

Capítulo 3

Técnicas de Projeto de Algoritmos

Elton J. Silva (eltonsilv@gmail.com)
Departamento de Computação
Universidade Federal de Ouro Preto

*“One of the most important aspects of algorithm design is
creating an algorithm that has an efficient run time”*

As técnicas simples de Análise Estruturada ou de Análise Orientada a Objetos aliadas a um pouco de bom senso e conhecimento sobre Estruturas de Dados geralmente são suficientes para que possamos elaborar algoritmos para resolver uma grande parte dos problemas do cotidiano.

Exemplos:

- Criar um sistema de contabilidade para uma empresa.
- Criar um programa que gerencie um estoque de um depósito.

Entretanto, existem alguns problemas onde o caminho a ser seguido para a solução não é tão direto. São problemas onde os algoritmos para resolvê-los ou parte deles não são tão óbvios.

Estes são problemas onde é recomendável aplicar **Técnicas de Projeto de Algoritmos** mais elaboradas para resolvê-los.

Exemplos:

- Criar um algoritmo eficiente para ordenar 500.000 endereços.
- Criar um algoritmo eficiente para distribuir tarefas entre máquinas de uma empresa de maneira a reduzir o tempo de ociosidade dessas máquinas.
- Criar um algoritmo eficiente para compactar arquivos com determinadas características.
- Criar um algoritmo de roteamento aproximado para uma empresa de entrega de pacotes.

Para a solução de problemas assim, existem conjuntos de técnicas que dividem os problemas em **classes** e apresentam **linhas gerais** para o projeto de um algoritmo. Essas técnicas frequentemente produzem algoritmos efetivos para resolver uma grande classe de problemas. Neste capítulo apresentamos algumas dessas técnicas mais importantes, tais como divisão e conquista, método guloso, programação dinâmica, backtracking e branch-and-bound. Ao tentar desenvolver um algoritmo para resolver um determinado problema é importante fazer a seguinte pergunta: *“Que tipo de solução o dividir-e-conquistar, programação dinâmica e estratégia gulosa, ou alguma outra técnica-padrão produz?”*.

Entretanto, devemos enfatizar que existem alguns tipos de problemas, tais como os NP-Completo, para os quais os métodos aqui abordados também não produzem soluções eficientes. Quando nos deparamos com problemas dessa natureza é frequentemente mais proveitoso determinarmos se as entradas para o problema possuem características especiais que possam ser exploradas na tentativa de desenvolver uma solução, ou se uma solução aproximada pode ser usada ao invés de uma solução exata mais difícil de computar.

1. Método *Dividir e Conquistar*

Divisão e Conquista (DC) é talvez uma das técnicas de projeto mais aplicadas. Consiste em dividir sucessivamente o problema original em problemas menores até que uma solução simples exista, de forma que a combinação simples destas soluções parciais forme a solução completa do problema.

A técnica de divisão e conquista consiste de 3 passos básicos:

1. **Divisão:** Dividir o problema original, em subproblemas menores.
2. **Conquista:** Resolver cada subproblema recursivamente.
3. **Combinação:** Combinar as soluções encontradas, compondo uma solução para o problema original.

Um algoritmo de “divisão e conquista” é normalmente relacionado a uma equação de recorrência que contém termos referentes ao próprio problema:

$$T(n) = aT(n/b) + f(n),$$

onde a indica o número de sub-problemas gerados, n/b está associado ao tamanho de cada um deles e $f(n)$ o custo para fazer a divisão e combinação das sub-soluções em uma solução única.

Uma das vantagens da estratégia de Dividir-e-Conquistar é que os algoritmos produzidos em geral são facilmente paralelizáveis. Se existem vários processadores disponíveis, a estratégia propicia eficiência.

1.1. DC em alto nível

Algoritmo DC(P)

```
{
  if small(P) then return S(P)
  //P é pequeno o suficiente para ser resolvido de forma que não
  precisará ser subdividido
  else {
    divide P em {P1, P2, ..., Pk} k>1
    //Pi é do mesmo tipo do problema original P
    return (combine (DC(P1), DC(P2), ... , DC(Pk)))
    //aplica DC recursivamente a cada um dos subproblemas gerados
    //combina as sub-soluções dos Pis para a solução de P
  }
}
```

Dessa forma, podemos definir uma relação de recorrência genérica para os algoritmos que seguem a estratégia Dividir-e-Conquistar:

$$T(n) = S(n), n \leq \text{smallsize}$$

$$T(n) = \left(\sum_{i=1}^k T(n_i) \right) + D(n) + C(n), \text{ onde } n_i \approx \text{tamanho}(P_i)$$

A técnica de divisão e conquista pode ser aplicada a uma variedade de problemas já estudados, entre eles o de busca em tabelas, com o algoritmo de *busca binária*, e o de classificação, com os algoritmos de *mergesort* e o *quicksort*. Para ilustrar melhor a técnica de DC, nas seções 1.2 e 1.3 vamos apresentar mais dois problemas (*Máximo e Mínimo* e *Multiplicação de Polinômios*) que também podem ser resolvidos com a técnica.

1.2. Máximo e Mínimo

O problema do MaxMin consiste em achar o máximo e/ou o mínimo de um conjunto de n elementos. O algoritmo padrão para resolver o problema (StraightMaxMin) é apresentado a seguir:

```
Algoritmo StraightMaxMin (A, n, max, min)
{
    max = min = A[1];
    for (i: 2 to n)
        if (A[i] > max) max = A[i];
        else if (A[i] < min) min = a[i];
}
```

- No algoritmo acima, o melhor caso ocorre quando os elementos estão em ordem crescente. O número de comparações é $n-1$.
- O pior caso ocorre quando os elementos estão em ordem decrescente. Neste caso o número de comparações é $2(n-1)$.

Podemos utilizar a seguinte técnica de Divisão e Conquista para resolver o problema:

- $P = (n, a[i], \dots, a[j])$, onde n é o número de elementos na lista $a[i], \dots, a[j]$, a qual estamos querendo encontrar o máximo e o mínimo.
- Small(P) retorna *true* quando $n \leq 2$. Neste caso, max e min são $a[i]$ se $n=1$. Se $n=2$, o problema pode ser resolvido fazendo apenas uma comparação.
- Se a lista tiver mais de dois elementos, P será dividido em instâncias menores. Após ter dividido P em dois subproblemas, podemos invocar o mesmo algoritmo recursivamente.

O algoritmo utilizando Divisão e Conquista para o problema MaxMin pode ser da seguinte forma:

```

Algoritmo MaxMin (i, j, max, min)
{
    if (i = j) max = min = A[i] // Small(P): 1 elemento
    else
        if (i = j - 1) // Small(P): 2 elementos
            if (A[i] < A[j]) {
                max = A[j];
                min = A[i];
            }
            else {
                max = A[i];
                min = A[j];
            }
        else {
            // se Small(P) = false, divide P em subproblemas.
            meio = ⌊(i + j)/2⌋;
            // resolve os subproblemas
            MaxMin (i, meio, max, min);
            MaxMin (meio+1, j, max1, min1);
            // Combina as soluções
            if (max < max1) max = max1;
            if (min > min1) min = min1;
        }
}

```

Relação de recorrência para o algoritmo MaxMin:

$$\begin{aligned}
 T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & , n > 2 \\
 T(n) = 1 & , n = 2 \\
 T(n) = 0 & , n = 1
 \end{aligned}$$

Resolvendo a relação de recorrência para $n=2^k$, temos:

$$\begin{aligned}
 T(n) &= 2T(n/2) + 2 \\
 &= 4T(n/4) + 4 + 2 \\
 &\vdots \\
 &= 2^{k-1} + 2^k - 2 = (3n/2) - 2
 \end{aligned}$$

Assim, conseguimos melhorar em 25% o número de comparações com o algoritmo convencional ($2n-2$). Podemos então dizer que o algoritmo MaxMin é melhor que o algoritmo StraightMaxMin? Talvez não na prática, pois o algoritmo MaxMin requer um custo adicional para a administração da sua pilha de recursão (armazenamento das variáveis i , j , max , min , $max1$ e $min1$). Dados n elementos, ocorrerão $\lfloor \lg n \rfloor + 1$ níveis de recursão e será necessário salvar esses 6 valores e o endereço de retorno para cada chamada recursiva.

1.3. Multiplicação de Polinômios

Operações de multiplicação geralmente são caras, mas aplicadas a um grande número de situações:

- Multiplicação de polinômios (ex.: Computação Gráfica)
- Multiplicação de matrizes (ex. operações de rotação e translação)
- Multiplicação de inteiros longos (ex.: Criptografia, representação de números inteiros grandes).

Existe um esforço de pesquisadores para tentar reduzir a complexidade desses problemas.

Nesta seção, apresentamos o problema da Multiplicação de Polinômios e como a técnica de divisão e conquista é aplicada para reduzir a complexidade do mesmo.

Seja $a_0 + a_1x + a_2x^2 + \dots + a_nx^{n-1}$ um polinômio de grau $n-1$, com n termos. Sejam dois polinômios $p(x)$ e $q(x)$ com n termos, um algoritmo para resolver o problema da multiplicação de $p(x)$ por $q(x)$ deve retornar um outro polinômio $r(x)=p(x)*q(x)$.

1ª tentativa: algoritmo "força bruta" (ingênuo)

Algoritmo mult_polinomio

```
{
  for (i:0 to 2n-2)      } O(n)
    r(i)=0;
  for (i:0 to n-1) //p(x)
    for (j:0 to n-1) //q(x)
      r(i+j) = r(i+j) + p(i) * q(j) } O(n^2)
}
```

Exemplo:

$$p(x) = 2x+3$$

$$q(x) = x^2 + 3x + 5$$

$$p(x)*q(x) = (2x^3 + 6x^2 + 10x) + (3x^2 + 9x + 15) = 2x^3 + 9x^2 + 17x + 15$$

É fácil perceber que o algoritmo mult_polinomio, usando a técnica de "força-bruta" é $O(n^2)$.

2ª tentativa: usar a idéia de convolução

Vamos enxergar um polinômio $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^{n-1}$ da seguinte forma:

$$p(x) = p_e(x) + p_d(x) \cdot x^{n/2}, \quad n = \text{número de termos.}$$

Para simplificar a análise, vamos considerar $n = 2^k$.

$$\text{Exemplo: } (5 + 3x) + (8x^2 - 7x^3) = \underbrace{(5 + 3x)}_{p_e} + \underbrace{(8 - 7x)}_{p_d}x^2$$

$$r(x) = p(x) * q(x) = (p_e(x) + p_d(x) \cdot x^{n/2}) * (q_e(x) + q_d(x) \cdot x^{n/2})$$

$$= r_e + r_m \cdot x^{1/2} + r_d \cdot x^n$$

\nwarrow
 $p_e * q_e$

\downarrow
 $p_e * q_d + p_d * q_e$

\searrow
 $p_d * q_d$

onde p_e, p_d, q_e, q_d têm grau $(n/2)-1$

Exemplo: $p(x) * q(x)$

$$(5 + 3x + 8x^2 - 7x^3) * (1 - 7x - 3x^2 + 4x^3)$$

$$[(5+3x) + (8-7x)x^2] * [(-1+7x) + (-3+4x)x^2]$$

$$(5+3x) * (1-7x) + [(5+3x) * (-3+4x) + (8-7x) * (-1+7x)]x^2 + [(8-7x) * (-3+4x)]x^4$$

$$p_e * q_e$$

$$p_e * q_d$$

$$p_d * q_e$$

$$p_d * q_d$$

A relação de recorrência do algoritmo que usa a idéia de convolução é:

$$T(n) = 4T(n/2) + \Theta(n)$$

$$\Theta(n) = \text{esforço p/ soma dos termos}$$

$$T(0) = 0$$

Resolvendo a relação de recorrência, encontramos $T(n) = O(n^2)$, mesma complexidade do algoritmo força bruta visto anteriormente. Será que dá para conseguir algo melhor?

3ª tentativa: usar a idéia de área=produto

$$\text{Vimos que } r(x) = (p_e * q_e) + [(p_e * q_d) + (p_d * q_e)]x^{n/2} + (p_d * q_d)x^n$$

Visualmente temos:

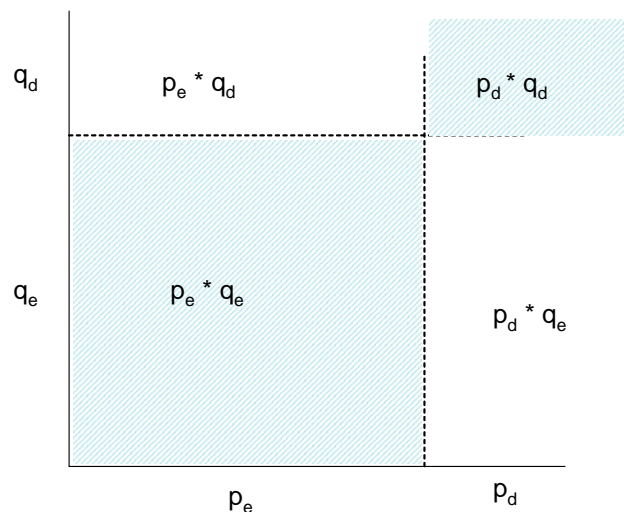


Figura 1 – Multiplicação de $p(x)$ por $q(x)$ vista como um cálculo de área

A idéia na Figura 1 é substituir as duas multiplicações de $p_e * q_d$ e $p_d * q_e$ por uma única multiplicação $(p_e + p_d) * (q_e + q_d)$ e subtrair desse valor as áreas de $p_e * q_e$ e $p_d * q_d$, já calculadas.

$$(p_e * q_d) + (p_d * q_e) \Leftrightarrow (p_e + p_d) * (q_e + q_d) - ((p_e * q_e) + (p_d * q_d))$$

$$r_e = p_e * q_e \quad (\text{subproblema 1})$$

$$r_d = p_d * q_d \quad (\text{subproblema 2})$$

$$r_{\text{aux}} = (p_e + p_d) * (q_e + q_d) \quad (\text{subproblema 3})$$

$$r_m = r_{\text{aux}} - (r_e + r_d)$$

Quebramos um problema maior em 3 subproblemas menores da mesma natureza. Dessa forma temos a seguinte relação de recorrência:

$$\begin{aligned} T(n) &= 3T(n/2) + \Theta(n) \\ T(0) &= 0 \end{aligned}$$

$$\text{Resolvendo, temos: } T(n) = O(n^{\lg 3}), T(n) = O(n^{1,58})$$

A seguir, apresentamos uma versão inicial do algoritmo para multiplicação de polinômios que utiliza as idéias discutidas nesta seção:

```

Algoritmo multiplica_polinomio(int n1, n2; vetor p, q): vetor;
{
  se (n1 = 1) ou (n2 = 1) // multiplicação simples
    return r
  senão { // divide em subproblemas menores
    encontre k; //ponto de divisão dos 2 vetores: "metade" do menor
    pe = subvetor (p, 1, k);
    pd = subvetor (p, k+1, n1);
    qe = subvetor (q, 1, k);
    qd = subvetor (q, k+1, n2);
    re = multiplica_polinomio (k, k, pe, qe);
    rd = multiplica_polinomio (n1-k, n2-k, pd, qd);
    raux = multiplica_polinomio (n1-k, n2-k, pe+pd, qe+qd);
    rm = raux - (re + rd);
    r = re + rm + rd; //combina resultados parciais
    return r;
  }
}

```

É possível melhorar ainda mais? Existe uma solução $O(n \log n)$ que envolve o uso de um algoritmo FFT (*Fast Fourier transform*) como subrotina.

1.4. Considerações finais sobre o DC

- **Top down** (divide um problema grande em problemas menores);
- **Recorrente** (os algoritmos são recursivos e a relação de recorrência pode ser facilmente extraída do código).
- **Balanceamento** é a chave de sucesso (relação entre a quantidade e tamanho dos subproblemas e o esforço para resolvê-los). Se bem aplicada, a técnica DC geralmente produz algoritmos ótimos.

2. Método Guloso (Greedy)

O Método Guloso (MG), ou ganancioso, é uma estratégia para resolver problemas de otimização, sempre realizando a escolha que parece ser a melhor no momento;

fazendo uma escolha ótima local, na esperança de que esta escolha leve até a solução ótima global.

Em cada iteração, o algoritmo faz uma escolha gulosa sobre a maneira de aumentar a solução parcial até então construída. A escolha feita é sempre da opção que leve a menor contribuição ao custo total (em caso de minimização) ou a maior contribuição ao valor total (em caso de maximização) segundo algum critério local à solução parcial.

2.1. Guloso em alto nível

```
Algoritmo guloso (A, n);  
// A[1..n] contém n entradas  
{ solução = {}  
  for i=1 to n do  
  {  
    x = SELECT(A);  
    if VIÁVEL(solução, x)  
      solução = UNIÃO (solução, x)  
  }  
  return solução  
}
```

Existem duas versões da técnica gulosa: uma mais apropriada para resolver problemas que se encaixam no chamado *paradigma de subconjunto* e outra mais apropriada para problemas que se encaixam no *paradigma de ordem*.

O paradigma de subconjuntos está relacionado a problemas que envolvem a identificação de um subconjunto de elementos do conjunto original. Alguns exemplos de problemas nesse paradigma são:

- *Knapsack Problem* (Problema da Mochila)
- *Tree Vertex Splitting*
- *Job Sequencing with Deadlines* (Sequenciamento de Jobs)
- *Minimum Cost Spanning Trees* (Árvore Geradora Mínima ou Árvore de Espalhamento Mínima)

O paradigma de ordem não usa o SELECT para a escolha do elemento a cada passo, pois as decisões são tomadas considerando-se as entradas em uma determinada ordem. Alguns exemplos de problemas nesse paradigma são:

- *Optimal Storage on Tapes* (Armazenamento Ótimo em Fitas)
- *Optimal Merge Patterns* (Código de Huffman)
- *Single-Source Shortest Paths* (Caminho mínimo em grafos)

Nas seções 2.2 e 2.3 a seguir são apresentados dois problemas resolvidos em cada um desses paradigmas: o problema da mochila, que se encaixa no paradigma de subconjuntos, e o problema do armazenamento ótimo em fitas, no paradigma de ordem.

2.2. Problema da Mochila (*Knapsack Problem*)

O **problema da mochila** é um problema de otimização combinatória. O nome é dado devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.

Dados:

- n objetos e uma mochila
- objeto i tem peso w_i e lucro p_i
- mochila tem capacidade m
- Se uma fração x_i , $0 \leq x_i \leq 1$ do objeto i é colocada na mochila então há um ganho $p_i \cdot x_i$

O objetivo do problema é obter um valor máximo na mochila (maximizar o ganho). O peso total de todos os objetos escolhidos deve ser menor ou igual a m .

Mais formalmente, o problema da mochila fracionária pode ser definido da seguinte forma:

$$\text{Maximizar } \sum_{1 \leq i \leq n} p_i \cdot x_i \text{ (função objetivo)}$$

$$\text{sujeita a: } \left(\sum_{1 \leq i \leq n} w_i \cdot x_i \right) \leq m$$

$$0 \leq x_i \leq 1,$$

$$1 \leq i \leq n,$$

$$w_i, p_i > 0$$

Exemplo:

Seja o seguinte conjunto de objetos com os respectivos ganhos e pesos:

Objeto	p_i (profit)	w_i (weight)	$d_i (p_i/w_i)$
1	25	18	1,39
2	24	15	1,6
3	15	10	1,5

$m = 20$ (capacidade máxima da mochila)

Guloso1: gula pelo ganho (colocar na mochila primeiro os objetos que dão maior lucro). Ganho total = $25 \cdot 1 + 24 \cdot (2/15) = 28,2$

Guloso2: gula pelo peso (colocar na mochila primeiro os objetos que possuem menor peso). Ganho total = $15 \cdot 1 + (10/15) \cdot 24 = 31,0$

Guloso3: gula pela "densidade". Ganho total = $24 \cdot 1 + (5/10) \cdot 15 = 31,5$

Para este problema, a estratégia gulosa 3 gera a solução ótima.

A seguir, apresentamos um algoritmo para o problema da mochila, baseado no algoritmo geral da seção 2.1:

```

algoritmo mochila (m, n);
// p[1..n] é o vetor de ganhos
// w[1..n] é o vetor de pesos
// m é a capacidade da mochila
// x[1..n] é o vetor com a solução
{
    ordenar p e w de forma que p[i]/w[i] • p[i+1]/w[i+1], i:1..n-1
    for i=1 to n do
        x[i] = 0.0;
    U := m //U é o peso residual da mochila
    for i=1 to n do
    {
        if (w[i]>U) break;
        x[i] = 1.0
        U = U-w[i];
    }
    if (i • n) x[i]=U/w[i];
}

```

É fácil verificar que a complexidade do algoritmo é $O(n \lg n)$, dada pela ordenação inicial dos elementos seguida pelo FOR que faz a distribuição dos itens na mochila.

2.3. Problema do Armazenamento Ótimo em Fitas

Seja $E=(l_1, l_2, \dots, l_n)$ um vetor com comprimentos de n arquivos a serem armazenados em uma fita suficientemente extensa. $S=(j, l_j)$, $j: 1, 2, \dots, n$, é a ordem de armazenamento dos arquivos na fita, de forma a minimizar o tempo médio de recuperação (TMR) dos arquivos na fita. Neste problema existe a seguinte restrição: quando um arquivo vai ser recuperado da fita, a mesma é reposicionada no início.

$$TMR = \frac{1}{n} \sum_{k=1}^n \sum_{j=1}^k l_j$$

Exemplo: $E=(7, 3, 5, 2)$, $n=4$

Se os arquivos forem armazenados nessa ordem (7, 3, 5, 2), o $TMR = 49/4$, como pode ser observado na Figura 2.

7	3	5	2		...	
7						
7	3					
7	3	5				
7	3	5	2			
<hr/>						
$7 \times 4 + 3 \times 3 + 5 \times 2 + 2 \times 1 = 49$						

Figura 2 – Uma possível distribuição de arquivos na fita

Se os arquivos forem armazenados na ordem (7, 3, 2, 5), o $TMR = 46/4$.

É fácil perceber que uma solução gulosa que minimiza o TMR armazena os arquivos em ordem crescente de tamanhos. Para o exemplo, se os arquivos forem armazenados na ordem (2, 3, 5, 7) o TMR é igual a 34/4.

Teorema: Se $l_1 \leq l_2 \leq \dots \leq l_n$, então a ordem $i_j = j$, $1 \leq j \leq n$ minimiza $\sum_{k=1}^n \sum_{j=1}^k l_{i_j}$ para todas as possíveis permutações de i_j .

Demonstração: a idéia básica é mostrar que toda seqüência não ordenada pode ser melhorada. Logo, a seqüência ordenada é a melhor (ótima).

$$\sum_{k=1}^n \sum_{j=1}^k l_{i_j} = n.l_1 + (n-1).l_2 + \dots + 1.l_n$$

Em uma solução diferente da gulosa apresentada, existe j tal que $l_{i_j} > l_{i_{j+1}}$. Então pode-se inverter as posições de l_{i_j} e $l_{i_{j+1}}$, reduzindo o valor de $\sum_{k=1}^n \sum_{j=1}^k l_{i_j}$. Essa idéia pode ser aplicada a toda a seqüência não ordenada, ou seja, toda seqüência não ordenada pode ser melhorada (invertendo posições). Logo, o ótimo é a seqüência ordenada.

Um algoritmo para o armazenamento ótimo em fitas poderia ser escrito da seguinte forma:

```

Algoritmo store(n);
// n é o número de arquivos
{
    ordena os arquivos pelo tamanho;
    for i:1 to n do
        Append arquivo i na fita
    }

```

A complexidade do algoritmo store é $O(n \lg n)$.

2.4. Matróides

Dizemos que a estratégia gulosa é correta quando ela encontra a solução ótima. Mas quando é que podemos garantir que o guloso é correto? Será que existe alguma propriedade intrínseca ao problema que garante o funcionamento correto do guloso?

Se no problema conseguirmos identificar a estrutura algébrica de MATRÓIDE então podemos garantir que existe solução gulosa correta para o problema. Vamos definir então o que é um Matróide, mas antes precisamos do conceito de um Sistema de Subconjuntos:

Seja $S = (M, \mathfrak{I})$ um sistema de subconjuntos. Para isto, as seguintes propriedades devem ser observadas:

- (i) $|M| < \infty$
- (ii) \mathfrak{I} é um conjunto de subconjuntos independentes de M

(iii) $I \in \mathfrak{I}$ e $J \subset I$, então $J \in \mathfrak{I}$ (fechado por inclusão)

Um Sistema de Subconjuntos S é um Matróide se ele possuir uma das seguintes propriedades:

(i) **Aumento de Base**

Se I_p e I_{p+1} , conjuntos independentes pertencentes a \mathfrak{I} , e $|I_p| < |I_{p+1}|$

então: $\exists e \in (I_{p+1} - I_p)$ tal que $I_p \cup \{e\} \in \mathfrak{I}$.

(ii) **Dimensão Única**

$\forall A \subset M, I, I' \in \mathfrak{I}, I \text{ e } I' \subset A \text{ e maximais, então } |I| = |I'|$.

2.5. Considerações finais sobre o MG

- O guloso encontra somente UMA solução ótima (construída incrementalmente). O problema pode ter várias soluções ótimas, mas o guloso só encontra uma. Cuidado: é errado dizer que "o guloso só encontra soluções ótimas", pois como vimos a otimalidade nem sempre é garantida.
- Algoritmos gulosos geralmente são simples e de fácil implementação. Também são eficientes e a análise de complexidade é trivial (uma ordenação seguida de um loop $O(n)$, em geral).
- A grande dificuldade do MG consiste em, uma vez encontrada uma solução, verificar se ela é ótima (provar a corretude).
- Os algoritmos gulosos têm a sua eficiência computacional fortemente dependente do procedimento usado para testar a viabilidade da extensão gulosa das soluções parciais obtidas passo a passo.
- Todo algoritmo guloso baseado na escolha sucessiva de elementos, por comparação entre seus valores, tem tempo de computação $\Omega(n \lg n)$, que é o tempo mínimo para ordenar as escolhas.
- Os algoritmos gulosos geralmente são iterativos, ao contrário dos algoritmos baseados na DC, que são geralmente recursivos.

3. Programação Dinâmica

A estratégia da Programação Dinâmica (PD), ou Planejamento Sequencial, costuma ser aplicada a certos problemas de otimização, resultando geralmente em algoritmos mais eficientes que os triviais, pois a cada passo são eliminadas subsoluções que certamente não farão parte da solução ótima do problema.

A diferença entre programação dinâmica e método guloso é que no método guloso somente uma sequência de decisão é gerada e na programação dinâmica muitas seqüências podem ser geradas.

A PD é especialmente útil quando não é fácil chegar a uma seqüência ótima de decisões sem testar todas seqüências possíveis para então escolher a melhor.

A idéia básica da programação dinâmica é construir por etapas uma resposta ótima combinando respostas já obtidas para partes menores.

- Inicialmente, a entrada é decomposta em partes mínimas, para as quais são obtidas respostas.
- Em cada passo, sub-resultados são combinados dando respostas para partes maiores, até que se obtenha uma resposta para o problema original.
- A decomposição é feita uma única vez e, além disso, os casos menores são tratados antes dos maiores.

Assim, esse método é chamado ascendente, ao contrário dos métodos recursivos, que são chamados descendentes.

O diagrama da Figura 3 ilustra a estrutura geral dessa abordagem:

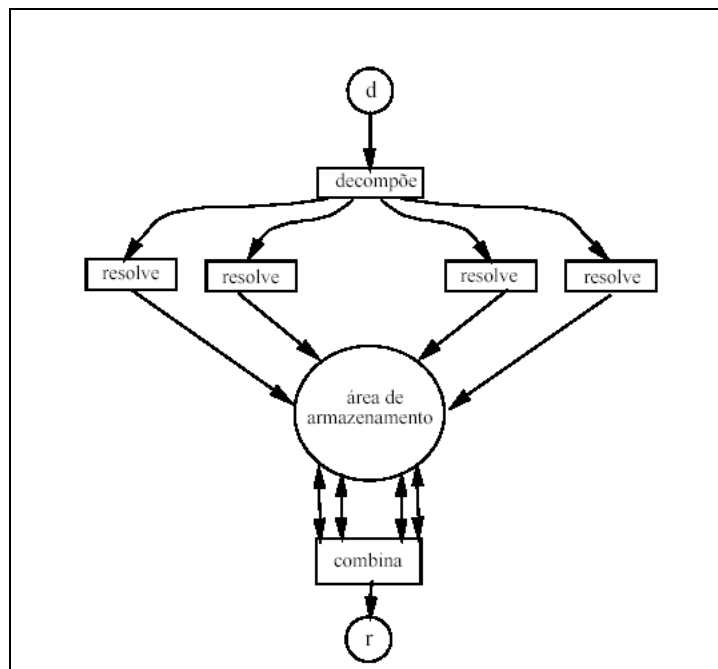


Figura 3 – Visão geral da Programação Dinâmica

Esta estratégia reduz drasticamente o número total de seqüências viáveis através de um mecanismo que evita aquelas que sabidamente não podem resultar

ótimas. Este mecanismo é conhecido como PRINCÍPIO DA OTIMALIDADE, e pode ser enunciado da seguinte forma: *“Toda solução parcial de uma solução ótima também é (localmente) ótima.* Isso significa que somente soluções ótimas dos subproblemas podem compor uma solução ótima do problema.

Na procura de uma solução ótima, as soluções não ótimas de subproblemas podem ser descartadas, e é esse o processo que torna os algoritmos desenvolvidos por programação dinâmica mais eficientes que os algoritmos diretos.

Além disso, os subproblemas são resolvidos uma única vez, e suas soluções são guardadas para serem usadas tantas vezes quantas for necessário.

3.1. PD em alto nível

A idéia básica da programação dinâmica sugere uma estrutura geral para algoritmos projetados por essa estratégia:

Inicialização; Iteração; Finalização.

Na *inicialização*, a entrada é decomposta em partes mínimas para as quais são obtidas respostas diretas.

A *iteração* vai aumentando o tamanho das partes e obtendo respostas correspondentes a partir das já geradas, até que as partes atinjam o tamanho da entrada original quando então sua solução é recuperada, e o processo finalizado.

Assim, chega-se ao seguinte esquema geral de algoritmo de programação dinâmica.

```
Algoritmo_PD(d: D): R //Algoritmo de programação dinâmica (abstrato)
// Entrada-Saída: saída r:R é resposta ótima para entrada d:D
{ Incz_PD ; Iter_PD ; Fnl_PD; } //estrutura geral
onde
    Incz_PD: //inicialização
        entrada é decomposta em partes mínimas;
        partes mínimas dão diretamente partes da saída
    Iter_PD : //iteração
        escolhe elemento “e” da parte da entrada ;
        combina parte da entrada com elemento “e” ;
        atualiza parte da saída com elemento “e” ;
    Fnl_PD : //finalização
        extrai a saída final para a entrada original
```

A programação dinâmica pode ser aplicada a uma variedade de problemas de otimização combinatória, como por exemplo:

- multiplicação de matrizes;
- problema do menor caminho entre dois nós;
- problema de alocação de recursos;
- problema da reposição de equipamentos;

- problema da edição de strings;
- problema do caixeiro viajante;
- problema da mochila.

Esses e outros problemas podem ser formulados como **grafos multiestágio** e têm solução por PD.

3.2. Grafos multiestágio

Um grafo multiestágio é um grafo:

- $G = (V, E)$ com V particionado em $k \geq 2$ subconjuntos disjuntos de forma que se (a, b) está em E então a está em V_i , e b está em V_{i+1} , para alguns subconjuntos na partição;
- e $|V_i| = |V_k| = 1$.

O vértice s em V_1 é chamado de vértice fonte; o vértice t em V_k é chamado de vértice destino.

Em outras palavras:

- Os vértices de G estão em camadas ou estágios.
- Os arcos saem de um estágio para um outro imediatamente seguinte.
- Não existem arcos entre vértices do mesmo estágio. Obs.: se um grafo tem arestas para vértices em um mesmo estágio então ele não é multiestágio, e sim dirigido acíclico.

3.3. Problema do menor caminho

Para o grafo da Figura 4, qual o menor caminho para ir do vértice 1 ao vértice 12?

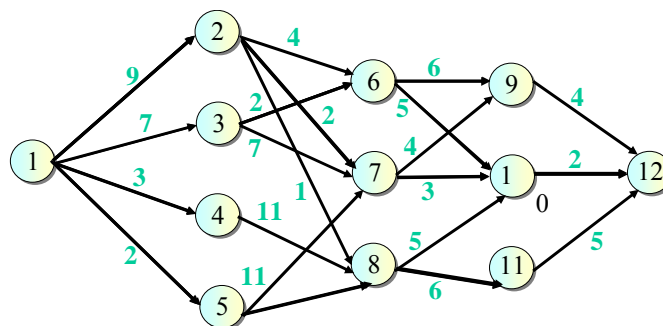


Figura 4 – Problema do caminho mínimo

Uma forma de enxergar a solução do problema utilizando Divisão e Conquista (seção 1) é a seguinte:

$$\text{ótimo}(1, 12) = \min \begin{cases} 2 : 9 + \text{ótimo}(2, 12) \\ 3 : 7 + \text{ótimo}(3, 12) \\ 4 : 3 + \text{ótimo}(4, 12) \\ 5 : 2 + \text{ótimo}(5, 12) \end{cases}$$

$$\text{ótimo}(2,12) = \min \begin{cases} 6:4 + \text{ótimo}(6,12) \\ 7:2 + \text{ótimo}(7,12) \\ 8:1 + \text{ótimo}(8,12) \end{cases}$$

$$\text{ótimo}(3,12) = \min \begin{cases} 6:2 + \text{ótimo}(6,12) \\ 7:7 + \text{ótimo}(7,12) \end{cases}$$

...

É fácil perceber que esta solução é ineficiente pela quantidade de chamadas recursivas repetidas que são geradas (ex.: $\text{ótimo}(6,12)$ e $\text{ótimo}(7,12)$).

A formulação em PD para o problema do caminho mínimo é obtida através da percepção de que o menor caminho do vértice 1 ao vértice 12 é o resultado de uma sequência de decisões. A idéia na PD é sequenciar os diversos subproblemas de forma iterativa.

A i -ésima decisão envolve determinar qual vértice no estágio V_{i+1} vai estar no caminho ($1 \leq i \leq k-2$, k é o número de estágios).

A solução correta é garantida pelo Princípio da Otimalidade: "toda subtrajetória da trajetória ótima é ótima com relação a suas extremidades inicial e final".

Repare que duas estratégias diferentes podem ser utilizadas na geração do caminho mínimo: forward (gera o caminho do vértice origem para o vértice destino) e backward (gera o caminho do vértice destino para o vértice origem). A seguir, apresentamos um algoritmo baseado em PD para o problema do caminho mínimo, que usa a estratégia *forward*.

```
int[] MStageForward(Graph G)
{
    // returns vector of vertices to follow through the graph
    // let c[i][j] be the cost matrix of G

    int n = G.n (number of nodes);
    int k = G.k (number of stages);
    float[] C = new float[n];
    int[] D = new int[n];
    int[] P = new int[k];
    for (i = 1 to n) C[i] = 0.0;
    for j = n-1 to 1 by -1 {
        r = vertex such that (j,r) in G.E and c(j,r)+C(r) is minimum
        C[j] = c(j,r)+C(r);
        D[j] = r;
    }
    P[1] = 1; P[k] = n;
    for j = 2 to k-1 {
        P[j] = D[P[j-1]];
    }
    return P;
}
```

- Se G for representado por listas de adjacências, r pode ser encontrado em tempo proporcional ao grau do vértice j .
- Se G tem $|E|$ arestas, o loop for $j=n-1$ to 1 é $O(|V|+|E|)$.
- O for $j=2$ to $k-1$ é $O(k)$.

- Logo, MStageForward é $O(|V| + |E|)$. Para grafos completos, o algoritmo é $O(n^2)$.

Na Figura 5 a seguir, temos o resultado do algoritmo MStageForward aplicado ao grafo da Figura 4.

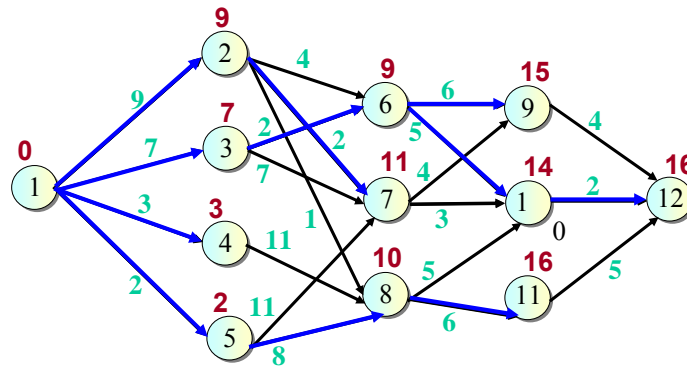


Figura 5 – Solução *forward* para o grafo da Figura 4

3.4. Problema de alocação de recursos

O problema de alocação de recursos consiste em dado um conjunto de filiais de uma empresa e os respectivos retornos de investimento de cada uma, decidir como dividir uma quantidade de dinheiro entre essas filiais, de forma a maximizar o lucro da empresa.

Por exemplo, como dividir 8 milhões entre quatro filiais de forma a maximizar o lucro de uma empresa? Os retornos de investimento de cada filial são dados pela tabela a seguir:

x	$r_1(x)$	$r_2(x)$	$r_3(x)$	$r_4(x)$
0	0	0	0	0
1	3	1	2	1
2	7	2	4	3
3	10	4	6	6
4	12	8	8	9
5	13	13	10	12
6	14	17	12	14
7	14	19	14	16
8	14	20	16	17

Solução 1: guloso (máximo lucro) $r_2(8) = 20$

mas, $r_1(1) + r_2(7) = 22$ (ainda não é o ótimo)

Solução 2: Formulação matemática da função objetivo, utilizando PD.

$$V_k(x) = \max\{r_k(y) + V_{k+1}(x-y)\}$$

$$0 \leq y \leq x$$

3 Retorno de investimento
se tomo a decisão por y

p_i O que sobra
para as outras

Investimentos

P = preço de uma máquina nova

$c(i)$ = custo de operação anual (manutenção)

$t(i)$ = reembolso na troca

$s(i)$ = valor no ano n

i = idade da máquina

idade i	0	1	2	3	4	5	6
$c(i)$	10	13	20	40	70	100	100
$t(i)$	-	32	21	11	5	0	0
$s(i)$	-	25	17	8	4	0	0

O problema da reposição de equipamentos consiste em determinar a política de reposição (troca ou manutenção) que minimiza a despesa total.

Formulação matemática da função objetivo:

despesa

$$V_n(i) = \min$$

↓

Recursão de cauda:
transformar em
iteração

{

máquina nova retorno na troca custo de operação da máquina nova No início do ano seguinte a máquina tem idade 1

Compra: $p - t(i) + c(0) + V_{n+1}(1)$

custo de manutenção máquina um ano mais velha

Manutenção: $c(i) + V_{n+1}(i+1)$

Neste problema, o grafo multiestágio está representado implicitamente, onde os nós são representados pelos anos e as arestas pelas decisões de manter/comprar.

3.6. Modelagem de Problemas com Programação Dinâmica

- Identificar **decisões e estágios**
- Identificar o **critério de otimização**
- Construir a **função critério**
 - Identificar o **critério de otimalidade**
 - Para cada decisão relacionar recursivamente os valores do critério definindo as variáveis de estado necessárias.

- Considerando as variáveis de estado, fornecer condições de contorno para a função critério.
- Determinar a **política ótima** para cada estágio e estado.

3.7. Considerações finais sobre a PD

- Os algoritmos PD geralmente têm complexidade polinomial $O(n^2)$, $O(n^3)$.
- A análise de complexidade dos algoritmos é fácil.
- A prova de corretude dos algoritmos é simples: baseada no Princípio da Otimalidade.
- A principal dificuldade está na modelagem adequada para o problema (escolha das decisões, critérios, estados, estágios, políticas, condições de contorno etc).
- As implementações recursivas são ineficientes por causa das chamadas repetidas. A solução é eliminar a recursão gerando soluções iterativas.

4. Backtracking

Backtracking é um método para realizar busca exaustiva, isto é, avaliar todas as possíveis soluções de um problema, através da "tentativa e erro".

O método, sistematicamente analisa diferentes caminhos que conduzam a uma solução válida. Alternativas que não satisfaçam as condições do problema são eliminadas.

Depois de analisar um caminho sem sucesso, retorna-se a posições anteriores tentando encontrar uma solução usando outros caminhos.

O método usa, implicitamente, uma árvore para organizar as soluções do problema.

A estratégia do backtracking pode ser aplicada a diversos problemas, como por exemplo:

- Problema das n rainhas;
- Problema do passeio do cavaleiro
- Soma de Subconjuntos
- Mochila Binária
- Problema do Labirinto
- Coloração de grafos

4.1. Estrutura Geral de algoritmos de Backtracking

```
Procedure Tenta
{
  Inicialize seleção de candidatos;
  repeat
    selecione o próximo candidato;
    if aceitável then
    {
      gravar;
      if solução incompleta then
      {
        Tenta próximo passo;
        {Chame Tenta recursivamente com
         parâmetros relativos ao próximo
         passo}
        if mal-sucedido then
          cancele gravação
      }
    }
  until (bem-sucedido) or
    (não há mais candidatos)
};
```

Se a cada passo o número de candidatos a ser investigado é fixo, temos:

```

Procedure Tenta (i: integer);
var k: integer;
{
  k := 0; {Inicialize seleção de candidatos}
  repeat
    k := k+1;
    selecione o k-ésimo candidato;
    if aceitável then
    {
      gravar;
      if i < n then {n = solução completa}
      {
        Tenta (i + 1);
        if mal-sucedido then
          cancele gravação
      }
    }
  until (bem-sucedido) or (k = m)
};

```

Sobre a complexidade:

Os fatores determinantes do esforço para determinação de uma resposta utilizando backtracking são: geração do próximo nó, fator de ramificação e cálculo da função de fronteira.

- **Melhor caso:** Ok, para problemas pequenos;
- **Média:** razoável, estimada empiricamente via simulação (Monte Carlo): roleta que sorteia trajetórias;
- **Pior caso:** exponencial

4.2. Problema do Passeio do Cavaleiro



Problema do Passeio do Cavaleiro (*Knight's Tour Problem*)

Dado um tabuleiro 8x8, o cavalo deve se mover cobrindo todas as casas. O percurso (se existir) deve ser feito de forma que todas as casas sejam visitadas uma única vez.

```

Algoritmo TentaProxMov(); {versão inicial}
{
  Inicialize seleção de movimentos;
  repeat
    selecione o próximo candidato;
    if aceitável then
      {movimento cai no tabuleiro e casa vazia}
      {

```

```

    registra movimento;
    if (tabuleiro não cheio) then
    {
        TentaProxMov( );
        if mal-sucedido then
            apague movimento
    }
}
until (bem-sucedido) or (não há candidatos)
};

```

```

Algoritmo TentaProxMov(i:int; x,y:indice; var q: boolean);
{
    k:=0;
    repeat
        k:=k+1; q1:=false;
        (u,v):=coordenadas do próximo movimento;
        if ( $1 \leq u \leq n$ ) and ( $1 \leq v \leq n$ ) and ( $tab[u,v]=0$ ) then
        {
            tab[u,v]:=i;
            if ( $i < \sqrt{n}$ ) then
            {
                TentaProxMov(i+1, u, v, q1);
                if not q1 then
                    tab[u,v]:=0;
            }
            else q1:=true
        }
    until (q1) or (k=8);
    q:=q1
};

```

4.3. Problema das n rainhas



Problema das 8 rainhas (*Eight Queen's Problem*)

Posicionar 8 rainhas em um tabuleiro de xadrez tal que nenhuma rainha ataque qualquer outra.

```

Algoritmo TentaProx( ); {versão inicial}
{
    selecione posição para i-ésima rainha;
    repeat
        selecione a próxima posição;
        if aceitável then {posição segura}
        {
            posicionar rainha;
            if i<8 then
            {

```

```

    TentaProx(i+1);
    if mal-sucedido then
        remover rainha
    }
}
until (bem-sucedido) or (não há posições)
};

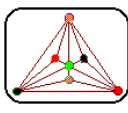
```

```

Algoritmo TentaProx(i:int; var q: boolean; var j: int);
{
    j:=0;
    repeat
        j:=j+1; q:=false;
        if (linha[j] and diagesq[i+j] and
            diagdir[i-j]) then {
            posicao[i]:=j; linha[j]:=false;
            diagesq[i+j]:=false; diagdir[i-j]:=false;
            if i<8 then {
                TentaProx(i+1, q);
                if not q then {
                    linha[j]:=true;
                    diagesq[i+j]:=true;
                    diagdir[i-j]:=true;
                }
            }
            else q:=true;
        }
    until q or (j=8);
};

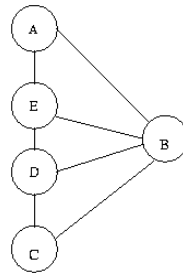
```

4.4. Problema da Coloração de Grafos

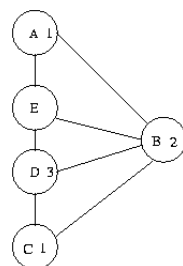
	<p>Problema da Coloração de Grafos $(x_1, x_2, \dots, x_n) \in C^n$ é uma coloração de $G \Leftrightarrow \{u, v\} \in E$ então $x_u \neq x_v$ Ou seja, queremos colorir os vértices de um grafo não direcional com n cores de maneira que cada vértice tenha uma cor diferente da cor de seus vizinhos.</p> <p>Dado um grafo, pode não haver solução para o problema, pode haver uma ou pode haver várias soluções.</p>
---	---

Exemplo:

Queremos colorir os vértices de um grafo não direcional com 3 cores de maneira que cada vértice tenha uma cor diferente da cor de seus vizinhos. Dado um grafo, pode não haver solução para o problema, pode haver uma ou pode haver várias soluções. Considere que vamos resolver o problema para o grafo a seguir:

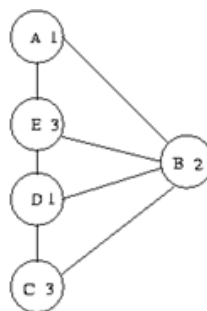


Tentando resolver o problema, poderíamos colocar a cor 1 no vértice A, a cor 2 no vértice B, a cor 1 no vértice C, a cor 3 no vértice D e então, não haveria nenhuma cor possível para colocar no vértice E.



Neste ponto então voltamos ao vértice D (mas a única cor possível para D seria 3 mesmo) e então, voltamos do vértice D ao vértice C, onde poderíamos ter escolhido a cor 3 (Backtracking).

Alteramos então nossa solução de cor para o vértice C que passa a ter cor 3. Neste ponto, a cor do vértice D ainda não havia sido escolhida, então vamos continuar a solucionar o problema a partir daqui. Somos então obrigados a escolher a cor 1 para o vértice D e a cor 3 para o vértice E.



Obs.: Número Cromático de um grafo G , $X(G)$, é o menor número de cores suficientes para gerar uma coloração de G . Resultado importante: G é planar $\Rightarrow X(G) \leq 4$.

4.5. Considerações finais do Backtracking

O backtracking trabalha com:

- **Geração de estados:** ao invés de representar o espaço, gera localmente o grafo.

- **Busca em profundidade:** caminha na árvore implícita através do caminhamento em profundidade que guarda o tempo todo o caminho percorrido.
- **Função de fronteira** (*bounding function*): descobre alguns pontos (fronteiras) a partir do qual a solução se torna inviável (exs.: 8-rainhas: teste de diagonal, passeio do cavaleiro: posição inválida, etc).

5. Branch and Bound (Ramificar e Podar)

O método *Branch&Bound* é parecido com o *backtracking*, entretanto, utiliza a “proximidade” da resposta para determinar o nó a expandir. A busca cega (por exemplo, busca em profundidade) não percebe a proximidade da solução.

Em relação ao algoritmo backtracking, o B&B requer dois itens adicionais:

- Uma maneira de estabelecer para cada nó da árvore espaço-estado um limite sobre o melhor valor da função objetiva sobre qualquer solução que possa ser obtida pela adição de mais componentes a solução parcial representada pelo nó.
- O valor da melhor solução encontrada até o momento

Se estas informações estiverem disponíveis, podemos comparar o valor limite de um nó com o valor da melhor solução vista até agora e se o valor do limite não for melhor, o nó não é promissor e pode ser podado, pois, nenhuma solução que o inclua levará a uma solução melhor que a solução já existente.

Em geral, o caminho de busca é encerrado quando:

- O valor do limite do nó não é melhor que o valor da solução obtida até agora;
- O nó representa uma solução não possível, pois as restrições dos problemas já foram violadas;
- O subconjunto das soluções possíveis representadas pelo nó consiste de um único ponto.

5.1. Branch&Bound em alto nível¹

```
listnode = record {
    listnode *next, *parent; float cost;
}
(1) algorithm B&B(t) //Least Cost Search: Search t for an answer node
(2) {
(3)   if *t is an answer node then output *t and return;
(4)   E := t; //E-node
(5)   Initialize the list of open nodes to be empty;
(6)   repeat
(7)     for each child x of E do {
(8)       if x is an answer node then output the path from x to t and return;
(9)       Add(x); //x is a new open node
(10)      (x->parent) := E; // Pointer for path to root
(11)    }
(12)    if there are no more open nodes then {
(13)      write ("No answer node"); return
(14)    }
(15)    E := Least() //finds a node with least f' in the list of open nodes
(16) until (false)
(17) }
```

¹ Em [Rich, 1993], o algoritmo de *Busca pela Melhor Escolha*, também conhecido como algoritmo A* é visto de forma mais detalhada, tratando dos casos de geração de nós repetidos.

Observação:

No caso do B&B aplicado a **problemas de otimização**, uma outra versão do algoritmo deve ser utilizada. Nessa nova versão, deve-se ir eliminando soluções parciais que sejam piores que soluções já encontradas.

A técnica Branch&Bound pode ser aplicada a vários tipos de problemas, como por exemplo:

- 15-Puzzle
- problema da mochila.
- Problema do caixeiro viajante

5.2. Escolha da Função Heurística

A determinação da proximidade da resposta para escolher o nó a expandir é feita através do cálculo de uma **função heurística** de classificação (*ranking function*), $f'(n) = g(n) + h'(n)$, calculada para cada nó gerado. $f'(n)$ é uma função que estima o custo de cada nó gerado. Isto permite que o algoritmo busque caminhos mais promissores antes. A função g representa o custo do menor caminho do nó inicial até o nó em questão. A função h' é uma estimativa do custo do caminho do nó corrente até um nodo associado ao estado final mais próximo (h' utiliza informações específicas sobre o domínio do problema). A função $f'(n) = g(n) + h'(n)$, então, representa uma estimativa do custo de sair do estado-inicial e chegar a um estado-meta através do caminho que gerou o nó corrente.

Na prática, quando o tamanho do espaço de estados torna inviável uma busca exaustiva, a especificação da função heurística h' , juntamente com a forma de representação do estado a partir da qual ela é calculada, são os pontos mais críticos no projeto de um sistema para solucionar um dado problema.

5.3. n-Puzzle

Estado inicial:

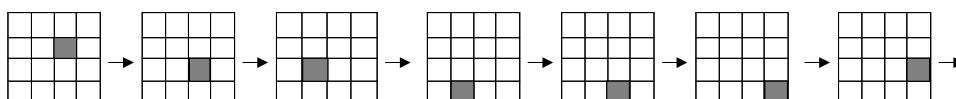
1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Estado final:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

A partir do estado inicial, consegue-se perceber que com apenas 3 movimentos (\uparrow peça 7, \leftarrow peça 11 e \uparrow peça 12), resolve-se o problema.

Já a busca em profundidade, pode gerar nós infinitamente na busca da solução.

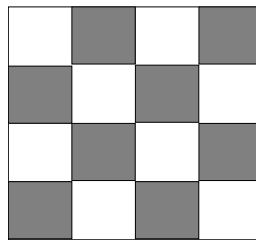


Neste problema, o *backtracking* não é recomendado, pois não temos o conceito de "beco sem saída". A partir de qualquer estado sempre se pode fazer um movimento.

Para o 15-puzzle, nem todo estado final pode ser alcançado a partir de uma determinada disposição inicial das peças. Em [Horowitz,1998], p. 383, é apresentado o seguinte teorema:

Para o problema do 15-puzzle, o estado final é alcançável a partir do estado inicial se e somente se $[\sum_{i=1}^{16} \text{menores}(i)] + x$ é par, onde:

- $\text{menores}(i)$ é o número de peças menores que a peça i que estão à sua frente. Obs.: 16 é considerado a peça do espaço vazio.
- $x=1$ se no estado inicial o espaço vazio estiver em uma das posições sombreadas da figura a seguir, e $x=0$ se o espaço vazio estiver em uma das posições restantes.



Em relação à escolha da função heurística, para o exemplo do puzzle, temos: $f'(n) = g(n) + h'(n)$, onde:

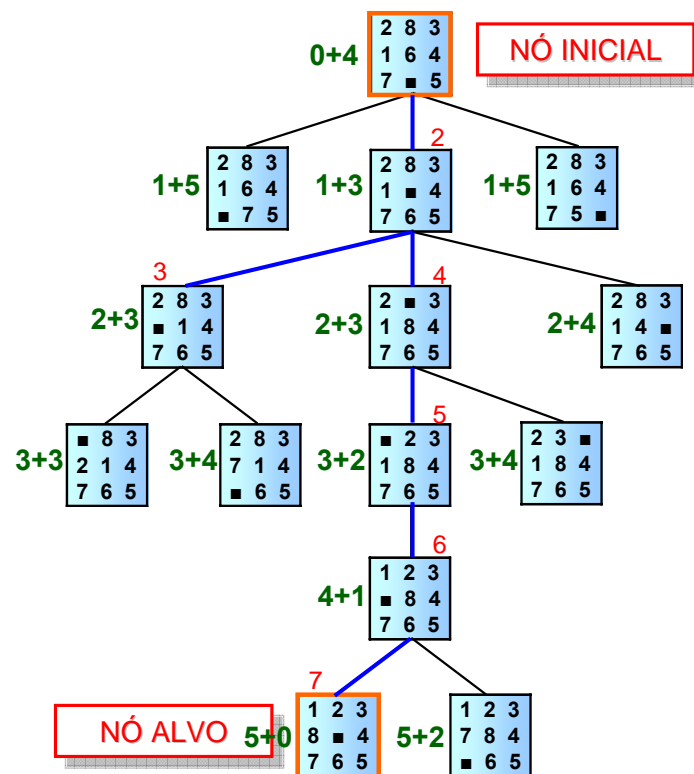
$g(n)$ = profundidade do nó, $g(\text{nó-inicial}) = 0$

$h_1'(n)$ = número de peças fora do lugar (o objetivo é ter zero peças fora do lugar)

$h_2'(n) = \sum_{i=1}^p m_i$, onde m_i é o número mínimo de movimentos para a peça i atingir a posição final e p é o número total de peças. Os movimentos permitidos são: $\leftarrow, \uparrow, \rightarrow$, e \downarrow .

A cada passo, o conjunto de nós ainda não expandidos é ordenado segundo a função $f'(n)$. Obs.: geralmente é utilizada uma fila de prioridades (implementada usualmente através de um *heap*), onde a cada passo sai o melhor (o mais urgente).

Na figura a seguir (extraída de [Ribeiro&Rocha, 2004]) é apresentado um exemplo da aplicação do algoritmo B&B na solução de um 8-puzzle utilizando a heurística $h_1'(n)$ = número de peças fora da posição.

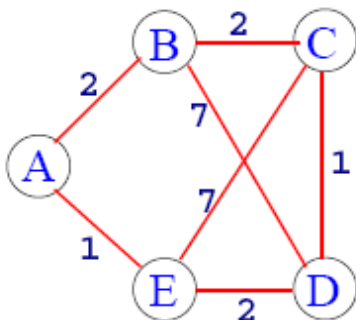


5.4. Problema do Caixeiro Viajante

Ramificação da árvore:

- Circuito do TSP é representado como sequência de n vértices distintos.
- Vamos enumerar todos os circuitos possíveis por sequências de vértices.
- Cada nó da árvore de enumeração representa um vértice.
- Cada ramificação de um nó para outro representa uma aresta percorrida ligando os dois nós.

Vamos fazer a árvore de ramificação do seguinte grafo:



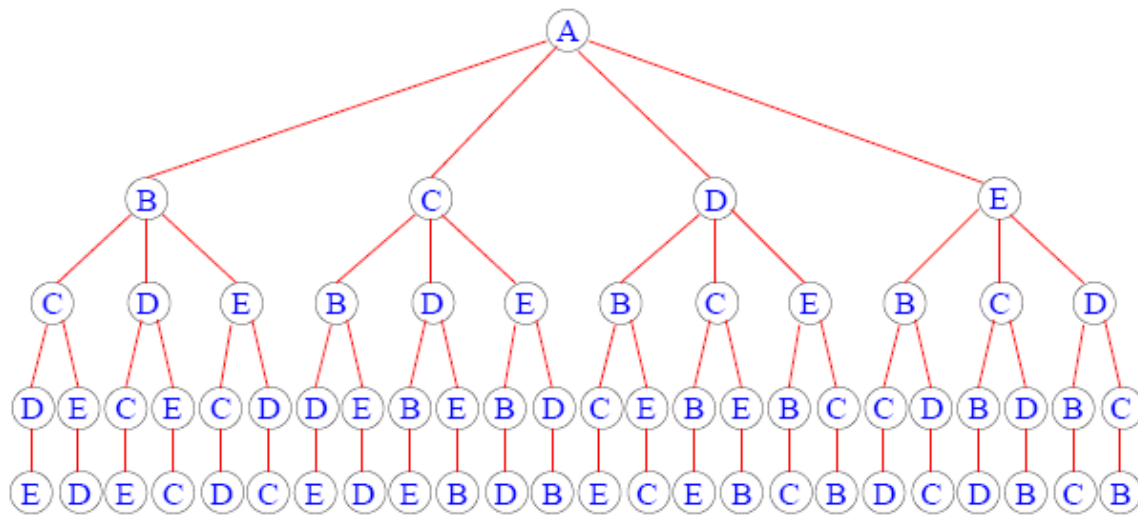


Figura 6 - Todas as seqüências possíveis dos vértices A, B, C, D e E

Algumas arestas da árvore de enumeração não existem. A Figura 7 seguinte mostra as podas que podemos fazer considerando apenas a falta de arestas.

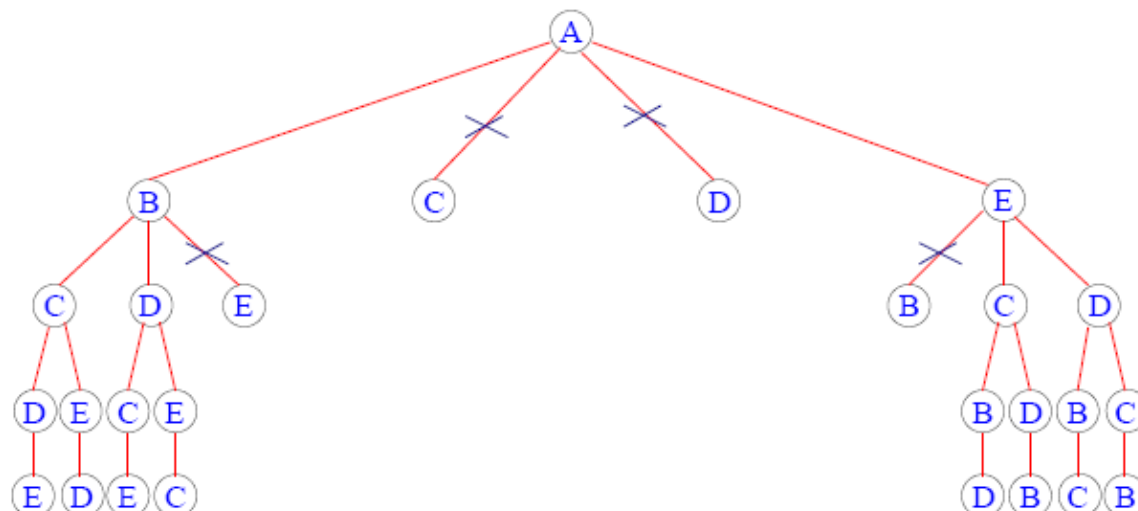


Figura 7 - Poda da árvore por inviabilidade

O grafo apresenta custos positivos. Vamos supor que o algoritmo escolheu de maneira gulosa um ramo mais promissor, digamos a seqüência A, B, C, D e E de custo 8. Se em algum percurso a soma das arestas já nos dê custo 8, podemos este ramo.

Além disso, o backtracking é exponencial em tempo, enquanto o B&B é exponencial em espaço de memória.

b) Comparando B&B com Programação Dinâmica

Os algoritmos de B&B não têm garantida eficiência computacional melhor que os baseados em Programação Dinâmica. A eficiência do B&B depende da função de fronteira escolhida e pode ser melhorado por uma função heurística de custo (ranking function). Essas heurísticas podem melhorar os algoritmos (no pior caso eles são exponenciais) e podem levar a um caso médio relativamente eficiente.

Como vimos, os algoritmos de Programação Dinâmica no pior caso são de ordem de complexidade polinomial (geralmente n^2 ou n^3). Portanto, os algoritmos B&B só devem ser preferidos quando se tem uma boa heurística que torne o caso médio melhor que n^2 ou n^3 .

Bibliografia

- [Brito, 2003] Brito, P. H. S, *MO417 - Atas das Aulas (Conceitos Básicos)*. Departamento de Computação, Unicamp.
- [Cormen, 2004] Cormen, T. H., et al., *Algoritmos: Teoria e Prática*, tradução da segunda edição [americana] Souza, Vendenberg D. de, Editora Campus, Rio de Janeiro, 2002.
- [Figueiredo, 2002] Figueiredo, J. C. A., *Apostila de Divisão e Conquista*, Departamento de Sistemas e Computação, Universidade Federal de Campina Grande.
- [Greve, 2003] Greve, F. G. P., *Lista de Exercícios 2 (Divisão e Conquista, Classificação interna)*. Departamento de Ciência da Computação, Universidade Federal da Bahia.
- [Horowitz et al., 1998] Horowitz, E., Sahni, S. & Rajasekaran, S., *Computer Algorithms*, Computer Science Press, 1998.
- [Koerich, 2005] Koerich, A. L., *Lidando com limitações do poder dos algoritmos*, Pontifícia Universidade Católica do Paraná, 2005.
- [Koerich, 2006] Koerich, Alessandro L., *Estratégia Gulosa: Técnicas de Projeto de Algoritmos*, Pontifícia Universidade Católica do Paraná.
- [Miyazawa, 2002] Miyazawa, F. K., *Otimização – Branch & Bound*, 2002
- [Pimentel & Cristina, 2006] Pimentel, G. & Cristina. M., *Algoritmo Guloso*, Departamento de Computação e Estatística, Universidade de São Paulo.
- [Pinto, 2003] Pinto, Guilherme, *Algoritmos Gulosos ("Greedy Algorithms")*, Unicamp.
- [Rich, 1993] Rich, E. *Inteligência Artificial*, Campus, 1993.
- [Ribeiro & Rocha, 2004] Ribeiro, C. C., Rocha, C. T. *Algoritmos em Grafos*, PUC-Rio, 2004.
- [Rudich, 2004] Rudich, Steven, *Lecture16: Great Theoretical Ideas In Computer Science*, Carnegie Mellon University.

Exercícios de Fixação

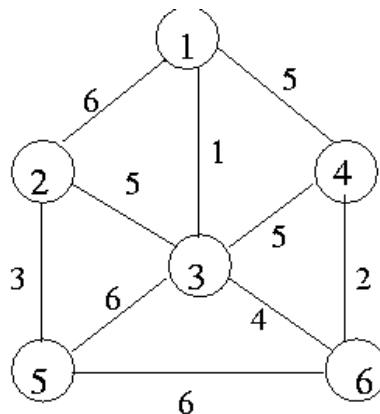
1) Seja $p(x) = 5 + 3x + 8x^2 - 7x^3$ e $q(x) = -1 + 7x - 3x^2 + 4x^3$. Faça a multiplicação passo a passo de $p(x)$ por $q(x)$ usando:

- a) O algoritmo de força bruta
- b) O algoritmo de convolução
- c) O algoritmo de convolução melhorado (usando a idéia de produto igual à área)

2) Faça uma revisão de alguns algoritmos que você já conhece que usam a estratégia de divisão e conquista (torres de Hanói, máximo e mínimo, busca binária, busca ternária, quicksort, mergesort, multiplicação de polinômios etc). Analise como o fator de balanceamento desses algoritmos (relação entre a quantidade e tamanho dos subproblemas e o esforço para resolvê-los) influencia em sua eficiência.

3) Faça uma pesquisa sobre os algoritmos de *Prim* e *Kruskal* para resolver o problema da Árvore Geradora Mínima.

4) Mostre passo a passo como os algoritmos de Prim e Kruskal (para o problema da **Árvore Geradora Mínima**) executam sobre o grafo a seguir:



5) Encontre a solução ótima para o **problema da mochila** com a seguinte instância: $n=7$, $m=15$, $(p_1, p_2, \dots, p_7)=(10, 5, 15, 7, 6, 18, 3)$, e $(w_1, w_2, \dots, w_7)=(2, 3, 5, 7, 1, 4, 1)$.

6) Problema do troco com moedas

Seja $a_n = \{a_0, a_1, a_2, \dots, a_n\}$ um conjunto finito de moedas de valores distintos. Vamos assumir que a_i tem valor v_i , com $v_i = 10^i$, para $i = 0, \dots, n$. Além disso, dispomos de um número ilimitado de moedas de cada tipo. O problema do troco com moedas consiste em obter exatamente um valor total C inteiro, não nulo, usando o menor número possível de moedas.

- (a) Formule o algoritmo correto mais eficiente que você puder baseado na estratégia do método guloso.
- (b) Encontre a complexidade do seu algoritmo.

7) Descreva um algoritmo $O(n)$ que dado um conjunto $\{x_1, x_2, \dots, x_n\}$ de pontos na reta real determine o menor conjunto de intervalos fechados de comprimento unitário (tamanho 1 no máximo) que contenha todos os pontos dados. Mostre informalmente que o seu algoritmo é correto.

Exs.:

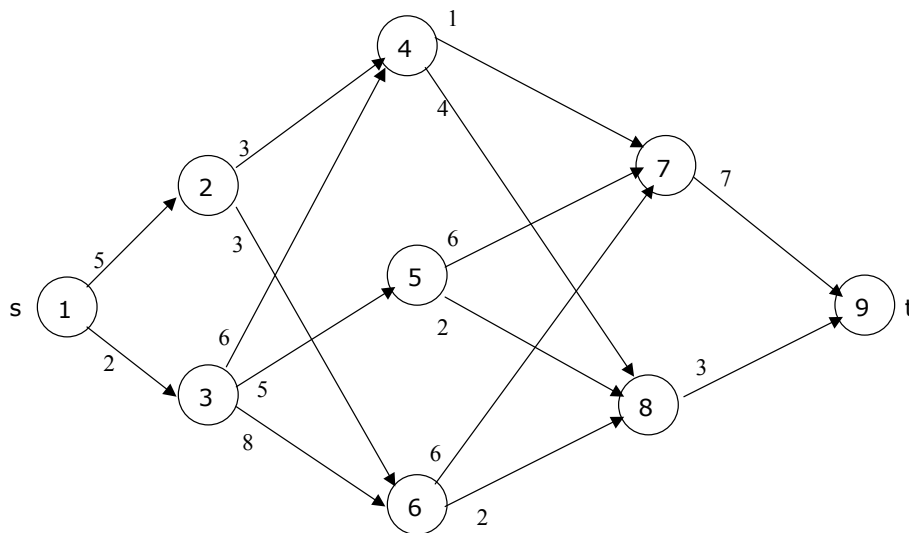
Entrada: $\{0.3, 0.4, 0.56, 0.78, 1.3, 1.4, 1.81, 1.9, 2.4\}$.

Saída: $\{[0.3, 1.3], [1.4, 2.4]\}$

Entrada: $\{0.3, 0.7, 1.4, 2.5, 2.8, 3.4, 4.1\}$.

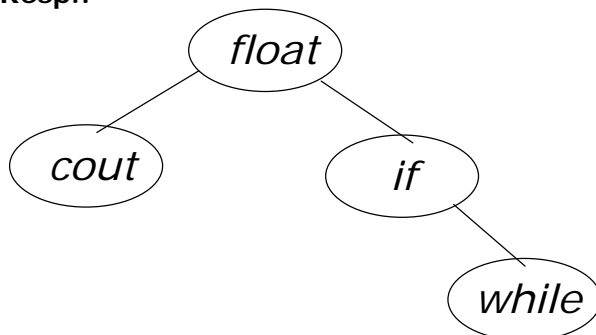
Saída: $\{[0.3, 0.7], [1.4], [2.5, 2.8, 3.4], [4.1]\}$

8) Encontre o custo do caminho mínimo entre s e t no grafo multiestágio a seguir, usando as estratégias (a) *forward* e (b) *backward*. [**Resp.: Custo = 12**]



9) Construa a árvore binária de pesquisa ótima para o seguinte conjunto de identificadores: $(a_1, a_2, a_3, a_4) = (cout, float, if, while)$, com $p(1)=1/20$, $p(2)=1/5$, $p(3)=1/10$, $p(4)=1/20$, $q(0)=1/5$, $q(1)=1/10$, $q(2)=1/5$, $q(3)=1/20$ e $q(4)=1/20$. Use o algoritmo OBST (Horowitz, p. 283).

Resp.:



10) Para cada um dos problemas a seguir, discutidos em sala, identifique o que são os *estágios*, a *decisão* em cada estágio e a *função-objetivo*.

Problema	Estágios	Decisão	Função-Objetivo
<i>Caminho mínimo entre as cidades A e B</i>	Instantes (representado por um conjunto de cidades)	ir/não ir por uma cidade	minimizar a distância total percorrida entre A e B
<i>Alocação de recursos</i>			
<i>Reposição de Equipamentos</i>			
<i>Edição de String</i>			
<i>Árvore Binária de Busca Ótima</i>			

11) Implementar os algoritmos de *backtracking* para:

- (i) Problema das 8 rainhas,
- (ii) Problema do Passeio do Cavaleiro. Aprimore a busca de uma nova posição válida (coordenadas do próximo movimento), utilizando uma função que sorteia o próximo passo dentre os passos possíveis.

12) Pesquisar na Internet outros exemplos de problemas com solução por *backtracking* não discutidos em sala. (obs.: de preferência exemplos com implementação e demo)

13) Utilize a estratégia de *backtracking* para desenvolver um algoritmo que resolva o problema de coloração de grafos.

14) O Professor Zé da Silva deseja marcar os exames finais de 7 cursos e gostaria de evitar que um estudante fizesse mais do que um exame por dia. Na tabela a seguir, uma entrada i,j significa que um curso i tem pelo menos um estudante em comum, ou seja, os exames desses cursos não podem estar no mesmo dia. Qual o menor número de dias que o Professor Zé da Silva vai precisar para alocar todos os exames? Mostre como ele poderia alocar os exames.

.	1	2	3	4	5	6	7
1	.	*	*	*	-	*	*
2	*	.	*	-	-	-	*
3	*	*	.	*	-	-	-
4	*	-	*	.	*	*	-
5	-	-	-	*	.	*	-
6	*	-	-	*	*	.	*
7	*	*	-	-	-	*	.

15) Resolva o problema do **15-puzzle**, partindo do estado inicial e chegando no estado final dados a seguir:

Estado inicial:

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Estado final:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Utilize como *ranking function* $f'(n)=g(n)+h'(n)$, $g(n)$ = profundidade do nó, $h'(n)$ = número de peças fora do lugar

Montar o grafo passo a passo, até obter a solução.

16) Aplique *B&B* para resolver o **Problema das 6 Rainhas**.

Utilize como *ranking function* $f'(n)=g(n)+h'(n)$, $g(n)$ = profundidade do nó, $h'(n)$ = número característico da diagonal. Montar o grafo passo a passo, até obter a solução.

17) Seja $G=(V,E)$ um grafo não dirigido. Mostre que $S=(E, \mathcal{F})$ é um matróide, onde E é o conjunto de arestas de G e \mathcal{F} é o conjunto das florestas de G .