Madan Bhandari Memorial College

Lab No. 1

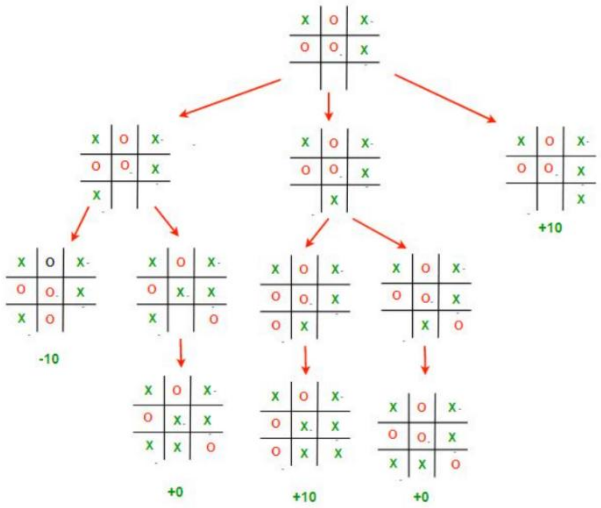
Title: Write a program for tic tac toe game in python.

Tic Tac Toe is a classic two-player strategy game where players alternately mark spaces in a 3×3 grid with their respective symbols (usually X and O). The goal is to be the first to place three marks in a horizontal, vertical, or diagonal row.

In Artificial Intelligence, Tic Tac Toe serves as an introductory example of game theory and search algorithms such as Minimax, demonstrating how an AI can make optimal decisions in a finite game space. This lab implements a simple console-based Tic Tac Toe game in Python. The game is be played between human and computer opponent.

Algorithm

1. Initialize Board: Create a 3×3 matrix to represent the game board.
2. Display Board: Show the current board state after each move.
3. Player Input: Take input from the current player for their move position.
4. Validate Move: Check if the chosen position is empty. If not, prompt again.
5. Update Board: Place the player's symbol in the chosen position.
6. Check Win Condition: After each move, check if the current player has won.
7. Check Draw: If the board is full and no one has won, declare a draw.
8. Switch Player: Alternate between Player X and Player O.
9. Repeat until a win or draw occurs.



This image depicts all the possible paths that the game can take from the root board state. It is often called the Game Tree.

The 3 possible scenarios in the above example are :

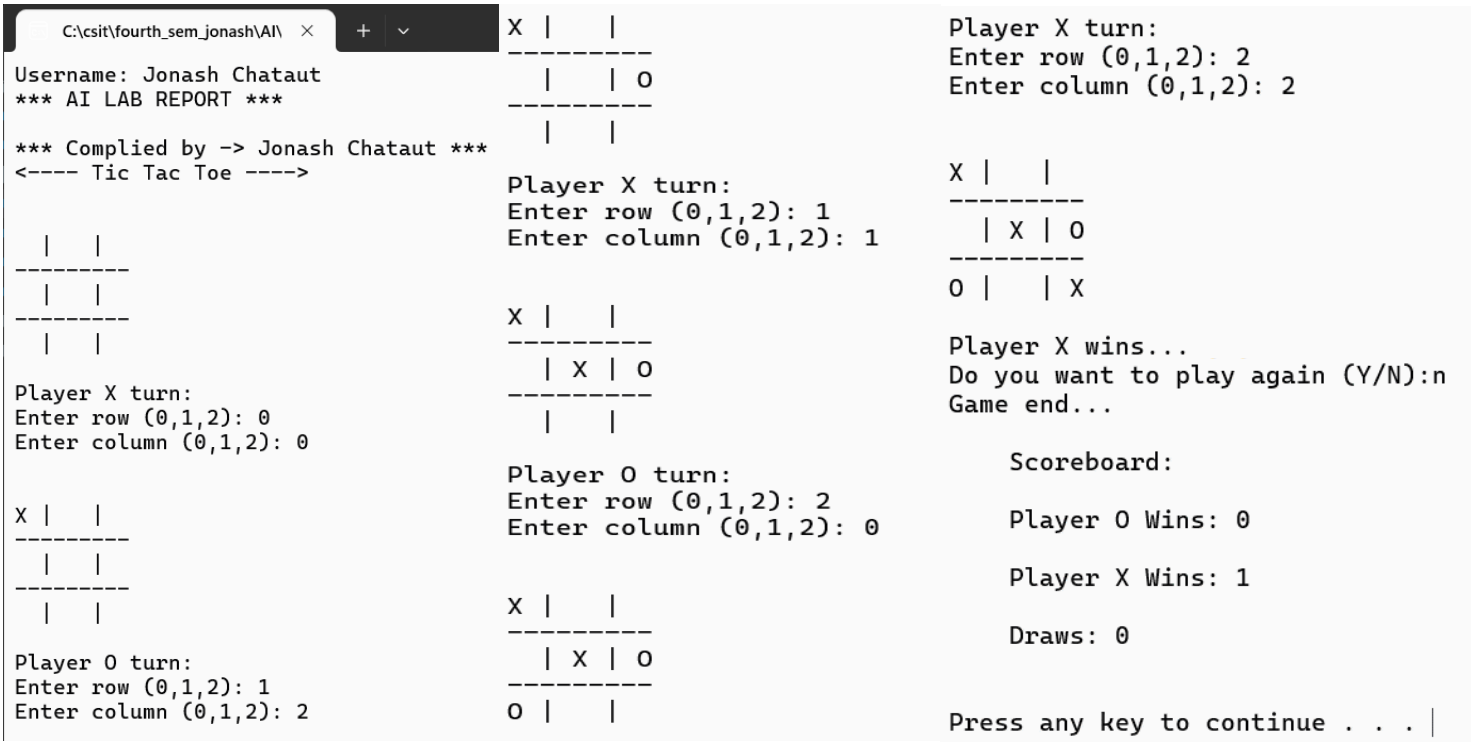
- Left Move : If X plays [2,0]. Then O will play [2,1] and win the game. The value of this move is -10
- Middle Move : If X plays [2,1]. Then O will play [2,2] which draws the game. The value of this move is 0
- Right Move : If X plays [2,2]. Then he will win the game. The value of this move is +10;

Source Code:

```
import random
print("*** Complied by -> Jonash Chataut ***")
print("<---- Tic Tac Toe ---->")
def print_board(board):
    print("\n")
    for i,row in enumerate(board):
        print(" | ".join(row))
        if i<len(board)-1:
            print("-"*9)
def check_winner(board,player):
# checking rows, columns and diagonals
    for i in range(3):
        if all(board[i][j]==player for j in range(3)): #rows
            return True
        if all(board[j][i]==player for j in range(3)): #columns
            return True
    if all(board[i][i]==player for i in range(3)): #diagonals
        return True
    if all(board[i][2-i]==player for i in range(3)): #anti-diagonals
        return True
    return False
def is_board_full(board):
    return all(cell !=" " for row in board for cell in row) #returns true only when each ierations are true
def display_scores(scores):
    print(f"""
    Scoreboard:\n
    Player O Wins: {scores['O']}\n
    Player X Wins: {scores['X']}\n
    Draws: {scores["Draw"]}
    """)
def tic_tac_toc_run():
    #Tracking score
    score = {'X':0, 'O':0, "Draw":0}
    while True:
        board = [[" " for _ in range(3)] for _ in range(3)]
        current_player = random.choice(["X","O"])
        while True:
            print_board(board)
            print(f"\nPlayer {current_player} turn:")
            try:
                row = int(input("Enter row (0,1,2): "))
                col = int(input("Enter column (0,1,2): "))
                if row  not in range(3) or col  not in range(3):
                    print("Invalid input.../n Please enter 0, 1 or 2.")
                    continue
                if board[row][col]!=" ":
                    print("This cell is already taken try another cell....")
                    continue
                board[row][col] = current_player
                if check_winner(board,current_player):
```

```
            print_board(board)
            print(f"\nPlayer {current_player} wins... ")
            score[current_player]+=1
            break
          if is_board_full(board):
            print_board(board)
            print("Game draw....")
            break
          current_player = "O" if current_player=="X" else "X"
        except ValueError:
          print("Invalid input.../n Enter a number")
      play_again = input("Do you want to play again (Y/N):")
      if play_again.lower() == 'n':
        print("Game end...")
        display_scores(score)
        break
if __name__ == "__main__":
  tic_tac_toc_run()
```

Output:



Madan Bhandari Memorial College

Lab No. 2

Title: Write a program to implement water jug problem in python.

The Water Jug Problem is a classic puzzle in Artificial Intelligence and problem-solving. It involves two jugs with different capacities and the goal of measuring an exact target quantity of water. It's often used to illustrate state space search techniques such as Breadth-First Search (BFS).

Key Concepts:

- State Space: Each state is represented as a pair (x, y) where x and y are the amounts of water in Jug 1 and Jug 2 respectively.
- Goal State: A state where either jug contains the target amount.
- Operators: Filling a jug, emptying a jug, or pouring from one jug to another.
- Search Algorithm: BFS explores all possible states level-by-level until the goal is reached.

Algorithm

1. Initialize:
    - Take capacities of Jug 1, Jug 2, and target amount as input.
    - Create an empty visited set and a BFS queue.
2. Start BFS:
    - Begin from the initial state (0, 0).
3. Goal Test:
    - If the current state meets the target in either jug, return the solution path.
4. Generate Successor States:
    - Fill Jug 1.
    - Fill Jug 2.
    - Empty Jug 1.
    - Empty Jug 2.
    - Pour Jug 1 into Jug 2.
    - Pour Jug 2 into Jug 1.
5. Enqueue Valid States:
    - Add new states to the queue if not visited.
6. Repeat until the queue is empty or the goal is found.
7. Output:
    - Display the sequence of steps to reach the target amount.

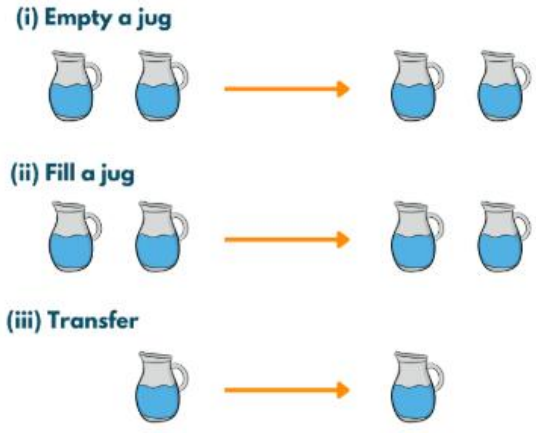
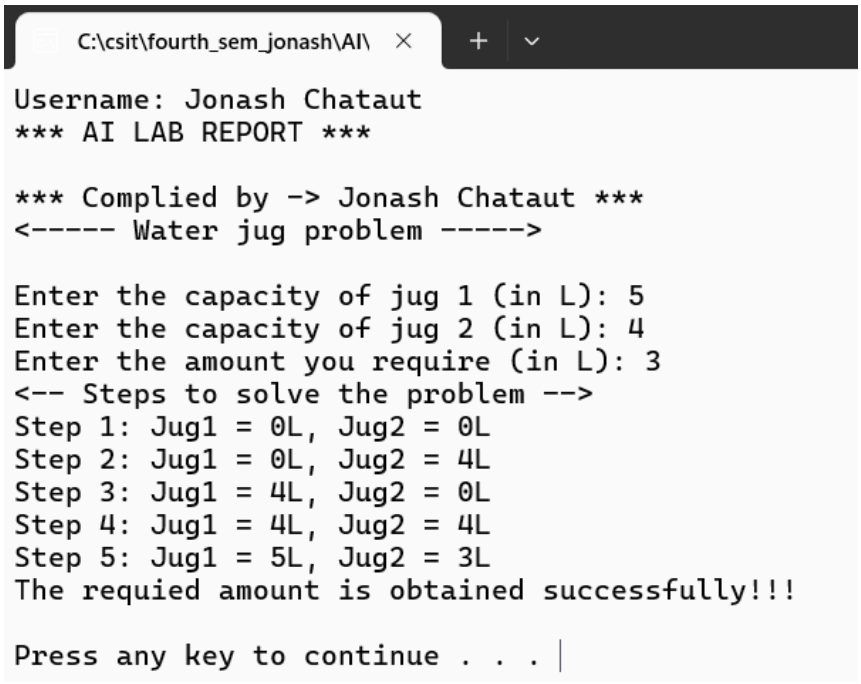


Figure: Basic Operations in The Water Jug Problem

Source Code:

```
from collections import deque
# Checking if the state is in visited set or not
def is_valid(state, visited):
    return state not in visited #return true if not in visited set
def water_jug_bfs(jug1, jug2, target):
    visited = set()
    queue = deque() #create a empty double ended queue
    # Each element: (current state, path to reach this state)
    queue.append(((0, 0), []))
    while queue:
        (a, b), path = queue.popleft()
        # Skip already visited states
        if (a, b) in visited:
            continue
        visited.add((a, b))
        # Add current state to path
        path = path + [(a, b)]
        # Check if target is reached
        if a == target or b == target:
            return path
        # Generate next possible states
        next_states = [
            (jug1, b),      # Fill Jug1
            (a, jug2),      # Fill Jug2
            (0, b),         # Empty Jug1
            (a, 0),         # Empty Jug2
            (0, a + b) if a + b <= jug2 else (a - (jug2 - b), jug2),  # Jug1 → Jug2
            (a + b, 0) if a + b <= jug1 else (jug1, b - (jug1 - a)),  # Jug2 → Jug1
        ]
        for state in next_states:
            if is_valid(state, visited):
                queue.append((state, path))
    return None  # No solution found
print("*** Complied by -> Jonash Chataut ***")
print("<----- Water jug problem ----->\n")
# Asking inputs from user
jug1_capacity = int(input("Enter the capacity of jug 1 (in L): "))
jug2_capacity = int(input("Enter the capacity of jug 2 (in L): "))
target = int(input("Enter the amount you require (in L): "))
# Calling the function
solution = water_jug_bfs(jug1_capacity, jug2_capacity, target)
if solution:
    print("<-- Steps to solve the problem -->")
    for i, (A_jug_1,A_jug_2) in enumerate(solution): #jug_1 and jug_2 are the amount in jugs stored in tuple in
      each steps
        print(f"Step {i+1}: Jug1 = {A_jug_1}L, Jug2 = {A_jug_2}L")
    print("The requied amount is obtained successfully!!!")
else:
    print("No solution found.")
```

Output:


C:\csit\fourth_sem_jonash\AI\
Username: Jonash Chataut
*** AI LAB REPORT ***
*** Complied by -> Jonash Chataut ***
<----- Water jug problem ----->
Enter the capacity of jug 1 (in L): 5
Enter the capacity of jug 2 (in L): 4
Enter the amount you require (in L): 3
<-- Steps to solve the problem -->
Step 1: Jug1 = 0L, Jug2 = 0L
Step 2: Jug1 = 0L, Jug2 = 4L
Step 3: Jug1 = 4L, Jug2 = 0L
Step 4: Jug1 = 4L, Jug2 = 4L
Step 5: Jug1 = 5L, Jug2 = 3L
The requied amount is obtained successfully!!!
Press any key to continue . . .

Madan Bhandari Memorial College

Lab No. 3

Title: Write a program for simple chatbot in python.

A chatbot is a computer program designed to simulate conversation with human users, especially over the Internet.

There are two main types of chatbots:

1. Rule-based Chatbots – respond to keywords or patterns using if-else rules.
2. AI-based Chatbots – use Natural Language Processing (NLP) and machine learning to understand and respond intelligently.

In this lab, we implement a simple rule-based chatbot. The program responds based on matching keywords from user input to predefined rules. This approach, while simple, demonstrates the basic structure of conversational agents. The chatbot runs in a loop, taking user input until the user types "bye".

Source Code:

```
def simple_chatbot():
  print("*** Complied by -> Jonash Chataut ***")
  print("Hi! I'm ChatBot. Let's talk... /nType 'bye' to exit.\n")

  while True:
    user_input = input("You: ").lower()

    if "hello" in user_input or "hi" in user_input:
      print("Bot: Hello !")
    elif "how are you" in user_input:
      print("Bot: I'm just code, but I feel fantastic! What about you?")
    elif "your name" in user_input:
      print("Bot: I'm your friendly chatbot, ChatBuddy haha")
    elif "thank" in user_input:
      print("Bot: You're welcome!")
    elif "joke" in user_input:
      print("Why do Python programmers wear glasses? Because they can't C!")
    elif "bye" in user_input:
      print("Bot: Goodbye, Have a great day !!")
      break
    else:
      print("Bot: I'm not sure how to respond to that...")

if __name__ == "__main__":
  simple_chatbot()
```

Output:


C:\csit\fourth_sem_jonash\AI\
Username: Jonash Chataut
*** AI LAB REPORT ***
*** Complied by -> Jonash Chataut ***
Hi! I'm ChatBot. Let's talk... /nType 'bye' to exit.
You: Hello
Bot: Hello !
You: Your name
Bot: I'm your friendly chatbot, ChatBuddy haha
You: Joke
Why do Python programmers wear glasses? Because they can't C!
You: Who are you
Bot: I'm not sure how to respond to that...
You: Bye
Bot: Goodbye, Have a great day !!
Press any key to continue . . .

Lab No. 4

Title: Write a program to implement Breadth-First Search.

Breadth-First Search (BFS) is a graph traversal algorithm used to explore vertices and edges of a graph in a systematic way. It works by visiting all neighbours of a node before moving on to the next level of nodes. BFS uses a queue data structure to keep track of nodes to visit next.

Key points about BFS:

- Works for both directed and undirected graphs.
- Guarantees the shortest path in an unweighted graph.
- Uses FIFO (First In First Out) order for traversal.

Algorithm

1. Create Graph:
    - Input number of nodes.
    - Input each node and its connected nodes.
    - Store them in a dictionary (adjacency list).
2. Initialize BFS:
    - Add the starting node to visited list and queue.
3. While Queue Not Empty:
    - Dequeue a node from the front.
    - Print the node.
    - If node equals the goal, stop and print "Goal reached".
    - Otherwise, enqueue all unvisited neighbors.
4. Repeat until goal is found or all reachable nodes are visited.

Source Code:

```
def createGraph(graph):
  print("***** BFS *****")
  n = int(input("Enter the number of nodes in graph:- "))
  print("Enter nodes and connected nodes in format -> node: connectedNode_1, connectedNode_2,...")
  for _ in range(n):
    node = input("Node -> ").split(":")
    graph[node[0]] = node[1].split(",")
  return graph
def bfs(graph, start, goal):  # function for BFS
  visited = []  # List for visited nodes.
  queue = []  # Initialize a queue
  visited.append(start)
  queue.append(start)
  while queue:        # Creating loop to visit each node
    current_Node = queue.pop(0)
    # print '->' after each node except the last one
    print(current_Node, end='->' if current_Node != goal else '')
    if current_Node == goal:
```

```
            print("\nGoal reached!")
            break
        for neighbour in graph[current_Node]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)
# Driver Code
graph = dict()
graph = createGraph(graph)
start = input("Enter the starting point:- ")
goal = input("Enter the goal point:- ")
print("Following is the Breadth-First Search")
bfs(graph,start,goal)
```

Output:

```
C:\csit\fourth_sem_jonash\AI\
Username: Jonash Chataut
*** AI LAB REPORT ***

***** BFS *****
Enter the number of nodes in graph:- 5
Enter nodes and connected nodes in format -> node: connectedNode_1, connectedNode_2,...
Node -> A:B,C
Node -> B:E,G
Node -> C:
Node -> E:
Node -> G:
Enter the starting point:- A
Enter the goal point:- G
Following is the Breadth-First Search
A->B->C->E->G
Goal reached!

Press any key to continue . . .
```

Lab No. 5

Title: Write a program to implement Depth-First Search.

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along one branch before backtracking. It uses the LIFO (Last In, First Out) principle, typically implemented using recursion or an explicit stack.

Key characteristics of DFS:

- Explores deep into a branch before moving to the next.
- Can be used for pathfinding, cycle detection, and topological sorting.
- Does not guarantee the shortest path in an unweighted graph.

Algorithm

1. Create Graph:
    - Input number of nodes.
    - Input each node and its connected nodes.
    - Store in an adjacency list (dictionary).
2. Initialize DFS:
    - Maintain a visited set and a path list.
3. Recursive DFS:
    - Mark the current node as visited and add it to the path.
    - If the current node is the goal, print the path and stop recursion.
    - Recursively visit all unvisited neighbors.
4. Backtrack:
    - Remove the node from the path if the goal was not found in its branch.
5. Output:
    - Display traversal path if the goal is found, otherwise indicate that it's unreachable.

Source Code:

```
def createGraph(graph):
  print("***** DFS *****")
  n = int(input("Enter the number of nodes in graph:- "))
  print("Enter nodes and connected nodes in format -> node: connectedNode_1, connectedNode_2,...")
  for _ in range(n):
    node = input("Node -> ").split(":")
    graph[node[0].strip()] = [n.strip() for n in node[1].split(",") if n.strip()]
  return graph

def dfs(graph, node, goal, visited=None, path=None):
  if visited is None:
    visited = set()
  if path is None:
    path = []
```

```
    visited.add(node)
    path.append(node)

    if node == goal:
        print("->".join(path))
        print("Goal reached!")
        return True  # stop further recursion

    for neighbour in graph.get(node, []): #if node not found then [] given
        if neighbour not in visited:
            if dfs(graph, neighbour, goal, visited, path):
                return True

    path.pop()  # backtrack if goal not found on this path
    return False  # continue searching

# Driver Code
graph = dict()
graph = createGraph(graph)
start = input("Enter the starting point:- ")
goal = input("Enter the goal point:- ")
print("Following is the Depth-First Search")

found = dfs(graph, start, goal)
if not found:
    print("Goal not reachable.")
```

Output:

```
C:\csit\fourth_sem_jonash\AI\
Username: Jonash Chataut
*** AI LAB REPORT ***

***** DFS *****
Enter the number of nodes in graph:- 6
Enter nodes and connected nodes in format -> node: connectedNode_1, connectedNode_2,...
Node -> A:B,C
Node -> B:D,E
Node -> C:
Node -> D:
Node -> E:G
Node -> G:
Enter the starting point:- A
Enter the goal point:- G
Following is the Depth-First Search
A->B->E->G
Goal reached!

Press any key to continue . . .
```

Lab No. 6

Title: Write a program to implement Uniform Cost Search.

Uniform Cost Search (UCS) is a search algorithm used to find the least-cost path from a start node to a goal node. It is a variant of Dijkstra's Algorithm and is an uninformed search technique that explores nodes based on path cost rather than depth or heuristic.

Key characteristics of UCS:

- Uses a priority queue to always expand the node with the lowest path cost.
- Guarantees the optimal solution if all edge costs are non-negative.
- Suitable for weighted graphs where the goal is to minimize cost.

Algorithm

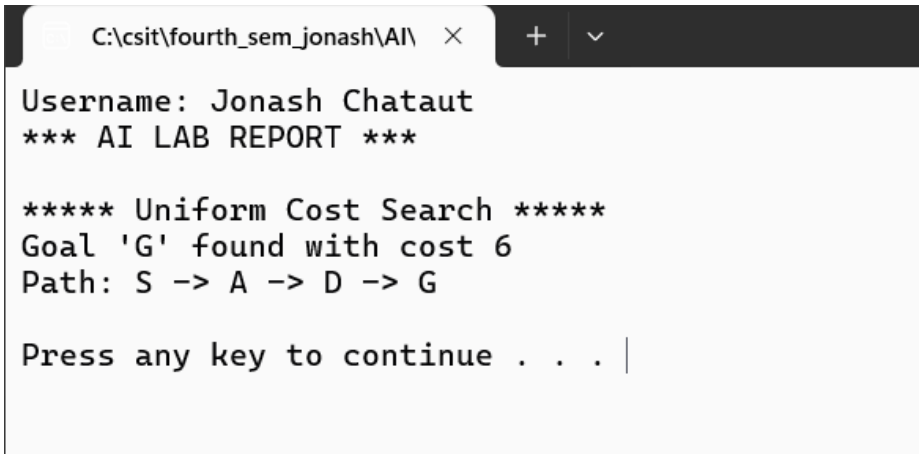
1. Initialize a priority queue with a tuple (cost, node, path) starting from (0, start, [start]).
2. While the queue is not empty:
    - Pop the node with the smallest cost.
    - If it is the goal, print the cost and path, then stop.
    - Mark the node as visited.
    - For each unvisited neighbor, push (new_cost, neighbor, updated_path) into the queue.
3. If the queue empties without finding the goal, report that the goal is unreachable.

Source Code:

```
import heapq #priority queue with heap (min heap)
def uniform_cost_search(graph, start, goal):
  # Priority queue: (cost, node, path)
  pq = [(0, start, [start])]
  visited = set()
  while pq:
    cost, node, path = heapq.heappop(pq) #removes the item with smallest cost
    if node == goal:
      print(f"Goal '{goal}' found with cost {cost}")
      print("Path:", " -> ".join(path))
      return
    if node not in visited:
      visited.add(node)
      for neighbor, weight in graph.get(node, []):
        if neighbor not in visited:
          #push item in heapq must in tuple and first element decides priority i.e cost in our case
          heapq.heappush(pq, (cost + weight, neighbor, path + [neighbor]))
  print(f"Goal '{goal}' not reachable from '{start}'.")
# Graph: adjacency list with (neighbor, cost) using dictionary
graph = {
  'S': [('A', 1), ('B', 4)],
  'A': [('C', 3), ('D', 2)],
  'B': [('G', 3)],
```

```
    'C': [('E',5)],
    'D': [('F', 4),('G',3)],
    'E': [('G',5)],
    'G': []
}

start = 'S'
goal = 'G'
print("***** Uniform Cost Search *****")
uniform_cost_search(graph, start, goal)
```

Output:



Lab No. 7

Title: Write a program to implement Depth-Limited Search.

Depth-Limited Search (DLS) is a variation of Depth-First Search (DFS) in which the search is restricted to a specified depth limit.

It is useful when:

- The search space is large or infinite.
- We want to avoid going too deep into the search tree.

Key points about DLS:

- Similar to DFS but stops expanding nodes beyond the given depth limit.
- Prevents infinite loops in cyclic or unbounded search spaces.
- May fail to find a solution if the depth limit is too small.

Algorithm

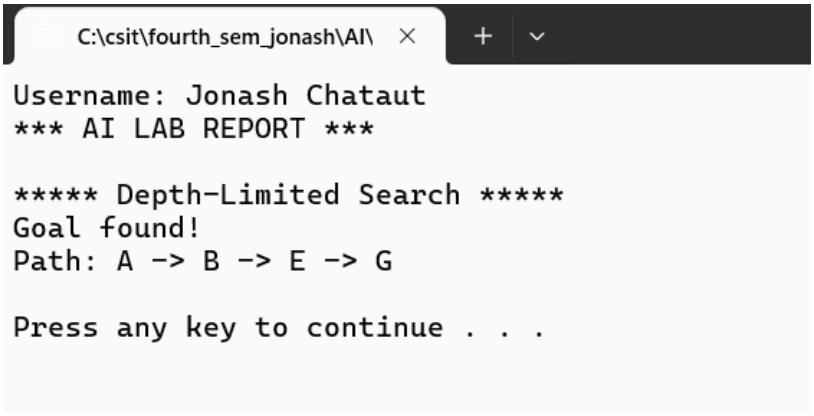
1. Start from the initial node and add it to the path.
2. Goal Test: If the current node is the goal, print the path and return success.
3. Depth Check: If the current depth limit is zero, backtrack.
4. Recursive Expansion: For each neighbor of the current node, recursively call DLS with a reduced depth limit.
5. Backtrack if no neighbor leads to the goal within the limit.
6. Return Failure if the goal is not found within the specified depth.

Source Code:

```
def depth_limited_search(graph, node, target, limit, path=[]):
  path.append(node)

  if node == target:
    print("Goal found!")
    print("Path:", " -> ".join(path))
    return True

  if limit <= 0:
    path.pop()  # backtrack
    return False

  for neighbor in graph.get(node, []):
    if depth_limited_search(graph, neighbor, target, limit - 1, path):
      return True

  path.pop()  # backtrack if goal not found through this node
  return False
```

```
    # Sample graph
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
    'F': [],
    'G': []
}

# Run DLS
start = 'A'
goal = 'G'
depth_limit = 3

print("***** Depth-Limited Search *****")
found = depth_limited_search(graph, start, goal, depth_limit)
if not found:
    print(f"Goal '{goal}' not found within depth {depth_limit}")
```

Output:



Lab No. 8

Title: Write a program to implement Bidirectional Search.

Bidirectional Search is a graph search algorithm that simultaneously explores from the start node forward and from the goal node backward until the two searches meet. This approach reduces the search space significantly compared to normal BFS or DFS.

Key characteristics of Bidirectional Search:

- Works best for unweighted graphs.
- Time complexity is approximately O(b^(d/2)) where b is branching factor and d is distance to the goal.
- Uses two simultaneous BFS traversals, one from the start and one from the goal.

Algorithm

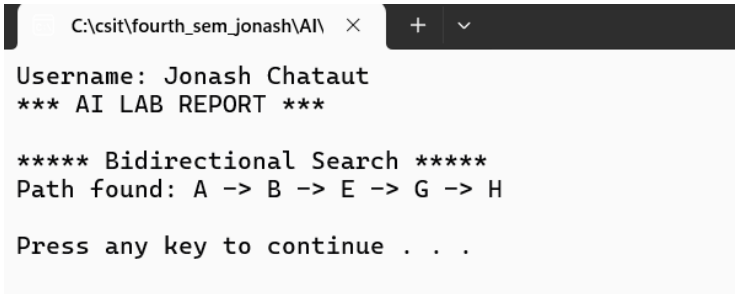
1. Reverse Graph: Create a reversed adjacency list for backward search.
2. Initialize Queues:
    - q_start for BFS from the start node.
    - q_goal for BFS from the goal node.
3. Visited Maps: Maintain visited dictionaries for both directions, storing the path taken.
4. Search Loop:
    - Expand one level forward from q_start.
    - Check if any forward-expanded node exists in the backward visited set.
    - Expand one level backward from q_goal.
    - Check if any backward-expanded node exists in the forward visited set.
5. Meeting Point: When found, combine paths from both directions.
6. Return Path: If no connection is found, output "No path found".

Source Code:

```
from collections import deque
def reverse_graph(graph):
  rev_graph = {node: [] for node in graph}
  for node, neighbors in graph.items():
    for neigh in neighbors:
      rev_graph.setdefault(neigh, []).append(node)
  return rev_graph
def bidirectional_search(graph, start, goal):
  if start == goal:
    print(f"Start is the same as goal: {start}")
    return [start]
  rev_graph = reverse_graph(graph)
  # Queues for BFS from both directions
  q_start = deque([(start, [start])])
  q_goal = deque([(goal, [goal])])
```

```
    # Visited dictionaries with paths
    visited_start = {start: [start]}
    visited_goal = {goal: [goal]}
    while q_start and q_goal:
        # Expand forward (normal graph)
        node_s, path_s = q_start.popleft()
        for neighbor in graph.get(node_s, []):
            if neighbor not in visited_start:
                visited_start[neighbor] = path_s + [neighbor]
                q_start.append((neighbor, path_s + [neighbor]))
                if neighbor in visited_goal:
                    # Meeting point found
                    full_path = visited_start[neighbor] + visited_goal[neighbor][-2::-1]
                    print(f"Path found: {' -> '.join(full_path)}")
                    return full_path
        # Expand backward (reverse graph)
        node_g, path_g = q_goal.popleft()
        for neighbor in rev_graph.get(node_g, []):
            if neighbor not in visited_goal:
                visited_goal[neighbor] = path_g + [neighbor]
                q_goal.append((neighbor, path_g + [neighbor]))
                if neighbor in visited_start:
                    # Meeting point found
                    full_path = visited_start[neighbor] + visited_goal[neighbor][-2::-1]
                    print(f"Path found: {' -> '.join(full_path)}")
                    return full_path
    print("No path found between the two nodes.")
    return None
# graph (directed)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
    'F': [],
    'G': ['H'],
    'H': []
}
start = 'A'
goal = 'H'
print("***** Bidirectional Search *****")
bidirectional_search(graph, start, goal)
```

Output:



Lab No. 9

Title: Write a program to implement Iterative Deepening Depth-First Search.

Iterative Deepening Depth-First Search (IDDFS) combines the space efficiency of Depth-First Search (DFS) with the completeness of Breadth-First Search (BFS). It repeatedly runs a Depth-Limited Search (DLS) with increasing depth limits until the goal is found or the maximum depth is reached.

Key points about IDDFS:

- Performs DFS up to depth 0, then depth 1, then depth 2, and so on.
- Uses O(bd) time complexity where b is branching factor and d is depth.
- Space complexity is O(bd), which is much less than BFS.
- Suitable for very large or infinite search spaces.

Algorithm

1. Graph Creation:
    - Take number of nodes and their connections from the user.
    - Store them as an adjacency list.
2. Iterative Deepening:
    - For each depth limit from 0 to max_depth:
        - Perform a Depth-Limited Search (DLS) from the start node.
3. Depth-Limited Search (DLS):
    - Visit the current node and add it to the path.
    - If depth limit is 0, check if node is the goal.
    - Otherwise, recursively visit each neighbor with reduced depth.
4. Goal Found:
    - If the goal is found, print the path and depth.
5. Goal Not Found:
    - If all depth limits are exhausted without success, report failure.

Source Code:

```
def createGraph(graph):
  print("*****  Iterative Deepening Depth-First Search  *****")
  n = int(input("Enter the number of nodes in graph:- "))
  print("Enter nodes and connected nodes in format -> node: connectedNode_1, connectedNode_2,...")
  for _ in range(n):
    node = input("Node -> ").split(":")
    graph[node[0].strip()] = [n.strip() for n in node[1].split(",") if n.strip()]
  return graph
def DLS(graph, node, target, depth,path):
  path.append(node)
  print("Visited:"," -> ".join(path))
  if depth == 0:
    if node == target:
      return True
```

```
        else:
            path.pop()
            return False
    if depth > 0:
        for neighbor in graph.get(node, []):
            if DLS(graph, neighbor, target, depth - 1,path):
                return True
        path.pop() #backtrack
    return False
def IDDFS(graph, start, target, max_depth):
    for depth in range(max_depth + 1):
        print(f"Searching at depth: {depth}")
        path =[]
        if DLS(graph, start, target, depth, path):
            print(f"Found target '{target}' at depth {depth}")
            print("Path:", " -> ".join(path))
            return True
    print(f"Target '{target}' not found within depth {max_depth}")
    return False
# main drive
graph = dict()
graph = createGraph(graph)
start = input("Enter the starting point:- ")
goal = input("Enter the goal point:- ")
max_depth_limit = int(input("Enter the maximum depth limit:- "))
print("Following is the IDDFS:-")
IDDFS(graph, start, goal, max_depth_limit)
```

Output:

```
C:\csit\fourth_sem_jonash\AI\
Username: Jonash Chataut
*** AI LAB REPORT ***

*****  Iterative Deepening Depth-First Search  *****
Enter the number of nodes in graph:- 10
Enter nodes and connected nodes in format -> node: connectedNode_1, connectedNode_2,...
Node -> A:B,C
Node -> B:D,E
Node -> C:F,G
Node -> D:H,I
Node -> E:
Node -> F:K
Node -> G:
Node -> H:
Node -> I:
Node -> K:
Enter the starting point:- A
Enter the goal point:- G
Enter the maximum depth limit:- 3
Following is the IDDFS:-
Searching at depth: 0
Visited: A
Searching at depth: 1
Visited: A
Visited: A -> B
Visited: A -> C
Searching at depth: 2
Visited: A
Visited: A -> B
Visited: A -> B -> D
Visited: A -> B -> E
Visited: A -> C
Visited: A -> C -> F
Visited: A -> C -> G
Found target 'G' at depth 2
Path: A -> C -> G
```