

Lab no: 1

Title: Create a tic tac toe game in python.

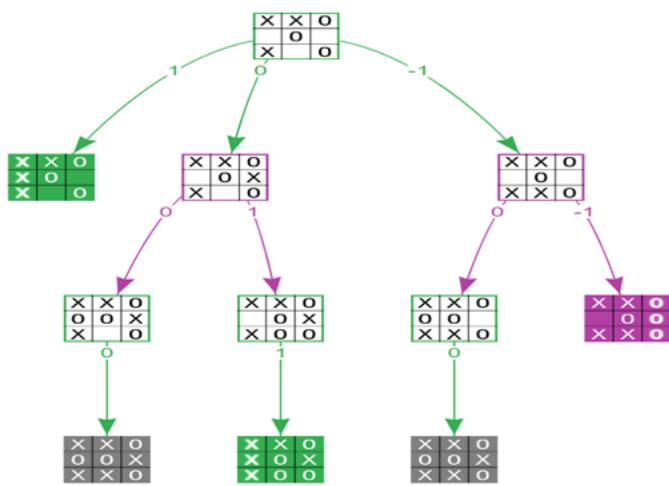
Tic-Tac-Toe

Tic-Tac-Toe is a classic two-player game played on a 3-by-3 grid where players alternate marking cells with an X or an O, aiming to align three of their marks in a row horizontally, vertically, or diagonally. Commonly enjoyed by children, the game has a straightforward structure, yet it has limited possibilities, making it relatively easy for the first player to adopt a winning strategy.

It's a great beginner project to practice:

- Loops
 - Conditionals
 - Lists
 - Functions

For example: Possible moves played by two players to win the game.



Source code:

```
def print_board(board):
    print("\n")
    print(f" {board[0]} | {board[1]} | {board[2]} ")
    print(" ---+---+---")
    print(f" {board[3]} | {board[4]} | {board[5]} ")
    print(" ---+---+---")
    print(f" {board[6]} | {board[7]} | {board[8]} ")
    print("\n")

def check_winner(board, player):
    win_combos = [
        (0, 1, 2), (3, 4, 5), (6, 7, 8), # rows
        (0, 3, 6), (1, 4, 7), (2, 5, 8), # cols
        (0, 4, 8), (2, 4, 6)           # diagonals
    ]
    for combo in win_combos:
        if all(board[i] == player for i in combo):
            return True
    return False

def is_draw(board):
    return all(cell in ["X", "O"] for cell in board)

def tic_tac_toe():
    board = [str(i) for i in range(1, 10)] # 1-9 positions
    current_player = "X"

    while True:
        print_board(board)
        move = input(f"Player {current_player}, choose a position (1-9): ")

        if not move.isdigit() or int(move) not in range(1, 10):
            print("Invalid input. Choose a number from 1 to 9.")
            continue

        move = int(move) - 1
        if board[move] in ["X", "O"]:
            print("That position is already taken. Choose another.")
            continue

        board[move] = current_player

        if check_winner(board, current_player):
            print_board(board)
```

```
print(f"🎉 Player {current_player} wins!")
break

if is_draw(board):
    print_board(board)
    print("It's a draw!")
    break

current_player = "O" if current_player == "X" else "X"

# Run the game
tic_tac_toe()
```

Output:

```
PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Tic Tac Toe\Tic Tac Toe 13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Tic Tac Toe/Tic Tac Toe 13.exe" -p 1  
●  
  
1 | 2 | 3  
---|---|---  
4 | 5 | 6  
---|---|---  
7 | 8 | 9  
  
Player X, choose a position (1-9): 1  
  
X | 2 | 3  
---|---|---  
4 | 5 | 6  
---|---|---  
7 | 8 | 9  
  
Player 0, choose a position (1-9): 3  
  
X | 2 | 0  
---|---|---  
4 | 5 | 6  
---|---|---  
7 | 8 | 9  
  
Player X, choose a position (1-9): 5  
  
Player X, choose a position (1-9): 5  
  
X | 2 | 0  
---|---|---  
4 | X | 6  
---|---|---  
7 | 8 | 9  
  
Player 0, choose a position (1-9): 6  
  
X | 2 | 0  
---|---|---  
4 | X | 0  
---|---|---  
7 | 8 | 9  
  
Player X, choose a position (1-9): 9  
  
X | 2 | 0  
---|---|---  
4 | X | 0  
---|---|---  
7 | 8 | X  
  
🎉 Player X wins!  
PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Tic Tac Toe\Tic Tac Toe 13.exe
```

Lab no: 2

Title: Create water jug puzzle in python.

Water Jug Puzzle:

The Water Jug Puzzle is a classic puzzle where you're given two jugs of different capacities and an unlimited supply of water. The goal is to measure an exact amount of water using only these two jugs.

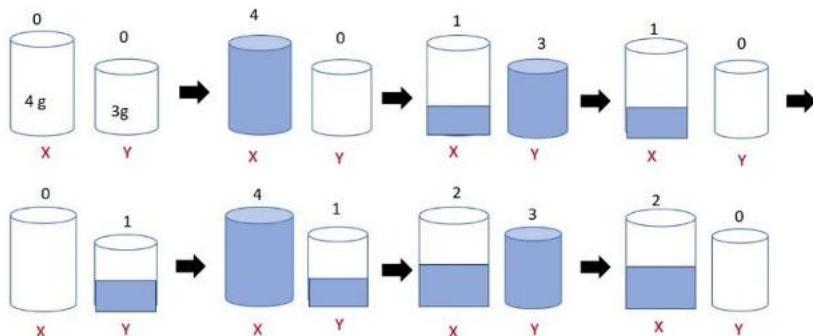
You can:

- Fill a jug completely
 - Empty a jug
 - Pour water from one jug to the other until one is full or the other is empty

This problem is often used to teach logical reasoning, problem-solving, and state-space search in computer science and artificial intelligence.

For example:

Solution



Source code:

```
from collections import deque

def water_jug_bfs(capacityA, capacityB, target):
    # Check if solution is possible
    def gcd(a, b):
        while b:
            a, b = b, a % b
        return a

    if target > max(capacityA, capacityB) or target % gcd(capacityA, capacityB) != 0:
        return "No solution possible"

    visited = set()
    queue = deque()
    queue.append((0, 0, [])) # initial state and empty path

    while queue:
        jugA, jugB, path = queue.popleft()

        if jugA == target or jugB == target:
            return path + [(jugA, jugB)]

        if (jugA, jugB) in visited:
            continue
        visited.add((jugA, jugB))

        next_states = []

        # Fill Jug A
        next_states.append((capacityA, jugB))
        # Fill Jug B
        next_states.append((jugA, capacityB))
        # Empty Jug A
        next_states.append((0, jugB))
        # Empty Jug B
        next_states.append((jugA, 0))

        # Pour A -> B
        pour_to_B = min(jugA, capacityB - jugB)
        next_states.append((jugA - pour_to_B, jugB + pour_to_B))

        # Pour B -> A
        pour_to_A = min(jugB, capacityA - jugA)
```

```

next_states.append((jugA + pour_to_A, jugB - pour_to_A))

for state in next_states:
    if state not in visited:
        queue.append((state[0], state[1], path + [(jugA, jugB)]))

return "No solution found"

def main():
    print("Water Jug Problem Solving By Sarfraz Alam")
    capacityA = int(input("Enter capacity of Jug A (liters): "))
    capacityB = int(input("Enter capacity of Jug B (liters): "))
    target = int(input("Enter the target amount to measure (liters): "))

    solution = water_jug_bfs(capacityA, capacityB, target)
    if isinstance(solution, str):
        print(solution)
    else:
        print("\nSteps to measure", target, "liters:")
        for step in solution:
            print(f"Jug A: {step[0]} liters, Jug B: {step[1]} liters")

if __name__ == "__main__":
    main()

```

Output:

```

PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/Users/Dell/AppData/13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/waterjugProblem.py"
● Water Jug Problem Solving By Sarfraz Alam
Enter capacity of Jug A (liters): 2
Enter capacity of Jug B (liters): 3
Enter the target amount to measure (liters): 1

Steps to measure 1 liters:
Jug A: 0 liters, Jug B: 0 liters
Jug A: 0 liters, Jug B: 3 liters
Jug A: 2 liters, Jug B: 1 liters
○ PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab>

```



Lab no: 3

Title: Create Chatbot in python.

Chatbot:

Chatbot is a computer program that uses rules or artificial intelligence to communicate with people automatically, usually through text or voice, to answer questions or help with tasks.

- It doesn't learn or adapt like AI chatbots.
- Only works within the limits of the programmed rules.

Purpose of the Program

This chatbot, named Jarvis, is a basic conversational agent implemented in Python. It interacts with users by recognizing predefined keywords or phrases and responding with appropriate answers. The chatbot can greet users, respond to simple questions, tell jokes, and remember the user's name during the conversation.

Key Features

- o Pattern Matching: The chatbot matches user input against keys in a dictionary to generate responses.
- o State Maintenance: Remembers the user's name once provided.
- o Simple Natural Language Handling: Handles common phrases and can recognize commands like "my name is" or "what is my name."
- o Exit Condition: Allows graceful exit on user command

Source code:

```
def chatbot():
    print("ChatBot: Hello! I'm a simple chatbot. Type 'bye' to exit.")

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

        if "hello" in user_input or "hi" in user_input:
            print("ChatBot: Hello Sarfraj! How can I help you?")
        elif "how are you" in user_input:
            print("ChatBot: I'm just a bunch of code, but thanks for asking!")
        elif "your name" in user_input:
            print("ChatBot: I'm just a chatbot with no name.")
        elif "bye" in user_input:
            print("ChatBot: Goodbye! Have a great day.")
            break
        elif "help" in user_input:
            print("ChatBot: Sure! You can ask me about my name, say hi, or just chat.")
        else:
            print("ChatBot: I'm not sure how to respond to that. Try asking something else.")

# Run the chatbot
chatbot()
```

Output:

```
PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/Users/Dell 13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/chatbot.py
● ChatBot: Hello! I'm a simple chatbot. Type 'bye' to exit.
You: hi
ChatBot: Hello Sarfraj! How can I help you?
You: how are you?
ChatBot: I'm just a bunch of code, but thanks for asking!
You: bye
ChatBot: Goodbye! Have a great day.
○ PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab>
```

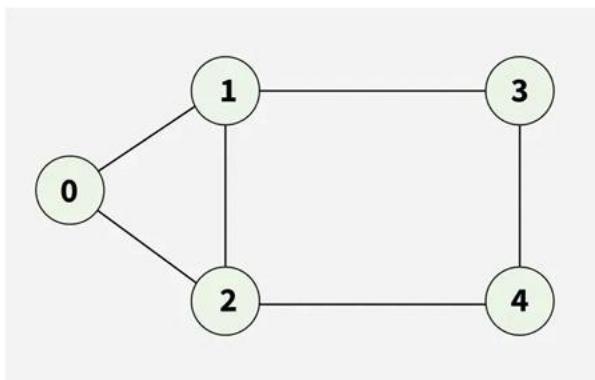
Lab no: 4

Title: Create Breadth First Search Algorithm in python.

Breadth First Search

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent nodes. Once all adjacent are visited, then their adjacent are traversed.

For example: Input: adj[][] = [[1,2], [0,2,3], [0,1,4], [1,4], [2,3]] Output: [0, 1, 2, 3, 4]



Explanation: Starting from 0, the BFS traversal will follow these steps:

Visit 0 → Output: [0]

Visit 1 (first neighbor of 0) → Output: [0, 1]

Visit 2 (next neighbor of 0) → Output: [0, 1, 2]

Visit 3 (next neighbor of 1) → Output: [0, 1, 2, 3]

Visit 4 (neighbor of 2) → Final Output: [0, 1, 2, 3, 4]

Source code:

```
import queue
adj_list = {
    "a": ["b", "c"],
    "b": ["d", "e", "f"],
    "c": ["g", "h"],
    "d": [],
    "e": ["i", "j"],
    "f": [],
    "g": ["k", "l"],
    "h": ["m"],
    "i": [],
    "j": [],
    "k": [],
    "l": [],
    "m": []
}

visited = [] # List for visited nodes.
queue = [] # Initialize a queue
bfs_traversal_output = []

def bfs(visited, graph, node): #function for BFS
    visited.append(node)
    queue.append(node)
    while queue: # Creating loop to visit each node
        m = queue.pop(0)
        bfs_traversal_output.append(m)
        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

print("The traversing of graph: ", bfs_traversal_output);

print("Breadth-First Search By Sarfraz Alam");
startnode = input("Enter the node from where you want to traverse: ");
bfs(visited, adj_list, startnode)
```

Output:

```
PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/Users/D
13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/BFS.py"
● Breadth-First Search By Sarfraz Alam
Enter the node from where you want to traverse: b
The traversing of graph:  ['b', 'd', 'e', 'f', 'i', 'j']
○ PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab>
```

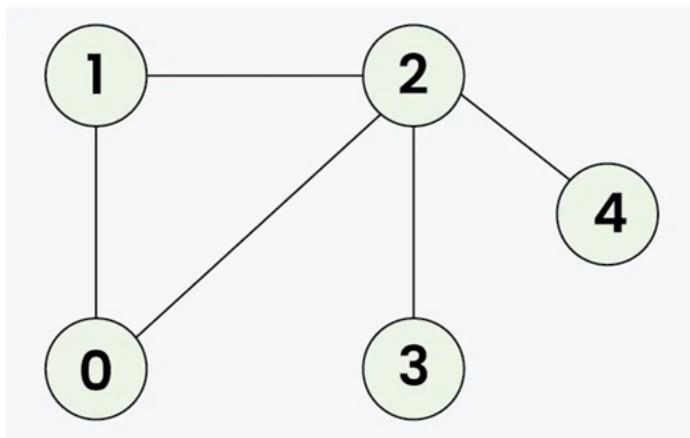
Lab no: 5

Title: Create Depth First Search Algorithm in python.

Depth First Search:

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. This is similar to a tree, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.

For example: Input: adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]] Output: [0 1 2 3 4]



Output: [0 1 2 3 4]

Explanation: The source vertex s is 0. We visit it first, then we visit an adjacent.

Start at 0: Mark as visited. Output: 0

Move to 1: Mark as visited. Output: 1

Move to 2: Mark as visited. Output: 2

Move to 3: Mark as visited. Output: 3 (backtrack to 2)

Move to 4: Mark as visited. Output: 4 (backtrack to 2, then backtrack to 1, then to 0)

(Note: There can be multiple DFS traversals of a graph according to the order in which we pick adjacent vertices).

Source code:

```
adj_list = {  
    "a": ["b", "c"],  
    "b": ["d", "e"],  
    "c": ["f", "g"],  
    "d": [],  
    "e": ["h", "i"],  
    "f": [],  
    "g": ["j", "k"],  
    "h": [],  
    "i": [],  
    "j": [],  
    "k": []  
}  
closedlist = {}  
dfs_traversal_output = []  
for node in adj_list.keys():  
    closedlist[node] = "notvisited"  
def dfs(u):  
    closedlist[u] = "visited"  
    dfs_traversal_output.append(u)  
    for v in adj_list[u]:  
        if closedlist[v] != "visited":  
            dfs(v)  
print("Depth-First Search By Sarfraz Alam")  
startnode = input("Enter the node from where you want to traverse: ")  
dfs(startnode)  
print("Traversed sequence of the graph: ", dfs_traversal_output)
```

Output:

```
PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/Users/D  
13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/DFS.py"  
● Depth-First Search By Sarfraz Alam  
Enter the node from where you want to traverse: b  
Traversed sequence of the graph:  ['b', 'd', 'e', 'h', 'i']  
○ PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> █
```

Lab no: 6

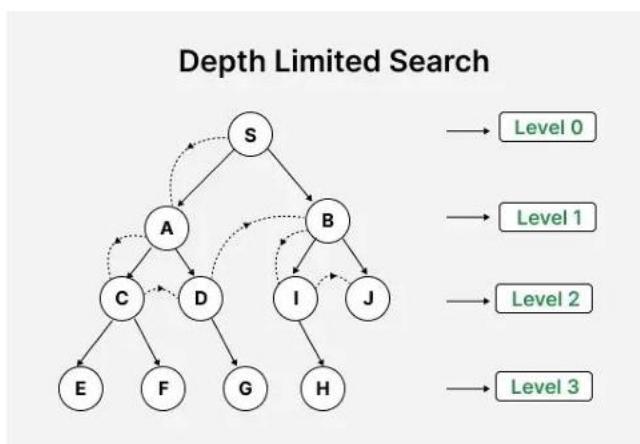
Title: Create Depth Limit Search Algorithm in python.

Depth Limit Search:

Depth Limited Search is a key algorithm used in solving problem space concerned with artificial intelligence. But before starting it lets first understand Depth First Search which is an algorithm that explores a tree or graph by starting at the root node and exploring as far as possible along each branch before backtracking. It follows a path from the root to a leaf node then backtracks to explore other path.

How Depth Limited Search Works:

Initialization: Begin at the root node with a specified depth limit. Exploration: Traverse the tree or graph, exploring each node's children. Depth Check: If the current depth exceeds the set limit, stop exploring that path and backtrack. Goal Check: If the goal node is found within the depth limit, the search is successful. Backtracking: If the search reaches the depth limit or a leaf node without finding the goal, backtrack and explore other branches.



Source code:

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

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

    if limit <= 0:
        path.pop()
        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

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

print(" Depth-Limited Search By Sarfraz Alam")
start = input("Enter the starting node: ").strip().upper()
goal = input("Enter the goal node: ").strip().upper()
try:
    depth_limit = int(input("Enter the depth limit: "))
except ValueError:
    print("Invalid depth limit. Please enter an integer.")
    exit()

found = depth_limited_search(graph, start, goal, depth_limit)
if not found:
    print(f"\nGoal '{goal}' not found within depth {depth_limit}")
```

Output:

```
● PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/Users/13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/DLS.py"
Depth-Limited Search By Sarfraz Alam
Enter the starting node: a
Enter the goal node: f
Enter the depth limit: 3

Goal found!
Path: A -> B -> C -> F
○ PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab>
```

Lab no: 7

Title: Create Iterative Deepening Depth First Search (IDDFS) Algorithm in python.

Iterative Deepening Depth First Search:

Iterative Deepening Depth-First Search (IDDFS), also known as Iterative Deepening Search (IDS), is an uninformed search algorithm that combines the memory efficiency of Depth-First Search (DFS) with the completeness and optimality (for unweighted graphs) of Breadth-First Search (BFS).

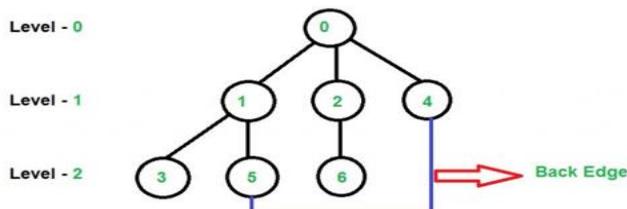
How does IDDFS works:

IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically, we do DFS in a BFS fashion.

Illustration: There can be two cases:

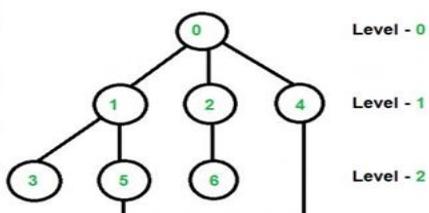
When the graph has no cycle: This case is simple. We can DFS multiple times with different height limits.

When the graph has cycles. This is interesting as there is no visited flag in IDDFS.



Although, at first sight, it may seem that since there are only 3 levels, so we might think that Iterative Deepening Depth First Search of level 3, 4, 5,...and so on will remain same. But, this is not the case. You can see that there is a cycle in the above graph, hence IDDFS will change for level- 3,4,5..and so on.

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

Source code:

```
graph={  
    "a":["b","c","d"],  
    "b":["e","f"],  
    "c":["g","h"],  
    "d":["i"],  
    "e":[],  
    "f":[],  
    "g":[],  
    "h":[],  
    "i":[]  
  
}  
  
def dfs(node, goal, d_limit, visited, path):  
    if node == goal:  
        return "FOUND", path + [node]  
    elif d_limit == 0:  
        return "NOT_FOUND", visited  
    else:  
        visited.add(node)  
        for child in graph[node]:  
            if child not in visited:  
                result, tv_path = dfs(child, goal, d_limit - 1, visited, path + [node])  
                if result == "FOUND":  
                    return "FOUND", tv_path  
        return "NOT_FOUND", visited  
  
def iddfs(root, goal):  
    d_limit = 0  
    while True:  
        visited = set()  
        result, tv_path = dfs(root, goal, d_limit, visited, [])  
        print("Depth Limit:", d_limit, "Traversed path:", ' -> '.join(tv_path))  
        if result == "FOUND":  
            return "Goal node found! Traversed path:" + ' -> '.join(tv_path)  
        elif result == "NOT_FOUND":  
            d_limit += 1  
  
print(" Iterative Deepening Depth First Search By Sarfraz Alam")  
root = input('Enter the start node: ')  
goal = input('Enter the goal node: ')  
result = iddfs(root, goal)  
print(result)
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/Users/Dell 13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/IDDFs.py"
● Iterative Deepening Depth First Search By Sarfraz Alam
Enter the start node: a
Enter the goal node: g
Depth Limit: 0 Traversed path:
Depth Limit: 1 Traversed path: a
Depth Limit: 2 Traversed path: a -> c -> g
Goal node found! Traversed path:a -> c -> g
○ PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab>
```

Lab no: 8

Title: Create Bidirectional Search Algorithm in python.

Bidirectional Search:

Bidirectional search is a graph search algorithm which finds smallest path from source to goal vertex. It runs two simultaneous search.

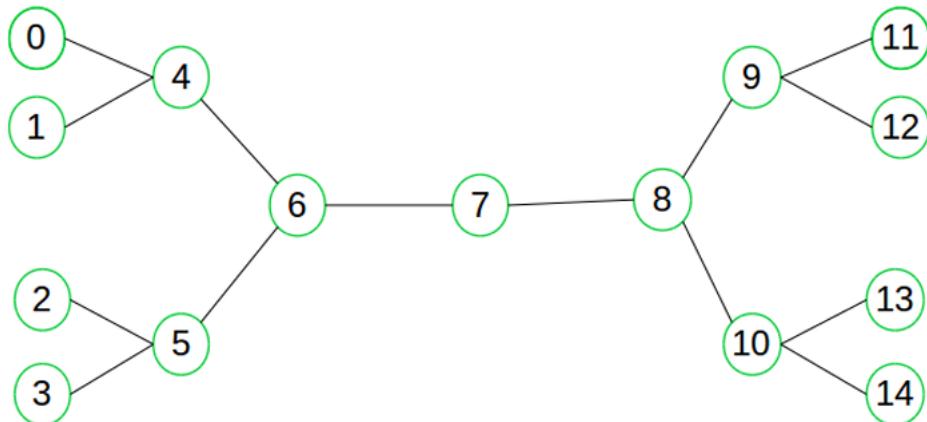
Forward search from source/initial vertex toward goal vertex. Backward search from goal/target vertex toward source vertex.

When to use bidirectional approach?

We can consider bidirectional approach when,

1. Both initial and goal states are unique and completely defined.
2. The branching factor is exactly the same in both directions.

For Example:



Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We can clearly see that we have successfully avoided unnecessary exploration.

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)
    q_start = deque([(start, [start])])
    q_goal = deque([(goal, [goal])])
    visited_start = {start: [start]}
    visited_goal = {goal: [goal]}
    while q_start and q_goal:
        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:
                full_path = visited_start[neighbor] + visited_goal[neighbor][-2::-1]
                print(f"\nPath found: {' -> '.join(full_path)}")
                return full_path
        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:
                full_path = visited_start[neighbor] + visited_goal[neighbor][-2::-1]
                print(f"\nPath found: {' -> '.join(full_path)}")
                return full_path
    print("\nNo path found between the two nodes.")
    return None

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
```

```

'C': ['F'],
'D': [],
'E': ['G'],
'F': [],
'G': ['H'],
'H': []
}
print(" Bidirectional Search Algorithm By Sarfraz Alam")
start = input("Enter start node: ").strip().upper()
goal = input("Enter goal node: ").strip().upper()

print("\nBidirectional Search by Sarfraz Alam")
bidirectional_search(graph, start, goal)

```

Output:

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/Users/Dell/AppData/Local
13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/BidirectionalSearch.py"
● Bidirectional Search Algorithm By Sarfraz Alam
Enter start node: a
Enter goal node: g

Bidirectional Search by Sarfraz Alam

Path found: A -> B -> E -> G
○ PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab>

```



Lab no: 9

Title: Create Uniform Cost Search Algorithm in python.

Uniform Cost Search:

Uniform Cost Search (UCS) is a search algorithm used in artificial intelligence (AI) for finding the least cost path in a graph. It is a variant of Dijkstra's algorithm and is particularly useful when all edges of the graph have different weights, and the goal is to find the path with the minimum total cost from a start node to a goal node.

How Uniform Cost Search Works:

Initialization: UCS starts with the root node. It is added to the priority queue with a cumulative cost of zero since no steps have been taken yet.

Node Expansion: The node with the lowest path cost is removed from the priority queue. This node is then expanded, and its neighbors are explored.

Exploring Neighbors: For each neighbor of the expanded node, the algorithm calculates the total cost from the start node to the neighbor through the current node. If a neighbor node is not in the priority queue, it is added to the queue with the calculated cost. If the neighbor is already in the queue but a lower cost path to this neighbor is found, the cost is updated in the queue.

Goal Check: After expanding a node, the algorithm checks if it has reached the goal node. If the goal is reached, the algorithm returns the total cost to reach this node and the path taken.

Repetition: This process repeats until the priority queue is empty or the goal is reached.

Source code:

```
import heapq

class Node:
    def __init__(self, position, cost=0, parent=None):
        self.position = position
        self.cost = cost
        self.parent = parent

    def __lt__(self, other):
        return self.cost < other.cost

def uniform_cost_search(start, goal, graph):
    open_list = []
    closed_list = set()

    start_node = Node(start, 0)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        current_position = current_node.position

        if current_position in closed_list:
            continue

        closed_list.add(current_position)

        if current_position == goal:
            path = []
            total_cost = current_node.cost
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1], total_cost

        for neighbor, cost in graph.get(current_position, {}).items():
            if neighbor in closed_list:
                continue

            neighbor_node = Node(neighbor, current_node.cost + cost, current_node)
            heapq.heappush(open_list, neighbor_node)

    return None, float('inf')
```

```

graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

print("Uniform Cost Search by Sarfraz Alam\n")
start = input("Enter start node: ").strip().upper()
goal = input("Enter goal node: ").strip().upper()

if start not in graph or goal not in graph:
    print("\nInvalid start or goal node. Please check the node names.")
else:
    path, cost = uniform_cost_search(start, goal, graph)
    if path:
        print("\nPath found:", " -> ".join(path))
        print("Total cost:", cost)
    else:
        print("\nNo path found between the given nodes.")

```

Output:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/Users/Dell/AppData/L
13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/UniformCostSearch.py"
Uniform Cost Search by Sarfraz Alam

Enter start node: a
Enter goal node: d

Path found: A -> B -> C -> D
Total cost: 4
PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab>
PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> █

```



Lab no: 10

Title: Create a Greedy Best-First Search algorithm in python.

Greedy Best-First Search:

Greedy Best-First Search is an informed search algorithm that uses a heuristic function to guide its search towards the goal. Unlike other search algorithms, it always chooses the path that appears best at the moment according to the heuristic function. It uses a priority queue to always expand the node that appears to be closest to the goal.

Key Features:

- **Heuristic-based:** Uses a heuristic function to estimate the cost from current node to goal
- **Greedy approach:** Always selects the node that appears best according to the heuristic
- **Informed search:** Unlike uninformed searches, it has knowledge about the goal location
- **Time complexity:** $O(b^m)$ where b is branching factor and m is maximum depth
- **Space complexity:** $O(b^m)$ in worst case

How Greedy Best-First Search Works:

1. **Initialization:** Start with the initial node in a priority queue
2. **Heuristic Evaluation:** Calculate heuristic value for each node (distance to goal)
3. **Node Selection:** Always select the node with lowest heuristic value
4. **Expansion:** Expand the selected node and add neighbors to queue
5. **Goal Check:** Continue until goal is reached or queue is empty

Source Code:

```
import heapq

# Graph representation with letter nodes
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5, 'E': 3},
    'C': {'A': 4, 'B': 2, 'F': 6},
    'D': {'B': 5, 'E': 1, 'G': 2},
    'E': {'B': 3, 'D': 1, 'F': 4, 'G': 1},
    'F': {'C': 6, 'E': 4, 'H': 3},
    'G': {'D': 2, 'E': 1, 'H': 2},
    'H': {'F': 3, 'G': 2}
}

# Heuristic values (estimated distance to goal)
# These represent straight-line distances to each node
def get_heuristic_values(goal):
    heuristics = {
        'A': {'A': 0, 'B': 7, 'C': 6, 'D': 8, 'E': 5, 'F': 9, 'G': 4, 'H': 3},
        'B': {'A': 7, 'B': 0, 'C': 3, 'D': 5, 'E': 2, 'F': 8, 'G': 3, 'H': 6},
        'C': {'A': 6, 'B': 3, 'C': 0, 'D': 8, 'E': 5, 'F': 4, 'G': 6, 'H': 7},
        'D': {'A': 8, 'B': 5, 'C': 8, 'D': 0, 'E': 2, 'F': 7, 'G': 1, 'H': 3},
        'E': {'A': 5, 'B': 2, 'C': 5, 'D': 2, 'E': 0, 'F': 4, 'G': 1, 'H': 3},
        'F': {'A': 9, 'B': 8, 'C': 4, 'D': 7, 'E': 4, 'F': 0, 'G': 5, 'H': 2},
        'G': {'A': 4, 'B': 3, 'C': 6, 'D': 1, 'E': 1, 'F': 5, 'G': 0, 'H': 2},
        'H': {'A': 3, 'B': 6, 'C': 7, 'D': 3, 'E': 3, 'F': 2, 'G': 2, 'H': 0}
    }
    return heuristics.get(goal, {})

# Greedy Best-First Search with letter nodes
def greedy_best_first_search(graph, start, goal):
    if start not in graph or goal not in graph:
        return None, "Invalid start or goal node"

    visited = set()
    priority_queue = []
    heuristic_values = get_heuristic_values(goal)

    # Heap queue with (heuristic, node)
    heapq.heappush(priority_queue, (heuristic_values.get(start, float('inf')), start))
    came_from = {start: None}
```

```

print(f"\nSearching from {start} to {goal}...")
print("Search Progress:")

while priority_queue:
    current_heuristic, current = heapq.heappop(priority_queue)

    if current in visited:
        continue

    visited.add(current)
    print(f"Visiting node {current} (heuristic: {current_heuristic})")

    if current == goal:
        print(f"Goal {goal} reached!")
        break

    # Explore neighbors
    for neighbor in graph.get(current, {}):
        if neighbor not in visited:
            neighbor_heuristic = heuristic_values.get(neighbor, float('inf'))
            heapq.heappush(priority_queue, (neighbor_heuristic, neighbor))
            if neighbor not in came_from:
                came_from[neighbor] = current

    # Reconstruct path
    if goal not in came_from:
        return None, f"No path found from {start} to {goal}"

path = []
node = goal
while node:
    path.append(node)
    node = came_from.get(node)
path.reverse()

return path, "Path found successfully"

# Display available nodes
def display_graph():
    print("Available nodes in the graph:")
    print("Nodes:", sorted(graph.keys()))
    print("\nGraph connections:")
    for node, neighbors in graph.items():
        print(f"{node}: {list(neighbors.keys())}")

```

```

# Main execution
def main():
    print("=" * 50)
    print("Greedy Best-First Search By Sarfraz Alam")
    print("=" * 50)

    display_graph()

    print("\n" + "-" * 30)

    # Get user input
    while True:
        start = input("\nEnter the source node (A-H): ").strip().upper()
        if start in graph:
            break
        print("Invalid node! Please choose from A, B, C, D, E, F, G, H")

    while True:
        goal = input("Enter the destination node (A-H): ").strip().upper()
        if goal in graph:
            break
        print("Invalid node! Please choose from A, B, C, D, E, F, G, H")

    # Perform search
    path, message = greedy_best_first_search(graph, start, goal)

    # Display results
    print("\n" + "=" * 50)
    print("SEARCH RESULTS")
    print("=" * 50)

    if path:
        print(f"Status: {message}")
        print(f"Path found: {' -> '.join(path)}")
        print(f"Number of nodes in path: {len(path)}")

    # Calculate total cost
    total_cost = 0
    for i in range(len(path) - 1):
        current_node = path[i]
        next_node = path[i + 1]
        cost = graph[current_node].get(next_node, 0)
        total_cost += cost
        print(f" {current_node} -> {next_node}: cost = {cost}")

```

```

        print(f"Total path cost: {total_cost}")
else:
    print(f"Status: {message}")

# Run the program
if __name__ == "__main__":
    main()

```

Output:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

- PS E:\Sarfraj\4th SEMESTER\Artificial Intelligence\13.exe "e:/Sarfraj/4th SEMESTER/Artificial Intelligence/13.py"

```

=====
Greedy Best-First Search By Sarfraj Alam
=====

Available nodes in the graph:
Nodes: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

Graph connections:
A: ['B', 'C']
B: ['A', 'C', 'D', 'E']
C: ['A', 'B', 'F']
D: ['B', 'E', 'G']
E: ['B', 'D', 'F', 'G']
F: ['C', 'E', 'H']
G: ['D', 'E', 'H']
H: ['F', 'G']

-----
Enter the source node (A-H): a
Enter the destination node (A-H): g

Searching from A to G...
Search Progress:
Visiting node A (heuristic: 4)
Visiting node B (heuristic: 3)
Visiting node D (heuristic: 1)
Visiting node G (heuristic: 0)
Goal G reached!

=====
SEARCH RESULTS
=====
Status: Path found successfully
Path found: A -> B -> D -> G
Number of nodes in path: 4
  A -> B: cost = 1
  B -> D: cost = 5
  D -> G: cost = 2
Total path cost: 8
PS E:\Sarfraj\4th SEMESTER\Artificial Intelligence\Lab>

```



Lab no: 11

Title: Create a A* Search Algorithm in python.

A* Search Algorithm:

A* (A-star) is one of the most popular and widely used informed search algorithms in artificial intelligence and pathfinding. It combines the advantages of Dijkstra's algorithm (guarantees shortest path) and Greedy Best-First Search (uses heuristics for efficiency). A* uses both the actual cost from start to current node (g) and the estimated cost from current node to goal (h) to make optimal decisions.

Key Features:

- **Optimal:** Guarantees the shortest path if heuristic is admissible
- **Complete:** Will find a solution if one exists
- **Informed Search:** Uses both actual cost (g) and heuristic (h)
- $f(n) = g(n) + h(n)$: Total estimated cost through node n
- **Admissible Heuristic:** Never overestimates the actual cost to goal

How A* Search Works:

1. **Initialization:** Start with initial node in open set with $f(\text{start}) = h(\text{start})$
2. **Node Selection:** Select node with lowest $f(n)$ value from open set
3. **Goal Check:** If current node is goal, reconstruct and return path
4. **Expansion:** For each neighbor, calculate g , h , and f values
5. **Update:** Add to open set if better path found or node is new
6. **Repeat:** Continue until goal found or open set is empty

Mathematical Foundation:

- $g(n)$: Actual cost from start to node n
- $h(n)$: Heuristic estimate from node n to goal
- $f(n) = g(n) + h(n)$: Total estimated cost of path through n

Source code:

```
import heapq

# Graph representation with letter nodes and costs
graph = {
    'A': {'B': 2, 'C': 3},
    'B': {'A': 2, 'C': 1, 'D': 4, 'E': 2},
    'C': {'A': 3, 'B': 1, 'F': 5},
    'D': {'B': 4, 'E': 1, 'G': 3},
    'E': {'B': 2, 'D': 1, 'F': 2, 'G': 2},
    'F': {'C': 5, 'E': 2, 'H': 1},
    'G': {'D': 3, 'E': 2, 'H': 2},
    'H': {'F': 1, 'G': 2}
}

# Heuristic function (admissible - never overestimates)
def get_heuristic_values(goal):
    # Straight-line distances to each possible goal
    heuristics = {
        'A': {'A': 0, 'B': 2, 'C': 3, 'D': 6, 'E': 4, 'F': 7, 'G': 6, 'H': 8},
        'B': {'A': 2, 'B': 0, 'C': 1, 'D': 4, 'E': 2, 'F': 5, 'G': 4, 'H': 6},
        'C': {'A': 3, 'B': 1, 'C': 0, 'D': 5, 'E': 3, 'F': 4, 'G': 5, 'H': 5},
        'D': {'A': 6, 'B': 4, 'C': 5, 'D': 0, 'E': 1, 'F': 3, 'G': 2, 'H': 3},
        'E': {'A': 4, 'B': 2, 'C': 3, 'D': 1, 'E': 0, 'F': 2, 'G': 2, 'H': 3},
        'F': {'A': 7, 'B': 5, 'C': 4, 'D': 3, 'E': 2, 'F': 0, 'G': 3, 'H': 1},
        'G': {'A': 6, 'B': 4, 'C': 5, 'D': 2, 'E': 2, 'F': 3, 'G': 0, 'H': 2},
        'H': {'A': 8, 'B': 6, 'C': 5, 'D': 3, 'E': 3, 'F': 1, 'G': 2, 'H': 0}
    }
    return heuristics.get(goal, {})

# A* Search Algorithm
def a_star_search(graph, start, goal):
    if start not in graph or goal not in graph:
        return None, float('inf'), "Invalid start or goal node"

    # Priority queue: (f_score, g_score, node)
    open_set = []
    heapq.heappush(open_set, (0, 0, start))

    # Track the path
    came_from = {}

    # Cost from start to each node
    g_score = {node: float('inf') for node in graph}
```

```

g_score[start] = 0

# Estimated total cost from start to goal through each node
heuristic_values = get_heuristic_values(goal)
f_score = {node: float('inf') for node in graph}
f_score[start] = heuristic_values.get(start, 0)

# Keep track of nodes in open set
open_set_nodes = {start}

print(f"\nA* Search from {start} to {goal}")
print("Search Progress:")
print(f'{Node}:<4} {g(n)}:{<6} {h(n)}:{<6} {f(n)}:{<6} {Action}')
print("-" * 35)

while open_set:
    # Get node with lowest f_score
    current_f, current_g, current = heapq.heappop(open_set)
    open_set_nodes.discard(current)

    h_value = heuristic_values.get(current, 0)
    print(f'{current}:<4} {current_g}:{<6} {h_value}:{<6} {current_f}:{<6} Exploring')

    # Goal reached
    if current == goal:
        print(f'{current}:<4} {current_g}:{<6} {h_value}:{<6} {current_f}:{<6} GOAL!')
        break

    # Explore neighbors
    for neighbor, cost in graph.get(current, {}).items():
        tentative_g = g_score[current] + cost

        # Better path found
        if tentative_g < g_score[neighbor]:
            came_from[neighbor] = current
            g_score[neighbor] = tentative_g
            h_neighbor = heuristic_values.get(neighbor, 0)
            f_score[neighbor] = tentative_g + h_neighbor

            if neighbor not in open_set_nodes:
                heapq.heappush(open_set, (f_score[neighbor], tentative_g, neighbor))
                open_set_nodes.add(neighbor)

# Reconstruct path
if goal not in came_from and start != goal:

```

```

        return None, float('inf'), f"No path found from {start} to {goal}"

    path = []
    node = goal
    while node in came_from:
        path.append(node)
        node = came_from[node]
    path.append(start)
    path.reverse()

    total_cost = g_score[goal]
    return path, total_cost, "Optimal path found"

# Display graph information
def display_graph():
    print("Available nodes in the graph:")
    print("Nodes:", sorted(graph.keys()))
    print("\nGraph connections with costs:")
    for node, neighbors in graph.items():
        connections = [f"\t{neighbor}({cost})" for neighbor, cost in neighbors.items()]
        print(f"\t{node}: {', '.join(connections)})")

# Main execution function
def main():
    print("\n" + "=" * 60)
    print("A* Search Algorithm By Sarfraz Alam")
    print("\n" + "=" * 60)

    display_graph()

    print("\n" + "-" * 40)

    # Get user input
    while True:
        start = input("\nEnter the source node (A-H): ").strip().upper()
        if start in graph:
            break
        print("Invalid node! Please choose from A, B, C, D, E, F, G, H")

    while True:
        goal = input("Enter the destination node (A-H): ").strip().upper()
        if goal in graph:
            break
        print("Invalid node! Please choose from A, B, C, D, E, F, G, H")

```

```

# Perform A* search
path, total_cost, message = a_star_search(graph, start, goal)

# Display results
print("\n" + "=" * 60)
print("A* SEARCH RESULTS")
print("=" * 60)

if path:
    print(f"Status: {message}")
    print(f"Optimal path: {' -> '.join(path)}")
    print(f"Total optimal cost: {total_cost}")
    print(f"Number of nodes in path: {len(path)}")

    print("\nDetailed path costs:")
    running_cost = 0
    for i in range(len(path) - 1):
        current_node = path[i]
        next_node = path[i + 1]
        edge_cost = graph[current_node][next_node]
        running_cost += edge_cost
        print(f" {current_node} -> {next_node}: +{edge_cost} (total: {running_cost})")
else:
    print(f"Status: {message}")
    print(f"Cost: {total_cost}")

# Run the program
if __name__ == "__main__":
    main()

```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

● PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab> & C:/13.exe "e:/Sarfraz/4th SEMESTER/Artificial Intelligence/Lab/A* Search Algorithm By Sarfraz Alam
=====
A* Search Algorithm By Sarfraz Alam
=====
Available nodes in the graph:
Nodes: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

Graph connections with costs:
A: B(2), C(3)
B: A(2), C(1), D(4), E(2)
C: A(3), B(1), F(5)
D: B(4), E(1), G(3)
E: B(2), D(1), F(2), G(2)
F: C(5), E(2), H(1)
G: D(3), E(2), H(2)
H: F(1), G(2)

-----
Enter the source node (A-H): a
Enter the destination node (A-H): h
A* Search from A to H
Search Progress:
Node g(n)    h(n)    f(n)    Action
-----
A    0        8        0        Exploring
B    2        6        8        Exploring
E    4        3        7        Exploring
F    6        1        7        Exploring
H    7        0        7        Exploring
H    7        0        7        GOAL!

=====
A* SEARCH RESULTS
=====
Status: Optimal path found
Optimal path: A -> B -> E -> F -> H
Total optimal cost: 7
Number of nodes in path: 5

Detailed path costs:
A -> B: +2 (total: 2)
B -> E: +2 (total: 4)
E -> F: +2 (total: 6)
F -> H: +1 (total: 7)
PS E:\Sarfraz\4th SEMESTER\Artificial Intelligence\Lab>
```