

Best First search !! Implement Breadth first search algorithm in python. Consider following tree. Start node = 2, Goal node = 11

```
from collections import deque

# Define the graph as an adjacency list
# Add node '1' and its neighbors (if any) to the graph
graph = {
    2: [7, 5],
    7: [1, 6],
    5: [9],
    6: [5, 11],
    9: [4],
    11: [],
    4: [],
    1: [] # Add node 1 with an empty list of neighbors (or its actual neighbors)
}

# BFS implementation
def bfs(start, goal):
    visited = set() # Set to keep track of visited nodes
    queue = deque([start]) # Queue to process nodes level by level

    while queue:
        node = queue.popleft() # Dequeue a node
        if node not in visited:
            print(f"Visiting: {node}") # Visit the node
            visited.add(node)

            if node == goal:
                print(f"Goal node {goal} found!")
                return

            # Add unvisited neighbors to the queue
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

    print(f"Goal node {goal} not found in the graph.")

# Call BFS function with start and goal nodes
bfs(start=2, goal=11)
```

```
➡ Visiting: 2
Visiting: 7
Visiting: 5
Visiting: 1
Visiting: 6
Visiting: 9
Visiting: 11
Goal node 11 found!
```

Implement Breadth first search algorithm in python. Consider following graph. (Root Node: 5, Goal Node = ?)

```
from collections import deque

# Define the graph as an adjacency list
graph = {
    5: [3, 7],
    3: [2, 4, 5],
    7: [5, 8],
    2: [3],
    4: [3],
    8: [7],
}

# BFS implementation
def bfs(graph, start, goal):
    visited = set() # Set to keep track of visited nodes
    queue = deque([start]) # Queue to process nodes level by level

    while queue:
        node = queue.popleft() # Dequeue a node
        if node not in visited:
            print(f"Visiting: {node}") # Visit the node
            visited.add(node)

            if node == goal:
                print(f"Goal node {goal} found!")
```

```

        return

    # Add unvisited neighbors to the queue
    for neighbor in graph[node]:
        if neighbor not in visited:
            queue.append(neighbor)

    print(f"Goal node {goal} not found in the graph.")

# Example usage
root_node = 5
goal_node = 8 # You can specify any goal node here
bfs(graph, start=root_node, goal=goal_node)

```

```

➡ Visiting: 5
  Visiting: 3
  Visiting: 7
  Visiting: 2
  Visiting: 4
  Visiting: 8
  Goal node 8 found!

```

Implement Depth first search algorithm in python. Consider following graph. ( Goal Node= 0 )

```

from collections import deque

# Define the graph as an adjacency list based on the image
graph = {
    'S': ['A', 'K'],
    'A': ['B', 'C'],
    'K': ['I', 'J'],
    'B': ['H', 'M'],
    'C': ['N'],
    'I': ['O'],
    'J': [],
    'H': [],
    'M': [],
    'N': [],
    'O': [],
}

# BFS implementation
def bfs(graph, start, goal):
    visited = set() # Set to keep track of visited nodes
    queue = deque([start]) # Queue to process nodes level by level

    while queue:
        node = queue.popleft() # Dequeue a node
        if node not in visited:
            print(f"Visiting: {node}") # Visit the node
            visited.add(node)

            if node == goal:
                print(f"Goal node {goal} found!")
                return

            # Add unvisited neighbors to the queue
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

    print(f"Goal node {goal} not found in the graph.")

# Example usage
root_node = 'S'
goal_node = 'O' # Specify the desired goal node
bfs(graph, start=root_node, goal=goal_node)

```

```

➡ Visiting: S
  Visiting: A
  Visiting: K
  Visiting: B
  Visiting: C
  Visiting: I
  Visiting: J
  Visiting: H
  Visiting: M
  Visiting: N
  Visiting: O
  Goal node O found!

```

Implement Depth first search algorithm in python. Consider following graph. (Goal Node= 6 ).

```
# Define the graph using an adjacency list
graph = {
    1: [2, 3],
    2: [4, 5],
    3: [6, 7],
    4: [],
    5: [],
    6: [],
    7: []
}

# Function to implement DFS
def depth_first_search(graph, start, goal):
    visited = set() # Keep track of visited nodes
    stack = [start] # Use a stack to hold nodes to visit

    while stack:
        node = stack.pop() # Get the current node
        if node not in visited:
            print(f"Visiting node: {node}")
            visited.add(node)

            if node == goal:
                print(f"Goal node {goal} found!")
                return True

            # Add neighbors to the stack
            # We reverse to maintain order, so the leftmost child is processed first
            stack.extend(reversed(graph[node]))

    print("Goal node not found.")
    return False

# Call the DFS function
depth_first_search(graph, 1, 6)
```

```
➡ Visiting node: 1
Visiting node: 2
Visiting node: 4
Visiting node: 5
Visiting node: 3
Visiting node: 6
Goal node 6 found!
True
```

Implement best first search algorithm in python for following tree. Starting node (10), Goal node (100).

```
import heapq

# Define the graph as an adjacency list with (node, cost) pairs
graph = {
    10: [(5, 3), (15, 2)],
    5: [(25, 4)],
    15: [(30, 1), (35, 1)],
    25: [(45, 1), (50, 1)],
    30: [(55, 5), (60, 2)],
    35: [(100, 3)],
    45: [],
    50: [],
    55: [],
    60: [],
    100: []
}

# Best-First Search function
def best_first_search(graph, start, goal):
    visited = set() # To keep track of visited nodes
    priority_queue = [(0, start)] # Min-heap to store (cost, node)

    while priority_queue:
        cost, node = heapq.heappop(priority_queue) # Pop the node with the lowest cost

        if node not in visited:
            print(f"Visiting node: {node}")
            visited.add(node)
```

```

    if node == goal:
        print(f"Goal node {goal} found!")
        return True

    # Add neighbors to the priority queue
    for neighbor, weight in graph[node]:
        if neighbor not in visited:
            heapq.heappush(priority_queue, (weight, neighbor))

    print("Goal node not found.")
    return False

# Call the BFS function
best_first_search(graph, 10, 100)

```

```

→ Visiting node: 10
Visiting node: 15
Visiting node: 30
Visiting node: 35
Visiting node: 60
Visiting node: 5
Visiting node: 100
Goal node 100 found!
True

```

Implement best first search algorithm in python for following graph. Starting node(S), Goal node(G)

```

import heapq

# Define the graph as an adjacency list with (neighbor, heuristic) pairs
graph = {
    'S': [('A', 5), ('B', 1)],
    'A': [('G', 1), ('C', 1), ('S', 5)],
    'B': [('C', 2), ('S', 1)],
    'C': [('G', 2), ('B', 2), ('A', 2)],
    'G': [] # Goal node has no outgoing edges
}

# Best-First Search function
def best_first_search(graph, start, goal):
    visited = set() # To keep track of visited nodes
    priority_queue = [(0, start)] # Min-heap to store (heuristic, node)

    while priority_queue:
        heuristic, node = heapq.heappop(priority_queue) # Pop the node with the lowest heuristic value

        if node not in visited:
            print(f"Visiting node: {node}")
            visited.add(node)

            if node == goal:
                print(f"Goal node {goal} found!")
                return True

            # Add neighbors to the priority queue
            for neighbor, h_value in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (h_value, neighbor))

    print("Goal node not found.")
    return False

# Call the BFS function
best_first_search(graph, 'S', 'G')

```

```

→ Visiting node: S
Visiting node: B
Visiting node: C
Visiting node: A
Visiting node: G
Goal node G found!
True

```

Implement A\* algorithm in python for following graph. Starting node (S), Goal node(G). Heuristic Value: H(S) = 5, H(A) = 3, H(B) = 4, H(C) = 2, H(D) = 6, H(G) = 0

```

import heapq

# Define the graph as an adjacency list

```

```

# Define the graph as an adjacency list
graph = {
    'S': [('A', 1), ('C', 10)],
    'A': [('B', 2), ('C', 1)],
    'B': [('D', 5)],
    'C': [('D', 4), ('G', 10)],
    'D': [('G', 2)],
    'G': []
}

# Define the heuristic values
heuristic = {
    'S': 5,
    'A': 3,
    'B': 4,
    'C': 2,
    'D': 6,
    'G': 0
}

def a_star_algorithm(start, goal):
    # Priority queue to store nodes with their f-cost (g + h)
    open_set = []
    heapq.heappush(open_set, (0 + heuristic[start], start))

    # Track the cost from start to each node (g-cost)
    g_costs = {node: float('inf') for node in graph}
    g_costs[start] = 0
    # Track the path
    came_from = {}

    while open_set:
        # Get the node with the lowest f-cost
        current_f, current_node = heapq.heappop(open_set)

        if current_node == goal:
            # Reconstruct the path from start to goal
            path = []
            while current_node in came_from:
                path.append(current_node)
                current_node = came_from[current_node]
            path.append(start)
            path.reverse()
            return path, g_costs[goal]


        for neighbor, cost in graph[current_node]:
            tentative_g_cost = g_costs[current_node] + cost
            if tentative_g_cost < g_costs[neighbor]:
                # Update g-cost and record the path
                g_costs[neighbor] = tentative_g_cost
                came_from[neighbor] = current_node
                f_cost = tentative_g_cost + heuristic[neighbor]
                heapq.heappush(open_set, (f_cost, neighbor))

    return None, float('inf') # Return None if no path is found

# Run the algorithm
start_node = 'S'
goal_node = 'G'
path, cost = a_star_algorithm(start_node, goal_node)

# Display the results
print(f"Path: {path}")
print(f"Total Cost: {cost}")

```

 Path: ['S', 'A', 'C', 'D', 'G']  
 Total Cost: 8

