UDESC

Projeto e Análise de Algoritmos

(Classes de Problemas)

Cristiano Damiani Vasconcellos

cristiano.vasconcellos@udesc.br

UDESC

Problemas tratáveis e intratáveis

Problemas tratáveis: resolvidos por algoritmos que executam em tempo polinomial.

Problemas intratáveis: não se conhece algoritmos que os resolvam em tempo polinomial.

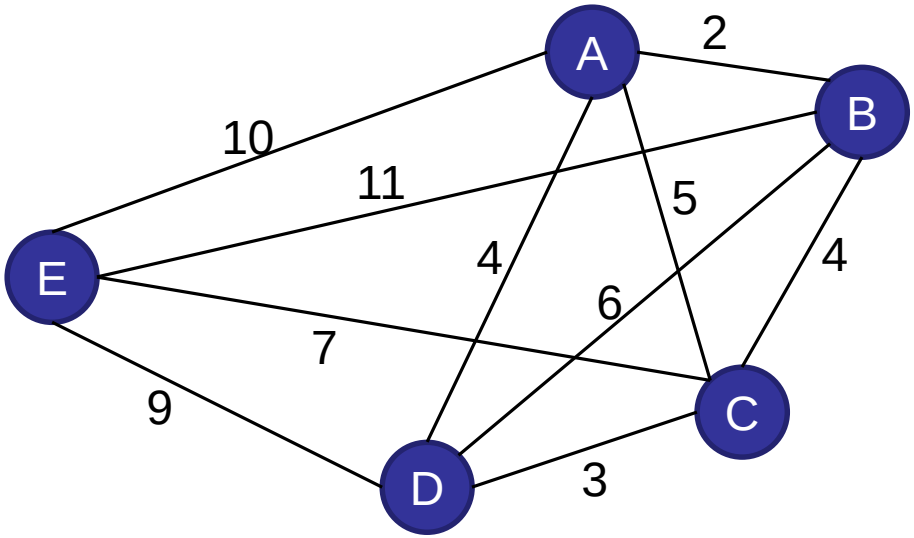
Problemas de Decisão.

Problemas de Otimização: Cada solução possível tem um valor associado e desejamos encontrar a solução com melhor valor.

Problemas de Decisão: Problemas que tem resposta sim ou não.

Problema do Caixeiro Viajante

Consistem em descobrir o caminho com custo mínimo para um vendedor que deseja visitar *n* cidades e retornar a cidade de origem. O problema é representado por um grafo não orientado completo ponderado com *n* vértices. Sendo que o valor *c*(*u*, *v*) (associado a cada aresta) representa o custo para viajar da cidade *u* a cidade *v*.



Problema do Caixeiro Viajante

Existe um caminho com custo menor que k?

Algoritmos Não Deterministas

Capaz de escolher uma entre várias alternativas possíveis a cada passo. A alternativa escolhida será sempre a alternativa que leva a conclusão esperada, caso essa alternativa exista.

```
int pesquisa(Estrutura *v, int n, int chave)
{
        int i;

        for (i = 0; i < n; i++)
                if (v[i].chave == chave)
                        return i;
        return -1;
}
```

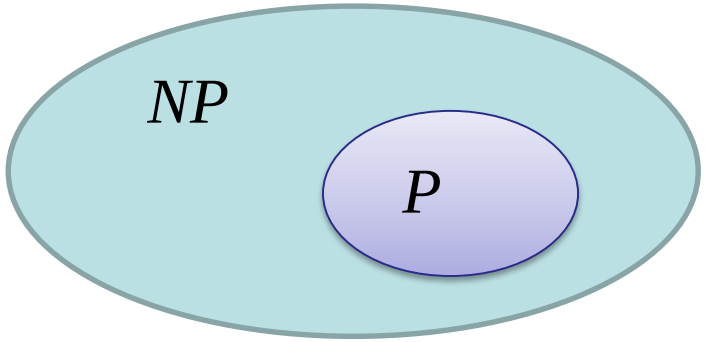
```
int pesquisa(Estrutura *v, int n, int chave)
{
        int i;

        i = escolheND(0, n - 1);
        if (v[i].chave == chave)
                return i;
        return -1;
}
```

Classes de Problemas *P* e *NP*

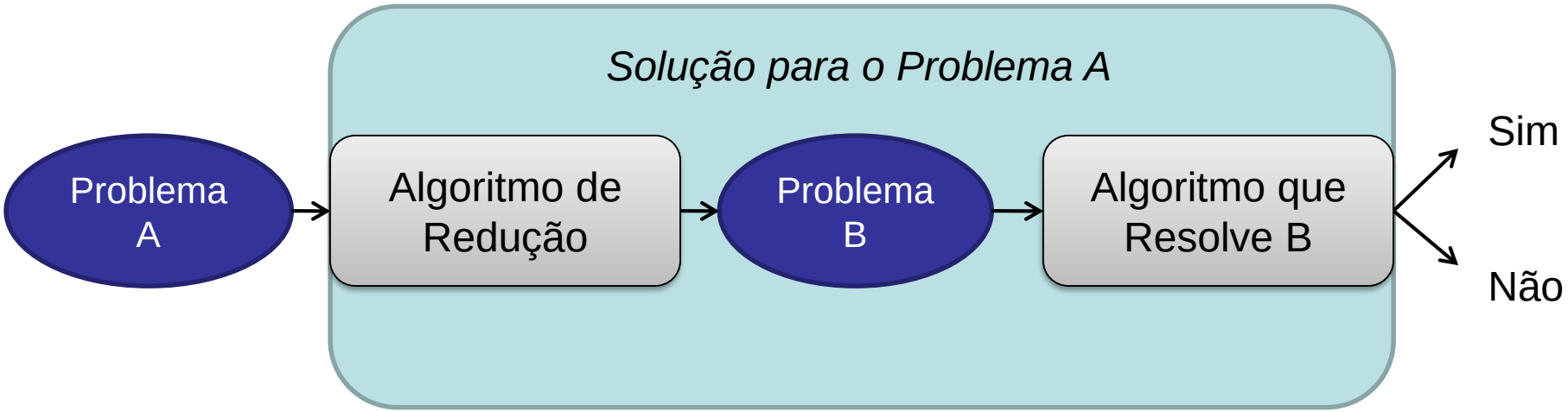
Classe de Problemas *P*: Problemas que podem ser resolvido (por algoritmos deterministas) em tempo polinomial.

Classe de Problemas *NP*: Problemas que podem ser resolvidos por algoritmos não deterministas em tempo polinomial. Ou problemas que a solução pode ser verificada em tempo polinomial.

Possível relação entre as classes:



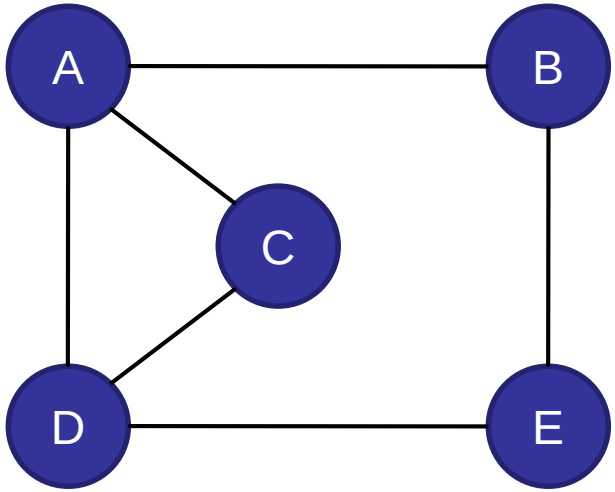
Redução de Problemas



Ciclo Hamiltoniano

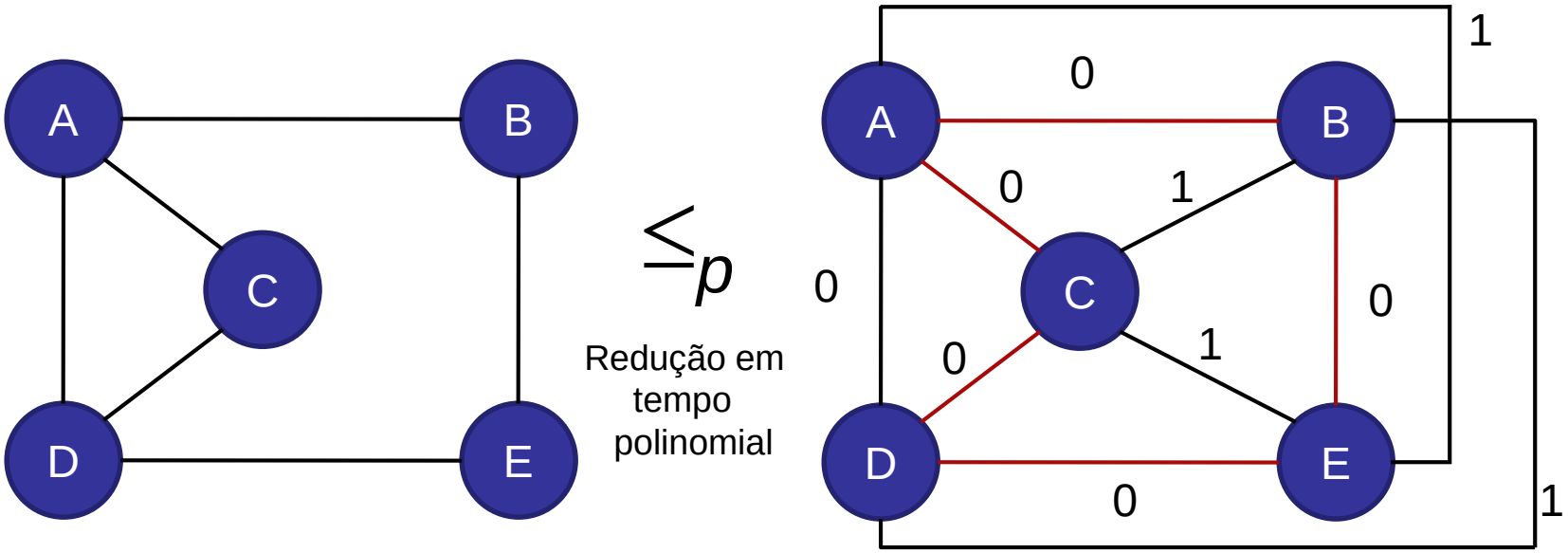
Um ***Ciclo Hamiltoniano*** em um grafo não orientado é um caminho que passa por cada vértice do grafo exatamente uma vez.

Problema do ***Ciclo Hamiltoniano***: Um grafo *G* possui um ciclo Hamiltoniano?



Redução do Problema do Ciclo Hamiltoniano ao Problema do Caixeiro Viajante

G $(G', 0)$



para cada vértice i

para cada vértice j

se $(i, j) \in H$ então $c(i, j) \leftarrow 0$

senão $c(i, j) \leftarrow 1$

UDESC

(SAT) Satisfazibilidade de Fórmulas Booleanas

O problema da ***Satisfazibilidade de fórmulas booleanas*** consiste em determinar se existe uma atribuição de valores booleanos, para as varáveis que ocorrem na fórmula, de tal forma que o resultado seja ***verdadeiro***.

Exemplo:

$$x_1 \wedge (x_2 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

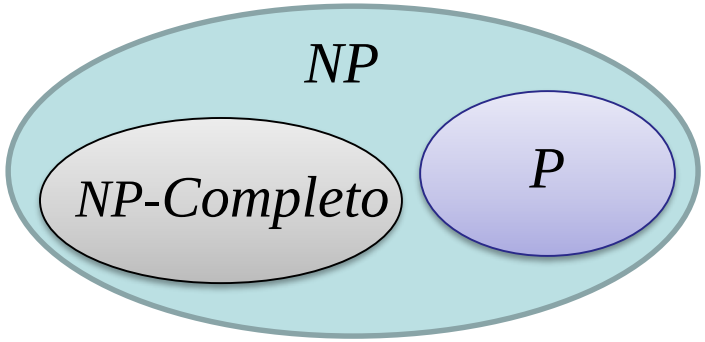
NP-Completo

Teorema de Cook: SAT está em *P* se e somente se *P* = *NP*.

Um problema X é ***NP-Completo*** se:

1. $X \in NP$
2. $X' \leq_p X$ para todo $X' \in NP$.

Possível relação entre as classes:



UDESC

Forma Normal Conjuntiva

Um ***literal*** é uma variável proposicional ou sua negação.

Uma formula booleana está na ***Forma Normal Conjuntiva*** (***CNF***) se é expressa por um grupo cláusulas AND, cada uma das quais formada por OR entre literais.

Uma fórmula booleana esta na ***k-CNF*** se cada cláusula possui exatamente *k* literais:

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

UDESC

3-CNF-SAT

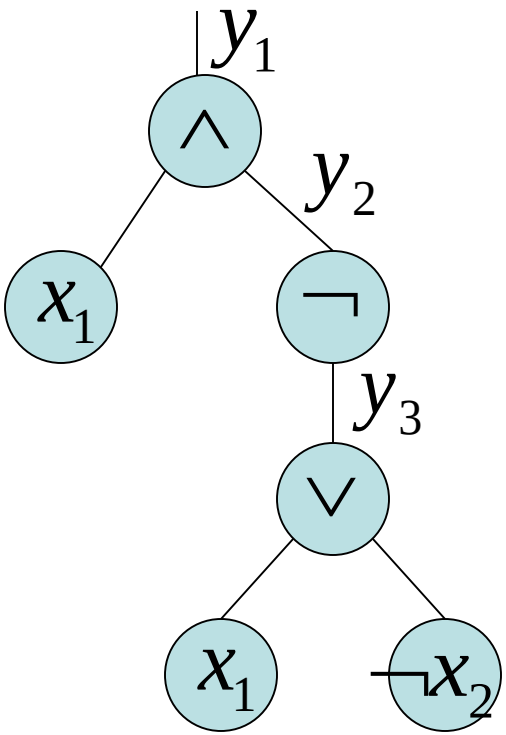
Verificar se uma fórmula booleana na 3-CNF é satisfazível.

3-CNF-SAT é *NP-Completo*:

- 3-CNF-SAT $\in$ *NP*.
- SAT $\leq_p$ 3-CNF-SAT.

$SAT \leq_p$ 3-CNF-SAT

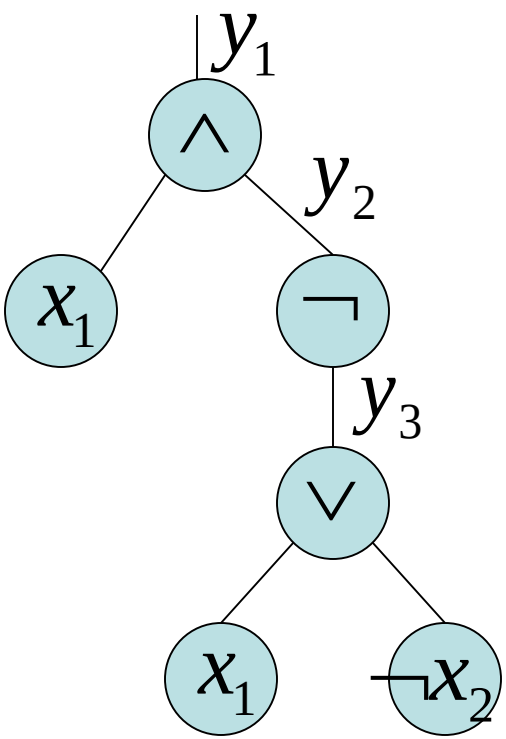
Dada uma fórmula booleana: $\phi = x_1 \wedge \neg(x_1 \vee \neg x_2)$



1. Construir uma árvore que represente à fórmula.
2. Introduzir uma variável y_i para a raiz e a saída de cada no interno.

$SAT \leq_p$ 3-CNF-SAT

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$



3. Reescrevemos a fórmula original como conjunções entre a variável raiz cláusulas que descrevem as operações de cada nó.

Introduz 1 variável e 1 cláusula para cada operador.

UDESC

$SAT \leq_p$ 3-CNF-SAT

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$

Para cada ϕ'_i construir uma tabela tabela verdade, usando as entradas que tornam $\neg \phi'_i$ verdade, construir uma forma normal disjuntiva para cada ϕ'_i .

$SAT \leq_p$ 3-CNF-SAT

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$

y_1	x_1	y_2	$\phi'_2 = y_1 \leftrightarrow (x_1 \wedge y_2)$
V	V	V	V
V	V	F	F
V	F	V	F
V	F	F	F
F	V	V	F
F	V	F	V
F	F	V	V
F	F	F	V

$$\neg\phi''_2 = (y_1 \wedge x_1 \wedge \neg y_2)$$
$$\vee (y_1 \wedge \neg x_1 \wedge y_2)$$
$$\vee (y_1 \wedge \neg x_1 \wedge \neg y_2)$$
$$\vee (\neg y_1 \wedge x_1 \wedge y_2)$$

Cada cláusula de ϕ' introduz no máximo 8 cláusulas em ϕ'', pois cada cláusula de ϕ' possui no máximo 3 variáveis.

UDESC

SAT $\leq_p$ 3-CNF-SAT

$$\neg\phi_2'' = (y_1 \wedge x_1 \wedge \neg y_2) \vee (y_1 \wedge \neg x_1 \wedge y_2) \vee$$
$$(y_1 \wedge \neg x_1 \wedge \neg y_2) \vee (\neg y_1 \wedge x_1 \wedge y_2)$$

Converter a fórmula para a CNF usando as leis de De Morgan:

$$\phi_2'' = (\neg y_1 \vee \neg x_1 \vee y_2) \wedge (\neg y_1 \vee x_1 \vee \neg y_2) \wedge$$
$$(\neg y_1 \vee x_1 \vee y_2) \wedge (y_1 \vee \neg x_1 \vee \neg y_2)$$

UDESC

$SAT \leq_p$ 3-CNF-SAT

O último passo faz com que cada cláusula tenha exatamente 3 literais, para isso usamos duas novas variáveis p e q. Para cada cláusula C_i em ϕ'' :

1. Se C_i tem 3 literais, simplesmente inclua C_i.
2. Se C_i tem 2 literais, $C_i = (l_1 \vee l_2)$, inclua:

   $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$
3. Se C_i tem 1 literal, l_1, inclua:

   $(l_1 \vee p \vee q) \wedge (l_1 \vee \neg p \vee \neg q) \wedge (l_1 \vee p \vee \neg q) \wedge (l_1 \vee \neg p \vee q)$

Introduz no máximo 4 cláusulas por cláusula em ϕ''.

UDESC

$SAT \leq_p$ 3-CNF-SAT

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$

$$\phi_1''' = (y_1 \vee p \vee q) \wedge (y_1 \vee \neg p \vee \neg q) \wedge (y_1 \vee p \vee \neg q) \wedge (y_1 \vee \neg p \vee q)$$

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$

$$(y_1 \vee p \vee q) \wedge (y_1 \vee \neg p \vee \neg q) \wedge (y_1 \vee p \vee \neg q) \wedge (y_1 \vee \neg p \vee q) \wedge$$

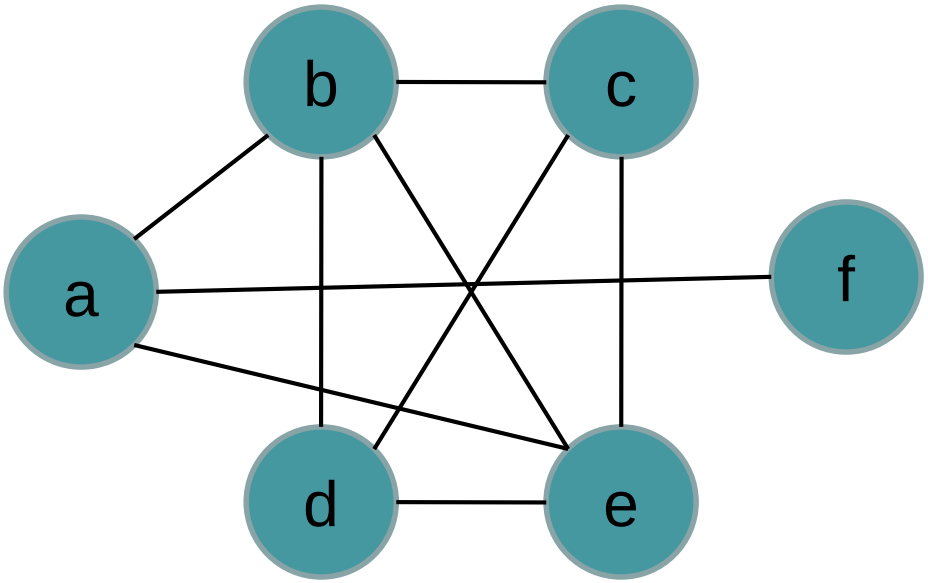
$$(\neg y_1 \vee \neg x_1 \vee y_2) \wedge (\neg y_1 \vee x_1 \vee \neg y_2) \wedge (\neg y_1 \vee x_1 \vee y_2) \wedge (y_1 \vee \neg x_1 \vee \neg y_2)$$

CLIQUE

Um *Clique* em um grafo não direcionado $G = (V, A)$ é um subconjunto de vértices $V' \subseteq V$, onde cada vértice está conectado por uma aresta. Ou seja um subgrafo completo.

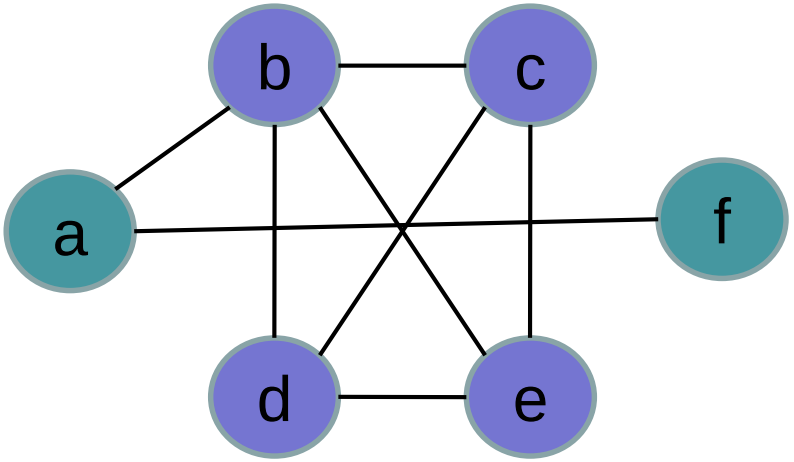
Versão de otimização: Encontrar o maior *Clique* possível.

Versão de decisão: Existe um *Clique* de tamanho $\geq k$?



CLIQUE

Clique $\in NP$



Dado um grafo $G = (V, A)$, a solução (certificado) V' e k

Verificar de $|V| \geq k$

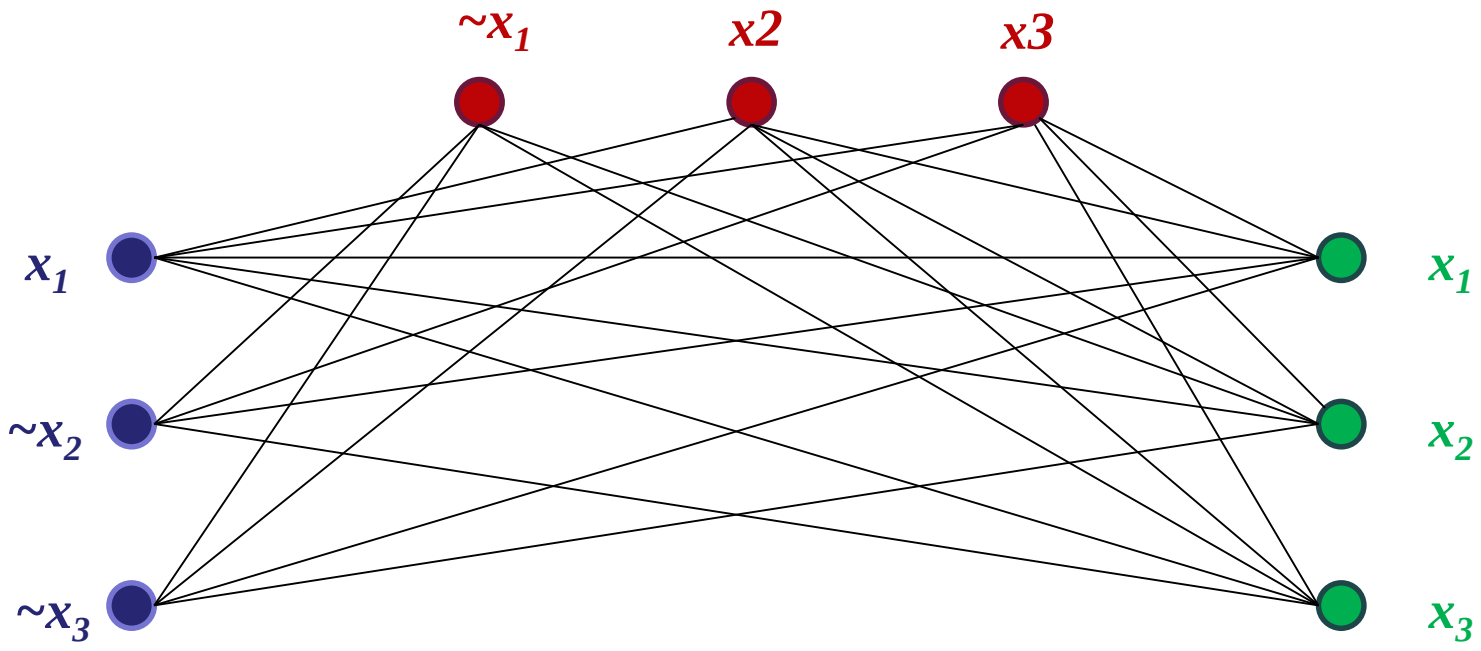
Para cada $u \in V'$

    Para cada $v \in V'$

        Se $u \neq v$ então verificar se $(u, v) \in A$

3-CNF-SAT $\leq_p$ CLIQUE

$\phi = (x_1 \vee \sim x_2 \vee \sim x_3) \wedge (\sim x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



Existe um clique de tamanho k? Sendo k o número de cláusulas.

UDESC

3-CNF-SAT $\leq_p$ CLIQUE

Dada uma instancia ϕ do problema 3-CNF-SAT, cada cláusula de ϕ gera 3 vértices, sendo que cada vértice corresponde a um literal da cláusula.

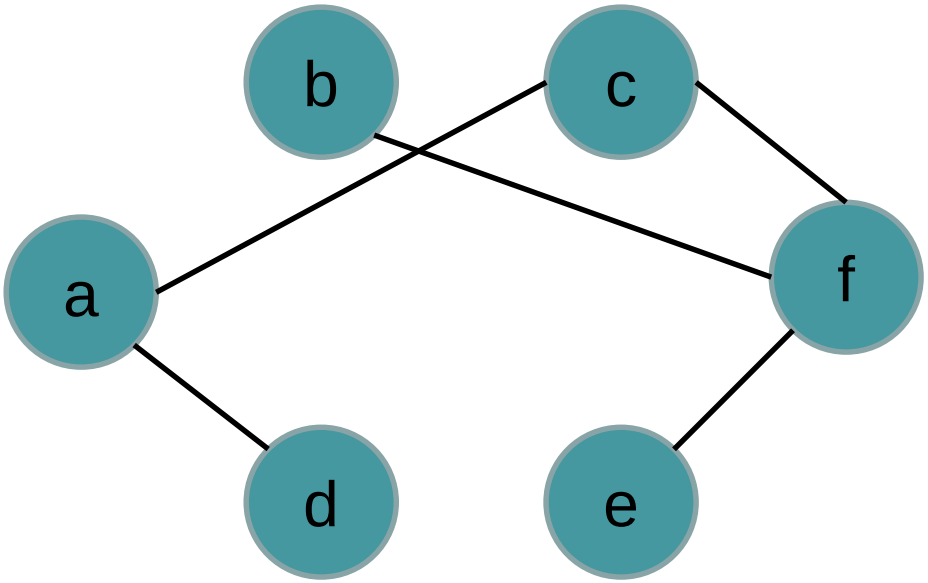
É adicionada uma aresta para cada par de vértices ***u*** e ***v*** se as duas condições a seguir forem satisfeitas:

- Se ***u*** e ***v*** não foram gerados a partir da mesma cláusula;
- Se ***u*** e ***v*** não foram gerados a partir de um literal que corresponde a uma variável e sua negação. Por exemplo, um vértice correspondente a variável $\boldsymbol{x_1}$ não pode ser conectado a um vértice correspondente a negação de $\boldsymbol{x_1}$.

UDESC

Cobertura de Vértices
(VERTEX-COVER)

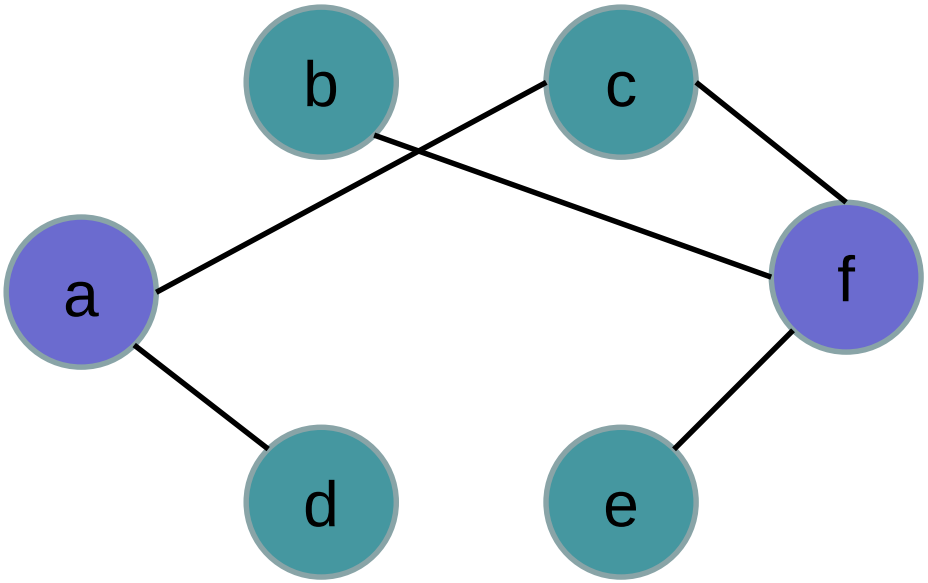
Uma *Cobertura de Vértices* de um grafo não orientado $G = (V, A)$ é um subconjunto $V' \subseteq V$ tal que se $(u, v) \in A$, então $u \in V'$ ou $v \in V'$.



Cobertura de Vértices
(VERTEX-COVER)

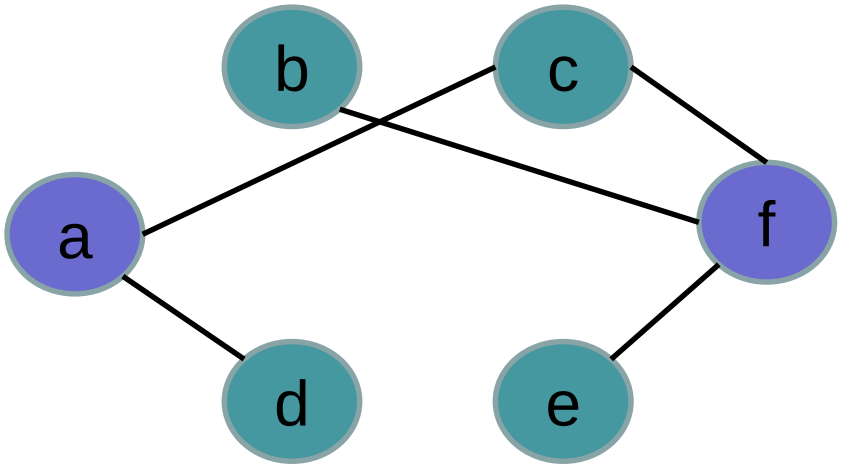
Versão de otimização: Encontrar menor Cobertura de Vértices.

Versão de decisão: Existe uma cobertura de tamanho *k*?



Cobertura de Vértices
(VERTEX-COVER)

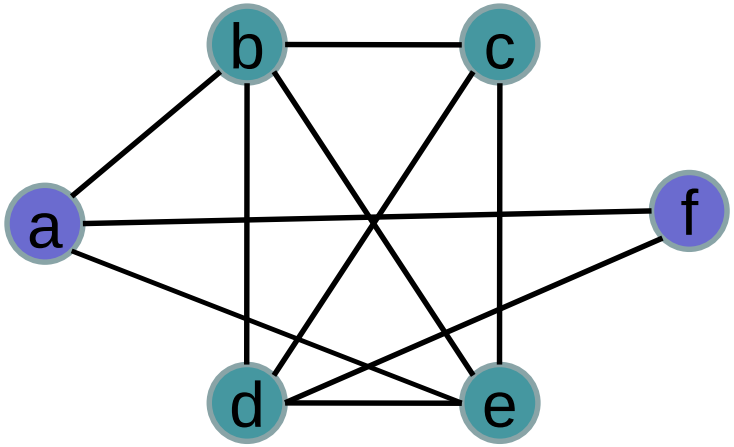
Cobertura de Vértices $\in$ *NP*.

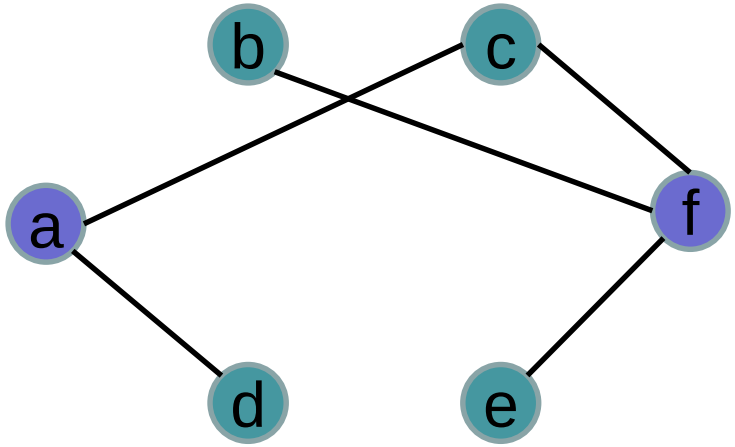


Dado um grafo $G = (V, A)$ e a solução (certificado) V'
Verificar de $|V| \geq k$
Para cada $(u, v) \in A$
    Verificar se $u \in V'$ ou $v \in V'$

UDESC

CLIQUE $\leq_p$ VERTEX-COVER





CLIQUE
Entrada (G, k), onde $G = (V, A)$

VERTEX-COVER
Entrada $(\overline{G}, |V| - k)$

UDESC

SUBSET-SUM

Dado um conjunto finito de de inteiros positivos S e um inteiro $t > 0$, determinar se existe um subconjunto $S' \subseteq S$ onde o somatório dos elementos de S' é igual a t.

$$\sum_{i=1}^{n} s'_i = t$$

UDESC

3-CNF-SAT $\leq_p$ SUBSET-SUM

$$(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

	x_1	x_2	x_3	C_1	C_2
v_1	1	0	0	0	1
v'_1	1	0	0	1	0
v_2	0	1	0	1	1
v'_2	0	1	0	0	0
v_3	0	0	1	0	0
v'_3	0	0	1	1	1
s_1	0	0	0	1	0
s'_1	0	0	0	2	0
s_2	0	0	0	0	1
s'_2	0	0	0	0	2
t	1	1	1	4	4

UDESC

Programação Dinâmica

Método de solução de problemas baseado no uso de tabelas.

Resolve problemas combinando as soluções de subproblemas. As soluções para cada um dos subproblemas são armezadas em uma tabela, dessa forma uma solução não necessita ser recalculada quando um subproblema ocorre repetidas vezes.

Programação Dinâmica

É indicada quando um subproblema ocorre repetidamente e sua solução pode ser reaproveitada.

Exemplo:

$$fib(n) = \begin{cases} n & \text{se } n = 0 \vee n = 1 \\ fib(n-1) + fib(n-2) & \text{se } n > 1 \end{cases}$$

UDESC

Programação Dinâmica

Abordagem *top-down* (*Memoization*)

Pode ser vista como o uso de recursão com apoio de tabelas:

- Como ocorre em algoritmos recursivos (divisão e conquista), um problema é resolvido dividindo-o em subproblemas menores, resolvendo esses subproblemas recursivamente e combinando suas soluções.
- A solução de cada subproblema é armazenada em uma tabela, dessa forma não é recalculada caso o subproblema ocorra repetidamente.

UDESC

Programação Dinâmica

Abordagem *top-down* (*Memoization*)

Fib(n)

    se $n \leq 1$ então

        retorne n

    senão

        se F[n] está indefinido

            F[n] ← Fib(n – 1) + Fib(n – 2)

        retorne F[n]

Programação Dinâmica

Abordagem *bottom-up*

Fib(n)

F[0] = 0

F[1] = 1

para $i \leftarrow 2$ até n

F[i] = F[$i - 2$] + F[$i - 1$]

retorne F[n]

UDESC

Programação Dinâmica

Change-Making Problem

Dado um conjunto de n moedas, cada uma com um valor x_i, e um valor t, achar o conjunto $\{f_1, f_2, \ldots, f_n\}$, onde f_i representa a quantidade de moedas de valor x_i, que minimiza:

$$\sum_{i=1}^{n} f_i$$

tal que:

$$\sum_{i=1}^{n} x_i f_i = t$$

Programação Dinâmica
Change-Making Problem

Imprimir-Troco (s, x, t)

  Enquanto $t > 0$

    Imprimir ($x[s[t]]$)

    $t \leftarrow t - x[s[t]]$

Onde $c[t]$ é o número mínimo de moedas para totalizar o valor t e $s[t]$ é o índice da última moeda que ocorre nessa solução.

Exemplo: $x = \{1, 2, 5\}$ e $t = 9$

t	*c*	*s*
0	0	-
1	1	1
2	1	2
3	2	2
4	2	2
5	1	3
6	2	3
7	2	3
8	3	3
9	3	3

UDESC

Programação Dinâmica
Change-Making Problem

```
Troco(x[1..n], t)
    c[0] ← 0
    para p ← 1 até t
        min ← ∞
        para i ← 1 até n
            se (x[i] ≤ p) e (c[p – x[i]] + 1 ≤ min) então
                min ← c[p – x[i]] + 1
                moeda ← i
        c[p] ← min
        s[p] ← moeda
    retorne (c, s)
```

UDESC

Programação Dinâmica (*Subset-Sum*)

Dado um conjunto de inteiros positivos, representados como um arranjo $S[1..n]$, e um inteiro t, existe algum subconjunto de S tal que a soma de seus elementos seja t.

$$SubsetS(i,t) = \begin{cases} Verdade & \text{se } t = 0 \\ Falsidade & \text{se } t < 0 \vee i > n \\ SubsetS(i+1,t) \vee SubsetS(i+1,t-x[i]) & \end{cases}$$

Programação Dinâmica (*Subset-Sum*)

Exemplo: $x = \{2, 3, 5\}$ e $t = 8$.

$$SubsetS(i,t) = \begin{cases} Verdade & \text{se } t = 0 \\ Falsidade & \text{se } t < 0 \vee i > n \\ SubsetS(i+1,t) \vee SubsetS(i+1,t-x[i]) \end{cases}$$

Programação Dinâmica (*Subset-Sum*)

SubsetSum($x[1..n]$, t)

S[$n + 1, 0$] ← Verdade

para j ← 1 até t

S[$n + 1, j$] ← Falsidade

para i ← n até 1

S[$i, 0$] ← Verdade

para j ← 1 até $x[i] - 1$

S[i, j] ← S[$i + 1, j$]

para j ← $x[i]$ até t

S[i, j] ← S[$i + 1, j$] v S[$i + 1, j - x[i]$]

retorne S[1,t]

Programação Dinâmica (*Subset-Sum*)

Exemplo: $x = \{1, 3, 5, 7\}$ e $t = 10$.

	0	1	2	3	4	5	6	7	8	9	10
1	V	V	F	V	V	V	V	V	V	V	V
2	V	F	F	V	F	V	F	V	V	F	V
3	V	F	F	F	F	V	F	V	F	F	F
4	V	F	F	F	F	F	F	V	F	F	F
5	V	F	F	F	F	F	F	F	F	F	F

UDESC

Algoritmos que Executam em Tempo Pseudo-Polinomial SUBSET-SUM

Usando programação dinâmica podemos implementar um algoritmo pseudo-polinomial com complexidade $O(nt)$, onde n é o numero de elemetos no conjunto e t o valor do somatório que se deseja alcançar.

Se restringirmos nossa atenção a instâncias do problema onde o valor de t é limitado por um polinómio existe uma solução eficiente.

UDESC

Algoritmos que Executam em Tempo Pseudo-Polinomial

Essa restrição pode ser bastante razoável na prática:

• Problemas onde é impossível a ocorrência de números muito grandes (*e.g.* problemas de escalonamento).

• Problemas onde o tamanho do número possa ser restrito ao tamanho da palavra do processador.

Note que esse não é o caso da redução do 3-CNF-SAT ao SUBSET-SUM, onde o valor de t cresce exponenciamente ao número de variáveis e cláusulas presentes na fórmula booleana.

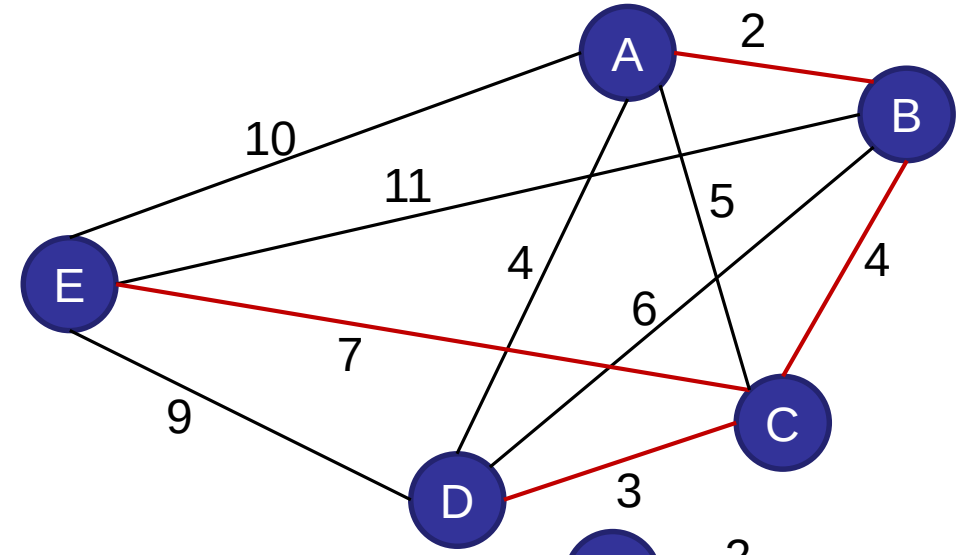
UDESC

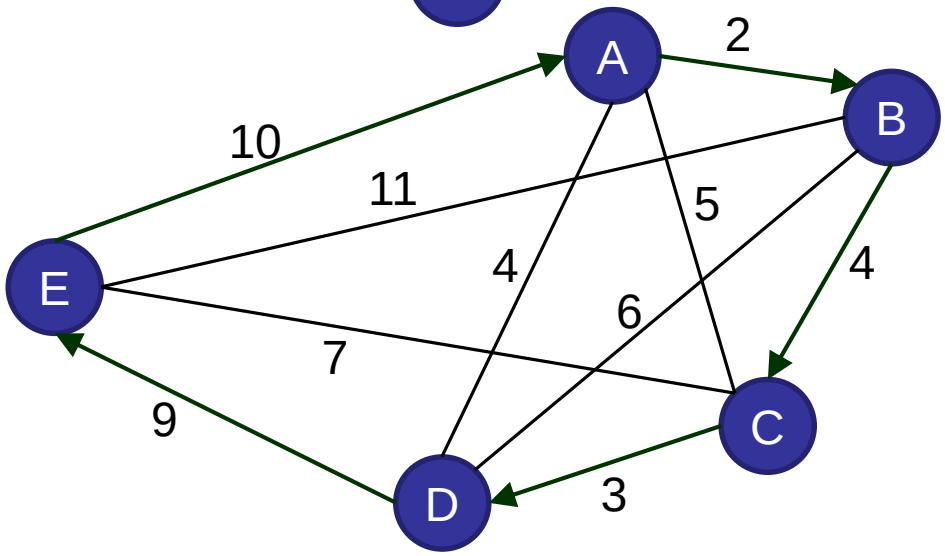
Heurísticas e Algoritmos de Aproximação

Procedimentos heurísticos são métodos que buscam soluções próximas a solução ótima de um problema. Utilizam alguma informação (ou intuição) sobre a instância do problema para resolvê-lo de forma eficiente.

Um ***algoritmo de aproximação***, além de uma solução eficiente, garante a qualidade da solução. É necessário provar a garantia de proximidade da solução ótima.

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 P ser a lista de vertices ordenados de acordo com a primeira visita, considerando um percurso em pré-ordem.

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

$c(T) < c(\textit{ótimo})$

$c(T \textit{ concatenado com } [r]) \leq 2c(\textit{ótimo})$

UDESC

Referências

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

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

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