Formation Based UAV Path Planning

GROUP 5

HEMA
KANAK
PONISH BHATIA
SANATH KOUNDINYA
TANISH GUPTA

NUMPY

NumPy (Numerical Python) is a Python library used for fast numerical and mathematical operations, with arrays and matrices
we can create arrays and matrices of different shapes
we can also create arrays filled with certain numbers:

np.zeros() ==> Creates an array of zeros
np.ones() ==> Creates an array of ones
np.eye() ==>Creates an identity matrix
np.full() ==>Creates an array with the same value
np.arange() ==>Creates an array with a range of numbers
np.linspace() ==> creates an array of evenly spaced numbers
np.random.randint() ==> Creates an array of random integers

importing numpy as np , we can perform various functions :
np.array() ==> create array
np.zeros() ==> array of zeros
np.ones() ==> array of ones
np.arange() ==> range array
np.reshape() ==> change shape

```
import numpy as np
o = np.ones((3, 2))
print(o)
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

```
b= np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
b.shape
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
(2, 2, 2)
```

arithmetics operation using numpy

code

```
import numpy as np
A = np.array([[2, 4, 6],
              [8, 10, 12],
              [14, 16, 18]])

B = np.array([[1, 3, 5],
              [7, 9, 11],
              [13, 15, 17]])


print("\nAddition (A + B):\n", A + B)
print("Subtraction (A - B):\n", A - B)
print("Multiplication (A * B):\n", A * B)
print("Division (A / B):\n", A / B)
print("Power (A ** 2):\n", A ** 2)

print("\nDot Product (A @ B):\n", A @ B)
```

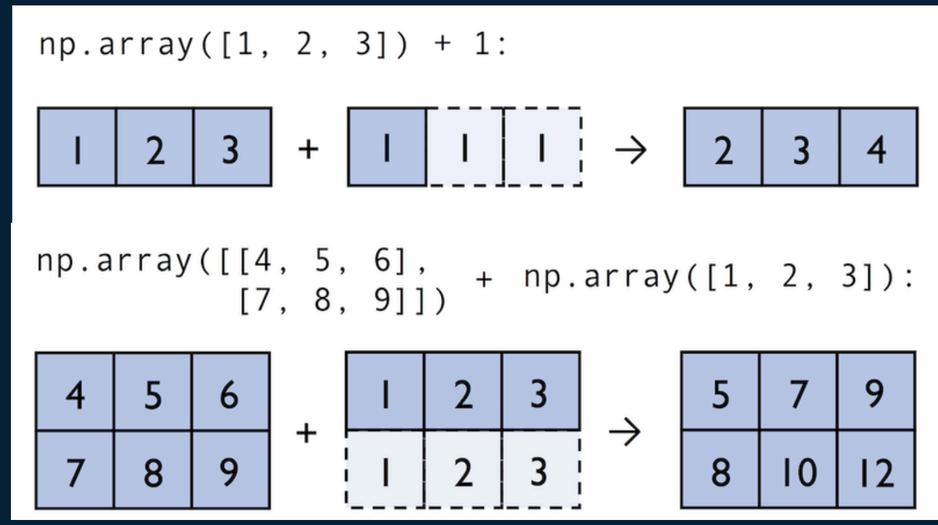
output

```
Addition (A + B):
 [[ 3  7 11]
 [15 19 23]
 [27 31 35]]
Subtraction (A - B):
 [[1 1 1]
 [1 1 1]
 [1 1 1]]
Multiplication (A * B):
 [[  2  12  30]
 [ 56  90 132]
 [182 240 306]]
Division (A / B):
 [[2.         1.33333333 1.2       ]
 [1.14285714 1.11111111 1.09090909]
 [1.07692308 1.06666667 1.05882353]]
Power (A ** 2):
 [[  4  16  36]
 [ 64 100 144]
 [196 256 324]]

Dot Product (A @ B):
 [[108 132 156]
 [234 294 354]
 [360 456 552]]
```

Broadcasting

Broadcasting is a feature in NumPy that allows arrays of different shapes to be combined in arithmetic operations automatically by stretching”the smaller array along the larger one without copying data.
Broadcasting two arrays together follows these rules:
Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape
Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised




```
#Adding a scalar to a 1D array
C = np.array([1, 2, 3])
result1 = C + 1
print(result1)
```

```
[2 3 4]
```

```
import numpy as np
# Adding a 2x3 array to a 1x3 array
# broadcasting along rows
A = np.array([[4, 5, 6],
              [7, 8, 9]])
B = np.array([1, 2, 3])
result2 = A + B
print(result2)
```

```
[[ 5  7  9]
 [ 8 10 12]]
```

Graphs

- Nodes/Vertices
- Edges (Links connecting Nodes)

Undirected Graphs

No direction
Connections on LinkedIn

Directed Graphs

Directed Edges
Followers on Instagram

Graph Traversal

Visiting all Nodes
Checking if two graphs are connected
Searching for a Particular Node

DFS

Depth-first Search

Go as deep as possible before backtracking

Implementation:
Recursion or Explicit Stack

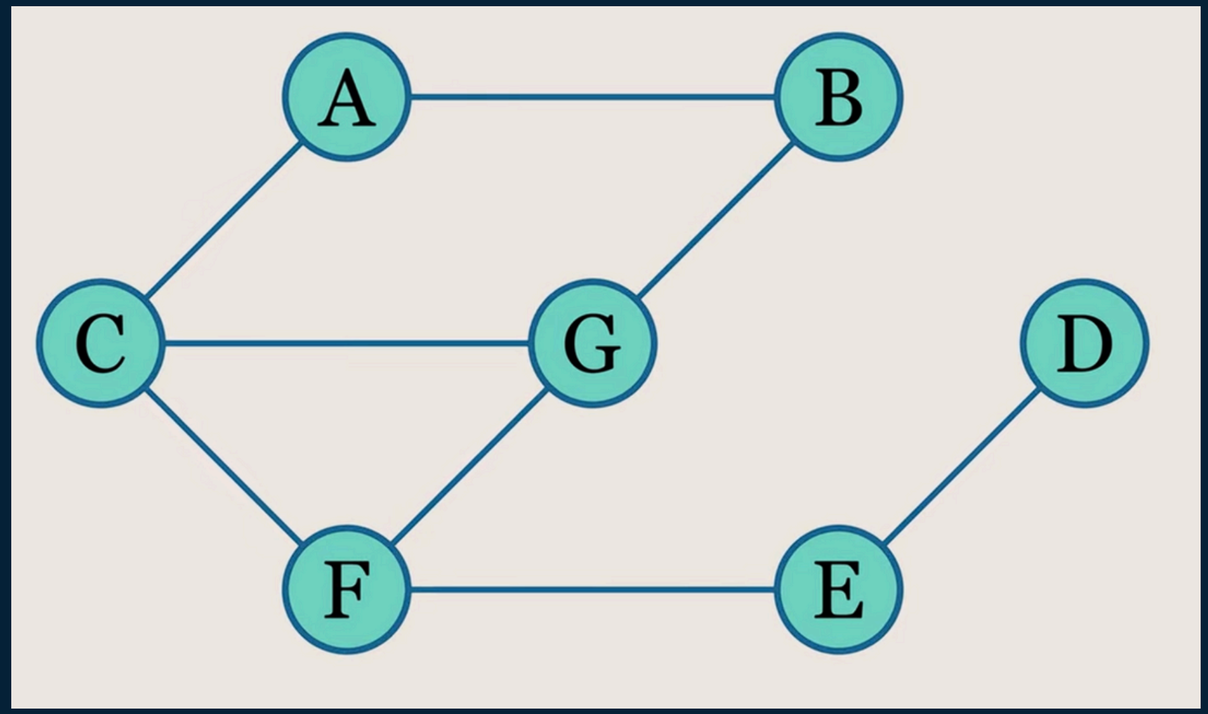
BFS

Breath-first Search

Explore level-by-level

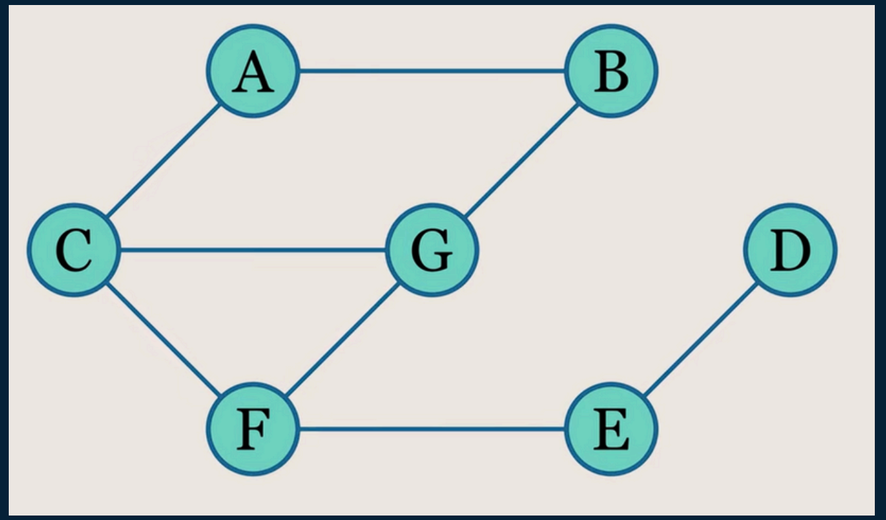
Implementation: Queue

DFS




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

Recursive Approach

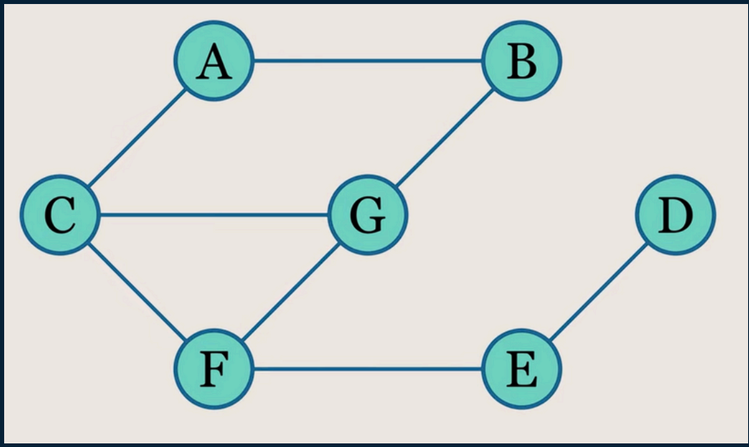



```
def DFS(graph, vertex, visited):
    visited.add(vertex)
    print(vertex, end=" ")
    for neighbor in graph[vertex]:
        if neighbor not in visited:
            DFS(graph, neighbor, visited)
```

Iterative Approach

```
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex, end=" ")
            for neighbor in reversed(graph[vertex]):
                if neighbor not in visited:
                    stack.append(neighbor)
```



Grid Pathfinding Algorithms

What is grid pathfinding?

Grid pathfinding is the process of finding an optimal route between a start cell and a goal cell on a grid-based map. The grid is typically divided into square cells, where each cell represents a traversable space or an obstacle.

- The environment is modeled as a 2D grid.
- Each cell can be free or blocked (obstacle).
- Movement is allowed in predefined directions (e.g., 4-directional or 8-directional).
- Each move may have an associated cost (uniform or weighted).

What is grid pathfinding?

Grid pathfinding provides a simple yet powerful abstraction for real-world navigation problems.

It is widely used in robotics, games, AI planning, maze solving, and autonomous navigation.

Shortest-path problems in constrained environments

Dijkstra’s Algorithm

Introduction:

- It's a grid pathfinding algorithm used to find the **shortest path** from a starting node to all other nodes.on a graph
- Works on weighted graphs where all edges weights are non negative.

Commonly used in

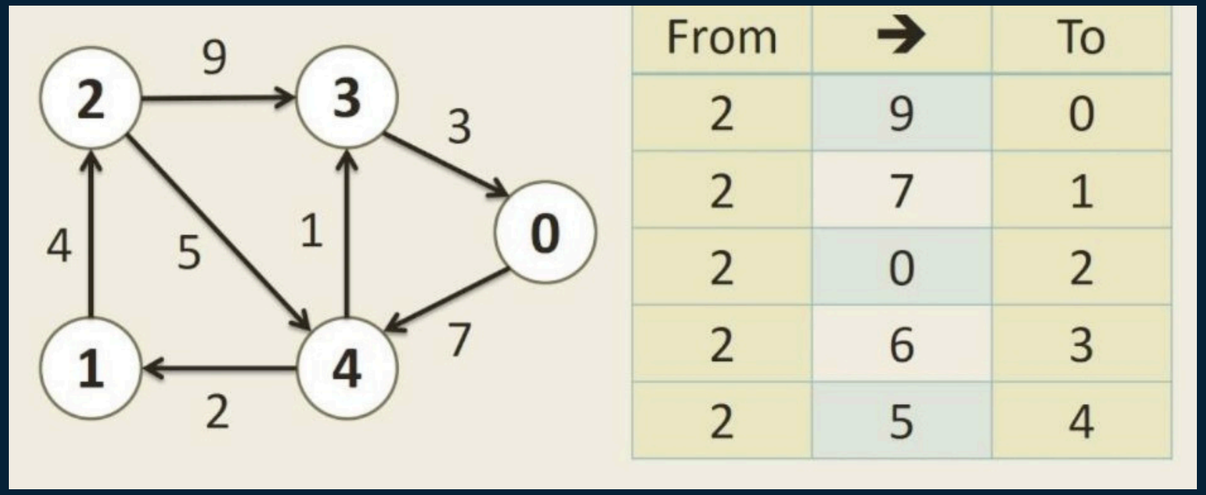
- GPS navigation system
- Network routing
- Maps and pathfinding problems

How Dijkstra's Algorithm works :

- Start with distance 0 at source .
- Repeatedly pick the closest visited node.
- Update(relax) distance to its neighbours.
- This is like BFS but using a **priority queue** instead of a normal queue.

👉 The algorithm always chooses the **shortest available path first.**

Example:



From	→	To
2	9	0
2	7	1
2	0	2
2	6	3
2	5	4

Why BFS and Dijkstra are not enough

- Guarantee correctness, but they become inefficient as problem size and complexity increase

Limitations of BFS:

- Assumes uniform edge costs; cannot handle weighted movement efficiently.
- Explores the grid uniformly in all directions, regardless of where the goal is.
- Results in excessive node expansion, especially on large grids.
- Computationally expensive in both time and memory for real-world map

Why BFS and Dijkstra are not enough

Limitations of Dijkstra's Algorithm:

- Handles weighted costs but still performs uninformed search.
- Expands nodes based solely on distance from the start, ignoring the goal's location.
- Explores many unnecessary nodes far away from the optimal path.
- Becomes slow for large-scale environments such as dense grids or maps with many obstacles.

Both BFS and Dijkstra treat all directions as equally promising.
They lack goal awareness, leading to wasted computation.

A* Algorithm

A* is a best-first search guided by f(n):

$$f(n) = g(n) + h(n)$$

Where:
g(n) = cost from start to n
h(n) = estimated cost from n to goal
f (n) = estimated total path cost

Motivation for A*

Real-world navigation requires algorithms that are both optimal and efficient.

A* introduces a heuristic function that estimates the remaining cost to the goal.

This heuristic guides the search toward the goal, significantly reducing exploration.

Motivation for A*

Conclusion:

- BFS and Dijkstra are correct but not scalable.
- A* improves performance by combining path cost so far with intelligent goal-directed search, making it better suited for practical applications.
- g(n) ensures correctness
- h(n) provides direction
- OPEN enables best-first search
- CLOSED avoids repeated work

With an admissible heuristic, A* is optimal.

Approximation Heuristics

A heuristic h(n) is an approximation of the remaining cost.

- Fast to compute
- Problem-specific
- Not necessarily exact

Distance-based heuristics are commonly used for grids.

Heuristic 1: Manhattan distance

$h = |x_{current} - x_{goal}| + |y_{current} - y_{goal}|$

never overestimates true cost in 4-direction grids

Used for problems involving:

- 4-direction movement
- No diagonals
- Each move cost = 1

Approximation Heuristics

Heuristic 2: Diagonal distance

$dx = |x_{current} - x_{goal}|$

$dy = |y_{current} - y_{goal}|$

$h = D(dx + dy) + (D2 - 2D)\min(dx, dy)$

Where:

D = 1 (straight move)

D2 = √2 (diagonal move)

Diagonal distance is used when:

Movement allowed in 8 directions

Straight move cost = 1

Diagonal move cost = √2

$$\min(dx, dy) \rightarrow \text{diagonal moves}$$

$$|dx - dy| \rightarrow \text{straight moves}$$

Heuristic	Grid Type	Move Cost
Manhattan	4-direction	All $= 1$
Diagonal	8-direction	$1, \sqrt{2}$
Euclidean	Any	Straight-line

Correct cost model is essential

Choosing the right heuristic matters

If the cost model does not match the environment, the computed path may be suboptimal or incorrect.
Example:

- Using Manhattan distance while allowing diagonal movement leads to systematic underestimation of diagonal efficiency.
- Treating diagonal moves as cost 1 instead of sqrt(2) distorts distance estimation.
- The heuristic defines how intelligently the search is guided toward the goal.
- A weak heuristic behaves like Dijkstra’s algorithm, exploring too many nodes.
- An incorrect heuristic can overestimate costs, breaking A*’s optimality guarantee.

Key properties of a good heuristic

- Admissible: never overestimates the true remaining cost.
- Consistent (monotonic): ensures stable and optimal expansion.
- Aligned with movement rules: reflects actual allowed moves and their costs.

Practical implication

- Correct cost model + appropriate heuristic
- → Faster search, fewer node expansions, optimal paths
- Mismatch between them
- → Inefficiency, incorrect paths, or loss of optimality

High level steps of the A* algorithm

1. Initialize OPEN and CLOSED lists
2. Insert start node into OPEN
3. While OPEN is not empty:

    Select node with smallest f (n)
    If goal reached, stop
    Generate valid neighbors
    Update g, h, and f
    Add better nodes to OPEN
    Move current node to CLOSED

Thank you