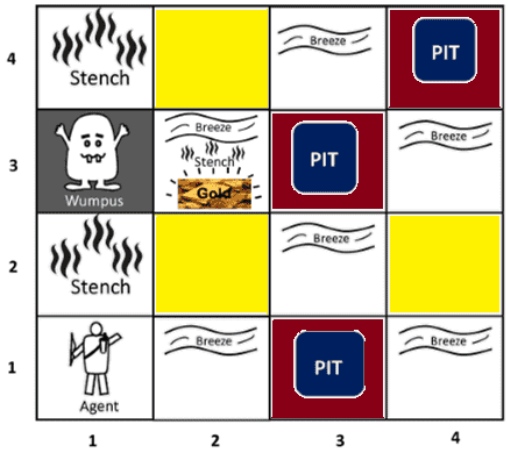
The Wumpus world is a cave with 4/4 rooms and pathways connecting them. As a result, there are a total of 16 rooms that are interconnected. We now have a knowledge-based AI capable of progressing in this world. There is an area in the cave with a beast named Wumpus who eats everybody who enters. The agent can shoot the Wumpus, but he only has a single arrow. There are some Pits chambers in the Wumpus world that are bottomless, and if an agent falls into one, he will be stuck there indefinitely. The intriguing thing about this cave is that there is a chance of finding a gold heap in one of the rooms. So the agent's mission is to find the gold and get out of the cave without getting eaten by Wumpus or falling into Pits. the agent returns with gold, he will be rewarded, but if he is devoured by Wumpus or falls into the pit, he will be penalized.

Note: Wumpus is immobile in this scene.

A sample diagram for portraying the Wumpus world is shown below. It depicts some rooms with Pits, one room with Wumpus, and one agent in the world's (1, 1) square position.

There are also some components which can help the agent to navigate the cave. These components are given as follows:

- The rooms adjacent to the Wumpus room are stinky, thus there is a stench there.
- The room next to PITs has a breeze, so if the agent gets close enough to PIT, he will feel it.
- If and only if the room contains gold, there will be glitter.
- If the agent is facing the Wumpus, the agent can kill it, and Wumpus will cry horribly, which can be heard anywhere.



PEAS description of Wumpus world:

We have given PEAS description as below to explain the Wumpus world:

Following are some basic facts about propositional logic:

Performance measure:

- If the agent emerges from the cave with the gold, he will receive 1000 bonus points.
- If you are devoured by the Wumpus or fall into the pit, you will lose 1000 points.
- For each action, you get a -1, and for using an arrow, you get a -10.
- If either agent dies or emerges from the cave, the game is over.

Environment:

- A 4*4 grid of rooms.
- Initially, the agent is in room square [1, 1], facing right.
- Except for the first square [1,1], the locations of Wumpus and gold are picked at random.
- Except for the initial square, every square of the cave has a 0.2 chance of being a pit.

Actuators:

- Left turn
- Right turn
- Move forward
- Grab
- Release
- Shoot

Sensors:

- If the agent is in the same room as the Wumpus, he will smell the stench. (Not on a diagonal.)
- If the agent is in the room directly adjacent to the Pit, he will feel a breeze.
- The agent will notice the gleam in the room where the gold is located.
- If the agent walks into a wall, he will feel the bump.
- RWhen the Wumpus is shot, it lets out a horrifying scream that can be heard from anywhere in the cave.
- These perceptions can be expressed as a five-element list in which each sensor will have its own set of indicators.
- For instance, if an agent detects smell and breeze but not glitter, bump, or shout, it might be represented as [Stench, Breeze, None, None, None].

The Wumpus world Properties:

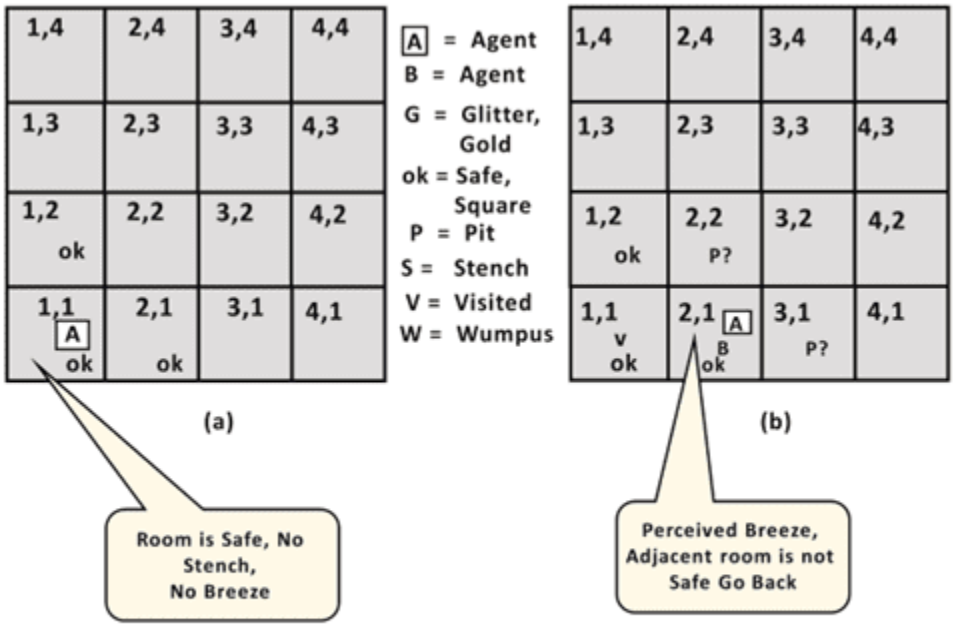
- Partially observable: The Wumpus universe is only partially viewable because the agent can only observe the immediate environment, such as a nearby room.
- Deterministic: It's deterministic because the world's result and outcome are already known.
- Sequential: It is sequential because the order is critical.
- Static: Wumpus and Pits are not moving, thus it is static.
- Discrete: There are no discrete elements in the environment.
- One agent: We only have one agent, and Wumpus is not regarded an agent, hence the environment is single agent.

Exploring the Wumpus world:

Now we will explore Wumpus' world a bit and will explain how the agent will find its goal applying logical reasoning.

Agent's First step:
At first, the agent is in the first room, or square [1,1], and we all know that this room is safe for the agent, thus we will add the sign OK to the below diagram (a) to represent that room is safe. The agent is represented by the letter A, the breeze by the letter B, the glitter or gold by the letter G, the visited room by the letter V, the pits by the letter P, and the Wumpus by the letter W.

Agent does not detect any wind or Stench in Room [1,1], indicating that the nearby squares are similarly in good condition.



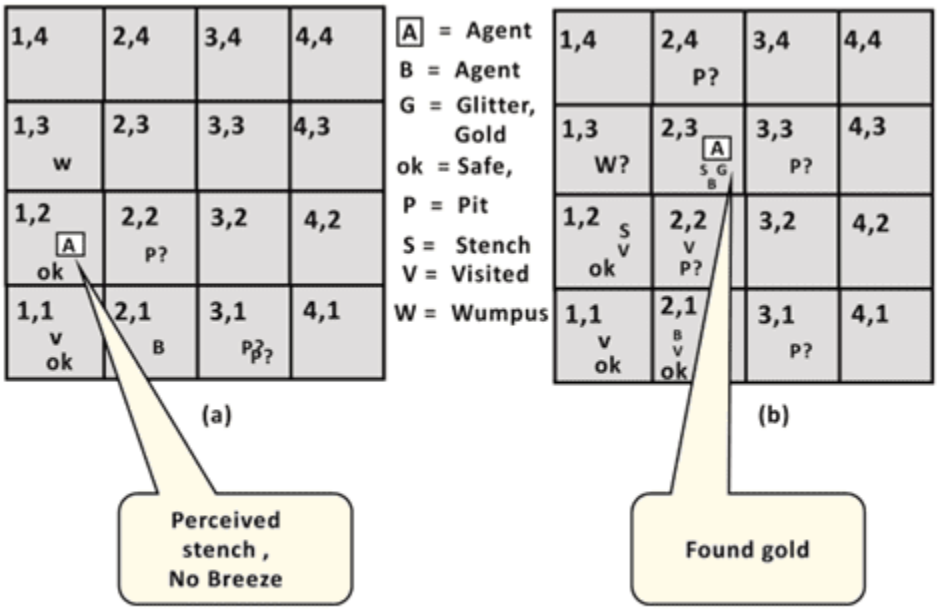
Agent's first step

Agent's second Step:
Now that the agent must go forward, it will either go to [1, 2] or [2, 1]. Let's say agent enters room [2, 1], where he detects a breeze, indicating Pit is present. Because the pit might be in [3, 1] or [2, 2], we'll add the sign P? to indicate that this is a Pit chamber.

Now the agent will pause and consider his options before doing any potentially destructive actions. The agent will return to room [1, 1]. The agent visits the rooms [1,1] and [2,1], thus we'll use the symbol V to symbolize the squares he's been to.

Agent's third step:

The agent will now proceed to the room [1,2], which is fine. Agent detects a stink in the room [1,2], indicating the presence of a Wumpus nearby. However, according to the rules of the game, Wumpus cannot be in the room [1,1], and he also cannot be in [2,2]. (Agent had not detected any stench when he was at [2,1]). As a result, the agent infers that Wumpus is in the room [1,3], and there is no breeze at the moment, implying that there is no Pit and no Wumpus in [2,2]. So that's safe, and we'll designate it as OK, and the agent will advance [2,2]



Agent's third step

Agent's fourth step:

Because there is no odor and no breeze in room [2,2], let's assume the agent decides to move to room [2,3]. Agent detects glitter in room [2,3], thus it should collect the gold and ascend out of the cave.

Propositional logic in Artificial intelligence

The simplest kind of logic is propositional logic (PL), in which all statements are made up of propositions. The term "Proposition"refers to a declarative statement that can be true or false. It's a method of expressing knowledge in logical and mathematical terms.

Example:

1. It is Sunday.
2. The Sun rises from West (False proposition)
3. 3 + 3 = 7 (False proposition)
4. 5 is a prime number.

Following are some basic facts about propositional logic:

- Because it operates with 0 and 1, propositional logic is also known as Boolean logic.

- In propositional logic, symbolic variables are used to express the logic, and any symbol can be used to represent a proposition, such as A, B, C, P, Q, R, and so on.
- Propositions can be true or untrue, but not both at the same time.
- An object, relations or functions, and logical connectives make up propositional logic.
- Logical operators are another name for these connectives.
- The essential parts of propositional logic are propositions and connectives.
- Connectives are logical operators that link two sentences together.
- Tautology, commonly known as a legitimate sentence, is a proposition formula that is always true.
- Contradiction is a proposition formula that is always false.
- Statements that are inquiries, demands, or opinions are not propositions, such as "Where is Rohini","How are you", and "What is your name" are not propositions.

Syntax of propositional logic:
The allowed sentences for knowledge representation are defined by the syntax of propositional logic. Propositions are divided into two categories:

1. Atomic Propositions.
2. Compound propositions.

- Atomic propositions: Simple assertions are referred to as atomic propositions. It is made up of only one proposition sign. These are the sentences that must be true or untrue in order to pass.

  Example:
  1. 2+2 is 4, it is an atomic proposition as it is a true fact.
  2. "The Sun is cold" is also a proposition as it is a false fact.
- Compound proposition: Simpler or atomic statements are combined with parenthesis and logical connectives to form compound propositions.

  Example:
  1. "It is raining today, and street is wet."
  2. "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:
Logical connectives are used to link two simpler ideas or to logically represent a statement. With the use of logical connectives, we can form compound assertions. There are five primary connectives, which are listed below:

1. Negation: A statement like ¬P is referred to as a negation of P. There are two types of literals: positive and negative literals.
   Example: Rohan is intelligent and hardworking. It can be written as,
   P = Rohan is intelligent,
   Q = Rohan is hardworking. →P ∧ Q.
2. Conjunction:A conjunction is a sentence that contains ∧ connective such as, P ∧ Q.
   Example: "Ritika is a doctor or Engineer",
   Here P = Ritika is Doctor. Q = Ritika is Doctor, so we can write it as P ∨ Q.
3. Disjunction: A disjunction is a sentence with a connective ∨ , such as P ∨ Q, where P and Q are the propositions.
4. Implication: An implication is a statement such as P → Q. If-then rules are another name for implications. It can be expressed as follows: If it rains, the street is flooded.
   Because P denotes rain and Q denotes a wet street, the situation is written as P and Q
5. Biconditional: A sentence like P Q, for example, is a biconditional sentence. I am alive if I am breathing. P= I am breathing, Q = I am alive, it can be represented as P ⇔ Q.

Following is the summarized table for Propositional Logic Connectives:

Connective Symbol	Technical Term	Word	Example
$\wedge$	Conjunction	AND	P $\wedge$ Q
V	Disjunction	OR	P V Q
$\rightarrow$	Implication	Implies	P $\rightarrow$ Q
$\Leftrightarrow$	Biconditional	If and only If	P $\Leftrightarrow$ Q
¬ or ~	Negation	Not	¬P or ¬Q

Truth Table:
We need to know the truth values of propositions in all feasible contexts in propositional logic. With logical connectives, we can combine all possible combinations, and the representation of these combinations in a tabular manner is known as a truth table. The truth table for all logical connectives is as follows:

For Negation:

P	¬P
true	false
false	true

For Conjunction:

P	Q	P $\wedge$ Q
true	true	true
true	false	false
false	true	false
false	false	false

For Disjunction:

P	Q	P V Q
true	true	true
true	false	true
false	true	true
false	false	false

For Implication:

P	Q	P → Q
true	true	true
true	false	false
false	true	true
false	false	true

For Biconditional:

P	Q	P ⇔ Q
true	true	true
true	false	false
false	true	false

false	false	true

Truth table with three propositions:
You can build a proposition composing three propositions P, Q, and R. The truth table is made up of 8Xn Tuples as we have taken three proposition symbols.

P	**Q**	**R**	**¬R**	**P V Q**	**P V Q → ¬R**
true	true	true	false	true	false
true	true	false	true	true	true
true	false	true	false	true	false
true	false	false	true	true	true
false	true	true	false	true	false
false	true	false	true	true	true
false	false	true	false	true	true
false	false	false	true	true	true

Precedence of connectives:
Propositional connectors or logical operators, like arithmetic operators, have a precedence order. When evaluating a propositional problem, this order should be followed. The following is a list of the operator precedence order:

Precedence	**Operators**
First Precedence	Parenthesis

Second Precedence	Negation
Third Precedence	Conjunction(AND)
Forth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Sixth Precedence	Biconditional

Note: Use parenthesis to make sure of the correct interpretations for a better understanding. For example: ¬R ∨ Q, It can be interpreted as (¬R) ∨ Q.

Logical equivalence:

One of the characteristics of propositional logic is logical equivalence. If and only if the truth table's columns are equal, two assertions are said to be logically comparable. Let's take two propositions P and Q, so for logical equivalence, we can write it as P⇔Q. In below truth table we can see that column for ¬P ∨ Q and P→Q, are identical hence P is Equivalent to P

P	**Q**	**¬P**	**¬P V Q**	**P → Q**
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

Properties of Operators:

- Commutativity:
  - P ∧ Q= Q ∧ P, or
  - P ∨ Q = Q ∨ P.
- Associativity:
  - (P ∧ Q) ∧ R = P ∧ (Q ∧ R),
  - (P ∨ Q) ∨ R= P ∨ (Q ∨ R).
- Identity element:

    - P ⋀ True = P,
    - P ⋁ True= True.
- Distributive:
    - P ⋀ (Q ⋁ R) = (P ⋀ Q) ⋁ (P ⋀ R).
    - P ⋁ (Q ⋀ R) = (P ⋁ Q) ⋀ (P ⋁ R).
- DE Morgan's Law:
    - ¬(P ⋀ Q) = (¬P) ⋁ (¬Q),
    - ¬(P ⋁ Q) = (¬ P) ⋀ (¬Q).
- Double-negation elimination:
    - ¬(¬P) = P.

Limitations of Propositional logic:

- This is not possible to represent relations like ALL, some, or none with propositional logic. Example:
    - All the girls are intelligent.
    - Some apples are sweet.
- The expressive power of propositional logic is restricted.
- We can't explain propositions in propositional logic in terms of their qualities or logical relationships.

Proving Propositional Theorem

This article discusses how to use inference rules to create proof—a series of conclusions that leads to the desired result. The most well-known rule is known as **Modus Ponens** (Latin for affirming mode) and is expressed as

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

Inferences in Proving Propositional Theorem

The notation signifies that the sentence may be deduced whenever any sentences of the type are supplied. If $(\text{WumpusAhead} \wedge \text{WumpusAlive}) \Rightarrow \text{Shoot}$ and $(\text{WumpusAhead} \wedge \text{WumpusAlive})$ are both supplied, Shoot may be deduced.

And-Elimination is another helpful inference rule, which states that any of the conjuncts can be inferred from conjunction:

$$\frac{\alpha \wedge \beta}{\alpha}$$

WumpusAlive can be deduced from $(\text{WumpusAhead} \wedge \text{WumpusAlive})$, for example. One may readily demonstrate that Modus Ponens and And-Elimination are sound once and for all by evaluating the potential truth values of α and β. These principles may then be applied to each situation in which they apply, resulting in good conclusions without the necessity of enumerating models.

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$
$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$
$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$
$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{De Morgan}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{De Morgan}$$
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$

The equations above show all of the logical equivalences that can be utilized as inference rules. The equivalence for biconditional elimination, for example, produces the two inference rules.

$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}$ and $\frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$

Some inference rules do not function in both directions in the same way. We can't, for example, run Modus Ponens in the reverse direction to get $\alpha \Rightarrow \beta$ and α from β.

Let's look at how these equivalences and inference rules may be applied in the wumpus environment. We begin with the knowledge base including R1 through R5 and demonstrate how to establish $\neg P_{1,2}$ i.e. that [1,2] does not include any pits. To generate R6, we first apply biconditional elimination to R2:

$R_6: \quad (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$

After that, we apply And-Elimination on R6 to get $R_7: \quad ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$

For contrapositives, logical equivalence yields $R_7: \quad ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$

With R8 and the percept R_4 (i.e., $\neg B_{1,1}$), we can now apply Modus Ponens to get $R_9: \quad \neg(P_{1,2} \vee P_{2,1})$.

Finally, we use De Morgan's rule to arrive at the following conclusion: $R_{10}: \quad \neg P_{1,2} \wedge \neg P_{2,1}$

That is to say, neither [1,2] nor [2,1] have a pit in them.

We found this proof by hand, but any of the search techniques may be used to produce a proof-like sequence of steps. All we have to do now is define a proof problem:

- **Initial State:** the starting point for knowledge.
- **Actions:** the set of actions is made up of all the inference rules that have been applied to all the sentences that fit the inference rule's upper half.
- **Consequences:** Adding the statement to the bottom part of the inference rule is the result of an action.
- **Objective:** The objective is to arrive at a state that contains the phrase we are attempting to verify.

As a result, looking for proofs is a viable alternative to counting models.