

## AI LAB 3

What is Breadth-First Search (BFS)?

- BFS is a graph traversal algorithm that explores all the vertices of a graph or tree level by level before moving to the next depth.
- It is commonly used for shortest path finding in unweighted graphs, network broadcasting, AI pathfinding, and maze-solving.
- BFS uses a queue (FIFO – First In, First Out) to keep track of the nodes to be explored.
- It ensures that the shortest path (in terms of edges) is found in an unweighted graph, making it complete and optimal in such cases.

Explore more about BFS [here](#).

What is Depth-First Search (DFS)?

- DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.
- It is used in maze generation, solving puzzles, topological sorting, cycle detection, and finding connected components.
- DFS uses a stack (LIFO – Last In, First Out) or recursion to keep track of visited nodes.
- Unlike BFS, DFS does not guarantee the shortest path but is useful in scenarios where a solution needs to be found without concern for the shortest distance.

Read more about dfs [here](#) .

What is A-star Algorithm (A\*)?



- It is a searching algorithm that is used to find the shortest path between an initial and a final point
- It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A\* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It remains a widely popular algorithm for graph traversal.
- It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

You can read more about A\* [here](#) .

## **TASK :**

### Question 1 (Estimated Time:25min , Marks:15)

A 2D list is given Ali is at position (1,1). Help him to find HOME (4,4). But you have to suggest the shortest path. But there is one restriction, you are not allowed to move in diagonal. Write a function that uses the matrix as an input and return the shortest path.

|  |   |  |  |  |  |
|--|---|--|--|--|--|
|  |   |  |  |  |  |
|  |  |  |  |  |  |
|  |   |  |  |  |  |
|  |   |  |  |  |  |
|  |   |  |  |  |  |
|  |   |  |  |  |  |

### Function Skeleton for BFS:

```
def find_shortest_path(matrix):
    # Directions: Up, Down, Left, Right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Initialize starting and ending positions
    start = (0, 0)
    end = (3, 3)

    # Initialize data structures for BFS
    queue = None # Use deque for BFS
    visited = None # Track visited positions

    # BFS Loop
    while queue:
        # Dequeue the next position
        # Check if HOME is reached
        # Mark the position as visited
        # Explore neighboring cells (non-diagonal and not obstacles)
        Pass
    #Return the shortest path If found, else indicate no path
```

Return None

## Question 2 (Estimated Time:25min , Marks:15)

Implement Depth-First Search (DFS) to solve the 8-puzzle problem. The 8-puzzle consists of a 3x3 grid with 8 numbered tiles and one blank space. The goal is to rearrange the tiles to form a particular configuration. Your program should accept the initial configuration of the puzzle as input and output the steps required to reach the goal state.

Sample Output

Enter start State: 120345678

Enter goal State: 012345678

-----

DFS Algorithm

-----

Time taken: 0.0010004043579101562 seconds

Path Cost: 2

No of Node Visited: 3

-----

1 2 0

3 4 5

6 7 8

-----

1 0 2

3 4 5

6 7 8

-----

0 1 2

3 4 5

6 7 8

-----

## Skeleton Code for DFS:

```
import time
```

```
def state_to_tuple(state):
```

```
    """Convert a string state to a tuple representation."""
```

```

pass

def tuple_to_state(matrix):
    """Convert a tuple representation back to a string state."""
    pass

def get_moves(matrix):
    """Generate possible moves from the given state."""
    pass

def dfs(start_state, goal_state):
    """Perform Depth-First Search (DFS) to find a solution path."""
    pass

def main():
    """Main function to take input and execute the DFS algorithm."""
    start_state = input("Enter start State: ")
    goal_state = input("Enter goal State: ")

    start_tuple = state_to_tuple(start_state)
    goal_tuple = state_to_tuple(goal_state)

    print("-----")
    print("DFS Algorithm")
    print("-----")

    start_time = time.time()
    solution_path = dfs(start_tuple, goal_tuple)
    end_time = time.time()

    if solution_path:
        print("Time taken:", end_time - start_time, "seconds")
        print("Path Cost:", len(solution_path))
        print("No of Nodes Visited:", len(solution_path) + 1)
        for state in solution_path:
            for row in state:
                print(' '.join(row))
            print("-----")
    else:
        print("No solution found.")

```

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

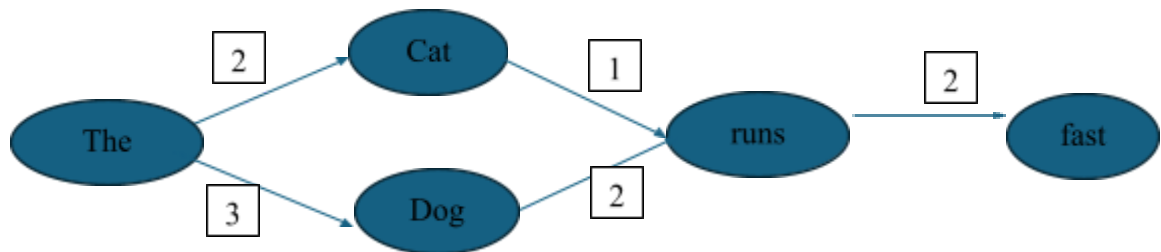
Question 3: (Estimated Time:30min , Marks:20)

Problem Statement: Sentence Completion using A\* Algorithm

You are developing an AI-powered autocomplete system for a search engine. When a user types an incomplete sentence, the system suggests the most coherent and relevant sentence completion.

You are given a **word graph**, where:

1. **Nodes** represent words.
2. **Edges** represent transitions between words, with a cost based on transition likelihood (**lower cost = more likely**).
3. A **heuristic function ( $h(n)$ )** estimates how relevant a word is in forming a meaningful sentence.



$h(n)$  is given as :

- $h(\text{The}) = 4$
- $h(\text{cat}) = 3$
- $h(\text{dog}) = 3$
- $h(\text{runs}) = 2$
- $h(\text{fast}) = 1$

Implement the *A\* algorithm* to determine the **best possible sentence completion** starting from "**The**" and ending at "**fast**". The algorithm should:

1. Use the given **heuristic function** to guide the search.
2. Compute the optimal **sentence completion path** and its **total cost**.

The output for this word graph given the heuristic function is:

Sentence: The cat runs fast

Total cost : 6

**Skeleton Code for A\* :**

```

from collections import deque

class Graph:
    def __init__(self, adjacency_list):
        """Initializes the graph with an adjacency list."""
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        """Returns the neighbors of a given node."""
        # Implement the function to return neighbors of node v
        pass

    def h(self, n):
        """Heuristic function: estimates the cost from node n to the goal."""
        H = { # define the heuristics here
        }
        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        """Implements the A* search algorithm to find the optimal path."""
        open_list = set([start_node])
        closed_list = set([])

        g = {} # Cost from start node to all other nodes
        g[start_node] = 0

        parents = {} # Keeps track of paths
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None # Current node

            # Find the node with the lowest f(n) = g(n) + h(n)

            if n == None:
                print("Path does not exist!")
                return None

            # If goal node is found, reconstruct and return the path
            #print the path found

```

```
        return reconst_path

    # Explore all neighbors of the current node

    # Update costs, parents, and move nodes between lists

    open_list.remove(n)
    closed_list.add(n)

    print("Path does not exist!")
    return None

# Define the graph given in adjacency list
adjacency_list = {
}

graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('startnode', 'endnode') # Define start and goal words
```