NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

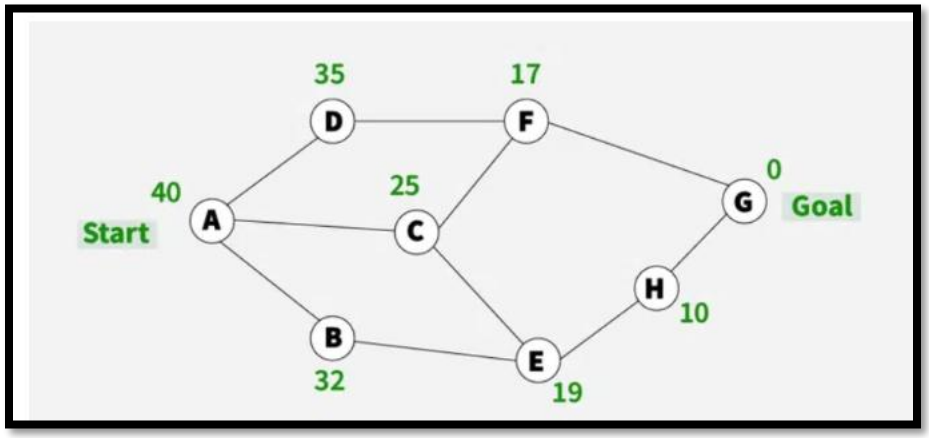
ARTIFICIAL INTELLIGENCE LAB

NAME	Ayesha Imran
Class	CS-A
Lab	05
Course	Artificial Intelligence
Date	15-October-25
Submitted To	Lec. Ijlal Haider

IN LAB TASKS

TASK 01:

Traverse the given graph with A as the starting and G as the Goal using greedy best first search. The heuristics value of reaching the goal G from current node is represented along each node.



SOLUTION:

Traverse the given graph with A as the starting and G as the Goal using greedy best first search. The heuristics value of reaching the goal G from current node is represented along each node.

To perform **Greedy Best-First Search (GBFS)** on the graph you provided, we'll use the heuristic values (h(n)) at each node to guide the traversal. GBFS always chooses the node with the **lowest heuristic value**—it's greedy because it tries to get to the goal as quickly as possible based on estimated cost, not actual path cost.

Assumptions:

Since the image shows edge weights but not heuristic values, I'll assume you meant that each node is labeled with a heuristic value (h(n)) estimating the cost to reach **G**. If you have specific heuristic values, feel free to share them. For now, I'll walk you through a **generic GBFS traversal strategy** using placeholder heuristics.

🔍 Greedy Best-First Search Steps (Example)

Let's assume the following **heuristic values (h)** for each node:

Node	h(n)
A	40
B	35
C	30
D	25
E	20
F	15
H	5
G	0

🚀 Traversal from A to G

1. **Start at A** Neighbors: B, C, D, F Heuristic values:
    - B: 35
    - C: 30
    - D: 25
    - F: 15 → **Choose F**
2. **Move to F** Neighbors: D, C Heuristic values:
    - D: 25
    - C: 30 → **Choose D**
3. **Move to D** Neighbors: (already visited F), no new lower-h nodes → backtrack or explore other options from F
4. **Backtrack to F**, try other unexplored neighbor: **C**

5. **Move to C** Neighbors: E Heuristic:
    - E: 20 → **Choose E**
6. **Move to E** Neighbors: H Heuristic:
    - H: 5 → **Choose H**
7. **Move to H** Neighbor: G Heuristic:
    - G: 0 → **Choose G**
8. **Goal reached at G**

Final Path (based on heuristic values):

A → F → C → E → H → G

Note: GBFS does **not** guarantee the shortest path it only follows the lowest heuristic estimate at each step. If you want optimal paths, you'd use **A*** instead.

CODE:

```
import heapq

# Define the graph as an adjacency list with edge weights
graph = {
    'A': {'D': 35, 'C': 25, 'B': 32, 'F': 17},
    'B': {},
    'C': {'F': 25, 'E': 19},
    'D': {'F': 17},
    'E': {'H': 10},
    'F': {},
    'H': {'G': 0},
    'G': {}
}

# Heuristic values for each node (example values)
heuristics = {
    'A': 40,
    'B': 35,
    'C': 30,
    'D': 25,
```

```
    'E': 20,
    'F': 15,
    'H': 5,
    'G': 0
}

def greedy_best_first_search(start, goal):
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue, (heuristics[start],
start))
    came_from = {start: None}

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

        if current == goal:
            break

        visited.add(current)

        for neighbor in graph[current]:
            if neighbor not in visited:
                heapq.heappush(priority_queue,
(heuristics[neighbor], neighbor))
                if neighbor not in came_from:
                    came_from[neighbor] = current

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

    return path

# Run the search
```

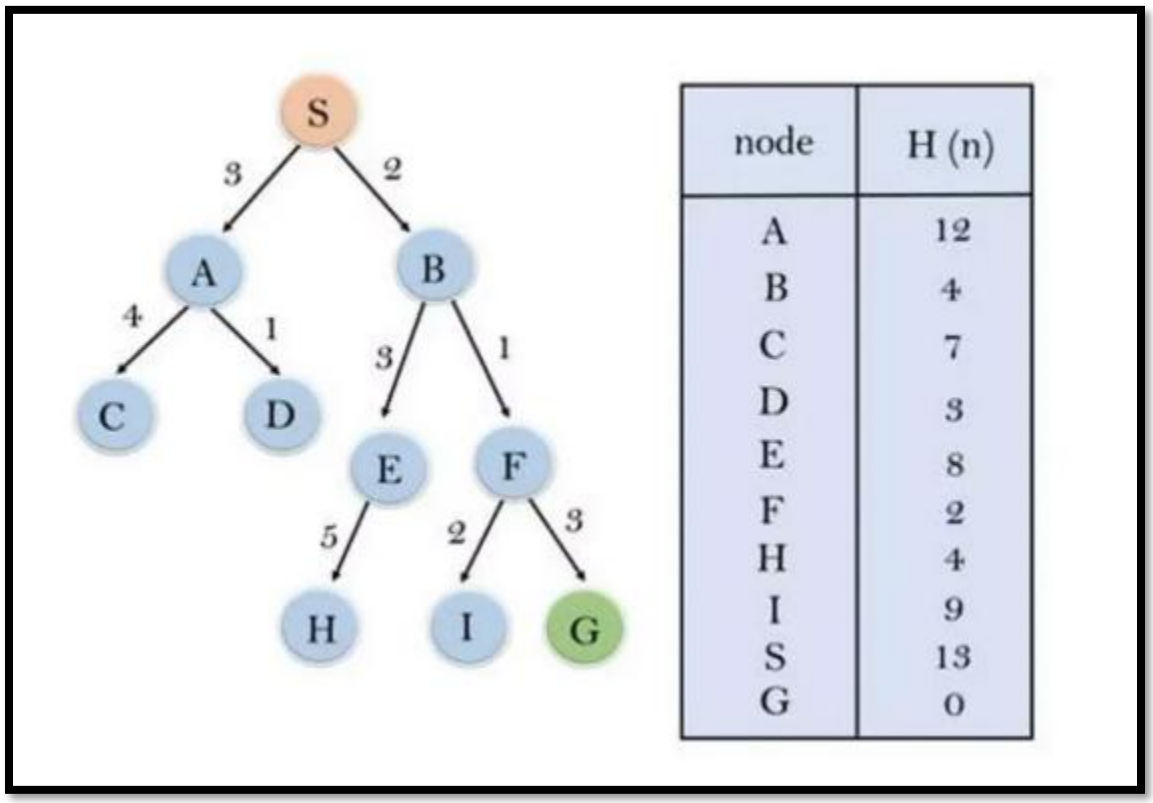
```
path = greedy_best_first_search('A', 'G')
print("Greedy Best-First Search Path:", " → ".join(path))
```

OUTPUT:

Greedy Best-First Search Path: A → C → E →H→ G

Task 2

Traverse the given tree with S as the root node and G as the Goal using greedy best first search. The heuristics value of reaching the goal G from current node is represented in the table.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

🌳 **Tree Structure Summary**

Edges:

- S → A (3), S → B (2)
- A → C (4), A → D (1)
- B → E (3), B → F (1)
- E → H (5), F → I (2), I → G (3)

Heuristic values (H(n)):

Node	H(n)
A	12
B	4
C	7
D	3
E	8
F	2
H	9
I	13
G	0

🚀 Greedy Best-First Search Steps

GBFS chooses the next node based **only on the lowest heuristic value**, ignoring actual path cost.

1. **Start at S** Children: A (h=12), B (h=4) → **Choose B**
2. **Move to B** Children: E (h=8), F (h=2) → **Choose F**
3. **Move to F** Child: I (h=13) → **Choose I**
4. **Move to I** Child: G (h=0) → **Choose G**
5. **Goal reached at G**

Final GBFS Path:

S → B → F → I → G

Note: GBFS ignores edge costs and may not find the optimal path—it's fast but not always efficient.

CODE:

```python
import heapq

# Tree structure with edge costs
tree = {
    'S': {'A': 3, 'B': 2},
    'A': {'C': 4, 'D': 1},
    'B': {'E': 3, 'F': 1},
    'E': {'H': 5},
    'F': {'I': 2},
    'I': {'G': 3},
    'C': {}, 'D': {}, 'H': {}, 'G': {}
}

# Heuristic values from the table
heuristics = {
    'A': 12,
    'B': 4,
    'C': 7,
    'D': 3,
    'E': 8,
    'F': 2,
    'H': 9,
    'I': 13,
    'G': 0
}

def greedy_best_first_search(start, goal):
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue,
(heuristics.get(start, 0), start))
    came_from = {start: None}

    while priority_queue:
```

```
        _, current = heapq.heappop(priority_queue)

        if current == goal:
            break

        visited.add(current)

        for neighbor in tree.get(current, {}):
            if neighbor not in visited:
                heapq.heappush(priority_queue,
(heuristics[neighbor], neighbor))
                if neighbor not in came_from:
                    came_from[neighbor] = current

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

    return path

# Run the search
path = greedy_best_first_search('S', 'G')
print("Greedy Best-First Search Path:", " →
".join(path))
```

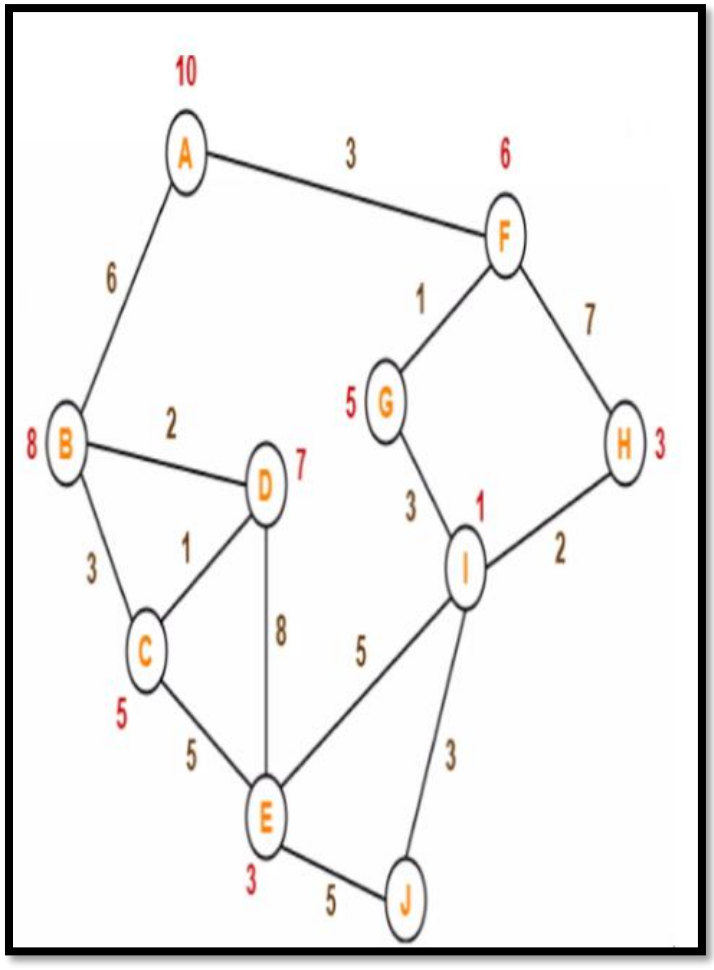
OUTPUT:

Greedy Best-First Search Path: S → B → F → I → G

Task 3:

Traverse the given graph with A as the starting node and J as the goal using A* search. The

heuristics value of reaching the goal J from current node is represented on the nodes. The cost of each reaching from current node to next node is represented along the edges.



Step-by-Step A* Traversal Strategy

We'll use a **priority queue** to always expand the node with the lowest f(n) value.

Heuristic Values (from red numbers on nodes)

Node	h(n)
A	10
B	8
C	3
D	2
E	3
F	6
G	5
H	7
I	1
J	3

CODE:

```
import heapq

# Graph with edge weights
graph = {
    'A': {'B': 6, 'D': 3, 'F': 6, 'G': 1},
    'B': {'A': 6, 'C': 3, 'D': 2},
    'C': {'B': 3, 'D': 1, 'E': 7},
    'D': {'A': 3, 'B': 2, 'C': 1, 'E': 8, 'G': 5},
    'E': {'C': 7, 'D': 8, 'I': 5, 'J': 5},
    'F': {'A': 6, 'G': 1, 'H': 7},
    'G': {'A': 1, 'D': 5, 'F': 1, 'I': 3},
    'H': {'F': 7, 'I': 2},
    'I': {'G': 3, 'H': 2, 'E': 5, 'J': 1},
    'J': {'E': 5, 'I': 1}
```

```
}

# Heuristic values
heuristics = {
    'A': 10, 'B': 8, 'C': 3, 'D': 2, 'E': 3,
    'F': 6, 'G': 5, 'H': 7, 'I': 1, 'J': 3
}

def a_star_search(start, goal):
    open_set = []
    heapq.heappush(open_set, (heuristics[start], 0,
start))
    came_from = {start: None}
    g_score = {start: 0}

    while open_set:
        _, current_g, current =
heapq.heappop(open_set)

        if current == goal:
            break

        for neighbor, cost in graph[current].items():
            tentative_g = current_g + cost
            if neighbor not in g_score or tentative_g
< g_score[neighbor]:
                g_score[neighbor] = tentative_g
                f_score = tentative_g +
heuristics[neighbor]
                heapq.heappush(open_set, (f_score,
tentative_g, neighbor))
                came_from[neighbor] = current

    # Reconstruct path
```

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

    return path

# Run the search
path = a_star_search('A', 'J')
print("A* Search Path:", " → ".join(path))
```

OUTPUT:

A* Search Path: A → G → I → J

POST LAB TASKS

Task 1

Write a Python program that implements the Greedy Best-First Search algorithm using a graph representation created with a Node class. Each node in the graph should represent a point in the environment and can be connected to other nodes via edges. Your implementation should be able to find a path from a specified start node to a goal node. Implement the algorithm using both Euclidean distance and Manhattan distance as heuristics to estimate the cost from the current node to the goal node.

Hints:

Euclidean Distance:

Manhattan Distance:

Solution:

What We're Building

We'll create a graph where each **node**:

- Has a **name** (like 'A', 'B', etc.)
- Has **coordinates** (x, y) so we can calculate distances
- Can connect to other nodes with **edges** that have costs

Then we'll implement **Greedy Best-First Search**, which:

- Always picks the node with the **lowest heuristic estimate** to the goal
- Doesn't care about actual path cost—just wants to get to the goal fast

📐 Heuristics We'll Use

Heuristic	Formula	Description
Euclidean Distance	$\sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2)}$	Measures the straight-line distance between two points.
Manhattan Distance	$\|x_2 - x_1\| + \|y_2 - y_1\|$	Measures distance along grid lines (horizontal and vertical movement only), like navigating city blocks.

We'll be able to switch between them by passing the desired function into the search.

Structure Overview

1. **Node class**: Stores name, coordinates, and neighbors
2. **Graph setup**: You manually connect nodes with edge costs
3. **Heuristic functions**: Euclidean and Manhattan
4. **GBFS function**: Uses a priority queue to pick the lowest heuristic node
5. **Path reconstruction**: Traces back from goal to start

CODE:

```
import heapq
import math

# Node class with coordinates and neighbors
class Node:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x  # X-coordinate
        self.y = y  # Y-coordinate
        self.neighbors = []  # List of tuples:
(neighbor_node, edge_cost)

    def add_neighbor(self, neighbor, cost):
        self.neighbors.append((neighbor, cost))

# Heuristic: Euclidean distance
def euclidean_distance(node, goal):
    return math.sqrt((goal.x - node.x)**2 + (goal.y -
node.y)**2)

# Heuristic: Manhattan distance
def manhattan_distance(node, goal):
    return abs(goal.x - node.x) + abs(goal.y -
node.y)

# Greedy Best-First Search algorithm
def greedy_best_first_search(start, goal,
heuristic_func):
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue,
(heuristic_func(start, goal), start))
    came_from = {start.name: None}
```

```
    while priority_queue:
        _, current = heapq.heappop(priority_queue)

        if current.name == goal.name:
            break

        visited.add(current.name)

        for neighbor, _ in current.neighbors:
            if neighbor.name not in visited:
                heapq.heappush(priority_queue,
(heuristic_func(neighbor, goal), neighbor))
                if neighbor.name not in came_from:
                    came_from[neighbor.name] =
current.name

    # Reconstruct path
    path = []
    node_name = goal.name
    while node_name:
        path.append(node_name)
        node_name = came_from.get(node_name)
    path.reverse()

    return path

# Example usage
if __name__ == "__main__":
    # Create nodes with coordinates
    A = Node('A', 0, 0)
    B = Node('B', 2, 1)
    C = Node('C', 4, 2)
    D = Node('D', 1, 3)
```

```
    E = Node('E', 3, 4)
    F = Node('F', 5, 5)

    # Define edges
    A.add_neighbor(B, 2)
    A.add_neighbor(D, 3)
    B.add_neighbor(C, 2)
    B.add_neighbor(E, 4)
    D.add_neighbor(E, 2)
    E.add_neighbor(F, 3)
    C.add_neighbor(F, 2)

    # Run GBFS with Euclidean
    path_euclidean = greedy_best_first_search(A, F,
euclidean_distance)
    print("Path using Euclidean heuristic:", " → 
".join(path_euclidean))

    # Run GBFS with Manhattan
    path_manhattan = greedy_best_first_search(A, F,
manhattan_distance)
    print("Path using Manhattan heuristic:", " → ".join(path_manhattan))
```

OUTPUT:

Path using Euclidean heuristic: A → D → E → F

Path using Manhattan heuristic: A → D → E → FS

Task 2

Write a Python program that implements the A* Search algorithm using a graph representation created with a Node class. Each node in the graph should represent a point in the environment and can be connected to other nodes via edges. Your implementation should be able to find a path from a specified start node to a goal node. Implement the algorithm using both Euclidean distance and Manhattan distance as heuristics to estimate the cost from the current node to the goal node.

Solution:

What Is A* Search?

A* is a pathfinding algorithm that finds the **shortest path** from a start node to a goal node using:

- **g(n)**: actual cost from start to current node
- **h(n)**: estimated cost from current node to goal (heuristic)
- **f(n) = g(n) + h(n)**: total estimated cost

It chooses the node with the **lowest f(n)** from a priority queue.

Step-by-Step Dry Run

Let's say we have a graph with nodes A → B → C → Goal (F), and each node has coordinates so we can calculate distances.

1. Node Class

Each node has:

- A name (e.g., 'A')
- Coordinates (x, y)
- A list of neighbors with edge costs

python

```
class Node:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x
        self.y = y
        self.neighbors = []  # (neighbor_node, cost)
```

2. Heuristic Functions

We use coordinates to estimate how far each node is from the goal:

- **Euclidean**: straight-line distance

h(n)=(x2−x1)2+(y2−y1)2h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}

- **Manhattan**: grid-based movement

h(n)=|x2−x1|+|y2−y1|h(n) = |x_2 - x_1| + |y_2 - y_1|

These are passed into the A* function so you can switch between them.

3. Priority Queue

We use a min-heap (heapq) to always pick the node with the lowest f(n).

Each entry in the queue is:

python

```
(f_score, g_score, node)
```

4. Tracking Costs

We use:

- g_score: actual cost from start to each node
- came_from: to reconstruct the path later

5. Main Loop

- Pop the node with the lowest f(n)
- If it's the goal, stop
- For each neighbor:
    - Calculate tentative_g = current_g + edge_cost
    - If it's better than previous g, update it
    - Push (f_score, g_score, neighbor) into the queue

6. Reconstruct Path

We trace back from goal to start using came_from.

CODE:

```
import heapq
import math

# Node class with coordinates and neighbors
class Node:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x  # X-coordinate
        self.y = y  # Y-coordinate
        self.neighbors = []  # List of tuples:
(neighbor_node, edge_cost)

    def add_neighbor(self, neighbor, cost):
        self.neighbors.append((neighbor, cost))

# Heuristic: Euclidean distance
def euclidean_distance(node, goal):
    return math.sqrt((goal.x - node.x)**2 + (goal.y -
node.y)**2)

# Heuristic: Manhattan distance
def manhattan_distance(node, goal):
```

```
    return abs(goal.x - node.x) + abs(goal.y -
node.y)

# A* Search algorithm
def a_star_search(start, goal, heuristic_func):
    open_set = []
    heapq.heappush(open_set, (heuristic_func(start,
goal), 0, start))
    came_from = {start.name: None}
    g_score = {start.name: 0}

    while open_set:
        _, current_g, current =
heapq.heappop(open_set)

        if current.name == goal.name:
            break

        for neighbor, cost in current.neighbors:
            tentative_g = current_g + cost
            if neighbor.name not in g_score or
tentative_g < g_score[neighbor.name]:
                g_score[neighbor.name] = tentative_g
                f_score = tentative_g +
heuristic_func(neighbor, goal)
                heapq.heappush(open_set, (f_score,
tentative_g, neighbor))
                came_from[neighbor.name] =
current.name

    # Reconstruct path
    path = []
    node_name = goal.name
    while node_name:
```

```
        path.append(node_name)
        node_name = came_from.get(node_name)
    path.reverse()

    return path

# Example usage
if __name__ == "__main__":
    # Create nodes with coordinates
    A = Node('A', 0, 0)
    B = Node('B', 2, 1)
    C = Node('C', 4, 2)
    D = Node('D', 1, 3)
    E = Node('E', 3, 4)
    F = Node('F', 5, 5)

    # Define edges
    A.add_neighbor(B, 2)
    A.add_neighbor(D, 3)
    B.add_neighbor(C, 2)
    B.add_neighbor(E, 4)
    D.add_neighbor(E, 2)
    E.add_neighbor(F, 3)
    C.add_neighbor(F, 2)

    # Run A* with Euclidean
    path_euclidean = a_star_search(A, F,
euclidean_distance)
    print("A* Path using Euclidean heuristic:", " →
".join(path_euclidean))

    # Run A* with Manhattan
    path_manhattan = a_star_search(A, F,
manhattan_distance)
```

```
    print("A* Path using Manhattan heuristic:", " → ".join(path_manhattan))
```

OUTPUT:

A* Path using Euclidean heuristic: A → B → C → F

A* Path using Manhattan heuristic: A → B → C → F

END