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

Trabalho 2 - AEDS 1

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

Florestal - MG
2024

Sumário

1. Introdução

Este trabalho tem como objetivo aferir o desempenho dos algoritmos em sua execução real. Para isso, foram utilizadas as linguagens de programação C — para a criação do sistema — e Python — para criar um gerador de casos de teste.

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. Somado a isso, tem-se, no repositório, a pasta “testes_rochas”, na qual estão os casos de teste utilizados para a medição do desempenho do algoritmo.

Além disso, dividiram-se os arquivos 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.

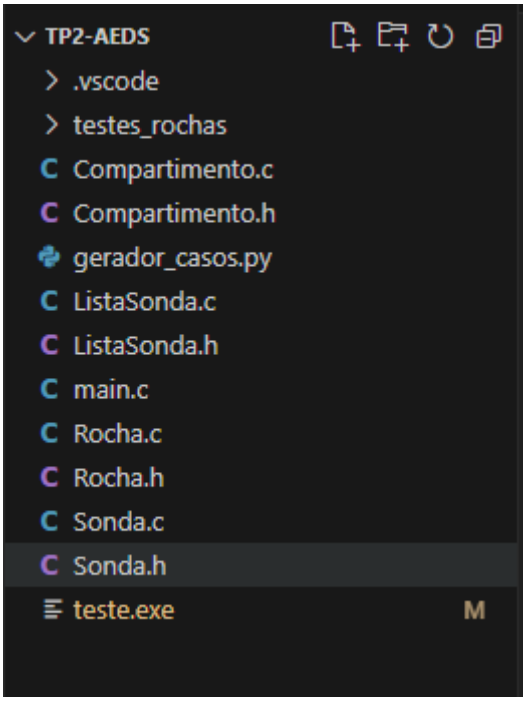

Figura 1: Repositório do projeto.

3. Desenvolvimento

Para analisar todas as possíveis combinações e inserir nas sondas as três melhores encontradas, foi utilizado o algoritmo de força bruta, uma técnica de resolução de problemas que testa todas as combinações possíveis até encontrar a solução desejada. Apesar de ser uma técnica ineficiente, especialmente para problemas de longa escala, o algoritmo apresenta a vantagem de garantir que a solução encontrada seja correta. Juntamente à força bruta, foi utilizado o método de busca exaustiva, que consiste em procurar as melhores combinações de todas as que foram encontradas pelo método de força bruta.

Além disso, para medir o desempenho do algoritmo, foi utilizada a função “clock”, presente na biblioteca “time.h”.

```c

void forcabruta(LCompartimento *rochas, int capacidade, int numSondas, int N, LSonda *melhorSolucao) {
    for (int i = 0; i < numSondas; i++) {
        int melhorValorSonda = -1;
        int melhorNumItensSonda = -1; // Para desempate
        LCompartimento melhorCombinacaoSonda;
        FLVaziaRocha(&melhorCombinacaoSonda);

        for (int k = 0; k < (1 << N); k++) { // Iterar por todas as combinações
            int pesoTotalCombinacao = 0;
            int valorTotalCombinacao = 0;
            int numItensCombinacao = 0; // Contar itens na combinação
            bool combinacaoValida = true;

            // Verificar a validade da combinação ANTES de calcular peso e valor
            for (int j = 0; j < N; j++) {
                if ((k >> j) & 1 && rochas->rochas[j].usada) {
                    combinacaoValida = false;
                    break; // Se uma rocha já foi usada, a combinação é inválida
                }
            }

            // Calcular peso e valor apenas para combinações válidas
            if (combinacaoValida) {
                for (int j = 0; j < N; j++) {
                    if ((k >> j) & 1) {
                        pesoTotalCombinacao += rochas->rochas[j].pesoI;
                        valorTotalCombinacao += rochas->rochas[j].valorI;
                        numItensCombinacao++;
                    }
                }

                // Verificar se a combinação é melhor que a atual
                if (pesoTotalCombinacao <= capacidade &&
                    (valorTotalCombinacao > melhorValorSonda ||
                     (valorTotalCombinacao == melhorValorSonda && numItensCombinacao > melhorNumItensSonda))) {
                    melhorValorSonda = valorTotalCombinacao;
                    melhorNumItensSonda = numItensCombinacao;
                    FLVaziaRocha(&melhorCombinacaoSonda); // Limpar a melhor combinação anterior

                    for (int j = 0; j < N; j++) {
                        if ((k >> j) & 1) {
                            LInsereRocha(&melhorCombinacaoSonda, rochas->rochas[j]);
                        }
                    }
                }
            }
        }
```

Figura 2: Função forcabruta.

4. Compilação e execução

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

- gcc ListaSonda.c Rocha.c Sonda.c Compartimento.c main.c -o teste.exe
- .\teste.exe

Após a inserção das informações, será solicitado o nome do arquivo de entrada, que deve ser colocado seguindo o exemplo a seguir:

```
PS D:\GIT HUB\TP2-AEDS> .\teste.exe
Nome do arquivo de entrada(com extensao): testes_rochas/teste10.txt
```

Figura 3: Exemplo de inserção de arquivo.

No exemplo, os nomes “testes_rochas” e “teste10.txt” nomeiam a placa que comporta o arquivo de entrada e o arquivo de entrada, respectivamente.

5. Resultados

Os computadores utilizados possuem as seguintes configurações:

- Computador 1 — utilizado para exibir os resultados abaixo):
    - Processador Intel Core I3 10100F;
    - Placa de vídeo NVIDIA GTX 1650;
    - SSD NVME - 240 GB;
    - 16 GB de memória RAM;
    - Sistema Operacional Windows 10.

- Computador 2:
    - Sistema Operacional: Windows 11 Home;
    - Processador: AMD Ryzen™ 7 (5700U);
    - Placa de vídeo: Placa de vídeo integrada AMD Radeon™;
    - Memória RAM: 12 GB;
    - Armazenamento: SSD M2 PCIe de 512GB.

- Computador 3:
    - Sistema Operacional: Windows 11 Home;
    - Processador Intel(R) Core(TM) i7-1165G7;
    - Placa de vídeo: Intel Iris Xe Graphics integrada com o processador;
    - Memória RAM: 16 GB;
    - Armazenamento: SSD NVMe de 512 GB.

Os resultados obtidos em cada caso de teste podem ser visualizados a seguir:

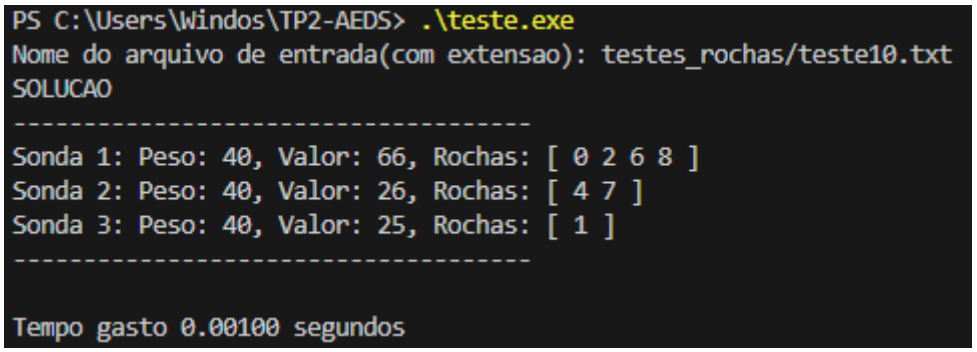

Figura 4: Caso de teste envolvendo 10 rochas.

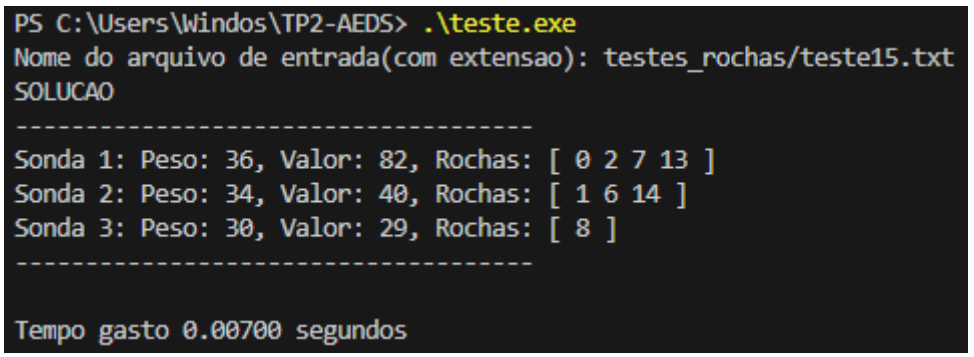

Figura 5: Caso de teste envolvendo 15 rochas.

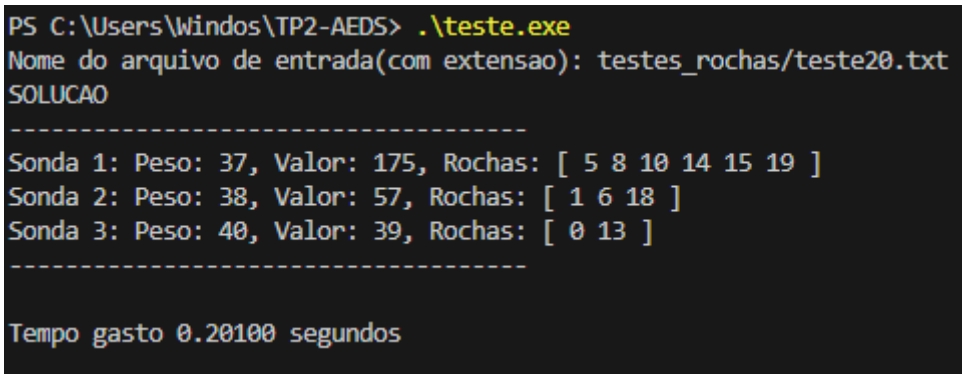

Figura 6: Caso de teste envolvendo 20 rochas.

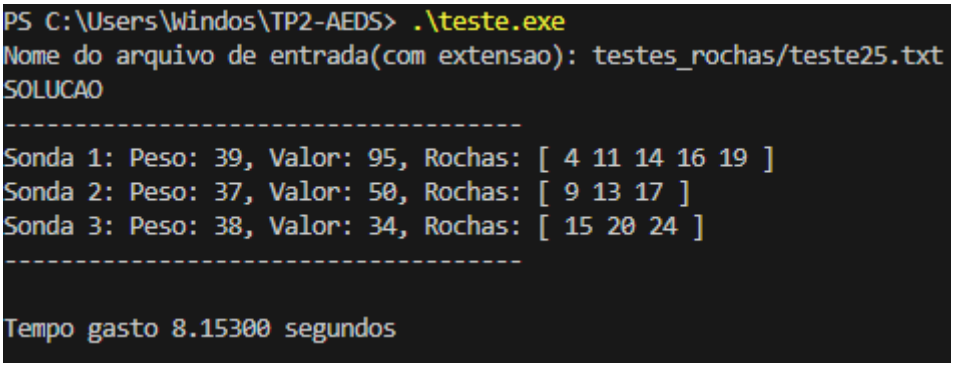

Figura 7: Caso de teste envolvendo 25 rochas.

Levando isso em consideração, afirma-se que o tempo de execução do código aumenta, de maneira gradativa, mediante o número de rochas presente em cada caso de teste. Isso porque, quanto maior o número de rochas, maior a quantidade de comparações a serem realizadas pelo algoritmo.

Além disso, para valores maiores ou iguais a 50, o código não consegue ser executado devido à falta de memória, que impossibilita o armazenamento do número de rochas solicitado. Essa limitação de memória pode ser explicada pelo fato de que técnica a força bruta é exponencial, ou seja, o número de comparações, o tempo de execução e o espaço necessário na memória aumentam exponencialmente.

6. Conclusão

A partir deste trabalho, compreende-se a importância de criar algoritmos eficientes, capazes de realizar tarefas de forma rápida e precisa. Além disso, constata-se que os algoritmos desempenham papéis cruciais por serem essenciais no processamento de grandes volumes de dados.

Tendo isso em vista, a utilização do algoritmo de força bruta não é recomendada para valores de rochas maiores que 100, haja vista a sua pouca eficiência diante de casos de elevada quantidade de rochas, principalmente. Outrossim, é válido destacar que outros algoritmos, como o da mochila, possuem melhor desempenho para a realização de um grande número de comparações, pois utilizam a técnica de programação dinâmica, o que resulta em soluções exatas de forma mais rápida quando comparadas ao algoritmo de força bruta.

7. Referências

[1] Github. Disponível em: <https://github.com/PapaArt/AEDSI_Practical_Works/blob/main/Tp2/doc/CCF_211___Algoritmos_e_Estruturas_de_Dados_I_Trabalho_Pr_tico_02___SAT%20.pdf>. Último acesso em: 12 de dezembro de 2024.

[2] Github. Disponível em: <https://github.com/ambarmodi/Knapsack-Problem/blob/master/Q1-1_brute_force_knapsack.c>. Último acesso em: 12 de dezembro de 2024.

[3] How to solve Knapsack problem using exhaustive search | Brute Force. 2021. Disponível em: <https://www.youtube.com/watch?v=zQyy7s4Lp6E>. Último acesso em: 12 de dezembro de 2024.

[4] Brute force algorithms: The power of exhaustive search. Disponível em: <https://dev.to/akashdev23/brute-force-algorithms-the-power-of-exhaustive-search-1bab>. Último acesso em: 12 de dezembro de 2024.