UNIT - III

Adversarial Search: 2-Player Games

➢ **Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.**

- we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions.
- But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.
- 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.
- So, **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**.
- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

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

- **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 has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games.
  Example: Backgammon, Monopoly, Poker, etc.

Zero-Sum Game

➢Zero-sum game :-

- 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 try to maximize one single value, while other 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.

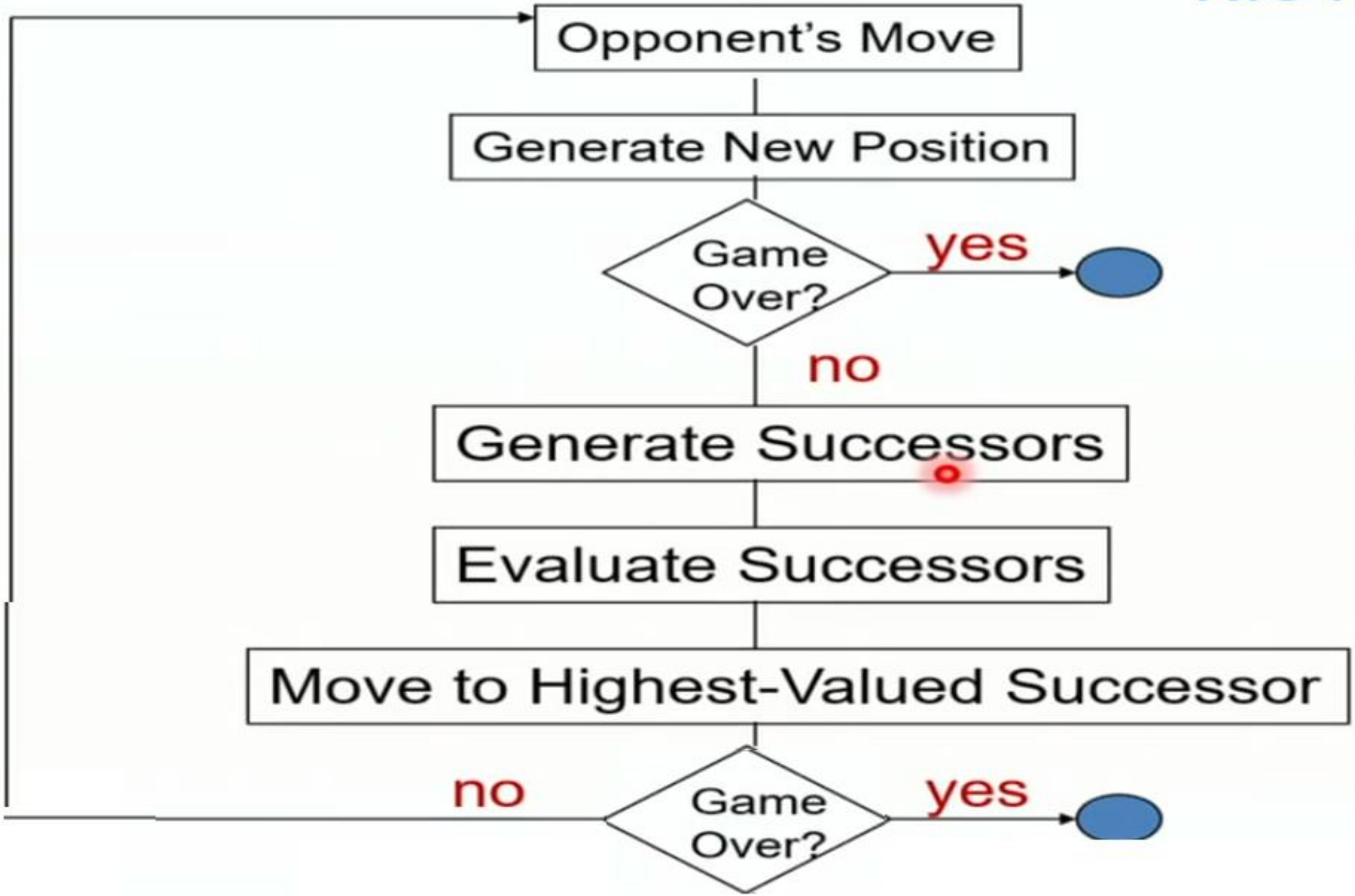
➢Zero-sum game: Embedded thinking

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

- What to do.
- How to decide the move
- Needs to think about his opponent as well
- The opponent also thinks what to do

Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Two-Player Game



➢ Formalization of the problem:

A game can be defined as a type of search in AI which can be formalized of the following elements:

- **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(win), -1(loss), 0(tie), And for tic-tac-toe, utility values are +1, -1, and 0.

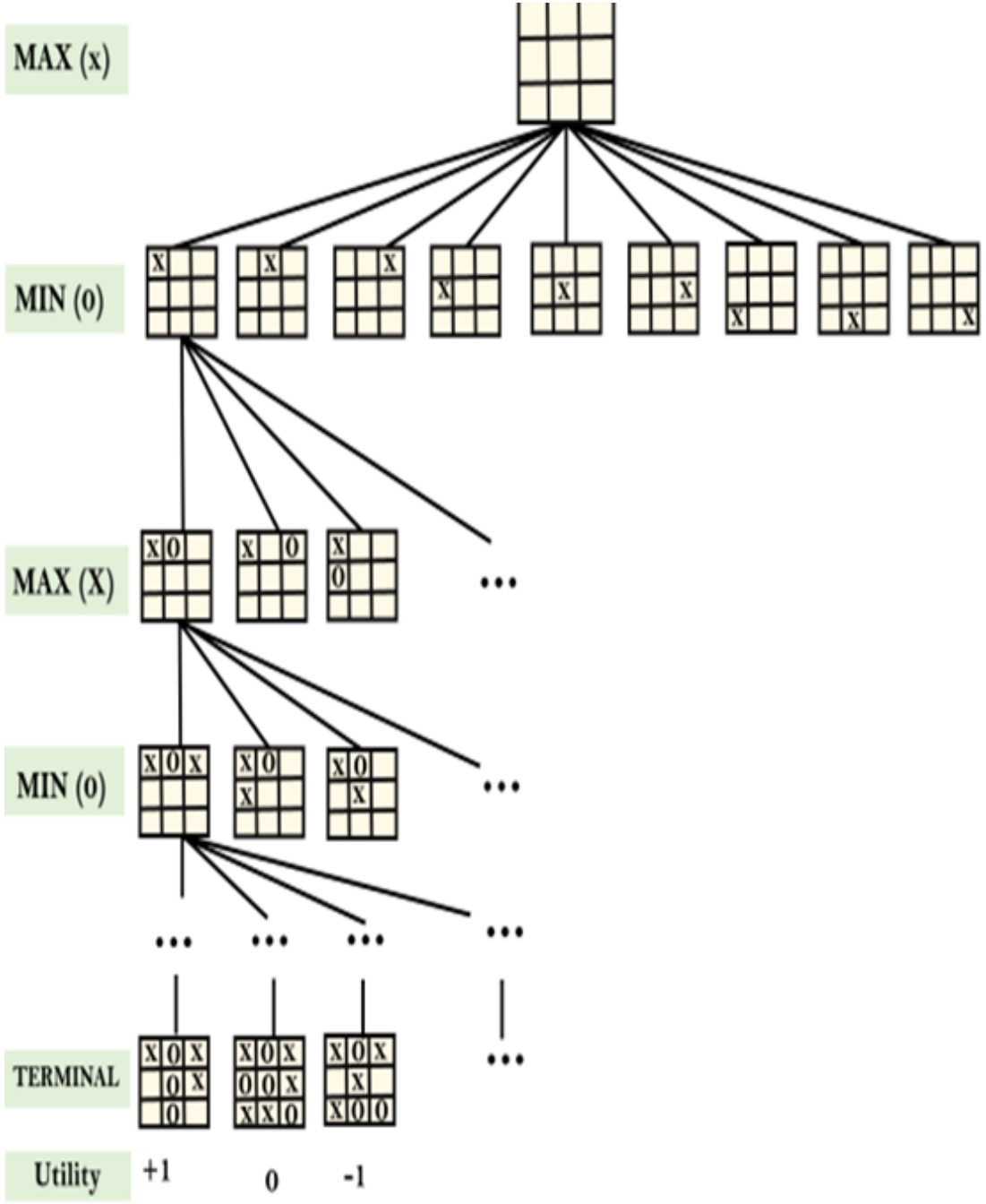
Game tree :

- 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, actions 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.



Example Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

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) =

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

In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

Optimal Decisions in Games: AND-OR graph

- Let us start with games with two players, whom we'll refer to as MAX and MIN for obvious reasons. MAX is the first to move, and then they take turns until the game is finished. At the conclusion of the game, the victorious player receives points, while the loser receives penalties. A game can be formalized as a type of search problem that has the following elements:

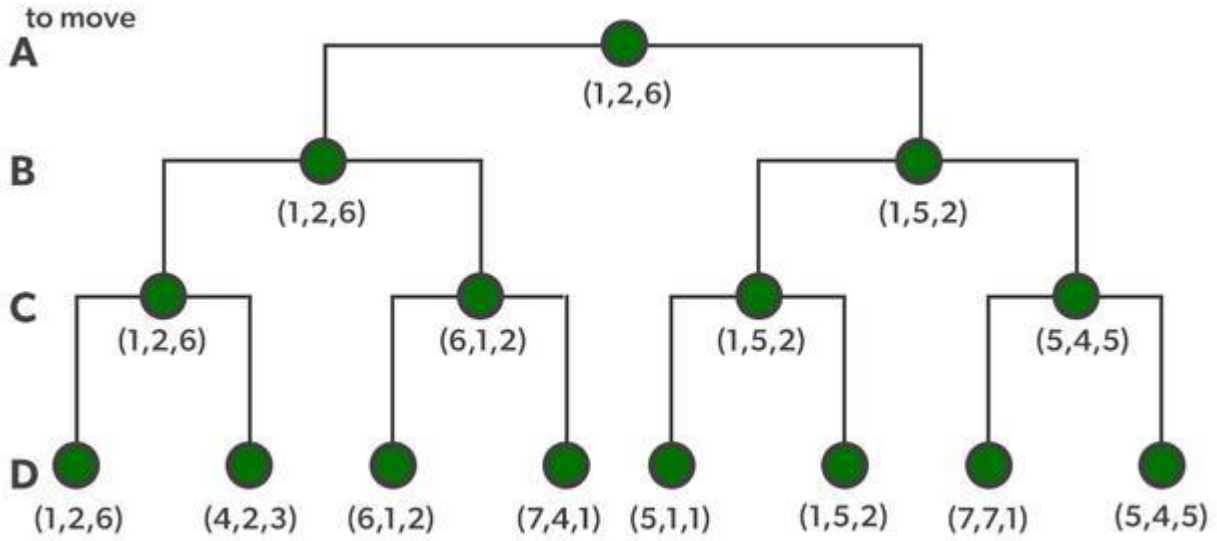
- **S_0:** The initial state of the game, which describes how it is set up at the start.
- **Player (s):** Defines which player in a state has the move.
- **Actions (s):** Returns a state's set of legal moves.
- **Result (s, a):** A transition model that defines a move's outcome.
- **Terminal-Test (s):** A terminal test that returns true if the game is over but false otherwise. Terminal states are those in which the game has come to a conclusion.
- **Utility (s, p):** A utility function (also known as a payout function or objective function) determines the final numeric value for a game that concludes in the terminal state ***s*** for player ***p***.

The result in chess is a win, a loss, or a draw, with values of +1, -1, or 0

Backgammon's payoffs range from 0 to +192, but certain games have a greater range of possible outcomes.

A zero-sum game is defined (confusingly) as one in which the total reward to all players is the same for each game instance.

- The game tree for the game is defined by the beginning state, ACTIONS function, and RESULT function—a tree in which the nodes are game states and the edges represent movements.
- The game tree for tic-tac-toe is relatively short, with just 9! = 362,880 terminal nodes. However, because there are over 1040 nodes in chess, the game tree is better viewed as a theoretical construct that cannot be realized in the actual world.

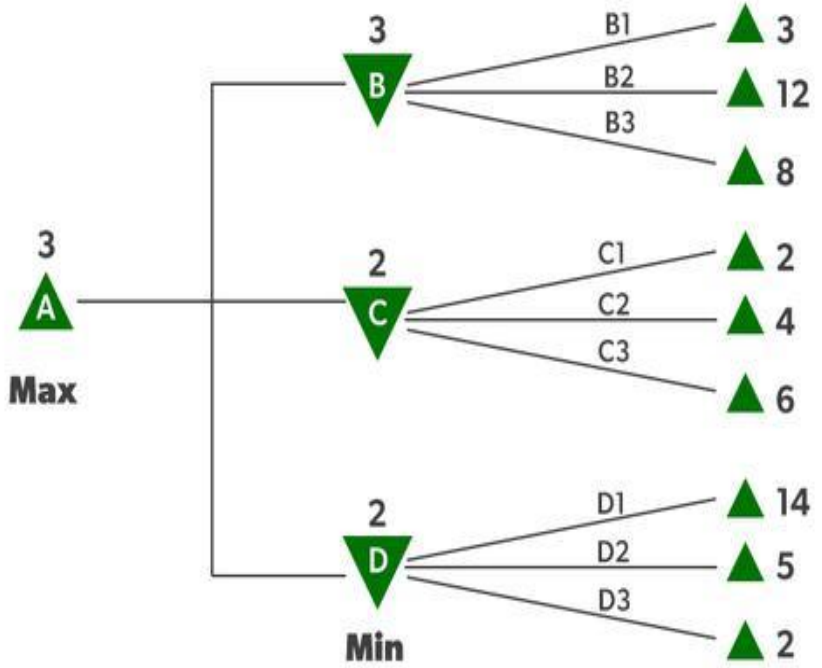


- A sequence of actions leading to a goal state—a terminal state that is a win—would be the best solution in a typical search problem.
- MIN has something to say about it in an adversarial search. MAX must therefore devise a contingent strategy that specifies MAX's initial state move, then MAX's movements in the states resulting from every conceivable MIN response, then MAX's moves in the states resulting from every possible MIN reaction to those moves, and so on.
- This is quite similar to the AND-OR search method, with MAX acting as OR and MIN acting as AND.
- We'll move to the trivial game in the ***figure below*** since even a simple game like tic-tac-toe is too complex for us to draw the full game tree on one page. MAX's root node moves are designated by the letters a1, a2, and a3. MIN's probable answers to a1 are b1, b2, b3, and so on. This game is over after MAX and MIN each make one move. (In game terms, this tree consists of two half-moves and is one move deep, each of which is referred to as a ply.) The terminal states in this game have utility values ranging from 2 to 14.

Game's Utility Function

- The optimal strategy can be found from the minimax value of each node, which we express as MINIMAX, given a game tree (n).
- Assuming that both players play optimally from there through the finish of the game, the utility (for MAX) of being in the corresponding state is the node's minimax value. The usefulness of a terminal state is obviously its minimax value. Furthermore, if given the option, MAX prefers to shift to a maximum value state, whereas MIN wants to move to a minimum value state. So here's what we've got:

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



- The game’s UTILITY function provides utility values to the terminal nodes on the bottom level. Because the first MIN node, B, has three successor states with values of 3, 12, and 8, its minimax value is 3. Minimax value 2 is also used by the other two MIN nodes. The root node is a MAX node, with minimax values of 3, 2, and 2, resulting in a minimax value of 3. We can also find the root of the minimax decision: action a1 is the best option for MAX since it leads to the highest minimax value.
- This concept of optimal MAX play requires that MIN plays optimally as well—it maximizes MAX’s worst-case outcome. What happens if MIN isn’t performing at its best? Then it’s a simple matter of demonstrating that MAX can perform even better. Other strategies may outperform the minimax method against suboptimal opponents, but they will always outperform optimal opponents.

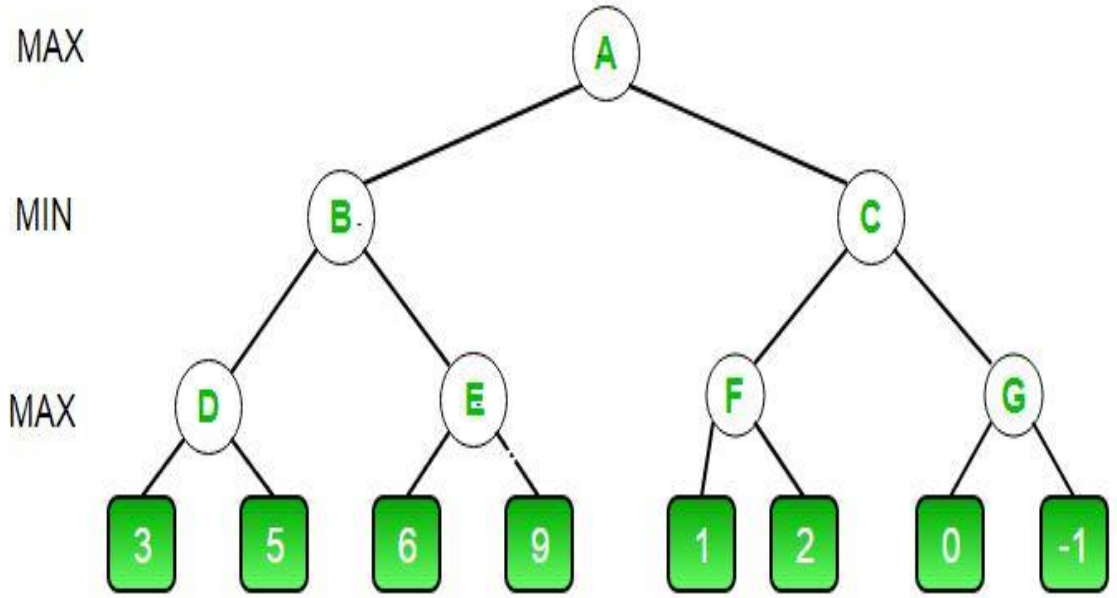
Alpha-Beta pruning

- Alpha-Beta pruning is not actually a new algorithm, but rather an optimization technique for the minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.
- Let’s define the parameters alpha and beta.

Alpha is the best value that the **maximizer** currently can guarantee at that level or above.
Beta is the best value that the **minimizer** currently can guarantee at that level or below.

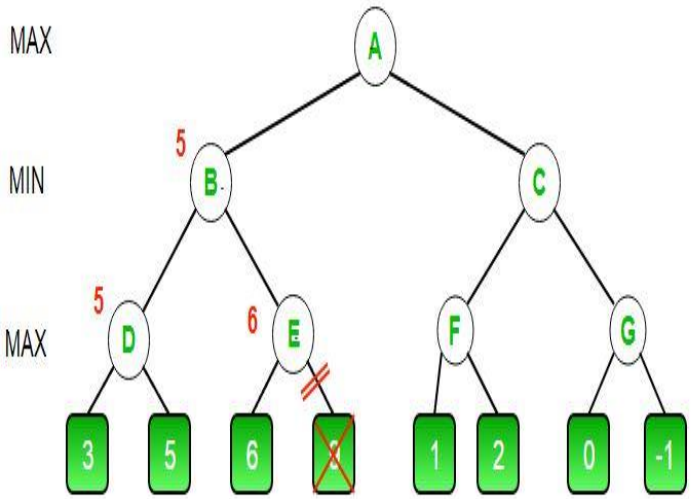
- Pseudocode :

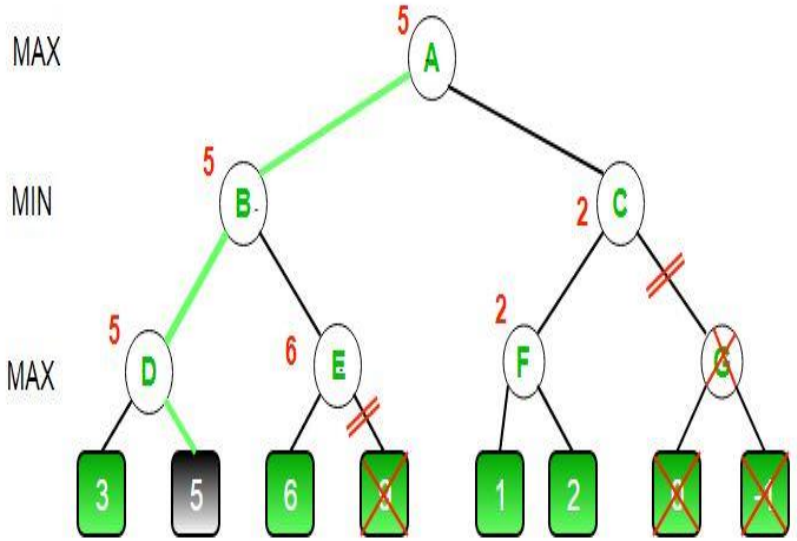
```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):
    if node is a leaf node :
        return value of the node
        if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal
    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```



- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the maximizer must choose max of **B** and **C**, so **A** calls **B** first
- At **B** it the minimizer must choose min of **D** and **E** and hence calls **D** first.
- At **D**, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at **D** is max(-INF, 3) which is 3.
- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.
- **D** now looks at its right child which returns a value of 5.At **D**, alpha = max(3, 5) which is 5. Now the value of node **D** is 5
- **D** returns a value of 5 to **B**. At **B**, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. **B** now calls **E** to see if he can get a lower value than 5.
- At **E** the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at **B** and that is what **B** passed down to **E**
- Now **E** looks at its left child which is 6. At **E**, alpha = max(-INF, 6) which is 6. Here the condition becomes true. beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and **E** returns 6 to **B**
- Note how it did not matter what the value of **E**‘s right child is. It could have been +INF or -INF, it still wouldn’t matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of **B**. This way we didn’t have to look at that 9 and hence saved computation time.
- **E** returns a value of 6 to **B**. At **B**, beta = min(5, 6) which is 5.The value of node **B** is also 5

- So far this is how our game tree looks. The 9 is crossed out because it was never computed.





B returns 5 to A. At A, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. A now calls C to see if it can get a higher value than 5.
At C, alpha = 5 and beta = +INF. C calls F
At F, alpha = 5 and beta = +INF. F looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
F looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
F returns a value of 2 to C. At C, beta = min(+INF, 2). The condition beta <= alpha becomes true as beta = 2 and alpha = 5. So it breaks and it does not even have to compute the entire sub-tree of G.
The intuition behind this break-off is that, at C the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose B. So why would the maximizer ever choose C and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub-tree.
C now returns a value of 2 to A. Therefore the best value at A is max(5, 2) which is a 5.
Hence the optimal value that the maximizer can get is 5
This is how our final game tree looks like. As you can see G has been crossed out as it was never computed.

Imperfect real-time decisions

- The minimax algorithm generates the entire game search space
- alpha–beta algorithm allows us to prune large parts of it. However, alpha–beta still has to search all the way to terminal states for at least a portion of the search space.
- To improve on this, terminate searches early based on a heuristic evaluation function.

  - Treats non-terminal nodes like leaves.

so, modify min-max or alpha-beta by;

- Replace utility function with heuristic function evaluation.
- Replace terminal test with a cutoff test.

- So, we will then have a heuristic minimax function for state s with a max depth of d:

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

➢ Evaluation functions :

- An evaluation function returns an estimate of the expected utility of the game from a given position
- just as the heuristic functions return an estimate of the distance to the goal.
- It should be clear that the performance of a game-playing program depends strongly on the quality of its evaluation function. An inaccurate evaluation function will guide an agent toward positions that turn out to be lost.
- Good evaluation function are a must, bad evaluation functions lose you the game

How exactly do we design good evaluation functions?

- First, the evaluation function should order the terminal states in the same way as the true utility function:

  states that are wins must evaluate better than draws, which in turn must be better than losses. Otherwise, an agent using the evaluation function might err even if it can see ahead all the way to the end of the game.

- Second, the computation must not take too long! (The whole point is to search faster.)
- Third, for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

How do evaluation functions work ?

- Most evaluation functions work by calculating various features of the state

for example:

in chess, we would have features for the number

of white pawns, black pawns, white queens, black queens,

and so on.

The features, taken together, define various categories or equivalence classes of states: the states in each category have the same values for all the features.

For example :

one category contains all two-pawn vs. one-pawn endgames. Any given category, generally speaking, will contain some states that lead to wins, some that lead to draws, and some that lead to losses.

- They calculate features off a state
- Defines categories or equivalence classes of the states
- Example : All 1 pawn vs 2 pawn states.
- Each category will win some, lose some, draw some
- Function figures out the ratio for each outcome.

For Example :

suppose our experience suggests that 72% of the states encountered in the two-pawns vs. one-pawn category lead to a win (utility+1); 20% to a loss (0), and 8% to a draw (1/2). Then a reasonable evaluation for states in the category is the **expected value**: $(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$.

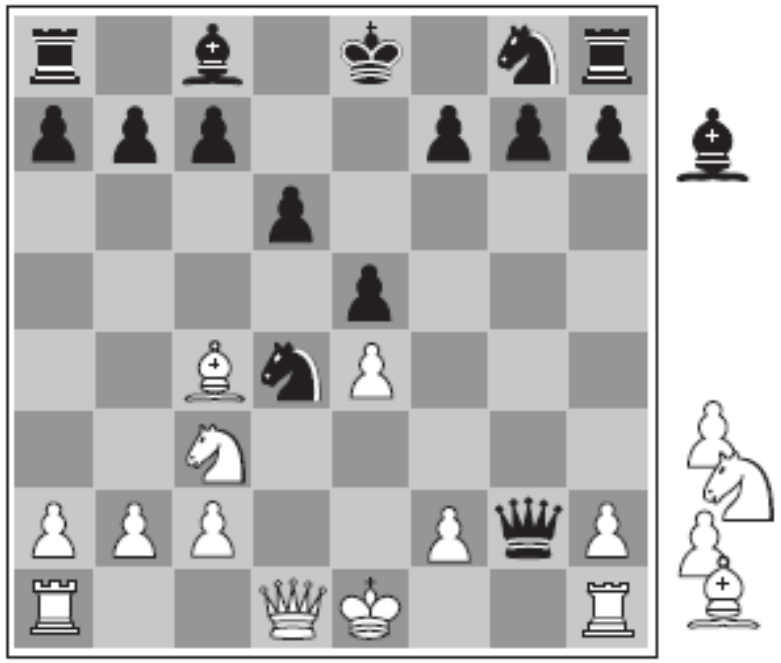
- In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities of winning.
- Instead, most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value.
- Mathematically, this kind of evaluation function is called a **weighted linear function** because it can be expressed as

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

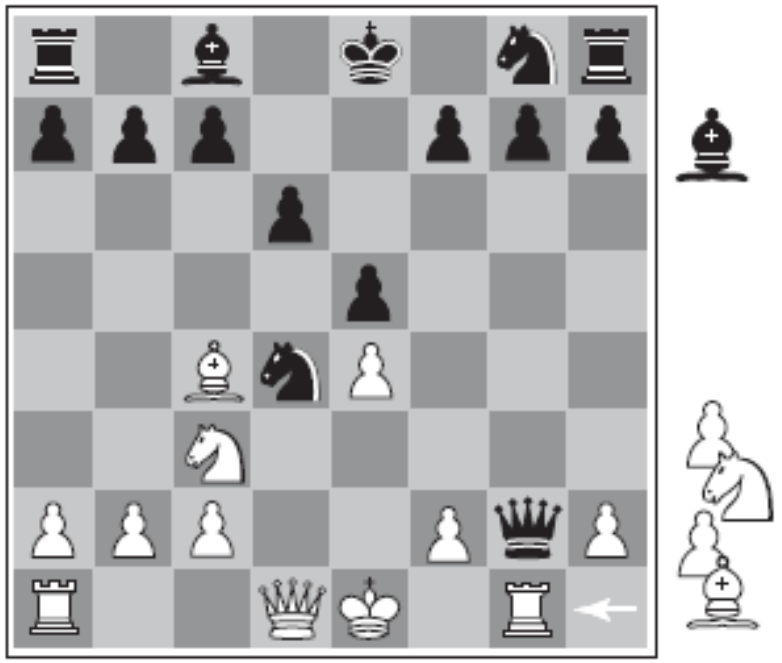
Where

each wi is a weight and each fi is a feature of the position.

For chess, the fi could be the numbers of each kind of piece on the board, and the wi could be the values of the pieces



(a) White to move



(b) White to move

Two chess positions that differ only in the position of the rook at lower right.
In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

Chance based games:
Games that include an Element of Chance

- Many games, such as dice tossing, have a rando element to reflect this unpredictability.
- The outcome of this games depending on the skills a well as luck.These are known as stochastic games.

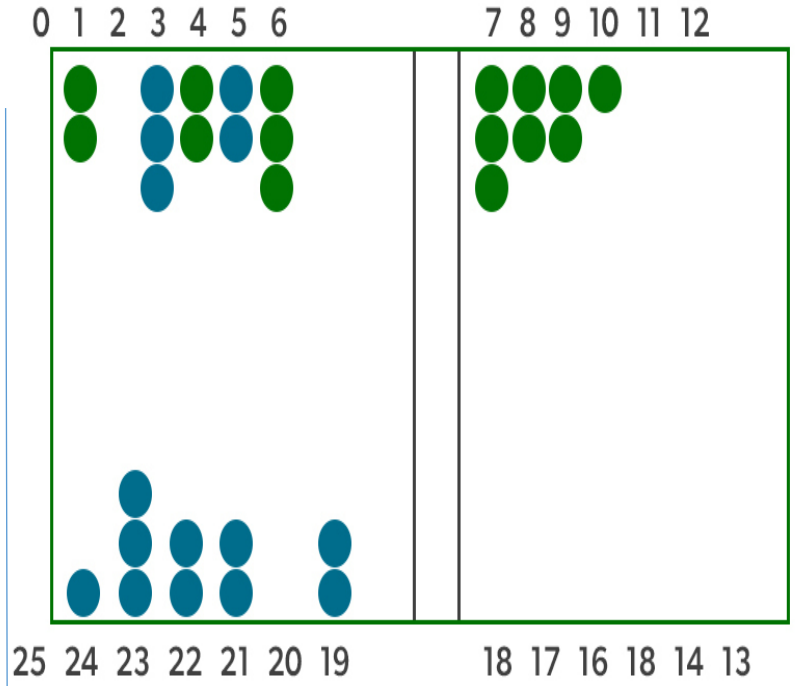
Examples are

Gambling game

Golf ball game

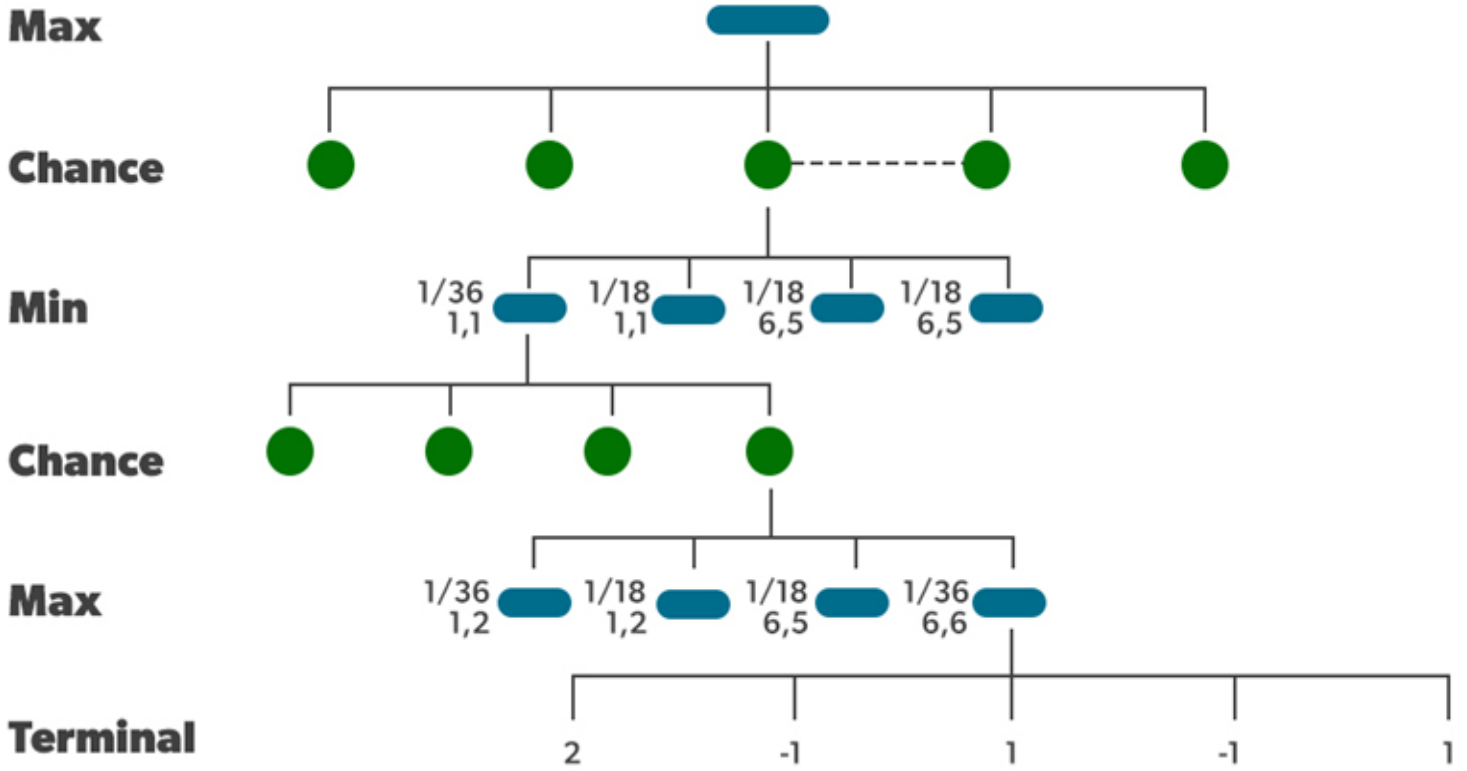
Backgammon etc

- Backgammon is a classic game that mixes skill ar.. luck.
- The legal moves are determined by rolling dice at the start of each player's turn white,
- The object of the game is to get all of one's pieces off the board as quickly as possible. green moves in a clockwise direction toward 25, while blue moves in a counter clockwise direction toward 0.



White has rolled a 6–5 and must pick between four valid moves:
(5–10,5–11),
(5–11,19–24),
(5–10,10–16), and
(5–11,11–16), where the notation (5–11,11–16) denotes moving one piece from position 5 to 11 and then another from 11 to 16.

- . In backgammon, in addition to M A X and M I N nodes, a game tree must include chance nodes. The figure below depicts chance nodes as circles. The possible dice rolls are indicated by the branches leading from each chance node; each branch is labelled with the roll and its probability.
- There are 36 different ways to roll two dice, each equally likely, yet there are only 21 distinct rolls because a 6–5 is the same as a 5–6. P (1–1) = 1/36 because each of the six doubles (1–1 through 6–6) has a probability of 1/36. Each of the other 15 rolls has a 1/18 chance of happening.



- As a result, we can generalize the deterministic minimax value to an expected-minimax value for games with chance nodes. Terminal nodes, MAX and MIN nodes (for which the dice roll is known), and MAX and MIN nodes (for which the dice roll is unknown) all function as before.
- We compute the expected value for chance nodes, which is the sum of all outcomes, weighted by the probability of each chance action.

$$\text{EXPECTIMINIMAX}\ (s) = \begin{cases} \text{UTILITY}\ (s) & \text{if TERMINAL-TEST}\ (s) \\ \max_a\ \text{EXPECTIMINIMAX}\ (\text{RESULT}(s,a)) & \text{if PLAYER}\ (s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\,\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

- where r is a possible dice roll (or other random events) and RESULT(s, r) denotes the same state as s, but with the addition that the dice roll's result is r.

State-of-the-Art Game Programs

- State-of-the-art (SOTA) DNNs(Deep Neural Networks) are the best models you can use for any particular task. A DNN can be identified as SOTA based on its accuracy, speed, or any other metric of interest.
- Deep learning models coupled with the right image data can be used to solve real-life problems that we come across every day,

  such as

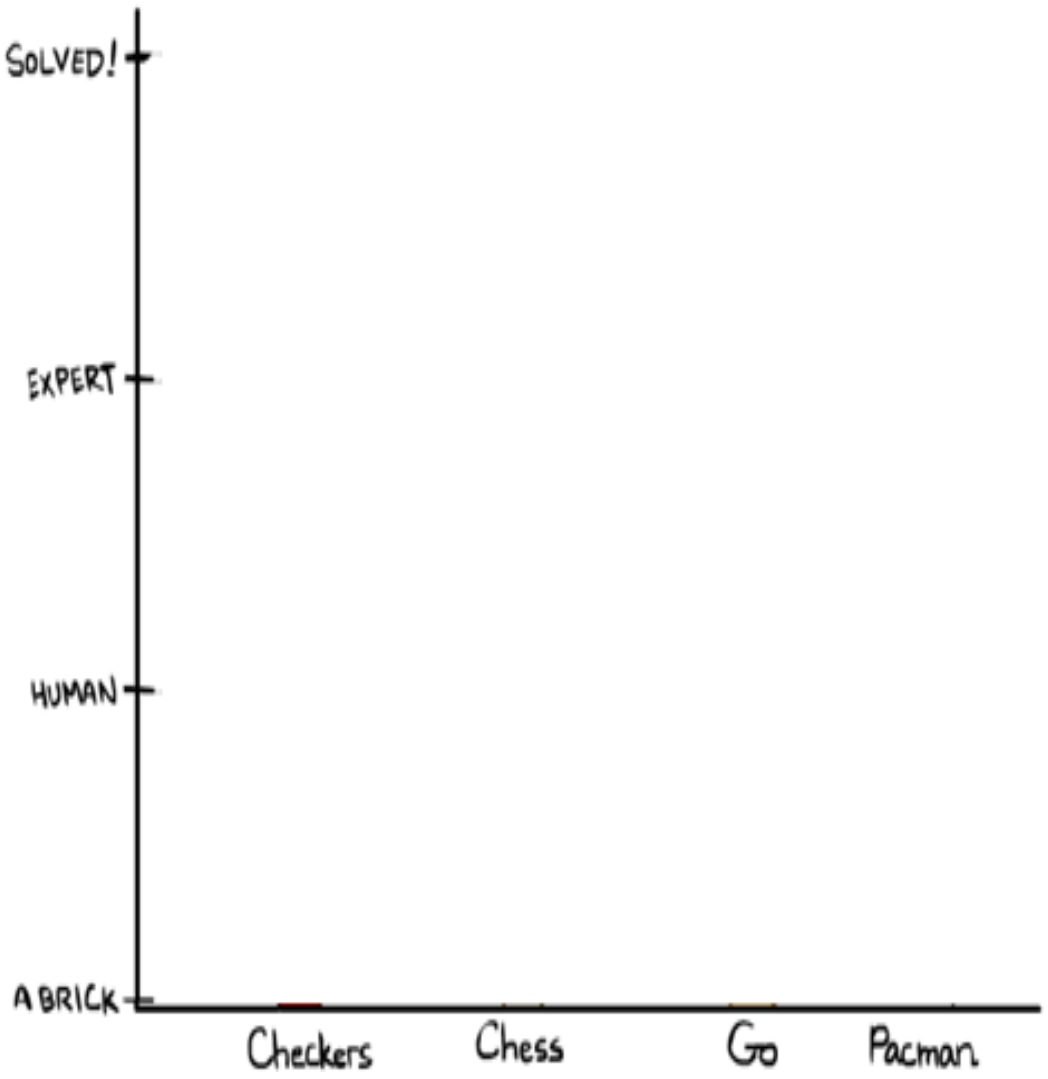
  medical image analysis,

  video conferencing, and autonomous driving.
- state-of-the-art game programs are fast, highly optimized machines that incorporate the latest engineering advances, but they aren’t much use for doing the shopping or driving off-road.
- racing and game-playing generate excitement and a steady stream of innovations that have been adopted by the wider community.

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go:** Human champions are now starting to be challenged by machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

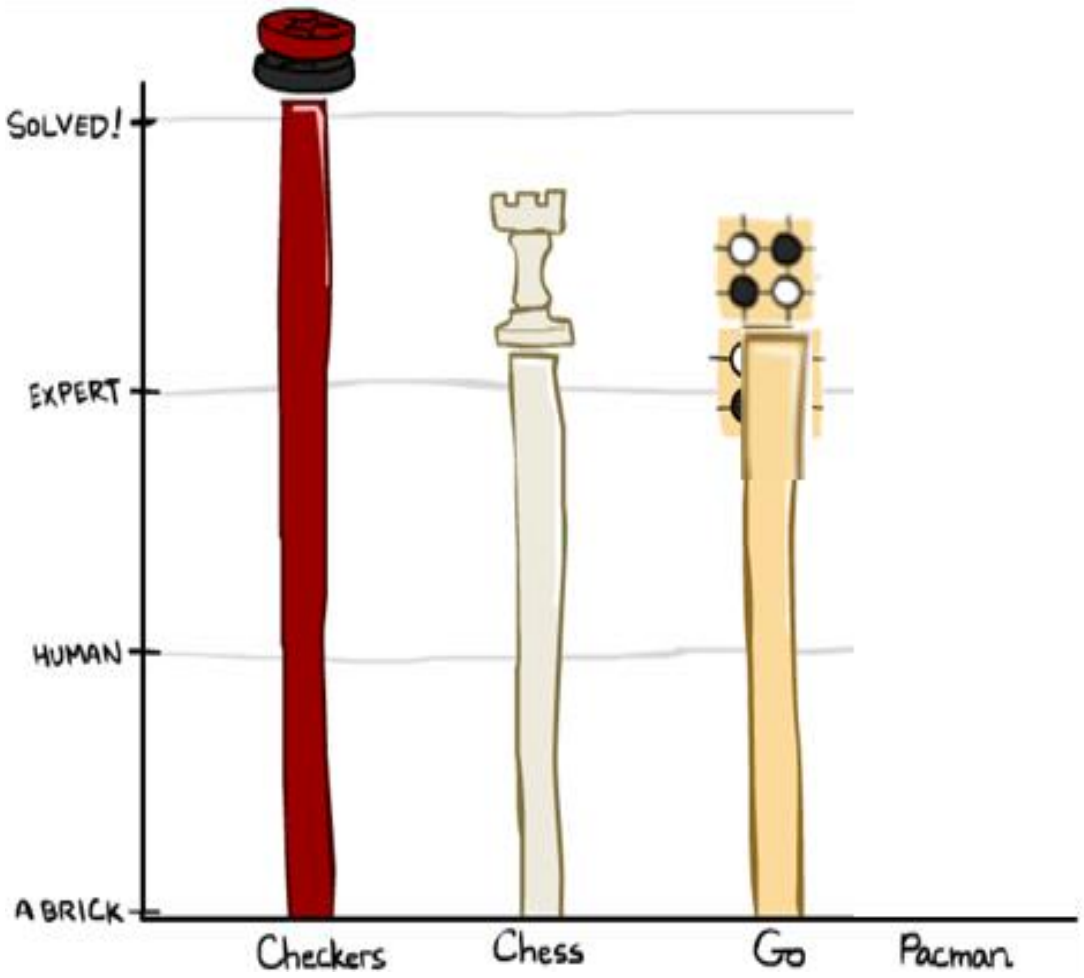


- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go: 2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.**

- **Pacman**



Constraint Satisfaction Problems

Defining Constraint Satisfaction Problems

- We have seen so many techniques like Local search, Adversarial search to solve different problems. The objective of every problem-solving technique is one, i.e., to find a solution to reach the goal.
- in adversarial search and local search, there were no constraints on the agents while solving the problems and reaching to its solutions.
- constraint satisfaction means *solving a problem under certain constraints or rules.*
- *Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem.* Such type of technique leads to a deeper understanding of the problem structure as well as its complexity.

- Constraint satisfaction depends on three components, namely:
- **X:** It is a set of variables.
- **D:** It is a set of domains where the variables reside. There is a specific domain for each variable.
- **C:** It is a set of constraints which are followed by the set of variables.

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

Solving Constraint Satisfaction Problems

The requirements to solve a constraint satisfaction problem (CSP) is:

- A state-space
- The notion of the solution.
- A state in state-space is defined by assigning values to some or all variables such as

$$\{X_1=v_1, X_2=v_2, \text{ and so on…}\}.$$

An assignment of values to a variable can be done in three ways:

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

Types of Domains in CSP

There are following two types of domains which are used by the variables :

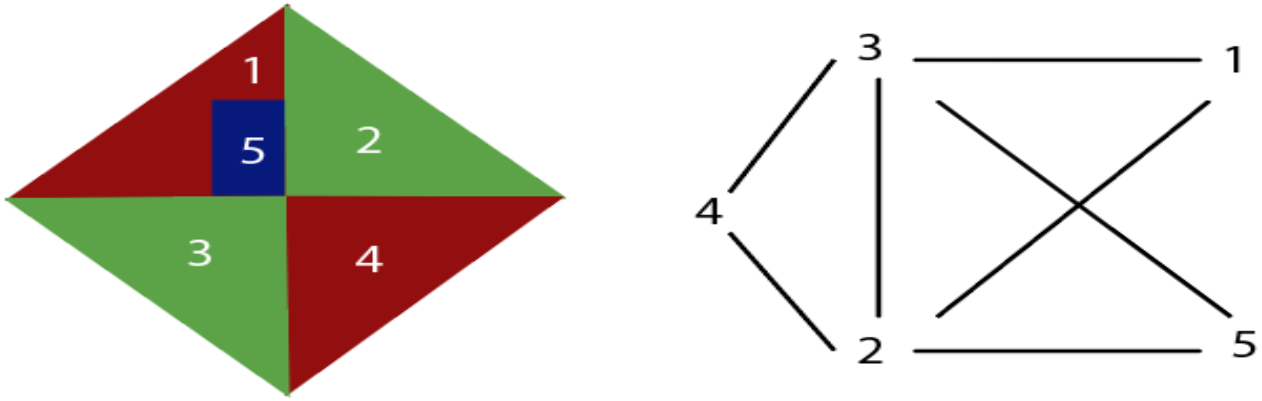
- **Discrete Domain:** It is an infinite domain which can have one state for multiple variables. **For example,** a start state can be allocated infinite times for each variable.
- **Finite Domain:** It is a finite domain which can have continuous states describing one domain for one specific variable.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

Constraint Types in CSP

- With respect to the variables, basically there are following types of constraints:
- **Unary Constraints:** It is the simplest type of constraints that restricts the value of a single variable.
- **Binary Constraints:** It is the constraint type which relates two variables. A value $\mathbf{x_2}$ will contain a value which lies between **x1** and **x3**.
- **Global Constraints:** It is the constraint type which involves an arbitrary number of variables.

CSP Problems

- Constraint satisfaction includes those problems which contains some constraints while solving the problem. CSP includes the following problems:
- **Graph Coloring:** The problem where the constraint is that no adjacent sides can have the same color.



Graph Coloring

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

SUDOKU

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

Puzzle

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

Solution

crypt-arithmetic problem

- In the crypt-arithmetic problem, some letters are used to assign digits to it. Like ten different letters are holding digit values from 0 to 9 to perform arithmetic operations correctly. There are two words are given and another word is given an answer of addition for those two words.
- As an example, we can say that two words ‘BASE’ and ‘BALL’, and the result is ‘GAMES’. Now if we try to add BASE and BALL by their symbolic digits, we will get the answer GAMES.

Input and Output

```
Input:
This algorithm will take three words.
             B A S E
             B A L L
           ----------
           G A M E S

Output:
It will show which letter holds which number from 0 – 9.
For this case it is like this.
```

Letter	A	B	E	G	L	M	S
Values	4	2	1	0	5	9	6

```
             B A S E                    2 4 6 1
             B A L L                    2 4 5 5
           ---------                  ---------
           G A M E S                  0 4 9 1 6
```

Backtracking Search

- **Backtracking** can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.
- Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.
- **Types of Backtracking Algorithm**

There are three types of problems in backtracking

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

- Backtracking search, a form of depth-first search, is commonly used for solving CSPs.
- Backtracking search: A depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution.
- If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.
- There is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

BACKTRACKING-SARCH keeps only a single representation of a state and alters that representation rather than creating a new ones.

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

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

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

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

SELECT-UNASSIGNED-VARIABLE

- Variable selection—fail-first
- Minimum-remaining-values (MRV) heuristic: The idea of choosing the variable with the fewest "legal" value.
- "most constrained variable" or "fail-first" heuristic, it picks a variable that is most likely to cause a failure soon thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.

ORDER-DOMAIN-VALUES

Value selection—fail-last

If we are trying to find all the solution to a problem (not just the first one), then the ordering does not matter.

Least-constraining-value heuristic: prefers the value that rules out the fewest choice for the neighboring variables in the constraint graph. (Try to leave the maximum flexibility for subsequent variable assignments.)

INFERENCE

forward checking: [One of the simplest forms of inference.] Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

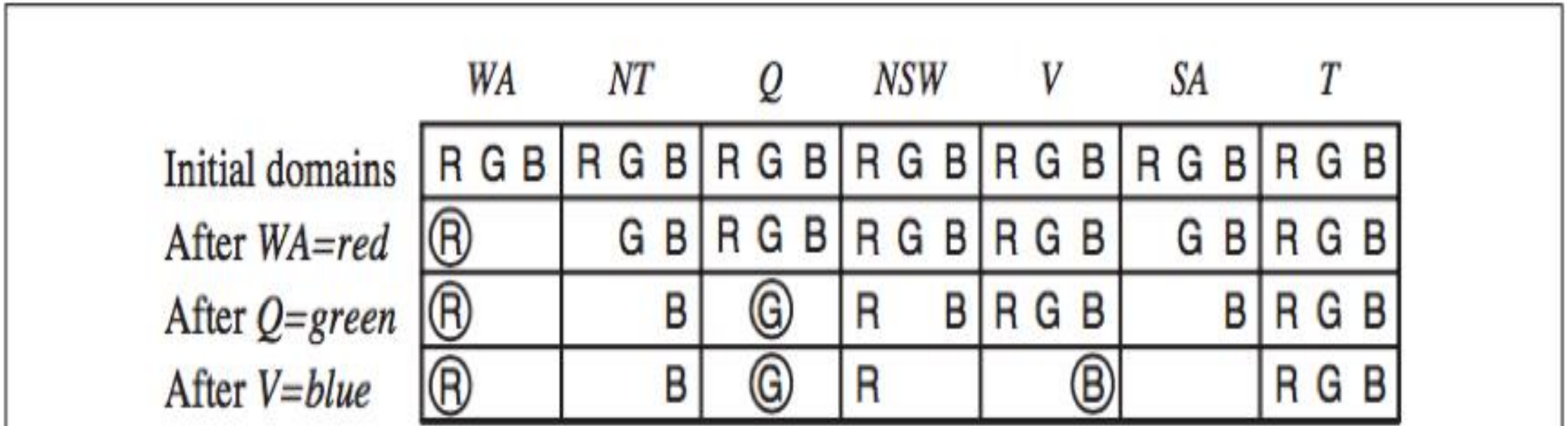


Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$ is assigned, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$ is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

Local search for CSPs

- In constraint satisfaction, **local search** is an incomplete method for finding a solution to a problem. It is based on iteratively improving an assignment of the variables until all constraints are satisfied.
- local search algorithms typically modify the value of a variable in an assignment at each step.

- Constraint weighting: a technique that can help concentrate the search on the important constraints.
- Each constraint is given a numeric weight Wi, initially all 1.
- At each step, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints.
- The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.
- Local search can be used in an online setting when the problem changes, this is particularly important in scheduling problems.

Idea:
– Consider the space of complete assignments of values to variables
(all possible worlds)
– Neighbours of a current node are similar variable assignments
– Move from one node to another according to a function that scores
how good each assignment is

Local Search Problem: Definition

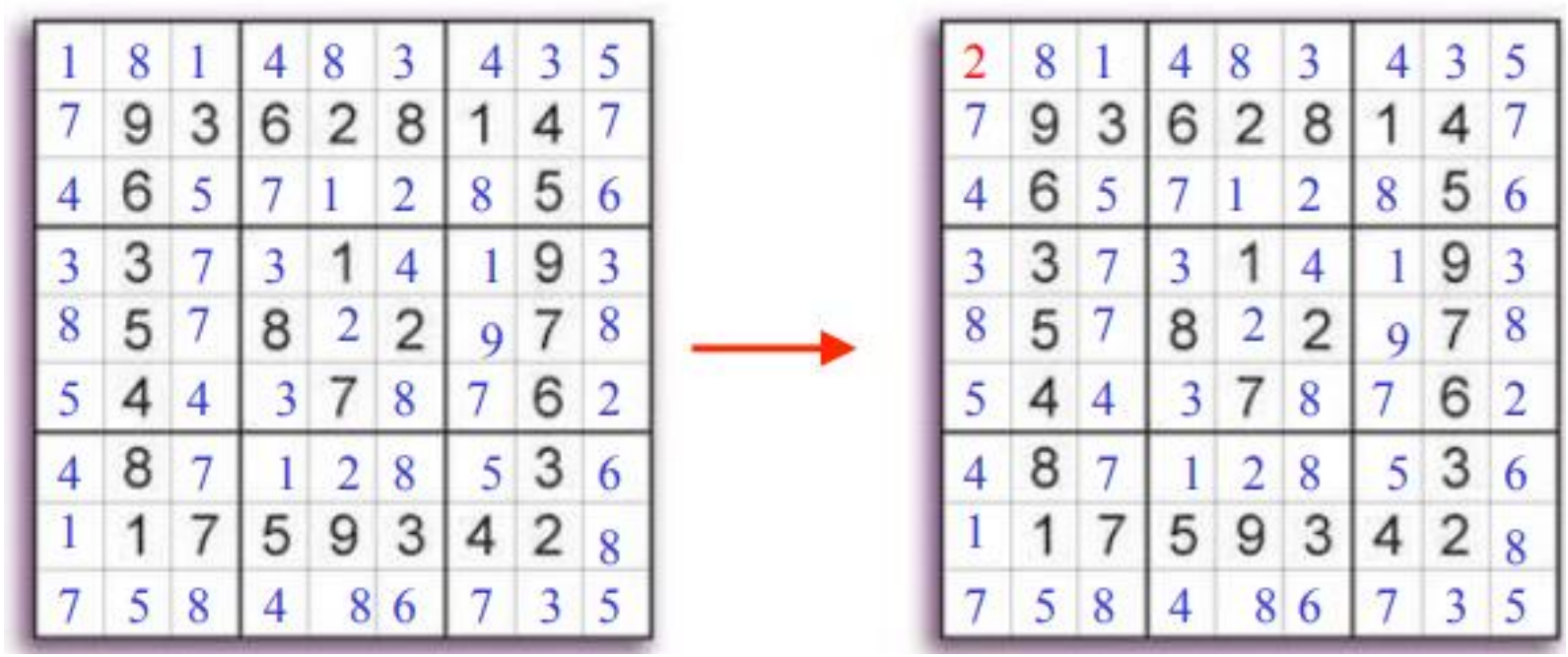
- Definition: A local search problem consists of a CSP , Neighbour relation and Scoring function
- CSP: a set of variables, domains for these variables, and constraints on their joint values. A node in the search space will be a complete assignment to all of the variables.
- Neighbour relation: an edge in the search space will exist when the neighbour relation holds between a pair of nodes.
- Scoring function: h(n), judges cost of a node (want to minimize) - E.g. the number of constraints violated in node n. - E.g. the cost of a state in an optimization context.

Example: Sudoku as a local search problem

- CSP: usual Sudoku CSP - One variable per cell;domains {1,…,9}; -

  Constraints: each number occurs once per row, per column, and

  per 3x3 box
- Neighbour relation: value of a single cell differs
- Scoring function: number of constraint violations



Constraint Networks

- A constraint network can be represented by a graph called a primal constraint graph or just a constraint graph, where each node represents a variable and the arcs connect all nodes whose variables are included in a constraint scope of the problem.
- The absence of an arc between two nodes indicates that there is no direct constraint

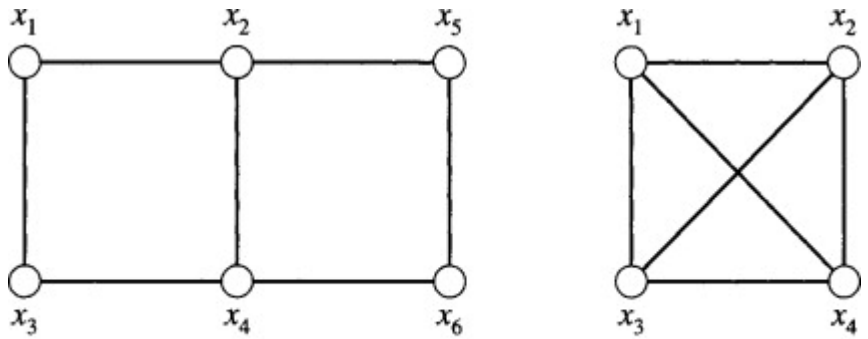


Figure (a) constraint graph associated with crossword puzzle, and
Figure (b) constraint graph of the 4-queens problem, which is complete.

A constraint network R consists of a finite set of variables X = {X1, …, Xn}, each associated with a domain of discrete values, D_1,D_2 …, Dn} and a set of constraints,

Constraints

In many domains, not all possible assignments of values to variables are permissible.

A hard constraint, or simply constraint,

specifies legal combinations of assignments of values to some of the variables.

A scope is a set of variables

A relation on a scope S is a function from assignments on S to {true , false}

it specifies whether each assignment is true or false.

A constraint c is a scope S and a relation on S.

A constraint is said to involve each of the variables in its scope.

A constraint can be evaluated on any assignment that extends its scope.

Consider constraint c on S.

Assignment A on S′, where $S \subseteq S'$ satisfies c if A, restricted to S, is mapped to true by the relation.

Otherwise, the constraint is violated by the assignment

- Formally, a constraint C_i has two parts:

(1) the subset of variables $S_i = \{X_i, \ldots, X_{i_{ji}}\}$ on which it is defined, called a *constraint-scope*

(2) a *relation*, R_i defined over $S_i: R_i \subseteq D_i \times \ldots \times D_{i_{ij}}$.

➢The relation denotes all compatible tuples of D_{Si} allowed by the constraint.

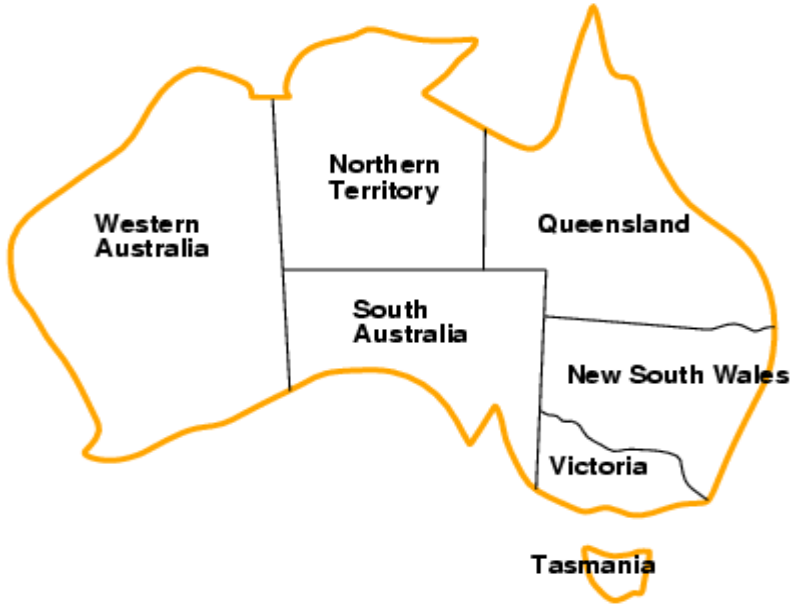
Thus a constraint network D_{Si} can be viewed as the triplet $R=(X, D, C)$. The *scheme* of a constraint network is its set of scopes, namely,

scheme (R) = $\{S_1, S_2, \ldots, S_t\}$,

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints

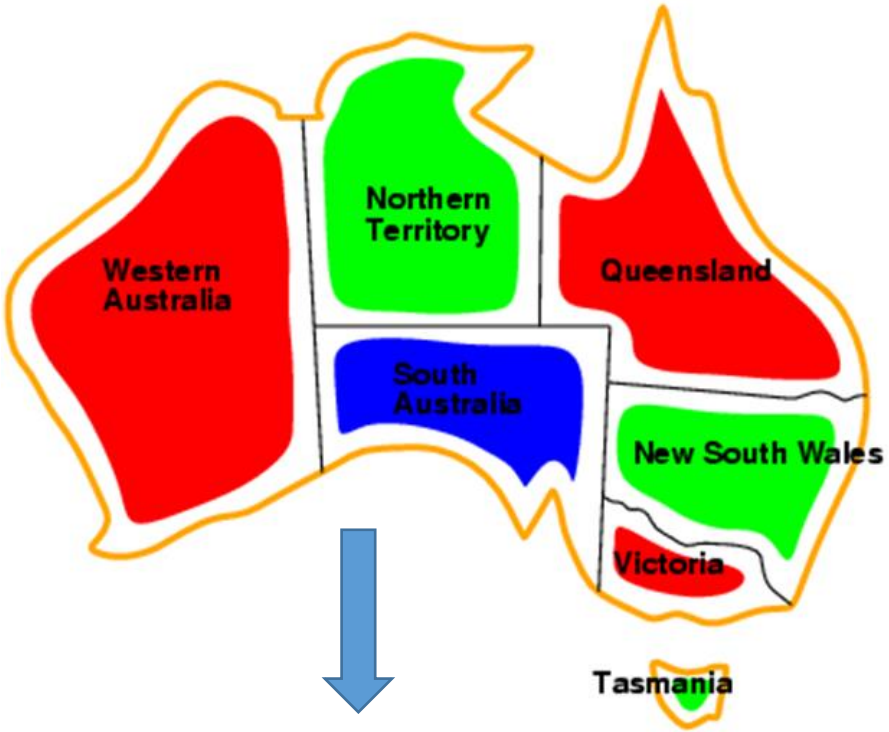
Example: Map-Coloring



Variables *WA, NT, Q, NSW, V, SA, T*
Domains D_i = {red,green,blue}
Constraints: adjacent regions must have different colors

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

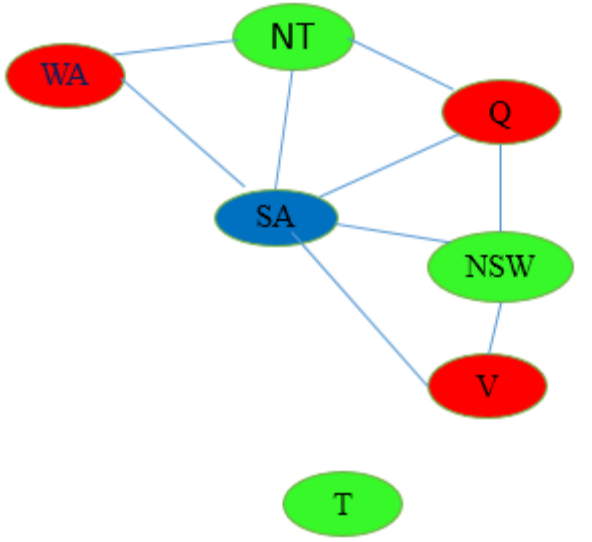
Map-Coloring



Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

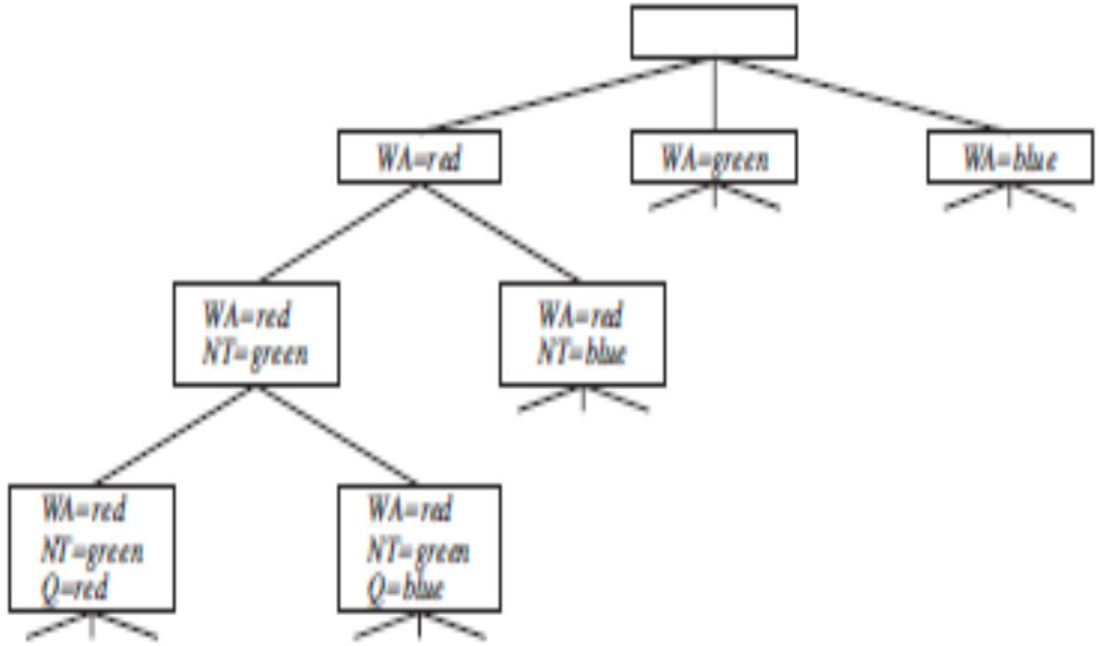
(b) The map-coloring problem represented as a constraint graph.

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



constraint Network

Search tree :



- **n-queen problem:** In n-queen problem, the constraint is that no queen should be placed either diagonally, in the same row or column.
- **Crossword:** In crossword problem, the constraint is that there should be the correct formation of the words, and it should be meaningful.

Logical Agents: Knowledge-based Agents

Solving CSPs Using Search

Generate-and-test algorithms assign values to all variables before checking the constraints.

individual constraints only involve a subset of the variables, some constraints can be tested before all of the variables have been assigned values.

If a partial assignment is inconsistent with a constraint, any total assignment that extends the partial assignment will also be inconsistent.

assignments A=1 and B=1 are inconsistent with the constraint A≠B

regardless of the values of the other variables.

If the variables A and B are assigned values first,

this inconsistency can be discovered before

any values are assigned to C, D, or E, saving a large amount of work.

Solving CSPs Using Search

construct a search space for the search strategies

- nodes are assignments of values to some subset of the variables.
- neighbors of a node n are obtained by selecting a variable Y that is not assigned in node n and
  by having a neighbor for each assignment of a value to Y that does not violate any constraint.

CSP with the variables A, B, and C each with domain (1,2,3,4)

constraints A<B and B<C

Search tree for the CSP

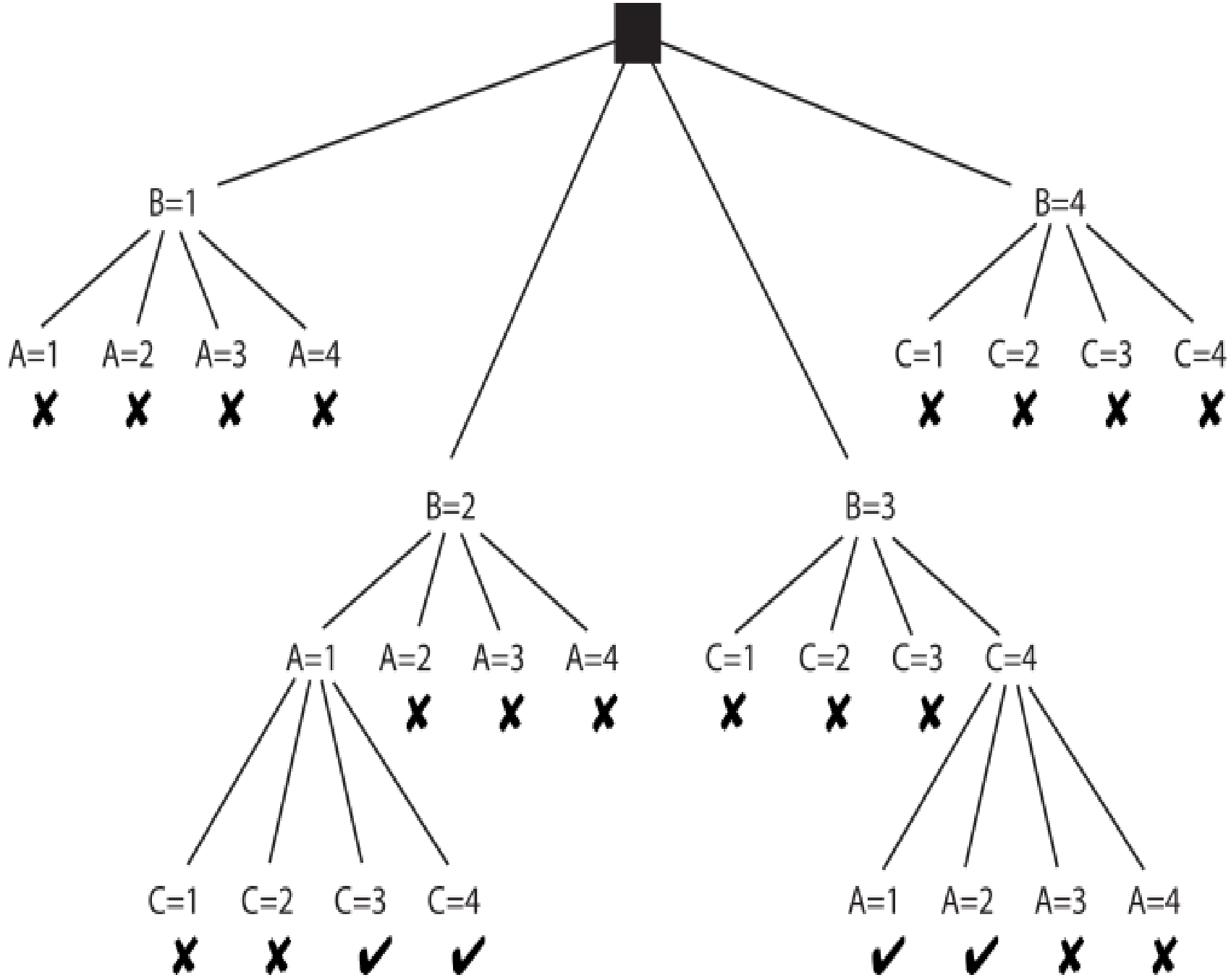
a node corresponds to all of the assignments from the root to that node.

The potential nodes that are pruned because they violate constraints are labeled with ✗.

CSP has four solutions.

inefficiency in back tracking

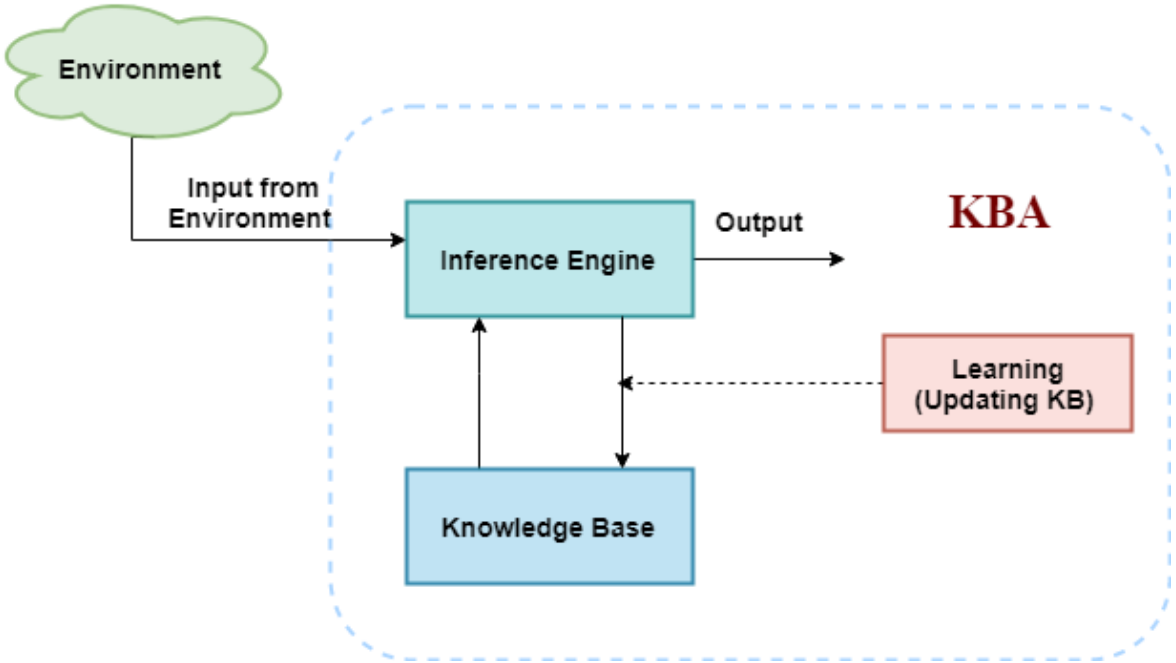
A = 4 inconsistency is rediscovered for different assignments of B, C.



Knowledge-based Agents

- An intelligent agent needs **knowledge** about the real world for taking decisions and reasoning to act efficiently.
- **Knowledge-based agents** are those agents who have the capability of maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions.
- These agents can represent the world with some formal representation and act intelligently.
- Knowledge-based agents are composed of two main parts:
  1) **Knowledge-base**
  2) **Inference system**
- A knowledge-based agent must able to do the following:
  - An agent should be able to represent states, actions, etc.
  - An agent Should be able to incorporate new percepts
  - An agent can update the internal representation of the world
  - An agent can deduce the internal representation of the world
  - An agent can deduce appropriate actions.

The architecture of knowledge-based agent



- **Knowledge base:** Knowledge-base is a central component of a knowledge-based agent, it is also known as KB.
- It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English).
- These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.

1) The knowledge-based agent (KBA) take input from the environment by perceiving the environment.
2) The input is taken by the inference engine of the agent and which also communicate with KB to decide as per the knowledge store in KB.
3)The learning element of KBA regularly updates the KB by learning new knowledge.

- Knowledge-base is required for updating knowledge for an agent to learn with experiences and take action as per the knowledge.

Inference system

- Inference means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information.
- Inference system generates new facts so that an agent can update the KB. An inference system works mainly in two rules which are given as:
  - **Forward chaining**
  - **Backward chaining**
- Operations Performed by KBA :

Following are three operations which are performed by KBA in order to show the intelligent behavior:

1) TELL: This operation tells the knowledge base what it perceives from the environment.

2) ASK: This operation asks the knowledge base what action it should perform.

3) Perform: It performs the selected action.

Various levels of **knowledge-based** agent

A knowledge-based agent can be viewed at different levels which are given below:

1. Knowledge level

Knowledge level is the first level of knowledge-based agent, and in this level, we need to specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.

2. Logical level:

At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs. At the logical level we can expect to the automated taxi agent to reach to the destination B.

3. Implementation level:

This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

Approaches to designing a knowledge-based agent

There are mainly two approaches to build a knowledge-based agent:

1. Declarative approach: We can create a knowledge-based agent by initializing with an empty knowledge base and telling the agent all the sentences with which we want to start with. This approach is called Declarative approach.

2. Procedural approach: In the procedural approach, we directly encode desired behavior as a program code. Which means we just need to write a program that already encodes the desired behavior or agent.

Propositional Logic

- 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:

  a) It is Sunday.

  b) The Sun rises from West (False proposition)

  c) 3+3= 7(False proposition)

  d) 5 is a prime number.

Following are some basic facts about propositional logic:

- 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:

1) Atomic Propositions

2) Compound propositions

- **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

  a) 2+2 is 4, it is an atomic proposition as it is a **true** fact.

  b) "The Sun is cold" is also a proposition as it is a **false** fact.

- **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

a) "It is raining today, and street is wet."

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.

Connective symbols	Word	Technical term	Example
∧	AND	Conjunction	A ∧ B
∨	OR	Disjunction	A ∨ B
→	Implies	Implication	A → B
⇔	If and only if	Biconditional	A⇔ B
¬ or ~	Not	Negation	¬ A or ¬ B

Truth Table

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

P	Q	P∧ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

For disjunction:

P	Q	P ∨ Q.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

P	Q	P→ Q
True	True	**True**
True	False	**False**
False	True	**True**
False	False	**True**

Truth table with three propositions:

P	Q	R	¬R	Pv Q	PvQ→¬R
True	True	True	False	True	False
True	True	False	True	True	True
True	False	True	False	True	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	True
False	False	True	False	False	True
False	False	False	True	False	True

Precedence of connectives :

Precedence	Operators
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Six Precedence	Biconditional

Properties of Operators

1) Commutativity:

- P∧ Q= Q ∧ P, or
- P ∨ Q = Q ∨ P.

2) Associativity:

- (P ∧ Q) ∧ R= P ∧ (Q ∧ R),
- (P ∨ Q) ∨ R= P ∨ (Q ∨ R)

3) Identity element:

- P ∧ True = P,
- P ∨ True= True.

4) Distributive:

- P∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R).
- P ∨ (Q ∧ R) = (P ∨ Q) ∧ (P ∨ R).

5) DE Morgan's Law:

- ¬ (P ∧ Q) = (¬P) ∨ (¬Q)
- ¬ (P ∨ Q) = (¬ P) ∧ (¬Q).

6)Double-negation elimination:

- ¬ (¬P) = P.

Limitations of Propositional logic:

•We cannot represent relations like ALL, some, or none with propositional logic. Example:

- **All the girls are intelligent.**
- **Some apples are sweet.**

•Propositional logic has limited expressive power.

•In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

Propositional Theorem Proving: Inference

- Entailment can be done through theorem proving : apply the inferences rules to the KB sentences to make a proof.
- Need to understand some additional concepts
  - logical equivalence
  - Validity
  - Satisfiability
  - **logical equivalence:**

  Sentence A and B are logically equivalent if true in the same set of models

  Written as : $A \equiv B$

  Example : $P \wedge Q = Q \wedge P$

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

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

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

Some inference rules do not function in both directions in the same way.
Example:

run Modus Ponens in the reverse direction to get $\alpha \Rightarrow \beta$ and α from β

➢ Validity :

- A Sentence is Valid if it is true in all models

  Example : P∨¬P is a valid
- **tautologies** : They are necessarily true

  A sentence true in all models is logically equivalent to true
- **leads to deduction theorem** :

  for any sentence A and B , A|=B iff the sentence (A→ B) is valid

Allow verification of A|=B by checking that (A→ B) in every model

Thus every valid implication sentence describes a legitimate inference

➢ Satisfiability:

a sentence is satisfiable if it is true, or satisfied by some model

The most well-known rule is known as **Modus Ponens** and is expressed as

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

Inferences in Proving Propositional Theorem:

- The notation signifies that the sentence may be deduced whenever any sentences of the type are supplied.

If(WumpusAhead ^WumpusAlive) } →{ Shoot } and (WumpusAhead ^ WumpusAlive) are both supplied, Shoot may be deduced.

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

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

WumpusAlive can be deduced from (WumpusAhead ^ WumpusAlive) ,

for example:

One may readily demonstrate that Modus Ponens and And-Elimination are sound once and for all by evaluating the potential truth values of α and β.

These principles may then be applied to each situation in which they apply, resulting in good conclusions without the necessity of enumerating models.

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

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

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

For contrapositives, logical equivalence yields $R_8 : \quad (\neg B_{1,1} \Rightarrow \neg (P_{1,2} \vee P_{2,1}))$

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

Finally, we use De Morgan's rule to arrive at the following conclusion:

$$R_{10} : \quad \neg P_{1,2} \wedge \neg P_{2,1}$$

That is to say, neither [1,2] nor [2,1] have a pit in them.
We found this proof by hand, but any of the search techniques may be used to produce a proof-like sequence of steps.
All we have to do now is define a proof problem:
Initial State: the starting point for knowledge.
Actions: the set of actions is made up of all the inference rules that have been applied to all the sentences that fit the inference rule's upper half.
Consequences: Adding the statement to the bottom part of the inference rule is the result of an action.
Objective: The objective is to arrive at a state that contains the phrase we are attempting to verify.
As a result, looking for proofs is a viable alternative to counting models.