Projeto e Análise de Algoritmos
Trabalho Prático 1 - Paradigmas

Vagner Clementino[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

`vagnercs@dcc.ufmg.br`

1. Introdução

Apesar da crença, um cérebro maior não indica maior inteligência. Um contraexemplo bastante conhecido é Albert Einsten, cujo cérebro possuía um volume que não era maior do que o da média dos indivíduos. Com a melhoria das imagens de ressonância magnética (IRM), diversos estudos vêm sendo proposto com o objetivo de correlacionar o *volume* do cérebro com o *Quociente de Inteligência* (QI). Um estudo utilizando imagens concluiu que a correlação entre QI e o volume do cérebro é consistente, todavia, as correlações são fracas, e não há como comprovar uma relação direta. Em síntese: ser um *"cabeção"* não é indicativo de inteligência.

A fim de refutar de vez esta crença, este trabalho se propõem em analisar os dados de *peso* e *QI* de um determinada população, com o objetivo de encontrar o maior número de pessoas cujo o peso do seu cérebro é menor, contudo, com um QI maior. O problema será formalmente definido na Seção 2.

Este documento está estruturado como segue. A Seção 2 define formalmente o problema, fazendo uma relação do mesmo com o problema da *Longest increasing subsequence*; a Seção 3 apresenta as três soluções propostas para resolver o problema: FORÇA BRUTA (subseção 3.1), GULOSA (subseção 3.2) e PROGRAMAÇÃO DINÂMICA (subseção 3.3); a Seção 4 faz uma discussão sobre resultados os experimentais de cada solução; a Seção 5 conclui o trabalho.

2. Definição Formal do Problema

O problema pode ser definido formalmente da seguinte forma. Dados $C = \langle c_1, c_2, \ldots, c_n \rangle$ um conjunto de indivíduos de tamanho n, bem como $P = \langle p_1, p_2, \ldots, p_n \rangle$ e $Q = \langle q_1, q_2, \ldots, q_n \rangle$, onde p_i e q_i representa, respectivamente, o peso e o QI do i-ésimo indivíduo. O problema consiste em encontrar o subconjunto $S \subseteq C$ tal que para todo $s_i, s_j \in S$ onde $i < j$ $p_i < p_j$ **e** $q_i > q_j$. Além disso, $|S|$ deve ser *máximo*.

O problema proposto neste trabalho pode ser facilmente mapeado para um bem conhecido problema de otimização denominado *Longest Increasing Subsequence* - *LIS*[Knuth 2013]. Para tanto, basta ordenar um dos conjuntos P ou Q de forma crescente ou decrescente. Por exemplo, caso o conjunto P seja ordenado de forma decrescente, o problema se transforma em encontrar a maior *LIS* no novo conjunto Q' que foi gerado pela ordenação de P.[1]

[1]Considera-se que para toda posição i em P e Q represente os dados do mesmo indivíduo

O LIS é um problema bastante estudado e existem diversas abordagens na literatura para resolvê-lo. Este trabalho utilizou algumas destas referências para desenvolver as soluções propostas na Seção 3.

3. Soluções propostas

Neste trabalho foi proposto três soluções para o problema utilizando os paradigmas Força Bruta, Gulosa e Programação Dinâmica. Para cada paradigma discute-se como foi realizada a modelagem, o funcionamento do algoritmo proposto e a análise da complexidade de tempo e de espaço. A descrição dos algoritmos será feita em mais alto nível por meio de pseudocódigo.

3.1. Força Bruta

3.1.1. Modelagem

O paradigma de Força Bruta é o único que garantidamente encontra uma solução ótima para qualquer problema computável. Contudo, o preço que se paga por esta aplicabilidade universal é o alto custo de tempo e/ou espaço necessário. A principal característica de uma abordagem Força Bruta é que ela faz uma *busca integral no espaço de soluções* [Kleinberg and Tardos 2005]. Neste sentido, ao desenvolvermos um algoritmo força bruta, ele deverá listar todas as possíveis soluções para posteriormente definir a melhor entre as soluções válidas.

No contexto do problema estudado, o espaço de soluções consiste de todas as permutações dos elementos do *power set* de C, cuja a notação é dada por 2^C. Desta forma, o algoritmo a ser proposto deverá ser capaz de criar cada permutação de tamanho $1, 2, \ldots, n$. Para cada permutação/solução criada deverá ser verificado se a solução é válida a fim de encontrar a melhor entre aquelas que são válidas. Na próxima subseção descreveremos o algoritmo proposto.

3.1.2. O Algoritmo Força Bruta

O algoritmo 1 apresenta em alto nível a abordagem Força Bruta utilizada. Por meio do método GENERATE-ALL-SOLUTIONS todas as possível soluções são geradas. A partir delas a solução ótima (de maior tamanho) é encontrada e posteriormente retornada S_{best}. O método IS-VALID verifica se uma dada solução é valida verificando se cada par de item respeita as restrições do problema.

3.1.3. Análise de Complexidade

Conforme exposto anteriormente, na abordagem de Força Bruta é realizada uma busca em cada item do espaço de soluções do problema. Na subseção 3.1.1 discutiu-se que este espaço de solução $O(2^n)$. Neste contexto, o método responsável por varrer o espaço de soluções necessariamente terá sua complexidade igual $O(2^n)$. No algoritmo 1 este trabalho é realizado pelo método GENERATE-ALL-SOLUTIONS. Como as demais funções do algoritmo possuem complexidade inferiores, podemos concluir

Algorithm 1: BRUTE-FORCE encontra a solução ótima listando todas elas.

Input: As sequências finitas $C = \langle c_1, c_2, \ldots, c_n \rangle$, $P = \langle p_1, p_2, \ldots, p_n \rangle$ e $Q = \langle q_1, q_2, \ldots, q_n \rangle$

Output: Uma sequência $S_{best} \subseteq C$ tal que atende as restrições do problema e seja máxima

```
U ← GENERATE-ALL-SOLUTIONS(C)
max ← 0
S_best ← ∅
for each S in U do
    if IS-VALID(S) then
        if S.legth() > max then
            max ← S.legth()
            S_best ← S
return S_best
```

que o algoritmo de força bruta proposto possui ondem de complexidade de tempo igual a $O(2^n)$.

No tocante a complexidade de espaço, o algoritmo necessita carregar os conjuntos C, P e Q a fim de poder gerar todas as soluções e verificar se elas são válidas. Apesar de no pior caso existir $O(2^n)$ possíveis soluções, o sistema apenas armazena a melhor solução encontrada no momento. Partindo desta estratégia teremos um custo de espaço igual a $O(1)$. Neste sentido, o algoritmo têm uma complexidade de espaço igual a $O(n)$.

3.2. Uma abordagem Gulosa

Apesar do algoritmo de força bruta resolver o problema, conforme poderá ser observado na Seção 4 onde descreve a análise experimental dos algoritmos, tal abordagem torna-se impraticável quando o tamanho da entrada cresce. Com o objetivo de encontrar uma solução de melhor desempenho, esta seção descreve uma solução baseada no paradigma *Guloso*.

3.2.1. Modelagem

Um problema é passível de ser resolvido utilizando a abordagem Gulosa caso ele possua a propriedade da *subestrutura ótima*. Um problema é dito ter subestrutura ótima se uma solução ótima pode ser construído de forma eficiente a partir de soluções ótimas de seus subproblemas [Cormen et al. 2009]. Vamos provar que o problema em questão possui subestrutura ótima.

Iniciemos, sem perda de generalidade, ordenando o conjunto de entrada C de forma decrescente pelo seu conjunto de pesos P. Conforme exposto na Seção 2 ao realizarmos a ordenação da entrada conforme proposto, o problema transforma-se em encontrar uma *LIS* no novo conjunto de QI's Q' gerado.

Seja S_{ij} a maior LIS entre os elementos i e j do conjunto $Q^{'}$, conforme descrito no parágrafo anterior, tal que $1 \leq i \leq (n-1)$ e $i < j$. Desta forma, S_{1n} é a solução ótima para o problema. Suponha que escolhamos um valor k qualquer de modo que $i \leq k < j$. As soluções S_{ik} e $S_{(K+1)j}$ *são ótimas* o que pode ser provado por absurdo. Suponha sem perda de generalidade que exista uma solução S'_{ik} tal $|S'_{ik}| > |S_{ik}|$, a existência de S'_{ik} vai contra a suposição de que S_{ij} seja ótima.

Após provado que o problema apresenta subestrutura ótima já é possível propor um algoritmo guloso para o problema. A solução proposta é detalhada na subseção 3.2.2.

3.2.2. O Algoritmo Guloso

O *Patience Sorting* é um algoritmo de ordenação baseado no jogo de cartas de mesmo nome. O funcionamento básico do jogo é descrito a seguir:

1. Inicialmente, não há pilhas. A primeira carta formará uma nova pilha de tamanho 1.
2. A cada nova carta retirada do baralho, caso ela seja menor que alguma carta no topo de qualquer uma das pilhas disponíveis, a carta será colocada no topo da pilha; caso contrário uma nova pilha será criada.
3. Quando não há mais cartas no baralho, o jogo termina.

A Figura 1 exibe o resultado da aplicação do Patience Sorting em um conjunto de números. Conforme pode ser observado ao final do algoritmo cada pilha terá uma *sequência decrescente*. Utilizando esta propriedade do algoritmo foi realizada uma versão modificada do algoritmo para resolver o problema proposto neste trabalho. Basicamente realiza-se a ordenação do conjunto P de forma crescente; a partir do novo conjunto $Q^{'}$ resultante aplica-se uma versão gulosa do Patience Sorting de que modo que ao final a maior pilha gerada será a solução do problema. O Algoritmo apresenta de forma genérica a solução proposta.

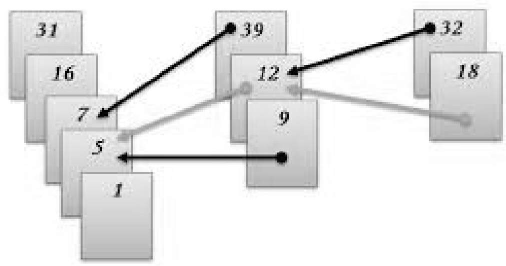

Figura 1. O resultado da aplicação Patience Sorting

Conforme exposto o algoritmo 2 é uma modificação gulosa do Patience Sorting. A parte gulosa do algoritmo está no método `GREEDY-CHOICE` (c_i, q_i) que consiste basicamente em buscar a pilha cujo elemento no topo seja maior do que q_i. Caso exista mais de uma pilha candidata será *retornada aquela cujo o elemento no topo seja o maior*.

Algorithm 2: GREEDY ENCONTRA A SOLUÇÃO DE FORMA GULOSA.

Input: As sequências finitas $C = \langle c_1, c_2, \ldots, c_n \rangle$, $P = \langle p_1, p_2, \ldots, p_n \rangle$ e $Q = \langle q_1, q_2, \ldots, q_n \rangle$

Output: Uma sequência $S_{best} \subseteq C$ tal que atende as restrições do problema e seja máxima

```
INITIALIZE o conjunto de n pilhas S_1, S_2, ..., S_n
SORT (C) //Ordenando pelos pesos
maxLength ← 0
bestStack ← 0
for i to n do
    j ← GREEDY-CHOICE (c_i, q_i)
    PUSH (S_j, c_i)
    if |S_j| > maxLength then
        maxLength ← |S_j|
        bestStack ← j
return S_bestStack
```

Não obstante, cabe um questionamento se a solução gulosa aqui proposta leva à solução ótima. Seja G uma solução de tamanho k obtida utilizado o algoritmo guloso ora proposto. Se G não for ótima, existe uma solução $\mathcal{O}$ de tamanho m tal que $m > k$. Suponha ainda, sem perda de generalidade, que em ambas as soluções houve a ordenação dos pesos, cabendo analisar apenas a maior lista decrescente de QI's. Como $m > k$ temos que existe um cérebro o_{j+1} em $\mathcal{O}$ tal que é menor do que o cérebro g_j em G[2]. Contudo, o algoritmo 2 realiza a interação sobre *todos* os cérebros e ao final sempre retorna a *maior pilha*. Logo, não é possível existir uma sequência decrescente maior do que G, logo a solução é ótima.

3.2.3. Análise de Complexidade

Como pode ser observado no Algoritmo 2, a solução gulosa proposta faz uma iteração sobre todos os elementos de entrada. Para cada entrada ele faz uma chamada para o método `GREEDY-CHOICE` (c_i, q_i). Este método realiza uma *busca sequencial* para encontrar a pilha no qual o item será inserido, desta forma a complexidade do `GREEDY-CHOICE` é $O(n)$. Tendo em vista que o método é chamando n vezes podemos concluir que a complexidade do algoritmo guloso é $O(n^2)$. Cabe ressaltar que existe a possibilidade de melhoria do algoritmo proposto ao implementar o método `GREEDY-CHOICE` através de busca binária. Neste caso o algoritmo teria a complexidade de $O(nlogn)$.

No que tange ao espaço utilizado o algoritmo necessitará armazenar os conjuntos C, P e Q. Além disso, no pior caso, será necessário criar um total de n pilhas. Todavia, mesmo no pior caso, a ordem de complexidade de espaço será $O(n)$, ou seja, proporcional ao tamanho da entrada.

[2] O índice j representa a posição do cérebro na solução

3.3. Programação Dinâmica

A *Programação Dinâmica* é aplicável em problemas que otimização que apresentem as seguintes características: (i) *subestrutura ótima* e (ii) *sobreposição de subproblemas*. A substrutura ótima do problema foi provado na subseção 3.2.1. Provaremos na próxima subseção que existe sobreposição entre os subproblemas.

3.3.1. Sobreposição de Subproblemas

De forma análoga as outras soluções propostas neste trabalho, iniciemos que a entrada foi ordenada pelos pesos de forma crescente. Neste sentido, o problema se resume em encontrar uma *LIS* para o novo conjunto $Q^{'}$ de QI's gerado. Seja $LIS(i)$ a maior subsequencia crescente que termina no elemento i do conjunto $Q^{'}$. $L(i)$ pode ser definido recursivamente conforme Equação 1.

$$LIS(x) = \begin{cases} 1 + MAX(LIS(j)) \text{ onde } j < i \text{ e } q_j < q_i \\ 1 \text{ caso contrário} \end{cases} \tag{1}$$

Para encontramos uma LIS em um conjunto de tamanho n devemos encontrar o $MAX(L(1), L(2), \ldots, L(n)$. A figura 2 apresenta a árvore de recursão para o cálculo da LIS para a entrada $A = \langle 1, 2, 3, 4 \rangle$. Como pode ser observado existem sobreposições dos subproblemas.

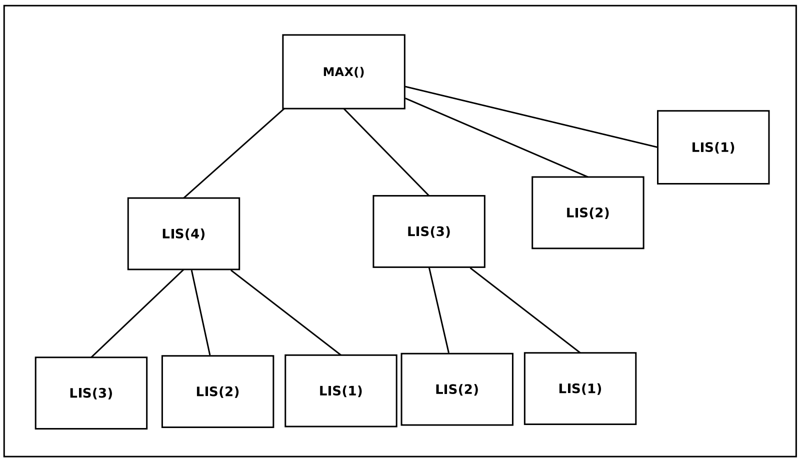

Figura 2. Arvore de recursão do LIS para a entrada A

3.3.2. O Algoritmo de Programação Dinâmica

O algoritmo via Programação Dinâmica proposto utilizou a abordagem de cima para baixo com memoização, ou seja, realiza o calculo para subproblemas menores de modo que estas soluções sejam aproveitadas por subproblemas maiores. O Algoritmo 3 exibe o código proposto:

O vetor MEMO armazena os resultados dos subproblemas a fim de serem utilizados pelos subproblemas menores. Para cada i é armazenado no vetor PREV

Algorithm 3: PROGRAMAÇÃO DINÂMICA DE CIMA PARA BAIXO COM MEMOIZAÇÃO

Input: As sequências finitas $C = \langle c_1, c_2, \ldots, c_n \rangle$, $P = \langle p_1, p_2, \ldots, p_n \rangle$ e $Q = \langle q_1, q_2, \ldots, q_n \rangle$ e n como tamanho da entrada

Output: Uma sequência $S_{best} \subseteq C$ tal que atende as restrições do problema e seja máxima

```
SORT (C) //Ordenando pelos pesos
maxLength ← 0
bestEnd ← 0
INITIALIZE MEMO com o valor 0 //MEMO é vetor que memoria o resultado de soluções menores
INITIALIZE PREV com o valor −1 //PREV armazena os índices da melhor solução
for i ← 1 to (n − 1) do
    for j ← (i − 1) to 0 do
        if q_j < q_i and MEMO[i] < MEMO[j] then
            MEMO[i] ← MEMO[j] + 1
            PREV[i] ← j
    if  bestEnd = i
    maxLength = memo[i]
    then MEMO[i] > maxLength
S ← BUILD-SOLUTION (PREV, n)
return S
```

Tabela 1. Tempo de execução do algoritmo de força bruta

Tamanho Entrada	Tempo Execução (s)
5	1
9	1620
10	3618
50	7250

os índices das maiores LIS encontrada de modo a solução ser construída através do método BUILD-SOLUTION.

3.3.3. Análise da Complexidade

Analisando o pseudocódigo do Algoritmo 3 verificamos que ele possui dois loop aninhados. Dentro do loop mais interno apenas existem operações de complexidade $O(1)$. Desta forma, o algoritmo possui complexidade igual a $O(n^2)$. No que tange ao espaço utilizado, o algoritmo utiliza de dois vetores para armazenarem os dados de soluções menores e dos índices da LIS. Neste sentido a sua complexidade de espaço é da ordem de $O(n)$. O algoritmo poderia ser otimizado em loop interno de modo a encontrar os valores armazenados em $O(log(n))$ resultado em uma complexidade igual à $O(nlogn)$

4. Análise Experimental

Com o objetivo de verificar a desempenho real das soluções propostas foi realizada uma bateria de testes. Os testes consistem em executar os algoritmos em conjunto de entradas de diferentes tamanhos com a medição dos respetivos tempos de execução. Os testes foram executados em computador com o sistema operacional Ubuntu versão 12.04 kernel 3.13.0-37-generic 64 bits e com 4GB de memória RAM. Os tempos de executam consistem da média de cinco execuções para cada entrada.

A bateria de testes foi executada prioritariamente nos algoritmos guloso e programação dinâmica tendo em vista que para algumas entradas o algoritmo de força bruta não foi possível executar. Os tamanho de entrada para o qual o força bruta executou são exibidos na Tabela 4.

Os tamanhos de entrada (eixo x) e tempo de execução (eixo y) são exibidos na Figura 3. Como pode ser observado o tempo de execução de ambos os algoritmos são similares para entradas menores. Quando o tamanho da entrada cresce verifica-se que o algoritmo guloso é mais eficiente. Esta diferença pode ser devido ao fato do algoritmo guloso executar em média um número menor de operações que seu concorrente.

A figura 4 exibe o tamanho da solução encontrada para uma determinada entrada. Naturalmente o numero de elementos da solução é proporcional ao tamanho da entrada, todavia, é possível verificar que é cada mais difícil encontrar indivíduos que atendam aos requisitos do problema. Por exemplo para uma entrada de tamanho 50, existem 18% de indivíduos que atendem ao requisito do problema, todavia, para uma entrada de tamanho 1000 o percentual cai para 3%.

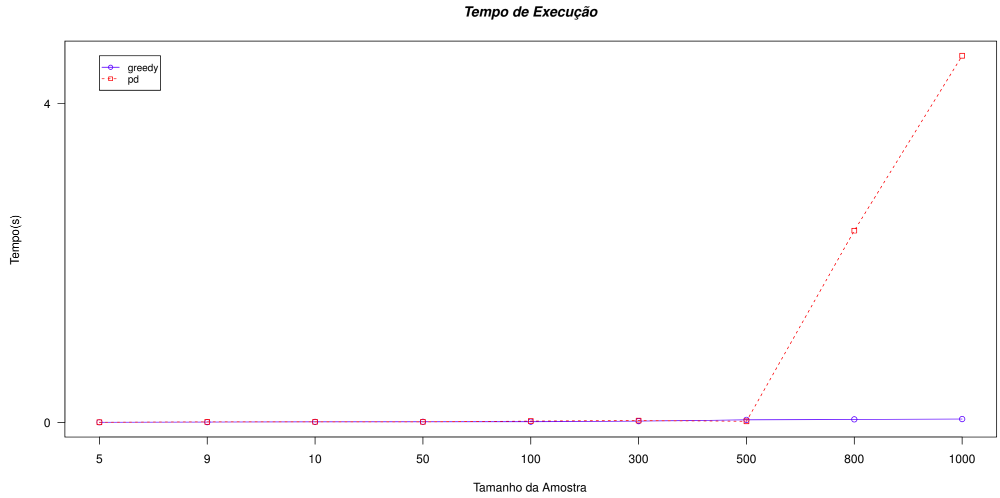

Figura 3. Tempos de execução Greedy vs PG

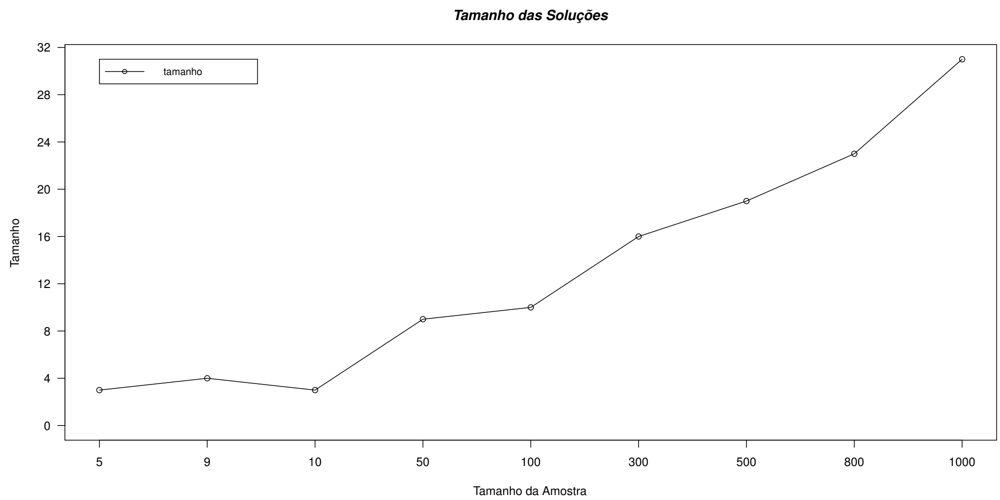

Figura 4. Tempos de execução Greedy vs PG

5. Conclusões

Este trabalho se propôs a resolver o problema de encontrar o maior número de pessoas em determinada população cujo peso do cérebro seja menor, contudo, apresentando um maior QI. Afim de resolver o problema foram propostos três algoritmos utilizando os paradigmas Força Bruta, Guloso e Programação Dinâmica. O algoritmo de Força Bruta, apesar de garantidamente retornar a solução ótima, mostrou-se inutilizável quando o tamanho das entradas foram crescendo. Para os paradigmas Guloso e Programação Dinâmica provou-se que o problema atendia aos requisitos de subestrutura ótima e sobreposição de subproblemas (necessário à Programação Dinâmica). Foi possível mostrar que os algoritmos resultam em soluções ótimas. A partir deste trabalho foi possível verificar a aplicação dos diferentes paradigmas de programação em um mesmo problema e como as especificidades de cada paradigma levam as diferentes soluções.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition.

Kleinberg, J. and Tardos, E. (2005). *Algorithm Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Knuth, D. (2013). *Art of Computer Programming, Volume 4, Fascicle 4,The: Generating All Trees–History of Combinatorial Generation.* Pearson Education.