UNIT II

Problem Solving by Search-II and Propositional Logic Adversarial Search:

Games,Optimal Decisions in Games, Alpha–Beta Pruning, Imperfect Real-Time Decisions.

Constraint Satisfaction Problems:

Defining Constraint Satisfaction Problems,Constraint Propagation,Backtracking Search for CSPs,

Local Search for CSPs,The Structure of Problems.

Propositional Logic:

Knowledge-Based Agents,The Wumpus World,Logic,Propositional Logic

Propositional Theorem Proving:

Inference and proofs,Proof by resolution,Horn clauses and definite clauses,Forward and backward chaining,Effective Propositional Model Checking,Agents Based on Propositional Logic.

Adversarial Search: Games , Decisions in Games

Adversarial search is a specialized area of problem-solving by search, where the "environment" is another intelligent agent (or agents) trying to optimize its own goals, which are often in conflict with the first agent's goals. This is most famously applied to **games**.

The agent needs to make decisions considering that an opponent will also make optimal (or near-optimal) moves to counter its plans.

Characteristics of Adversarial Games:

- **Multi-agent:** More than one player.
- **Zero-Sum (often):** One player's gain is another's loss (e.g., chess).
- **Deterministic (often):** No random elements (e.g., chess, checkers).
- **Perfect Information (often):** Both players know the full state of the game (e.g., chess, not poker).
- **Finite:** The game eventually ends.

Game Tree: Games are typically represented as a **game tree**, where:

- Nodes are game states.
- Edges are moves.
- Levels alternate between players (e.g., MAX's turn, MIN's turn).
- Terminal nodes have utility values (e.g., +1 for win, -1 for loss, 0 for draw)

Minimax Algorithm:

- **Concept:** A recursive algorithm for selecting the optimal move in a two-player, zero-sum, perfect-information game. It assumes that the opponent will also play optimally.
- **How it works:**
    - **MAX player** tries to maximize the utility.
    - **MIN player** tries to minimize the utility (which is equivalent to maximizing their own, opposite, utility).
    - It evaluates terminal nodes, then propagates these values up the tree, assuming optimal play at each step.

Alpha-Beta Pruning:

- **Concept:** An optimization to the Minimax algorithm. It prunes (cuts off) branches of the game tree that can be proven to be irrelevant to the final decision.
- **How it works:** It keeps track of the best alternatives found so far for MAX (alpha value) and MIN (beta value) and prunes branches that can't possibly beat these current best options.
- **Effect:** Significantly reduces the number of nodes that need to be evaluated, making deeper searches feasible.

Evaluation Functions:

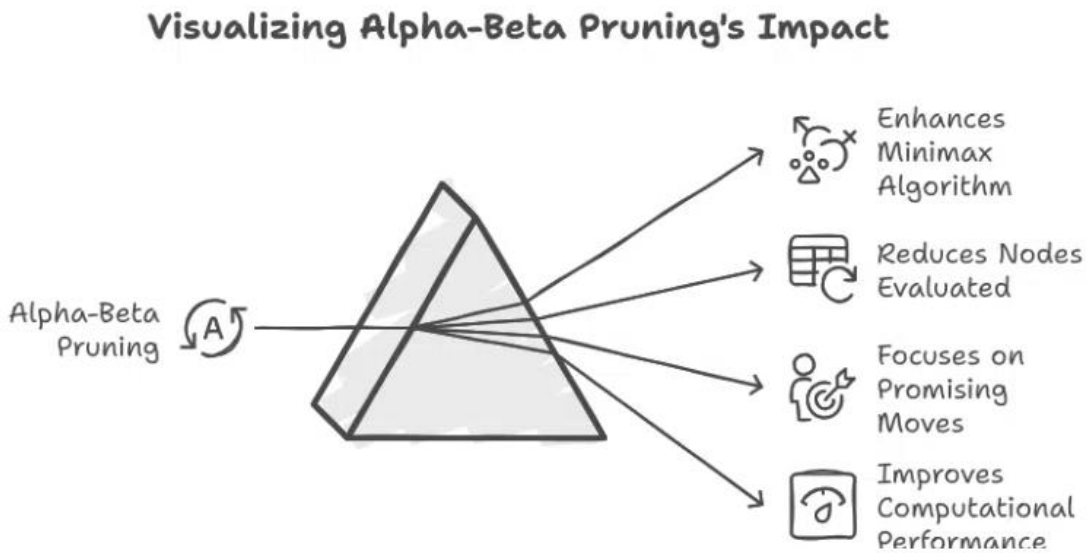
- For games too complex to search to the very end (like Chess), an **evaluation function** (or heuristic) is used to estimate the utility of non-terminal game states. This function assigns a score to a board configuration, allowing the Minimax (or Alpha-Beta) algorithm to search to a limited depth and then use the evaluation function at the leaf nodes of its limited search tree.

Other Concepts in Games:

- **Transposition Tables:** Store results of previously computed game states to avoid redundant calculations.
- **Opening Books / Endgame Databases:** Precomputed optimal moves for initial and final game phases.
- **Stochastic Games:** Introduce chance nodes (e.g., dice rolls) and require expected minimax.
- **Partially Observable Games:** Players don't know the full state (e.g., Poker), leading to belief states.

- **Monte Carlo Tree Search (MCTS):** A powerful search technique, especially popular in games with large branching factors (like Go), that uses random simulations to guide tree exploration.

Alpha–Beta Pruning



- **Alpha-Beta Pruning** is an optimization technique used to enhance the performance of the **Minimax algorithm** in game tree search.
- Alpha Beta Pruning optimizes AI decision-making by reducing unnecessary computations in the minimax algorithm.
- The word 'pruning' means cutting down branches and leaves.
- In Artificial Intelligence, Alpha-beta pruning is the pruning of useless branches in decision trees.
- The minimax algorithm is a decision-making process commonly used in two-player, zero-sum games like chess
- The minimax algorithm operates by recursively exploring all possible game states and assigning values to the leaf nodes based on the potential outcomes of the game. With this the complexity of the game increases, the number of possible states grows exponentially, leading to ***very high computational costs***.

- It helps in reducing the number of nodes that need to be evaluated in the search tree by eliminating branches that cannot possibly influence the final decision.

- Alpha-Beta pruning allows us to skip the evaluation of branches in the tree that have already been proven to be irrelevant.

Concepts in Alpha-Beta Pruning:

- **Alpha**: The best value that the maximizer can guarantee so far. It's the value of the best (highest) choice found so far at any point along the path of the maximizer.
  The initial value of alpha is $-\infty$.
- **Beta**: The best value that the minimizer can guarantee so far. It's the value of the best (lowest) choice found so far at any point along the path of the minimizer.
  The initial value of alpha is $+\infty$

- **Pruning**: If a node is found to be worse than the current best choice, it can be "pruned" (i.e., skipped) because it doesn't need to be explored further. If a move is found to be worse than the current best choice, there's no need to explore it further.

Algorithm :

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(%α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

How Alpha-Beta Pruning Works:

- The **Maximizing Player** (often the AI in a game) explores the game tree starting from the **root**.
- As the algorithm progresses down the tree, it calculates the best possible move for the current player while simultaneously maintaining **alpha** (the best score the maximizer can guarantee) and **beta** (the best score the minimizer can guarantee).

- If a node's score is worse than the best available option for the current player (i.e., **beta ≤ alpha**), the branch is pruned, and the algorithm moves on to the next branch.

This process allows for a **dramatic reduction** in the number of nodes that need to be evaluated, especially if the tree is large.

Time Complexity :

Without pruning, the **Minimax algorithm** has a time complexity of $O(b^d)$, where b is the branching factor (the number of possible moves from a given state) and d is the depth of the tree.

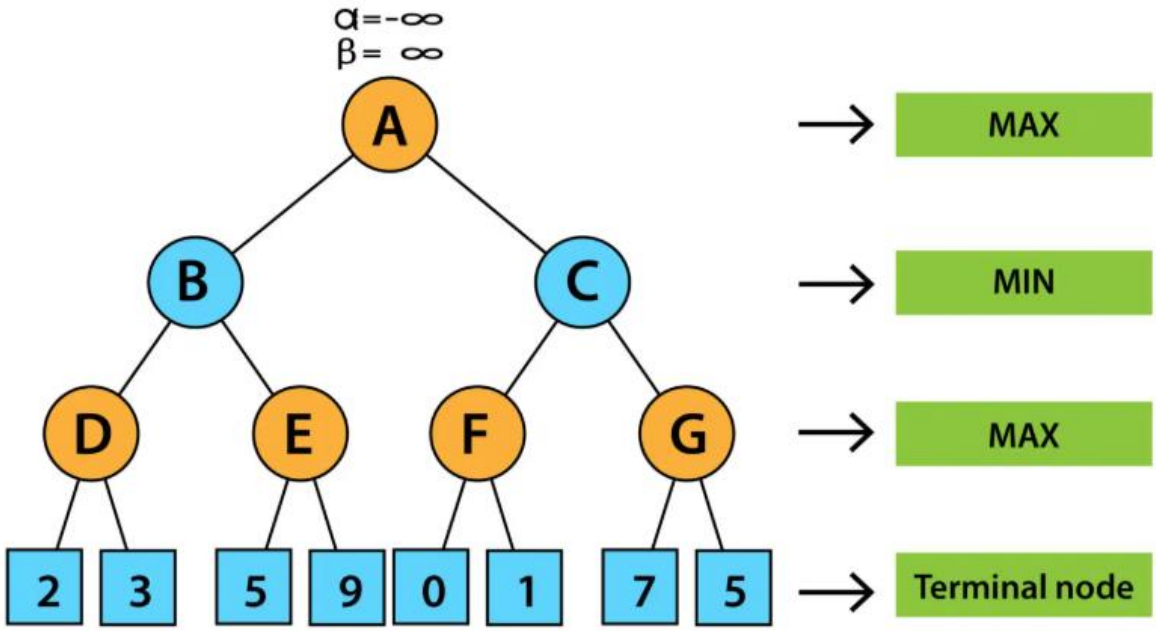
With Alpha-Beta pruning, the best-case time complexity improves to $O(b^{d/2})$, because pruning allows the search to effectively examine fewer nodes by cutting off unpromising branches.

Space Complexity:

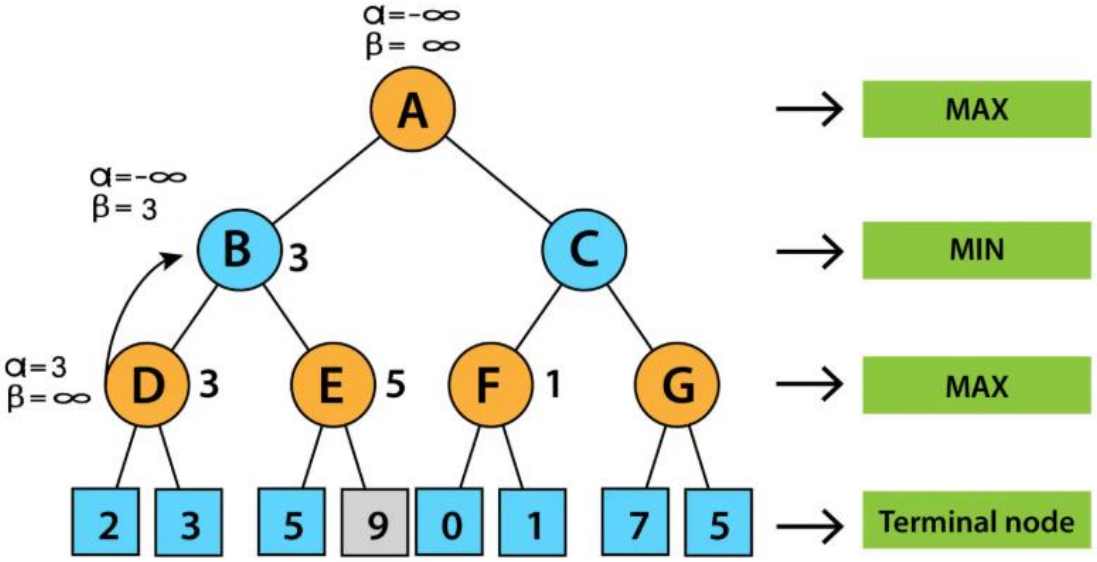
The space complexity remains O(b·d), where:

- b is the branching factor, and d is the depth of the search tree, because the algorithm uses recursion and stores nodes in memory.

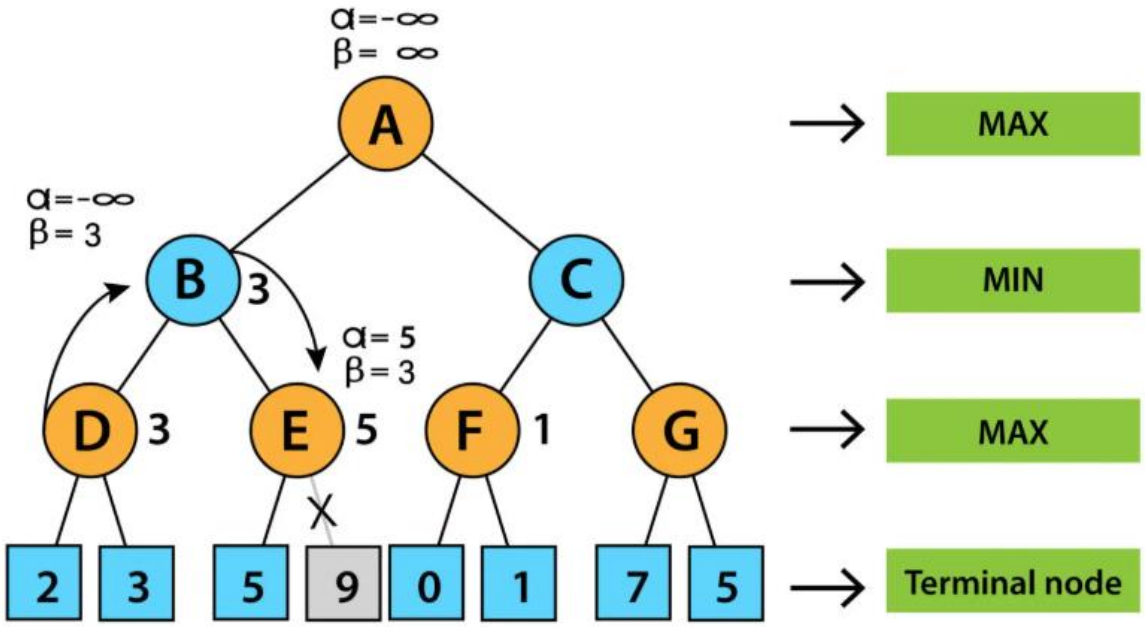
Example :



- First, we will take care of the first move. So initially, we will define the worst case α = −∞ and β = +∞. If alpha is greater than or equal to beta, we will prune the node.
- We didn't prune it since the initial value of alpha is less than beta. Now it's turn for MAX. Therefore, we will calculate the value of alpha at node D. At node D, the value of alpha will be **max (2, 3) = 3.**
- Now, node B's turn is MIN. That means that the value of alpha beta at node B will be **min (3,** ∞**).** Therefore, alpha will be – ∞ and beta will be 3 at node B.
- Next, algorithms will pass the values of α= -∞ and β= 3 to the next successor of Node B, that is node E.



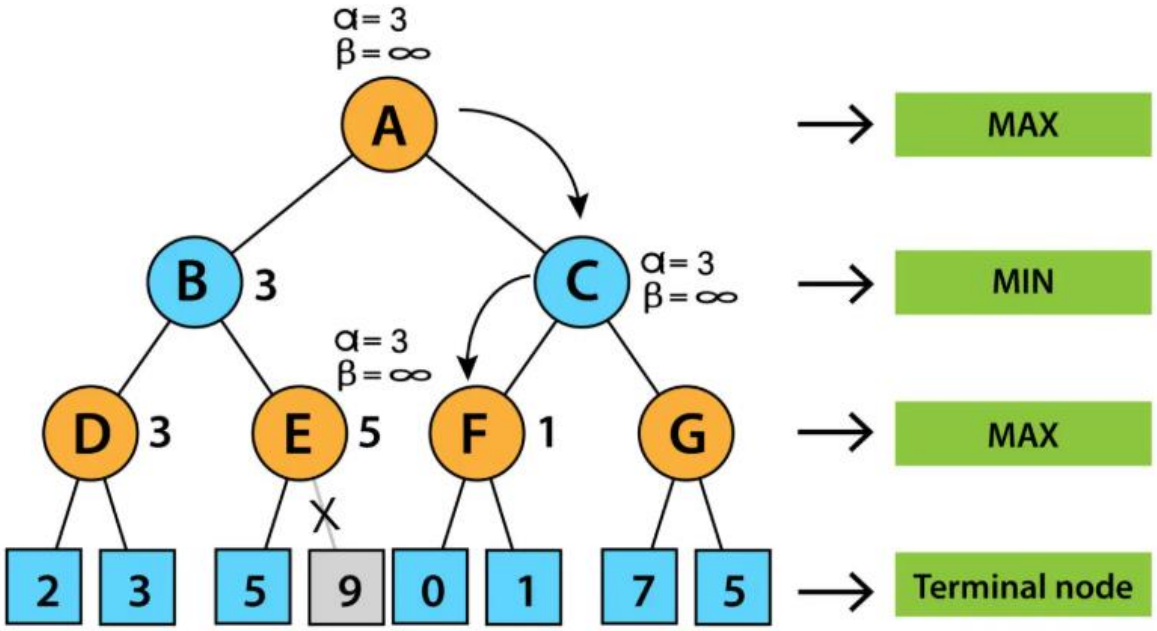
- Now it's turn for MAX. Therefore, we will search for MAX at node E. The value of alpha at E is – ∞ and will be compared with 5. So, MAX (- ∞, 5) will be
- Thus, at node E, alpha = 5, Beta = 5.



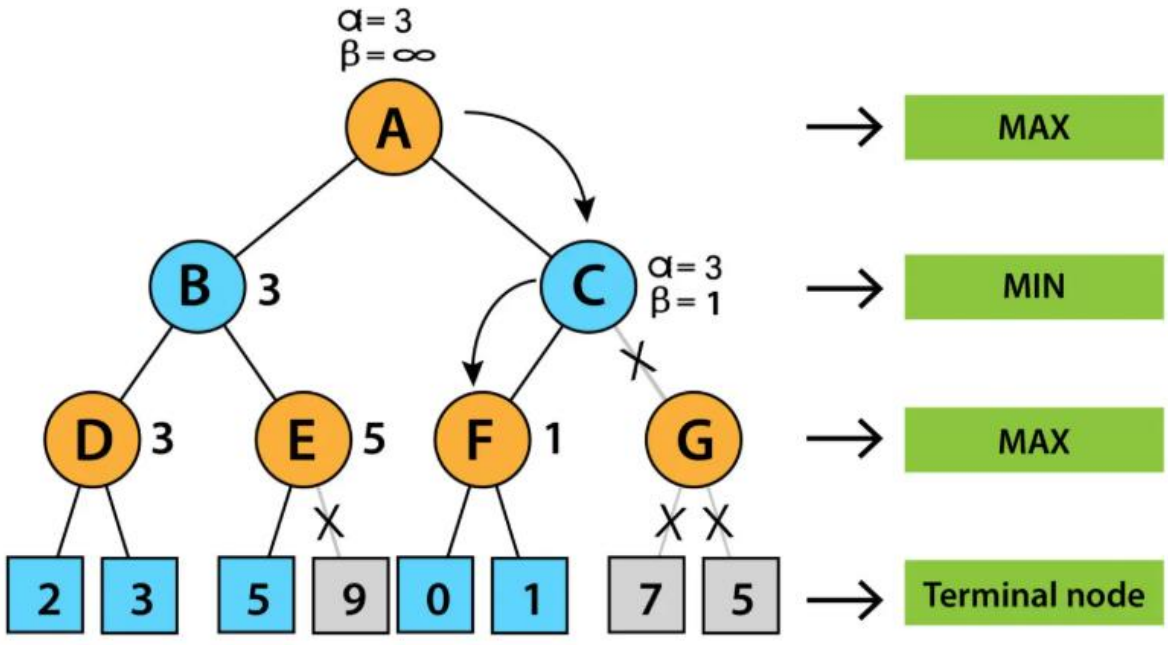
- We now know that alpha is greater than beta and is also a pruning condition, so we can prune the right successor of node E, and the algorithm will not be visited, and the value of node E will be 5.

- In the next step, the algorithm again comes from node B to node A. The alpha of node A is changed to the max value of MAX(- ∞, 3). Now, alpha and beta at node A are (3, + ∞) and will be transmitted to node C. The same values will be transferred to node F.

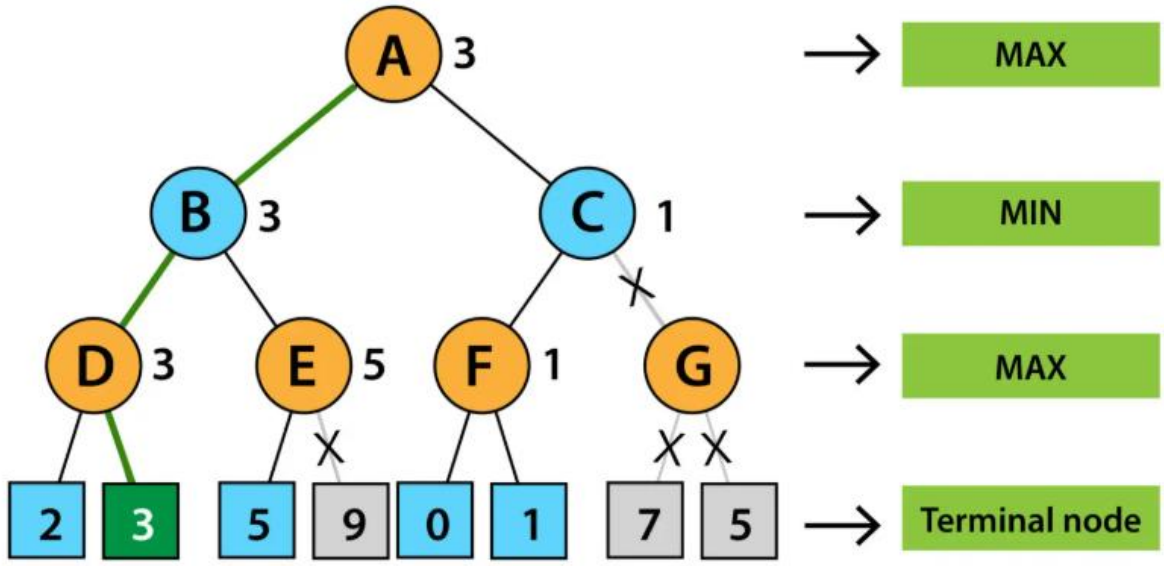
❖ The value of alpha will be compared to the left branch, which is 0 at node F. Now, MAX (0, 3) will be 3 and is compared with the right child (1) and MAX(3, 1) = 3 still α is 3, but the node value of F will be 1.



❖ Next, node F will pass the node value 1 to C and compare it to the beta value at C. Now it's turn for MIN. So, MIN (+ ∞, 1) will be 1. Now, at node C, α = 3, β= 1, and alpha is greater than beta, which again satisfies the pruning condition. Then, the successor of node C, G, will be pruned, and the algorithm did not calculate the whole subtree G.



- Now, C will return its node value to A, and A will calculate MAX(1, 3), resulting in 3.



- The above-represented tree is the final tree that has nodes that are computed and nodes that are not computed. Therefore, in this example, the optimal value of the maximizer would be 3.

- Now it's turn for MAX. Therefore, we will search for MAX at node E. The value of alpha at E is – ∞ and will be compared with 5. So, MAX (- ∞, 5) will be 5. Thus, at node E, alpha = 5, Beta = 5.

Imperfect Real-Time Decisions.

An AI agent in an adversarial game cannot afford the time to perform a full, deep search (like a perfect mini max with alpha-beta pruning) to find the absolute optimal move. Instead, it must make a decision *quickly*, with limited computational resources, and often with imperfect information or a simplified understanding of the game.

This is relevant for many real-world AI applications, especially in areas like:

- **Video Games:** Most game AI in complex games (RTS, FPS, RPGs) cannot run deep search trees for every decision due to real-time constraints and the need for believable, not just optimal, behavior.
- **Robotics:** A robot navigating a dynamic environment with an adversary needs to react quickly.
- **Financial Trading:** Decisions must be made in milliseconds.
- **Autonomous Driving:** Reacting to other drivers.

Why "Imperfect Real-Time Decisions" are Necessary?

- **Sufficient Time:** They can search the game tree to a reasonable depth, or even to the end for simple games.
- **Perfect Information:** The agent knows the complete state of the game.
- **Deterministic Actions:** Actions have predictable outcomes.
- **Known Opponent:** The opponent is assumed to be rational and play optimally.

Defining Constraint Satisfaction Problems

A **Constraint Satisfaction Problem (CSP)** is a mathematical problem defined by a set of variables, each of which has a domain of possible values, and a set of constraints that restrict the combinations of values that these variables can take. The goal is to find an assignment of values to all variables such that all the constraints are satisfied.
CSPs provide a powerful and flexible framework for representing and solving a wide range of problems in AI

A CSP is formally defined by three components: (X,D,C)

1. **Variables (X):**
   - A finite set of variables: X={X1,X2,…,Xn}.
   - Each variable represents a specific decision point or attribute in the problem.
2. **Domains (D):**
   - A finite set of possible values for each variable: D={D1,D2,…,Dn}.
   - Each Di is the **domain** of variable Xi, specifying the set of all possible values that Xi can take.
   - Domains can be discrete (e.g., colors, numbers, true/false) or continuous (though continuous CSPs are harder and often approximated).
3. **Constraints (C):**
   - A finite set of constraints: C={C1,C2,…,Cm}.
   - Each constraint Cj is a relation that restricts the values that a subset of variables can simultaneously take.
   - A **constraint** consists of:
     - **Scope:** A tuple of variables involved in the constraint (e.g., (Xi,Xj)).
     - **Relation:** A specification of the allowed combinations of values for the variables in its scope. This can be an explicit list of allowed (or forbidden) tuples, or an algebraic expression .

A **solution** to a CSP is a complete assignment of values to all variables, A={X1=v1,X2=v2 ,…,Xn=vn}, such that:

1. Each value vi is drawn from the domain of its respective variable Di (vi∈Di).
2. All constraints Cj are satisfied by this assignment. That is, for every constraint Cj, the tuple of values assigned to the variables in its scope is consistent with the relation defined by Cj

Constraint Satisfaction Problems (CSPs): Defining Constraint Satisfaction Problems

A **Constraint Satisfaction Problem (CSP)** is a mathematical problem defined by a set of variables, each of which has a domain of possible values, and a set of constraints that restrict the combinations of values that these variables can take. The goal is to find an assignment of values to all variables such that all the constraints are satisfied.

CSPs provide a powerful and flexible framework for representing and solving a wide range of problems in AI, operations research, and various other fields.

Formal Definition of a CSP

A CSP is formally defined by three components: (X,D,C)

1. **Variables (X):**
    - A finite set of variables: X={X1,X2,…,Xn}.
    - Each variable represents a specific decision point or attribute in the problem.
2. **Domains (D):**
    - A finite set of possible values for each variable: D={D1,D2,…,Dn}.
    - Each Di is the **domain** of variable Xi, specifying the set of all possible values that Xi can take.
    - Domains can be discrete (e.g., colors, numbers, true/false) or continuous (though continuous CSPs are harder and often approximated).
3. **Constraints (C):**
    - A finite set of constraints: C={C1,C2,…,Cm}.
    - Each constraint Cj is a relation that restricts the values that a subset of variables can simultaneously take.
    - A constraint consists of:
        - **Scope:** A tuple of variables involved in the constraint (e.g., (Xi,Xj)).
        - **Relation:** A specification of the allowed combinations of values for the variables in its scope. This can be an explicit list of allowed (or forbidden) tuples, or an algebraic expression (e.g., X1□=X2, X3<X4, X5=X6+X7).

Solution to a CSP

A **solution** to a CSP is a complete assignment of values to all variables, A={X1=v1,X2=v2 ,…,Xn=vn}, such that:

1. Each value vi is drawn from the domain of its respective variable Di (vi∈Di).
2. All constraints Cj are satisfied by this assignment. That is, for every constraint Cj, the tuple of values assigned to the variables in its scope is consistent with the relation defined by Cj.

Types of Constraints :

a. **Unary Constraints** : Involve a single variable. They restrict the values that a specific variable can take.
b. **Binary Constraints** : Involve exactly two variables. They specify allowed/forbidden combinations for a pair of variables
c. **Ternary Constraints** : Involve three variables
d. **Global Constraints (or N-ary Constraints)** : Involve an arbitrary number of variables, sometimes representing complex relationships that are best handled by specialized algorithms

Why CSPs are Important in AI ?

- **Declarative Representation:** CSPs allow problems to be defined declaratively (what the solution looks like), rather than procedurally (how to find it). This separation of problem definition from the solution algorithm simplifies modeling.

- **Generality:** Many seemingly diverse problems can be naturally formulated as CSPs, demonstrating the power of the framework.

- **Efficient Algorithms:** Specialized algorithms have been developed for CSPs that are much more efficient than general-purpose search for many problems. These include:

  - **Backtracking Search**
  - **Constraint Propagation (Consistency Techniques**
  - **Variable and Value Ordering Heuristics**
  - **Local Search for CSPs**

Examples of CSPs :

- N-Queens Problem
- Sudoku
- Map Coloring
- Scheduling Problems
- Timetabling

Constraint Propagation

Constraint propagation is the process of **reducing the domains of variables** by enforcing consistency conditions

- **Identify values that are inconsistent** with one or more constraints.

- **Remove these inconsistent values** from the domains of the variables.

- **Propagate the effects** of these removals to other variables and constraints, potentially leading to further domain reductions

Why CSP propagation is Important?

- **Reduces Search Space:** By removing values from domains, the number of possible assignments to explore during search (e.g., with backtracking) is significantly reduced. This is critical for solving large CSPs.

- **Early Failure Detection:** If at any point during propagation a variable's domain becomes empty, it means the current partial assignment or the initial problem formulation has no solution. This allows the search to backtrack or terminate early, avoiding fruitless exploration.

- **Efficiency:** While propagation itself takes time, it often saves far more time by pruning the search tree.

Types of Consistency and Propagation Algorithms :

1. **Node Consistency**
2. **Arc Consistency**
3. **Path Consistency**
4. **K-Consistency**

Backtracking Search for CSPs

Backtracking Search is the most common and fundamental algorithm for solving Constraint Satisfaction Problems (CSPs). It's a general-purpose recursive algorithm that systematically searches for a valid assignment to variables by incrementally building a partial solution and checking at each step if the current partial assignment violates any constraints. If a violation occurs, it "backs up" (backtracks) to a previous decision point and tries a different path.

Purpose of Backtracking :

For example : A set of empty slots (variables) that you need to fill with items (values) from a limited set (domains), subject to some rules (constraints). Backtracking search works like this:

1. **Start with the first variable.**
2. **Assign it a value** from its domain.
3. **Check if this assignment is consistent** with all relevant constraints with already assigned variables.
4. **If consistent:** Move to the next unassigned variable and repeat the process.

5. **If inconsistent (or no values left for the current variable):** Unassign the current variable, backtrack to the *previous* variable, and try a different value for that variable. If no more values are left for that variable, backtrack further.
6. **Success:** If all variables are assigned values and all constraints are satisfied, a solution is found.
7. **Failure:** If the search exhausts all possible assignments without finding a solution, the problem has no solution.

Example : MAP Coloring

Efficiency and Improving Backtracking Search

Basic backtracking search can be very inefficient for large CSPs, often exploring huge portions of the search space unnecessarily. Several heuristics and techniques are used to improve its performance

1. Ordering Heuristics
2. Constraint Propagation (Look-Ahead)

Advantages of Backtracking Search

- **Generality:** Can solve any CSP, given enough time.
- **Completeness:** Guaranteed to find a solution if one exists, or prove that no solution exists.
- **Foundation:** Forms the basis for more advanced CSP algorithms.

Disadvantages of Backtracking Search

- **Inefficiency (without improvements):** Can be very slow for large or tightly constrained problems.
- **"Thrashing":** Repeatedly exploring similar failed paths due to local inconsistencies that are not detected early. Constraint propagation helps mitigate this.

Local Search for CSPs

This is an alternative to systematic search (like backtracking) that can be very effective for certain types of CSPs, especially very large ones or those where finding *any* solution is sufficient, rather than proving no solution exists

Purpose of Local Search

1. **Start with a complete assignment** of values to all variables. This assignment will likely violate some constraints.
2. **Define a cost function :** This function quantifies how "bad" the current assignment is. A common cost function for CSPs is simply the **number of violated constraints**. The goal is to minimize this cost to zero.

3. **Define a "neighborhood" of states:** A neighbor of the current assignment is typically formed by changing the value of *one* variable.
4. **Iteratively move to a neighboring state** that improves the cost function (or makes progress towards a solution).

Key Algorithms and Concepts:

1. **Min-Conflicts Heuristic:**
    - This is the most popular and effective local search heuristic for CSPs.
    - **Algorithm:**
        1. Start with a random complete assignment.
        2. While there are still violated constraints:
            - Randomly select a variable `X` that is involved in a violated constraint.
            - Choose a new value for `X` from its domain that **minimizes the number of conflicts** (i.e., minimizes the number of violated constraints) with the other variables, given their current assignments.
            - If there are ties, pick one randomly.
        3. If no constraints are violated, a solution is found.

        Example (N-Queens): If you have queens on a chessboard and some are attacking each other, Min-Conflicts would pick a queen that's under attack and move it to a square in its column where it attacks the fewest other queens.

2. **Hill Climbing (Greedy Descent)**
3. **Simulated Annealing**
4. **Genetic Algorithms (GAs) / Evolutionary Algorithms**
5. **Tabu Search**

Advantages of Local Search for CSPs:

- **Memory Efficiency:** Typically requires very little memory (only needs to store the current assignment).
- **Scalability:** Can often find good solutions for very large CSPs where systematic search methods would run out of time or memory.
- **Anytime Algorithm:** Can return a "best effort" solution if stopped at any time (though it might not be a perfect solution if conflicts still exist).

Disadvantages of Local Search for CSPs:

- **Incompleteness:** Not guaranteed to find a solution even if one exists, especially if it gets stuck in a local optimum.
- **Cannot Prove Unsatisfiability:** If it fails to find a solution, it cannot definitively prove that no solution exists (unlike backtracking).
- **Heuristic Dependent:** Performance heavily relies on the quality of the cost function and the choice of heuristics.

The Structure of Problems (for CSPs)

Understanding the structure of a CSP's constraint graph can be critical for designing efficient algorithms. Exploiting this structure can sometimes yield exponential reductions in complexity.

The **constraint graph** is a graphical representation where:

- Each **node** represents a variable.
- An **edge** connects two nodes if there is a constraint involving those two variables.

How problem structure impacts CSP solving ?

1. **Independent Sub problems (Connected Components):**
2. **Tree-Structured CSPs:**
3. **Nearly Tree-Structured CSPs (Tree Decomposition / Cutset Conditioning):**
4. **Constraint Graph Density/Sparsity:**

Knowledge-Based Agents

A **knowledge-based agent (KBA)** is an AI agent that decides what to do by **reasoning about its knowledge** of the world. Unlike simple reflex agents that directly map percepts to actions, KBAs maintain an internal representation of the world, often called a **knowledge base**, and use logical inference to derive conclusions and choose actions.

Components of a Knowledge-Based Agent:

- **Knowledge Base (KB):**
  - A set of sentences (statements) expressed in a **formal language**.
  - Represents the agent's beliefs about the world, its environment, its actions, and its goals.
  - Can contain facts (e.g., "The alarm is on"), rules (e.g., "If the alarm is on and the door is open, then there is an intruder"), and relationships.
- **Inference Engine:**
  - A set of procedures or algorithms that operate on the knowledge base.
  - Its role is to:
    - **Query the KB:** Ask questions about what is true or false.
    - **Derive new sentences:** Infer conclusions that are logically entailed by the existing knowledge.
    - **Update the KB:** Add new percepts and the consequences of actions to the knowledge base.

How a KBA Works (The TELL and ASK Interface):

- **TELL:** The agent's **percepts** (inputs from sensors) are added to the knowledge base. The effects of the agent's **actions** are also "told" to the KB.
- **ASK:** The agent's **inference engine** queries the KB to determine what action to perform. This might involve inferring whether a goal state has been reached, or which action is most likely to lead to a desired outcome

Why use Propositional Logic for KBAs?

- **Clarity and Formality:** Provides a precise and unambiguous way to represent statements and their relationships.

- **Inference Rules:** Well-defined rules of inference allow the agent to logically derive new conclusions from existing knowledge.

- **Completeness:** Inference procedures for Propositional Logic can be proven to be sound (only derive true conclusions from true premises) and complete (can derive all true conclusions).

Example: Wumpus World Agent

Limitations of Propositional Logic for KBAs :

- **Lack of Expressiveness:**

  - Cannot represent properties of objects (e.g., `Color(Car1, Red)`).
  - Cannot represent relationships between objects (e.g., `Parent(John, Mary)`).
  - Cannot quantify over individuals (e.g., "All birds can fly," "Some people are happy").
  - This leads to a large number of propositions even for simple domains.
  - **For example,** if you have 100 rooms and want to say "There is a pit in every room where there is a breeze," you'd need a separate rule for each room, leading to a huge KB.

- **Combinatorial Explosion:** For even moderately complex domains, the number of distinct propositions (and thus rows in a truth table) becomes astronomically large, making truth-table-based inference computationally intractable.

Transition to First-Order Logic (FOL):

Due to these limitations, real-world knowledge-based agents almost universally use **First-Order Logic (FOL)** (also known as Predicate Logic) as their formal language.

FOL extends propositional logic by introducing:

- **Predicates:** To represent properties and relations (e.g., `Is_Red(Car)`, `Parent(John, Mary)`).
- **Functions:** To map terms to other terms (e.g., `Leg(John)`).
- **Constants:** To represent specific objects (e.g., `John`, `Car1`).
- **Variables:** To stand for arbitrary objects (e.g., `x`, `y`).

- **Quantifiers:** ∀ (for all/universal) and ∃ (there exists/existential) to make general statements (e.g., `∀x. Bird(x) => Flies(x)` - "All birds fly").

The Wumpus World

The **Wumpus World** is a classic artificial intelligence problem that serves as a simplified, yet rich, environment for demonstrating and testing various AI concepts, particularly in the areas of **knowledge representation, logical reasoning (inference), planning, and decision-making in partially observable environments.**

It was introduced by Michael Genesereth and popularized in the textbook "Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig.

The Environment

The Wumpus World typically consists of a **4x4 grid of rooms** (caves) connected by passageways.

Key Elements in the World:

- **Agent:** The intelligent entity that navigates the world. Starts at a fixed location (usually [1,1]) facing a specific direction (usually right).
- **Wumpus:** A dangerous monster. If the agent enters the Wumpus's room, it dies. The Wumpus stays in one room. The agent has one arrow to shoot the Wumpus. If killed, the Wumpus emits a "scream."
- **Pits:** Bottomless pits. If the agent enters a room with a pit, it falls in and dies. Pits are stationary.
- **Gold:** The ultimate goal for the agent to find. It's in one room.
- **Walls:** The boundaries of the 4x4 grid.

Agent's Sensors (Percepts):

The agent does not have a direct "map" of the cave. It relies on local sensory information from adjacent squares (not diagonally adjacent):

- **Stench:** Perceived in a room if the **Wumpus is in an adjacent room**.
- **Breeze:** Perceived in a room if a **Pit is in an adjacent room**.
- **Glitter:** Perceived in the room **where the Gold is located**.
- **Scream:** Heard anywhere in the cave if the **Wumpus is killed**.
- **Bump:** Perceived if the agent tries to move into a wall.

Agent's Actuators (Actions):

The agent can perform the following actions:

- **Move Forward:** Moves the agent one square in the direction it's facing (if not blocked by a wall).

- **Turn Left (90 degrees counter-clockwise):** Changes the agent's orientation.
- **Turn Right (90 degrees clockwise):** Changes the agent's orientation.
- **Grab:** Picks up the Gold if it's in the current room.
- **Shoot:** Fires the single arrow in the direction the agent is facing. If the arrow hits the Wumpus, the Wumpus dies and a scream is heard. If it misses, the arrow is lost.
- **Climb Out:** Exits the cave from the starting square [1,1] (only possible if the agent has the gold to win).

Performance Measures

The agent's performance is typically measured by a score:

- **+1000 points:** For climbing out of the cave with the Gold.
- **-1000 points:** For being eaten by the Wumpus or falling into a Pit (game over).
- **-1 point:** For each action taken (encourages efficient paths).
- **-10 points:** For using the arrow (encourages careful shooting).

Key Properties of the Wumpus World Environment

The Wumpus World is chosen as a testbed because it exhibits several important properties relevant to AI research:

1. **Partially Observable:** The agent does not have a global view of the cave. It can only sense its immediate surroundings. This forces the agent to use **inference** to deduce information about hidden parts of the environment.
2. **Deterministic:** The outcomes of the agent's actions are predictable (e.g., if you move forward, you *will* end up in the next square unless there's a wall). There's no randomness in movement or the environment's response (unless specifically added for advanced variants).
3. **Static:** The Wumpus, Pits, and Gold do not move once the game starts. This simplifies reasoning, as the agent doesn't need to track moving objects.
4. **Sequential:** The agent's actions influence future states and percepts. A plan (sequence of actions) is required.
5. **Discrete:** The environment is a grid, and actions are distinct steps (move, turn, grab, shoot).

How AI Concepts are Applied

The Wumpus World is a perfect demonstration for:

- **Knowledge Representation:** How to represent facts about the world (e.g., "Breeze in [1,2]", "Pit is not in [1,1]"), and rules (e.g., "If there's a Breeze in [x,y], then a Pit must be in an adjacent square"). This is often done using **Propositional Logic** (for simpler examples) or more commonly **First-Order Logic** (for a more general and expressive representation).
- **Inference/Logical Deduction:** The agent uses its knowledge base and inference rules to deduce unobserved facts. For example, if it perceives a breeze in [1,2] and [2,1], and knows [1,1] is safe, it can infer where pits *might* be, and also where it *must* be safe to move.

- **Planning:** Based on its inferred knowledge, the agent needs to plan a sequence of actions to reach the gold and return safely. This involves exploring possible paths and considering the risks.
- **Decision-Making Under Uncertainty:** While the environment itself is deterministic, the *agent's knowledge* about it is uncertain (due to partial observability). The agent must make decisions based on probabilities or logical possibilities, minimizing risk.
- **PEAS Framework:** The Wumpus World is an excellent example to illustrate the PEAS (Performance, Environment, Actuators, Sensors) description of an intelligent agent.

Propositional Logic

Propositional Logic (PL) is the most fundamental and foundational logical system used in Artificial Intelligence.

What is Propositional Logic?

At its core, propositional logic deals with **propositions**, which are declarative statements that can be definitively assigned a truth value of either **True** or **False**, but not both. It's like the "yes or no" way of thinking for a computer.

Components of Propositional Logic:

1. **Atomic Propositions (Propositional Variables):**
   - These are the simplest, indivisible declarative statements. They are typically represented by uppercase letters like P,Q,R,S, or more descriptive names like `IsRaining`, `DoorIsOpen`, `AlarmOn`.
   - Examples: "The sky is blue," "It is raining," "2 + 2 = 4."
2. **Logical Connectives (Operators):**
   - These symbols are used to combine atomic propositions to form more complex **compound propositions**.
   - **Negation (NOT):** ¬P (e.g., "It is *not* raining") - Inverts the truth value.
   - **Conjunction (AND):** P∧Q (e.g., "It is raining *and* the sun is shining") - True only if both P and Q are true.
   - **Disjunction (OR):** P∨Q (e.g., "It is raining *or* the sun is shining") - True if at least one of P or Q is true (inclusive OR).
   - **Implication (IF...THEN):** P⇒Q (or P→Q) (e.g., "If it is raining, then the ground is wet") - False only if P is true and Q is false.
   - **Biconditional (IF AND ONLY IF / IFF):** P⇔Q (or P↔Q) (e.g., "The alarm is on if and only if there's an intruder") - True if P and Q have the same truth value.
3. **Syntax:**
   - Defines the rules for constructing well-formed formulas (WFFs) or sentences in propositional logic. It dictates how propositions and connectives can be combined legally.
   - E.g., `(P ∧ Q) ⇒ ¬R` is a valid sentence. `P Q ∧` is not.
4. **Semantics:**

- Defines the meaning of sentences by specifying how their truth values are determined based on the truth values of their atomic propositions.
- **Truth Tables:** A common way to define the semantics of connectives and to determine the truth value of compound propositions for all possible truth assignments to their atomic components.

How Propositional Logic is Used in AI ?

1. **Knowledge Representation (KR):**
   - PL provides a formal language to encode facts, rules, and relationships about a specific domain into an AI system's **knowledge base (KB)**.
   - **Encoding Facts:** Simple observations can be stored as true or false propositions (e.g., `Is_Door_Open`, `Has_Fever`).
   - **Modeling Rules/Relationships:** Conditional statements are crucial for representing "if-then" rules (e.g., `(Patient_Has_Fever ∧ Patient_Has_Cough) ⇒ Patient_Might_Have_Flu`).
2. **Logical Reasoning and Inference:**
   - The primary power of PL in AI is its ability to perform **inference** .
   - An AI agent can use a set of known facts (premises) in its KB to logically derive new conclusions that are entailed by those facts.
   - **Inference Rules:** Algorithms within an AI's "inference engine" apply rules like:
     - **Modus Ponens:** From `P` and `P ⇒ Q`, infer `Q`. (If "It's raining" is true, and "If it's raining then the ground is wet" is true, then "The ground is wet" is inferred.)
     - **Resolution:** A complete inference rule used to prove if a query is **entailed** by a KB, often by refutation.
   - This allows AI systems to:
     - **Answer queries:** "Is the patient infected?"
     - **Make decisions:** "If (Temperature > 37 AND Headache), then Prescribe_Painkillers."
     - **Problem-solving:** Deduce missing information needed to achieve a goal.
3. **Basic Decision-Making:**
   - In simple rule-based systems or expert systems, propositional logic rules can directly map conditions to actions.
   - Example: An AI for a smart home:
     - `(Motion_Detected ∧ NOT Light_On) ⇒ Turn_Light_On`
     - `(Temperature < 20 ∧ NOT Heater_On) ⇒ Turn_Heater_On`
4. **Game Playing (Simple States):**
   - For very simple games where the state can be described by a finite set of true/false propositions (e.g., specific cells occupied in Tic-Tac-Toe), propositional logic could be used to represent game rules and winning conditions.
5. **Boolean Logic in Computer Science:**
   - At a fundamental level, propositional logic is synonymous with Boolean logic, which is the basis for all digital circuits and computer operations. This underpins how computers represent and manipulate information at the hardware level.

Limitations of Propositional Logic in AI

1. **Limited Expressiveness:**
    - **Cannot represent relationships between objects:** You can't express "John is taller than Mary" as a single proposition that captures the relationship between John and Mary. You'd need a new proposition for every specific pairing (e.g., `John_Taller_Than_Mary`).
    - **Cannot represent properties of objects:** You can't say "Car1 is red." Again, you'd need `Car1_Is_Red`.
    - **Cannot quantify over individuals:** This is the most severe limitation. You cannot express general statements like:
        - "All birds can fly." (You'd need `Bird1_Can_Fly`, `Bird2_Can_Fly`, etc., for every individual bird).
        - "Some people are happy." (You can't say "there exists" a happy person).
    - This makes the KB grow unmanageably large for even moderately sized domains.
2. **Combinatorial Explosion:**
    - Inference methods based on truth tables become computationally intractable very quickly. For n propositional variables, a truth table has 2n rows. Even for n=30, 230 is over a billion, making it impossible to enumerate.
    - While proof-based methods like resolution are more efficient, they still face scaling challenges for KBs with many propositions.

The Need for First-Order Logic (FOL):

Because of these limitations, **First-Order Logic (FOL)**, also known as **Predicate Logic**, was developed and is widely used in AI for knowledge representation and reasoning. FOL extends propositional logic by introducing:

- **Predicates:** To represent properties and relations (e.g., `Red(Car1)`, `TallerThan(John, Mary)`).
- **Constants:** To represent specific objects (e.g., `Car1`, `John`).
- **Variables:** To stand for arbitrary objects (e.g., `x`, `y`).
- **Functions:** To map terms to other terms (e.g., `LegOf(John)`).
- **Quantifiers:** ∀ (universal quantifier, "for all") and ∃ (existential quantifier, "there exists") to make general statements (e.g., `∀x. Bird(x) ⇒ Flies(x)` - "For all x, if x is a bird, then x flies").

Propositional Theorem Proving:

Inference and proofs

In propositional logic, **theorem proving** is the process of determining whether a given sentence is logically entailed by a set of existing sentences (the knowledge base or premises). In other words,

If we assume all sentences in our knowledge base are true, must the theorem also be true?

This is achieved through **inference**, which is the process of deriving new sentences from existing ones.

1. Inference Rules

An **inference rule** is a template for deriving valid conclusions from a set of premises. A rule is **sound** if it only produces true conclusions when its premises are true.

Here are some common and important inference rules used in propositional logic:

- **Modus Ponens (Implication Elimination):**
  - **Rule:** From P and P⇒Q, infer Q.
  - **Example:**
    - Premise 1: "It is raining" (P)
    - Premise 2: "If it is raining, then the ground is wet" (P⇒Q)
    - Conclusion: "The ground is wet" (Q)
- **And-Elimination:**
  - **Rule:** From P∧Q, infer P. (Also, from P∧Q, infer Q.)
  - **Example:**
    - Premise: "It is sunny AND it is warm" (P∧Q)
    - Conclusion: "It is sunny" (P)
- **And-Introduction:**
  - **Rule:** From P and Q, infer P∧Q.
  - **Example:**
    - Premise 1: "The light is on" (P)
    - Premise 2: "The door is closed" (Q)
    - Conclusion: "The light is on AND the door is closed" (P∧Q)
- **Or-Introduction:**
  - **Rule:** From P, infer P∨Q. (Any Q can be added.)
  - **Example:**
    - Premise: "John is sick" (P)
    - Conclusion: "John is sick OR John is tired" (P∨Q)
- **Double Negation Elimination:**
  - **Rule:** From ¬¬P, infer P.
  - **Example:**
    - Premise: "It is NOT NOT true that the car is red" (¬¬P)

        - Conclusion: "The car is red" (P)
- **Unit Resolution:**
    - **Rule:** From P∨Q and ¬P, infer Q.
    - **Example:**
        - Premise 1: "The alarm is on OR the lights are out" (P∨Q)
        - Premise 2: "The alarm is NOT on" (¬P)
        - Conclusion: "The lights are out" (Q)
- **Resolution (General Resolution):**
    - **Rule:** From P∨Q and ¬Q∨R, infer P∨R. (This is the most powerful and general single inference rule in PL.)
    - **How it works:** If you have two clauses (disjunctions of literals), and one contains a literal Q while the other contains its negation ¬Q, you can "resolve" them by canceling out Q and ¬Q and forming a new clause with the remaining literals.
    - **Example:**
        - Premise 1: "It's cold OR it's raining" (C ∨ R)
        - Premise 2: "It's NOT raining OR the street is wet" (¬ R ∨ W)
        - Conclusion: "It's cold OR the street is wet" (C ∨ W)
    - Resolution is particularly useful when all sentences in the Knowledge Base and the negation of the query are converted into **Conjunctive Normal Form (CNF)**, which is a conjunction of disjunctions (clauses).

2. Proofs

A **proof** is a sequence of inference rule applications that leads from the initial premises (the knowledge base) to the desired conclusion (the theorem). If a proof exists, it means the theorem is **entailed** by the knowledge base.

Types of Proof Methods in AI:

1. **Direct Proof (Forward Chaining / Data-Driven):**
    - **Concept:** Starts with the known facts in the knowledge base and repeatedly applies inference rules to derive new facts until the goal sentence (theorem) is derived.
    - **Process:**
        1. Initialize a set of facts with the initial KB.
        2. Loop:
            - Find an inference rule whose premises are all present in the current set of facts.
            - Add the conclusion of that rule to the set of facts.
            - If the goal sentence is added, the proof is found.
            - If no new facts can be derived, and the goal sentence is not in the set, then it cannot be proven by this method.
    - **Use Cases:** Useful for systems that need to react to new information as it comes in, or for deriving all possible conclusions from a set of facts.
    - **Example:** Expert systems, rule-based agents.
2. **Indirect Proof (Backward Chaining / Goal-Driven):**
    - **Concept:** Starts with the goal sentence (the theorem) and tries to find a chain of inference rules that would allow it to be derived from the initial facts. It works backward from the goal.

- **Process:**

    1. Set the goal as the current "subgoal."
    2. If the subgoal is already in the KB (a known fact), then it's proven.
    3. Otherwise, find an inference rule whose conclusion matches the current subgoal.
    4. Set the premises of that rule as new subgoals.
    5. Recursively try to prove these new subgoals.
- **Use Cases:** Efficient for answering specific queries, as it only explores relevant parts of the KB.
- **Example:** Logic programming (like Prolog), diagnostic systems.

3. **Proof by Refutation (Resolution Algorithm):**
    - **Concept:** This is a very powerful and general proof method, especially when combined with the Resolution inference rule. It works by assuming the negation of the goal sentence is true, adding it to the knowledge base, and then trying to derive a **contradiction** (e.g., an empty clause, `false`). If a contradiction is derived, it means the initial assumption (negation of the goal) must be false, and therefore the original goal must be true.
    - **Process:**

        1. Convert all sentences in the Knowledge Base (KB) and the negation of the query ($\neg Q$) into **Conjunctive Normal Form (CNF)**. A CNF sentence is a conjunction of clauses, where each clause is a disjunction of literals (a literal is an atomic proposition or its negation).
        2. Place all CNF clauses into a set called `S`.
        3. Repeatedly apply the **Resolution inference rule** to pairs of clauses in `S`.
        4. Add the resulting new clauses (resolvents) to `S`.
        5. If the **empty clause** (representing `False` or a contradiction) is derived, then the query `Q` is entailed by the KB.
        6. If no new clauses can be derived, and the empty clause has not been derived, then the query `Q` is NOT entailed.
    - **Completeness:** Resolution is **refutation complete**, meaning if a sentence is entailed by a KB, resolution will eventually derive the empty clause.

Proof by Resolution

Proof by Resolution is a powerful and widely used method for automated theorem proving in propositional logic

It's based on the principle of **refutation**, meaning you try to prove a statement by showing that its negation leads to a contradiction.

1. **Assume the opposite:** To prove that a statement (let's call it Q) is logically entailed by your Knowledge Base (KB), you assume that Q is **false**. This means you add ¬Q (the negation of Q) to your KB.
2. **Look for a contradiction:** The goal then becomes to show that the augmented KB (KB ∧¬Q) is **unsatisfiable**
3. **Derive the Empty Clause:** Systematically applying the **Resolution inference rule** until you derive the **empty clause** .The empty clause represents a **contradiction.**
4. **Conclude the original statement is true:** If you derive the empty clause, it means your initial assumption (¬Q) must have been false, and therefore, the original statement Q must be true.

Steps for Proof by Resolution

step-by-step breakdown of the resolution refutation algorithm:

1. **Convert all sentences to Conjunctive Normal Form (CNF):**
    - **Conjunctive Normal Form (CNF)** is a standardized form for logical sentences. A sentence is in CNF if it is a **conjunction** (AND) of one or more **clauses**, where each clause is a **disjunction** (OR) of one or more **literals**.
    - A **literal** is an atomic proposition (e.g., P) or its negation (e.g., ¬P).
    - **Conversion Steps (General Idea):**
        1. **Eliminate implications (⇒) and biconditionals (⇔):**
            - $A \Rightarrow B \equiv \neg A \vee B$
            - $A \Leftrightarrow B \equiv (\neg A \vee B) \wedge (A \vee \neg B)$
        2. **Move negations inwards:** Use De Morgan's Laws and double negation elimination.
            - $\neg(A \wedge B) \equiv \neg A \vee \neg B$
            - $\neg(A \vee B) \equiv \neg A \wedge \neg B$
            - $\neg\neg A \equiv A$
        3. **Distribute ∨ over ∧:** This is the key step to get the "AND of ORs" form.
            - $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
            - $(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$

4. **Convert the conjunctions into a set of clauses:** Each conjunct in the CNF becomes a separate clause in your set.

2. **Negate the goal (query) and convert it to CNF:** Add this negated CNF to your set of clauses.
3. **Repeatedly apply the Resolution Rule:**
    - **The Resolution Rule:** Given two clauses C1 and C2, if C1 contains a literal L and C2 contains its complementary literal ¬L, then you can resolve them to produce a new clause (called the **resolvent**) that contains all the literals from C1 and C2 *except* L and ¬L.
        - Formally: $(A \vee L) \wedge (B \vee \neg L) \Longrightarrow A \vee B$
        - Here, A and B represent the remaining literals in the respective clauses.
    - **Process:**

        1. Select two clauses from your set that contain a complementary pair of literals (e.g., P and ¬P).
        2. Create a new clause (the resolvent) by combining all literals from the two selected clauses, *excluding* the complementary pair.
        3. Add this new resolvent clause to your set of clauses.
        4. Remove any duplicate literals within the new clause (e.g., $P \vee P \equiv P$).
        5. Discard any new clause that is a tautology (i.e., contains both a literal and its negation, e.g., $P \vee \neg P$). These clauses are always true and don't help in deriving a contradiction.

4. **Check for the Empty Clause:**
    - If, at any point, you derive the **empty clause** (□ or `False`), it means you've found a contradiction. This proves that the original goal (query) is entailed by the knowledge base.
    - If you can no longer apply the resolution rule (no more complementary literals to resolve) and you haven't derived the empty clause, then the goal is *not* entailed by the knowledge base.

Example: Proving a Statement by Resolution

Knowledge Base (KB):

1. P (It is raining)
2. $P \Rightarrow Q$ (If it is raining, then the ground is wet)

Goal (Q): Q (The ground is wet)

Proof Steps:

Step 1: Convert all sentences to CNF.

- KB1: P (Already a clause) $\Rightarrow$ {P}
- KB2: $P \Rightarrow Q \equiv \neg P \vee Q$ (Already a clause) $\Rightarrow$ {¬P,Q}

Step 2: Negate the goal and convert to CNF.

- Goal: Q
- Negated Goal: $\neg Q$ (Already a clause) $\Rightarrow$ {¬Q}

Step 3: Collect all clauses into a set S:

$S = \{ \{P\}, \{\neg P, Q\}, \{\neg Q\} \}$

Step 4: Repeatedly apply the Resolution Rule.

1. **Resolve {P} and {¬P, Q}:**
   - Complementary literals: P and ¬P.
   - Resolvent: {Q} (The P and ¬P cancel out).
   - Add {Q} to S. Now $S = \{ \{P\}, \{\neg P, Q\}, \{\neg Q\}, \{Q\} \}$
2. **Resolve {Q} (the new clause) and {¬Q}:**
   - Complementary literals: Q and ¬Q.
   - Resolvent: □ (The empty clause, since all literals cancel out).
   - Add □ to S.

Step 5: Check for the Empty Clause.

- We derived the empty clause (□).

Horn clauses and definite clauses

In Artificial Intelligence, particularly in the realm of knowledge representation and logical reasoning, **Horn clauses** and **definite clauses** are special forms of logical sentences that possess highly desirable computational properties. They are fundamental to logic programming languages like Prolog and enable very efficient inference mechanisms

Horn Clauses

A **Horn clause** is a disjunction (OR) of literals that has **at most one positive (unnegated) literal**.

In its general form, a Horn clause looks like: ¬P1∨¬P2∨…∨¬Pn∨Q

Where:

- P1,P2,…,Pn are positive literals (atomic propositions).
- Q is a single positive literal (or no positive literal at all).

This form is logically equivalent to an implication: (P1∧P2∧…∧Pn)⇒Q

Types of Horn clauses:

Definite Clauses: These are Horn clauses with **exactly one** In Artificial Intelligence, particularly in the realm of knowledge representation and logical reasoning, **Horn clauses** and **definite clauses** are special forms of logical sentences that possess highly desirable computational properties. They are fundamental to logic programming languages like Prolog and enable very efficient inference mechanisms.

Horn Clauses

A **Horn clause** is a disjunction (OR) of literals that has **at most one positive (unnegated) literal**.

In its general form, a Horn clause looks like: ¬P1∨¬P2∨…∨¬Pn∨Q

Where:

- P1,P2,…,Pn are positive literals (atomic propositions).
- Q is a single positive literal (or no positive literal at all).

This form is logically equivalent to an implication: (P1∧P2∧…∧Pn)⇒Q

Let's break down the types of Horn clauses:

1. **Definite Clauses:** (See next section for detailed explanation). These are Horn clauses with **exactly one positive literal**.
    - Example: `(P AND Q) => R` (equivalent to `¬P OR ¬Q OR R`)
2. **Facts (Unit Clauses):** A definite clause with **no negative literals**. It's simply a single positive literal.
    - Example: `P` (equivalent to `P` itself)
    - This represents a basic truth or assertion in the knowledge base.
3. **Goal Clauses (Query Clauses / Integrity Constraints):** A Horn clause with **no positive literals** (i.e., all literals are negative).
    - Example: `¬P1 OR ¬P2 OR ... OR ¬Pn`

Definite Clauses

A **definite clause** is a specific type of Horn clause that contains **exactly one positive literal**.

It can be expressed in two common ways:

1. **Disjunctive Form (CNF):** $\neg P1 \vee \neg P2 \vee \ldots \vee \neg Pn \vee Q$ Here, Q is the single positive literal.
2. **Implicational Form (Rule Form):** $(P1 \wedge P2 \wedge \ldots \wedge Pn) \Rightarrow Q$ This is often read as

   "If P1 AND P2 AND ... AND Pn are true, THEN Q is true."

    - The conjunction of Pi on the left side is called the **body** or **antecedent**.
    - Q on the right side is called the **head** or **consequent**.
    - If n=0 (the body is empty), then the clause is just `Q`, which is a **fact** (a definite clause without a body).

Examples of Definite Clauses:

- `Bird(Tweety)` (Fact: Tweety is a bird)
- `HasFeather(x) AND LaysEggs(x) => Bird(x)` (Rule: If x has feathers and lays eggs, then x is a bird)
- `Happy` (Fact: Happy is true)
- `BuyCoffee AND HasMoney => DrinkCoffee` (Rule)

Differences between Horn Clauses and Definite Clauses

Feature	Horn Clause	Definite Clause
Positive Literals	At most one (0 or 1)	Exactly one
Category	A broader category, includes definite clauses, facts, and goal clauses.	A specific type of Horn clause.
Purpose (General)	Used for efficient inference, including refutation.	Primarily for representing facts and rules (implications).
Examples	`P`, `¬P`, `¬P ∨ Q`, `¬P ∨ ¬Q`	`P`, `¬P ∨ Q`, `¬P ∨ ¬Q ∨ R`

Forward and backward chaining

In **Propositional Theorem Proving**, **forward chaining** and **backward chaining** are two fundamental and complementary inference mechanisms used by AI systems to derive conclusions from a knowledge base (KB) or to prove a specific goal.

Both methods are typically applied to knowledge bases composed of **definite clauses**

Forward Chaining (Data-Driven / Bottom-Up)

Concept: Forward chaining starts with the known facts in the knowledge base and repeatedly applies inference rules (primarily Modus Ponens) to derive all possible new facts. It continues until no new facts can be inferred, or until the specific goal is reached.

Analogy: Imagine you have a set of ingredients (facts) and a cookbook (rules). Forward chaining is like looking at your ingredients, seeing which recipes you can make, making them, and then seeing if those new dishes can be used as ingredients for *other* recipes, and so on. You keep cooking until you can't make anything new, or you've made the specific dish you were looking for.

Algorithm :

1. **Initialization:**
    - Maintain a set of `known_facts` (initially, all the facts given in the KB).
    - Maintain a list of `rules` in the KB.
    - Keep track of the `count` of premises for each rule that are not yet satisfied (initially, this is the number of literals in the rule's body).
2. **Iteration:**
    - Loop until no new facts can be derived:
        - For each rule `(P1 ^ ... ^ Pn) => Q` in the `rules` list:
            - If all premises (P_1,ldots,P_n) of the rule are present in `known_facts`:
                - If Q is not already in `known_facts`:
                - Add Q to `known_facts`.

        - If Q is the `goal`, then the proof is found.
        - Mark this rule as "fired" or "satisfied" to avoid redundant processing.
3. **Termination:**
    - If the `goal` is added to `known_facts`, return "Proven".
    - If no new facts can be added in an entire iteration, and the `goal` is not in `known_facts`, return "Not Proven".

Example:

- **KB:**
    1. `A`
    2. `B`
    3. `A ^ C => D`
    4. `B ^ D => E`
    5. `F ^ G => H`
- **Goal:** `E`

Forward Chaining Steps:

1. **Initial `known_facts`:** `{A, B}`
2. **Iteration 1:**
    - Rule 1 (`A`): Already a fact.
    - Rule 2 (`B`): Already a fact.
    - Rule 3 (`A ^ C => D`): `A` is known, but `C` is not. Cannot fire.
    - Rule 4 (`B ^ D => E`): `B` is known, but `D` is not. Cannot fire.
    - Rule 5 (`F ^ G => H`): `F` and `G` are not known. Cannot fire.
    - *No new facts added from rules in this specific order initially.*

Advantages of Forward Chaining:

- **Data-Driven:** Naturally reacts to new data or observations as they come in.
- **Generates All Possible Conclusions:** Useful when you need to know all consequences of a set of facts.
- **Good for Monitoring/Alerting Systems:** If certain conditions (facts) become true, trigger appropriate actions/alerts.
- **Intuitive for some problems:** Like building up a complex structure from basic components.

Disadvantages of Forward Chaining:

- **Can derive irrelevant conclusions:** It might derive many facts that are not relevant to the current goal, leading to inefficiency if the goal is specific.
- **Can be inefficient for specific queries:** If the KB is large and the goal is deep in the inference chain, it might spend a lot of time deriving unnecessary intermediate facts.

Backward Chaining (Goal-Driven / Top-Down)

Concept: Backward chaining starts with the goal (the statement to be proven) and attempts to find rules that could conclude that goal. It then recursively sets the premises of those rules as new subgoals. This process continues until all subgoals are either found to be known facts or are proven by further backward chaining.

Analogy: You want to bake a specific cake (goal). Backward chaining is like looking up the cake recipe, seeing what ingredients you need. For each ingredient, if you don't have it, you then look up *its* recipe to see how to make it (new subgoal), and so on, until all ingredients are either found in your pantry (known facts) or can be bought directly.

Example:

- **KB:**
  1. `A`
  2. `B`
  3. `C`
  4. `A ^ C => D`
  5. `B ^ D => E`
- **Goal:** `E`

Backward Chaining Trace:

1. **Goal:** `E`
2. **Look for rules that conclude `E`:**
   - Rule 5: `B ^ D => E` is the only rule.
3. **New Subgoals:** To prove `E`, we need to prove `B` AND `D`.
   - **Subgoal 1: `B`**
     - Is `B` in KB as a fact? Yes.
     - `B` is proven.
   - **Subgoal 2: `D`**
     - **Look for rules that conclude `D`:**
       - Rule 4: `A ^ C => D` is the only rule.
     - **New Subgoals:** To prove `D`, we need to prove `A` AND `C`.

        - **Sub-subgoal 1:** `A`
            - Is `A` in KB as a fact? Yes.
            - `A` is proven.
        - **Sub-subgoal 2:** `C`
            - Is `C` in KB as a fact? Yes.
            - `C` is proven.
    - Since `A` and `C` are both proven, `D` is proven.
4. Since `B` and `D` are both proven, `E` is proven.
5. **Conclusion:** `E` is proven.

Advantages of Backward Chaining:

- **Goal-Driven:** Only explores paths relevant to the goal, making it highly efficient for specific queries.
- **Interactivity:** Ideal for expert systems that need to ask clarifying questions or gather specific data to prove a hypothesis (e.g., in medical diagnosis: "Does the patient have a fever?").
- **Good for Diagnostic Systems:** Works well when diagnosing a problem, starting from the observed symptom (goal) and working back to the root cause.

Disadvantages of Backward Chaining:

- **May repeat computations:** If the same subgoal is encountered multiple times through different paths, it might be re-proven repeatedly (though caching can mitigate this).
- **Not suitable for "what-if" analysis:** Doesn't naturally generate all consequences of new facts.

Differences between Forward and Backward Chaining

Feature	Forward Chaining	Backward Chaining
Direction	Data-driven (Bottom-up)	Goal-driven (Top-down)
Starting Point	Known facts/premises	The goal/query to be proven
Process	Infers all possible conclusions	Infers only conclusions relevant to the goal
Efficiency	Good when all consequences are needed; can be inefficient for specific goals.	Efficient for specific queries; focuses search.
Questions?	Doesn't typically ask questions (just uses available data).	Can ask questions to gather missing facts/evidence for subgoals.
Applications	Monitoring, control, simulation, planning, generating all consequences of an action.	Diagnosis, expert systems, query answering, proof assistants, logical programming (Prolog).

Effective Propositional Model Checking

You're asking about **Effective Propositional Model Checking** as a method for propositional proving. This is a very important area, especially for the theoretical understanding and practical implementation of AI reasoning systems.

In propositional logic, a **model** is a truth assignment (an assignment of 'true' or 'false' to each atomic proposition) that makes a given sentence or a set of sentences true.

Propositional Model Checking is a method of proving entailment by checking whether a given sentence (the query or conclusion) is true in *all models* where the knowledge base (KB) is true.

KB ⊨α (KB entails α) if and only if every model of KB is also a model of α.(if α is true in every possible world where everything in KB is true, then KB entails α.).

The Process of Propositional Model Checking

The most straightforward (though often inefficient) way to perform propositional model checking is through **truth table enumeration**:

1. **Identify all atomic propositions:** List all unique atomic propositions (P1,P2,…,Pn) present in the knowledge base (KB) and the query (α).
2. **Construct a truth table:** Create a truth table with 2n rows, representing all possible truth assignments to these n atomic propositions.
3. **Evaluate KB for each row:** For each row (each truth assignment/model), determine whether the entire knowledge base (KB) is true.
4. **Evaluate the query for each row:** For each row, determine whether the query (α) is true.
5. **Check for entailment:**

- Iterate through the rows of the truth table.
- For every row where the **KB is true**, check if α is also true in that same row.
- If for *even one* row where KB is true, α is false, then KB does **NOT** entail α.
- If α is true in *all* rows where KB is true, then KB **DOES** entail α.

Example: Proving by Truth Table Model Checking

KB:

1. P
2. $P \Rightarrow Q$

Query (α): Q

Atomic Propositions: `P, Q` (n=2, so 22=4 rows)

P	Q	KB1 (P)	KB2 (P ⇒ Q)	KB (P ∧ (P ⇒ Q))	Query (Q)	Is KB ⇒ Q?
True	True	True	True	True	True	(KB True) ⇒ Q True
True	False	True	False	False	False	
False	True	False	True	False	True	
False	False	False	True	False	False	

- There is only one row where `KB` (i.e., `P ∧ (P ⇒ Q)`) is True (the first row).
- In that row, the `Query (Q)` is also True.
- Since the query `Q` is true in all models where KB is true, **KB entails Q**.

Agents Based on Propositional Logic

An AI agent, in the context of propositional logic, is essentially a **Knowledge-Based Agent (KBA)** whose internal knowledge representation language is Propositional Logic (PL).

Agent Architecture (Simplified for PL):

1. **Percepts:** The agent receives information about its environment (e.g., through sensors). These percepts are translated into atomic propositions (e.g., "It's raining," "Alarm is on," "Door is open").
2. **Knowledge Base (KB):** The agent maintains a collection of sentences (facts and rules) expressed in propositional logic. This KB represents its current beliefs about the world.
3. **Inference Engine:** This is the "brain" of the agent. It contains algorithms (based on propositional theorem proving techniques) to manipulate the KB, derive new conclusions, and answer queries.
4. **Actions:** Based on the conclusions drawn by the inference engine, the agent decides on and executes actions.

Life Cycle of a Propositional Logic-Based Agent:

The agent operates in a continuous cycle, often described as a **TELL-ASK cycle**:

1. **TELL:** The agent receives new percepts from its environment. It translates these percepts into new propositional sentences and adds them to its KB.
    - Example: If the agent's sensor detects motion, it adds `MotionDetected` to its KB.
    - It also *tells* its KB the consequences of its own actions (e.g., if it turned on a light, it adds `LightIsOn` to its KB).

2. **ASK:** The agent then "asks" its KB questions to decide what to do next. These questions are typically queries in the form of propositional sentences.
    - Example: `ASK(ShouldTurnOnLight?)`, `ASK(IsThereAnIntruder?)`
3. **Inference (Propositional Theorem Proving):** When the agent "asks" a question, the inference engine employs propositional theorem proving techniques to determine if the query is logically entailed by the current KB.
4. **Action Selection:** Based on the answer from the inference engine (`True` or `False`), the agent decides on its action.

Limitations of Propositional Logic for General Agents :

While foundational, agents based solely on propositional logic face significant limitations for real-world complexity:

1. **Lack of Expressiveness:** As discussed before, PL cannot represent properties of objects, relationships between objects, or general quantified statements ("All intruders are dangerous"). This means:
    - **Scaling Issues:** You'd need a separate proposition for every specific fact (e.g., `IntruderInRoom1`, `IntruderInRoom2`, `IntruderInRoom3` instead of `Intruder(RoomX)`).
    - **Redundant Rules:** Rules like "If there's an intruder, send an alert" would need to be replicated for every possible location or type of intruder if propositions are specific.
2. **Computational Intractability:** Even with "effective" model checking (SAT solvers), the number of atomic propositions can become too large for complex environments, leading to performance bottlenecks.
3. **Difficulty with Change:** Representing changes over time (e.g., actions causing state transitions) explicitly in PL can be cumbersome, often requiring a separate set of propositions for each time step.

Because of these limitations, practical knowledge-based agents for complex domains usually employ more expressive logics like **First-Order Logic (FOL)**, which allows for predicates, variables, and quantifiers.