Documentação TP2

Matheus Ibrahim Fonseca Magalhães
Matrícula: 2018046912

1. Introdução

O problema passado, contextualizado como uma viagem para diferentes ilhas(cada uma com um custo e uma pontuação dada pelos viajantes), consiste em duas variações do problema da mochila, uma com repetição de itens(ilhas) permitidas, cuja solução deve ser realizada através de um algoritmo guloso, e a outra sem essa permissão, que deve ser resolvida utilizando programação dinâmica.

2. Código

2.1) Linguagem e ambiente

O programa foi escrito na linguagem de programação C++ em uma máquina utilizando o sistema operacional Linux e compilado por um arquivo *makefile*(incluído no envio).

2.2) Arquivos

O programa possui cinco arquivos: *main.cpp*, com a função *main*, responsável por realizar as operações de entrada e chamar as operações das outras classes. *Island.h* e *Island.cpp*, que definem, respectivamente, o contrato e a implementação da classe *Island*. *Trip.h* e *Trip.cpp*, similarmente utilizados para definir contrato e implementar a classe *Trip*.

2.3)Classes

O programa possui duas classes: *Island* e *Trip*.

A classe *Island* define uma entidade para cada ilha do programa, possuindo três atributos: custo, pontuação e custo-benefício, calculado através de uma relação entre custo e pontuação. Possui apenas funções *getters*, para os atributos, e uma função que calcula o custo benefício, utilizada apenas internamente, chamada FindValue().

A classe *Trip* representa a viagem, e possui todos os atributos necessários para calcular a pontuação máxima e a duração da viagem, com ou sem repetição de ilhas. Esses atributos são o dinheiro máximo que pode ser gasto, a quantidade de ilhas que existem para ser visitadas e uma lista com todas as ilhas e seus respectivos atributos. Possui uma função de ordenação, auxiliar para a função que calcula a viagem com repetição de ilhas, implementada com base no algoritmo *merge sort*. Possui outra função auxiliar *max* que retorna o maior entre dois inteiros. E, por último, possui as duas funções que calculam as viagens com ou sem repetição de ilhas,

chamadas MaxPointsRepeatingIslands() e MaxPointsWithoutRepeating(), respectivamente.

2.4) Detalhamento das principais funções do programa

O programa possui duas funções principais para resolver o problema proposto(cada uma possui funções auxiliares), cada uma responsável por realizar o cálculo de um dos problemas.

A função MaxPointsRepeatigIslands() calcula a pontuação máxima e a duração da viagem que permite passar mais de um dia na mesma ilha, a partir de um algoritmo guloso. Para tal, as ilhas são ordenadas de forma decrescente com base no seu custo benefício, utilizando uma implementação adaptada do *merge sort* para tal(A ordenação também leva em conta outros detalhes, para lidar com possíveis exceções, como se duas ilhas possuírem o mesmo custo benefício qual é a mais vantajosa). Com as ilhas ordenadas, basta percorrer essa lista, e, para cada ilha, calcular se ainda existe dinheiro suficiente para ser gasto nela, e, em caso positivo, quantos dias é possível gastar na ilha, encontrando, assim, a duração da viagem e a pontuação obtida, já que, sabendo quantos dias será gasto em cada ilha, é possível também calcular quantos pontos cada ilha irá render à viagem.

A função MaxPointsWithoutRepeating() calcula a pontuação máxima e a duração da viagem em que só é possível passar um dia em cada ilha, utilizando para tal a técnica de programação dinâmica. O algoritmo usa a programação dinâmica com estratégia *bottom-up,* portanto calcula todos os valores anteriores ao valor final de entrada. Para isso, é criada uma matriz de tamanho quantidadeIlhas x dinheiroDisponivel, cuja posição final possui o resultado do problema. A matriz é preenchida dentro de dois laços, um para cada dimensão da matriz. Para cada posição, dentro dos laços, existem três possibilidades: Primeiro, a quantidade de ilhas ou o dinheiro disponível é 0, e a pontuação que pode ser obtida é também 0. Segundo, existem ilhas e dinheiro disponível, e o custo da ilha é menor ou igual ao dinheiro, caso em que é necessário descobrir se a solução ótima existe com a inserção daquela ilha ou sem a inserção dela, por isso a matriz naquela posição recebe o valor máximo entre uma solução sem ela(já calculada, pontuação de uma solução que desconsidera essa ilha e possui o mesmo custo total de viagem) e uma solução com ela(que é a pontuação atribuída a ilha somada a uma solução que desconsidera essa ilha e possui um custo que é igual ao custo da viagem com a ilha incluída menos o custo da ilha, também já calculada). Terceiro, existem ilhas e dinheiro disponível, mas o custo da ilha é superior ao dinheiro, caso em que, sem verificação, é garantido que a ilha não está incluída na solução, e a posição da matriz recebe o valor já calculado da solução sem a ilha.

Em seguida, para calcular a duração dessa viagem, é preciso fazer o caminho de volta na matriz, para identificar quais ilhas foram incluídas na solução. Dessa forma, a matriz é percorrida a partir da última posição. Se o valor na matriz for igual ao valor da matriz na mesma coluna e uma linha acima(posição com mesmo custo, desconsiderando a ilha da linha atual), isso quer dizer que a ilha não foi incluída. Caso contrário, a ilha foi incluída, incrementado a duração da viagem em um dia, e a matriz passa para a posição com o custo da ilha incluída retirado, para verificar se a ilha anterior foi incluída. Dessa forma, a cada vez que uma ilha for descoberta como parte da solução, o número de dias da viagem é incrementado, encontrando, ao final, a duração total da viagem.

2.5) Fluxo do programa

O programa começa recebendo um parâmetro, o nome do arquivo que possui as entradas para a execução do programa. Esse arquivo é armazenado em uma variável do tipo *ifstream*, e, a partir dele, as entradas dos programas são lidas e armazenadas em variáveis. Após a leitura das entradas são inicializadas as entidades do programa(as ilhas e, em seguida, a viagem). Após a inicialização da viagem, o programa chama as funções que calculam os resultados, já detalhadas. As próprias funções imprimem os resultados na tela.

2.6) Decisões de implementação

a) **Linguagem C++**

A linguagem C++ permite, ao contrário de C, uso de técnicas de Programação Orientada a Objetos, além de possuir uma construção intuitiva para leitura de entradas via arquivos, especialmente com a biblioteca *fstream* e o tipo *ifstream*. Por isso, a linguagem C++ oferece mais suporte a forma como a implementação foi montada.

b) **Entidades**

O problema da mochila clássico considera dois vetores(um com pesos, outro com valor dos itens), que seriam os vetores de custo e pontuação das ilhas nessa versão do problema. Contudo, aproveitando as vantagens da programação Orientada a Objetos e considerando que cada ilha precisaria também do próprio custo benefício, foi entendido que a solução seria mais intuitiva criar uma entidade que possuísse todas as informações, permitindo utilizar apenas um *array* de ilhas ao invés de dois vetores separados de inteiros.

A classe *Trip* foi criada especialmente para auxiliar a modularização do código, que fica mais organizado com a divisão de funções entre os arquivos.

c) **Casos de teste**

Além dos três casos de teste fornecidos pelos monitores, foram criados casos de teste extra, para verificar completamente o funcionamento do programa e das operações. Todos os casos de teste(fornecidos e criados, com as entradas e as respectivas saídas obtidas) estão incluídos no envio, na pasta dataset.

d) **Instruções de compilação e execução**

No envio já foi incluído um arquivo executável chamado **tp2**, portanto não é necessário compilar o programa. De qualquer forma, também está incluído no envio um arquivo *makefile* que, se executado com o comando *make*, irá gerar outro arquivo executável chamado **tp2**. Para executar o programa, basta utilizar, em um terminal do linux, o comando ./tp1 dataset/CasoDeTeste.txt, ou, em um terminal do windows, tp1.exe dataset/CasoDeTeste.txt.

3. Análise de complexidade

3.1) Tempo

Definindo quantidade de ilhas como **m** e dinheiro disponível como **n**, o custo do programa é:

Leitura das entradas: **O(m)**.

Construção do vetor de ilhas: as operações das ilhas possuem custo O(1), e existem m ilhas, portanto **O(m)**.

Construção de entidade Trip: **O(1)**.

Cálculo da viagem com repetição de ilhas: Custo da ordenação com *merge sort*(O(m log m)) mais custo linear para percorrer o array ordenado (O(m)), portanto **O(m log m)**.

Cálculo da viagem sem repetição de ilhas: Custo para percorrer uma matriz de tamanho m x n, já que qualquer operação em cada posição será constante(O(m n)), mais o custo de encontrar a duração, que, percorre a matriz, no máximo, com custo linear em m, já que realiza apenas operações de custo constante para cada ilha para verificar se ela faz parte da solução(O(m)), portanto **O(m n)**.

3.2) Espaço

Definindo novamente a quantidade de ilhas como **m** e dinheiro disponível como **n**, o custo de espaço do programa é:

Leitura das entradas: **O(m)**

Construção da entidade Trip: como possui a criação de um array de **m** ilhas, **O(m).**

Cálculo da viagem com repetição de ilhas: custo de espaço auxiliar do merge sort, **O(m)** somado com custo da criação do array que armazena o retorno do merge sort, **O(m)**.

Cálculo da viagem sem repetição de ilhas: custo de criação da matriz para utilizar programação dinâmica, portanto **O(m n)**

4. **Prova de corretude**

Algoritmo guloso: Como o vetor já está ordenado de forma decrescente por custo benefício, é garantido que $v_0 / w_0 > v_1 / w_1 > v_2 / w_2 > ... > v_n / w_n$. Portanto, como a repetição de itens é permitida, a solução ótima vai possuir o máximo de vezes as ilhas com melhor custo benefício, somando seus pontos, ou seja, o máximo de dias que puderem ser gastos na primeira ilha somado com o máximo a ser gasto na segunda e assim por diante. Dessa forma, o algoritmo guloso encontra a pontuação máxima com repetição de ilhas permitida, na maioria dos casos. Contudo, devido à limitação de tempo imposta, existem casos em que a solução falha. Por exemplo, passando a entrada 6 2 3 5 4 8, a saída correta seria 10 2, passando dois dias na ilha 1, enquanto o algoritmo produz a saída 8 1, passando um dia na ilha 2.

Algoritmo dinâmico: Assuma uma posição do array temporário de soluções construído pela solução [i][j]. Por indução, sabemos que essa posição encontra a solução ótima para i itens e peso j, e todos os itens e pesos anteriores. Agora assuma uma posição [i+1][j], com o item i+1 possuindo valor v e peso w. A solução para essa posição é encontrada pela solução ótima entre escolher ele(posição [i][j-w]) e não escolher ele(posição[i][j]). Como ambas já foram calculadas e, por indução, ambas são ótimas, a solução para [i+1][j] também será ótima.

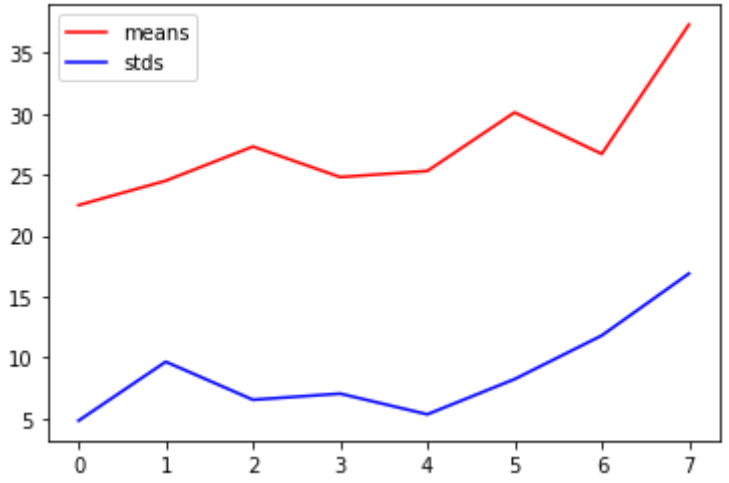
5. **Avaliação Experimental**

5.1) Análise do tempo

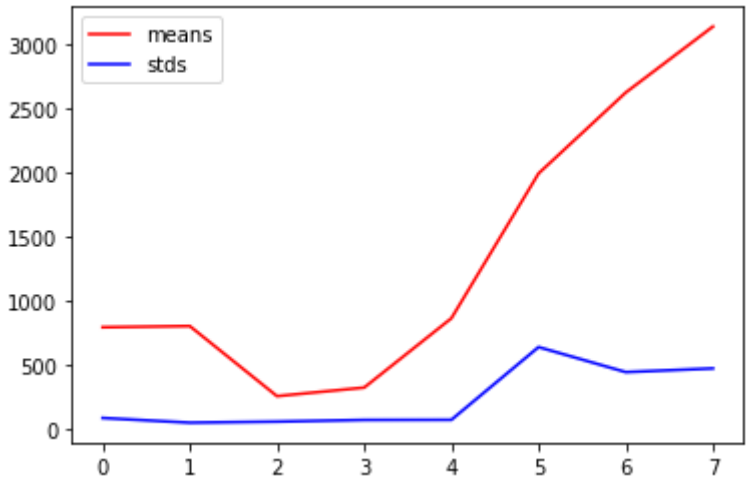
Para a análise de tempo, o programa foi executado, para cada caso de teste, 10 vezes, e os tempos encontrados para cada algoritmo a seguir estão em microssegundos

Algoritmo com repetição		
Quantidade de Ilhas	**Média**	**Desvio Padrão**
5	22.5	4.836
5	24.5	9.664
4	27.3	6.550
3	24.8	7.052
8	25.5	5.355

12	30.1	8.239
16	26.7	11.814
20	37.3	16.892



Algoritmo sem repetição		
Quantidade de Ilhas	**Média**	**Desvio Padrão**
5	791.6	83.116
5	800.6	46.471
4	253.7	59.963
3	320.4	67.843
8	861.6	68.842
12	1992.3	636.703
16	2623.9	441.364
20	3139.0	469.991



Considerando os números usados nos exemplo, em que o custo das ilhas é muito maior que sua pontuação(casa dos milhares x casa das dezenas, em maior parte), a grande diferença não é surpreendente, e ambos seguem padrões próximos ao esperado.

5.2) Saídas de cada abordagem

Na maior parte dos casos, a melhor solução vai ser retornada pelo algoritmo guloso com repetição de ilhas, já que, pegando o custo benefício das ilhas, ele repete ao máximo as melhores ilhas. Contudo, devido à limitação do tempo que pode ser gasto no algoritmo e da não possibilidade de se utilizar frações das ilhas, existem casos em que a abordagem dinâmica sem repetição vai encontrar soluções melhores.

6. Conclusão

O trabalho foi interessante para compreender o funcionamento de diferentes paradigmas de programação, especialmente aplicados em problemas tão semelhantes. É possível perceber a importância de se compreender cada um dos paradigmas pois para diferentes problemas, a solução ótima pode ser encontrada através de diferentes estratégias.

7. Referências

Geeks for Geeks - 0-1 Knapsack Problem | DP-10. Disponível em: <https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/>. Acesso em: 07/10/2019.

Geeks for Geeks -Merge Sort. Disponível em: <https://www.geeksforgeeks.org/merge-sort/>. Acesso em: 07/10/2019.

C plus plus. Disponível em: <http://www.cplusplus.com/reference/>. Acesso em: 05/10/2019