Constraint Satisfaction Problems (CSP)

- Constraint Satisfaction Problems (CSP) play a crucial role in artificial intelligence (AI) as they help solve various problems that require decision-making under certain constraints.
- CSPs 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.

Components of Constraint Satisfaction Problems

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.

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.

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)($V1,V2,\dots,Vn$)**:** 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)($D1,D2,\dots,Dn$)**:** 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)($C1,C2,\dots,Cm$)**:** Constraints restrict the possible values that variables can take. Each constraint defines a rule or relationship between variables.
4. **ConstraintRepresentation:**
   Each constraint CiCi 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$V1$ and V2$V2$. A possible constraint could be V1≠V2$V1$□$=V2$, which means the values assigned to these variables must not be equal.
   - **Detailed Explanation:**
     - **Scope:** The variables V1$V1$ and V2$V2$.

- **Relation:** A list of valid value combinations where V1$V1$ is not equal to V2$V2$.

Types of Domains in CSP

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

Discrete Domain: It is an infinite domain which can have one state for multiple variables. For example, a start state can be allocated infinite times for each variable.

Finite Domain: It is a finite domain which can have continuous states describing one domain for one specific variable. It is also called a continuous domain.

Constraint Types in CSP

With respect to the variables, basically there are following types of constraints:

Unary Constraints: It is the simplest type of constraints that restricts the value of a single variable.

Binary Constraints: It is the constraint type which relates two variables. A value x2 will contain a value which lies between x1 and x3.

Global Constraints: It is the constraint type which involves an arbitrary number of variables.

Constraint Propagation: Inference in CSPs

Constraint propagation is a fundamental concept in constraint satisfaction problems (CSPs). A CSP involves variables that must be assigned values from a given domain while satisfying a set of constraints. Constraint propagation aims to simplify these problems by reducing the domains of variables, thereby making the search for solutions more efficient.

How Constraint Propagation Works

Constraint propagation works by iteratively narrowing down the domains of variables based on the constraints. This process continues until no more values can be eliminated from any domain. The primary goal is to reduce the search space and make it easier to find a solution.

Steps in Constraint Propagation

1. **Initialization**: Start with the initial domains of all variables.
2. **Propagation**: Apply constraints to reduce the domains of variables.

3. **Iteration**: Repeat the propagation step until a stable state is reached, where no further reduction is possible.

Example

Consider a simple CSP with two variables, X and Y, each with domains {1, 2, 3}, and a constraint X ≠ Y. Constraint propagation will iteratively reduce the domains as follows:

- If X is assigned 1, then Y cannot be 1, so Y's domain becomes {2, 3}.
- If Y is then assigned 2, X cannot be 2, so X's domain is reduced to {1, 3}.
- This process continues until a stable state is reached.

Algorithms for Constraint Propagation

- Several algorithms are used for constraint propagation, each with its strengths and weaknesses. Some common algorithms include:

Arc Consistency

- Arc consistency ensures that for every value of one variable, there is a consistent value in another variable connected by a constraint. This algorithm is often used as a preprocessing step to simplify CSPs before applying more complex algorithms.

Path Consistency

- Path consistency extends arc consistency by considering triples of variables. It ensures that for every pair of variables, there is a consistent value in the third variable. This further reduces the domains and simplifies the problem.

k-Consistency

- k-Consistency generalizes the concept of arc and path consistency to k variables. It ensures that for every subset of k-1 variables, there is a consistent value in the kth variable. Higher levels of consistency provide more pruning but are computationally more expensive.

Applications of Constraint Propagation

Constraint propagation is widely used in various AI applications. Some notable areas include:

Scheduling

- In scheduling problems, tasks must be assigned to time slots without conflicts. Constraint propagation helps by reducing the possible time slots for each task based on constraints like availability and dependencies.

Planning

- AI planning involves creating a sequence of actions to achieve a goal. Constraint propagation simplifies the planning process by reducing the possible actions at each step, ensuring that the resulting plan satisfies all constraints.

Resource Allocation

- In resource allocation problems, resources must be assigned to tasks in a way that meets all constraints, such as capacity limits and priority rules. Constraint propagation helps by narrowing down the possible assignments, making the search for an optimal allocation more efficient.

Backtracking search for CSPs

- Backtracking search, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.
- Commutativity: CSPs are all commutative. A problem is commutative if the order of application of any given set of actions has no effect on the outcome.
- Backtracking search: A depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.
- There is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.
- BACKTRACKING-SARCH keeps only a single representation of a state and alters that representation rather than creating a new ones.

Steps in Backtracking search

Initialization: Start with an empty assignment.

Selection: Choose an unassigned variable.

Assignment: Assign a value to the chosen variable.

Consistency Check: Check if the current assignment is consistent with the constraints.

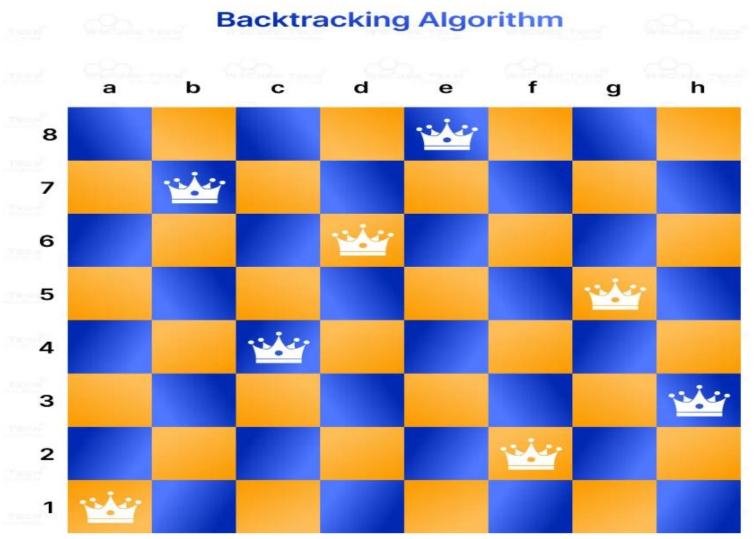
Recursion: If the assignment is consistent, recursively try to assign values to the remaining variables.

Backtrack: If the assignment is not consistent, or if further assignments do not lead to a solution, undo the last assignment (backtrack) and try the next possible value.

Examples of Backtracking Algorithms

1. **N-Queens Problem**

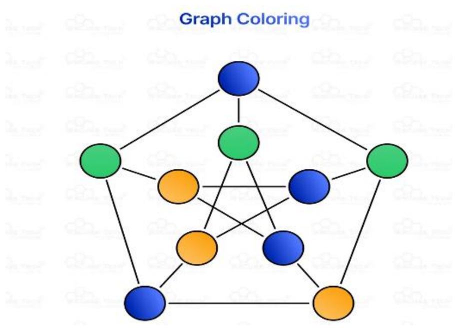
   - The N queen problem using backtracking algorithm involves placing N queens on an N×N chessboard such that no two queens threaten each other.
   - This means no two queens can share the same row, column, or diagonal.
   - The backtracking algorithm places queens one by one in different rows, checking for conflicts, and backtracking when a conflict is found until all queens are safely placed.



2. **Graph Coloring**

   - In the Graph Coloring problem, the goal is to color the vertices of a graph using the minimum number of colors so that no two adjacent vertices share the same colour.

   - The backtracking algorithm assigns colors to vertices one by one, ensuring that each color assignment is valid. If a conflict arises, it backtracks and tries a different color.



Logical agents

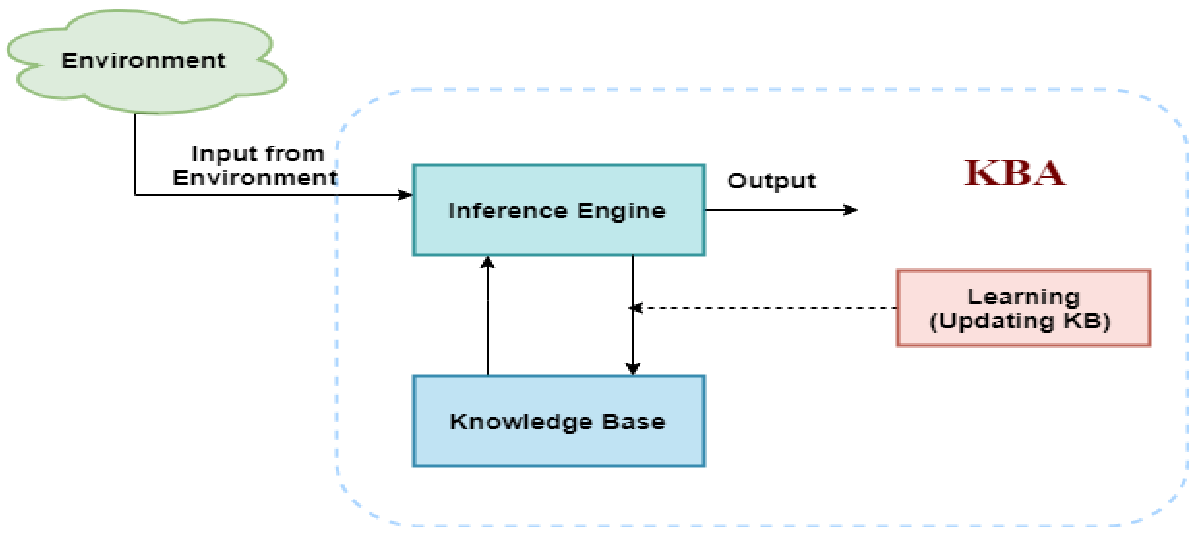
Knowledge–Based Agent

- 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 behavior:

1. **TELL:** This operation tells the knowledge base what it perceives from the environment.
2. **ASK:** This operation asks the knowledge base what action it should perform.
3. **Perform:** It performs the selected action.

Various levels of knowledge-based agent:

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

1. Knowledge level

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

2. Logical level:

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

3. Implementation level:

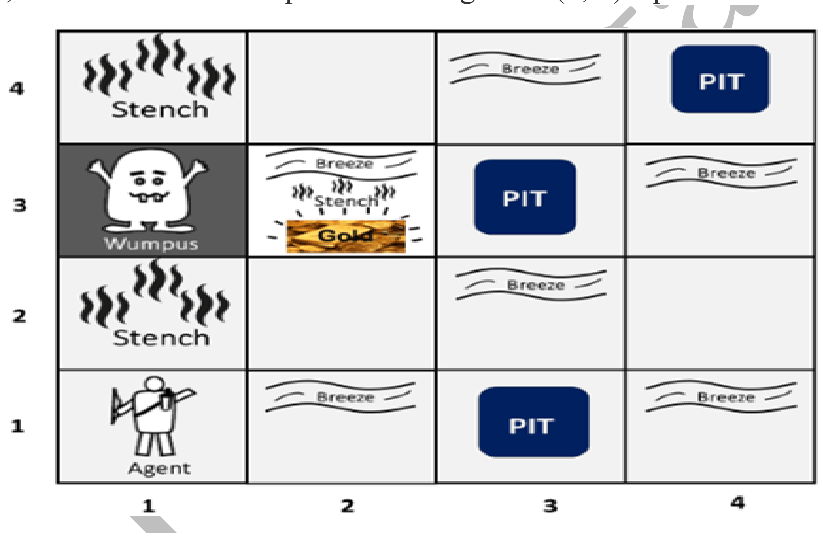
This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

The Wumpus World in Artificial intelligence

- The Wumpus world is a simple world example to illustrate the worth of a knowledge-based agent and to represent knowledge representation. It was inspired by a video game **Hunt the Wumpus** by Gregory Yob in 1973.
- The Wumpus world is a cave which has 4/4 rooms connected with passageways. So there are total 16 rooms which are connected with each other.

- We have a knowledge-based agent who will go forward in this world. The cave has a room with a beast which is called Wumpus, who eats anyone who enters the room.
- The Wumpus can be shot by the agent, but the agent has a single arrow. In the Wumpus world, there are some Pits rooms which are bottomless, and if agent falls in Pits, then he will be stuck there forever.
- The exciting thing with this cave is that in one room there is a possibility of finding a heap of gold.
- So the agent goal is to find the gold and climb out the cave without fallen into Pits or eaten by Wumpus.
- The agent will get a reward if he comes out with gold, and he will get a penalty if eaten by Wumpus or falls in the pit.

Following is a sample diagram for representing the Wumpus world. It is showing some rooms with Pits, one room with Wumpus and one agent at (1, 1) square location of the world.



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

- The rooms adjacent to the Wumpus room are smelly, so that it would have some stench.
- The room adjacent to PITs has a breeze, so if the agent reaches near to PIT, then he will perceive the breeze.
- There will be glitter in the room if and only if the room has gold.
- The Wumpus can be killed by the agent if the agent is facing to it, and Wumpus will emit a horrible scream which can be heard anywhere in the cave.

PEAS description of Wumpus world:

To explain the Wumpus world we have given PEAS description as below:

Performance measure:

- +1000 reward points if the agent comes out of the cave with the gold.
- -1000 points penalty for being eaten by the Wumpus or falling into the pit.
- -1 for each action, and -10 for using an arrow.
- The game ends if either agent dies or came out of the cave.

Environment:

- A 4*4 grid of rooms.
- The agent initially in room square [1, 1], facing toward the right.
- Location of Wumpus and gold are chosen randomly except the first square [1,1].
- Each square of the cave can be a pit with probability 0.2 except the first square.

Actuators:

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

Sensors:

- The agent will perceive the **stench** if he is in the room adjacent to the Wumpus. (Not diagonally).
- The agent will perceive **breeze** if he is in the room directly adjacent to the Pit.
- The agent will perceive the **glitter** in the room where the gold is present.
- The agent will perceive the **bump** if he walks into a wall.
- When the Wumpus is shot, it emits a horrible **scream** which can be perceived anywhere in the cave.
- These percepts can be represented as five element list, in which we will have different indicators for each sensor.
- Example if agent perceives stench, breeze, but no glitter, no bump, and no scream then it can be represented as:**[Stench, Breeze, None, None, None]**.

The Wumpus world Properties:

- **Partially observable:** The Wumpus world is partially observable because the agent can only perceive the close environment such as an adjacent room.
- **Deterministic:** It is deterministic, as the result and outcome of the world are already known.
- **Sequential:** The order is important, so it is sequential.
- **Static:** It is static as Wumpus and Pits are not moving.
- **Discrete:** The environment is discrete.
- **One agent:** The environment is a single agent as we have one agent only and Wumpus is not considered as an agent.

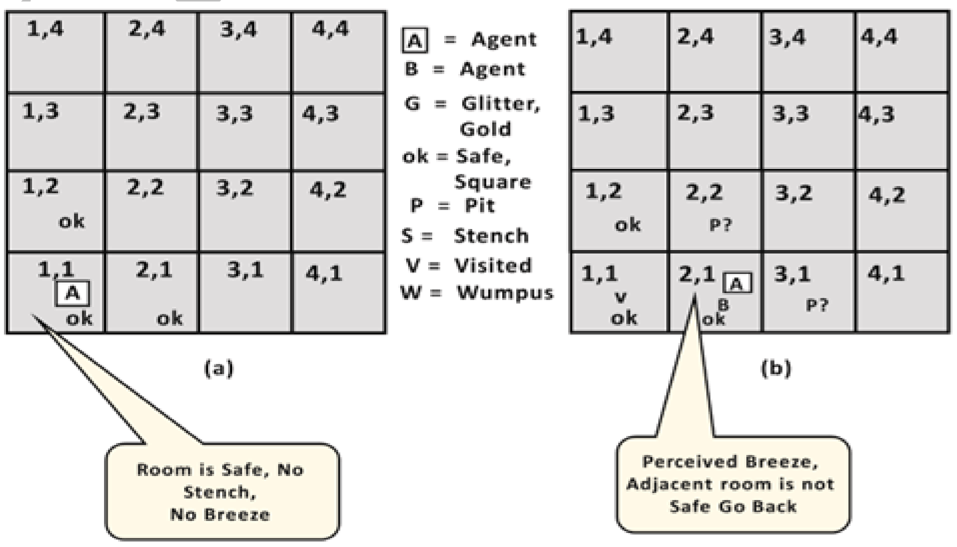
Exploring the Wumpus world:

Now we will explore the Wumpus world and will determine how the agent will find its goal by applying logical reasoning.

Agent's First step:

Initially, the agent is in the first room or on the square [1,1], and we already know that this room is safe for the agent, so to represent on the below diagram (a) that room is safe we will add symbol OK. Symbol A is used to represent agent, symbol B for the breeze, G for Glitter or gold, V for the visited room, P for pits, W for Wumpus.

At Room [1,1] agent does not feel any breeze or any Stench which means the adjacent squares are also OK.



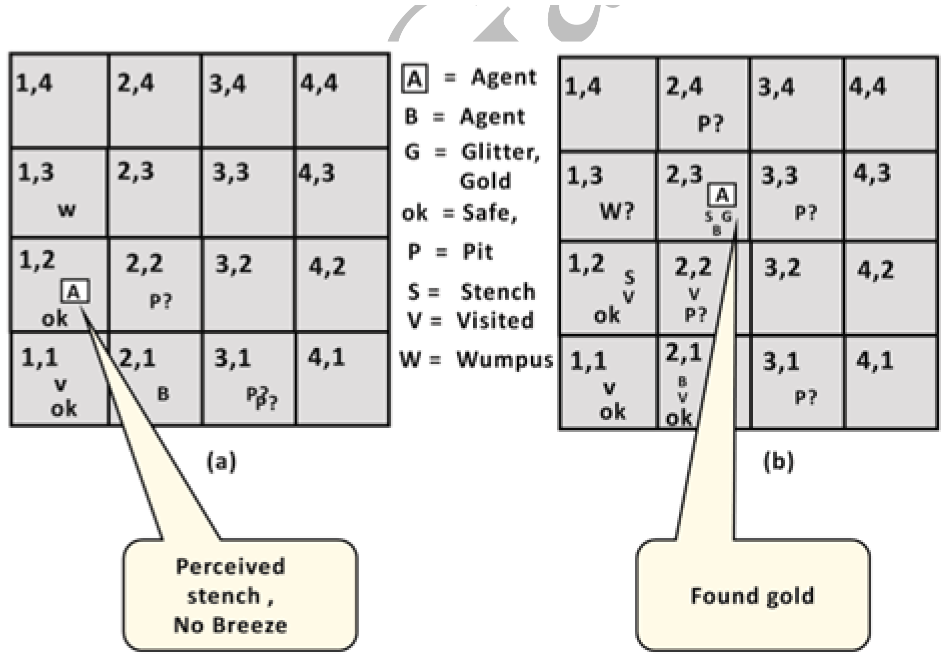
Agent's second Step:

Now agent needs to move forward, so it will either move to [1, 2], or [2,1]. Let's suppose agent moves to the room [2, 1], at this room agent perceives some breeze which means Pit is around this room. The pit can be in [3, 1], or [2,2], so we will add symbol P? to say that, is this Pit room?

Now agent will stop and think and will not make any harmful move. The agent will go back to the [1, 1] room. The room [1,1], and [2,1] are visited by the agent, so we will use symbol V to represent the visited squares.

Agent's third step:

At the third step, now agent will move to the room [1,2] which is OK. In the room [1,2] agent perceives a stench which means there must be a Wumpus nearby. But Wumpus cannot be in the room [1,1] as by rules of the game, and also not in [2,2] (Agent had not detected any stench when he was at [2,1]). Therefore agent infers that Wumpus is in the room [1,3], and in current state, there is no breeze which means in [2,2] there is no Pit and no Wumpus. So it is safe, and we will mark it OK, and the agent moves further in [2,2].



Agent's fourth step:

At room [2,2], here no stench and no breezes present so let's suppose agent decides to move to [2,3]. At room [2,3] agent perceives glitter, so it should grab the gold and climb out of the cave.

Propositional logic :a very simple logic

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

Example:

- a) It is Sunday.
- b) The Sun rises from West (False proposition)
- c) 3+3= 7(False proposition)
- d) 5 is a prime number.

Following are some basic facts about propositional logic:

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

lhar

Syntax of propositional logic:

There are two types of Propositions:

- **Atomic Propositions**
- **Compound propositions**

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

Example:

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

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

Example:

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

Logical Connectives:

- Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives.

There are mainly five connectives, which are given as follows:

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

Gangadhar

- **Biconditional:** A sentence such as **$P \Leftrightarrow Q$ is a Biconditional sentence, example If I am breathing, then I am alive**
  P= I am breathing, Q= I am alive, it can be represented as $P \Leftrightarrow Q$.

Following is the summarized table for Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
$\wedge$	AND	Conjunction	$A \wedge B$
$\vee$	OR	Disjunction	$A \vee B$
$\rightarrow$	Implies	Implication	$A \rightarrow B$
$\Leftrightarrow$	If and only if	Biconditional	$A \Leftrightarrow B$
$\neg$ or $\sim$	Not	Negation	$\neg A$ or $\neg B$

Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called **Truth table**. Following are the truth table for all logical connectives:

For Negation:

P	$\neg P$
True	**False**
False	**True**

For Conjunction:

P	Q	$P \wedge Q$
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

For disjunction:

P	Q	$P \vee Q$.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

P	Q	$P \rightarrow Q$
True	True	**True**
True	False	**False**
False	True	**True**
False	False	**True**

For Biconditional:

P	Q	P⇔ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**True**

Truth table with three propositions:

We can build a proposition composing three propositions P, Q, and R. This truth table is made-up of 8n Tuples as we have taken three proposition symbols.

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

Logical equivalence:

Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.

Let's take two propositions A and B, so for logical equivalence, we can write it as A⇔B. In below truth table we can see that column for ¬A∨ B and A→B, are identical hence A is Equivalent to B

A	B	¬A	¬AV B	A→B
T	T	F	**T**	**T**
T	F	F	**F**	**F**
F	T	T	**T**	**T**
F	F	T	**T**	**T**

Properties of Operators:

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

- (P ∨ Q) ∨ R= P ∨ (Q ∨ R)

- **Identity element:**
  - P ∧ True = P,
  - P ∨ True= True.
- **Distributive:**
  - P∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R).
  - P ∨ (Q ∧ R) = (P ∨ Q) ∧ (P ∨ R).
- **DE Morgan's Law:**
  - ¬ (P ∧ Q) = (¬P) ∨ (¬Q)
  - ¬ (P ∨ Q) = (¬ P) ∧ (¬Q).
- **Double-negation elimination:**
  - ¬ (¬P) = P.

Propositional theorem proving:

Inference:

artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, **so generating the conclusions from** In **evidence and facts is termed as Inference**.

Inference rules:

- Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.
- In inference rules, the implication among all the connectives plays an important role. Following are some terminologies related to inference rules:
- **Implication:** It is one of the logical connectives which can be represented as P → Q. It is a Boolean expression.
- **Converse:** The converse of implication, which means the right-hand side proposition goes to the left-hand side and vice-versa. It can be written as Q → P.
- **Contrapositive:** The negation of converse is termed as contrapositive, and it can be represented as ¬ Q → ¬ P.
- **Inverse:** The negation of implication is called inverse. It can be represented as ¬ P → ¬ Q.

From the above term some of the compound statements are equivalent to each other, which we can prove using truth table:

P	Q	P→Q	Q→P	¬Q→¬P	¬P→¬Q.
T	T	T	T	T	T
T	F	F	T	F	T
F	T	T	F	T	F
F	F	T	T	T	T

Hence from the above truth table, we can prove that P → Q is equivalent to ¬ Q → ¬ P, and Q→ P is equivalent to ¬ P → ¬ Q.

Types of Inference rules:

Modus Ponens:

The Modus Ponens rule is one of the most important rules of inference, and it states that if P and P → Q is true, then we can infer that Q will be true. It can be represented as:

Notation for Modus ponens: $\dfrac{P \rightarrow Q,\quad P}{\therefore Q}$

Example:

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q
Statement-2: "I am sleepy" ==> P
Conclusion: "I go to bed." ==> Q.
Hence, we can say that, if P→ Q is true and P is true then Q will be true.

Proof by Truth table:

P	Q	$P \rightarrow Q$
0	0	0
0	1	1
1	0	0
1	1	1

Modus Tollens:

The Modus Tollens rule state that if P→ Q is true and ¬ **Q is true, then ¬ P** will also true. It can be represented as:

Notation for Modus Tollens: $\dfrac{P \rightarrow Q,\quad \sim Q}{\sim P}$

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q
Statement-2: "I do not go to the bed."==> ~Q
Statement-3: Which infers that "**I am not sleepy**" => ~P

Proof by Truth table:

P	Q	~P	~Q	$P \to Q$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	0	0	1

Hypothetical Syllogism:

The Hypothetical Syllogism rule state that if P→R is true whenever P→Q is true, and Q→R is true. It can be represented as the following notation:

Example:

Statement-1: If you have my home key then you can unlock my home. **P→Q**
Statement-2: If you can unlock my home then you can take my money. **Q→R**
Conclusion: If you have my home key then you can take my money. **P→R**

Proof by truth table:

P	Q	R	$P \to Q$	$Q \to R$	$P \to R$
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	1	1

Disjunctive Syllogism:

The Disjunctive syllogism rule state that if P∨Q is true, and ¬P is true, then Q will be true. It can be represented as:

Notation of Disjunctive syllogism: $\dfrac{P \lor Q,\ \neg P}{Q}$

Example:

Statement-1: Today is Sunday or Monday. ==>P∨Q
Statement-2: Today is not Sunday. ==> ¬P
Conclusion: Today is Monday. ==> Q

Proof by truth-table:

P	Q	$\neg P$	$P \vee Q$
0	0	1	0
0	1	1	1 ←
1	0	0	1
1	1	0	1

Addition:

The Addition rule is one the common inference rule, and it states that If P is true, then P∨Q will be true.

Notation of Addition: $\dfrac{P}{P \vee Q}$

Example:

Statement: I have a vanilla ice-cream. ==> P
Statement-2: I have Chocolate ice-cream.
Conclusion: I have vanilla or chocolate ice-cream. ==> (P∨Q)

Proof by Truth-Table:

P	Q	$P \vee Q$
0	0	0
1	0	1 ←
0	1	1
1	1	1 ←

Simplification:

The simplification rule state that if **P∧ Q** is true, then **Q or P** will also be true. It can be represented as:

Notation of Simplification rule: $\dfrac{P \wedge Q}{Q}$ Or $\dfrac{P \wedge Q}{P}$

Proof by Truth-Table:

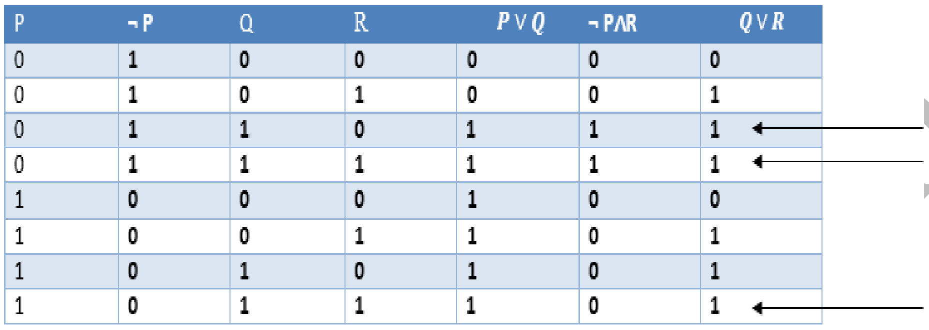
P	Q	$P \wedge Q$
0	0	0
1	0	0
0	1	0
1	1	1 ←

Resolution:

The Resolution rule state that if P∨Q and ¬ P∧R is true, then Q∨R will also be true. **It can be represented as**

Notation of Resolution $\dfrac{P \vee Q,\quad \neg P \wedge R}{Q \vee R}$

Proof by Truth-Table:



P	¬P	Q	R	$P \vee Q$	¬P∧R	$Q \vee R$
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	1
1	0	1	0	1	0	1
1	0	1	1	1	0	1

N.P. Gang ar