# *INTRODUCTION TO ARTIFICIAL INTELLIGENCE*

## UNIT-II

### GAMES:

Optimal Decisions in Games
Alpha–Beta Pruning
Defining Constraint Satisfaction Problems
Constraint Propagation
Backtracking Search for CSPs
Knowledge-Based Agents
Propositional Logic

### PROPOSITIONAL THEOREM PROVING:

Inference and proofs
Proof by resolution
Horn clauses and definite clauses

## Optimal Decisions in Games

In **Artificial Intelligence (AI)**, **optimal decisions in games** refer to the strategies or choices made by an agent (or player) that maximize its chances of achieving the best possible outcome, given the game's rules, the environment, and the behaviour of other agents. The idea is to make decisions that lead to the best result, whether that involves maximizing a player's reward, minimizing their losses, or achieving some predefined objective.

**Key Concepts in Optimal Decision Making in AI Games**

1. **Game Theory**:
   - Game theory provides the foundation for analysing optimal decision-making in games, where multiple agents or players interact. It studies how rational players make decisions in strategic environments, where the outcome depends not just on their decisions but also on the decisions of others.
   - **Nash Equilibrium**: In many games, an optimal decision for an agent may involve finding a Nash Equilibrium, where no player can improve their outcome by changing their strategy, assuming the others keep theirs unchanged. This concept applies to games with multiple agents or players, such as poker, trading, or even complex strategic games.
   - **Zero-Sum Games**: In zero-sum games (e.g., chess), one player's gain is exactly another player's loss. In these games, finding optimal strategies often involves minimizing the opponent's gain while maximizing one's own.
2. **Minimax Algorithm**:
   - The **Minimax** algorithm is a common technique used for decision-making in two-player, zero-sum games (e.g., chess or tic-tac-toe). It involves minimizing the possible loss for a worst-case scenario. The idea is that each player assumes the opponent will play optimally, and they choose a move that minimizes the opponent's ability to win, while maximizing their own advantage.
   - The Minimax algorithm operates by building a **game tree**, which represents all possible moves and counter-moves. It recursively evaluates all potential outcomes and selects the move with the best guaranteed result.
   - **Alpha-Beta Pruning**: An optimization of Minimax, alpha-beta pruning reduces the number of nodes the algorithm needs to evaluate in the game tree, making the search for the optimal move

much faster. It prunes branches that do not affect the outcome, significantly improving performance.

3. **Reinforcement Learning (RL)**:
   - In games where agents need to learn optimal strategies over time, **reinforcement learning (RL)** plays a crucial role. In RL, agents learn by interacting with the environment and receiving rewards or penalties for their actions.
   - The goal is to learn a policy—a mapping from states to actions—that maximizes the expected cumulative reward. Through exploration and exploitation, the agent can discover optimal strategies. Algorithms like **Q-learning** and **Deep Q-Networks (DQN)** are commonly used in RL to make optimal decisions in complex environments.
   - **Deep Reinforcement Learning** is used for high-dimensional problems (like video games or robotics), where deep neural networks are employed to represent the value functions or policies.
4. **Monte Carlo Tree Search (MCTS)**:
   - **Monte Carlo Tree Search** is an algorithm used for decision-making in games with large state spaces and incomplete information, such as **Go** or **video games**. MCTS combines **random simulations** and **tree search** to explore the game tree, balancing between trying new moves (exploration) and leveraging known good moves (exploitation).
   - The algorithm builds a tree structure where each node represents a state, and it simulates the game from that state multiple times to estimate the outcome. The AI then selects the move that maximizes its chances of success.
5. **Imperfect Information and Probabilistic Decision-Making**:
   - In many games, like **Poker** or **Bluffing Games**, AI agents face **imperfect information**—meaning that they don't know all the details about the game state (such as hidden cards or the strategies of opponents). Here, the AI needs to make decisions probabilistically based on what it knows.
   - **Bayesian Reasoning** and **Markov Decision Processes (MDPs)** are often used to model such decision-making scenarios, where the AI updates its beliefs about the game state based on new information.
   - **Counterfactual Regret Minimization (CFR)** is a game-theoretic algorithm used in imperfect information games like Poker. CFR helps find strategies that perform well even when not all information is available, by simulating multiple possible scenarios.

**Applications of Optimal Decision-Making in AI**

1. **Board Games**:
   - In classical games like **chess**, **Go**, and **tic-tac-toe**, AI agents use algorithms like **Minimax**, **Alpha-Beta Pruning**, or **MCTS** to make optimal decisions. In **Go**, **AlphaGo** used a combination of **deep neural networks** and **MCTS** to achieve superhuman performance.
   - **Chess engines** like **Stockfish** also use a combination of search algorithms and evaluation functions to determine the best moves based on an in-depth understanding of the game state.
2. **Video Games**:
   - **Reinforcement Learning (RL)** has enabled AI to perform exceptionally well in video games, from **Atari 2600 games** to complex environments like **StarCraft II**. For example, **DeepMind's AlphaStar** used RL to learn how to play StarCraft II at a high level, competing against human professionals.
   - **OpenAI's Dota 2 AI (OpenAI Five)** used deep RL to play **Dota 2**, a complex multiplayer real-time strategy game, and was able to defeat human teams.
3. **Poker and Other Card Games**:
   - In games like **Poker**, where there is incomplete information, AI uses probabilistic models and game-theoretic strategies to make optimal decisions. **Libratus**, an AI developed by Carnegie Mellon, defeated professional poker players in **Heads-Up No-Limit Texas Hold'em** by using advanced game-theoretic methods like **counterfactual regret minimization**.
4. **Multi-Agent and Cooperative Games**:
   - In multi-agent systems (e.g., robotic teams or competitive markets), AI uses **game theory** and **multi-agent reinforcement learning (MARL)** to model and decide on optimal strategies for cooperation, competition, or negotiation.
   - **Cooperative games** involve agents working together to achieve a common goal, while **non-cooperative games** involve agents competing against each other to maximize their own individual payoff.
5. **Autonomous Systems and Robotics**:
   - AI makes optimal decisions in real-world systems such as **self-driving cars**, where the vehicle must continuously make decisions about navigation, obstacle avoidance, and route planning while interacting with other agents (e.g., pedestrians, other vehicles).

    - **Robotic teams** performing tasks like exploration, disaster response, or coordinated action (e.g., search-and-rescue operations) require optimal decision-making for cooperation, task allocation, and conflict resolution.

**Challenges in Optimal Decision-Making**

1. **Complexity of Game Environments**:
    - As the size of the game state or the number of possible actions increases, the search space grows exponentially, making it infeasible to compute optimal decisions using brute-force methods. Techniques like **pruning**, **heuristic evaluation**, and **deep learning** help mitigate this complexity.
2. **Real-Time Decision Making**:
    - In real-time games (e.g., real-time strategy games like **StarCraft II**), decisions need to be made rapidly, often under time constraints. AI agents must balance the exploration of new strategies with the need to act immediately.
3. **Incomplete Information**:
    - In games with incomplete information (e.g., **Poker**, **Strategic negotiations**), optimal decision-making requires handling uncertainty and making decisions based on limited or hidden data.
4. **Dynamic and Evolving Environments**:
    - In many games or real-world applications (e.g., **autonomous vehicles**, **trading systems**), the environment is dynamic and continuously evolving. AI must adapt to changing conditions in real time, often with imperfect feedback about the results of its actions.

**Alpha-Beta Pruning**

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

**The Importance of Alpha-Beta Pruning**

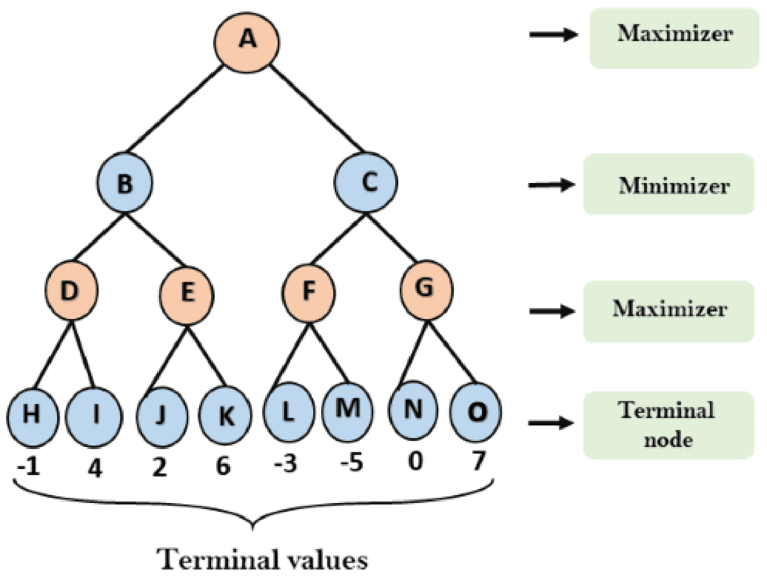
Alpha-Beta Pruning plays a pivotal role in optimizing the minimax algorithm, which is used for decision-making in two-player games. Its significance lies in its ability to drastically reduce the search space, allowing the algorithm to explore only the most promising branches of the game tree while discarding unfruitful ones.

**Applications of Alpha-Beta Pruning**

Alpha-Beta Pruning finds extensive use in game-playing AI, where computational efficiency is crucial. It's the technique that allows AI agents to make intelligent decisions within a reasonable time frame in games like chess, checkers, and even video games. Additionally, Alpha-Beta Pruning is applicable in decision trees used in fields such as finance, logistics, and optimization problems, where making the right decisions quickly is essential.

**Overview of the Minimax Algorithm**

Before we dive into Alpha-Beta Pruning, let's start with a brief overview of the Minimax algorithm. Minimax is a decision-making algorithm used in two-player games, where one player maximizes their outcome, and the other player aims to minimize it. It's a fundamental concept in game theory and artificial intelligence.



**Minimax in Two-Player Games**

Minimax is employed to determine optimal strategies in games like chess, checkers, and tic-tac-toe. In a two-player game, one player takes on the role of the maximizer, seeking the best move to maximize their chances of winning, while the other player acts as the minimizer, attempting to minimize the maximizer's chances.

## Understanding Alpha and Beta Values

Alpha and Beta values are key components of Alpha-Beta Pruning, and they play a crucial role in optimizing the minimax search algorithm. Let's explore these concepts:

### Alpha and Beta

- **Alpha:** Alpha represents the best score that the maximizing player has found so far in a particular branch of the game tree. It is the highest score the maximizing player can achieve up to this point. Essentially, it tracks the maximizer's best-known option.
- **Beta:** On the other hand, Beta represents the best score that the minimizing player has found in a specific branch. It is the lowest score the minimizing player can allow up to this point. Beta tracks the minimizer's best-known option.

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

- $\alpha >= \beta$

## Initial Values of Alpha and Beta

At the beginning of the Alpha-Beta Pruning process, we set the initial values of Alpha and Beta to represent the extremes of possible scores:

- **Alpha** is initially set to negative infinity, symbolizing the worst possible score for the maximizer.
- **Beta** is initially set to positive infinity, indicating the best possible score for the minimizer.

As the Alpha-Beta Pruning algorithm in AI progresses and evaluates nodes, Alpha and Beta are updated to reflect the best-known values for each player within a specific branch. These initial values are crucial to the pruning process, allowing us to efficiently identify promising branches and discard unproductive ones.

## Pseudo Code for Alpha-beta Pruning

```
function AlphaBeta(node, depth, alpha, beta, maximizingPlayer):

  if depth == 0 or node is a terminal node:
    return the heuristic value of node

  if maximizingPlayer:
    maxEval = -∞
    for each child of node:
      eval = AlphaBeta(child, depth - 1, alpha, beta, false)
      maxEval = max(maxEval, eval)
      alpha = max(alpha, eval)
      if beta <= alpha:
        break  # Beta cutoff (prune the branch)
    return maxEval
  else:
    minEval = ∞
    for each child of node:
      eval = AlphaBeta(child, depth - 1, alpha, beta, true)
      minEval = min(minEval, eval)
      beta = min(beta, eval)
      if beta <= alpha:
        break  # Alpha cutoff (prune the branch)
    return minEval
```

**Step by Step Breakdown of the Alpha-Beta Pruning Pseudo-Code:**

1. **Base Case**:
    - **Depth 0 or Terminal Node**: If the current depth is 0 or the node is terminal (no further moves possible), return the heuristic value. This provides a score for the current state.
2. **Maximizing Player's Turn**:
    - **Initialization**: Start with maxEval set to negative infinity, as we're trying to find the highest possible value.
    - **Iterate Over Children**: For each child node, call the AlphaBeta function recursively to evaluate it.
    - **Update maxEval**: After evaluating a child, update maxEval to be the maximum of its current value and the evaluated child's value.
    - **Update alpha**: Adjust alpha to the highest value found so far. If alpha exceeds or equals beta, prune the remaining children (cut off further exploration).

3. **Minimizing Player's Turn**:
    - **Initialization**: Start with minEval set to positive infinity, as we're trying to find the lowest possible value.
    - **Iterate Over Children**: For each child node, call the AlphaBeta function recursively to evaluate it.
    - **Update minEval**: After evaluating a child, update minEval to be the minimum of its current value and the evaluated child's value.
    - **Update beta**: Adjust beta to the lowest value found so far. If beta is less than or equal to alpha, prune the remaining children (cut off further exploration).
4. **Pruning Condition**:
    - **Cutoff**: If at any point, alpha (best for maximizer) becomes greater than or equal to beta (best for minimizer), prune the remaining child nodes as they can't improve the outcome. This is the core of Alpha-Beta pruning.
5. **Return**:

- **Best Value**: Finally, return maxEval for the maximizing player or minEval for the minimizing player, representing the best decision possible at this node given the current search depth and evaluated branches.

**Alpha Beta Pruning Algorithm with an Example**

In this example, we'll use a small game tree to demonstrate how Alpha-Beta Pruning works.

1. **Initialization:**

- We begin with the root node of the game tree.
- Initialize Alpha as negative infinity and Beta as positive infinity. This means that the maximizer's best score is initially set to negative infinity, and the minimizer's best score is initially set to positive infinity.

2. **Exploring the Tree:**

- We explore the tree depth-first, considering each node and its possible moves.
- As we evaluate each node, we update the Alpha and Beta values.

3. **Updating Alpha and Beta:**

- When we evaluate a maximizing (Max) node, we update Alpha to the maximum of its current value and the evaluation result. It represents the best-known option for the maximizing player.
- When we evaluate a minimizing (Min) node, we update Beta to the minimum of its current value and the evaluation result. It represents the best-known option for the minimizing player.

4. **Pruning:**

- The magic of Alpha-Beta Pruning happens when we determine that a branch is not worth exploring further. We do this by comparing Alpha and Beta.
- If at any point, Alpha becomes greater than or equal to Beta, it indicates that we've found a better option elsewhere. So, we can safely prune the rest of the branch because it won't affect the final result.

**Alpha Beta Pruning Example Step by Step:**
For a practical demonstration, let's consider a small game tree, such as a Tic-Tac-Toe scenario, where we can apply Alpha-Beta Pruning to see how it efficiently prunes unproductive branches. We'll show how the pruning process reduces the number of nodes to evaluate, saving time and computational resources. This practical example will make the Alpha-Beta Pruning concept clearer.
**Tic-Tac-Toe Example:**
Imagine a Tic-Tac-Toe game tree where the AI is the maximizing player, and the opponent is the minimizing player. The game tree looks like this:

```
Max's Turn (AI)
  |   | X
---------
  | O |
---------
X |   | O

Min's Turn (Opponent)
```

- We start at the root node (Max's turn) with Alpha as negative infinity and Beta as positive infinity.
- We evaluate the first child node, which represents Max's move to the top-right corner. The evaluation result is -10 (Max doesn't have a winning move here).
- We update Alpha to -10.

Now, as we move to the next child node (Min's turn), we find that the opponent has a choice between the bottom-left and bottom-right corners. We evaluate the first child node (bottom-left) and get an evaluation result of 5. We update Beta to 5.
At this point, we can prune the rest of the branch because Alpha (-10) is less than Beta (5). It means that, as the maximizing player, we already have a better option elsewhere. So, we don't need to explore further.

This pruning process continues as we traverse the tree. We quickly identify branches that won't lead to better results, and we eliminate the need to evaluate them fully. In this way, Alpha-Beta Pruning efficiently reduces the search space, making the AI's decision-making process faster and more resource-efficient.

**Example of Alpha-Beta Pruning with a More Complex Game Tree:**

Let's consider a more complex Tic-Tac-Toe game tree to showcase the efficiency of Alpha-Beta Pruning. In this tree, we'll apply Alpha-Beta Pruning to compare the number of nodes evaluated with and without pruning.

```
Max's Turn (AI)
  | X |
---------
  | O | O
---------
X |   |
```

Min's Turn (Opponent)

- We start at the root node (Max's turn) with Alpha as negative infinity and Beta as positive infinity.
- We evaluate the first child node, where Max places an 'X' in the top-center position. The evaluation result is -10 (Max doesn't have a winning move here). We update Alpha to -10.

As we move to the next child node (Min's turn), we find that the opponent has two available moves, which leads to a branching factor of 2:

- In the first branch, Min selects the bottom-left corner. The evaluation result is 5. We update Beta to 5.
- In the second branch, Min chooses the bottom-center position. The evaluation result is -5. We update Beta to -5.

At this point, we can prune the second branch (Min's move to the bottom-center) because Alpha (-10) is already greater than or equal to Beta (-5). This branch won't affect the final result, so we eliminate the need to explore it further.

**Practical Applications of Alpha-Beta Pruning**

Alpha-Beta Pruning is a fundamental technique with practical applications in various domains. Let's discuss some of these applications:

- **Chess Engines:** Alpha-Beta Pruning is widely used in chess engines to evaluate and compare various move sequences efficiently. It allows chess

AI to explore deep into the game tree and choose the best move while significantly reducing computational requirements.

- **Board Games:** Beyond chess, Alpha-Beta Pruning is applied to various board games like checkers, Othello, and Go, helping AI players make strategic decisions.
- **Video Games:** In video game development, Alpha-Beta Pruning is used to create AI opponents that can make intelligent moves in real-time strategy games, ensuring challenging gameplay.
- **Route Planning:** Alpha-Beta Pruning finds applications in pathfinding and route planning, such as GPS navigation systems, where it helps identify the most efficient routes.
- **Decision Trees:** Alpha-Beta Pruning can be employed in decision trees for decision-making processes. This is applicable in financial planning, logistics, and other decision-support systems.

**Challenges in Alpha-Beta Pruning**

Alpha-Beta Pruning is a powerful technique, but it's not without its challenges. Let's explore a couple of key challenges:

1. **Heuristic Evaluation Function:** Alpha-Beta Pruning relies on an accurate heuristic evaluation function to estimate the value of a game position. If the heuristic is poorly designed or inaccurate, it can lead to premature pruning or inefficient search. Developing a good heuristic is both an art and a science and requires domain-specific knowledge.
2. **Memory Consumption:** While Alpha-Beta Pruning improves the efficiency of the search process, it can still consume a significant amount of memory, especially in games with large branching factors. Balancing computational resources is a challenge in resource-constrained environments.
3. **Fail-Soft and Fail-Hard Issues:** In some cases, if the heuristic function underestimates or overestimates a position, it can lead to unexpected outcomes in the pruning process. Handling these "fail-soft" (underestimate) and "fail-hard" (overestimate) scenarios can be challenging.

**Situations Where Alpha-Beta Pruning May Not Perform Optimally:**

While Alpha-Beta Pruning is highly effective in many scenarios, there are situations where it may not perform optimally:

1. **Games with Uncertainty:** In games involving chance elements, like poker or backgammon, Alpha-Beta Pruning may not be as effective due to the difficulty of accurately estimating probabilities.

2. **Complex Heuristics:** Games with complex or poorly understood evaluation functions can pose challenges for Alpha-Beta Pruning. The effectiveness of the pruning depends on the quality of the heuristic.
3. **Non-Admissible Heuristics:** If the heuristic used is inadmissible (overestimates the true cost), Alpha-Beta Pruning can lead to suboptimal or incorrect decisions.
4. **Parallel Execution:** In distributed or parallel computing environments, coordinating and synchronizing Alpha-Beta Pruning can be complex, particularly in real-time applications.

In such scenarios, other techniques or variations of Alpha-Beta Pruning, such as Monte Carlo Tree Search (MCTS) or probabilistic approaches, may be more suitable. Understanding the limitations of Alpha-Beta Pruning is essential for making informed decisions about its application in AI systems.

## DEFINING CONSTRAINT SATISFACTION PROBLEMS

Constraint Satisfaction Problems (CSP) represent a class of problems where the goal is to find a solution that satisfies a set of constraints. These problems are commonly encountered in fields like scheduling, planning, resource allocation, and configuration.

A **Constraint Satisfaction Problem** is a mathematical problem where the solution must meet a number of constraints. In a CSP, the objective is to assign values to variables such that all the constraints are satisfied. CSPs are used extensively in artificial intelligence for decision-making problems where resources must be managed or arranged within strict guidelines.

**Common applications of CSPs include:**

- **Scheduling:** Assigning resources like employees or equipment while respecting time and availability constraints.
- **Planning:** Organizing tasks with specific deadlines or sequences.
- **Resource Allocation:** Distributing resources efficiently without overuse.

**Components of Constraint Satisfaction Problems**

CSPs are composed of three key elements:

1. **Variables:** The things that need to be determined are variables. Variables in a CSP are the objects that must have values assigned to them in order to satisfy a particular set of constraints. Boolean, integer, and categorical variables are just a few examples of the various types of variables, for instance, could stand in for the many puzzle cells that need to be filled with numbers in a sudoku puzzle.
2. **Domains:** The range of potential values that a variable can have is represented by domains. Depending on the issue, a domain may be finite

or limitless. For instance, in Sudoku, the set of numbers from 1 to 9 can serve as the domain of a variable representing a problem cell.

3. **Constraints:** The guidelines that control how variables relate to one another are known as constraints. Constraints in a CSP define the ranges of possible values for variables. Unary constraints, binary constraints, and higher-order constraints are only a few examples of the various sorts of constraints. For instance, in a sudoku problem, the restrictions might be that each row, column, and 3×3 box can only have one instance of each number from 1 to 9.

**Types of Constraint Satisfaction Problems**

CSPs can be classified into different types based on their constraints and problem characteristics:

1. **Binary CSPs**: In these problems, each constraint involves only two variables. For example, in a scheduling problem, the constraint could specify that task A must be completed before task B.
2. **Non-Binary CSPs**: These problems have constraints that involve more than two variables. For instance, in a seating arrangement problem, a constraint could state that three people cannot sit next to each other.
3. **Hard and Soft Constraints**: Hard constraints must be strictly satisfied, while soft constraints can be violated, but at a certain cost. This distinction is often used in real-world applications where not all constraints are equally important.

## Representation of Constraint Satisfaction Problems (CSP)

In **Constraint Satisfaction Problems (CSP)**, the solution process involves the interaction of variables, domains, and constraints. Below is a structured representation of how CSP is formulated:

1. **Finite Set of Variables** (V1,V2,…,Vn)($V1,V2,…,Vn$)**:**
   The problem consists of a set of variables, each of which needs to be assigned a value that satisfies the given constraints.
2. **Non-Empty Domain for Each Variable** (D1,D2,…,Dn)($D1,D2,…,Dn$)**:**
   Each variable has a domain—a set of possible values that it can take. For example, in a Sudoku puzzle, the domain could be the numbers 1 to 9 for each cell.
3. **Finite Set of Constraints** (C1,C2,…,Cm)($C1,C2,…,Cm$)**:**
   Constraints restrict the possible values that variables can take. Each constraint defines a rule or relationship between variables.
4. **Constraint Representation:**
   Each constraint Ci$Ci$ is represented as a pair **<scope, relation>**, where:

- **Scope:** The set of variables involved in the constraint.
- **Relation:** A list of valid combinations of variable values that satisfy the constraint.

5. **Example:**
   Let's say you have two variables V1 and V2. A possible constraint could be V1≠V2, which means the values assigned to these variables must not be equal.
   - **Detailed Explanation:**
     - **Scope:** The variables V1 and V2.
     - **Relation:** A list of valid value combinations where *V*1 is not equal to *V*2.

Some relations might include explicit combinations, while others may rely on abstract relations that are tested for validity dynamically.

## CSP Algorithms: Solving Constraint Satisfaction Problems Efficiently

**Constraint Satisfaction Problems (CSPs)** rely on various algorithms to explore and optimize the search space, ensuring that solutions meet the specified constraints. Here's a breakdown of the most commonly used CSP algorithms:

### 1. Backtracking Algorithm

The **backtracking algorithm** is a depth-first search method used to systematically explore possible solutions in CSPs. It operates by assigning values to variables and backtracks if any assignment violates a constraint.

**How it works:**

- The algorithm selects a variable and assigns it a value.
- It recursively assigns values to subsequent variables.
- If a conflict arises (i.e., a variable cannot be assigned a valid value), the algorithm backtracks to the previous variable and tries a different value.
- The process continues until either a valid solution is found or all possibilities have been exhausted.

This method is widely used due to its simplicity but can be inefficient for large problems with many variables.

### 2. Forward-Checking Algorithm

The **forward-checking algorithm** is an enhancement of the backtracking algorithm that aims to reduce the search space by applying **local consistency** checks.

**How it works:**

- For each unassigned variable, the algorithm keeps track of remaining valid values.

- Once a variable is assigned a value, local constraints are applied to neighbouring variables, eliminating inconsistent values from their domains.
- If a neighbor has no valid values left after forward-checking, the algorithm backtracks.

This method is more efficient than pure backtracking because it prevents some conflicts before they happen, reducing unnecessary computations.

**3. Constraint Propagation Algorithms**

**Constraint propagation** algorithms further reduce the search space by enforcing **local consistency** across all variables.

**How it works:**

- Constraints are propagated between related variables.
- Inconsistent values are eliminated from variable domains by leveraging information gained from other variables.
- These algorithms refine the search space by making **inferences**, removing values that would lead to conflicts.

Constraint propagation is commonly used in conjunction with other CSP algorithms, such as **backtracking**, to increase efficiency by narrowing down the solution space early in the search process.

**Solving Sudoku with Constraint Satisfaction Problem (CSP) Algorithms**

Sudoku is a perfect example of a CSP that can be solved using backtracking, forward-checking, and constraint propagation algorithms. Here's a step-by-step guide to solving a Sudoku puzzle using CSP principles:

1. **Define the Problem**: In Sudoku, the variables are the cells of the grid, the domains are the numbers 1 through 9, and the constraints are that no two cells in the same row, column, or 3×3 subgrid can have the same value.
2. **Create the CSP Solver Class**: A Python class is created to represent the Sudoku puzzle and handle the solving process. This class will include methods for assigning values to variables and checking constraints.
3. **Implement Helper Functions for Backtracking**: Backtracking is used to try different assignments of values to variables, and helper functions are implemented to ensure that constraints are satisfied at each step.
4. **Define Variables, Domains, and Constraints**: The variables are the cells of the grid, the domains are the possible values (1-9), and the constraints ensure that no two cells in the same row, column, or subgrid have the same value.

5. **Solve the Sudoku Puzzle**: The puzzle is solved by applying CSP techniques like backtracking and forward-checking. These techniques ensure that the solution satisfies all Sudoku constraints.

```
class SudokuSolver:
  def __init__(self, grid):
    self.grid = grid
  def solve(self):
    # Backtracking logic to solve the puzzle
    pass
  def is_valid(self, row, col, num):
    # Check row, column, and subgrid constraints
    pass
# Initialize the Sudoku grid and solve the puzzle
sudoku_grid = [[5, 3, 0, 0, 7, 0, 0, 0, 0], ...]
solver = SudokuSolver(sudoku_grid)
solver.solve()
```

## Applications of CSPs in AI

CSPs are widely used across different fields and industries due to their flexibility and capability of solving real-world problems efficiently. Some notable applications include:

1. **Scheduling**: CSPs are extensively used in scheduling problems, such as employee shifts, flight schedules, and course timetabling. The goal is to allocate tasks or resources while respecting constraints like time, availability, and precedence.
2. **Puzzle Solving**: Many logic puzzles, such as Sudoku, crosswords, and the N-Queens problem, can be formulated as CSPs, where the constraints ensure the puzzle rules are followed.
3. **Configuration Problems**: In product design or software configuration, CSPs help in choosing the right components or settings based on given requirements and restrictions. For example, in configuring a computer system, constraints could specify that certain components are incompatible with others.

4. **Robotics and Planning**: CSPs are used in pathfinding and task planning for autonomous agents and robots. For instance, a robot might need to navigate an environment while avoiding obstacles and minimizing energy consumption, which can be modeled as a CSP.
5. **Natural Language Processing (NLP)**: In NLP, CSPs can be applied to tasks like sentence parsing, where the goal is to find a valid syntactic structure based on grammar rules (constraints).

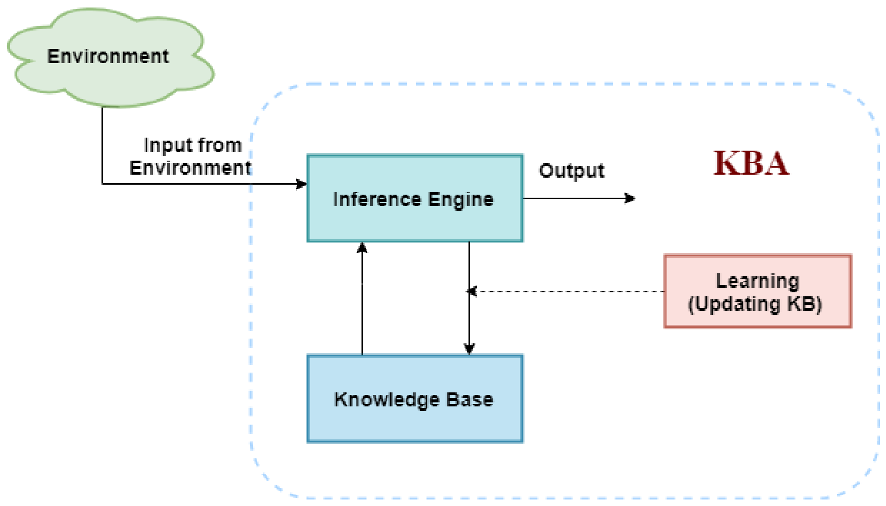
## KNOWLEDGE-BASED AGENT IN ARTIFICIAL INTELLIGENCE

- 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:
    - **Knowledge-base and**
    - **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:



The above diagram is representing a generalized architecture for a knowledge-based agent. The knowledge-based agent (KBA) take input from the environment by perceiving the environment. 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. The learning element of KBA regularly updates the KB by learning new knowledge.

**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.

**Why use a knowledge base?**

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 behaviour:

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.

**A generic knowledge-based agent:**

Following is 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 takes percept as input and returns an action as output. The agent maintains the knowledge base, KB, and it initially has some background knowledge of the real world. It also has a counter to indicate the time for the whole process, and this counter is initialized with zero.
Each time when the function is called, it performs its three operations:

- Firstly it TELLs the KB what it perceives.
- Secondly, it asks KB what action it should take
- Third agent program TELLS the KB that which action was chosen.

The **MAKE-PERCEPT-SENTENCE** generates a sentence as setting that the agent perceived the given percept at the given time.
The **MAKE-ACTION-QUERY** generates a sentence to ask which action should be done at the current time.
**MAKE-ACTION-SENTENCE** generates a sentence which asserts that the chosen action was executed.

## 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. **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. **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.

However, in the real world, a successful agent can be built by combining both declarative and procedural approaches, and declarative knowledge can often be compiled into more efficient procedural code.

## PROPOSITIONAL LOGIC

**Propositional Logic (PL)** is a branch of logic that focuses on **statements (propositions)** that can be **either true or false**. It is also known as **Boolean logic** since the truth values are binary—either **True (1)** or **False (0)**.

In AI, propositional logic forms the **foundation for logical reasoning**, allowing systems to represent **facts** and **rules** about a problem domain. These rules help the system infer new information or make decisions based on the given inputs.

Propositional logic simplifies **knowledge representation** by breaking down reasoning into **atomic statements** or **propositions**. For example, an AI system used in home automation might have propositions such as:

- **P**: “The light is on.”
- **Q**: “The window is open.”

Using **logical connectives**, the system can combine these propositions to represent more complex statements like:
“If the light is on and the window is open, turn off the light.”

By using propositional logic, AI systems can **reason** effectively and perform tasks like **automated decision-making, knowledge representation**, and **game playing**.

**Basic Facts About Propositional Logic**

**1. Propositions are Declarative Statements:**

- In **propositional logic**, each statement, known as a **proposition**, is either **True** or **False**.
  Example:
- **P**: "It is raining." (True or False)
- **Q**: "The heater is on." (True or False)

**2. Atomic Propositions:**

- These are **simple, indivisible statements** that cannot be broken down further. Each atomic proposition represents a basic fact or condition. Example: "The door is closed."

**3. Compound Propositions:**

- **Multiple atomic propositions** can be combined using **logical connectives** (like AND, OR, NOT) to create **compound propositions**. Example: "The door is closed AND the heater is on."

**4. Binary Truth Values:**

- Every proposition has a **binary truth value**: it can only be **True (1)** or **False (0)**. There are no intermediate states. This simplicity makes propositional logic ideal for **clear-cut decisions**.

**5. Logical Connectives Combine Propositions:**

- Logical connectives such as **AND, OR, NOT, IF-THEN,** and **IF AND ONLY IF** allow us to create more complex propositions from simple ones.

**Syntax of Propositional Logic**

The **syntax** of propositional logic defines the **rules for creating valid propositions**. In propositional logic, we combine **atomic propositions** using **logical connectives** to form more complex statements, known as **compound propositions**.

**Building Blocks of Propositional Logic Syntax**

1. **Atomic Propositions:**
   - These are **basic statements** that represent individual facts or conditions.
     Example:
   - **P**: "It is raining."
   - **Q**: "The heater is on."
2. **Logical Connectives:**
   - Connectives are used to combine atomic propositions to form **compound propositions**.

- **AND ( ∧ )**: True if both propositions are true.
- **OR ( ∨ )**: True if at least one proposition is true.
- **NOT ( ¬ )**: Negates the truth value of a proposition.
- **IF-THEN ( → )**: True unless the first proposition is true and the second is false.
- **IF AND ONLY IF ( ↔ )**: True if both propositions have the same truth value.

3. **Compound Propositions:**
    - These are **more complex statements** formed by connecting atomic propositions using logical connectives. Example:
    - "If it is raining and the heater is on, then the room will be warm." This can be written in **propositional logic syntax** as: $(P \land Q) \rightarrow R$

Where:

- **P**: "It is raining."
- **Q**: "The heater is on."
- **R**: "The room is warm."

**Example of Propositional Logic**

Let's explore a **real-world scenario** where propositional logic is applied in AI. Consider a **home automation system** that needs to decide whether to **turn on the air conditioner** based on the weather conditions and indoor temperature.

**Scenario:**

- **P**: "It is hot outside."
- **Q**: "The windows are open."
- **R**: "Turn on the air conditioner."

Using **propositional logic**, we can represent the system's decision-making with the following compound proposition:

$(P \land \neg Q) \rightarrow R$

This logic reads as:
"If it is hot outside **AND** the windows are not open, then **turn on the air conditioner**."

**Explanation of the Logic:**

- **AND ( ∧ )** ensures that both conditions must be true (hot outside and windows closed) for the air conditioner to turn on.
- **NOT ( ¬ )** negates the condition, meaning the windows must be **closed**.
- **IF-THEN ( → )** states that if the first part is true, the second part (turning on the AC) will follow.

## Logical Connectives in Propositional Logic

Logical connectives are essential operators that combine **atomic propositions** to form **compound propositions**. These connectives allow AI systems to build more complex rules and perform logical reasoning. Below are the most common connectives used in **propositional logic**:

**Common Logical Connectives**

| Connective | Symbol | Meaning | Example |
|---|---|---|---|
| AND | ∧ | True if both propositions are true. | (P ∧ Q): "It is raining AND cold." |
| OR | ∨ | True if at least one proposition is true. | (P ∨ Q): "It is raining OR cold." |
| NOT | ¬ | Negates the truth value of a proposition. | ¬P: "It is not raining." |
| IF-THEN | → | True unless the first is true and second is false. | (P → Q): "If it rains, then it will flood." |
| IF AND ONLY IF | ↔ | True if both propositions are either true or false. | (P ↔ Q): "It rains if and only if it is cloudy." |

These connectives allow us to create **logical rules** that AI systems can use to make decisions. Let's take a quick look at how each works:

1. **AND ( ∧ ):**
   - The result is **True** only if both propositions are true. Example: If **P** is "It is hot" and **Q** is "The fan is on", then (P ∧ Q) means both conditions are satisfied.
2. **OR ( ∨ ):**
   - The result is **True** if at least one of the propositions is true. Example: (P ∨ Q) will be true if either it is hot or the fan is on.
3. **NOT ( ¬ ):**

- This **inverts** the truth value of the proposition. Example: If **P** is true, ¬P will be false.

4. **IF-THEN ( → ):**
    - This implies that if the first proposition is true, the second must also be true for the compound statement to be true. Example: “If it rains, then the ground will be wet” (P → Q).
5. **IF AND ONLY IF ( ↔ ):**
    - This is true only when both propositions have the **same truth value** (either both true or both false). Example: “It is cloudy if and only if it will rain” (P ↔ Q).

**Truth Table**

A **truth table** is a useful tool for determining the **truth value** of a compound proposition based on the truth values of its atomic propositions. It systematically lists all **possible combinations** of truth values and the corresponding output for a given logical expression.

**How Truth Tables Work**

Let’s consider two propositions:

- **P**: “It is raining.”
- **Q**: “The ground is wet.”

We can build a truth table to evaluate the compound proposition **P ∧ Q** (It is raining AND the ground is wet).

| **P** | **Q** | **P ∧ Q** |
|---|---|---|
| TRUE | TRUE | TRUE |
| TRUE | FALSE | FALSE |
| FALSE | TRUE | FALSE |
| FALSE | FALSE | FALSE |

- The result of **P ∧ Q** is **True** only when both **P** and **Q** are True.

**Truth Table with Three Propositions**

Let’s extend the concept to three propositions:

- **P**: “It is hot.”
- **Q**: “The air conditioner is on.”
- **R**: “The windows are closed.”

We can create a truth table for the compound proposition **(P ∨ Q) ∧ R** (It is hot OR the air conditioner is on, AND the windows are closed).

| P | Q | R | (P ∨ Q) ∧ R |
|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE |
| TRUE | TRUE | FALSE | FALSE |
| TRUE | FALSE | TRUE | TRUE |
| TRUE | FALSE | FALSE | FALSE |
| FALSE | TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE | FALSE |
| FALSE | FALSE | TRUE | FALSE |
| FALSE | FALSE | FALSE | FALSE |

**Purpose of Truth Tables**

- Truth tables help in **evaluating the outcomes** of complex logical expressions.
- They ensure **correct reasoning** by listing all possibilities, making them a vital tool for **AI systems** that rely on logical reasoning.

**Precedence of Connectives in Propositional Logic**

When evaluating **compound propositions** with multiple logical connectives, it's important to follow a specific **order of precedence** to ensure accurate results. Similar to arithmetic operations, **logical operators** are evaluated in a defined sequence, from highest to lowest precedence.

**Order of Precedence**

1. **NOT ( ¬ )** – Negation has the **highest precedence** and is evaluated first.
2. **AND ( ∧ )** – Conjunction is evaluated next, after negations are resolved.
3. **OR ( ∨ )** – Disjunction comes after AND operations.
4. **IF-THEN ( → )** – Implication is evaluated after OR.
5. **IF AND ONLY IF ( ↔ )** – Biconditional has the **lowest precedence**.

**Example: Precedence in Action**

Consider the following logical expression:

**¬P∨(Q∧R)**

- **Step 1:** Evaluate **¬P** (Negation).
- **Step 2:** Evaluate **Q ∧ R** (AND).
- **Step 3:** Evaluate **¬P ∨ (Q ∧ R)** (OR).

The final result depends on the proper **evaluation order**, ensuring the correct outcome.

**Using Parentheses for Clarity**

To avoid ambiguity, it’s good practice to use **parentheses** in complex expressions. For example:

**$(P \lor Q) \rightarrow R$**

In this case, **$(P \lor Q)$** is evaluated first, followed by the implication **$\rightarrow$**

**Logical Equivalence in Propositional Logic**

**Logical equivalence** occurs when two or more logical expressions produce the **same truth values** for all possible combinations of their propositions. In other words, two statements are logically equivalent if they always have the **same result**, regardless of the truth values of the individual propositions.

**Definition of Logical Equivalence**

Two propositions **P** and **Q** are logically equivalent if:

**$P \equiv Q$**

This means that both **P** and **Q** yield identical truth values for all possible cases. Logical equivalence allows AI systems to **simplify complex expressions** without changing their meaning.

**Example of Logical Equivalence**

1. **De Morgan’s Laws:**
    - These laws show how **negations of conjunctions** and **disjunctions** behave:

$\neg(P \land Q) \equiv (\neg P \lor \neg Q)$

$\neg(P \lor Q) \equiv (\neg P \land \neg Q)$

2. **Double Negation:**
    - Negating a negation gives the original proposition:

$\neg(\neg P) \equiv P$

3. **Implication and Disjunction:**
    - An implication can be rewritten as:

$P \rightarrow Q \equiv \neg P \lor Q$

**Tautologies and Contradictions**

- **Tautology:** A tautology is a statement that is **always true**, no matter the truth values of its individual propositions.
    - **Example:** $P \lor \neg P \equiv \text{True}$
- **Contradiction:** A contradiction is a statement that is **always false**.
    - **Example:** $P \land \neg P \equiv \text{False}$

**Properties of Operators in Propositional Logic**

In propositional logic, **logical operators** follow specific properties that allow us to **manipulate and simplify logical expressions**. Understanding these properties is essential for building efficient AI systems that rely on logical reasoning.

**1. De Morgan's Laws**

These laws describe how **negations** distribute over **AND (∧)** and **OR (∨)** operations:

- **First Law:**
  - $\neg(P \land Q) \equiv (\neg P \lor \neg Q)$

This means that the negation of a conjunction is equivalent to the disjunction of the negated propositions.

- **Second Law:**
  - $\neg(P \lor Q) \equiv (\neg P \land \neg Q)$

This means that the negation of a disjunction is equivalent to the conjunction of the negated propositions.

**2. Commutative Property**

This property states that the **order of the propositions** does not affect the result of **AND ( ∧ )** and **OR ( ∨ )** operations:

- **AND:**
  - $P \land Q \equiv Q \land P$
- **OR:**
  - $P \lor Q \equiv Q \lor P$

**3. Associative Property**

This property allows us to **group propositions** in any order when using **AND** or **OR** operations:

- **AND:**
  - $(P \land Q) \land R \equiv P \land (Q \land R)$
- **OR:**
  - $(P \lor Q) \lor R \equiv P \lor (Q \lor R)$

**4. Distributive Property**

This property states that **AND distributes over OR**, and vice versa:

- **AND over OR:**
  - $P \land (Q \lor R) \equiv (P \land Q) \lor (P \land R) P$
- **OR over AND:**
  - $P \lor (Q \land R) \equiv (P \lor Q) \land (P \lor R)$

**Applications of Propositional Logic in AI**

**1. Knowledge Representation in Expert Systems:**

- Represents **rules and facts** to solve domain-specific problems (e.g., medical diagnosis systems).

**2. Reasoning and Decision-Making:**

- AI agents use logical rules to make **decisions** (e.g., robot vacuum cleaners deciding when to start cleaning).

**3. Natural Language Processing (NLP):**

- Helps analyze text and respond logically (e.g., chatbots understanding weather-related queries).

**4. Game-Playing AI:**

- Uses logic to **make strategic moves** (e.g., deciding checkmate in chess).

**Limitations of Propositional Logic**

**1. Inability to Handle Complex Relationships**

- Propositional logic cannot represent **relationships between multiple objects** or deal with hierarchies of information.

**2. No Handling of Uncertainty**

- It works only with **true or false** values and cannot deal with **probabilities or uncertain outcomes**, limiting its use in real-world applications involving incomplete data.

**3. Limited Expressiveness**

- It cannot represent **time-based sequences** or dynamic events, which are crucial in some AI systems like speech recognition and robotics.

**4. Scalability Issues**

- As the number of propositions grows, the **complexity of expressions** increases, making reasoning slower and harder to manage.

**PROPOSITIONAL THEOREM PROVING**

Propositional theorem proving is a different approach to using logic to solve problems is to use logical rules of inference to generate logical implications in some cases, this can be less work than model-checking (i.e. generating a truth table) or even SAT solving plus, logical rules of inference are at the level of logical reasoning that humans consciously strive to do **theorem-proving** is interested in **entailment**, e.g. given a logical sentence A and a logical sentence B, we can ask if A entails B the general idea is that sentence A is what the agent knows, and sentence B is something that the agent can infer from what it knows sentence A might be very big, i.e. the entire "brain" of an agent encoded in logic an important basic result in logic is the **deduction theorem**, which says:

A entails B if, and only if, A => B (A implies B) is a tautology (i.e. valid for all assignments of values to its variables)

so we can answer the question "Does A entail B?" by showing that the sentence A => B is a tautology

recall that a sentence is unsatisfiable if no assignment of values to its variables makes it true

so, A => B is a tautology, then !(A=>B) is unsatisfiable, and !(A=>B) == !(!(A & !B)) == A & !B

so, we can re-write the deduction theorem like this:

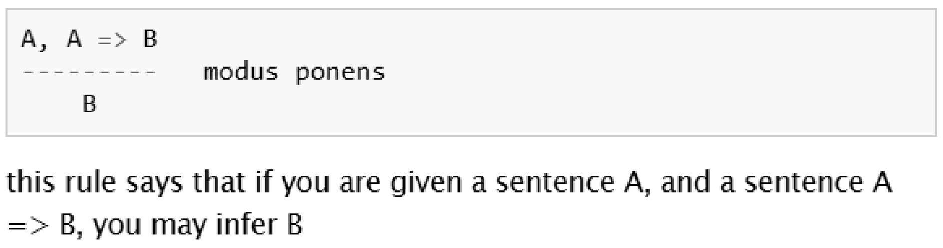
A entails B if, and only if, A & !B is unsatisfiable

this means you can use a SAT solver to figure out entailment!

**Rules of Inference**

recall that there are various rules of inference that can be used to create proofs, i.e. chains of correct inferences

e.g. **modus ponens** is this rule of inference:



```
A, A => B
---------    modus ponens
    B
```

this rule says that if you are given a sentence A, and a sentence A => B, you may infer B e.g. **and-elimination** is this pair of rules:

```
A & B
-----        and-elimination
  A

A & B
-----
  B
```

these two rules encode the (obvious!) fact that if the sentence A & B is true, then A is true, and also B is true

logical equivalences can also be stated as inference rules, e.g.:

```
A <==> B
---------------
(A=>B) & (B=>A)
```

there are many inference rules, and choosing a reasonable set of rules for a theorem-prover turns out to be important

- we can think of the application of one rule of inference as an action performed to the state of the world, i.e. the rule of inference adds more facts to the knowledge base
- if there are multiple inference rules to choose from, then we need knowledge (i.e. heuristics) to help decide which rule to use
    - or we could rely on backtracking, i.e. just pick a rule at random, apply it, and keep going until it “seems” like a proof is not being reached
- plus, how do we know that the rules of inference we are using are **complete**, i.e. if a proof exists, how do we know our set of inference rules will find it?
    - e.g. suppose the two and-elimination rules were our only rules of inference; is this enough to prove any entailment in propositional logic?
        - no!
        - for example, (P | P) -> P is clearly true, but and-elimination doesn’t apply to this sentence

**Proof by Resolution**

it turns out that only one particular inference rule is needed to prove any logical entailment (that can be proved): the **resolution rule**

in a 2-variable form, the resolution inference rule is this (| here means “or”):

```
A | B,  !B
----------
   A
              resolution

A | !B,  B
----------
   A
```

in English, the first rules says that if A or B is true, and B is not true, then A must be true; the second rule says the same thing, but B and !B are swapped
note that A | !B is logically equivalent to B -> A, and so the resolution rules can be translated to this:

```
!B -> A, !B
-----------    variation of modus ponens
   A


B -> A, B
---------    modus ponens
   A
```

in other words, resolution inference could be viewed as a variation of inference with modus ponens
we can generalize resolution as follows:

```
A_1 | A_2 | ... | A_i | ... | A_n    !A_i
-----------------------------------------    resolution generalized
A_1 | A_2 | ... | A_i-1 | A_i+1... | A_n
```

we've written A_i and !A_i, but those could be swapped: the key is that they are complementary literals
notice that both sentence above and below the inference line are CNF clauses

- recall that a CNF clause consists of literals (a variable, or the negation of a variable), or-ed together

so resolution can be stated conveniently in terms of clauses, e.g.:

```
Clause_1   L_i     opposite of literal L_i is in Clause_1
-------------
Clause_2           Clause_2 is Clause_1 with opposite of literal L_i removed
```

here, L_i is a literal that has the opposite sign of a literal in Clause_1

- i.e. L_i is the complement of a literal in Clause_1

Clause_2 is the same as Clause_1, but with the literal that is the opposite of L_i removed
e.g.:

```
(A | !B | !C)  !A      A and !A are complementary literals
-----------------
    !B | !C

(A | !B | !C)  C       C and !C are complementary literals
-----------------
    A | !B
```

**full resolution** generalizes this even further to two clauses:

```
Clause_1  Clause_2
------------------    full resolution
    Clause_3
```

here, we assume that some literal L appears in Clause_1, and the complement of L appears in Clause_2

Clause_3 contains all the literals (or-ed together) from both Clause_1 and Clause_2, *except* for L and its opposite — those are not in Clause_3

e.g.:

```
(A | !B | !C)  (!A | !C | D)    A and !A are complementary literals
----------------------------
        !B | !C | D
```

what is surprising is that full resolution is *complete*: it can be used to proved any entailment (assuming the entailment can be proven)

to do this, resolution requires that all sentences be converted to CNF

- this can always be done ... see the textbook for a sketch of the basic idea
- the same requirement for most SAT solvers!

next, recall from above that if A entails B, then A & !B is unsatisfiable

- this is a consequence of the deduction theorem

resolution theorem proving proves that A entails B by proving that A & !B is unsatisfiable

it follows these steps:

- A & !B is converted to CNF
- clauses with complementary literals are chosen and resolved (i.e. the resolution rule is applied to them to remove the literal and its complement) and added as a new clause in the knowledge base
- this continues until one of two things happens:
    1. no new clause can be added, which means that A does **not** entail B
        - in other words, A & !B is satisfiable
    2. two clauses resolve to yield the **empty clause**, which means that A entails B
        - the empty clause means there is a contradiction, i.e. if P and !P are clauses then we can apply resolution to them to get the "empty clause"

        - clearly, if both P and !P are in the same knowledge base, then it is inconsistent since P & !P is false for all values of P

Example. KB = {P->Q, Q->R}

- does KB entail P->R?
- yes it does, and lets use resolution to prove this
- to prove KB entails P->R, we will prove that KB & !(P->R) is unsatisfiable
- first we convert KB & !(P->R) to CNF:

  ```
  !P | Q
  !Q | R
  P
  !R
  ```

  next we pick pairs of clauses and resolve them (i.e. apply the resolution inference rule) if we can; we add any clauses produced by this to the collection of clauses:

  ```
  !P | Q
  !Q | R
  P
  !R

  !P | R     // from resolving (!P | Q) with (!Q | R)
  Q          // from resolving (!P | Q) with (P)
  !Q         // from resolving (!Q | R) with (!R)
  ```

- we have Q and !Q, meaning we've reach a contradiction
- this means KB & !(P->R) is unsatisfiable
- which means KB entails P->R

this is a *proof* of entailment, and the various resolutions are the steps of the proof

the exact order in which clauses are resolved could result in shorter or longer proofs, and in practice you usually want short proofs, and so heuristic would be needed to help make decisions

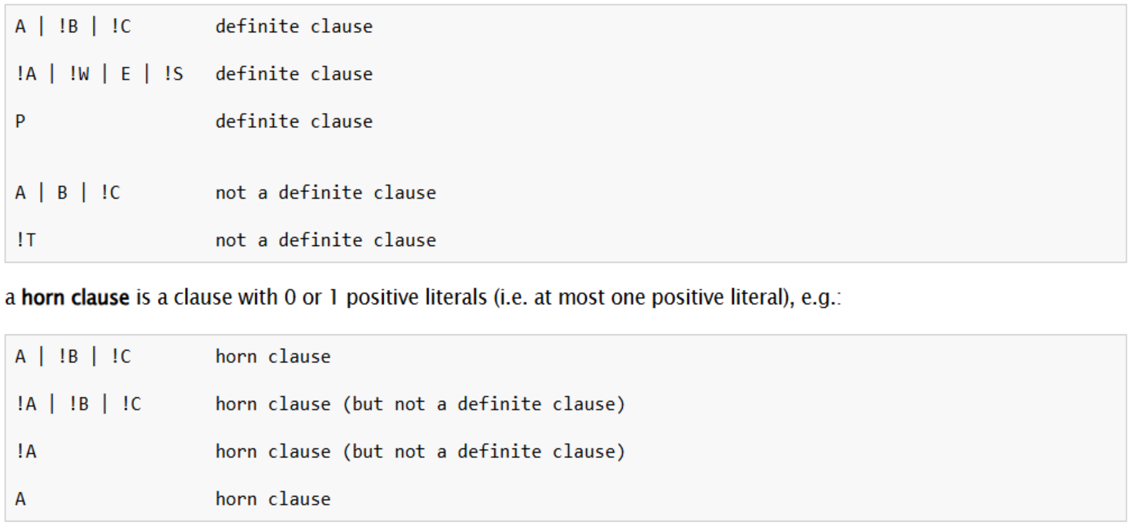
**Horn Clauses and Definite Clauses**

a fundamental difficulty with SAT solvers and algorithms like resolution is that, for propositional logic, they have exponential running time

- SAT solvers are fast, but in the worst case they can still take an exponential amount of time to finish
- plus, many problems require thousands or more variables when encoded in propositional logic, and that alone makes some problems simply too big to be handled by SAT solvers

so, some AI researchers have explored the possibility of using a logic that is *less* expressive than propositional logic, but for which entilaments can be computed more efficiently

for example, a **definite clause** is a clause in which *exactly 1* literal is positive, e.g.:



```
A | !B | !C         definite clause

!A | !W | E | !S    definite clause

P                   definite clause


A | B | !C          not a definite clause

!T                  not a definite clause
```

a **horn clause** is a clause with 0 or 1 positive literals (i.e. at most one positive literal), e.g.:

```
A | !B | !C         horn clause

!A | !B | !C        horn clause (but not a definite clause)

!A                  horn clause (but not a definite clause)

A                   horn clause
```

note that every definite clause is a horn clause, but some horn clauses are not definite clauses

these restricted forms of clauses are interesting for a few reasons:

- entailment of horn clauses can be decided in time **linear** with respect to the size of the knowledge base – much more efficient than full resolution
- reasoning with horn clauses can be done in a straightforward way that humans find easier to follow than resolution
- definite clauses can be written in an appealing form using implication, i.e. the definite clause A | !B | !C can be written (B & C) -> A
    - more generally, the definite clause A | !B1 | !B2 | … | !Bn can be written as (B1 & B2 & … & Bn) -> A
        - the implication is sometimes written in reverse:
        - A <- B1 & B2 & ... & Bn
    - for some real-world problems, this is can be a natural way to encode knowledge, i.e. A is true if B1, B2, …, and Bn are all true
    - horn clauses are used extensively in the logic programming language.