Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Algorítmo Guloso, Programação Dinâmica – Viagem ao Caribe

Vinícius Julião Ramos
viniciusjuliao@dcc.ufmg.br

28 de Outubro de 2019

Resumo

O trabalho aqui apresentado consiste na elaboração de um modelo computacional que consiga computar a rota da viagem de um grupo de pessoas, levando em consideração o custo de cada ponto de parada e uma nota classificatória, dada para cada ponto. Para isso solicitou-se que duas soluções fossem apresentadas, uma com a aplicação de um algoritmos guloso e outra através de um problema de decisão de acordo com todas as opções possíveis.

1 Introdução

Antes de iniciar a viagem para as ilhas do Caribe, um grupo de amigas deseja estudar a melhor opção levando em conta dois parâmetros fundamentais, preço e classificação. Tal estudo tem como objetivo maximizar o aproveitamento da viagem conduzindo o grupo para a maior quantidade de ilhas possível desde que o custo seja minimizado e o somatório total de classificação maximizado. O custo respectivo de cada ilha refere-se ao gasto que se tem em passar um dia utilizando as dependências de cada local, nesse sentido, o problema possui duas versões diferentes para o tipo de viagem realizado pelo grupo.

No primeiro tipo de viagem, deve-se considerar a possibilidade de ficar por mais de um dia na mesma ilha, logo a cada dia o custo total aumenta e também a pontuação total é somada ao contador. Já na outra solução, é possível passar apenas um dia em cada ilha, sendo assim uma ilha não pode ser visitada em dois dias de viagem. Por fim, para cada versão do problema, deve-se tomar uma diferente abordagem de solução, sendo que, para o modo que permite repetição de ilhas, utiliza-se um algoritmo guloso, no qual através do calculo de uma terceira variável, que sintetiza a razão entre o preço de visitação e a classificação, foi possível ordenar as ilhas em função desse novo parâmetro. Contanto, para a solução que não permite repetição de ilhas, pediu-se a implementação de um algoritmo de programação dinâmica.

Após o desenvolvimento dos algoritmos para solucionar o problema, na implementação prática usou-se da linguagem de programação **C++**[**?**], que permite implementar o conceito de orientação a objetos, além de possuir uma biblioteca padrão robusta que contém diversas estruturas de dados já implementadas (fatores que foram essenciais para uma boa implementação dos códigos).

2 Modelagem

A partir da descrição dada para o problema, necessitou-se da criação de uma estrutura capaz de computar e armazenar as informações sobre cada ilha. Tal estrutura é composta por três atributos: Preço de visitação, pontuação da ilha (valor classificatório) e custo *vs.* benefício (que é a razão entre preço de visitação e pontuação). Essa modelagem será útil para acessar com maior facilidade cada atributo das respectivas ilhas.

Diante de tal modelo, agora pode-se criar o cenário ideal para armazenar e computar as respostas baseadas em programação dinâmica e em um algoritmo guloso. Portanto, para melhor esclarecer as decisões tomadas na implementação de cada uma dessas soluções, considere a existência de uma estrutura *Ilha*, que possui os atributos *preço*, *pontuação*, *custo_benefício* em que as operações de encaminhamento sobre inclusão ou exclusão de cada ilha na solução final, serão tomadas a partir da leitura de tais atributos.

2.1 Algoritmo Guloso

Para esse ponto, consideramos que a instância de cada execução do algoritmo é um vetor **V**[ ] composto por n elementos do tipo *Ilha*. A entrada é responsável por ler sequencialmente o *preço* e a *pontuação* de cada ilha, sendo que, à partir da leitura desses dois dados calcula-se o valor de *custo_benefício*(*pontuação*÷*preço*). Esse valor será usado para ordenar as *Ilhas* de acordo com a preferência em visitá-las, para que assim o algoritmo guloso tome a decisão correta em adicionar ou não uma ilha ao plano de viagem.

Portanto, temos que esse algoritmo trata a otimização do custo *vs.* benefício da viagem de forma gulosa, ou seja, a cada etapa, opta-se por tomar a melhor decisão segundo o valor de *Ilha.custo_beneficio*. Para demonstrar o funcionamento da solução observe o seguinte pseudo código:

Algorithm 1 Greedy

```
1:  function MAXIMIZE_COST_VS_BENEFIT(IslandV[ ], Integer max_cost)        ▷ O(n log n)
2:      decreasing_sort(Islands)                                           ▷ O(n log n)
3:      Total_points ← 0
4:      Total_days ← 0
5:      for all {Locale_i ∈ V[ ] \ 0 < i ≤ length(V)} do                   ▷ O(n)
6:          quocient ← max_cost ÷ Locale_i.price                           ▷ Integer division
7:          if quocient > 0 then          ▷ Agregate quantity of days and price to result
8:              remain ← max_cost%Locale_i.price
9:              max_cost ← remain
10:             Total_points ← Total_points + (quocient × Locale_i.price)
11:             Total_days ← Total_days + (quocient)
12:         end if
13:     end for
          return Total_points, Total_days
14: end function
```

A primeira etapa do algoritmo consiste em ordenar o arranjo de ilhas *Islands* de forma decrescente de acordo com o parâmetro *custo_beneficio*, depois disso, a partir desse rearranjamento percorre-se o vetor desde o início até o final, comparando se é possível adicionar a ilha ou não de acordo com o valor ainda disponível para a viagem. Essa fase consiste na obtenção do máximo local, que é a chave do paradigma de algoritmos gulosos. Por fim, ao adicionar uma ilha na solução final, deve-se subtrair do custo máximo o custo de visitação da ilha multiplicado pela quantidade de dias que é possível permanecer na ilha.

O parâmetro *custo_beneficio* é conhecido como um otimizador de gastos, uma vez que ao tentar adquirir um bem olha-se o beneficio causado pela compra mediante o preço do produto. Como neste problema tem-se uma classificação que pontua o preço e o benéfico de visitação das ilhas, utilizou-se a razão entre tais classificadores para obter o *custo_beneficio* a fim de maximizar a pontuação (benefício) total. Uma vez que há um valor máximo que o grupo de viajantes está disposto a gastar, existe por consequência a necessidade de analisar a possibilidade em visitar determinado local, isso faz com que tome-se o cuidado por minimizar o gasto, logo a ordenação do vetor de ilhas através do *custo_beneficio*, em ordem decrescente, garante que

para a i-ésima ilha obtenha-se a maior pontuação possível, desde que o valor máximo disponível para pagar (max_cost) seja menor que $Island_{i-1}$ e maior que $Island_i$.

Permite-se concluir, portanto, que ao percorrer a lista ordenada decrescente garante-se a preferência pela ilhas que maximizam a pontuação e reduzem o custo. Nesse encaminhamento, adiciona-se à solução final a quantidade de dias em que o grupo poderá permanece no mesmo local, o que é referente ao quociente da divisão inteira entre max_cost e $Island_i.price$. Esse passo configura a obtenção do máximo local (fator determinante para a sequencialidade dos algoritmos gulosos), da mesma forma como o resto da divisão dos operandos citados dará respaldo para a obtenção do $i+1$-ésimo máximo local até que se percorra todo o vetor.

2.1.1 Corretude

Dada a instância *IN* do problema, composta por um conjunto de n elementos do tipo *Ilha* com os respectivos preços e pontuações, além de uma constante m que representa o custo máximo que o grupo de amigas está disposto a pagar pela viagem. Demonstra-se abaixo, que nem sempre esse algoritmo produz a solução ótima, ou seja, nem sempre o benefício é maximizado. Um exemplo de uma entrada que não gera a solução ótima pode ser dada como, duas ilhas (considere o calculo de *custo_beneficio* assim que as entradas são lidas):

Ilha$_1$: *preço:* 3, *pontuação*: 5, *custo_beneficio*: 1,666
Ilha$_2$: *preço:* 4, *pontuação*: 8, *custo_beneficio*: 2

Ao ordenar as ilhas tem-se que a primeira ilha checada será **Ilha**$_2$ e em seguida **Ilha**$_1$. Dado um custo máximo da viagem como o valor 6 analisaremos qual a saída produzida pela solução apresentada. Então faz-se a leitura do laço central do algoritmo após a ordenação:

1. $Total\_dias \leftarrow 0, Total\_pontos \leftarrow 0$
2. Primeira iteração do laço – Leitura de $Ilha_2$.
3. $quociente \leftarrow max\_cost \div Ilha_2.preço$ $[= (6 \div 4) = 1]$.
4. $resto \leftarrow max\_cost \% Ilha_2.preço [= 2]$.
5. $max\_cost \leftarrow resto$ $[= 2]$
6. $Total\_dias \leftarrow Total\_dias + quocient$ $[= 0 + 1]$
7. $Total\_pontos \leftarrow Total\_pontos + (quocient \times Ilha_2.pontuação)$ $[= 0 + (1 \times 8)]$.
8. Segunda iteração do laço – Leitura de $Ilha_1$.
9. $quociente \leftarrow max\_cost \div Ilha_1.preço$ $[= (2 \div 3) = 0]$.
10. $resto \leftarrow max\_cost \% Ilha_1.preço$ $[= 2]$.
11. Condição do *If* não é satisfeita
12. Fim do laço
13. *Total_dias*: 1, *Total_pontos*: 8

Essa execução demonstra a saída dada para o problema, no caso da instância informada. Mas há uma solução mais otimizada que representa permanecer dois dias em **Ilha**$_1$. Dessa forma o gasto total seria de 6, que é o mesmo que o custo máximo do grupo. Além disso, permanecer por dois dias em $Ilha_1$ garante um total de 10 pontos. Assim, demonstra-se que a solução gulosa não satisfaz em todos os casos a otimização.

2.2 Programação dinâmica

O caso em que não são permitidas repetições de ilhas, implementado através de programação dinâmica, possui semelhanças com outros problemas já conhecidos e estudados sobre o mesmo assunto. Portanto, a solução implementada neste ponto é dada través de pequenas alterações em um dos algoritmos já conhecidos em aula, denominado "*Problema da mochila*", em que também será utilizada estrutura *Ilha* definida no início desta seção para simular o que seriam as cargas colocadas na mochila.

A fim de entender a adaptação do problema da mochila para o problema da viagem do grupo de amigas, imagine a conversão dos nomes dos atributos do problema da mochila da seguinte maneira: No problema da mochila há o atributo central m que representa o carga máxima da mochila, e há n objetos que, individualmente, possuem um peso e um valor, o objetivo é carregar a mochila com cada um desses objetos fazendo com que o atributo valor seja otimizado sem que haja extrapolação da carga máxima m no somatório dos pesos de cada item da carregado. Assim, a conversão para problema da viagem é destacada na tabela:

Tabela 1: Adaptação do problema da mochila.

Problema da mochila	**Problema da viagem**
Carga máxima m	Custo máximo m
Objeto	*Ilha*
Objeto.valor	*Ilha.pontuação*
Objeto.peso	*Ilha.preço*

Nesse novo cenário deseja-se otimizar a pontuação total das ilhas visitadas desde que o somatório total dos gastos não ultrapasse o valor do custo máximo m. Para isso, fez-se necessário a criação de uma matriz $M_{n\times m}$, sendo que, utilizando dois laços, navega-se no laço interno entre as colunas de M e o laço externo destaca as linhas da mesma matriz. Dessa forma, o número de colunas de M refere-se ao valor total do custo máximo disponibilizado pelas viajantes, e a quantidade de linhas da matriz é a mesma que a quantidade de ilhas pretendidas pelo grupo. Antes de demonstrar como é o funcionamento do algoritmo, utilizaremos uma estrutura de dados auxiliar, que será uma tupla $Tuple$ na qual o primeiro elemento ($first$) representará a quantidade de pontos acumulados e o segundo elemento da tupla ($second$) representará a quantidade de dias que a viagem durará. O pseudo código demonstrará como navegar na matriz montada para a programação dinâmica:

A explicação sobre o funcionamento do problema envolve a garantia de que para uma determinada célula e_{ij} de M tem-se o estado ótimo para todas as outra células de e_{xy}, para $x < i$ e $y < j$. Ao averiguar a i-ésima linha da matriz, é dado o estado em que deseja-se incluir a i-ésima Ilha na solução e portanto, adicionando esse novo local deseja-se encontrar a solução ótima que considera a solução ótima antes da adição para produzir um novo melhor cenário. Dessa maneira, ao percorrer as colunas de M deseja-se obter o cenário em que, dada a j-ésima linha, deseja-se obter a maior pontuação visto que o valor total disponível considerado em tal coluna é igual a $j \leq max\_cost$.

Uma vez que os valores são salvos na matriz M, não é necessário recalcular todos valores para obter determinada solução ótima para o caso de e_{ij}. Basta acessar tal célula da matriz que terá-se a melhor solução que considera a inclusão das i primeiras ilhas com um custo máximo igual a j e ao percorrer toda a matriz, chegando até a célula e_{nm} obtém-se a melhor solução com o custo máximo igual a m e que considera a inclusão, ou não, das n primeiras linhas. A equação de Bellman demonstra o que fora explicado.

$$OPT(i,j) = \begin{cases} 0, & \text{if } i = 0 \\ OPT(i-1,j), & \text{if } Ilha_i.price > j \\ max(OPT(i-1,j), Ilha_i.points + OPT(i-1, j - Ilha_i.price)) & \text{para os demais} \end{cases}$$

2.2.1 Corretude

Para provar que a solução de programação dinâmica retorna a saída esperada, o método da indução será utilizado. Através da equação de Bellman pode-se observar os três casos que devem ser considerados no desenvolvimento da prova.

Como **caso base**, tem-se que $OPT(0, j) = 0$.

Algorithm 2 Dynamic Programming

```
1: function TRAVEL_PROBLEM_BY_KNAPSACK_ADAPTATION(Island[ ], Integer max_cost)        ▷ Θ(nm)
2:     n ← length_of(V[ ])
3:     m ← max_cost
4:     M ← Tuple[n][m]
5:     for all i \ 0 ≤ i ≤ m do                                                       ▷ Θ(m)
6:         M[0][i] ← (0, 0)
7:     end for
8:     for all i \ 1 ≤ i ≤ n do                                                       ▷ Θ(n)
9:         for all j \ 0 ≤ j ≤ m do                                                   ▷ Θ(m)
10:            if Island_i.price > m then
11:                M[i][m] ← M[i − 1][m]
12:            else
13:                if M[i − 1][m] ≤ Island_i.points + M[i − 1][m − Island_i.price].first then
14:                    M[i][j].first ← Island_i.points + M[i − 1][m − Island_i.price].first
15:                    M[i][j].second ← M[i − 1][m − Island_i.price].second + 1
16:                else
17:                    M[i][j] ← M[i − 1][m]
18:                end if
19:            end if
20:        end for
21:    end for                                                                        ▷ Θ(nm)
       return M[n][m]
22: end function
```

Observe no Algoritmo da solução de programação dinâmica que o primeiro laço da função trata-se de preencher toda a primeira linha da matriz M com o valor). Portanto, ao acessar quaisquer valores da primeira linha não é possível obter um melhor caso que há um acumulo da pontuação das ilhas visitadas que seja maior que 0. Em outras palavras, $OPT(0, j)$ procura a melhor solução da quantidade de pontos somados em que não se considera nenhuma ilha, para um valor de custo máximo igual a j. Logo, como não há ilhas para se considerar, não se pode obter um valor máximo de pontos que seja maior que 0.

Observando o segundo caso, em que $OPT(i, j) = OPT(i - 1, j)$, if $Ilha_i.price > j$. Por hipótese consideremos que $OPT(i - 1, j)$ retorna a melhor solução para as primeiras $i - 1$ ilhas em que se pretende despender-se da quantidade j de dinheiro.

Nesse caso, observa-se que se o custo em visitar a i-ésima ilha for maior do que o valor atual que deseja-se gastar (j), então não é possível incluir $Ilha_i$ na solução final. Logo o melhor cenário em que deseja-se incluir $Ilha_i$ para um custo máximo igual a j, é não incluir a visitação de tal local no plano de viagem e considerar a melhor solução em que pretende-se gastar a quantia j para as outras $i - 1$ ilhas.

A ultima parte da equação de Bellman, é possível de se abordar desde que considere-se que $OPT(i-1, j)$ retorne o valor correto. Assim, pela recorrência $OPT(i-1, j-Ilha_i.price)$ também retornará o valor correto, uma vez que o atributo *price* deve ser maior ou igual a zero, então ao obter $OPT(i-1, j)$, é possível dizer que $OPT(i - 1, j - Ilha_i.price)$ foi reconhecido anteriormente, ou ao mesmo tempo (no caso em que $Ilha_i.price$ é igual a 0).

Ao lidar com a situação em que $Ilha_i.price < j$, pode-se dizer que é possível adicionar a i-ésima ilha no plano de viagem, entretanto, como deseja-se maximizar a pontuação da viagem, deve-se considerar o melhor cenário em que $Ilha_i$ não foi adicionada. Portanto, surge a necessidade de calcular o valor de $s_1 := Ilha_i.points + OPT(i - 1, j - Ilha_i.price)$ que será retratado como a quantidade de pontos que é possível acumular ao adicionar $Ilha_i$ à solução final.

O valor de s_1 é dado através da soma entre a quantidade de pontos de $Ilha_i$ e a maior quantidade dede pontos em que o custo máximo é reduzido pela diferença entre o custo máximo anterior e o custo de visitação da i-ésima ilha. O que representa a diminuição do valor que o grupo de garotas pretende

gastar com a viagem, ao adicionar tal ilha no plano de viagem. Já a necessidade de visitar a linha $i-1$ é reconhecida pelo motivo de não haver repetições: ao buscar por $OPT(i-1, j-Ilha_i.price)$ destaca-se a obtenção da melhor solução para o caso da não inclusão de $Ilha_i$ em que já gastou-se o valor necessário para visitar esse lugar.
Por fim, o valor de $OPT(i, j)$ deve maximizar o valor da pontuação total, portanto, ao receber o valor de s_1, deve-se compará-lo com a opção em não incluir $Ilha_i$, pois é possível que a não inclusão dessa ilha acarrete numa pontuação maior do que a obtida em s_1. Assim, a consideração por obter a otimização da solução tem seu ponto chave na parte em que deve-se considerar o máximo entre a inclusão e a não inclusão da i-ésima ilha no plano de viajem, mesmo quando é possível adicioná-la.

Ao procurar pelo valor de $OPT(n, m)$ considera-se que a solução envolveu o calculo de todos os casos menores possíveis, iniciando com $i = 0, j = 0$. E assim optou-se por obter sempre o menor caso que maximizasse a soma da quantidade de pontos até que se chegasse no valor de $i = n, j = m$ que resultou na resposta final. □

3 Análise de Complexidade

A análise do comportamento da solução seguirá um cenário teórico que servirá de base comparativa para identificar o custo de cada algoritmo utilizado. Portanto, para melhor esclarecer as soluções aplicadas, é necessário explicar as estruturas que fornecem o cenário ideal para o uso dos algoritmos, logo inicia-se a descrição explicando o custo de memória do problema, para que se entenda como há elevação do custo de memória em determinada chamada de função

3.1 Custo de Espaço

Por critério de legibilidade e qualidade de construção do código, implementou-se uma estrutura do tipo **Ilha** que armazenará os dados recebidos da entrada do programa, além de uma atributo derivado das informações obtidas a partir da entrada. As informações que tal estrutura possui são:

1. Preço: atributo do tipo **Inteiro**.
2. Pontuação: atributo do tipo **Inteiro**.
3. Custo_Benefício: atributo do tipo **ponto flutuante de precisão dupla**.

A solução do problema, por ambos algoritmos usarão um arranjo de elementos do tipo **Ilha** para solucionarem o problema. Desde já, é preciso destacar que independente de qual dos algoritmos seja aplicado, é sempre necessário manter um vetor n posições do tipo **Ilha**. Então ambas as soluções iniciam com um custo de memória igual a n, antes mesmo de iniciarem a execução da resolução do problema.

3.1.1 Algoritmo Guloso

Como já possui-se as ilhas armazenadas em um vetor, a composição do algoritmo guloso faz uso de poucas varáveis auxiliares para executarem seu trabalho. O trabalho de ordenação é executado através da *std::sort()*[1] da biblioteca padrão da linguagem C++ e este algoritmo não utiliza uma estrutura auxiliar para ordenar o arranjo. Portanto, no trabalho de ordenação há apenas a utilização de poucas variáveis que não interferirão no comportamento assintótico do custo de memória, uma vez que o custo de memória para a ordenação pode ser descrito por uma constante C. Já na execução da função **TravelIslands::get_better_by_greedy()** há a instanciação de três variáveis do tipo **Inteiro**.

Sendo assim, o somatório da função de custo pode ser detalhado como: $f(n) = 3+C+n$, sendo 3 inteiros instanciados na função que executa o algoritmo guloso, uma constante C para a função de ordenação e n ilhas armazenadas no arranjo da estrutura auxiliar. Esse custo não pode abaixar em função de nenhum dos parâmetros, logo a função $f(n)$ pode descrever tanto um limite superior quanto o limite infereior do custo de espaço para esse algoritmo. Dessa forma temo que o custo de memória é: guloso é de:

$$f(n) = n + 3 + C = \Theta(n)$$

[1] http://www.cplusplus.com/reference/algorithm/sort

3.1.2 Algoritmo de Programação Dinâmica

Além da utilização do arranjo de elementos do tipo *Ilha*, para a essa solução necessitou-se a implementação de uma matriz de elementos do tipo **std::pair**[2], uma vez que necessita-se armazenar o somatório de pontuações e também o somatório da quantidade de dias total que o grupo de amigas fará.

Tal matriz terá terá dimensões $(n+1) \times (m+1)$, em que n representa a quantidade de ilhas recebidas na entrada e m é o valor recebido na entrada em que se especifica o custo máximo que o grupo de amigas pode gastar na viagem. Os elementos do tipo **std::pair** armazenam dois valores do tipo **Inteiro**. Portanto, para a exeução do algoritmo temos os seguintes gastos:

1. $(n+1) \times (m+1)$: quantidade de elementos do tipo **std::pair** na matriz auxiliar.
2. n: quantidade de elementos do tipo **Ilha** no arranjo de recebimento da entrada de dados.
3. 3: quantidade de variáveis criadas no meio do código para salvarem os estados do programa.

Uma vez que não é possível dividir a execução em casos diferentes de utilização da memória, tem-se que o gasto será o mesmo para qualquer instância. Logo o somatório dos três itens acima será definido como a função de complexidade dessa parte do programa, sendo assim é possível afirmar que:

$$f(n,m) = (n+1)(m+1) + n + 3 = \Theta(nm)$$

3.2 Custo de Execução

3.2.1 Algoritmo Guloso

Essa versão de implementação depende unicamente de dois processos chave: ordenar um arranjo de elementos e depois percorrer uma vez todo o conjunto, executando duas divisões e três somas. Considerando que as operações de atribuição realizadas no algoritmo de ordenação tenham custo $O(1)$ e que também o custo de as operações aritméticas executadas dentro do laço do Algoritmo 1 possuam o mesmo custo de $O(1)$, não é possível afirmar que o custo de execução do algoritmo gulosos seja linear.

Para esta solução o custo envolvido sobre todo o processo de desenvolvimento de uma resposta para o plano de viagem do grupo de amigas envolve uma ordenação de um vetor de n elementos ($O(n \log n)$) e o encaminhamento de um arranjo de mesmo tamanho de forma linear ($O(n)$). Sendo assim, o custo por trás da execução desse programa é de:

$$f(n) = n + n \log n = O(n \log n)$$

3.2.2 Algoritmo de Programação Dinâmica

Pode-se entender o custo de execução da parte de programação dinâmica como o custo de montagem de uma matriz de dimensões $n \times m$ (sendo n a quantidade de ilhas e m o valor da variável *custo_máximo*), uma vez que ao montar tal matriz executa-se exatamente a mesma operação, uma única vez para cada célula da estrutura. Considerando que a operação de maior custo constante dentro do laço de manipulação de uma célula seja definida pela soma dos custos de duas comparações, quatro atribuições e três somas e podendo configurar tal custo como contante, temos que essas operações são $O(1)$.

Seguindo o Algoritmo 2, observa-se duas operações que envolvem variáveis de entrada: Um *loop* para adicionar 0 para toda primeira linha da matriz e um segundo aninhamento de laços que executa as operações de definição do resultado final da tabela. O custo do primeiro laço é definido pela função $h(n) = n$, já o aninhamento é demonstrado pelo custo de se executar $g(n,m) = n \times (m+1)$ operações de custo constante. Tais operações são executadas independente dos valores de entrada, o que faz com que a entrada não interfira na limitação superior ou inferior, mas sim em ambas as assíntotas, dessa maneira:

$$n + n \times (m+1) = h(n) + g(n,m) = O(n \times m)$$

Por fim, vale destacar, que, apesar de se parecer com uma notação polinomial, o custo de execução e de utilização da memória para esse algoritmo não pode ser definido como polinomial na entrada. A definição de

[2] http://www.cplusplus.com/reference/utility/pair

custo polinomial toma como base a entrada do problema e uma vez que o comportamento assintótico dessa solução dependa do valor de uma variável e não do valor da quantidade de entidades da entrada, esse não é um problema de custo polinomial. Observe que pode-se executar uma instância do problema com apenas uma ilha, mas com um valor de custo_máximo muito alto e nesse caso o valor de $n \times m$ será extremamente grande apesar de não mudar a quantidade de elementos recebidos na entrada do programa.

4 Análise Experimental

A etapa experimental demandou a criação de *scripts* capazes de criarem instâncias do problema de forma aleatória. Tais instâncias foram executadas quinze vezes, sendo que gerou-se quatro instâncias para cada uma das seguintes quantidades de ilhas: 2, 4, 8, 12, 16 e 20. Após tal etapa, pôde-se criar uma tabela para observar o comportamento das médias e do desvio padrão de cada uma das intâncias, permitindo a validação do teste.

Para os dois primeiros gráficos mostrados na Figura 1, é possível notar o comportamento não linear em ambos os casos, tanto para o algorítimo guloso (gráfico da esquerda) quanto para a solução de programação dinâmica (direita). A variável de análise dos gráficos é o tempo de execução registrado em nanosegundo, sendo representado no eixo das ordenadas. Note que os dois gráficos são muito parecidos quanto ao comportamento, entretanto é possível afirma que esse é um efeito causado pelo pequeno tamanho da entrada, uma vez que para entradas pequenas, mesmo algoritmos exponenciais podem executar rapidamente. Porém, é preciso realizar um estudo mais detalhado a cerca de cada um dos dados de tempo obtidos nos gráficos.

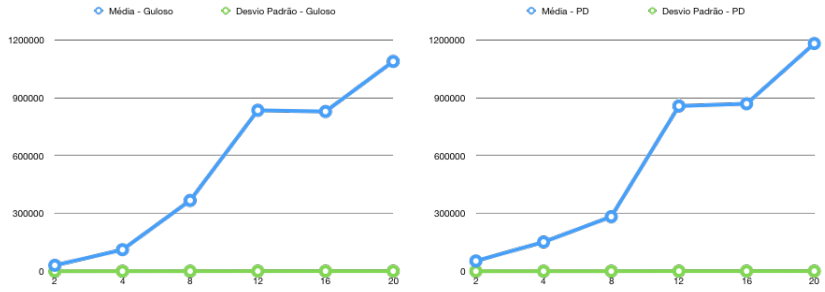

Figura 1: Análise do comportamento médio dos custos de cada comando.

Para uma visualização mais detalhada, observe na Figura 2 que mesmo com o comportamento muito parecido, a média de execução do algoritmo guloso se coloca abaixo da média de tempo de execução do algoritmo de programação dinâmica. Isso se deve ao fato citado anteriormente, que mesmo que sutil, há um maior custo de execução para algoritmos $\Theta(nm)$ em relação ao custo $O(n \log n)$ Tal diferença ficará mais nítida à medida em que o tamanho da entrada cresce, ou até mesmo à medida em que o valor do custo máximo (m) aumenta, pois para resolver usando programação dinâmica, o custo de execução é polinomialmente afetado pelo valor de m, em contrapartida, a solução gulosa não é afetada pela mesma variável.

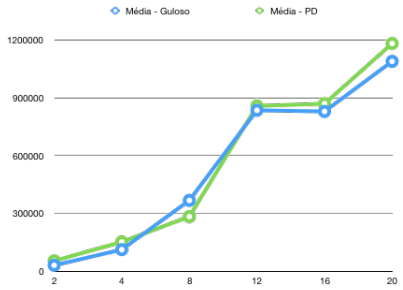

Figura 2: Análise do comportamento médio dos custos de cada comando.

A tabela a seguir resume os dados obtidos na análise experimental:

Tabela 2: Dados da análise experimental.

n	**Média - Guloso**	**Desv. Padr. - Guloso**	**Média - PD**	**Desv. Padr. - PD**
2	30910	167	53570	224
4	112015	328	152009	385
8	367656	586	283500	528
12	835516	897	857704	923
16	829381	908	869426	929
20	1089109	1040	1181926	1085

Por fim, pode-se observar o benefício de execução do algoritmo guloso, entretanto é valido considerar que nem sempre esta solução retornará um valor ótimo, apesar de possuir uma boa aproximação para a solução ótima. Já a segunda solução é excelente em retornar a melhor solução, porém paga-se o preço de um alto custo de execução e de ocupação da memória.

5 Conclusão

Ao permitir que se repitam as ilhas visitadas, pode-se concluir que enquanto o desejo do grupo de garotas for otimizar as férias para a quantidade de dias em viagem, a opção pelo algoritmo guloso será mais satisfatória, pesar de nem sempre prover a melhor saída possível. Entretanto, quando o quesito central for otimizar a quantidade de ilhas visitadas, o algoritmo que utiliza programação dinâmica será de maior valia, visto que fornece a saída correta de otimização da pontuação da viagem e que não permite repetições. Logo, por consequência o segundo algoritmo pode ser uma ótima opção para aumentar a gama de lugares visitados.

Por outro lado, o estudo das duas soluções permitiu analisar que o algoritmo guloso executa de forma mais rápida que algoritmo de programação dinâmica. Isso é justificado pelo fato do custo de execução dos dois casos se diferir muito, logo o esperado é que a solução $O(n \log n)$ execute mais rápida que $\Theta(nm)$, fato que é comprovado pela etapa de experimentação.