Trabalho Prático 2 - Algoritmos 1

Luiz A. D. Berto
2018047099

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

luizberto@dcc.ufmg.br

Abstract. *This work aims to solve two variations of the Knapsack problem (under a island travel context) using a greedy approach and a dynamic programming approach. We compare the problems, analysing which one has the best expected sum of scores, and find that under the constraints of the repeating islands problem, the greedy algorithm fails to deliver the optimal output for certain cases.*

Resumo. *Esse trabalho visa resolver duas variações do problema da Mochila (sob um contexto de viagens a ilhas) utilizando uma abordagem gulosa e uma abordagem de programação dinâmica. Comparamos os problemas, analisando qual tem a melhor soma de pontuação esperada, e verificamos que sob as restrições do problema de ilhas com repetição, o algoritmo guloso não retorna a saída ótima para certos casos.*

1. Introdução

Esse trabalho aborda um problema que é uma adaptação do famoso problema da Mochila. Basicamente, um grupo de amigas deseja realizar uma viagem a um conjunto de ilhas, e elas pontuaram cada ilha de acordo com o quão vantajoso/interessante é passar um dia na ilha. Cada ilha, então, possui um custo por dia (que considera transporte, alimentação e estadia), e uma pontuação (também diária). Dado isso, considerando uma restrição orçamentária, o programa deve determinar qual é a pontuação máxima e a quantidade de dias que a viagem poderá durar, em duas situações:

1. caso as amigas possam permanecer mais de um dia numa mesma ilha;
2. caso as amigas só possam permanecer um único dia numa mesma ilha.

Podemos ver a relação direta com o problema da Mochila, no qual as ilhas seriam os itens. A pontuação de cada ilha determinada pelas amigas seria o valor do item, e o custo diário da ilha seria o peso do item. Além disso, a restrição orçamentária das amigas seria o peso máximo que a mochila consegue carregar. O programa faz uso de um algoritmo guloso e um algoritmo de programação dinâmica, respectivamente, para resolver os problemas acima citados. Nas próximas seções, haverá explicações mais aprofundadas sobre o funcionamento desses algoritmos.

2. Implementação

2.1. Estruturas de dados

Foi criada uma classe *Island*, que encapsula os dados de uma ilha (custo e pontuação). Ela também possui o método *getCostPerPoint*, que retorna o custo por unidade de pontuação,

que é calculado sempre que o método é chamado. Além disso, existe a classe *MergeSortIslands*, que é simplesmente uma classe estática que expõe um método para ordenar vetores do tipo *Island[]* baseado no método *getCostPerPoint*.

2.2. Algoritmo guloso

Para resolver esse problema utilizando uma abordagem gulosa, para cada ilha é calculado o seu custo por ponto, ou seja, o custo da ilha diário dividido pela quantidade de pontos diários desta. A partir desse valor, as ilhas são ordenadas em ordem crescente de custo por ponto. Após isso, o vetor de *Island* é simplesmente processado do menor para o maior custo por ponto, verificando se o dinheiro restante (*dr*) é suficiente para adicionar cada ilha (uma ou mais vezes). Para cada ilha *i*, o algoritmo faz o seguinte processamento:

Seja *c* o custo da ilha atual, *p* sua pontuação, *q* a quantidade de dias da viagem, *dr* o dinheiro restante, e a pontuação final *v*:

- calcula quantas vezes t que a ilha "cabe" no orçamento restante dr, com a seguinte fórmula: $\frac{dr}{c}$;
- adiciona à pontuação final v a pontuação diária da ilha vezes a quantidade de vezes que ela cabe no orçamento ($t \times p$);
- adiciona à q o valor t;
- subtrai do dinheiro restante o preço da ilha vezes a quantidade de vezes que ela cabe no orçamento: $dr = dr - t \times c$.

Essa abordagem visa minimizar o custo por pontos de maneira gulosa. Com isso, ao final da execução, teremos a pontuação final v e a quantidade q de dias que a viagem durará.

2.2.1. Prova de corretude

Para algumas instâncias do problema, o algoritmo consegue retornar a solução ótima. No entanto, ele pode falhar, pois ele é o algoritmo utilizado para uma variação do problema da Mochila, no qual itens podem ser adicionados parcialmente (Fractional Knapsack Problem) [gee 2019]. Com essa possibilidade, o algoritmo sempre iria adicionar uma única ilha: a que possuir o menor custo por ponto, quantas vezes fosse necessário, e utilizaria frações de dias para garantir a otimalidade. Como no caso desse trabalho essa situação não é possível, então o algoritmo pode falhar. Prova por contradição que o algoritmo pode falhar:

- Considere, por contradição, que o algoritmo retorne a solução ótima para qualquer instância do problema.
- Seja uma instância do problema com as seguintes características:
  custo máximo da viagem de 15000
  ilha A, com custo diário de 9000 e pontuação 140
  ilha B, com custo diário de 8000 e pontuação 80
  ilha C, com custo diário de 7000 e pontuação 70
- O algoritmo iria ordenar as ilhas por custo por ponto, e iria processá-las na seguinte ordem: ilha A (custo por ponto de aproximadamente 64), ilha B (custo por ponto de 100), ilha C (custo por ponto também de 100).

- O algoritmo iria incluir a ilha A uma vez, e não conseguiria incluir nenhuma outra ilha, retornando como solução ótima a pontuação 140 e a quantiade de dias 1.
- No entanto, se as ilhas B e C fossem utilizadas, teríamos uma pontuação de 150, que é maior que a pontuação retornada pelo algoritmo guloso (contradição).

2.3. Algoritmo de programação dinâmica

No algoritmo de programação dinâmica foi utilizada uma abordagem bottom-up, ou seja, são computados todos os subproblemas menores que o problema real (de tamanho $n \times m$), partindo dos subproblemas menores, e seus resultados são armazenados numa matriz de tamanho $n \times m$. Isso é particularmente interessante para esse problema, pois ele tem a propriedade *overlapping subproblems*: o problema pode ser quebrado em subproblemas que aparecem várias vezes. Com isso, é extremamente vantajoso armazenar o resultado dos subproblemas, pois eles podem ser utilizados várias vezes durante a construção da solução do problema real.

Seja *i* a quantidade de ilhas que serão inclusas por iteração, *p* a pontuação da ilha atual sendo processada, *c* seu custo, e *dr* a quantidade de dinheiro restante, a equação de Bellman para o algoritmo é a seguinte [Kleinberg and Tardos 2005]:

$OPT(i, dr) = 0$ se $i = 0$

$OPT(i - 1, dr)$ se $c_i > dr$

$max\{OPT(i - 1, dr), p_i + OPT(i - 1, dr - c_i)\}$ caso contrário

No final das $n \times m$ iterações, o elemento no índice $[n][m]$ da matriz de lookup terá a solução ótima para o problema em termos de pontuação.

Para recuperar a quantidade de dias, precisamos de fazer uma pesquisa na tabela de lookup. Essa pesquisa começa da última posição da tabela, e vai procurando o primeiro elemento b da última coluna, de baixo para cima, que difere do elemento anterior a. Caso isso ocorra, significa que a faz parte da solução final para o problema, então incrementa-se o contador de dias. Para verificar qual a pontuação da ilha que foi incluída naquele ponto, apenas referenciamos ao vetor de ilhas na posição $i - 1$ (o índice deve ser normalizado, pois a tabela de lookup tem uma linha e uma coluna a mais que o número de elementos do vetor de ilhas). A partir disso, passamos para o elemento $[i - 1][c - island[i].cost]$, e fazemos essa verificação até que a pontuação total, que é decrementada a cada iteração junto com o custo, seja igual a zero.

2.3.1. Prova de corretude

Partindo do pressuposto que o algoritmo utilizado é o mesmo algoritmo do problema da Mochila, podemos assumir que este é correto. Seja n o valor máximo da viagem, e m a quantidade de ilhas. Esse algoritmo irá realizar no máximo $n \times m$ iterações, pois é a quantidade de operações necessárias para preencher a tabela de todos os subproblemas menores que o problema real, de tamanho $n \times m$ (algoritmo termina). Quanto ao fato de retornar o resultado ótimo, podemos provar por indução:

- Passo base: para um subproblema de tamanho $(0, i]$, o algoritmo retorna a solução 0, que é ótima.

- Hipótese indutiva: para todos os subproblemas de tamanho i, o algoritmo retorna a solução ótima.
- Passo indutivo: considere um subproblema de tamanho i, sua solução será o máximo entre o subproblema de tamanho $i-1$ (ótimo pelo passo base) e o subproblema de tamanho $i-1$ acrescido da pontuação da i-ésima ilha. Com isso, garante-se que a sequência de soluções é monotonicamente crescente, e que a ilha i só será adicionada caso seja vantajoso adicioná-la, garantindo que a solução seja ótima.

3. Instruções de compilação e execução

Para compilar o programa, é necessário o uso do *make*. Após o download do *.zip*, descompacte-o, e dentro da pasta *alg1-tp2*, abra um terminal e execute o seguinte comando:

- make

Para executar o programa, basta referenciar o binário criado (dentro da pasta *bin/*), da seguinte maneira:

./bin/tp2 input-file-name

4. Análise de complexidade

4.1. Espaço

4.1.1. Algoritmo guloso

O algoritmo guloso utiliza $O(m)$ de espaço adicional, visto que faz uso do *MergeSort*, que em cada chamada recursiva aloca um vetor auxiliar no procedimento *merge*.

4.1.2. Algoritmo de programação dinâmica

O algoritmo de programação dinâmica utiliza $O(n \times m)$ de espaço adicional, que é a tabela de lookup dos subproblemas.

4.2. Tempo

4.2.1. Algoritmo guloso

Basicamente, esse algoritmo ordena as ilhas ($O(mlogm)$) pelo custo por ponto, e depois percorre todas as ilhas ($O(m)$) tentando as adicionar à solução final. Esse algoritmo custa $O(mlogm)$, pois seu custo dominante é o de ordenar o vetor de ilhas - $O(mlogm)$ domina assintóticamente $O(m)$. O algoritmo de ordenação utilizado é o *MergeSort*, que apesar de possuir um caso médio pior que o *QuickSort*, garante a complexidade de $O(mlogm)$ para qualquer configuração da entrada, o que é um requisito desse trabalho.

4.2.2. Algoritmo de programação dinâmica

O algoritmo de programação dinâmica faz no máximo $O(n \times m)$ operações para resolver o problema de tamanho $n \times m$, pois resolve cada subproblema de tamanho até $n \times m$

em uma operação de custo constante $O(1)$, preenchendo a tabela de lookup. Após isso, para determinar a quantidade de dias da viagem, são feitas no máximo $O(m)$ operações, pois o *for* utiliza como contador uma variável que varia de m até 0. Portanto, o custo assintótico do algoritmo é de $O(n \times m)$ se n for maior do que 0, visto que $O(n \times m)$ domina assintoticamente $O(m)$ nesse caso.

5. Análise experimental

Para analisar o desempenho do programa de maneira empírica, foram feitos testes com entradas de diferentes tamanhos para comparar o tempo de execução entre elas, de forma que possamos verificar que os tempos de execução cresçam de acordo com o esperado pela análise assintótica.

Para isso, foram gerados automaticamente 10 casos de teste customizados. O menor deles possui 100 ilhas, e cada caso de teste subsequente tinha 100 ilhas a mais que o anterior, até o máximo de 1000 ilhas. Em cada caso de teste, todas as ilhas (menos a primeira) tinham custo e pontuação iguais, ou seja, o "custo benefício" de todas era o mesmo: 1 ($n = m$). No caso da primeira ilha, sua pontuação era uma unidade maior que seu custo, fazendo com que sempre a primeira ilha seja a mais vantajosa. A fim de aumentar a assertividade da análise experimental, cada entrada foi executada 100 vezes, e a média e desvio padrão dos dados coletados foi calculada. *Nota de implementação: o código de checagem de tempo de execução não foi incluído junto ao código fonte final, pois a inclusão desse poderia resultar em erros de correção automática, visto que este imprimia na saída padrão.*

Os dados de cada execução foram incluídos anexos à entrega. Abaixo, as figuras 1 e 2 mostram as médias e desvios padrões calculados para cada tamanho de entrada, para o algoritmo guloso e o algoritmo de programação dinâmica, variando de 100 a 1000, conforme especificado acima.

Algoritmo guloso		
N	Média (s)	Desvio padrão (s)
100	0.0000329918	0.0000177929
200	0.0000352170	0.0000141254
300	0.0000437127	0.0000159606
400	0.0000518425	0.0000141598
500	0.0000654140	0.0000229131
600	0.0000778776	0.0000287363
700	0.0000800599	0.0000170391
800	0.0000968822	0.0000284807
900	0.0001048613	0.0000275254
1000	0.0001202578	0.0000354999

Figure 1. Tabela com as estatísticas dos tempos de execução do algoritmo guloso.

Algoritmo Programação Dinâmica		
N	Média (s)	Desvio padrão (s)
100	0.0000518082	0.0000130519
200	0.0001912790	0.0000332400
300	0.0004284042	0.0000544047
400	0.0007554963	0.0000760973
500	0.0011401108	0.0000825569
600	0.0016802689	0.0001612987
700	0.0023259371	0.0004115717
800	0.0030113308	0.0004100344
900	0.0037708999	0.0002714771
1000	0.0046004984	0.0002826752

Figure 2. Tabela com as estatísticas dos tempos de execução do algoritmo de programação dinâmica.

Comparando as tabelas, podemos perceber que o tempo de execução médio do algoritmo de programação dinâmica é maior em todos os casos, e também cresce a uma taxa mais elevada. Podemos visualizar isso mais facilmente com o gráfico da figura 3, que compara o crescimento da média do tempo de execução dos dois algoritmos para o mesmo tamanho de entrada.

Figure 3. Gráfico comparando o tempo de execução dos dois algoritmos para entradas do mesmo tamanho.

O gráfico mostra que o tempo médio de execução do algoritmo de programação dinâmica é bem superior, exibindo comportamento ”quadrático” (lembrando que $n = m$ para as entradas utilizadas), enquanto o algoritmo guloso tem tempo muito inferior, apresentando comportamento ”linearitmico”. O gráfico da figura 4 confirma a hipótese de que o algoritmo de programação dinâmica roda em $O(n \times m)$, pois nele o eixo x é $n \times m$, e a tendência passa a ser linear:

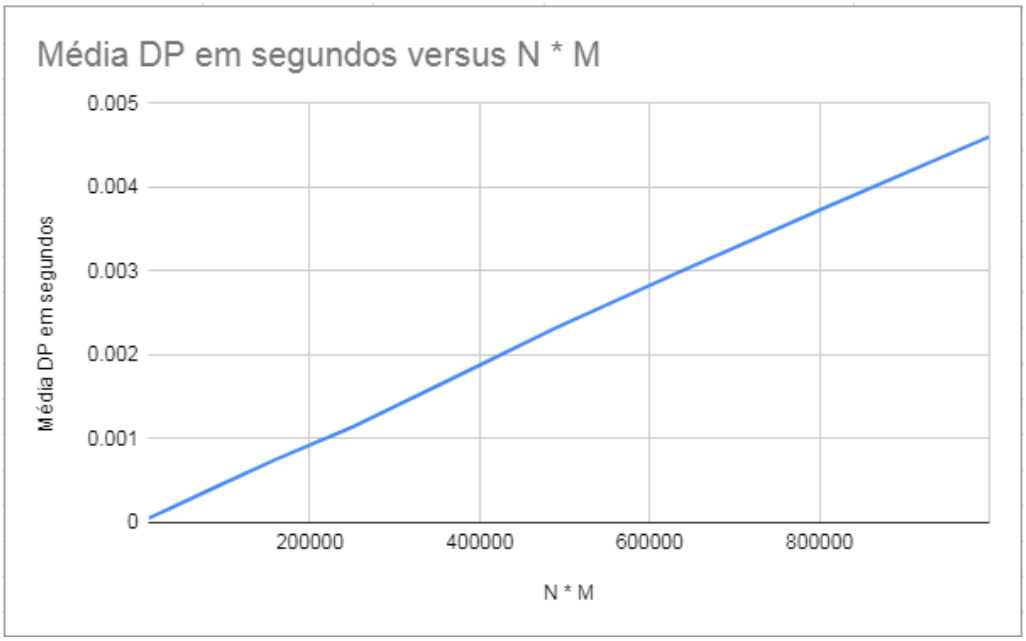

Figure 4. Gráfico comparando o tempo médio de execução do algoritmo de programação dinâmica com $n \times m$.

5.1. Comparação das alternativas

O algoritmo guloso é mais interessante assintoticamente tanto em termos de tempo de execução quanto espaço. Portanto, se o tempo de execução/espaço for um problema, talvez seja mais interessante o uso do algoritmo guloso. Entretanto, devemos nos atentar que os algoritmos resolvem **problemas diferentes**. É importante considerar também que nem sempre o algoritmo guloso consegue retornar a resposta ótima para um problema, conforme discutido na seção de implementação.

Quanto à comparação dos problemas, podemos verificar que em algumas das vezes o problema que admite repetição de ilhas terá uma solução ótima melhor que a do problema que não admite repetição de ilhas, pois em instâncias em que existe uma ilha muito vantajosa (com um custo por ponto muito inferior às outras), ela seria incluída múltiplas vezes, o que não seria possível no problema que não admite repetição. Todavia, isso vem com um custo, que é a inabilidade do algoritmo guloso determinar a resposta ótima para todas as instâncias do problema que admite repetição de ilhas.

6. Conclusão

Conclui-se que, por mais que o algoritmo guloso seja uma **aproximação** boa para o problema da Mochila com repetição, ele é apenas uma heurística, e não deve-se depender de sua corretude para resolver problemas em computação.

References

(2019). Fractional knapsack problem. https://www.geeksforgeeks.org/fractional-knapsack-problem/. Acessado em: 2019-10-17.

Kleinberg, J. and Tardos, E. (2005). *Algorithm design*. Pearson, 1st edition.