Elevator System Project

Celaleddin Ömer Sağlam / Salih Can Erer

Introduction

This project presents an elevator control system optimized to minimize both energy consumption and passenger waiting time. The system consists of multiple elevators managed by the `ElevatorSystem` class, which is responsible for processing passenger requests, updating elevator states, and running simulations to determine the best constants for system operation.

1 Class Overview: Elevator

The ‘Elevator‘ class models an elevator system with methods to manage stops, movement between floors, and tracking the elevator's state, such as idle status, current floor, and direction of travel.

1.1 Class Attributes

- **isMoving (bool)**: Indicates whether the elevator is currently in motion.
- **stops (std::vector¡int¿)**: A list of floors where the elevator is scheduled to stop.
- **currentFloor (int)**: The elevator's current floor.
- **passengers (int)**: Current number of passengers.
- **isGoingUp (bool)**: Direction of movement; true if moving up.
- **maxPQFloorTime** and **minPQFloorTime** (priority_queue<pair<int, int>>): Priority queues for tracking floors based on expected stop times.
- **MaxPQFloorEnergy** and **minPQFloorEnergy** (priority_queue<pair<int, int>>): Priority queues for energy optimization between stops.

1.2 Constructors

- **Elevator()**: Default constructor initializing the elevator at floor 0, with no passengers, and idle.
- **Elevator(int startFloor)**: Parameterized constructor to start the elevator at a specified floor.

1.3 Public Methods

- **void addStop(int floor)**: Adds a floor to the stop list, increments the passenger count, and sets the elevator in motion.
- **int distanceToFloor(int floor, int currentFloor) const**: Calculates and returns the distance in floors between the elevator's current floor and a specified floor.
- **bool isIdle() const**: Checks if the elevator is idle (i.e., not moving and has no stops).
- **int timeUntilStop()**: Estimates time until the next stop, based on current and next stop positions.
- **int move()**: Simulates one unit of movement towards the next stop and adjusts ‘currentFloor‘. Returns 1 if moved, or 0 if at a stop.

2 Detailed Method Explanations

2.1 addStop

Definition: Adds a stop at a specified floor and sets the elevator in motion if it isn't already. Increments the passenger count.

2.2 distanceToFloor

Definition: Calculates the absolute difference in floors between the elevator's current location and the target floor.

2.3 move

Definition: Controls movement towards the next stop. If at the next stop, it removes the stop and decrements the passenger count. If there are no more stops, sets the elevator as idle.

3 Class Overview: Passenger

The ‘Passenger‘ class models a passenger within the elevator system, including attributes and methods to manage passenger's origin and destination, calculate energy and waiting times, and select the most optimal elevator based on these metrics.

3.1 Class Attributes

- **currentFloor (int)**: Represents the current floor of the passenger.
- **destinationFloor (int)**: The floor the passenger wishes to reach.
- **isInsideElevator (bool)**: Indicates whether the passenger is inside the elevator.
- **cWT (int)**: Weighting constant for waiting time optimization.
- **cEC (int)**: Weighting constant for energy consumption optimization.
- **waitingTimes (vector¡int¿)**: A vector of calculated waiting times for available elevators.
- **consumedEnergies (vector¡int¿)**: Stores calculated energy consumption for each elevator.
- **summedUpWeights (vector¡pair¡int, int¿ ¿)**: Stores combined weights (waiting time and energy consumption) for each elevator to aid in optimal selection.

3.2 Constructors

- **Passenger(int currentFloor, int destinationFloor)**: Initializes the passenger's current and destination floors and sets `isInsideElevator` to false.

3.3 Public Methods

- **void setConstants(int cWT, int cEC)**: Configures constants used for weighting waiting time and energy consumption.
- **Elevator calculateWaitingTimeAndEnergy(vector¡Elevator¿ elevators)**: Calculates the optimal elevator based on waiting time and energy consumption. Returns the most suitable `Elevator` object.
- **int calculateEnergySingle(Elevator elevator)**: Estimates the energy needed for a specific elevator to serve the passenger.
- **int calculateWaitingTimeSingle(Elevator elevator)**: Determines the waiting time for a specific elevator to reach the passenger's current floor.

- **int timeInElevator(int currentFloor, int destinationFloor)**: Returns the time required for an elevator to travel between two floors.
- **Elevator findBestElevator(vector¡Elevator¿ elevators)**: Uses waiting time and energy consumption data to find and return the optimal elevator for the passenger.

4 Detailed Method Explanations

4.1 setConstants

Definition: Configures constants for optimizing both waiting time (`cWT`) and energy consumption (`cEC`), used in determining the best elevator for the passenger.

4.2 calculateWaitingTimeAndEnergy

Definition: Iterates through a list of elevators to calculate both waiting time and energy consumption for each. Uses `calculateWaitingTimeSingle` and `calculateEnergySingle` methods and then calls `findBestElevator` to return the most optimal elevator based on combined weights.

4.3 calculateEnergySingle

Definition: Computes energy consumption for an elevator to reach the passenger's current floor and then travel to the passenger's destination floor, adjusting calculations based on whether the elevator is moving up or down.

Algorithm: The method accounts for elevator direction and uses priority queues (MaxPQ and minPQ) to evaluate the highest and lowest floors in the elevator's current direction, thereby computing a realistic estimate of the energy required.

4.4 calculateWaitingTimeSingle

Definition: Calculates the waiting time for a specific elevator to reach the passenger's floor. Considers the elevator's current state (moving or idle) and direction to estimate an accurate waiting time.

4.5 findBestElevator

Definition: Iterates through the calculated waiting times and energy consumption values for each elevator, calculating a weighted sum based on `cWT` and `cEC`. The elevator with the lowest combined weight is selected as the best option.

5 ElevatorSystem Class Description

The `ElevatorSystem` class is designed to simulate and manage the operation of multiple elevators in a building. It efficiently handles passenger requests, optimizes energy consumption, and evaluates performance metrics. This class integrates several core algorithms to determine the best elevator for each request and to manage the movement of elevators effectively.

5.1 Overview

The main purpose of the `ElevatorSystem` is to ensure efficient transportation of passengers between floors while minimizing waiting times and energy consumption. The class achieves this through various algorithms and data structures, allowing it to adapt to changing conditions, such as fluctuating passenger demand.

5.2 Attributes

- `vector<Elevator> elevators`: A dynamic array of `Elevator` objects representing each elevator in the system. Each elevator maintains its own state, including its current position, direction of movement, and queue of stops.
- `int totalEnergyConsumed`: A cumulative measure of the total energy consumed by all elevators throughout the simulation. This metric is crucial for assessing the system's efficiency.
- `int totalWaitingTime`: A cumulative measure of the total waiting time experienced by all passengers. This statistic helps evaluate the performance of the elevator system from the passengers' perspective.
- `int numVisitors`: A counter that tracks the total number of passengers served during the simulation. This value is used to calculate averages and other performance metrics.

5.3 Constructor

`ElevatorSystem(int numElevators, int startingFloor)`:

- Initializes the elevator system with a specified number of elevators, all starting from the same floor.
- Sets `totalEnergyConsumed`, `totalWaitingTime`, and `numVisitors` to zero.
- Constructs each `Elevator` object and adds it to the `elevators` vector.

5.4 Key Methods

5.4.1 New Call Handling

`void newCall(int from, int to, Passenger pass)`:

- Handles new elevator calls from a specified `from` floor to a `to` floor for a given `Passenger`.
- Calls `pass.calculateWaitingTimeAndEnergy(elevators)` to determine the best elevator for the request based on current states and metrics. This function evaluates all elevators to find the one with the least waiting time and energy consumption.
- If a suitable elevator is found (indicated by `bestElevatorIndex`), it updates the chosen elevator's stop queue, incrementing the `numVisitors` counter.
- Adds the stop at the originating floor and the destination floor to the elevator's queue for processing.

5.4.2 Elevator Updates

`void updateElevators()`:

- Iterates through each elevator in the `elevators` vector.
- Calls the `move()` method on each elevator to update its position based on its current direction and queue.
- Updates the `totalEnergyConsumed` by adding the energy used during the move.
- Calculates and adds to the `totalWaitingTime` based on the number of passengers currently in the elevator and the distance moved.

5.4.3 Finding Optimal Constants

`pair<int,int> findBestConstants()`:

- This method attempts to optimize the constants used for calculating waiting time and energy consumption.
- It generates random values for `cWT` (waiting time constant) and `cEC` (energy consumption constant) and simulates multiple scenarios to find the best combination.
- The method runs 100 simulations, each with random passenger requests, and stores the results in a vector.
- After sorting the results, it returns the pair of constants that yielded the best performance, defined by the sum of waiting time and energy constants.

5.4.4 Simulation Execution

`void runSimulation(int numFloors, double visitorFrequency, int totalTime, Passenger passenger)`:

- This method executes the entire simulation for a specified duration (`totalTime`).
- It randomly generates passenger requests based on the `visitorFrequency`.
- Each time a new passenger is generated, it calls `newCall()` to process the request.
- After processing the requests, it updates the elevators through `updateElevators()` to reflect their movements and status.
- The simulation runs until the total time elapses, simulating real-time operations of the elevator system.

5.4.5 Statistics Reporting

`void printStatistics() const`:

- Outputs the results of the simulation to the console, displaying the total energy consumed and the average waiting time for passengers.
- Calculates average waiting time by dividing `totalWaitingTime` by `numVisitors`, ensuring to handle the case where no visitors have been served.

5.5 Algorithms Used

The `ElevatorSystem` utilizes several algorithms for its operations:

- **Best Elevator Selection Algorithm**: This algorithm evaluates all available elevators to determine which one can best serve a new passenger request. It considers factors such as waiting time and energy consumption. The elevator that minimizes these factors is chosen for the request.
- **Simulation Algorithm**: This algorithm generates passenger requests based on a defined frequency and simulates the passing of time in the elevator system. It incorporates random elements to model realistic visitor behavior and updates the system accordingly.
- **Energy Consumption and Waiting Time Calculation**: During each elevator movement, the energy consumed is calculated based on the distance traveled, and waiting time is updated based on the number of passengers currently being served. This is crucial for performance evaluation.
- **Sorting Algorithm for Constant Optimization**: After running multiple simulations with different constants, the results are sorted to identify the optimal constants for waiting time and energy consumption. This is done using a simple comparison-based sorting approach.

5.6 Conclusion

The `ElevatorSystem` class provides a robust framework for managing elevator operations in a simulated environment. Through its various algorithms and methods, it ensures efficient handling of passenger requests while optimizing key performance metrics such as energy consumption and waiting time. This implementation serves as a foundational model for further enhancements and real-world applications in elevator management systems.

6 Overall Optimization Algorithm for Elevator System

6.1 Algorithm: Optimize Elevator Performance

1. **Initialization**:
   - Define the number of simulations to run (e.g., `numSimulations`).
   - Create a vector to store the results of each simulation.
2. **Define Constants**:
   - Set initial ranges for the waiting time constant (`cWT`) and the energy consumption constant (`cEC`).
   - These constants influence how waiting time and energy consumption are calculated and can be varied during optimization.
3. **Randomly Generate Passenger Requests**:
   - For each simulation, do the following:
     (a) **Randomly Set Constants**:
       – Generate random values for `cWT` and `cEC` within defined ranges.
     (b) **Initialize Elevator System**:
       – Create an instance of the `ElevatorSystem` with a predefined number of elevators and starting floors.
     (c) **Run the Simulation**:
       – Call the `runSimulation()` method of the `ElevatorSystem` instance, passing in the random constants, number of floors, visitor frequency, and total simulation time.
       – This method generates passenger requests based on the specified frequency and tracks the performance of the elevator system.
4. **Collect Results**:
   - After running the simulation, collect the following performance metrics:
     – Total energy consumed.
     – Average waiting time for passengers.
     – Number of visitors served.
5. **Evaluate Performance**:
   - Calculate a performance score for the current constants:
     – Combine the total energy consumed and average waiting time using a weighted formula.

- For example, define a score as follows:

$$\text{score} = \text{average waiting time} + \alpha \cdot \text{total energy consumed}$$

where α is a weight factor that balances the importance of waiting time versus energy consumption.

6. **Store Results**:
   - Store the tuple of $(\text{cWT}, \text{cEC}, \text{score})$ in the results vector.
7. **Sort Results**:
   - Sort the results vector based on the performance score.
   - This can be done using a simple sorting algorithm (e.g., quicksort) or using STL sort functionality in C++.
   - The result with the lowest score is deemed the best combination of constants.
8. **Select Optimal Constants**:
   - Retrieve the first element from the sorted results vector, which contains the optimal constants:

$$\text{optimalConstants} = \text{results}[0]$$

9. **Output Optimal Constants**:
   - Print or return the optimal constants for further use in the elevator system.
10. **Terminate Optimization**:
    - End the optimization process and implement the optimal constants in the elevator operation calculations.

6.2 Pseudocode Representation

6.3 Explanation of the Optimization Process

- **Random Sampling**: The algorithm employs a random sampling technique to explore a wide range of constant values (`cWT` and `cEC`). This randomness helps to discover potentially effective combinations that may not be evident through systematic approaches.
- **Performance Evaluation**: The performance score incorporates both energy consumption and waiting time, allowing for a balanced evaluation of elevator performance. By adjusting the weight factor α, you can prioritize either energy efficiency or passenger satisfaction based on specific needs.

Algorithm 1 Optimize Elevator Performance

```
function OptimizeElevatorPerformance():
  numSimulations ← 100
  results ← []
for i from 1 to numSimulations do
    cWT ← random value in [min_cWT, max_cWT]
    cEC ← random value in [min_cEC, max_cEC]
    elevatorSystem ← new ElevatorSystem(numElevators, startingFloor)
    elevatorSystem.runSimulation(numFloors, visitorFrequency, totalTime,
  passenger)
    totalEnergy ← elevatorSystem.totalEnergyConsumed
    averageWaitingTime ← elevatorSystem.totalWaitingTime / elevatorSys-
  tem.numVisitors
    score ← averageWaitingTime + α * totalEnergy
    results.push((cWT, cEC, score))
end for
results.sort by score
optimalConstants ← results[0]
print("Optimal Constants:", optimalConstants)
```

- **Result Sorting**: By sorting the results based on the performance score, the algorithm ensures that the best-performing constant values are selected for actual implementation, leading to improved overall system efficiency.

7 Acknowledgments

This work has been created under considerable time constraints, and we acknowledge that our proficiency in C++ is still developing. As students balancing multiple responsibilities, including exams and quizzes, we have done our best to implement the elevator system and its optimization features.

Unfortunately, we have not found a suitable time window to thoroughly test the code, which may impact its functionality. We appreciate your understanding as we navigate these challenges while striving to enhance our programming skills and deepen our knowledge in computer science.

This project has been a valuable learning experience, and we look forward to further refining our abilities in future endeavors.

Thank you for your support and consideration.