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.

INTRODUCTION TO GAME PLAYING AND CSP

Multiple Agent Environments:

- **Search strategies - single agent** that aims to find the solution which often expressed in the form of a **sequence of actions.**
- More than one agent leads to game theory. **(Multiple agent)**
- The environment with more than one agent is termed as multi-agent environment, in which each agent is an opponent of other agent and playing against each other.
- Each agent needs to consider the action of other agent and effect of that action on their performance.
- **Definition for Game: Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution are called adversarial searches, often known as Games**.

1. 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.

Factors in Game Theory:

Factors associated with the game theory are:

- **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 approximating the cost value at each level of the search tree, before reaching the goal node.

Types of games in AI:

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

Types of information:

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. **Examples** are Chess, Checkers, Go, etc.
- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. **Examples** are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and have a factor of chance or luck. Such games are also called as stochastic games. **Example:** Backgammon, Monopoly, Poker, etc.

Zero sum theory:

- Zero-sum games are adversarial search which involves pure competition.

- In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
- One player of the game tries to maximize one single value, while another player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.

Formalization of the problem:

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result (s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility (s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

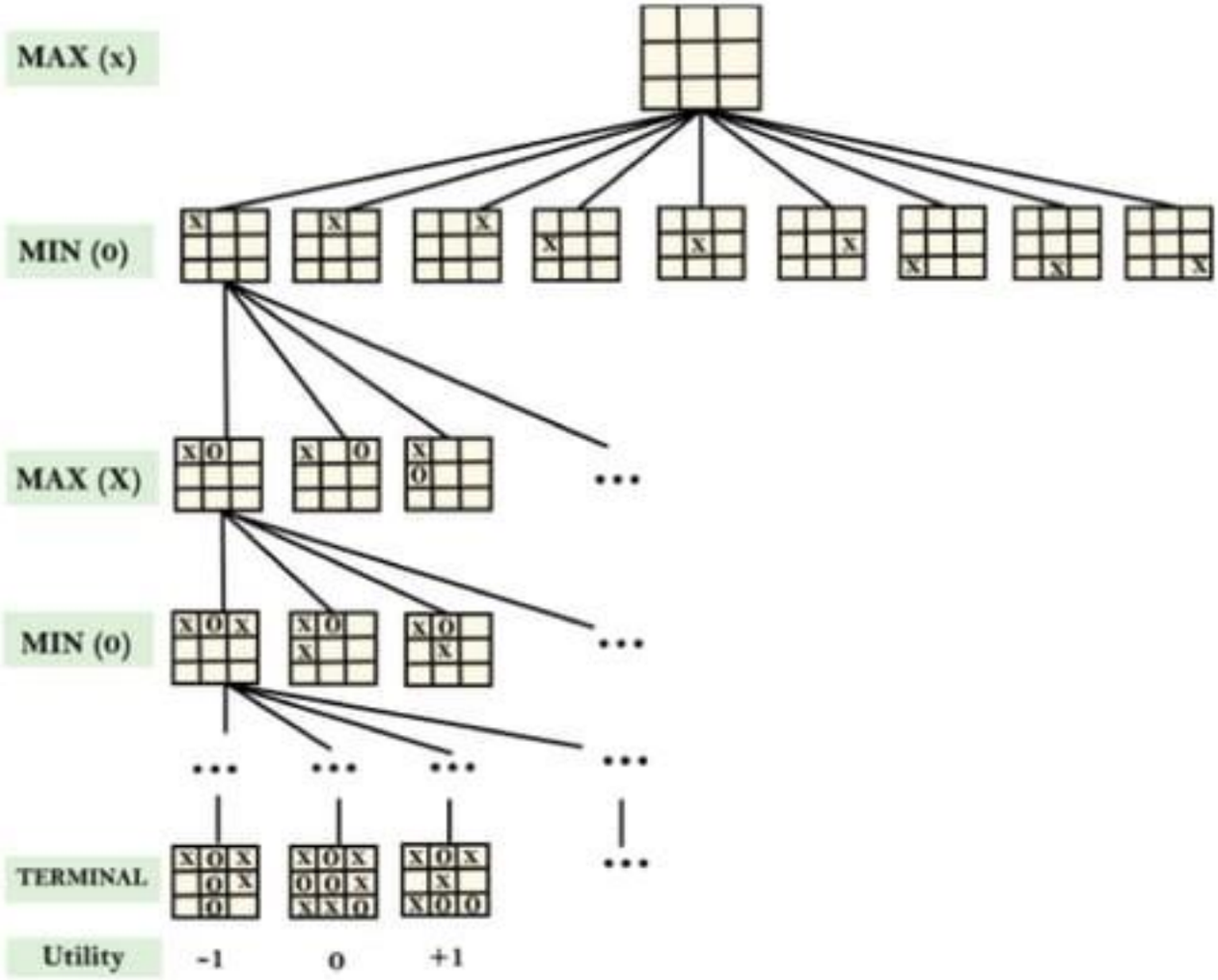
2. GAME TREE:

- A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, action's function, and result Function.

Example: Tic-Tac-Toe game tree

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.



- **INITIAL STATE (S_0):** 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**.**

Hence adversarial Search for the minimax procedure works as follows:

- It aims to find the optimal strategy for MAX to win the game.

- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN}. \end{cases}$$

3. TYPES OF ALGORITHMS IN ADVERSARIAL SEARCH:

- Adversarial search is a **game-playing** technique where the agents are surrounded by a competitive environment.
- 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:
  i. **Min-max Algorithm**
  ii. **Alpha-beta Pruning.**

i. MIN-MAX ALGORITHM:

- 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 '**Min-max,**' '**MM,**' **or** '**Saddle point.**'
- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
  Example: Chess, Checkers, tic-tac-toe
- In this algorithm two players play the game; one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
  - **MIN:** Decrease the chances of **MAX** to win the game.
  - **MAX:** Increases his chances of winning the game.
    - The minimax algorithm performs a **depth-first search algorithm** for the exploration of the complete game tree.
    - The minimax algorithm proceeds all the way down to the terminal node of the tree, then **backtrack the tree** as the recursion.

Working of MIN-MAX 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.

Pseudo code for Minimax Algorithm:

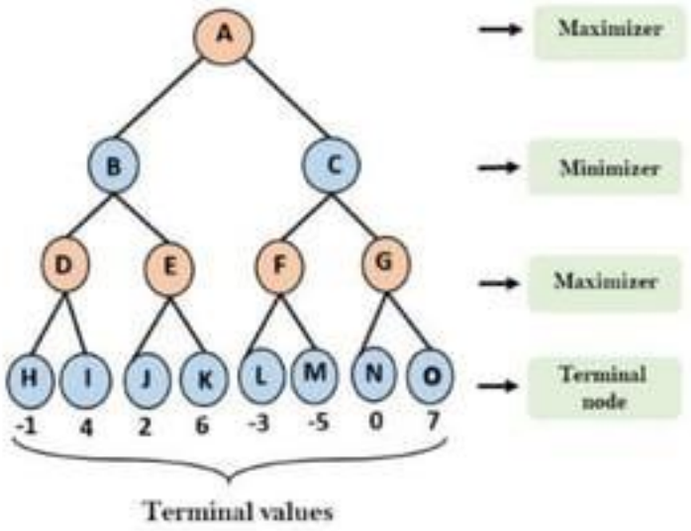
```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← -∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

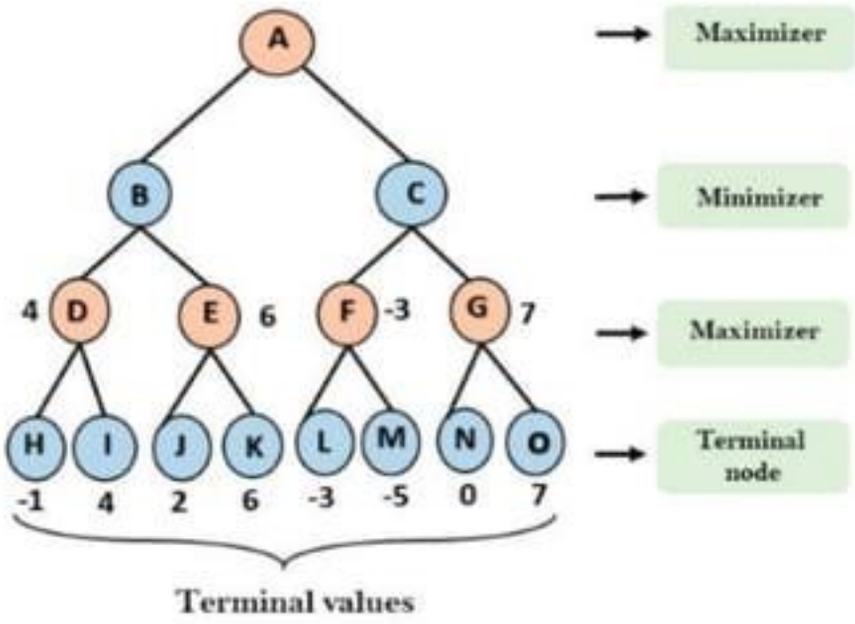
Step1:

- In the first step, the algorithm generates the entire game-tree and applies the utility function to get the utility values for the terminal states.
- In the below tree diagram, let's take A is the initial state of the tree.
- Suppose maximize takes first turn which has worst-case **initial value =- infinity**, and minimize will take next turn which has worst-case **initial value = +infinity.**



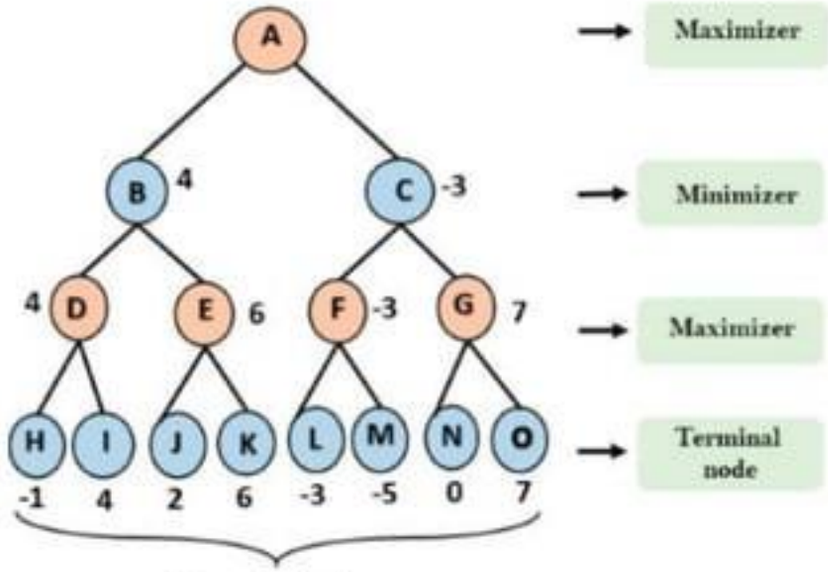
Step 2:

- Now, first we find the utilities value for the Maximize, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximize and determines the higher nodes values. It will find the maximum among the all.
  - **For node D** **max (-1,- -∞) => max (-1,4)= 4**
  - **For Node E** **max (2, -∞) => max (2, 6) = 6**
  - **For Node F** **max (-3, -∞) => max (-3,-5) = -3**
  - **For node G** **max (0, -∞) = max (0, 7) = 7**



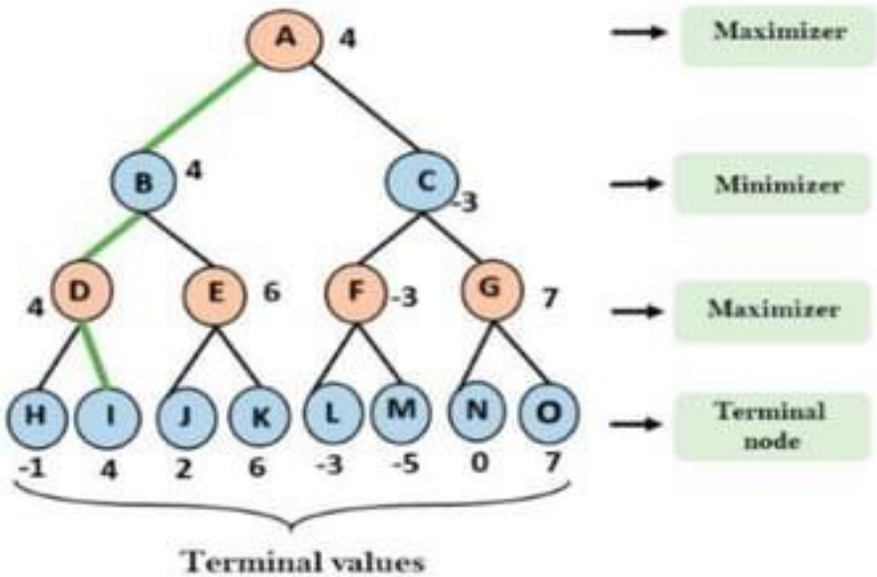
Step 3:

- In the next step, it's a turn for minimize, so it will compare all nodes value with +∞ and will find the 3rd layer node values.
    - **For node B= min (4,6) = 4**
    - **For node C= min (-3, 7) = -3**



Step 4:

- Now it's a turn for Maximize, and it will again choose the maximum of all nodes value and find the maximum value for the root node.
- In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.
    - **For node A max (4, -3)= 4**



Practical problem 2:

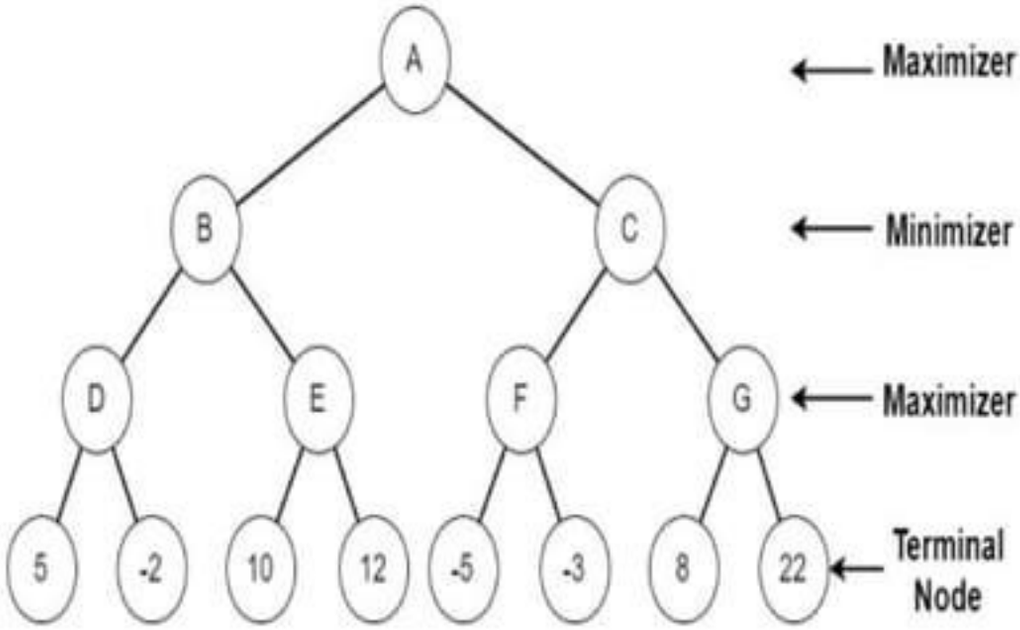


Figure 7. Minimax algorithm start

Solution:

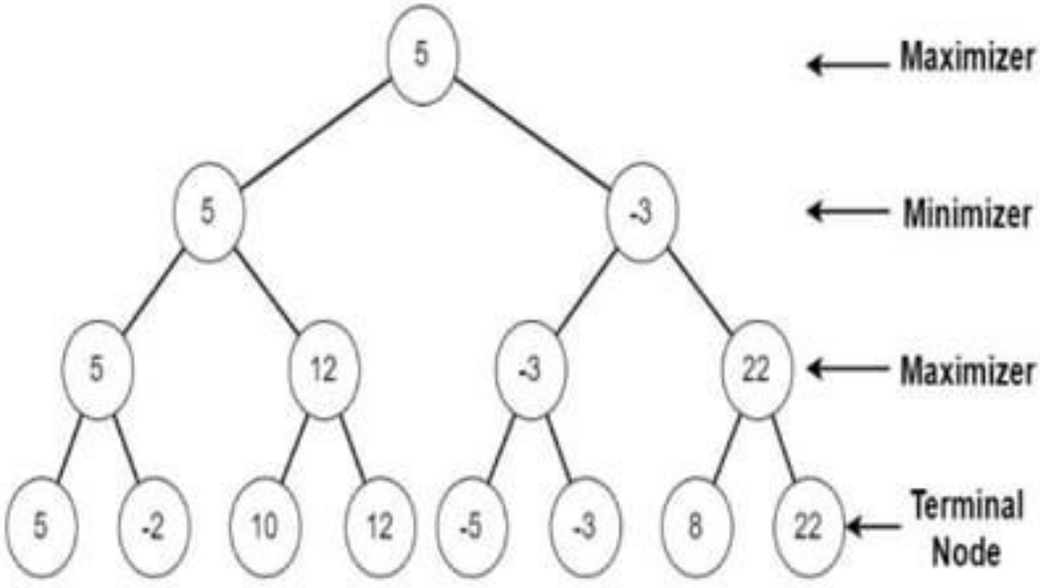


Figure 9. Minimax algorithm start

Limitations of MIN-MAX Algorithm:

- The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc.
- This type of games has a huge branching factor, and the player has lots of choices to decide.
- This limitation of the minimax algorithm can be improved from alpha-beta pruning.

ii. ALPHA –BETA PURNING:

- A modified variant of the minimax method is alpha-beta pruning. It's a way for improving the minimax algorithm.
- As a result, there is a technique known as Pruning that allows us to compute the correct minimax choice without having to inspect each node of the game tree.
- It's named alpha-beta pruning because it involves two threshold parameters, Alpha and beta, for future expansion.
- Alpha-beta pruning can be done at any depth in a tree, and it can sometimes prune the entire sub-tree as well as the tree leaves.

Conditions of Alpha-Beta Pruning:

The main condition which required for alpha-beta pruning is:

$$\alpha >= \beta$$

Key points about Alpha-Beta Pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Pseudo code for Alpha-Beta Pruning:

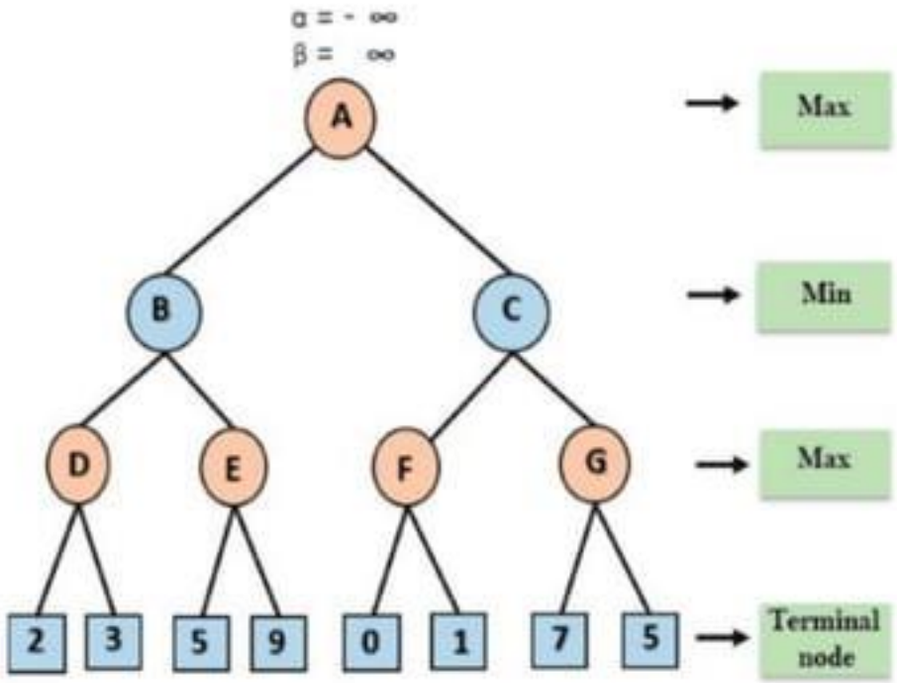
```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
  maxEva= -infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, False)
 maxEva= max(maxEva, eva)
 alpha= max(alpha, maxEva)
  if beta<=alpha
 break
 return maxEva
```

```
else                    // for Minimizer player
  minEva= +infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, true)
  minEva= min(minEva, eva)
  beta= min(beta, eva)
   if beta<=alpha
 break
return minEva
```

Working of Alpha-Beta Pruning:

Step 1:

The Max player will begin by moving from node A, where = - and = +, and passing these values of alpha and beta to node B, where again = - and = +, and Node B passing the same value to its offspring D.



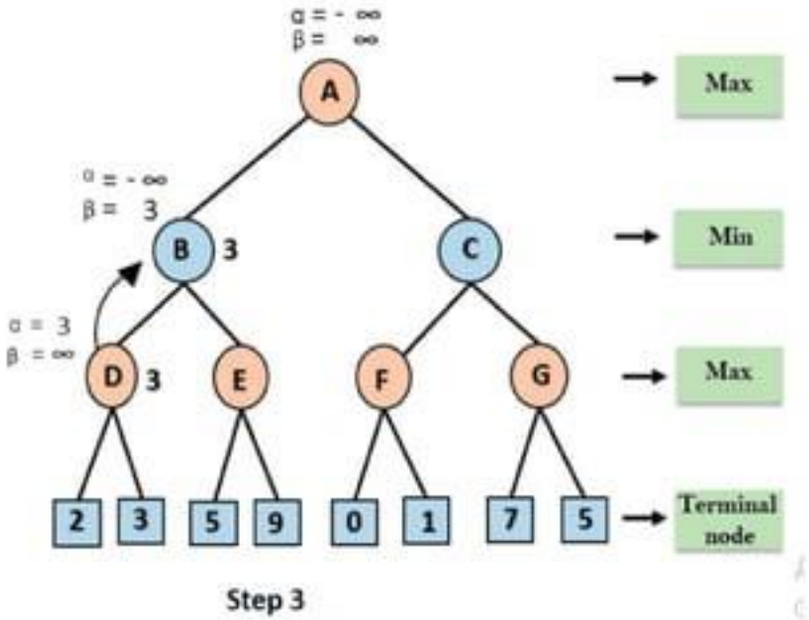
Step 1

Step 2:

- The value of will be determined as Max's turn at Node D. The value of is compared to 2, then 3, and the value of at node D will be max (2, 3) = 3, and the node value will also be 3.

Step 3:

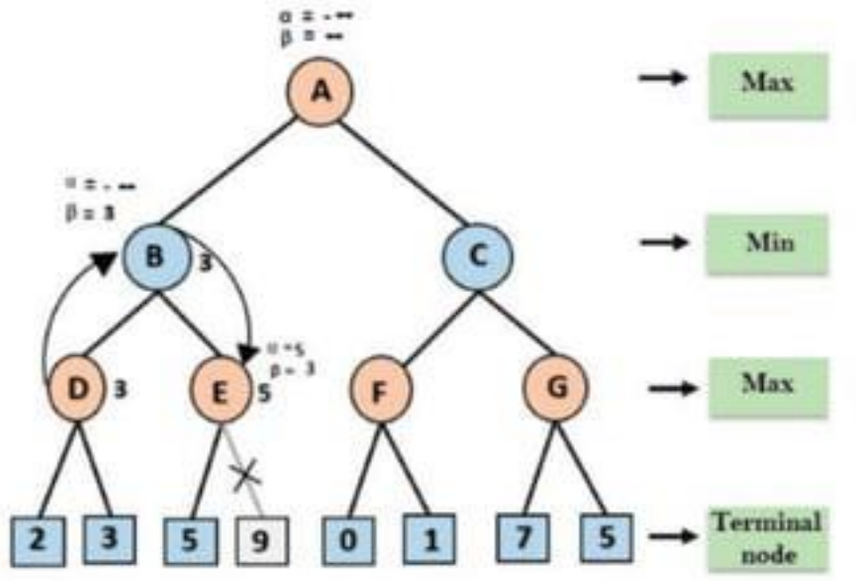
- The algorithm now returns to node B, where the value of will change as this is a turn of Min, now = +, and will compare with the value of the available subsequent nodes, i.e., min (, 3) = 3, so at node B now = -, and= 3.
- In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.



Step 3

Step 4:

Max will take its turn at node E, changing the value of alpha. The current value of alpha will be compared to 5, resulting in max (-, 5) = 5, and at node E= 5 and= 3, where >=, the right successor of E will be pruned, and the algorithm will not traverse it, and the value at node E will be 5.



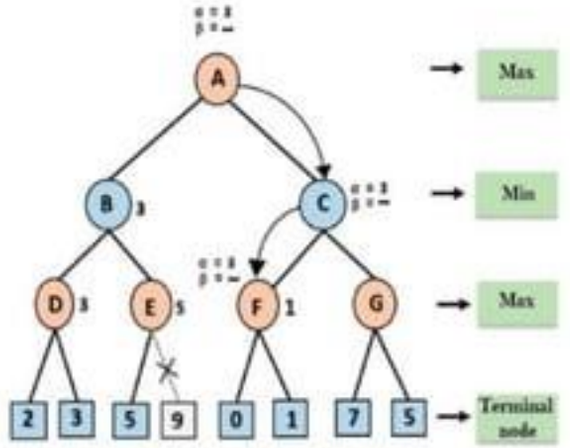
Step 4

Step 5:

- The method now goes backwards in the tree, from node B to node A.
- The value of alpha will be modified at node A, and the highest available value will be 3 as max (-, 3) = 3, and = +.
- These two values will now transfer to A's right successor, Node C.
- =3 and = + will be passed on to node F at node C, and the same values will be passed on to node F.

Step 6:

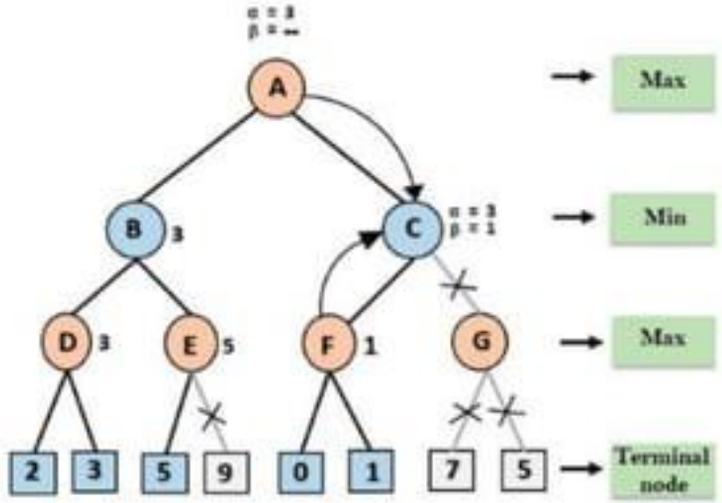
- At node F, the value of will be compared with the left child, which is 0, and max(3,0)= 3, and then with the right child, which is 1, and max(3,1)= 3 will remain the same, but the node value of F will change to 1.



Step 6

Step 7:

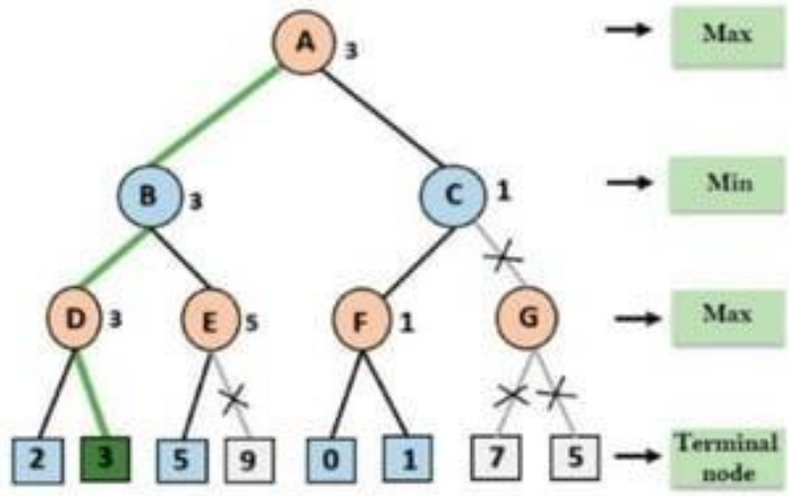
Node F returns the node value 1 to node C, at C = 3 and = +, the value of beta is modified, and it is compared to 1, resulting in min (, 1) = 1. Now, at C, =3 and = 1, and again, it meets the condition >=, the algorithm will prune the next child of C, which is G, and will not compute the complete sub-tree G.



Step 7

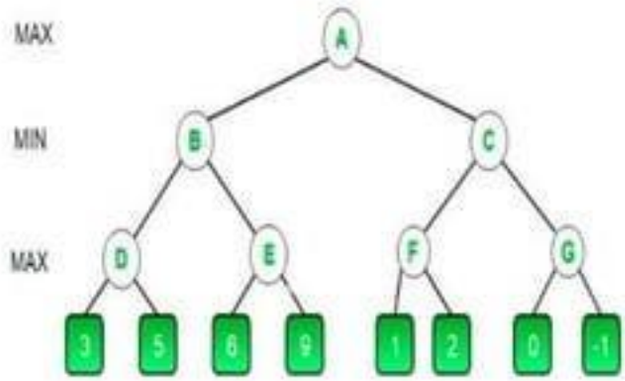
Step 8:

C now returns 1 to A, with max (3, 1) = 3 being the greatest result for A. The completed game tree, which shows calculated and uncommuted nodes, is shown below. As a result, in this case, the best maximize value is 3.

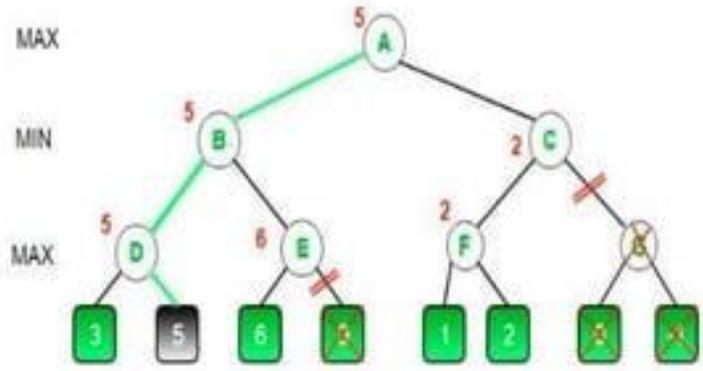


Step 8

Example 2 and Solution:



Solution:



Example 3 and Solution:

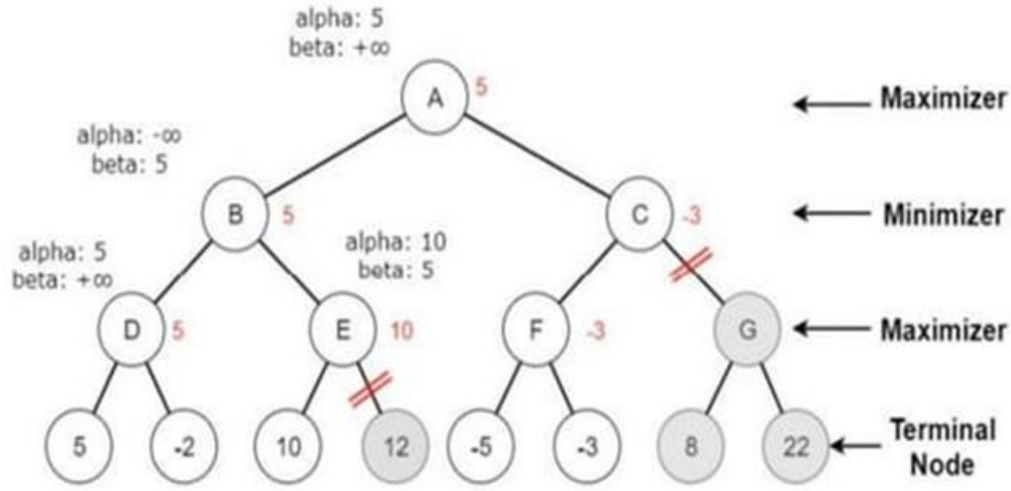
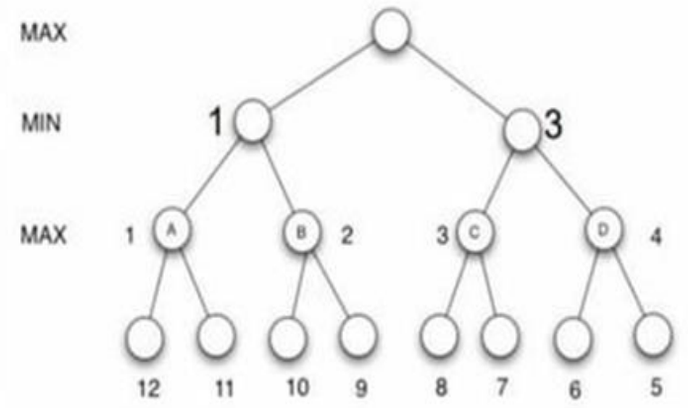


Figure 10. Alpha-Beta pruning algorithm finished calculation

Example 4 and Solution:



7. CONSTRAINT SATISFACTION PROBLEMS:

A **constraint satisfaction problem** (or CSP) is a special kind of problem that satisfies some additional structural properties beyond the basic requirements for problems in general.

Definition:

- ***State*** is defined by ***variables Xi*** with values from ***domain Di***
- ***Goal test*** is a set of ***constraints*** specifying allowable combinations of values for subsets of variables
- ***Solution*** is a ***complete, consistent assignment.***
- In a CSP, the states are defined as,
    - Finite set of **variables V1, V2, …, Vn**.
    - Finite set of **constrainsC1, C2, …, Cm.**
    - Non-empty domain of possible values for each variable **DV1, DV2, … DVn.**
    - Each **constraint Ci** limits the values that variables can take, **e.g., V1 ≠ V2**

CSP EXAMPLE:

GRAPH COLORING:

- ***Variables:*** WA, NT, Q, NSW, V, SA, T
- ***Domains: Di***={red , green , blue}
- ***Constraints:*** adjacent regions must have different colors.
  - E.g. WA ≠ NT (if the language allows this)
  - E.g. (WA, NT) ≠ {(red, green), (red, blue), (green, red)…}
- A ***state*** is defined as an ***assignment*** of values to some or all variables.
- ***Consistent assignment:*** assignment does not violate the constraints.
- A solution to a CSP is a ***complete assignment*** that satisfies all constraints.

Solution:

- **{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}**
- Simple example of a formal representation language

CSP benefits

- Standard representation language
- Generic goal and successor functions
- Useful general-purpose algorithms with more power than standard search algorithms, including generic heuristics.

Applications:

- Time table problems (exam/teaching schedules)
- Assignment problems (who teaches what)

Varieties of CSP's:

*i. **Discrete variables***

- Finite domains of size d ⇒O (dn) complete assignments.

  Eg: a Boolean CSP, NP-Complete problem
- Infinite domains (integers, strings, etc.)

Eg: job scheduling, variables are start/end days for each job

- Need a constraint language

Eg: StartJob1 +5 ≤ StartJob3.

- Linear constraints solvable, nonlinear undecidable.

ii. ***Continuous variables***

- Linear constraints solvable in poly time by linear programming methods (deal with in the field of operations research).

iii. **Our focus: discrete variables and finite domains**

iv. ***Unary constraints*** involve a single variable.

E.g. SA ≠ green

v. ***Binary constraints*** involve pairs of variables.

E.g. SA ≠ WA

vi. ***Global constraints*** involve an arbitrary number of variables.

Eg: Crypth-arithmetic column constraints.

- Preference (soft constraints) e.g. red is better than green often represent able by a cost for each variable assignment; not considered here.

REAL WORLD CSP's:

- Assignment problems
  - **E.g., who teaches what class**
- Timetable problems
  - **E.g., which class is offered when and where?**
- Transportation scheduling
- Factory scheduling

CSP as a standard search problem:

Incremental formulation:

- ***States:*** Variables and values assigned so far
- ***Initial state:*** The empty assignment
- ***Action:*** Choose any unassigned variable and assign to it a value that does not violate any constraints
  - Fail if no legal assignments
- ***Goal test:*** The current assignment is complete and satisfies all constraints.

- Same formulation for all CSPs!!!
- Solution is found at depth n (n variables).
  - **What search method would you choose?**
  - **How can we reduce the branching factor?**

Commutative:

- ***CSPs are commutative.***
  - The order of any given set of actions has no effect on the outcome.
  - ***Example:*** choose colors for Australian territories one at a time
    - **[WA=red then NT=green] same as [NT=green then WA=red]**
    - All CSP search algorithms consider a single variable assignment at a time ⇒ there are dn leaves.

CRYPTARITHMETIC PROBLEM

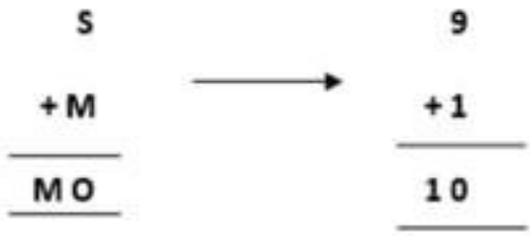
- Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols.
- In cryptarithmetic problem, the digits (0-9) get substituted by some possible alphabets or symbols.
- The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.
- We can perform all the arithmetic operations on a given cryptarithmetic problem.

Constraints for cryptarithmetic problem:

- Unique digit to be replaced with a unique alphabet (no repeated digits).
- The result should satisfy the predefined arithmetic rules, i.e., 2+2 =4
- Digits should be from 0-9 only.
- In addition operation only one carry forward.
- The problem can be solved from both sides, i.e., lefthand side (L.H.S), or right-hand side (R.H.S)

Example 1: Given a cryptarithmetic problem, i.e., S E N D + M O R E = M O N E Y.

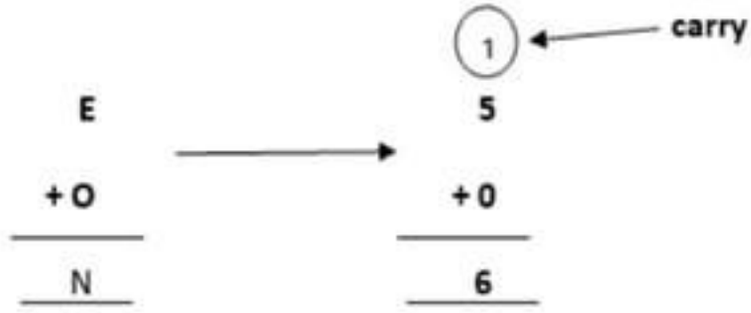
Step 1: Starting from the left hand side (L.H.S) , the terms are S and M. Assign a digit which could give a satisfactory result. Let's assign **S->9 and M->1**.



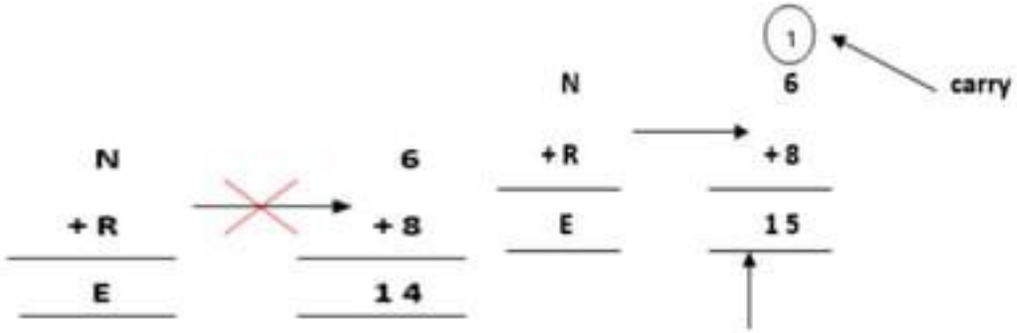
Hence, we get a satisfactory result by adding up the terms and got an assignment for **O** as **O->0** as well.

Step 2: Now, move ahead to the next terms E and O to get N as its output.

- Adding E and O, which means 5+0=0, which is not possible because we cannot assign the same digit to two letters.
- Add carry 1 to the value E to change the value of alphabet.



Step 3: Further, adding the next two terms **N** and **R** we get,



But, we have already assigned E->5. Not possible with 5 to E. Again, after solving the whole problem, we will get a carryover on this term, so our answer will be satisfied.

Step 4: Again, on adding the last two terms, i.e., the rightmost terms D and E, we get Y as its result.

where 1 will be carry forward to the above term

Keeping all the constraints in mind, the final resultant is as follows:

```
  SEND
 +MORE
------
 MONEY
```

Alphabets	Values
S	9
E	5
N	6
D	7
M	1
O	0
R	8
Y	2

Example 2:

```
  BASE
 +BALL
------
 GAMES
```

→

B	7
A	4
S	8
E	3
L	5
G	1
M	9

Example 3:

```
  YOUR
 +YOU
------
 HEART
```

→

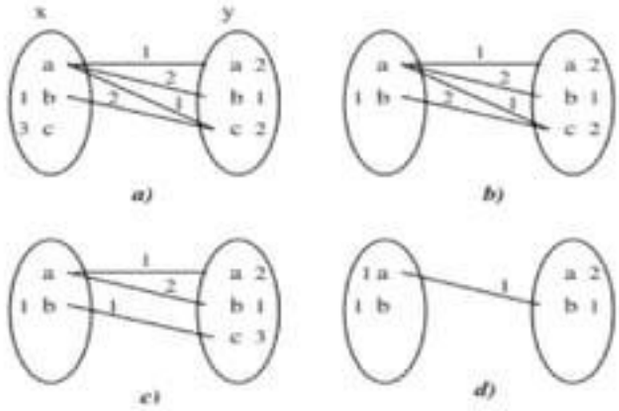
Y	9
O	4
U	2
R	6
H	1
E	0
A	3
T	8

8. CONSTRATIN PROPAGATION:

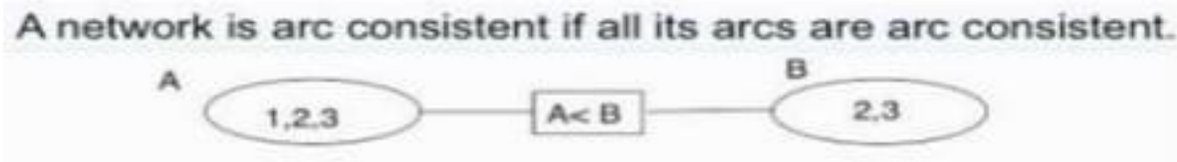
- In local state-spaces, the choice is only one, i.e., to search for a solution. But in CSP, we have two choices either:
    - We can search for a solution or
    - We can perform a special type of inference called constraint propagation.
- ***Constraint propagation is a special type of inference which 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:

  1. **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.



  2. **Arc Consistency:** A variable is arc consistent if every value in its domain satisfies the binary constraints of the variables.



  3. **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.
  4. **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.

9. BACKTRACKING CSP's:

- In CSP's, variable assignments are commutative

  For example:

  [WA = red then NT = green]

  Is the same as?

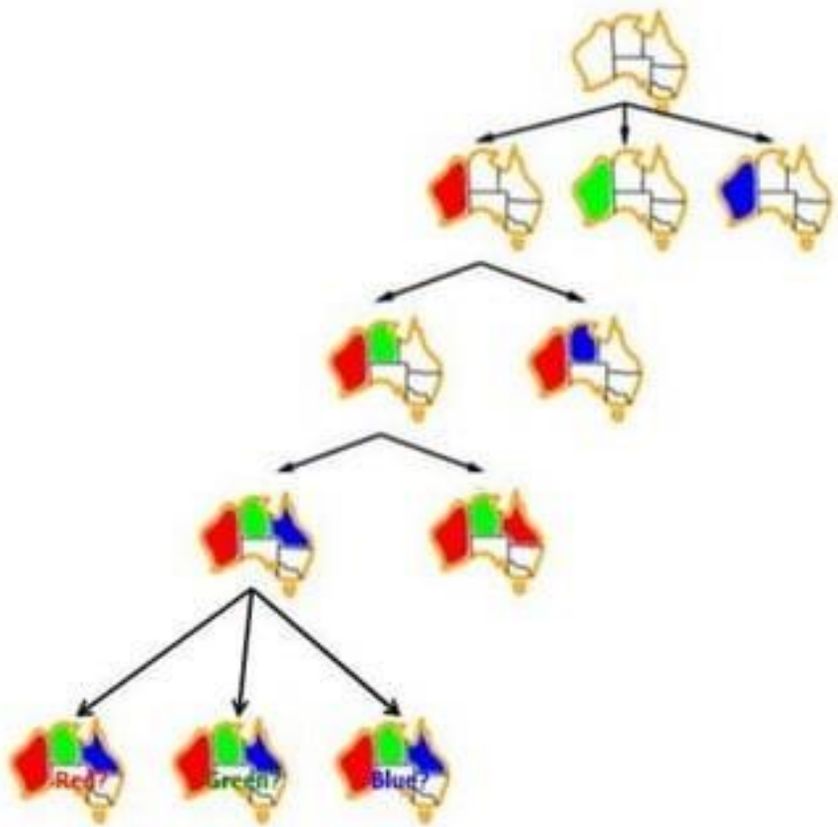
  [NT =green then WA = red]

- We only need to consider assignments to a single variable at each level (i.e., we fix the order of assignments)
- Depth-first search for CSPs with single-variable assignments is called backtracking search.

Pseudocode for Backtracking:

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given 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
```

Example:



The above diagram explanation in terms of words

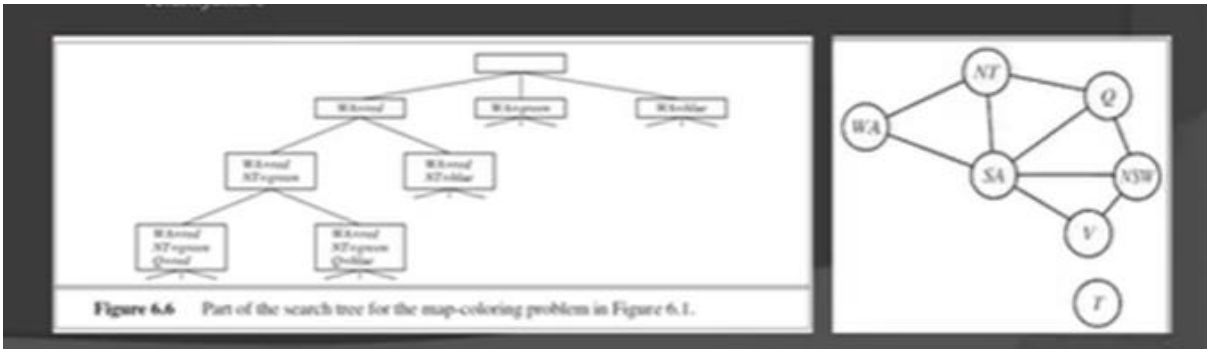


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

Unanswered questions

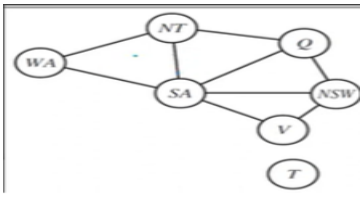
- Which variable should be assigned next (variable ordering)?
- In what order should its values be tried (value ordering)?
- What inferences should be performed at each step in the search (INFERENCE)?

Variable ordering

(a) Minimum-Remaining-Values (MRV) heuristic:

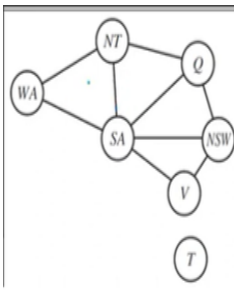
Chooses the variable with the fewest "legal" values

Also called the "most constrained variable" or "fail-first" heuristic.For example, after the assignments 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. In fact, after SA is assigned, the choices for Q, NSW, and V are all forced.



(b). Degree heuristic

selects the variable that is involved in the largest number of constraints on other unassigned variables reduces the branching factor on future choices E.g. SA is the variable with highest degree, 5. So, it will be chosen. The minimum-remaining-values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

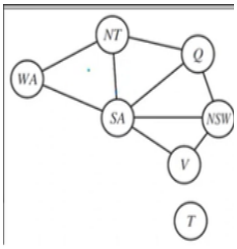


Value ordering

Least-Constraining-Value heuristic

prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph For example, 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. The least-constraining-value heuristic therefore prefers red to blue.

In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

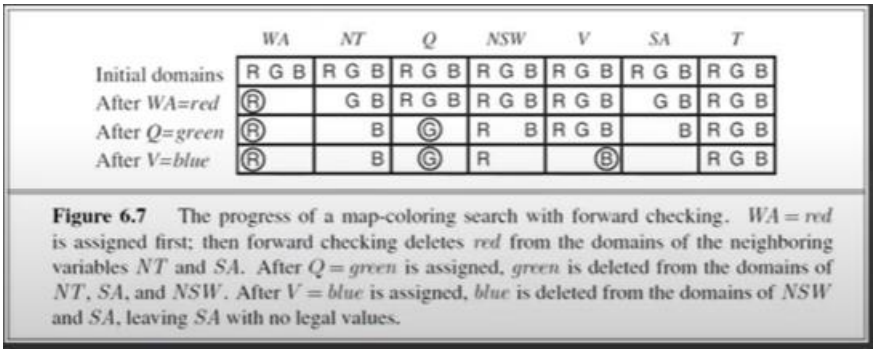


Inference

Forward Checking

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.



	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.

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.

LOCAL SEARCH FOR CSPS:

- Local search algorithms turn out to be effective in solving many CSPs. They 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.
- For example, in the 8-queens problem (see Figure 4.3), the initial state might be a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column.

- Typically, the initial guess violates several constraints. The point of local search is to eliminate the violated constraints In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic. The algorithm is shown in Figure 6.8 and its application to an 8-queens problem is diagrammed in Figure 6.9.

```
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.

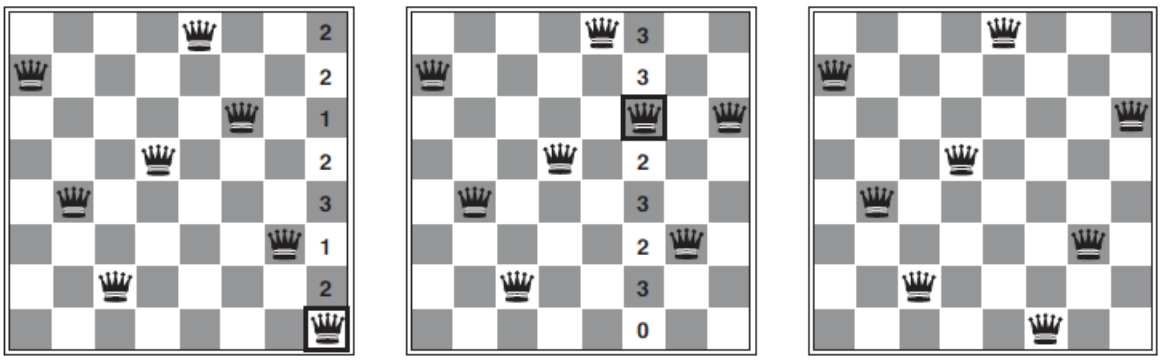


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

Min-conflicts is surprisingly effective for many CSPs. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the run time of min-conflicts is roughly *independent of problem size*. It solves even the *million*-queens problem in an averageof 50 steps (after the initial assignment). Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems.

THE STRUCTURE OF PROBLEMS

To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a **topological sort**. Figure 6.10(a) shows a sample tree and (b) shows one possible ordering.

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 two variables,

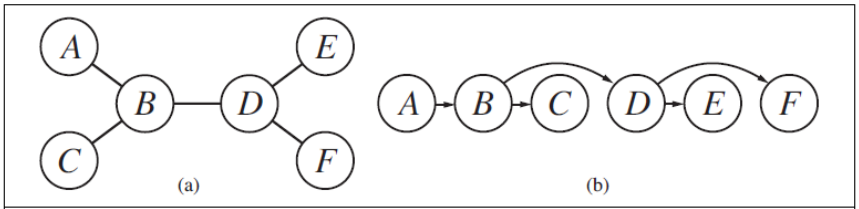


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

Two ways to reduce tree in to constraint Graphs:

. The complete algorithm is shown in Figure 6.11.

```
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.

Two ways to reduce the trees in to constraint graphs:

(1) Cut Set Conditioning:

The general algorithm is as follows:

1. 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.

2. For each possible assignment to the variables in S that satisfies all constraints on S,

(a) remove from the domains 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.

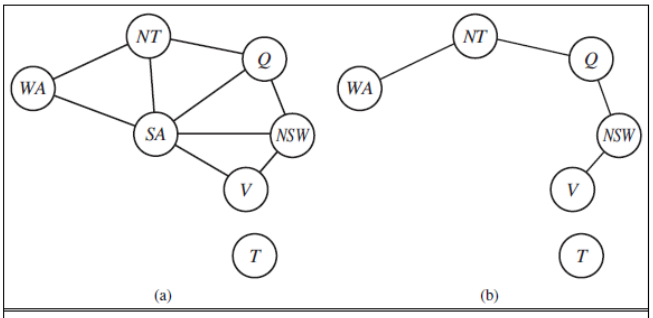


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of *SA*.

2. Tree decomposition

A tree decomposition must satisfy the following three requirements:

Every variable in the original problem appears in at least one of the subproblems.

• If two 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 two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

Figure 6.13 Atree decomposition of the constraint graph in Figure 6.12)

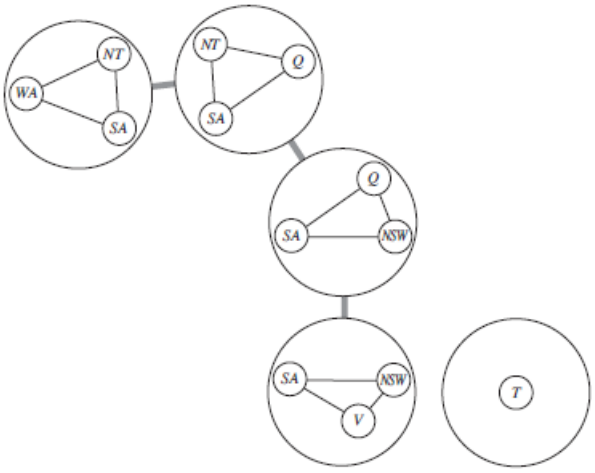


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

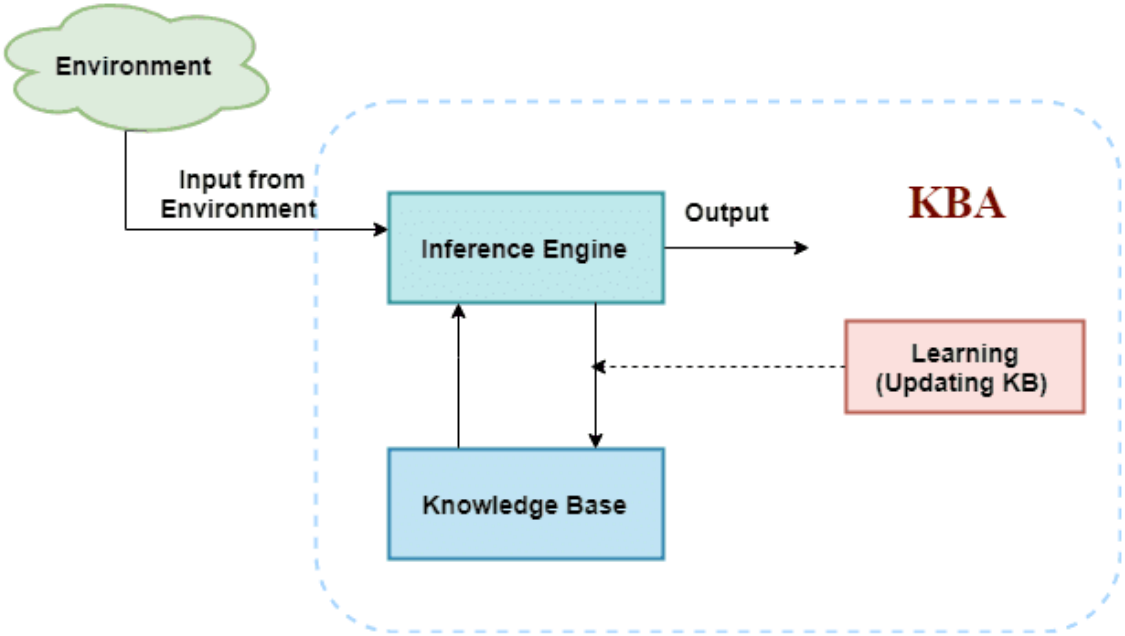
KNOWLEDGE-BASED AGENT IN ARTIFICIAL INTELLIGENCE

- 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

Operations Performed by KBA

- **TELL:** This operation tells the knowledge base, what it discern from the environment.
- **ASK:** This operation asks the knowledge base what action it should perform.
- **Perform:** It performs the selected action.

A generic knowledge-based agent:

The structure outline of a generic knowledge-based agents program:

```
function KB-AGENT(percept):
   persistent: KB, a knowledge base
t, a counter, initially 0, indicating time
TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
Action = ASK(KB, MAKE-ACTION-QUERY(t))
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t = t + 1
return action
```

The knowledge-based agent receives a percept as input and responds with an action. The agent is in charge of the knowledge base, KB, and it comes with some real-world experience. It also features a counter that starts at zero to indicate how long the entire operation will take.
When the function is called, it conducts the following three operations:

- To begin, it TELLS the KB what it sees.
- Second, it asks the KB what action it should take.
- Finally, the third agent program informs the KB of the action that was chosen.

The MAKE-PERCEPT-SENTENCE command constructs a sentence indicating that the agent perceived the specified percept at the specified time.

The MAKE-ACTION-QUERY command provides a statement that asks which action should be taken right now.

MAKE-ACTION-Statement creates a sentence stating that the selected action was carried out.

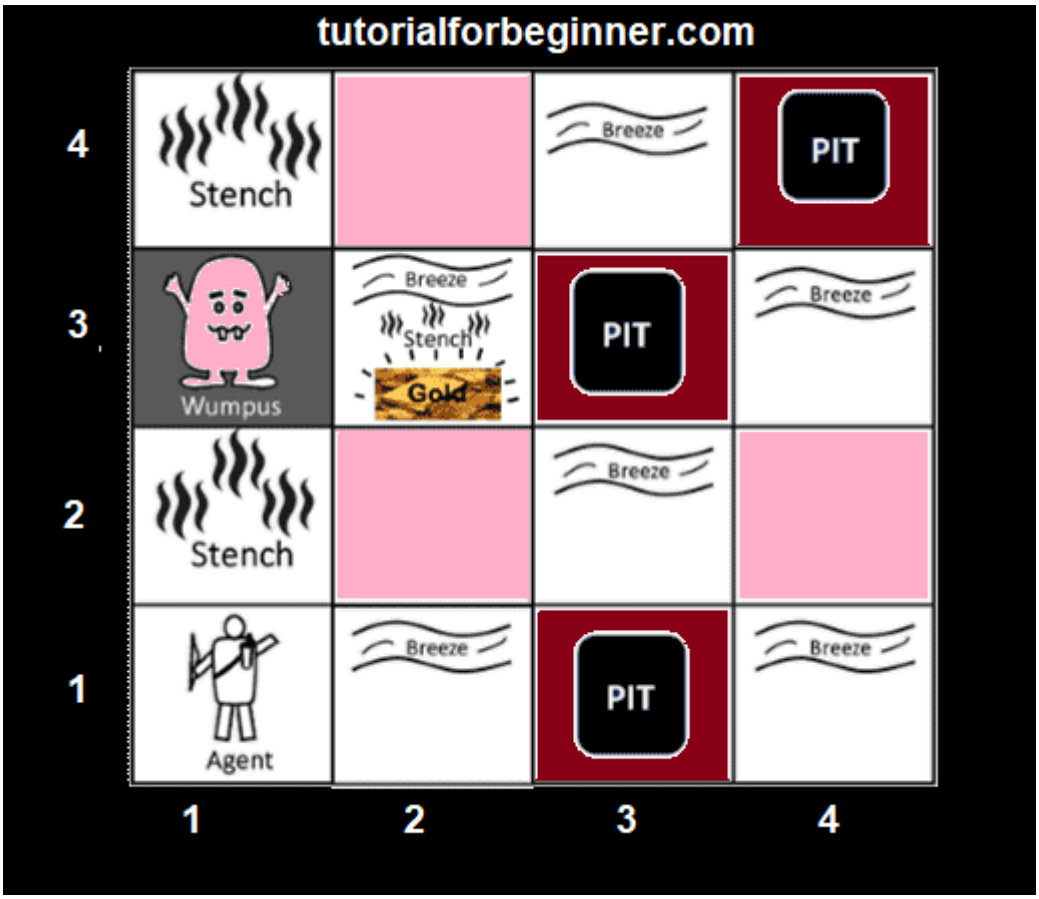
WUMPUS WORLD

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.

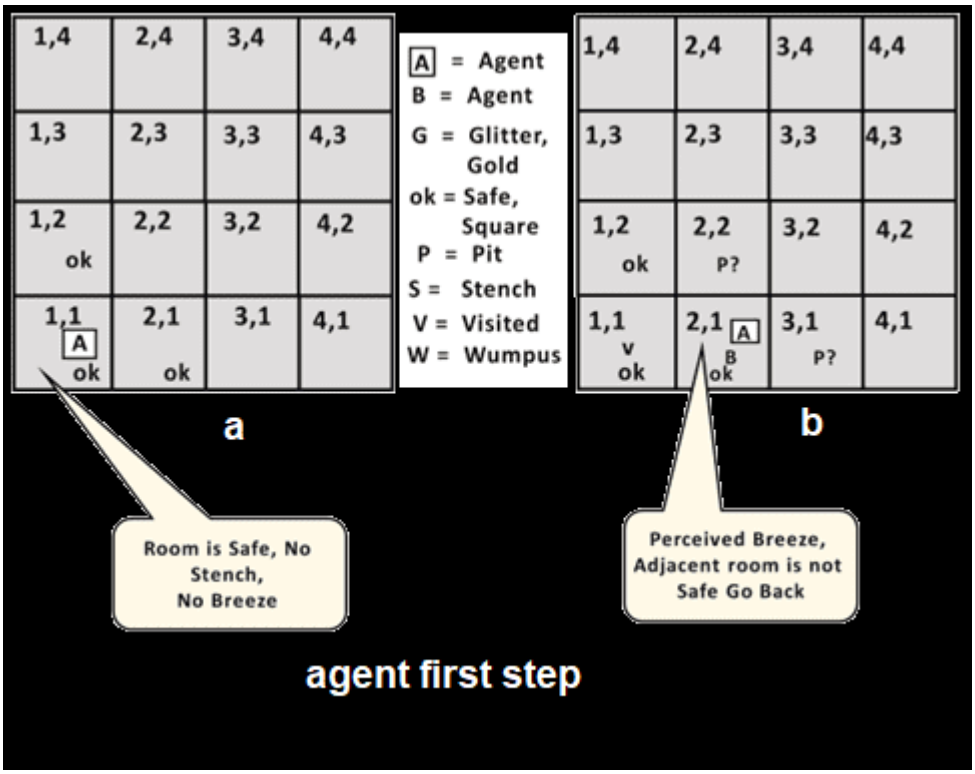
Exploring the Wumpus world:

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.



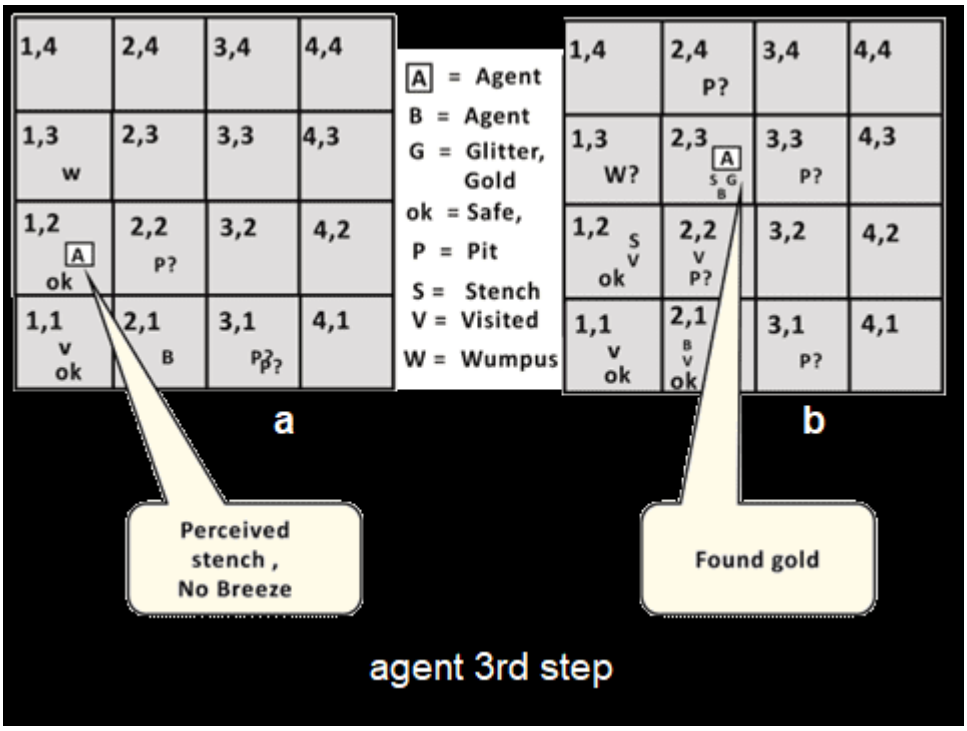
Agent's second Step:

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]



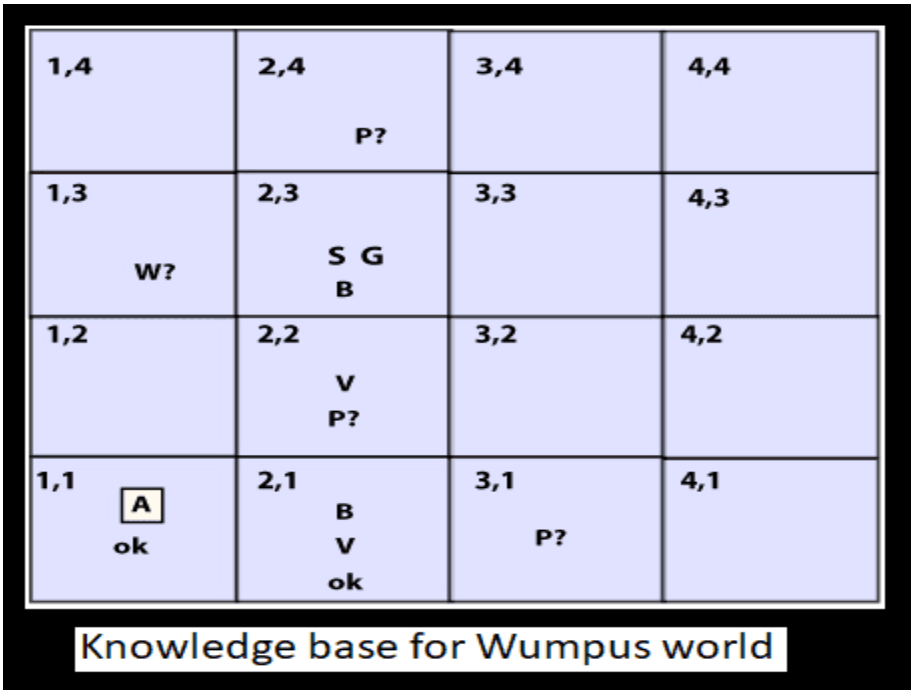
Agent's fourth step:

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.

Knowledge base for Wumpus world in Artificial intelligence

We studied about the wumpus world and how a knowledge based agent evolves the world in the previous topic. Now, in this topic, we'll establish a knowledge base for the Wumpus world and use propositional logic to deduce some Wumpus-world facts.

The agent begins his visit in the first square [1, 1], and we already know that the agent is secure in this room. We'll utilize certain rules and atomic propositions to create a knowledge base for the wumpus world. For each place in the wumpus world, we need the symbol [I j], where I stands for row location and j for column location.

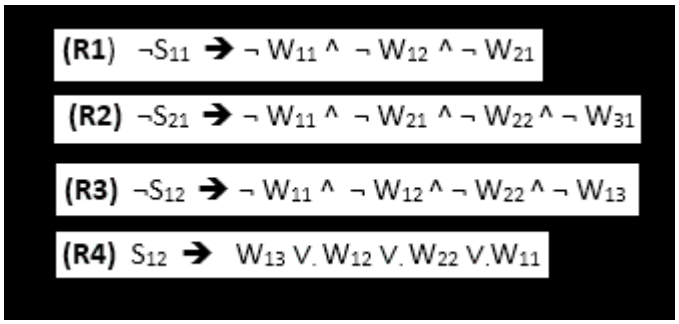


Atomic proposition variable for Wumpus world:

- Let $P_{i,j}$ be true if there is a Pit in the room [i, j].
- Let $B_{i,j}$ be true if agent perceives breeze in [i, j], (dead or alive).
- Let $W_{i,j}$ be true if there is wumpus in the square[i, j].
- Let $S_{i,j}$ be true if agent perceives stench in the square [i, j].
- Let $V_{i,j}$ be true if that square[i, j] is visited.
- Let $G_{i,j}$ be true if there is gold (and glitter) in the square [i, j].
- Let $OK_{i,j}$ be true if the room is safe.

[Note: There will be 7*4*4= 122 propositional variables for a 4 * 4 square board.]

Representation of Knowledgebase for Wumpus world:
The Simple KB for wumpus world when an agent moves from room [1, 1], to room [2,1] is as follows:



We mentioned propositional variables for room[1,1] in the first row, indicating that the room has no wumpus ($\neg W_{11}$), no smell ($\neg S_{11}$), no Pit ($\neg P_{11}$), no breeze ($\neg P_{11}$), no gold ($\neg G_{11}$), has been visited (V_{11}), and is safe (OK_{11}).

We mentioneds propositional variables for room [1,2] in the second row, indicating that there are no wumpus, stink, or breeze because an agent has not visited room [1,2], no Pit, and the room is safe.

We mentioned a propositional variable for room[2,1] in the third row, which shows that there are no wumpus($\neg W_{21}$), no stink ($\neg S_{21}$), no Pit ($\neg P_{21}$), Perceives breeze(B_{21}), no glitter($\neg G_{21}$), visited (V_{21}), and the room is secure (OK_{21}).

LOGIC

Logical AI involves representing knowledge of an agent's world, its goals and the current situation by sentences in logic. The agent decides what to do by inferring that a certain action or course of action is appropriate to achieve the goals.

PROPOSITIONAL LOGIC IN ARTIFICIAL INTELLIGENCE

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example:

1. a) It is Sunday.
2. b) The Sun rises from West (False proposition)
3. c) 3+3= 7(False proposition)

4. d) 5 is a prime number.

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- A proposition formula which is always false is called **Contradiction**.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

a. **Atomic Propositions**
b. **Compound propositions**

(a) Atomic Proposition: Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.

Example:

1. a) 2+2 is 4, it is an atomic proposition as it is a **true** fact.
2. b) "The Sun is cold" is also a proposition as it is a **false** fact.

(b)Compound proposition: Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

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

Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

1. **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has ∧ connective such as, **P ∧ Q** is called a conjunction.
   Example: Rohan is intelligent and hardworking. It can be written as,
   P= Rohan is intelligent,
   Q= Rohan is hardworking. → P∧ Q.
3. **Disjunction:** A sentence which has ∨ connective, such as **P ∨ Q**. is called disjunction, where P and Q are the propositions.
   Example: "Ritika is a doctor or Engineer",
   Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as **P ∨ Q**.
4. **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as
   If it is raining, then the street is wet.
   Let P= It is raining, and Q= Street is wet, so it is represented as P → Q
5. **Biconditional:** A sentence such as **P⇔ Q is a Biconditional sentence, example If I am breathing, then I am alive**
   P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

Sentence	→	*AtomicSentence* | *ComplexSentence*
AtomicSentence	→	*True* | *False* | *P* | *Q* | *R* | ...
ComplexSentence	→	(*Sentence*) | [*Sentence*]
	|	¬ *Sentence*
	|	*Sentence* ∧ *Sentence*
	|	*Sentence* ∨ *Sentence*
	|	*Sentence* ⇒ *Sentence*
	|	*Sentence* ⇔ *Sentence*
OPERATOR PRECEDENCE	:	¬, ∧, ∨, ⇒, ⇔

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

SEMANTICS

- The semantics defines the rules for determining the truth of a sentence with respect to a particular model. The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five

connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives.

- Atomic sentences are easy:
  - True is true in every model and False is false in every model.
  - The truth value of every other proposition symbol must be specified directly in the model.
- Complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here "iff" means "if and only if"):
  - ¬ P is true iff P is false in m.
  - P ∧ Q is true iff both P and Q are true in m.
  - P ∨ Q is true iff either P or Q is true in m.
  - P ⇒ Q is true unless P is true and Q is false in m.
  - P ⇔ Q is true iff P and Q are both true or both false in m.
- The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

A simple knowledge base for WUMPUS problem

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We focus first on the *immutable* aspects of the wumpus world,leaving the mutable aspects for a later section. For now, we need the following symbols for each [x, y] location:

- $P_{x,y}$ is true if there is a pit in [x, y].
- $W_{x,y}$ is true if there is a wumpus in [x, y], dead or alive.
- $B_{x,y}$ is true if the agent perceives a breeze in [x, y].
- $S_{x,y}$ is true if the agent perceives a stench in [x, y].

We label each sentence R_i so that we can refer to them:

• There is no pit in [1,1]:

$R_1 : \neg P_{1,1}$.

• A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.

$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$.

• The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation in Figure 7.3(b).

$R_4 : \neg B_{1,1}$.

$R_5 : B_{2,1}$.

- Returning to our wumpus-world example, the relevant proposition symbols are $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$, and $P_{3,1}$. With seven symbols, there are $2^7 = 128$ possible models; in three of these, KB is true (Figure 7.9). In those three models, $\neg P_{1,2}$ is true, hence there is no pit in [1,2]. On the other hand, $P_{2,2}$ is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2]. Figure 7.9

reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is shown in Figure 7.10.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 A truth table constructed for the knowledge base given in the text. *KB* is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, { })

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
      if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
      else return true // when KB is false, always return true
  else do
      P ← FIRST(symbols)
      rest ← REST(symbols)
      return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
              and
              TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword "**and**" is used here as a logical operation on its two arguments, returning *true* or *false*.

PROPOSITIONAL THEOREM PROVING

Before we plunge into the details of theorem-proving algorithms, we will need some logical additional concepts related to entailment.

The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$. For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure 7.11.

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

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

The **second concept** we will need is **validity**. A sentence is valid if it is true in *all* models. For example, the sentence P ∨ ¬ P is valid. Valid sentences are also known as **tautologies**—they are *necessarily* true. Because the sentence True is true in all models, every valid sentence is logically equivalent to True. From our definition , we can derive the **deduction theorem**, which was known to the ancient Greeks:

For any sentences α *and* β, α |= β *if and only if the sentence* (α ⇒ β) *is valid.*

The **final concept** we will need is **satisfiability**. SATISFIABILITY A sentence is satisfiable if it is true in, or satisfied by, *some* model. For example, the knowledge base given earlier, (R1 ∧ R2 ∧ R3 ∧ R4 ∧ R5), is satisfiable because there are three models in which it is true, as shown in Figure 7.9. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence.

Inference and proofs

This section covers **inference rules** that can be applied to derive a **proof**—a chain of conclusions that leads to the desired goal. The best-known rule is called **Modus Ponens** (Latin for *mode that affirms*) and is written

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

The notation means that, whenever any sentences of the form α ⇒ β and α are given, then the , entence β can be inferred.
For example, if (WumpusAhead ∧WumpusAlive) ⇒ Shoot,and (WumpusAhead ∧ WumpusAlive) are given, then Shoot can be inferred.

Another useful inference rule is **And-Elimination**, which says that, from a conjunction,any of the conjuncts can be inferred:

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

For example, from (WumpusAhead ∧ WumpusAlive), WumpusAlive can be inferred.
By considering the possible truth values of α and β, one can show easily that Modus Ponens and And-Elimination are sound once and for all. These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.
All of the logical equivalences in Figure 7.11 can be used as inference rules. For example, the equivalence for biconditional elimination yields the two inference rules

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

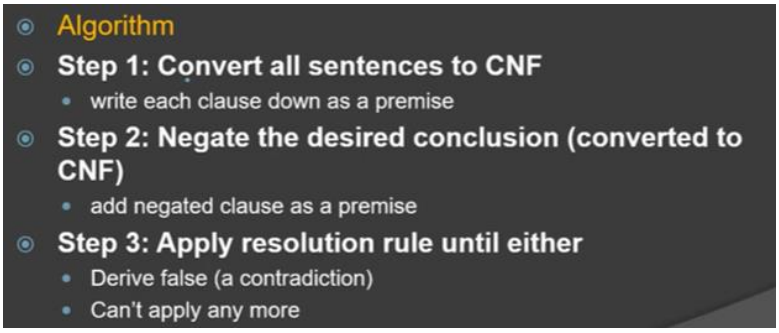
Not all inference rules work in both directions like this. For example, we cannot run Modus Ponens in the opposite direction to obtain α ⇒ β and α from β. Let us see how these inference rules and equivalences can be used in the wumpus world.

We start with the knowledge base containing R1 through R5 and show how to prove ¬ P1,2, that is, there is no pit in [1,2].

- First, we apply biconditional elimination to R2 to obtain
  R6: (B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .
  Then we apply And-Elimination to R6 to obtain
  R7: ((P1,2 ∨ P2,1) ⇒ B1,1) .
  Logical equivalence for contrapositives gives
  R8: (¬ B1,1 ⇒ ¬ (P1,2 ∨ P2,1)) .
  Now we can apply Modus Ponens with R8 and the percept R4 (i.e., ¬ B1,1), to obtain
  R9 : ¬ (P1,2 ∨ P2,1) .
  Finally, we apply De Morgan's rule, giving the conclusion
  R10 : ¬ P1,2 ∧ ¬ P2,1 .
  That is, neither [1,2] nor [2,1] contains a pit.

A Resolution Algorithm

A resolution algorithm is shown in Figure 7.12. First, (KB ∧ ¬ α) is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:



Converting to CNF:

A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** (see Figure 7.14). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence B1,1 ⇔ (P1,2 ∨ P2,1) into CNF. The steps are as follows:

There are following steps used to convert into CNF:

1) Eliminate bi-conditional implication by replacing A ⇔ B with (A → B) ∧ (B →A)
2) Eliminate implication by replacing A → B with ¬A ∨ B.
3) In CNF, negation(¬) appears only in literals, therefore we move it inwards as:
   - **¬ (¬A) ≡ A** (double-negation elimination)
   - **¬ (A ∧ B) ≡ (¬A ∨ ¬B)** (De Morgan's law)
   - **¬(A ∨ B) ≡ (¬A ∧ ¬B)** (De Morgan 's law)
4) Finally, using distributive law on the sentences, and form the CNF as:
   - **$(A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge \ldots \wedge (A_n \vee B_n)$.**

Resolution Examples:

(1)

Que 1. P ∨ Q
P → R
Q → R
Prove R.

Step	Formula	Derivation
1	P ∨ Q	Given
2	¬ P ∨ R	Given
3	¬ Q ∨ R	Given
4	¬ R	Negated conclusion

Step	Formula	Derivation
1	P ∨ Q	Given
2	¬ P ∨ R	Given
3	¬ Q ∨ R	Given
4	¬ R	Negated conclusion
5	Q ∨ R	1,2
6	¬ P	2,4
7	¬ Q	3,4
8	R	5,7
9	•	4,8

(2)

Que 2. (P → Q) → Q
(P → P) → R
(R → S) → ¬(S → Q)
Prove R.

1	P ∨ Q	
2	P ∨ R	
3	¬ P ∨ R	
4	R ∨ S	
5	R ∨ ¬ Q	
6	¬ S ∨ ¬ Q	
7	¬ R	Neg

	P ∨ Q	
	P ∨ R	
	¬ P ∨ R	
	R ∨ S	
	R ∨ ¬ Q	
	¬ S ∨ ¬ Q	
	¬ R	Neg
	S	4,7
	¬ Q	6,8
	P	1,9
	R	3,10
	•	7,11

Horn clauses and definite clauses

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm

Definite clause: A clause which is a disjunction of literals with exactly one positive literal is known as a definite clause or strict horn clause.

Example: (~p V q V k).

Horn clause: A clause which is a disjunction of literals with at most one positive literal is known as horn clause. Hence all the definite clauses are horn clauses.

• Example: (~p V ¬ q V k). It has only one positive literal k.

Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.13.)

   For example, the definite clause (¬ L1,1 ∨ ¬ Breeze ∨ B1,1) can be written as the implication (L1,1 ∧ Breeze) ⇒ B1,1. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy. In Horn form, the premise is called the **body** and the conclusion is called the **head**. A sentence consisting of a single positive literal, such as L1,1, is called a **fact**. It too can be written in implication form as True ⇒ L1,1, but it is simpler to write just L1,1.

2. Inference with Horn clauses can be done through the **forward chaining** and **backward chaining** algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow. This type of inference is the basis for **logic programming**

3. Deciding entailment with Horn clauses can be done in time that is *linear* in the size of the knowledge base—a pleasant surprise.

Forward and Backward Chaining

Forward chaining

• It determines if a single proposition symbol q—the query-is entailed by a knowledge base of definite clauses.
• It begins from known facts (positive literals) in the knowledge base.
• If all the premises of an implication are known, then its conclusion is added to the set of known facts.
• This process continues until the query q is added or until no further inferences can be made.

Algorithm:

• It determines if a single proposition q—the query-is entailed by a knowledge base of definite clauses.
• It begins from known facts (positive literals) in the knowledge base.
If all the premises of an implication are known, then its conclusion is added to the set of known facts.
• This process continues until the query q is added or until no further inferences can be made.
P = Q

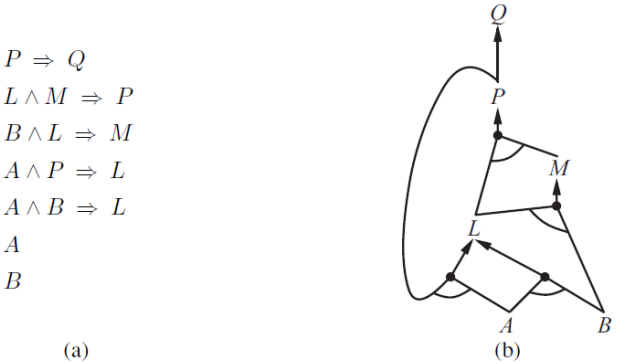
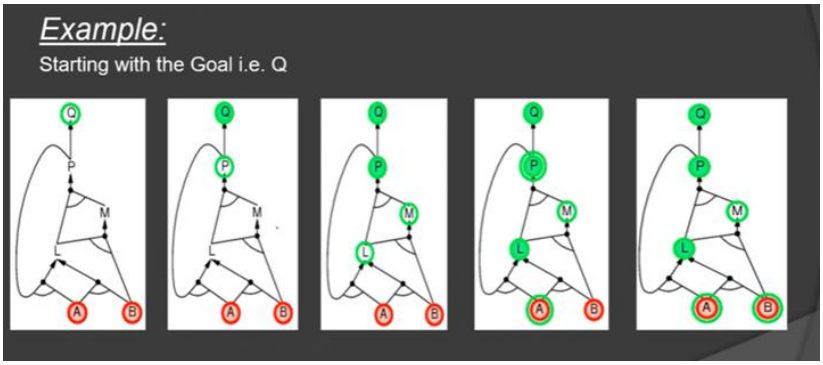


Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

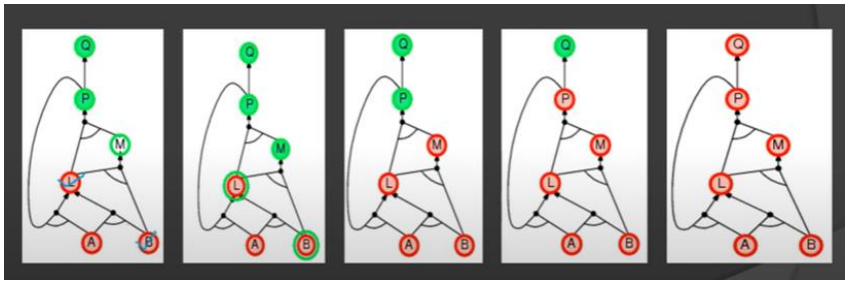
- The best way to understand the algorithm is through an example and a picture. Figure 7.16(a) shows a simple knowledge base of Horn clauses with A and B as known facts.Figure 7.16(b) shows the same knowledge base drawn as an **AND–OR graph** .
- In AND–OR graphs, multiple links joined by an arc indicate a conjunction—every link must be proved—while multiple links without an arc indicate a disjunction—any link can be proved. It is easy to see how forward chaining works in the graph.
- The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Wherever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding.

Backward chaining Algorithm:

- If the query q is known to be true, then no work is needed. Otherwise, the algorithm finds those implications in the knowledge base whose conclusion is q.
- If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 7.16, it works back down the graph until it reaches a set of known facts, A and B, that forms the basis for a proof.



Example(contd..)



EFFECTIVE PROPOSITIONAL MODEL CHECKING

- The algorithms we describe are for checking satisfiability: the SAT problem. (As noted earlier, testing entailment, $\alpha \models \beta$, can be done by testing *un*satisfiability of $\alpha \wedge \neg\beta$.) We have already noted the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of algorithms closely resemble the backtracking algorithms.

Two algorithms for the SAT problem(satisfiability problem) based on model checking:

- a. based on backtracking
- b. based on local hill-climbing search
- **(a) A complete backtracking algorithm**

- The first algorithm we consider is often called the Davis–Putnam algorithm. DPLL takes as input a sentence in conjunctive normal form—a set of

- **EARLY TERMINATION**: The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if any literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete.

- For example, the sentence (A ∨ B) ∧ (A ∨ C) is true if A is true, regardless of the values of B and C. Similarly, a sentence is false if *any* clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.

- **PURE SYMBOL HEURISTIC:** A **pure symbol** is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses (A ∨ ¬B), (¬B ∨ ¬C), and (C ∨ A), the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure. It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals true, because doing so can never make a clause false.

- **UNIT CLAUSE HEURISTIC**: A unit clause was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model. For example, if the model contains B= true, then simplifies to , which is a unit clause. assigning one unit clause can create another unit clause. for example, when is set to false,(C V A) becomes a unit clause, causing true to be assigned to A. This "cascade" of forced assignments is called **unit propagation**. It resembles the process of forward chaining with definite clauses. If the CNF expression contains only definite clauses then DPLL essentially replicates forward chaining.

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })

function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P=true}) or
         DPLL(clauses, rest, model ∪ {P=false}))
```

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

The DPLL algorithm is shown in Figure 7.17, which gives the the essential skeleton of the search process.What Figure 7.17 does not show are the tricks that enable SAT solvers to scale up to large problems. It is interesting that most of these tricks are in fact rather general, and we have seen them before in other guises:

1. **Component analysis**: As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.
2. **Variable and value ordering** : Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*. The **degree heuristic** suggests choosing the variable that appears most frequently over all remaining clauses.
3. **Intelligent backtracking** : Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of **conflict clause learning** to record conflicts so that they won't be repeated later in the search . Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.
4. **Random restarts**: Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree,rather than trying to continue. After restarting, different random choices are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.
5. **Clever indexing** : The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things as "the set of clauses in which variable Xi appears as a positive literal." This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so the indexing structures must be updated dynamically as the computation proceeds. With these enhancements, modern solvers can handle problems with tens of millions of variables. They have revolutionized areas such as hardware verification and security protocol verification, which previously required laborious, hand-guided proofs.

LOCAL SEARCH ALGORITHMS

- We have seen several local search algorithms like HILL-CLIMBING and SIMULATED-ANNEALING and Min max algorithm
- These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job.
- One of the simplest and most effective algorithms to emerge from all this work is called **WALKSAT** (Figure 7.18).
- On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip:
  (1) a "min-conflicts" step that minimizes the number of unsatisfied clauses in the new stated
  (2) a "random walk" step that picks the symbol randomly.
- When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns failure, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time. If we set max flips $=\infty$ and $p > 0$, WALKSAT will eventually return a model (if one exists), because the random-walk steps will eventually hit

```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
      if model satisfies clauses then return model
      clause ← a randomly selected clause from clauses that is false in model
      with probability p flip the value in model of a randomly selected symbol from clause
      else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

AGENTS BASED ON PROPOSITIONAL LOGIC

The first stage is to enable the agent to deduce the state of the world from its percept history to the greatest extent possible. This necessitates the creation of a thorough logical model of the consequences of actions. We also demonstrate how the agent may keep track of the world without having to return to the percept history for each inference. Finally, we demonstrate how the agent may develop plans that are guaranteed to meet its objectives using logical inference.

Wumpus World's Current State

A logical agent works by deducing what to do given a knowledge base of words about the world. Axioms are the general information about how the universe works combine with percept sentences gleaned from the agent's experience in a specific reality to form the knowledge base.

Understanding Axioms

- If there is a pit in $[x, y]$, $P_{x,y}$ is true.
- If there is a Wumpus in $[x, y]$, whether dead or living, $W_{x,y}$ is true.
- If the agent perceives a breeze in $[x, y]$, $B_{x,y}$ is true.
- If the agent detects a smell in $[x, y]$, $S_{x,y}$ is true.

The sentences we write will be adequate to infer $P_{1,2}$ (there is no pit in $[1,2]$ labelled). Each sentence is labeled R_i so that we can refer to it:

- In $[1,1]$, there is no pit:
  $R_1 : \quad \neg P_{1,1}$
- A square is breezy if and only if one of its neighbours has a pit. This must be stated for each square; for the time being, we will only add the relevant squares:
  $R_2 : \quad B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
  $R_3 : \quad B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
- In all Wumpus universes, the previous sentences are correct. The breeze percepts for the first two squares visited in the specific environment the agent is in are now included.
  $R_4 : \quad \neg B_{1,1}$
  $R_5 : \quad B_{2,1}.$

8

The agent is aware that there are no pits $(\neg P_{1,1})$ or Wumpus $(\neg W_{1,1})$ in the starting square. It also understands that a square is windy if and only if a surrounding square has a pit, and that a square is stinky if and only if a neighbouring square has a Wumpus. As a result, we include a huge number of sentences of the following type:

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$
$\dots$

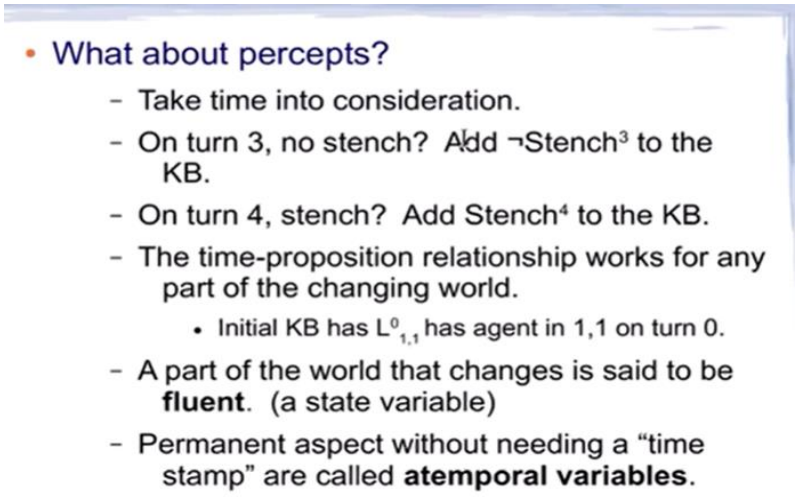
The agent is also aware that there is only one wumpus on the planet. This is split into two sections. First and foremost, we must state that there is at least one wumpus:

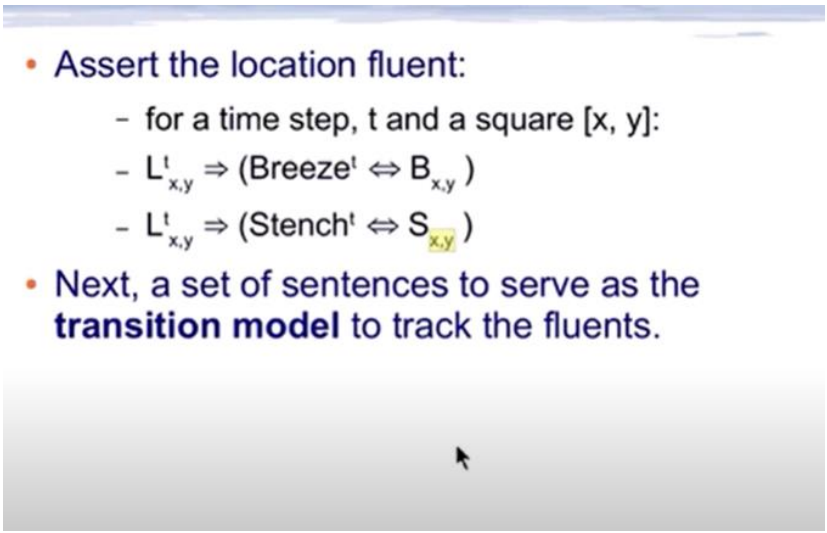
$W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,3} \vee W_{4,4}$

Then we must conclude that there is only one wumpus. We add a statement to each pair of places stating that at least one of them must be wumpus-free:

$\neg W_{1,1} \vee \neg W_{1,2}$
$\neg W_{1,1} \vee \neg W_{1,3}$
$\dots$
$\neg W_{4,3} \vee \neg W_{4,4}$

So far, everything has gone well. Let's look at the agent's perceptions now.





Of course, axioms are required to allow the agent to keep track of fluents like L_{x, y}^{t} . These fluents change as a result of the agent's activities, thus we need to write down the wumpus world's transition model as a series of logical statements.

For starters, we'll need proposition symbols for action occurrences. These symbols, like percepts, are indexed by time; for example, Forward 0 indicates that the agent performs the Forward action at time 0. The percept for a given time step occurs first, followed by the action for that time step, and then a transition to the next time step, according to the convention.

We can attempt defining effect axioms that explain the outcome of an action at the following time step to describe how the world changes. If the agent is at [1,1] facing east at time 0 and goes Forward, the consequence is that the agent is now in square [2, 1] and no longer in [1, 1] :

$$L^0_{1,1} \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L^1_{2,1} \wedge \neg L^1_{1,1})$$

Each potential time step, each of the 16 squares, and each of the four orientations would require a separate statement. For the other actions, we'd need comparable sentences: grab, shoot, climb, turnLeft, and turnRight.

Assume the agent decides to travel Forward at time 0 and records this information in its knowledge base. The agent can now derive that it is in [2, 1] using the effect axiom in the above equation and the initial statements about the state at time 0. \operatorname{ASK}\left(K B, L_{2,1}^{1}\right)=\operatorname{true}, in other words. So far, everything has gone well. Unfortunately, the news isn't so good elsewhere: if we \operatorname{ASK}\left(K B, \text { HaveArrow }^{1}\right), the result is false, which means the agent can't show it still has the arrow or that it doesn't! Because the effect axiom fails to explain what remains unchanged as a result of an action, the knowledge has been lost. The frame problem arises from the need to do so. Adding frame axioms explicitly expressing all the propositions that remain the same could be one answer to the frame problem. For each time t, we would have

$$\text{Forward}^t \Rightarrow \left(\text{HaveArrow}^t \Leftrightarrow \text{HaveArrow}^{t+1}\right)$$
$$\text{Forward}^t \Rightarrow \left(\text{WumpusAlive}^t \Leftrightarrow \text{WumpusAlive}^{t+1}\right)$$

Despite the fact that the agent now knows it still retains the arrow after going ahead and that the wumpus hasn't been killed or resurrected, the proliferation of frame axioms appears to be incredibly inefficient. The set of frame axioms in a universe with m distinct actions and n fluents will be of size O.

- Solution: write axioms about fluents, not actions.
  - Have axioms defining truth for fluents F^{t+1}, and actions that could happen at that time.
  - Truth can be set by either the action at time t which makes F true at t+1 OR F was already true and the t-time action doesn't make it false.
  - Referred to as a successor-state axiom:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$$

- HaveArrow is a simple example. Replace ActionCausesFt part with:

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t) . \quad (7.2)$$

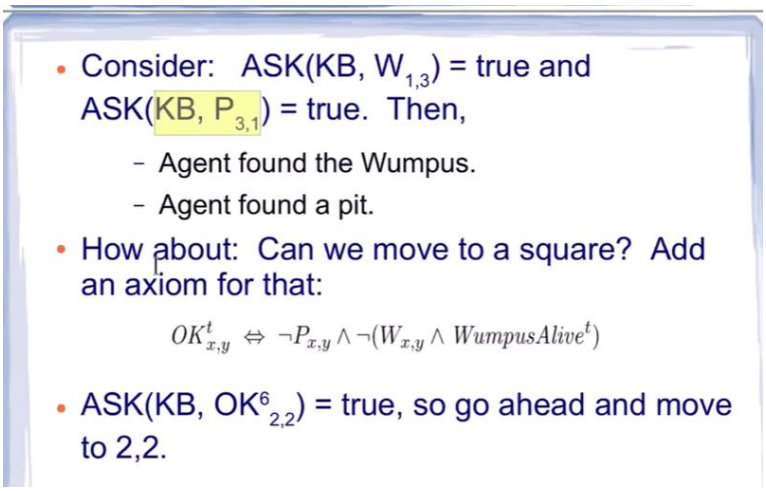
- Location more involved. $L^{t+1}_{1,1}$ is true if the agent moved Forward from [1,2] when facing south, or from [2,1] when facing west; OR $L^t_{1,1}$ was already true and we didn't move:

$$\begin{aligned} L^{t+1}_{1,1} \Leftrightarrow\ & (L^t_{1,1} \wedge (\neg Forward^t \vee Bump^{t+1})) \\ \vee\ & (L^t_{1,2} \wedge (South^t \wedge Forward^t)) \\ \vee\ & (L^t_{2,1} \wedge (West^t \wedge Forward^t)) . \end{aligned} \quad (7.3)$$

- With successor-state axioms and the previously defined axioms, the agent can ASK anything that can be answered in the world.
- Given the initial sequence of percepts and actions:

$$\begin{aligned} &\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 \ ; \ Forward^0 \\ &\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 \ ; \ TurnRight^1 \\ &\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 \ ; \ TurnRight^2 \\ &\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 \ ; \ Forward^3 \\ &\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 \ ; \ TurnRight^4 \\ &\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 \ ; \ Forward^5 \\ &Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6 \end{aligned}$$

  - ASK(KB, $L^6_{1,2}$) = true, and the agent knows where it is.



Finally, ASK(KB,OK62,2)=true, so the square [2, 2] is OK to move into. In fact, given a sound and complete inference algorithm such as DPLL, the agent can answer any answerable question about which squares are OK—and can do so in just a few milliseconds for small-to-medium wumpus worlds.