# WEEK 2

# Question 1

Use the CartPole-v0 environment, the status of the system is specified by an “observation” of four parameters (x, v, θ, ω), where

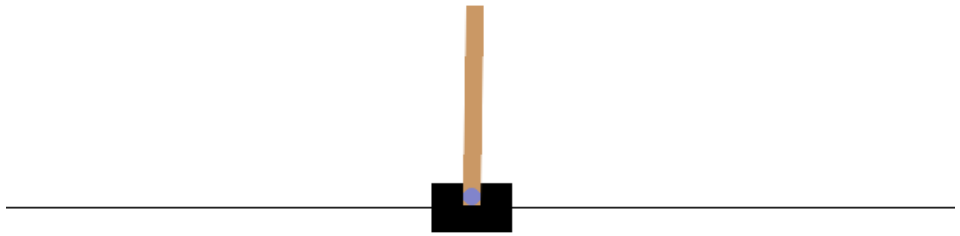
- x: the horizontal position of the cart (positive means to the right)
- v: the horizontal velocity of the cart (positive means moving to the right)
- θ: the angle between the pole and the vertical position (positive means clockwise)
- ω: angular velocity of the pole (positive means rotating clock-wise)

Use gym and write the following policies for 100 episodes :

a. Theta policy: if the pole is tilted to the left ($\theta<0$), then push the cart to the left and vice versa.
b. Omega policy : when the pole is moving away from the vertical position ($\omega<0$) then push the cart to the left and vice versa.
c. Plot the cumulative rewards (based on steps) for each policy, and write down the average rewards with standard deviation.
d. Comment on the performance of policies.

# Cart Pole

- A pole is attached by an unactuated joint to a cart, which moves along a frictionless track.
- The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.



## Action Space ---- Discrete(2)

- 0: Push cart to the left
- 1: Push cart to the right

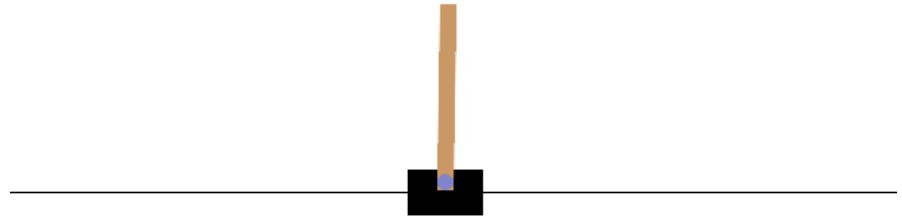
## Observation Space

| Num | Observation | Min | Max |
|---|---|---|---|
| 0 | Cart Position | -4.8 | 4.8 |
| 1 | Cart Velocity | -Inf | Inf |
| 2 | Pole Angle | ~ -0.418 rad (-24°) | ~ 0.418 rad (24°) |
| 3 | Pole Angular Velocity | -Inf | Inf |

- The cart x-position (index 0) can be take values between `(-4.8, 4.8)`, but the episode terminates if the cart leaves the `(-2.4, 2.4)` range.
- The pole angle can be observed between `(-.418, .418)` radians (or **±24°**), but the episode terminates if the pole angle is not in the range `(-.2095, .2095)` (or **±12°**)

# Reward

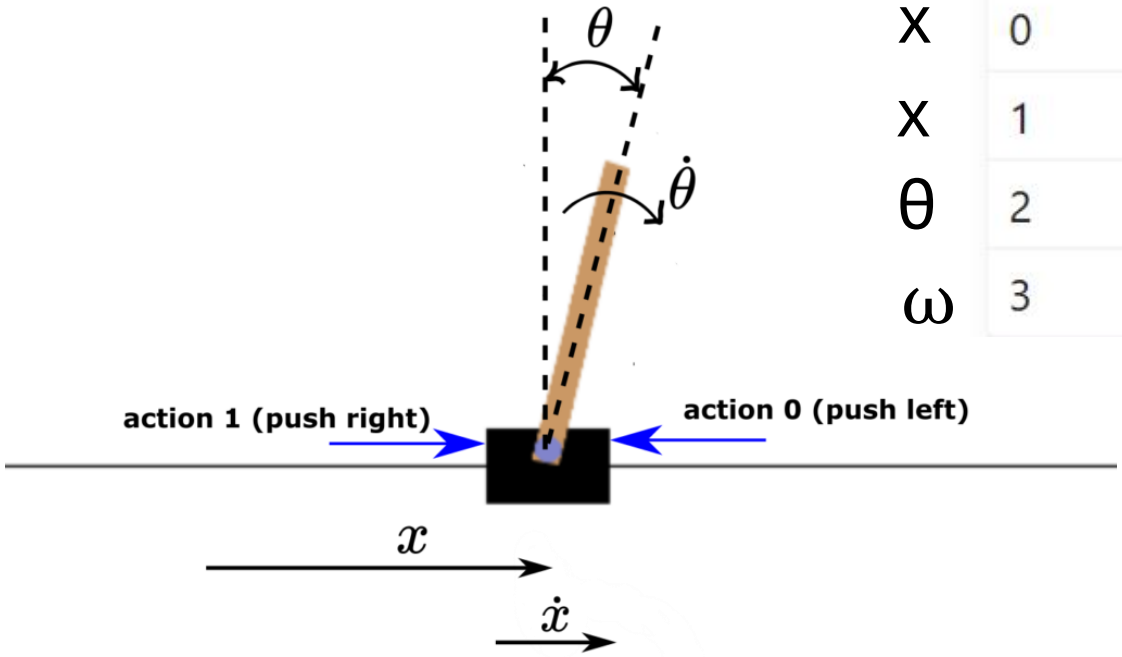
You get **a reward of 1 point for every step** you keep the pole balanced.



# Episode End

The episode ends if any one of the following occurs:

1. Termination: Pole Angle is greater than ±12°
2. Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
3. Truncation: Episode length is greater than 500 (200 for v0)




| | Num | Observation |
|---|---|---|
| x | 0 | Cart Position $x$ |
| x | 1 | Cart Velocity $\dot{x}$ |
| θ | 2 | Pole Angle $\theta$ |
| ω | 3 | Pole Angular Velocity $\dot{\theta}$ |

1. **Theta Policy**:
    - Push the cart left if the pole is tilted to the left ($\theta < 0$).
    - Push the cart right if the pole is tilted to the right ($\theta \geq 0$).
2. **Omega Policy**:
    - Push the cart left if the pole is rotating left ($\omega < 0$).
    - Push the cart right if the pole is rotating right ($\omega \geq 0$).

# Steps to implement

- Initialize the CartPole environment
- Define  # of episodes
- Store rewards for each policy

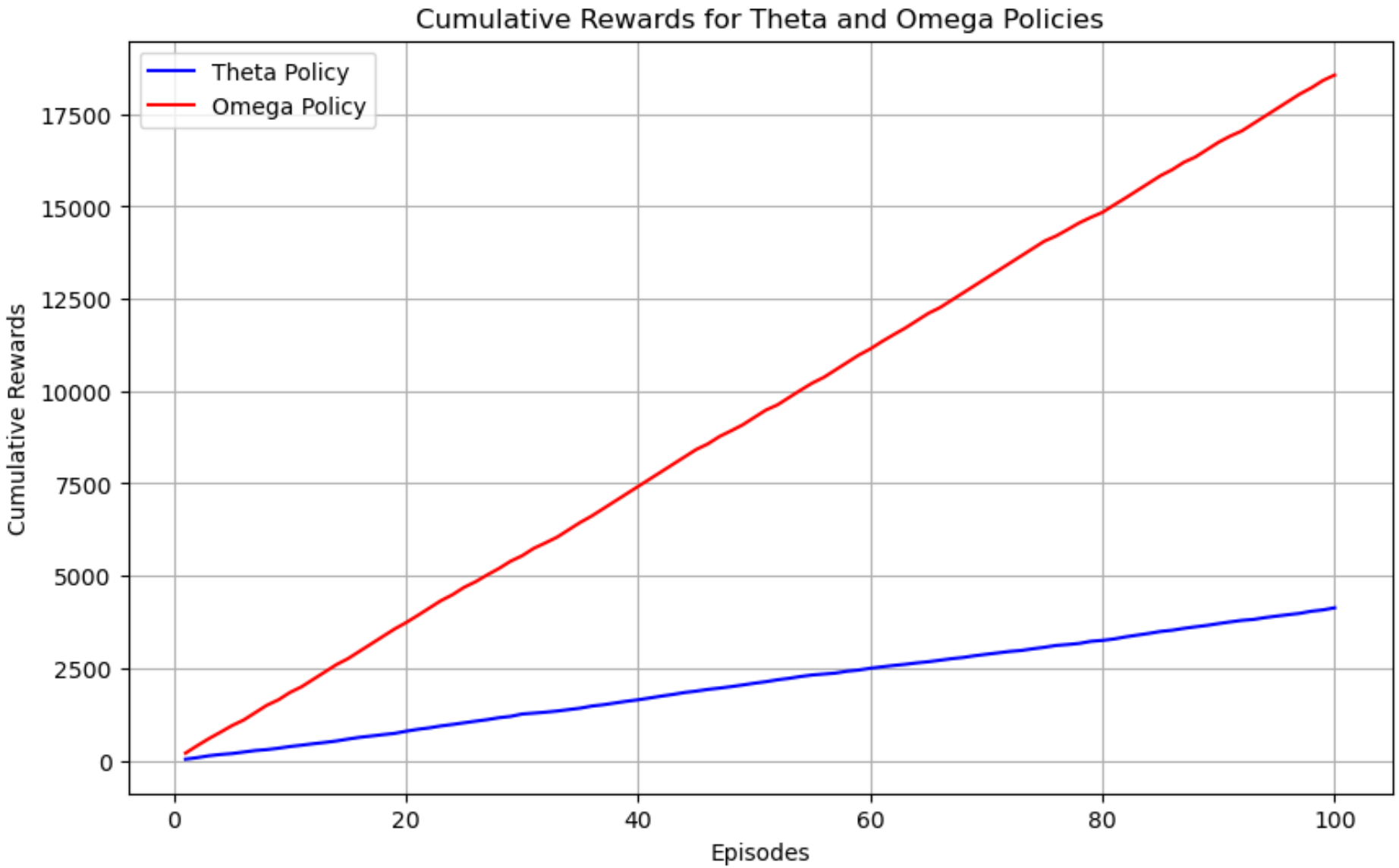
```
theta_rewards = []
omega_rewards = []
```

- Theta Policy: Push left if θ < 0, otherwise push right
  - loop through episodes, render env, calculate reward, display reward, sleep after each episode

- Omega Policy: Push left if $\omega < 0$, otherwise push right
  - loop through episodes, render env, calculate reward, display reward, sleep after each episode
- Close the environment rendering
- Calculate average and standard deviation for each policy
- Print results
- Plot cumulative rewards

```python
# Theta Policy: Push left if θ < 0, otherwise push right
for episode in range(num_episodes):
    observation = env.reset()
    total_reward = 0
    done = False

    while not done:
        env.render()  # Render the environment
        theta = observation[2]
        action = 0 if theta < 0 else 1  # Push left (0) or right (1)
        observation, reward, done, _ = env.step(action)
        total_reward += reward

    theta_rewards.append(total_reward)
    print(f"Theta Policy - Episode {episode + 1}: Reward = {total_reward}")
    time.sleep(1)  # Add sleep after each episode
```

```python
# Omega Policy: Push left if ω < 0, otherwise push right
for episode in range(num_episodes):
    observation = env.reset()
    total_reward = 0
    done = False

    while not done:
        env.render()  # Render the environment
        omega = observation[3]
        action = 0 if omega < 0 else 1  # Push left (0) or right (1)
        observation, reward, done, _ = env.step(action)
        total_reward += reward

    omega_rewards.append(total_reward)
    print(f"Omega Policy - Episode {episode + 1}: Reward = {total_reward}")
    time.sleep(1)  # Add sleep after each episode
```


Cumulative Rewards for Theta and Omega Policies
Theta Policy
Omega Policy
Cumulative Rewards
17500
15000
12500
10000
7500
5000
2500
0
0
20
40
60
80
100
Episodes

# Search Algorithms in AI

- A search problem consists of:
    - **A State Space.** Set of all possible states where you can be.
    - **A Start State.** The state from where the search begins.
    - **A Goal State.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.

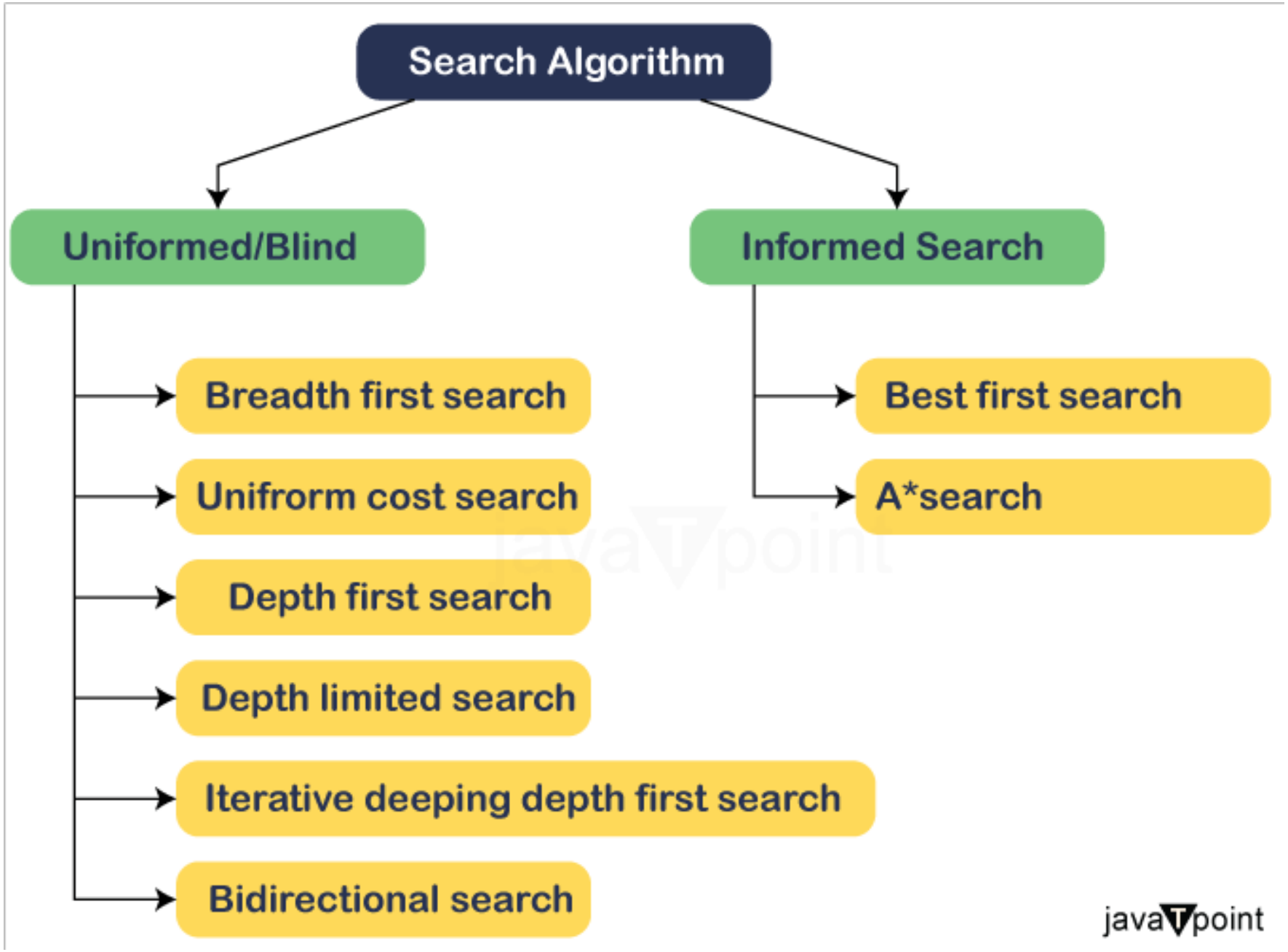
The set of all leaf nodes available for expansion at any given point is called the **Frontier ( Open List)**

The tree-search algorithm can be augmented with a data structure called the **explored set (Closed list),** which remembers every expanded node

# Properties of Search Algorithms

- **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
- **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution is said to be an optimal solution.
- **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
- **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms



# Uninformed/Blind Search

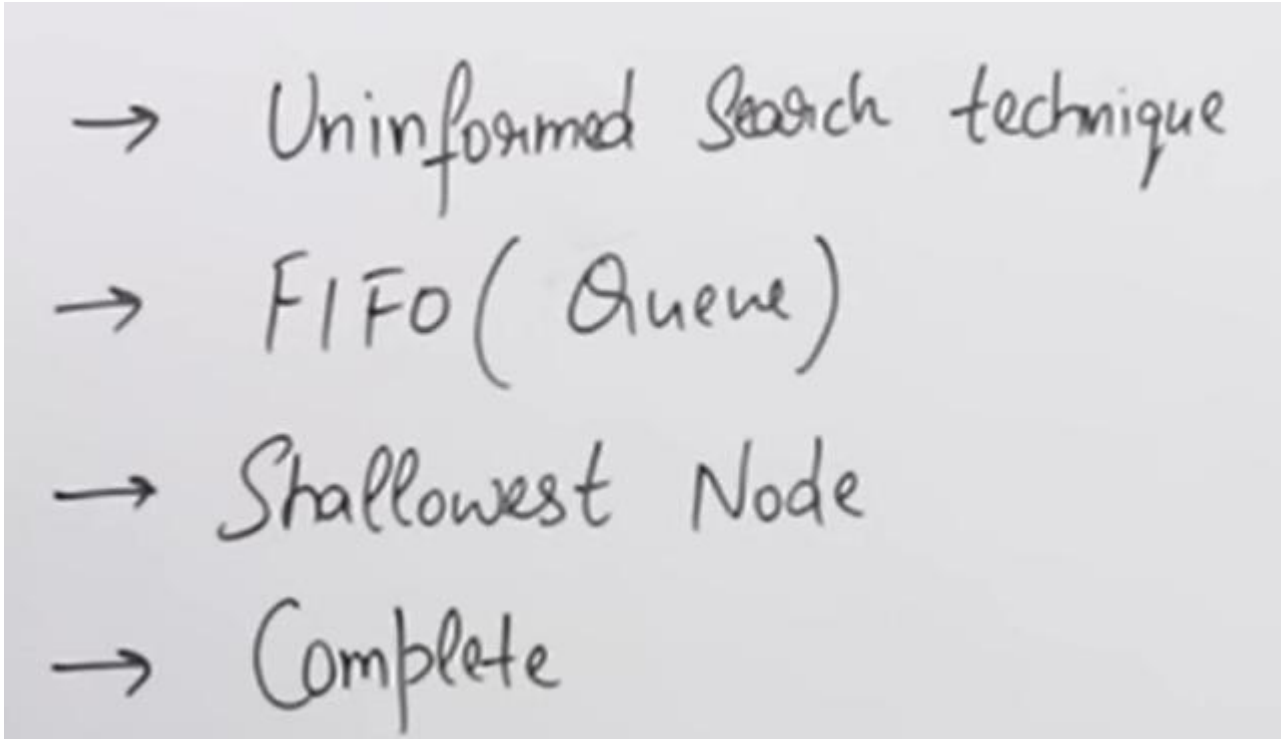
- Does not contain any domain knowledge such as closeness, or the location of the goal.
- It operates in a brute-force way
- The tree is searched without any information about the search space like initial state operators and tests for the goal
- Examines each node of the tree until it achieves the goal node.

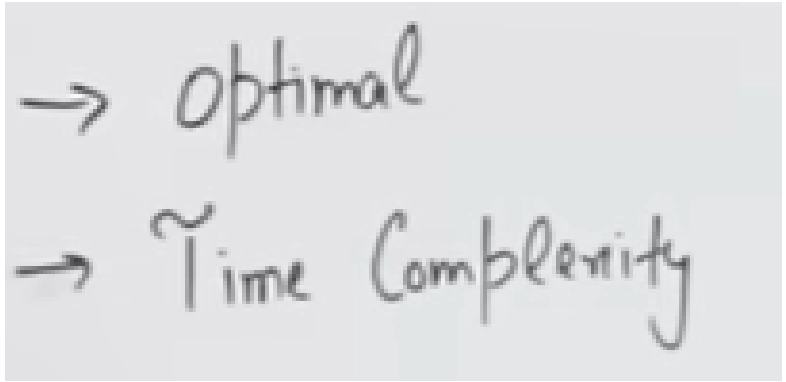
# Informed Search

- Use domain knowledge
- Informed search strategies can find a solution more efficiently than an uninformed search strategy.
- Also called a Heuristic search.

***A heuristic is a way that might not always be guaranteed for the best solutions but guaranteed to find a good solution in a reasonable time.***

# Breadth First Search (BFS)



i) Enter Starting nodes in Queue

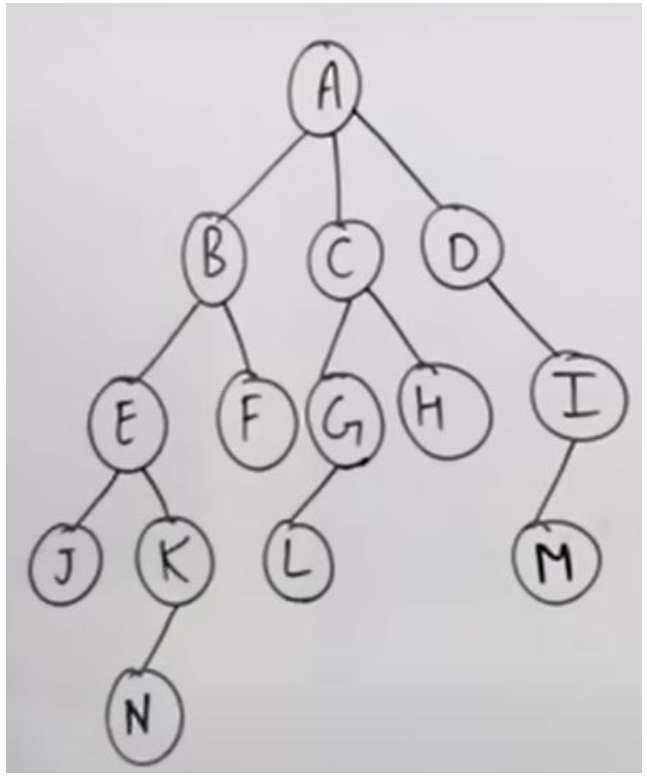
ii) if → Queue Empty return fail and Stop

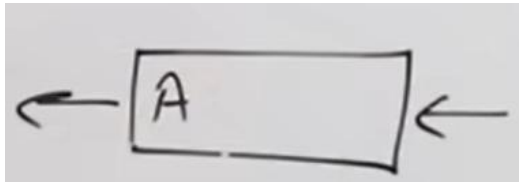
iii) if → first element in queue is goal node
then return Success & Stop

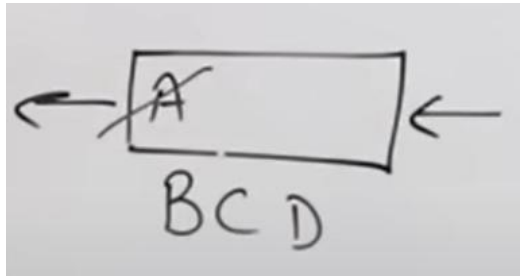
Else

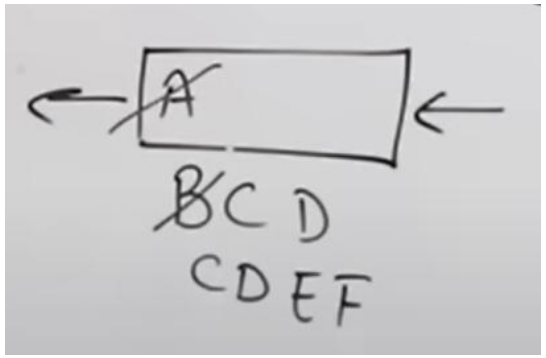
iv) Remove and EXPand first Element in Queue
and place Children nodes at the End of the Queue.

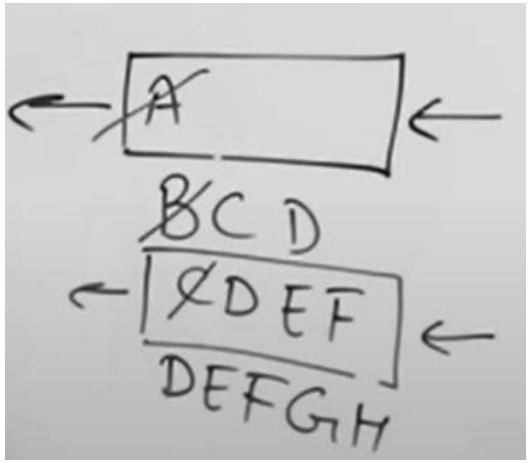
v) Goto Step(ii)

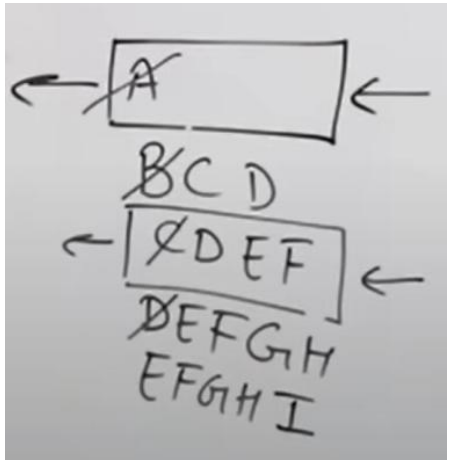

A
B
C
D
E
F
G
H
I
J
K
L
M
N

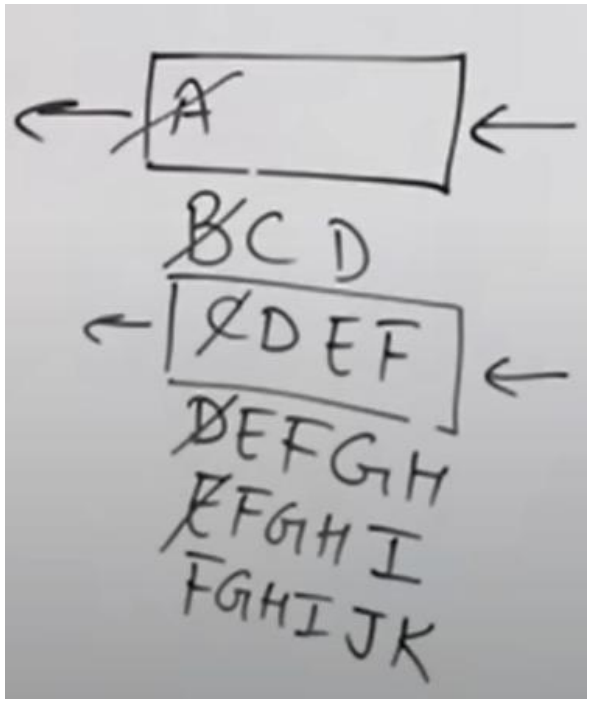

A

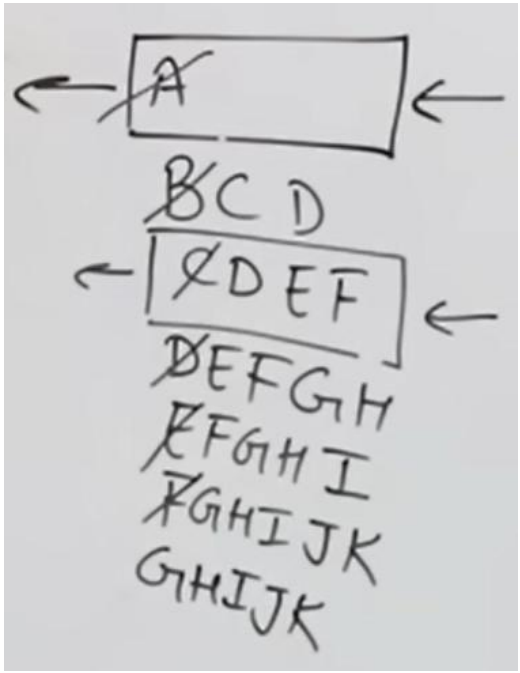

A
BCD

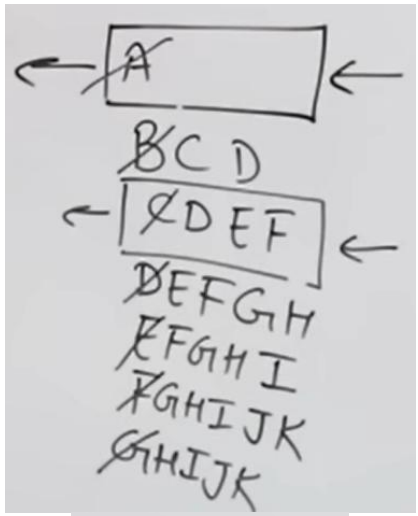

A
BCD
CDEF

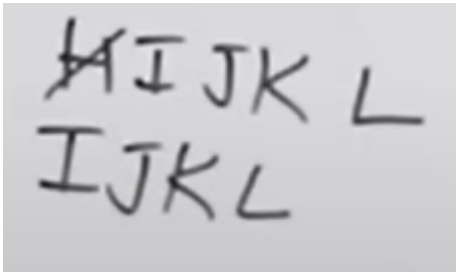

A
BCD
CDEF
DEFGH

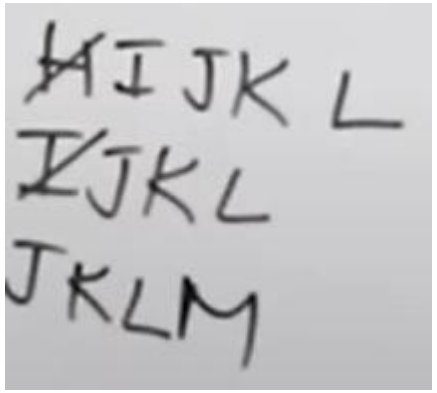

A
BCD
CDEF
DEFGH
EFGHI

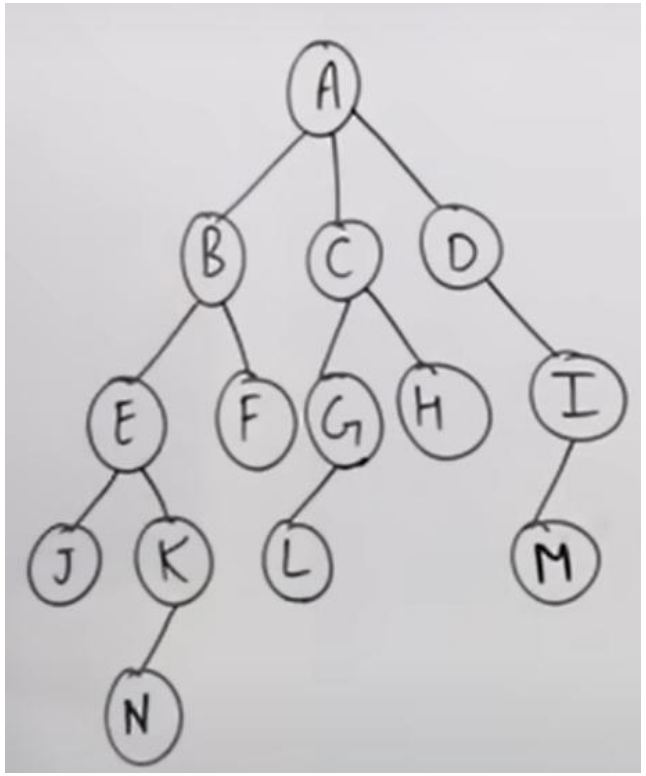

A
BCD
CDEF
DEFGH
EFGHI
FGHIJK

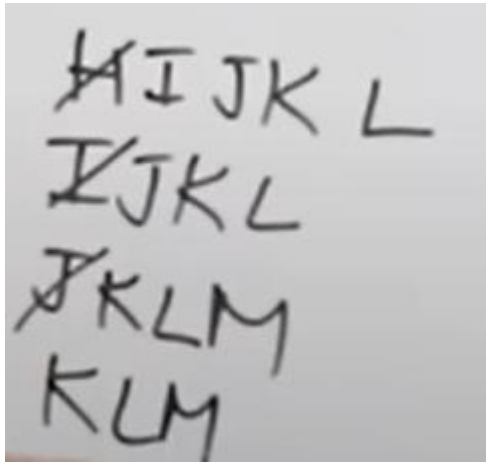

A
BCD
CDEF
DEFGH
EFGHI
FGHIJK
GHIJK

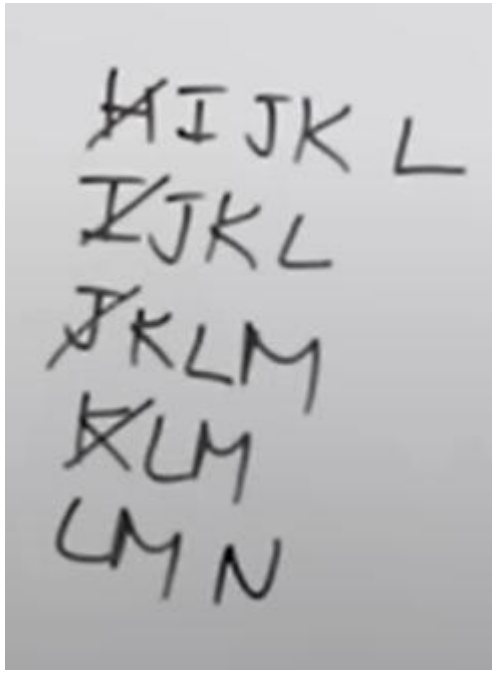

A
BCD
CDEF
DEFGH
EFGHI
FGHIJK
GHIJK
HIJKL

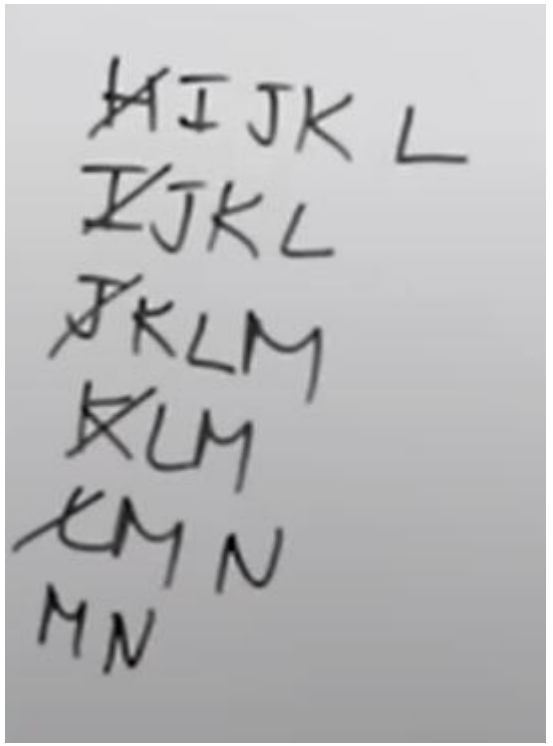

AHIJKL
IJKL

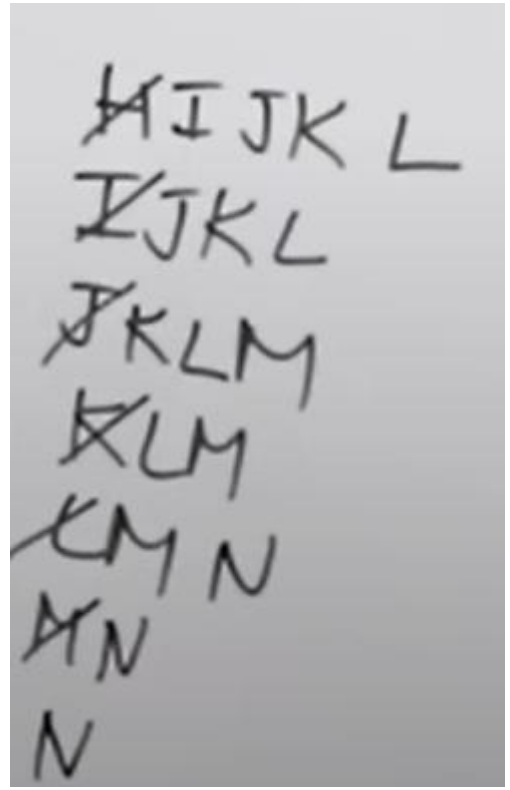

AHIJKL
IJKL
JKLM


A
B
C
D
E
F
G
H
I
J
K
L
M
N


HIJKL
IJKL
JKLM
KLM


HIJKL
IJKL
JKLM
KLM
LMN


HIJKL
IJKL
JKLM
KLM
LMN
MN


HIJKL
IJKL
JKLM
KLM
LMN
MN
N

# Algorithm

```
BREATH-FIRST-SEARCH(problem) returns a solution node or failure
        node ← NODE(problem.INITIAL)
        if problem.IS-GOAL(node.STATE) then return node

        frontier ← a FIFO queue, initialized with node
        reached ← {problem.INITIAL}

        while not IS-EMPTY(frontier) do
                    node ← POP(frontier)

                     for each child in EXPAND(problem, node) do
                           s ← child.STATE

                           if problem.IS-GOAL(s) then return child

                           if s is not in reached then
                               add s to reached
                               add child to frontier
         return failure
```

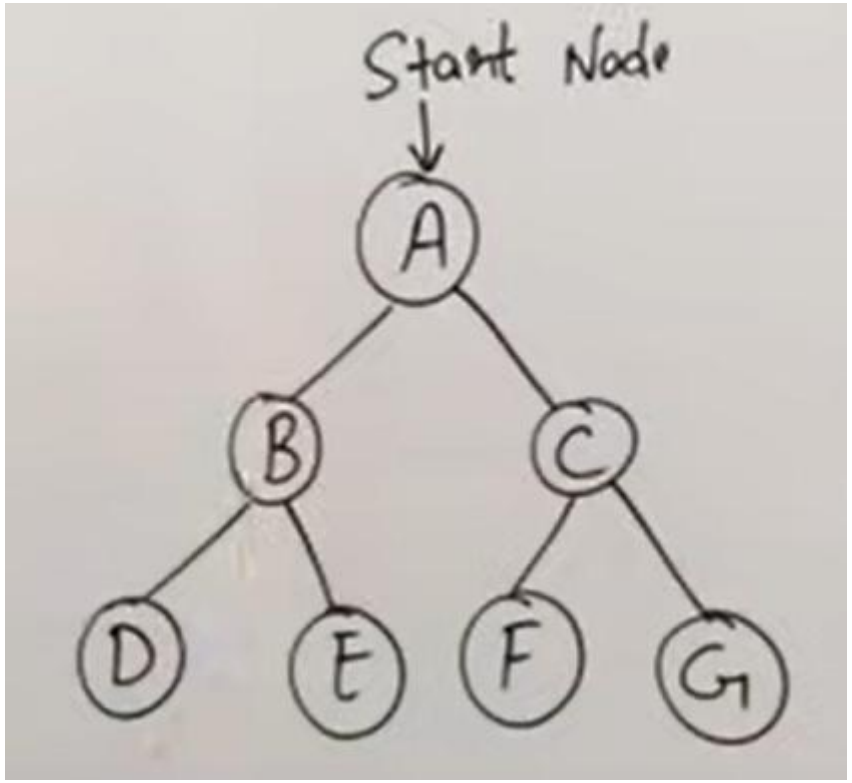
```
def breadth_first_search(problem):
    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node

    frontier = deque([node])
    reached = {problem.initial}

    while frontier:
        node = frontier.popleft()

        for child in problem.expand(node):
            s = child.state

            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.add(s)
                frontier.append(child)
    return None
```

```python
# BFS function to find and return the path
def bfs_with_path(graph, start_node):
    frontier = [start_node]  # List to store the frontier
    explored = []  # List to store explored nodes
    path = []  # List to store the BFS traversal path

    while frontier:
        # Pop the first element from the frontier
        current_node = frontier.pop(0)
        if current_node not in explored:
            explored.append(current_node)  # Mark the node as explored
            path.append(current_node)  # Add to path
            # Add neighbors to the frontier
            for neighbor in sorted(graph.neighbors(current_node)):
                if neighbor not in explored and neighbor not in frontier:
                    frontier.append(neighbor)

    return path
```

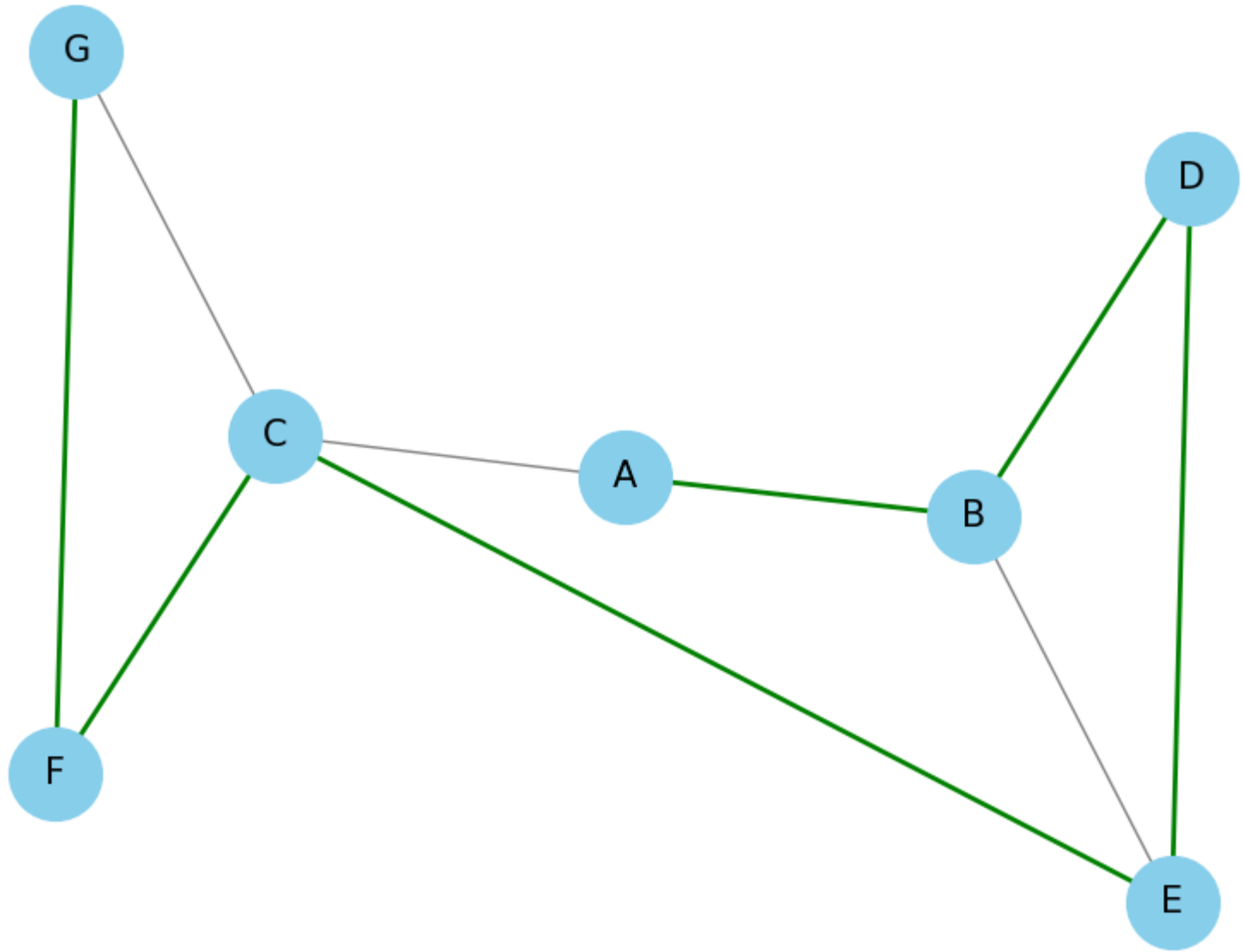
# Depth First Search (DFS)

- Uninformed Search technique
- Stack (LIFO)
- Deepest Node
- ~Incomplete
- Non optimal
- ~Time Complexity

1. Push the root node in the Stack
2. While (Stack is not empty)
    (a) Pop a node from the stack
        (i) If node is a goal node return success
        (ii) Push all children of node onto the stack
3. return failure


Start Node
A
B
C
D
E
F
G

```python
def dfs_with_path(graph, start_node):
    stack = [start_node]  # Stack to store the frontier
    explored = []  # List to store explored nodes
    path = []  # List to store the DFS traversal path

    while stack:
        # Pop the last element from the stack
        current_node = stack.pop()
        if current_node not in explored:
            explored.append(current_node)  # Mark the node as explored
            path.append(current_node)  # Add to path
            # Add neighbors to the stack (sorted to maintain order)
            for neighbor in sorted(graph.neighbors(current_node), reverse=True):
                if neighbor not in explored:
                    stack.append(neighbor)

    return path
```


DFS Traversal Path Starting from A
G
D
C
A
B
F
E