FORMATION BASED UAV-PATH PLANNING

MID-EVAL

Graph?

WHAT IS A GRAPH?

- A graph is a data structure that consists of a set of nodes (vertices) and edges connecting them.
- Vertices (nodes): Represent entities (e.g., cities, computers).
- Edges (links): Represent connections or relationships between vertices

TYPES OF GRAPH

- **Undirected** graph: edges have no direction
- **Directed** graph: edges have direction.
- **Unweighted** graph: edges treated as cost 1.
- **Weighted** graph: edges have different cost.

Why only Graph?

- It contains loops like real- world systems.
- Makes complex systems understandable, navigable and solvable.
- Represents dynamic connections or multiple paths.

Uses of Graphs

Navigation (maps)

Cities as nodes, roads as edges–used in GPS and route optimization.

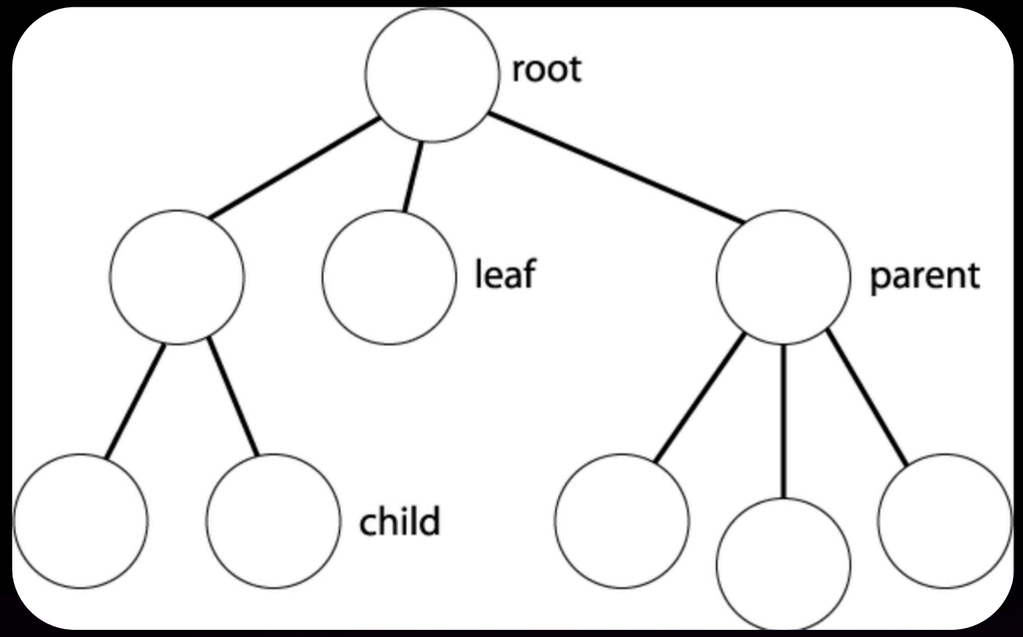
Social Media

USERS AS NODES, FRIENDSHIPS/FOLLOWS AS EDGES–USED IN FRIEND SUGGESTIONS

Network

Devices as nodes, connections as edges –used in routing and data flow

Tree ?



- It is a special graph which has no cycles, no loops.
- If it has N nodes then it has N-1 edges.
- Root: Starting point.
- Child: Directly connected to parent.
- Leaf: Nodes with no children.

Graph Traversal Algorithms

SIMPLE GRAPH TRAVERSAL ALGORITHM

- BREADTH FIRST SEARCH(BFS)
- DEPTH FIRST SEARCH(DFS)

ADVANCED / INFORMED GRAPH SEARCH ALGORITHMS

- DIJKSTRA'S ALGORITHM
  (COST-BASED SHORTEST PATH)
- A* SEARCH ALGORITHM
  (HEURISTIC-BASED OPTIMAL SEARCH)

DEPTH FIRST SEARCH(DFS)

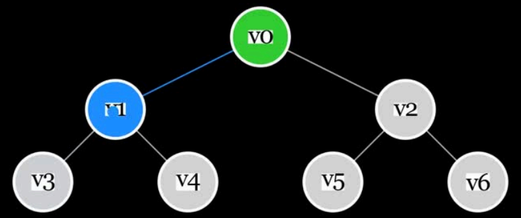
WHAT IS DFS?

- Depth-First Search (DFS) is a graph/tree traversal algorithm
- DFS explores nodes by going as deep as possible along a path before backtracking.
- It uses a stack (or recursion) and is useful for path finding, cycle detection, and connectivity checks.

DEPTH FIRST SEARCH(DFS)

HOW DFS WORKS?

DFS starts from a node and explores as deep as possible along each branch before backtracking to explore other paths.



- Starts at the root node.
- Visit one adjacent node and keep going deeper
- Backtrack when no unvisited nodes remain
- Continue until all nodes are visited

DEPTH FIRST SEARCH(DFS)

APPLICATIONS

- Path finding and maze solving
- Cycle detection in graphs
- Topological sorting in DAGs
- Finding connected components
- Used in backtracking problems (e.g., puzzles)

CODE

```python
def dfs(graph, start, visited=None, order=None):
    if visited is None:
        visited = set()
    if order is None:
        order = []

    visited.add(start)
    order.append(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited, order)

    return order
```

BREADTH FIRST SEARCH(BFS)

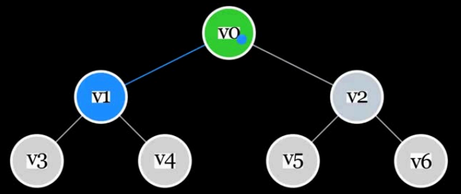
WHAT IS BFS?

- Breadth-First Search (BFS) is a graph/tree traversal algorithm
- BFS explores a graph level by level, visiting all neighbors of a node before moving deeper.
- Uses a queue data structure
- Guarantees the shortest path in unweighted graphs
- Commonly used in level-order traversal and connectivity problems

BREADTH FIRST SEARCH(BFS)

HOW BFS WORKS?

- Start from the source node, mark it as visited, and insert it into a queue
- Remove the front node from the queue and visit all its unvisited adjacent nodes
- Mark each adjacent node as visited and add them to the queue
- Repeat this process, ensuring nodes are processed level by level, until the queue becomes empty and all reachable nodes are visited



BREADTH FIRST SEARCH(BFS)

APPLICATIONS

- Finding shortest path in unweighted graphs
- Level-order traversal of trees
- Web crawling and social network analysis
- Finding minimum moves in games and puzzles
- Network broadcasting and routing

CODE

```
def bfs(graph, start):
    visited = set()
    order = []
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            order.append(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

    return order
```

Dijkstra’s Algorithm

Why Dijkstra?

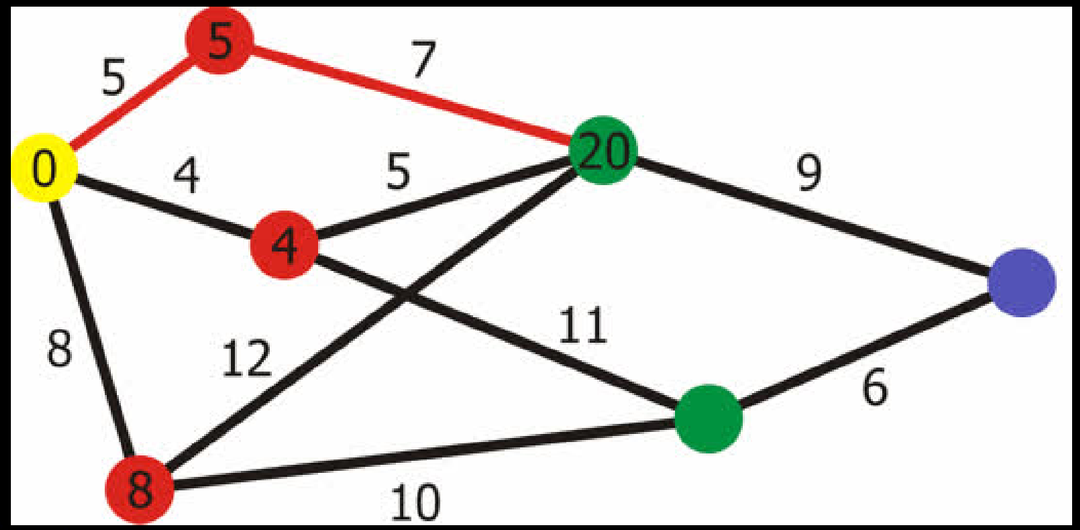
BFS works only when all edges have equal cost, but real-world graphs often have weighted edges such as road lengths or tolls. We need an algorithm to find the shortest path in weighted graphs.

Key Idea

Dijkstra’s algorithm repeatedly selects the unvisited node with the smallest distance from the source and updates its neighboring edges. Once selected, the node’s shortest distance is finalized.

Algorithm

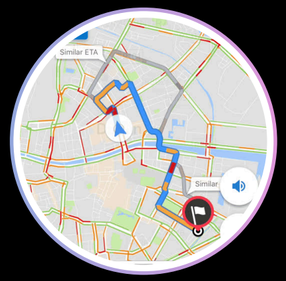
- **Input**: Takes a weighted graph with **non-negative edge weights** and a source vertex
- Initialize distance of the source to 0 and all other vertices to ∞
- Use a **priority queue** to always select the vertex with minimum distance
- Repeatedly:
  - Extract the closest unvisited vertex
  - Relax edges to update distances of its neighbors
- **Output**: Computes the shortest distances (and paths) from the source to all vertices



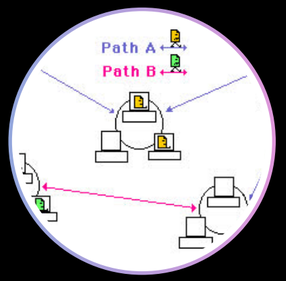
Algorihtm Code

```python
import heapq

def dijkstra(graph, source, target):
    dist = {node: float('inf') for node in graph}
    dist[source] = 0
    pq = [(0, source)]
    visited = set()

    while pq:
        curr_dist, u = heapq.heappop(pq)
        if curr_dist > dist[u]:
            continue
        if u in visited:
            continue
        visited.add(u)
        if u == target:
            return dist[u]

        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))

    return float('inf')
```

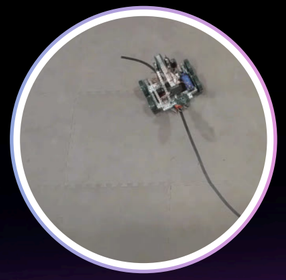
Applications



GPS / Google Maps routing



Network routing protocols



Robotics Navigation

Game Pathfinding

A* Search Algorithm

- A* is an informed **best-first search algorithm**
  Level-order traversal of trees.
- It is a variant of Dijkstra's algorithm.
- The only difference is that an evaluation function is used to
  determine which node to explore next.
- The evaluation function, f(x) = g(x) + h(x), is the estimated
  total cost of a path going through Node x.
  - g(x): Cost of the best path discovered so far from the start node to node x
  - h(x): Heuristic estimate of the cheapest cost from Node x to the goal

Note: to gain a correct result, f(x) should never overestimates the cost to arrive at the goal node, i.e. h(x) must be less than the actual optimal cost to reach the goal

The Algorithm

Two Lists are used:

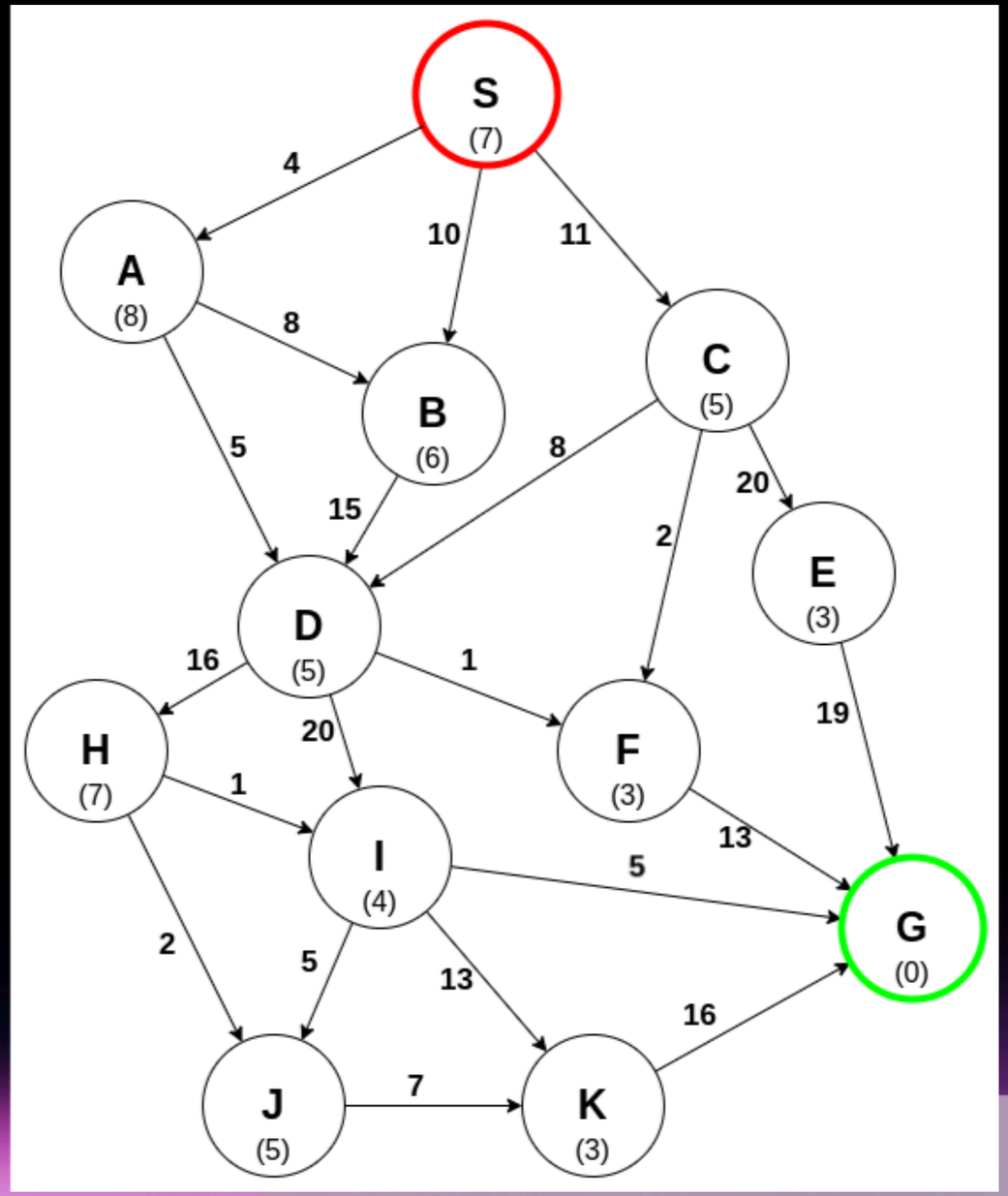
- An **open list**, implemented as a priority queue, which stores the next nodes to be explored. Initially, the only node in this list is the start node S.
- A closed list which stores the nodes that have already been evaluated. When a node is in the closed list, it means that the lowest-cost path to that node has been found.

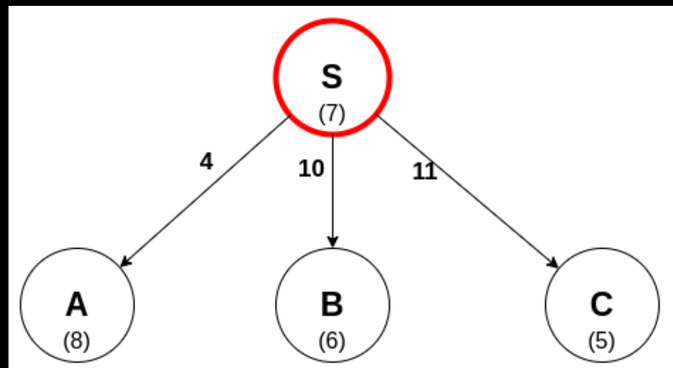
Steps:

1. Initialize a tree with the root node being the start node **S**.
2. Remove the top node from the open list for exploration.
3. Add the current node to the closed list.
4. Add all nodes that have an incoming edge from the current node as child nodes in the tree.
5. Update the lowest cost to reach the child node.
6. Compute the evaluation function for every child node and add them to the open list.

All nodes except for the start node start with a lowest cost of infinity. The start node has an initial lowest cost of 0.

Note: The algorithm ends when the goal node **G** has been explored, NOT when it is added to the open list.

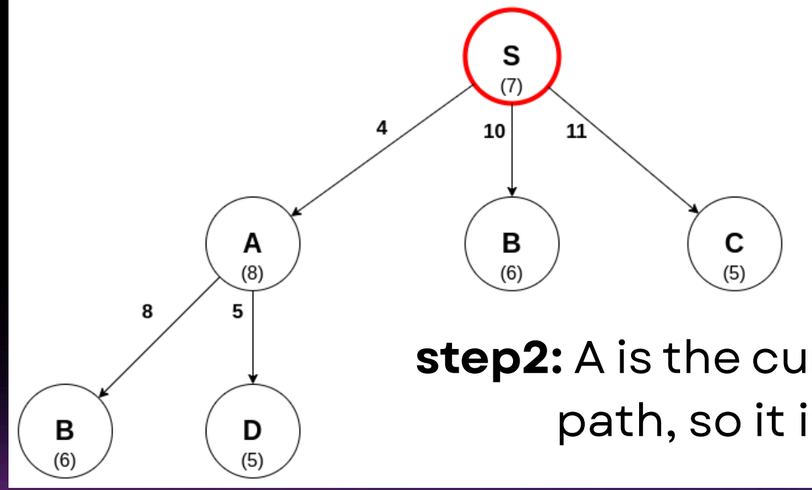




Node[cost]
A[12]
B[16]
C[16]

Closed List
S

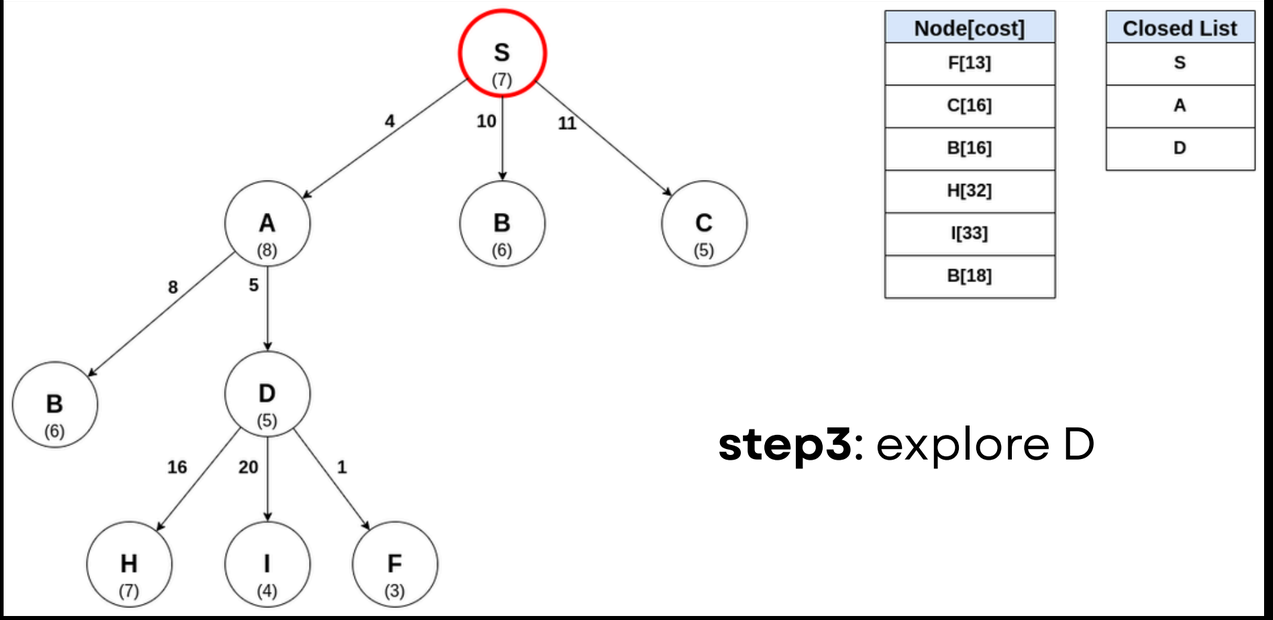
step1: explore S



Node[cost]
D[14]
C[16]
B[16]
B[18]

Closed List
S
A

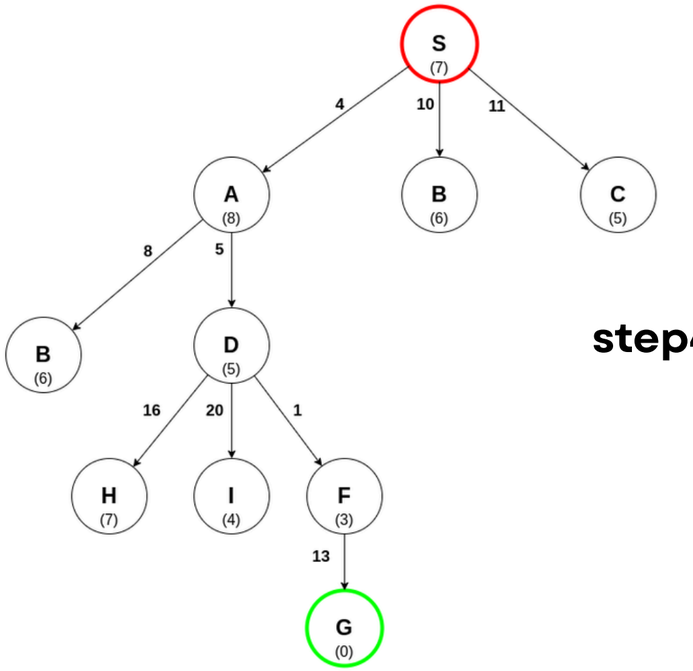
step2: A is the current most promising path, so it is explored next



Node[cost]
F[13]
C[16]
B[16]
H[32]
I[33]
B[18]

Closed List
S
A
D

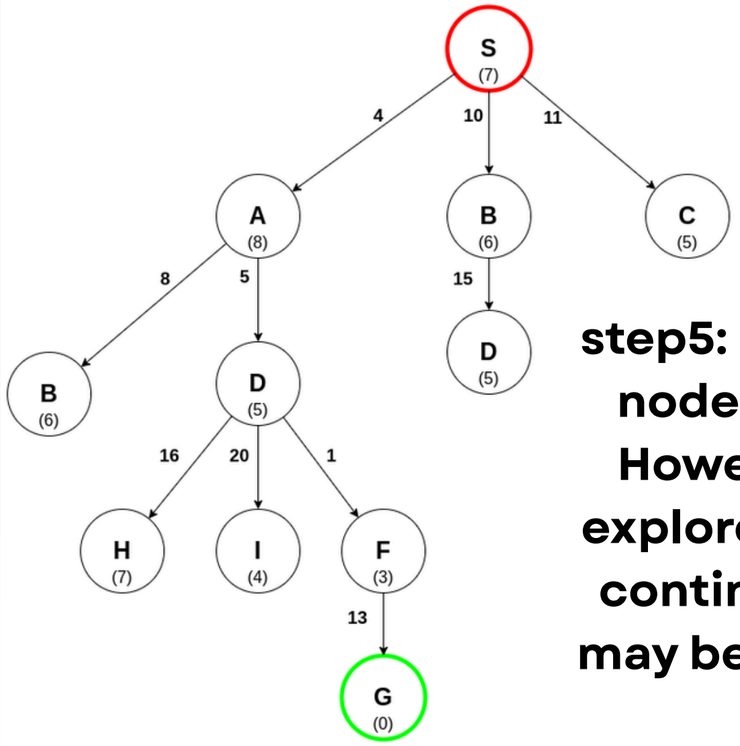
step3: explore D



Node[cost]
B[16]
C[16]
B[18]
H[32]
I[33]
G[23]

Closed List
S
A
D
F

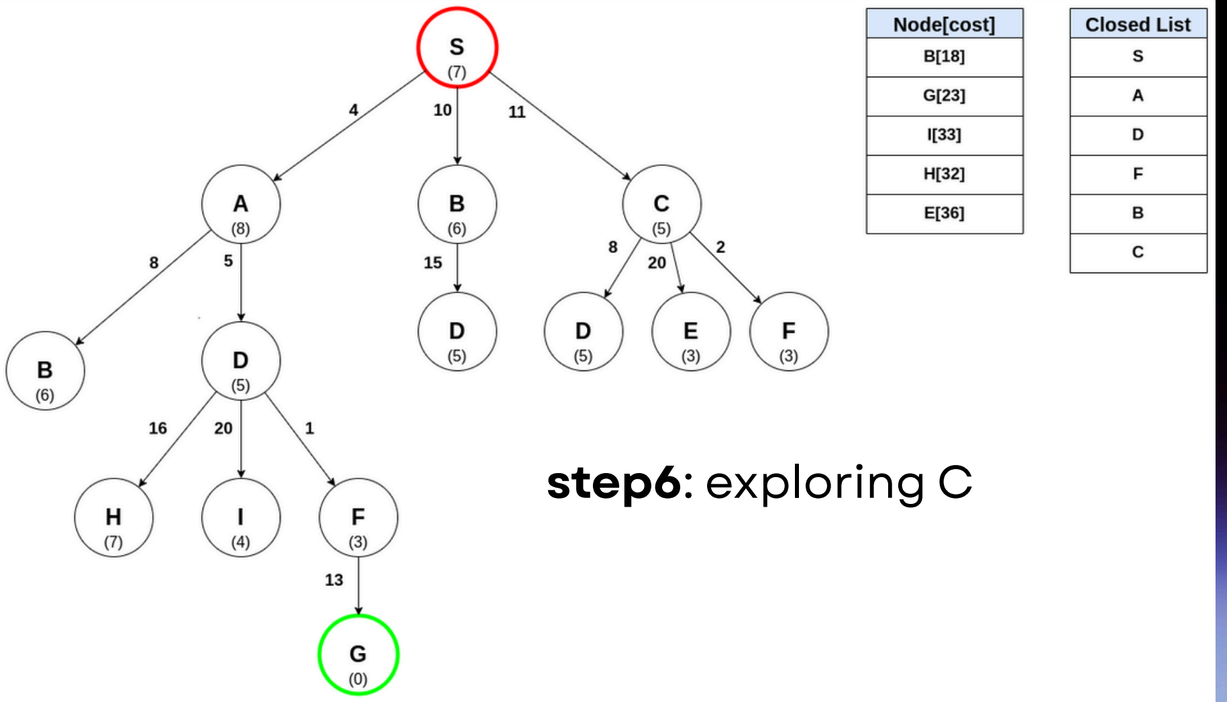
step4: explore F



Node[cost]
C[16]
G[23]
B[18]
H[32]
I[33]

Closed List
S
A
D
F
B

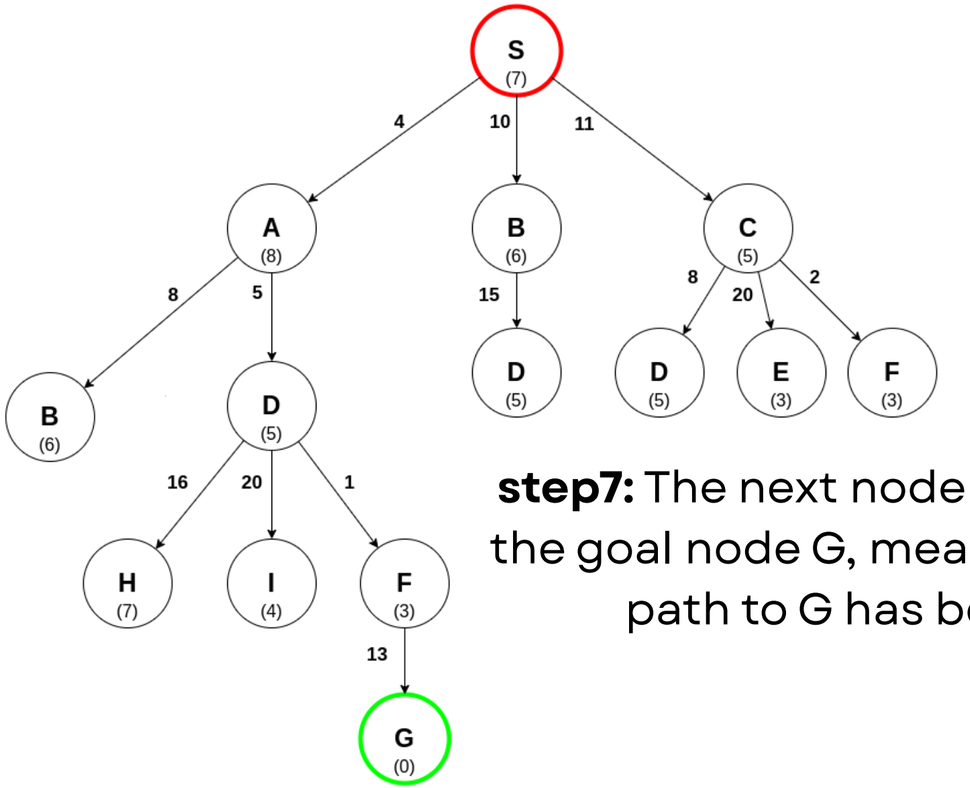
step5: Notice that the goal node G has been found. However, it hasn't been explored, so the algorithm continues because there may be a shorter path to G. Eploring B



Node[cost]
B[18]
G[23]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C

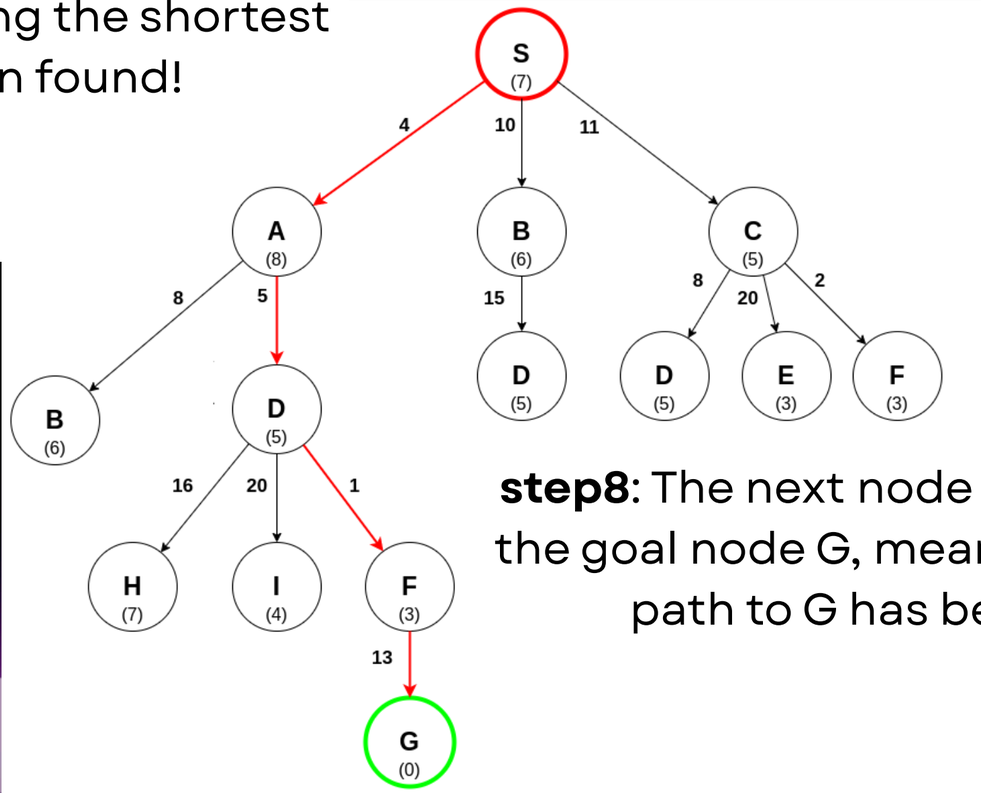
step6: exploring C



Node[cost]
G[23]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C

step7: The next node to be explored is the goal node G, meaning the shortest path to G has been found!



Node[cost]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C
G

step8: The next node to be explored is the goal node G, meaning the shortest path to G has been found!

```python
from heapq import heappop, heappush

def a_star_search(graph: dict, start: str, goal: str, heuristic_values: dict) -> int:

    open_list, closed_list = [(heuristic_values[start], start)], set()

    while open_list:
        cost, node = heappop(open_list)
        if node == goal:
            return cost

        if node in closed_list:
            continue

        closed_list.add(node)

        cost -= heuristic_values[node]

        for neighbor, edge_cost in graph[node]:
            if neighbor in closed_list:
                continue
            neighbor_cost = cost + edge_cost + heuristic_values[neighbor]
            heappush(open_list, (neighbor_cost, neighbor))

    return -1
```

THANK YOU!

Members

1. Rajkumar Ahirwar
2. Aditya Kumar
3. Ahmad Akthar
4. Divya Mina