UDESC

Complexidade de Algoritmos

Prof. Diego Buchinger
diego.buchinger@outlook.com
diego.buchinger@udesc.br

Prof. Cristiano Damiani Vasconcellos
cristiano.vasconcellos@udesc.br

UDESC

Abordagens para Resolução de Problemas

UDESC

Abordagens para Resolução de Problemas

Muitos problemas podem ser resolvidos de diversas maneiras diferentes.

Contudo, dependendo da abordagem escolhida, uma solução pode ser melhor ou pior do que outra em questões de tempo ou uso de memória.

UDESC

Abordagens para Resolução de Problemas

Podemos classificar os principais métodos de resolução de problemas em relação a abordagem utilizada:

- Indução Matemática – Fórmula
- Divisão e Conquista (*divide and conquer*)
- Algoritmos Gulosos (*greedy*)
- Algoritmos de Tentativa e Erro (*backtracking*)
- Programação dinâmica (*dynamic programming*)
- Algoritmos de aproximação (*approximation*)

Indução Matemática – Fórmula

Abordagem mais simples possível.

O problema pode ser reduzido a uma fórmula ou a um conjunto de fórmulas matemáticas.

Exemplo: descobrir as raízes de uma equação de segundo grau.

UDESC

Algoritmos de Divisão-e-Conquista
(*Divide and Conquer*)

Já estudamos esta abordagem.

Desmembrar o problema original em vários subproblemas semelhantes (menores), resolver os subproblemas (executando o mesmo processo recursivamente) e combinar as soluções.

Exemplos: ordenar um vetor de elementos (*Merge-Sort*, *Quick-Sort*) e multiplicação entre inteiros grandes

UDESC

Algoritmos Gulosos
(*Greedy Algorithms*)

Uma técnica para resolver problemas de otimização: minimizar ou maximizar um determinado valor.

Um algoritmo guloso sempre faz a escolha que parece ser a melhor em um determinado momento, esperando que essa melhor escolha local leve ao melhor resultado do problema como um todo.

Uma técnica simples, mas que na maioria das vezes não leva ao resultado ótimo.

UDESC

Algoritmos Gulosos
(*Greedy Algorithms*)

Alguns problemas que podem ser resolvidos com algoritmos gulosos:

- Problema do melhor troco - canônico
- Problema da Árvore Geradora Mínima
- Escalonamento de tarefas
- Codificação de Huffman
- Problema da Mochila Fracionada

Problema do Melhor Troco - canônico

Considere que o sistema econômico de um local utiliza unidades monetárias de valor: M_0, M_1, M_2, ... M_n;

Qual é o menor número de unidades monetárias que representam um determinado valor V?

Exemplo: Moedas disponíveis: 1, 5, 10, 25, 50, 100

Valor do troco: 3.82

Qual a menor quantidade de moedas a serem entregues e quais são elas?

Problema do Melhor Troco - canônico

Para sistemas monetários **canônicos** como os usualmente utilizados, podemos adotar a abordagem de escolher o maior número possível das maiores moedas:

Exemplo: Moedas disponíveis: 1, 5, 10, 25, 50, 100

Valor do troco: $ 3.82

382 / 100 = 3 e sobram $ 82	3 moedas de 100
82 / 50 = 1 e sobram 32	1 moeda de 50
32 / 25 = 1 e sobram 7	1 moeda de 25
7 / 5 = 1 e sobram 2	1 moeda de 5
2 / 1 = 2 e sobram 0	2 moedas de 1

Problema do Melhor Troco - canônico

CUIDADO! Para sistemas monetários **não canônicos** a abordagem apresentada pode não resultar no melhor troco possível:

Exemplo Trivial: Moedas disponíveis: 1, 3, 4

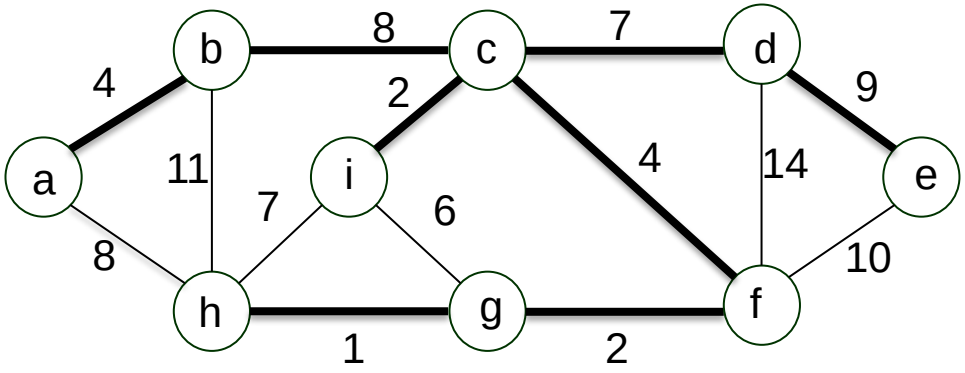
Valor do troco: $ 0.06

Qual é o número mínimo de moedas para o troco?

Problema da Árvore Geradora Mínima

Dado um grafo com seus respectivos vértices e arestas, determinar qual a árvore que contém todos os vértices deste grafo com o menor custo possível.

Escolhemos sempre a melhor aresta dos nós já inclusos que não formem um circuito (Prim)



Escalonamento de Tarefas

Dada uma lista de tarefas a serem executadas com um horário de início e um horário de término, determinar qual a quantidade máxima de atividades que podem ser executadas

Exemplo: Um auditório só pode ser utilizado para um evento por vez. Em um dia com muitos eventos, deseja-se determinar qual é o maior número de eventos que podem ser realizados no auditório, e quais são eles (OBS: pode haver mais de uma solução).

Evento	1	2	3	4	5	6	7	8	9	10	11
Inicio	3	8	5	1	6	12	0	8	5	2	3
término	5	11	7	4	10	14	6	12	9	13	8

Escalonamento de Tarefas

Solução gulosa: ordenamos os eventos pelo horário de término (em ordem crescente) e sempre que possível pegamos o evento com menor horário de término.

Evento	4	1	7	3	11	9	5	2	8	10	6
Inicio	1	3	0	5	3	5	6	8	8	2	12
término	4	5	6	7	8	9	10	11	12	13	14

Codificação de Huffman

Um método de compressão de dados. Esse método usa a frequência com que cada símbolo ocorre, em uma coleção de dados, para determinar um código de tamanho variável para o símbolo.

Caractere	Frequência	Código (fixo)	Código (variável)
a	45	000	0
b	13	001	101
c	12	010	100
d	16	011	111
e	9	100	1101
f	5	101	1100

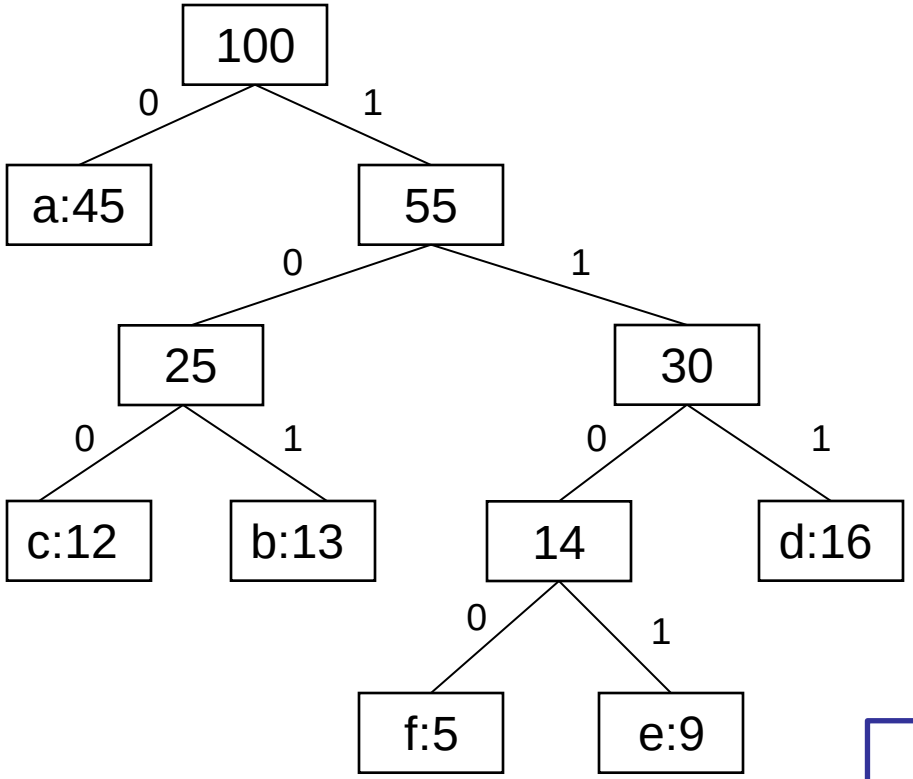
Exemplo:

... faca ...

... 101000010000 ...

... 110001000 ...

Codificação de Huffman (Exemplo)



Exemplo:

... faca ...

... 101000010000 ...

... 110001000 ...

Codificamos usando a notação:
Filhos a esquerda: 0
Filhos a direita: 1
Até chegar a letra desejada

Codificação de Huffman

Huffman(C)

$n \leftarrow |C|$

$Q \leftarrow C$

para $i \leftarrow 1$ **até** n - 1

$z \leftarrow$ AlocaNo()

$x \leftarrow z.esq \leftarrow$ Extrair-Minimo(Q)

$y \leftarrow z.dir \leftarrow$ Extrair-Minimo(Q)

$z.valor = x.valor + y.valor$

Inserir(Q, z)

retorne Q

Testando o algoritmo:
Como ficaria a codificação para a string:

"testando o teste"

Construir a árvore e o mapa de codificação

Importante:

Note que para decodificar uma string, é necessário conhecer a árvore que foi usada para gerá-la!
Ou seja, espaço utilizado: nova string (menor) + árvore

Problema da Mochila Fracionada

Dark Version: Um ladrão está assaltando uma loja e possui uma mochila que pode carregar até **P** kg sem arrebentar. Sabendo os pesos e valores dos itens, qual é o maior valor possível que o ladrão conseguirá roubar?

Na versão da mochila fracionada os elementos podem ser pegos em fração (apenas parte do objeto, com proporção linear entre peso e valor)

Exemplo: Mochila com capacidade 9 kg

Objetos	1	2	3	4	5	6
Peso	1	3	2	5	7	9
Valor	3	2	4	7	9	12

Problema da Mochila Fracionada

Solução Gulosa: Ordenamos o vetor de itens em relação a proporção: [valor / peso]

Objetos	1	3	4	6	5	2
Peso	1	2	5	9	7	3
Valor	3	4	7	12	9	2
Proporção	3	2	1.4	1.33	1.29	0.67

UDESC

Tentativa e Erro
(*Backtracking*)

Outra técnica para resolver problemas de otimização

Testa todas as possíveis soluções até encontrar a(s) melhor(es) solução(ões) para um problema, utilizando as ideias de busca em profundidade ou busca em largura

Pode utilizar validações para diminuir (de maneira não muito significativa) o escopo de busca [ex: desconsiderar soluções parciais piores do que uma solução final já encontrada]

Geralmente não adequada para instâncias muito grandes

UDESC

Tentativa e Erro
(*Backtracking*)

Alguns problemas clássicos que são resolvidos (também) com *backtracking* (instâncias pequenas):

- Problema do passeio do cavalo
- Puzzle 3x3
- Sokoban
- Caixeiro Viajante

Passeio do Cavalo

Partindo de uma posição inicial, qual é o menor número de movimentos que um cavalo deve realizar para chegar a qualquer uma das casas de um tabuleiro?

LEMBRETE: o cavalo (*knight*) anda apenas em movimentos 'L'

Exemplo: tab. 5x5 c->2,1

- Busca em Profundidade...
- Busca em Largura...

UDESC

Tentativa e Erro
(*Backtracking*)

Solução usando Busca em Profundidade:

[+] usa menos memória (apenas tabuleiro)

[–] repete muitos movimentos (mais lento)

Solução usando Busca em Largura:

[+] não repete movimentos (mais rápido)

[–] usa mais memória (para salvar as posições)

em alguns casos se torna inviável

Puzzle 3x3

Montar a imagem original reorganizando as peças, podendo movimentar as peças usando a única lacuna.

Como resolver computacionalmente?

Puzzle 3x3

Como resolver computacionalmente?

- Abstrair que os movimentos se baseiam na lacuna: ela pode ir para cima, baixo, direita e esquerda;
- Representar o ‘puzzle’ como uma estrutura de dados;
- Armazenar a quantidade de movimentos (mínima) necessárias para alcançar uma dada posição.
- Descobrir o mínimo para chegar à posição final.

Puzzle 3x3

Exemplo:

Peças são representadas por números entre 1 e 8

Lacuna é representada pelo número 0

1	**0**	**3**
4	2	5
7	8	6

Sokoban

Levar as caixas para pontos objetivos, em qualquer ordem, com o menor número de movimentos possível.

Quais valores representam um estado neste problema?

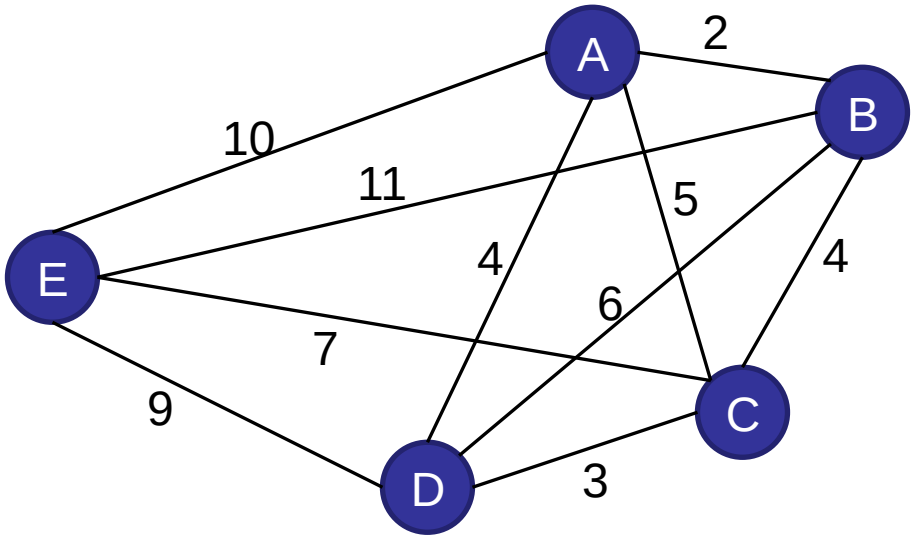
Sokoban

Posição do jogador (x,y)	(1,2) e (2,2)
Posição das caixas (x,y)	[1][2][2][2] array

Problema do Caixeiro Viajante

Um vendedor deseja visitar *n* cidades e retornar a cidade de origem. Dado um grafo não orientado completo com *n* vértices, onde existe um custo *c*(*i*, *j*) (associado a cada aresta) para viajar da cidade *i* a cidade *j*.

Qual o trajeto com custo mínimo?



UDESC

Programação Dinâmica
(*Dynamic Programming*)

Uma abordagem auxiliar para reduzir a computação em problemas de otimização.

Consiste em dividir o problema original em subproblemas mais simples, resolvê-los, armazenar os resultados em memória e consultar estes resultados posteriormente a fim de reduzir o espaço de busca.

Resultados são calculados apenas uma vez (nunca são recalculados por outra iteração)

UDESC

Programação Dinâmica
(*Dynamic Programming*)

Alguns problemas que podem ser resolvidos com programação dinâmica:

- Fibonacci
- Problema do Menor Troco não canônico
- Problema da Mochila Binária (0/1)
- Maior Subsequência Comum
- Multiplicação de Cadeias de Matrizes

UDESC

Fibonacci

O modelo recursivo de Fibonacci pode ser extremamente ruim dependendo da implementação:

```
int fib(int n){
    if( n<=1 ) return n;
    return fib(n-1) + fib(n-2);
}
```

UDESC

Fibonacci

Para entender a programação dinâmica, talvez o problema da sequencia de Fibonacci seja o exemplo mais simples para demonstrar o seu uso.

A ideia é utilizar uma estrutura auxiliar para armazenar os resultados e utilizá-los posteriormente

Fibonacci

Abordagem *top-down* (*Memoization*)

```
Fib(n)
    se n ≤ 1 então
        retorne n
    senão
        se F[n] está indefinido
        F[n] ← Fib(n − 1) + Fib(n − 2)
        retorne F[n]
```

O vetor **F** é inicializado antes da chamada da função com um valor padronizado que nunca será registrado [ex: -1]

UDESC

Fibonacci

Abordagem *bottom-up*

```
Fib(n)
        F[0] = 0
        F[1] = 1
        para i ← 2 até n
                F[i] = F[i − 2] + F[i − 1]
        retorne F[n]
```

UDESC

Problema do Melhor Troco não canônico

Como visto anteriormente, para sistemas monetários **não canônicos** a abordagem gulosa pode não resultar no melhor resultado!

Nesse caso, podemos usar programação dinâmica como um método eficiente para encontrar o melhor troco:

Exemplo Trivial: Moedas disponíveis: 1, 3, 4

Valor do troco: $ 0.06

Qual é o número mínimo de moedas para o troco?

UDESC

Problema do Melhor Troco não canônico

Exemplo Trivial: Moedas disponíveis: 1, 3, 4

Valor do troco: $ 0.06

Considere as moedas como um vetor: M = { 1, 3, 4 }

Inicialmente assumimos que o troco para $ 0.00 precisa de zero moedas

$$T(0) = 0$$

E seguimos iterativamente até o troco desejado usando a fórmula:

$$T(n) = \min(\quad 1+T(n-M[0]) , 1+T(n-M[1]) , 1+T(n-M[2]) \quad)$$

Problema do Melhor Troco não canônico

Exemplo Trivial: Moedas disponíveis: 1, 3, 4

Valor do troco: $ 0.06

$$T(0) = 0$$

$$T(n) = \min(\ 1+T(\ n-M[0]\ ),\ 1+T(\ n-M[1]\ ),\ 1+T(\ n-M[2]\ )\ )$$

$$T(1) = \min(\ 1+T(1-1),\ 1+\sout{T(1-3)},\ 1+\sout{T(1-4)}\ ) = 1$$

$$T(2) = \min(\ 1+T(2-1),\ 1+\sout{T(2-3)},\ 1+\sout{T(2-4)}\ ) = 2$$

$$T(3) = \min(\ 1+T(3-1),\ 1+T(3-3),\ 1+\sout{T(3-4)}\ ) = 1$$

(...)

Complexidade de tempo => Θ (n * |M|)

Complexidade de espaço => Θ (n)

Problema da Mochila Binária (0/1)

Dark Version: Um ladrão está assaltando uma loja e possui uma mochila que pode carregar até **P** kg sem arrebentar. Sabendo os pesos e valores dos itens, qual é o maior valor possível que o ladrão conseguirá roubar?

- Na versão da mochila binária os elementos só podem ser pegos como um todo (não podemos pegar metade de um item – ou pega, ou não pega).

- **Exemplo**: Mochila com capacidade 9 kg

Objetos	1	2	3	4	5	6
Peso	1	3	2	5	7	9
Valor	3	2	4	7	9	12

Problema da Mochila Binária (0/1)

Solução Matricial

Montamos uma matriz para armazenar as melhores soluções intermediárias e as usamos para gerar as próximas soluções;

Objetos	1	2	3	4	5	6
Peso	1	3	2	5	7	9
Valor	3	2	4	7	9	12

Problema da Mochila Binária (0/1)

Solução Matricial

Objetos	1	2	3	4	5	6
Peso	1	3	2	5	7	9
Valor	3	2	4	7	9	12

Item / Capacidade	0	1	2	3	4	5	6	7	8	9
1	0	3	3	3	3	3	3	3	3	3
2	0	3	3	3	5	5	5	5	5	5
3	0	3	4	7	7	7	9	9	9	9
4	0	3	4	7	7	7	10	11	14	14
5	0	3	4	7	7	7	10	11	14	14
6	0	3	4	7	7	7	10	11	14	**14**

```
m[item][k] = max( m[item-1][k], valor + m[item-1][k-peso] )
```

Problema da Mochila Binária (0/1)

Solução Matricial

Complexidade de tempo => Θ (peso * qtd-itens)

Complexidade de espaço => Θ (peso * qtd-itens)

Maior Subsequência Comum

Longest Common Subsequence

Dadas duas strings S e T, qual é maior subsequencia (sequencia não necessariamente contígua), da esquerda para a direita, entre estas strings?

- **Exemplo**: S = ABAZDC
  T = BACBAD

  Resposta ?

Resolução similar a abordagem matricial para resolver o problema da mochila.

Problema da Mochila Binária (0/1)

Solução Matricial

pA / pB		A	B	A	Z	D	C
	0	0	0	0	0	0	0
B	0						
A	0						
C	0						
B	0						
A	0						
D	0						

1 – Preenche linha e coluna fictícia

2 – usar regra:

```
m[i][j] = max( m[i-1][j],
               m[i][j-1] )

SE pA[i] == pB[j]:
    m[i][j] = max( m[i][j],
                   m[i-1][j-1] + 1 )
```

Problema da Mochila Binária (0/1)

Solução Matricial

pA / pB		A	B	A	Z	D	C
	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1
A	0	1	1	2	2	2	2
C	0	1	1	2	2	2	3
B	0	1	2	2	2	2	3
A	0	1	2	3	3	3	3
D	0	1	2	3	3	4	4

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

Dadas uma sequencia de *n* matrizes, qual é a melhor ordem para multiplicá-las a fim de realizar menos multiplicações.

OBS: a multiplicação entre matrizes é associativa, logo (AB)(CD) = A(B(CD)) = (AB)C)D = (A(BC))D = A((BC)D)

- **Exemplo**: São dadas 4 matrizes com os seguintes tamanhos:

$M_1 = (2,3)$

$M_2 = (3,6)$

$M_3 = (6,4)$

$M_4 = (4,5)$

$M_1 * M_2 = 2 * 3 * 6 = 36$

$M_2 * M_3 = 3 * 6 * 4 = 72$

$M_3 * M_4 = 6 * 4 * 5 = 120$

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

- **Exemplo**: São dadas 4 matrizes com os seguintes tamanhos:

$M_1 = (2,3)$ $\quad M_1 * M_2 = 2 * 3 * 6 = 36$

$M_2 = (3,6)$ $\quad M_2 * M_3 = 3 * 6 * 4 = 72$

$M_3 = (6,4)$ $\quad M_3 * M_4 = 6 * 4 * 5 = 120$

$M_4 = (4,5)$

MATRIZ	1	2	3	4
1	0	36		
2		0	72	
3			0	120
4				0

MATRIZ	1	2	3	4
1	0	1		
2		0	2	
3			0	3
4				0

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

- **Exemplo**: São dadas 4 matrizes com os seguintes tamanhos:

$M_1 = (2,3)$ $(M_1 * M_2)\ M_3 = 36 + 2 * 6 * 4 = 84$

$M_2 = (3,6)$ $M_1\ (M_2 * M_3) = 2 * 3 * 4 + 72 = 96$

$M_3 = (6,4)$ $(M_2 * M_3)\ M_4 = 72 + 3 * 4 * 5 = 132$

$M_4 = (4,5)$ $M_2\ (M_3 * M_4) = 3 * 6 * 5 + 120 = 210$

MATRIZ	1	2	3	4
1	0	36	84	
2		0	72	132
3			0	120
4				0

MATRIZ	1	2	3	4
1	0	1	1	
2		0	2	2
3			0	3
4				0

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

- **Exemplo**: São dadas 4 matrizes com os seguintes tamanhos:

$M_1 = (2,3)$

$M_2 = (3,6)$

$M_3 = (6,4)$

$M_4 = (4,5)$

$(M_1\, M_2\, M_3)\, M_4$ = 84 + 2 * 4 * 5 = 124

$(M_1 M_2)*(M_3 M_4)$ = 36+2*6*5+120 = 216

$M_1\, (M_2\, M_3\, M_4)$ = 2 * 3 * 5 + 132 = 162

MATRIZ	1	2	3	4
1	0	36	84	124
2		0	72	132
3			0	120
4				0

MATRIZ	1	2	3	4
1	0	1	1	1
2		0	2	2
3			0	3
4				0

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

PSEUDOCÓDIGO

```
MATRIX-CHAIN-MULT(   v[n]  )
    para  i de 1 a n:                       // zera a primeira linha
        m[ i , i ] = 0
    para  c de 2 a n:                       // para todas as linhas posteriores
        para  i de 1 a (n - c + 1):         // para todos os elem. da diagonal
            j = i + c - 1
            m[ i , j ] = ∞
            para  k de i a (j - 1):         // para todas as permutações
                q = m[ i , k ] + m[ k+1 , j ] + v[ i-1 ] * v[ k ] * v[ j ]
                se  q < m[ i , j ]:
                    m[ i , j ] = q
                    s[ i, j] = k
```

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

PSEUDOCÓDIGO

```
PRINT-EQUACAO (  s[n][n] , i , j  )
    se i == j
        print  "M" + i
    senão
        print  "("
        PRINT-EQUACAO(   s , i , s[ i , j ] )
        PRINT-EQUACAO(   s , s[ i , j ] + 1 , j )
        print  ")"
```

MATRIZ	1	2	3	4
1	0	1	1	1
2		0	2	2
3			0	3
4				0

P-E (1,4)
- P-E(1,1) => "M1"
- P-E(2,4)
  - P-E(2,2) => "M2"
  - P-E(3,4)
    - P-E(3,3) => "M3"
    - P-E(4,4) => "M4"

(M1 (M2 (M3 M4)))

UDESC

Algoritmos de Aproximação
(*Approximation Algorithms*)

Existem alguns problemas os quais não se conhece uma solução eficiente. Dizemos que estes problemas são “difíceis” ou intratáveis pois, para instâncias grandes, o tempo de processamento seria inviável.

Nestas situações é comum remover a exigência de procurar pela solução ótima e passamos a procurar por uma solução próxima da ótima (solução boa / razoável)

UDESC

Algoritmos de Aproximação
(*Approximation Algorithms*)

Existem duas abordagens principais:

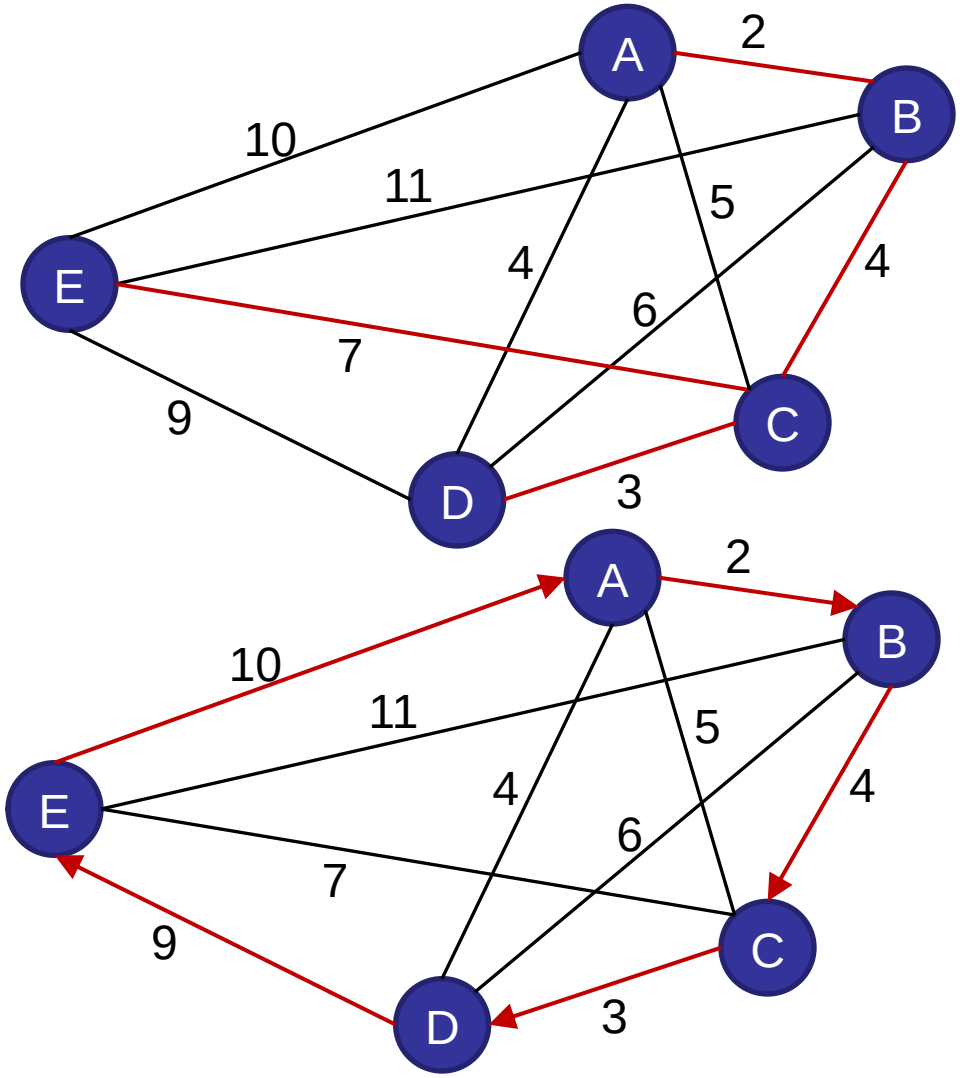
- **Heurística**: algoritmo que buscam soluções próximas a solução ótima. Utilizam alguma informação (ou intuição) sobre a instância do problema para resolvê-lo de forma eficiente. Podem ser genéricas (servem p/ vários problemas).

- **Algoritmo Aproximado**: algoritmo que resolve um problema de forma eficiente e ainda garante a qualidade da solução. É necessário provar que a solução está próxima da ótima. Geralmente únicos para cada problema.

Heurísticas

- **Algoritmos Bio-inspirados**: algoritmo baseados no comportamento de seres vivos:
  - **Algoritmos genéticos**: gera soluções, escolhe as melhores (seleção natural) e gera um novo ciclo, oferecendo espaço para mutações (alterações nos componentes das soluções).
  - **Colônia de formigas**: modela a geração de soluções baseado no comportamento das formigas (como elas sempre encontram um doce?!) .

Algoritmos de Aproximação
Caixeiro Viajante



Dado o grafo $G = (V, A)$ e o custo c:

1. Selecione um vértice $r \in V$ para ser o vértice *raiz*.
2. Obtenha a ávore geradora mínima a partir de r.
3. Faça H ser a lista de vertices ordenados de acordo com a primeira visita, considerando um percurso em pré-ordem, retornando a r.

Se a função custo satisfaz a desigualdade de triângulos: $c(u, w) < c(u, v) + c(v, w)$

$$c(H) < 2c(\textit{ótimo})$$

UDESC

Exercícios

Analisar e resolver os seguintes problemas:

- https://www.urionlinejudge.com.br/judge/pt/problems/view/1034
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1286
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1288
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1310
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1351
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1458
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1602
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1976
- https://www.urionlinejudge.com.br/judge/pt/problems/view/2032
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1912
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1166

UDESC

Referências

Algoritmos. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein. Campus.

Algorithms. Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani. McGraw Hill.

Concrete Mathematics: A Foundation for Computer Science (2nd Edition). Ronald L. Graham, Donald E. Knuth, Oren Patashnik. Addison Wesley.

M. R. Garey and D. S. Johnson. 1978. *"Strong" NP-Completeness Results: Motivation, Examples, and Implications.* J. ACM 25, 3 (July 1978)