AI Practical Presentation

TE Computer A

Index

Tic Tac Toe Using Bruteforce and Heuristic Apprcoch

Problem Statement

1. Tic Tac Toe Using Bruteforce approch
2. Tic Tac Toe Using Heuristic Approch

What is Bruteforce Technique

the bruteforce approach is like trying every possible option until you find the right one, without using any clever shortcuts or strategies.
Example:reallife example of a bruteforce approach is searching for a lost item in your house. You might start by checking every room, opening every drawer, and looking under every piece of furniture until you find the item. This method doesn't involve any special strategy or shortcuts; you're simply systematically checking every possible location until you find what you're looking for. While this approach may eventually lead you to the lost item, it can be timeconsuming, especially if you have a large house with many rooms and hiding spots.

TIC TAC TOE USING BRUTEFORCE APPROCH

- **Bruteforce Usage:**
- **Minimax Algorithm: Bruteforce is used in the minimax algorithm to recursively evaluate all possible future game states.**
- **Generate Moves: Bruteforce generates all possible next moves for a given board position to exploe all potential game outcomes.**
- **Move Evaluation: Bruteforce evaluates each move by simulating future game states to determine the best move for the computer.**
- **Move Table Usage:**
- **Optimization: The move table stores evaluated scores for each board position to avoid redundant calculations.**
- **Caching Scores: Previously computed scores are stored in the move table to prevent recalculating them during subsequent moves.**
- **Efficiency: By storing and retrieving scores from the move table, the algorithm improves efficiency and reduces computation time.**
-

What is Heuristic Technique

Heuristic approach is a problemsolving strategy that uses practical experience and rules of thumb to find solutions quickly, even if they may not be optimal. It's like using shortcuts based on common sense rather than exhaustive search. Example: Imagine you're trying to find the fastest route to work. Instead of examining every possible road and calculating the exact travel time, you might use a heuristic approach by taking the main highway because it's usually faster during rush hour based on past experience.

TIC TAC TOE USING HEURISTIC APPROACH

- This heuristic_move() function implements strategies for the computer's move:
- Winning Move: Checks for an immediate win.
- Blocking Opponent: Prevents the opponent from winning.
- Forking Strategy: Creates opportunities for multiple winning paths.
- Center Strategy: Occupies the center or corners strategically.
- Random Move: Chooses a random empty cell when no strategic moves are available.

Results

main.py

Save Run

```python
import random

# Function to print the Tic Tac Toe board
def print_board(board):
    for row in board:
        print(' '.join(row))

# Function to check if any player has won the game
def check_winner(board, player):
    # Check rows and columns
    for i in range(3):
        if all(cell == player for cell in board[i]) or all
            (board[j][i] == player for j in range(3)):
            return True
    # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all
        (board[i][2 - i] == player for i in range(3)):
        return True
```

Shell

```
O X O
- X -
O - -
Computer's move:
O X O
X X -
O - -
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 1
O X O
X X -
O O -
Computer's move:
O X O
X X X
O O -
Computer wins!
>
```

Results

main.py

Save Run

```python
import random

def print_board(board):
    for row in board:
        print(' '.join(row))

def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != '-':  # Rows
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != '-':  # Columns
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != '-':  # Diagonal 1
        return board[0][0]
```

Shell

```
User's turn (O)
Enter row (0, 1, 2): 1
Enter column (0, 1, 2): 2
O O X
X X O
O - -
Computer's turn (X)
O O X
X X O
O - X
User's turn (O)
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 1
O O X
X X O
O O X
It's a draw!
>
```

https://replit.com/@joelpawarwork/AI-Presentations#bruteforce.py

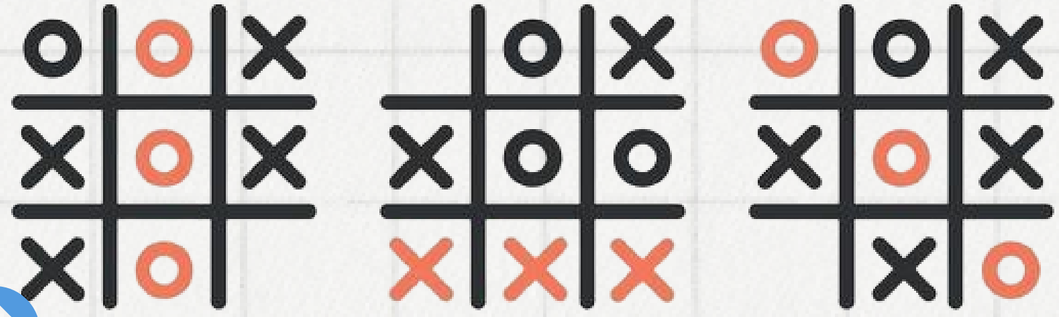
https://replit.com/@joelpawarwork/AI-Presentations#heuristic.py

Tic-Tac-Toe Game Playing using Magic Square

Problem Statement

It’s a paper and pencil game of two players X and O, who choose to mark a space on a grid of 3×3.

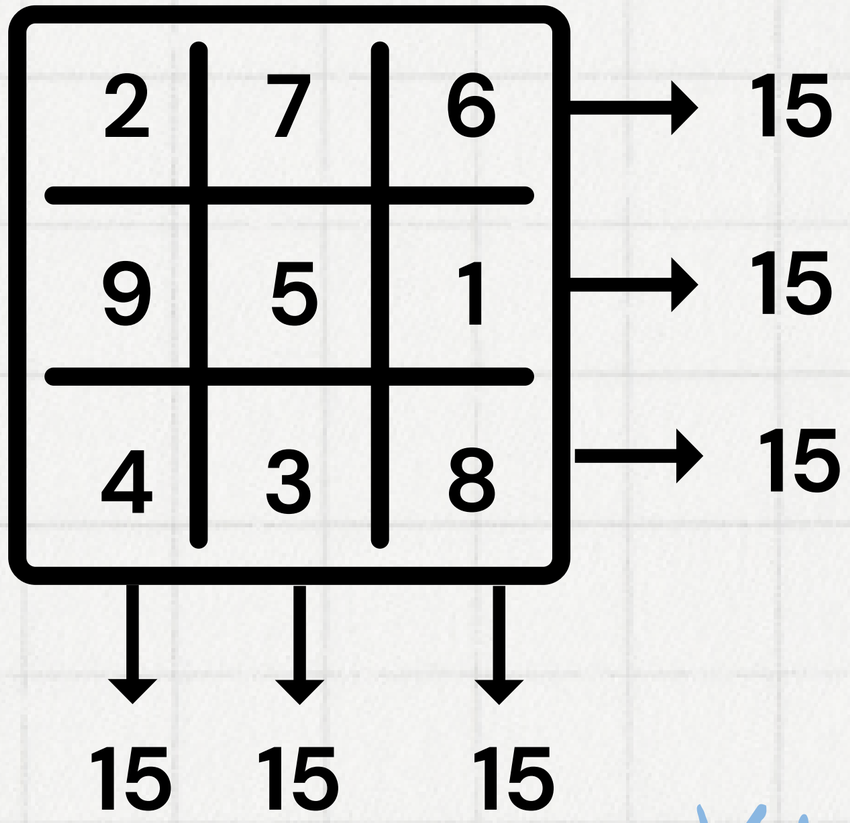
The game is won by the player who succeeds in putting three of their marks in a **horizontal line**, **vertical line**, or **diagonal line.**



This game is played by two players – One is a **human** and the other is a **computer**.

The objective is to write a computer program in such a way that the computer wins most of the time.

A magic square is a square grid of numbers where the sum of the numbers in each row, column, and diagonal is the same. For a 3x3 magic square, this sum is known as the magic sum.



Algorithm — Tic-Tac-Toe Game Playing using Magic Square —

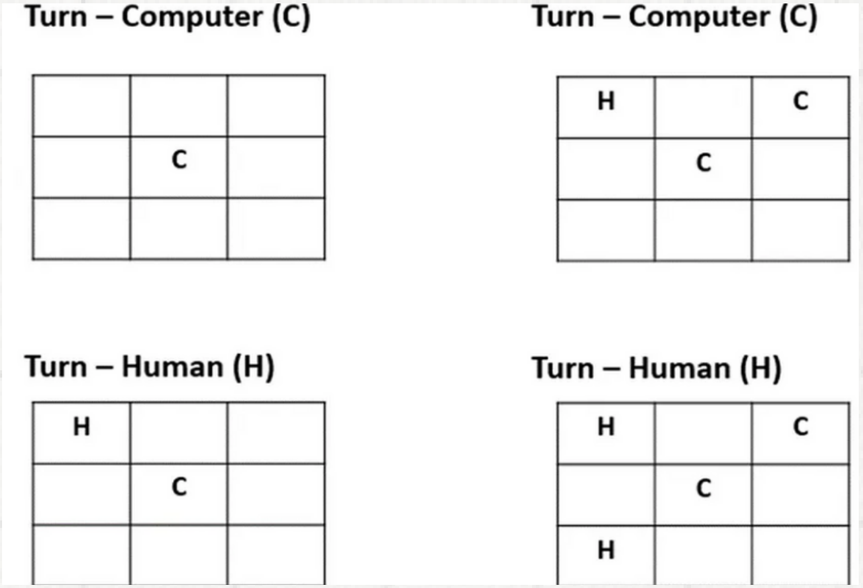
It checks if adding the numbers on two squares **equals 15**. If this difference is **not a positive number or if it is greater than 9**, it means the two squares are not in a straight line, so it ignores them as potential winning moves.

Alternatively, the machine also looks at its opponent's moves to block any chances of the opponent winning.

After each move, you check if any row, column, or diagonal sums up to the magic constant.

Example

8	3	4
1	5	9
6	7	2



calculate the difference between the 15 and the sum of two positions.

Diff = 15 — (5+4) = 6

6 is not empty, hence Computer can’t win the game

Now, the computer checks the possibility of opponents winning the match. If the opponent is winning black it.

Diff = 15 — (8+6) = 1

1 is empty, hence the human can win the game.

Hence Computer Blocks it.

Turn – Computer (C)

H		
C	C	
H		

Computer go to 1

8	3	4
1	5	9
6	7	2

Turn – Human (H)

H		C
C	C	
H	H	

Now, the computer will check its possibility of winning the game.

Diff = 15 — (5+4) = 6

6 is not empty, hence Computer can't win the game.

Diff = 15 — (1+4) = 10

10 is greater than 9, hence Computer can't win the game.

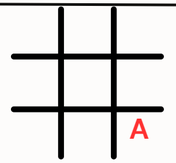
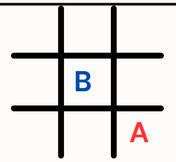
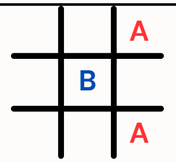
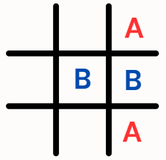
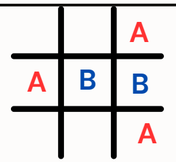
Diff = 15 — (1+5) = 9

9 is empty, hence Computer can win the game. Computer — go to 9

Turn – Computer (C)

H		C
C	C	C
H	H	

Computer wins

PLAYER A	PLAYER B	MOVES	POSTION
Take square 8		Random start	
	Take sqaure 5	15-87. Player B should take an appropriate square to prevent A from winning. B could choose one of 5,2 or 6,1 or 4.3 (sums to 7)	
Take sqaure 6		A needs a score of 7 now. Since 5 is taken 2,5 diagonal has no chance of winning. A should go for 6,1 or 4,3 combination	
	Take sqaure 1	A now has 8+6= 14. B must take 15-14=1 to stop A from winning	
Take sqaure 9		B now has 5+1 = 6. A must now take 15-6=9 to stop B from winning	

At this point, the game is drawn – B takes square 7 or 3, which has better chance of winning, then A blocks the win by taking 3 or 7 and so on.

4 X 4 MATRIX

1	8	12	13
14	11	7	12
15	10	6	3
4	5	9	16

sum of every row and col – 34

Results

```
PS C:\Users\hp> python -u "c:\Users\hp\Downloads\Tic Tac Toe implemented by Magic Square.py"
Welcome to Tic-Tac-Toe using Magic Square technique!
  |   |
-------------
  |   |
-------------
  |   |
-------------
Enter your move (1-9): 1
X |   |
-------------
  |   |
-------------
  |   |
-------------
Computer chooses position 6
X |   |
-------------
  |   | O
-------------
  |   |
-------------
Enter your move (1-9): 4
X |   |
-------------
X |   | O
-------------
  |   |
-------------
Computer chooses position 7
X |   |
-------------
X |   | O
-------------
O |   |
-------------
```

```
Enter your move (1-9): 3
X |   | X
-------------
X |   | O
-------------
O |   |
-------------
Computer chooses position 2
X | O | X
-------------
X |   | O
-------------
O |   |
-------------
Enter your move (1-9): 5
X | O | X
-------------
X | X | O
-------------
O |   |
-------------
Computer chooses position 9
X | O | X
-------------
X | X | O
-------------
O |   | O
-------------
Enter your move (1-9): 8
X | O | X
-------------
X | X | O
-------------
O | X | O
-------------
It's a tie!
```

Implementation link:

https://docs.google.com/document/d/1WyEHWPIBkPTK8KLu93cFSEbD3XxZWALgt0uz8C7KYRw/edit

Advantages of Magic Square Method in Tic Tac Toe:

Balanced Gameplay: Ensures fairness by giving equal opportunities to players.
Strategic Depth: Allows for deeper planning and decision-making based on square values.
Reduced Predictability: Adds excitement and challenge by making outcomes less predictable.

Disadvantages of Magic Square Method in Tic Tac Toe
Complexity: Adds complexity, potentially discouraging those seeking simplicity.
Learning Curve: Requires understanding of magic squares, posing a challenge for new players.
Increased Time: Analyzing squares and planning may prolong games, unsuitable for quick matches.

Thankyou !

WATER JUG PROBLEM

Artificial Intelligence

Problem Statement

- The Water Jug Problem, also known as the **Die Hard** Problem, is a classic puzzle in mathematics and computer science.
- The problem involves two jugs of different capacities and the objective of measuring a specific quantity of water using only these jugs and an unlimited water supply
- Given two jugs with capacities m and n liters, where m and n are positive integers, and the task is to measure out a desired quantity of water, d liters, where d is a positive integer less than or equal to the capacity of the larger jug.
- The challenge is to determine if it's possible to measure d liters using the given jugs and find a sequence of actions that accomplishes this task.

EXAMPLE

You have a 2-liter jug and a 3-liter jug. Can you measure exactly 1 liters of water into 2-liter of jug

01

Fill the 3-liter jug full

02

Pour the water from the 3-liter jug into the 2-liter jug

03

Empty the 2-liter jug

04

Pour the remaining 1 liter from the 3-liter jug into the 2-liter jug

Approaches

There are several approaches you can consider to solve the water jug problem, Each approach has its advantages and limitations, and the choice of method depends on factors such as problem complexity, available computational resources, and desired solution quality.

01.

Brute Force Method

This approach involves systematically trying every possible combination of actions until a solution is found. It may involve creating a tree-like structure to represent all possible states of the jugs and then searching through this tree for a solution

02.

Breadth-First Search

BFS is a graph traversal algorithm that can be applied to solve the water jug problem. In this approach, you start with an initial state representing the empty jugs and explore all possible actions from that state. Then, you move to the next level of states and continue exploring until you find a solution.

03.

Depth-First Search

DFS is another graph traversal algorithm that can be used to solve the water jug problem. In this approach, you start with an initial state and explore as far as possible along each branch before backtracking.

Depth-First Search

1) Define Initial State

Start with an empty initial state.

2) Explore Possible Actions

Actions include filling, emptying, or transferring water between jugs.

3) Recursively Explore States

For each action, recursively explore resulting states.

4) Terminate or Backtrack

Terminate upon reaching goal state or backtrack encountering a dead-end.

5) Repeat

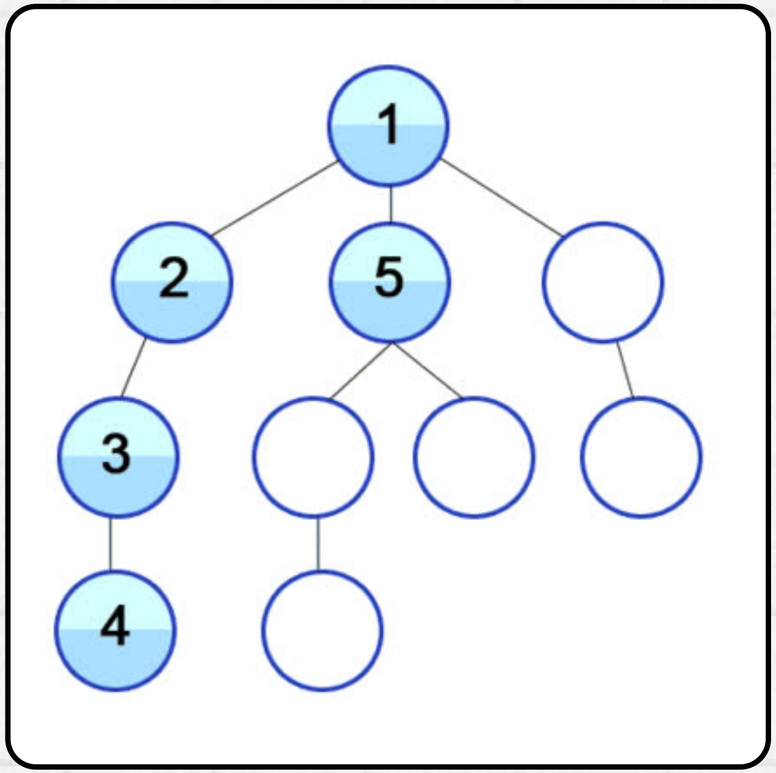
Continue exploring actions until solution found or all states are explored.

6) Track Visited States

Keep track of visited states to avoid revisiting.

7) Return Solution

Return sequence of actions leading to solution and final state of jugs.



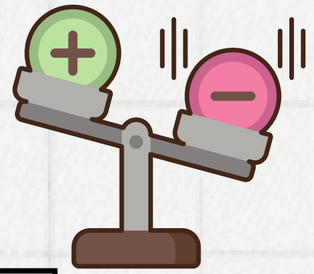
Results

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

(base) PS C:\Users\simon\Downloads\TE PRACS\CSS> python waterjug.py
enter the capacity for Jug1:4
enter the capacity for Jug2:3
enter the Target Capacity:2
('fill', 'jug1') (4, 0)
('fill', 'jug2') (4, 3)
('empty', 'jug1') (0, 3)
('transfer', 'jug2', 'jug1') (3, 0)
('fill', 'jug2') (3, 3)
('transfer', 'jug2', 'jug1') (4, 2)
Number of steps: 6
(base) PS C:\Users\simon\Downloads\TE PRACS\CSS>
```

Implementation Link

Advantages & Disadvantages



	Advantages	Disadvantages
DFS	**Memory Efficiency** • DFS is more memory-efficient compared to BFS in the water jug problem, exploring a single path at a time.	**Completeness and Optimality Not Guaranteed** DFS does not ensure completeness or optimality in finding the shortest path to the goal state; it may encounter infinite loops or explore non-optimal solutions.
BFS	**Completeness and Optimality** BFS guarantees finding the shortest path to the goal state in the water jug problem	**Memory Usage** • BFS may consume a significant amount of memory, especially for problems with extensive search spaces.

THANK YOU

Addressing Missionaries&Cannibals' Problem

By Yashika Haritwal

Understanding problem statement

The missionaries and cannibals problem are classic river-crossing logic puzzles and AI toy problem.

Three missionaries and three cannibals must cross a river using a boat which can carry at most two people

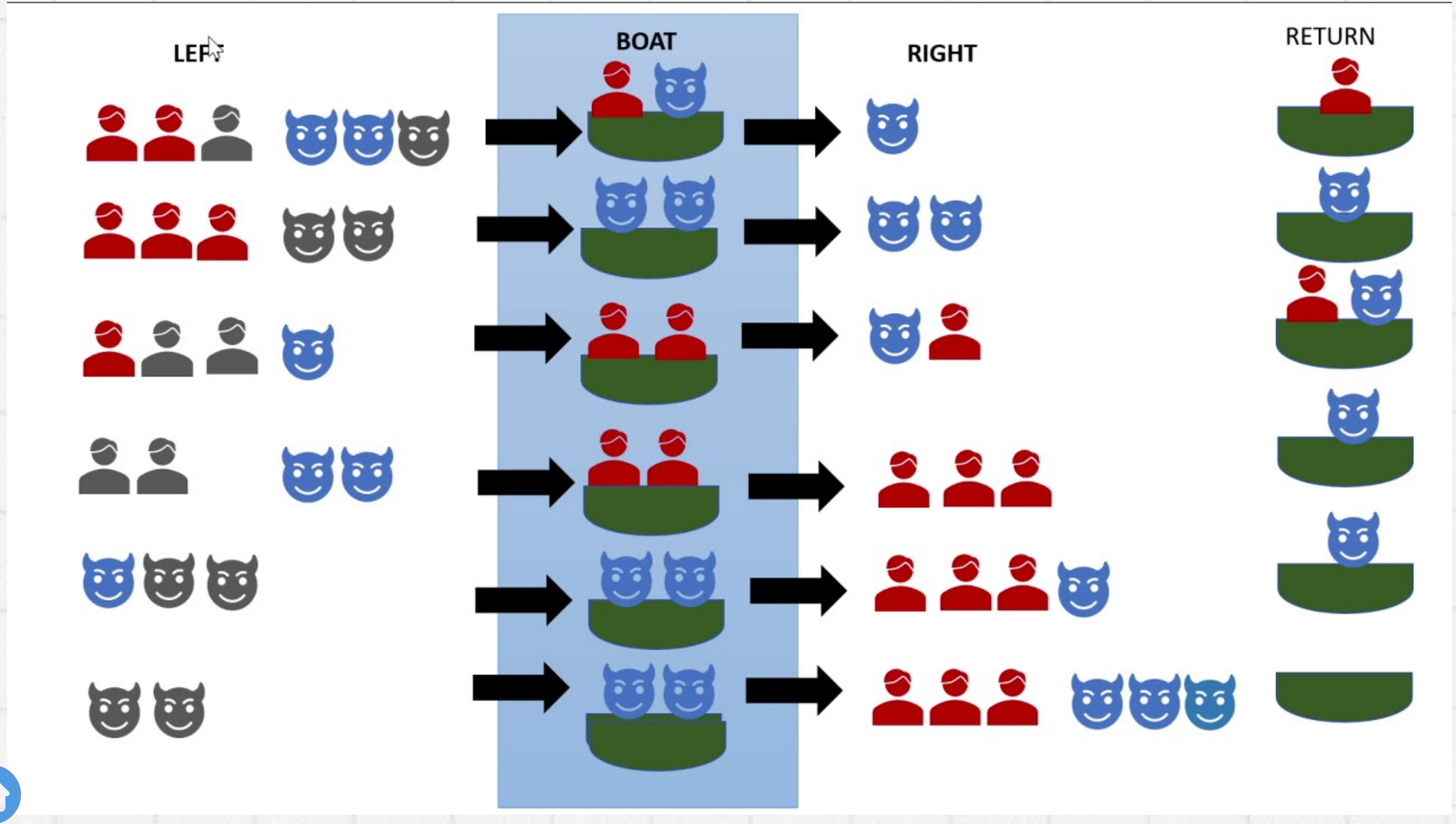
Constraints:

If the missionaries present on the bank are outnumbered by cannibals they will be eaten by the cannibals.

1.Missionaries cannot be outnumbered by cannibals on either banks.

2.The boat cannot cross the river by itself with no people on board.

Possible solution example:



Depth First Search Approach

1. Depth-first search is an algorithm for traversing or searching tree or graph data structures.
2. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.
3. **Missionaries and Cannibals problem**: DFS explores a path from the initial state to the goal state by diving as deeply as possible down one branch of the search tree before backtracking.
4. **State Representation:**

- Each state in the Missionaries and Cannibals problem represents the configuration of missionaries and cannibals on both sides of the river, along with the position of the boat.
- The state space consists of all possible combinations of missionaries and cannibals on both sides of the river.

5. **Next States:**

- At each step, DFS generates all possible valid moves as per constraints from the current state to reach the next possible states.

Depth First Search Tree

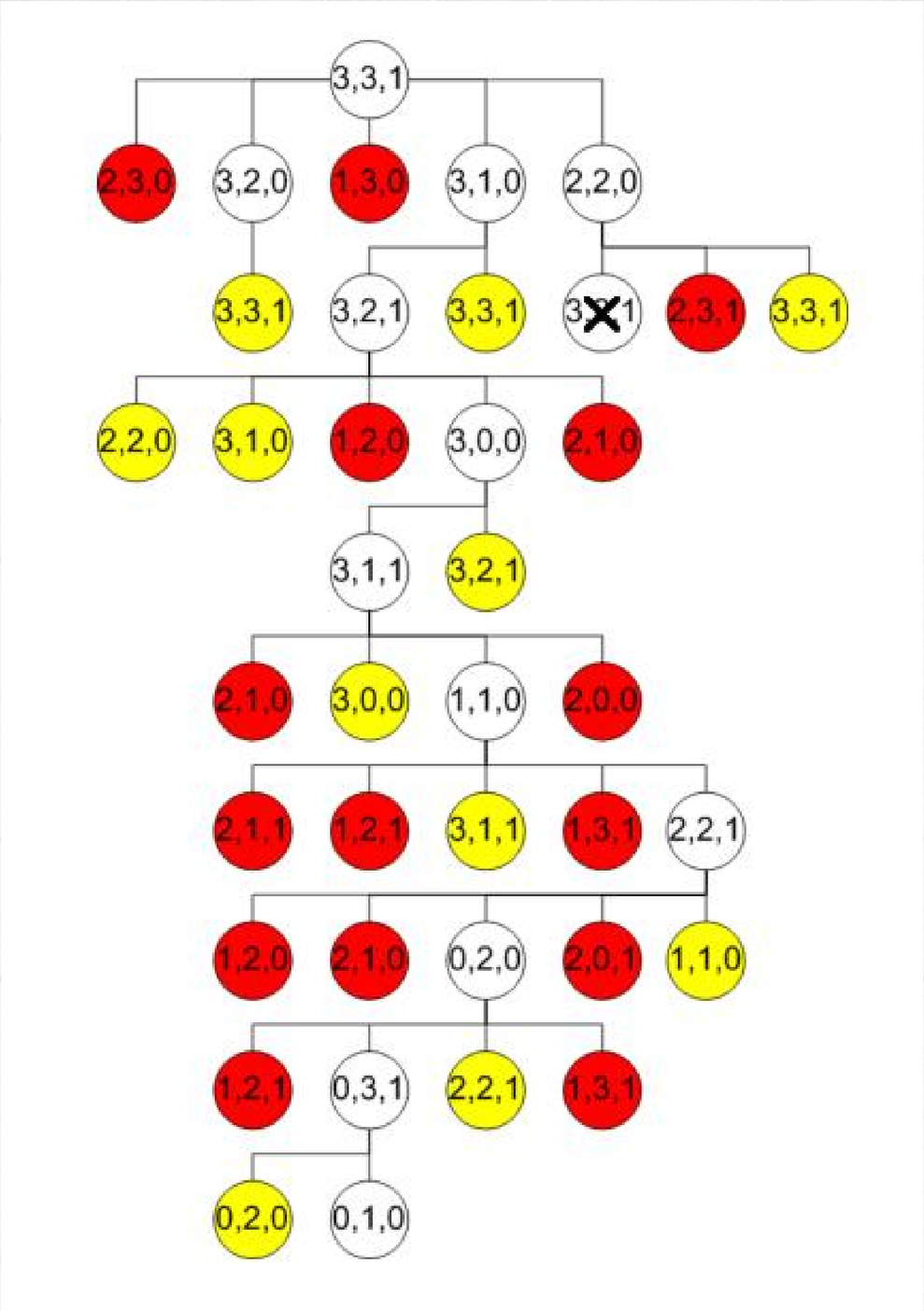
5.**Backtracking:**

- If DFS reaches a dead end it backtracks to the previous state and explores other possible moves.
- This process continues recursively until a goal state is reached:where all missionaries and cannibals are on the right side of the river

Nodes represention for tree
: (massionaries,cannibals,side)

side=1 represents Left
side=0 represents Right

Red States :Missionaries get eaten
Yellow States:Repeated states



Solution1:

01

send 1 cannibal and 1 missionary to right bank

02

send 1 missionary from right to left bank

03

send two cannibals to right bank

04

send 1 cannibal to left

05

send 1 cannibal and 1 missionary to right bank

06

send 1 cannibal and 1 missionary to leftt bank

07

send two missionaries to right bank

08

send 1 cannibal to left bank

09

send 2 cannibal to right bank

10

send 1 missionary to leftt bank

11

send 1 cannibal and 1 missionary to right bank

12

Final state reached

Implementation

CODE:

Result::

```
[yashika@fedora AI]$ python3 mc.py
Left Bank   Right Bank
M   C  Towards   M   C
(3, 3, 'left', 0, 0)
(2, 2, 'right', 1, 1)
(3, 2, 'left', 0, 1)
(3, 0, 'right', 0, 3)
(3, 1, 'left', 0, 2)
(1, 1, 'right', 2, 2)
(2, 2, 'left', 1, 1)
(0, 2, 'right', 3, 1)
(0, 3, 'left', 3, 0)
(0, 1, 'right', 3, 2)
(1, 1, 'left', 2, 2)
(0, 0, 'right', 3, 3)
```

Data structure used: Stack,List,Set

Breadth-First Search (BFS):

- Explores all possible moves at each depth level before moving to the next level,starting from the initial state, until it finds the goal state.
- It guarantees finding the shortest path to the goal state in terms of the number of moves.
- Requires more memory for large search spaces.
- Slower

Depth-First Search (DFS):

- Explores one branch of the search tree as deeply as possible before backtracking to explore other branches.
- Finding shortest path is not guaranteed
- It is memory-efficient and easy to implement.
- Faster

Mind map

Exploring creativity

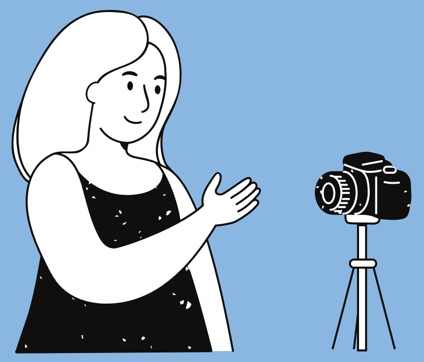
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

why DFS?

Since the problem space for Missionaries and Cannibals is relatively small, DFS can efficiently find a solution by exploring one possible sequence of moves deeply, potentially leading to a solution faster than BFS.

DFS typically requires less memory compared to BFS, making it more suitable for problems with limited memory resources.

Thank you very much!

WATER JUG PROBLEM

Artificial Intelligence

Problem Statement

- The Water Jug Problem, also known as the **Die Hard** Problem, is a classic puzzle in mathematics and computer science.
- The problem involves two jugs of different capacities and the objective of measuring a specific quantity of water using only these jugs and an unlimited water supply
- Given two jugs with capacities m and n liters, where m and n are positive integers, and the task is to measure out a desired quantity of water, d liters, where d is a positive integer less than or equal to the capacity of the larger jug.
- The challenge is to determine if it's possible to measure d liters using the given jugs and find a sequence of actions that accomplishes this task.

EXAMPLE

You have a 2-liter jug and a 3-liter jug. Can you measure exactly 1 liters of water into 2-liter of jug

01

Fill the 3-liter jug full

02

Pour the water from the 3-liter jug into the 2-liter jug

03

Empty the 2-liter jug

04

Pour the remaining 1 liter from the 3-liter jug into the 2-liter jug

Approaches

There are several approaches you can consider to solve the water jug problem, Each approach has its advantages and limitations, and the choice of method depends on factors such as problem complexity, available computational resources, and desired solution quality.

01.

Brute Force Method

This approach involves systematically trying every possible combination of actions until a solution is found. It may involve creating a tree-like structure to represent all possible states of the jugs and then searching through this tree for a solution

02.

Breadth-First Search

BFS is a graph traversal algorithm that can be applied to solve the water jug problem. In this approach, you start with an initial state representing the empty jugs and explore all possible actions from that state. Then, you move to the next level of states and continue exploring until you find a solution.

03.

Depth-First Search

DFS is another graph traversal algorithm that can be used to solve the water jug problem. In this approach, you start with an initial state and explore as far as possible along each branch before backtracking.

Depth-First Search

1) Define Initial State

Start with an empty initial state.

2) Explore Possible Actions

Actions include filling, emptying, or transferring water between jugs.

3) Recursively Explore States

For each action, recursively explore resulting states.

4) Terminate or Backtrack

Terminate upon reaching goal state or backtrack encountering a dead-end.

5) Repeat

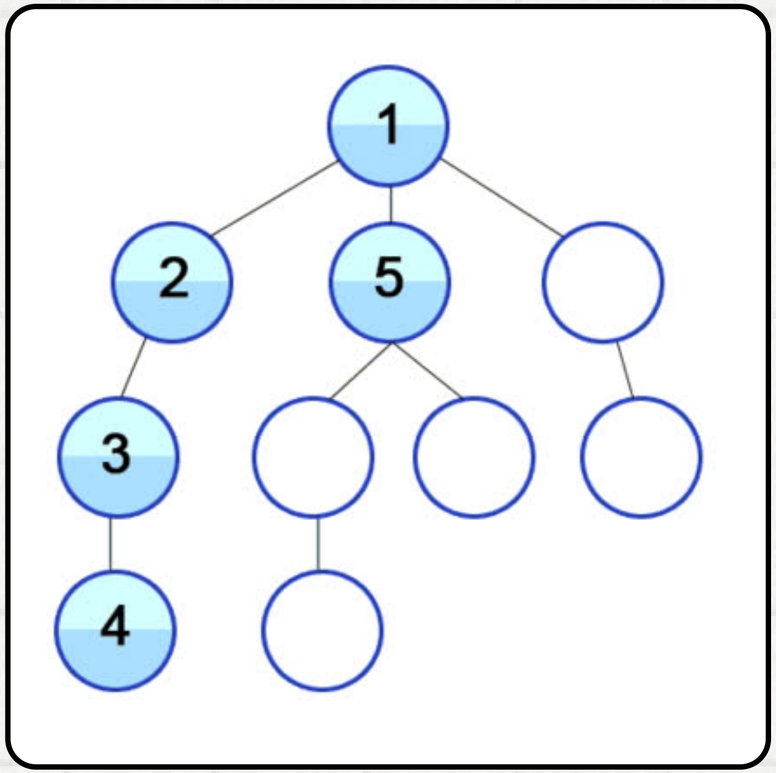
Continue exploring actions until solution found or all states are explored.

6) Track Visited States

Keep track of visited states to avoid revisiting.

7) Return Solution

Return sequence of actions leading to solution and final state of jugs.



Results

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

(base) PS C:\Users\simon\Downloads\TE PRACS\CSS> python waterjug.py
enter the capacity for Jug1:4
enter the capacity for Jug2:3
enter the Target Capacity:2
('fill', 'jug1') (4, 0)
('fill', 'jug2') (4, 3)
('empty', 'jug1') (0, 3)
('transfer', 'jug2', 'jug1') (3, 0)
('fill', 'jug2') (3, 3)
('transfer', 'jug2', 'jug1') (4, 2)
Number of steps: 6
(base) PS C:\Users\simon\Downloads\TE PRACS\CSS>
```

Implementation Link

Breadth-First Search

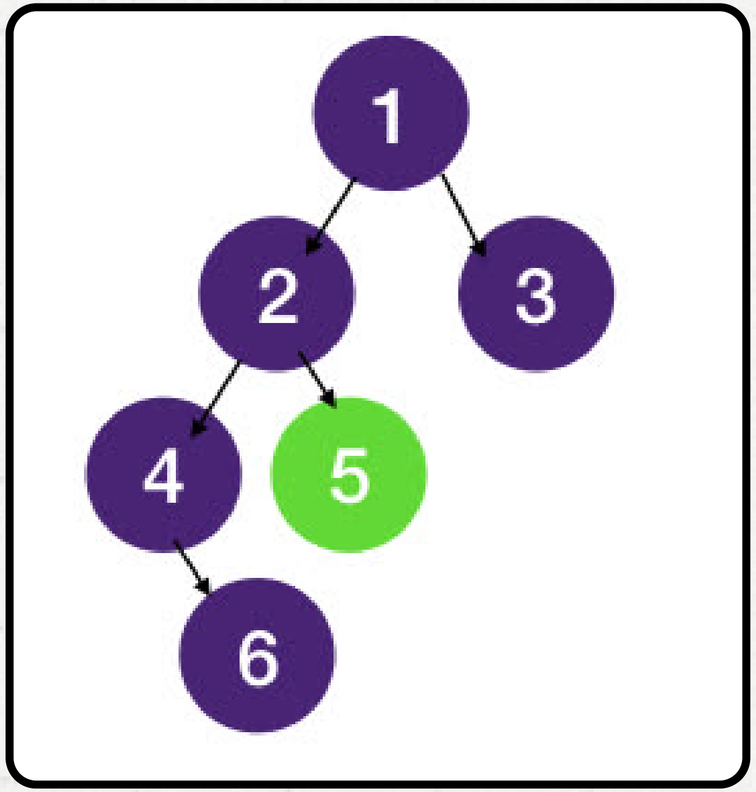
Initial State:
- Start with an initial state representing the current state as empty.

Explore Possible Actions:
- Actions include filling a jug, emptying a jug, or transferring water from one jug to another.

Queue Initialization:
- Use a queue to keep track of states to explore. Enqueue the initial state.

BFS Iteration:
- While the queue is not empty:
  - Dequeue a state.
  - For each possible action:
    - Generate the new state.
    - If the new state is not visited:
      - Enqueue the new state.
      - Mark it as visited.

Goal State Check:
- If a goal state is reached, return the solution.



Breadth-First Search

Repeat:

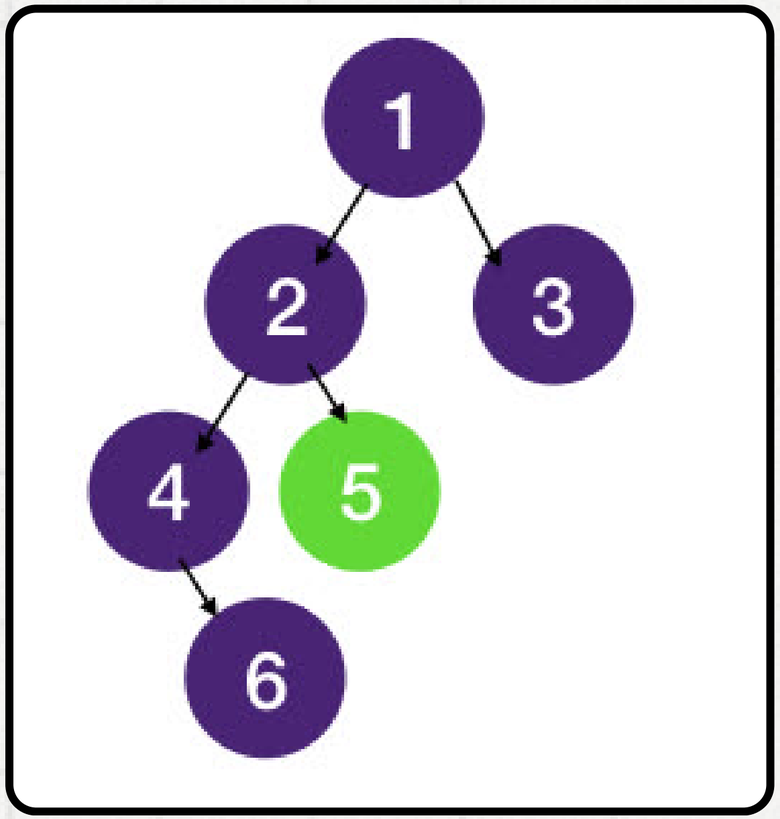
- Repeat the BFS iteration until a solution is found or until all possible states have been explored.

Track Visited States:

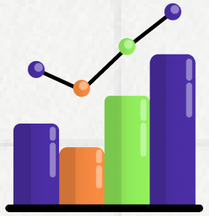
- Use a data structure like a set to keep track of visited states.

Return Solution:

- Once a solution is found, return the sequence of actions that lead to the solution along with the final state of the jugs.



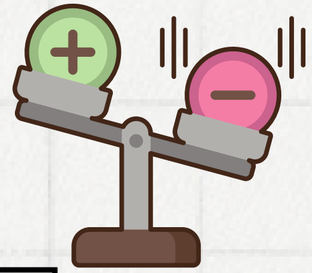
Results



```
Current State: Jug1=0, Jug2=0
Current State: Jug1=4, Jug2=0
Current State: Jug1=0, Jug2=3
Current State: Jug1=4, Jug2=3
Current State: Jug1=1, Jug2=3
Current State: Jug1=3, Jug2=0
Current State: Jug1=1, Jug2=0
Current State: Jug1=3, Jug2=3
Current State: Jug1=0, Jug2=1
Current State: Jug1=4, Jug2=2
Goal state reached! Solution found.


...Program finished with exit code 0
Press ENTER to exit console.
```

Advantages & Disadvantages



	Advantages	Disadvantages
DFS	**Memory Efficiency** • DFS is more memory-efficient compared to BFS in the water jug problem, exploring a single path at a time.	**Completeness and Optimality Not Guaranteed** DFS does not ensure completeness or optimality in finding the shortest path to the goal state; it may encounter infinite loops or explore non-optimal solutions.
BFS	**Completeness and Optimality** BFS guarantees finding the shortest path to the goal state in the water jug problem	**Memory Usage** • BFS may consume a significant amount of memory, especially for problems with extensive search spaces.

THANK YOU

AI Presentation

Presented by kim lopes

Problem Statement

Three missionaries and three cannibals are on one side of a river. They have a boat that can carry at most two people. If at any time the cannibals outnumber the missionaries on either side of the river, the missionaries will be eaten. The goal is to find a way to get all three missionaries and three cannibals safely to the other side of the river without violating these rules.
The problem can be formulated as a search problem, where the states represent the configurations of people on each side of the river and the actions represent moving people from one side to the other. The task is then to find a sequence of actions (moves) that lead from the initial state to the goal state without violating the constraints.

Process To Solution

01

LStart with all three missionaries and three cannibals on the left side of the river.
Take two cannibals across the river to the right side.

02

Send one cannibal back to the left side.
Take two missionaries across to the right side

03

Send one cannibal back to the left side.
Take two cannibals across to the right side.

04

Send one cannibal back to the left side.
Take two missionaries across to the right side.

State-space search

This is a general problem-solving technique where you explore a set of possible states to find a solution. It involves defining the initial state, possible actions, transitions between states, and a goal state.

Depth-first search

DFS is an algorithm for traversing or searching tree or graph data structures. It starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking

A search*: A*

A* is a best-first search algorithm that finds the least-cost path from a given initial node to one goal node (out of one or more possible goals). It evaluates nodes by combining the cost to reach the node and an estimate of the cost to reach the goal from the node

Breadth-first search

BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Solution using BFS

1. Data Structures Definition:
   - State struct: Represents the state of the problem, including the number of missionaries and cannibals on the left side of the river, the boat's position, and is_valid() and is_goal() functions to validate the state and check if it's a goal state.
   - Node struct: Represents a node in the search tree, containing a state and the index of its parent node.
   - Queue struct: Implements a basic queue for BFS using an array.
2. Queue Operations:
   - initialize_queue(): Initializes the queue.
   - is_empty(): Checks if the queue is empty.
   - enqueue(): Adds a node to the rear of the queue.
   - dequeue(): Removes a node from the front of the queue.
3. BFS Function (bfs()):
   - Initializes a queue and enqueues the initial state node with a parent index of -1.
   - Enters a while loop that continues until the queue is empty.
   - Dequeues a node from the front of the queue.
   - Checks if the dequeued node's state is the goal state. If so, it prints the solution path and returns.
   - Generates child nodes by iterating over possible moves (adding or subtracting 1 or 2 missionaries and cannibals).
   - Checks if the generated child state is valid and enqueues it if it is.
   - Repeats the process until a solution is found or the queue becomes empty.
4. Printing Solution Path (print_path()):
   - Recursively prints the solution path by traversing the parent nodes starting from the last node in the queue.
5. Main Function:
   - Initializes the initial state.
   - Calls the bfs() function with the initial state to start the breadth-first search

Advantage

1. Completeness: BFS is guaranteed to find the shortest path between a starting node and any other reachable node in an unweighted graph. If there is a solution, BFS will find it.
2. Optimality: Because BFS explores all nodes at a given depth level before moving to the next depth level, it ensures that the first solution found is the shortest path to the goal.
3. Avoidance of Redundancy: BFS avoids exploring duplicate states or nodes, ensuring that each state is visited only once.
4. Applications: BFS is widely applicable and commonly used in various problems, such as shortest path problems, graph traversal, and maze solving.

Disadvantages Disadvantages

1. Memory Usage: BFS requires storing the entire frontier of nodes at each depth level in memory, which can be memory-intensive, especially for large graphs. This limits its applicability to graphs with limited sizes.
2. Time Complexity: While BFS guarantees optimality, its time complexity can be high, especially for graphs with many nodes and edges. The time complexity of BFS is O(V + E), where V is the number of vertices (nodes) and E is the number of edges in the graph.
3. Performance on Dense Graphs: BFS may perform poorly on dense graphs with a large number of edges, as it may need to explore a large number of nodes before finding a solution.
4. Not Suitable for Large Graphs: Due to its memory and time complexity constraints, BFS may not be suitable for solving problems on very large graphs or networks.

Output

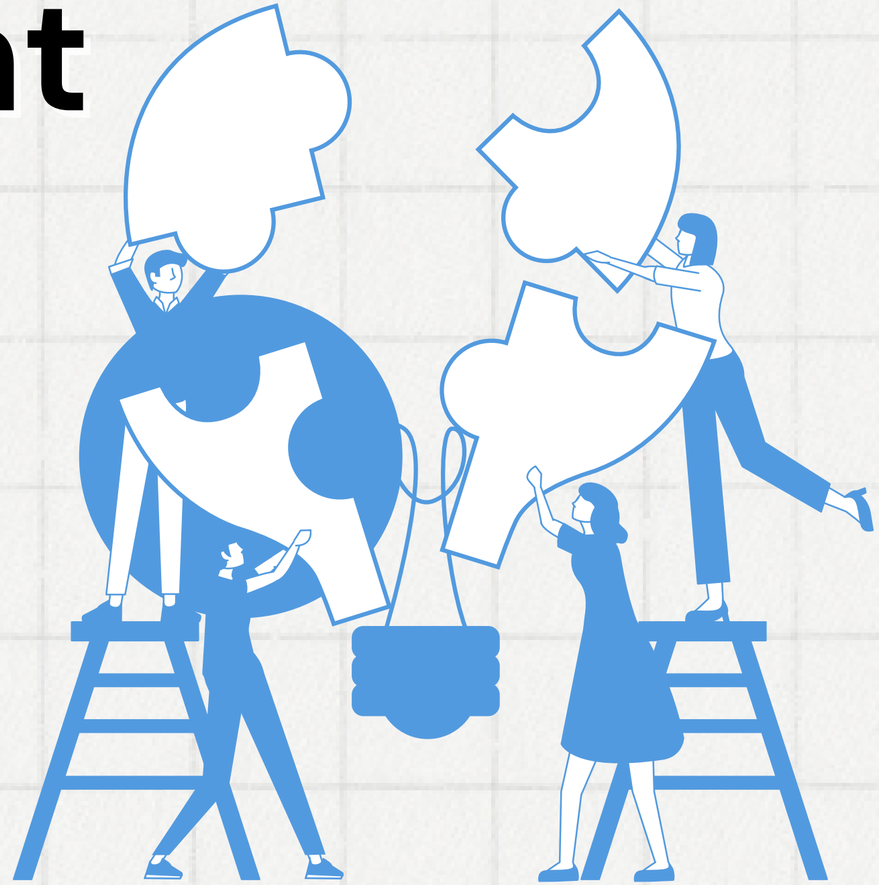
```
Solution Found:
Move: 2 missionaries and 0 cannibals from the left side to the right side.
Move: 0 missionaries and 2 cannibals from the right side to the left side.
Move: 2 missionaries and 0 cannibals from the left side to the right side.
Move: 0 missionaries and 2 cannibals from the right side to the left side.
Move: 1 missionaries and 1 cannibals from the left side to the right side.
Move: 0 missionaries and 2 cannibals from the right side to the left side.
Move: 0 missionaries and 2 cannibals from the left side to the right side.
Move: 0 missionaries and 2 cannibals from the right side to the left side.
Move: 0 missionaries and 2 cannibals from the left side to the right side.
```

Thank you very much!

8-Puzzle Problem using A* search

Problem Statement

- Given an 8-puzzle with a starting configuration of numbered tiles (1 to 8) and one empty space on a 3x3 grid, develop an algorithm that finds a sequence of legal moves (solution path) to reach the goal state from the starting configuration.
- The 8-puzzle problem asks for an algorithm to solve a sliding tile puzzle: rearrange numbered tiles on a 3x3 grid to reach a specific goal state using legal moves (sliding tiles into the empty space).
- Many possibilities: There are numerous ways to move the tiles, leading to a vast number of potential states to explore.
- Finding the best path: Not all moves lead directly to the goal. The challenge is to identify the sequence of moves that gets you there in the fewest steps or using some other optimality criteria.



Approaches

01.

Brute Force Method

This approach involves systematically exploring all possible states of the puzzle until the goal state is reached. While guaranteed to find a solution if one exists, it is generally impractical due to the vast number of possible states, resulting in exponential time complexity.

02.

Breadth-First Search

BFS explores all nodes at the current depth level before moving to the next level. It guarantees finding the shortest solution path, but it may require a large amount of memory, especially for deeper search trees.

03.

Depth-First Search

DFS explores as far as possible along each branch before backtracking. It can be memory efficient but may not always find the shortest solution, as it can get stuck in deep branches.

04.

A* Search Algorithm

A* search combines the advantages of informed and uninformed search by using a heuristic function to guide the search towards the goal state efficiently. It evaluates nodes based on a combination of the cost to reach them from the start and an estimate of the cost to reach the goal from them.

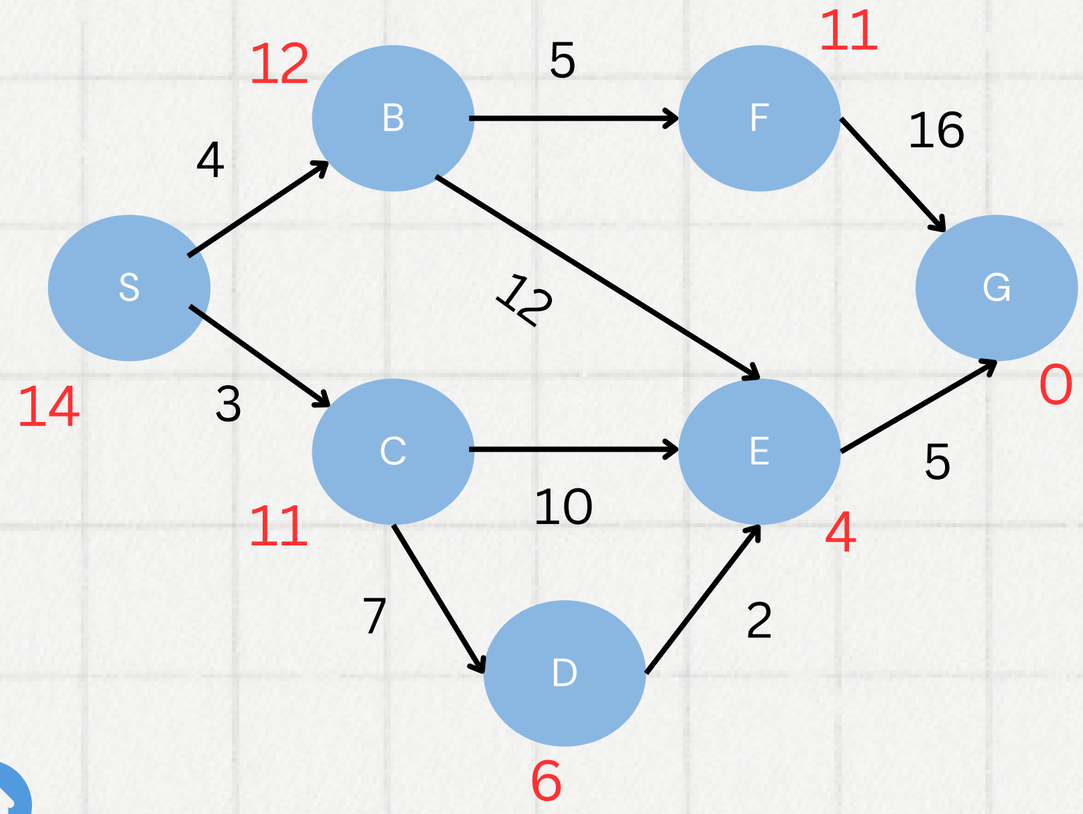
A* Algorithm

1. **Start:** Add the initial state to the Open List. Set its f(n) score to 0 (initial cost is 0).
2. **Iteration:** While the Open List is not empty:
    a. Select the state with the lowest f(n) score from the Open List (**prioritizes states closer to the goal**).
    b. Move this state to the Closed List.
    c. If the selected state is the goal state, the solution path is found (**stop**).
    d. Generate all possible successor states (reachable by legal moves).
    e. For each successor:
    f. Calculate its g(n) score (actual cost to reach that state from the start).
    g. Estimate its h(n) score using a heuristic (e.g., number of misplaced tiles or Manhattan distance).
    h. Calculate its total cost f(n) = g(n) + h(n).
    i. If the successor is not in the Closed List:
    j. Add the successor to the Open List with its f(n) score.
    k. If the successor is already in the Open List with a higher f(n) score (less promising path):
    l. Update the Open List with the lower f(n) score for this successor (**explores more efficient paths first**).
3. **No Solution:** If the Open List becomes empty, there's no solution reachable from the starting configuration.

A* Algorithm example

f(n) = g(n) + h(n)
g(n): actual cost from start node to n
h(n): estimate cost from n to Goal node using Manhattan distance

T.C = O(V + E) = O (b^d) , S.C = O(b^d)
V = vertices , E = edges
b = branching factor / no. of children of a particular node
d = depth



S->S : f(S) = 0 + 14

S->B: f(B) = 4+12 = 16 , S->C : f(B) = 3 + 11 = **14**

SC->E : f(E)= 3+ 10 + 4 = 17 ,

SC->D : f(D)=3 +7+ 6 = **16**

SCD->E : f(E)= 3 +7 + 2+ 4 = **16**

SCDE->G : f(G)= 3 +7 + 2+ 5 = **17**

Solving 8-puzzle using A*

1	2	3
0	4	6
7	5	8

Initial State

1	2	3
4	5	6
7	8	0

Final State

g=0 ,

h=3
(no. of miss-placed tiles)

f = 0 + 3 = **3**

1	2	3
4	0	6
7	5	8

g=1 , h=2

f = 1 + 2 = **3**

1	2	3
4	5	6
7	0	8

g=2 , h=1

f = 2 + 1 = **3**

1	2	3
4	5	6
7	8	0

g=3 , h=0

f = 3 + 0 = **3**

Data Structures

A priority queue implemented using the **heapq** module is utilized to efficiently explore states while solving the 8-puzzle problem with the A* search algorithm. Let's delve into how data structures like priority queues are utilized in the code:

1. **Priority Queue Initialization**:
   a. A priority queue is initialized using the **heapq** module's **heap** list. This list serves as the underlying data structure for the priority queue.
   b. The priority queue is used to store tuples **(cost, puzzle, path)**, where **cost** represents the cumulative cost of reaching the current state, **puzzle** represents the current state of the puzzle, and **path** represents the sequence of moves taken to reach the current state.
2. **Pushing States into the Priority Queue**:
   a. Initially, the priority queue is populated with the initial state of the puzzle, along with a cost of 0 and an empty path. This is done using **heapq.heappush(heap, (0, puzzle, []))**.
   b. States are pushed into the priority queue with their associated costs, ensuring that states with lower costs are explored first.
3. **Popping States from the Priority Queue**:
   a. States are popped from the priority queue using **heapq.heappop(heap)**. This operation retrieves the state with the lowest cost from the priority queue.
   b. The popped state represents the current state of the puzzle that needs to be explored further.
4. **Visited States**:
   a. A set named **visited** is used to keep track of visited states to avoid revisiting the same state multiple times. This helps in preventing infinite loops and redundant exploration.
   b. Before pushing a state into the priority queue, it is checked if the state has been visited before. If not, the state is added to the **visited** set, and then it is pushed into the priority queue.

Results

```
Found solution with cost 4

Start State:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Step 1:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 2:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 3:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

```
Found solution with cost 7

Start State:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]

Step 1:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Step 2:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 3:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 4:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Implementation Link

Advantages & Disadvantages

Pros	Cons
1. Finds the best path (if the guesstimate is good). 2. Saves time by focusing on promising paths first. 3. Adaptable to different problems with different guesstimates (heuristics).	1. Can be computationally expensive for very complex problems. 2. Relies on a good guesstimate (heuristic) - a bad one can lead it astray. 3. Might miss the best path if the guesstimate is overly optimistic.

1	2	3
4	5	6
7	8	0

THANK YOU

AO* Algorithm

Problem Statement

- The problem statement is to study and implement the AO* algorithm, a heuristic search algorithm based on problem decomposition, for efficient pathfinding in dynamic environments.
- The algorithm focuses on decomposing the problem into smaller pieces, calculating a cost function (F(n) = G(n) + H(n)) where H represents the heuristic or estimated value of nodes, and G represents the actual cost or edge value (with a unit value of 1 in this case).
- The problem involves developing a robust implementation of the AO* algorithm, focusing on effective problem decomposition and heuristic evaluation.
- The algorithm's performance will be evaluated in dynamic environments where the cost of edges is uniform (1) and the heuristic values play a crucial role in determining the optimal path

Approaches

01.

Brute Force Method

This approach involves systematically exploring all possible states of the puzzle until the goal state is reached. While guaranteed to find a solution if one exists, it is generally impractical due to the vast number of possible states, resulting in exponential time complexity.

02.

A* Search Algorithm

A* search combines the advantages of informed and uninformed search by using a heuristic function to guide the search towards the goal state efficiently. It evaluates nodes based on a combination of the cost to reach them from the start and an estimate of the cost to reach the goal from them.

03.

Dijkstras Algorithm

This approach involves systematically exploring all possible states of the puzzle until the goal state is reached. While guaranteed to find a solution if one exists, it is generally impractical due to the vast number of possible states, resulting in exponential time complexity.

04.

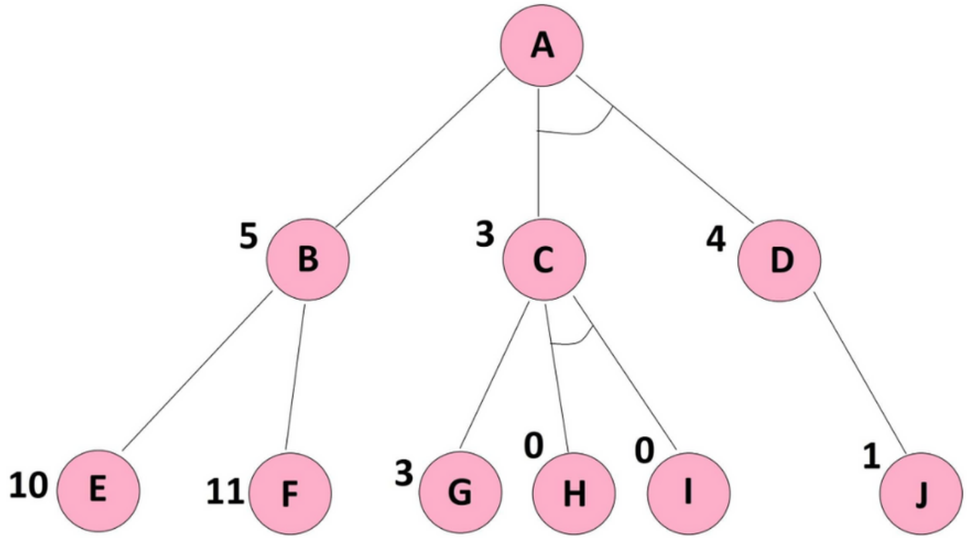
AO* Algorithm

The AO* (AO star) algorithm is a heuristic search approach designed for solving problems represented by And-Or graphs, emphasizing problem decomposition and a cost function This algorithm efficiently explores the graph, dynamically adapting to changes and providing an optimal path.

AO* Algorithm

1. **START:** Create an initial graph with a single node (start node).
2. Transverse the graph following the current path, accumulating node that has not yet been expanded or solved.
3. Select any of these nodes and explore it. If it has no successors then call this value- FUTILITY else calculate f'(n) for each of the successors.
4. If f'(n)=0, then mark the node as SOLVED.
5. Change the value of f'(n) for the newly created node to reflect its successors by backpropagation.
6. Whenever possible use the most promising routes, if a node is marked as SOLVED then mark the parent node as SOLVED.
7. If the starting node is SOLVED or value is greater than FUTILITY then **stop** else **repeat from Step-2.**

Example

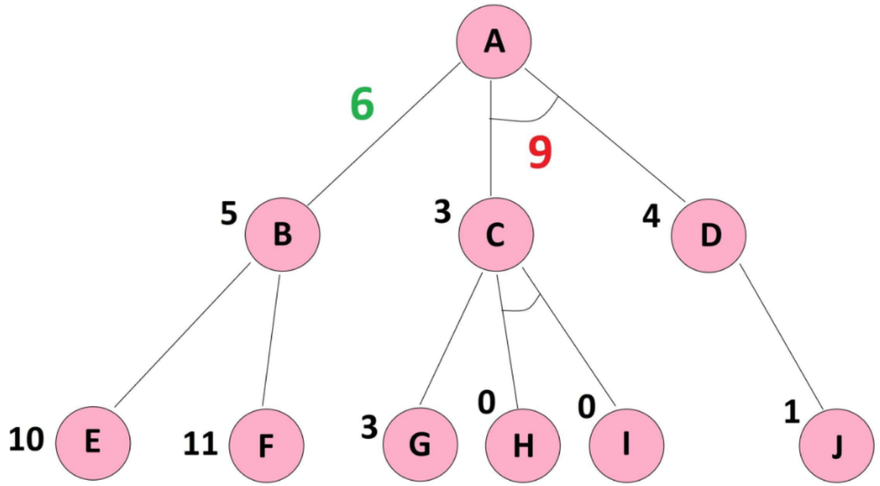


f(n) = g(n) + h(n)
g(n): actual cost from start node to n
h(n): estimate cost from n to Goal node

Note: For simplicity, the value for g(n) for all the edges is taken as 1

f(A-B) = g(AB) + h(B)
f(A-B) = 1 + 5 = 6

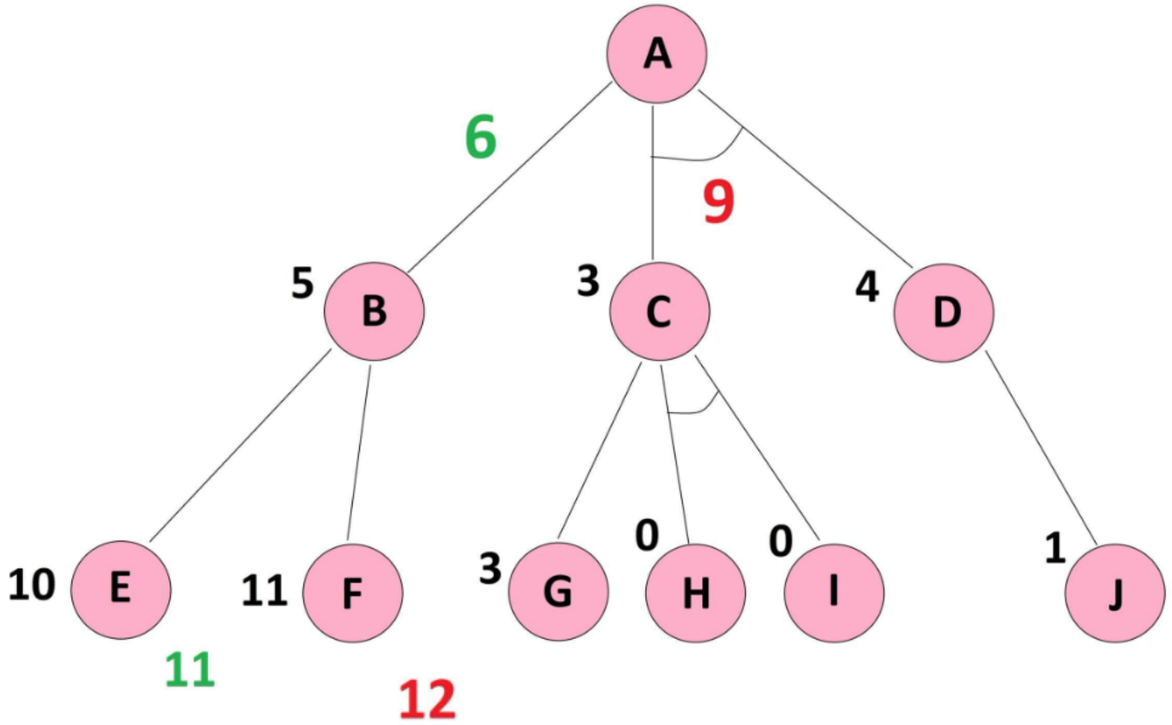
As there is an And Symbol f(A-CD) will be
f(A-CD) = g(AC) + h(C) + g(AD) + h(D)
f(A-CD) = 1 + 3 + 1 + 4 = 9

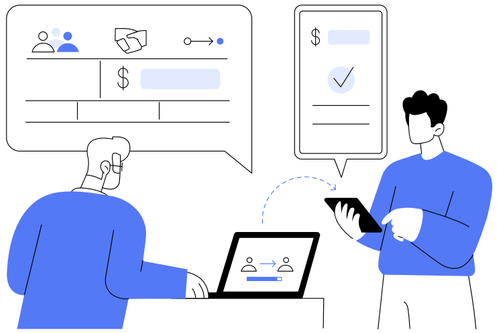


Example

f(A-B) is smaller then f(A-CD)
Hence we will take the A->B path.
Now similarly calculate the f(n) values for f(B-E) and f(B-F)

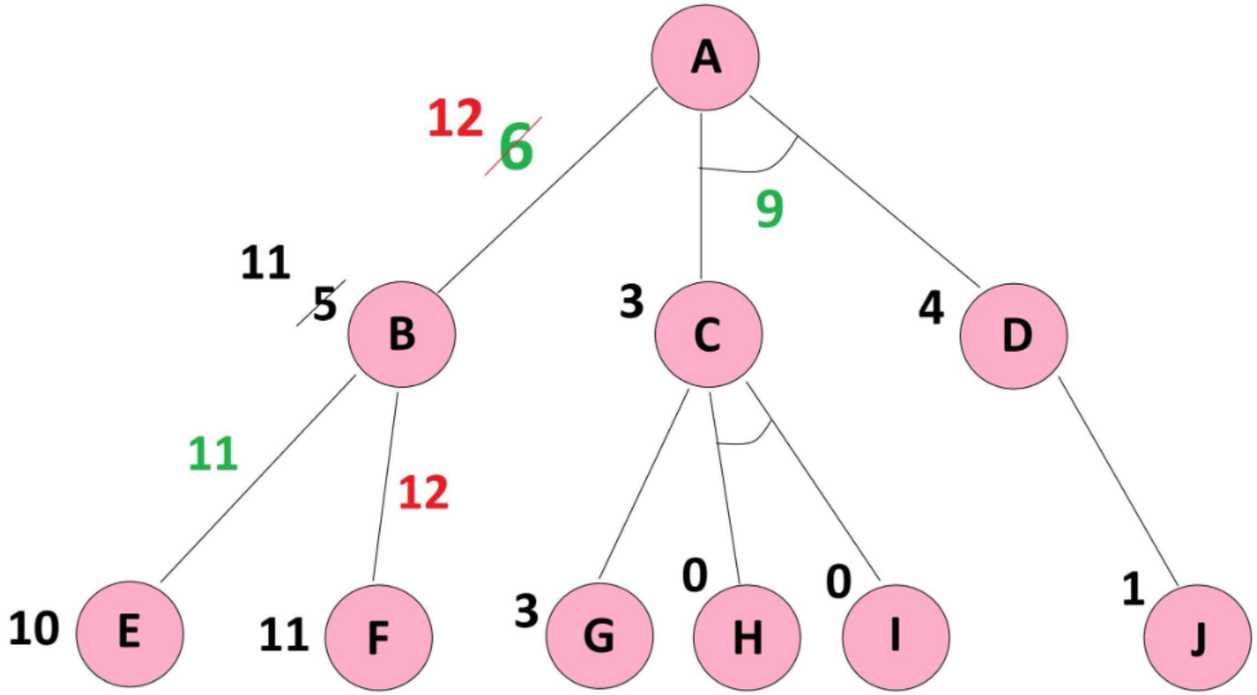
f(B -F) = g(BF) + h(F)
f(B -F) = 12
f(B -E) = g(BE) + h(E)
f(B -E) = 11
Here, f(B-E) has a lower cost and would be chosen





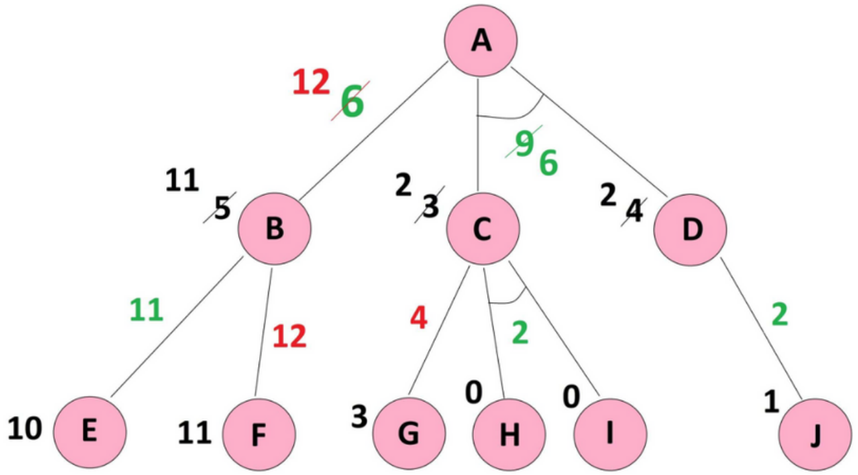
Example

We have reached the leaf node now we shall back track
Therefore, we can do the backpropagation and correct the heuristics of upper levels the updated , and as a consequence the updated .
The updated value of h(B) = f(B-E) = 11
and as a consequence the updated
f(A-B) = f(B) + updated h(B) = 1 + 11 = 12



Example

Similarly we have to do for other branches



The updated f(A-C-D) with the cost of 6 is still less then the updated f(A-B) with the cost of 12, and therefore, the minimum cost path from A to the goal node goes from f(A-C-D) by the cost of 6. We are done.

Advantages & Disadvantages

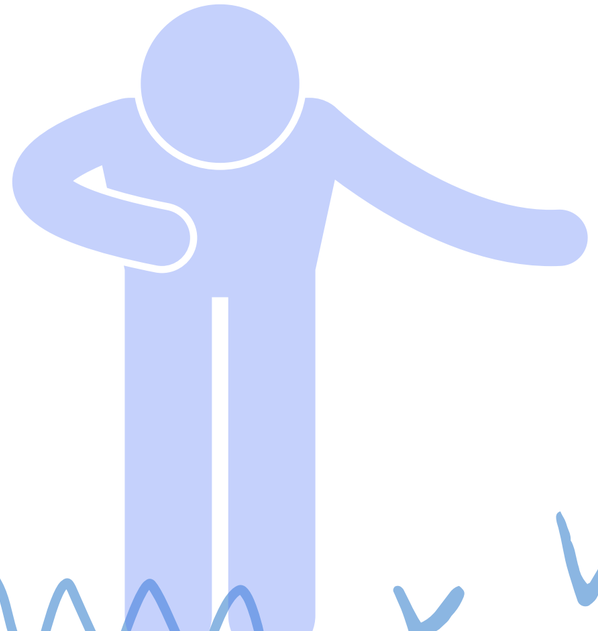
Pros	Cons
1. Finds the best path (if the guesstimate is good). 2. Saves time by focusing on promising paths first. 3. Adaptable to different problems with different guesstimates (heuristics).	1. Can be computationally expensive for very complex problems. 2. Relies on a good guesstimate (heuristic) - a bad one can lead it astray. 3. Might miss the best path if the guesstimate is overly optimistic.

Results

```
TRAVERSE THE GRAPH FROM THE START NODE: A
--------------------------------------------------------
{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}
--------------------------------------------------------
```

Implementation:- Document

THANK YOU



Hill Climb Algorithm

Artificial Intelligence

Problem Statement

You are given a set of blocks of different shapes and colors, arranged in a certain configuration in the Block World. The goal is to rearrange these blocks to achieve a specific desired configuration.

Initial State

The initial state consists of a set of blocks arranged in a specific configuration. Each block has a unique identifier, shape, and color, and they may be stacked on top of each other or arranged in different positions.

Goal State

The goal state specifies the desired configuration of the blocks. It defines how the blocks should be arranged, including their positions and any stacking relationships.

Example

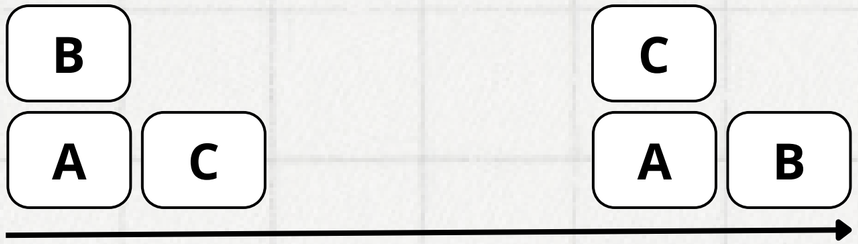
You are given a set of blocks with the following properties:

Block A | **Block B** | **Block C**

Initial State

The initial configuration of the blocks is as follows:

- Block A is on the table.
- Block B is stacked on top of Block A.
- Block C is on the table.



Final State

The goal is to arrange the blocks in the following configuration:

- Block A is on the table.
- Block C is stacked on top of Block A.
- Block B is on the table.

Approaches

There are several approaches you can consider to solve the Block World problem, Each approach has its advantages and limitations, and the choice of method depends on factors such as problem complexity, available computational resources, and desired solution quality.

01. **Brute Force Search:** Generate all possible sequences of actions and evaluate each sequence until the goal state is reached. This approach guarantees finding a solution if one exists but may be inefficient due to the large search space.

02. **Heuristic Search Algorithms:** Hill climbing iteratively improves solutions by moving to neighboring states with higher scores. A* search combines breadth-first & greedy best-first search, ensuring optimality with heuristic functions. IDA enhances memory efficiency by iteratively deepening search depth until a solution is found

03. **Constraint Satisfaction Problem (CSP) Solvers**: Formulate the Block World problem as a CSP and use constraint satisfaction techniques to find a solution. CSP solvers like backtracking, constraint propagation, and arc consistency can efficiently handle problems with discrete variables and constraints.

Approaches

There are several approaches you can consider to solve the Block World problem, Each approach has its advantages and limitations, and the choice of method depends on factors such as problem complexity, available computational resources, and desired solution quality.

04.

Planning Algorithms:
STRIPS: Formulate the problem as a series of states and actions and use planning algorithms like STRIPS to generate a plan to reach the goal state.
Graph Plan: Graph Plan is another planning algorithm that represents the problem as a planning graph and performs a search to find a valid plan.

05.

Evolutionary Algorithms: Use evolutionary algorithms such as genetic algorithms or simulated annealing to search for a solution by evolving a population of candidate solutions over multiple generations.

06.

Machine Learning Approaches: Train a machine learning model, such as a neural network, to predict the best action to take given a particular state of the Block World. Reinforcement learning techniques can also be applied to learn an optimal policy for solving the problem.

Actions & Contstraints

in Shaping Solution

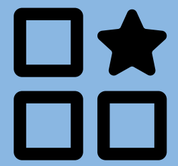
Actions

Move a block from one position to another.

Stack a block on top of another block.

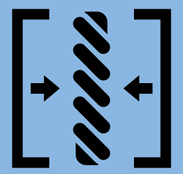
Unstack a block from the top of another block.

Constraints



Only one block can be moved at a time

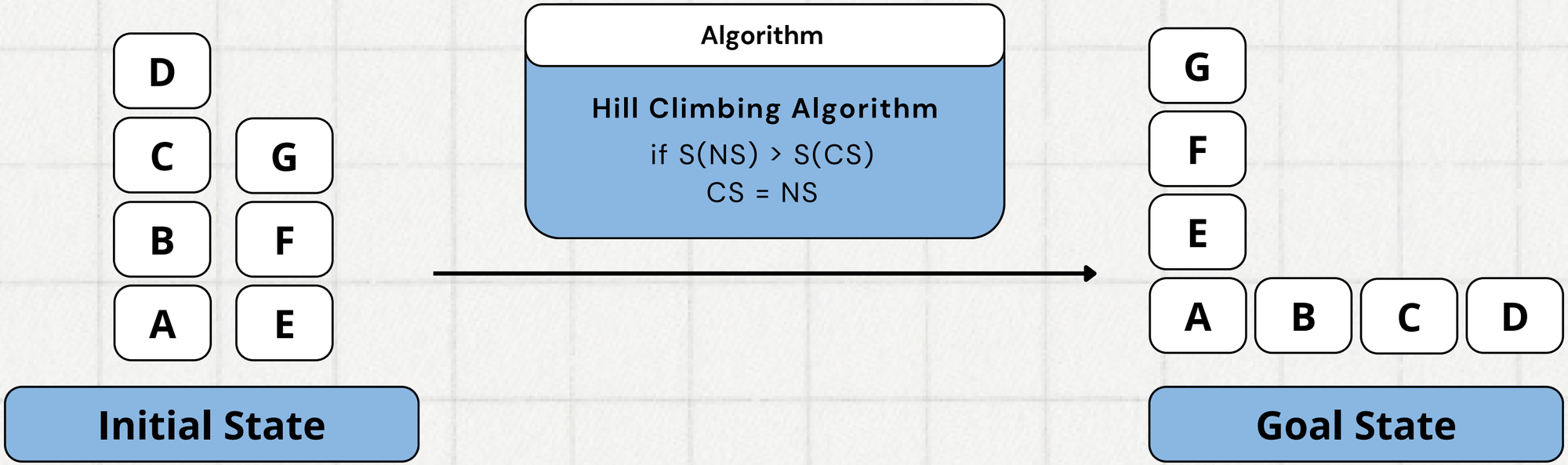
A block cannot be stacked on top of itself.



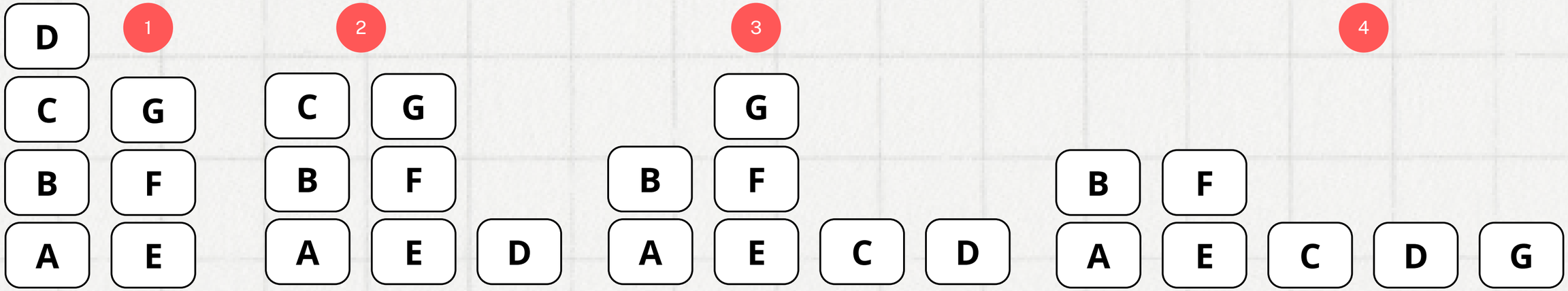
Stacking relationships must adhere to the physical constraints of the Block World.

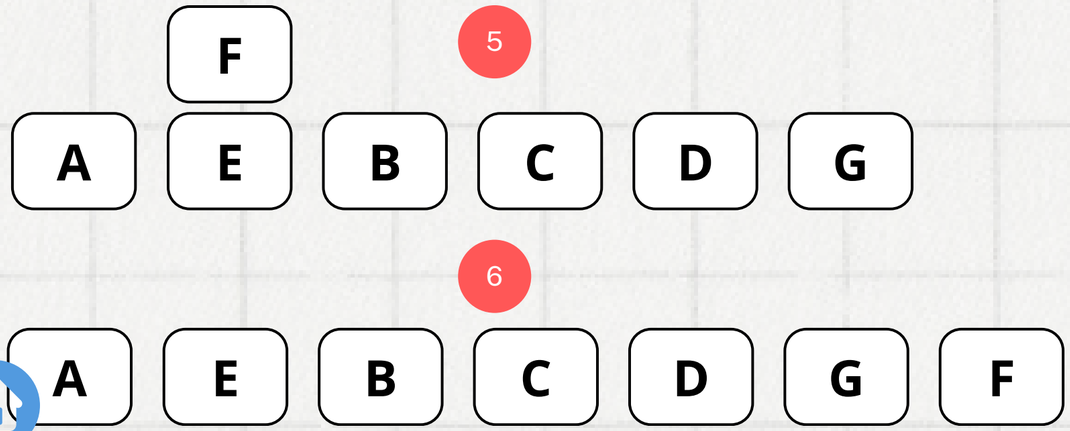
Example

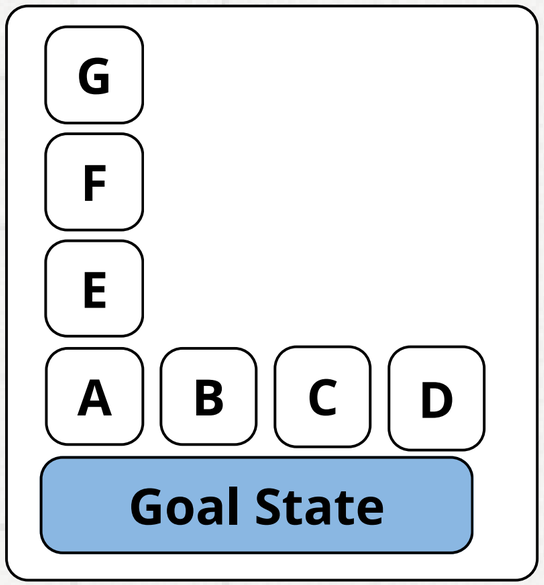
In this example, seven blocks (A, B, C, D, E, F, G) are arranged initially. The goal is to have Blocks A, B, C, and D on the table, with Blocks E, F, and G stacked as specified.



Example Step By Step







Example Step By Step

7

A E B C D G F

E

8

A B C D G F

9

F
E
A B C D G

G
F
E
A B C D

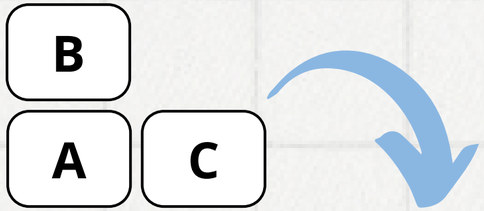
10

G
F
E
A B C D

Goal State

State (Blocks) Representation

In the provided implementation, the structure of the state is represented using a Python dictionary. Each block in the Block World is represented as a key in the dictionary, and the value associated with each key represents the position of that block. The position is represented as a tuple containing two elements:




```
state = {
    'A': ('on_table', None),  # Block A is on the table
    'B': ('on_top_of', 'A'),  # Block B is stacked on top of Block A
    'C': ('on_table', None)   # Block C is on the table
}
```

Hill Climbing Algorithm

Process

01

Initialization

The algorithm starts with the initial state of the Block World.

02

Evaluation

It evaluates the current state using the evaluate_state() method, which calculates a score indicating how close the current state is to the goal state.

03

Neighbor Generation

The algorithm generates neighboring states from the current state using the generate_neighbors() method. These neighboring states represent possible actions that can be taken from the current state, such as moving, stacking, or unstacking blocks.

04

Selection of Best Neighbor

It selects the best neighboring state with the highest evaluation score as the next state to explore. The best neighbor is chosen based on the evaluation function.

05

Iteration

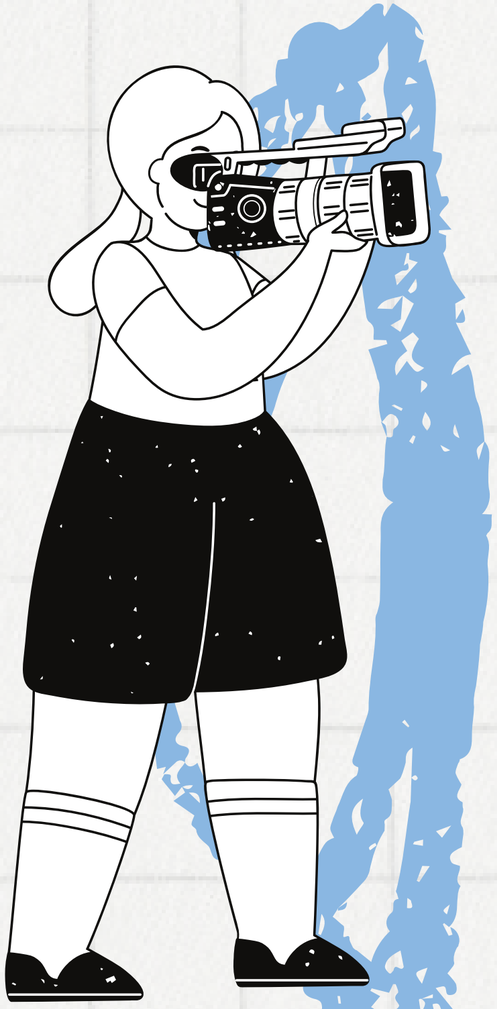
Steps 2-4 are repeated iteratively until one of the termination conditions is met:

06

Termination

The algorithm terminates when one of the following conditions is met:

- A goal state is reached
- No better neighboring state can be found



Results

The algorithm will either reach the goal state or terminate when no better neighboring state is found, leading to two possible outcomes for termination.

```
Final State:
Block A: on_table None
Block B: on_table None
Block C: on_table None
Block D: on_table None
Block E: on_table None
Block F: on_top_of E
Block G: on_top_of F
```

Code Implementation Link

Advantages & Disadvantages

Hill climbing is a simple and efficient optimization algorithm, it may not always be suitable for complex optimization problems with non-convex search spaces or where finding the global optimum is essential. Here are some advantages and disadvantages of using the hill climbing algorithm:

Advantages	Disadvantages
Simple and memory-efficient local search algorithm suitable for finding local optima in optimization problems.	Prone to getting stuck in local optima and lacking global perspective, limiting its effectiveness in complex search spaces.

THANK YOU!!