Trabalho Prático 1 (TP1)
Estrutura de Dados

Ítalo Leal Lana Santos | Matrícula: 2024013893 | italolealanasantos@ufmg.br
Universidade Federal de Minas Gerais (UFMG) | Belo Horizonte/MG | Maior de 2025

---

1. Introdução

Este trabalho prático (TP1) visa implementar um Tipo Abstrato de Dados (TAD) "Ordenador Universal". O objetivo é selecionar dinamicamente o algoritmo de ordenação mais eficiente – entre **Insertion Sort** e **Quicksort (com mediana de 3 e partição de 3 vias)** – com base nas características do vetor de entrada.

A otimização busca minimizar um custo ponderado, calculado a partir do número de comparações, movimentações e chamadas de função, cujos pesos (a, b, c) são fornecidos na entrada.
O TAD calibra empiricamente dois limiares cruciais:

- **minTamParticao**: Tamanho mínimo de partição para o Quicksort, abaixo do qual o Insertion Sort é usado.

- **limiarQuebras**: Número de "quebras" (definidas como inversões adjacentes) no vetor que torna o Insertion Sort preferível comparado ao Quicksort para o vetor.

A calibração utiliza um método de refinamento iterativo até que a diferença de custo entre os extremos da faixa de busca seja menor que um limiarCusto também fornecido na entrada.

2. Método

2.1. Ambiente e Ferramentas

O desenvolvimento do projeto foi realizado utilizando a linguagem C++, com o compilador GCC (GNU Compiler Collection) versão padrão com C++11.

O ambiente de desenvolvimento primário foi o Windows 11, utilizando o WSL (Windows Subsystem for Linux).

Processador: Intel Core i5-10210U @ 2.11 GHz
Memória: 8 GB RAM

2.2. Organização do Projeto

```
TP1/
├── bin/
├── include/
│   ├── ordenador.hpp
│   ├── sorting_algorithms.hpp
│   ├── stats.hpp
│   └── vector_manager.hpp
├── src/
│   ├── main.cpp
│   ├── ordenador.hpp
│   ├── sorting_algorithms.hpp
│   ├── stats.hpp
│   └── vector_manager.hpp
├── obj/
│   ├── files/
│   └── Makefile
```

2.3. Classes Principais

a) **main.cpp:** Ponto de entrada. Lê os parâmetros de entrada (incluindo seed, limiarCusto, coeficientes a,b,c, e dados do vetor), orquestra a calibração dos limiares e imprime informações relevantes.

b) **ordenador.cpp:** Encapsula a lógica do "Ordenador Universal" e os métodos de calibração LimPart_determinaLimiar e LimQueb_determinaLimiar. Contém getNumeroQuebras para contar inversões adjacentes.

c) **sorting_algorithms.cpp:** Contém as implementações do InsertionSort e QuickSort. O Quicksort utiliza mediana de 3 para seleção do pivô e partição de 3 vias. Gerencia um objeto estático da classe Stats e o QUICK_SORT_SIZE (limiar interno do Quicksort).

d) **stats.cpp:** Classe para contabilizar comparações, movimentações e chamadas. Calcula o custo total usando a fórmula cost = |a*cmp + b*move + c*calls|.

e) **vector_manager.cpp:** Funções utilitárias para manipulação de vetores: inicialização com semente (srand48/drand48), geração de números, cópia e embaralhamento (shuffleVector que realiza um número especificado de trocas aleatórias).

2.4. Métodos Principais

1. Ordenador::getNumeroQuebras(vetor[], tam)

- Calcula o número de **inversões adjacentes** no vetor.
- Conta quantos pares consecutivos vetor[i] > vetor[i+1] existem no vetor, indicando seu nível de desordem local.

2. Ordenador::OrdenadorUniversal(vetor[], tam, minTamParticao, limiarQuebras)

- Recebe um vetor e os limiares (pré-calibrados ou para teste).
- Funcionamento:
  - Se getNumeroQuebras(vetor, tam) < limiarQuebras, utiliza **SortingAlgorithms::InsertionSort**.
  - Caso contrário, se tam > minTamParticao, utiliza **SortingAlgorithms::QuickSort**.
  - Se nenhuma das condições anteriores for satisfeita (ou seja, o vetor possui mais quebras que o limiar, mas tamanho menor ou igual a minTamParticao), utiliza novamente **SortingAlgorithms::InsertionSort**.

3. Ordenador::LimPart_determinaLimiar(...)

- Responsável por calibrar o valor de **minTamParticao** (corresponde ao parâmetro QUICK_SORT_SIZE do SortingAlgorithms).
- Funcionamento:
  - Começa com uma faixa inicial ampla (minMPS até maxMPS).
  - Refinamento iterativo:
    - Em cada iteração, testa 5 pontos dentro da faixa atual.
    - Executa **OrdenadorUniversal** com limiarQuebras = 0 (para garantir foco no QuickSort).
    - Mede o custo associado a cada ponto.
  - A faixa é ajustada ao redor do ponto que apresentou o menor custo.
  - O processo é encerrado quando:
    - A diferença de custo entre os extremos da faixa é menor que um **limiarCusto**, ou
    - Um número máximo de iterações é atingido.

4. Ordenador::LimQueb_determinaLimiar(...)

- Responsável por calibrar o valor de **limiarQuebras**, também utilizando refinamento iterativo de uma faixa.
- Funcionamento:
  - Para cada valor de limiarQuebras testado (gerando diferentes níveis de desordem via shuffleVector):
    - Executa **SortingAlgorithms::QuickSort** e **SortingAlgorithms::InsertionSort** separadamente.
    - Avalia a diferença de custo entre os dois algoritmos.
  - O objetivo é encontrar o limiarQuebras que **minimiza a diferença de custo** entre QuickSort e InsertionSort.
  - O processo de refinamento da faixa e os critérios de parada seguem a mesma lógica de **LimPart_determinaLimiar**.

5. SortingAlgorithms::QuickSort

- Implementa o algoritmo QuickSort com:
  - **Mediana de 3** para escolha do pivô.
  - **Particionamento em 3 vias**, eficiente para tratar elementos repetidos.
  - Utiliza **_insertionSort** para sub-partições menores que QUICK_SORT_SIZE (valor de minTamParticao calibrado).

6. Stats::calculateCost()

- Calcula o custo total de execução de um algoritmo, utilizando a fórmula:
  Custo = | a*cmp + b*move + c*calls |

Onde:
cmp = número de comparações,
move = número de movimentações,
calls = número de chamadas recursivas ou operacionais,
a, b, c = pesos configuráveis para cada métrica.

3. Análise Complexidade

Considerando N como o tamanho do vetor de entrada.

3.1. Complexidade de Tempo

- **Ordenador::getNumeroQuebras**
  Percorre o vetor uma única vez. Complexidade **O(N)**.

- **SortingAlgorithms::InsertionSort**
  Melhor caso (vetor já ordenado): **O(N)**.
  Pior e caso médio: **O(N²)**.

- **SortingAlgorithms::QuickSort**
  Caso médio: **O(N * log(N))**.
  Pior caso: **O(N²)**, mitigado pelo uso de **mediana de 3** e **partição em 3 vias**.

- **Ordenador::OrdenadorUniversal**
  Executa getNumeroQuebras (**O(N)**) e depois InsertionSort ou QuickSort.
  Complexidade dominada pelo algoritmo de ordenação:
  O(N * log(N)) em média, **O(N²)** no pior caso.

- **Ordenador::LimPart_determinaLimiar** e **LimQueb_determinaLimiar**
  Ambos realizam um número constante e pequeno de iterações (ex.: até 5).

- **LimPart_determinaLimiar:**
  Cada ponto testa OrdenadorUniversal **O(N * log(N))**. em média.

- **LimQueb_determinaLimiar:** Cada ponto envolve:
  - shuffleVector (**O(N)**),
  - QuickSort (**O(N \log N)**),
  - InsertionSort (**O(N²)**).
    Dominado por InsertionSort: **O(N²)** por ponto.

- **Complexidade da Calibração**
  Dominada por **LimQueb_determinaLimiar**, portanto aproximadamente **O(N²)** (com fator constante de iterações e pontos por iteração).

- **Complexidade Total (com calibração)**
  O(N) + 6 * O(N²) + O(N * log(N)) = O(N²)

3.2 Complexidade de Espaço

- **getNumeroQuebras** e **InsertionSort**
  Espaço auxiliar constante: **O(1)**.

- **QuickSort**
  Espaço da pilha de recursão:
  O(log(N)) em média, **O(N)** no pior caso.

- **Ordenador**, **vectorManager** e **Stats**
  Espaço para armazenamento do vetor e vetorCópia: **O(2N)**.

- **Complexidade Total de Espaço**
  O(1) + O(N) + O(2N) = O(N)

4. Estratégias de Robustez

- **Validação de Arquivo:** main.cpp verifica se o arquivo de entrada foi aberto corretamente.
- **Mediana de 3 e Partição de 3 Vias no Quicksort:** Aumentam a robustez do Quicksort contra piores casos e lidam eficientemente com chaves repetidas.
- **srand48/drand48:** Utilizados em vectorManager para geração de números aleatórios de melhor qualidade, contribuindo para a reprodutibilidade e qualidade dos testes baseados em dados aleatórios**.**
- **Convergência da Calibração:** Os laços de calibração em LimPart_determinaLimiar e LimQueb_determinaLimiar usam um limiarCusto para determinar a convergência e um número máximo de iterações para evitar loops excessivos.
- **Coeficientes de Custo não Negativos:** O uso de fabs() na função de custo da classe Stats garante que o custo seja sempre positivo, evitando problemas com coeficientes negativos.

5. Análise Experimental

O objetivo dessa análise é determinar os limiares minTamParticao e limiarQuebras que otimizam o desempenho da ordenação, de acordo com a função de custo:

Cost = | a*cmp + b*move + c*calls |

5.1. Configuração da Entrada

O programa recebe como parâmetros:

- Seed (para controle da aleatoriedade),
- LimiarCusto (critério de convergência),
- Coeficientes a, b e c (ponderações da função de custo),
- O tamanho e os elementos do vetor a ser ordenado.

5.2. Calibração de minTamParticao

- Busca o tamanho ideal de partição, onde o QuickSort deve recorrer ao InsertionSort.
- O método OrdenadorUniversal é executado com limiarQuebras = 0, forçando foco no QuickSort e suas sub-partições.
- A busca é iterativa e encerra quando a diferença de custo entre os extremos da faixa é menor que limiarCusto.

5.3. Calibração de limiarQuebras

- Busca o ponto onde a diferença de custo entre QuickSort e InsertionSort é minimizada, considerando o nível de desorganização do vetor (medido em quebras via shuffleVector).
- Ambos os algoritmos são testados sobre vetores com o mesmo nível de desordem.
- A busca também utiliza limiarCusto como critério de parada.

5.4. Resultados Esperados

- Obtêm-se valores ideais para minTamParticao e limiarQuebras, específicos para os coeficientes a, b, c e para a definição de "quebra" adotada.
- O programa imprime os limiares encontrados e estatísticas detalhadas, permitindo acompanhar o processo de calibração e sua convergência.
- O Ordenador Universal, uma vez calibrado, tende a apresentar bom desempenho médio para novas entradas, adaptando-se automaticamente conforme o grau de desordem e o tamanho do vetor.
- A eficácia desse sistema depende diretamente da escolha dos coeficientes a, b, c e da capacidade da métrica getNumeroQuebras (inversões adjacentes) em representar corretamente o conceito de "quase ordenado", fundamental para o bom desempenho do InsertionSort.

6. Conclusões

O trabalho implementou um "Ordenador Universal" capaz de calibrar dinamicamente os limiares minTamParticao e limiarQuebras através de um processo iterativo de refinamento, guiado por uma função de custo parametrizável e um critério de convergência (limiarCusto).

6.1. Aprendizados principais:

- A definição de "quase ordenado" (aqui, por inversões adjacentes) e a função de custo são cruciais para a calibração.
- Métodos de busca adaptativa podem encontrar limiares razoáveis sem testar exaustivamente todas as possibilidades.
- A separação de responsabilidades em módulos (Ordenador, SortingAlgorithms, Stats, vectorManager) facilita o desenvolvimento e a manutenção.
- O Quicksort com mediana de 3 e partição de 3 vias é uma escolha robusta.

O sistema construído oferece uma estrutura flexível para explorar a seleção dinâmica de algoritmos de ordenação, com a calibração sendo um passo explícito e observável.

7. Bibliografia

- cppreference.com. Online C++ language reference.
- Manuais do GCC, GNU Make.