2019-20

Marks: 60

Answer any five Questions from the followings.

1. a) How is Machine Learning related to Artificial Intelligence? [3]

   b) Define Agent and Rational Agent through real-time example [3]

   c) Distinguish between the following properties of a task environment: [3]
      i. Static or dynamic
      ii. Discrete or continuous
      iii. Single or multi-agent

   d) What is local maxima? How to escape local maxima? [3]

2. a) Describe the function of vacuum cleaner as an Agent with suitable diagram. [4]

   b) Imagine, Mary is a model-based AI that builds and maintains an internal representation of the world. And Greg, on the other hand, is a goal-based AI that focuses on achieving specific objectives. [4]
   Describe the fundamental differences between Model-Based Mary and Goal-Based Greg in terms of their operational strategies. How does each approach affect their decision-making processes and adaptability to changes in their environment

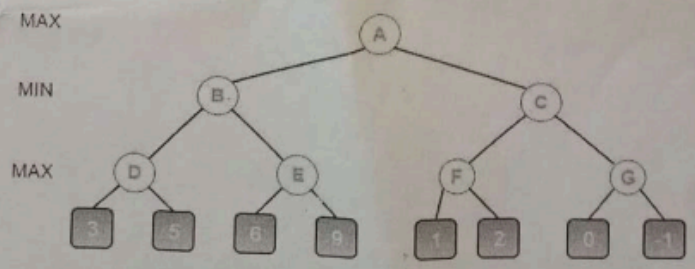
   c) Define Rational Agent with example. Is vacuum cleaner agent Rational? Why or why not? Explain with suitable reasons. [4]

3. a) Define in your own words the following terms: state, state space, search tree, search node, goal, action, transition model, and branching factor. [4]

   b) What is Greedy Best First Search? Explain with an example the different stages of Greedy Best First search. [4]

   c) Explain iterative deepening search with example. [4]

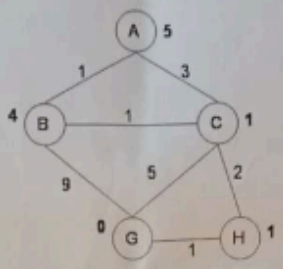
4. a) Explain the process of Minimax without alpha-beta pruning and how it determines the optimal move. [6]



   b) How does alpha-beta pruning improve the efficiency of the Minimax algorithm in evaluating game trees like the tree? [4]

   c) Explain the terms "max node" and "min node" in the Minimax algorithm. How are they used to represent players in a game? [2]

5. a) Consider the graph shown below where the numbers on the links are link costs and the numbers next to the states are heuristic estimates. Note that the arcs are undirected. Let A be the start state and G be the goal state. [6]

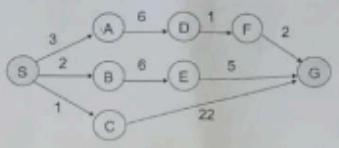


Simulate A* search with a strict expanded list on this graph. At each step, show the path to the state of the node that's being expanded, the length of that path, the total estimated cost of the path (actual + heuristic), and the current value of the expanded list (as a list of states).

b) Is the heuristic given in Problem 5.a admissible? Explain. [3]

c) Is the heuristic given in Problem 5.a consistent? Explain. [3]

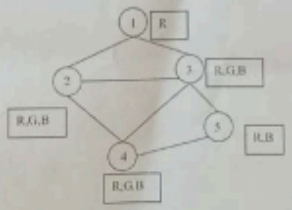
6. a) [8]



The graph above shows the step costs for different paths going from the start (S) to the goal (G). Use uniform cost search to find the optimal path to the goal.

b) Write down the advantage of IDS over BFS and DFS. Also, shows times and space complexity among them. [4]

7. a) Consider the following constraint graph for a graph coloring problem (the constraints indicate that connected nodes cannot have the same color). The domains are shown in the boxes next to each variable node. [8]



Now answer the following questions:

I. What are the variable domains after a full constraint propagation?

II. Show the sequence of variable assignments during a pure backtracking search (do not assume that the propagation above has been done), assume that the variables are examined in numerical order and the values are assigned in the order shown next to each node.

b) What is forward checking algorithm? Give an example. [4]

8. a) What is logic, syntax and semantics ? Write down some propositional logic syntax. [4]

b) Show that $p \rightarrow (q \rightarrow r)$ is logically equivalent to $(p \wedge q) \rightarrow r$ [4]

c) Translate each of the following sentences into First Order Logic (FOL) [4]

i. Not all cars have carburetors ii. All babies are illogical iii. Every connected and

1.a)Khata
b)khata
c)khata
2.a)khata
)b) Lusagine, Mary is a model-based AI that builds and maintains an internal representation of the world. And Greg, on the other hand, is a goal-based At that focuses on achieving specific @bjectives
Describe the fundamental differences between Model-Based Mary and Goal-Based Greg in lents of their operational strategies. How does each approach affect their decision-making processes and adaptability to changes in their environment?

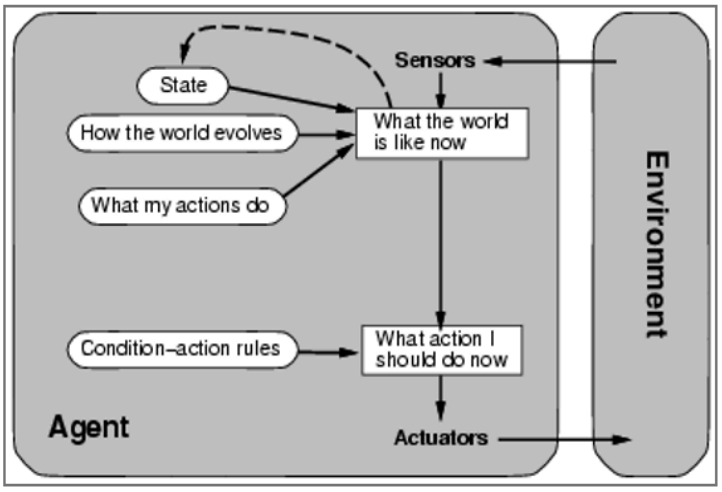
Ans:

- **Model-Based Mary** → Thinks before acting, predicts, and adapts via her internal model.

- **Goal-Based Greg** → Acts toward a defined goal, focusing on outcomes more than understanding the environment.

Model-based reflex agent

- Maintains internal state that keeps track of aspects of the environment that cannot be currently observed



How it works: Maintains an internal model of the world to handle partially observable environments.

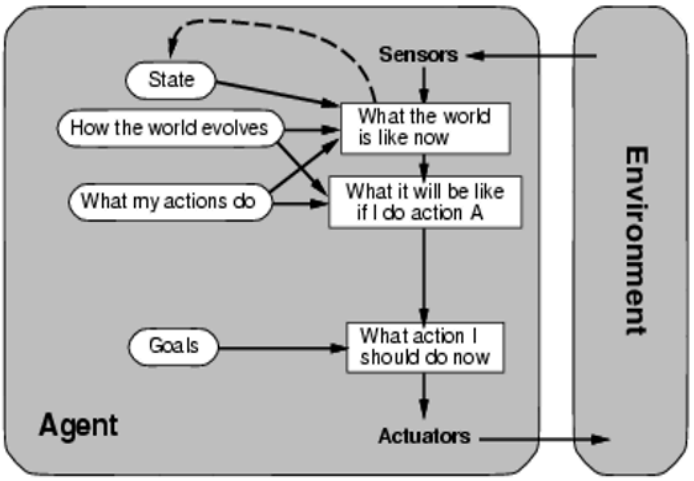
Example:

A vacuum cleaner robot:

Remembers which rooms it has cleaned and where dirt may still remain.
Self-parking car: Keeps track of car position and obstacles.

Goal-based agent

- The agent uses goal information to select between possible actions in the current state



Goal-Based Agent

How it works: Chooses actions by considering **future states** and whether they achieve a goal.

Example:

A GPS navigation system: Chooses the shortest path to the destination.
Chess AI: Selects moves that lead to checkmate.

Feature	Model-Based Mary	Goal-Based Greg
Knowledge	Maintains internal model of environment	Focuses on goal definitions
Decision Basis	Predictions from model	Goal satisfaction
Adaptability	High (can re-plan when model updates)	Medium (depends on goal flexibility)
Complexity	Higher computationally	Simpler and faster
Example Use	Autonomous planning, simulation	Search and optimization problems
Planning	Plans actions based on modeled state transitions.	Plans actions based on goal satisfaction.

C) Define Rational Agent Agent with example. Is vacuum cleaner agent Rational? Why or why not? Explain with suitable reasons.

A **Rational Agent** is an AI agent that **acts to achieve the best possible outcome** (or, when there is uncertainty, the best expected outcome) **based on its perception and knowledge** of the environment.

Formally:

> A rational agent chooses an action that **maximizes its performance measure**, given the percept sequence and its built-in knowledge.

•Performance measure (utility function):

An *objective* criterion for success of an agent's behavior

ExpectedUtility(action) = sum_outcomes Utility(outcome) * P(outcome)

Example of a Rational Agent:

Specifying the task environment

- Problem specification: **Performance measure, Environment, Actuators, Sensors (PEAS)**

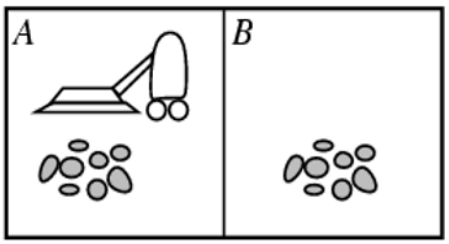
- **Example: automated taxi driver**
  - **Performance measure**
    - Safe, fast, legal, comfortable trip, maximize profits
  - **Environment**
    - Roads, other traffic, pedestrians, customers
  - **Actuators**
    - Steering wheel, accelerator, brake, signal, horn
  - **Sensors**
    - Cameras, sonar, speedometer, GPS, odometer, engine sensors, keyboard

Is the Vacuum Cleaner Agent Rational?

Yes, it can be rational — depending on its design and performance measure.

Back to vacuum-cleaner world

- **Percepts:**
  Location and status,
  e.g., [A,Dirty]
- **Actions:**
  Left, Right, Suck, NoOp



function Vacuum-Agent([location,status]) returns an action

- *if* status = Dirty *then* return Suck
- *else if* location = A *then* return Right
- *else if* location = B *then* return Left

- Is this agent rational?
  - Depends on performance measure, environment properties

Condition	Explanation
Percepts	The vacuum cleaner perceives whether the current square is dirty or clean and where it is located.
Actions	It can move **Left**, **Right**, or **Suck (clean)**.
Performance Measure	Keep the room as clean as possible using minimal actions and time.
Rational Behavior	If the vacuum cleaner sucks dirt when the square is dirty and moves efficiently to find dirt when clean, it acts rationally.

When It's *Not* Rational:

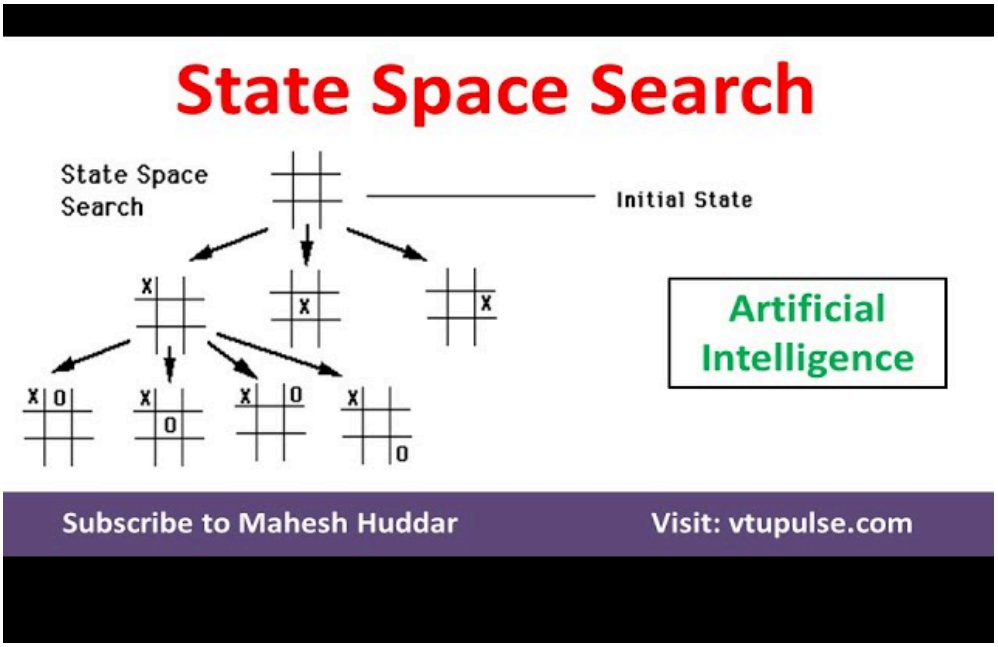
If the vacuum cleaner:

- **Cleans already clean areas repeatedly**, or
- **Moves randomly without sensing dirt**,
  then it's **not rational**, since it fails to maximize cleanliness efficiently.

3.a) Defire in your own words the following term: state, state space, search tree, search node. goal, action, transition model, and branching factor.

Term	Definition
State	A configuration of the problem
State Space	The set of all possible states
Search Tree	A tree structure representing the search process
Search Node	A node in the search tree
Goal	The state that the search process is trying to reach
Action	A choice that can be made in a state
Transition Model	Describes the result of taking an action in a state
Branching Factor	The average number of successors that any node in the search tree has

Term	Tic-Tac-Toe Example
State	The current layout of the Tic-Tac-Toe board (e.g., X in the center, O in the top-left).
State Space	All possible board configurations that can occur during the game (from empty board to full board).
Search Tree	The tree formed by all possible sequences of moves by X and O starting from an empty board.
Search Node	One specific board position in the search tree after a certain number of moves.
Goal	The winning state where a player gets three in a row (horizontal, vertical, or diagonal), or a draw if the board is full.
Action	Placing an X or O in an empty cell on the board.
Transition Model	The rule showing how the board changes after a move (e.g., X places in an empty cell → that cell now contains X).
Branching Factor	The number of possible moves from a state (e.g., at the start 9 possible moves, then 8, 7, etc.).



b)What is Greedy Best First Search? Explain with an example the different stages of Greedy Best First search

Greedy Best-First Search is a **search algorithm** that selects the next node to explore based on which node **appears to be closest to the goal**, according to a **heuristic function** h(n).

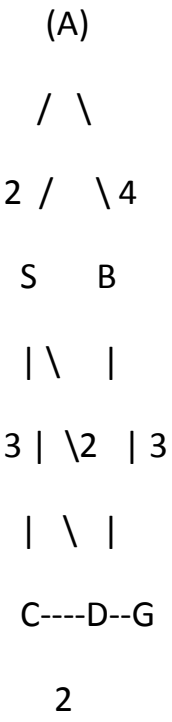
It is called *"greedy"* because it **always tries to expand the node that looks best at the moment**, without considering the total cost so far.

Expand the node that has the lowest value of the heuristic function $h(n)$.

Formula:

f(n)=h(n)Where:

- h(n) = heuristic estimate of the cost from node nnn to the goal.
- GBFS ignores the path cost g(n)); it only uses h(n)



Heuristic values h(n) (estimated straight-line distance to G):

node	S	A	B	C	D	G
h(n)	7	6	2	6	3	0

Step	Frontier (Nodes to Explore)	Chosen Node (Lowest h)	Action / Result
1	{S}	S (h=7)	Start from S
2	Expand S ⟶ {A(6), C(6), D(3)}	D (h=3)	Move to D
3	Expand D ⟶ {G(0)} + previous nodes	G (h=0)	Goal found ✅

S ⟶ D ⟶ G

Properties of greedy best-first search

- **Complete?**
  No – can get stuck in loops
- **Optimal?**
  No
- **Time?**
  Worst case: $O(b^m)$
  Best case: $O(bd)$ – If $h(n)$ is 100% accurate
- **Space?**
  Worst case: $O(b^m)$

C)Explain iterative deepening search with example.

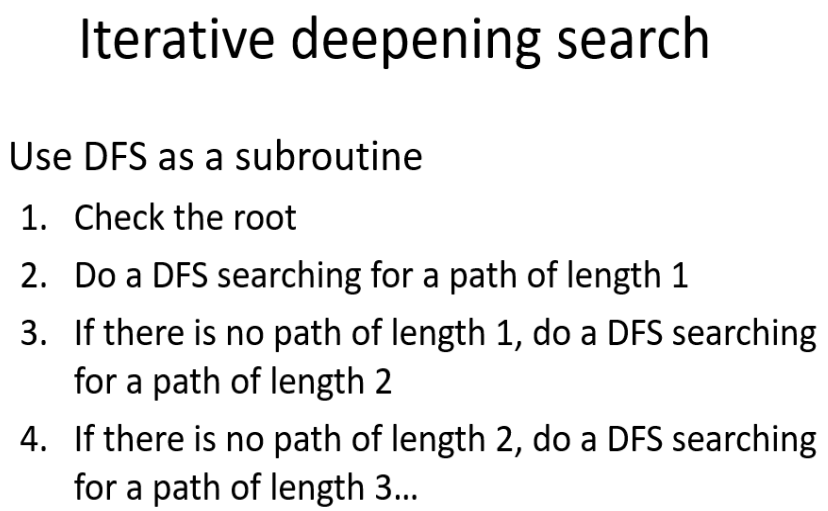
🧠 **What is Iterative Deepening Search (IDS)?**

Iterative Deepening Search (IDS) is a **search algorithm** that combines the advantages of **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**.

👉 It performs **repeated depth-limited searches**, increasing the limit each time until the goal is found.

⚙️ **How It Works:**

1. Start with depth limit = 0.
2. Perform a **Depth-Limited Search (DLS)** up to that depth.
3. If the goal is not found, increase the depth limit by 1 and repeat.
4. Continue until the goal is found.



🧩 **Why Use IDS?**

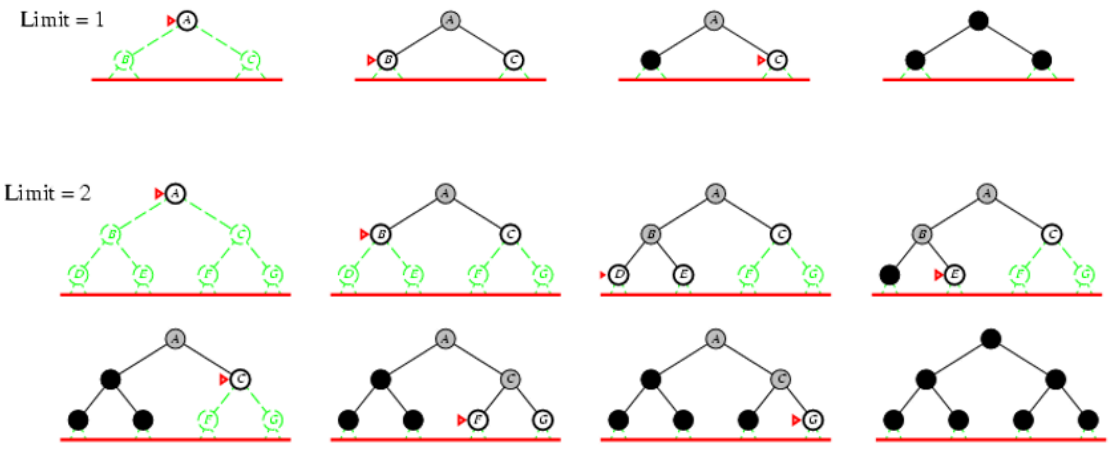
- Like **BFS**, it will **find the shallowest (optimal) goal**.
- Like **DFS**, it uses **low memory**.
- It avoids DFS's infinite loop problem and BFS's high memory usage.

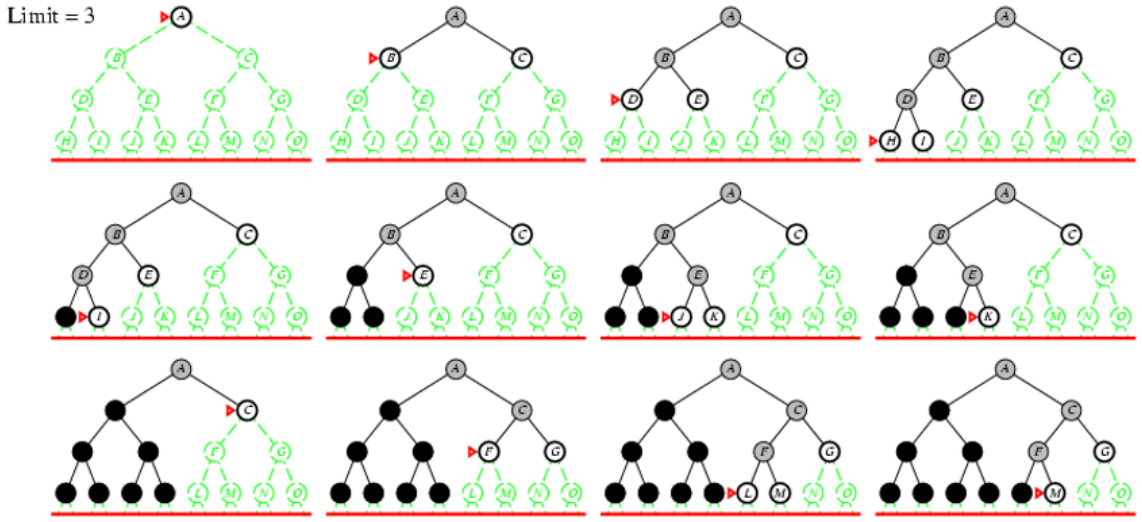
🌳 **Example:**

Consider this simple search tree:

```
    A
   / | \
  B  C  D
 / \   / \
E   F G   H
```

Goal: **Find node G**





🔁 Stages of Iterative Deepening Search

Iteration	Depth Limit	Nodes Visited (in order)	Goal Found?
1	0	A	❌ No
2	1	A, B, C, D	❌ No
3	2	A, B, E, F, C, D, G	✅ Yes (found at depth 2)

✅ Result:

The algorithm finds **G** at **depth 2**, after exploring nodes gradually deeper each time.

Feature	Description
Completeness	Yes, it always finds the goal if one exists.
Optimality	Yes, if all step costs are equal.
Time Complexity	$O(b^d)$ (like BFS)
Space Complexity	O(bd) (like DFS, very memory efficient)
Best Use	When the search space is large and depth of goal is unknown.

- **Complete?**

  Yes

- **Optimal?**

  Yes, if step cost = 1

- **Time?**

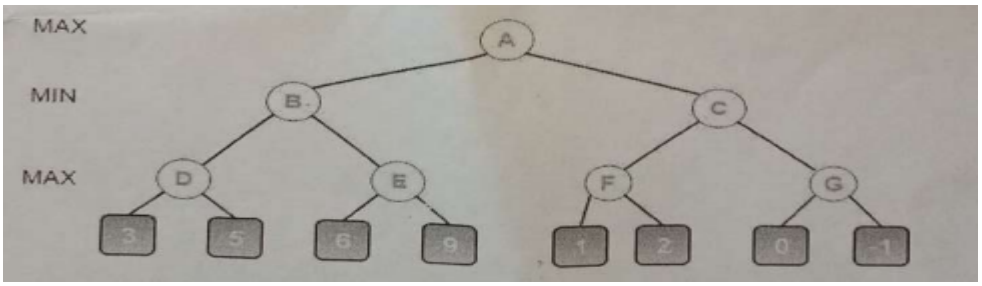
  $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

- **Space?** $O(bd)$

🧾 **In short:**

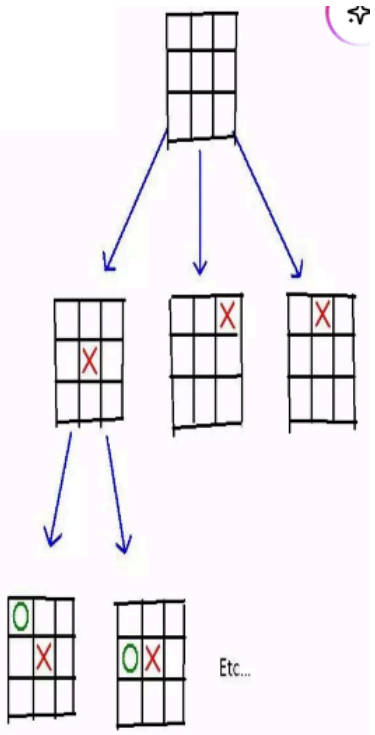
Iterative Deepening Search repeatedly performs depth-limited searches with increasing limits until the goal is found — combining the **low memory** of DFS and the **optimality** of BFS.

4.a)Explain the process of Minimax without alpha-beta pruning and how it determines the optimal move.



Minimax Algorithm (Without Alpha-Beta Pruning)

- **Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory.**
- **It provides an optimal move for the player assuming that opponent is also playing optimally.**
- **Mini-Max algorithm uses recursion to search through the game-tree.**
- **Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.**



Purpose:

Minimax is used in two-player zero-sum games (like Tic-Tac-Toe or Chess) to determine the optimal move by assuming that:

- MAX player tries to maximize the score.
- MIN player tries to minimize the score.

Process:

1. Start from the leaf nodes (terminal states) and assign their utility values.
2. Move up the tree, alternating between MIN and MAX levels:
    - At MAX nodes, choose the maximum value of child nodes.
    - At MIN nodes, choose the minimum value of child nodes.
3. Continue until the root node is assigned a value.
4. The optimal move for MAX is the child node that gives this value.

Step-by-Step Minimax Computation

Step 1: Evaluate leaf nodes

- Leaves are: 3, 5, 6, 9, 1, 2, 0, -1

Step 2: Compute MAX nodes (D, E, F, G)

- D = max(3, 5) = **5**
- E = max(6, 9) = **9**
- F = max(1, 2) = **2**
- G = max(0, -1) = **0**

Step 3: Compute MIN nodes (B, C)

- B = min(D, E) = min(5, 9) = **5**
- C = min(F, G) = min(2, 0) = **0**

Step 4: Compute MAX node (A)

- A = max(B, C) = max(5, 0) = **5**

Optimal Move

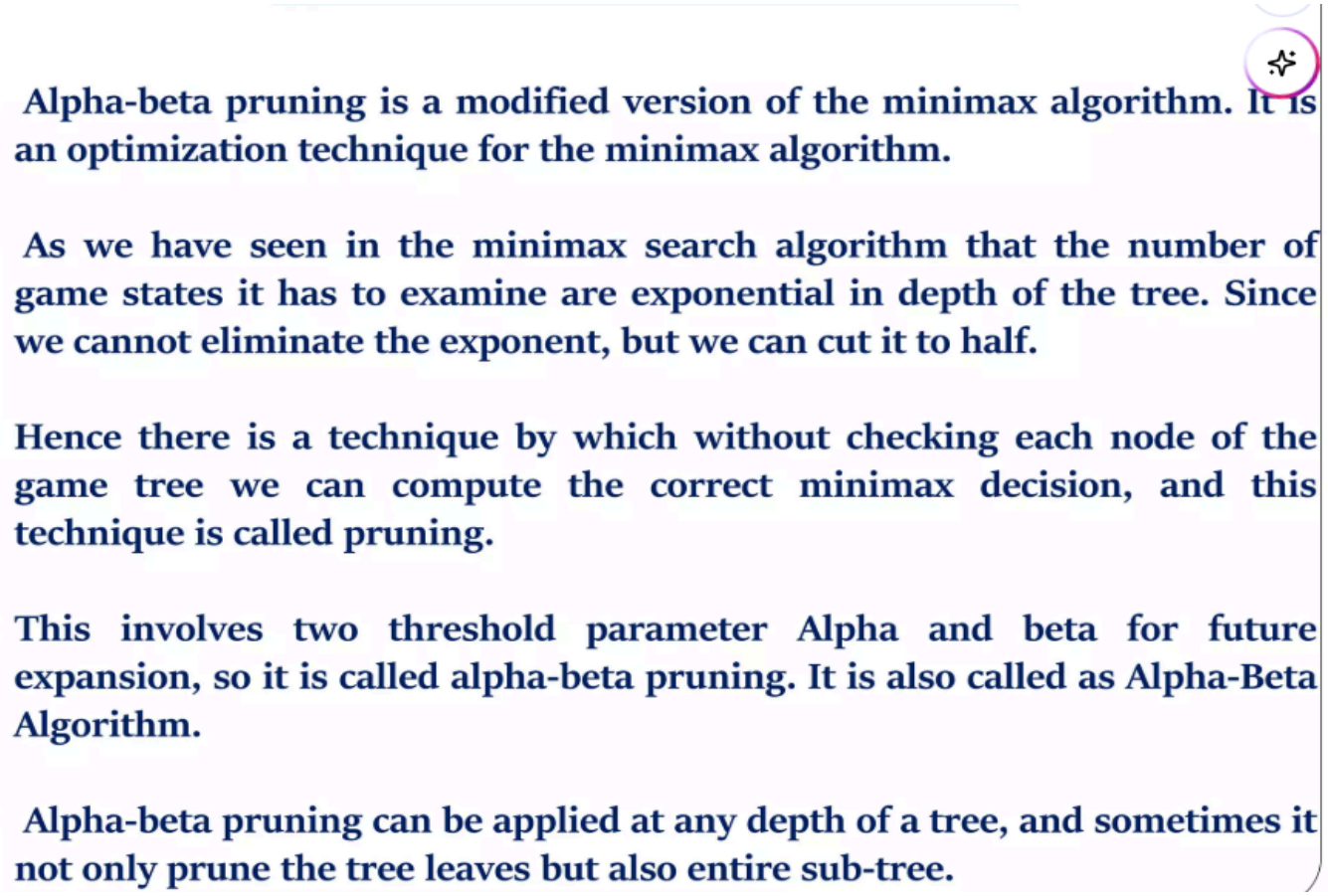
- **MAX (A)** should choose **B**, because it leads to the **highest guaranteed value (5)**.

Feature	Value
Optimal Move	A → B
Optimal Value	5
Completeness	Yes (tree is finite)
Time Complexity	O(b^m) = O(2^3) = O(8) nodes in this example
Space Complexity	O(m) = O(3) (depth of tree)

LIMITATION OF THE MINIMAX ALGORITHM

❖ The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide.

b) How does alpha-beta pruning improve the efficiency of the Minimax algorithm in evaluating game trees like the tree?



Alpha (α) = best value that **MAX** can guarantee along the current path.

Beta (β) = best value that **MIN** can guarantee along the current path.

While evaluating a node:

- If **current node value > β** at a MIN node → **stop exploring**, because MIN will avoid it.
- If **current node value < α** at a MAX node → **stop exploring**, because MAX will avoid it.

This is called **pruning** because the branch is **cut off early**.

Efficiency Gains

- Without pruning: Minimax evaluates **all nodes** → **O(b^m)** time complexity.
- With alpha-beta pruning:
    - Best-case time complexity: **O(b^(m/2))** (almost **square root** of nodes).
    - Worst-case: still **O(b^m)** if the tree is poorly ordered.
- Space complexity remains the same: **O(m)**.

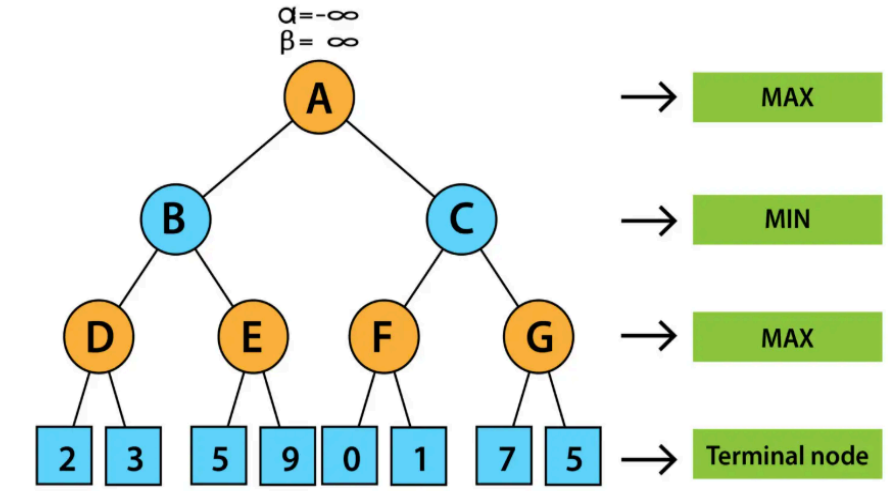
Without pruning: all 8 leaves evaluated.
With alpha-beta pruning (assuming good move ordering):

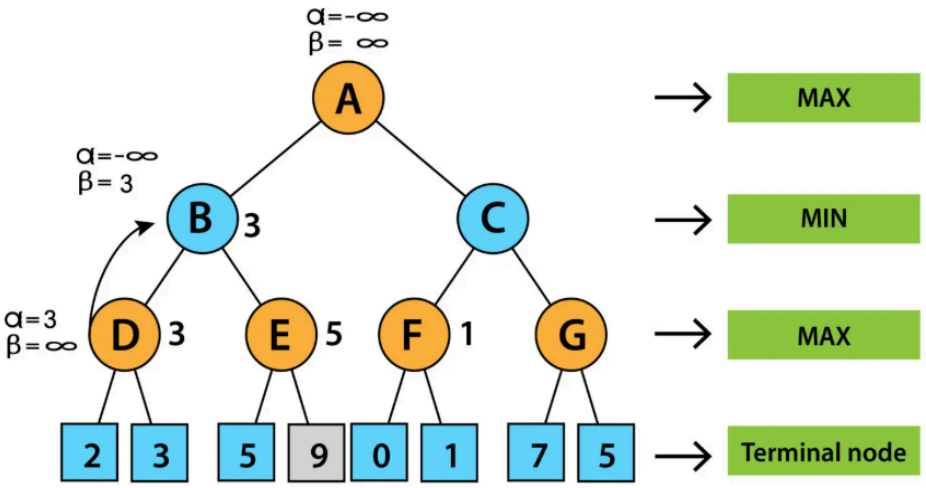
1. Start at **A** → **B** → **D**
    - D = max(3,5) = 5
    - α at A = 5
2. Next **E** under B: max(6,9) = 9
    - MIN(B) = min(5, 9) = 5 → β at B = 5
    - α = 5 → if we explore **C**, some branches may be pruned because **MAX already has 5**, so some values under C cannot improve the outcome for MAX.

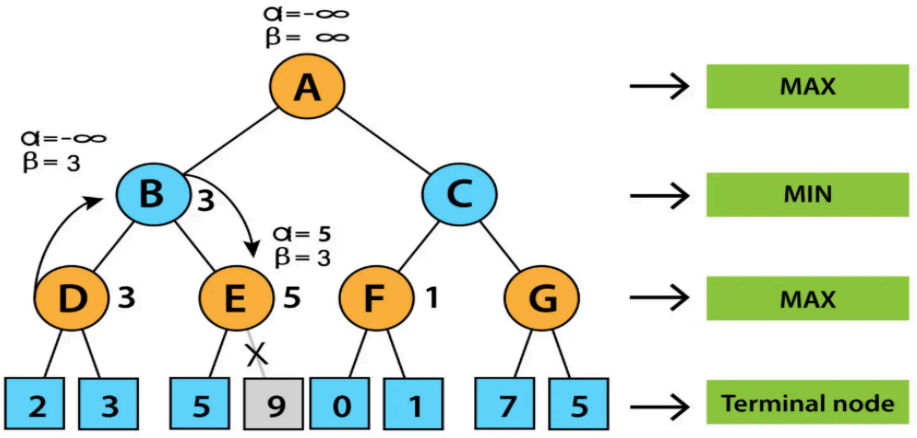
In effect, alpha-beta pruning **avoids exploring F and part of G** because they cannot yield a better result than **5** for MAX.

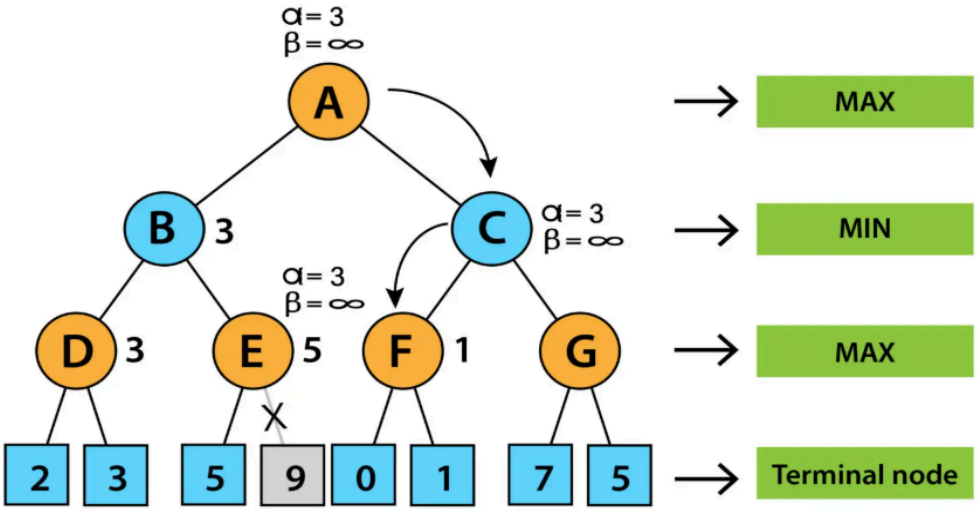
1. **Reduces computation:** fewer nodes are evaluated.
2. **Speeds up decision-making:** especially important for deeper game trees like Chess.

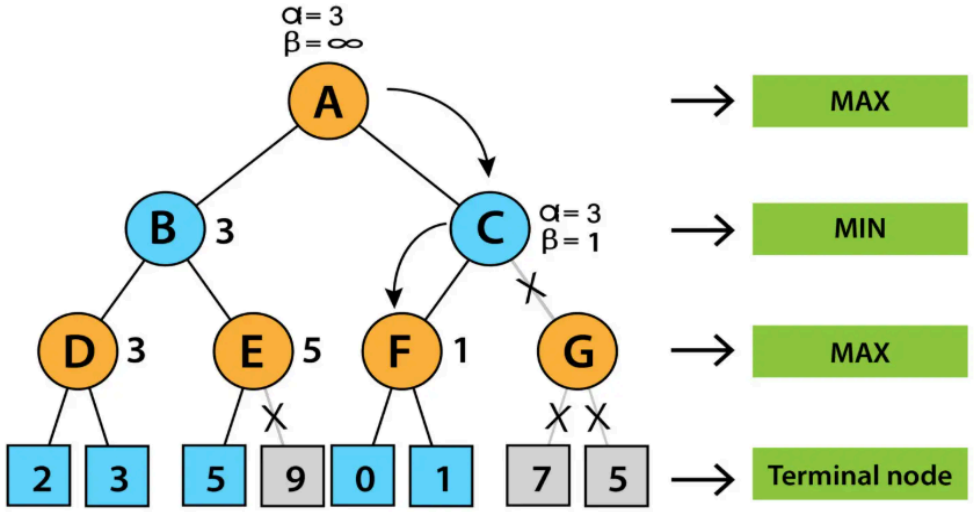
3. **Does not affect optimality:** the final Minimax value and move remain the same.

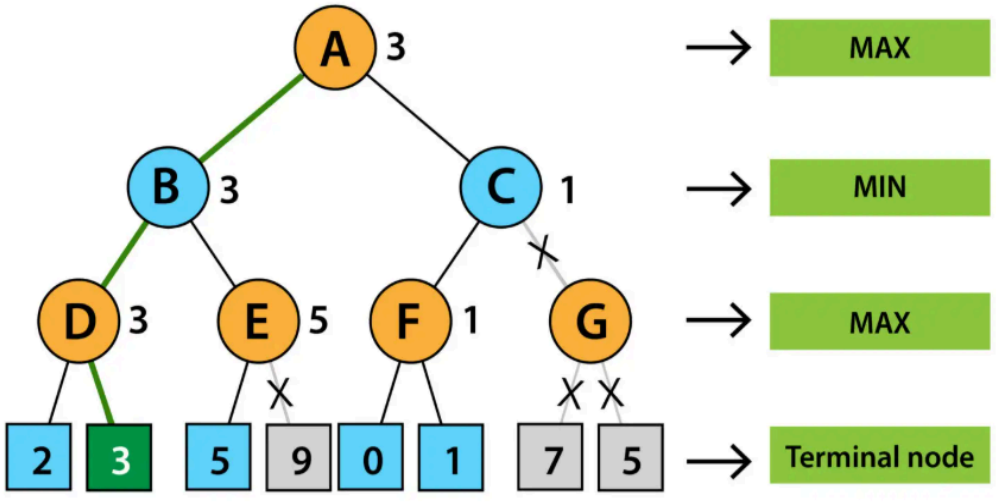












C) Explain the terms 'max node" and "min node" in the Minimax algorithm ,how they are used to represent players in a game?

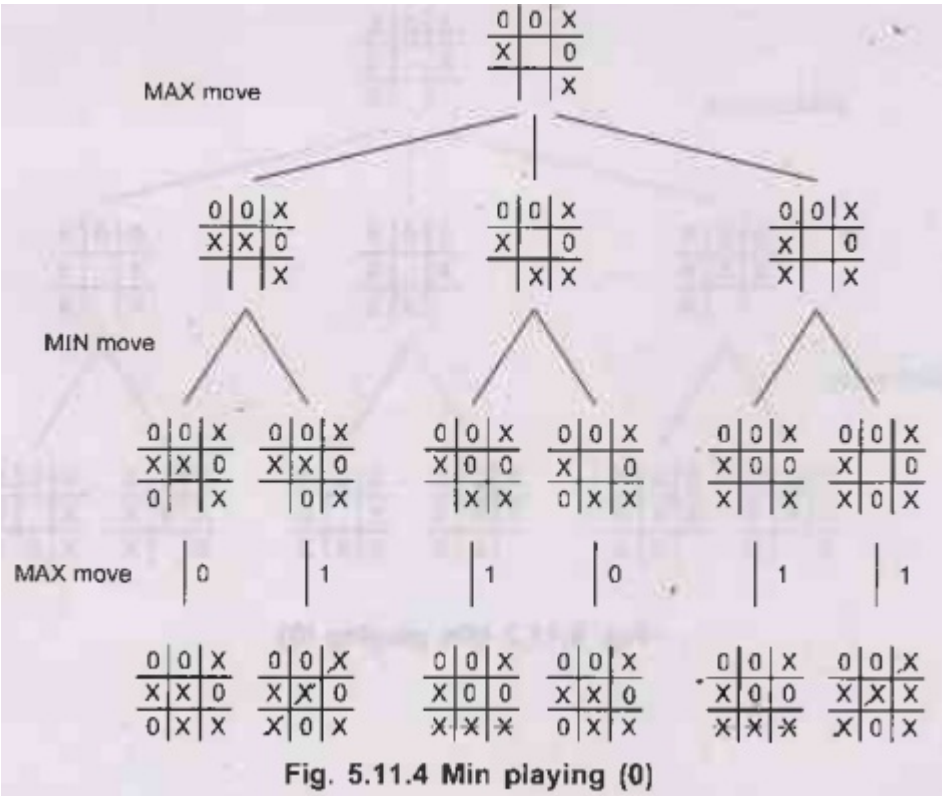


Fig. 5.11.4 Min playing (0)

🧠 Max Node and Min Node in Minimax Algorithm

In the **Minimax algorithm**, nodes in the game tree represent **game states**, and they are classified based on **which player's turn it is**.

1️⃣ Max Node

- **Represents:** The turn of the **MAX player** (the player trying to maximize the score).
- **Purpose:** Chooses the **child node with the maximum value**.
- **Role in Game:** MAX player is trying to select the move that **gives the best possible outcome**.
- **Example:** In Tic-Tac-Toe, if X is the MAX player, nodes where X makes a move are **max nodes**.

2️⃣ Min Node

- **Represents:** The turn of the **MIN player** (the opponent trying to minimize MAX's score).
- **Purpose:** Chooses the **child node with the minimum value**.
- **Role in Game:** MIN player tries to **prevent MAX from winning** or **minimize MAX's advantage**.
- **Example:** In Tic-Tac-Toe, if O is the MIN player, nodes where O makes a move are **min nodes**.

🔄 How They Work in Minimax

1. **Leaf nodes** contain the **utility value** of that game state.
2. At **MAX nodes**, the algorithm picks the **highest value** from its children.
3. At **MIN nodes**, it picks the **lowest value** from its children.
4. This alternation continues **up the tree** until the root, which determines the **optimal move for MAX**.

🌳 Visual Example (Partial Tree)

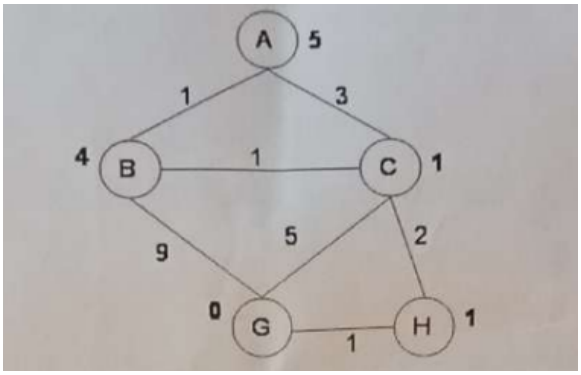
```
    A (MAX)
    /     \
B (MIN)    C (MIN)
```

- **A (MAX node):** MAX chooses between B and C to **maximize outcome**.
- **B, C (MIN nodes):** MIN chooses moves that **minimize MAX's score**.

✅ **In short:**

> **Max nodes** represent the moves of the player trying to **maximize the score**, and **Min nodes** represent the opponent's moves trying to **minimize that score**. This alternation models the **strategic behavior of both players** in a game tree.

5.a)Consider the graph shown below where the numbers on the links are link costs and the numbers next to the states are heuristic estimates. Note that the arcs are undirected. Let A be the start state and G be the goal state.



Simulate A search with a strictly expanded list on this graph. At each step, show the path to the state of the node that's being expanded, the length of that path, the total estimated cost of the path (actual heuristic), and the current value of the expanded list (as a list of states).

Simulate A* search with a strict expanded list on this graph. At each step, show the path to the state of the node that's being expanded, the length of that path, the total estimated cost of the path (actual + heuristic), and the current value of the expanded list (as a list of states). You are welcome to use scratch paper or the back of the exam pages to simulate the search. However, please transcribe (only) the information requested into the table given below.

Path to State Expanded	Length of Path	Total Estimated Cost	Expanded List
A	0	5	(A)
C-A	3	4	(C A)
B-A	1	5	(B C A)
H-C-A	5	6	(H B C A)
G-H-C-A	6	6	(G H B C A)

Assumptions / rules used

- Heuristic values read from the picture: (h(A)=5,; h(B)=4,; h(C)=1,; h(H)=1,; h(G)=0).

- Edge costs:A−B=1,A−C=3,B−C=1,B−G=9,C−G=5,C−H=2,H−G=1.
- Standard A* tie-breaks arbitrarily (no tie occurred that affected order here).

- “Strictly expanded list” = closed list: once a node is expanded it is not re-opened even if a cheaper path is later discovered.

Step-by-step A* expansions

I'll simulate A* on that graph and report, at each expansion, (1) the path to the node being expanded, (2) the path length (g), (3) the total estimated cost (f=g+h) (they called this the “total estimated cost”), and (4) the current expanded (closed) list.

Assumptions / rules used

- Heuristic values read from the picture: (h(A)=5,; h(B)=4,; h(C)=1,; h(H)=1,; h(G)=0).

- Edge costs: (A!-!B=1,; A!-!C=3,; B!-!C=1,; B!-!G=9,; C!-!G=5,; C!-!H=2,; H!-!G=1).

- Standard A* tie-breaks arbitrarily (no tie occurred that affected order here).

- “Strictly expanded list” = closed list: once a node is expanded it is not re-opened even if a cheaper path is later discovered.

Step-by-step A* expansions

1. **Expand: A**
    - Path to node: A

- (g = 0)
- (f = g + h = 0 + 5 = 5)
- Expanded list (closed): [A]

2. (When expanding A we add neighbors B and C to open:
   B: (g=1,; f=1+4=5).
   C: (g=3,; f=3+1=4).)

3. **Expand: C** (lowest (f=4))
   - Path to node: A - C
   - (g = 3)
   - (f = 3 + 1 = 4)
   - Expanded list: [A, C]

4. (Expanding C generates/updates:
   - B via C: (g=3+1=4) but B already in open with (g=1) (keep the better (g=1)).
   - G via C: (g=3+5=8,; f=8+0=8).
   - H via C: (g=3+2=5,; f=5+1=6).)

5. **Expand: B** (next lowest (f=5))
   - Path to node: A - B
   - (g = 1)
   - (f = 1 + 4 = 5)
   - Expanded list: [A, C, B]

6. (Expanding B would give C via B a path (g=1+1=2), which is better than C's current (g=3), **but** C is already in the closed list and we are following the strict rule of not reopening expanded nodes — so we ignore that improvement. B also gives G via B: (g=1+9=10) (worse than the existing G via C with (g=8)).)

7. **Expand: H** (next lowest (f=6), path from C)
    - Path to node: A - C - H
    - (g = 5)
    - (f = 5 + 1 = 6)
    - Expanded list: [A, C, B, H]

8. (Expanding H updates G via H to (g=5+1=6,; f=6+0=6); this is better than the previous G in open with (g=8), so G becomes (g=6,f=6).)

9. **Expand: G** (goal reached; lowest (f=6))
    - Path to node: A - C - H - G
    - (g = 6)
    - (f = 6 + 0 = 6)
    - Expanded list: [A, C, B, H, G]

Stop — goal expanded. The found path is A - C - H - G with cost (6).

Final result

- **Returned path:** A → C → H → G
- **Path cost (g):** 6
- **Expanded (in expansion order):** A, C, B, H, G

- b) is the heuristic given in Problem 5.a admissible? Explain.

Yes. The heuristic is admissible because it is less than or equal to the actual shortest distance to the goal.

To determine if the heuristic is **admissible**, I need to check if it **never overestimates** the actual cost to reach the goal from any node.

A heuristic h(n) is admissible if: **h(n) ≤ actual shortest path cost from n to G** for all nodes n.

Let me find the actual shortest paths from each node to G:

Actual shortest paths to G:

1. **From A to G:**
    - Path: A → C → H → G, Cost: 3 + 2 + 1 = **6**
    - h(A) = 5 ≤ 6 ✓
2. **From B to G:**
    - Path: B → C → H → G, Cost: 1 + 2 + 1 = **4**
    - h(B) = 4 ≤ 4 ✓
3. **From C to G:**
    - Path: C → H → G, Cost: 2 + 1 = **3**
    - h(C) = 1 ≤ 3 ✓
4. **From H to G:**
    - Path: H → G, Cost: **1**
    - h(H) = 1 ≤ 1 ✓
5. **From G to G:**
    - Cost: **0**
    - h(G) = 0 ≤ 0 ✓

Yes, the heuristic is admissible. ✓

C) Is the heuristic given in Problem 2 consistent? Explain.

No, the heuristic is not consistent. There are two places in the graph where consistency fails. One is between A and C where the drop in heuristic is 4, but the path length is only 3. The other is between B and C where the drop in heuristic is 3 but the path length is only 1.

To determine if the heuristic is **consistent** (also called monotone), I need to check if it satisfies the triangle inequality for all edges.

A heuristic h(n) is consistent if: **$h(n) \leq c(n, n') + h(n')$** for every edge from node n to neighbor n', where c(n, n') is the edge cost.

Let me check all edges in the graph:

Checking consistency for each edge:

1. **Edge A → B (cost = 1):**
    - $h(A) \leq c(A,B) + h(B)$?
    - $5 \leq 1 + 4 = 5$ ✓
2. **Edge A → C (cost = 3):**
    - $h(A) \leq c(A,C) + h(C)$?
    - $5 \leq 3 + 1 = 4$
    - **$5 \leq 4$ is FALSE ✗**
3. **Edge B → C (cost = 1):**
    - $h(B) \leq c(B,C) + h(C)$?
    - $4 \leq 1 + 1 = 2$
    - **$4 \leq 2$ is FALSE ✗**
4. **Edge B → G (cost = 9):**
    - $h(B) \leq c(B,G) + h(G)$?
    - $4 \leq 9 + 0 = 9$ ✓
5. **Edge C → H (cost = 2):**
    - $h(C) \leq c(C,H) + h(H)$?
    - $1 \leq 2 + 1 = 3$ ✓
6. **Edge C → G (cost = 5):**

- h(C) ≤ c(C,G) + h(G)?
- 1 ≤ 5 + 0 = 5 ✓

7. **Edge H → G (cost = 1):**

- h(H) ≤ c(H,G) + h(G)?
- 1 ≤ 1 + 0 = 1 ✓

Conclusion:

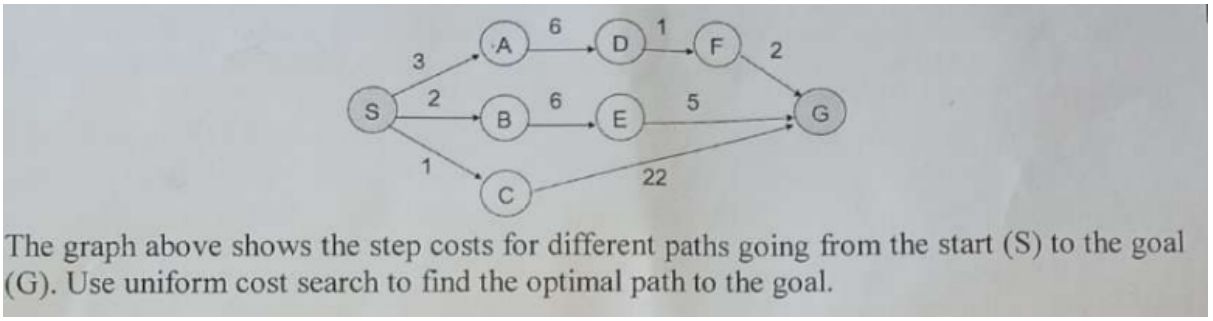
No, the heuristic is NOT consistent. ✗

The heuristic violates the consistency condition on two edges:

- **A → C:** h(A) = 5 > 4 = c(A,C) + h(C)
- **B → C:** h(B) = 4 > 2 = c(B,C) + h(C)

This means the heuristic is **admissible but not consistent**.

6.a)



The graph above shows the step costs for different paths going from the start (S) to the goal (G). Use uniform cost search to find the optimal path to the goal.

Let's run **Uniform-Cost Search (UCS)** on the drawn graph and find the optimal path S → G.

Graph edges (interpreted from picture)
S–A:3, S–B:2, S–C:1
A–D:6, D–F:1, F–G:2
B–E:6, E–G:5
C–G:22

Start: (S) with cost 0.

Step 0 — initialize

- Open (frontier): `S(0)`
- Closed (expanded): `[]`

Step 1 — expand S

- Expand: **S**
- Path to node: `S` , (g=0)
- Add neighbors to open:
    - A via S: `S–A` cost (3) → A(3)
    - B via S: `S–B` cost (2) → B(2)
    - C via S: `S–C` cost (1) → C(1)
- Open (sorted by cost): `C(1), B(2), A(3)`
- Closed: `[S]`

Step 2 — expand C (lowest cost)

- Expand: **C**
- Path: `S–C` , (g=1)
- Successors: G via C cost (1 + 22 = 23) → add G(23)
- Open: `B(2), A(3), G(23)`

- Closed: `[S, C]`

Step 3 — expand B

- Expand: **B**
- Path: `S–B` , (g=2)
- Successors: E via B cost (2 + 6 = 8) → add E(8)
- Open: `A(3), E(8), G(23)`
- Closed: `[S, C, B]`

Step 4 — expand A

- Expand: **A**
- Path: `S–A` , (g=3)
- Successors: D via A cost (3 + 6 = 9) → add D(9)
- Open: `E(8), D(9), G(23)`
- Closed: `[S, C, B, A]`

Step 5 — expand E

- Expand: **E**
- Path: `S–B–E` , (g=8)

- Successors: G via E cost (8 + 5 = 13).
    - G already in open at 23 → update to better cost 13.
- Open: `D(9), G(13)`
- Closed: `[S, C, B, A, E]`

Step 6 — expand D

- Expand: **D**
- Path: `S–A–D` , (g=9)
- Successors: F via D cost (9 + 1 = 10) → add F(10)
- Open: `F(10), G(13)`
- Closed: `[S, C, B, A, E, D]`

Step 7 — expand F

- Expand: **F**
- Path: `S–A–D–F` , (g=10)
- Successors: G via F cost (10 + 2 = 12).
    - G currently in open at 13 → update to better cost 12.
- Open: `G(12)`
- Closed: `[S, C, B, A, E, D, F]`

Step 8 — expand G (goal)

- Expand: **G**
- Path: `S–A–D–F–G` , (g=12)
- Closed: `[S, C, B, A, E, D, F, G]`
  Stop — goal expanded.

Result

- **Optimal path found by UCS:** `S → A → D → F → G`
- **Total path cost:** (12)
- **Order of expansions (closed list):** `S, C, B, A, E, D, F, G`

(UCS updated the best G path as better routes were discovered; the final best route is S–A–D–F–G with cost 12.)

b) Write down the advantage of IDS over BFS and DFS. Also, shows times and space complexity among them.

Here's a clear comparison among **IDS (Iterative Deepening Search)**, **BFS (Breadth-First Search)**, and **DFS (Depth-First Search)** — including their **advantages**, **time complexity**, and **space complexity**:

✅ Advantage of IDS over BFS and DFS

Compared With	Advantage of IDS

Over BFS	IDS uses **much less memory**. BFS stores all nodes at the current level (which can be exponential in size), but IDS stores only one path from the root to a leaf (like DFS).
Over DFS	IDS is **complete and optimal** (for uniform step cost), whereas DFS can get stuck in deep or infinite paths and may not find the shallowest goal.

👉 **In summary:**
IDS combines the **space efficiency of DFS** with the **completeness and optimality of BFS**.

⚙️ Complexity Comparison

Let:

- **b** = branching factor (average number of successors per node)
- **d** = depth of the shallowest goal node

Algorithm	Completeness	Optimality	Time Complexity	Space Complexity
BFS	✅ Yes	✅ Yes (for equal step cost)	(O(b^{d+1}))	(O(b^{d+1}))
DFS	❌ No (can go infinite)	❌ No	(O(b^{m})) where *m* = max depth	(O(bm))
IDS	✅ Yes	✅ Yes (for equal step cost)	(O(b^{d}))	(O(bd))

🧠 Explanation

- **BFS:** Explores level by level → finds the shallowest goal → optimal, but needs huge memory.
- **DFS:** Explores deep paths first → low memory but may miss the optimal or even any goal if loops exist.
- **IDS:** Repeats DFS up to increasing depths → revisits nodes multiple times but the overhead is small ((<10%)) compared to the exponential cost saved in memory.

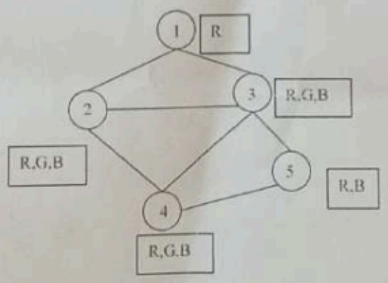
🏁 Summary Table

Feature	BFS	DFS	IDS
Completeness	Yes	No	Yes
Optimality (unit cost)	Yes	No	Yes
Time Complexity	(O(b^{d+1}))	(O(b^{m}))	(O(b^{d}))
Space Complexity	(O(b^{d+1}))	(O(bm))	(O(bd))
Memory Usage	Very High	Low	Moderate (Low like DFS)

7.

---

a) Consider the following constraint graph for a graph coloring problem (the constraints indicate that connected nodes cannot have the same color). The domains are shown in the boxes next to each variable node. [8]



Now answer the following questions:

I. What are the variable domains after a full constraint propagation?

II. Show the sequence of variable assignments during a pure backtracking search (do not assume that the propagation above has been done), assume that the variables are examined in numerical order and the values are assigned in the order shown next to each node.

Question I: Variable Domains After Full Constraint Propagation

Initial Domains:

- **Node 1:** {R}
- **Node 2:** {R, G, B}
- **Node 3:** {R, G, B}
- **Node 4:** {R, G, B}
- **Node 5:** {R, B}

After Constraint Propagation:

- **Node 1:** {R}
- **Node 2:** {G, B}
- **Node 3:** {G, B}
- **Node 4:** {G, B}
- **Node 5:** {B}

Propagation Steps:

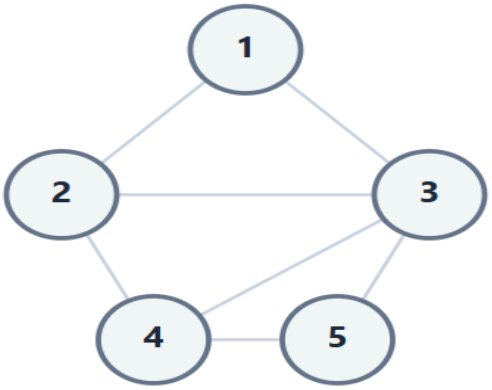
1. **Node 1 = R** (only option) → Remove R from neighbors 2 and 3
2. **Node 2:** {R,G,B} → {G,B} (R removed due to edge 1-2)
3. **Node 3:** {R,G,B} → {G,B} (R removed due to edge 1-3)
4. **Node 5 = B** (after checking): Remove R (would conflict with potential assignments)
5. **Node 4:** Remains {G,B} (multiple valid options)

II.

Initial State (Before Backtracking)

Starting with original domains. Will assign variables in order: 1, 2, 3, 4, 5

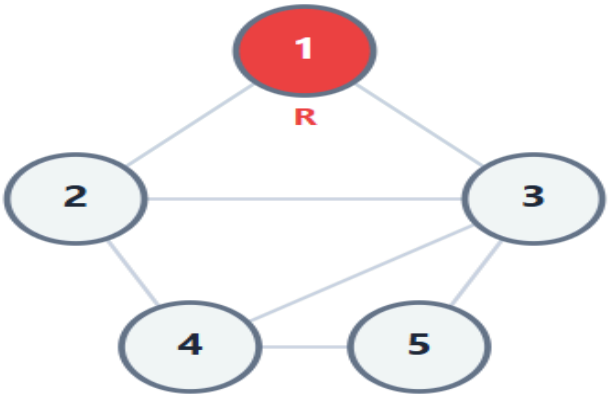
> Ready to start backtracking



Step 1: Assign Variable 1

Variable 1 has only one value in domain: R

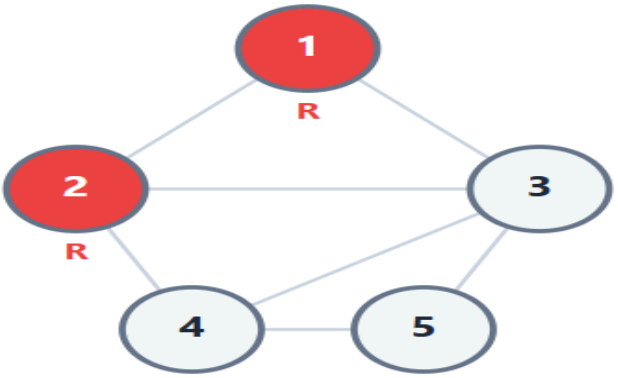
> Assigned 1 = R



Step 2: Assign Variable 2

Variable 2: Try first value R from domain {R, G, B}

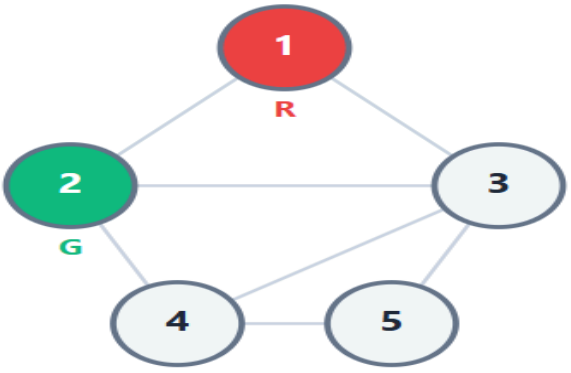
> **Assigned 2 = R**
> Check constraints: 2-1 edge → R ≠ R? NO! Constraint violated!



Step 2 (retry): Backtrack and try next value

R failed for variable 2. Try next value: G

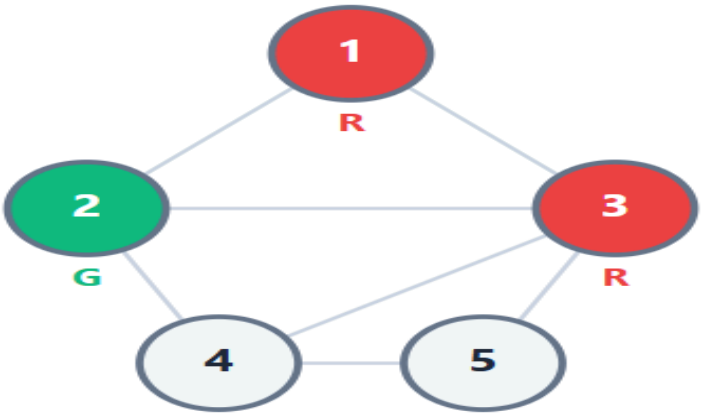
> **Assigned 2 = G**
> Check constraints: 2-1 (G≠R ✓), 2-3 (will check later), 2-4 (will check later)



Step 3: Assign Variable 3

Variable 3: Try first value R from domain {R, G, B}

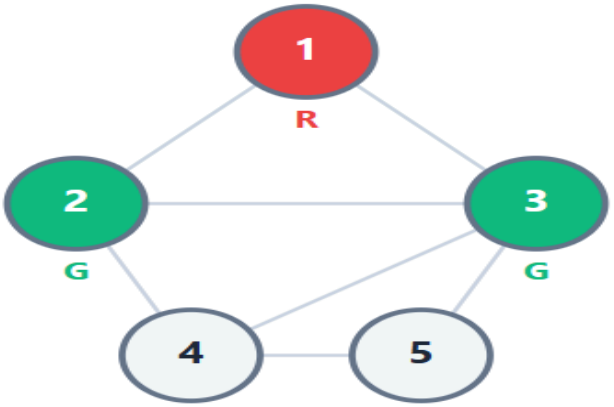
> **Assigned 3 = R**
> Check: 3-1 (R≠R? NO!), Constraint violated!



Step 3 (retry): Try next value

R failed for variable 3. Try next value: G

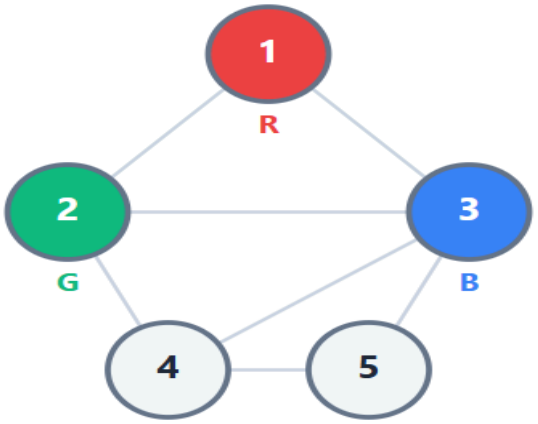
> **Assigned 3 = G**
> Check: 3-1 (G≠R ✓), 3-2 (G≠G? NO!), Constraint violated!



Step 3 (retry): Try next value

G failed for variable 3. Try next value: B

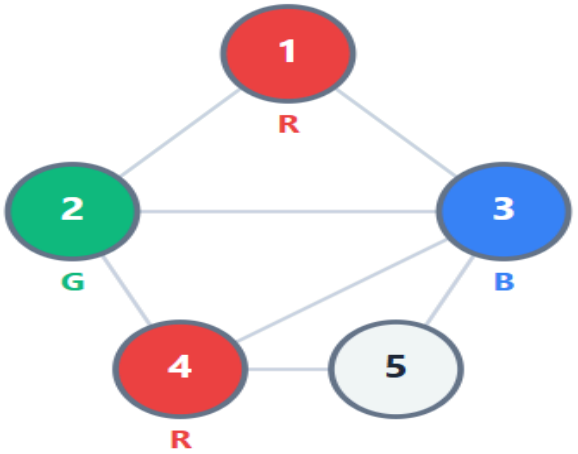
> **Assigned 3 = B**
> Check: 3-1 (B≠R ✓), 3-2 (B≠G ✓), 3-4 (will check), 3-5 (will check) ✓



Step 4: Assign Variable 4

Variable 4: Try first value R from domain {R, G, B}

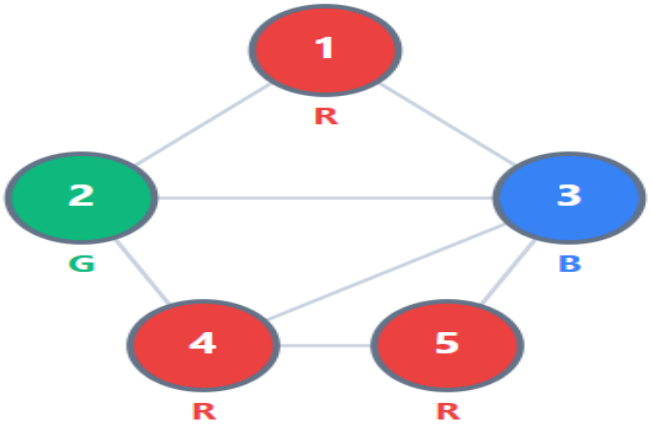
> **Assigned 4 = R**
> Check: 4-2 (R≠G ✓), 4-3 (R≠B ✓), 4-5 (will check) ✓



Step 5: Assign Variable 5

Variable 5: Try first value R from domain {R, B}

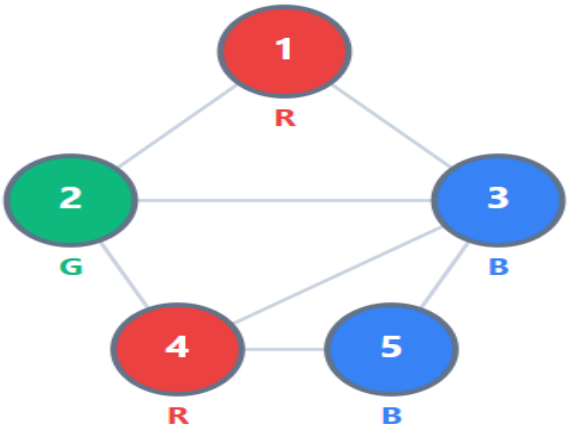
> **Assigned 5 = R**
> Check: 5-3 (R≠B ✓), 5-4 (R≠R? NO!), Constraint violated!



Step 5 (retry): Try next value

R failed for variable 5. Try next value: B

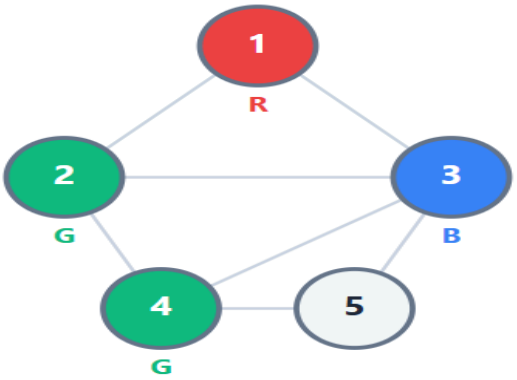
> **Assigned 5 = B**
> Check: 5-3 (B≠B? NO!), Constraint violated! No more values in domain!



Backtrack to Variable 4

Variable 5 has no valid assignment. Backtrack to variable 4 and try next value.

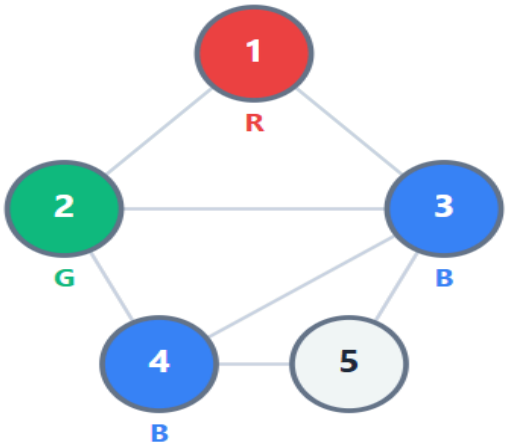
> **Changed 4 = G (trying next value)**
> Check: 4-2 (G≠G? NO!), Constraint violated!



Step 4 (retry): Try next value

G failed for variable 4. Try next value: B

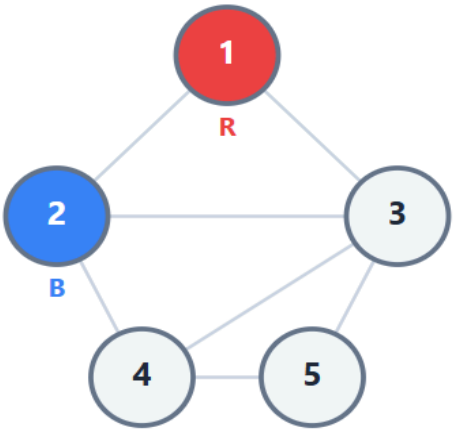
> **Assigned 4 = B**
> Check: 4-2 (B≠G ✓), 4-3 (B≠B? NO!), Constraint violated! No more values!



Backtrack to Variable 3

Variable 4 exhausted. Backtrack to variable 3. All values tried. Backtrack to variable 2!

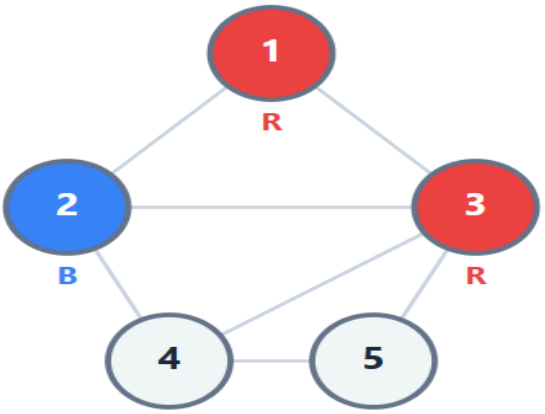
> **Backtracked to variable 2, trying 2 = B**
> Check: 2-1 (B≠R ✓), 2-3 (will check), 2-4 (will check) ✓



Step 3 (new branch): Assign Variable 3

With 2=B, try variable 3: first value R

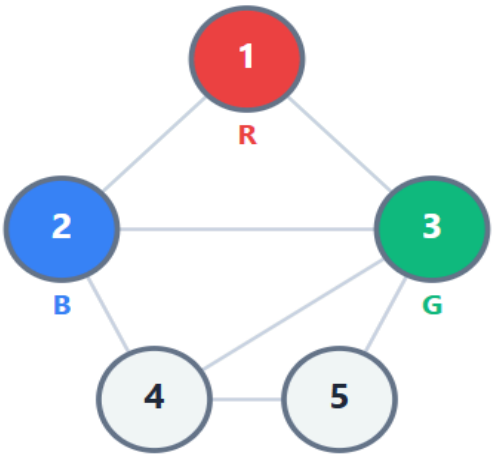
> **Assigned 3 = R**
> Check: 3-1 (R≠R? NO!), Constraint violated!



Step 3 (retry): Try G

R failed. Try G for variable 3

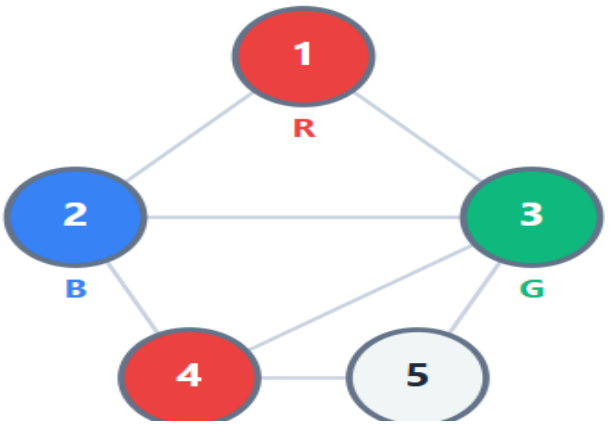
> **Assigned 3 = G**
> Check: 3-1 (G≠R ✓), 3-2 (G≠B ✓), 3-4 (will check), 3-5 (will check) ✓



Step 4: Assign Variable 4

Variable 4: Try first value R

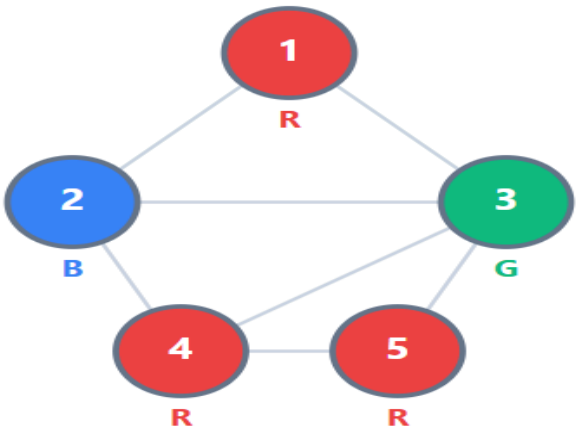
> **Assigned 4 = R**
> Check: 4-2 (R≠B ✓), 4-3 (R≠G ✓), 4-5 (will check) ✓



Step 5: Assign Variable 5

Variable 5: Try first value R

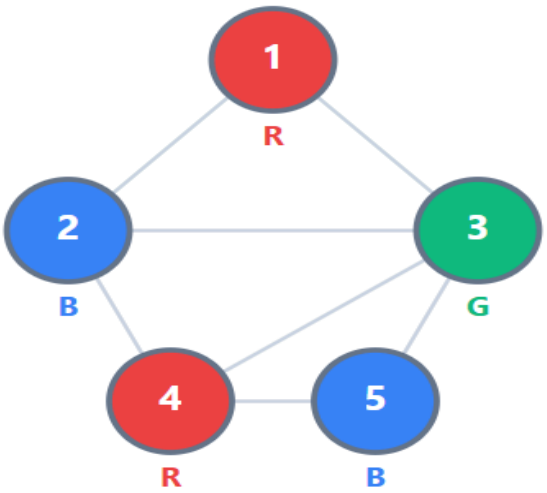
> **Assigned 5 = R**
> Check: 5-3 (R≠G ✓), 5-4 (R≠R? NO!), Constraint violated!



Step 5 (retry): Try B

R failed. Try B for variable 5

> **Assigned 5 = B**
> Check: 5-3 (B≠G ✓), 5-4 (B≠R ✓) ✓✓✓ SOLUTION FOUND!



b) What is forward cheeking algorithm? Give an example

Let's explain it clearly 👇

🔹 **Forward Checking Algorithm — Definition**

Forward Checking is a **constraint satisfaction** technique used in **Artificial Intelligence** and **Constraint Satisfaction Problems (CSPs)** (like Sudoku, map coloring, etc.).

It is used during **backtracking search** to **reduce the search space** by **looking ahead** and eliminating inconsistent values from the domains of the unassigned variables.

🔹 **How It Works**

When a variable is assigned a value, **forward checking**:

1. Looks ahead to the **remaining (unassigned) variables**.
2. Removes any value from their domain that **conflicts** with the current assignment.
3. If any variable's domain becomes **empty**, it means the current path **cannot lead to a solution**, and **backtracking** occurs immediately.

This helps detect **inconsistencies early**, saving computation time.

🔹 **Algorithm Steps**

1. Select an unassigned variable.
2. Assign a value from its domain.
3. For each unassigned variable:
    - Remove all values inconsistent with the current assignment.

4. If any domain becomes empty → **Backtrack**.

5. Else, continue assigning next variable.

🔹 **Example: Map Coloring Problem**

Let's consider three regions:
A, **B**, and **C**.
Each can be colored with **{Red, Green, Blue}**,
and adjacent regions **cannot have the same color**.

Adjacency:

- A ↔ B
- B ↔ C

Step 1: Assign

Assign **A = Red**.

Step 2: Forward Checking

Since A and B are adjacent:

- Remove **Red** from B's domain.
  Now:
- B: {Green, Blue}
- C: {Red, Green, Blue}

Step 3: Assign next variable

Assign **B = Green**.

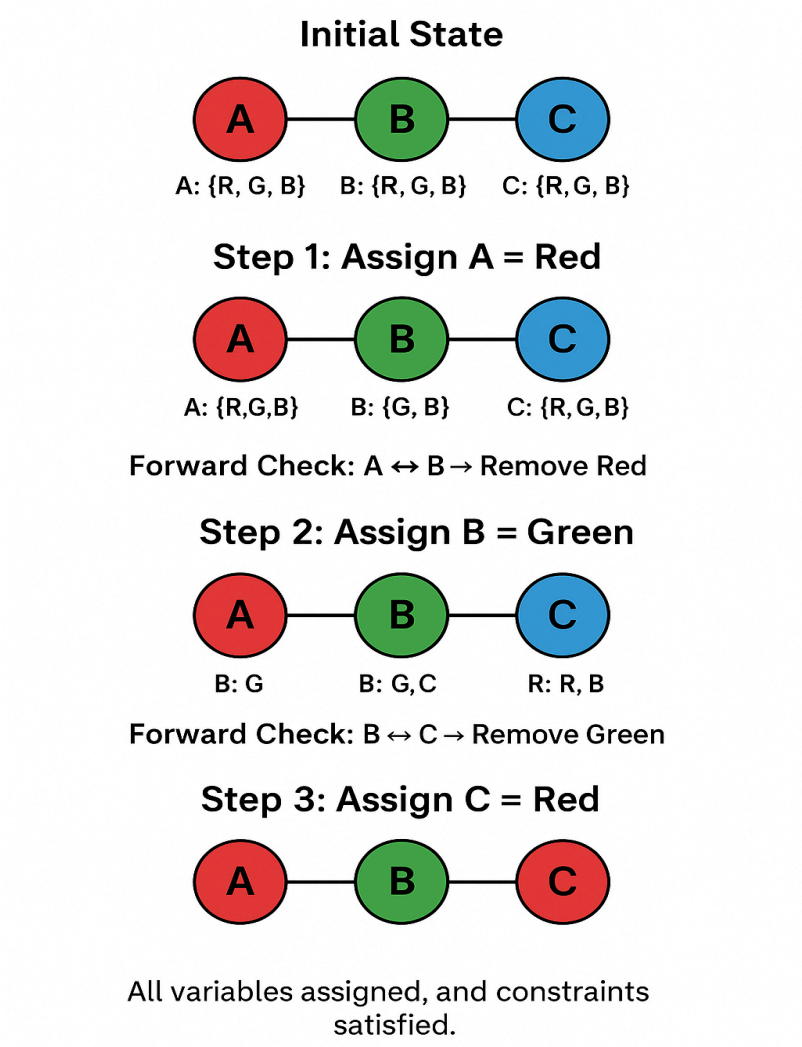
Forward Checking:

- B and C are adjacent → remove **Green** from C's domain.
  Now:

- C: {Red, Blue}

Step 4: Assign C = Red → ✅ No conflict.

All variables assigned → **Solution found**.



- **Advantages**

- Detects conflicts early.
- Reduces unnecessary backtracking.
- More efficient than pure backtracking.

🔹 **Disadvantages**

- Still not as strong as full arc consistency (like in MAC algorithm).
- Requires extra computation to maintain domains.

✅ **Summary Table**

Step	Variable Assigned	Domains After Forward Checking
Start	A,B,C = {R,G,B}	Initial domains
1	A = R	B={G,B}, C={R,G,B}
2	B = G	C={R,B}
3	C = R	Solution found ✅

Would you like me to show this example with a **diagram or pseudocode** of the algorithm next?

8.

a) What is logic, syntax and semantics? Write down some propositional logie syntax.

- **Logic** is a formal system for manipulating facts so that true conclusions may be drawn
  - *"The tool for distinguishing between the true and the false" – Averroes (12th cen.)*
- **Syntax:** rules for constructing valid sentences
  - E.g., $x + 2 \geq y$ is a valid arithmetic sentence, $\geq x2y +$ is not
- **Semantics:** "meaning" of sentences, or relationship between logical sentences and the real world
  - Specifically, semantics defines truth of sentences
  - E.g., $x + 2 \geq y$ is true in a world where $x = 5$ and $y = 7$

Propositional Logic:

- **Atomic sentence:**
  - A *proposition symbol* representing a true or false statement
- **Negation:**
  - If **P** is a sentence, $\neg$**P** is a sentence
- **Conjunction:**
  - If **P** and **Q** are sentences, **P** $\wedge$ **Q** is a sentence
- **Disjunction:**
  - If **P** and **Q** are sentences, **P** $\vee$ **Q** is a sentence
- **Implication:**
  - If **P** and **Q** are sentences, **P** $\Rightarrow$ **Q** is a sentence
- **Biconditional:**
  - If **P** and **Q** are sentences, **P** $\Leftrightarrow$ **Q** is a sentence

- $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ are called *logical connectives*

1. Logic — Definition

Logic is the study of **reasoning and inference** — how we can derive conclusions from given facts or premises.

In **Artificial Intelligence (AI)** and **Computer Science**, logic provides a formal language to represent **knowledge** and to perform **reasoning** automatically.

2. Syntax — Definition

Syntax refers to the **rules that define the structure or form** of valid statements (or formulas) in a logical language.

👉 In simple words:
Syntax = grammar or structure of logical expressions.

For example, in **Propositional Logic**, a well-formed formula (WFF) follows these syntactic rules:

- Atomic propositions: $P, Q, R, \ldots$
- Logical connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- Parentheses are used to group subformulas.

Example of Syntax (valid formulas):

- P
- $\neg P$
- $P \wedge Q$
- $(P \vee \neg Q) \rightarrow R$

❌ **Invalid syntax example:**

- $PQ \vee$
- $\rightarrow PQ$

3. Semantics — Definition

Semantics gives **meaning** to the syntactically correct sentences.
It defines **how truth values** (True or False) are assigned to logical expressions.

👉 In simple words:
Semantics = meaning / interpretation of logical symbols.

For example:

- If P is true and Q is false,
  then the meaning of P⋀Q (P AND Q) is **False**.

✅ Truth Table Semantics Example

P	Q	$P \wedge Q$	$P \vee Q$	$\neg P$	$P \rightarrow Q$
T	T	T	T	F	T
T	F	F	T	F	F
F	T	F	T	T	T
F	F	F	F	T	T

🔷 4. Summary Table

Concept	Meaning	Example
Logic	Study of reasoning and inference	Deductive reasoning, Propositional logic
Syntax	Rules of how sentences are written	$(P \vee Q) \rightarrow R$
Semantics	Meaning or truth of the sentences	If P=T, Q=F ⇒ $P \vee Q = T$

🔷 Some Common Propositional Logic Syntax Rules

Symbol	Meaning	Example
P, Q, R	Atomic propositions	P = "It is raining"
¬	NOT	$\neg P$: "It is not raining"
∧	AND	$P \wedge Q$: "It is raining and cold"
∨	OR	$P \vee Q$: "It is raining or cold"
→	IMPLIES	$P \rightarrow Q$: "If it rains, the ground is wet"
↔	IFF (if and only if)	$P \leftrightarrow Q$: "It rains if and only if it's cloudy"

b) Show that p->(q->r) is logically equivalent to (p^q)→r

Logical Equivalence

Two propositions (logical statements) **P** and **Q** are said to be **logically equivalent** if they always have **the same truth value** (True or False) in **every possible case** of their variables.

That means:

P≡Q if and only if P and Q are both true or both false for all possible truth assignments

🔹 Condition of Logical Equivalence

P↔Q must be a tautology.

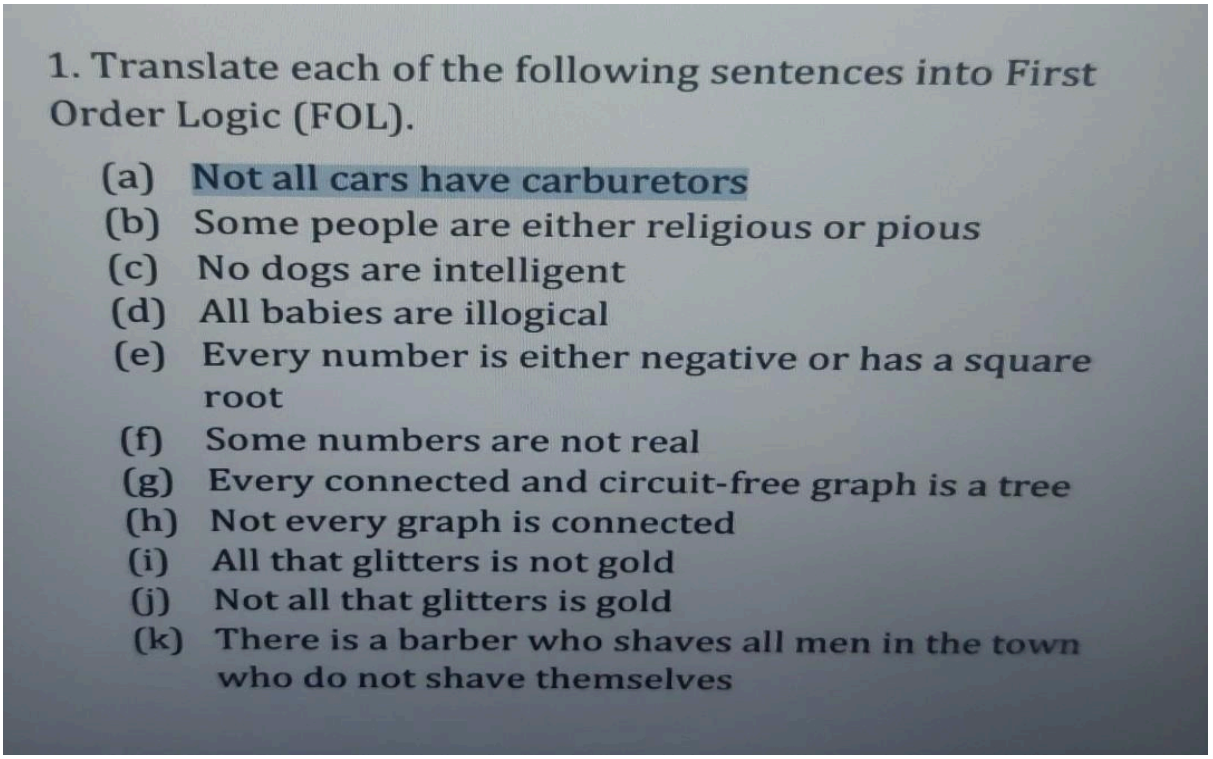
👉 That is, the compound statement (P↔Q) should be **True in all rows** of the truth table

🔷 Step 5: Verification by Truth Table

p	q	r	q→r	p→(q→r)	(p∧q)→r
T	T	T	T	T	T
T	T	F	F	F	F
T	F	T	T	T	T
T	F	F	T	T	T
F	T	T	T	T	T
F	T	F	F	T	T
F	F	T	T	T	T
F	F	F	T	T	T

c) Translate each of the following sentences into First Order Logic (FOL)

Not all cars have carburetors ii. All babies are illogical it. Every connected and



1. Translate each of the following sentences into First Order Logic (FOL).

(a) `Not all cars have carburetors`
$\neg\forall x\ [car(x) \rightarrow carburetors(x)]$ or
$\exists x\ [car(x) \wedge \neg carburetors(x)]$

(b) `Some people are either religious or pious`
$\exists x\ (R(x) \oplus P(x)) \equiv \exists x\ \neg(R(x) \leftrightarrow P(x))$ or
$\neg\forall x\ [R(x) \leftrightarrow P(x)]$ or
$\exists x((R(x) \wedge \neg P(x)) \vee (\neg R(x) \wedge P(x))$

(c) `No dogs are intelligent`
$\forall x\ (dog(x) \rightarrow \neg Intelligent(x))$ or
$\neg\exists x\ (dog(x) \wedge Intelligent(x))$

(d) `All babies are illogical`
$\forall x\ (baby(x) \rightarrow illogical(x))$ or
$\neg\exists x\ (baby(x) \wedge \neg illogical(x))$

(e) `Every number is either negative or has a square root`
$\forall x\ \neg(negative(x) \leftrightarrow sqroot(x))$ or
$\neg\exists x\ (negative(x) \leftrightarrow sqroot(x))$ or
$\forall x((negative(x) \wedge \neg sqroot(x)) \vee (\neg negative(x) \wedge sqroot(x))$

(f) Some numbers are not real
$\exists x \neg Real(x)$ or $\neg \forall x\ Real(x)$

(g) Every connected and circuit-free graph is a tree
$\forall x\ [(conn(x) \wedge \neg cir(x)) \rightarrow tree(x)]$ or
$\neg \exists x\ [(conn(x) \wedge \neg cir(x)) \wedge \neg tree(x)]$

(h) Not every graph is connected
$\neg \forall x\ connected(x)$ or $\exists x\ \neg connected(x)$

(i) All that glitters is not gold
$\forall x\ [glitter(x) \rightarrow \neg gold(x)]$ or
$\neg \exists x\ [glitter(x) \wedge gold(x)]$

(j) Not all that glitters is gold
$\neg \forall x(glitter(x) \rightarrow gold(x))$ or
$\exists x(glitter(x) \wedge \neg gold(x))$.

(k) There is a barber who shaves all men in the town who do not shave themselves
$\exists x\ [Barber(x) \wedge \forall y\ [man(y) \wedge \neg shaves(y, y)] \rightarrow shaves(x, y)]$

(l) There is no business like show business
$\forall x\ [(business(x) \wedge (x \neq show\ business)) \rightarrow \neg like(x, show\ \ business)]$

2. Rewrite each proposition symbolically, given that the universe of discourse is a set of real numbers

(a) For each integer x, there exist an integer y such that $x + y = 0$
$\forall x\ [int(x) \rightarrow \exists y\ (int(y) \wedge (x + y = 0))]$

(b) There exist an integer x such that $x + y = y$ for every integer y
$\exists x\ [int(x) \wedge \forall y\ (int(y) \rightarrow (x + y = y))]$

(c) For all integers x and y, $x.y = y.x$
$\forall x\ \forall y\ [[int(x) \wedge int(y)] \rightarrow x.y = y.x]$

(d) There are integers x and y such that x+y=5
$\exists x\ \exists y\ [(int(x) \wedge int(y)) \wedge (x + y = 5)]$