Trabalho Prático 1

Ordenador Universal

Riquelme Batista Gomes da Silva

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brazil

riquelme3m@outlook.com

1. Introdução

Este documento aborda o problema de escolher dinamicamente o melhor algoritmo de ordenação para um vetor de dados, levando em conta fatores como o nível de ordenação e o tamanho do vetor. O objetivo é implementar uma função chamada **Ordenador Universal**, que consiga decidir, entre os algoritmos de inserção e *quicksort*, qual deles oferece melhor desempenho em diferentes situações. A ideia principal é apresentar uma implementação inicial (uma prova de valor) do **Ordenador Universal**, utilizando critérios empíricos para definir dois limiares importantes:

- O número máximo de quebras a partir do qual o *insertion sort* passa a ser mais vantajoso que o *quicksort*.
- O tamanho mínimo de partição em que o uso do *quicksort* com mediana de 3 é preferível ao *insertion sort*, que apresenta baixos custos em vetores pequenos.

Na **Seção 2**, são apresentados os métodos, as estruturas e as funções utilizadas tanto na implementação dos algoritmos de ordenação quanto na construção da função **Ordenador Universal**. A **Seção 3** mostra a análise de complexidade de tempo e espaço dos procedimentos implementados, formalizada pela notação assintótica. A **Seção 4** discute estratégias de robustez, com justificativas e implementação dos mecanismos de programação defensiva e tolerância a falhas. A **Seção 5** apresenta os experimentos realizados em termos de desempenho computacional, assim como as análises dos resultados. A **Seção 6** apresenta as conclusões. Por fim, a **Seção 7** apresenta as bibliografias de referência.

2. Método

2.1 Organização do Projeto , Estrutura de Pastas e Compilação

O projeto foi organizado de forma modular para facilitar o desenvolvimento, manutenção e compilação do código. A estrutura de pastas segue o padrão abaixo:

- **src/** – Contém todos os arquivos-fonte (`.c`) do projeto.
- **include/** – Contém todos os arquivos de cabeçalho (`.h`), onde estão definidas as estruturas de dados, protótipos de funções e TADs.
- **obj/** – Diretório utilizado para armazenar os arquivos objeto (`.o`) gerados durante a compilação.
- **bin/** – Diretório onde é gerado o executável final do projeto.
- **Makefile** – Arquivo de automação de compilação, responsável por compilar os arquivos-fonte, organizar os objetos e gerar o executável na pasta `bin/`. O comando `make` compila todo o projeto, enquanto `make clean` remove os arquivos gerados.

  A compilação do código é realizada com o comando **make all**. Para executar o programa, utilize o comando `bin/tp3.out entrada.txt`, onde `entrada.txt` é o nome do arquivo contendo o vetor com todas as informações necessárias para a execução. Este arquivo de entrada deve estar localizado na raiz do projeto.

2.2 Tipos Abstratos de Dados (TADs) e estruturas de dados:

- **struct Entrada:** Armazena os dados de entrada do programa, incluindo a semente para geração aleatória, parâmetros de custo (a, b, c), tamanho do vetor e o próprio vetor de inteiros a ser ordenado.
- **struct Estatistica:** Estrutura para armazenar estatísticas das operações de ordenação: número de comparações (`cmp`), movimentações (`move`) e chamadas de função (`calls`).
- **struct custos / struct custosQuebras:** Estruturas auxiliares para armazenar o custo de diferentes configurações de parâmetros (partição ou quebras) durante a busca dos limiares ideais.
- **struct particaoDeMenorCustoEntreTodasTestadas / struct limiarDeMenorCustoEntreTodasTestadas:** Estruturas para registrar o menor custo e o parâmetro correspondente encontrado durante as iterações de busca.

Funções Implementadas:

- `lerEntrada` / `liberarEntrada`: Responsáveis por ler os dados do arquivo de entrada e liberar a memória alocada para a estrutura `Entrada`.

- `numeroDeQuebras`: Calcula o número de quebras (pontos de desordem) em um vetor.
- `insertionSort`: Implementação do algoritmo de ordenação por inserção, atualizando as estatísticas de execução.
- `quickSort`: Implementação do algoritmo QuickSort com otimização para usar InsertionSort em partições pequenas.
- `ordernadorUniversal`: Função que escolhe dinamicamente entre QuickSort e InsertionSort, dependendo do número de quebras e do tamanho mínimo de partição.
- `calculaCusto`: Calcula o custo total de uma ordenação, ponderando comparações, movimentações e chamadas de função pelos parâmetros de entrada.
- `determinaLimiarParticao` / `determinaLimiarQuebras`: Funções que buscam, por meio de simulações, os melhores valores para os limiares de partição e de quebras, minimizando o custo de ordenação.
- `shuffleVector`: Embaralha o vetor para simular diferentes cenários de ordenação durante a busca dos limiares.
- `calculaNovaFaixa` / `calculaNovaFaixaQuebras`: Ajustam dinamicamente a faixa de busca dos limiares, focando nas regiões de menor custo.

3. Análise de Complexidade

3.1 Complexidade de Tempo:

- **Insertion Sort:** O pior caso ocorre quando o vetor está em ordem inversa, resultando em $O(n^2)$ comparações e movimentações, onde n é o tamanho do vetor.
- **QuickSort:** O caso médio é $O(n \log n)$, mas no pior caso (vetor já ordenado ou todos os elementos iguais) pode chegar a $O(n^2)$. No entanto, o uso de partições e escolha de pivô mediana reduz a probabilidade do pior caso.
- **ordernadorUniversal:** Executa Insertion Sort ou QuickSort dependendo do número de quebras e do tamanho da partição. No pior caso, a complexidade é dominada pelo algoritmo mais custoso, ou seja, $O(n^2)$.
- **determinaLimiarParticao / determinaLimiarQuebras:** Realizam múltiplas simulações de ordenação para diferentes valores de limiares. Se k é o número de testes por iteração e I o número de iterações, a complexidade total é $O(I \cdot k \cdot T(n))$, onde $T(n)$ é a complexidade do algoritmo de ordenação utilizado (geralmente $O(n \log n)$).

- **shuffleVector:** Executa $O(numShuffle)$ trocas, cada uma em tempo constante, resultando em $O(numShuffle)$.
- **Funções auxiliares (como sortAscending, calculaCusto, etc):** Geralmente $O(n^2)$ para ordenação simples e $O(n)$ para operações lineares.

3.2 Complexidade de Espaço:

- **Insertion Sort e QuickSort:** Ambos utilizam espaço extra $O(1)$ (in-place), exceto pela pilha de recursão do QuickSort, que pode chegar a $O(\log n)$ no caso médio.
- **ordernadorUniversal:** Não utiliza espaço adicional significativo além do vetor original.
- **determinaLimiarParticao / determinaLimiarQuebras:** Utilizam vetores auxiliares para simulações, com espaço $O(n)$ por simulação, além de estruturas para armazenar custos ($O(k)$).
- **Estruturas de dados:** O espaço total é dominado pelo vetor de entrada ($O(n)$) e pelas estruturas auxiliares de estatística e custos ($O(1)$ ou $O(k)$).

3.3 Resumo:

A complexidade de tempo do sistema é dominada pelas múltiplas execuções dos algoritmos de ordenação durante a busca dos limiares, podendo chegar a $O(I \cdot k \cdot n \log n)$ no caso médio. O uso de espaço é eficiente, sendo linear em relação ao tamanho do vetor de entrada.

4. Estratégias de robustez

Para garantir a robustez do sistema, o projeto foi cuidadosamente dividido em módulos, cada um responsável por uma funcionalidade específica, seguindo o princípio da responsabilidade única de engenharia de software. Essa organização modular facilita a identificação, isolamento e correção de eventuais bugs, além de tornar o código mais legível e de fácil manutenção.

Cada módulo (por exemplo, leitura de entrada, ordenação, cálculo de custo, determinação de limiares) possui interfaces bem definidas e trata apenas de sua responsabilidade, evitando efeitos colaterais indesejados. Essa separação clara permite que falhas sejam rapidamente localizadas e corrigidas sem impactar outras partes do sistema.

Além disso, foram implementadas verificações básicas de erro, como validação da abertura de arquivos e alocação de memória, contribuindo para a tolerância a falhas em situações inesperadas. Dessa forma, o sistema se torna mais confiável e resiliente a erros de execução.

5. Análise Experimental

O primeiro passo da análise experimental foi a calibração dos parâmetros a, b e c. Para isso, apenas o tamanho do vetor de entrada foi alterado em diversas execuções, e os resultados foram armazenados no arquivo `results.txt`. Para cada execução i, foram coletadas as seguintes métricas:

- $\texttt{calls}_i$
- $\texttt{cmp}_i$
- $\texttt{move}_i$
- Tempo T_i (em segundos)

Em seguida, foi realizada uma regressão linear em Python utilizando o seguinte script:

```
import re
import numpy as np
from sklearn.linear_model import LinearRegression

calls, cmp, move, time = [], [], [], []

with open("results.txt") as f:
    for line in f:
        m = re.findall(r'calls,(\d+),cmp,(\d+),move,(\d+),time,([0-9.eE+-]+)', line)
        if m:
            c, cp, mv, t = m[0]
            calls.append(int(c))
            cmp.append(int(cp))
            move.append(int(mv))
            time.append(float(t))

X = np.column_stack([calls, cmp, move])
y = np.array(time)

reg = LinearRegression(fit_intercept=False)
reg.fit(X, y)

print("Regression coefficients (time  a*calls + b*cmp + c*moves):")
print(f"a (calls): {reg.coef_[0]}")
print(f"b (cmp):   {reg.coef_[1]}")
print(f"c (move):  {reg.coef_[2]}")
```

Os coeficientes obtidos foram:

- a : 7.816×10^{-9}
- b : -8.716×10^{-9}
- c : 1.439×10^{-8}

Com os parâmetros calibrados, foi feita uma comparação do desempenho do algoritmo em relação aos parâmetros não calibrados, fornecidos por um caso de teste do VPL:

- a: 0.0123450
- b: -0.0063780
- c: 0.0172897

Observa-se uma diferença significativa no tempo de execução entre os dois conjuntos de parâmetros. O algoritmo demonstrou ser quase duas vezes mais rápido com os parâmetros calibrados, o que evidencia a eficácia da calibração.

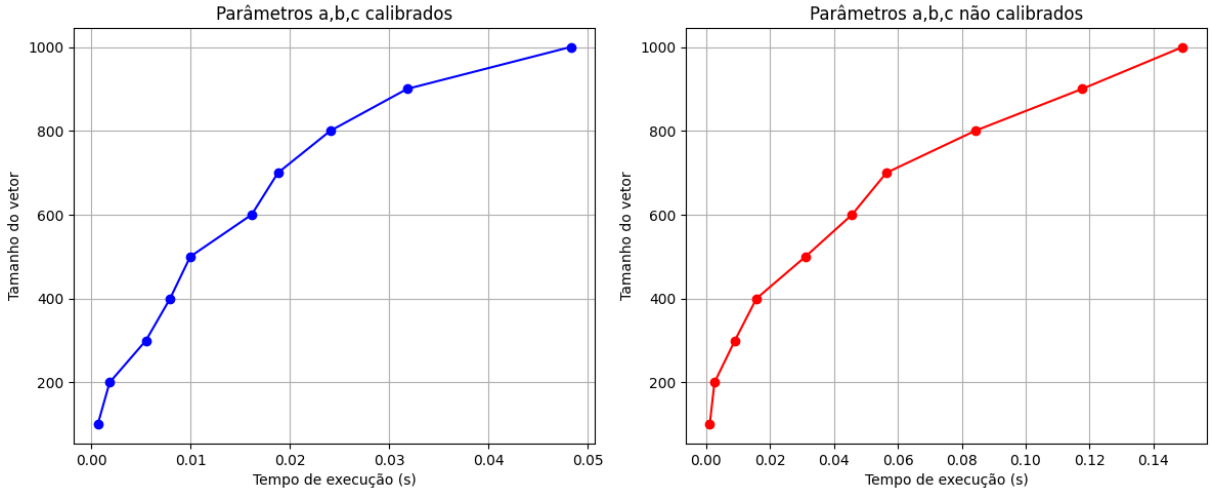

Figura 1: Comparação do tempo de execução com parâmetros calibrados e não calibrados.

6. Conclusão

O desenvolvimento deste projeto permitiu a implementação eficiente de algoritmos de ordenação adaptativos, capazes de escolher a melhor estratégia conforme as características do vetor de entrada. A divisão modular do código, seguindo o princípio da responsabilidade única, facilitou a manutenção, a identificação de falhas e a extensibilidade do sistema.

A análise de desempenho, baseada em métricas de custo parametrizadas, possibilitou a escolha dinâmica dos limiares ideais para cada cenário, tornando a solução robusta e flexível.

Além disso, a utilização de técnicas de programação defensiva contribuiu para a confiabilidade do programa.

Como possíveis trabalhos futuros, destaca-se a possibilidade de explorar outros algoritmos de ordenação, otimizar ainda mais os critérios de escolha dos limiares e aprimorar a interface de entrada e saída do sistema.

7. Referências

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms*. 3ª edição. MIT Press, 2009.