FUNCIONAMENTO E DESAFIOS NA IMPLEMENTAÇÃO DO SIMULADOR DE COLETA DE LIXO PARA TERESINA

Vitor Daniel Alves Mousinho Alexandre Medeiros Cavalcante

Maio 2025

1. Introdução

Este relatório apresenta o funcionamento e os principais desafios enfrentados durante o desenvolvimento do Simulador de Coleta de Lixo para a cidade de Teresina. O projeto foi desenvolvido como trabalho final da disciplina de Estrutura de Dados, com o objetivo de modelar o processo de coleta de resíduos sólidos na cidade, integrando conceitos de gestão ambiental e implementação de estruturas de dados personalizadas.

2. Funcionamento do Sistema de Simulação

2.1. Visão Geral do Funcionamento

O simulador modela o processo de coleta de lixo em Teresina através de um sistema que inclui zonas urbanas, caminhões de diferentes capacidades, estações de transferência e um aterro sanitário. O funcionamento do sistema pode ser resumido em:

1. A cidade é dividida em cinco zonas (Sul, Norte, Centro, Leste e Sudeste), cada uma gerando lixo em intervalos configuráveis.
2. Caminhões pequenos (2, 4, 8 e 10 toneladas) são distribuídos entre as zonas para coletar o lixo acumulado.
3. Quando um caminhão pequeno atinge sua capacidade, dirige-se a uma estação de transferência.
4. Nas estações, os caminhões pequenos aguardam em fila para transferir o lixo para caminhões grandes (20 toneladas).
5. Quando um caminhão grande está cheio ou atinge seu tempo máximo de espera, parte para o aterro sanitário.

2.2. Implementação das Estruturas Básicas

Para implementar o simulador, desenvolvemos estruturas de dados personalizadas em vez de utilizar as coleções prontas do Java. As principais estruturas implementadas foram:

```
public class Lista<T> {
    private Object[] elementos;
    private int tamanho;
    private static final int CAPACIDADE_INICIAL = 10;

    public Lista() {
        this.elementos = new Object[CAPACIDADE_INICIAL];
        this.tamanho = 0;
    }

    // Metodos para adicionar, remover e acessar elementos...
}
```

Listing 1: Implementação da classe Lista

```java
public class Fila<T> {
    private No inicio;
    private No fim;
    private int tamanho;

    private class No {
        private T elemento;
        private No proximo;

        // Construtor e metodos...
    }

    // Metodos para adicionar, remover e acessar elementos...
}
```

Listing 2: Implementação da classe Fila

Estas estruturas foram fundamentais para o gerenciamento das entidades do simulador e para a implementação dos algoritmos de distribuição e processamento de filas.

2.3. Modelagem da Cidade com MapaUrbano

Para representar a cidade de Teresina, implementamos a classe `MapaUrbano` que define as distâncias relativas entre as diferentes zonas e estações:

```java
public class MapaUrbano implements Serializable {
    private int[][] distancias;
    private Map<String, Integer> indicesZonas;
    private List<String> zonasRegistradas;

    public MapaUrbano() {
        // Inicializa o mapa com as cinco zonas de Teresina
        indicesZonas = new HashMap<>();
        zonasRegistradas = new ArrayList<>();

        // Registra as zonas
        registrarZona("Sul");
        registrarZona("Norte");
        registrarZona("Centro");
        registrarZona("Leste");
        registrarZona("Sudeste");

        // Inicializa a matriz de distancias
        int numZonas = zonasRegistradas.size();
        distancias = new int[numZonas][numZonas];

        // Define as distancias entre as zonas
        setDistancia("Sul", "Sul", 0);
        setDistancia("Sul", "Norte", 8);
        setDistancia("Sul", "Centro", 3);
        // ...
    }

    // Metodos para acessar e manipular as distancias
}
```

Listing 3: Implementação da classe MapaUrbano

Esta classe permite calcular tempos de viagem mais realistas, considerando a posição relativa de cada zona na cidade.

2.4. Sistema de Geração e Coleta de Lixo

Cada zona da cidade gera lixo em intervalos configuráveis:

```java
public class ZonaUrbana {
    private String nome;
    private int lixoAcumulado;
    private int geracaoMinima;
    private int geracaoMaxima;
    private Random random;

    public void gerarLixo() {
```

```
        int quantidade = random.nextInt(geracaoMaxima - geracaoMinima + 1) +
    geracaoMinima;
        lixoAcumulado += quantidade;
        System.out.println(nome + ": Gerou " + quantidade + "kg de lixo. Total: " +
    lixoAcumulado);
    }

    public int coletarLixo(int quantidade) {
        int coletado = Math.min(quantidade, lixoAcumulado);
        lixoAcumulado -= coletado;
        return coletado;
    }

    // Outros metodos...
}
```

Listing 4: Implementação da classe ZonaUrbana

Os caminhoes pequenos sao responsaveis por coletar o lixo nas zonas:

```
public abstract class CaminhaoPequeno {
    protected int capacidade;
    protected int cargaAtual;
    protected String placa;
    protected StatusCaminhao status;
    protected int tempoRestanteViagem;
    protected ZonaUrbana zonaAtual;
    protected EstacaoTransferencia estacaoDestino;

    public abstract int coletar(int quantidade);

    // Outros metodos...
}

public class CaminhaoPequenoPadrao extends CaminhaoPequeno {
    @Override
    public int coletar(int quantidade) {
        if (quantidade <= 0 || estaCheio()) {
            return 0;
        }

        int espacoDisponivel = capacidade - cargaAtual;
        int quantidadeColetada = Math.min(quantidade, espacoDisponivel);

        if (quantidadeColetada > 0) {
            cargaAtual += quantidadeColetada;
            return quantidadeColetada;
        }

        return 0;
    }
}
```

Listing 5: Implementação da classe CaminhaoPequeno

2.5. Funcionamento das Estações de Transferência

As estações de transferência recebem os caminhões pequenos e transferem o lixo para caminhões grandes:

```
public class EstacaoPadrao extends EstacaoTransferencia {
    private Fila<CaminhaoPequeno> filaCaminhoesRequenos;
    private CaminhaoGrande caminhaoGrandeAtual;
    private int tempoEsperaCaminhaoGrandeAtual;
    private int tempoPrimeiroCaminhaoNaFila;

    public ResultadoProcessamentoFila processarFila(int tempoSimuladoAtual) {
        if (filaCaminhoesRequenos.estaVazia()) {
            tempoPrimeiroCaminhaoNaFila = 0;
            return new ResultadoProcessamentoFila(null, 0);
        }

        tempoPrimeiroCaminhaoNaFila++;
```

```
14
15          if (caminhaoGrandeAtual == null) {
16              return new ResultadoProcessamentoFila(null, 0);
17          }
18
19          CaminhaoPequeno caminhaoPequeno = filaCaminhoesRequenos.primeiroElemento();
20
21          if (caminhaoGrandeAtual.getCargaAtual() + caminhaoPequeno.getCargaAtual() <=
    caminhaoGrandeAtual.getCapacidadeMaxima()) {
22              caminhaoPequeno = filaCaminhoesRequenos.remover();
23
24              int cargaDescarregada = caminhaoPequeno.descarregar();
25              caminhaoGrandeAtual.carregar(cargaDescarregada);
26
27              long tempoDeEspera = 0;
28              if (caminhaoPequeno.getTempoChegadaNaFila() != -1) {
29                  tempoDeEspera = tempoSimuladoAtual - caminhaoPequeno.
    getTempoChegadaNaFila();
30                  if (tempoDeEspera < 0) {
31                      tempoDeEspera = 0;
32                  }
33              }
34
35              tempoPrimeiroCaminhaoNaFila = 0;
36
37              return new ResultadoProcessamentoFila(caminhaoPequeno, tempoDeEspera);
38          } else {
39              // Caminhao grande sem espaco, precisa partir
40              return new ResultadoProcessamentoFila(null, 0);
41          }
42      }
43
44      // Outros m todos...
45  }
```

Listing 6: Implementação da classe EstacaoPadrao

2.6. Gerenciamento de Caminhoes Grandes

Os caminhoes grandes possuem uma tolerancia de espera e partem para o aterro quando estao cheios ou quando o tempo de espera e excedido:

```
1  public abstract class CaminhaoGrande {
2      public static final int CAPACIDADE_MAXIMA_KG = 20000;
3
4      protected int capacidadeMaxima = CAPACIDADE_MAXIMA_KG;
5      protected int cargaAtual;
6      protected int toleranciaEspera;
7
8      public boolean prontoParaPartir() {
9          return cargaAtual >= capacidadeMaxima;
10     }
11
12     public void descarregar() {
13         System.out.println("Caminhao grande partiu para o aterro com " + cargaAtual + "kg
    .");
14         cargaAtual = 0;
15     }
16
17     // Outros m todos...
18 }
```

Listing 7: Implementação da classe CaminhaoGrande

2.7. Sistema de Distribuicao de Caminhoes

O coracao do simulador e o sistema de distribuicao inteligente de caminhoes entre as zonas, implementado na classe `DistribuicaoCaminhoes`:

```
1  public class DistribuicaoCaminhoes implements Serializable {
2      private MapaUrbano mapaUrbano;
3      private Map<String, ScoreZona> scoresZonas;
```

```
4
5       public int distribuirCaminhoes(Lista<CaminhaoPequeno> caminhoes,
6                                      Lista<ZonaUrbana> todasZonas,
7                                      Lista<ScoreZona> zonasOrdenadas) {
8           int distribuidos = 0;
9
10          // Primeira fase - distribuir caminhoes iniciais com base nos scores
11          int caminhoesFirstCota = Math.min(caminhoes.tamanho(), todasZonas.tamanho() * 3);
12          int caminhoesForZonaBasico = Math.max(1, caminhosPrimeraCota / todasZonas.tamanho
    ());
13
14          for (int z = 0; z < zonasOrdenadas.tamanho() && !caminhoes.estaVazia(); z++) {
15              ScoreZona scoreZona = zonasOrdenadas.obter(z);
16              ZonaUrbana zona = scoreZona.getZona();
17
18              for (int i = 0; i < caminhoesForZonaBasico && !caminhoes.estaVazia(); i++) {
19                  // Encontra o caminhao mais proximo ou o melhor candidato
20                  CaminhaoPequeno melhorCaminhao = encontrarCaminhaoMaisProximo(caminhoes,
    zona);
21                  // Logica de envio...
22              }
23          }
24
25          // Segunda fase - distribuir caminhoes restantes de forma circular
26          if (!caminhoes.estaVazia()) {
27              int indiceZona = 0;
28              while (!caminhoes.estaVazia()) {
29                  ZonaUrbana zona = todasZonas.obter(indiceZona);
30                  CaminhaoPequeno caminhao = caminhoes.obter(0);
31                  caminhoes.remover(0);
32                  // L gica de envio...
33                  // Avanca para a proxima zona (distribuicao circular)
34                  indiceZona = (indiceZona + 1) % todasZonas.tamanho();
35              }
36          }
37
38          return distribuidos;
39      }
40
41      // Outros metodos para calculo de scores, ordenacao, etc.
42  }
```

Listing 8: Implementação da classe DistribuicaoCaminhoes

3. Interface Gráfica

Para visualização e controle do simulador, implementamos uma interface gráfica básica utilizando JavaFX. A interface mostra o estado das zonas, estações, caminhões em atividade e estatísticas em tempo real.

```
1   private void exibirRelatorioFinalGrafico() {
2       // Graficos de barras para comparacao entre lixo gerado e coletado por zona
3       XYChart.Series<String, Number> seriesLixoGerado = new XYChart.Series<>();
4       seriesLixoGerado.setName("Lixo Gerado");
5
6       XYChart.Series<String, Number> seriesLixoColetado = new XYChart.Series<>();
7       seriesLixoColetado.setName("Lixo Coletado");
8
9       String[] nomesZonasOrdenadas = {"Sul", "Norte", "Centro", "Leste", "Sudeste"};
10
11      if (stats.getEstatisticasZonas() != null && stats.getEstatisticasZonas().tamanho() >
    0) {
12          Lista<Estatisticas.EntradaZona> zonasStats = stats.getEstatisticasZonas();
13          for (String nomeZona : nomesZonasOrdenadas) {
14              // Adiciona dados ao grafico...
15          }
16      }
17      // ...
18
19      // Cria a visualizacao tabular das metricas por estacao
20      ObservableList<MetricaEstacaoModel> metricasEstacaoList = FXCollections.
        observableArrayList();
```

```
21      if (stats.getEstatisticasEstacoes() != null && stats.getEstatisticasEstacoes().
        tamanho() > 0) {
22          Lista<Estatisticas.EntradaEstacao> estacoesStats = stats.getEstatisticasEstacoes
        ();
23          for (int i = 0; i < estacoesStats.tamanho(); i++) {
24              // Adiciona dados a tabela...
25          }
26      }
27
28      // Graficos de caminhoes em atividade
29      // Visualizacao da lista de caminhoes em atividade
30      if (vboxCaminhoesEmAtividade != null) {
31          atualizarListaCaminhoesAtividade();
32      }
33  }
```

Listing 9: Implementação da visualização gráfica

Esta representação visual permitiu um acompanhamento claro do estado de cada caminhão, facilitando a depuração e compreensão do sistema como um todo.

4. Desafios e Soluções na Implementação

4.1. Desafio das Estruturas Próprias

Um dos requisitos mais importantes do projeto era implementar nossas próprias estruturas de dados personalizadas, sem utilizar as coleções prontas do Java (ArrayList, LinkedList, etc.). Isso representou um desafio significativo, pois precisávamos garantir eficiência e corretude nas operações básicas dessas estruturas.

Para a lista genérica, enfrentamos o desafio de garantir que o array interno fosse redimensionado quando necessário:

```
1  private void garantirCapacidade() {
2      if (tamanho == elementos.length) {
3          int novaCapacidade = elementos.length * 2;
4          elementos = Arrays.copyOf(elementos, novaCapacidade);
5      }
6  }
```

Listing 10: Método de garantia de capacidade na Lista

Para a fila, optamos por uma implementação baseada em encadeamento, o que simplificou as operações de adição e remoção:

```
1  public void adicionar(T elemento) {
2      No novoNo = new No(elemento);
3      if (estaVazia()) {
4          inicio = novoNo;
5      } else {
6          fim.proximo = novoNo;
7      }
8      fim = novoNo;
9      tamanho++;
10 }
```

Listing 11: Implementação da adição na Fila

4.2. Desafio da Distribuição Equilibrada

Um dos maiores desafios que enfrentamos foi desenvolver um algoritmo eficiente para distribuir os caminhões pequenos entre as cinco zonas da cidade. Inicialmente, tentamos uma abordagem simples de distribuição uniforme, mas logo percebemos que isso não era eficiente:

1. Diferentes zonas geram quantidades diferentes de lixo
2. O número de caminhões disponíveis poderia ser ímpar ou insuficiente para atender todas as zonas
3. A distância entre as zonas e as estações afetava o tempo de coleta

Ao testar a aplicação, observamos que com um número ímpar de caminhões para cinco zonas, algumas zonas ficavam sem atendimento, resultando em acúmulo excessivo de lixo. Além disso, quando alguns caminhões se deslocavam para as estações de transferência, certas zonas ficavam temporariamente descobertas.

4.3. Solução com Sistema de Score

Para resolver esse problema, desenvolvemos um sistema de pontuação (score) para as zonas, que considerava múltiplos fatores:

1. **Quantidade de lixo acumulado**: Zonas com mais lixo recebiam maior prioridade
2. **Tempo desde a última coleta**: Áreas que não eram atendidas há mais tempo tinham prioridade aumentada
3. **Caminhões já alocados**: Zonas com muitos caminhões tinham sua prioridade reduzida
4. **Taxa de geração**: Zonas que geravam mais lixo por hora recebiam uma prioridade adicional

A implementação desse sistema de score foi realizada através da classe `ScoreZona`:

```
public void calcularScore() {
    // Aumentar o peso do lixo acumulado quando ha poucos caminhoes
    double pesoLixoAjustado = PESO_LIXO_ACUMULADO;
    if (caminhoesRequenosAtivos <= 1) {
        // Dobrar o peso do lixo acumulado quando ha poucos caminhoes ativos
        pesoLixoAjustado = PESO_LIXO_ACUMULADO * 2.0;
    }

    double scoreLixoAcumulado = zona.getLixoAcumulado() * pesoLixoAjustado;
    double scoreTempoSemColeta = tempoDesdeUltimaColeta * PESO_TEMPO_SEM_COLETA;
    double scoreCaminhoesAtivos = caminhoesRequenosAtivos * PESO_CAMINHOES_ATIVOS;
    double scoreTaxaGeracao = ((zona.getGeracaoMaxima() + zona.getGeracaoMinima()) / 2.0)
     * PESO_TAXA_GERACAO;

    this.score = scoreLixoAcumulado + scoreTempoSemColeta + scoreCaminhoesAtivos +
    scoreTaxaGeracao;
    this.scoreFinal = this.score;
}
```

Listing 12: Implementação do cálculo de score

4.4. Desafio do Balanceamento do Score

Este balanceamento foi crucial para garantir uma distribuição mais equitativa dos caminhões, sem deixar nenhuma zona sem atendimento por períodos prolongados.

```
private void aplicarFatorBalanceamento(Lista<ScoreZona> zonasOrdenadas) {
    // Se tivermos pelo menos duas zonas, ajusta scores para nao ter diferenca excessiva
    if (zonasOrdenadas.tamanho() >= 2) {
        double scoreMaximo = zonasOrdenadas.obter(0).getScore();
        double scoreMinimo = zonasOrdenadas.obter(zonasOrdenadas.tamanho() - 1).getScore
    ();

        // Se a diferenca for muito grande, aproxima valores
        if (scoreMaximo > 8 * scoreMinimo) {
            for (int i = 0; i < zonasOrdenadas.tamanho(); i++) {
                ScoreZona score = zonasOrdenadas.obter(i);
                // Modificacao na formula para reduzir menos o score de zonas
    prioritarias
                double novoScore = (score.getScore() * 0.9) + (scoreMinimo * 0.1);
                score.setScoreFinal(novoScore);
            }
            // Reordena com scores ajustados
            ordenarZonasPorScoreFinal(zonasOrdenadas);
        }
    }
}
```

Listing 13: Implementação do fator de balanceamento

Esta abordagem de duas fases garantiu que todos os caminhões fossem aproveitados, usando primeiro o sistema de score para priorizar zonas com maior necessidade, e depois uma distribuição circular para garantir que todos os caminhões fossem utilizados.

4.5. Desafio do Fluxo nas Estações

Outro desafio significativo foi implementar o comportamento das estações de transferência, onde os caminhões pequenos descarregam o lixo coletado em caminhões grandes. Era necessário simular filas, calcular tempos de espera e decidir quando um caminhão grande deveria partir para o aterro.

```
public CaminhaoGrande gerenciarTempoEsperaCaminhaoGrande() {
    if (caminhaoGrandeAtual == null) {
        tempoEsperaCaminhaoGrandeAtual = 0;
        return null;
    }

    tempoEsperaCaminhaoGrandeAtual++;
    boolean devePartir = false;
    String motivoPartida = "";

    if (caminhaoGrandeAtual.prontoParaPartir()) {
        devePartir = true;
        motivoPartida = "atingiu a capacidade maxima";
    }
    else if (tempoEsperaCaminhaoGrandeAtual >= caminhaoGrandeAtual.getToleranciaEspera())
     {
        if (caminhaoGrandeAtual.getCargaAtual() > 0) {
            devePartir = true;
            motivoPartida = "excedeu a tolerancia de espera";
        } else {
            tempoEsperaCaminhaoGrandeAtual = 0;
        }
    }
    // Outras condicoes...

    if (devePartir) {
        CaminhaoGrande caminhaoQuePartiu = caminhaoGrandeAtual;
        this.caminhaoGrandeAtual = null;
        this.tempoEsperaCaminhaoGrandeAtual = 0;
        return caminhaoQuePartiu;
    }

    return null;
}
```

Listing 14: Gerenciamento de tempo de espera nos caminhões grandes

Este algoritmo considera três fatores principais:

1. Se o caminhão grande está cheio (capacidade máxima atingida)
2. Se o tempo de espera excedeu a tolerância configurada
3. Se o próximo caminhão pequeno na fila não pode ser atendido por falta de espaço

5. Visualização Gráfica das Estatísticas

Para complementar o relatório textual, implementamos também uma visualização gráfica dos dados:

```
private void exibirRelatorioFinalGrafico() {
    // Gr ficos de barras para compara  o entre lixo gerado e coletado por zona
    XYChart.Series<String, Number> seriesLixoGerado = new XYChart.Series<>();
    seriesLixoGerado.setName("Lixo Gerado");

    XYChart.Series<String, Number> seriesLixoColetado = new XYChart.Series<>();
    seriesLixoColetado.setName("Lixo Coletado");

    // Visualizacao de metricas das estacoes em tabelas
    ObservableList<MetricaEstacaoModel> metricasEstacaoList = FXCollections.
    observableArrayList();

```

```
12      // Gr ficos de caminh es em atividade
13      // Visualiza  o da lista de caminh es em atividade
14      if (vboxCaminhoesEmAtividade != null) {
15          atualizarListaCaminhoesAtividade();
16      }
17  }
```

Listing 15: Visualização estatística

Esta visualização gráfica complementou o relatório textual, permitindo uma compreensão mais intuitiva do estado do sistema e do desempenho da simulação.

6. Conclusão

O desenvolvimento do Simulador de Coleta de Lixo para Teresina apresentou diversos desafios que demandaram soluções criativas e eficientes. A implementação de estruturas de dados próprias, o desenvolvimento de algoritmos de distribuição inteligente e o gerenciamento eficiente das estações de transferência foram os principais obstáculos superados durante o projeto.

O sistema de score desenvolvido para a distribuição de caminhões entre as zonas foi particularmente bem-sucedido, demonstrando como algoritmos inteligentes podem otimizar a alocação de recursos limitados. O modelo híbrido de distribuição, combinando a abordagem baseada em score com uma distribuição circular, garantiu um equilíbrio entre eficiência global e cobertura.

As simulações realizadas permitiram responder à pergunta central sobre o número mínimo de caminhões grandes de 20 toneladas no mínimo o município deverá possuir para atender a demanda da geração de lixo em Teresina. Foi obtida através de múltiplas simulações com diferentes configurações, revelando o número ideal de caminhões necessários para manter a cidade limpa com o mínimo de recursos possível.

Este projeto não apenas atendeu aos requisitos acadêmicos da disciplina de Estrutura de Dados, mas também desenvolveu uma ferramenta com potencial aplicação prática na gestão urbana, demonstrando como soluções computacionais podem contribuir para a sustentabilidade ambiental e a melhoria da qualidade de vida nas cidades brasileiras.

7. Referências

ABNT. NBR 6023: Informação e documentação - Referências - Elaboração. Rio de Janeiro, 2018.