Projeto e Análise de Algoritmos

A. G. Silva

Baseado nos materiais de
Souza, Silva, Lee, Rezende, Miyazawa – Unicamp
Miyazawa, Xavier – Unicamp
Feofiloff, Pina, Cris – USP

15 de junho de 2018

Conteúdo programático

- Introdução (4 horas/aula)
- Notação Assintótica e Crescimento de Funções (4 horas/aula)
- Recorrências (4 horas/aula)
- Divisão e Conquista (12 horas/aula)
- Grafos (4 horas/aula)
- Buscas (4 horas/aula)
- Algoritmos Gulosos (8 horas aula)
- Programação Dinâmica (8 horas/aula)
- NP-Completude e Reduções (6 horas/aula)
- Algoritmos Aproximados e Busca Heurística (6 horas/aula)

Cronograma

- **02mar** – Apresentação da disciplina. Introdução.
- **09mar** – *Prova de proficiência/dispensa.*
- **16mar** – Notação assintótica. Recorrências.
- **23mar** – *Dia não letivo.* Exercícios.
- **30mar** – *Dia não letivo.* Exercícios.
- **06abr** – Recorrências. Divisão e conquista.
- **13abr** – Divisão e conquista. Ordenação.
- **20abr** – Ordenação. Estatística de ordem.
- **27abr** – **Primeira avaliação.**
- **04mai** – Estatística de ordem. Grafos. Buscas.
- **11mai** – Buscas. Algoritmos gulosos.
- **18mai** – Algoritmos gulosos.
- **25mai** – Algoritmos gulosos. Programação dinâmica.
- **01jun** – *Dia não letivo.* Exercícios.
- **08jun** – *Semana Acadêmica.* Exercícios.
- **15jun** – Programação dinâmica. NP-Completude e reduções.
- **22jun** – Exercícios *(copa)*.
- **29jun** – **Segunda avaliação.**
- **06jul** – **Avaliação substitutiva** *(opcional)*.

Programação Dinâmica – O Problema Binário da Mochila

O Problema Binário da Mochila

O Problema da Mochila

Dada uma mochila de capacidade W (inteiro) e um conjunto de n itens com tamanho w_i (inteiro) e valor c_i associado a cada item i, queremos determinar quais itens devem ser colocados na mochila de modo a **maximizar** o valor total transportado, respeitando sua capacidade.

- Podemos fazer as seguintes suposições:
  - $\sum_{i=1}^{n} w_i > W$;
  - $0 < w_i \leq W$, para todo $i = 1, \ldots, n$.

O Problema Binário da Mochila

- Podemos formular o problema da mochila com Programação Linear Inteira:
  - Criamos uma variável x_i para cada item: $x_i = 1$ se o item i estiver na solução ótima e $x_i = 0$ caso contrário.
  - A modelagem do problema é simples:

$$\max \sum_{i=1}^{n} c_i x_i \tag{1}$$

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

$$x_i \in \{0, 1\} \tag{3}$$

- (1) é a função objetivo e (2-3) o conjunto de restrições.

O Problema Binário da Mochila

- Como podemos projetar um algoritmo para resolver o problema?
- Existem 2^n possíveis subconjuntos de itens: um algoritmo de força bruta é impraticável!
- É um problema de otimização. Será que tem subestrutura ótima?
- Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n - 1$ primeiros itens.
- Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n - 1$ primeiros itens.

O Problema Binário da Mochila

- Seja $z[k, d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original.
- A fórmula de recorrência para computar $z[k, d]$ para todo valor de d e k é:

$$z[0, d] = 0$$

$$z[k, 0] = 0$$

$$z[k, d] = \begin{cases} z[k-1, d], & \text{se } w_k > d \\ \max\{z[k-1, d], z[k-1, d-w_k] + c_k\}, & \text{se } w_k \leq d \end{cases}$$

O Problema Binário da Mochila - Complexidade Recursão

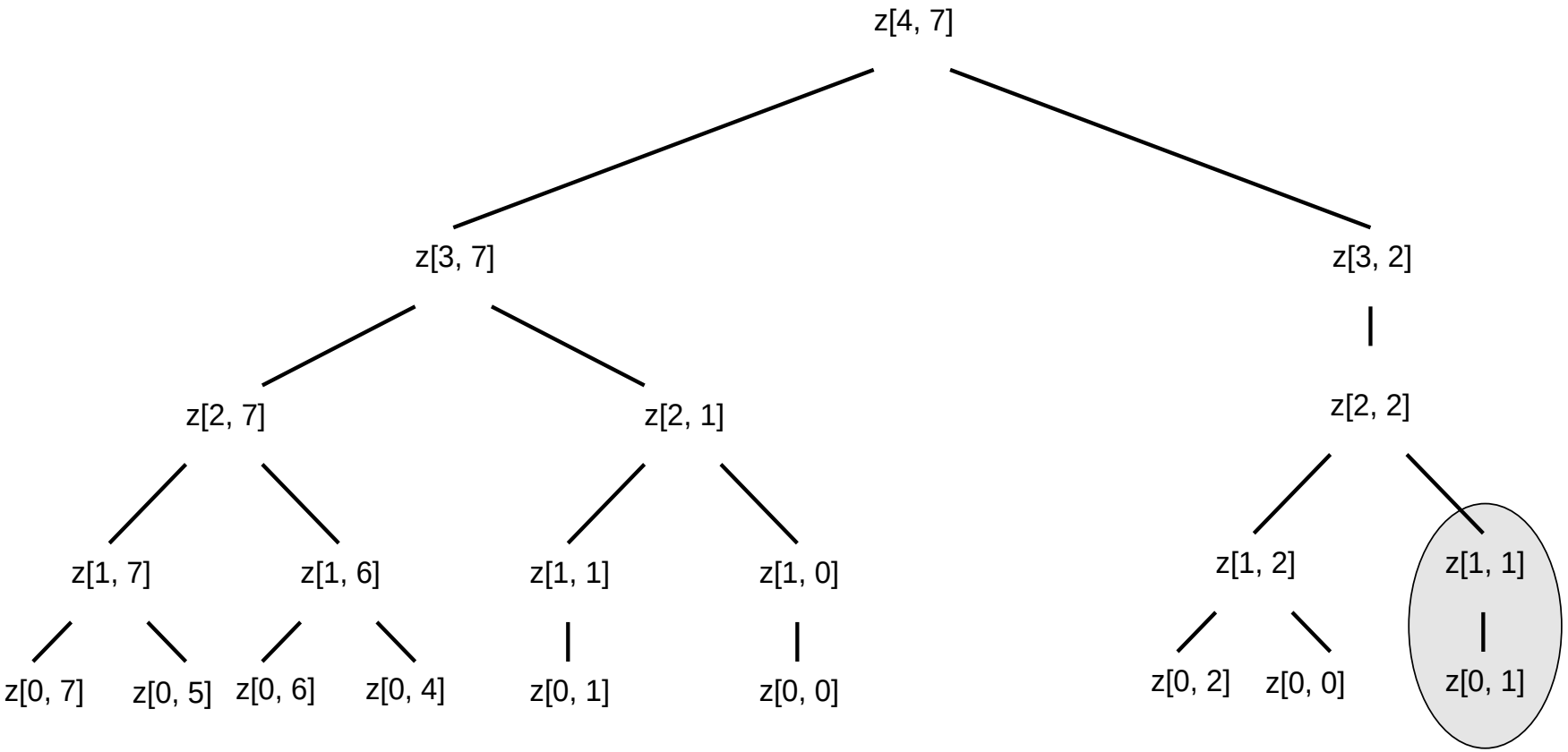
- A complexidade do algoritmo recursivo para este problema no pior caso é dada pela recorrência:

$$T(k,d) = \begin{cases} 1, & k = 0 \text{ ou } d = 0 \\ T(k-1,d) + T(k-1,d-w_k) + 1 & k > 0 \text{ e } d > 0. \end{cases}$$

- Portanto, no pior caso, o algoritmo recursivo tem complexidade $\Omega(2^n)$. É impraticável!
- Mas há sobreposição de subproblemas: o recálculo de subproblemas pode ser evitado!

Mochila - Sobreposição de Subproblemas

- Considere vetor de tamanhos $w = \{2, 1, 6, 5\}$ e capacidade da mochila $W = 7$. A árvore de recursão seria:

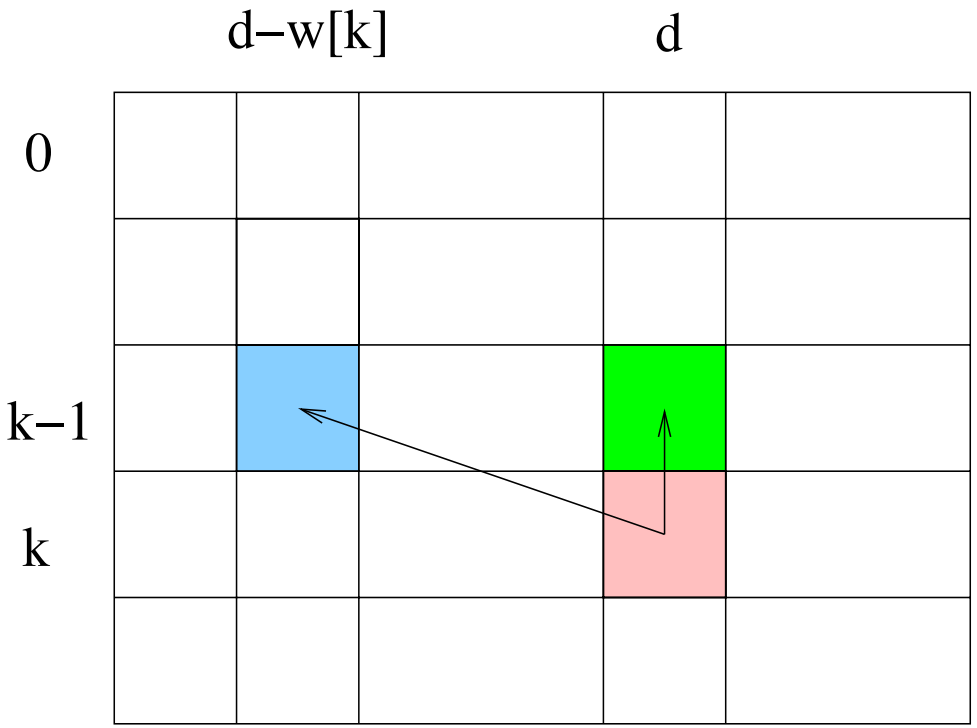


- O subproblema $z[1, 1]$ é computado duas vezes.

Mochila - Programação Dinâmica

- O número total máximo de subproblemas a serem computados é nW.
- Isso porque tanto o tamanho dos itens quanto a capacidade da mochila são inteiros!
- Podemos então usar programação dinâmica para evitar o recálculo de subproblemas.
- Como o cálculo de $z[k, d]$ depende de $z[k-1, d]$ e $z[k-1, d-w_k]$, preenchemos a tabela linha a linha.

Mochila



$$z[k,d] = \max \{ z[k-1,d] , z[k-1,d-w[k]] + c[k] \}$$

O Problema Binário da Mochila - Algoritmo

Mochila(c, w, W, n)

▷ **Entrada:** Vetores c e w com valor e tamanho de cada item, capacidade W da mochila e número de itens n.

▷ **Saída:** O valor máximo do total de itens colocados na mochila.

1. **para** $d := 0$ **até** W **faça** $z[0, d] := 0$
2. **para** $k := 1$ **até** n **faça** $z[k, 0] := 0$
3. **para** $k := 1$ **até** n **faça**
4. **para** $d := 1$ **até** W **faça**
5. $z[k, d] := z[k-1, d]$
6. **se** $w_k \leq d$ **e** $c_k + z[k-1, d-w_k] > z[k, d]$ **então**
7. $z[k, d] := c_k + z[k-1, d-w_k]$
8. **devolva** $(z[n, W])$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Complexidade

- Claramente, a complexidade do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É um algoritmo pseudo-polinomial: sua complexidade depende do valor de W, parte da entrada do problema.
- O algoritmo não devolve o subconjunto de valor total máximo, apenas o valor máximo.
- É fácil recuperar o subconjunto a partir da tabela z preenchida.

Mochila - Recuperação da Solução

MochilaSolucao(z, n, W)

▷ **Entrada:** Tabela z preenchida, capacidade W da mochila e número de itens n.

▷ **Saída:** O vetor x que indica os itens colocados na mochila.

para $i := 1$ **até** n **faça** $x[i] := 0$

MochilaSolucaoAux(x, z, n, W)

devolva (x)

MochilaSolucaoAux(x, z, k, d)

se $k \neq 0$ **então**

  se $z[k, d] = z[k-1, d]$ **então**

    $x[k] := 0$; *MochilaSolucaoAux*$(x, z, k-1, d)$

  senão

    $x[k] := 1$; *MochilaSolucaoAux*$(x, z, k-1, d-w_k)$

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

$x[1] = x[4] = 1\ ,\ x[2] = x[3] = 0$

Mochila - Complexidade

- O algoritmo de recuperação da solução tem complexidade $O(n)$.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É possível economizar memória, registrando duas linhas: a que está sendo preenchida e a anterior. Mas isso inviabiliza a recuperação da solução.

Programação Dinâmica – Subcadeia Comum Máxima

Subcadeia comum máxima

Definição: Subcadeia

Dada uma cadeia $S = a_1 \ldots a_n$, dizemos que $S' = b_1 \ldots b_p$ é uma *subcadeia* de S se existem p índices $i(j)$ satisfazendo:

(a) $i(j) \in \{1, \ldots, n\}$ para todo $j \in \{1, \ldots, p\}$;

(b) $i(j) < i(j+1)$ para todo $j \in \{1, \ldots, p-1\}$;

(c) $b_j = a_{i(j)}$ para todo $j \in \{1, \ldots, p\}$.

- **Exemplo:** $S = ABCDEFG$ e $S' = ADFG$.

Problema da Subcadeia Comum Máxima

Dadas duas cadeias de caracteres X e Y de um alfabeto Σ, determinar a maior subcadeia comum de X e Y

Subcadeia comum máxima (cont.)

- É um problema de otimização. Será que tem subestrutura ótima?
- **Notação:** Seja S uma cadeia de tamanho n. Para todo $i = 1, \ldots, n$, o prefixo de tamanho i de S será denotado por S_i.
- **Exemplo:** Para $S = ABCDEFG$, $S_2 = AB$ e $S_4 = ABCD$.
- **Definição:** $c[i, j]$ é o tamanho da subcadeia comum máxima dos prefixos X_i e Y_j. Logo, se $|X| = m$ e $|Y| = n$, $c[m, n]$ é o valor ótimo.

Subcadeia comum máxima (cont.)

- **Teorema (subestrutura ótima):** Seja $Z = z_1 \ldots z_k$ a subcadeia comum máxima de $X = x_1 \ldots x_m$ e $Y = y_1 \ldots y_n$, denotado por $Z = \mathtt{SCM}(X, Y)$.
  1. Se $x_m = y_n$ então $z_k = x_m = y_n$ e $Z_{k-1} = \mathtt{SCM}(X_{m-1}, Y_{n-1})$.
  2. Se $x_m \neq y_n$ então $z_k \neq x_m$ implica que $Z = \mathtt{SCM}(X_{m-1}, Y)$.
  3. Se $x_m \neq y_n$ então $z_k \neq y_n$ implica que $Z = \mathtt{SCM}(X, Y_{n-1})$.

- **Fórmula de Recorrência:**

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Subcadeia comum máxima (cont.)

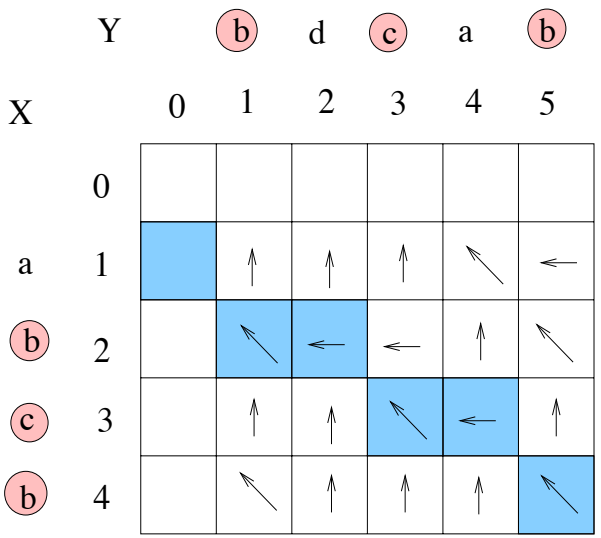
SCM(X, m, Y, n, c, b)

```
01.  para i = 0 até m faça c[i,0] := 0
02.  para j = 1 até n faça c[0,j] := 0
03.  para i = 1 até m faça
04.      para j = 1 até n faça
05.          se x_i = y_j então
06.              c[i,j] := c[i-1,j-1] + 1 ; b[i,j] := "↖"
07.          senão
08.              se c[i,j-1] > c[i-1,j] então
09.                  c[i,j] := c[i,j-1] ; b[i,j] := "←"
10.              senão
11.                  c[i,j] := c[i-1,j]; b[i,j] := "↑";
12.  devolva (c[m,n], b).
```

Subcadeia comum máxima - Exemplo

- **Exemplo:** $X = abcb$ e $Y = bdcab$, $m = 4$ e $n = 5$.

		Y	b	d	c	a	b
		0	1	2	3	4	5
X	0	0	0	0	0	0	0
a	1	0	0	0	0	1	1
b	2	0	1	1	1	1	2
c	3	0	1	1	2	2	2
b	4	0	1	1	2	2	3



Subcadeia comum máxima - Complexidade

- Claramente, a complexidade do algoritmo é $O(mn)$.
- O algoritmo não encontra a subcadeia comum de tamanho máximo, apenas seu tamanho.
- Com a tabela b preenchida, é fácil encontrar a subcadeia comum máxima.

Subcadeia comum máxima (cont.)

- Para recuperar a solução, basta chamar $Recupera\_MSC(b, X, m, n)$.

Recupera_SCM(b, X, i, j)

1. **se** $i = 0$ e $j = 0$ **então devolva**
2. **se** $b[i,j] =$ "$\nwarrow$" **então**
3. $Recupera\_MSC(b, X, i-1, j-1)$; **imprima** x_i
4. **senão**
5. **se** $b[i,j] =$ "$\uparrow$" **então**
6. $Recupera\_MSC(b, X, i-1, j)$
7. **senão**
8. $Recupera\_MSC(b, X, i, j-1)$

Subcadeia comum máxima - Complexidade

- A determinação da subcadeia comum máxima é feita em tempo $O(m+n)$ no pior caso.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da subcadeia comum máxima é $O(mn)$.
- Note que a tabela b é dispensável, podemos economizar memória recuperando a solução a partir da tabela c. Ainda assim, o gasto de memória seria $O(mn)$.
- Caso não haja interesse em determinar a subcadeia comum máxima, mas apenas seu tamanho, é possível reduzir o gasto de memória para $O(\min\{n,m\})$: basta registrar apenas a linha da tabela sendo preenchida e a anterior.

NP-Completude e Reduções

Histórico

- O trabalho de Cook de 1971 publicado no congresso STOC mostrou:
  - Todos os problemas da classe NP podem ser reduzidos em tempo polinomial para o problema de Satifatibilidade Booleana (SAT).
  - Ou seja se tivermos um algoritmo rápido (polinomial) para o SAT teremos um algoritmo rápido para todos os problemas em NP!
  - Este é o primeiro problema NP-Completo.
- Em 1972 Richard Karp mostrou como reduzir em tempo polinomial o SAT para outros 21 problemas importantes.
- Até hoje ninguém conseguiu encontrar um algoritmo polinomial para qualquer um dos problemas em NP-Completo!
- Conjectura: P=NP ? Prêmio de 1 milhão de US$.

21 Problemas NP-Completos provados por Karp

- *Satisfatibilidade* em forma normal conjuntiva
- *Programação Inteira 0-1*
- *Clique*
- *Empacotamento de Conjuntos*
- *Cobertura de Vértices*
- *Cobertura de Conjuntos*
- *Feedback Node Set*
- *Feedback Arc Set*
- *Circuito Hamiltoniano em grafo orientado*
- *Circuito Hamiltoniano em grafo não orientado*
- *3-Satisfatibilidade*
- *Coloração de Grafos*
- *Partição em Cliques*
- *Cobertura Exata*
- *Transversal*
- *Árvore de Steiner*
- *Emparelhamento Tridimensional*
- *Mochila*
- *Sequenciamento de Tarefas*
- *Partição*
- *Corte Máximo*

Comparando tempos polinomiais e exponenciais

$f(n)$	$n = 20$	$n = 40$	$n = 60$	$n = 80$	$n = 100$
n	$2{,}0\times10^{-11}$seg	$4{,}0\times10^{-11}$seg	$6{,}0\times10^{-11}$seg	$8{,}0\times10^{-11}$seg	$1{,}0\times10^{-10}$seg
n^2	$4{,}0\times10^{-10}$seg	$1{,}6\times10^{-9}$seg	$3{,}6\times10^{-9}$seg	$6{,}4\times10^{-9}$seg	$1{,}0\times10^{-8}$seg
n^3	$8{,}0\times10^{-9}$seg	$6{,}4\times10^{-8}$seg	$2{,}2\times10^{-7}$seg	$5{,}1\times10^{-7}$seg	$1{,}0\times10^{-6}$seg
n^5	$2{,}2\times10^{-6}$seg	$1{,}0\times10^{-4}$seg	$7{,}8\times10^{-4}$seg	$3{,}3\times10^{-3}$seg	$1{,}0\times10^{-2}$seg
2^n	$1{,}0\times10^{-6}$seg	1,0seg	13,3dias	$1{,}3\times10^{5}$séc	$1{,}4\times10^{11}$séc
3^n	$3{,}4\times10^{-3}$seg	140,7dias	$1{,}3\times10^{7}$séc	$1{,}7\times10^{19}$séc	$5{,}9\times10^{28}$séc

Supondo um computador com velocidade de 1 Terahertz (mil vezes mais rápido que um computador de 1 Gigahertz).

Comparando tempos polinomiais e exponenciais

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Fixando o tempo de execução

Histórico

- Desde então muitos outros problemas práticos foram demonstrados como pertencentes a classe NP-Completo.
- Estas demonstrações são feitas utilizando-se reduções.
- Veremos as principais classes de complexidade: P, NP, NP-Difícil e NP-Completo

Problemas práticos

Muitas vezes problemas NP-completos são casos particulares ou podem ser reduzidos facilmente para outros de carater mais prático, conhecidos como NP-difíceis

- Problema do Caixeiro Viajante
- Atribuição de Freqüências em Telefonia Celular
- Empacotamento de Objetos em Contêineres
- Escalonamento de Funcionários em Turnos de Trabalho
- Escalonamento de Tarefas em Computadores
- Classificação de Objetos
- Coloração de Mapas
- Projetos de Redes de Computadores
- Vários e vários outros problemas práticos...

O que acontece se nos depararmos com um problema NP-difícil ?

- Você pode buscar por boas soluções mas sem nenhuma garantia de tempo ou da qualidade da solução (heurísticas).
- Tentar achar soluções boas rapidamente com garantia de qualidade da solução obtida (algoritmos aproximados).
- Tentar resolver de forma ótima o problema, mas para instâncias pequenas (backtracking e branch-and-bound).
- Tentar resolver de forma ótima combinando com técnicas de programação linear (programação linear inteira)

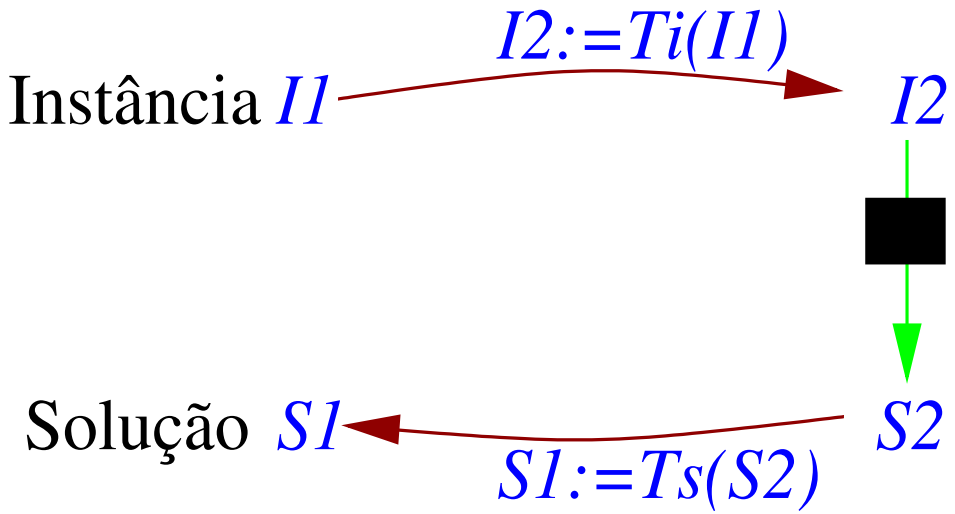
Reduções

- As vezes é fácil mapear instâncias de um problema P_1 em instâncias de um outro problema P_2 que já sabemos resolver.
- Neste caso podemos usar o resolvedor R_2 de P_2 como uma caixa preta para resolver instâncias de P_1.
- Note que é necessário transformar a solução dada para o problema P_2 em uma solução para P_1.
- Este processo é conhecido como redução de P_1 para P_2 e denotaremos por $P_1 \triangleright P_2$.

Redução entre problemas

P_1 é redutível a P_2 ($P_1 \triangleright P_2$) se

- $\exists$ *Ti* que transforma instância I_1 de P_1 para instância I_2 de P_2
- $\exists$ *Ts* que transforma solução S_2 de I_2 para solução S_1 de I_1



Claramente vale transitividade:

Se $P_1 \triangleright P_2$ e $P_2 \triangleright P_3$ então $P_1 \triangleright P_3$

Redução entre problemas

Usaremos as reduções para:

- Resolver um problema P_1 supondo que temos a resposta para um problema P_2.
- Mostrar que se P_1 for “difícil” e a redução for suficientemente rápida, é possível mostrar que P_2 também é “difícil”.

Redução 1

Problema (de Edição de String - ES)

As seguintes operações são possíveis em strings:

- *Inserção de um caracter*
- *Remoção de um caracter*
- *Troca de um caracter por outro*

Dados strings A e B, transformar A em B com o menor número de operações.

Redução 1

Exemplo

Se $A = babb$ e $B = bbc$, transformamos A em B com duas operações:

$$babb \Downarrow \text{ Remover } a$$
$$bbb \Downarrow \text{ Trocar último } b \text{ por } c$$
$$bbc$$

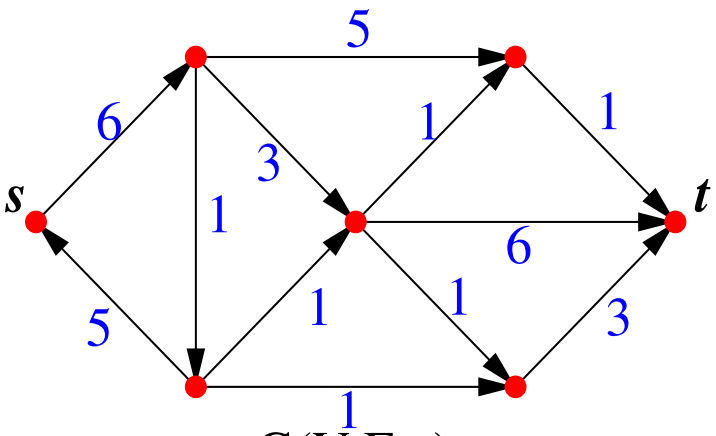
Exercício

O Problema de Edição de String pode ser resolvido por programação dinâmica.

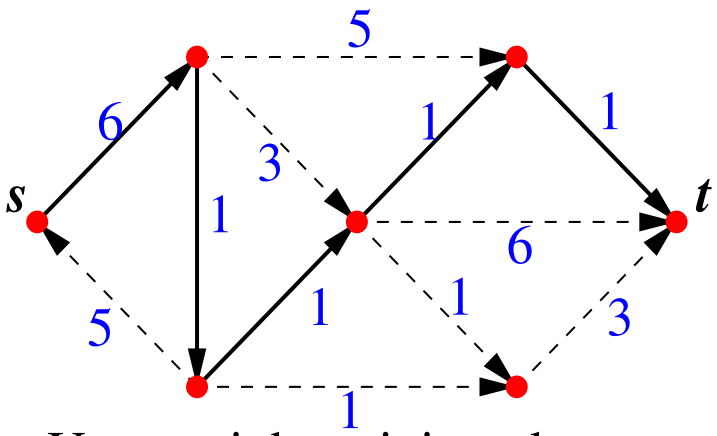
Redução 1

Problema (do Caminho Mínimo em Grafo Orientado)

Dado grafo orientado $G(V, E)$, onde cada aresta ij possui custo $c_{ij} > 0$, e vértices s e t, , encontrar um caminho de custo total mínimo de s a t em G.



G(V,E,c)



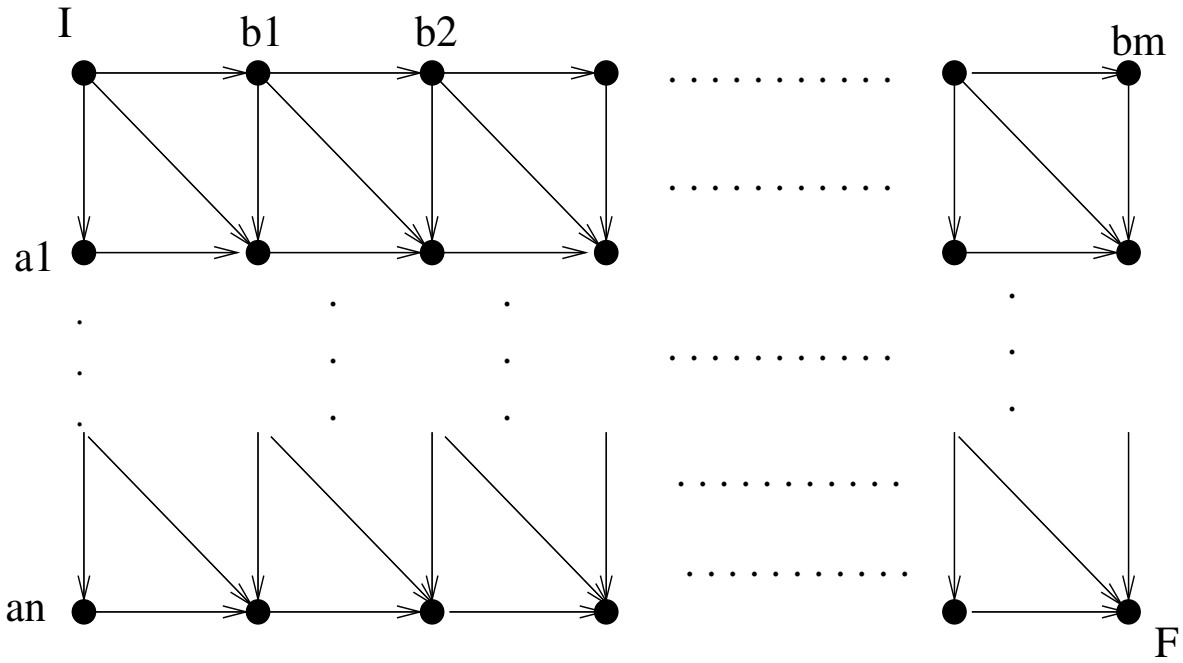
Um caminho minimo de s para t

Redução 1

Proposição

Problema de Edição de String $\triangleright$ *Problema do Caminho Mínimo em grafos orientados.*

Prova. Dados strings $A = a_1 a_2 \ldots a_n$ e $B = b_1 b_2 \ldots b_m$ (instância do Problema ES) construimos um grafo da seguinte forma:



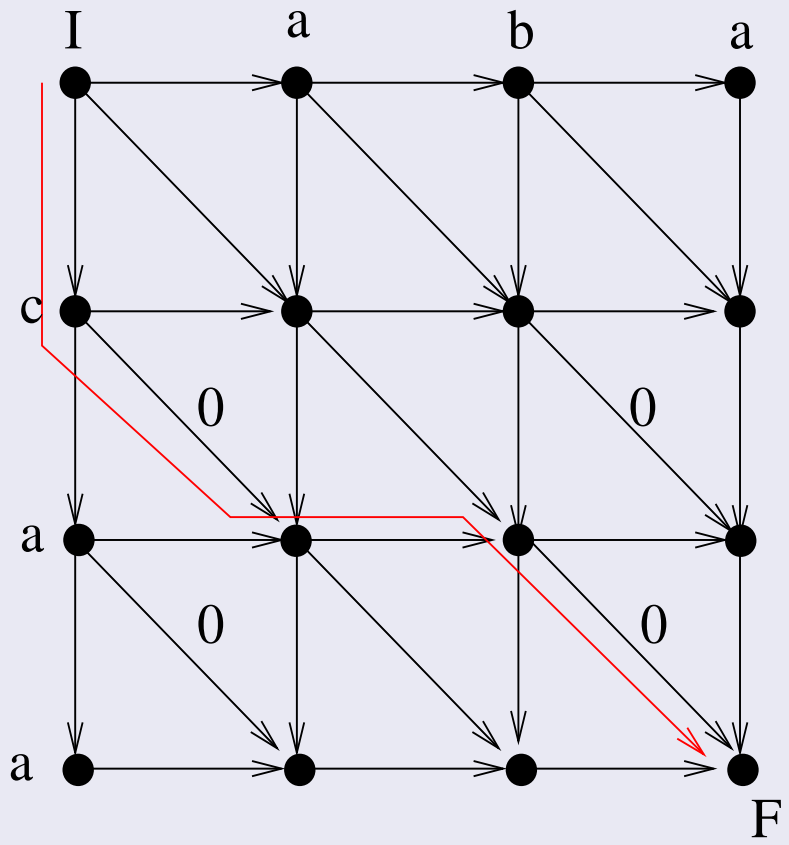
Redução 1

- Arestas horizontais correspondem a inserção de um caractere e possuem custo 1.
- Arestas verticais correspondem a remoção de um caractere e possuem custo 1.
- Arestas diagonais correspondem a uma troca e tem custo 1 caso os caracteres sejam diferentes, e 0 caso sejam iguais.
- O problema é encontrar um caminho mínimo do vértice I ao F.

Redução 1

Exemplo

Dados strings $A = caa$ e $B = aba$ construa grafo G:



custo de edição é $2 = 1 + 0 + 1 + 0$

Redução 1

Temos que mostrar que um caminho mínimo em G de I até F corresponde a uma edição mínima.

- Dado uma edição mínima, a partir do caracter vazio temos 4 opções (inserir, remover, trocar/match) que correspondem as arestas no grafo.
- Uma edição de strings corresponde a um caminho e este por sua vez corresponde a uma edição.
- Portanto um caminho mínimo será uma edição mínima.

Redução 2

Problema (do Triângulo em um Grafo)

Dado grafo $G = (V, E)$ não-orientado, decidir se G tem um triângulo (subgrafo completo de 3 vértices).

Proposição

O Problema dos Triângulo em um grafo pode ser resolvido em tempo $O(n^3)$.

Prova. Basta verificar todos os $\binom{n}{3}$ subconjuntos de 3 vértices de G. □

Redução 2

Vamos reduzir o Problema do Triângulo em um Grafo para o seguinte problema

Problema (Multiplicação de Matrizes)

Dadas matrizes A e B, computar a matriz $C = A \cdot B$.

Teorema (Strassen'69)

Existe algoritmo que computa multiplicação de matrizes de ordem n com complexidade de tempo $O(n^{2,807})$.

Teorema (Coppersmith-Winograd'90)

Existe algoritmo que computa multiplicação de matrizes de ordem n com complexidade de tempo $O(n^{2,376})$.

Redução 2

Definição

Dado grafo $G = (V, E)$, onde $V = \{1, \ldots, n\}$, a matriz de adjacência A de G é uma matriz de ordem $n \times n$ onde

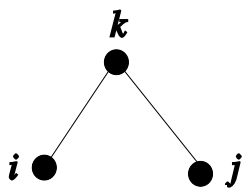
$$A[i,j] = \begin{cases} 1 & \text{se } \{i,j\} \in E \\ 0 & \text{caso contrário.} \end{cases}$$

O que acontece se calcularmos A^2 ? Sabemos que

$$A^2[i,j] = \sum_{k=1}^{n} A[i,k] \cdot A[k,j]$$

Portanto $A^2[i,j] > 0 \Leftrightarrow$ existe índice k tal que $A[i,k] = 1$ e $A[k,j] = 1$.
I.e.,

$A^2[i,j] > 0 \Leftrightarrow$ existe k tal que



Redução 2

Proposição

O Problema do Triângulo em um grafo G pode ser resolvido em tempo $O(n^{2,376})$.

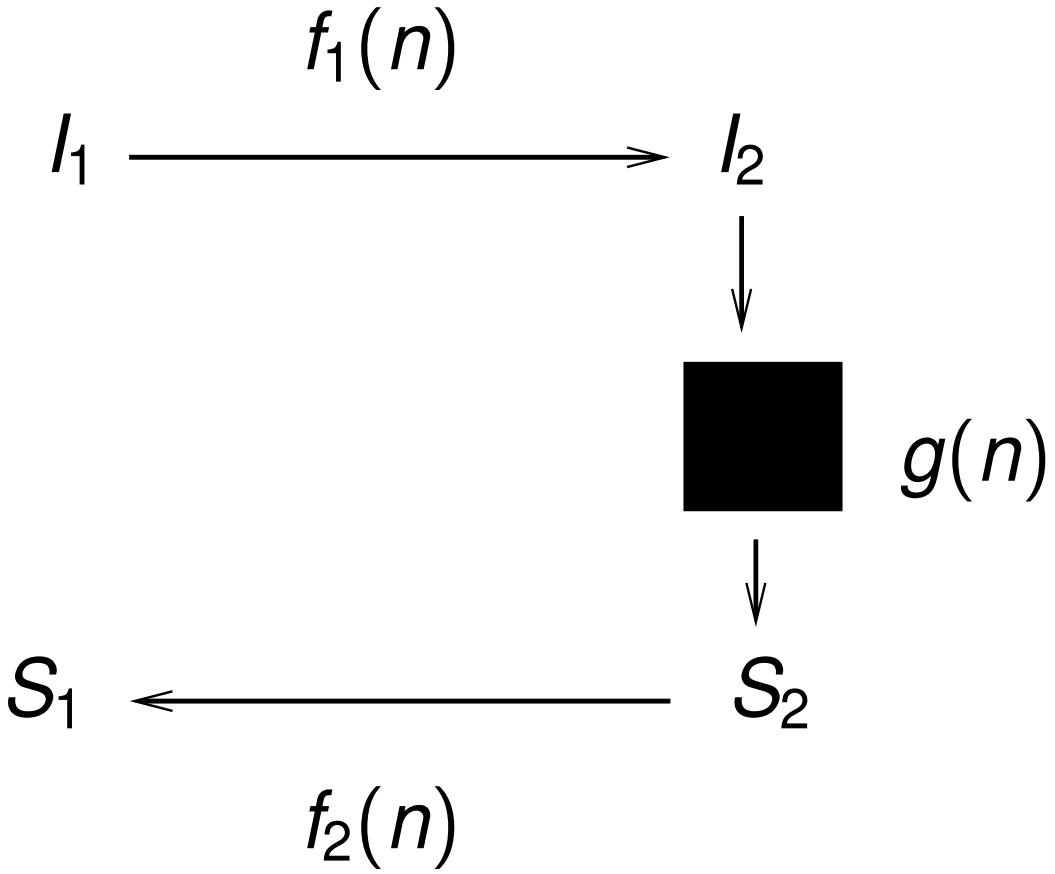
Prova. Seja A matriz de incidência de G.

- Compute A^2 em tempo $O(n^{2,376})$.
- Para cada posição (i, j), verifique em A e A^2 se
$$A[i, j] = 1 \text{ e } A^2[i, j] > 0.$$
- Caso se verifique para algum par, temos um triângulo em G.

□

Reduções e Cotas

- Suponha que temos uma redução de um problema P_1 para um problema P_2.



Reduções e Cotas

- $f_1(n)$ é o tempo para transformar instância P_1 para instância de P_2.
- $g(n)$ é o tempo para resolver o problema P_2.
- $f_2(n)$ é o tempo para transformar solução de P_2 em solução de P_1.
- Cota superior para P_1 será $f_1(n) + f_2(n) + g(n)$.
- No caso especial em que $g(n) = \Omega(f_1(n) + f_2(n))$ a cota superior para P_1 é $O(g(n))$.

Reduções e Cotas

- Se conhecemos cota inferior $\Omega(I(n))$ para P_1 e
- se $f_1(n) + f_2(n) = o(I(n))$ então $I(n)$ também é uma cota inferior para P_2.

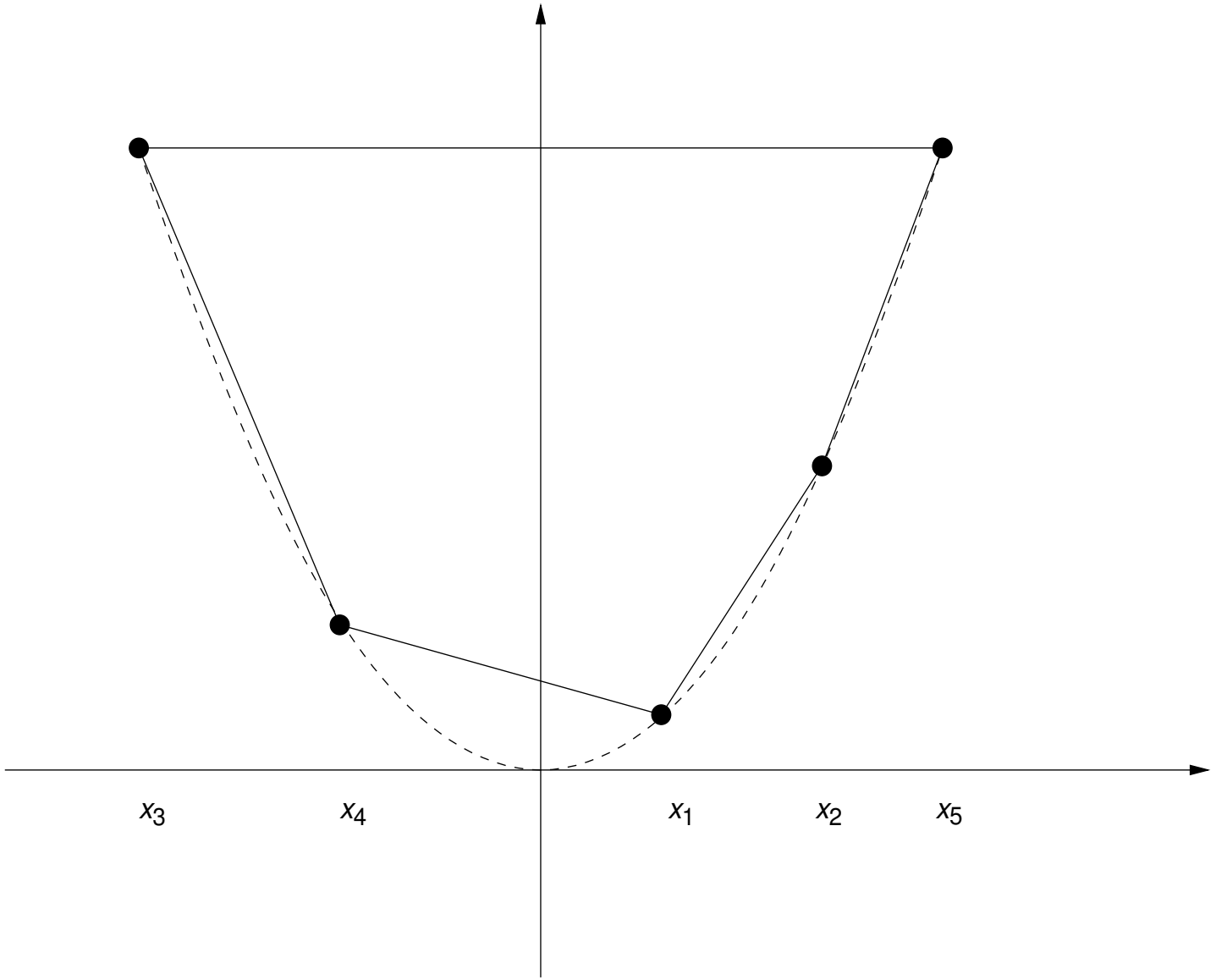
Exemplo: Polígono

- Considere o problema de se conectar pontos no plano por um polígono com semi-retas que não se cruzam (P_2).
- Será que existe uma redução do problema de ordenação P_1 para P_2.
- Lembre-se que P_1 gasta pelo menos $\Omega(n \log n)$ comparações.
- Será $\Omega(n \log n)$ uma cota inferior para P_2?

Exemplo: Polígono

- Sejam $x_1, x_2, \ldots, x_n$ números inteiros distintos que compõem uma entrada para o problema P_1.
- Podemos transformar estes números em pontos distintos no plano da forma (x_i, x_i^2).
- Notem que os pontos (x_i, x_i^2) preservam a ordem relativa dos x_i.

Exemplo: Polígono



Exemplo: Polígono

Teorema

Temos uma cota inferior $\Omega(n \log n)$ no número de comparações para a resolução do problema do polígono (P_2).

Prova.

- A transformação de uma instância de ordenação para uma instância de P_2 toma tempo $f_1(n) = O(n)$.
- Notem que uma solução de P_2 para esta instância necessariamente vai conectar os pontos vizinhos.
- Dado a descrição do polígono como solução, basta percorrermos os pontos para obtermos a ordenação dos x com custo $f_2(n) = O(n)$.
- Como $f1(n) + f2(n) = o(n \log n)$, temos uma cota inferior $\Omega(n \log n)$ para o custo de resolução de P_2.

□

Exemplo: Multiplicação de Matrizes Simétricas

- Considere o problema P_2 de multiplicação de matrizes quadradas simétricas(MMS).
- Considere o problema P_1 de multiplicação de matrizes quadradas (MMQ).
- Poderíamos pensar que P_2 é mais fácil de se resolver do que P_1.
- Para provar o contrário vamos mostrar que $P_1 \triangleright P_2$ com custos de transformações “baratos”.
- Seja n o número de linhas/colunas das matrizes. Qualquer algoritmo para multiplicação de matrizes (simétricas ou não) gasta $\Omega(n^2)$.

Exemplo: Multiplicação de Matrizes Simétricas

- Sejam A, B duas matrizes como entrada do problema MMQ.
- Denotamos por A^T a matriz transposta da matriz A.
- Não é difícil notar que as matrizes: $\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$ e $\begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}$ são simétricas.
- Além do mais multiplicando-as temos:
$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}$$
- Inspecionando a matriz resultante temos AB.

Exemplo: Multiplicação de Matrizes Simétricas

Teorema

Se existir um algoritmo para o problema MMS com tempo $O(g(n))$ então temos um algoritmo $O(g(n))$ para o problema MMQ. Ou seja MMS é pelo menos tão difícil quanto MMQ.

Prova.

- A redução proposta de uma instância do MMQ para o MMS tem custo $O(n^2)$ para montar as matrizes simétricas de dimensões $2n \times 2n$.

- Portanto o custo para gerar uma solução para o MMQ usando o resolvedor do MMS é $O(g(n) + n^2) = O(g(n))$ pois $g(n) = \Omega(n^2)$.

□

Exemplo: Matriz ao Quadrado

- Considere o problema P_2 de se elevar matrizes ao quadrado(QM).
- Considere o problema P_1 de multiplicação de matrizes quadradas (MMQ).
- Será que QM é mais fácil do que MMQ.
- Sejam A, B matrizes de uma instância para o problema MMQ.
- Não é difícil notar que: $\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}^2 = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$

Teorema

Se existir um algoritmo para o problema QM com tempo $O(g(n))$ então temos um algoritmo $O(g(n))$ para o problema MMQ. Ou seja QM é pelo menos tão difícil quanto MMQ.

Prova. Exercício □

Introdução

- Há milhares de problemas práticos para os quais não se conhece algoritmos de tempo polinomial
- Consideramos algoritmos eficientes aqueles com tempo de execução polinomial, ou seja, $O(n^k)$ para alguma constante k.
- Acredita-se que não haja tais algoritmos para muitos destes problemas
- Fundamentaremos esta dificuldade e a complexidade de tais problemas através de reduções
- Para este estudo, simplificaremos a "cara" destes problemas para problemas de decisão, mas que mantém sua complexidade/dificuldade computacional

Algumas Classes de Complexidade:

Informalmente, as classes P, NP, NP-Completo, NP-Difícil são conjuntos de problemas e se

$X \in$ P:

X pode ser decidido em tempo polinomial.

$X \in$ NP:

X é de decisão e tem certificado curto e verificável em tempo polinomial.

$X \in$ NP-Completo:

$X \in$ NP e $\forall Y \in NP$, Y é redutível polinomialmente a X.

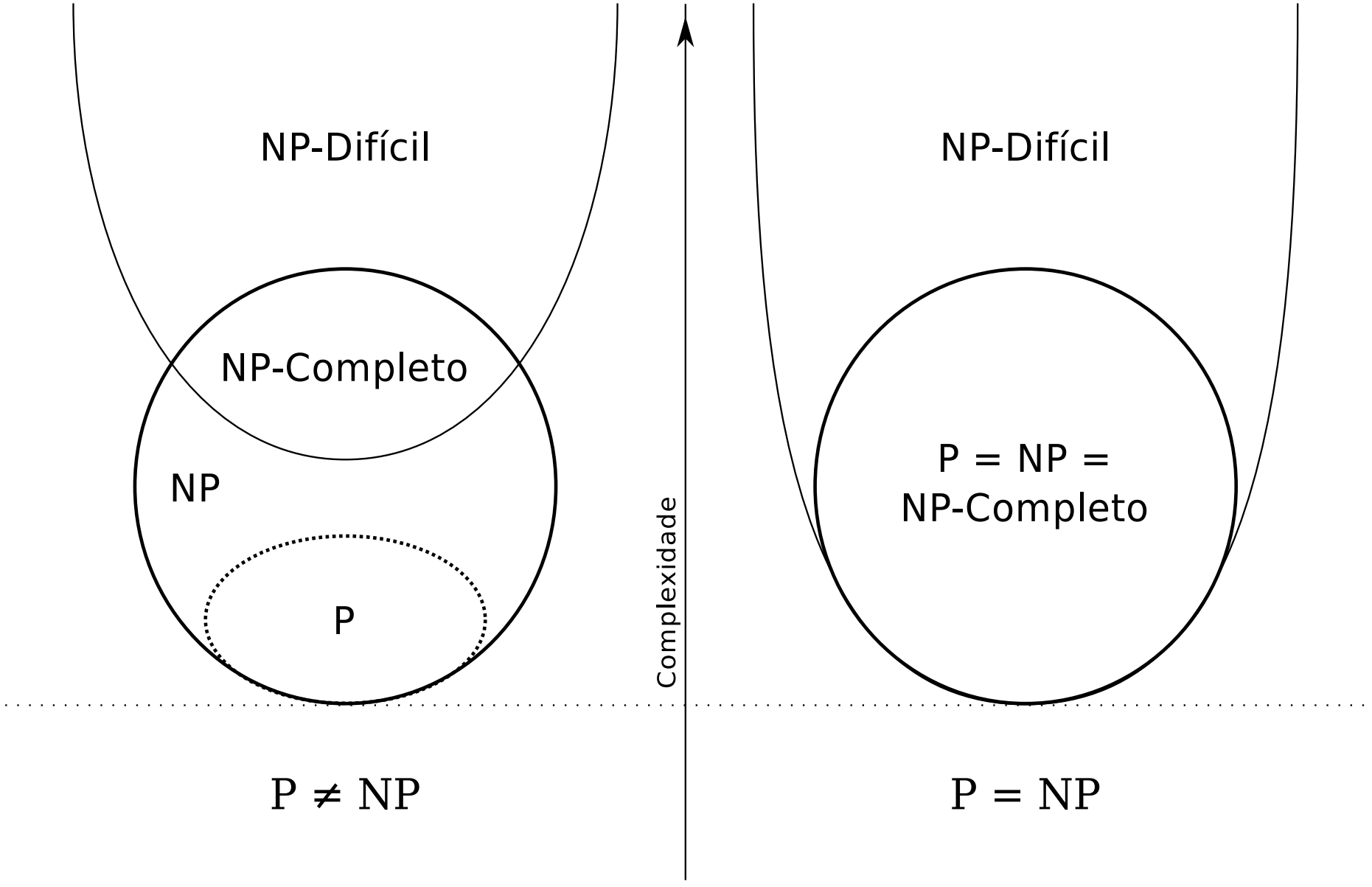
$X \in$ NP-Difícil:

$\forall Y \in$ NP, Y é redutível polinomialmente a X, onde X não necessariamente pertence a NP.

Observações:

- Veremos que a classe NP é gigantesca
- Todo problema de NP é redutível polinomialmente a um problema NP-completo
- Se existir algoritmo de tempo polinomial para um problema NP-completo então todos os problemas de NP se tornam de tempo polinomial

Possível configuração:



https://en.wikipedia.org/wiki/NP-completeness

Exemplos de problemas em **P**

- Dado um conjunto de números S, existe $n \in S$ tal que $n = \sum_{i \in S \setminus \{n\}} i$?
- Decida se um dado grafo G é conexo ?
- Dado grafo completo $G = (V, E)$, pesos nas arestas $c : E \to \mathbb{Z}^+$, vértices s e t e um inteiro positivo K, existe um caminho em G, de s para t, de peso no máximo K ?
- Posso colorir um mapa com 4 cores sem conflitos ?

Exemplos de problemas em NP-Completo

- Dado um conjunto de números S, existe $N \subseteq S$ tal que $\sum_{i \in N} i = \sum_{i \in S \setminus N} i$?
- Dado grafo G, existe um ciclo hamiltoniano em G ?
- Dado grafo completo $G = (V, E)$, pesos nas arestas $c : E \to \mathbb{Z}^*$ e um inteiro positivo K, existe um ciclo hamiltoniano em G, de peso no máximo K ?
- Dado grafo completo $G = (V, E)$, pesos nas arestas $c : E \to \mathbb{Z}$, vértices s e t e um inteiro positivo K, existe um caminho em G, de s para t, de peso no máximo K ?
- Posso colorir um mapa com 3 cores sem conflitos ?

Exemplos de problemas NP-difíceis

- Vários problemas práticos são NP-difíceis
  Exemplos:
  - Problema do Caixeiro Viajante
  - Atribuição de Freqüências em Telefonia Celular
  - Empacotamento de Objetos em Contêineres
  - Escalonamento de Funcionários em Turnos de Trabalho
  - Escalonamento de Tarefas em Computadores
  - Classificação de Objetos
  - Coloração de Mapas
  - Projetos de Redes de Computadores
  - Vários outros...
- P$\neq$NP $\Rightarrow$ não existem algoritmos eficientes para problemas NP difíceis

Problemas de Busca $\times$ Prolemas de Decisão

A teoria de NP-Completude é desenvolvida sobre problemas de decisão: resposta do tipo sim/não ou 1/0.

Problema (do Ciclo Hamiltoniano)

Dado grafo $G(V, E)$, encontrar ciclo em G que passa por cada vértice exatamente uma vez.

Problema (de Decisão do Ciclo Hamiltoniano)

Dado grafo $G(V, E)$, decidir se G possui ciclo que passa por cada vértice exatamente uma vez.

Exercício. Faça uma redução de tempo polinomial do Problema (de busca) do Circuito Hamiltoniano para o Problema de Decisão do Circuito Hamiltoniano.

Problemas de Otimização para Problemas de Decisão

Problema (do Caminho Mínimo)

Dado grafo $G(V, E)$ e dois vértices s e t, encontrar um caminho mais curto entre s e t.

Problema (de Decisão do Caminho Mínimo)

Dado grafo $G(V, E)$, dois vértices s e t e um inteiro K, decidir se há caminho de s a t de comprimento no máximo K.

Codificação de Problemas

- As instâncias de um problema são codificadas por strings de bits.
- Na prática não consideramos codificações “caras” como a unária.
- Utilizaremos codificações “baratas”/compactas, como a base 2 para representar inteiros
- Codificações compactas com bases diferentes não alteram a polinomialidade de um algoritmo

A classe P - Definição Informal

- Um problema Π pertence a classe P se existe algoritmo de tempo polinomial que resolve Π.
- Dizemos que os problemas pertencentes a P são **tratáveis** e admitem algoritmos **eficientes**.
- É claro que algoritmos com complexidade $O(n^{100})$ ou $O(10^{100}n)$ não são eficientes.
- Mas no estudo desta teoria consideramos tais algoritmos eficientes.
    - Geralmente quando um problema pertence a P, o tempo de execução do algoritmo é um polinômio de grau baixo.
    - Geralmente o algoritmo é eficiente na prática.

Linguagens Formais

- Um *alfabeto* Σ é um conjunto finito de símbolos.
- Uma *linguagem* sobre Σ é qualquer conjunto cujos elementos são strings formadas com símbolos de Σ.
- Uma string vazia é denotada por ϵ.
- Uma linguagem vazia é denotada por $\emptyset$.
- A linguagem contendo todas possíveis strings sobre Σ é denotada por Σ^*.
- $\langle I \rangle$ a codificação binária da estrutura I.
- $|I|$ é o tamanho da instância I, dado pelo número de bits desta string.

Exemplo

Se $\Sigma = \{0, 1\}$ *então* $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$

Linguagens Formais

- Admitimos as operações usuais $\in$, $\subset$, $\cup$ e $\cap$ sobre linguagens.
- Definimos o complemento de uma linguagem L como

$$\overline{L} = \Sigma^* - L$$

- A concatenação de linguagens L_1 e L_2 é uma nova linguagem L,

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ e } x_2 \in L2\}$$

- L^i denota a concatenação da linguagem L, i vezes.

Exemplo

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup \ldots$$

Linguagens Formais

- Usualmente assumiremos que $\Sigma = \{0, 1\}$.
- Σ^* corresponde a todas possíveis instâncias de um problema Π.
- Dado um problema de decisão Π, assumimos que este mapea instâncias de Σ^* para $\{0, 1\}$.
- Também assumimos que o problema Π é uma linguagem:

$$\Pi = \{x \in \Sigma^* : \Pi(x) = 1\}$$

- Ou seja, o problema é uma linguagem composta por todas as instâncias cuja solução é “sim”.

Problema (PATH)

$$PATH = \{\langle G, u, v, k\rangle : \quad G = (V, E) \text{ é um grafo}$$
$$u, v \in V,$$
$$k \geq 0 \text{ inteiro}$$
$$\text{existe caminho entre } u \text{ e } v \text{ em } G$$
$$\text{de tamanho no máximo } k\}$$

Usaremos PATH para representar tanto o problema de decisão quanto a linguagem.

- Dizemos que A decide uma linguagem L, se para todo $x \in \Sigma^*$ temos
  - $A(x) = 1$ se $x \in L$ e
  - $A(x) = 0$ se $x \in \overline{L}$.

- Se A tiver tempo de execução polinomial, dizemos que A decide a linguagem em tempo polinomial.

Definição (Classe P)

$$P = \{L \subseteq \Sigma^* : \quad \textit{existe um algoritmo } A \textit{ que decide } L \textit{ em tempo polinomial}\}$$

Certificado e Verificação

Certificados são estruturas que *certificam* que um problema tem resposta SIM

- Considere uma instância $I = \langle G, u, v, k \rangle$ para o problema PATH.
- Se I tem resposta SIM, deve existir caminho P de u até v de tamanho no máximo k.
- Neste caso, dizemos que P é um *certificado* para I.

Queremos certificados que

- sejam de tamanho curto (de tamanho polinomial)
- e que possam ser verificados eficientemente, em tempo polinomial.
  I.e., podemos *verificar* eficientemente se P é um caminho de u a v e se tem tamanho no máximo k.

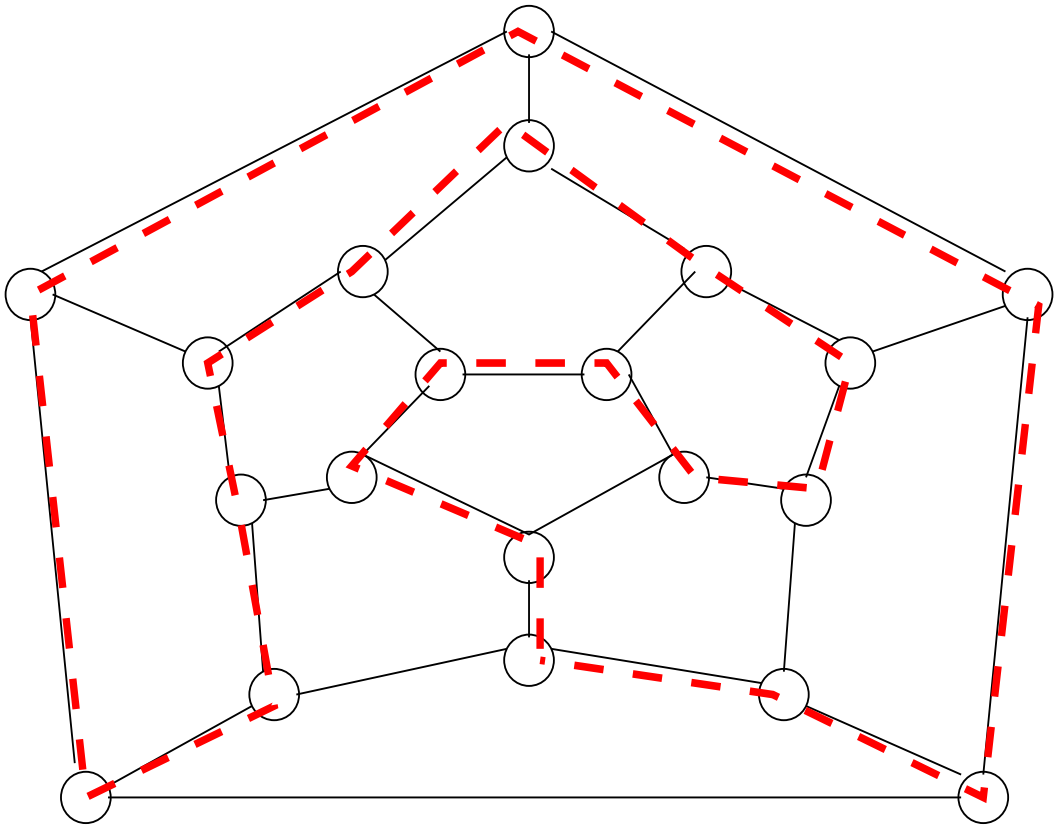
Certificado e Verificação

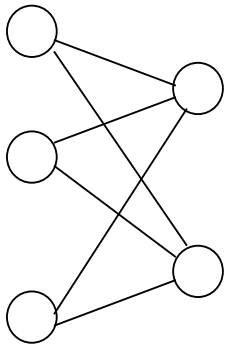
- Seja $G = (V, E)$ um grafo. O problema do ciclo hamiltoniano (HAM-CICLO) consiste em achar um ciclo que passa por todos os vértices de G exatamente uma vez.

- A versão de decisão consiste em dizer se um grafo possui ou não um ciclo hamiltoniano.

- HAM-CICLO$= \{\langle G \rangle : G$ é um grafo hamiltoniano $\}$.

Certificado e Verificação





Certificado e Verificação

- Um algoritmo simples para decidir se um grafo é hamiltoniano tem tempo $O(n!)$, onde n é o número de vértices do grafo.
- Mas dado um possível ciclo, é fácil verificar se este ciclo pertence ao grafo e é ou não hamiltoniano.
- Definimos então um **algoritmo verificador**, aquele que dado uma instância e um certificado, testa a validade do certificado.

Certificado e Verificação

- Se A é um algoritmo verificador de um problema Q, então:
  - Dado uma instância $x \in \{0,1\}^*$ existe uma prova/certificado y tal que $A(x,y) = 1$ se e somente se $x \in Q$.
- Note que dado uma instância $x \notin Q$, para todo $y \in \{0,1\}^*$ teremos $A(x,y) \neq 1$.
- Dado um algoritmo verificador A, a linguagem verificada por ele é:

$$L = \{x \in \{0,1\}^* : \text{ existe } y \in \{0,1\}^* \text{ tal que } A(x,y) = 1\}$$

Certificado e Verificação

- Uma linguagem Q é verificada por A
  - Se para todo $x \in L$ existir um $y \in \Sigma^*$ tal que $A(x, y) = 1$.
  - Para $x \notin L$ NÃO HÁ $y \in \Sigma^*$ tal que $A(x, y) = 1$.

Exemplo

- *Se um grafo é hamiltoniano então um ciclo-hamiltoniano é um certificado.*
- *Se um grafo não é hamiltoniano então qualquer sequência de vértices não corresponde a um ciclo hamiltoniano.*
- *Não há como enganar o algoritmo.*

Exemplo

Se Π *é um problema em P, qualquer* $y \in \Sigma^*$ *pode ser considerado certificado.*

A classe de problemas NP - Definição Informal

Definição

A classe NP é a classe de linguagens/problemas podem ser verificadas em tempo polinomial. Isto é, para instâncias SIM*, existe um certificado de tamanho polinomial e um algoritmo que testa a validade do certificado em tempo polinomial.*

- O termo NP refere-se a Não-determinístico Polinomial. Existe uma outra definição equivalente baseada em determinismo.

A classe de problemas NP - Definição Formal

Definição

Uma linguagem Q pertence a NP se existir um algoritmo verificador A com tempo de execução polinomial tal que

$$Q = \{x \in \Sigma^* : \quad \textit{existe um certificado } y \textit{ tal que } |y| = O(|x|^c) \textit{ onde } c \textit{ é uma constante e } A(x, y) = 1\}$$

- Dizemos que A verifica Q em tempo polinomial.

A classe NP é muito grande, pois basta que exista que o problema possa ser verificado eficientemente.

- Na grande maioria dos casos, o certificado é a própria solução do correspondente problema de busca

A classe de problemas NP

- Para mostrar que um problema de decisão **pertence a P**, devemos mostrar que **existe um algoritmo que decide/resolve** o problema em tempo polinomial.

- Para mostrar que um problema de decisão **pertence a NP**, devemos mostrar que existe um algoritmo de tempo polinomial que **verifica** o problema.

  Se a resposta é SIM,

  - existe um certificado que prova a resposta SIM do problema
  - tal certificado é de tamanho polinomial
  - existe um algoritmo de tempo polinomial que confere a resposta SIM pela instância e certificado

A classe de problemas NP

- Considere a linguagem HAM-CICLO. Seja um grafo $x \in$ HAM-CICLO, então existe uma sequência de vértices y que corresponde a um ciclo-hamiltoniano em x.
- É fácil implementar um algoritmo A' que dado um grafo x e uma sequência de vértices y verifica se y é um ciclo-hamiltoniano em x.
- Note que A' tem tempo polinomial e y tem tamanho polinomial em relação a x.
- Portanto mostramos que a linguagem HAM-CICLO $\in$ NP.

A classe de problemas NP

$\mathbf{P \subseteq NP}$

- Seja L uma linguagem pertencente a P.
- Existe um algoritmo A que decide L em tempo $p(n)$, para algum polinômio $p(n)$.
- Podemos escrever um algoritmo A' que recebe $x, y \in \Sigma^*$, e emula/executa A de tal forma que se $A(x) = 1$ então $A'(x, y) = 1$ independente do valor de y.
- Portanto $L \in$ NP.
- Com isso mostramos que P $\subseteq$ NP.

O problema em aberto de um milhão de dólares é P $=$ NP?

A classe de problemas co-NP

- Outro problema importante em aberto é: Dado $L \in$ NP, então $\overline{L} \in$ NP?

- Definimos a classe co-NP como o conjunto de linguagens L tal que $\overline{L} \in$ NP.

$$\text{co-NP} = \{L \subseteq \Sigma^* : \quad \overline{L} \in \text{NP}\}$$

- Ou seja para cada $x \notin L$ existe um algoritmo verificador de tempo polinomial e certificado y de tamanho polinomial tal que $A(x, y) = 1$.

- $\overline{\text{HAM-CICLO}}$ por exemplo pertence a co-NP.

- O problema em aberto é co-NP=NP?

Classes de Complexidade

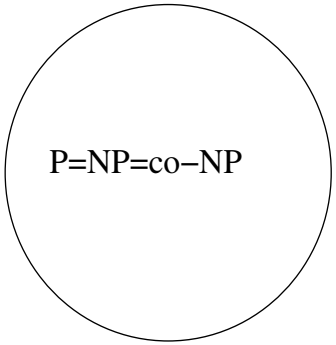
$$\text{P} = \{L \subseteq \Sigma^* : \quad \text{existe alg. pol. } A \text{ que decide L}\}$$

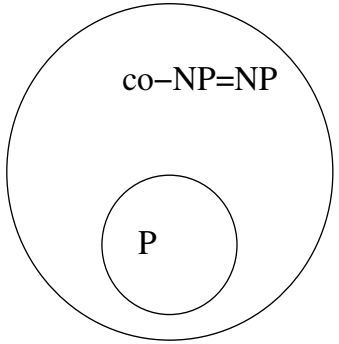
$$\text{NP} = \{L \subseteq \Sigma^* : \quad \text{existe alg. pol. } A \text{ que verifica } L\text{, ou seja,}$$
$$\text{para cada } x \in L \text{ existe } y \in \Sigma^* \text{ de tam. pol.}$$
$$\text{tal que A(x,y)=1 } \}$$

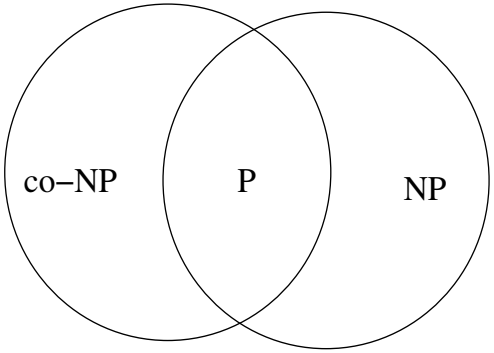
$$\text{co-NP} = \{L \subseteq \Sigma^* : \quad \text{alg. pol. } A \text{ que verifica } \overline{L}\text{, ou seja,}$$
$$\text{para cada } x \notin L \text{ existe } y \in \Sigma^* \text{ de tam. pol.}$$
$$\text{tal que A(x,y)=1}\}$$

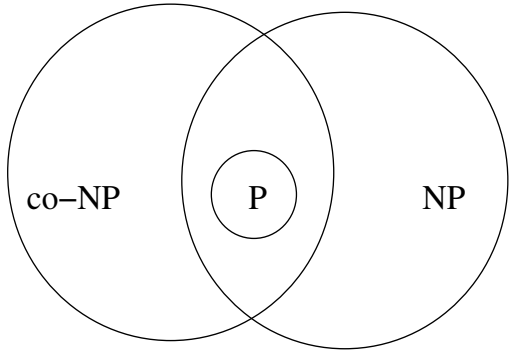
Classes de Complexidade

Possíveis configurações destas classes:









Classe NP-Completo (NPC)

- A maioria dos teóricos da computação acredita que P $\neq$ NP.
- Isto se deve em grande parte a existência da classe NP-Completo.
- Os problemas pertencentes a esta classe tem uma propriedade importante:
  - Se existir um algoritmo polinomial para algum problema em NP-Completo então todos problemas em NP possuem um algoritmo polinomial.

Reduções de Tempo Polinomial

- Nos concentraremos agora em reduções que gastam tempo polinomial.
- A função que transforma a instância deve ser de tempo polinomial.
- Uma linguagem L_1 é *redutível em tempo polinomial* para outra linguagem L_2 ($L_1 \triangleright_p L_2$) se
  - Existir uma função (algoritmo) $F : \{0,1\}^* \rightarrow \{0,1\}^*$.
  - F transforma instâncias $x_1 \in \{0,1\}^*$ de L_1 em instâncias $x_2 \in \{0,1\}^*$ de L_2 com custo polinomial $f_1(|x_1|)$.
  - $x_1 \in L_1$ se e somente se $x_2 \in L_2$.
- Neste tipo de redução, não precisamos transformar a solução, uma vez que a resposta é 0/1.

Reduções de Tempo Polinomial

- Desta forma,
  - se existir um algoritmo de tempo polinomial que decide L_2 e
  - se existir um algoritmo de tempo polinomial que reduz $L_1 \triangleright_p L_2$
- Então teremos um algoritmo polinomial que decide L_1.
- Note a importância de $x_1 \in L_1$ *SE E SOMENTE SE* $F(x_1) \in L_2$.

Reduções de Tempo Polinomial

Teorema (1.1)

Sejam $L_1, L_2 \subseteq \{0,1\}^*$ linguagens tais que $L_1 \triangleright_p L_2$, então se $L_2 \in P$ implica que $L_1 \in P$.

Prova.

- Se $L_2 \in P$ então, mostraremos que existe algoritmo de tempo polinomial A_1 que decide L_1.
- Seja A_2 algoritmo que decide L_2 em tempo polinomial e F algoritmo de tempo polinomial que reduz $L_1 \triangleright_p L_2$.
- Dado $x_1 \in \{0,1\}^*$ o algoritmo A_1 computa $F(x_1) = x_2$.
- Depois A_1 executa $A_2(x_2)$ e retorna 1 caso $A_2(x_2) = 1$ e 0 caso $A_2(x_2) = 0$.
- A_1 tem tempo polinomial e se $x_1 \in L_1$ então $A_1(x_1) = 1$; se $x_1 \notin L_1$ então $A_1(x_1) = 0$.

□

Classe NP-Completo (NPC)

Definição

Uma linguagem $L \subseteq \{0,1\}^*$ pertence a NP-completo se:

1. $L \in NP$
2. $L' \triangleright_p L$ *para todo* $L' \in NP$

- Se uma linguagem satisfaz apenas a propriedade 2 então esta linguagem pertence a classe **NP-Difícil**.

Classe NP-Completo (NPC)

Teorema

Se alguma linguagem NPC for decidida em tempo polinomial, então P=NP.

Se algum problema em NP não for decidido em tempo polinomial então nenhum problema NP-completo pode ser decidido em tempo polinomial.

Prova. Seja $L \in P$ e $L \in$ NPC. Para qualquer linguagem $L' \in$ NP sabemos que $L' \triangleright L$ pois $L \in$ NPC e como $L \in P$ então pelo teorema 1, L' é decidida em tempo polinomial.

Se existir $L \in NP$ que não seja decidido em tempo polinomial então nenhum problema NPC pode ser decidido em tempo polinomial, pois caso contrário L também seria. □

NP-Completude

Teorema

Seja $L' \in$ NP-Completo. Se L é tal que $L' \triangleright_p L$ então L é NP-Difícil. Se além disso $L \in$ NP então L é NP-Completo.

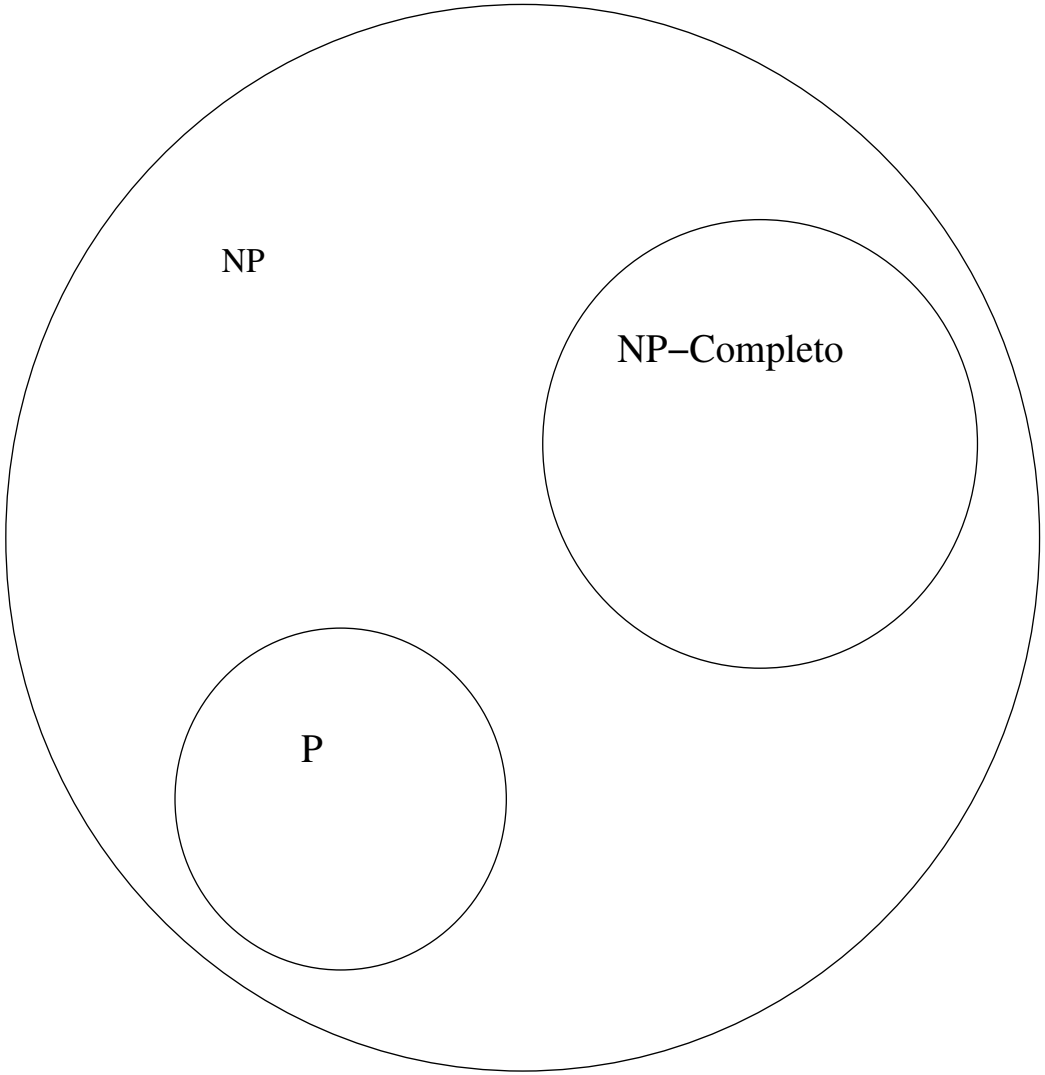
Prova. Como L' é NP-Completo, então

- para todo $L'' \in$ NP temos $L'' \triangleright_p L'$.
- Como $L' \triangleright_p L$ então (exercício) $L'' \triangleright_p L$ para todo $L'' \in$ NP.
- Portanto L é NP-Difícil.

Se além disso $L \in$ NP, então por definição L é NP-Completo. □

Classe NP-Completo (NPC)

Como acredita-se que seja a relação das classes:



Satisfatibilidade (SAT)

- O problema de Satisfatibilidade (SAT) consiste em determinar se há uma atribuição para um conjunto de variáveis de uma fórmula booleana, tal que o resultado desta seja verdadeiro.

- A fórmula pode usar parênteses e ser escrita com os seguintes operadores:

  1. AND ($\wedge$).
  2. OR ($\vee$).
  3. NOT ($\neg$).
  4. Implicação ($\rightarrow$).
  5. Se e Somente Se ($\leftrightarrow$).

Satisfatibilidade

Tabela verdade dos operadores

OP1	OP2	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$
F	F	F	F	V	V
F	V	F	V	V	F
V	F	F	V	F	F
V	V	V	V	V	V

Satisfatibilidade

Exemplo

$f = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

Atribuição: $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$

$$\begin{aligned} f &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg((1 \leftrightarrow 1) \vee 1)) \wedge 1 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1. \end{aligned}$$

Satisfatibilidade

- A linguagem SAT é aquela que contém fórmulas booleanos com uma atribuição verdadeira.

$$\text{SAT} = \{\langle f \rangle : \quad f \text{ é uma fórmula booleana que admite uma atribuição verdadeira } \}$$

Teorema

SAT é NP-Completo.

Prova. Veremos a prova posteriormente. □

NP-Completude (NCP)

- Sabemos que SAT $\in$ NPC
- Agora temos duas possibilidades para mostrar que um problema Q é NP-Difícil:
  1. Mostrar que existe uma redução polinomial de todo $L \in$ NP para Q.
  2. Mostrar que existe uma redução polinomial de SAT para Q.
- De uma maneira geral, para mostrarmos que Q é NP-Difícil, basta mostrarmos que há uma redução polinomial de algum $L \in$ NP-Difícil para Q.

3CNF-SAT

- Uma fórmula está na CNF (**forma normal conjuntiva**) se ela pode ser expressa como uma conjunção (AND) de cláusulas que são a disjunção (OR) de uma ou mais variáveis (as variáveis podem estar negadas).

  Exemplo: $(x_1) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$.

- Uma fórmula está na 3CFN se ela está na CFN e cada cláusula possui exatamente 3 **literais**.

- O problema 3CNF-SAT (Satisfibilidade de fórmulas na 3CNF) consiste em determinar se uma fórmula na 3CNF possui ou não uma atribuição verdadeira.

3CNF-SAT

Lema

3CNF-SAT pertence a NP.

Prova. Exercício.

□

3CNF-SAT

Lema

3CNF-SAT pertence a NP-Difícil.

Prova. Vamos fazer uma redução do SAT para o 3CNF-SAT.

Dado instancia f_1 do SAT vamos construir em tempo polinomial uma fórmula f_2 do 3CNF-SAT tal que $f_1 \in$ SAT se e somente se $f_2 \in$ 3CNF-SAT.

Construiremos a fórmula f_2 em três etapas.

1. Primeiramente transformaremos f_1 em uma fórmula que é uma conjunção de cláusulas contendo até 3 variáveis.
2. A fórmula resultante então será transformada em outra que estará na CNF. Deveremos tirar os operadores $\leftrightarrow$ e $\rightarrow$.
3. Por último deixaremos cada cláusula com exatamente 3 variáveis.

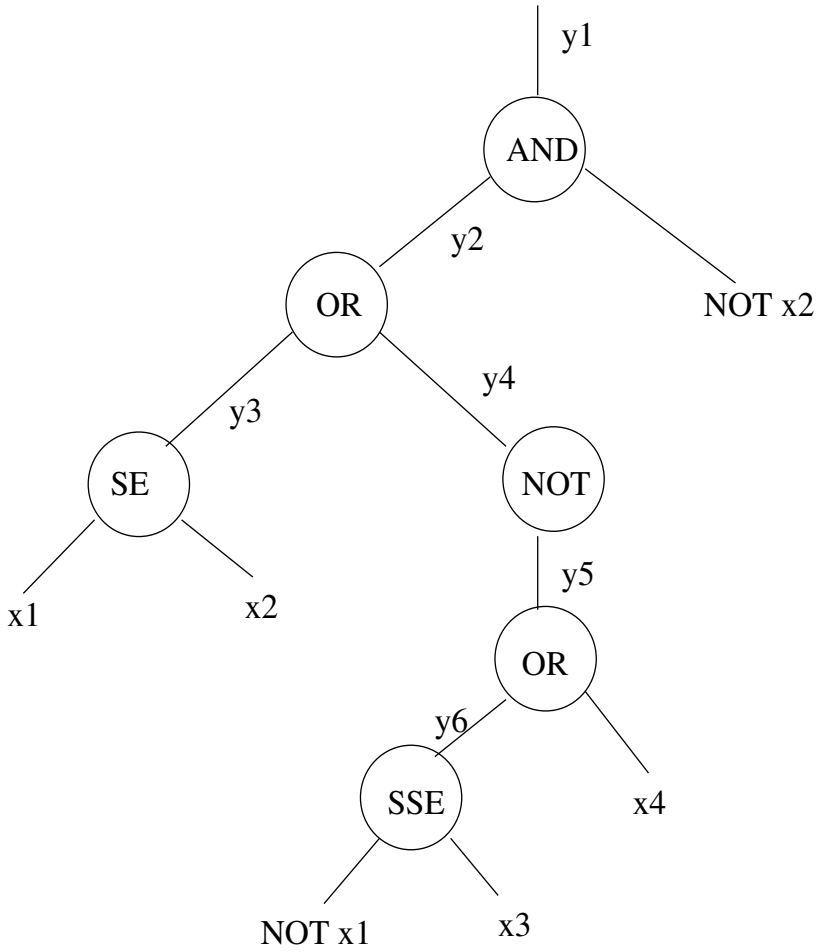
$\square$

Continuação da prova

- Primeiramente devemos separar a fórmula f_1 em cláusulas. Para fazer isto, construímos uma árvore de avaliação da fórmula.
- Cada aresta da árvore corresponde a uma variável e cada nó corresponde a um operador da fórmula.

Continuação da prova

Exemplo: $f_1 = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$



Continuação da prova

- Podemos “ver” esta árvore como um circuito que queremos transformar numa fórmula.
- Cada fio tem uma variável associada. Queremos que a saída (y_1 no exemplo) seja 1 mas desde que o resultado de cada fio/aresta corresponda a exatamente a operação no nó. Considerando a árvore como exemplo teremos a fórmula:

$$
\begin{aligned}
f' = y_1 \wedge \quad & (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
\wedge \quad & (y_2 \leftrightarrow (y_3 \vee y_4)) \\
\wedge \quad & (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
\wedge \quad & (y_4 \leftrightarrow (\neg y_5)) \\
\wedge \quad & (y_5 \leftrightarrow (y_6 \vee x_4)) \\
\wedge \quad & (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$

Continuação da prova

- Note que cada cláusula terá até 3 variáveis, no máximo duas como entrada do operador, mais uma nova (correspondente aos *y*s).
- A fórmula resultante f' tem portanto tamanho polinomial em relação a f_1 e pode ser gerada em tempo polinomial. Além disso f' tem atribuição verdadeira se e somente se f_1 tiver atribuição verdadeira.
- Para deixarmos cada cláusula f'_i de f' na CNF construímos a tabela verdade de cada cláusula. A partir da tabela verdade da cláusula f'_i é fácil construir uma fórmula na DNF (formal normal disjuntiva) que é equivalente a $\neg f'_i$.

Continuação da prova

Como exemplo, seja a cláusula $(y_1 \leftrightarrow (y_2 \wedge \neg x_2)$.

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Continuação da prova

A partir desta tabela podemos escrever a fórmula $\neg f_1'$ (terá valor 1 quando f_1' tem valor 0) como:

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

Portanto f_1' é equivalente a:

$$\overline{(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)}$$

Aplicando DeMorgan temos a fórmula equivalente:

$$(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

Continuação da prova

- Isto deve ser feito para cada cláusula de f'. Note que a tabela verdade tem no máximo 8 entradas pois há no máximo 3 variáveis em cada cláusula. Portanto construímos uma nova fórmula f'' que é equivalente a f' e tem tamanho polinomial em relação a f'.
- Finalmente vamos transformar f'' em uma fórmula equivalente f_2 em que cada cláusula tem exatamente 3 variáveis

Continuação da prova

Para cada cláusula f_i'' de f'' faremos o seguinte:

1. Se f_i'' tiver duas variáveis $(l_1 \vee l_2)$, trocamos esta pela equivalente $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$.
2. Se f_i'' tiver apenas uma variável então trocamos (l_1) por

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

Notem que as cláusulas novas serão verdadeiras somente se as originais forem, independente dos valores de p ou q.

Continuação da prova

- Nesta última transformação acrescentamos no máximo mais duas variáveis por cláusula e criamos no máximo mais três cláusulas extras.
- Portanto, a cláusula f_2 pode ser computada em tempo polinomial e terá tamanho polinomial em relação a f_1. Além disso f_2 é equivalente a f_1, pois terá atribuição verdadeira se e somente se f_1 tiver atribuição verdadeira.

Clique

- Uma clique em um grafo não-direcionado $G = (V, E)$ é um subconjunto de vértices $V' \subseteq V$ tal que qualquer par de vértices está conectado.
- O problema Clique consiste em achar a clique de tamanho máximo em um grafo.
- A versão de decisão é decidir se um grafo possui uma clique de tamanho k.

$$\text{Clique} = \{\langle G, k\rangle : \quad G \text{ é um grafo que possui clique de tamanho } k\}$$

Clique

Lema

Clique pertence a NP.

Prova. Exercício. □

Clique

Lema

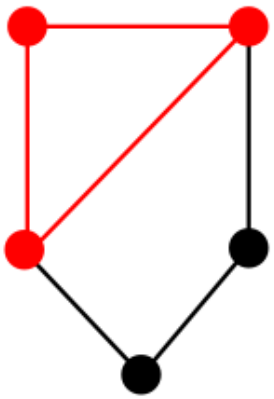
Clique pertence a NP-Difícil.

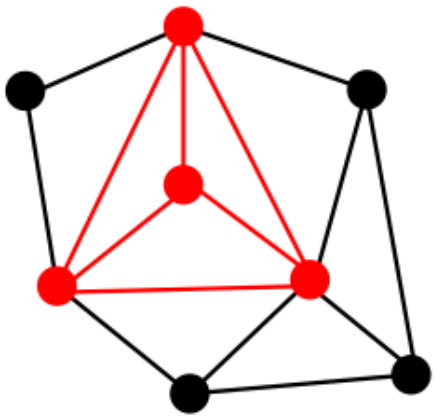
- Faremos uma redução do problema 3CNF-SAT para Clique. Dado uma fórmula f com k cláusulas, transformaremos esta em um grafo G tal que G possui clique de tamanho k se e somente se f puder ser satisfeita.
- Seja $f = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ uma fórmula na 3CNF. Cada cláusula C_i de f possui exatamente três literais l_1^i, l_2^i e l_3^i.
- Para cada cláusula C_i construimos 3 vértices v_1^i, v_2^i e v_3^i que correspondem aos 3 literais da cláusula. Seja C_j uma outra cláusula da fórmula com os respectivos vértices v_1^j, v_2^j e v_3^j.
- Hávera uma aresta entre v_x^i e v_y^j se seus correspondentes literais nas cláusulas são consistentes, ou seja, l_x^i não é negação de l_x^j.

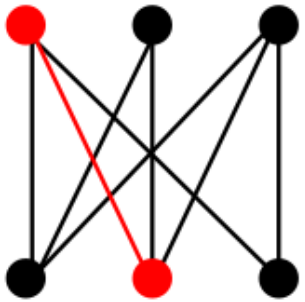
Clique

Problema: Dado um grafo G e um inteiro k, G possui um clique com $\geq k$ vértices?

Exemplos:



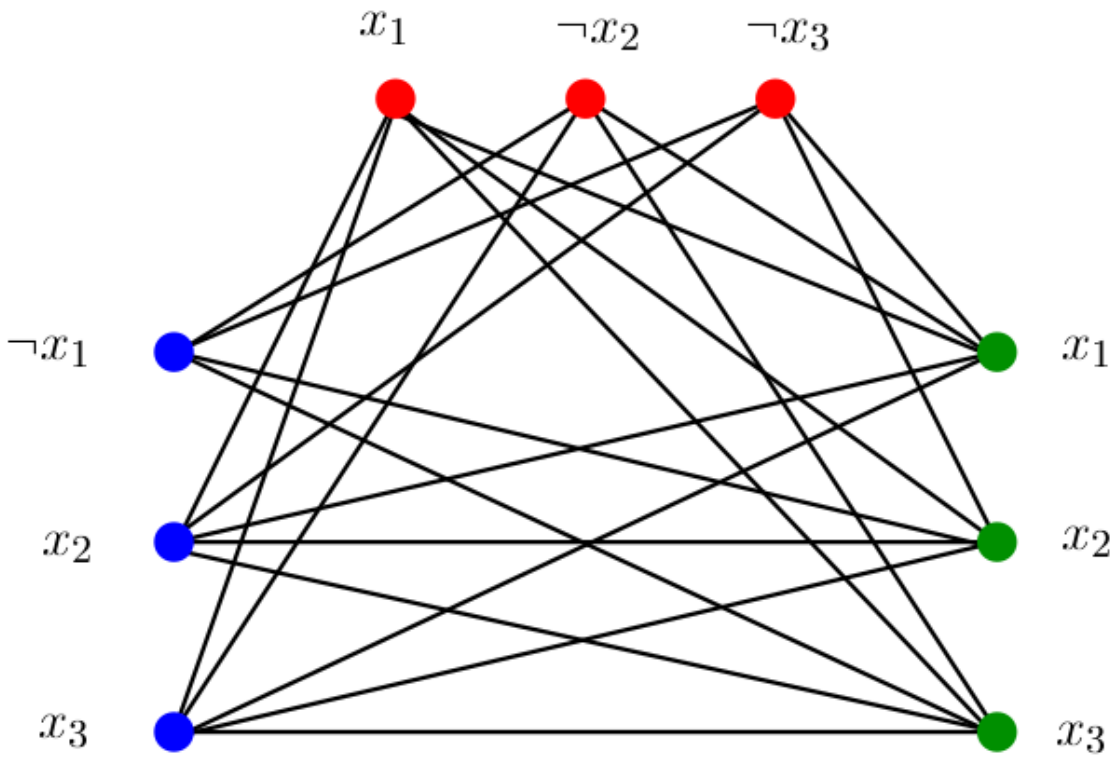




clique com k vértices $=$ subgrafo completo com k vértices

Clique é NP-Completo

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



Continuação da prova

- A redução tem tempo polinomial.
- $\Rightarrow$) Se f pode ser satisfeita, então pelo menos um literal de cada uma de suas cláusulas tem valor 1. Para cada cláusula escolhemos um literal que tem valor 1, e a clique será formada pelos vértices correspondentes. Note que para quaisquer dois vértices x e y desta clique há uma aresta entre eles pois só não haveria se x fosse negação de y.
- $\Leftarrow$) Se houver uma clique de tamanho k, estes vértices correspondem a uma atribuição verdadeira para f. Primeiro note que pode haver no máximo um vértice de cada cláusula, pois não há arestas entre vértices de uma mesma cláusula. Segundo, como não há arestas entre vértices que correspondem a literais que se negam, então temos uma atribuição verdadeira para f.

Vertex Cover (VC)

- Um VC em um grafo não-direcionado $G = (V, E)$ é um subconjunto de vértices $V' \subseteq V$ tal que qualquer aresta do grafo é incidente a pelo menos um vértice em V'.
- O problema é achar um VC de tamanho mínimo.
- A versão de decisão é saber se há um VC de tamanho k.

$$\text{VC} = \{\langle G, k\rangle : \quad G \text{ é um grafo que possui uma cobertura por vértices de tamanho } k\}$$

Vertex Cover (VC)

Lema

VC pertence a NP.

Prova. Exercício. □

Vertex Cover (VC)

Lema

VC é NP-Difícil.

- Faremos uma redução do problema Clique para o VC.
- Dado um grafo $G = (V, E)$ calcule o seu complemento $\overline{G} = (V, \overline{E})$. $\overline{G}$ é tal que $\overline{E} = \{(u, v) : (u, v) \notin E\}$.
- Vamos mostrar que G possui um Clique de tamanho k se e somente se $\overline{G}$ possui um VC de tamanho $|V| - k$.

Continuação da Prova

($\Rightarrow$)

- Seja $C \subseteq V$ uma clique em G de tamanho k. Seja $V' = V \setminus C$. Vamos mostrar que V' é um VC em $\overline{G}$.
- Suponha por absurdo que não seja. Então existe aresta (u, v) em $\overline{G}$ tal que $u \notin V'$ e $v \notin V'$. Então ambos u e v pertencem a C mas isto é um absurdo pois C é uma clique em G e $(u, v) \notin G$, já que $(u, v) \in \overline{G}$.

Continuação da Prova

($\Leftarrow$)

- Seja V' um VC de tamanho $|V| - k$ em $\overline{G}$. Seja $C = V \setminus V'$ (note que $|C| = k$). Vamos mostrar que C é uma clique em G.
- Como V' é um VC em $\overline{G}$ então todas as arestas em $\overline{G}$ incidem em pelo menos um vértice de V'. Portanto não pode haver arestas entre quaisquer dois vértices de C em $\overline{G}$ e portanto C é um clique em G.

Subset-Sum

- O problema Subset-Sum (S-Sum) tem como entrada um conjunto de números naturais S e um valor natural t.
- Deve-se achar um subconjunto $S' \subseteq S$ tal que $\sum_{s \in S'} s = t$.
- O problema de decisão consiste em determinar se existe tal subconjunto S'.

$$\text{S-Sum} = \{\langle S, t\rangle : \text{ existe } S' \subseteq S \text{ tal que } t = \sum_{s \in S'} s\}$$

Subset-Sum

- Suponha $S = \{1, 4, 10, 14, 54, 100, 1004, 1003\}$ e $t = 1027$.
- Neste caso há $S' = \{10, 14, 1003\}$ que é solução para esta instância.
- Vamos mostrar que S-Sum é NP-Completo.

Subset-Sum

Lema

S-Sum pertence a NP.

Prova. Exercício. □

Subset-Sum

Teorema

S-Sum é NP-completo.

- Falta mostrar que S-Sum é NP-Difícil. Para tanto vamos reduzir em tempo polinomial o problema 3CNF-SAT para S-Sum.

- Temos uma fórmula ϕ com variáveis $x_1, \ldots, x_n$, e cláusulas $C_1, \ldots, C_k$ cada uma com exatamente 3 literais.

- Sem perda de generalidade desconsideramos variáveis que não aparecem em nenhuma cláusula e também desconsideramos cláusulas que tenham um literal e sua negação.

Subset-Sum

- Vamos criar dois números para cada variável e para cada cláusula, e um número especial t.
- Cada número é composto por $n + k$ dígitos, e cada dígito corresponde a uma variável ou uma cláusula.
- Dígitos estão em ordem $x_1, x_2, \ldots, x_n, C_1, \ldots, C_k$.

Continuação da prova

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

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	1	0	0	1	0	0	1
v_1'	1	0	0	0	1	1	0
v_2	0	1	0	0	0	0	1
v_2'	0	1	0	1	1	1	0
v_3	0	0	1	0	0	1	1
v_3'	0	0	1	1	1	0	0
s_1	0	0	0	1	0	0	0
s_1'	0	0	0	2	0	0	0
s_2	0	0	0	0	1	0	0
s_2'	0	0	0	0	2	0	0
s_3	0	0	0	0	0	1	0
s_3'	0	0	0	0	0	2	0
s_4	0	0	0	0	0	0	1
s_4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

Subset-Sum

- Para uma variável x_i criamos um número v_i com um 1 no dígito x_i e um 1 em cada dígito C_j tal que x_i aparece na cláusula C_j.
- Também criamos para x_i um número v_i' com 1 no dígito x_i e um 1 em cada dígito C_i tal que $\neg x_i$ aparece em C_i.
- Para cada C_i criamos um número s_i com um 1 no dígito C_i.
- Também criamos para C_i um número s_i' com um 2 no dígito C_i.
- O número t tem um 1 em cada dígito x_i e um 4 em cada dígito C_i.

Continuação da prova

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

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	1	0	0	1	0	0	1
v_1'	1	0	0	0	1	1	0
v_2	0	1	0	0	0	0	1
v_2'	0	1	0	1	1	1	0
v_3	0	0	1	0	0	1	1
v_3'	0	0	1	1	1	0	0
s_1	0	0	0	1	0	0	0
s_1'	0	0	0	2	0	0	0
s_2	0	0	0	0	1	0	0
s_2'	0	0	0	0	2	0	0
s_3	0	0	0	0	0	1	0
s_3'	0	0	0	0	0	2	0
s_4	0	0	0	0	0	0	1
s_4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

Continuação da prova

- Note que cada um dos números tem valor diferente. Uma cláusula não tem um literal e sua negação!
- Os dígitos $x_1, \ldots, x_n$ são todos diferentes para números que correspondem a variáveis e cláusulas diferentes.
- Note ainda que ao somarmos todos os números, o maior valor que um dígito pode atingir é 6 (para um dígito de cláusula pois cada cláusula tem no máximo 3 literais).
- Portanto não há como valores de dígitos mais significativos serem afetados pela soma dos valores nos dígitos menos significativos.

Continuação da prova

- A redução é feita em tempo polinomial. O conjunto S tem $2(n+k)$ números cada um com $n+k$ dígitos e temos mais o número t.
- Temos que mostrar agora que uma fórmula ϕ para 3CNF-SAT pode ser satisfeita sse para (S,t) construído como especificado possui $S' \subseteq S$ tal que $\sum_{s \in S'} s = t$.

Continuação da prova

Ida:

- Suponha ϕ possua atribuição para literais $l_1 = 1, \ldots, l_n = 1$ que faça ϕ verdadeiro.
- Para cada literal l_i correspondendo a x_i atribuímos v_i para S' e caso l_i corresponda a $\neg x_i$ atribuímos v_i' para S'.
- Para cada cláusula C_j atribuímos para S' o número s_i se 3 literais forem verdadeiros na cláusula, o número s_i' se 2 literais forem verdadeiros, e ambos s_i e s_i' se apenas um literal for verdadeiro na cláusula.
- Seja $t' = \sum_{s \in S'} s$.
- Para cada dígito x_i temos um 1 em t' pois a variável ou sua negação tem valor 1.
- Pelo modo como escolhemos os números s_i e s_i' de cada cláusula sabemos que o digito de t' que correspondente a cada cláusula é 4.
- Portanto $t' = t$.

Continuação da prova

Volta:

- Seja $S' \subseteq S$ tal que $\sum_{s \in S'} s = t$.
- Primeiro note que nunca ambos v_i e v'_i podem pertencer a S' pois senão teríamos o valor 2 no dígito x_i correspondente.
- Para cada cláusula C_j, o valor correspondente da soma é 4 e portanto pelo menos um número correspondendo a um literal da cláusula pertence a S'.
- Logo atribuindo 1 para os literais correspondendo aos números v_i ou v'_i pertencentes a S', temos uma atribuição válida e verdadeira para ϕ.

Ciclo Hamiltoniano

- Vamos mostrar que o problema do ciclo hamiltoniano (HAM-CYCLE) é NPC.
- Para tanto vamos fazer uma redução do problema VERTEX-COVER para HAM-CYCLE.

Ciclo Hamiltoniano

Teorema

O problema HAM-CYCLE pertence à NP.

Exercício.

Ciclo Hamiltoniano

Teorema

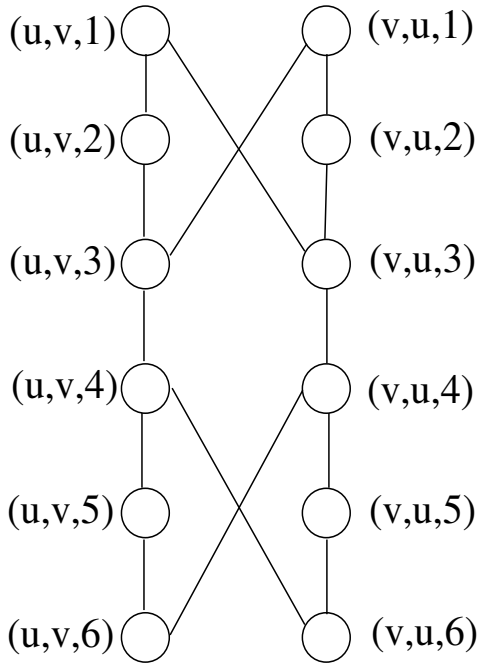
O problema HAM-CYCLE é NP-Difícil.

- Iremos fazer a redução VERTEX-COVER $\triangleright_p$ HAM-CYCLE.
- Dado instância (G, k) para VERTEX-COVER iremos construir um grafo G' instância para HAM-CYCLE tal que

$$(G, k) \in \text{VERTEX-COVER sse } G' \in \text{ HAM-CYCLE}$$

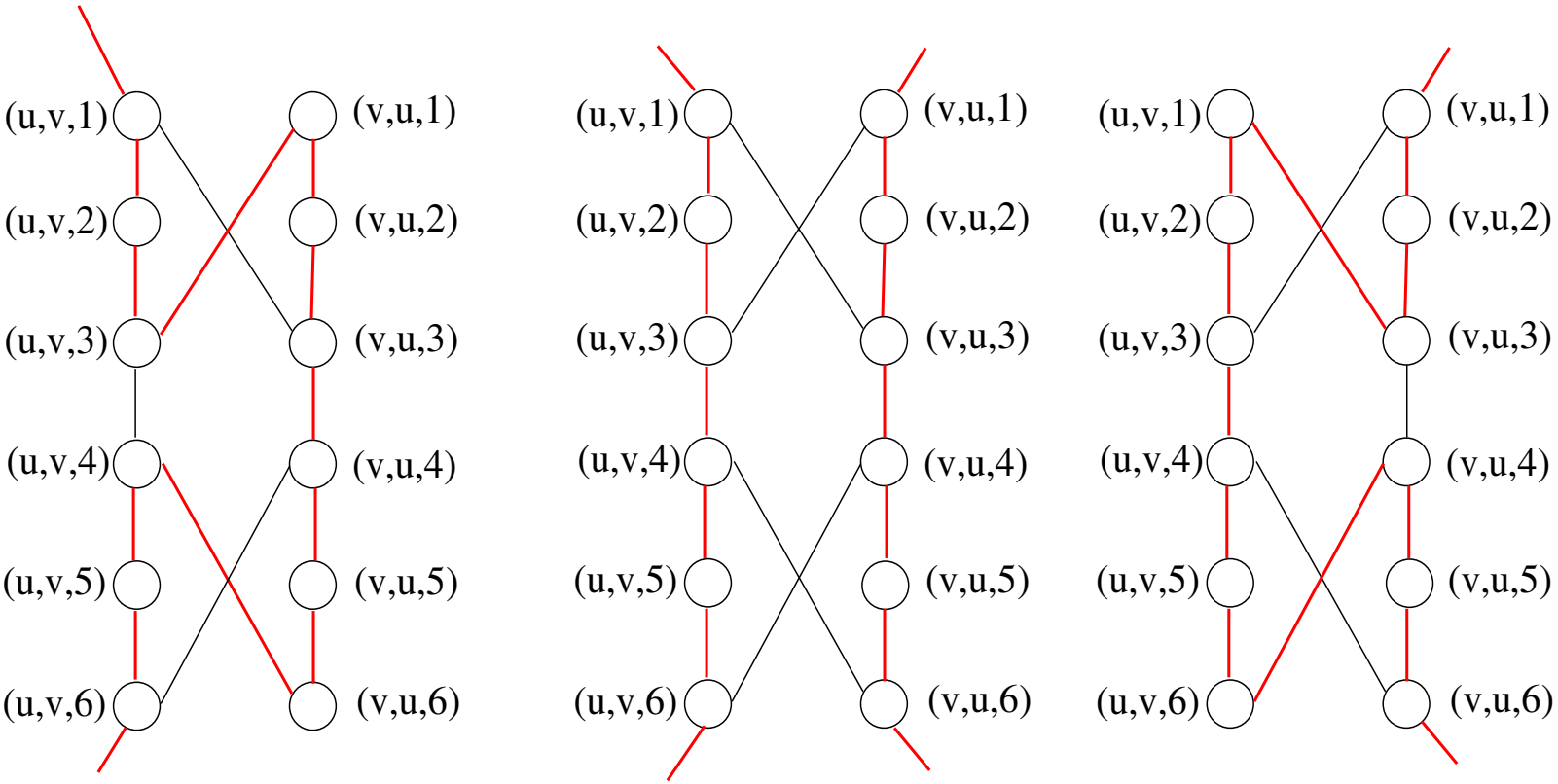
Ciclo Hamiltoniano

- Na redução usamos uma estrutura especial (denotada W_{uv}) para cada aresta (u, v) de G:



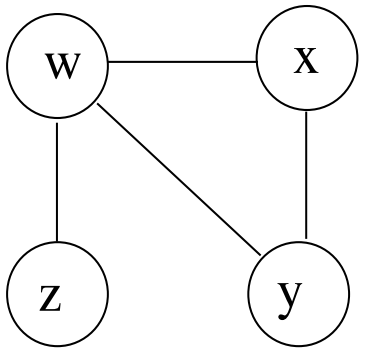
Ciclo Hamiltoniano

- Somente os vértices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$ e $[v, u, 6]$ tem ligação para fora da estrutura.
- O importante é que para passar por todos os vértices da estrutura temos apenas 3 opções:



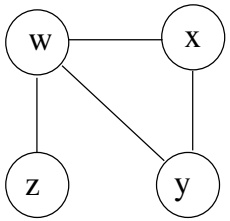
Ciclo Hamiltoniano

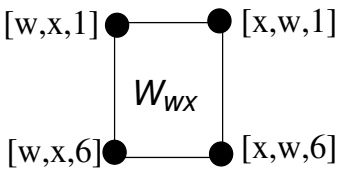
- Para cada aresta (u, v) de G teremos uma destas estruturas.
- Também teremos mais k vértices $s_1, \ldots, s_k$ chamados seletores.
- Para ligar as estruturas W_{uv} entre si e os vértices seletores iremos criar algumas arestas.
- Para deixar mais claro, vamos usar como exemplo o seguinte grafo G instância do VERTEX-COVER ($k = 2$):

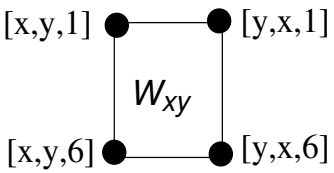


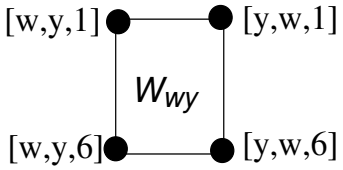
Ciclo Hamiltoniano

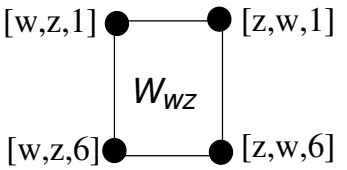
Com uma estrutura para cada aresta e seletores s_1 e s_1:









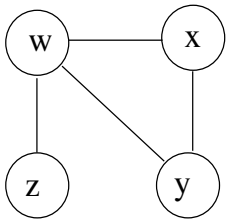


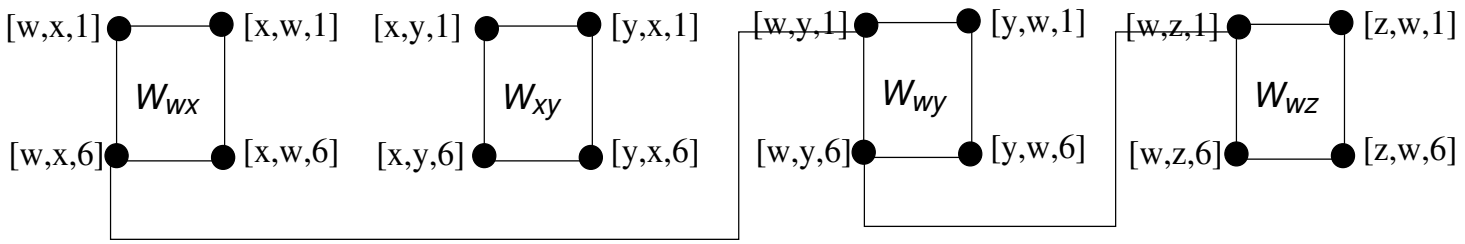
Ciclo Hamiltoniano

- Para cada vértice u de G sejam $u^1, \ldots, u^{\delta(u)}$ os seus vizinhos em uma ordem qualquer ($\delta(u)$ é o grau de u).
- Adicionamos as arestas em G': $\{([u, u^i, 6], [u, u^{(i+1)}, 1) : 1 \leq i \leq \delta(u) - 1\}$
- Estas arestas formam um “caminho”pelas arestas incidentes a u.

Ciclo Hamiltoniano

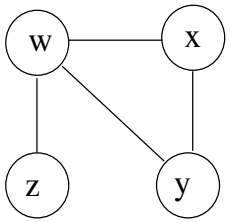
Para o vértice *w* acrescentamos arestas:





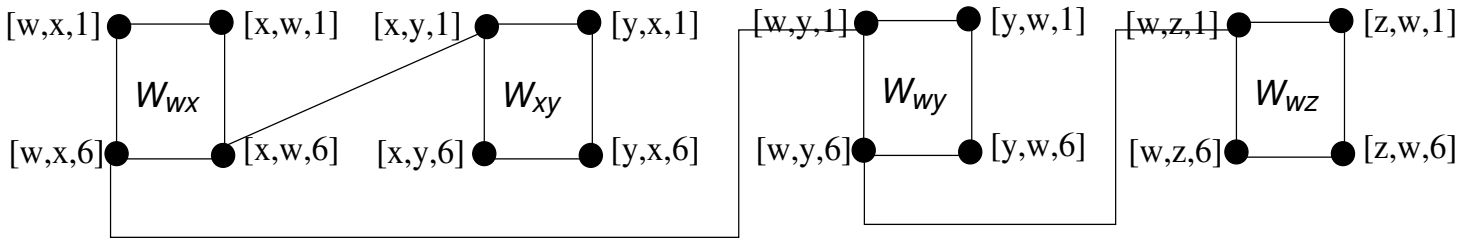
Ciclo Hamiltoniano

Para o vértice *x* acrescentamos arestas:



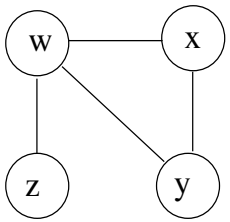
s_1

s_2



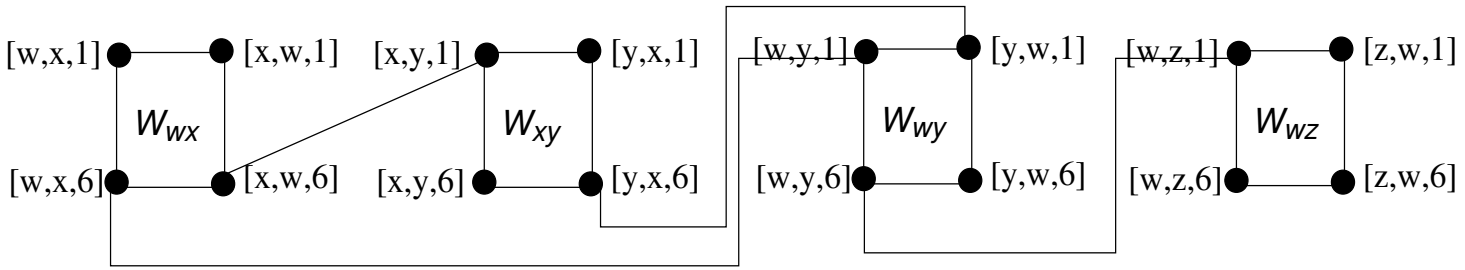
Ciclo Hamiltoniano

Para o vértice *y* acrescentamos arestas:



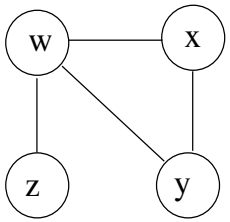
s_1

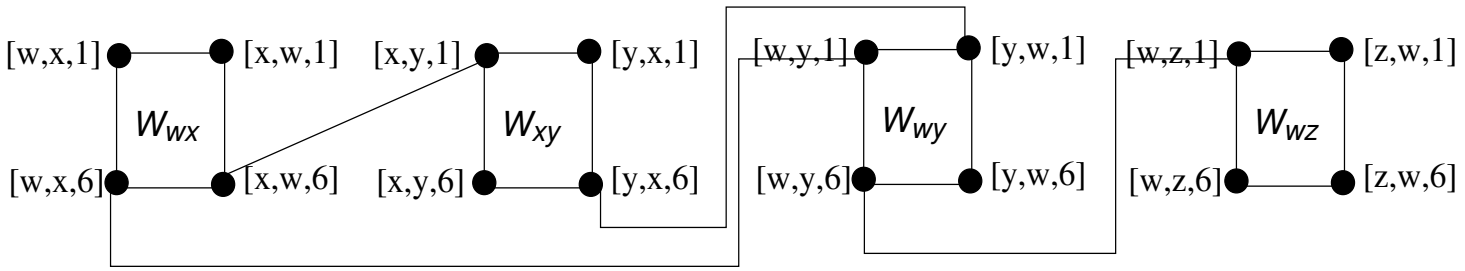
s_2



Ciclo Hamiltoniano

Para o vértice *z* não acrescentamos arestas:





Ciclo Hamiltoniano

- Agora incluímos arestas ligando cada vértice $[u, u^1, 1]$ com os seletores s_i's.
- Também incluímos arestas de cada vértice $[u, u^{\delta(u)}, 6]$ com os seletores.
- Incluir

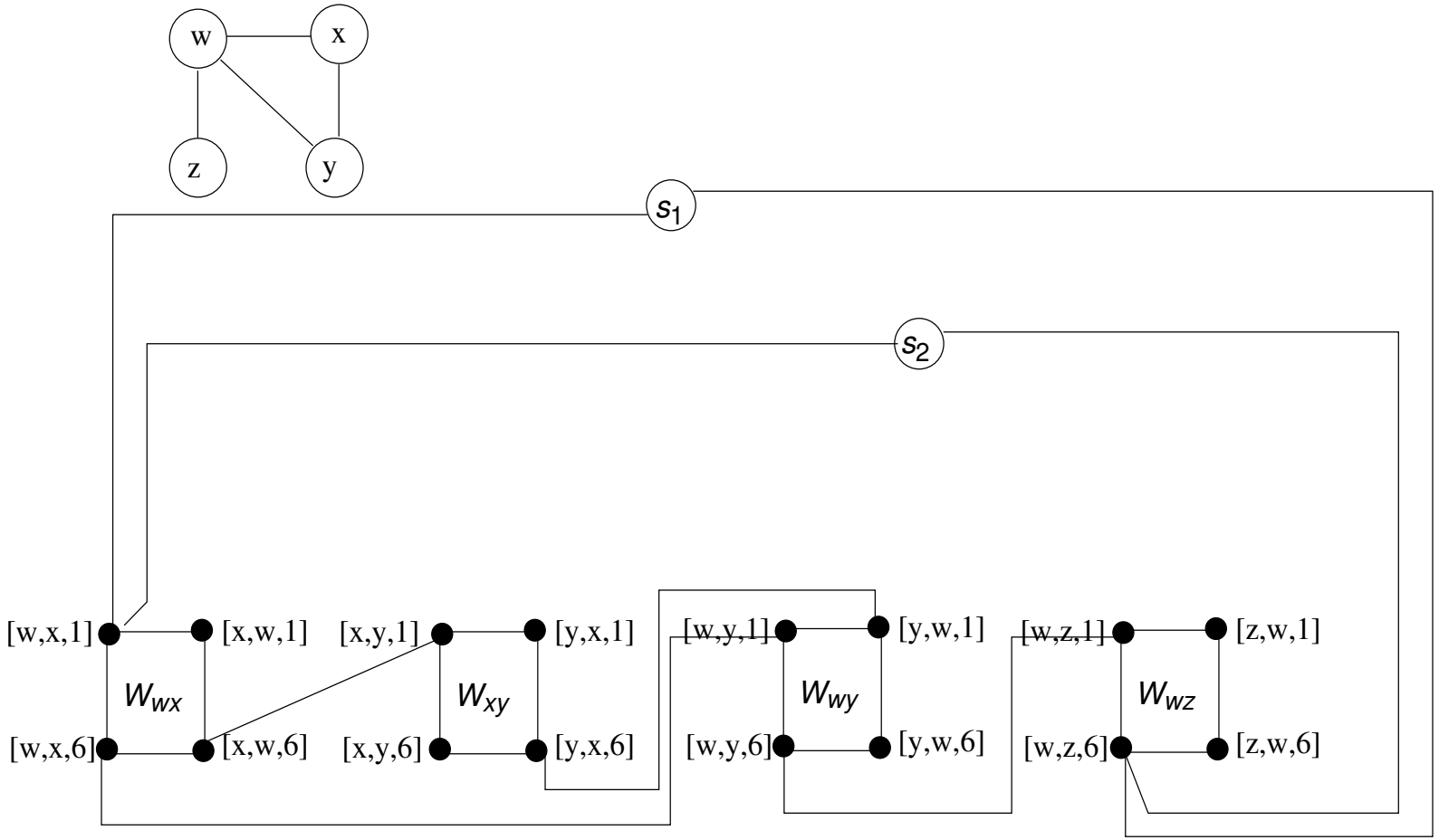
$$\{(s_j, [u, u^1, 1]) : u \in V \text{ e } 1 \leq j \leq k\}$$

e

$$\{(s_j, [u, u^{\delta(u)}, 6]) : u \in V \text{ e } 1 \leq j \leq k\}$$

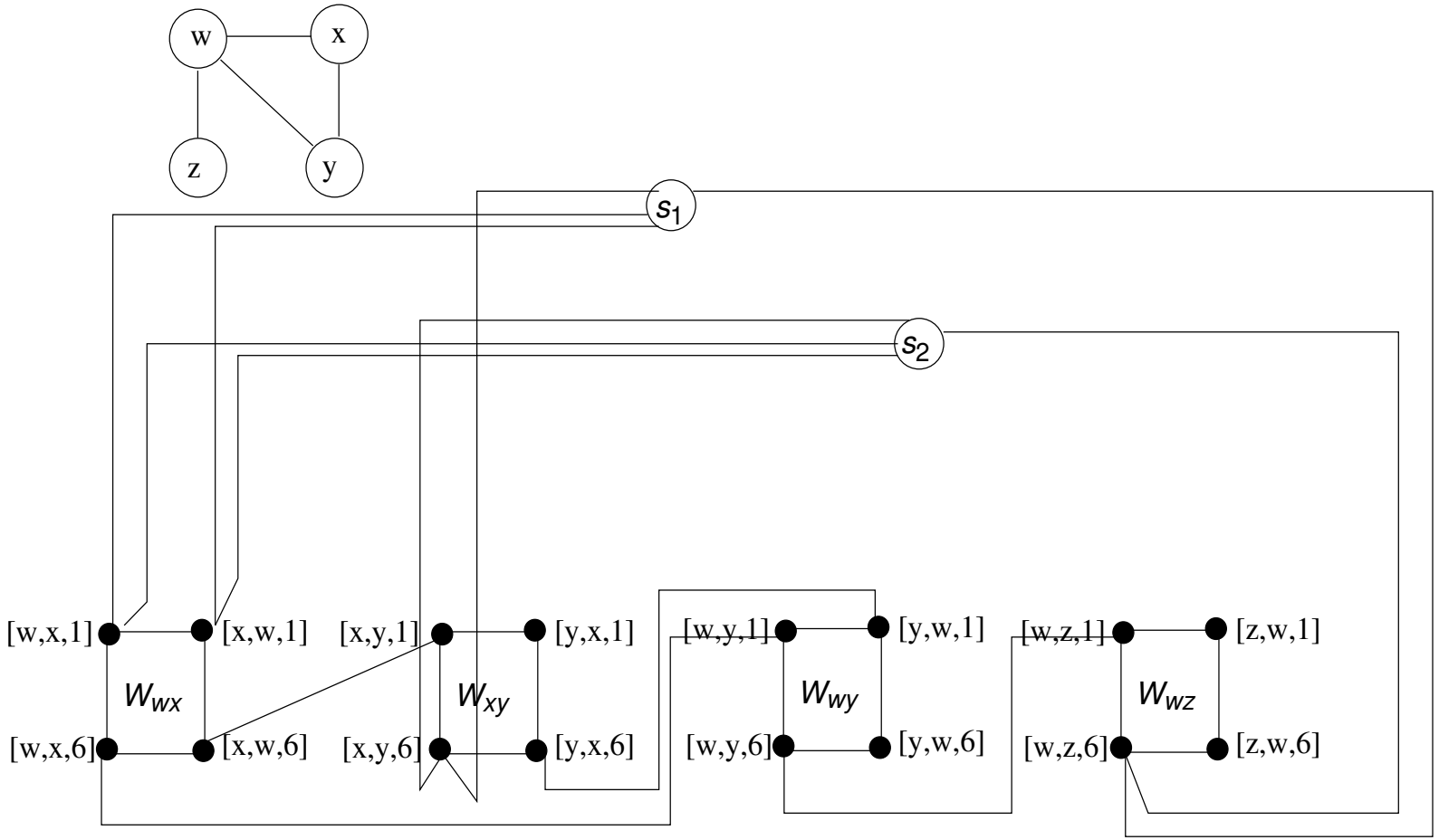
Ciclo Hamiltoniano

Para o vértice w ligar $[w, x, 1]$ e $[w, z, 6]$ com seletores:



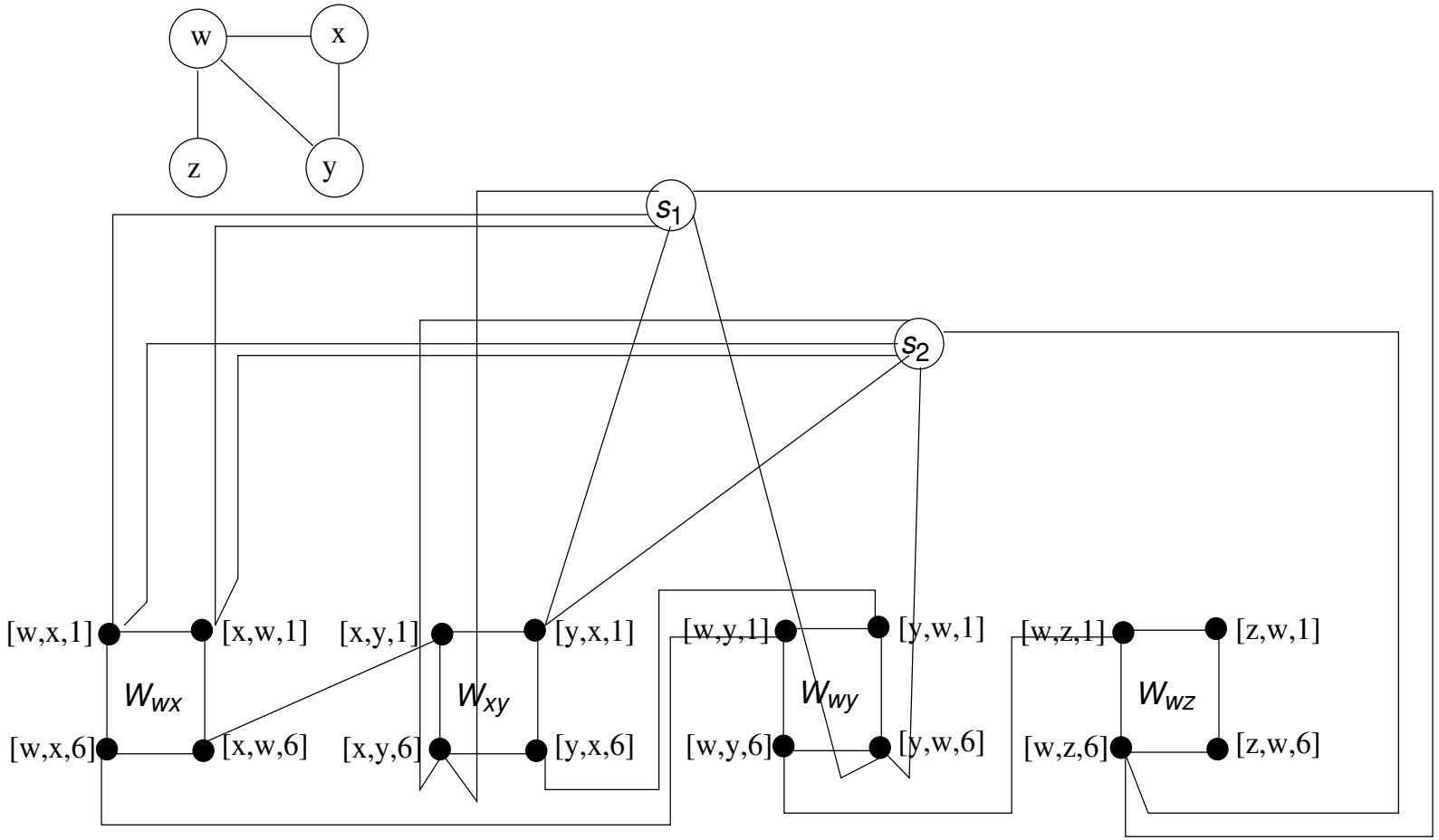
Ciclo Hamiltoniano

Para o vértice x ligar $[x, w, 1]$ e $[x, y, 6]$ com seletores:



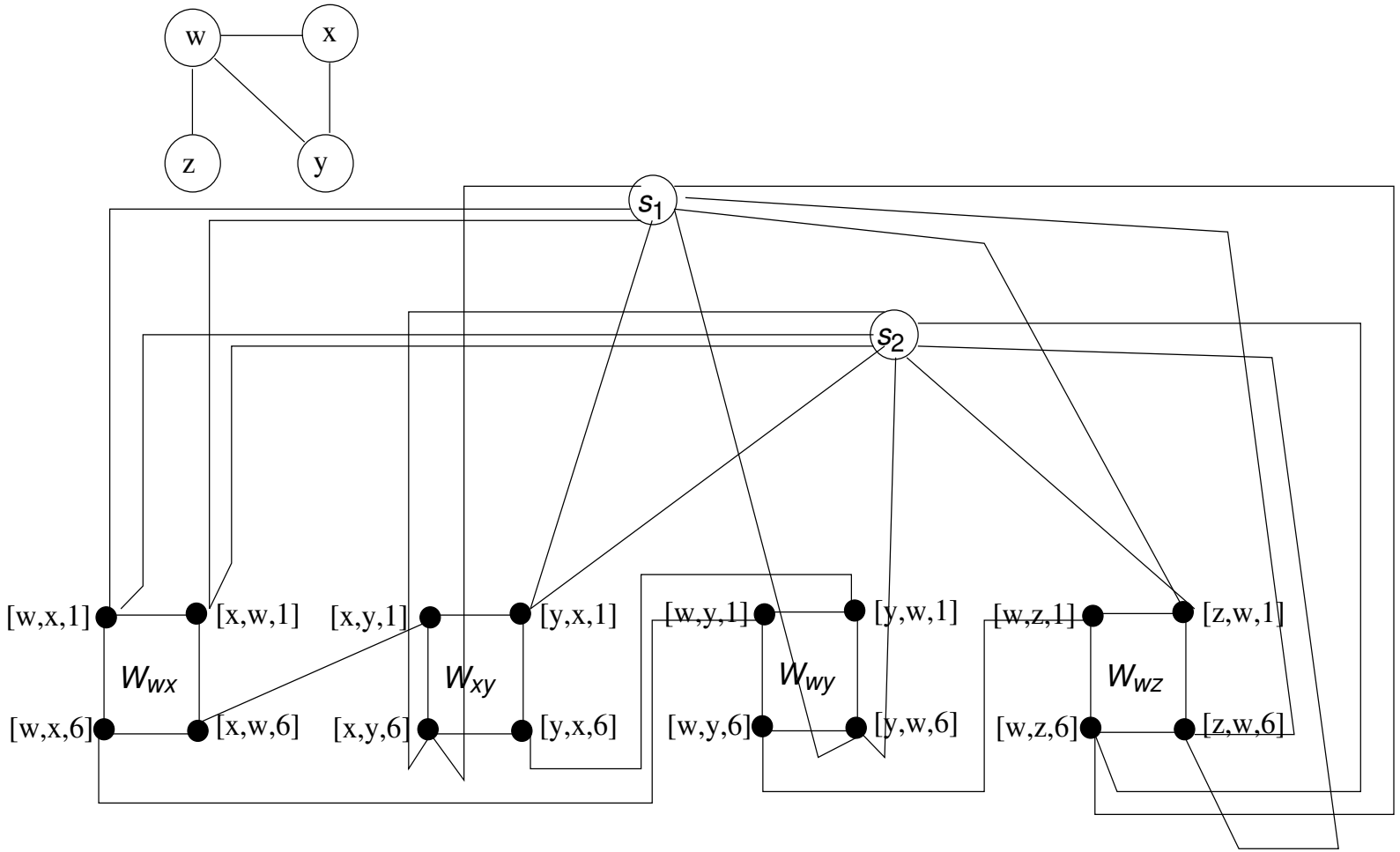
Ciclo Hamiltoniano

Para o vértice y ligar $[y, x, 1]$ e $[y, w, 6]$ com seletores:



Ciclo Hamiltoniano

Para o vértice z ligar $[z, w, 1]$ e $[z, w, 6]$ com seletores:



Ciclo Hamiltoniano

- A primeira coisa a mostrar é que esta construção é polinomial.
- Note que dado $G = (V, E)$ construímos $G' = (V', E')$ onde

$$|V'| = k + 12 \cdot |E|, \text{ com } k \leq |V|$$

- O número de arestas de G' é menor ou igual a $|V'|(|V'| - 1)/2$.
- Portanto a construção é polinomial.

Ciclo Hamiltoniano

- Temos que mostrar que efetivamente temos uma redução.
- G tem Vertex-Cover de tamanho k sse G' possui um Ciclo-Hamiltoniano.

Ciclo Hamiltoniano

Ida:

- Seja G instância do VERTEX-COVER e seja $V^* \subseteq V$ uma cobertura de tamanho k, $V^* = \{u_1, \ldots, u_k\}$.
- Ciclo-Hamiltoniano em G' é formado pelas arestas

(1) para cada $u_j \in V^*, \{([u_j, u_j^i, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \leq i \leq \delta(u_j) - 1\}$

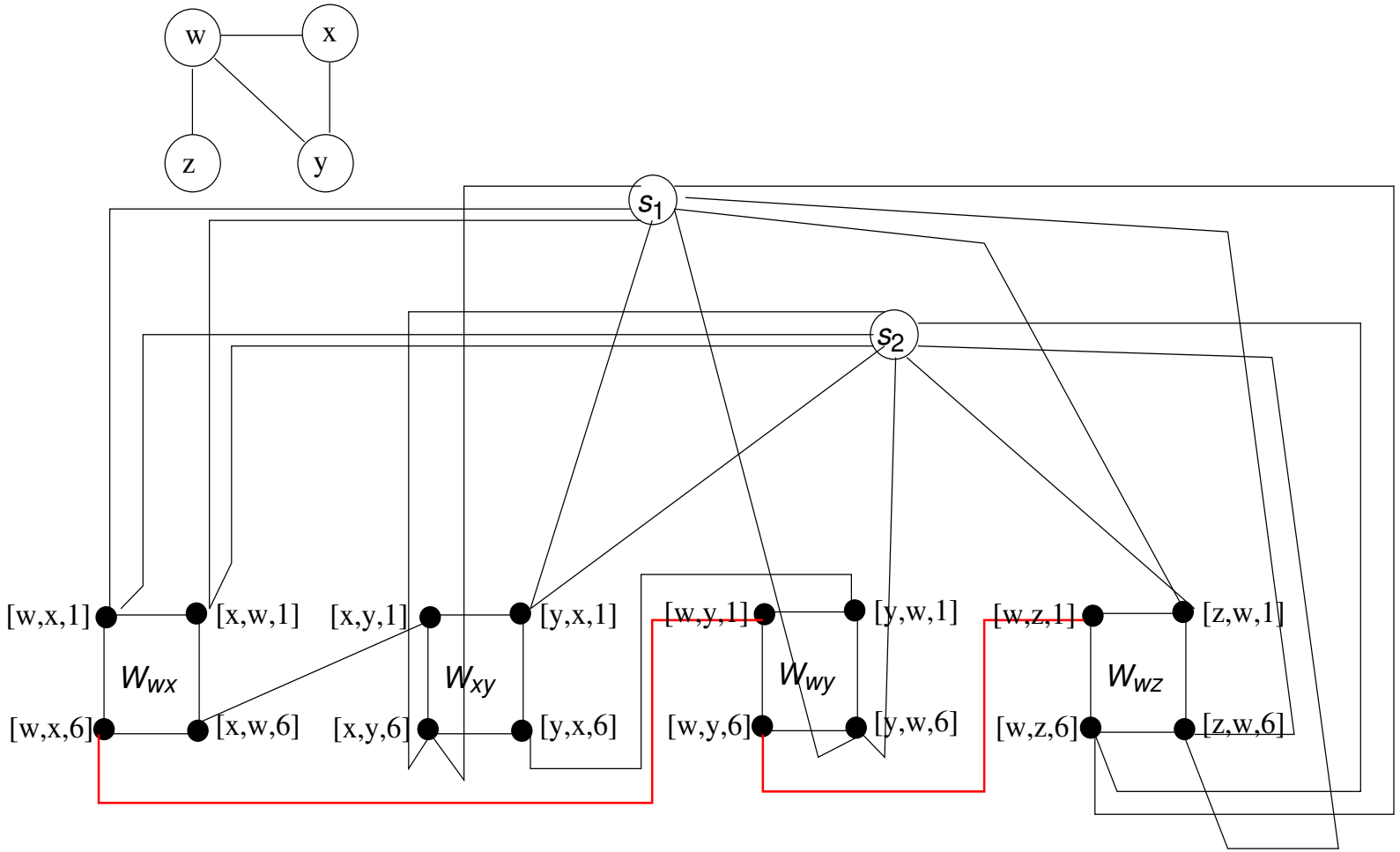
(2) $$\{(s_j, [u_j, u_j^1, 1]) : 1 \leq j \leq k\}$$

(3) $$\{(s_{j+1}, [u_j, u_j^{\delta(u_j)}, 6]) : 1 \leq j \leq k - 1\}$$

(4) $$\{(s_1, [u_k, u_k^{\delta(u_k)}, 6])\}$$

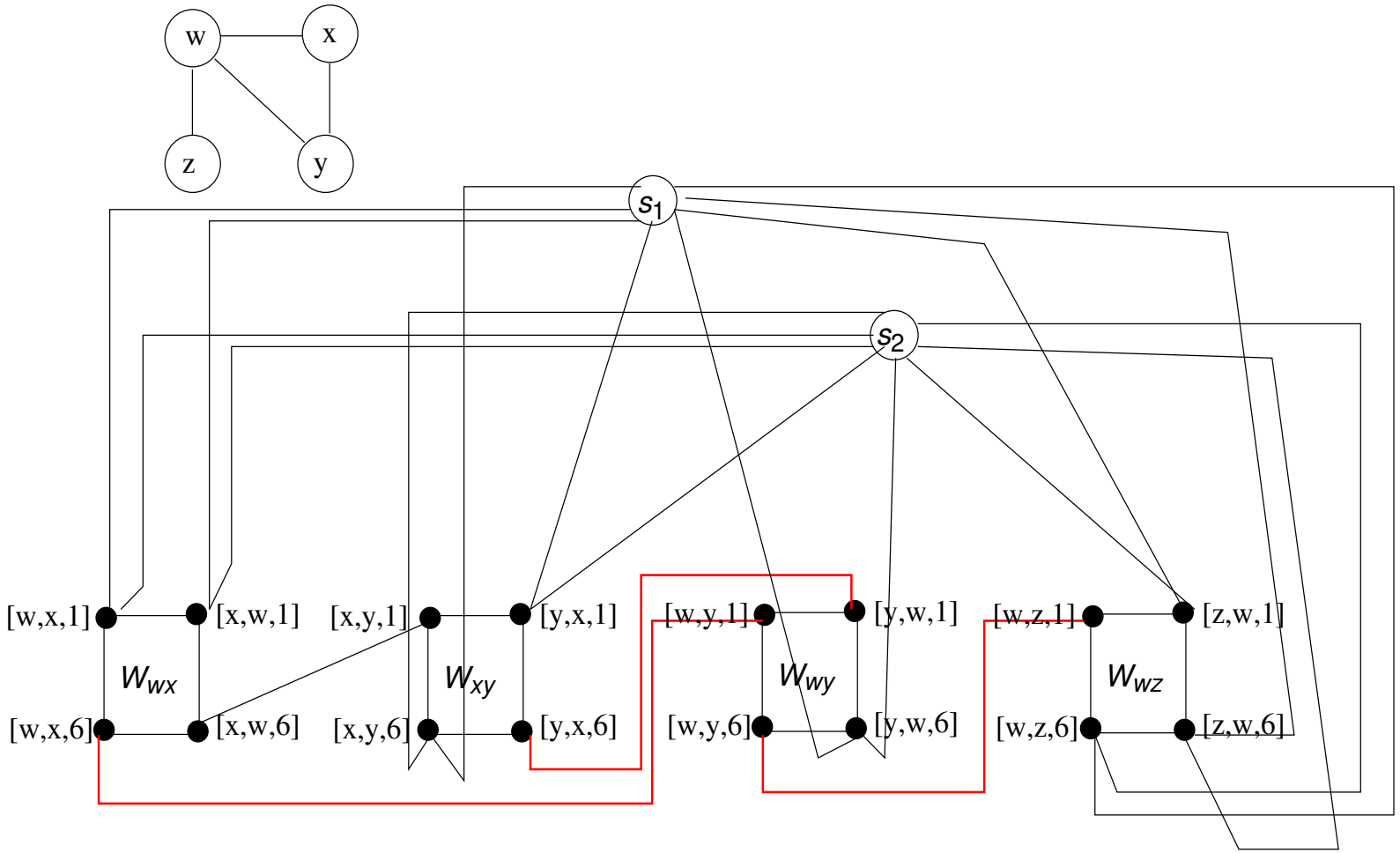
Ciclo Hamiltoniano

No nosso exemplo um VC é $\{w, y\}$. Incluir arestas (1) para w que conectam estruturas que correspondem a arestas incidentes em w:



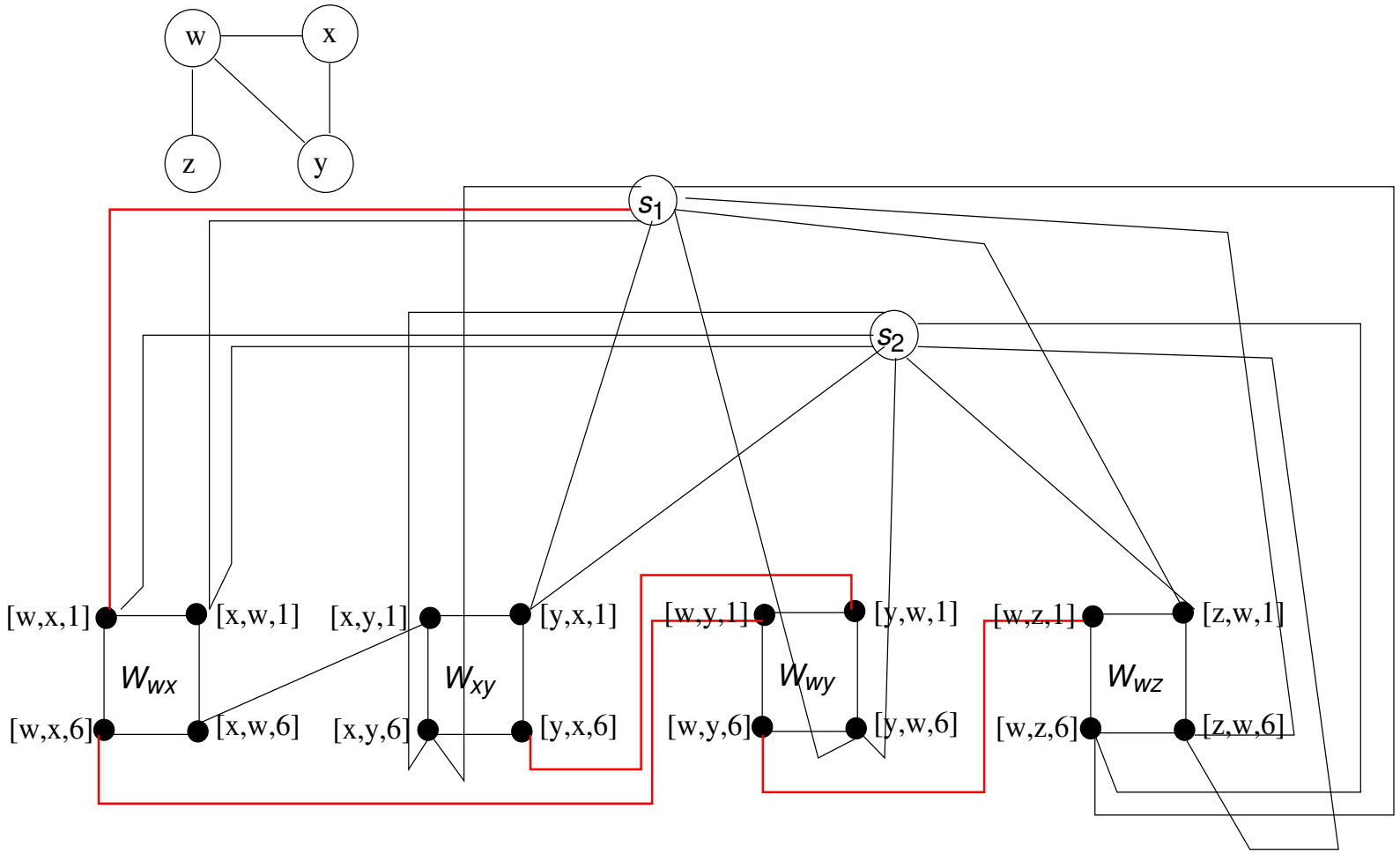
Ciclo Hamiltoniano

Incluir arestas (1) para *y* que conectam estruturas que correspondem a arestas incidentes em *y*:



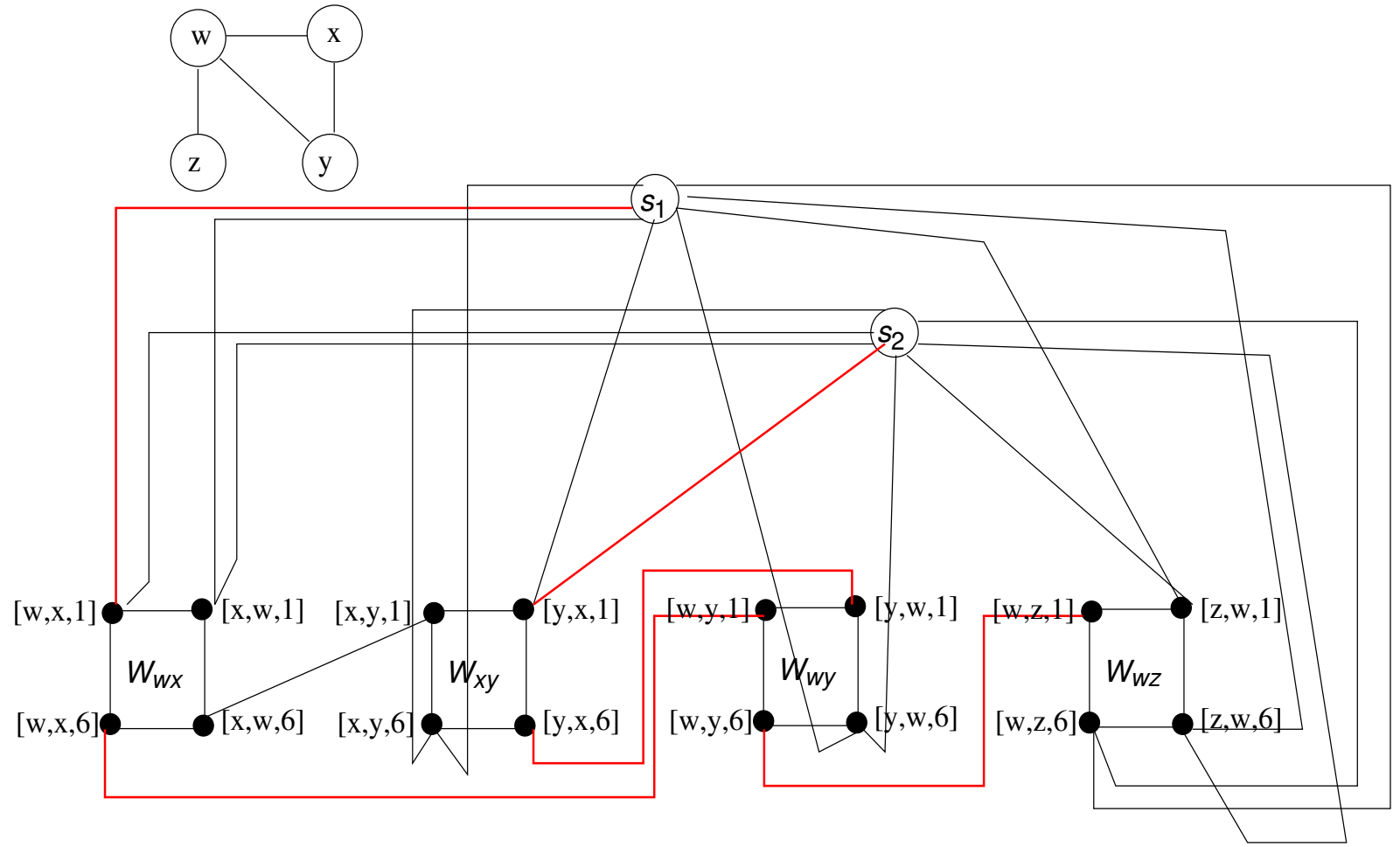
Ciclo Hamiltoniano

Incluir arestas (2) para *w*:



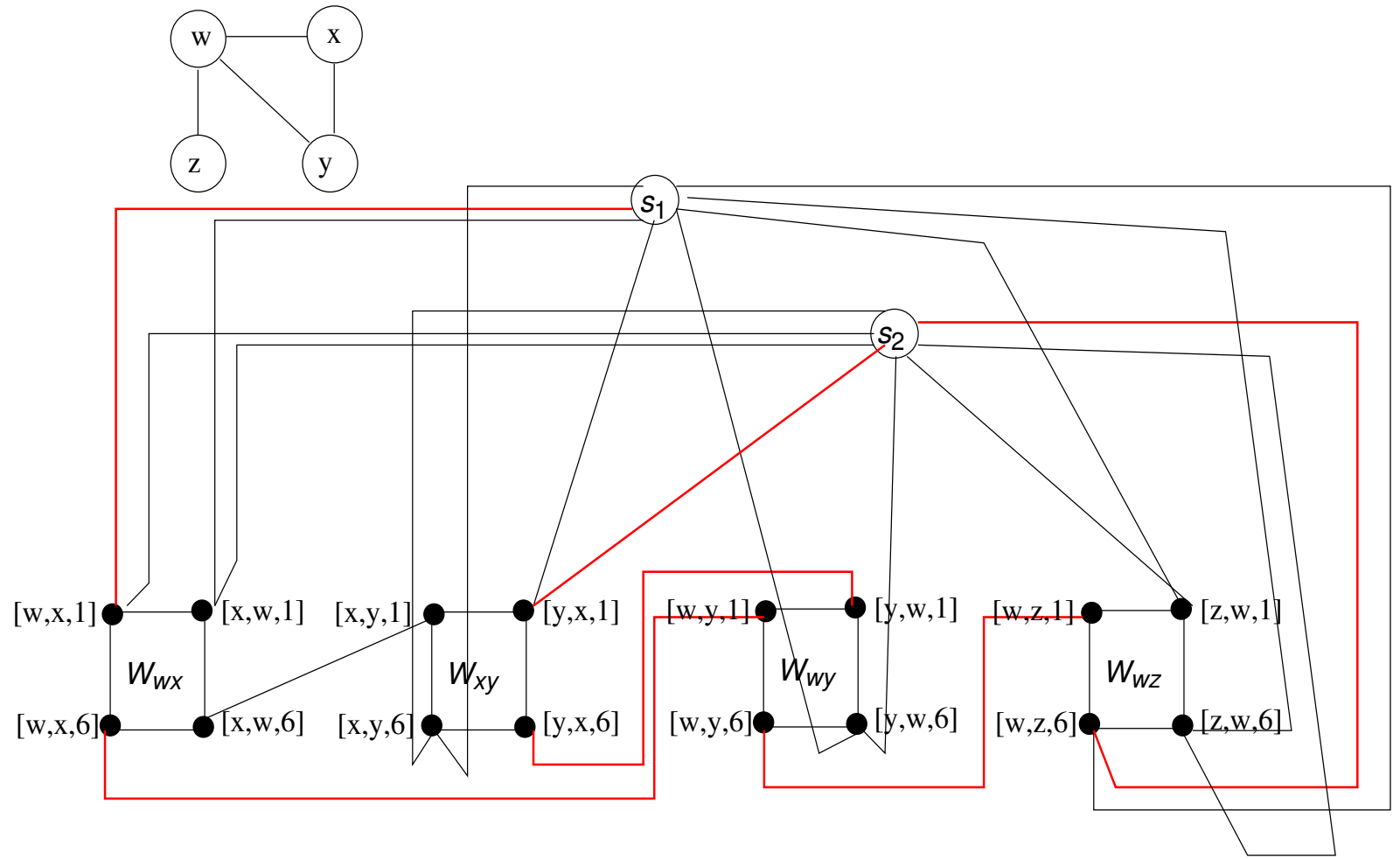
Ciclo Hamiltoniano

Incluir arestas (2) para y:



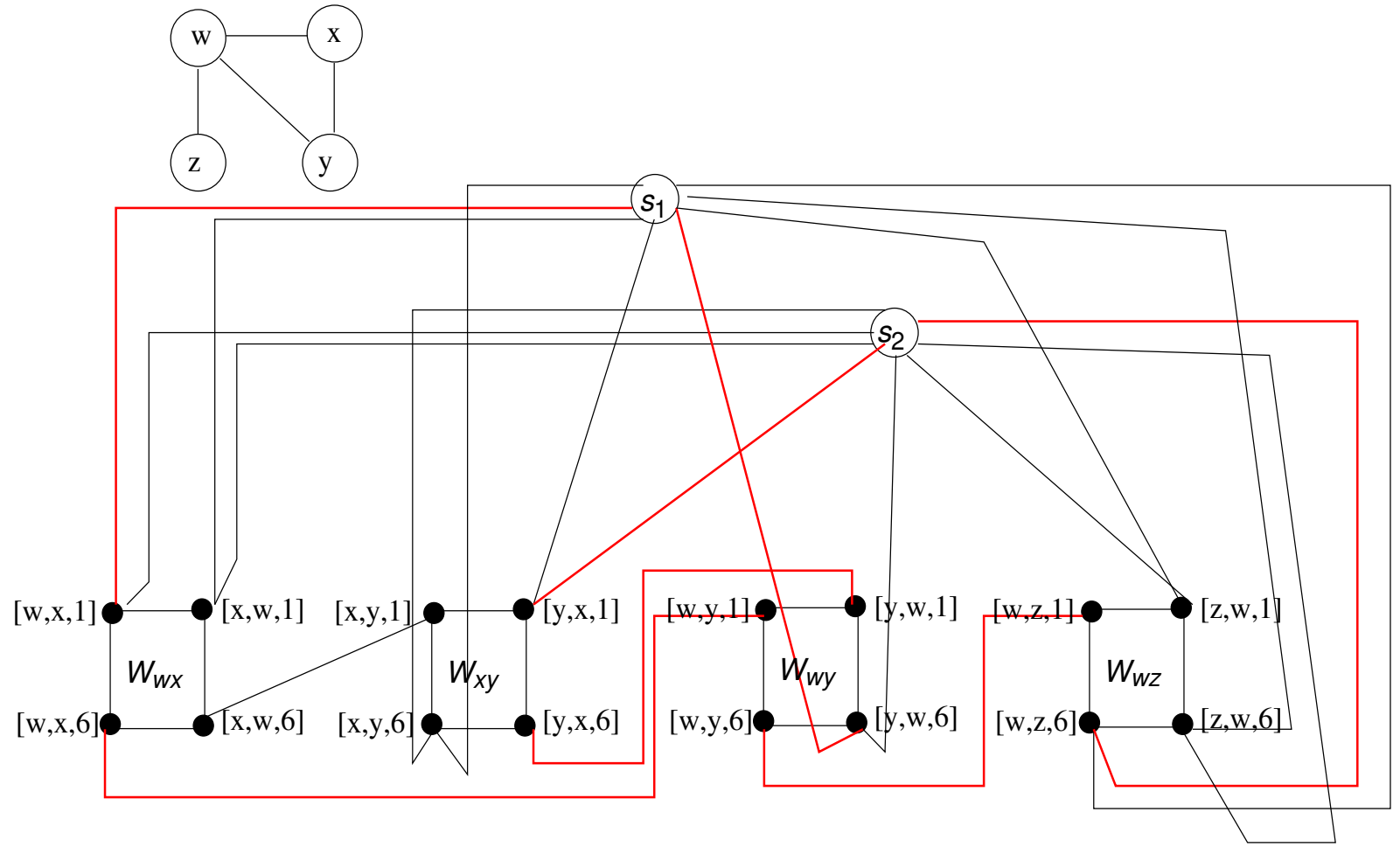
Ciclo Hamiltoniano

Incluir arestas (3) para *w*:



Ciclo Hamiltoniano

Incluir arestas (4) para y:



Ciclo Hamiltoniano

Ida:

- O importante é que como V^* é um VC então todas arestas estão cobertas e portanto todas as estruturas em G' estarão cobertas.
- Cada estrutura de G' ou tem um vértice incidente ou dois: Iremos formar um sub-caminho passando por todos os 12 vértices ou 2 sub-caminhos passando por 6 vértices cada.
- Pelo jeito que G' foi construído podemos ligar todas as estruturas que correspondem à arestas incidentes a um u_i.
- Ao terminar um sub-caminho correspondente as arestas incidentes a um u_i terminamos em um seletor que ligaremos a primeira estrutura de um outro vértice u_j do VC, e assim sucessivamente.

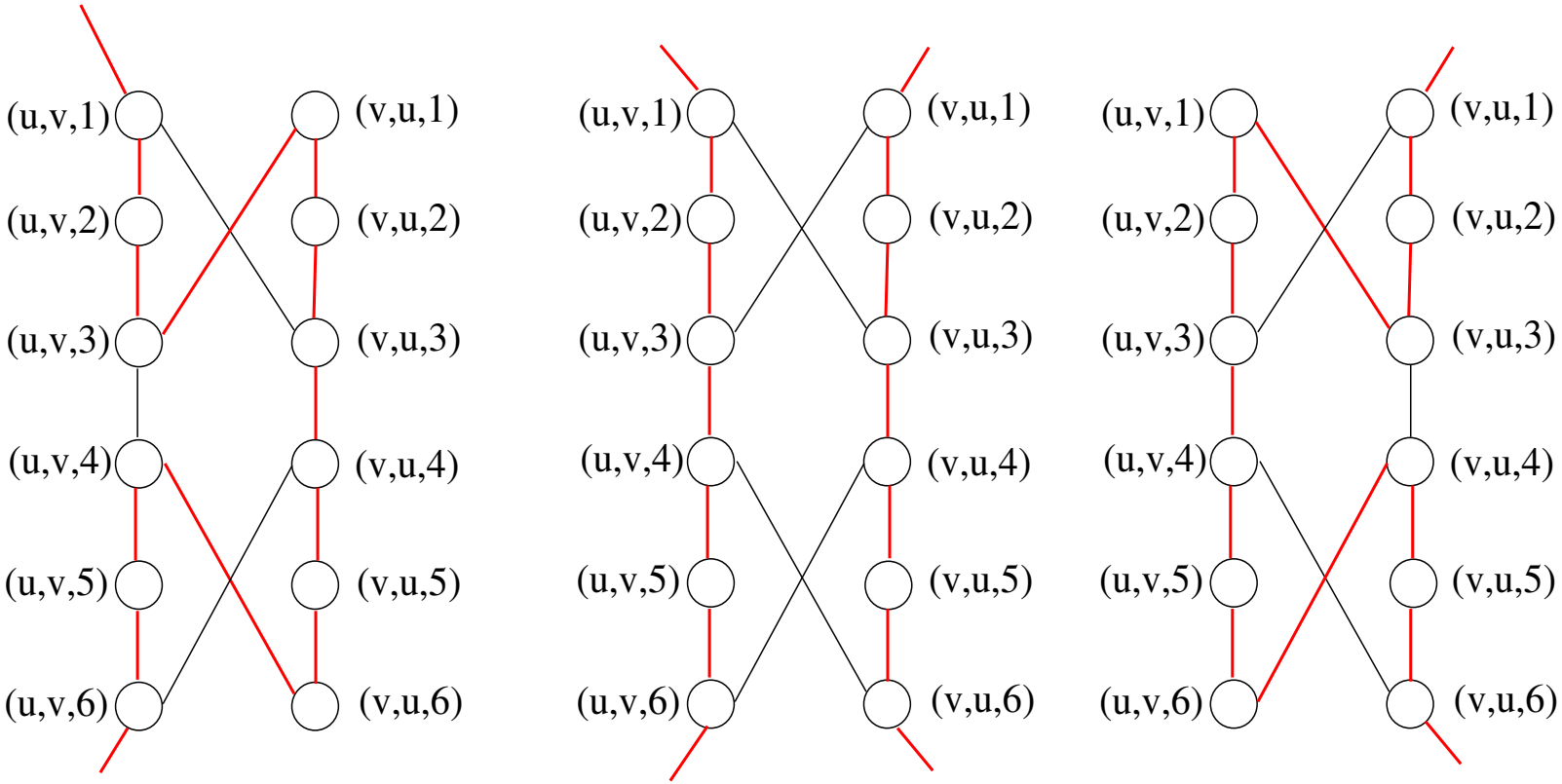
Ciclo Hamiltoniano

Volta:

- Seja $C' \subseteq E'$ arestas de G' que correspondem a um ciclo-hamiltoniano.
- Vamos mostrar que podemos achar uma cobertura V^* de G de tamanho k.
- Seja V^* definido como:

$$V^* = \{u \in V : (s_j, [u, u^1, 1]) \in C \text{ para algum } 1 \leq j \leq k\}$$

Ciclo Hamiltoniano



Ciclo Hamiltoniano

- Pela construção de uma estrutura, ao entrarmos por um vértice $[u, u^1, 1]$ da estrutura a única saída será pelo vértice $[u, u^1, 6]$.
- O vértice $[u, u^1, 6]$ só está ligado com $[u, u^2, 1]$ (ou algum s_i se $\delta(u) = 1$).
- A idéia chave é que ao entrarmos em um $[u, u^1, 1]$, visitamos todas as estruturas que correspondem a arestas incidentes a u, e só depois iremos para um próximo seletor s_j.
- Como C' é um ciclo-hamiltoniano sabemos que todas as estruturas são visitadas.
- Como cada estrutura representa uma aresta de G, isto implica que cada aresta de G é coberta pelos vértices selecionados.

Caixeiro Viajante (TSP)

- O problema do Caixeiro Viajante (Traveling Salesman Problem) é um dos mais conhecidos problemas de otimização.
- Sua entrada é um grafo com custo nas arestas.
- Devemos achar um ciclo de custo mínimo passando por todos os vértices exatamente uma vez.

Caixeiro Viajante (TSP)

- Há uma funcão de custo inteiro $c(i, j)$ para cada aresta que liga vértices i e j.
- Notem que não podemos mostrar que TSP é NPC, mas podemos estabelecer a dificuldade do problema mostrando que este é NP-Difícil.
- Vamos assumir que o grafo é completo.

Caixeiro Viajante (TSP)

Teorema

TSP é NP-Difícil.

- Vamos fazer uma redução em tempo polinomial do HAM-CICLO para o TSP tal que $G = (V, E)$ possui um ciclo hamiltoniano se e somente se $G' = (V', E')$ tiver um ciclo hamiltoniano com custo 0.

Continuação da Prova

Dado G como instância para HAM-CICLO vamos montar G' da seguinte forma:

$$V' = V$$

$$E' = \{(i,j) : \text{ para todo par } i,j \in V\}$$

$$c(i,j) = \begin{cases} 0 & \text{se aresta } (i,j) \in E \\ 1 & \text{se aresta } (i,j) \notin E \end{cases}$$

Continuação da Prova

$\Rightarrow$) Se G possuir um ciclo hamiltoniano então este ciclo tem custo 0 em G'.

$\Leftarrow$) Se G' possui um ciclo hamiltoniano de custo 0 então todas as arestas deste ciclo pertencem a G, e portanto G possui um ciclo hamiltoniano.

TSP

- Podemos considerar uma versão de decisão do problema:

$$\begin{aligned}\text{TSP} = \{\langle G, c, k\rangle : \quad & G = (V, E) \text{ é um grafo completo} \\ & c \text{ é uma funcão de custo } V \times V \to \mathbb{Z} \\ & k \in \mathbb{Z} \\ & G \text{ tem ciclo ham. de custo no máximo } k\}\end{aligned}$$

- É fácil mostrar que a versão de decisão do TSP é NPC.

Circuit Satisfiability

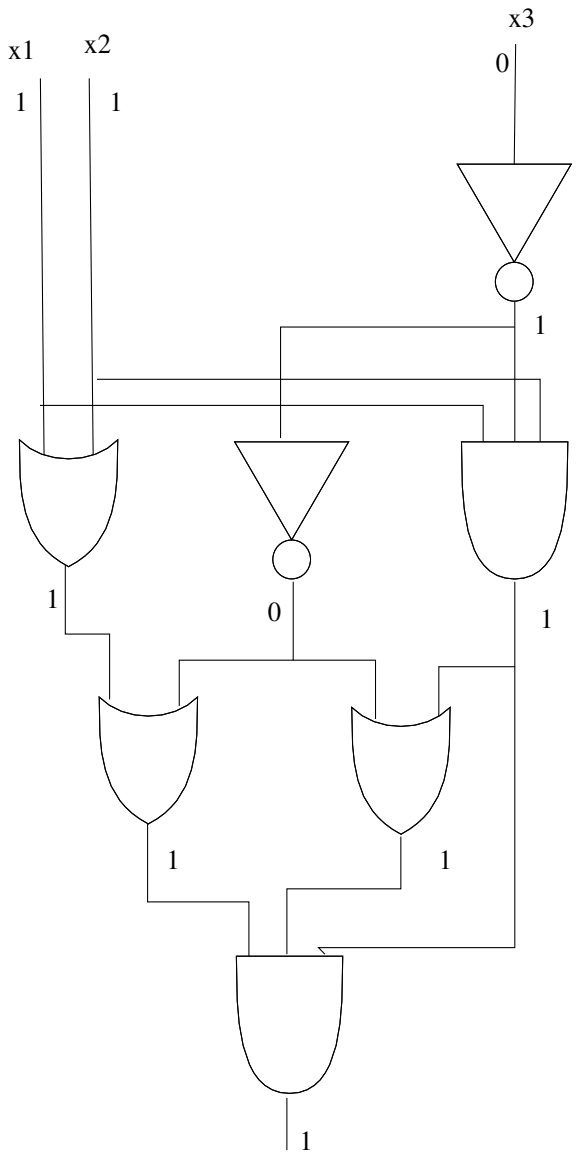
Para mostrar que SAT $\in$ NPC, vamos mostrar que Circuit Satisfiability (C-SAT) $\in$ NPC e em seguida reduzir C-SAT para SAT.

- No problema *Circuit Satisfiability* (C-SAT) consideramos um circuito lógico composto por portas:
  - AND denotado por $\wedge$.
  - OR denotado por $\vee$.
  - NOT denotado por $\neg$.
- Podemos formar um circuito booleano conectando diversas destas portas lógicas.
- A entrada do circuito é dada por n fios.
- A saída do circuito é dada por um fio.

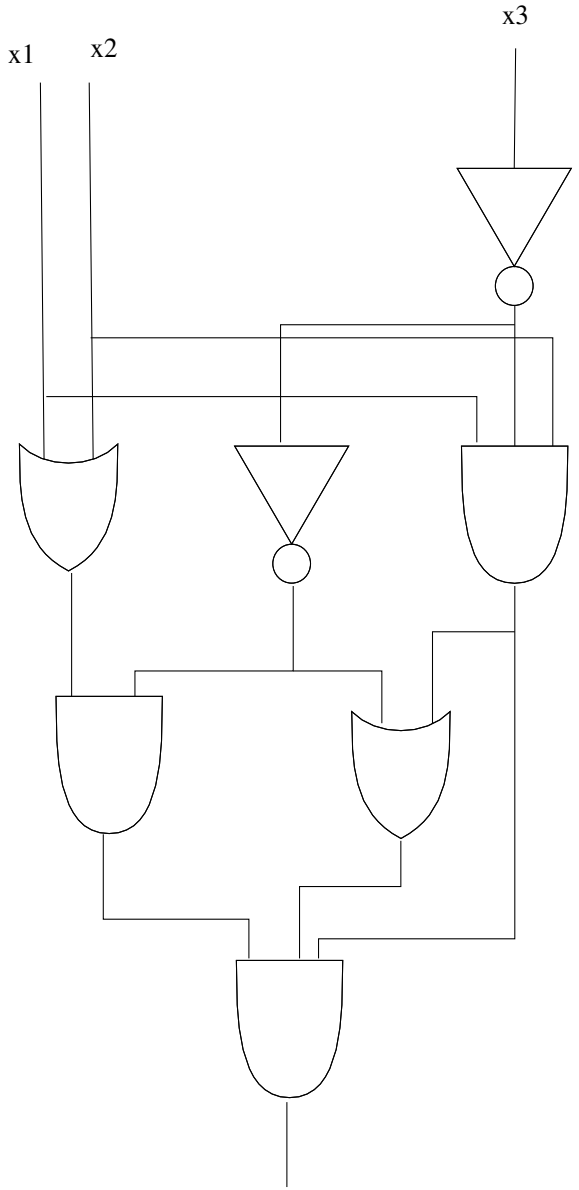
Circuit Satisfiability

- Uma atribuição para o circuito é uma atribuição lógica (0 ou 1) para os fios de entrada.
- Uma atribuição é dita ser *verdadeira* se resulta em uma saída com valor 1.
- Um circuito pode ser *satisfeito* se ele possui uma atribuição verdadeira.

Circuit Satisfiability



Circuit Satisfiability



Circuit Satisfiability

- Dado um circuito booleano, o problema C-SAT consiste em determinar se o circuito possui uma atribuição verdadeira.

  C-SAT $= \{\langle C \rangle : C$ é um circuito que possui atribuição verdadeira.

- Um algoritmo simples para o C-SAT tem complexidade de tempo $O(2^n(n+m))$ onde n é o número de portas de entrada e m o número de ligações.

Circuit Satisfiability

Lema

O problema C-SAT pertence a NP.

Prova.

- Temos que mostrar que existe um algoritmo de tempo polinomial que verifica C-SAT.
- Construir um algoritmo $A(x, y)$ que dado um circuito x, considera y como uma atribuição de valores de entrada para x
- O algoritmo simula o circuito e checa se a saída foi 1 ou 0. O algoritmo tem tempo polinomial.
- Se $x \in$ C-SAT, existe uma atribuição y tal que $A(x, y) = 1$.
- Se $x \notin$ C-SAT, nenhuma atribuição causará a saída 1 do circuito e portanto não tem como enganar o algoritmo.

□

Circuit Satisfiability

- Queremos mostrar que C-SAT pertence a classe NP-Completo.
- Falta mostrar então que C-SAT é NP-difícil, ou seja, para todo $Q \in$ NP, existe um algoritmo de redução polinomial F ($Q \triangleright_p$C-SAT).
- Algoritmo: Dado $x \in \{0,1\}^*$ construir circuito $F(x)$ tal que $x \in Q$ se e somente se $F(x) \in$ C-SAT.
- Vamos dar uma idéia do teorema de Cook (que considerou na verdade o problema SAT).

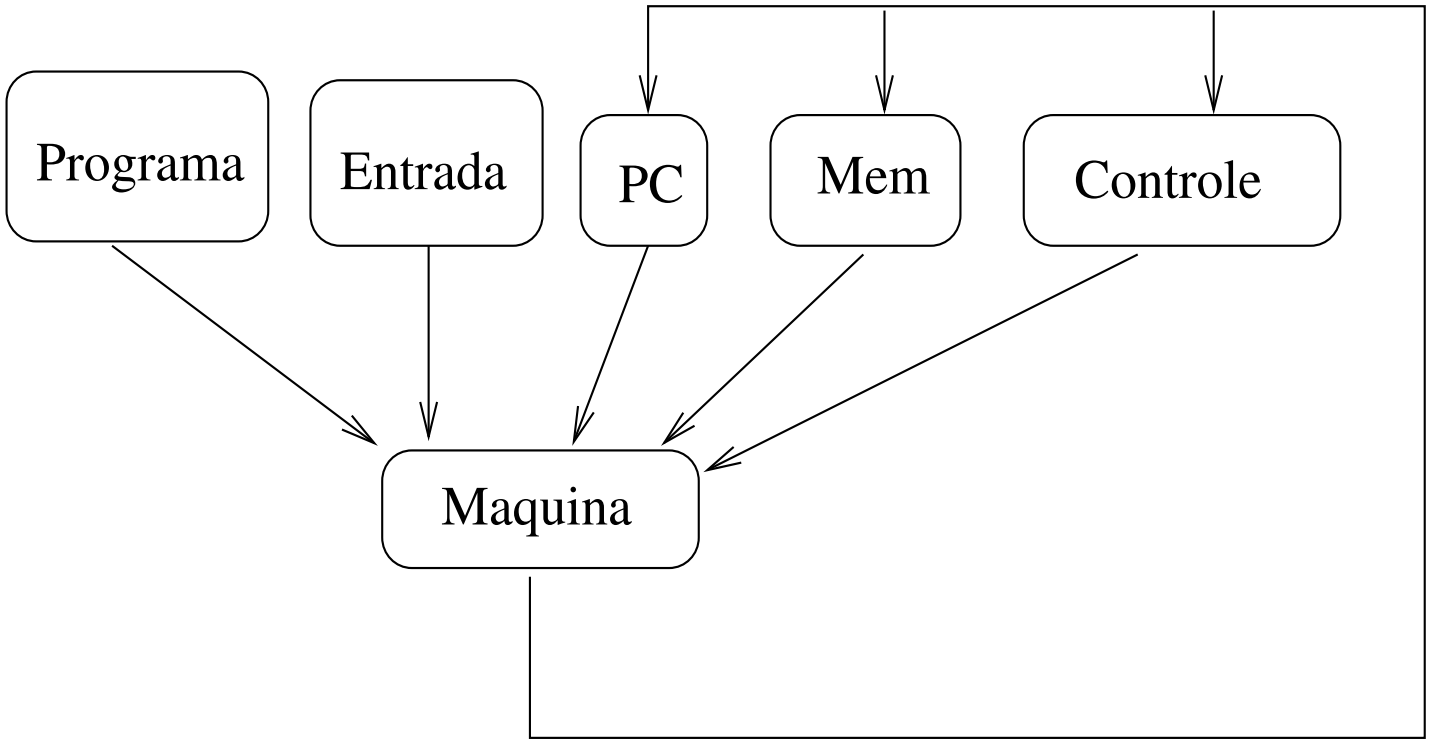
Circuit Satisfiability

Lema

C-SAT é NP-Difícil.

- Seja $Q \in$ NP.
- Existe um algoritmo verificador polinomial A para Q.
- Dado uma instância x para Q devemos montar uma instância $F(x)$ para C-SAT.
- O algoritmo verificador de Q recebe duas strings, x e y, de entrada. Sabemos que o tempo de execução limite para A é n^k e o tamanho limite para y é $|y| = n^{k'}$, onde k e k' são constantes.
- A idéia é montar um circuito que simula o algoritmo A executando em um computador de circuitos lógicos.

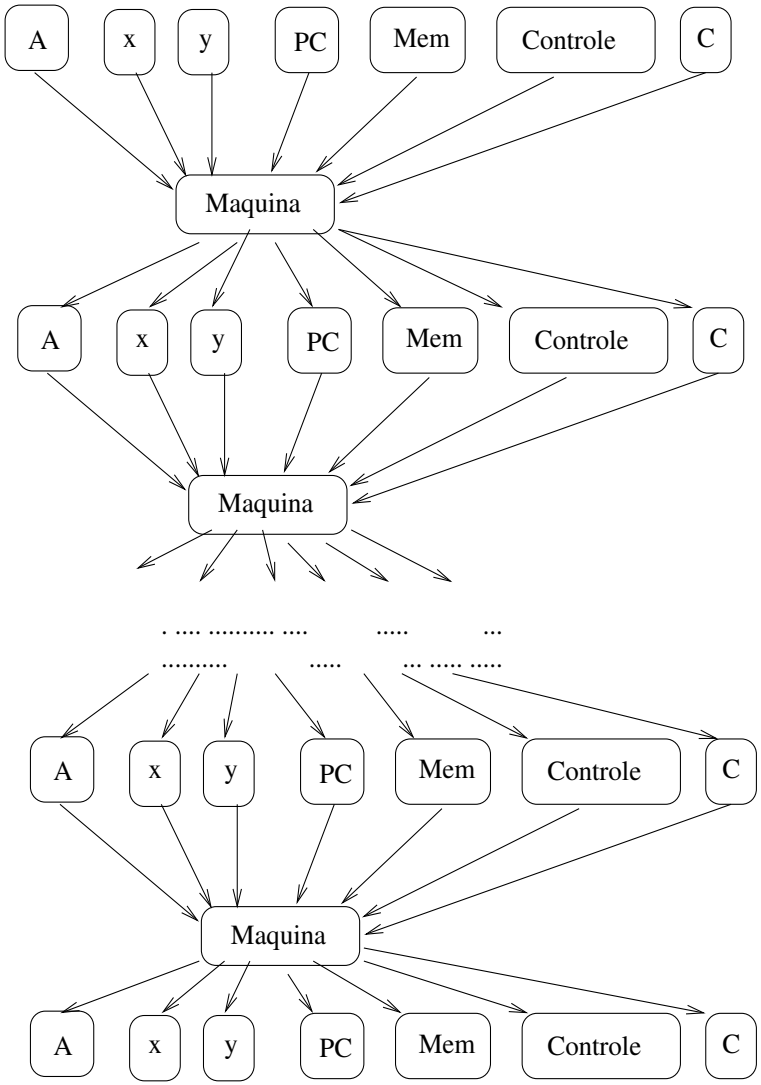
Continuação da prova



Continuação da prova

- Para um determinado x de tamanho n, o algoritmo executará no máximo n^k instruções, e o tamanho máximo de y é $n^{k'}$.
- Podemos montar n^k cópias desta "máquina" de tal forma que a saída de uma instrução do algoritmo re-alimente a "máquina" de um estado seguinte.
- O algoritmo A escreve sua saída final em um local específico que denotaremos por C.

Continuação da prova



Continuação da prova

- A máquina, o controle, o PC, e A tem sempre o mesmo tamanho independente de x.
- A memória e o espaço para y e x, tem tamanho polinomial em x. Como serão executados no máximo n^k instruções, todo este circuito tem tamanho polinomial.
- Todo o circuito, com exceção de y, é fixo, ou seja, dado um x construímos todo este circuito cuja **entrada** corresponde ao valor de y. A saída do circuito é um teste se em algum dos campos C está setado o valor 1.
- A instância $F(x)$ foi construída em tempo polinomial. Nos resta mostrar que $x \in Q$ se e somente se $F(x) \in$ C-SAT.

Continuação da prova

- Se $x \in Q$ então há um y tal que $A(x, y) = 1$. Portanto há uma entrada para o circuito $F(x)$ cujo valor de saída deste é 1.

- Se $x \notin Q$, então nenhum valor de y fará $A(x, y) = 1$ e portanto não existe valor de entrada para o circuito que faça com que ele tenha valor de saída 1.

□

Teorema

C-SAT é NP-Completo.

Satisfatibilidade

- Relembrando: A linguagem SAT é aquela que contém fórmulas booleanos com uma atribuição verdadeira.

$$\text{SAT} = \{\langle f \rangle : \quad f \text{ é uma fórmula booleana que admite uma atribuição verdadeira } \}$$

Satisfatibilidade

Lema

SAT pertence a NP.

- Dado uma fórmula f para o problema, se $f \in$ SAT existe uma atribuição y de valores para as variáveis que fazem com que f seja verdadeira.
- O algoritmo deve atribuir os valores definidos por y para cada uma das variáveis e então deve avaliar a fórmula.
- Se y for uma atribuição verdadeira a fórmula terá uma resposta verdadeira.
- É claro que este algoritmo pode ser implementado para executar em tempo polinomial.

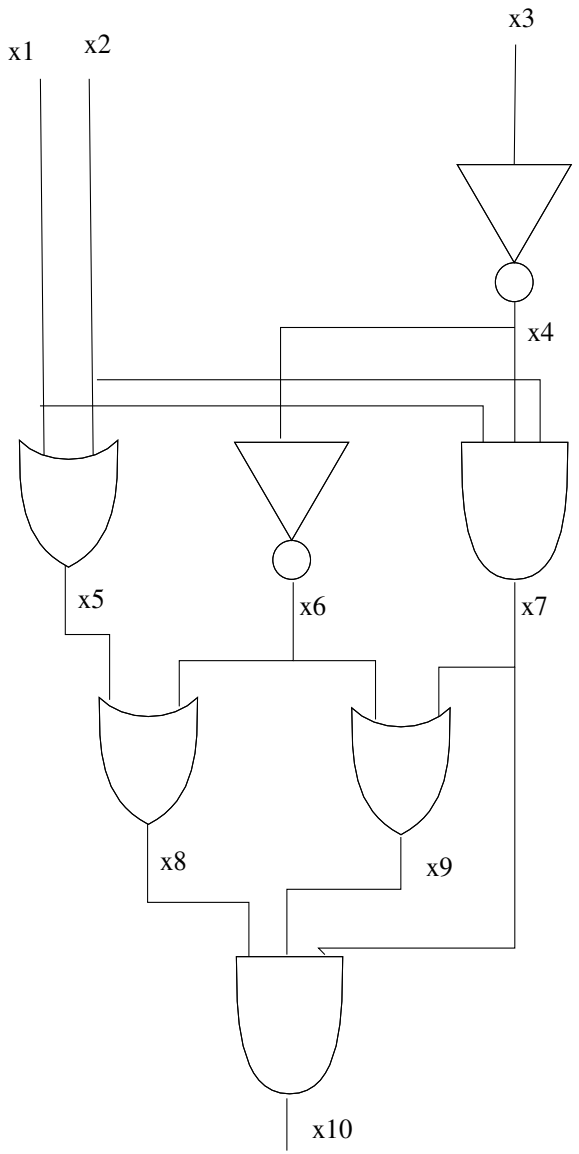
Satisfatibilidade

Lema

SAT é NP-Difícil.

- Vamos mostrar uma redução polinomial do problema C-SAT para SAT.
- Dado um circuito c temos que montar uma fórmula f tal que $c \in$ C-SAT se e somente se $f \in$ SAT. A transformação deve ter tempo polinomial.
- Dado um circuito c criamos uma variável x_i para cada ligação i do circuito.

Continuação da Prova



Continuação da Prova

Para cada fio escrevemos uma fórmula que computa a saída da porta lógica correspondente.

Ex: $x_5 \leftrightarrow (x_1 \vee x_2)$

Assim, x_5 terá o valor correspondente a saída da porta lógica correspondente. O circuito completo será descrito como:

$$
\begin{aligned}
f = x_{10} \wedge \quad & (x_4 \leftrightarrow (\neg x_3)) \\
\wedge \quad & (x_5 \leftrightarrow (x_1 \vee x_2)) \\
\wedge \quad & (x_6 \leftrightarrow (\neg x_4)) \\
\wedge \quad & (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
\wedge \quad & (x_8 \leftrightarrow (x_5 \vee x_6)) \\
\wedge \quad & (x_9 \leftrightarrow (x_6 \vee x_7)) \\
\wedge \quad & (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

Continuação da Prova

- Dado um circuito c geramos uma fórmula f como exemplificado e isto pode ser feito em tempo polinomial em relação ao tamanho de c.
- Se c tiver uma atribuição verdadeira então atribuindo os valores das variáveis x_is com os correspondentes valores dos fios i, a formúla f será satisfeita.
- Se f tiver uma atribuição verdadeira é porque a última variável, que corresponde ao último fio, tem valor 1 e todas as demais cláusulas também possuem valor 1.
- Como cada uma das demais cláusulas possuem valor 1, elas definem corretamente os valores dos fios. Portanto obtemos uma atribuição verdadeira para o circuito c.

Satisfatibilidade

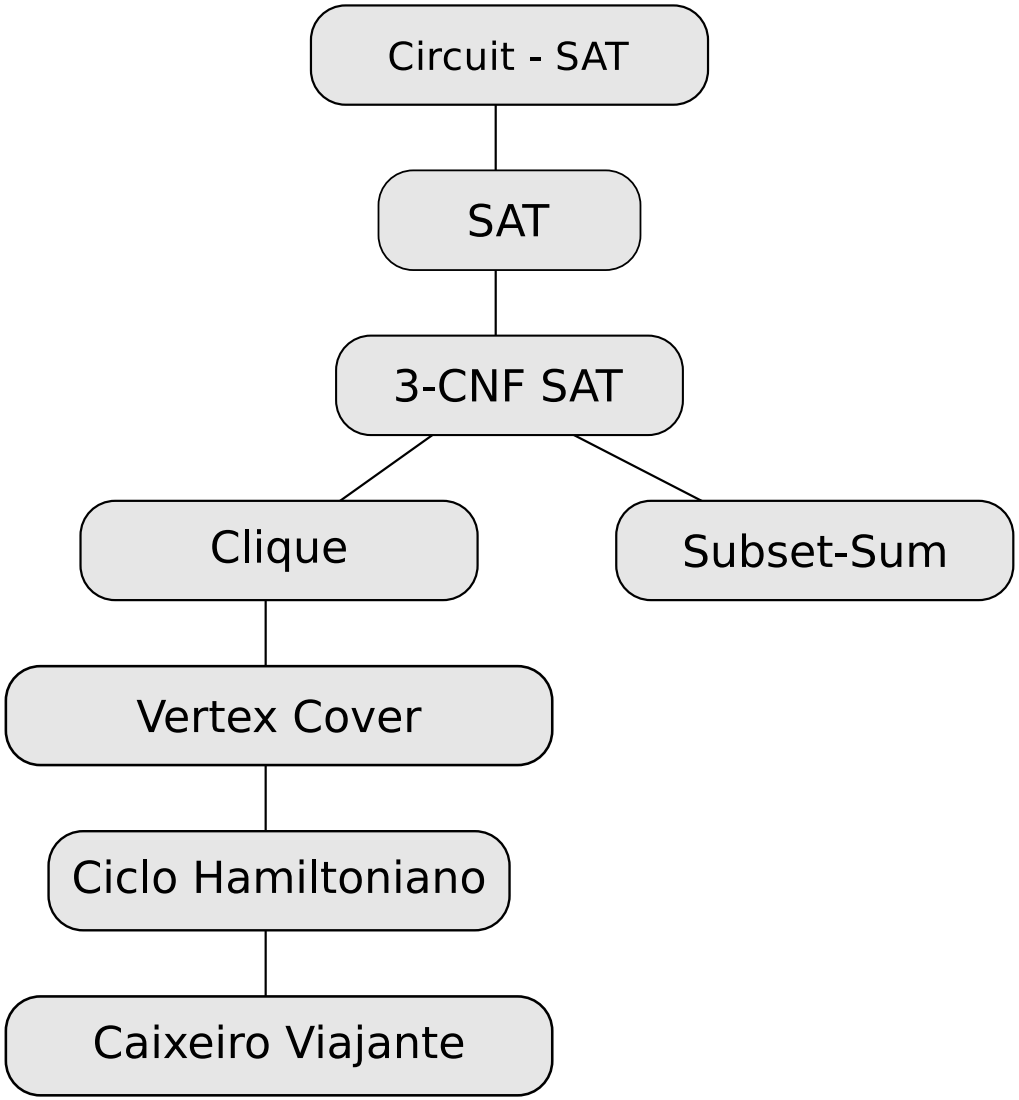
Teorema

SAT é NP-Completo.

Prova. Direto usando-se os dois últimos lemas. □

Resumo

Alguns problemas tipicamente usados para provar NP-completude



https://en.wikipedia.org/wiki/NP-completeness