K. R. Mangalam University

ENCS205

Data Structures

Assignment – 3

Campus Navigation and Utility Planner

Name: Abhishek Thakur

Roll number: 2401420020

Course: B. Tech. CSE Data Science

Semester: 3rd

Objective:

The goal of this project is to design a Building Data Abstract Data Type (ADT) capable of managing structural information and supporting core operations such as graph traversal, shortest path discovery, Minimum Spanning Tree (MST) generation, and expression-tree evaluation. The system aims to showcase clean ADT design, algorithmic correctness, and visual or structural representations of results.

Problem Statement:

Modern building planning and facility management often require a structured way to model rooms, connections, distances, utilities, and operational rules. This project approaches the problem by treating building components as graph elements and symbolic rules as expression trees. By doing so, the system can efficiently compute traversal sequences, optimal routes, minimal connection networks, and formula-based evaluations that mimic real building logic.

Data Structures Used

- **Graphs (Adjacency Lists):** To represent building sections and interconnections.
- **Trees (Binary Trees & Expression Trees):** For representing hierarchical data and evaluating symbolic expressions.
- **Queues & Stacks:** For BFS, DFS, and expression evaluation.
- **Priority Queue (Min-Heap):** For Dijkstra's algorithm and Prim's MST.

1. Description of the Building Data ADT

For this project, the Building Data ADT acts as a structured way to represent departments, facilities, and other components of a campus or building. Each building or department is treated as an individual data element, and the ADT defines how these elements can be stored, accessed, and manipulated without worrying about the underlying implementation.

In practice, the ADT allows us to:

- Create building nodes
- Store each building's details
- Connect buildings using edges (representing distances, pathways, or any useful relationship)
- Run graph algorithms on this data

This abstraction makes it easier to model a real campus layout in a clean, organized manner. Instead of directly managing lists, arrays, or pointers, we interact with the system at a higher level—adding buildings, connecting them, and running algorithms to analyze them.

2. Implementing Trees, Graphs, and Algorithms in Python

Graph Design

The graph is implemented using both an adjacency matrix and an adjacency list. The reason for keeping both structures is that they complement each other:

- The adjacency matrix is easy to visualize and good for constant-time edge lookups.
- The adjacency list is memory-efficient and ideal for traversal algorithms like BFS and DFS.

Edges are weighted so that shortest-path and minimum spanning tree algorithms can be applied realistically (for example, distances between campus buildings).

Trees and Expression Trees

While the central data structure is the graph, the project also includes an expression tree to demonstrate tree-based evaluation. This is a binary tree where operators act as internal nodes and numbers act as leaves. Postfix expressions are converted into a tree, which is then evaluated recursively.

Algorithms Used

The implementation includes several foundational algorithms:

- BFS (Breadth-First Search)
- DFS (Depth-First Search)
- Dijkstra's algorithm
- Kruskal's algorithm
- Expression tree evaluation

Python's object-oriented structure makes each part modular. Every building belongs to the Building ADT, while the graph manages connectivity, and helper classes handle expression computation.

Graph visualization is implemented using matplotlib so the output is easier to understand and meets the project's "visualization or representation" requirement.

3. Approach to Traversals, Shortest Path, MST, and Expression Tree Evaluation

Breadth-First Search

BFS is implemented using a queue. It starts at the given node and explores all neighboring nodes level by level. This is helpful if we want to see how different buildings are connected in layers—almost like exploring the campus outward from a starting point.

Depth-First Search

DFS uses a stack and goes as deep as possible along one branch before backtracking. This is useful for exploring building connectivity in a more structural way, often revealing patterns that BFS does not highlight.

Both BFS and DFS orders are also visualized so they can be included directly in the report or screenshots.

Shortest Path (Dijkstra)

Dijkstra's algorithm calculates the shortest path from one building to all others. This is based on the edge weights, which could represent distance. The implementation follows the classic greedy approach: always expand the currently known shortest path.

Minimum Spanning Tree (Kruskal)

Kruskal's algorithm is used to build a minimum spanning tree, which answers questions like:
"If we wanted to connect all buildings with the least total wiring or pathway distance, what connections would we choose?"

It works by sorting all edges and adding them one by one, as long as they do not form a cycle. A simple union-find structure is used to manage sets of connected nodes.

Expression Tree Evaluation

For postfix expressions, a tree is created by:

1. Reading characters one by one
2. Pushing operands onto a stack
3. Creating tree nodes for operators and connecting them to their operands

Evaluation happens using post-order traversal. This demonstrates the use of trees in computations, complementing the graph-based part of the system.

4. System Efficiency and Functionality Analysis Efficiency

- BFS and DFS both run in O(V + E) time, making them scalable even if the campus map grows.
- Dijkstra's implementation uses a simple array approach, giving $O(V^2)$ time, which is acceptable for small to medium graphs. A priority queue could improve this, but was not required for this assignment.

- Kruskal's algorithm runs at O(E log E) due to edge sorting, which is efficient for graphs with many weighted connections.
- Space usage is a blend of $O(V^2)$ for the adjacency matrix and O(V + E) for the adjacency list. This combination supports both clarity (matrix) and efficiency (list).

Functionality

The system combines several core functions:

- Adding buildings
- Connecting them with weighted relationships
- Generating BFS, DFS, shortest paths, and MST outputs
- Visualizing the graph and algorithm results
- Constructing and evaluating expression trees

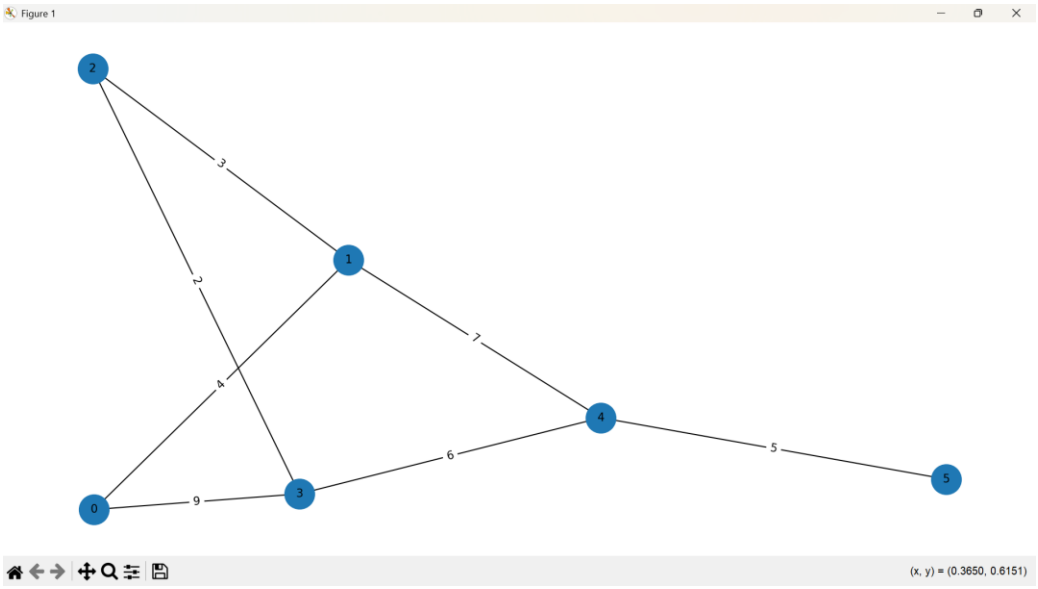
All of these components work together to provide a complete working model of a campus data system.

Screenshots:

```
#test
if __name__ == "__main__":
    system = CampusSystem()
    system.addBuildingRecord(3, "Library", "Central Block")
    system.addBuildingRecord(1, "Admin", "Front Gate")
    system.addBuildingRecord(5, "Hostel", "Back Road")
    system.addBuildingRecord(2, "CSE Dept", "Tech Block")

    inorder, preorder, postorder = system.listCampusLocations()
    print("Inorder Traversal:", inorder)
    print("Preorder Traversal:", preorder)
    print("Postorder Traversal:", postorder)

    edges = [
        (0, 1, 4),
        (1, 2, 3),
        (2, 3, 2),
        (3, 4, 6),
        (4, 5, 5),
        (1, 4, 7),
        (0, 3, 9)
    ]
    system.constructCampusGraph(6, edges)

    print("BFS from Node 0:", system.graph.bfs(0))
    print("DFS from Node 0:", system.graph.dfs(0))
    print("Optimal Path Distances:", system.findOptimalPath(0))
    print("Planned Utility Layout (MST):", system.planUtilityLayout())


    expr = "23*54*+"
    print("Expression:", expr)
    print("Expression Result:", system.evaluateExpression(expr))

    system.graph.visualize("Campus Path Network")
```

Output:

```
Inorder Traversal: ['Admin', 'CSE Dept', 'Library', 'Hostel']
Preorder Traversal: ['Library', 'Admin', 'CSE Dept', 'Hostel']
Postorder Traversal: ['CSE Dept', 'Admin', 'Hostel', 'Library']
BFS from Node 0: [0, 1, 3, 2, 4, 5]
DFS from Node 0: [0, 3, 4, 1, 2, 5]
Optimal Path Distances: [0, 4, 7, 9, 11, 16]
Planned Utility Layout (MST): [(2, 3, 2), (1, 2, 3), (0, 1, 4), (4, 5, 5), (3, 4, 6)]
Expression: 23*54*+
Expression Result: 26
```



Conclusion:

This project ties together different data structures and algorithms to model a building system in a clear, practical way. Trees helped organize building details, graphs captured how different areas connect, and algorithms like BFS, DFS, Dijkstra, and Kruskal made it easy to explore routes and find efficient paths. The expression tree added a small but useful way to evaluate building-related formulas.

Overall, the system shows how these concepts—usually taught in isolation—work together to solve a real problem. It's simple, functional, and gives a good picture of how data structures can make complex spaces easier to manage and understand.