# Introduction to Artificial Intelligence

## Lecture 2: Breadth-First Search Algorithm

**Exercise:**

1. Declare a function that takes three numbers and returns their average.
2. Declare the following variables a, b, and c, and allow the user to enter their values.
3. Apply your function to a,b,c, and print the result.

---

**Solution:**

```
def average3 (x,y,z):
    return (x+y+z)/3

a = int(input('enter first number: '))
b = int(input('enter second number: '))
c = int(input('enter third number: '))
avg = average3(a,b,c)
print(type(avg))
```

**Notes:**

- The values are cast to integers because input() returns strings.
- The / operator in Python performs float division, regardless of whether the operands are integers or floats.
- The // operator performs integer (floor) division, regardless of operand types.

---

# Matplotlib Library:

Matplotlib is a graph plotting library in Python that serves as a visualization utility.

**Importing the library:**

```
import matplotlib.pyplot as plt
```

**Plotting a graph using Matplotlib:**

- Xdata=[content..]
  Ydata=[content..]
  plt.plot(Xdata, Ydata)
  plt.show()

Example:

```
import matplotlib.pyplot as plt
xpoints = [1, 8]
ypoints = [3, 10]
plt.plot(xpoints, ypoints)
plt.show()
```

---

**Exercise:**

You have the following data:

| Time | 0 | 2.2 | 4 | 5.6 | 7.2 | 9 |
|---|---|---|---|---|---|---|
| Car Speed | 0 | 1 | 3 | 5.1 | 7.8 | 10 |

Plot a graph showing car speed over time using matplotlib.

---

**Solution:**

```
import matplotlib.pyplot as plt
time=[0,2.2,4,5.6,7.2,9]
car_speed=[0,1,3,5.1,7.8,10]
plt.plot(time,car_speed,color='red') #color is not mandatory
plt.show()
```

---

# Breadth-First Search (BFS) Algorithm:

Breadth-first search (BFS) starts at the given start node S and explores the graph outward in all directions level by level, first visiting all nodes that are directly reachable from S via outgoing edges (i.e., have distance 1 from S), then nodes that have distance two from S, then distance three, etc.
To achieve this, the frontier in BFS is represented as a queue (First In First Out).

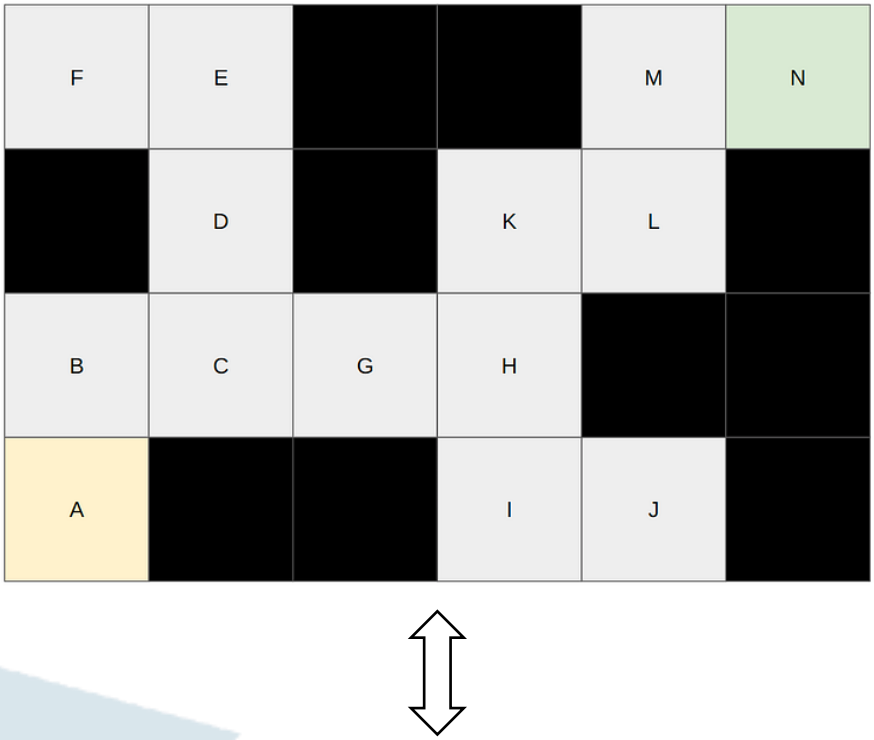
**Approach:**

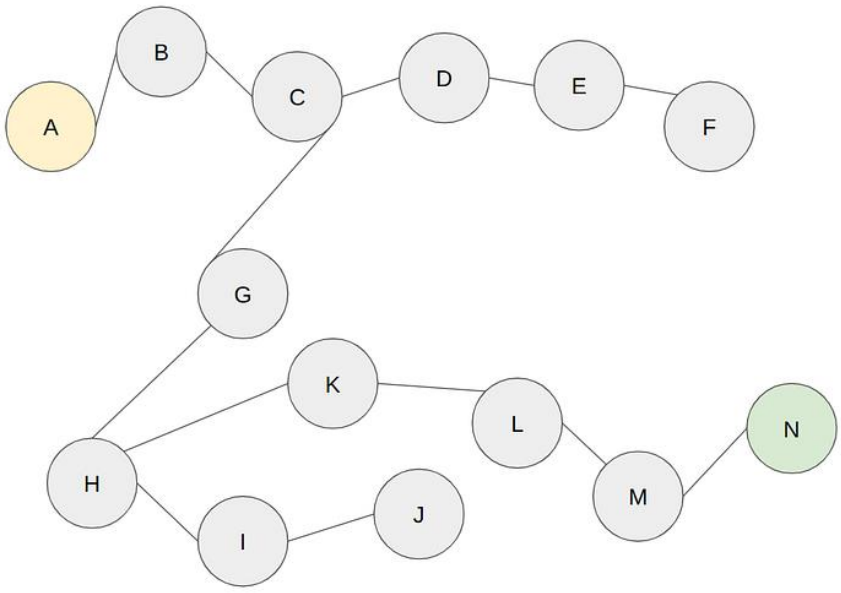
- Start with a frontier that contains the start node.
- Start with an empty explored set.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove the **shallowest** node from the frontier.
  - If the node is the goal state, return the solution.
  - Add the node to the explored set.
  - Expand node, add resulting nodes to the frontier if they aren't already in the frontier or the explored set.

**Solving Maze Problems with BFS:**

To apply BFS to a maze, we first **convert the maze into a graph**, where each open cell becomes a node and edges connect neighboring cells that can be moved between.

**Example:**

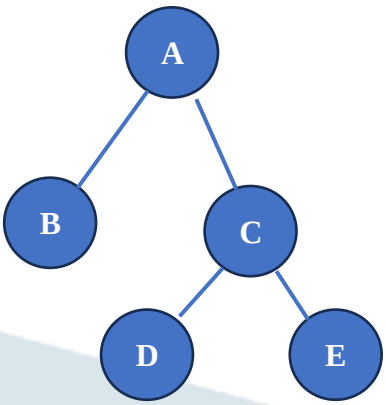


**Graph representation in Python:**

To deal with a graph in a programming language (like Python), we have to represent this graph using some data structure. In this lecture, we present one way to represent a graph in Python, which is a ***Dictionary***.
Declaring a graph as a dictionary in Python is basically a way to describe which node connects to which other nodes. Think of it as writing down a list of "friendships" between points in a network.
A dictionary in Python is a data structure that stores ***key → value*** pairs.
When we use it to represent a graph, each key is a node, and its value is a list of neighboring nodes (children).

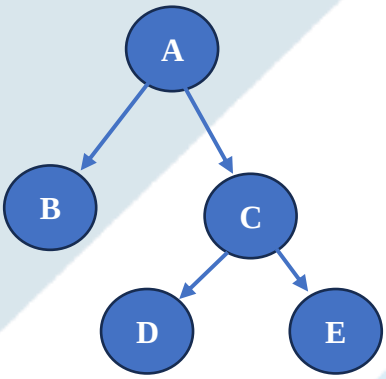
Example:




```
Graph= {
"A": ["B","C"],
"B": ["A"],
"C": ["A","D","E"],
"D": ["C"],
"E": ["C"]
}
```

Note: This is an undirected graph, so for example, B is a child of A, and A is a child of B at the same time.
This is obvious in the dictionary as B exists in A's children list, and A exists in B's children list.

**What if the graph is directed?**




```
Graph= {
"A": ["B","C"],
"B": [],
"C": ["D","E"],
"D": [],
"E": []
}
```

As shown in the dictionary, now B is a child of A, but A is not a child of B.

**BFS Python code**

This code demonstrates a function that implements the BFS algorithm on a graph, returning both the solution (the search steps taken from the start node to the goal node) and the cost (the number of links jumped to reach the solution). Comments in the code explain every step.

```
def bfs(graph, start_node, end_node): # function declaration

    solution = []          # List to store the order of visited nodes to the solution
    costs = 0              # Counter for the number of BFS iterations (number of links jumped)

    frontier = []          # Queue for BFS (nodes to be explored)
    visited = []           # List of already visited nodes

    frontier.append(start_node) # Add the start node to the queue

    while frontier:             # Continue while there are still nodes to explore

        selected_node = frontier.pop(0)    # Remove and return the first node (FIFO)
        visited.append(selected_node)      # Mark the current node as visited
        solution.append(selected_node)     # Add the current node to the solution path

        if selected_node == end_node:      # Check if the goal node is reached
            break                           # Stop the BFS search

        for neighbour in graph[selected_node]:     # Loop through all neighbors (children)
            if neighbour not in visited:           # Only consider unvisited neighbors
                frontier.append(neighbour)         # Add neighbor to the BFS queue

        costs += 1     # Increase cost after each BFS step

    return solution, costs     # Return visited path and BFS cost
```

Example: (calling the bfs function on the aforementioned graph)

1. For an existing node:

```
Graph= {
"A": ["B","C"],
"B": [],
"C": ["D","E"],
"D": [],
"E": []
}
start_state = "A"
goal_state = "D"
[s,c]=bfs(Graph,start_state,goal_state)
print("solution is: ", s)
print("cost is: ", c)
```

output:

```
solution is: ['A', 'B', 'C', 'D']
cost is:  3
```

2. For a node that does not exist: (or may exist in a "disconnected graph")

```
Graph= {
"A": ["B","C"],
"B": [],
"C": ["D","E"],
"D": [],
"E": []
}
start_state = "A"
goal_state = "F"
[s,c]=bfs(Graph,start_state,goal_state)
print("solution is: ", s)
print("cost is: ", c)

output:

solution is: ['A', 'B', 'C', 'D', 'E']
cost is:  5
```