Thadomal Shahani Engineering College

Bandra (W.), Mumbai- 400 050.

CERTIFICATE

Certify that Mr./~~Miss~~ Om Aniruddha Shete of COMPUTER ENGG. Department, Semester VI with Roll No. 2103163 has completed a course of the necessary experiments in the subject Artificial Intelligence under my supervision in the **Thadomal Shahani Engineering College** Laboratory in the year 2023 - 2024

Teacher In-Charge

Head of the Department

Date 06-04-2024

Principal

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

CONTENTS

SR. NO.	EXPERIMENTS	DATE ~~NO.~~	PAGE NO	TEACHERS SIGN.
1.	Case study on AI application published in IEEE/ACM/ Springer or any prominent journel.	22/1	1-6	
2.	Implement DFS/DLS/DFID Search algorithm in Python	29/1	7-19	
3.	Implement BFS/UCS algorithm in Python	5/2	20-27	
4.	Implement Greedy Best First Search / A* algorithm in Python	26/2	28-38	
5.	Implement Genetics / Hill climbing in Python.	4/3	39-46	
6.	Knowledge Representation and Creating a knowledge base for Wumpus World	11/3	47-48	
7.	Planning for Blocks World problem.	18/3	49-50	
8.	Implementing Family tree using prolog.	18/3	51-56	
9.	Assignment - 1	29/1	57-68	
10.	Assignment - 2	4/3	69-81	

Experiment No 1

Aim : Case Study On AI applications Published in IEEE/ACM/Springer or any prominent journal

Theory :

Artificial Intelligence in Human Computing Games

1. Introduction

   The introduction sets the stage by highlighting the burgeoning interest in character-based AI and its diverse applications beyond entertainment. It underscores the transformative potential of AI in military and medical training, educational games, and those advocating for social causes. The main aim of the research is emphasized: to develop AI techniques that profoundly impact the gaming industry

2. AI in Games:

   2.1. Traditional Approaches:

   This subsection briefly outlines traditional approaches to AI in games, such as rule-based systems and decision trees.

   2.2. Minimax Algorithm with Alpha-Beta Pruning:

   The Minimax algorithm is a decision-making algorithm widely used in two-player turn-based games. It aims to minimize the possible loss for a worst-case scenario. To improve its efficiency, Alpha-Beta pruning is employed to reduce the number of nodes evaluated in the minimax tree.

   2.2.1. Minimax Algorithm:

   In the context of gaming, the Minimax algorithm involves a recursive search through the game tree, where each node represents a possible game state. It assigns a value to each node based on the outcome of the game from that state.

   The algorithm alternates between maximizing and minimizing players until a terminal state is reached.

   2.2.2. Alpha-Beta Pruning:

   Alpha-Beta pruning is a technique used to cut off branches of the search tree that will not affect the final decision. By maintaining two values, alpha and beta, representing the minimum score the maximizing player is assured of and the maximum score the minimizing player is assured of, respectively, the algorithm can disregard subtrees that are known to be irrelevant.

   2.2.3. Application in Computing Games:

   Consider a simple example of applying Minimax with Algorithm to Tic-Tac-Toe. The game tree represents all possible moves and counter-moves until a terminal state is reached. The algorithm assigns scores to each leaf node, and as it traverses back up the tree, it selects the move that maximizes or minimizes the score, depending on the player.

   This algorithm allows an AI agent to make optimal moves in Tic-Tac-Toe, considering all possible outcomes and efficiently pruning irrelevant branches in the search space.

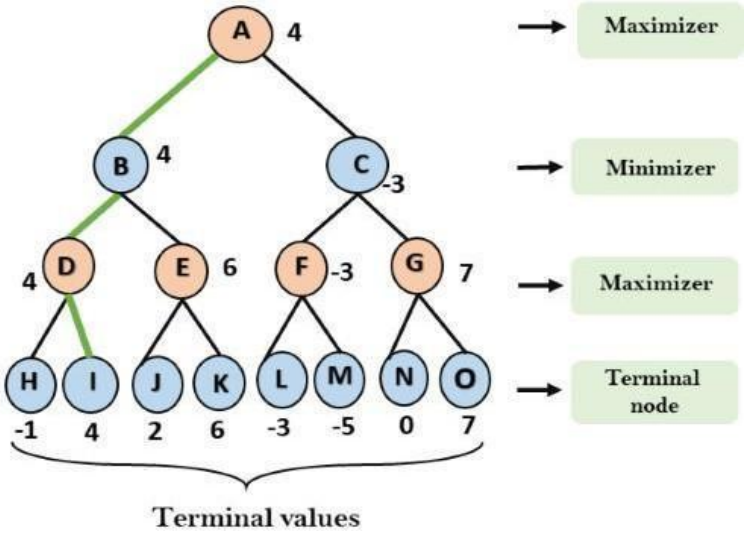
3. Challenges in AI:
   How Minimax addresses decision-making challenges in game AI and how Alpha-Beta pruning improves adaptability in the context of AI:
   3.1. Learning and Memory: Minimax in Game AI

   Minimax is a decision-making algorithm commonly used in game AI to handle complex decision spaces, especially in games with two players and perfect information (meaning that the entire state of the game is visible to both players). It's often applied in games like chess, tic-tac-toe, and checkers.

   Major Points:

   - Tree-based Decision Model: Minimax creates a game tree that represents all possible moves and counter-moves by both players. This tree allows the AI to explore different decision paths and evaluate the potential outcomes.
   - Maximization and Minimization: The algorithm alternates between maximizing and minimizing players. The maximizing player aims to achieve the highest possible score, while the minimizing player aims to minimize that score.
   - Backtracking and Decision Evaluation: As the algorithm explores the tree, it backtracks and assigns scores to different game states. The AI evaluates the desirability of a move based on the potential outcomes and selects the move with the highest score for the maximizing player and the lowest score for the minimizing player.

   Example : -The algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram



Properties of Mini-Max algorithm:

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.

- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

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. This limitation of the minimax algorithm can be improved from **alpha-beta pruning**

Addressing Challenges:

- Handling Complexity: Minimax effectively manages complex decision spaces by systematically exploring the possibilities in a game tree. It ensures a methodical approach to decision-making, even in situations with numerous potential moves.
- Strategic Decision-Making: By evaluating potential outcomes and selecting the move with the highest score, Minimax helps the AI make strategic decisions, especially in games where long-term planning is crucial.

3.2. Adaptability: Alpha-Beta Pruning

Alpha-Beta pruning is an optimization technique used in conjunction with the Minimax algorithm to enhance adaptability and efficiency in decision-making.

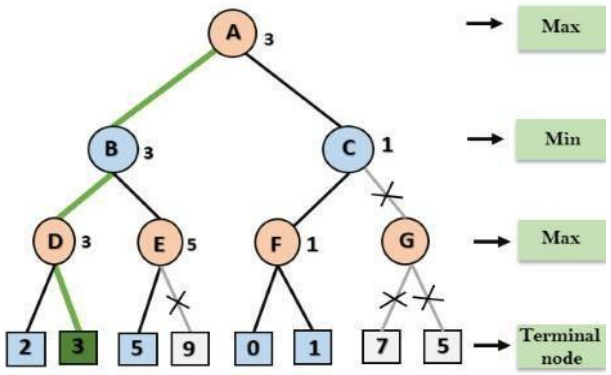
Major Points:

- Pruning Unnecessary Branches: Alpha-Beta pruning efficiently reduces the search space by eliminating branches in the game tree that are guaranteed to be irrelevant to the final decision. It does this by maintaining two values, alpha and beta, which represent the minimum score the maximizing player is assured of and the
  maximum score the minimizing player is assured of, respectively.
- Early Stopping: As the algorithm progresses, it stops exploring branches that cannot affect the final decision, saving computational resources and time.

Improving Adaptability:

- Faster Decision-Making: Alpha-Beta pruning significantly accelerates decision- making by avoiding unnecessary exploration of unpromising paths. This is particularly beneficial in games with large decision spaces, making the AI more adaptable in real-time scenarios.
- Resource Efficiency:The adaptability of the AI is improved by conserving computational resources. Alpha-Beta pruning allows the AI to focus on the most promising moves, leading to quicker and more resource-efficient decision-making.

Example: Let's take an example of two-player search tree to understand the working of Alpha-beta pruning



Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

In summary, Minimax with Alpha-Beta pruning addresses challenges in AI by providing a systematic approach to decision-making in complex game scenarios and enhancing adaptability through efficient search space reduction.

4. Behavior Adaptation for Believable Characters:This section delves into the intricacies of creating characters with distinct personalities in interactive games. The action transformation system is explored through a vivid example involving a game of tag. Emotion-based personality constraints and reasoning modules are detailed, showcasing how they determine behavior revisions in response to personality contract violations.

5. Case-Based Planning for Strategy Games:

   The paper introduces a novel approach that extracts behavioral knowledge from expert demonstrations to address the vast search spaces in computer games, particularly in strategy games. A detailed explanation is provided on how the case-based planning engine retrieves suitable behaviors from expert demonstrations and adapts them to the current game state.

6. Drama Management in Interactive Stories:
   This section focuses on drama management, a crucial aspect of interactive storytelling. The goal is to give players a significant impact during interaction, deviating from pre-written scripts. The paper discusses the three-module approach involving a game engine, player modeling module, and drama management module. The player modeling module's construction of player models based on reactions and the drama management module's planning actions for a more appealing story direction are explored.

7. Proposed Approach and Future Work:
   The proposed approach, comprising a game engine, player modeling module, and drama management module, is elucidated in detail. The player modeling module's construction of player models based on reactions and the drama management module's planning
   actions for a more appealing story direction are explored. Future work, including extensive player evaluation and the expansion of player modeling modules, is outlined.

8. Conclusion:
   This section synthesizes the challenges and AI techniques discussed throughout the paper. It emphasizes the experiments showing the positive impact of Drama Management techniques on player experience. The belief in AI's significant impact on entertainment, education, and training is reiterated, emphasizing the need for continuous innovation and incorporation of AI in games for truly adaptive and immersive experiences.

9. Acknowledgment:
   A brief acknowledgment expresses gratitude to collaborators involved in the projects mentioned in the paper. Special thanks are extended for assistance in drama management work, real-time strategy games, and behavior modification work.

In conclusion, this comprehensive exploration of AI in human-computer interaction within gaming provides a detailed understanding of its challenges, innovative approaches, and potential future developments. The four-page document encapsulates the complexity and significance of AI in shaping the future of gaming experiences.

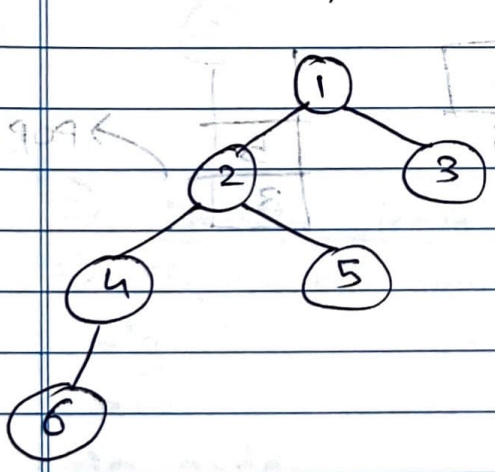
Experiment No : 2

Aim : Implement DFS / DFID / DLS search algorithm in python.

Theory :

1) **Depth First Search :**

- DFS stands for Depth First search where we first traversal in depth wise that is vertical search that is first we travel to left until we reach the end node its all level order traversal.
- And then we would backtrack to the previous ancestor node.
- It uses stack ferit's implementation.
- For e.g.,



Step 1 :

1	2	3	4	5	6
0	0	0	0	0	0

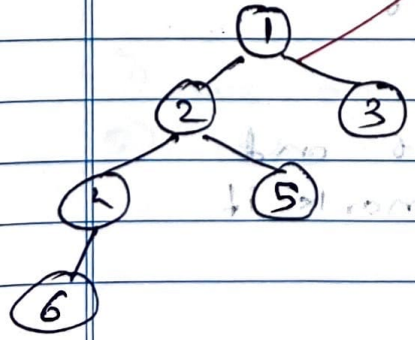
visited

1

stack

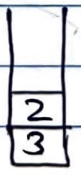
Insert root node into stack.

Step 2 : Now mark 1 as visited and put stack into its left and right node.



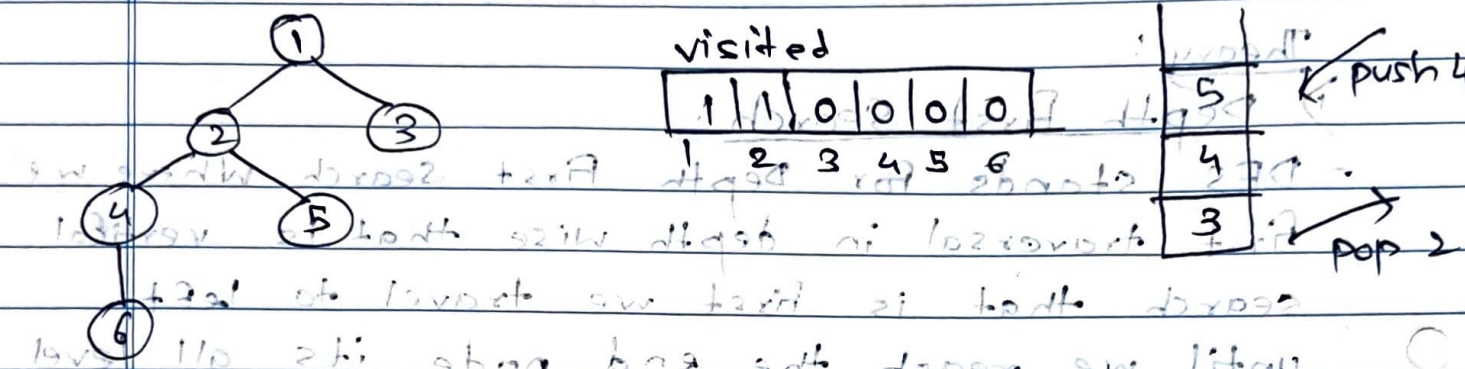
1	2	3	4	5	6
1	0	0	0	0	0

visited

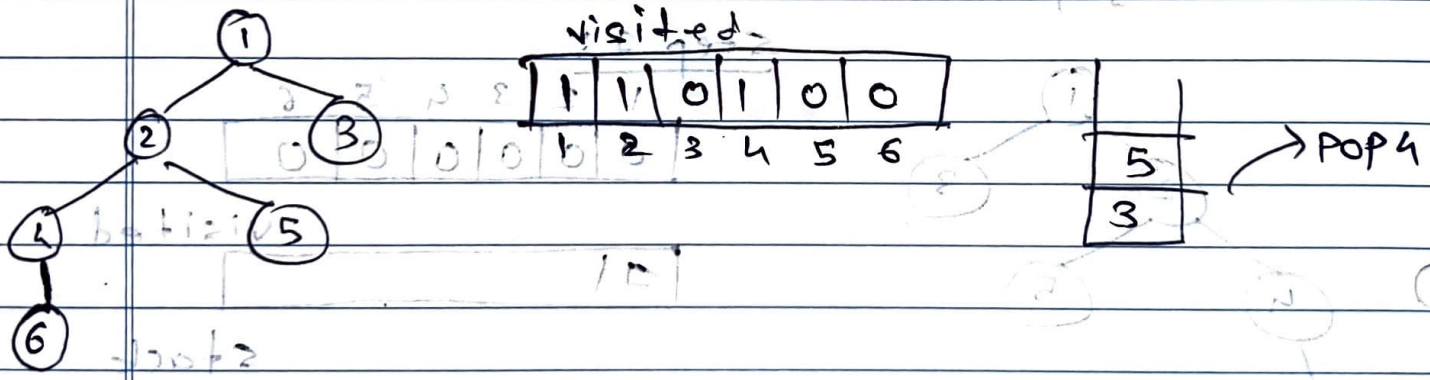


Pop 1

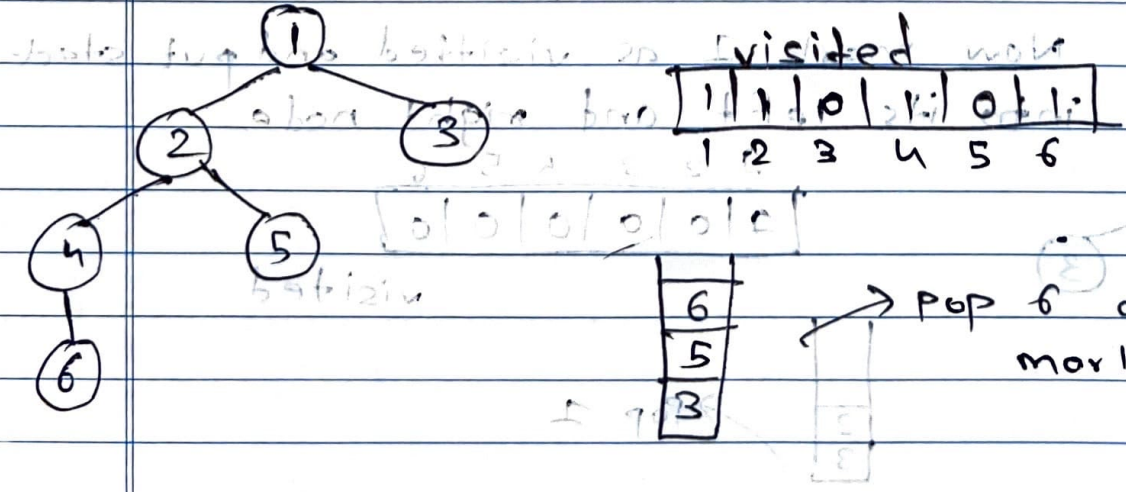
Step 3: Now will pop 2 and explore it and mark visited as 2



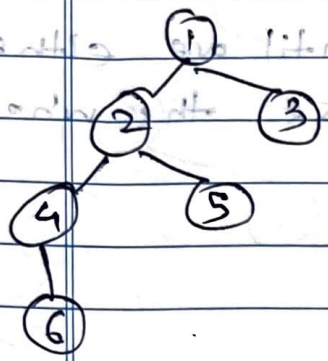
Step 4: Now explore 4 so make it as visited and push it corresponding left and right node



Step 5:

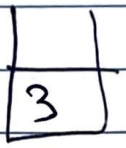


Step 6:



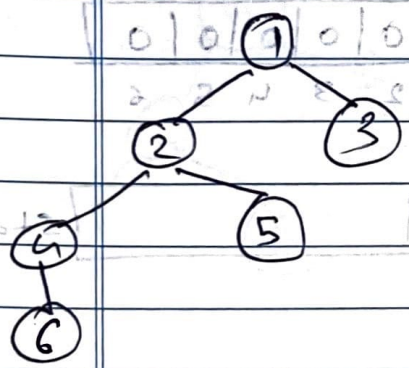
visited

1	1	0	1	1	1
1	2	3	4	5	6



→ pop 5 and mark it.

Step 7:



visited

1	1	1	1	1	1
1	2	3	4	5	6

Empty.

2) **Depth Limiting Search (DLS)**

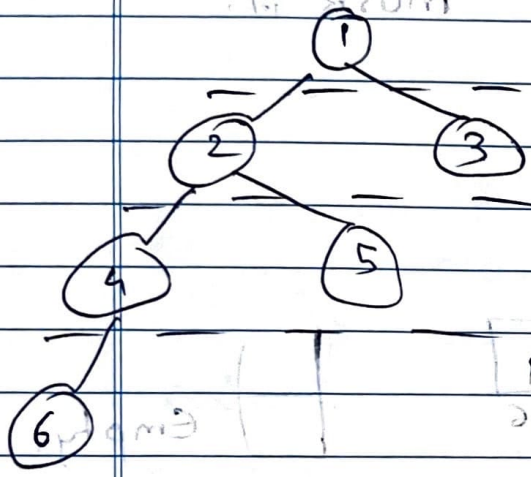
⇒ It's an variation of DFS travelling algorithm. It takes care of an edge case problem with DFS by implementing limiting in order to remove infinity problem.

- Basic DFS algo :-

1) We push the root node into the stack
2). We pop the root node before pushing all its child nodes into the stack.

3) We pop a child node before pushing all of its node back into the stack.
4) We repeat this process until we either find our result or traverse the whole tree.

For eg,



let say limit = 2

Step 1:

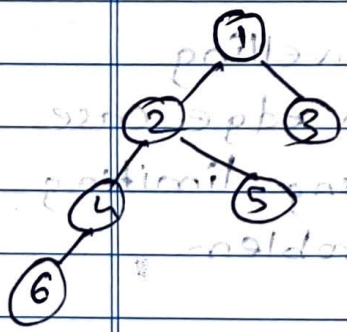
0	0	0	0	0	0
1	2	3	4	5	6

1

stack

Step 2: level counter = 0

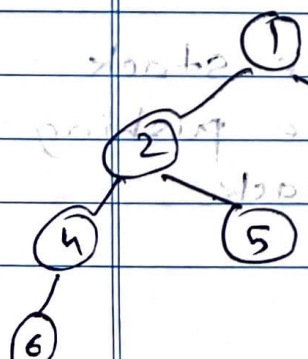
1	0	0	0	0	0
1	2	3	4	5	6



3
2

Step 3: level counter = 1

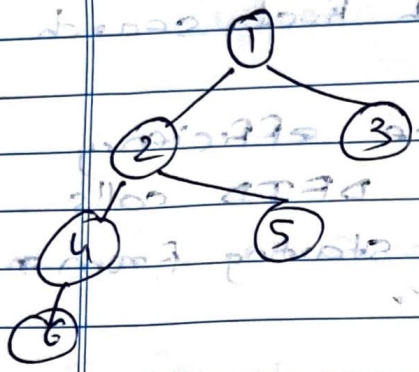
1	1	0	0	0	0
1	2	3	4	5	6



4
5
3

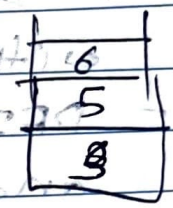
stack

Step 4: level counter = 2

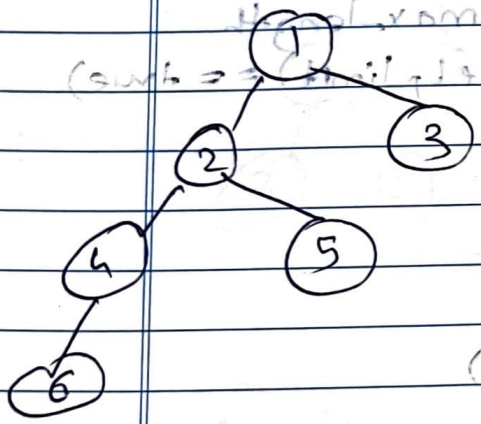


visited :-

1	2	3	4	5	6
1	1	0	1	0	0



Step 5: level counter = 2

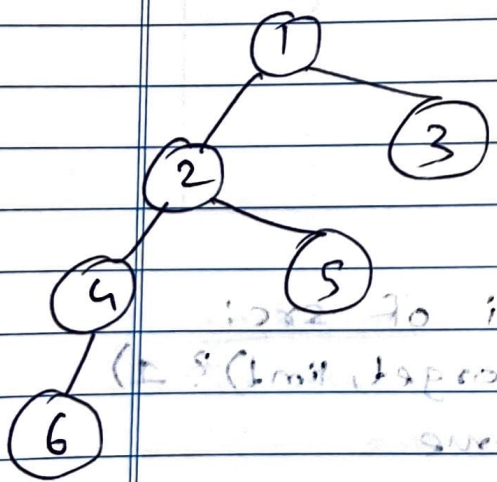


visited :-

1	2	3	4	5	6
1	1	0	1	1	0

can travers to 6 as level 3.

Step 6: level counter = 3



visited :-

1	2	3	4	5	6
1	1	0	1	1	0

Stack

3) DFID

⇒ DFID stands for the Depth first search with Iterative Deepning.

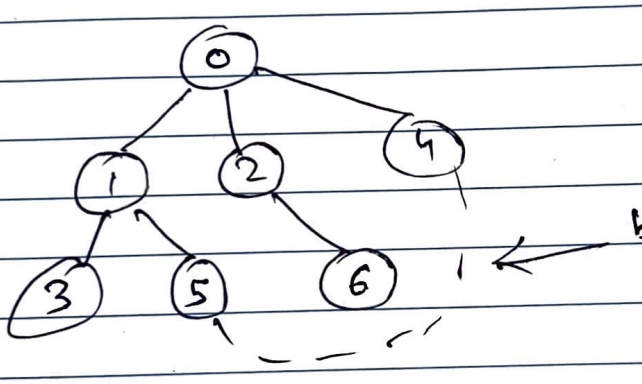
- DFID combines DFS space efficiency and BFS fast search DFID calls DFS for different depth starting from an initial value.
- pseudo code :-

```
bool DFID (src, target, max-depth)
    for limit from 0 to max-length
        if (DLS(src, target, limit) == true)
            return true
    return False.

bool DLS (src, target, limit)

    if (src == target)
        return true

    if (limit <= 0)
        return false

    for each adjacent i of src:
        if (DLS(src, target, limit) ? 1)
            return true
    return false.
```

- It's time complexity is $-O(b^d)$, where
  b = branching factor,
  d = depth limit.

- Space complexity - $O(d)$

- Examples :-



Depth	DFID
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4
3	0 1 3 5 4 2 6 4 5 2

- Conclusion :
In this experiment, we have learnd DFS, DLS, and DFID algorithms.

Code: (DFS)

```
graph = {
    '5': {
        '3': 2,
        '7': 1
    },
    '3': {
        '2': 3,
        '4': 4
    },
    '7': {
        '8': 5
    },
    '2': {},
    '4': {
        '8': 6
    },
    '8': {}
}

visited = set()
partial_explored = set()


def dfs_with_cost(visited, partial_explored, graph, node, path_cost,
goal_node):
  if node not in visited:
    print(f"{node} (Cost: {path_cost})")
    visited.add(node)

    if node == goal_node:
      print(f"Goal node {goal_node} reached!")
      return

    partial_explored.add(node)
    print(
        f"Nodes with partially explored children in current iteration:
{partial_explored}"
    )

    for neighbour, edge_cost in graph[node].items():
      if neighbour not in visited:
```

```
        dfs_with_cost(visited, partial_explored, graph, neighbour,
                      path_cost + edge_cost, goal_node)

goal_node = '8'
print(
    f"Following is the Depth-First Search with Path Cost to reach goal
node {goal_node}"
)
dfs_with_cost(visited, partial_explored, graph, '5', 0, goal_node)

print("\nNodes with partially explored children in each iteration:")
print(partial_explored)
```

Output:

```
Run
Following is the Depth-First Search with Path Cost to reach goal node 8
5 (Cost: 0)
Nodes with partially explored children in current iteration: {'5'}
3 (Cost: 2)
Nodes with partially explored children in current iteration: {'5', '3'}
2 (Cost: 5)
Nodes with partially explored children in current iteration: {'5', '2', '3'}
4 (Cost: 6)
Nodes with partially explored children in current iteration: {'5', '4', '2', '3'}
8 (Cost: 12)
Goal node 8 reached!
7 (Cost: 1)
Nodes with partially explored children in current iteration: {'5', '4', '2', '3', '7'}

Nodes with partially explored children in each iteration:
{'5', '4', '2', '3', '7'}
```

Code: (DLS)

```
graph = {
    '5': {
        '3': 2,
        '7': 1
    },
    '3': {
        '2': 3,
        '4': 4
    },
    '7': {
        '8': 5
    },
```

```
    '2': {},
    '4': {
        '8': 6
    },
    '8': {}
}

def dls_with_cost(graph, node, goal_node, depth_limit, path_cost,
visited,
                  partial_explored):
  if node not in visited:
    print(f"{node} (Cost: {path_cost})")
    visited.add(node)

  if node == goal_node:
    print(f"Goal node {goal_node} reached!")
    return True

  if depth_limit == 0:
    return False

  partial_explored.add(node)
  print(
      f"Nodes with partially explored children in current iteration:
{partial_explored}"
  )

  for neighbour, edge_cost in graph[node].items():
    if dls_with_cost(graph, neighbour, goal_node, depth_limit - 1,
                     path_cost + edge_cost, visited, partial_explored):
      print(f"{node} (Cost: {path_cost})")
      return True

  return False

goal_node_dls = '8'
depth_limit_dls = 3
visited_dls = set()
partial_explored_dls = set()
print(
    f"Following is the Depth-Limited Search with Path Cost (Depth
Limit: {depth_limit_dls})"
)
```

```
if not dls_with_cost(graph, '5', goal_node_dls, depth_limit_dls, 0,
                     visited_dls, partial_explored_dls):
  print(
      f"Goal node {goal_node_dls} not found within depth limit
{depth_limit_dls}"
  )
```

Output:

```
Following is the Depth-Limited Search with Path Cost (Depth Limit: 3)
5 (Cost: 0)
Nodes with partially explored children in current iteration: {'5'}
3 (Cost: 2)
Nodes with partially explored children in current iteration: {'5', '3'}
2 (Cost: 5)
Nodes with partially explored children in current iteration: {'2', '5', '3'}
4 (Cost: 6)
Nodes with partially explored children in current iteration: {'2', '5', '3', '4'}
8 (Cost: 12)
Goal node 8 reached!
4 (Cost: 6)
3 (Cost: 2)
5 (Cost: 0)
```

Code: (DFID)

```
graph = {
    '5': {
        '3': 2,
        '7': 1
    },
    '3': {
        '2': 3,
        '4': 4
    },
    '7': {
        '8': 5
    },
    '2': {},
    '4': {
        '8': 6
    },
    '8': {}
}
```

```
def dls_with_cost(graph, node, goal_node, depth_limit, path_cost,
visited, partial_explored):
  if node not in visited:
    print(f"{node} (Cost: {path_cost})")
    visited.add(node)

  if node == goal_node:
    print(f"Goal node {goal_node} reached!")
    return True

  if depth_limit == 0:
    return False

  partial_explored.add(node)
  print(
      f"Nodes with partially explored children in current iteration:
{partial_explored}"
  )

  for neighbour, edge_cost in graph[node].items():
    if dls_with_cost(graph, neighbour, goal_node, depth_limit - 1,
path_cost + edge_cost, visited, partial_explored):
      print(f"{node} (Cost: {path_cost})")
      return True

  return False


def dfid_with_cost(graph, start_node, goal_node, max_depth, visited,
partial_explored):
  for depth_limit in range(max_depth + 1):
    print(f"Depth-Limited Search with Depth Limit: {depth_limit}")
    if dls_with_cost(graph, start_node, goal_node, depth_limit, 0,
visited, partial_explored):
      return

goal_node_dfid = '8'
max_depth_dfid = 3
visited_dfid = set()
partial_explored_dfid = set()
print(
    f"Following is the Depth-First Iterative Deepening with Path Cost
(Max Depth: {max_depth_dfid})"
```

```
)
dfid_with_cost(graph, '5', goal_node_dfid, max_depth_dfid,
visited_dfid,
               partial_explored_dfid)
```

Output:

```
Following is the Depth-First Iterative Deepening with Path Cost (Max Depth: 3)
Depth-Limited Search with Depth Limit: 0
5 (Cost: 0)
Depth-Limited Search with Depth Limit: 1
Nodes with partially explored children in current iteration: {'5'}
3 (Cost: 2)
7 (Cost: 1)
Depth-Limited Search with Depth Limit: 2
Nodes with partially explored children in current iteration: {'5'}
Nodes with partially explored children in current iteration: {'3', '5'}
2 (Cost: 5)
4 (Cost: 6)
Nodes with partially explored children in current iteration: {'3', '7', '5'}
8 (Cost: 6)
Goal node 8 reached!
7 (Cost: 1)
5 (Cost: 0)
```

Experiment No: 3

Aim: Implementation of BFS and UCS

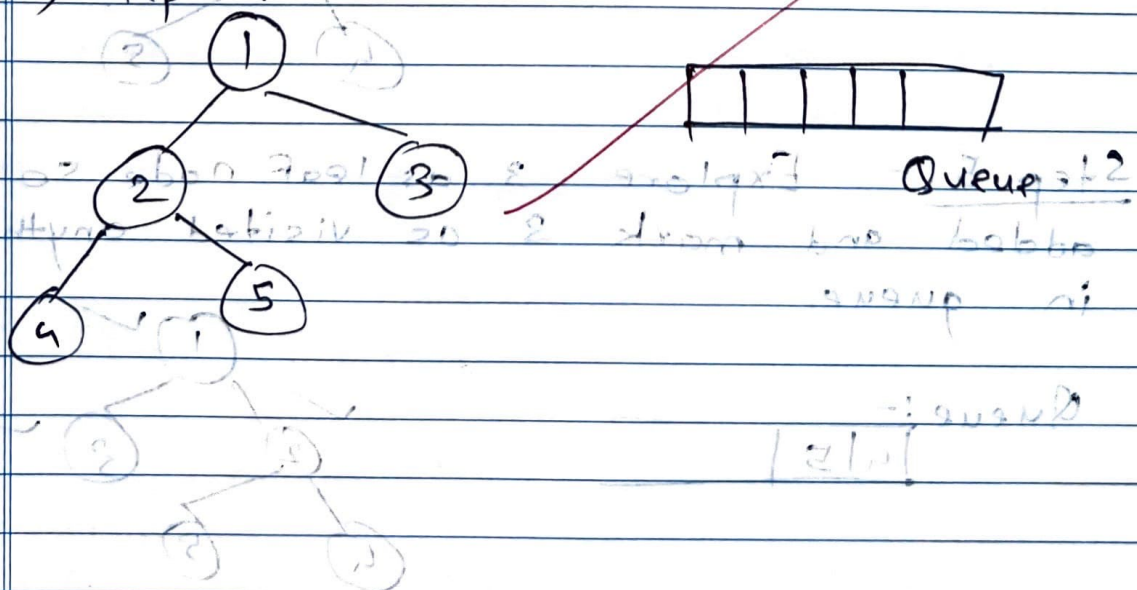
Theory:

1) Breadth first search (BFS)

⇒ BFS is a type of uniformed search, in this the search in horizontal. It's a level search algorithm.

- In BFS, queue data structure is used. In this first the root is pushed to the stack and then the front element is dequeued and then the children are pushed into the queue.
- This steps are repeated till the queue is empty.
- It's time complexity → $O(b^d)$
- It's space complexity → $O(b^d)$
- For e.g.;

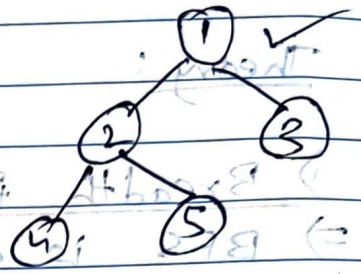
1) Step 1:



Step 2 :- Push the root node into a queue.

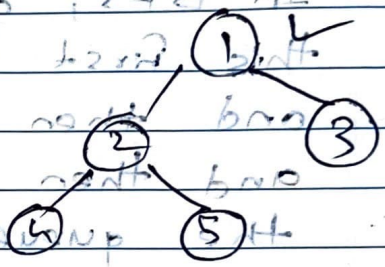
visited :- | 1 |

Queue :- | 1 |



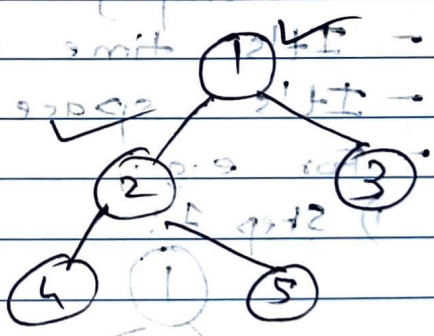
Step 3 :- Explore the children of queue.front() so push 2 and 3 into and pop 1 and mark it visited.

Queue :- | 2 | 3 |



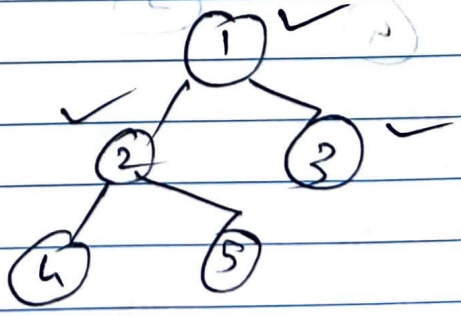
Step 4 :- Explore 2 so push 4 and 5

Queue :- | 3 | 4 | 5 |



Step 5 :- Explore 3 as leaf node so not added and mark 3 as visited anything in queue.

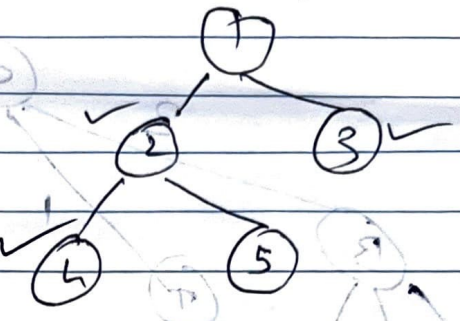
Queue :- | 4 | 5 |



Step 6:- Explore 4.

Queue :-

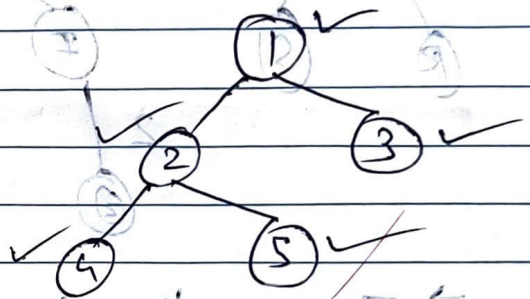
| 5 |



Step 7 :-

Queue :-

Empty



2) Uniformed Cost Search (UCS)

⇒ UCS is variant of Dijstra's algorithm.

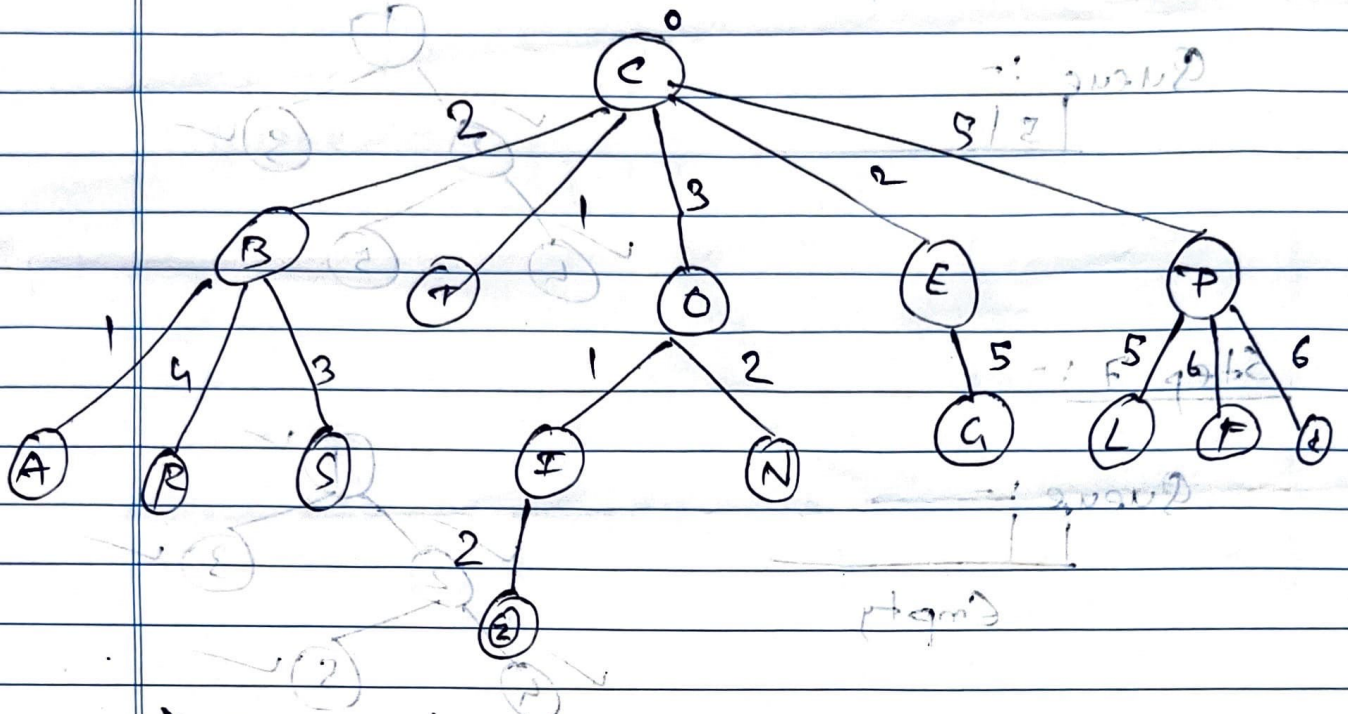
- Then instead of inserting all vertices into a priority queue, we insert only the same then one by one, we check if item is already in priority queue.
- Time complexity :-

$$O\left(b^{(1+c/2)}\right)$$

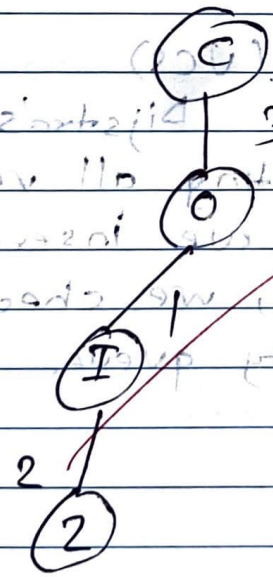
- Space complexity :-

$$O\left(b^{(1+c/2)}\right)$$

- For e.g.,



⇒ The path is the -



∴ The path cost = 3 + 1 + 2
= 6

∴ Traversal path ⇒ C - O - I - 2

- Conclusion: In this experiment, we learnt about BFS and UCS algorithm.

Code: (BFS)

```
from collections import deque

graph = {
    '5': {
        '3': 2,
        '7': 1
    },
    '3': {
        '2': 3,
        '4': 4
    },
    '7': {
        '8': 5
    },
    '2': {},
    '4': {
        '8': 6
    },
    '8': {}
}
visited = set()
partial_explored = set()


def bfs_with_cost(graph, start_node, goal_node):

  queue = deque([(start_node, 0)])

  while queue:
    node, path_cost = queue.popleft()

    if node not in visited:
      print(f"{node} (Cost: {path_cost})")
      visited.add(node)

      if node == goal_node:
        print(f"Goal node {goal_node} reached!")
        return

      partial_explored.add(node)
```

```
      for neighbour, edge_cost in graph[node].items():
        if neighbour not in visited:
          queue.append((neighbour, path_cost + edge_cost))

    print(
        f"Nodes with partially explored children in current iteration:
{partial_explored}"
    )

goal_node = '8'
print(
    f"Following is the Breadth-First Search with Path Cost to reach
goal node {goal_node}"
)
bfs_with_cost(graph, '5', goal_node)

print("\nNodes with partially explored children in each iteration:")
print(partial_explored)
```

Output:

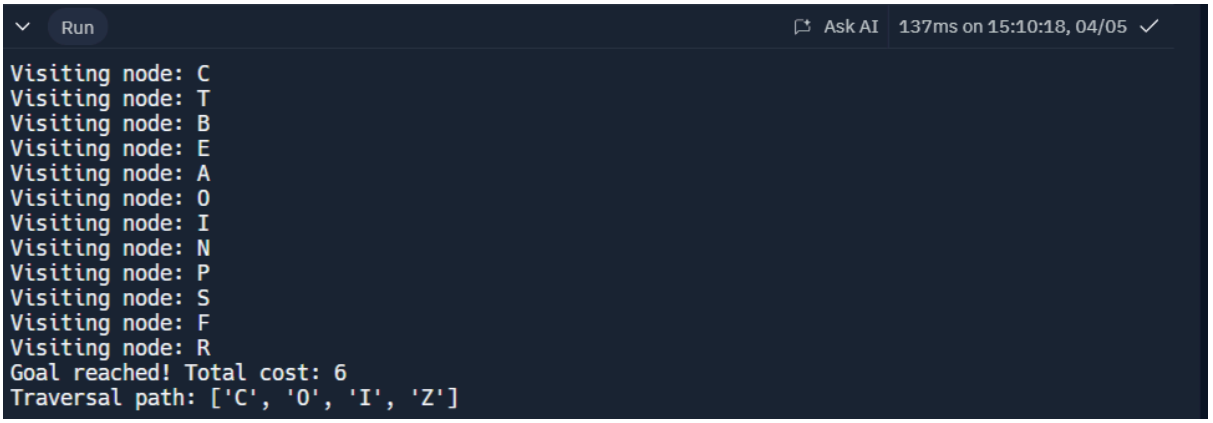
```
Following is the Breadth-First Search with Path Cost to reach goal node 8
5 (Cost: 0)
Nodes with partially explored children in current iteration: {'5'}
3 (Cost: 2)
Nodes with partially explored children in current iteration: {'3', '5'}
7 (Cost: 1)
Nodes with partially explored children in current iteration: {'7', '3', '5'}
2 (Cost: 5)
Nodes with partially explored children in current iteration: {'7', '3', '2', '5'}
4 (Cost: 6)
Nodes with partially explored children in current iteration: {'4', '3', '7', '2', '5'}
8 (Cost: 6)
Goal node 8 reached!

Nodes with partially explored children in each iteration:
{'4', '3', '7', '2', '5'}
```

Code: (UCS)

```
import heapq



def ucs(graph, start, goal):
  priority_queue = [(0, start, [start])]
  visited = set()
```

```
  while priority_queue:
    cost, current_node, path = heapq.heappop(priority_queue)

    if current_node == goal:
      print("Goal reached! Total cost:", cost)
      print("Traversal path:", path)
      return

    if current_node not in visited:
      print("Visiting node:", current_node)
      visited.add(current_node)

      for neighbor, edge_cost in graph[current_node]:
        if neighbor not in visited:
          heapq.heappush(priority_queue,
                         (cost + edge_cost, neighbor, path +
[neighbor]))


graph = {
    'A': [('B', 1)],
    'B': [('C', 2), ('A', 1), ('R', 4), ('S', 3)],
    'C': [('B', 2), ('T', 1), ('O', 3), ('E', 2), ('P', 5)],
    'R': [('B', 4)],
    'S': [('B', 3)],
    'T': [('C', 1)],
    'O': [('C', 3), ('I', 1), ('N', 2)],
    'I': [('O', 1), ('Z', 2)],
    'N': [('O', 2)],
    'Z': [('I', 2)],
    'E': [('C', 2), ('G', 5)],
    'G': [('E', 5)],
    'P': [('C', 5), ('L', 5), ('F', 1), ('D', 3)],
    'L': [('P', 5)],
    'F': [('P', 1)],
    'D': [('P', 3)]
}

ucs(graph, 'C', 'Z')
```

Output:


Run
Ask AI
137ms on 15:10:18, 04/05
Visiting node: C
Visiting node: T
Visiting node: B
Visiting node: E
Visiting node: A
Visiting node: O
Visiting node: I
Visiting node: N
Visiting node: P
Visiting node: S
Visiting node: F
Visiting node: R
Goal reached! Total cost: 6
Traversal path: ['C', 'O', 'I', 'Z']

Experiment No : 4

Aim : Implement Greedy Best First search / A* search algorithm in python.

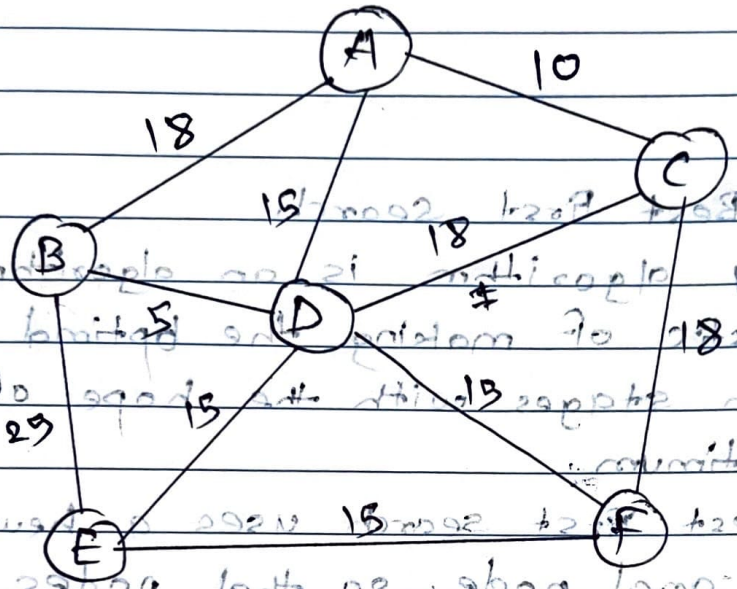
Theory :

- Greedy Best First Search

⇒ A greedy algorithm is an algorithm that follows the heuristic of making the optimal choice locally at each stages with the hope of finding a global optimum.

- When Best First search uses a heuristic that leads to goal node, so that nodes seems to be more promising, are expanded first.
- This particular type of search is called greedy-best-first search.
- In greedy best first search algorithm first successor of the parent is expanded. For the successor node, check the following :
  1) If the successor node's heuristic is better than the parent, the successor is set at the front of the queue (with the parent reinserted directly behind it) and the loop restarts.
  2) Else the successor is inserted into the queue, in a location determined by its heuristic value. The procedure will evaluate the remaining successor, if any of the parent.

- For eg:- Travelling salesman problem example :-



- As greedy algorithm, it will always make a local optimum choice. Hence it will select node C first as it found to be the one with less distance from the next non-visited node from node A, and then the generated will be A → C → D → B → E → F with the total cost = 10 + 18 + 5 + 25 + 15 = 73
- While by observing the graph one can find the optimal path and optimal distance the salesman needs to travel.
- It turns out to be, A → B → D → E → F → C, where the cost comes out to be :-
  18 + 5 + 15 + 15 + 18 = 68
- It's time complexity : $O(b^m)$
- It's space complexity : $O(b^m)$

- **A* Algorithm**

→ A* search algorithm is one of the best and popular technique used in path finding and graph traversals.

- Consider a square grid having many obstacles and we are given a starting cell and target cell.
- We want to reach the target cell from the starting cell as quickly as possible. Here A* algorithm comes to rescue.
- What A* search algorithm does is that at each step it picks the node according to a value 'f' which is a parameter equal to the sum of the two other parameters - 'g' and 'h'.
- At each step it picks the node/cell having the lowest 'f' and process that node/cell.
- **'g'** : the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
- **'h'** : the estimated movement cost to move from that given square on the grid to the final destination.

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

- For e.g.:
8 Puzzle problem using A* algorithm.

Start state

3	7	6
5	1	2
4	□	8

goal state

5	3	6
7	□	2
4	1	8

- Here, Firstly we will move the blank space to the up (1+3). Then we will again move the blank space to up (2+3).
- Now we will move the blank space to the left (3+2). Then we will move the blank space to the down (4+1).
- Finally we will move the blank space to the right and we will reach to the goal state.
- Its time complexity is $O(b^m)$.
- Its space complexity is $O(b^m)$.
- Conclusion: In this experiment, we have implemented Greedy Best first search and A* algorithm successfully.

Code: (A*)

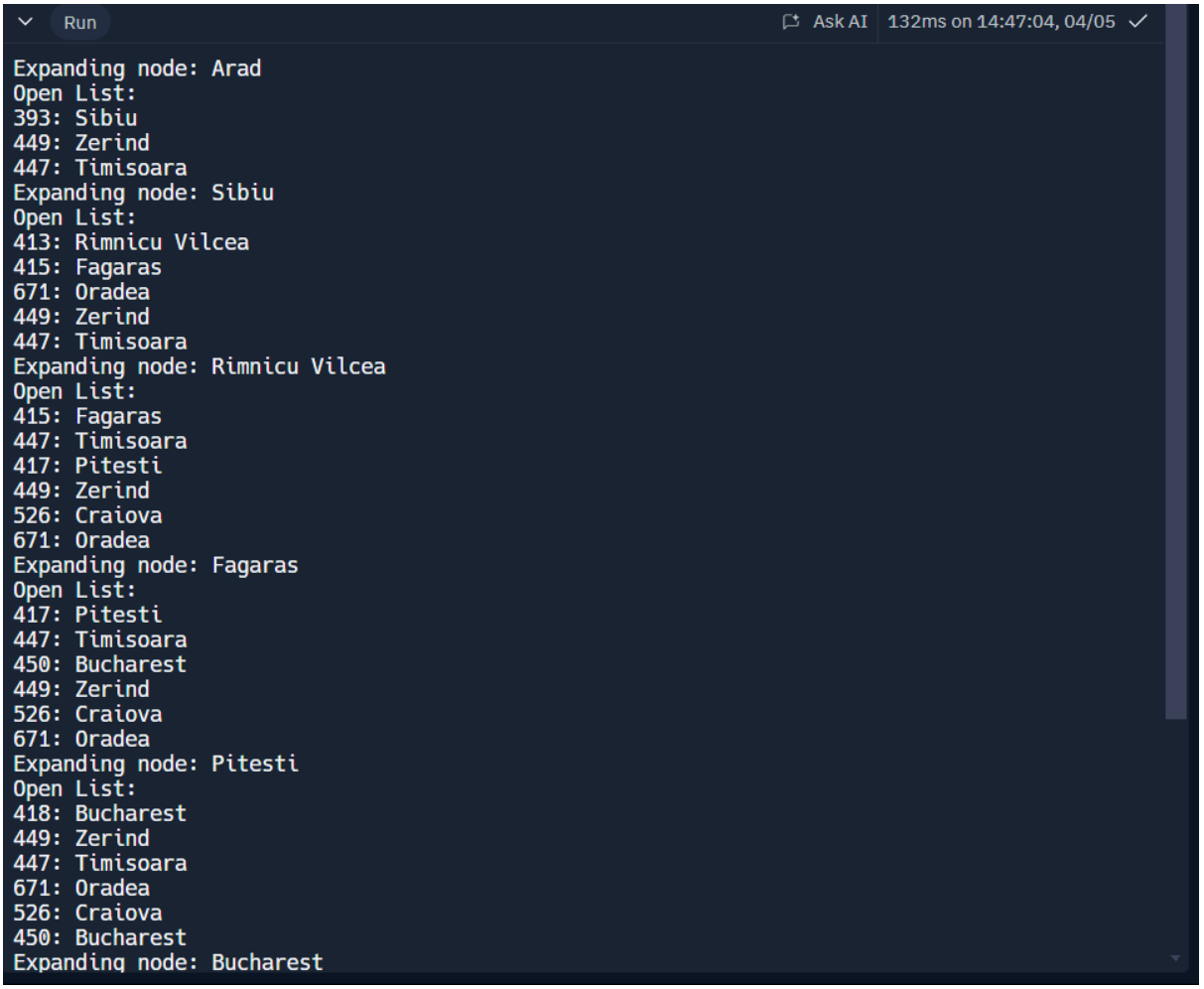
```
import heapq

romania_graph = {
    'Arad': {
        'Zerind': 75,
        'Sibiu': 140,
        'Timisoara': 118
    },
    'Zerind': {
        'Arad': 75,
        'Oradea': 71
    },
    'Oradea': {
        'Zerind': 71,
        'Sibiu': 151
    },
    'Sibiu': {
        'Arad': 140,
        'Oradea': 151,
        'Fagaras': 99,
        'Rimnicu Vilcea': 80
    },
    'Timisoara': {
        'Arad': 118,
        'Lugoj': 111
    },
    'Lugoj': {
        'Timisoara': 111,
        'Mehadia': 70
    },
    'Mehadia': {
        'Lugoj': 70,
        'Drobeta': 75
    },
    'Drobeta': {
        'Mehadia': 75,
        'Craiova': 120
    },
    'Craiova': {
        'Drobeta': 120,
        'Rimnicu Vilcea': 146,
```

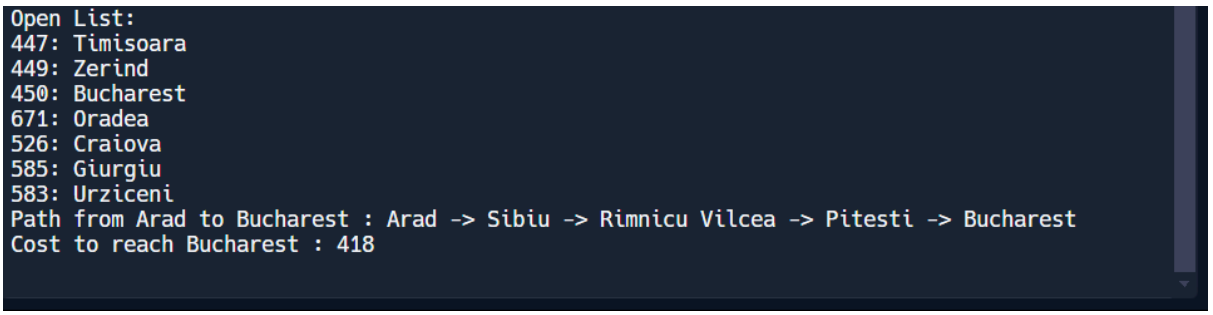
```
        'Pitesti': 138
    },
    'Rimnicu Vilcea': {
        'Sibiu': 80,
        'Craiova': 146,
        'Pitesti': 97
    },
    'Fagaras': {
        'Sibiu': 99,
        'Bucharest': 211
    },
    'Pitesti': {
        'Rimnicu Vilcea': 97,
        'Craiova': 138,
        'Bucharest': 101
    },
    'Bucharest': {
        'Fagaras': 211,
        'Pitesti': 101,
        'Giurgiu': 90,
        'Urziceni': 85
    },
    'Giurgiu': {
        'Bucharest': 90
    },
    'Urziceni': {
        'Bucharest': 85,
        'Vaslui': 142,
        'Hirsova': 98
    },
    'Hirsova': {
        'Urziceni': 98,
        'Eforie': 86
    },
    'Eforie': {
        'Hirsova': 86
    },
    'Vaslui': {
        'Urziceni': 142,
        'Iasi': 92
    },
    'Iasi': {
        'Vaslui': 92,
```

```
        'Neamt': 87
    },
    'Neamt': {
        'Iasi': 87
    }
}

heuristic = {
    'Arad': 366,
    'Bucharest': 0,
    'Craiova': 160,
    'Drobeta': 242,
    'Eforie': 161,
    'Fagaras': 176,
    'Giurgiu': 77,
    'Hirsova': 151,
    'Iasi': 226,
    'Lugoj': 244,
    'Mehadia': 241,
    'Neamt': 234,
    'Oradea': 380,
    'Pitesti': 100,
    'Rimnicu Vilcea': 193,
    'Sibiu': 253,
    'Timisoara': 329,
    'Urziceni': 80,
    'Vaslui': 199,
    'Zerind': 374
}


def a_star(graph, start, goal):
  open_nodes = []
  closed_nodes = set()

  heapq.heappush(open_nodes, (0 + heuristic[start], 0, start, []))

  while open_nodes:

    f, g, current_node, path = heapq.heappop(open_nodes)

    print("Expanding node:", current_node)
    for neighbor, cost in graph[current_node].items():
```

```
      neighbor_g = g + cost
      neighbor_f = neighbor_g + heuristic[neighbor]

      if neighbor in closed_nodes:
        continue

      neighbor_in_open_list = False
      for i, (f_val, g_val, node, _) in enumerate(open_nodes):
        if node == neighbor:
          neighbor_in_open_list = True
          break

      if neighbor_in_open_list and neighbor_g >= g_val:
        continue

      heapq.heappush(open_nodes,
                     (neighbor_f, neighbor_g, neighbor, path +
[current_node]))

    closed_nodes.add(current_node)

    print("Open List:")
    for f_val, _, node, _ in open_nodes:
      print(f"{f_val}: {node}")

    if current_node == goal:
      return path + [current_node], g
  return None, None


start_city = 'Arad'
goal_city = 'Bucharest'
path, cost = a_star(romania_graph, start_city, goal_city)
if path:
  print("Path from", start_city, "to", goal_city, ":", " ->
".join(path))
  print("Cost to reach", goal_city, ":", cost)
else:
  print("No path found from", start_city, "to", goal_city)
```

Output:


Run
Ask AI
132ms on 14:47:04, 04/05
Expanding node: Arad
Open List:
393: Sibiu
449: Zerind
447: Timisoara
Expanding node: Sibiu
Open List:
413: Rimnicu Vilcea
415: Fagaras
671: Oradea
449: Zerind
447: Timisoara
Expanding node: Rimnicu Vilcea
Open List:
415: Fagaras
447: Timisoara
417: Pitesti
449: Zerind
526: Craiova
671: Oradea
Expanding node: Fagaras
Open List:
417: Pitesti
447: Timisoara
450: Bucharest
449: Zerind
526: Craiova
671: Oradea
Expanding node: Pitesti
Open List:
418: Bucharest
449: Zerind
447: Timisoara
671: Oradea
526: Craiova
450: Bucharest
Expanding node: Bucharest

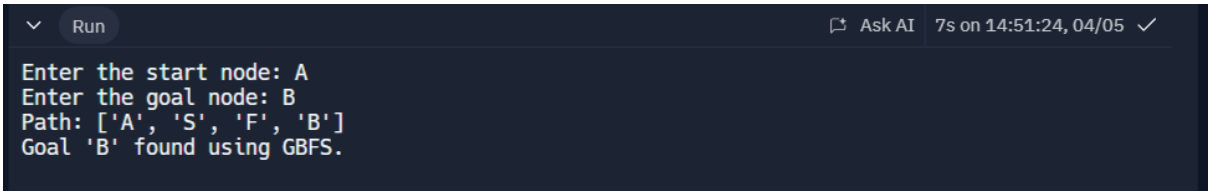

Open List:
447: Timisoara
449: Zerind
450: Bucharest
671: Oradea
526: Craiova
585: Giurgiu
583: Urziceni
Path from Arad to Bucharest : Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
Cost to reach Bucharest : 418

Code:

```
import heapq

def gbfs(graph, start, goal, heuristic):
  visited = set()
  priority_queue = [(heuristic[start], start)]
  path = {start: None}

  while priority_queue:
    _, current_node = heapq.heappop(priority_queue)
    if current_node == goal:
      return construct_path(path, start, goal)
    visited.add(current_node)

    for neighbor in graph[current_node]:
      if neighbor not in visited:
        heapq.heappush(priority_queue, (heuristic[neighbor], neighbor))
        path[neighbor] = current_node

  return None


def construct_path(path, start, goal):
  current_node = goal
  path_sequence = []

  while current_node:
    path_sequence.insert(0, current_node)
    current_node = path[current_node]

  return path_sequence


graph = {
    'A': ['S', 'T', 'Z'],
    'S': ['A', 'F', 'O', 'R'],
    'T': [],
    'Z': [],
    'F': ['S', 'B']
}

start_node = input("Enter the start node: ")
```

```
goal_node = input("Enter the goal node: ")

heuristic = {
    'A': 366,
    'S': 253,
    'F': 176,
    'T': 329,
    'O': 380,
    'Z': 374,
    'R': 193,
    'B': 0
}

gbfs_path = gbfs(graph, start_node, goal_node, heuristic)

if gbfs_path:
  print('Path:', gbfs_path)
  print(f"Goal '{goal_node}' found using GBFS.")
else:
  print(f"Goal '{goal_node}' not found using GBFS.")
```

Output:



Experiment No : 5

Aim : Implement Genetics / Hill Climbing in python

Theory :

- **Hill Climbing**

⇒ Hill climbing is simply a combination of depth first with generate and test where a feedback is used here to decide on the direction of motion in the search space.

- Hill climbing technique is used widely in AI, to solve computationally hard problems, which has multiple possible solutions.
- In the DFS, the test function will accept or reject a solution. But in Hill climbing the test function is provided with a heuristic function which provides an estimate of how close a given state is to goal state.
- In hill climbing, each state is provided with the additional information needed to find the solution, i.e. the heuristic value.
- The algorithm is memory efficient since it does not maintain the complete search tree.
- For eg. if you want to find a mall from your current location.
- There are n possible paths with different directions to reach the mall. The heuristic function will just give you the distance of each path which is reaching to the mall,

so that it becomes very simple and time efficient for you to reach to the mall.

- Algorithm :

1) Evaluate the initial state. If it is a goal state, then return and quit; otherwise make it a current state and go to step 2.
2) Loop until a solution is found or there are no new operators left to be applied.
   a) Select and apply new operator.
   b) Evaluate the new state :
      i) If it is a goal state, then return and quit.
      ii) If it is better than current state then make it a new current state.
      iii) If it is not better than the current state then continue the loop, go to step 2.

- Genetics Algorithm

⇒ GAs are adpative heuristic search algorithms based on the evolutionary ideas of natural selection and genetics.

- As they represent an intelligent exploitation of a random search used to solve optimization problems.
- Although "randomized", GAs are by no means random, instead they exploit historical information to direct the search

- for better performance within the search space.
- The basic techniques of the GAs are designed to simulate processes in natural systems necessary for evolution, especially those of following the principles of "Survival of the fittest" laid down by Charles Darwain
- For e.g.,

Parents chromosomes

Chromosome A	1 5 3 2 6 4 7 9 8
Chromosome B	8 5 6 7 2 3 1 4 9

- Let's consider the example of pair chromosome encoded using permutation encoding technique and are undergoing the complete process of GA.
- Assuming that they are selected using rank selection method and will be applied, arithmetic crossover and value encoding mutation techniques.
- Child chromosome after arithmetic crossover i.e. adding bits of both chromosomes.

Child chromosome :

Chromosome C	9 0 9 9 8 7 8 3 7

- After apply value encoding mutation i.e. adding or substraction a small value to selected values e.g. substraction 1 to 3rd and 4th bit

- Child chromosome

Chromosome C	90 8 -8 8 -7 8 37

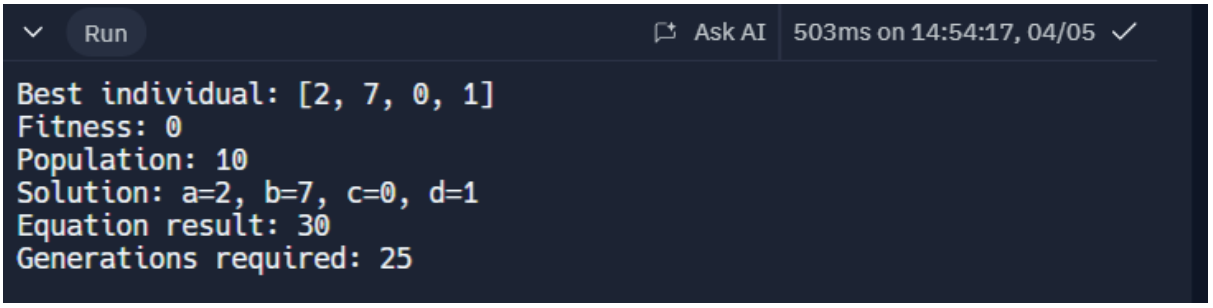
It can be observed that this child produced is much better than both parents.

Code: (Genetics*)

```
import random

def equation(a, b, c, d):
  return a + 2 * b + 17 * c + 14 * d


def eval_equation(individual):
  a, b, c, d = individual
  result = equation(a, b, c, d)
  return abs(result - 30)


def generate_individual(size):
  return [random.randint(0, 10) for _ in range(size)]


def crossover(parent1, parent2, crossover_prob):
  if random.random() < crossover_prob:
    crossover_point = random.randint(0, len(parent1) - 1)
    child = parent1[:crossover_point] + parent2[crossover_point:]
  else:
    child = parent1
  return child


def mutate(individual, mutation_rate):
  mutated_individual = individual[:]
  for i in range(len(mutated_individual)):
    if random.random() < mutation_rate:
      mutated_individual[i] = random.randint(0, 10)
  return mutated_individual


def genetic_algorithm(population_size, mutation_rate, crossover_prob):
  best_fitness = float('inf')
  best_individual = None
  generations = 0

  while True:
    generations += 1
```

```
    population = [generate_individual(4) for _ in
range(population_size)]

    for gen in range(population_size):
      fitnesses = [eval_equation(ind) for ind in population]

      min_fitness = min(fitnesses)
      best_index = fitnesses.index(min_fitness)
      best_individual = population[best_index]

      if min_fitness == 0:
        break

      selected = [random.choice(population) for _ in
range(population_size)]

      offspring = []
      for i in range(0, population_size, 2):
        parent1, parent2 = selected[i], selected[i + 1]
        child = crossover(parent1, parent2, crossover_prob)
        child = mutate(child, mutation_rate)
        offspring.extend([child])

      population[:] = offspring

    if min_fitness == 0 or generations >= 40:
      break

  print("Best individual:", best_individual)
  print("Fitness:", min_fitness)
  a, b, c, d = best_individual
  print("Population:", population_size)
  print("Solution: a={}, b={}, c={}, d={}".format(a, b, c, d))
  print("Equation result:", equation(a, b, c, d))
  print("Generations required:", generations)


if __name__ == "__main__":
  population_size = 10
  mutation_rate = 0.2
  crossover_prob = 0.5
  genetic_algorithm(population_size, mutation_rate, crossover_prob)
```

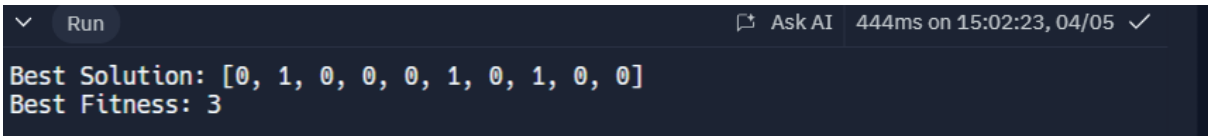
Output:



Code:

```
import random

def objective_function(solution):
  return sum(solution)

def generate_neighbor(current_solution):
  neighbor = current_solution[:]
  index = random.randint(0, len(neighbor) - 1)
  neighbor[index] = 1 - neighbor[index]
  return neighbor

def hill_climbing():
  current_solution = [random.randint(0, 1) for _ in range(10)]
  current_fitness = objective_function(current_solution)

  while True:
    neighbor = generate_neighbor(current_solution)
    neighbor_fitness = objective_function(neighbor)

    if neighbor_fitness >= current_fitness:
      current_solution = neighbor
      current_fitness = neighbor_fitness
    else:
      break

  return current_solution, current_fitness

best_solution, best_fitness = hill_climbing()
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:


Run
Ask AI
444ms on 15:02:23, 04/05
Best Solution: [0, 1, 0, 0, 0, 1, 0, 1, 0, 0]
Best Fitness: 3

Experiment No : 6

Aim : Knowledge representation and creating a knowledge base for Wumpus world

Theory :

- Humans are best at understanding, reasoning and interpreting knowledge.
- Human knows things, which is knowledge and as per their knowledge they perform various actions in the real world.
- But how machines do all these things comes under knowledge representation and reasoning.
- Hence we can describe knowledge representation as following :

1) Knowledge representation and Reasoning (KR, KRR) is the part of Artificial intelligence which concerned with AI agents thinking and how thinking contributes to intelligent behaviour of agents.
2) It is responsible for representing info about the real world so that a computer can understand and can utilize this knowledge to solve the complex real world problems such as diagnosis a medical condition or communicating with humans in natural language.
3) It is also a way which describes how we can represent knowledge in AI. Knowledge representation is not just storing data into some database, but it also enables intelligent

machine to learn from that knowledge and experiences so that it can behave intelligently like a human.

- The Wumpus World Environment

⇒ The Wumpus world's agent is an example of a knowledge-based agent that represents knowledge, representation, reasoning and planning.

- PEAS properties of Wumpus World problem:

1) Performance Measure

⇒ -200 if the player is killed.
+100 for grabbing the gold and coming back to the starting position

2) Environment

⇒ Empty rooms
Rooms with Wumpus

3) Sensors

⇒ Camera to get the view
odour sensor to smell the stench

4) Actuators

⇒ Move forward
Turn right
Turn Left

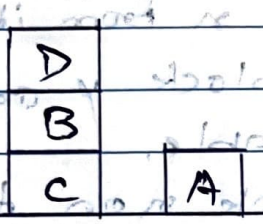
- Conclusion: In this experiment, we learnt about the Wumpus world.

Experiment No: 7

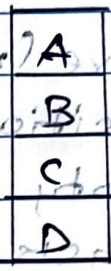
Aim: Planning for blocks world problem

Theory:

Initial state



goal state



A on Table
B on C
C on Table

A on B
B on C
C on D
D on Table

- The block world problem is one of the most famous planning domains in artificial intelligent.
- The goal state is to built one or more vertical stacks of blocks. i.e. turn the initial state into the goal state.
- Only one block may be moved at a time, it may be placed either on the table or on top of another block.
- A block may not be moved if there is another block on top of it.
- The world represented as a set of blocks and their positions. Each block has a unique identifier, and the state is defined by the arrangement of blocks on the table or on other blocks.

- The goal state specifies the desired arrangement of blocks.
- The following actions are performed :-

1) Move(x,y) : Move block x from its current position to the top of block y or an empty location on the table.
2) Stack (x,y) : Place block x on top of y.
3) Unstack (x,y) : Remove block x from the top of block y.

- The block world problem helps illustrate concepts of planning and problem-solving and is often used as a starting point for discussing the computational complexity of planning tasks.

Conclusion

⇒ Understanding and solving the block world problem provides insights into the challenges and strategies involved in automated planning, which is crucial in various fields.

Experiment No : 8

Aim : Implementing Family Tree using prology.

Theory :

- Implementing a family tree in Prolog involves defining facts and rules to represent relationships between family members.
- Prolog operates using a declarative paradigm, where you specify what relationships exist rather than how to find them.
- **Facts :**
  Facts represent the basic relationships in the family tree. These are the atomic pieces of information that define who is related to whom.
- For e.g. :

```
parent(bob, alice).
parent(alice, charlie).
```

These facts state that Bob is the parent of Alice, and Alice is the parent of Charlie.

- **Rules :**
  Rules define relationship based on existing facts. They allow us to infer additional relationships from the ones we have explicitly stated.
- For e.g. :

```
father(Father, child) :-
    parent(Father, child),
    male(Father).
```

- This rule states that someone is considered a father if they are a parent of the child and if they are male.
- This allows Prolog to infer fathers based on existing parent relationships and the gender of individuals.

- Queries

Queries are used to ask questions about the relationships defined in the family tree.

- For e.g.,

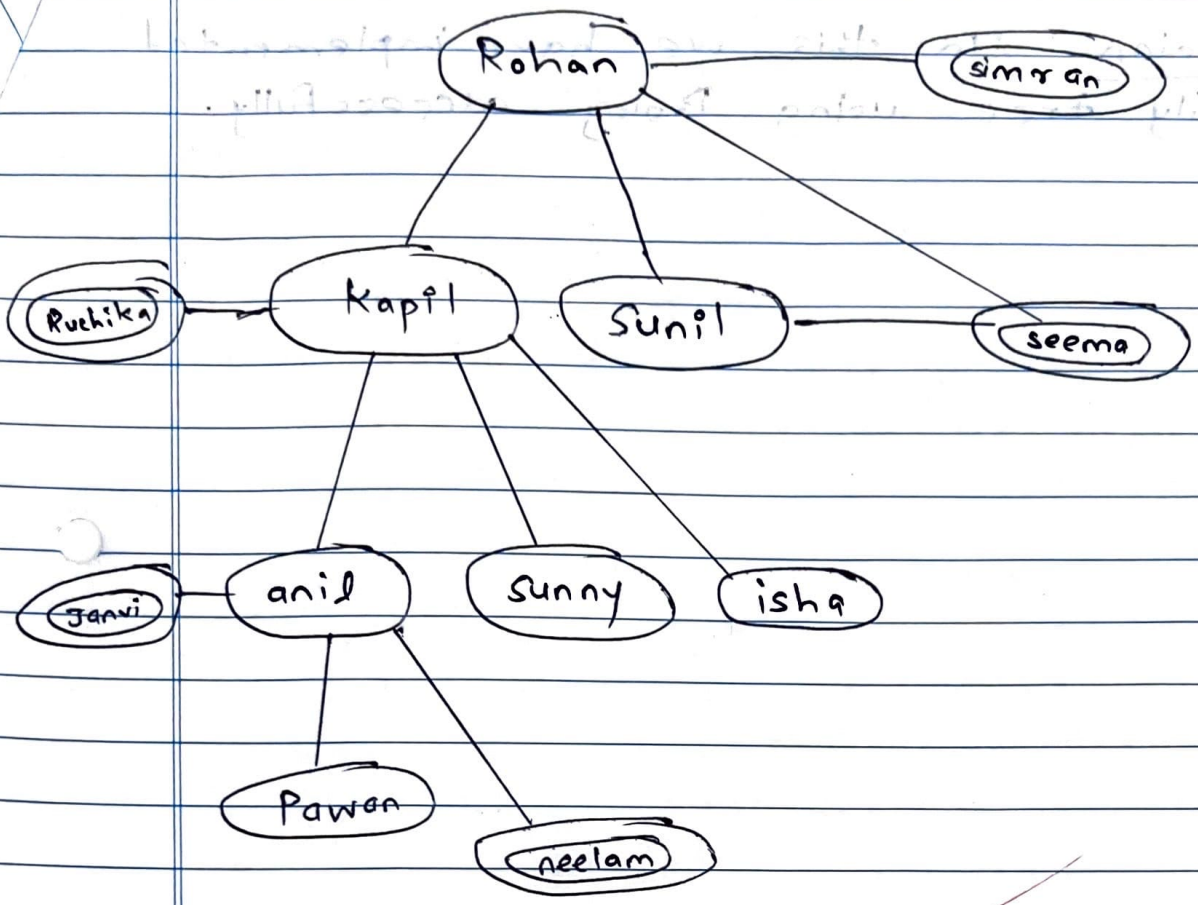
```
?- father(father, emma).
```

This query asks Prolog to find the father of Emma.

- Prolog will then search its database of facts and rules to find a matching answer.

- For e.g.,

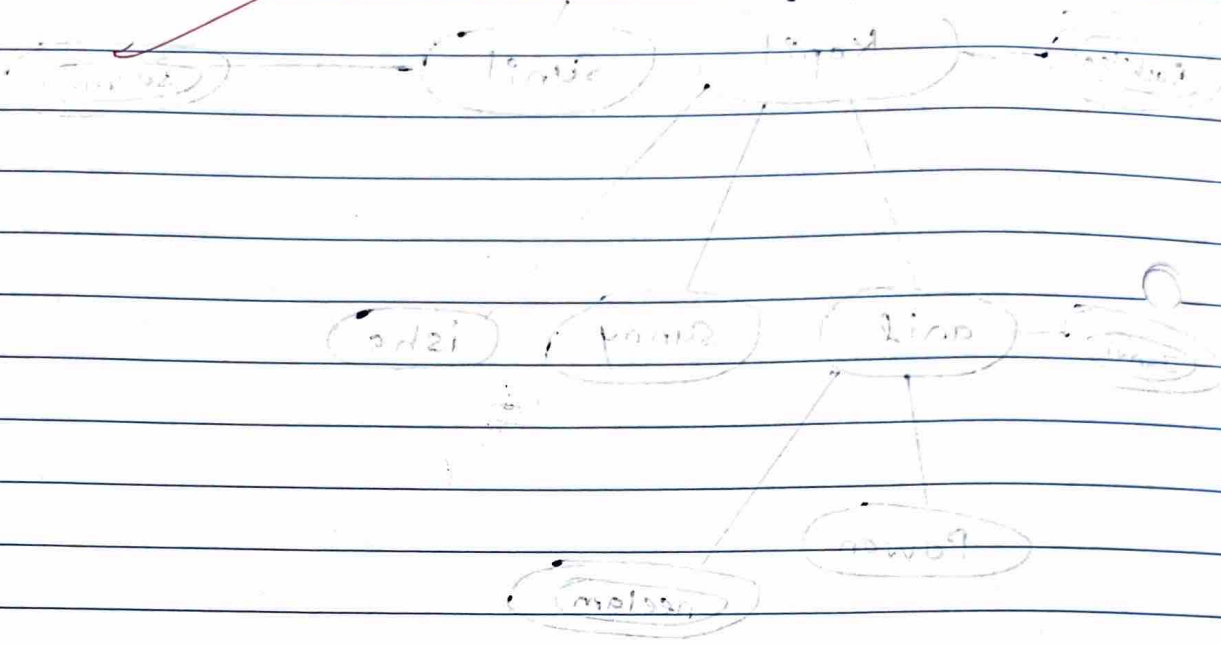
```
?- female(X).
?- mother(X, mike).
?- mother(pat, Y).
?- father(X, Y).
```



- Dataset :

father (Rohan, Kapil)
father (Rohan, sunil)
father (Rohan, seema)
father (Kapil, anil)
father (Kapil, sunny)
father (Kapil, isha)
father (anil, neelam)
wife (simran, rohan)
wife (seema, sunil)
wife (ruchika, Kapil)
wife (janvi, anil)

- Conclusion : In this we have implemented Family tree using Prolog successfully.



Code:

```
% Facts: Define the relationships in the family tree
parent(john, bob).
parent(john, lisa).
parent(mary, bob).
parent(mary, lisa).
parent(bob, tom).
parent(bob, ann).
parent(lisa, patrick).
parent(lisa, emily).

% Rules: Define other relationships based on the facts
father(Father, Child) :-
    parent(Father, Child),
    male(Father).

mother(Mother, Child) :-
    parent(Mother, Child),
    female(Mother).

% Define the gender
male(john).
male(bob).
male(tom).
male(patrick).

female(mary).
female(lisa).
female(ann).
female(emily).

% Define siblings
siblings(X, Y) :-
    parent(Z, X),
    parent(Z, Y),
    X \= Y.

% Define grandparents
grandparent(Grandparent, Grandchild) :-
    parent(Grandparent, Parent),
    parent(Parent, Grandchild).
```

Output:

father(john, Child).

Child = bob
Child = lisa

```
?- father(john, Child).
```

siblings(bob, lisa).

true 1

```
?- siblings(bob, lisa).
```

grandparent(Grandparent, patrick).

Grandparent = john
Grandparent = mary

```
?- grandparent(Grandparent, patrick).
```

male(Person).

Person = john
Person = bob
Person = tom
Person = patrick

```
?- male(Person).
```

siblings(X, Y).

X = bob,
Y = lisa
X = lisa,
Y = bob
X = bob,
Y = lisa
X = lisa,
Y = bob
X = tom,
Y = ann
X = ann,
Y = tom
X = patrick,
Y = emily
X = emily,
Y = patrick

```
?- siblings(X, Y).
```

Om Shete
THADOMAL SHAHANI TE C32
TSEC 2103163
ENGINEERING COLLEGE 23/01/24

Assignment No: I

Q.1 Give one definition on AI for each of the following approaches :

i) Acting Humanly

⇒ The study of how to make computers do things at which, at the moment, people are better.

ii) Thinking Humanly

⇒ The exciting new effort to make computers think ... machines with minds, in the full and literal sense.

iii) Acting Rationally

⇒ Computation Intelligence is the study of the design of intelligent agents.

iv) Thinking Rationally

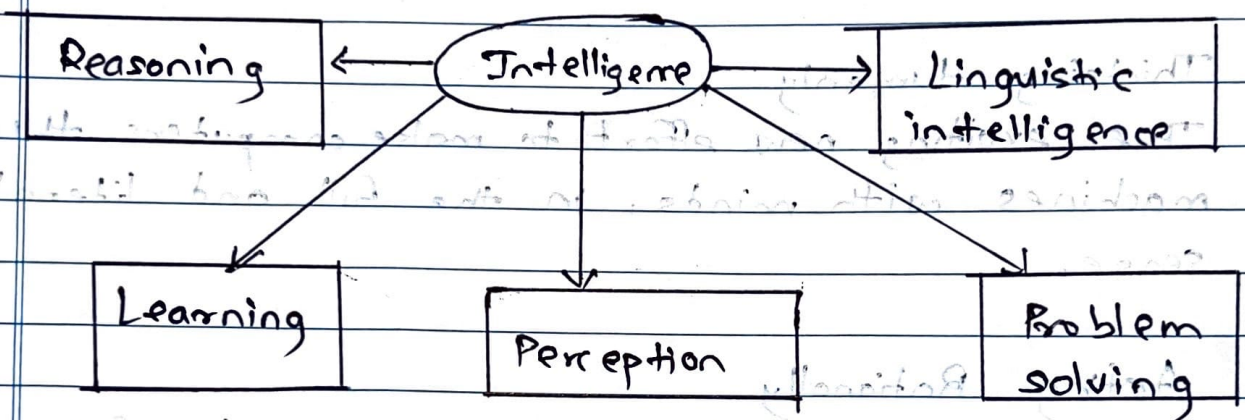
⇒ The study of mental faculties through the use of computational models.

Q.2 Explain the components of AI system in detail.

⇒

- AI is a vast field for research and it has got applications in almost all possible domains.
- By keeping this in mind, components of AI can be identified as follows :
  1) Perception
  2) Knowledge representation
  3) Learning

4) Reasoning
5) Problem Solving
6) Natural ~~Proce~~ Language Processing (NLP)



1) Perception

⇒ In order to work in the environment, intelligent agents need to scan the environment and the various objects in it.

- Agents scans the environment using various ~~scene~~ sense organs like camera, temperature sensor, etc.

2) Knowledge representation

⇒ The information obtained from various environment through sensors may not be in the format required by the system.

- Hence, it need to be represented in standard formats for further processing like learning various patterns, deducing interference, etc.

3) Learning

⇒ Learning is a very essential part and it happens in various forms. The simplest form of learning is by trial and error.

- In this form the program remembers the action that has given desired output and discards the other trial actions and learnes by itself.
- It is also called as unsupervised learning.

4) Reasoning

⇒ Reasoning is also called as logic or generating interference form the given set of facts.

- Reasoning is carried out based on strict rule of validity to perform a specified task.

5) Problem-solving

⇒ AI addresses huge variety of problems. For example, finding out winning moves on the board games, planning actions in order to achieve the defined task, identifying various objects from given images, etc.

6) NLP

⇒ Natural Language Processing, involves machine or robots to understand and process the language that human speak, and infer knowledge from the speech input.

Q.3 Write short note on categorization of AI.

⇒ • Based on capabilities

1) Weak AI

⇒ Weak AI is AI that specilizes in one area. It is not a general purpose intelligence.

- An Intelligent agent is built to solve a particular problem or to perform a specific task is termed as narrow intelligence or weak AI.

2) Strong AI

⇒ Strong AI or general AI refers to an intelligence demonstrated by machines in performing any intellectual task that human can perform.

- Developing strong AI is much harder than developing weak AI.

3) Artificial Super Intelligence

⇒ As defined by leading AI thinker Nick Bostrom, "Super intelligence is an intellect that is much smarter than the best human brains in practicdly every field, including scientific creativity, general wisdom and social skills.

- Artificial super intelligence is the ultimate power of AI.

* Based on functionalities

1) Reactive AI

⇒ Reactive AI systems operate based on predefined rules and respond to specific inputs with predetermined outputs.

- Chess-playing programs that follow fixed algorithms are examples of reactive AI.

2) Limited Memory AI

⇒ Limited Memory AI systems, also known as narrow AI, with memory, can learn and make decisions based on historical data.

- Self-driving cars, which learn from real-world driving scenarios, exemplify limited memory AI.

3) Theory of Mind Machines

⇒ Theory of Mind refers to the ability to understand and attribute mental states, such as beliefs and intentions, to oneself and others.

- Theory of Mind Machines aim to develop AI systems capable of understanding and predicting human emotions, thoughts and intentions.

4) Self-Aware AI

⇒ Self-Aware AI represents the highest level of artificial intelligence, where machines possess consciousness and self-awareness.

Q.4 Explain problem formulation with the help of example.

⇒ • Problems can be defined formally using five components as follows:

1) Initial state
2) Actions
3) Successor function
4) Goal state
5) Path cost

1) Initial state

⇒ The initial state is the one in which the agents starts in.

2) Actions

⇒ It is the set of actions that can be executed or applicable in all possible states. A description of what each action does; the formal name for this is the transition model.

3) Successor function

⇒ It is a function that returns a state on executing an action on the current state.

4) Goal test

⇒ It is a test to determine whether the current state is goal state. In some problems the goal state can be carried out just by compairing current state with the defined

goal state, called as explicit goal state.
- Whereas, in some of the problems, state cannot be defined explicitly but needs to be generated by carrying out some computations, it is called implicit goal test.
- For e.g.: In Tic-Tac-Toe game making diagonal or vertical or horizontal combination declares the winning state which can be compared explicitly; but in the case of chess game, the goal state cannot be predefined but it's a scenerio called as "Checkmate", which has to be ~~taken~~ evaluated implicitly.

5) Path cost

⇒ It is simply the cost associated with each step to be taken to reach goal state. To determine the cost to reach to each state, there is a cost function, which is chosen by the problem solving agent.

- A general solution sequence followed by a simple problem solving agent is, first it formulates the problem with the goal to be acheived, then it searches for a sequence of actions that would solve the problem, and then executes the actions one at a time.

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Q.5 Explain PEAS properties in detail.

⇒

- PEAS stands for Performance Measure, Environment, Actuators, and Sensors.
- It is the short form used for performance issues grouped under Task Environment.

1) Performance Measure

⇒ It is the objective function to judge the performance of the agent.

- For example, in case of pick and place robot, number of correct parts in a bin can be the performance measure

2) Environment

⇒ It is the real environment where the agent need to deliberate actions.

3) Actuators

⇒ These are the tools, equipments or organs using which agent performs actions in the environment. This works as the output of the agent.

4) Sensors

⇒ These are the tools, equipments or organs using which agent captures the state of the environment. This works as the input of the agent.

For e.g.)
Automated Car driving agent

1) Performance

⇒

a) safety : Automated system should be able to drive the car safely without dashing anywhere.

b) Optimum speed : Automated system should be able to maintain the optimal speed depending upon the surroundings.

2) Environment

⇒

a) Roads

⇒ Automated car driver should be able to drive on any kind of a road ranging from city roads to highway.

3) Actuators

⇒

a) steering wheel : which can be used to direct car in desired direction.

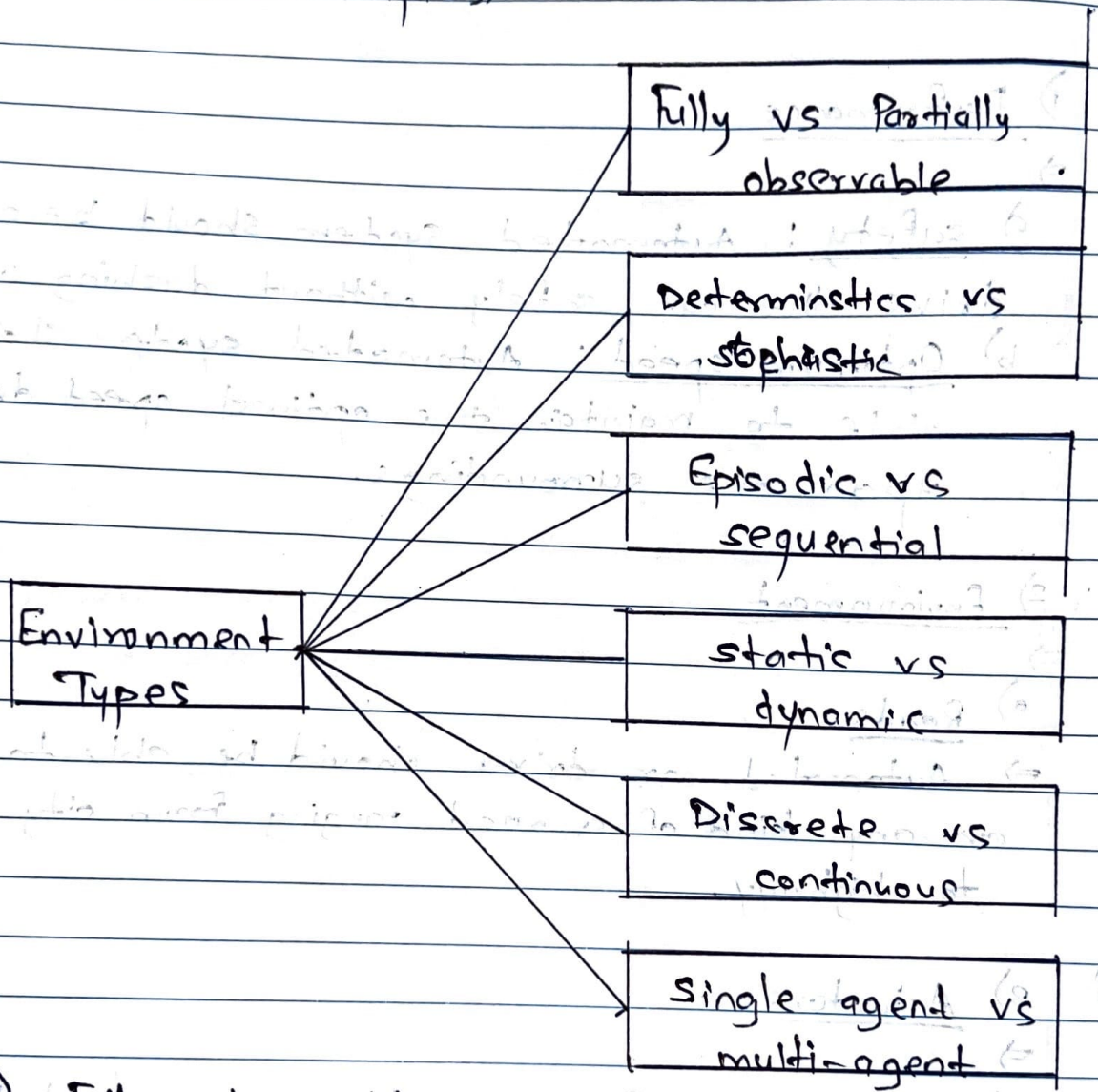
b) Brake : is used to stop the car.

4) Sensors

⇒ To take input from environment in car driving examples cameras, sonar system, GPS, engine sensors, etc. are used as sensors.

Q.6 Describe different types of environments with suitable examples.

⇒



1) Fully observable vs Partially Observable

⇒ In fully observable environments agents are able to gather all the necessary information required to take actions.

- Also in case of fully observable environments agents don't have to keep records of internal states.

- Environments are called partially observable when sensors cannot provide errorless information at any given time for every internal states, as the environment is not seen completely at any point of time.

2) Single agent vs multi-agent

⇒ The second type of an environment is based on the number of agents acting in the environment.

- Whethere the agent is operating on its own or in collaboration with other agents decides if it is a Single agent or a multi-agent environment.

3) Deterministic vs stochastic

⇒ An environment is called deterministic ~~vs stochastic~~ environment, when the next state of the environment can be completely determined by the previous state and the action executed by the agent.

- Stochastic environment generally means that the indecision about the actions is enumerated in terms of probabilities.
- That means environment changes while agent is taking actions, hence the next state of the world does not merely depends on the current state and agent's action.

4) Episodic vs Sequential

⇒ An episodic task environment is the one where each of the agent's action is divided into an atomic incidents or episodes.

- the current incident is different than the previous incident and there is no dependency between the current and the previous incident.
- In sequential environments, as per the name suggests, the previous decision can affect all future decisions.

5) Static vs dynamic

⇒ Dynamic environment changes over time, which static environments remain constants. dynamic environment could be traffic system while static environment might be a fixed puzzle.

Assignment No: 2

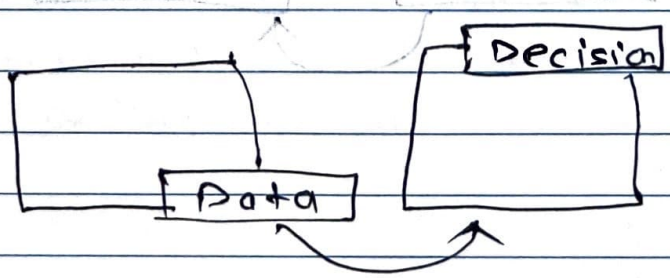
Q.1 Discuss the forward and backward chainning algorithm. Illustrate the working of forward and backward chainning for following problem.

⇒

Forward Chaining

⇒ For any type of inference there should be a path from start to goal. When based on the available data a decision is taken then the process is called as the forward chaining.

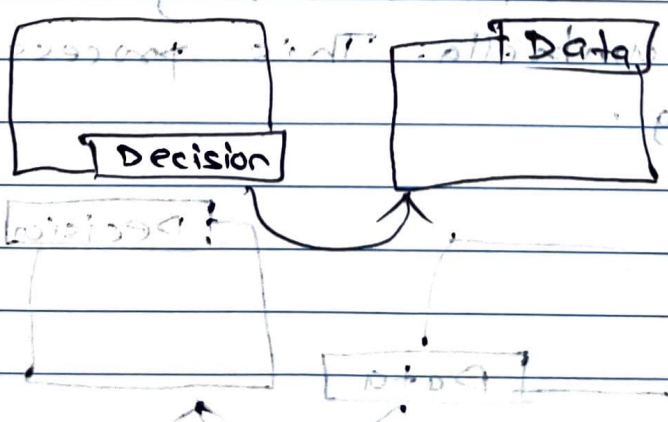
- Forward chaining or data driven inference works from an initial state, and by looking at the premises of the rules, performs the action, possibly updating the knowledge base or working memory.
- This continues until the no more rules can be applied or some loyale limit is met.
- for example; If it is raining then, we will take umbrella.
- Here, "it is raining" is the data and "we will take umbrella" is a decision.
- This means it was already known that it's raining that's why it was decided to take umbrella. This process is forward chaining.



- Backward chaining

⇒ If based on the decision the initial data is fetched, then it is called as backward chaining.

- Backward chaining or goal driven inference works towards a final state, and by looking at the working memory to see if goal already there.
- If not look at the actions of rules that will establish goal and set up sub-goals for achieving premises of rules. This continues until some rules can be applied, apply to achieve goal state.
- For e.g., If while going out one has taken umbrella. Then based on the decision it can be guessed that it is raining.
- Here, "taking umbrella" is a decision based on the data is generated that "it is raining".
- This process is backward chaining



• Illustration for given problem:

- Given Problem:
The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles and all of its missiles were sold to it by Colonel West, who is American.

- Facts:
1) Nono is an enemy of America.
2) Colonel west is American

- Rules:
1) selling weapons to hostile nations is crime for Americans.
2) All of Nono's missiles were sold by Colonel West

- Forward Chaining:
1) Start with known facts.
2) Apply rule 2 : Colonel West sold missiles to Nono.
3) Apply rule 1 : Selling weapons to hostile nations is crime for Americans.
4) Conclusion : Colonel west committed a crime by selling missiles to Nono.

- Backward Chaining :

⇒ 1) Start with the goal : Colonel West committed a crime.

2) Check if there is rule that concludes Colonel West committed a crime. Rule 1 applies.

3) Check if the condition for Rule 1 satisfied.

4) To prove Colonel west sold weapons to Nono check if there's a rule or fact supporting this. Rule 2 applies.

5) Check if the condition for Rule 2 is satisfied

6) Conclusion: Colonel West committed crime by selling missiles to Nono.

Q.2 Consider following examples and prove using Resolution Curiosity killed Cat

- Everyone who loves all animals is loved by someone
- Anyone who kills an animal is loved by no one
- Jack loves all animals
- Either Jack or Curiosity killed the cat, who is named tuna
- Did curiosity kill the cat ?

⇒

Step 1 : Negate the statement to be proved

⇒ ~Kills (curiosity, tuna)

Step 2 : Converting given statement to FOL

a) $\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow \exists y \text{ Loves}(x, y)$

b) $\forall x [\exists z \text{ Animal}(z) \wedge \text{kills}(x, z)] \Rightarrow \forall y \sim \text{Loves}(y, x)$

c) $\forall x \text{ Animal}(x) \Rightarrow \text{Loves}(\text{Jack}, x)$

d) $\text{Kills}(\text{Jack}, \text{tuna}) \vee \text{Kills}(\text{curiosity}, \text{tuna})$

e) $\text{Cat}(\text{tuna})$

F) $\sim \text{kills}(\text{curiosity}, \text{tuna})$

Step 3: Converting FOLs to CNF

a) $\text{Animal}(F(x)) \vee \text{Loves}(g(x), x)$
$\sim \text{Loves}(x, F(x)) \vee \text{Loves}(g(x), x)$

b) $\sim \text{Loves}(y, x) \vee \sim \text{Animal}(z) \vee \sim \text{kills}(x, z)$

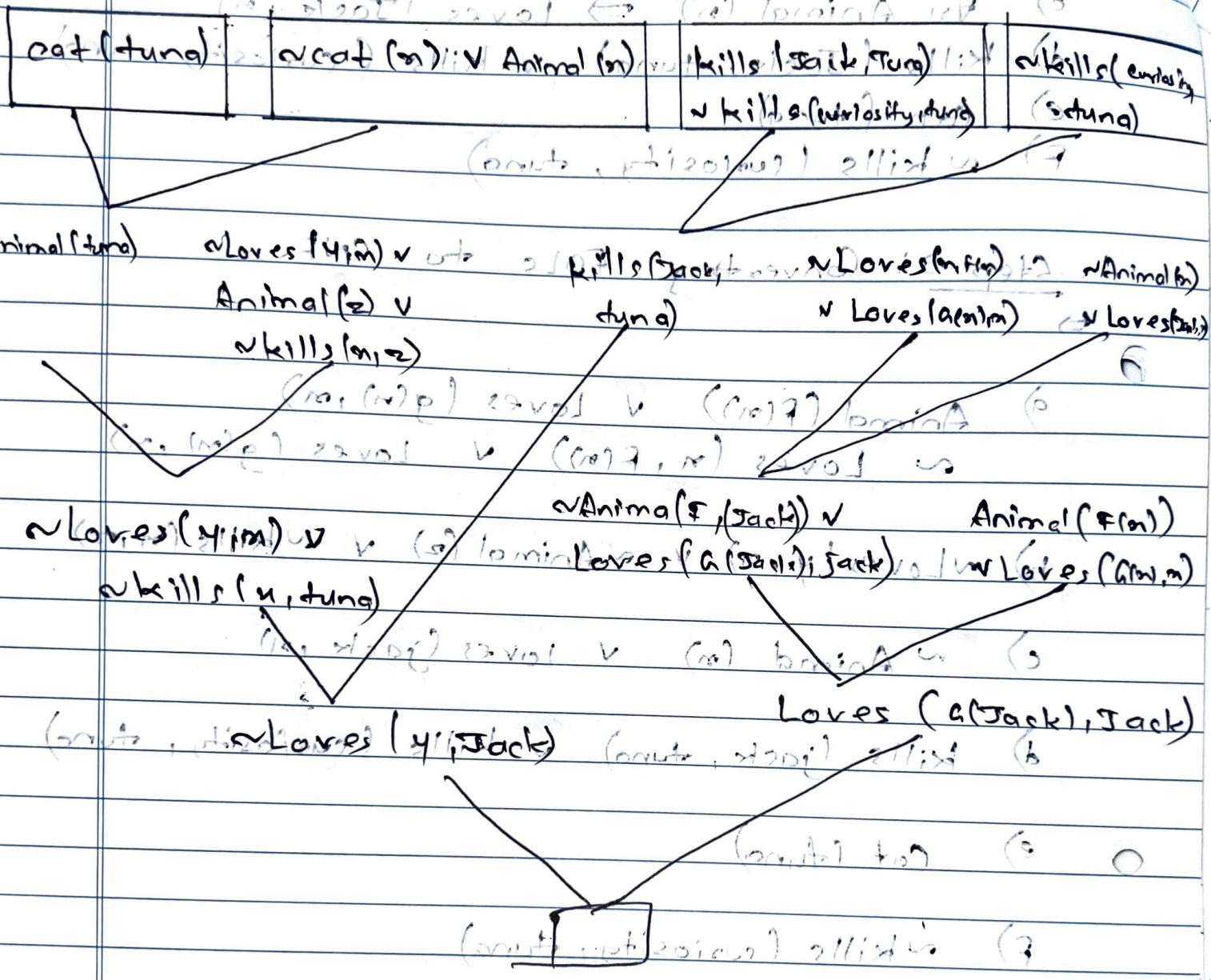
c) $\sim \text{Anind}(x) \vee \text{Loves}(\text{jack}, x)$

d) $\text{kills}(\text{jack}, \text{tuna}) \vee \text{kills}(\text{curiosity}, \text{tuna})$

e) $\text{Cat}(\text{tuna})$

f) $\sim \text{kills}(\text{curiosity}, \text{tuna})$

Step 4: Proof by resolution


cat(tuna)
∀x cat(x) : V Animal(x)
kills(Jack,Tuna) :
∀x kills(curiosity,tuna)
∀x kills(curiosity, tuna)
Animal(tuna)
∀x Loves(y,x) ∨
Animal(z) ∨
∀x kills(x,z)
kills(Jack, tuna)
∀x Loves(x,Fx) ∨
∀x Animal(x)
∀x Loves(G(x),x)
∀x Loves(x,x)
∀Animal(F(Jack)) ∨
∀x Loves(G(x),Jack)
Animal(F(x))
∀x Loves(G(x),x)
∀x Loves(y,x) ∨
∀x kills(y,tuna)
∀x Loves(y,Jack)
Loves(G(Jack),Jack)

Q.3 Discuss in detail NLP elaborating on the following point :

1) What is NLP ?

⇒ As the name suggests, Natural language Processing, involves machines or robots to understand and process the language that human speak, and infer knowledge from the speech input.

- It also involves the active participation from machine in the form of dialog i.e NLP aims at the text or verbal output from the machine or robot.
- The input and output of an NLP system can be speech and written text respectively.

2) Components of NLP

⇒ Mainly there are two components of NLP :

a) Natural Language Understanding (NLU)

⇒ In this part of the processing the speech input gets transformed into the useful representation in order to analyze various aspects of the language.

- As the natural language is very rich in forms and structures, it is also very ambiguous.
- There can be different forms of ambiguities like lexical ambiguity, which is a very basic i.e word level ambiguity.

b) Natural Language Generation (NLG)

⇒ In order to generate the output text, the intermediate representation requires to be converted back to the natural language format.

Hence, in this process there are multiple sub process involves.

They are as follows

i) Text planning
ii) Sentence planning
iii) Text Realization

3) Difficulties in NLP

⇒

a) Ambiguity : Natural language is inherently ambiguous, with words and phrases often having multiple meanings depending on context.

b) Variability : Language usages varies greatly across different regions, dialects and social groups, making it challenging to create generalized models.

c) Lack of context

⇒ Understanding the context in which words or phrases are used is crucial for accurate interpretation

4) Steps involved in NLP

⇒ Lexical analysis
↓
Syntactic Analysis
↓
Semantic Analysis
↓
Disclosure integration
↓
Pragmatic analysis

1) Lexical Analysis

⇒ Lexicon is the words and phrases in language. Lexicon analysis deals with the recognition and identification of structure of the sentences.

2) Syntactic Analysis

⇒ In syntactic analysis the sentences are parsed as noun, verbs, adjective and other parts of sentences.

3) Semantic Analysis

⇒ In this phase the actual meaning of the sentence is extracted from the structure and the words used.

4) Discourse Integration

⇒ The meaning of discourse with respect to NLP is nothing but the context of the sentence or a word.

5) Pragmatic Analysis

⇒ Pragmatic deals with meaning of the sentences in various situations. In this, the sentences are reinterpreted to verify the correctness of the meaning in their given context.

5) Role of NLP in AI

⇒

a) Language translation

⇒ NLP powers machine translation system that can translate text between different languages, breaking down languages barriers and facilating languages on a global scale.

b) Information retrieval

⇒ NLP techniques are used to extract relevant information from large volumes of text, enabling systems to search, index and organize textual data effectively.

Examples of NLP

1) Voice Assistants
2) Chatbots
3) Autocomplete tools
4) Language translation
5) Sentiment analysis
6) Text extraction
7) Content moderation

Q.4 What is robotics? Discuss the role of AI in robotics. Brief the applications of robotics in health care and agriculture.

⇒ - Robotics is a multidisciplinary field that involves the design, construction, operation and use of robots.
- A robot is a machine that is capable of carrying out tasks automatically, typically with some degree of autonomy.
- Roles of AI in Robotics :-

1) Perception

⇒ AI algorithm enables robot to perceive and interpret this environment using sensors such as camera, lidar, radar, etc.

2) Planning and Decision making

⇒ AI algorithms empower robots to plan and execute tasks autonomously by analyzing sensor data, generating trajectories, and making decision in real-time.

3) Learning and Adaption

⇒ AI techniques such as machine learning and reinforcement learning enables robots to learn from experience and improve their performance over time.

- Application of Robotics :-

1) Healthcare

⇒ When we talk about human lives and health, any technologies that can give more efficient, helpful and faster analysis to hand out a proper treatment plan in time are tremendously valuable

- Artificial intelligence and its subdivision ML is taking over the world right now.
- One of the best ML of healthcare application is a bot system that makes the treatment period much easier.
- There is huge application of ML in healthcare algorithms in the fields of oncology, pathology and Rare diseases.

2) Agriculture

⇒ Robotics is transforming agriculture with the development of autonomous vehicles such as drones and robotics tractor.

- These vehicles can perform tasks such as planting, spraying pesticides and harvesting crops with precision and efficiency, reducing labor costs and increasing productivity.