City Emergency Response Management System Project

boxi li wanting wang
zizheng liu zihan liu shu li

contents

01

Introduction

This powerpoint details the design, implementation, and features of the City Emergency Response Management System. The project's core purpose is to provide an efficient and intuitive platform for managing urban emergencies by leveraging and comparing different priority queue data structures. Through this system, we demonstrate the practical application of various data structures, their performance characteristics, and their impact on realworld emergency management scenarios.

02

Project Structure

```
emergency_system/
├── emergency_response/
│   ├── data_structures/
│   │   ├── emergency.py       # Emergency class
│   │   ├── linked_list.py     # Linked list priority queue
│   │   ├── binary_tree.py     # Binary tree priority queue
│   │   └── heap.py            # Heap priority queue
│   ├── gui/
│   │   ├── interface.py       # Main GUI interface
│   │   ├── knn_visualization.py # KNN visualization interface
│   │   ├── statistics.py      # Statistics analysis interface
│   │   ├── emergency_simulation.py # Simulation module
│   │   └── main_app.py        # Main application interface
│   └── utils/
│       ├── data_loader.py     # Data loader utility
│       └── performance_analyzer.py # Performance analyzer utility
├── tests/
│   ├── test_emergency.py
│   ├── test_linked_list.py
│   ├── test_binary_tree.py
│   ├── test_heap.py
│   ├── test_data_loader.py
│   └── test_performance_analyzer.py
├── data/
│   └── emergency_dataset.csv   # Emergency dataset
└── main.py                     # Main program entry
```

03

Backend Design

3.1. Emergency Class

The `Emergency` class represents an emergency instance with the following attributes:

- `emergency_id`: Unique identifier for the emergency.
- `type`: Emergency type (using `EmergencyType` enum).
- `severity_level`: Severity (1-10, where 1 is the most severe/highest priority).
- `location`: Location description.
- `coordinates`: Location coordinates (x, y).

The class implements comparison operators to prioritize emergencies by severity, then by ID:

```python
def __lt__(self, other):
    # Lower severity level means higher priority
    if self.severity_level == other.severity_level:
        return self.emergency_id < other.emergency_id
    return self.severity_level < other.severity_level
```

Additional methods in the Emergency class include:

```python
def __eq__(self, other):
    if not isinstance(other, Emergency):
        return False
    return self.emergency_id == other.emergency_id

def __hash__(self):
    # Hash implementation allows emergency objects to be used in sets and as dictionary keys
    return hash(self.emergency_id)
```

5.2. Priority Queue Data Structures

5.2.1. Linked List Implementation (LinkedListPriorityQueue)

Our linked list implementation uses a custom `Node` class and maintains a sorted list where the highest priority element (lowest severity) is always at the head.

Node Implementation:

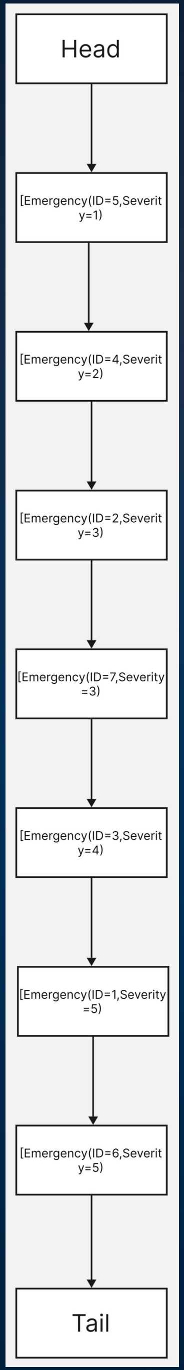
```python
class Node:
    def __init__(self, data):
        self.data = data  # Emergency object
        self.next = None  # Reference to next node
```

Key Methods:

```python
def enqueue(self, item):
    # Create a new node with the emergency data
    new_node = Node(item)

    # If queue is empty or new item has higher priority than head
    if self.head is None or item < self.head.data:
        new_node.next = self.head
        self.head = new_node
        self.size += 1
        return

    # Traverse the list to find the correct position
    current = self.head
    while current.next and item >= current.next.data:
        current = current.next

    # Insert the new node
    new_node.next = current.next
    current.next = new_node
    self.size += 1
```

```python
def dequeue(self):
    # Remove and return the highest priority item (at head)
    if self.is_empty():
        return None

    item = self.head.data
    self.head = self.head.next
    self.size -= 1
    return item
```

5.2. Priority Queue Data Structures



3.2.2. Binary Tree Implementation (BinaryTreePriorityQueue)

Our binary search tree implementation uses a custom `TreeNode` class and maintains a tree where nodes are ordered by emergency priority.

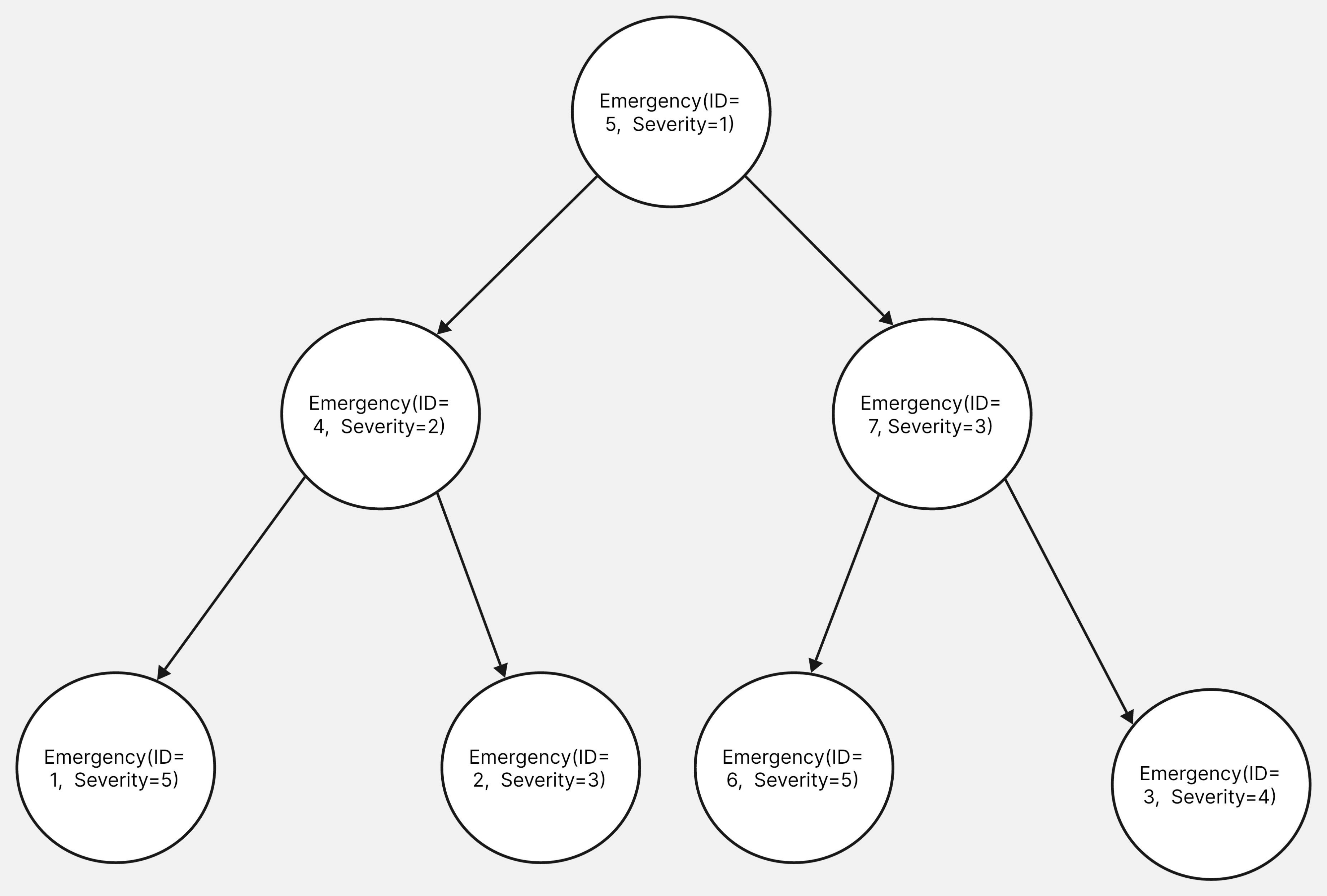
TreeNode Implementation:

```python
class TreeNode:
    def __init__(self, data):
        self.data = data  # Emergency object
        self.left = None  # Left child (higher priority)
        self.right = None  # Right child (lower priority)
```

Key Methods:

```python
def enqueue(self, item):
    # Recursive helper function to insert a node
    def _insert(node, item):
        if node is None:
            return TreeNode(item)

        if item < node.data:
            node.left = _insert(node.left, item)
        else:
            node.right = _insert(node.right, item)
        return node

    self.root = _insert(self.root, item)
    self.size += 1
def dequeue(self):
    # Remove and return the highest priority item (leftmost node)
    if self.is_empty():
        return None

    # Special case: only one node
    if self.root.left is None:
        item = self.root.data
        self.root = self.root.right
        self.size -= 1
        return item

    # Find the leftmost node and its parent
    parent = None
    current = self.root
    while current.left:
        parent = current
        current = current.left

    # Get the highest priority item
    item = current.data

    # Connect parent to the right child of the leftmost node
    parent.left = current.right

    self.size -= 1
    return item
```

3.2.3. Heap Implementation (HeapPriorityQueue)



3.2.2. Binary Tree Implementation (BinaryTreePriorityQueue)

Our binary search tree implementation uses a custom `TreeNode` class and maintains a tree where nodes are ordered by emergency priority.

TreeNode Implementation:

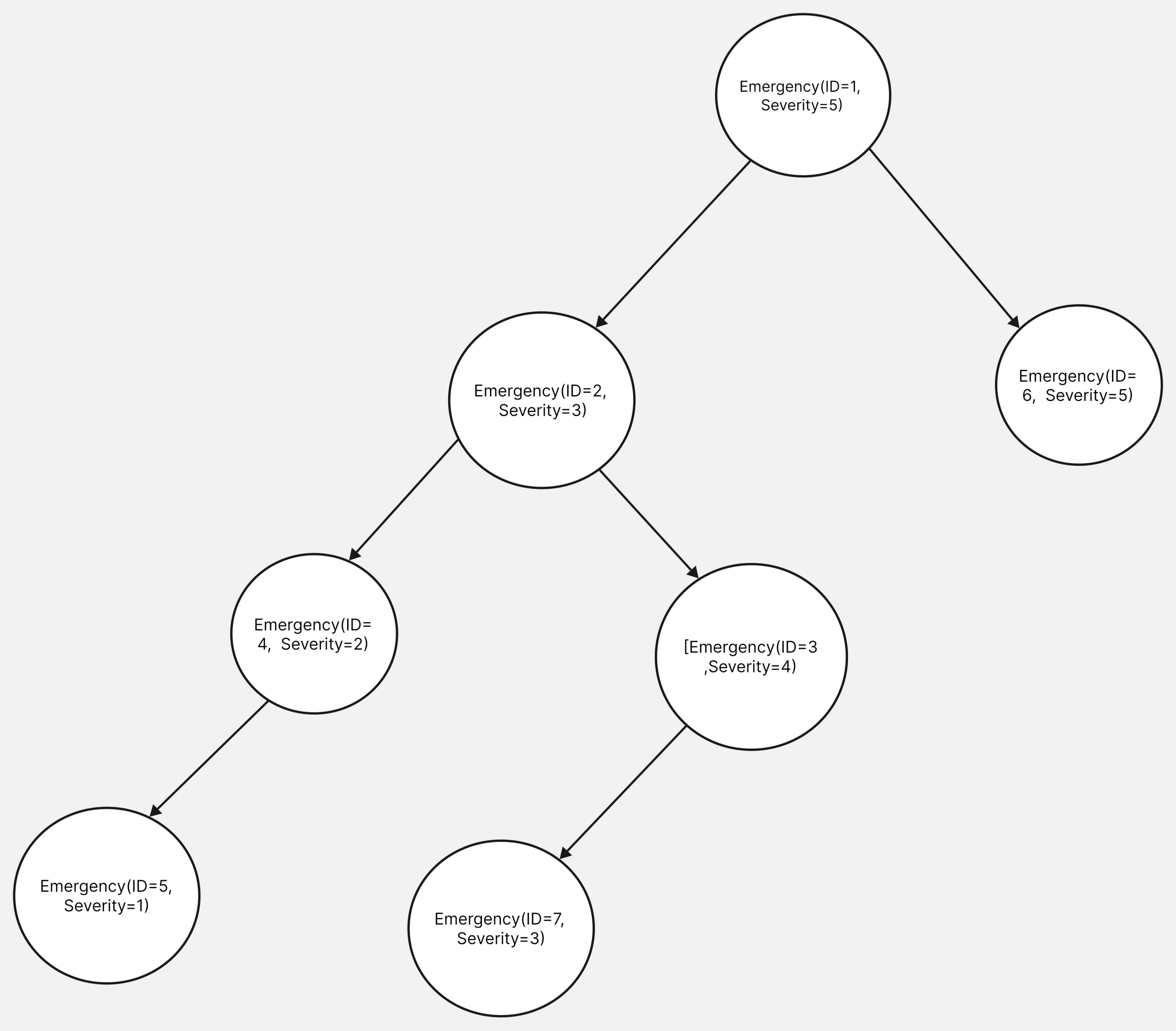
```python
class TreeNode:
    def __init__(self, data):
        self.data = data  # Emergency object
        self.left = None  # Left child (higher priority)
        self.right = None  # Right child (lower priority)
```

Key Methods:

```python
def enqueue(self, item):
    # Recursive helper function to insert a node
    def _insert(node, item):
        if node is None:
            return TreeNode(item)

        if item < node.data:
            node.left = _insert(node.left, item)
        else:
            node.right = _insert(node.right, item)
        return node

    self.root = _insert(self.root, item)
    self.size += 1
```

```python
def dequeue(self):
    # Remove and return the highest priority item (leftmost node)
    if self.is_empty():
        return None

    # Special case: only one node
    if self.root.left is None:
        item = self.root.data
        self.root = self.root.right
        self.size -= 1
        return item

    # Find the leftmost node and its parent
    parent = None
    current = self.root
    while current.left:
        parent = current
        current = current.left

    # Get the highest priority item
    item = current.data

    # Connect parent to the right child of the leftmost node
    parent.left = current.right

    self.size -= 1
    return item
```

Heap Property Maintenance:
```python
def _shift_up(self, index):
    # "Bubble up" operation to maintain heap property
    if index > 1:
        parent = index // 2
        if self._is_higher_priority(index, parent):
            self._swap(index, parent)
            self._shift_up(parent)

def _shift_down(self, index):
    # "Sift down" operation to maintain heap property
    if index <= self.count:
        left = 2 * index
        right = 2 * index + 1
        smallest = index

        if left <= self.count and self._is_higher_priority(left, smallest):
            smallest = left

        if right <= self.count and self._is_higher_priority(right, smallest):
            smallest = right

        if smallest != index:
            self._swap(index, smallest)
            self._shift_down(smallest)
```


Emergency(ID=1, Severity=5)
Emergency(ID=2, Severity=3)
Emergency(ID= 6, Severity=5)
Emergency(ID= 4, Severity=2)
[Emergency(ID=3 ,Severity=4)
Emergency(ID=5, Severity=1)
Emergency(ID=7, Severity=3)

3.3. Complexity Analysis

3.3.1. Time Complexity

Operation	Linked List	Binary Tree	Heap
Enqueue	O(n)	O(log n)	O(log n)
Dequeue	O(1)	O(log n)	O(log n)
Search	O(n)	O(n)	O(1)

3.3.2. Space Complexity

Data Structure	Space Complexity	Implementation Notes
Linked List	O(n)	Each node requires additional pointer overhead
Binary Tree	O(n)	Each node requires two child pointers
Heap	O(n)	Array-based implementation with additional mapping dictionary

Our measurements using the pympler library confirm these theoretical complexities, with some interesting observations:

- The linked list structure shows consistent memory usage regardless of the number of elements
- The binary tree's memory growth tapers off as the tree size increases
- The heap structure shows the most linear relationship between memory usage and data size

3.4. Design Rationale and Reflections on Optimization

In developing this project, we gave special consideration to how we used our data structures and the overall efficiency of the system.

3.4.1. Our Current Design and Its Justification

In the current implementation, we decided to maintain three separate instances of our priority queues (`LinkedList`, `BinarySearchTree`, and `Min-Heap`) simultaneously.

Our Rationale:

This decision was made primarily for comparative purposes. We wanted to build a system where a user could easily switch between different views and observe, in real-time, how each data structure behaves with the exact same dataset.

Acknowledged Trade-offs:

Data Redundancy: Every emergency object is stored three times, tripling the memory footprint.

Performance Overhead: Write operations are less efficient, as they must be executed three times.

3.4.2. Reflections on Future Optimizations

If we were to refactor this project for a real-world production environment, we would implement a more efficient design.

1. A Single Source of Truth: Use a hash map (a Python dictionary) for `O(1)` average time complexity lookups by `emergency_id`.
2. On-Demand Instantiation: Dynamically create the active priority queue from the hash map only when needed.
3. Balanced BST: Implement a self-balancing binary search tree (like AVL or Red-Black) to guarantee O(log n) operations.
4. Rebuilding on Switch: If the user switches views, dynamically build the new queue from the hash map.

This optimized design would eliminate data redundancy and restore the efficiency of write operations.

04

System Features and Use Cases

4.1. Main Features

1. Emergency Management: Add, process, and search for emergencies using three different priority queue implementations.
2. KNN Visualization: Visualize emergencies and response units on a map and recommend the nearest units.
3. Statistical Analysis: Analyze emergency type and severity distributions.
4. Performance Comparison: Compare the operational performance of the three data structures.
5. Emergency Dispatch Simulation: Simulate handling large-scale emergencies to compare data structure performance.
6. Space Complexity Analysis: Measure and visualize memory usage of different data structures.

4.2. Application Scenarios

1. **City Emergency Centers**: Managing and dispatching resources for various incidents.
2. **Hospitals**: Optimizing patient intake and resource allocation.
3. **Fire Departments**: Handling alarms and dispatching units efficiently.
4. **Police Departments**: Managing and responding to emergency calls.

05

GUI Overview

5.1. Interface Description

The GUI, built with tkinter, includes:

1.**Main Interface**: A central hub with access to all major functions.
2.**Emergency Management Interface**: Allows direct interaction with the priority queues.
3.**KNN Visualization Interface**: Maps emergencies and recommends response units.
4.**Statistical Analysis Interface:** Displays emergency data distributions and complexity information.
5. **Performance Comparison Interface**: Compares the performance of the data structures with charts.

5.2. Suggestions for Improvement

1. **Enhanced Visualization**: Implement heatmaps for emergency density or animations for response unit movements.
2. **Improved User Guidance**: Add an interactive tutorial or contextsensitive help tips.

06

Testing and Coverage

6.1. Testing Framework

The project uses Python's builtin unittest framework for all unit tests.

6.2. Test Coverage

The test suite covers all core data structures and utility functions, including edge cases and basic operations. The overall project test coverage is 89%, ensuring high reliability of the core functionalities.

| Test File | Coverage

test_emergency.py| Emergency class initialization and comparison

test_linked_list.py | Linked list queue operations and edge cases

test_binary_tree.py | Binary tree queue operations and structure

test_heap.py | Heap queue operations and performance

test_data_loader.py | Data loading and queue initialization

test_performance_analyzer.py| Performance analyzer functionality

07

Team Roles and Responsibilities

Name | Main Responsibilities

Li Boxi | Core architecture & data structures: Responsible for all core data structure implementations (binary_tree.py, heap.py, linked_list.py, emergency.py); main program architecture design (main.py); GUI and data structure integration; drawing data structure flowcharts

Liu Zhihan| GUI Development (Part 1): Develop main interface (interface.py); main application (main_app.py); emergency simulation module (emergency_simulation.py)

Liu Zizheng| GUI Development (Part 2): Develop custom dialogs (custom_dialogs.py); statistics module (statistics.py); KNN visualization module (knn_visualization.py)

Li Shu | Data Processing & Simulation: Develop data loader (data_loader.py); dataset preparation and management (emergency_dataset.csv); performance analyzer module (performance_analyzer.py)

| **Wang Wanting**| Testing & Documentation: Write all unit tests; system documentation; user manual (readme.md)

08

Daily Work Schedule

emergency response systems

Day 1: Project planning and requirement gathering.
Day 2: Designing the system architecture and data structures.
Day 3: Implementing the main application and GUI.
Day 4: Developing the KNN algorithm and visualization.
Day 5: System testing and integration.
Day 6: Debugging and bug fixing.
Day 7-9: Finalizing the project, creating the presentation, and writing this report.

09

Challenges and Lessons Learned

9.1. Main Challenges

1. **Data Structure Selection**: Balancing the performance tradeoffs between the linked list, binary tree, and heap.
2. **Data Consistency**: Ensuring data remained synchronized across all three data structures during operations.
3. **Performance Optimization**: Addressing potential bottlenecks, such as tree balancing and efficient heap updates.
4. **GUI Performance**: Ensuring the GUI remained responsive when visualizing large datasets.

9.2. Lessons Learned

1. Design for Edge Cases： The importance of considering special conditions, such as identical priorities.

2. TestDriven Development: High test coverage is crucial for ensuring code quality and reliability.

3.Balance Performance and Maintainability: Choose implementations appropriate for the specific use case.

4. Modular Design: A clear separation of concerns makes the project easier to extend and maintain.

10

Conclusion

emergency response systems

This City Emergency Response Management System successfully provides an effective solution for emergency management by implementing and comparing three distinct priority queue data structures. The system's combination of an intuitive visual interface and powerful analysis tools helps demonstrate key computer science concepts and offers a practical framework for optimizing emergency response.

THANKS