SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho 3 - AEDS 1

Ordenação de rochas minerais

Bruno Vicentini Ribeiro - 5907
Pedro Paulo Paz - 5937
Vitor Mathias Andrade - 5901

Florestal - MG
2025

Sumário

1. Introdução

Este trabalho dá continuidade ao estudo sobre rochas minerais em desenvolvimento e consiste na criação de um programa em linguagem C para a ordenação de rochas minerais. Para isso, serão aplicados conceitos de Tipos de Dados Abstratos (TADs), que estruturam os dados e facilitam sua manipulação, bem como estratégias para potencializar a eficiência computacional. Ademais, serão coletadas métricas de desempenho, como o número de comparações, movimentações e tempo de execução, para comparar os algoritmos SelectionSort e QuickSort.

2. Organização

Na figura 1, observa-se o panorama geral da organização do projeto, no qual são elencados os TADs necessários para a criação do algoritmo.

Além disso, os arquivos foram divididos em .h (responsável pela declaração das funções e pela criação das structs) e .c (responsável pela implementação das funções previamente declaradas e pela utilização das structs), para garantir uma melhor organização e funcionamento do código.

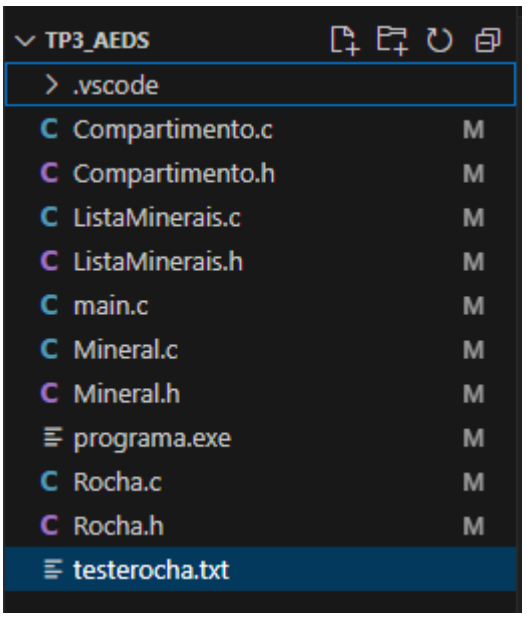

Figura 1: Repositório do projeto.

3. Desenvolvimento

A seguir, o trabalho envolve a reimplementação dos TADs: (3.1) TAD RochaM.c e (3.2) TAD Compartimento.

3.1. TAD RochaM.c

O TAD RochaM.c é responsável por representar rochas minerais e suas características, incluindo a categorização das rochas com base em propriedades como latitude, longitude, peso e materiais constituintes.

3.2. TAD Compartimento

O TAD Compartimento possui a função de armazenar informações acerca das rochas minerais. Assim, ele foi utilizado para armazenar as rochas minerais em um vetor e permitir operações de ordenação. Para a ordenação, foram implementados dois algoritmos: um simples, (3.2.1) SelectionSort, e um sofisticado, (3.2.2) QuickSort.

3.2.1. SelectionSort

O SelectionSort é um algoritmo de ordenação simples que funciona selecionando repetidamente o menor elemento do vetor e movendo-o para a posição correta. A sua complexidade é de $O(n^2)$.

A imagem a seguir ilustra a implementação do algoritmo SelectionSort na linguagem C.

```c
void SelectionSort(Compartimento *LRocha, int n) {
    int i, j, Min;
    int mov = 0, comp = 0;
    RochaMineral Aux;

    for (i = 0; i < n - 1; i++) {
        Min = i;
        for (j = i + 1 ; j < n; j++)
            if (LRocha->ListaR[j].peso < LRocha->ListaR[Min].peso) {
                Min = j;
                comp += 1;
            }
        Aux = LRocha->ListaR[Min];
        LRocha->ListaR[Min] = LRocha->ListaR[i];
        LRocha->ListaR[i] = Aux;
        mov += 1;
    }
```

Figura 2: Implementação do SelectionSort com vetores.

3.2.2. QuickSort

O QuickSort é um algoritmo mais sofisticado que utiliza o método de divisão e conquista, escolhendo um pivô para dividir a lista em duas partes, ordenando-as recursivamente. A sua complexidade é de O(n log n), sendo mais eficiente na maioria dos casos.

A implementação do QuickSort foi estruturada em três partes:

1. Função Ordena: realiza a ordenação da lista;

2. Função Particao: divide o vetor em duas partes;
3. Função QuickSort: chama a função Ordena com os parâmetros necessários.

```c
void Ordena(int esq, int dir, RochaMineral *A)
{
    int i, j;
    Particao(esq, dir, &i, &j, A);
    if (esq < j)
    {
        Ordena(esq, j, A);
    }
    if (i < dir)
    {
        Ordena(i, dir, A);
    }
}
```

Figura 3: Implementação da função Ordena.

```c
void QuickSort(RochaMineral *A, int n)
{
    Ordena(0, n - 1, A);
}
void Particao(int esq, int dir, int *i, int *j, RochaMineral *A)
{
    RochaMineral pivo, aux;
    *i = esq;
    *j = dir;
    pivo = A[(*i + *j) / 2];
    do
    {
        while (pivo.peso > A[*i].peso)
            (*i)++;
        while (pivo.peso < A[*j].peso)
            (*j)--;
        if (*i <= *j)
        {
            aux = A[*i];
            A[*i] = A[*j];
            A[*j] = aux;
            (*i)++;
            (*j)--;
        }
    } while (*i <= *j);
}
```

Figura 4: Implementação da função Particao.

```c
void QuickSort(RochaMineral *A, int n)
{
    Ordena(0, n - 1, A);
}
```

Figura 5: Implementação da função QuickSort.

4. Compilação e execução

Para compilar e executar o código é necessário inserir, no terminal do Visual Studio Code, os seguintes comandos:

- gcc Compartimento.c ListaMinerais.c Mineral.c Rocha.c main.c -o programa.exe
- .\programa.exe

Após a inserção dos comandos de compilação e execução, será exibido uma mensagem solicitando ao usuário que insira o nome do arquivo de entrada (arquivo .txt).

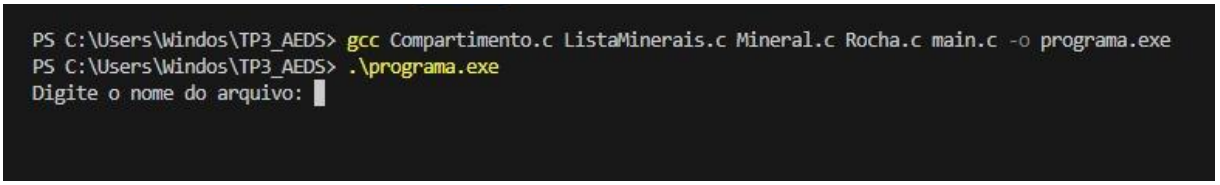

Figura 6: Solicitação para inserir o nome do arquivo de entrada.

Em seguida, é exibida uma mensagem solicitando ao usuário que selecione o algoritmo de ordenação desejado: SelectionSort ou QuickSort.

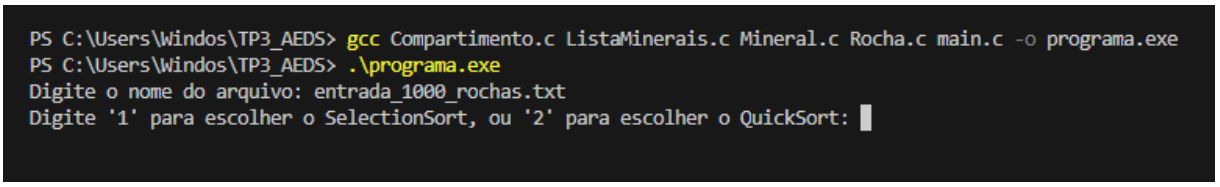

Figura 7: Solicitação para selecionar o algoritmo desejado.

5. Resultados

Para cada algoritmo de ordenação, foram registradas as seguintes métricas de desempenho: (5.1) SelectionSort e (5.2) QuickSort.

5.1. SelectionSort

5.1.1. Compartimento com 250 rochas:

5.1.1.1. Número de comparações realizadas: 31.125

5.1.1.2. Número de movimentações de itens no vetor: 239

5.1.1.3. Tempo total de execução: 0.00000 segundos

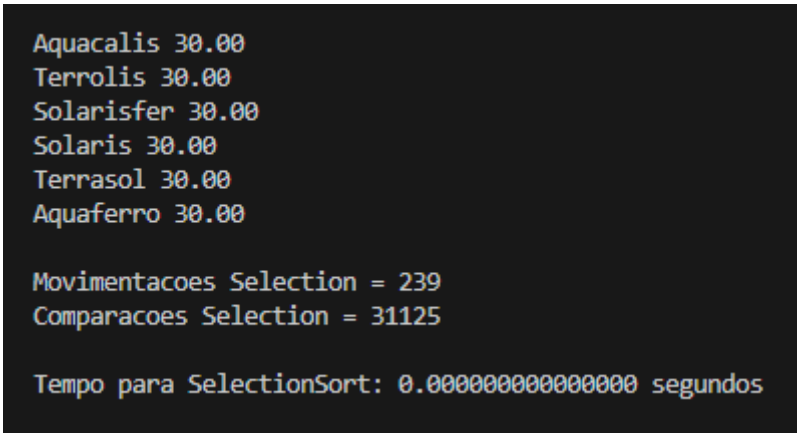

Figura 8: Resultados do SelectionSort com 250 rochas.

5.1.2. Compartimento com 1000 rochas:

5.1.2.1. Número de comparações realizadas: 499.500

5.1.2.2. Número de movimentações de itens no vetor: 964

5.1.2.3. Tempo total de execução: 0.00200 segundos

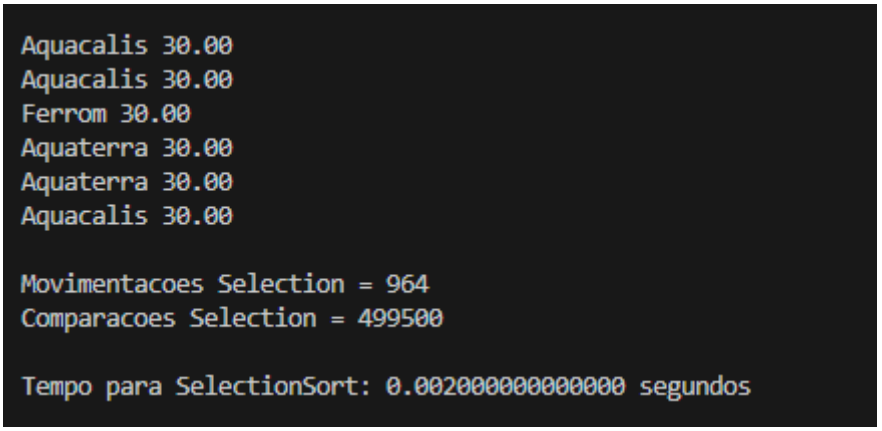

Figura 9: Resultados do SelectionSort com 1000 rochas.

5.1.3. Compartimento com 10000 rochas:

5.1.3.1. Número de comparações realizadas: 49.995.000

5.1.3.2. Número de movimentações de itens no vetor: 9.656

5.1.3.3. Tempo total de execução: 0.18200 segundos

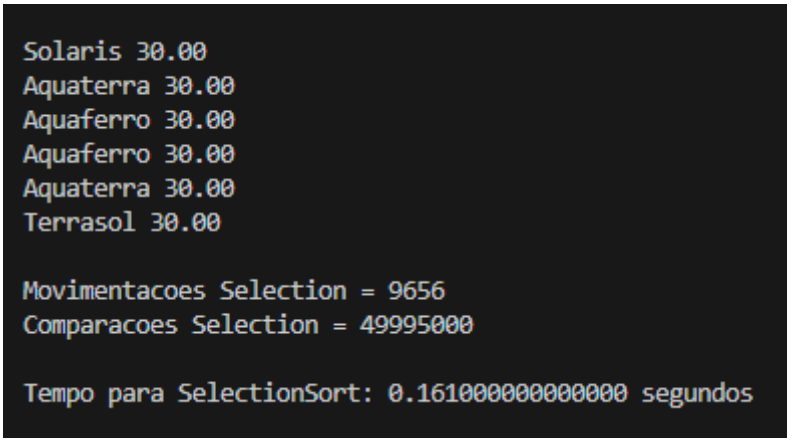

Figura 10: Resultados do SelectionSort com 10000 rochas.

5.2. QuickSort

5.2.1. Compartimento com 250 rochas:

5.2.1.1. Número de comparações realizadas: 3.118

5.2.1.2. Número de movimentações de itens no vetor: 650

5.2.1.3. Tempo total de execução:0.00000 segundos

```
Aquacalis 30.00
Solaris 30.00
Aquaferro 30.00
Aquaterra 30.00
Ferrom 30.00

Movimentacoes QuickSort: 650
Comparacoes QuickSort: 3118

Tempo para QuickSort: 0.0000000 segundos
```

Figura 11: Resultados do QuickSort com 250 rochas.

5.2.2. Compartimento com 1000 rochas:

5.2.2.1. Número de comparações realizadas: 14.570

5.2.2.2. Número de movimentações de itens no vetor: 3.452

5.2.2.3. Tempo total de execução: 0.00000 segundos

```
Aquaterra 30.00
Aquacalis 30.00
Aquacalis 30.00
Calquer 30.00
Aquaterra 30.00
Aquacalis 30.00

Movimentacoes QuickSort: 3452
Comparacoes QuickSort: 14570

Tempo para QuickSort: 0.0000000 segundos
```

Figura 12: Resultados do QuickSort com 1000 rochas.

5.2.3. Compartimento com 10000 rochas:

5.2.3.1. Número de comparações realizadas: 196.951

5.2.3.2. Número de movimentações de itens no vetor: 50.219

5.2.3.3. Tempo total de execução: 0.00500 segundos

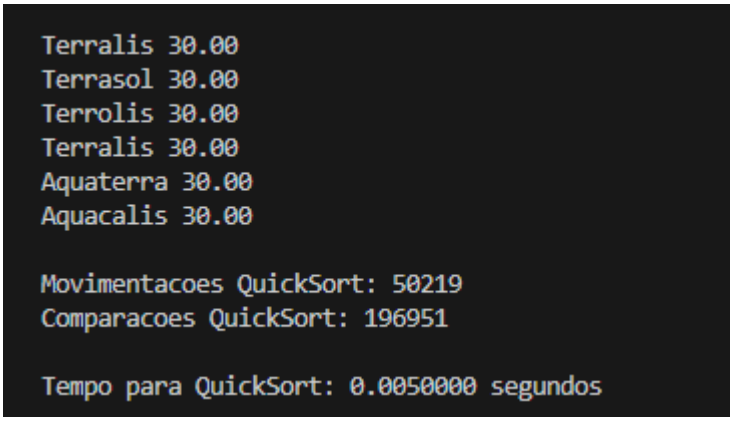

Figura 13: Resultados do QuickSort com 10000 rochas.

Por meio dos resultados, conclui-se que o algoritmo de ordenação QuickSort é mais eficiente que o SelectionSort. Embora o QuickSort tenha feito mais movimentações devido à escolha do pivô, o número de comparações foi muito menor em comparação com o SelectionSort, o que justifica seu tempo reduzido.

Além disso, é possível perceber que o aumento do tempo de execução do QuickSort é inferior ao crescimento do tempo de execução do SelectionSort. Isso pode ser justificado pelo comportamento assintótico dos dois algoritmos, já que o caso médio do QuickSort é O(n log n), enquanto o do SelectionSort é O(n²).

Tabela 1: Tabela sintética comparativa de métricas de desempenho.

Algoritmo	Tamanho do compartimento	Número de comparações realizadas	Número de movimentações de itens no vetor	Tempo total de execução (segundos)
SelectionSort	250	31.125	239	0.00000
SelectionSort	1000	499.500	964	0.00200
SelectionSort	10000	49.995.000	9.656	0.16100
QuickSort	250	3.118	650	0.00000
QuickSort	1000	14.570	3.452	0.00000
QuickSort	10000	196.951	50.219	0.00500

6. Conclusão

A interpretação das métricas de desempenho permitiu uma noção nítida das vantagens e desvantagens de cada algoritmo em situações distintas:

Tabela 2: Tabela comparativa de vantagens e desvantagens dos algoritmos de ordenação.

Algoritmo	Vantagens	Desvantagens

SelectionSort	● Fácil de implementar; ● Independente da entrada.	● Ineficiente para grandes conjuntos; ● Não é estável; ● Não é adaptativo, ou seja, sempre executa $O(n^2)$ comparações.
QuickSort	● Rápido na prática, com complexidade O(n log n), sendo mais eficiente que algoritmos quadráticos como o SelectionSort; ● Bom desempenho para grandes conjuntos.	● No pior caso, com o pivô mal escolhido, pode ter desempenho $O(n^2)$; ● Não é estável.

Em resumo, a análise deste trabalho demonstra que o QuickSort apresenta um melhor desempenho para grandes conjuntos de dados, sendo consideravelmente mais rápido que o SelectionSort. Contudo, observa-se que, para pequenas listas, a diferença de tempo entre os dois algoritmos de ordenação é quase insignificante.

Além disso, este trabalho evidencia a importância da escolha correta do algoritmo de ordenação a ser implementado, visto que o algoritmo escolhido pode influenciar tanto positivamente, quanto negativamente no desempenho do sistema criado. Em casos com grande número de dados a serem ordenados, os algoritmos de ordenação simples são altamente inviáveis, pois o desempenho deles é inferior aos algoritmos mais complexos, como o QuickSort, ShellSort e HeapSort.

7. Referências

[1] SILVA, Thais. Algoritmos e estruturas de dados I: ordenação simples. Universidade Federal de Viçosa, campus Florestal, 2023. Acesso em: 22 de janeiro de 2025.
[2] SILVA, Thais. Algoritmos e estruturas de dados I: QuickSort. Universidade Federal de Viçosa, campus Florestal, 2023. Acesso em: 22 de janeiro de 2025.