# TASK 1:

Great, let's dive into the first task of your assignment. You're asked to apply **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** on trees and graphs. Both BFS and DFS are fundamental algorithms used for traversing or searching through graphs and trees. Here's a brief overview of each:

- **Breadth-First Search (BFS)**: This algorithm traverses the graph level by level. It starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores all of the neighbor nodes at the present depth prior to moving on to nodes at the next depth level[1].
- **Depth-First Search (DFS)**: This algorithm traverses the graph depth-wise. It starts at the root and explores as far as possible along each branch before backtracking[1].

Here's a simple example to illustrate both algorithms on a graph:

```
Graph:
    A
   / \
  B   C
 /   / \
D   E   F
```

**BFS Traversal**:

1. Start at node A.
2. Visit A's neighbors: B, C.
3. Visit B's neighbor: D.
4. Visit C's neighbors: E, F.

**Resulting BFS Order**: A, B, C, D, E, F

**DFS Traversal**:

1. Start at node A.
2. Visit A's neighbor: B.
3. Visit B's neighbor: D.
4. Backtrack to B, then to A, and visit A's other neighbor: C.
5. Visit C's neighbors: E, F.

**Resulting DFS Order**: A, B, D, C, E, F

You can practice these traversals using the graph editor provided by your instructor. To implement these algorithms, you can use pseudocode or write actual code in a programming language of your choice. If you need help with writing the code or understanding the algorithms in more depth, feel free to ask!

Would you like to see a code example in a specific programming language, or do you have any other questions on BFS and DFS?

# TASK 2:

## Step 1: Generate Random Unique Numbers

You need to generate five sets of unique numbers for the given ranges. In Python, you can use the `random.sample` function to achieve this.

```
import random

# Function to generate a set of unique numbers for a given range and size
def generate_unique_numbers(range_start, range_end, size):
    return random.sample(range(range_start, range_end), size)

# Generating the five sets
set_1000 = generate_unique_numbers(1, 100000, 1000)
set_40000 = generate_unique_numbers(1, 100000, 40000)
set_80000 = generate_unique_numbers(1, 100000, 80000)
set_200000 = generate_unique_numbers(1, 100000, 200000)
set_1000000 = generate_unique_numbers(1, 100000, 1000000)
```

## Step 2: Build Trees for Each Set

For simplicity, let's assume we're building binary search trees (BSTs) from these sets. You can create a BST class and insert the numbers.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

# Function to insert a value into the BST
def insert_into_bst(root, value):
    if root is None:
        return TreeNode(value)
    if value < root.value:
        root.left = insert_into_bst(root.left, value)
    else:
        root.right = insert_into_bst(root.right, value)
    return root

# Function to build a BST from a set of numbers
def build_bst(numbers):
    root = None
    for number in numbers:
        root = insert_into_bst(root, number)
    return root

# Building BSTs for each set
bst_1000 = build_bst(set_1000)
bst_40000 = build_bst(set_40000)
bst_80000 = build_bst(set_80000)
bst_200000 = build_bst(set_200000)
bst_1000000 = build_bst(set_1000000)
```

## Step 3: Apply BFS and DFS

You'll need to implement BFS and DFS algorithms to traverse the trees and find the goal.

```
from collections import deque
import time

# BFS implementation
def bfs(root, goal):
    if root is None:
        return False
    queue = deque([root])
    while queue:
        node = queue.popleft()
        if node.value == goal:
            return True
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return False

# DFS implementation
def dfs(root, goal):
    if root is None:
        return False
    if root.value == goal:
        return True
    return dfs(root.left, goal) or dfs(root.right, goal)
```

## Step 4: Calculate Execution Time and Find Goal

You will measure the time taken by BFS and DFS to find the goal in each tree.

```
# Function to calculate execution time for BFS and DFS
def calculate_execution_time(bst, goal):
    start_time = time.time()
    bfs_found = bfs(bst, goal)
    bfs_time = time.time() - start_time

    start_time = time.time()
    dfs_found = dfs(bst, goal)
    dfs_time = time.time() - start_time

    return bfs_time, dfs_time

# Calculate the goal for each set
goal_1000 = set_1000[-220]
goal_40000 = set_40000[-220]
goal_80000 = set_80000[-220]
goal_200000 = set_200000[-220]
goal_1000000 = set_1000000[-220]

# Calculate execution times
bfs_time_1000, dfs_time_1000 = calculate_execution_time(bst_1000,
goal_1000)
bfs_time_40000, dfs_time_40000 = calculate_execution_time(bst_40000,
goal_40000)
# ... Repeat for other sets
```

## Step 5: Create DataFrame and Plot Bar Chart

Finally, you'll create a DataFrame with the results and plot a bar chart.

```
import pandas as pd
import matplotlib.pyplot as plt

# Create a DataFrame
df = pd.DataFrame({
    'Tree Size': [1000, 40000, 80000, 200000, 1000000],
    'BFS Time': [bfs_time_1000, bfs_time_40000, bfs_time_80000,
bfs_time_200000, bfs_time_1000000],
    'DFS Time': [dfs_time_1000, dfs_time_40000, dfs_time_80000,
dfs_time_200000, dfs_time_1000000]
})

# Plot the bar chart
df.plot(x='Tree Size', y=['BFS Time', 'DFS Time'], kind='bar')
plt.ylabel('Seconds')
plt.title('BFS vs DFS Execution Time')
plt.show()
```

Remember to run these codes in a Jupyter Notebook environment, and make sure you have all the necessary libraries installed. If you encounter any issues or have questions about the code, feel free to ask for clarification. Good luck with your assignment!

# TASK 3:

To implement an A* search algorithm for the maze you've provided, you'll need to follow these steps:

1. **Define the Nodes**: Each empty tile in the maze is a node. Create a graph structure where each node represents a tile and is connected to its adjacent non-wall tiles.
2. **Set Start and Goal**: Identify the start node (A) and the goal node (Y).
3. **Create Heuristic Function**: Define a heuristic function that estimates the cost from the current node to the goal. A common heuristic for a grid is the Manhattan distance, calculated as the sum of the absolute differences of their coordinates.
4. **Initialize**: Create two sets, one for the nodes already evaluated (`closedSet`) and one for the nodes to be evaluated (`openSet`). Initially, `openSet` contains only the start node.
5. **Algorithm Loop**:
    - While the `openSet` is not empty, take the node with the lowest `fScore` (initially, the start node).
    - If this node is the goal, reconstruct the path from the goal to the start by following the `cameFrom` links.
    - Otherwise, move the current node from `openSet` to `closedSet`.
    - For each neighbor of the current node, if it's not in `closedSet`, calculate the `gScore` (the cost from the start to the neighbor through the current node).
    - If the new `gScore` is lower than the neighbor's known `gScore` or the neighbor is not in `openSet`, update the neighbor's `cameFrom`, `gScore`, and `fScore`.
    - Add the neighbor to `openSet` if it's not already there.

6. **Reconstruct Path**: Once you reach the goal, reconstruct the path from the goal to the start by following the `cameFrom` links.

Here's a pseudocode outline for the A* algorithm:

```
def A_star(start, goal, h):
    openSet = set([start])
    cameFrom = {}
    gScore = defaultdict(lambda: float('inf'))
    gScore[start] = 0
    fScore = defaultdict(lambda: float('inf'))
    fScore[start] = h(start)

    while openSet:
        current = min(openSet, key=lambda x: fScore[x])
        if current == goal:
            return reconstruct_path(cameFrom, current)

        openSet.remove(current)
        for neighbor in current.neighbors():
            if neighbor in closedSet:
                continue
            tentative_gScore = gScore[current] + d(current, neighbor)

            if neighbor not in openSet:
                openSet.add(neighbor)
            elif tentative_gScore >= gScore[neighbor]:
                continue

            cameFrom[neighbor] = current
            gScore[neighbor] = tentative_gScore
            fScore[neighbor] = gScore[neighbor] + h(neighbor)

    return failure

def reconstruct_path(cameFrom, current):
    total_path = [current]
    while current in cameFrom:
        current = cameFrom[current]
        total_path.append(current)
    return total_path[::-1]
```

Remember to replace `h` with your heuristic function and `d` with the function that calculates the cost between two nodes. The `reconstruct_path` function will give you the sequence of nodes from the start to the goal.

If you need further assistance with the code or understanding the algorithm, feel free to ask!

# TASK 4:

Alpha-Beta Pruning is an optimization technique for the Minimax algorithm, which is used in decision-making and game theory. It reduces the number of nodes that are evaluated in the

search tree, thus speeding up the decision-making process. Here's a Python implementation of the Alpha-Beta Pruning algorithm:

```
def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
    if depth == 0 or is_terminal(node):
        return get_heuristic_value(node)

    if maximizing_player:
        value = float('-inf')
        for child in get_children(node):
            value = max(value, alpha_beta_pruning(child, depth - 1, alpha,
beta, False))
            alpha = max(alpha, value)
            if alpha >= beta:
                break  # Beta cut-off
        return value
    else:
        value = float('inf')
        for child in get_children(node):
            value = min(value, alpha_beta_pruning(child, depth - 1, alpha,
beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break  # Alpha cut-off
        return value

# Helper functions
def is_terminal(node):
    # Implement your logic to check if the node is a terminal node
    pass

def get_heuristic_value(node):
    # Implement your logic to calculate the heuristic value of the node
    pass

def get_children(node):
    # Implement your logic to generate the children of the node
    pass
```

In this code:

- `node` represents the current position in the game.
- `depth` is the maximum depth of the tree that the algorithm will explore.
- `alpha` is the best value that the maximizer currently can guarantee at that level or above.
- `beta` is the best value that the minimizer currently can guarantee at that level or below.
- `maximizing_player` is a boolean indicating whether the current move is by a maximizing player.

You'll need to implement the `is_terminal`, `get_heuristic_value`, and `get_children` functions based on the specific game you're applying the algorithm to. These functions are crucial as they define the game's rules and the evaluation criteria for the game positions.

Remember to test your implementation thoroughly to ensure it works correctly. If you have any questions or need further assistance, feel free to ask!