UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Algoritmos e Estruturas de Dados III
Trabalho Prático 3 – Legado da Copa

Caio Felipe Zanatelli

Belo Horizonte
10 de julho de 2017

1 Introdução

Em Ciência da Computação, diversas vezes um problema pode ser abordado de formas distintas. Neste contexto, os paradigmas de programação são de fundamental importância, uma vez que, com seu entendimento correto, pode-se chegar a um algoritmo mais adequado para um problema específico.

Com isso, este trabalho tem como objetivo propor uma solução para o seguinte problema: dada uma rua, a qual possui alguns bares e casas, sabendo que de um lado a numeração é par e do outro é ímpar, além de ambos os lados possuírem numeração em ordem crescente, determinar a quantidade máxima de bandeirolas que podem ser conectadas de um bar à casa de seu respectivo dono sem que haja cruzamento entre bandeirolas, sabendo também que a casa do dono sempre está na calçada oposta ao seu bar. Para tanto, são apresentadas as abordagens por força bruta, gulosa e por programação dinâmica.

2 Solução do Problema

Esta seção tem como objetivo apresentar a solução implementada. Para tanto, inicialmente será introduzida a modelagem proposta, ilustrando as etapas da resolução do problema. Em seguida serão apresentados três algoritmos para a sua resolução: força bruta, algoritmo guloso e programação dinâmica. Por fim, a estrutura de código utilizada é apresentada.

2.1 Modelagem do Problema

Dada a rua do problema, primeiro é necessário tratar a entrada. Como de um lado da rua só existem números ímpares e do outro só existem números pares, no momento da leitura da entrada deve ser feito o mapeamento correto. Assim, os pares (x, y) da entrada são mapeados em uma estrutura *pair<int, int>* definida. O primeiro elemento, por convenção adotada, é intitulado *bar*, e o segundo foi chamado de *owner*, representando assim que uma bandeirola deve sair de um bar e chegar ao seu respectivo dono.

Para entender a modelagem, suponhamos que os pares $< bar, owner >$ estejam ordenados de acordo com as coordenadas *bar*. Se duas bandeirolas se cruzam, então deve existir uma bandeirola com coordenadas (a_i, b_i) e alguma outra bandeirola com coordenadas (a_j, b_j) tal que (1) $a_i < a_j$ e $b_i > b_j$ ou (2) $a_i > a_j$ e $b_i < b_j$. Assim, para que não haja cruzamentos, é preciso garantir que todos os pares conectados atendam um dos dois casos: (1) $a_i \leq a_j$ e $b_i \leq b_j$ ou (2) $a_i \geq a_j$ e $b_i \geq b_j$. Com isso, percebe-se que, se os pares forem ordenados de acordo uma das coordenadas, o grupo das outras coordenadas que fazem parte da solução também devem estar em ordem crescente. Um resultado importante disso é que, se a instância do problema foi ordenada de acordo com uma das coordenadas, ou seja, o conjunto está parcialmente ordenado, então a maior quantidade possível de bandeirolas que podem ser conectadas sem cruzamento é exatamente o tamanho da maior subsequência crescente (LIS) do outro conjunto de coordenadas, sendo uma subsequência um conjunto de elementos não necessariamente contíguos.

Neste contexto, o ponto central da solução deste problema está na redução apresentada acima. Assim, após tratar devidamente a entrada fornecida, a mesma é ordenada de acordo com a coordenada *owner*, utilizando o método de ordenação *QuickSort*. Em seguida, a resposta esperada é encontrada através do cálculo do tamanho da maior subsequência crescente presente no conjunto das coordenadas *bar*. De forma a proporcionar um melhor entendimento, a Figura 1 ilustra este processo. Neste exemplo, percebe-se pela parte inferior da imagem que o tamanho da LIS é cinco, que é justamente a quantidade máxima de conexões que podem ser feitas sem cruzamento algum. Essa configuração é mostrada pelas setas entre os pares correspondentes.

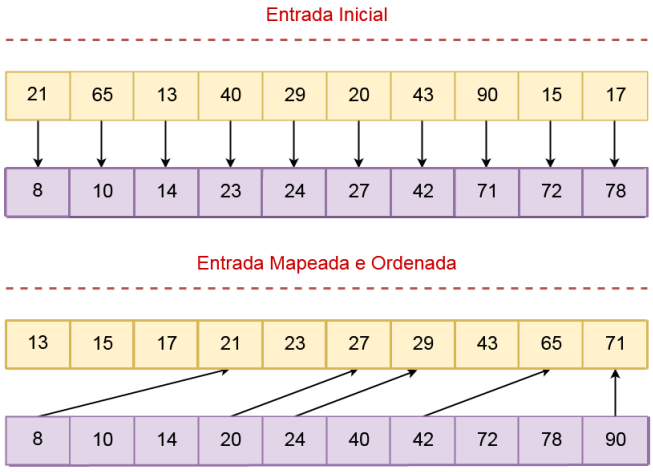

Figura 1 – Exemplo de resolução do problema (LIS)

.

2.2 Maior Subsequência Crescente

Como abordado, as implementações deste trabalho são pautadas na redução do problema original para um problema de encontrar a maior subsequência crescente no conjunto de coordenadas *bar*, dado que a instância do problema foi previamente ordenada parcialmente em função do conjunto de coordenadas *owner*. Assim, de forma a proporcionar um melhor entendimento acerca dos algoritmos que serão introduzidos posteriormente, faz-se necessário definir formalmente o conceito LIS apresentado.

Portanto, seja π uma permutação randômica dos inteiros $1, 2, 3, ..., n$ e $\pi(i)$ o *i-ésimo* elemento dessa permutação. Definimos $l(\pi)$ a maior subsequência crescente em π, sendo uma subsequência crescente $(i_1, i_2, ..., i_k)$ de π uma subsequência que satisfaz a seguinte propriedade:

$$i_1 < i_2 < i_3 < ... < i_k$$

$$\pi(i_1) < \pi(i_2) < \pi(i_3) < ... < \pi(i_k)$$

2.3 Algoritmo Força-Bruta

A primeira opção de algoritmo fornecida por esta implementação é a opção força-bruta. A ideia desse algoritmo é utilizar a modelagem proposta anteriormente para resolver o problema através da exploração de todo o espaço de busca para encontrar a maior subsequência crescente. Dessa forma, uma função recursiva é executada de forma a tentar todas as possibilidades de sequências crescentes e então retornar o tamanho da maior delas. O pseudocódigo para obter tal resultado é apresentado abaixo.

```
1:  function GetLISBackTracking(A[0...n], previous, left, right):
2:      if left > right then
3:          return 0
4:      else
5:          if A[left] ≤ previous then
6:              return GetLISBackTracking(A[0...n], previous, left + 1, right)
7:          else
8:              x ← GetLISBackTracking(A[0...n], previous, left + 1, right)
9:              y ← 1 + GetLISBackTracking(A[0...n], A[left], left + 1, right)
10:             return max(x, y)
11:         end if
```

12: **end if**
13: **end function**

2.4 Algorimo Guloso

O segundo algoritmo implementado neste trabalho é com abordagem gulosa. Para tanto, seguindo a metodologia LIS, foi implementado o algoritmo *Patience Sort*, o qual fornece o resultado desejado através de escolhas locais ótimas e, para este caso, fornece também uma solução ótima. O algoritmo em questão é apresentado a seguir.

2.4.1 Patience Sorting

Primeiramente, descrevemos o jogo *Patience Sorting*, o qual foi inventado como um método de ordenação de cartas, o qual funciona da seguinte maneira. Pegue um deque de cartas numeradas como $1, 2, 3, ..., n$. Inicialmente o deque está embaralhado. As cartas são então viradas para cima uma a uma e são postas em cima de pilhas de cartas de acordo com a seguinte regra: Uma carta menor pode ser colocada em cima de uma carta maior ou em outra pilha à direita das pilhas existentes. A cada estágio, temos acesso somente à carta que está no topo de cada pilha. Se a carta que acabou de ser virada é maior do que aquelas do topo, então ela *deve* ser colocada em uma nova pilha à direita das pilhas já existentes. Assim, o objetivo do jogo é terminar com a menor quantidade possível de pilhas.

Embora pareça um jogo um tanto quanto aleatório, uma estratégia simples para jogar produz uma forma eficaz de computação de $l(\pi)$. Neste caso, a estratégia é gulosa, na qual a carta atual é colocada na pilha mais à esquerda possível. Para visualizar melhor, a Figura 2 ilustra um exemplo de *Patience Sorting* com jogadas gulosas para um deque com 10 cartas, as quais são dispostas na seguinte ordem: $< 8, 3, 7, 9, 2, 5, 4, 1, 10, 6 >$.

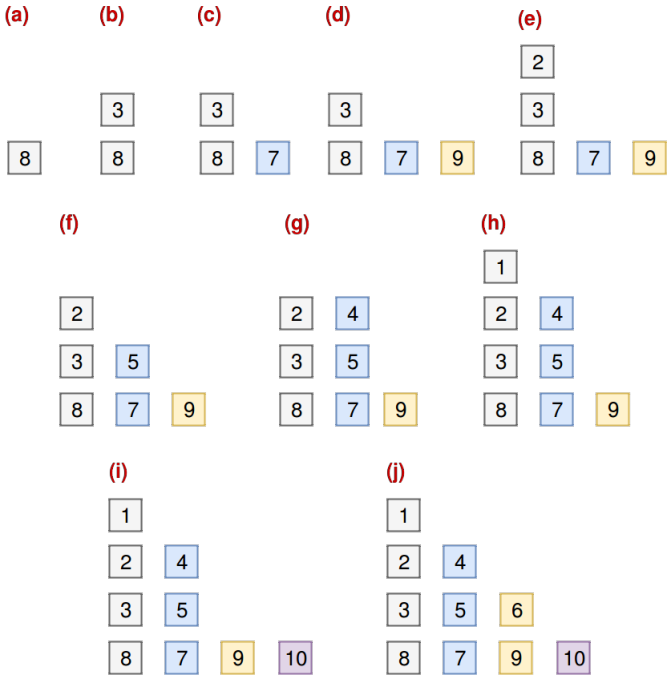

Figura 2 – Exemplo de estratégia gulosa para solução *Patience Sorting*

.

Seja π a ordenação de um deque de cartas numeradas $1, 2, 3, ..., n$. Através de uma estratégia gulosa para jogar *patience sorting* utilizando π irá resultar em exatamente $l(\pi)$ pilhas. Para

provar isso, primeiro provamos que o número de pilhas é pelo menos $l(\pi)$ e então provamos que é no máximo $l(\pi)$.

Seja $i_1 < i_2 < ... < i_k$ uma subsequência crescente em π. Como só é permitido colocar cartas no topo das cartas de menor valor, temos que $\pi(i_j)$ deve sempre ficar em uma pilha à direita de $\pi(i_{j+1})$. Portanto, a estratégia gulosa, assim como qualquer outra estratégia válida, irá gerar no mínimo $l(\pi)$ pilhas.

Patience Sorting pode ser também utilizado para encontrar uma instância de uma subsequência crescente, e com isso mostrar que pode haver no máximo $l(\pi)$ pilhas. Em qualquer situação que uma carta c for colocada em uma pilha i que não seja a primeira, desenhamos uma aresta de c para o topo da carta d da pilha anterior $i-1$. Mas $d < c$, pois na estratégia gulosa utilizada c seria colocada no topo de d se $d > c$. Ainda, cada aresta vai de uma carta à frente para uma carta anterior, como ilustra a Figura 3.

Assim, se existirem k pilhas, seja a_k a carta do topo da pilha mais à direita. Então, seguindo a aresta de a_k para a_{k-1} e assim sucessivamente, uma subsequência crescente será construída em π. Portanto, há no máximo $l(k)$ pilhas e, portanto, o *Patience Sorting* fornece uma solução ótima para o problema proposto utilizando uma abordagem gulosa. ■

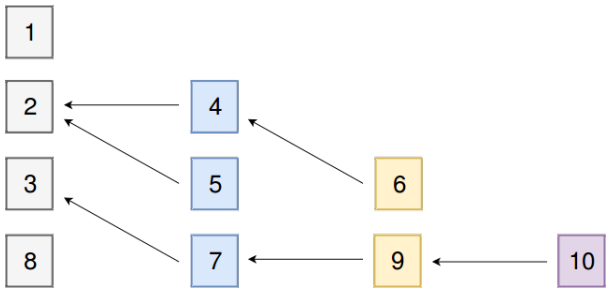

Figura 3 – Construção de uma subsequência crescente (*Patience Sorting*)

.

2.5 Programação Dinâmica

Por fim, o último algoritmo implementado neste trabalho foi através de Programação Dinâmica. A ideia para resolver o problema dado é a mesma descrita durante todo este documento: ordenar a instância do problema em relação a um conjunto de coordenadas e então aplicar um algoritmo para descobrir o tamanho da maior subsequência crescente do outro conjunto de coordenadas.

O algoritmo utilizando Programação Dinâmica funciona da seguinte forma. Primeiro é criada uma tabela de memorização, aqui chamada de DP, com tamanho n, onde n é o tamanho do vetor de entrada A, ou seja, as coordenadas $< bar, owner >$). Depois, iterando sobre essa tabela, de $i = 1$ até n, fazemos DP[i] = $max\{1, max_{j<i, A[j]<A[i]}\{1 + DP[j]\}\}$. Por fim, retornamos o maior valor contido no vetor DP.

Precisamos agora provar que este algoritmo está correto. Para tanto, consideremos A$[1, i]$ os primeiros i elementos do vetor. Seja OPT(i) o tamanho da LIS de A$[1, i]$, que acaba na posição i. Afirmamos que OPT(i) satisfaz a seguinte relação de recorrência:

$$OPT(i) = \begin{cases} 0, & \text{se } i = 0 \\ max \begin{cases} 1, & \\ max_{j<i, A[i]<A[j]}\{1 + OPT(j)\} \end{cases} & \text{caso contrário} \end{cases} \quad (1)$$

Primeiro, provamos que o algoritmo satisfaz a relação de recorrência dada. Se $i = 0$, então existe apenas uma subsequência de zero elementos do vetor, isto é, a subsequência vazia. Assim, OPT(i) = 0. Se $i \geq 1$, então seja S^* uma LIS de A$[1, i]$. Considere então S todo o conjunto S^* retirando o último elemento. Se S é vazio, então S^* tem tamanho um, logo OPT(i) = 1. Caso

contrário, se S não for vazio, então S deve terminar em alguma posição j do vetor. Como S^* é uma subsequência crescente, sabemos que A[j] < A[i]. Mais do que isso, S deve ser uma LIS de A[$1, k$] para algum $k < i$, onde A[k] < A[i]. Senão, não poderíamos extrair S para uma LIS do subvetor em questão, aumentando o tamanho de S^* e contradizendo o fato de que S^* é ótimo. Portanto, o tamanho de S^* é dado pelo valor $max_{j<i, A[j]<A[i]}\{1 + OPT(j)\}$. ■

Em seguida, provamos que o algoritmo retorna o tamanho da maior subsequência crescente de A. O algoritmo avalia a recorrência OPT(i) de forma *bottom-up*, armazenando OPT(i) em DP[i]. Note que, como avaliamos DP[i] enquanto i aumenta de 1 até n, todos os valores para os subproblemas referenciados pela recorrência OPT(i) já terão sido computados. Ao fim do processo, o algoritmo retorna o maior valor contido no vetor DP, o que corresponde ao tamanho da maior LIS que termina em algum ponto no vetor, o que na verdade é a LIS de todo o vetor A. ■

Por fim, o algoritmo apresentado pode ser implementado em $O(n^2)$ da seguinte forma: na iteração i, percorra os índices $k = 1, 2, ..., i - 1$ e dos valores lidos e menores do que A[i], escolha aquele que fornece o maior valor DP[k] correspondente (se existir algum). Então, faça DP[i] ser o maior valor (se existir) entre aquele valor e DP[$i - 1$]. Isso custa $O(n)$, então com n iterações todo o processo é feito em $O(n^2)$. O final do algoritmo é percorrer o vetor DP, de forma a retornar seu maior elemento, o que pode ser feito em $O(n)$, levando à complexidade de $O(n^2)$. Entretanto, neste trabalho o método de busca utilizado foi a busca binária, substituindo um termo da complexidade por $\log n$, fazendo que a complexidade total do algoritmo seja $O(n \cdot \log n)$.

2.6 Organização do Código

De forma a proporcionar uma melhor organização e modularização do código, cada implementação de paradigmas diferentes foi efetuada através de uma biblioteca específica. Dessa forma, cada biblioteca possui uma função acessível pelo correspondente ao algoritmo selecionado. Portanto, a organização do código em termos de bibliotecas implementadas ficou da seguinte forma:

- **Instance.h**: fornece uma abstração para o problema, transformando a entrada em uma variável do tipo de dados *Instance*, definido internamente na biblioteca. O tipo *Instance* possui um vetor do tipo *Pair*, o qual é alocado dinamicamente, e um atributo inteiro *size*, o qual indica a quantidade de itens que o vetor possui. O tipo *Pair* também é definido internamente nessa biblioteca, pois é utilizado para compor as coordenadas $< bar, owner >$ do problema. Além disso, fornece as operações de leitura da entrada e comparações entre chaves dessa instância. Como essas operações são de propósito geral, ou seja, são utilizadas por todas as bibliotecas de resolução do problema, elas são declaradas como funções públicas acessíveis pelo usuário.
- **List.h**: fornece uma implementação de lista por arranjos. É utilizada pela biblioteca *SolveGreedy.h* de forma a proporcionar a criação de uma listas de pilhas para o bom funcionamento do *Patience Sort*.
- **SolveBackTracking.h**: fornece as funções necessárias para a resolução do problema proposto através de um algoritmo recursivo por força bruta. Recebe como parâmetro uma instância do problema, ou seja, uma variável do tipo *Instance*. Complexidade temporal total: $O(2^n)$.
- **SolveGreedy.h**: fornece as funções necessárias para a resolução do problema proposto através de um algoritmo guloso, mais especificicamente o *Patience Sort*. Recebe como parâmetro uma instância do problema, ou seja, uma variável do tipo *Instance*. Complexidade temporal total: $O(n \cdot \log n)$.
- **SolveDynamicProgramming.h**: fornece as funções necessárias para a resolução do problema proposto através de um algoritmo por programação dinâmica. Recebe como parâmetro uma instância do problema, ou seja, uma variável do tipo *Instance*. Complexidade temporal total: $O(n \cdot \log n)$.

3 Análise de Complexidade

Nesta seção serão apresentadas as análises de complexidade de tempo e espaço dos algoritmos implementados.

3.1 Complexidade Temporal

Inicialmente, a biblioteca *Instance.h* é responsável por fornecer as funções de leitura da entrada do problema. Como são n pares no formato $< bar, owner >$, esse procedimento é feito em $O(n)$. Para isso, é preciso antes alocar uma instância do problema, uma variável do tipo *Instance* que contém um vetor do tipo $pair < int, int >$, um tipo definido internamente na biblioteca. Como se trata de uma alocação dinâmica de vetor, isso é feito em $O(1)$. Antes de terminar a execução do algoritmo, porém, a estrutura do tipo *Instance*, a qual contém um vetor de $pairs$, deve ser desalocada, o que é equivalente a desalocar o vetor de $pairs$. Neste caso, a complexidade temporal também é constante. Por fim, a função *solveInstance*, ainda na biblioteca *Instance.h*, faz a ordenação da entrada em função das coordenadas *owner* e então chama uma função para encontrar o tamanho da maior subsequência crescente, de acordo com o algoritmo selecionado. Para tanto, a ordenação utiliza o método *QuickSort*, o qual possui complexidade $O(n \cdot \log n)$. Assim, a complexidade total dessa função, a qual é acessada pelo usuário para a solução do problema, portanto sendo a função que determina a complexidade final do algoritmo, é $O(n \cdot \log n) + O(f(n))$, onde $O(f(n))$ é a complexidade do algoritmo selecionado. Como será abordado no decorrer desta seção, a menor das complexidades é $O(n \cdot \log n)$, e portanto a complexidade temporal resultante desta implementação é $O(f(n))$ [1].

Quando o algoritmo selecionado é por força bruta, a função *solveBacktracking*, da biblioteca *SolveBackTracking.h*, é chamada. Essa função invoca então o método *getLISLengthBackTracking*, o qual implementa uma recursão para encontrar a LIS desejada. Neste caso, todo o espaço de busca é percorrido, logo todas as possibilidades são testadas. Sendo $l(i)$ a maior subsequência crescente até o momento, a solução pode incluir ou não o elemento $\mathrm{A}[i+1]$, com $i+1 < n$, dependendo se $\mathrm{A}[i] < \mathrm{A}[i+1]$. Portanto, como o elemento pode ou não ser incluído na solução, a complexidade temporal total desse método é $O(2^n)$, uma vez que existem n ítens no vetor. Portanto, a complexidade total da função *solveBackTracking* também é $O(2^n)$, uma vez que esta apenas invoca a função *getLISLengthBackTracking*.

Por outro lado, se o algoritmo selecionado foi a estratégia gulosa, a função *solveGreedy*, da biblioteca *SolveGreedy.h*, é executada. Como abordado na seção sobre o *Patience Sort*, este procedimento itera sobre todos os elementos do vetor e então seleciona a pilha na qual o elemento avaliado no momento deverá entrar, ou se deverá ser criada uma nova pilha. Como, por natureza da estratégia gulosa do *Patience Sort*, se um elemento for maior do que o topo das pilhas mais à esquerda este deve ser colocado em uma pilha criada à direita, os elementos que compõem os topos das pilhas na lista estão sempre ordenados em ordem crescente. Assim, para localizar a pilha do elemento atual, uma busca binária é executada. Sendo a complexidade da busca binária $O(\log n)$, a complexidade temporal total neste caso é $O(n \cdot \log n)$. Vale ressaltar, neste ponto, que as operações de alocação/desalocação de memória, inserção e remoção no TAD *List* são executadas todas em $O(1)$, uma vez que a lista é implementada por arranjos, o que não contrubui para o aumento da complexidade do algoritmo apresentado.

[1] A complexidade pode mudar, uma vez que o pior caso do *QuickSort* é $O(n^2)$. Entretanto, como esse caso é bem improvável, isso está sendo desconsiderado, embora possa ocorrer na prática.

Por fim, se o algoritmo selecionado foi por programação dinâmica, a função *solveDynamicProgramming*, da biblioteca *SolveDynamicProgramming.h* é executada. Como abordado na seção *Programação Dinâmica*, é feita uma iteração sobre o todos os elementos do vetor. Como também mencionado, a relação de recorrência pode ser resolvida em $O(n^2)$, mas com a implementação de uma busca binária essa complexidade se reduziu para $O(n \cdot \log n)$. O vetor de memorização é alocado dinamicamente, e portanto a alocação e desalocação deste é realizada em $O(1)$. Assim, o algoritmo por programação dinâmica possui complexidade temporal total de $O(n \cdot \log n)$.

3.2 Complexidade Espacial

Todos os algoritmos compartilham da mesma instância de entrada, a qual é definida pela variável do tipo *Instance*, que contém um vetor de *pairs*. Esse vetor armazena os pares $< bar, owner >$ fornecidos pela entrada, o que ocupa um total de n posições, e portanto a complexidade espacial resultante é $O(n)$. No algoritmo força bruta, a altura da árvore resultante é no máximo n, e portanto a complexidade espacial também é $O(n)$. Para o algoritmo guloso e por programação dinâmica, é utilizada uma lista e um vetor de memorização, respectivamente. Como ambas as estruturas armazenam no máximo n itens, dadas as caracteristicas apresentadas em suas respectivas seções neste documento, a complexidade espacial de ambos também é $O(n)$. Com isso, temos que a complexidade espacial total desta implementação é $O(n)$.

4 Análise Experimental

Para analisar o comportamento dos algoritmos desenvolvidos, foi implementado um gerador de testes para o pior caso, em que a única configuração válida possível é uma única conexão, principalmente para o algoritmo por força bruta, de modo a forçar todas as análises possíveis na árvore de recursão. Os resultados obtidos para os três algoritmos são apresentados na sequência.

A Figura 4 ilustra o comportamento assintótico analisado experimentalmente para o pior caso do algoritmo por força bruta. Como esperado, o comportamento foi essencialmente exponencial, uma vez que, por natureza do algoritmo, todas as possibilidades são testadas.

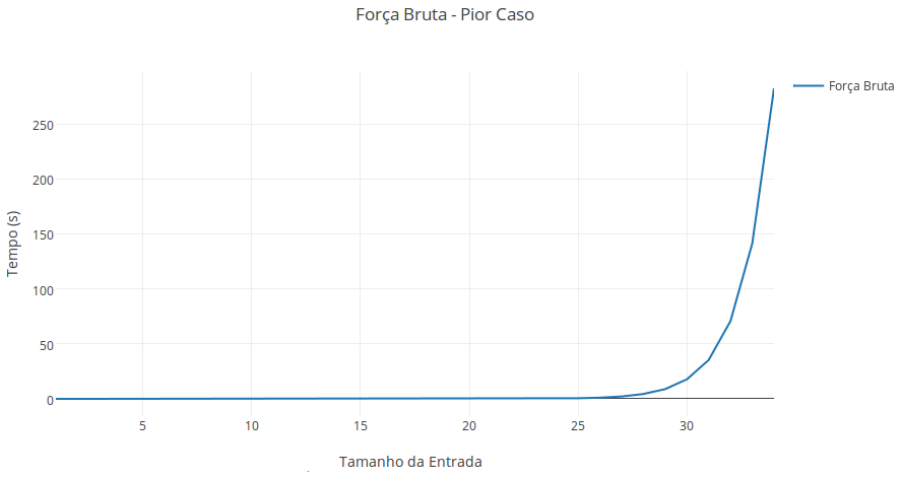

Figura 4 – Análise experimental do algoritmo por força bruta

.

A Figura 5, por sua vez, ilustra o comportamento assintótico do algoritmo guloso e por programação dinâmica. Como é perceptível no gráfico, o resultado de ambos condiz com as análises de complexidade avaliadas anteriormente, pois as curvas são bem próximas de $n \cdot \log n$. Neste experimento, a variação de tamanho dos testes executados foi até 10^6, comprovando a eficiência de ambos os algoritmos. Em contraste com a Figura 4, que apresenta o comportamento assintótico do algoritmo por força bruta, percebe-se como a programação dinâmica e a estratégia gulosa foram bastante eficientes, diminuindo drasticamente a complexidade e o tempo de execução. Um ponto interessante sobre o gráfico da Figura 5 está na comparação da curva gerada pela programação dinâmica e aquela gerada pelo algoritmo guloso. Embora ambos possuam complexidade temporal $O(n \cdot \log n)$, o algoritmo por programação dinâmica é mais rápido do que o guloso. Isso ocorre porque no método *Patience Sort*, a busca binária é sempre executada para definir em qual pilha o elemento atual da comparação deve entrar. Já no algoritmo por programação dinâmica essa busca só é executada quando as duas seguintes condições *não* são atendidas: (1) $A[i] < DP[0]$ e (2) $A[i] > DP[length - 1]$, onde *length* é o tamanho da maior subsequência crescente encontrada até o momento. Ou seja, em apenas um de três casos a busca binária é executada, o que contribui significativamente para o desempenho do algoritmo por programação dinâmica para entradas grandes. Por fim, vale ressaltar também que tanto a programação dinâmica quanto o algoritmo guloso implemetado garantem a otimalidade da solução.

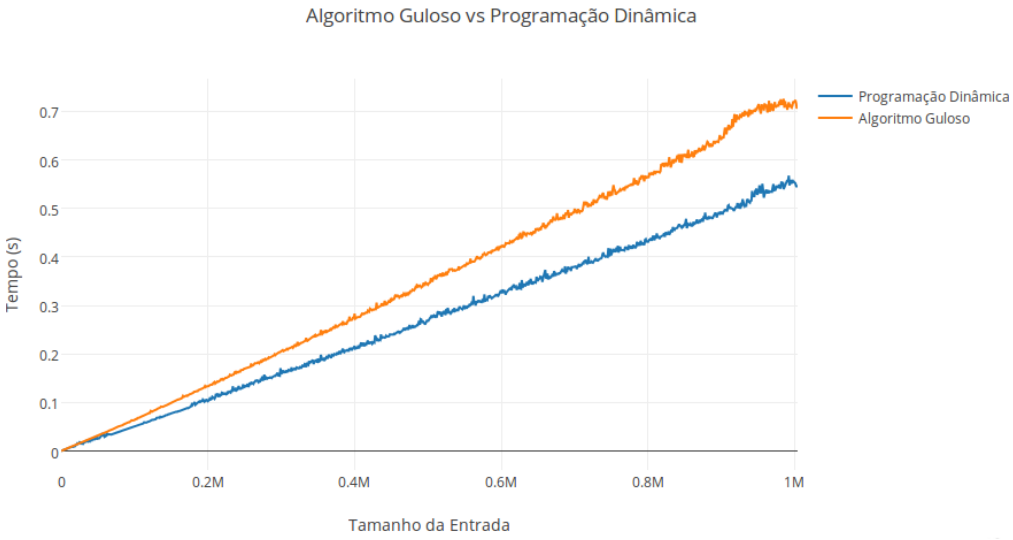

Figura 5 – Comparação de desempenho entre o algoritmo guloso e por programação dinâmica

.

5 Conclusão

Este trabalho apresentou uma abordagem por três paradigmas de programação distintos para a resolução do problema de encontrar o maior número possível de bandeirolas que podem ser conectadas de um lado da rua ao outro sem que haja nenhum cruzamento. Para tanto, foi desenvolvido um algoritmo por força bruta, com estratégia gulosa e por programação dinâmica. Com isso, ficou claro o quanto a escolha de um determinado paradigma pode influenciar na solução final de um problema, e como técnicas mais elaboradas de memorização podem ser bastante vantajosas em problemas de otimização. Além disso, um resultado interessante é o ponto do algoritmo guloso, que, implementado através do método *Patience Sort*, também fornece uma solução ótima para o problema, dadas as características do jogo e a estratégia gulosa por trás disso.

6 Referências Bibliográficas

Aldous, D.; Diaconis, P. Longest Increasing Subsequences: from Patience Sorting to the Baik-Deift-Johansson Theorem. Department of Statistics, University of California at Berkeley, and Departments of Mathematics and Statistics, Stanford University. March 31, 1999.

Cormen, Thomas H., and Thomas H. Cormen. Introduction to Algorithms. Cambridge, Mass: MIT Press, 2001.

Guide to Dynamic Programming. CS161, Summer 2013. Acesso em: 09 jul 2017. Disponível em: <https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/handouts/140 Guide to Dynamic Programming.pdf>.

Longest Increasing Subsequences. MIT Handouts, LIS – Chapter 33. Acesso em: 09 jul 2017. Disponível em: <http://web.mit.edu/18.338/www/2012s/handouts/LIS.pdf>.

ZIVIANI, Nivio. Projeto de Algoritmos com implementações em PASCAL e C, 3 ed. Cengage Learning, 2011.