SISTEMA DE SIMULAÇÃO DE ELEVADORES: MODELAGEM, ESTRUTURAS DE DADOS E ALGORITMOS

RESUMO

Este documento técnico apresenta a análise detalhada do Sistema de Simulação de Elevadores, uma aplicação Java que modela e simula o comportamento de múltiplos elevadores em um edifício com diversos andares. O sistema implementa diferentes estratégias heurísticas para otimização, incluindo minimização do tempo de espera dos usuários e redução do consumo de energia. São descritas as estruturas de dados personalizadas (Lista, Fila e FilaPrioridade), os algoritmos de controle implementados e a arquitetura geral do sistema. A simulação permite configurar parâmetros como número de andares, elevadores, capacidade, tipos de painel de controle e diversos aspectos energéticos e temporais. A interface gráfica desenvolvida com Swing possibilita a visualização do comportamento do sistema em tempo real, facilitando a análise dos resultados e comparação entre diferentes estratégias heurísticas.

1. INTRODUÇÃO

A simulação computacional de sistemas de elevadores representa uma área importante tanto para o

ensino de estruturas de dados e algoritmos quanto para o desenvolvimento e teste de estratégias de controle em edifícios reais. Este documento apresenta uma análise técnica do Sistema de Simulação de Elevadores, um projeto desenvolvido em Java que implementa uma simulação completa do funcionamento de múltiplos elevadores em um edifício.

O sistema analisado é uma aplicação robusta que permite configurar diversos parâmetros de simulação, como número de andares, quantidade de elevadores, tipos de painéis de controle, capacidade dos elevadores, tempo de deslocamento e diferentes modelos heurísticos para tomada de decisão. A simulação permite analisar o comportamento do sistema sob diferentes condições e comparar a eficiência das estratégias implementadas.

A implementação do sistema segue princípios de programação orientada a objetos, com uso de interfaces, herança, polimorfismo e encapsulamento. Além disso, o sistema utiliza estruturas de dados personalizadas, implementadas do

zero, em vez de utilizar as coleções padrão da linguagem Java, o que proporciona uma visão aprofundada sobre o funcionamento dessas estruturas.

O estudo deste sistema é relevante não apenas do ponto de vista educacional, mas também prático, pois os algoritmos de controle de elevadores têm impacto significativo na eficiência energética e no tempo de espera dos usuários em edifícios reais. A análise apresentada neste documento abrange a modelagem do sistema, as estruturas de dados utilizadas, os algoritmos implementados e a interface gráfica desenvolvida.

2. MODELAGEM DO SISTEMA

2.1. Arquitetura Geral

O Sistema de Simulação de Elevadores foi desenvolvido seguindo o padrão arquitetural Model-View-Controller (MVC), proporcionando uma clara separação entre os componentes de modelo, visualização e controle. Esta arquitetura facilita a manutenção, extensão e teste do sistema.

A arquitetura do sistema pode ser dividida em três camadas principais:

1. **Camada de Modelo (Model)**: Contém as classes que representam as entidades do domínio e implementam as estruturas de dados utilizadas.
    - Pacote com.erasmo.structure: Contém as classes Predio, Andar, Elevador, Pessoa, PainelChamadas e PainelElevador.
    - Pacote com.erasmo.tads: Contém as implementações das estruturas de dados Lista, Fila e FilaPrioridade.
    - Pacote com.erasmo.enums: Contém as enumerações Direcao, ModeloHeuristica e TipoPainel.
2. **Camada de Controle (Controller)**: Responsável pela lógica de negócio e coordenação das entidades do sistema.
    - Pacote com.erasmo.controls: Contém as classes CentralDeControle e Simulador.
3. **Camada de Visualização (View)**: Implementa a interface gráfica do sistema.
    - Pacote com.erasmo.graphics: Contém as classes ConfiguracaoSimulador e SimuladorGI.
    - Pacote com.erasmo.relatory: Contém a classe RelatorioSimulacao.

2.2. Principais Entidades

O sistema é composto por diversas entidades que representam os elementos físicos e lógicos de um sistema de elevadores. A seguir, são descritas as principais entidades:

1. **EntidadeSimulavel (Classe Abstrata)**
    - Classe base que define o comportamento comum para entidades que participam da simulação.
    - Método abstrato atualizar(int minutoSimulado) que deve ser implementado pelas subclasses.
2. **Predio**
    - Representa o edifício contendo andares e elevadores.
    - Gerencia a lista de andares e a central de controle.
    - Implementa EntidadeSimulavel, atualizando seu estado a cada ciclo da simulação.
    - Principais atributos:
        - numeroAndares: Quantidade de andares no prédio.
        - andares: Lista de objetos Andar.
        - centralDeControle: Referência para a central de controle.
        - elevadores: Lista de objetos Elevador.
3. **Andar**
    - Representa um andar do prédio.
    - Mantém uma fila de pessoas esperando pelos elevadores.
    - Contém um painel de chamadas para solicitar elevadores.
    - Principais atributos:
        - numero: Número do andar.
        - painelChamadas: Referência para o painel de chamadas.
        - filaPessoas: Fila de prioridade contendo as pessoas esperando.
4. **Elevador**
    - Representa um elevador físico no prédio.
    - Responsável por sua própria movimentação e controle de passageiros.
    - Implementa EntidadeSimulavel.

- Principais atributos:
    - id: Identificador único do elevador.
    - andarAtual: Andar onde o elevador está no momento.
    - andarDestino: Próximo andar de destino (-1 se não houver).
    - direcao: Direção atual de movimento (SUBINDO, DESCENDO, PARADO).
    - passageiros: Lista de pessoas dentro do elevador.
    - emMovimento: Flag indicando se o elevador está em movimento.
    - capacidadeMaxima: Número máximo de passageiros.

5. **Pessoa**
    - Representa um usuário do elevador.
    - Mantém informações sobre origem, destino e características especiais.
    - Principais atributos:
        - id: Identificador único da pessoa.
        - andarOrigem: Andar onde a pessoa entrou no sistema.
        - andarDestino: Andar para onde a pessoa deseja ir.
        - cadeirante: Flag indicando se a pessoa é cadeirante.
        - idoso: Flag indicando se a pessoa é idosa.
        - tempoEspera: Tempo acumulado de espera.
        - dentroDoElevador: Flag indicando se está dentro de um elevador.
6. **PainelChamadas**
    - Representa o painel de botões em um andar para chamar elevadores.
    - Varia conforme o tipo de painel configurado.
    - Principais atributos:
        - tipoPainel: Tipo de painel (UNICO_BOTAO, DOIS_BOTOES, PAINEL_NUMERICO).
        - chamadaSubida: Flag indicando se

há chamada para subir.

- chamadaDescida: Flag indicando se há chamada para descer.
- chamadasPorAndar: Array de booleanos para painel numérico.

7. **PainelElevador**
    - Representa o painel interno do elevador.
    - Permite que passageiros selecionem seu andar de destino.
    - Principais atributos:
        - botoesAndares: Array de booleanos indicando andares selecionados.
        - tipoPainelConfig: Configuração do tipo de painel.
8. **CentralDeControle**
    - Gerencia a distribuição dos elevadores para atender chamadas.
    - Implementa as estratégias heurísticas de controle.
    - Principais atributos:
        - elevadores: Lista dos elevadores sob controle.
        - modeloHeuristica: Estratégia de controle utilizada.
        - elevadorDesignadoPara: Matriz que mapeia elevadores a andares.
        - andaresPendentes: Lista de andares com chamadas não atendidas.
9. **Simulador**
    - Coordena toda a simulação, controlando o tempo e os ciclos.
    - Principais atributos:
        - predio: Referência para o prédio sendo simulado.
        - tempoSimulado: Tempo atual da simulação em minutos.
        - simulacaoAtiva: Flag indicando se a simulação está em andamento.
        - tempoLimiteSimulacao: Limite máximo de tempo para a simulação.

2.3. Relacionamentos entre Entidades

As entidades do sistema se relacionam formando uma estrutura hierárquica e de composição.

Os principais relacionamentos são:

- O **Simulador** contém exatamente um **Predio**.
- O **Predio** contém múltiplos **Andares** e uma **CentralDeControle**.
- A **CentralDeControle** gerencia múltiplos **Elevadores**.
- Cada **Andar** contém um **PainelChamadas** e múltiplas **Pessoas** (na fila).
- Cada **Elevador** contém um **PainelElevador** e múltiplas **Pessoas** (passageiros).

Estes relacionamentos refletem a estrutura de um sistema de elevadores real, onde o prédio contém andares e elevadores, e as pessoas se movem dos andares para os elevadores e vice-versa.

2.4. Enumerações e Tipos

O sistema utiliza enumerações para representar estados discretos e configurações. As principais enumerações são:

1. **Direcao (Direcao.java)**
    - Representa a direção de movimento de um elevador.
    - Valores: SUBINDO, DESCENDO, PARADO.
    - Métodos:
        - oposto(): Retorna a direção oposta.
        - obterDirecaoPara(int andarAtual, int andarDestino): Calcula a direção para ir de um andar a outro.
        - toString(): Retorna uma representação textual da direção (↑, ↓, •).
2. **ModeloHeuristica (ModeloHeuristica.java)**
    - Define o modelo de tomada de decisão para a central de controle.
    - Valores:
        - SEM_HEURISTICA: Atendimento na ordem de chegada.
        - OTIMIZACAO_TEMPO_ESPERA: Prioriza minimizar o tempo de espera dos usuários.
        - OTIMIZACAO_CONSUMO_ENERGIA: Prioriza minimizar o consumo de energia.
3. **TipoPainel (TipoPainel.java)**
    - Define o tipo de painel de controle para chamada de elevadores.
    - Valores:
        - UNICO_BOTAO: Um único botão

para chamar o elevador.

- DOIS_BOTOES: Dois botões (subir/descer) para indicar a direção desejada.
- PAINEL_NUMERICO: Botões numéricos para selecionar diretamente o andar de destino.

Estas enumerações são utilizadas em todo o sistema para representar estados e configurações de forma clara e consistente, facilitando o desenvolvimento e a manutenção do código.

3. ESTRUTURAS DE DADOS

3.1. Implementações Personalizadas

O sistema implementa estruturas de dados genéricas personalizadas em vez de utilizar as coleções padrão do Java. Essa abordagem proporciona maior controle sobre a implementação e serve como um exercício educativo sobre estruturas de dados.

3.1.1. Lista (Lista.java)

A classe Lista<T> implementa uma lista encadeada simples genérica, que pode armazenar elementos de qualquer tipo. A implementação utiliza uma classe interna No<T> para representar os nós da lista encadeada.

Principais características:

- **Estrutura**: Lista encadeada simples com referências para o nó inicial e final.
- **Classe interna**: No<T> para representar cada elemento da lista.
- **Atributos principais**:
  - inicio: Referência para o primeiro nó.
  - fim: Referência para o último nó.
  - tamanho: Contador de elementos.

Principais operações:

- **adicionar(T elemento)**: Adiciona um elemento ao final da lista. Complexidade: O(1).
- **adicionar(T elemento, int posicao)**: Adiciona um elemento em uma posição específica. Complexidade: O(n).
- **remover(int posicao)**: Remove um elemento por posição. Complexidade: O(n).
- **remover(T elemento)**: Remove um elemento por valor. Complexidade: O(n).
- **obter(int posicao)**: Recupera um elemento por posição. Complexidade: O(n).
- **contem(T elemento)**: Verifica se um elemento está na lista. Complexidade: O(n).
- **vazia()**: Verifica se a lista está vazia. Complexidade: O(1).

- **tamanho()**: Retorna o número de elementos na lista. Complexidade: O(1).
- **limpar()**: Remove todos os elementos da lista. Complexidade: O(1).

Pseudocódigo das principais operações:

Método adicionar(elemento):

Criar novoNo com elemento

Se lista vazia:

inicio = novoNo

fim = novoNo

Senão:

fim.proximo = novoNo

fim = novoNo

Incrementar tamanho

Método remover(posicao):

Se lista vazia ou posicao inválida:

Lançar exceção

Se posicao == 0:

elementoRemovido = inicio.dado

inicio = inicio.proximo

Se inicio == null:

fim = null

Senão:

Percorrer até o nó anterior à posição

elementoRemovido = nó.proximo.dado

Atualizar referências para remover o nó

Se o nó removido era o último:

Atualizar referência fim

Decrementar tamanho

Retornar elementoRemovido

3.1.2. Fila (Fila.java)

A classe Fila<T> implementa uma fila genérica, seguindo o princípio FIFO (First-In-First-Out), onde o primeiro elemento adicionado é o primeiro a ser removido.

Principais características:

- **Estrutura**: Baseada em lista encadeada com referências para o primeiro e último elemento.
- **Classe interna**: No<T> para representar cada elemento da fila.
- **Atributos principais**:
  - inicio: Referência para o primeiro nó.
  - fim: Referência para o último nó.
  - tamanho: Contador de elementos.

Principais operações:

- **enfileirar(T elemento)**: Adiciona um elemento ao final da fila. Complexidade: O(1).

- **desenfileirar()**: Remove e retorna o primeiro elemento da fila. Complexidade: O(1).
- **primeiro()**: Retorna o primeiro elemento sem removê-lo. Complexidade: O(1).
- **vazia()**: Verifica se a fila está vazia. Complexidade: O(1).
- **tamanho()**: Retorna o número de elementos na fila. Complexidade: O(1).
- **limpar()**: Remove todos os elementos da fila. Complexidade: O(1).
- **contem(T elemento)**: Verifica se um elemento está na fila. Complexidade: O(n).
- **paraLista()**: Converte a fila em uma Lista. Complexidade: O(n).

Pseudocódigo das principais operações:

Método enfileirar(elemento):

Criar novoNo com elemento

Se fila vazia:

inicio = novoNo

Senão:

fim.proximo = novoNo

fim = novoNo

Incrementar tamanho

Método desenfileirar():

Se fila vazia:

Lançar exceção

elementoRemovido = inicio.dado

inicio = inicio.proximo

Decrementar tamanho

Se inicio == null:

fim = null

Retornar elementoRemovido

3.1.3. FilaPrioridade (FilaPrioridade.java)

A classe FilaPrioridade<T> implementa uma fila de prioridade genérica, onde elementos com maior prioridade são atendidos primeiro, independentemente da ordem de chegada.

Principais características:

- **Estrutura**: Baseada em lista encadeada ordenada por prioridade.
- **Classe interna**: No<T> para representar cada elemento da fila, incluindo sua prioridade.
- **Atributos principais**:
  - inicio: Referência para o primeiro nó.
  - tamanho: Contador de elementos.

Principais operações:

- **enfileirar(T elemento, int prioridade)**: Adiciona um elemento com uma prioridade. Complexidade: O(n).

- **desenfileirar()**: Remove e retorna o elemento de maior prioridade. Complexidade: O(1).
- **primeiro()**: Retorna o elemento de maior prioridade sem removê-lo. Complexidade: O(1).
- **vazia()**: Verifica se a fila está vazia. Complexidade: O(1).
- **tamanho()**: Retorna o número de elementos na fila. Complexidade: O(1).
- **limpar()**: Remove todos os elementos da fila. Complexidade: O(1).
- **contem(T elemento)**: Verifica se um elemento está na fila. Complexidade: O(n).
- **paraLista()**: Converte a fila em uma Lista. Complexidade: O(n).

Pseudocódigo da operação enfileirar:

Método enfileirar(elemento, prioridade):

Criar novoNo com elemento e prioridade

Se fila vazia ou prioridade > inicio.prioridade:

// Inserir no início

novoNo.proximo = inicio

inicio = novoNo

Senão:

// Encontrar posição correta baseada na prioridade

atual = inicio

Enquanto atual.proximo não for null e atual.proximo.prioridade >= prioridade:

atual = atual.proximo

// Inserir entre atual e atual.proximo

novoNo.proximo = atual.proximo

atual.proximo = novoNo

Incrementar tamanho

3.2. Uso das Estruturas de Dados no Sistema

As estruturas de dados implementadas são utilizadas em diversas partes do sistema:

1. **Lista**:
   - Em Predio para armazenar a lista de andares e elevadores.
   - Em Elevador para armazenar os passageiros.
   - Em CentralDeControle para manter a lista de elevadores e andares pendentes.
   -
2. **Fila**:
   - Não é diretamente utilizada nas classes principais, mas está disponível como parte da implementação.

3. **FilaPrioridade**:
    - Em Andar para gerenciar a fila de pessoas esperando, ordenadas por prioridade.
    - Pessoas com necessidades especiais (cadeirantes e idosos) recebem maior prioridade.

A implementação dessas estruturas específicas permite um maior controle sobre o comportamento do sistema e facilita a adaptação para requisitos específicos. Por exemplo, a FilaPrioridade é crucial para implementar políticas de acessibilidade, garantindo que pessoas com necessidades especiais sejam atendidas primeiro.

Exemplos de uso das estruturas no código:

1. Lista de andares em Predio:

```
// Em Predio.java
this.andares = new Lista<>();
// Adicionando andares
for (int i = 0; i < numeroAndares; i++) {
  Andar andar = new Andar(i, numeroAndares, tipoPainel);
  andares.adicionar(andar);
}
```

2. FilaPrioridade em Andar:

```
// Em Andar.java
this.filaPessoas = new FilaPrioridade<>();
// Adicionando pessoa com prioridade baseada em características
filaPessoas.enfileirar(pessoa, pessoa.calcularPrioridade());
```

3.3. Análise de Complexidade

A tabela abaixo resume a complexidade das principais operações nas estruturas de dados implementadas:

Lista	Remoção	O(n)	Precisa percorrer até encontrar o elemento
Lista	Acesso	O(n)	Precisa percorrer até a posição desejada
Fila	Enfileirar	O(1)	Inserção sempre no final
Fila	Desenfileirar	O(1)	Remoção sempre do início
FilaPrioridade	Enfileirar	O(n)	Precisa encontrar a posição correta
FilaPrioridade	Desenfileirar	O(1)	Remoção sempre do início (maior

prioridade)

As implementações utilizadas são adequadas para o contexto da aplicação, considerando o número tipicamente limitado de andares e elevadores em um prédio. No entanto, para sistemas com um número muito grande de elementos, estruturas de dados mais eficientes poderiam ser consideradas, como árvores binárias de busca balanceadas para a fila de prioridade.

4. ALGORITMOS IMPLEMENTADOS

4.1. Algoritmo de Movimentação de Elevadores

O algoritmo de movimentação dos elevadores é implementado no método atualizar(int minutoSimulado) da classe Elevador. Este algoritmo coordena o movimento do elevador entre andares e a tomada de decisão sobre próximos destinos.

Funcionamento do algoritmo:

1. **Parte 1: Movimentação**
    - Se o elevador está em movimento (emMovimento == true):
        - Decrementar o tempo para o próximo andar (tempoParaProximoAndar--).
        - Quando o tempo de deslocamento acabar (tempoParaProximoAndar <= 0):
            - Atualizar a posição do elevador baseado na direção.
            - Verificar se chegou ao destino:
                - Se sim, marcar como parado e executar procedimento de parada.
                - Se não, reiniciar temporizador para o próximo trecho de

deslocamento.

2. **Parte 2: Decisão (apenas se não estiver em movimento)**
    - Se tem um destino e não está no destino, começa a se mover.
    - Se não tem destino e não há botões pressionados no painel, procura novo destino.

Pseudocódigo do algoritmo de movimentação:

```
Método atualizar(minutoSimulado):
  // PARTE 1: MOVIMENTAÇÃO
  Se emMovimento:
    tempoParaProximoAndar--

    Se tempoParaProximoAndar <= 0:
      // Atualizar posição do elevador
      Se direcao == SUBINDO:
        andarAtual++
      Senão:
        andarAtual--

      // Verificar se chegou ao destino
      Se andarAtual == andarDestino:
        emMovimento = false
        pararNoAndar()
      Senão:
        // Continuar em movimento para o próximo andar
        tempoParaProximoAndar = calcularTempoDeslocamento()
  // PARTE 2: DECISÃO
  Senão:
    Se andarDestino != -1 e andarAtual != andarDestino:
      iniciarMovimento()
    Senão se andarDestino == -1 e não tem botões pressionados:
      determinarProximoDestino()
```

Procedimento de parada (método pararNoAndar()):

```
Método pararNoAndar():
  // Registrar consumo de energia
  consumoEnergiaTotal += consumoEnergiaPorParada

  // Liberar passageiros que chegaram ao destino
  liberarPassageiros()

  // Resetar o botão deste andar no painel
painelInterno.resetarBotao(andarAtual)

  // Determinar próximo destino
```

proximoAndarInterno = painelInterno.proximoAndarSelecionado(andarAtual, direcao)

Se proximoAndarInterno != -1:

// Ainda há andares para atender na mesma direção

andarDestino = proximoAndarInterno

Senão:

// Verificar na direção oposta

direcaoOposta = direcao.oposto()

proximoAndarInterno = painelInterno.proximoAndarSelecionado(andarAtual, direcaoOposta)

Se proximoAndarInterno != -1:

// Há andares na direção oposta

direcao = direcaoOposta

andarDestino = proximoAndarInterno

Senão:

// Não há mais destinos - ficar parado

direcao = PARADO

andarDestino = -1

4.2. Algoritmo de Distribuição de Elevadores

O algoritmo de distribuição de elevadores é implementado na classe CentralDeControle e é responsável por determinar qual elevador deve atender a cada chamada. Este algoritmo varia de acordo com o modelo heurístico configurado.

Componentes principais do algoritmo de distribuição:

1. **Mapeamento de chamadas pendentes**:
    - A central mantém uma lista de andares com chamadas pendentes (andaresPendentes).
    - Também mantém uma matriz elevadorDesignadoPara[andar][direcao] que mapeia qual elevador está designado para cada andar/direção.
2. **Processo de atualização**:
    - No método atualizar(int minutoSimulado), a central atualiza o mapa de designações e os tempos de espera.
    - A cada três ciclos, aplica a heurística de distribuição configurada.
3. **Tratamento de chamadas:**
    - **Quando um andar registra uma chamada, o método processarChamadas(Andar andar) é chamado.**
    - **Este método adiciona o andar à lista de andares**

pendentes se ainda não estiver lá.

- O método aplicarHeuristicaAtual() é responsável por escolher e aplicar a estratégia de distribuição adequada.

Pseudocódigo do processamento de chamadas:

Método processarChamadas(andar):

painelChamadas = andar.getPainelChamadas()

numeroAndar = andar.getNumero()

// Se não há chamada neste andar, não faz nada

Se !painelChamadas.temChamada(PARADO) e !andar.temPessoasEsperando():

Retornar

// Verificar se este andar já está na lista de pendentes

jaEstaNaLista = verificarSeAndarEstaNaLista(numeroAndar)

// Se não está na lista de pendentes, adiciona

Se !jaEstaNaLista:

andaresPendentes.adicionar(numeroAndar)

Registrar no log que andar foi adicionado

// Resetar o tempo de espera para este andar

temposEsperaAndares[numeroAndar][0] = 0

temposEsperaAndares[numeroAndar][1] = 0

4.3. Algoritmos Heurísticos

O sistema implementa três modelos heurísticos diferentes para a distribuição de elevadores, cada um com suas próprias prioridades e estratégias.

4.3.1. Sem Heurística (SEM_HEURISTICA)

Este modelo representa a abordagem mais simples, onde os elevadores são distribuídos para atender as chamadas na ordem em que foram registradas, sem considerar otimizações específicas.

Funcionamento:

1. Obter a lista de elevadores disponíveis.
2. Obter a lista de andares pendentes (na ordem em que foram registrados).

3. **Para cada elevador disponível, atribuir o próximo andar pendente não atendido.**
4. **Remover os andares atendidos da lista de pendentes.**

Pseudocódigo:

```
Método distribuirElevadoresSemHeuristica():
  // Se não há andares pendentes, não faz nada
  Se andaresPendentes.tamanho() == 0:
    Retornar

  // Lista de elevadores disponíveis
  elevadoresDisponiveis = obterElevadoresDisponiveis()

  // Se não há elevadores disponíveis, não faz nada
  Se elevadoresDisponiveis.tamanho() == 0:
    Retornar

  // Distribui os elevadores disponíveis entre os andares pendentes
  elevadorAtual = 0
  andaresPendentesTemp = copiarListaAndaresPendentes()
  // Limpa a lista original para reconstruir
  andaresPendentes = nova Lista()

  // Distribui elevadores para andares, por ordem de chegada
  Enquanto elevadorAtual < elevadoresDisponiveis.tamanho() e andaresPendentesTemp.tamanho() > 0:
    // Obtém o próximo andar da lista (o mais antigo que chegou)
    andarAtual = andaresPendentesTemp.obter(0)

    andaresPendentesTemp.remover(0)

    // Obtém o elevador atual
    elevador = elevadoresDisponiveis.obter(elevadorAtual)

    // Se o elevador já está no andar, pula para o próximo andar
    Se elevador.getAndarAtual() == andarAtual:
      // Coloca o andar de volta na lista de pendentes

      andaresPendentes.adicionar(andarAtual)
      Continuar
```

// Designa o elevador para este andar

sucesso = elevador.definirDestinoExterno(andarAtual)

Se sucesso:

// Registra a designação na matriz de controle

registrarElevadorParaAndarDirecao(andarAtual, PARADO, elevador.getId())

Registrar no log a designação

Senão:

// Se não conseguiu definir o destino, coloca o andar de volta na lista

andaresPendentes.adicionar(andarAtual)

// Passa para o próximo elevador

elevadorAtual++

// Se sobrou algum andar pendente, adiciona de volta à lista

Para cada andar em andaresPendentesTemp:

andaresPendentes.adicionar(andar)

Vantagens:

- **Simples de implementar e entender.**
- **Atendimento previsível e justo na ordem de chegada.**

Desvantagens:

- **Não otimiza para tempo de espera ou consumo de energia.**
- **Pode resultar em movimentação ineficiente dos elevadores.**

4.3.2. Otimização de Tempo de Espera (OTIMIZACAO_TEMPO_ESPERA)

Este modelo prioriza a minimização do tempo de espera dos usuários, atendendo primeiro os andares onde as pessoas estão esperando há mais tempo.

Funcionamento:

1. **Manter um registro do tempo de espera para cada andar.**
2. **Ordenar os andares pendentes por tempo de espera (do maior para o menor).**
3. **Atribuir os elevadores disponíveis aos andares com maior tempo de espera primeiro.**

Pseudocódigo:

Método distribuirElevadoresOtimizandoTempoEspera():

Se andaresPendentes.tamanho() == 0:

Retornar

```
  elevadoresDisponiveis = obterElevadoresDisponiveis()

  Se elevadoresDisponiveis.tamanho() == 0:
    Retornar

  // Ordenar andares por tempo de espera (maior para menor)
  andaresPorPrioridade = nova Lista()
  tempAndaresPendentes = copiarListaAndaresPendentes()

  // Enquanto houver andares pendentes para ordenar
  Enquanto tempAndaresPendentes.tamanho() > 0:
    andarMaiorEspera = -1
    maiorTempoEspera = -1

    // Encontrar o andar com maior tempo de espera
    Para cada andar em tempAndaresPendentes:
      tempoEsperaTotal = temposEsperaAndares[andar][0] + temposEsperaAndares[andar][1]
      Se tempoEsperaTotal > maiorTempoEspera:
        maiorTempoEspera = tempoEsperaTotal
        andarMaiorEspera = andar

    // Adicionar à lista ordenada e remover da temporária
    Se andarMaiorEspera != -1:
      andaresPorPrioridade.adicionar(andarMaiorEspera)
      Remover andarMaiorEspera de tempAndaresPendentes
    Senão:
      Sair do loop // Não deveria acontecer, mas previne loop infinito

  // Limpa a lista de andares pendentes para reconstrução
  andaresPendentes = nova Lista()

  // Distribui elevadores para os andares priorizados
  Para i = 0 até min(andaresPorPrioridade.tamanho(), elevadoresDisponiveis.tamanho()) - 1:
    andar = andaresPorPrioridade.obter(i)
    elevador = elevadoresDisponiveis.obter(i)
```

// Verifica se o elevador já está no andar

Se elevador.getAndarAtual() == andar:

andaresPendentes.adicionar(andar)

Continuar

// Tenta designar o elevador para o andar

sucesso = elevador.definirDestinoExterno(andar)

Se sucesso:

registrarElevadorParaAndarDirecao(andar, PARADO, elevador.getId())

Registrar no log a designação

// Resetar o tempo de espera para este andar

temposEsperaAndares[andar][0] = 0

temposEsperaAndares[andar][1] = 0

Senão:

andaresPendentes.adicionar(andar)

// Adicionar andares que não foram atendidos de volta à lista

Para i = elevadoresDisponiveis.tamanho() até andaresPorPrioridade.tamanho() - 1:

andaresPendentes.adicionar(andaresPorPrioridade.obter(i))

Vantagens:

- **Reduz o tempo médio de espera dos usuários.**
- **Prioriza atendimento em andares com espera prolongada.**
- **Contribui para melhorar a percepção de qualidade do serviço.**

Desvantagens:

- **Pode resultar em maior consumo de energia.**
- **Algoritmo mais complexo que o SEM_HEURISTICA.**

4.3.3. Otimização de Consumo de Energia (OTIMIZACAO_CONSUMO_ENERGIA)

Este modelo prioriza a minimização do consumo de energia dos elevadores, atribuindo elevadores de forma a minimizar deslocamentos e maximizar a eficiência energética.

Funcionamento:

1. **Identificar andares críticos (com pessoas esperando).**
2. **Calcular uma pontuação energética para cada par elevador-andar, considerando:**

- **Distância a percorrer (menor é melhor).**
- **Ocupação do elevador (elevadores com passageiros são priorizados).**
- **Direção atual do elevador (priorizar deslocamentos na mesma direção).**

3. **Atribuir elevadores aos andares de forma a minimizar o consumo energético total.**

Pseudocódigo:

```
Método distribuirElevadoresOtimizandoEnergia():
  Se andaresPendentes.tamanho() == 0:
    Retornar

  elevadoresDisponiveis = obterElevadoresDisponiveis()

  Se elevadoresDisponiveis.tamanho() == 0:
    Retornar

  // Filtrar andares críticos (com pessoas esperando)
  andaresCriticos = nova Lista()
  Para cada andar em andaresPendentes:
    objAndar = obterAndarPorNumero(andar)

    Se objAndar != null e objAndar.getNumPessoasEsperando() > 0:

andaresCriticos.adicionar(andar)

  // Matriz de pontuações energéticas

pontuacoesEnergeticas[elevadoresDisponiveis.tamanho()][andaresPendentes.tamanho()]

  // Calcular pontuação energética (menor é melhor)
  Para e = 0 até elevadoresDisponiveis.tamanho() - 1:
    elevador = elevadoresDisponiveis.obter(e)
    andarAtualElevador = elevador.getAndarAtual()

    Para a = 0 até andaresPendentes.tamanho() - 1:
      andarDestino = andaresPendentes.obter(a)

      // Calcular custo energético baseado na distância
```

```
        distancia = abs(andarAtualElevador - andarDestino)

        // Penalidade para grandes deslocamentos
        pontuacao = distancia * 10

        // Verificar se o andar é crítico
        andarEhCritico = andaresCriticos.contem(andarDestino)

        // Se o andar for crítico, reduzir a pontuação para priorizá-lo
        Se andarEhCritico:
            pontuacao -= 50

        // Se o elevador estiver vazio, penalizar deslocamentos longos
        Se elevador.getNumPassageiros() == 0:
            pontuacao += distancia * 5

        // Direção favorável
        Se (elevador.getDirecao() == SUBINDO e andarDestino > andarAtualElevador) ou
           (elevador.getDirecao() == DESCENDO e andarDestino < andarAtualElevador):
            pontuacao -= 30

        pontuacoesEnergeticas[e][a] = pontuacao

    // Atribuir elevadores a andares minimizando a pontuação energética
    andaresAtendidos = nova Lista()
    elevadoresAtribuidos = nova Lista()

    Enquanto elevadoresAtribuidos.tamanho() < elevadoresDisponiveis.tamanho() e
          andaresAtendidos.tamanho() < andaresPendentes.tamanho():

        // Encontrar o par com menor pontuação
        melhorElevador = -1
        melhorAndar = -1
        melhorPontuacao = INFINITO

        Para cada elevador não atribuído e andar não atendido:
            Se pontuacoesEnergeticas[e][a] < melhorPontuacao:
                melhorPontuacao = pontuacoesEnergeticas[e][a]
                melhorElevador = e
                melhorAndar = a

        // Se encontrou um par válido
```

Se melhorElevador != -1 e melhorAndar != -1:

elevador = elevadoresDisponiveis.obter(melhorElevador)

andarDestino = andaresPendentes.obter(melhorAndar)

// Verificar se o elevador já está no andar

Se elevador.getAndarAtual() == andarDestino:

// Marcar como atendidos e continuar

andaresAtendidos.adicionar(melhorAndar)

elevadoresAtribuidos.adicionar(melhorElevador)

Continuar

// Tentar designar o elevador para o andar

sucesso = elevador.definirDestinoExterno(andarDestino)

Se sucesso:

registrarElevadorParaAndarDirecao(andarDestino, PARADO, elevador.getId())

Registrar no log a designação

// Marcar como atendidos

andaresAtendidos.adicionar(melhorAndar)

elevadoresAtribuidos.adicionar(melhorElevador)

Senão:

Sair do loop // Não há mais pares válidos

// Reconstruir a lista de andares pendentes

novosAndaresPendentes = nova Lista()

Para i = 0 até andaresPendentes.tamanho() - 1:

Se !andaresAtendidos.contem(i):

novosAndaresPendentes.adicionar(andaresPendentes.obter(i))

andaresPendentes = novosAndaresPendentes

Vantagens:

- **Reduz o consumo total de energia do sistema.**
- **Minimiza deslocamentos desnecessários.**

- **Prioriza deslocamentos mais eficientes energeticamente.**

Desvantagens:

- **Pode resultar em tempos de espera mais longos para alguns usuários.**
- **Algoritmo mais complexo e difícil de ajustar.**

5. INTERFACE GRÁFICA E SIMULAÇÃO

5.1. Configuração da Simulação

A interface de configuração da simulação é implementada pela classe ConfiguracaoSimulador. Esta interface permite ao usuário definir diversos parâmetros antes de iniciar a simulação.

Principais parâmetros configuráveis:

1. **Configurações Básicas:**
    - **Número de andares (2 ou mais)**
    - **Número de elevadores (1 ou mais)**
    - **Quantidade inicial de pessoas**
    - **Capacidade máxima dos elevadores**
    - **Tempo limite da simulação (em minutos)**
    - **Velocidade da simulação (milissegundos por ciclo)**
    - **Tipo de painel (UNICO_BOTAO, DOIS_BOTOES, PAINEL_NUMERICO)**
    - **Modelo de heurística (SEM_HEURISTICA, OTIMIZACAO_TEMPO_ESPERA, OTIMIZACAO_CONSUMO_ENERGIA)**
2. **Configurações Avançadas:**
    - **Tempo de deslocamento padrão (minutos)**
    - **Tempo de deslocamento em horário de pico (minutos)**
    - **Consumo de energia por deslocamento**
    - **Consumo de energia por parada**
    - **Tempo máximo de espera aceitável**
3. **Geração Automática de Pessoas:**
    - **Ativar/Desativar geração automática**
    - **Intervalo de geração (ciclos)**
    - **Quantidade de pessoas por geração**

O design da interface de configuração utiliza o framework Swing com um layout organizado em abas para separar as diferentes categorias de configuração. Depois que o usuário configura todos os parâmetros, o sistema cria uma instância de Simulador com as configurações especificadas e inicia a interface gráfica de simulação.

5.2. Visualização e Interação

A interface de visualização e interação com a simulação é implementada pela classe SimuladorGI. Esta interface permite ao usuário visualizar o estado atual da simulação, interagir com ela e monitorar estatísticas.

Principais componentes da interface:

1. **Visualização do Prédio:**
    - **Representação gráfica dos andares, elevadores e pessoas**
    - **Codificação por cores do estado dos elevadores (parado, em movimento)**
    - **Indicação visual da direção dos elevadores**
    - **Visualização das filas de pessoas em cada andar**
2. **Controles da Simulação:**
    - **Botões para iniciar, pausar e encerrar a simulação**
    - **Controle para adicionar pessoas manualmente**
    - **Ajuste da velocidade de simulação**
3. **Painel de Estatísticas:**
    - **Tempo atual da simulação**
    - **Consumo total de energia**
    - **Número de pessoas esperando**
    - **Estado da geração automática**
4. **Log de Eventos:**
    - **Registro em tempo real dos eventos da simulação**
    - **Opções para limpar e salvar o log**

A interface utiliza um esquema de cores intuitivo para facilitar a compreensão do estado do sistema:

- **Verde: Elevadores parados**
- **Amarelo/Laranja: Elevadores em movimento**
- **Azul: Elementos de interface e cabeçalhos**
- **Vermelho: Alertas e situações críticas**

Além disso, a interface usa símbolos especiais para representar diferentes tipos de pessoas:

- **"•": Pessoa comum**
- **"♦": Pessoa idosa**
- **"◊": Pessoa cadeirante**

5.3. Geração de Relatórios

O sistema inclui a funcionalidade de geração de relatórios de simulação, implementada pela classe RelatorioSimulacao. Esta funcionalidade permite analisar o desempenho do sistema ao final de uma simulação.

Informações incluídas no relatório:

1. **Configurações da Simulação:**
    - **Número de andares e elevadores**
    - **Tipo de painel e modelo heurístico utilizado**
    - **Outros parâmetros relevantes**
2. **Estatísticas dos Elevadores:**
    - **Consumo individual e total de energia**
    - **Número de passageiros transportados**
    - **Posição final de cada elevador**
3. **Estatísticas de Pessoas:**
    - **Número de pessoas ainda esperando em cada andar**
    - **Total de pessoas no sistema**
4. **Estatísticas de Tempo:**
    - **Tempo médio de espera**
    - **Tempo médio de deslocamento**
    - **Tempo máximo de espera e deslocamento**
5. **Resumo da Simulação:**
    - **Duração total**
    - **Consumo total de energia**
    - **Utilização média dos elevadores**

Pseudocódigo da geração de relatório:

```
Método gerarRelatorio():
  // Inicializar o relatório
  relatorio = nova StringBuilder()

  // Cabeçalho
adicionar_cabecalho_ao_relatorio(relatorio)

  // Data e hora
adicionar_data_hora_ao_relatorio(relatorio)

  // Configurações gerais
  adicionarConfiguracoes(relatorio)

  // Estatísticas de elevadores
adicionarEstatisticasElevadores(relatorio)

  // Estatísticas de pessoas
adicionarEstatisticasPessoas(relatorio)

  // Estatísticas de tempo
adicionarEstatisticasTempo(relatorio)
```

// Resumo

adicionarResumo(relatorio)

Retornar relatorio.toString()

O relatório pode ser visualizado na interface gráfica e também salvo em um arquivo de texto para análise posterior. Isso permite comparar diferentes configurações e estratégias heurísticas para identificar a mais eficiente para um determinado cenário.

6. CONCLUSÃO E TRABALHOS FUTUROS

6.1. Síntese do Sistema

O Sistema de Simulação de Elevadores apresentado neste documento demonstra uma implementação robusta e flexível para modelar o comportamento de múltiplos elevadores em um edifício. O sistema utiliza estruturas de dados personalizadas e algoritmos especializados para simular diferentes estratégias de controle, permitindo análises comparativas de eficiência.

Os principais pontos fortes do sistema incluem:

1. **Modularidade e Extensibilidade: A arquitetura MVC adotada facilita a manutenção e extensão do sistema.**
2. **Implementação Didática: As estruturas de dados personalizadas (Lista, Fila, FilaPrioridade) oferecem uma visão clara de suas implementações e comportamentos.**
3. **Estratégias Heurísticas: Os diferentes modelos heurísticos permitem comparar abordagens de otimização para tempo de espera e consumo de energia.**
4. **Interface Gráfica Interativa: A visualização em tempo real facilita a compreensão do comportamento do sistema.**
5. **Geração de Relatórios: A análise estatística dos resultados permite avaliar o desempenho das diferentes configurações.**

6.2. Limitações Atuais

Apesar de suas qualidades, o sistema apresenta algumas limitações:

1. **Escalabilidade das Estruturas de Dados: As implementações atuais podem não ser eficientes para sistemas muito grandes (prédios com muitos andares e elevadores).**
2. **Simplicidade das Heurísticas: As estratégias implementadas são relativamente simples comparadas a algoritmos avançados de controle de elevadores reais.**

3. **Ausência de Previsão de Demanda: O sistema não implementa algoritmos preditivos para antecipar demandas em horários específicos.**
4. **Limitações da Interface Gráfica: A visualização atual, embora funcional, poderia ser mais rica e detalhada.**