Problem 1

Using the graph presented in `Figure 1` as input, and assuming that the goal state is `11`, list the order in which the nodes will be visited (i.e., expanded) if they are generated in `ascending order`:

1. When using breadth-first search.
2. When using depth-first search.
3. When using iterative deepening depth-first search.

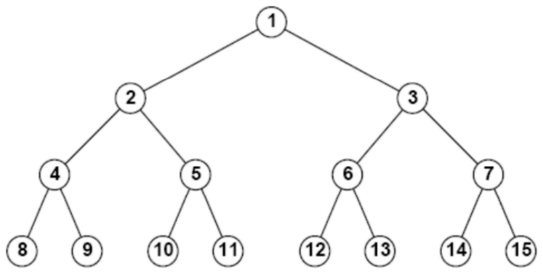


Figura 1: Grafo para explorar las diferentes estrategias de búsqueda.

As we can see is obvious that a solution exists, and the graph is a tree, so we can use the search algorithms to find the solution and the order in which the nodes will be visited.

Main Objectives

The main objective of this problem is to understand the differences between the three search algorithms presented in the question. In particular, the goal is to understand how the order in which the nodes are generated can affect the performance of the search algorithms.

Secondary Objectives

The secondary objectives are to compare the performance of the three search algorithms in terms of time taken.

Methodology

For this purpose, the following libraries will be necessary:

- **Networkx**: For efficient handling of graphs and/or trees
- **Matplotlib**: To display both the problem and the results of the analysis

- **Deque**: For efficient handling of queues (Optional)
- **NumPy**: For handling numerical data (Optional - Plot the results)

What we will do is to create a graph then we will implement the three search algorithms and we will analyze the results.

Implementation

The graph implementation will be a dictionary-based representation. In this one, the graph will be represented as a dictionary where the keys are the nodes and the values are the neighbors of the nodes. The following cell will present the implementation of the graph.

In [ ]:
```
from collections import deque
from typing import Dict, List, Tuple
import matplotlib.pyplot as plt
import networkx as nx
import time
import numpy as np
```

For the implementation of the graph as a dictionary a function called `create_graph` will be created. This function will return a dictionary where the keys are the nodes and the values are the neighbors of the nodes. The following cell will present the implementation of the graph.

In [ ]:
```
def create_graph() -> Dict[int, List[int]]:
    graph: Dict[int, List[int]] = {
        1: [2, 3],
        2: [4, 5],
        3: [6, 7],
        4: [8, 9],
        5: [10, 11],  # Goal
        6: [12, 13],
        7: [14, 15],

        # Leaf nodes
        8: [],
        9: [],
        10: [],
        11: [],  # Goal
        12: [],
        13: [],
        14: [],
        15: []
    }
    return graph
```

Then it will be necessary to implement the three search algorithms. The first algorithm to be implemented is the breadth-first search. The following cell will present the implementation of the breadth-first search algorithm. The function `bfs` will receive the graph, the start node, and the goal node as input and will return the order in which the nodes were visited until the goal node was found.

In [ ]:
```
def bfs(graph: Dict[int, List[int]], start: int, goal: int) -> List[int]:
    visited: List[int] = [] # Reached Nodes
    queue: deque = deque([start]) # Frontier

    while queue:
        node: int = queue.popleft()
        if node == goal:
            visited.append(node)
            return visited
        if node not in visited:
            visited.append(node)
            queue.extend([n for n in graph[node] if n not in visited])

    return visited
```

The second algorithm to be implemented is the depth-first search. The following cell will present the implementation of the depth-first search algorithm. The function `dfs` will receive the graph, the start node, and the goal node as input and will return the order in which the nodes were visited until the goal node was found.

In [ ]:
```
def dfs(graph: Dict[int, List[int]], start: int, goal: int) -> List[int]:
    visited: List[int] = [] # Reached Nodes
    stack: List[int] = [start] # Frontier

    while stack:
        node: int = stack.pop()
        if node not in visited:
            visited.append(node)
            if node == goal:
                break
            stack.extend(reversed(graph[node]))

    return visited
```

The third algorithm to be implemented is the iterative deepening depth-first search. The following cell will present the implementation of the iterative deepening depth-first search algorithm. The function `iddfs` will receive the graph, the start node, the goal and it will return the order in which the nodes were visited until the goal node was found and the path to the goal node.

In [ ]:
```
def iddfs(
        graph: Dict[int, List[int]],
        start: int,
        goal: int
        ) -> Tuple[List[int], List[int]]:
    visited_order: List[int] = []
    path_to_goal: List[int] = []

    def dls(node: int, depth: int, path: List[int]) -> bool:
        if node not in path:
            visited_order.append(node)

        if node == goal:
            path_to_goal.extend(path + [node])
            return True
        if depth == 0:
            return False

        for neighbour in graph[node]:
            if dls(neighbour, depth - 1, path + [node]):
                return True

        return False

    depth = 0
    while not dls(start, depth, []):
        depth += 1
        if depth > len(graph):
            break

    return visited_order, path_to_goal
```

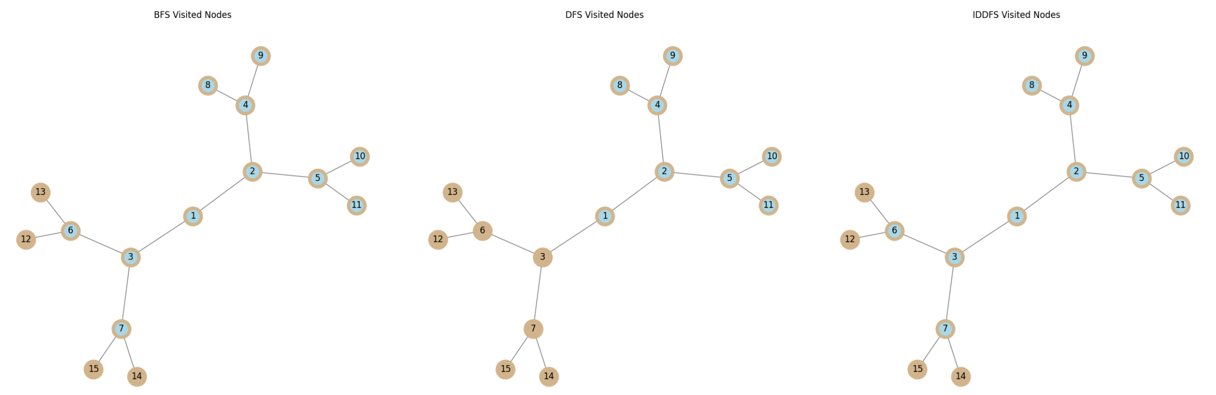
We now need to test if the algorithms are working correctly. So we will run the three algorithms with the graph presented in `Figure 1` and the goal state is `11`.

In [ ]:
```
graph: Dict[int, List[int]] = create_graph()
start_node: int = 1
goal_node: int = 11

bfs_visited: List[int] = bfs(graph, start_node, goal_node)
dfs_visited: List[int] = dfs(graph, start_node, goal_node)
iddfs_visited: List[int] = iddfs(graph, start_node, goal_node)

print(f"BFS: {bfs_visited}")
print(f"DFS: {dfs_visited}")
print(f"IDS: {iddfs_visited}")
```

```
BFS: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
DFS: [1, 2, 4, 8, 9, 5, 10, 11]
IDS: ([1, 1, 2, 3, 1, 2, 4, 5, 3, 6, 7, 1, 2, 4, 8, 9, 5, 10, 11], [1, 2, 5, 11])
```

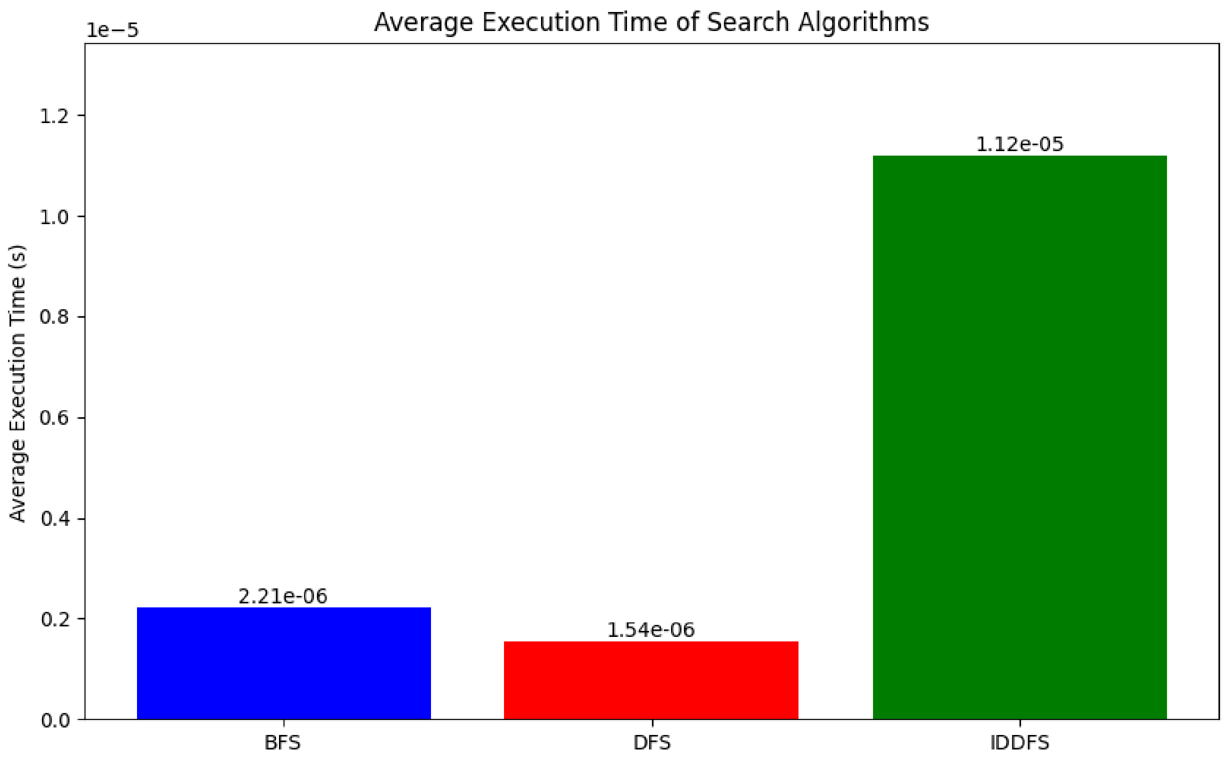
To see the the visited nodes in the three search algorithms we will use the `networkx` library to plot the graph and the visited nodes. So we will create a function called `plot_search_algorithms` that will receive the graph and the visited nodes as input and will plot the graph and the visited nodes.

In [ ]:
```
def plot_search_algorithms(graph: Dict[int, List[int]], bfs_order: List[int], dfs_orde
    # Initialize the graph
    G = nx.Graph()

    # Add nodes and edges
    for node, edges in graph.items():
        G.add_node(node)
        for edge in edges:
            G.add_edge(node, edge)

    # Define positions for all nodes
    pos = nx.spring_layout(G, seed=42)

    # Create a figure with 3 subplots
    fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(24, 8))
    # Plot BFS
    nx.draw(G, pos, ax=axes[0], with_labels=True, node_size=700, node_color='tan', edg
    nx.draw_networkx_nodes(G, pos, nodelist=bfs_order, node_color='lightblue', ax=axes
    axes[0].set_title('BFS Visited Nodes')

    # Plot DFS
    nx.draw(G, pos, ax=axes[1], with_labels=True, node_size=700, node_color='tan', edg
    nx.draw_networkx_nodes(G, pos, nodelist=dfs_order, node_color='lightblue', ax=axes
    axes[1].set_title('DFS Visited Nodes')

    # Plot IDDFS
    nx.draw(G, pos, ax=axes[2], with_labels=True, node_size=700, node_color='tan', edg
    nx.draw_networkx_nodes(G, pos, nodelist=iddfs_order[0], node_color='lightblue', ax
    axes[2].set_title('IDDFS Visited Nodes')

    plt.tight_layout()
    plt.show()

plot_search_algorithms(graph, bfs_visited, dfs_visited, iddfs_visited)
```



In order to compare the performance of the search algorithms, we can measure the time taken to find the goal node. We can use the time module to measure the time taken by each algorithm to find the goal node. For that we will create a function called `measure_average_execution_time` that will receive the graph, and the number of iterations to measure the average execution time of the three search algorithms. The function will return the average execution time of the three search algorithms. The start node and the goal node will be randomly chosen. Also a function called `plot_average_execution_time` will be created to plot the average execution time of the three search algorithms.

In [ ]:
```
# define a seed for reproducibility
np.random.seed(100)

def measure_average_execution_time(graph: Dict[int, List[int]], iterations: int = 100)
    # Initialize lists to store execution times
    bfs_times, dfs_times, iddfs_times = [], [], []

    # Randomly select start and goal nodes for each iteration
    nodes = list(graph.keys())
    for _ in range(iterations):
        start, goal = np.random.choice(nodes, 2, replace=False)

        # Measure BFS time
        start_time = time.time()
        bfs(graph, start, goal)
        bfs_times.append(time.time() - start_time)

        # Measure DFS time
        start_time = time.time()
        dfs(graph, start, goal)
        dfs_times.append(time.time() - start_time)

        # Measure IDDFS time
        start_time = time.time()
        iddfs(graph, start, goal)
        iddfs_times.append(time.time() - start_time)

    # Calculate average times
    avg_bfs_time = np.mean(bfs_times)
    avg_dfs_time = np.mean(dfs_times)
    avg_iddfs_time = np.mean(iddfs_times)

    return avg_bfs_time, avg_dfs_time, avg_iddfs_time

def plot_average_times(algorithms: list, avg_times: list, colors: list, title: str, yl
    plt.figure(figsize=(10, 6))

    # Creating the bar plot with labels
    bars = plt.bar(algorithms, avg_times, color=colors)
    for bar, time in zip(bars, avg_times):
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width() / 2.0, height, f'{time:.2e}', ha='cente

    # Setting titles and labels
    plt.ylabel(ylabel)
    plt.title(title)
    plt.ylim(0, max(avg_times) * 1.2)  # Add some space above the highest bar

    plt.show()

# Measure and plot the average execution times

avg_bfs_time, avg_dfs_time, avg_iddfs_time = measure_average_execution_time(graph, 10

plot_average_times(
    algorithms=['BFS', 'DFS', 'IDDFS'],
    avg_times=[avg_bfs_time, avg_dfs_time, avg_iddfs_time],
    colors=['blue', 'red', 'green'],
    title='Average Execution Time of Search Algorithms',
    ylabel='Average Execution Time (s)'
)
```


Average Execution Time of Search Algorithms
1e−5
Average Execution Time (s)
1.2
1.0
0.8
0.6
0.4
0.2
0.0
2.21e-06
1.54e-06
1.12e-05
BFS
DFS
IDDFS