Trabalho Prático - Grupo 02

Integrantes:	Lucas Novais - 18.1.8046; Thiago Figueiredo - 18.1.8017
Disciplina:	CSI-546 Projeto e Análise de Algoritmos
Professor:	Vinícius Dias

1 Introdução

O problema aqui representado visa minimizar os sub-caminhos de um trajeto. Em resumo, dadas n+1 distâncias entre n planetas consecutivos de uma rota (n+1 distâncias pois iniciamos a rota em I e terminamos em F – veja Figura 1) e o número k de planetas a serem conquistados, o objetivo do algoritmo é determinar quais planetas serão conquistados de forma a minimizar as sub-distâncias percorridas entre planetas, o início e o fim do caminho.

A solução deve respeitar a seguinte regra: A rota não irá repetir trechos. Deve-se ir do ponto inicial I até o final F passando uma única vez por cada trecho entre os planetas. Ou seja, a rota é uma linha reta tal que não é possível voltar para um planeta anterior.

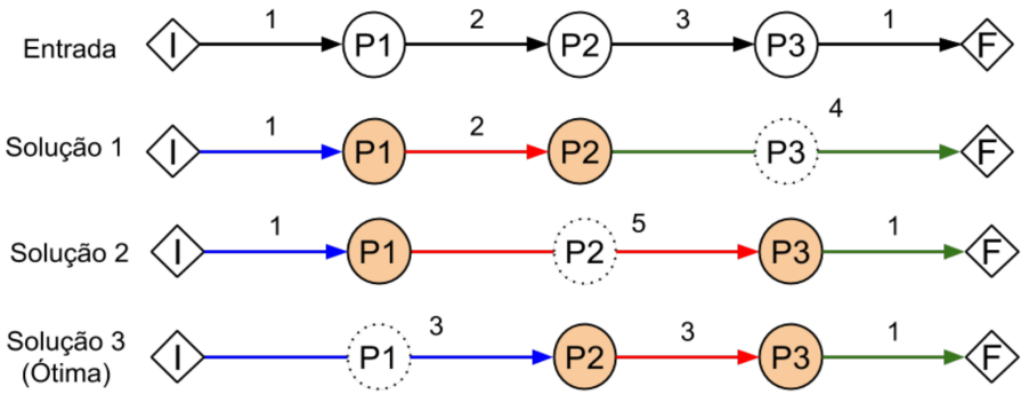

Figura 1: Exemplo de rota [Fonte: Texto base do trabalho prático]

Exemplo: Considerando que a rota entre os planetas é uma linha reta e que o início e fim do caminho não são planetas. Na Figura 1 temos um exemplo de rota com n=3 planetas e deseja-se conquistar apenas k=2 deles. Também temos as 3 soluções possíveis para o problema. Podemos observar que na última solução, a solução ótima, escolhemos os 2 últimos planetas, desta forma as distâncias entre o início do caminho, os planetas e o fim são 3, 3 e 1.

Para solucionar o problema serão propostas três abordagens distintas. A primeira delas será por meio de Força Bruta, isto é, um algoritmo que testará todas as possibilidades de trajetos e retornará o maior sub-caminho da solução ótima. Além desta estratégia, faremos também uma abordagem baseada em Escolha Gulosa, para isto, definiremos uma escolha local visando encontrar a solução ótima. Por fim, utilizaremos o paradigma da Programação Dinâmica, onde empregaremos o conceito de *memoization*[1] para otimizar o tempo de execução. Nos tópicos a seguir cada abordagem será explorada de maneira a demonstrar a solução aplicada, o seu custo e uma avaliação experimental para o problema.

2 Força Bruta

2.1 Solução do problema

Algoritmos de Força Bruta trazem uma abordagem direta para solucionar problemas. Normalmente segue-se o caminho mais fácil sem utilizar nenhuma estratégia específica de maneira a experimentar exaustivamente todas as soluções possíveis. Este tipo de abordagem faz com que sua complexidade tenha custo alto quando comparado a outras estratégias. Porém, devido ao fato de ter um baixo custo de implementação, e também por poder ser aplicado a uma variedade de problemas, a força bruta é de extrema importância. Portanto, para testar todas as soluções é necessário primeiramente gerá-las. Para isso, utilizamos funções de combinação, isto é, no problema apresentado foi necessário gerar todas as combinações de tamanho k dos n planetas. A quantidade de caminhos possíveis pode ser obtido pela fórmula abaixo:

$$C(n,k) = \frac{n!}{k!(n-k)!} \tag{1}$$

Para encontrar o custo da viagem de um planeta para outro foi gerada uma matriz de tamanho (n+2)x(n+2), pois é necessário entrar com o Início e Fim da rota. Então para cada par (i, j) da matriz será atribuído o valor da viajem saindo de i com destino a j. A matriz para o exemplo da Figura 1 pode ser visto na Figura 2.

[1] *Memoization* consiste em uma técnica de otimização utilizada, inicialmente, para aumentar a velocidade com que alguns algoritmos são executados. Seu funcionamento ocorre através do armazenamento do resultado de funções que são chamadas várias vezes. Nesse caso, ao invés de chamar a função novamente, basta acessar o seu resultado na estrutura de dados escolhida sempre que ela ocorrer.

	I	P1	P2	P3	F
I	0	1	3	6	7
P1	0	0	2	5	6
P2	0	0	0	3	4
P3	0	0	0	0	1
F	0	0	0	0	0

Figura 2: Matriz de caminhos

Por fim, para cada um dos caminhos gerados é testado se o valor do seu maior sub-caminho é o menor dentre todos. O algoritmo retorna então o menor máximo. O pseudocódigo pode ser visto no Algoritmo 1.

Listing 1: Implementação Força Bruta

```
forcaBruta():
    minMax <- inf
    comb <- conjunto de todas combinacoes
    for c in comb:
        maiorSub <- maior distancia entre planetas de c
        if maiorSub < minMax:
            minMax = maiorSub
    return minMax
```

2.2 Análise de complexidade

Como apresentado na seção anterior, os algoritmos de força bruta são conhecidos por sua característica de teste exaustivo, ou seja, por testarem todas as soluções possíveis. Porém, antes de gerar as soluções será criada uma matriz contendo o custo da viagem de um local i para um destino j. Para preencher esta matriz será necessário então um laço encaixado percorrendo os n planetas, além do Início e Fim. O custo pode ser representado pela Equação 2, o que significa uma complexidade $\Theta(n^2)$.

$$\sum_{i=1}^{n+2}\sum_{j=1}^{n+2} 1 = (n+2)^2 \tag{2}$$

Com a matriz devidamente preenchida, o algoritmo chamará a função para gerar as combinações possíveis. Para cada combinação gerada o algoritmo irá acessar a matriz k vezes, para verificar qual o maior sub-caminho. Portanto o custo para encontrar o maior sub-caminho de um dado conjunto de tamanho k é dado por $\Theta(k)$. Porém, como já dito, todas as combinações

serão geradas, e segundo a Equação 1 podemos inferir um custo de $\Theta(n!)$. Ou seja, o custo será $\Theta(n^2 + k + n!)$ sendo dominado, portanto, por $\Theta(n!)$.

2.3 Avaliação experimental

A análise de complexidade de tempo do algoritmo de força bruta apresentada anteriormente permite concluir que o custo de tempo está fortemente ligado à quantidade de combinações possíveis. A Tabela 1 apresenta o número de combinações e tempo de execução para diferentes instâncias do problema, a Figura 3 permite visualizar o crescimento do tempo em relação à quantidade de combinações para cada entrada n, k distinta. Note que para uma entrada cujo a quantidade de combinações é na casa dos bilhões, o tempo para executar o código foi de quase 6 minutos.

n	**k**	**Comb.**	**tempo (s)**
50	2	1,23E+03	2,00E-04
50	3	1,96E+04	1,00E-03
50	4	2,30E+05	1,34E-02
50	5	2,12E+06	1,33E-01
50	6	1,59E+07	1,14E+00
100	5	7,53E+07	4,41E+00
150	5	5,92E+08	3,40E+01
500	4	2,57E+09	1,27E+02
45	10	3,19E+09	3,43E+02

Tabela 1: Avaliação experimental de Força Bruta

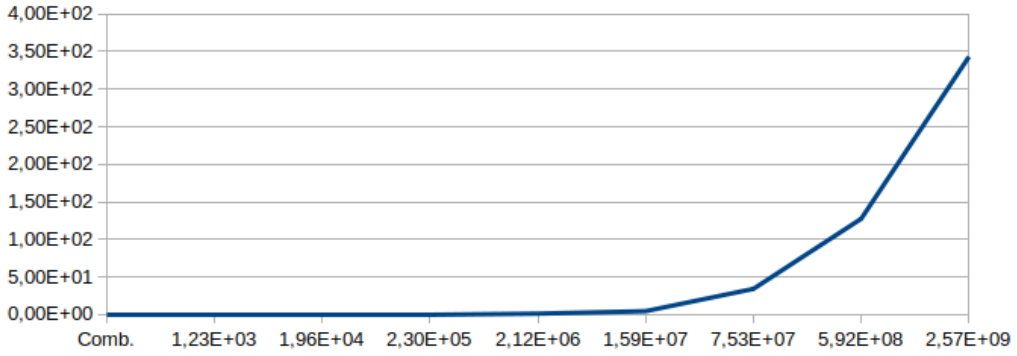

Figura 3: Custo de Tempo x Combinações

Em relação ao espaço, foi utilizada uma matriz quadrada conforme dito anteriormente, neste caso, tem-se um custo delimitado por $\Theta(n^2)$. Apesar de

o pseudocódigo apresentado anteriormente (Algoritmo 1) utilizar um arranjo para armazenar todas as combinações, esta estratégia foi utilizada apenas para apresentar a ideia. No código implementado, cada combinação é testada e armazenada sempre no mesmo arranjo de tamanho k. Nesse caso, o custo para o armazenamento é de $\Theta(k)$.

3 Algoritmo Guloso

3.1 Solução do problema

A próxima abordagem aplicada foi utilizando a estratégia dos Algoritmos Gulosos. Em geral este tipo de algoritmo é baseados em escolhas imediatas e locais, que aparentemente resultarão em uma solução ótima. Eles são interessantes para problemas em que o algoritmo tem um conjunto de escolhas, opções, e a cada momento uma dessas opções levará a solução total ou parcial do problema. Além disso, entende-se por guloso as escolhas incrementais, imediatas e locais, melhores para o momento, que buscam o que é mais correto para a solução global.

A utilização das escolhas gulosas devem ser factíveis para o problema, ou seja, não podem violar restrições caso hajam. Além disso, elas devem ser localmente ótimas e não podem ser desfeitas. Apesar de uma implementação de dificuldade razoável, existe grande dificuldade na prova da corretude de que a estratégia levará a solução ótima global, sendo necessário provar que: (1) Sempre existe uma solução ótima que contenha a escolha gulosa, ou seja, a escolha gulosa não impede que se alcance a solução ótima; (2) Combinar soluções ótimas dos subproblemas com a opção gulosa ajuda a compor a solução ótima total.

Para elaborar o algoritmo, foi considerada a escolha gulosa de pular planetas sempre que o custo da viagem da posição atual até o próximo planeta não exceder o valor indicado pela Equação 3.

$$C_{MAX} = \frac{S_{TOTAL}}{k+1} * 1.4 \tag{3}$$

onde

$$S_{TOTAL} = d_1 + d_2 + ... + d_{n+1} \tag{4}$$

Sendo C_{MAX} a média dos custos das viagens multiplicado por um fator de correção de quarenta porcento, S_{TOTAL} a soma total das distâncias da rota e $[d_1, d_2, ..., d_{n+1}]$ as n distâncias da rota. O fator de correção foi introduzido na equação para melhorar as aproximações geradas.

Porém, apesar de a estratégia resolver o exemplo adotado anteriormente, em que:

$$C_{MAX} = \frac{(1+2+3+1)}{2+1}.1,4 = \frac{7}{3}.1,4 = 3,26 \tag{5}$$

É fácil encontrar um exemplo em que a estratégia não retorna o valor ótimo. Suponha uma entrada com 4 planetas em que é necessário conquistar 2 deles, tal que o arranjo de distâncias é dado por $[4,1,1,1,4]$. Nesse caso, temos:

$$C_{MAX} = \frac{(4+1+1+1+4)}{2+1}.1,4 = \frac{11}{3}.1,4 = 5,13 \tag{6}$$

Porém, a solução ótima tem resultado igual a 4, onde conquista-se o primeiro e o último planeta, ou seja, a estratégia gulosa não leva ao ótimo global. Neste exemplo, tem-se uma razão de precisão igual à 1,25. Repetimos o teste para mais de 300 instâncias geradas de maneira aleatória. O gráfico da Figura 4 mostra o resultado obtido na comparação entre a solução ótima, e aquela encontrada através da estratégia gulosa. Vale ressaltar ainda que a correlação entre os resultados analisados foi superior à 99%, além disso, o valor médio calculado para a razão de precisão foi de 1.17, isto é, a solução aproximada está em média 17% acima do valor ótimo.

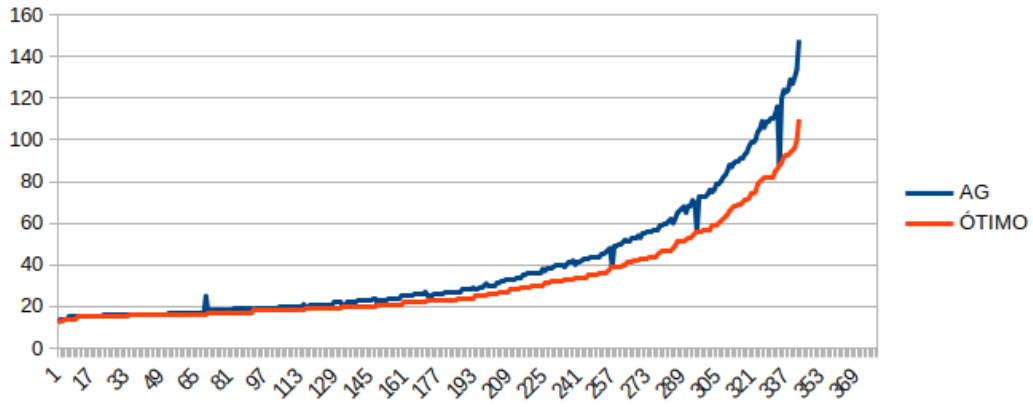

Figura 4: Resultado x Instância

3.2 Análise de complexidade

Em relação à complexidade de tempo para a estratégia gulosa temos os seguintes custos:

- Soma total do percurso: Devemos percorrer todo o arranjo contendo o valor dos caminhos entre planetas. Nesse sentido o custo é $\Theta(n)$;
- Percorrer $k + 1$ caminhos: Essa etapa é realizada através de um laço encaixado, executa-se k vezes um laço interno de tamanho máximo igual à n. Portanto no pior caso o custo é dado por $\Theta(k.n)$

Portanto, podemos afirmar que o custo que delimita o algoritmo é dado por $\Theta(k.n)$.

3.3 Avaliação experimental

A Tabela 2 mostra o tempo em segundos crescendo de acordo com a entrada n vezes k, o que é pertinente ao custo encontrado na seção anterior. O gráfico referente à tabela pode ser visto na Figura 5.

n	**k**	**n*k**	**Tempo (s)**
50	6	300	0,000090
45	10	450	0,000095
100	5	500	0,000096
150	5	750	0,000102
250	10	2500	0,000117
300	50	15000	0,000121
400	100	40000	0,000130
450	200	90000	0,000145
500	250	125000	0,000167

Tabela 2: Avaliação experimental para Algoritmo Guloso

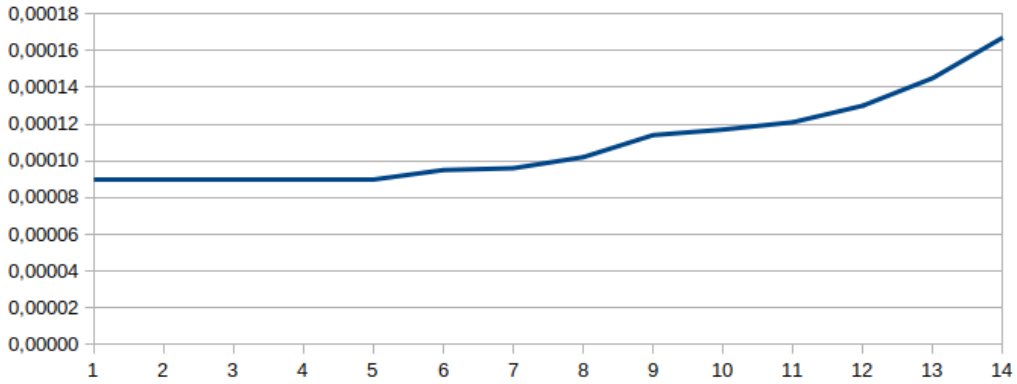

Figura 5: Custo de Tempo x n*k

4 Programação Dinâmica

4.1 Solução do problema

Por fim, aplicamos também o paradigma da Programação Dinâmica. Esta estratégia utiliza estruturas de dados específicas para resolver problemas de sobreposição, dessa forma guarda-se os resultados parciais melhorando a complexidade de tempo (*memoization*). Vale ressaltar ainda que, na programação dinâmica resolve-se os subproblemas de forma deliberada, ou seja, seguindo uma ordem específica de forma 'esperta'. Além disso, nesse tipo de abordagem os subproblemas não são disjuntos, isto é, existem subproblemas contidos em outros subproblemas, o que possibilita a utilização do *memoization* para evitar operações repetidas.

A partir disso, criou-se uma equação de recorrência C que possui três parâmetros: *atual* que indica o planeta em que se está posicionado; n que indica o planeta a ser analisado, isto é, conquistá-lo ou não; e k que representa a quantidade de planetas a serem conquistados. Além disso utilizamos a matriz de distâncias apresentada na Figura 2, em que *size* é equivalente ao tamanho da matriz. Sendo assim, o custo C pode ser dado por:

Listing 2: Implementação Força Bruta

```
C(atual, n, k):
    se k = 0
        return matriz[atual][size]
    se k = numero de planetas a frente
        return max(matriz[atual][n], C(n, n+1, k-1)
    se chegou ao utltimo planeta
        return max(matriz[atual][n], matriz[n][size])
    caso contrario
        return min(max(matriz[atual][n],C(n, n+1, k-1),
                   max(matriz[atual][n+1], C(atual, n+1, k))
```

A partir do Algoritmo 2, é possível notar a sobreposição de problemas. Perceba que $C(n, n+1, k-1)$ é chamado mais de uma vez, além disso, $matriz[atual][n]$ também ocorre mais de uma vez. Portanto, é interessante utilizar a estratégia de *memoization*. Para isso, é necessário adotar uma estrutura de dados para a tarefa de memorização de resultados que se repetem. Como o custo C possui três variáveis, optamos por utilizar uma matriz de três dimensões.

Para preencher a matriz, iniciamos definindo como zero todos as posições para $n = 0$. Em seguida, definimos as posições em que $k = 0$ com a distância do inicio até o planeta atual. Por fim, utiliza-se a estratégia definida na

equação de recorrência. Nesse sentido, utilizamos laços encaixados em que o laço mais externo parte do primeiro planeta até o último, para cada planeta será executado conquistas de 1 a k, e para cada par (planeta, conquista), iteraremos sobre a posição atual, de 1 até *atual*. Uma representação pode ser vista no Algoritmo 3.

Listing 3: Implementação Programação Dinâmica com *Memoization*

```
PD_memoization:
    //caso base em que n = 0;
    for i <- 1 to atual:
        for j <- 1 to k:
            mat[i][0][j] <- 0;

    //caso base em que k = 0;
    for i <- 1 to atual:
        for j <- 1 to n:
            mat[i][j][0] <- matriz[0][i];

    //estrategia baseada na recorrencia
    for i <- 1 to n:
        for j <- 1 to k:
            for l <- 1 to atual:
                mat[i][j][l] <- c(atual, n, k)
```

4.2 Análise de complexidade

A estrutura de dados adotada para realizar a memorização dos subproblemas foi uma matriz de três dimensões, pois é necessário saber o maior custo até chegar em um dado planeta baseado na quantidade de coquistas já realizada. Sendo assim, a matriz possui *n+1* linhas, n colunas, e profundidade k. Portanto, para preencher esta matriz é necessário três laços encaixados, o primeiro de tamanho n, o segundo de tamanho k e um terceiro de tamanho *n+1*. Neste sentido torna-se direta a conclusão de que o custo envolvido é de $\Theta(n^2.k)$. Além disso, é importante notar que por utilizar a estratégia de *memoization* haverá também um custo de $\Theta(n^2.k)$ para a quantidade de espaço demandado pelo código.

4.3 Avaliação experimental

A avaliação experimental confirma que o custo de tempo está atrelado ao valor de $n^2.k$ conforme pode ser observado na Tabela 3. Além disso, o custo de tempo é ilustrado na Figura 6.

n	k	n^2 x k	tempo (s)
50	3	7,50E+03	4,00E-04
50	4	1,00E+04	5,00E-04
50	5	1,25E+04	6,00E-04
45	10	2,03E+04	7,39E-04
100	5	5,00E+04	2,02E-03
150	5	1,13E+05	4,19E-03
250	10	6,25E+05	2,02E-02
400	100	1,60E+07	4,05E-01
450	200	4,05E+07	9,85E-01
500	250	6,25E+07	1,65E+00

Tabela 3: Avaliação experimental para Programação Dinâmica

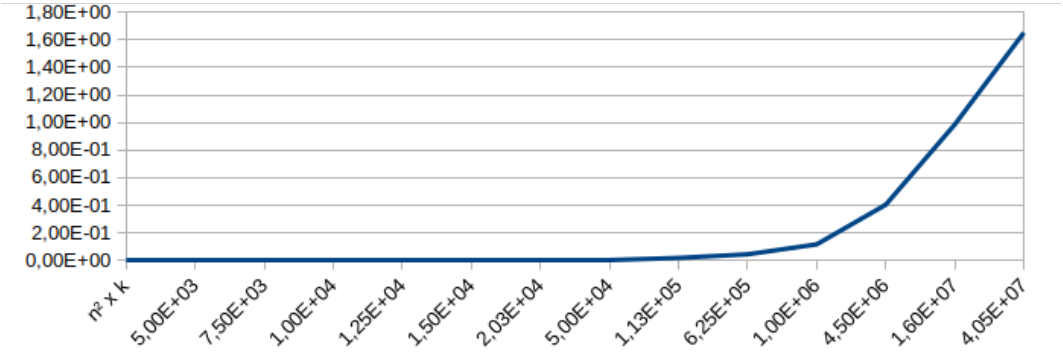

Figura 6: Custo de Tempo x Combinações

5 Contribuição dos Integrantes

O integrante Thiago ficou responsável pela implementação do algoritmo de Força Bruta, algoritmo Guloso e pela avaliação experimental. O integrante Lucas realizou a criação do *makefile*, a implementação do algoritmo de Programação Dinâmica, e realizou a padronização e refatoração do código. Ambos contribuíram igualmente para a documentação e análise de complexidade.