Projeto e Análise de Algoritmos

NP-completude

Lehilton Pedrosa

Primeiro Semestre de 2020

Introdução à NP-complexidade

Algoritmos eficientes e ineficientes

- Para vários problemas, vimos algoritmos rápidos:
  - Ordenação: $O(n \log n)$
  - Multiplicação de matrizes: $O(n^{2,72})$
  - Caminho mais curto*: $O(mE)$
  - Circuito euleriano: $O(V + E)$
- Para outros, só conhecemos algoritmos lentos:
  - Problema da mochila: $O(2^n)$
  - Caminho mais longo*: $O(m!2^m E)$
  - Circuito hamiltoniano: $O(2^V)$

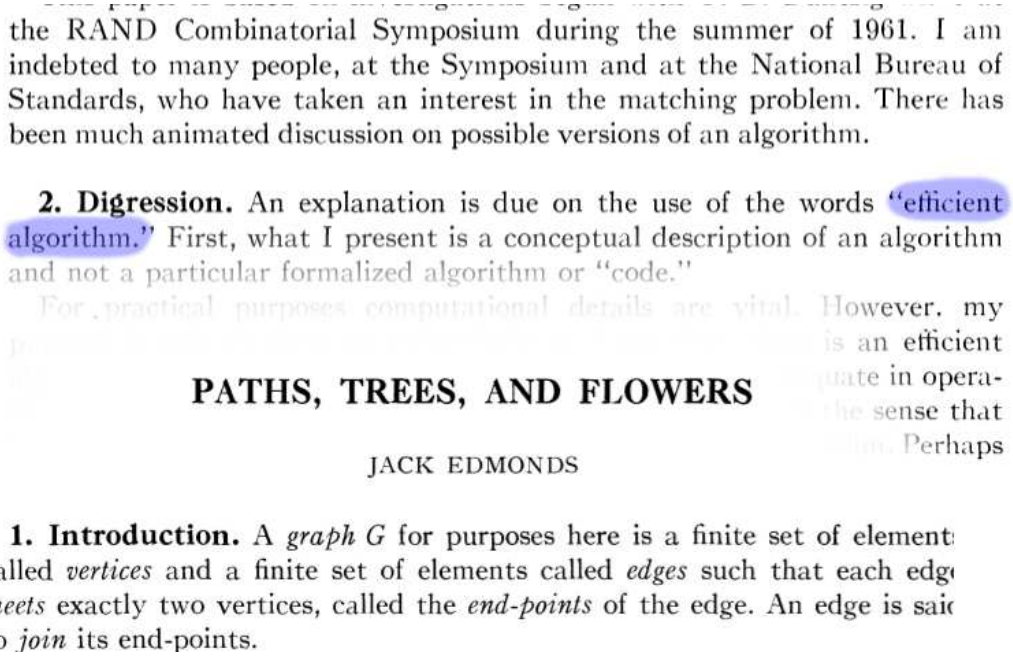
Como identificar algoritmos rápidos?

*m é o tamanho do caminho

Algoritmos de tempo polinomial

Um algoritmo é polinomial se o tempo de execução for limitado por $O(n^k)$ para alguma constante k.

- nesse caso, dizemos que ele é um algoritmo eficiente
- não necessariamente é rápido na prática
- mas exclui muitos dos algoritmos considerados lentos



the RAND Combinatorial Symposium during the summer of 1961. I am indebted to many people, at the Symposium and at the National Bureau of Standards, who have taken an interest in the matching problem. There has been much animated discussion on possible versions of an algorithm.

2. Digression. An explanation is due on the use of the words "efficient algorithm." First, what I present is a conceptual description of an algorithm and not a particular formalized algorithm or "code."

For practical purposes computational details are vital. However, my is an efficient uate in opera- the sense that Perhaps

PATHS, TREES, AND FLOWERS

JACK EDMONDS

1. Introduction. A *graph* G for purposes here is a finite set of element called *vertices* and a finite set of elements called *edges* such that each edg *meets* exactly two vertices, called the *end-points* of the edge. An edge is sai to *join* its end-points.

Organizando os problemas

Por que se preocupar com isso?

- desconhecemos algoritmos rápidos para vários problemas
- acreditamos que não há algoritmos eficientes para eles
- queremos saber quais deles têm algoritmos polinomiais!

Antes vamos discutir:

1. como representar problemas?
2. como comparar problemas?

Respondemos essas perguntas da seguinte forma:

1. representamos problemas como linguagens formais
2. comparamos problemas com um tipo de redução

Problemas de decisão

Um problema de decisão é um problema em que a resposta de cada elemento da entrada é **sim** ou **não**.

São problemas de decisão:

- Dado um número m, m é primo?
- Dadas as posições das peças em um tabuleiro de xadrez, o rei está em xeque?

Não são problemas de decisão:

- soma, ordenação, caminho mínimo etc.

Por que estudar problemas de decisão?

- é mais simples estudá-los do que problemas em geral
- várias situações são postas como problemas de decisão
- às vezes, decidir se existe alguma solução para um problema em geral é tão difícil quanto encontrá-la

Versão de decisão de problema de busca

Problema de busca do ciclo hamiltoniano

- **Entrada:** um grafo G
- **Solução:** um ciclo em G que percorre todos vértices

Problema de decisão do ciclo hamiltoniano

- **Entrada:** um grafo G
- **Pergunta:** existe ciclo em G que percorre todos vértices?

Suponha que sabemos decidir em tempo polinomial

- como encontrar em tempo polinomial?
- descubra uma aresta por vez (exercício)

Versão de decisão de problema de otimização

Problema do Caixeiro Viajante

- **Entrada:** grafo ponderado G
- **Solução:** um ciclo hamiltoniano C
- **Objetivo:** minimizar o peso de C

Versão de decisão

- **Entrada:** grafo ponderado G, parâmetro k
- **Solução:** um ciclo hamiltoniano C
- **Pergunta:** existe ciclo C de peso no máximo k?

Suponha que sabemos decidir em tempo polinomial

- como determinar o valor ótimo em tempo polinomial?
- faça uma busca binária (exercício)

Olhando para frente

Vamos classificar problemas de decisão nas classes:

P: podem ser resolvidos em tempo polinomial

NP: têm soluções curtas que podem ser verificadas em tempo polinomial

NP-difícil: pelo menos tão difíceis quanto quaisquer outros problemas em NP

NP-completo: são NP e NP-difícil

Exemplos de problemas em P

Soma de elemento:

- **Entrada:** conjunto de números S
- **Pergunta:** existe $n \in S$ tal que $n = \sum_{m \in S \setminus \{n\}} m$?

Conexidade:

- **Entrada:** grafo G
- **Pergunta:** o grafo é conexo?

Caminho mínimo:

- **Entrada:** grafo ponderado G, vértices s e t e inteiro k
- **Pergunta:** existe caminho de s até t de peso no máximo k?

4-Coloração:

- **Entrada:** uma mapa de regiões
- **Pergunta:** é possível colorir as regiões com até 4 cores de forma que regiões adjacentes tenham cores distintas?

Exemplos de problemas em NP-completo

Bipartição:

- **Entrada:** conjunto de números S
- **Pergunta:** existe $N \subseteq S$ tal que $\sum_{m \in N} m = \sum_{m \in S \setminus N} m$?

Ciclo hamiltoniano:

- **Entrada:** grafo G
- **Pergunta:** existe um ciclo hamiltoniano?

Caminho mais longo:

- **Entrada:** grafo ponderado G, vértices s e t e inteiro k
- **Pergunta:** existe caminho de s até t de peso no mínimo k?

3-Coloração:

- **Entrada:** uma mapa de regiões
- **Pergunta:** é possível colorir as regiões com até 3 cores de forma que regiões adjacentes tenham cores distintas?

Uma motivação prática

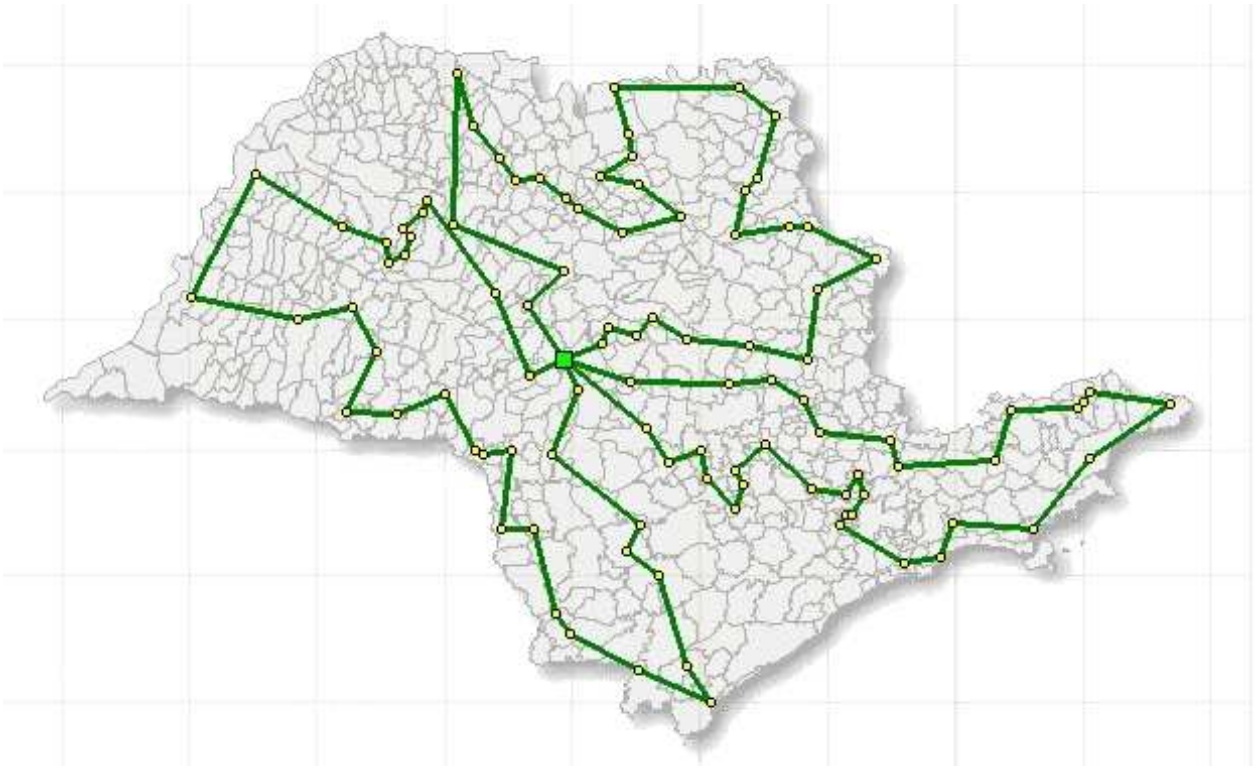
Vários problemas importantes são NP-difíceis

- roteamento de veículos de cadeias de distribuição
- atribuição de frequências 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
- otimização de código
- muitos outros...

É imprescindível saber se nosso problema é NP-difícil!

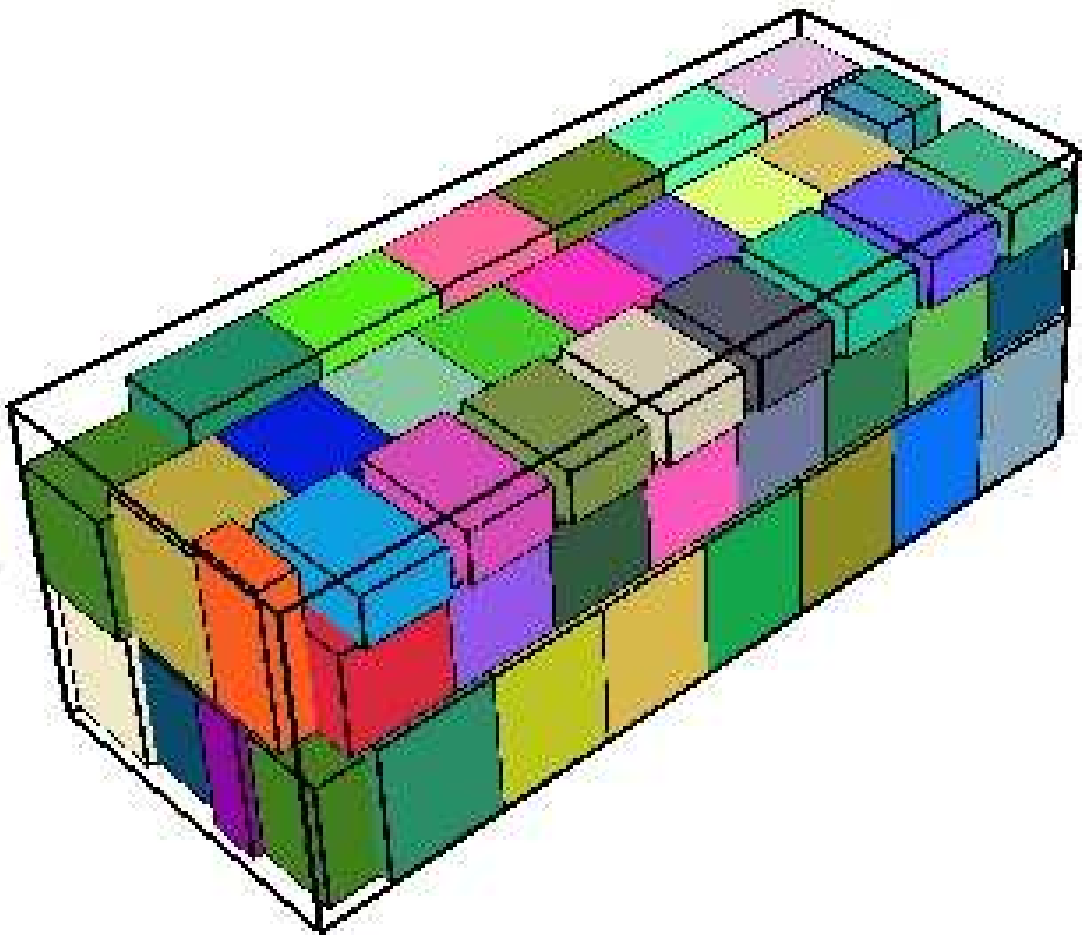
Um problema de roteamento

Calcular as rotas dos caminhões de entrega de uma distribuidora de bebidas em São Paulo, minimizando a distância percorrida.



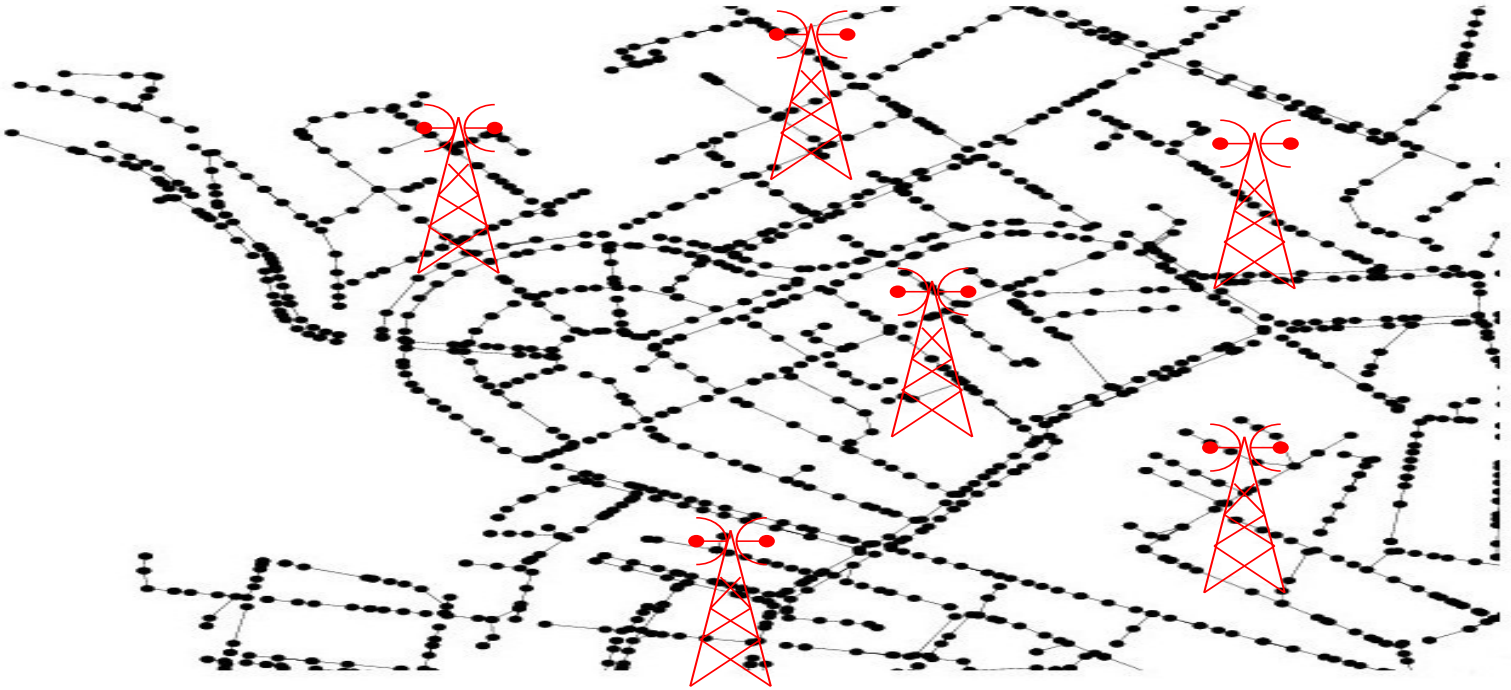
Um problema de empacotamento

Calcular o número mínimo de contêineres para transportar um conjunto de caixas com produtos.



Um problema de projeto de rede

Calcular a localização e o número mínimo de antenas de celulares para garantir a cobertura de uma certa região geográfica.



Algumas sutilezas

Já sabemos

- que o problema do caminho mais curto está em P
- que o problema da 4-coloração está em P

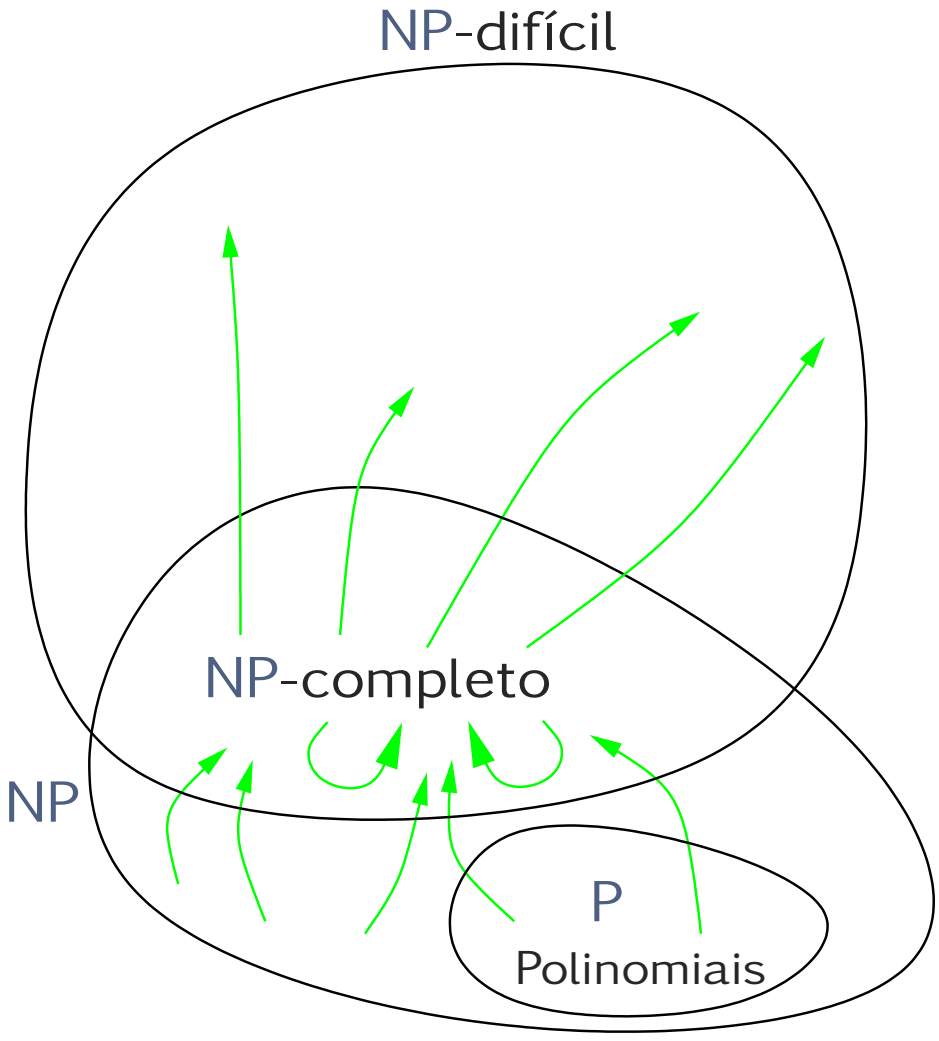
Mas ainda não sabemos

- o problema do caminho mais longo está em P?
- o problema da 3-coloração está em P?

Parecem perguntas bem diferentes

- mas as duas têm respostas idênticas
- responder sim é o mesmo que dizer $\mathrm{P} = \mathrm{NP}$
- mas conjecturamos que a resposta é não!

Possível configuração



No restante do curso, queremos entender essa figura!

Evidências para essa configuração

Por que acreditamos que não há algoritmo rápido para problemas NP-difíceis?

- demonstrou-se que um problema é NP-completo pela primeira vez na década de 1970 (Cook-Levin)
- bem antes, vários desses problemas já eram estudados
- desde então, mostrou-se que inúmeros problemas importantes também são NP-completos ou NP-difíceis
- vários pesquisadores estudaram seus problemas NP-completos preferidos, mas ninguém descobriu qualquer algoritmo polinomial
- basta que um problema NP-difícil tenha algoritmo de tempo polinomial para que todos problemas em NP tenham algoritmos de tempo polinomial

O que fazer se um problema for NP-difícil?

Se sabemos que um problema é NP-difícil, podemos concentrar os recursos em busca de

- algoritmos para instâncias pequenas
- algoritmos que obtêm soluções aproximadas
- algoritmos eficientes exatos, mas para casos particulares
- algoritmos e métodos heurísticos
- etc.

Antes, queremos identificar que problemas são NP-difíceis

Classes de problemas

Classes de problemas

- Linguagens formais

Linguagens formais

Alguns termos:

- um alfabeto Σ é um conjunto finito de símbolos
- uma palavra é uma sequência finita de símbolos de Σ
- o conjunto de todas as palavras é denotado por Σ^*, que inclui a sequência vazia, ε

Uma linguagem formal sobre Σ é um subconjunto $L \subseteq \Sigma^*$.

- a linguagem vazia é $L = \emptyset$
- uma finita é $F = \{banana, uva, pera\}$ sobre $\Sigma = \{a, b, \ldots, z\}$
- uma infinita é $P = \{2, 3, 7, 11, \ldots\}$ sobre $\Sigma = \{0, 1, \ldots, 9\}$

Representando problemas de decisão

Um problema de decisão é uma função $Q : \{0,1\}^* \to \{0,1\}$.

- o conjunto $\{0,1\}$ representa o alfabeto do computador
- uma sequência de bits $x \in \{0,1\}^*$ representa uma entrada
- um bit 0 ou 1 representa a resposta da pergunta

A linguagem correspondente a um problema de decisão Q é o conjunto de instâncias L com resposta sim, isso é,

$$L = \{x \in \{0,1\}^* \ : \ Q(x) = 1\}.$$

- identificamos um problema Q com sua linguagem L

Codificação

Uma codificação é o mapeamento utilizado para representar um objeto como uma palavra de Σ^*.

- dado um objeto A, sua codificação é denotada por $\langle A \rangle$

Problema do caminho mínimo

$$\text{PATH} = \{\langle G, u, v, k \rangle \ : \ G \text{ é um grafo, } u, v \text{ são vértices de } G \text{ e } k \text{ é um inteiro tais que existe caminho de } u \text{ até } v \text{ em } G \text{ de tamanho no máximo } k\}$$

Tamanho da instância

O tamanho de uma instância $x \in \{0,1\}^*$ é o número de bits de x.

- denotamos o tamanho de x por $n = |x|$
- para um objeto de alto nível A, temos $n = |\langle A \rangle|$
- precisamos tomar cuidado com a codificação

Considere a linguagem $\text{PARES} = \{\langle k \rangle : k \text{ é par}\}$

- em unário
  - representamos 100 como $1111111\ldots1111111$
  - nesse caso, o tamanho é $n = |\langle k \rangle| = \Theta(k)$
- em binário
  - representamos 100 como 1100100
  - nesse caso, o tamanho é $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- em ternário, quaternário etc.
  - também, o tamanho é $n = |\langle k \rangle| = \lceil \log_b k \rceil = \Theta(\log k)$

Classes de problemas

- A classe P

Linguagem aceita

Considere um algoritmo A e uma entrada $x \in \{0,1\}^*$

- pode ser que A termina ao receber x e devolve $A(x) = 1$
- pode ser que A termina ao receber x e devolve $A(x) = 0$
- pode ser que A não termina ao receber x

Um algoritmo A aceita uma linguagem L se

$$L = \{x \in \{0,1\}^* \ : \ A(x) = 1\}.$$

Um algoritmo A decide L se para todo $x \in \{0,1\}^*$:

- se $x \in L$, então $A(x) = 1$ (A aceita x)
- se $x \notin L$, então $A(x) = 0$ (A rejeita x)

Classe P

Definição

A classe P é o conjunto de linguagens $L \subseteq \{0,1\}^*$ para as quais existe algoritmo A que decide L em tempo polinomial.

Em outras palavras, se $L \in \mathrm{P}$, então

1. existe algoritmo $A(x)$ que decide L
2. esse algoritmo executa em tempo polinomial em $|x|$

Aceita e decide em tempo polinomial

Teorema

$\mathrm{P} = \{L \subseteq \{0,1\}^* : L \text{ é aceita por um algoritmo polinomial}\}$

Demonstração:

$\subseteq$
- considere $L \in \mathrm{P}$
- por definição de P, existe um algoritmo que decide L
- logo, também existe um algoritmo que aceita L

$\supseteq$
- suponha que L é aceita por um algoritmo polinomial A
- o tempo do algoritmo é n^k, para alguma constante k
- construa um algoritmo A' para uma entrada x:
  1. simule o algoritmo A executando no máximo n^k passos
  2. se A aceitou x, então responda sim
  3. se A rejeitou x ou não terminou, então responda não
- observe que o algoritmo A' decide L em tempo polinomial

Classes de problemas

- A classe NP

Certificado

Considere uma linguagem L

- tome uma instância x do problema correspondente
- queremos encontrar uma sequência de bits $y \in \{0,1\}^*$
- de forma que verificar y permite concluir que $x \in L$
- normalmente y é a codificação de uma solução para x
- chamamos y de certificado para x

Certificado para PATH

- tome uma instância $x = \langle G, u, v, k \rangle$ de PATH
  1. se x é sim, existe caminho P de u a v com até k arestas
  2. se x é não, não existe caminho de u a v com até k arestas
- no primeiro caso, $y = \langle P \rangle$ é um certificado de que $x \in$ PATH

Verificador

Um verificador para uma linguagem L é um algoritmo que recebe uma instância x e um sequência de bits y tal que

- se $x \in L$, ele devolve sim para algum certificado y
- se $x \notin L$, ele devolve não independentemente de y

VERIFICA-PATH$(\langle G, u, v, k\rangle, \langle P\rangle)$

1. se $\langle P\rangle$ não é codificação de um caminho de G, devolva **não**
2. se P tem mais que k arestas, devolva **não**
3. se P não sai de u e chega em v, devolva **não**
4. senão devolva **sim**

- normalmente, omitimos o passo que valida a codificação

Tempo de verificação

Queremos executar o verificador em tempo polinomial:

1. o tempo do verificador deve ser polinomial em $|x|$ e $|y|$
2. o tamanho do certificado $|y|$ deve ser polinomial em $|x|$

Por quê?

- queremos diferenciar as tarefas de decidir e verificar
- para certos problemas, decidir uma instância é difícil
- mas pode ser que verificar uma solução seja fácil
- veremos um exemplo em seguida

Ciclo hamiltoniano

Um ciclo hamiltoniano em um grafo G é um ciclo que passa por todos os vértices.

- se houver esse ciclo, dizemos que G é hamiltoniano

Exemplo

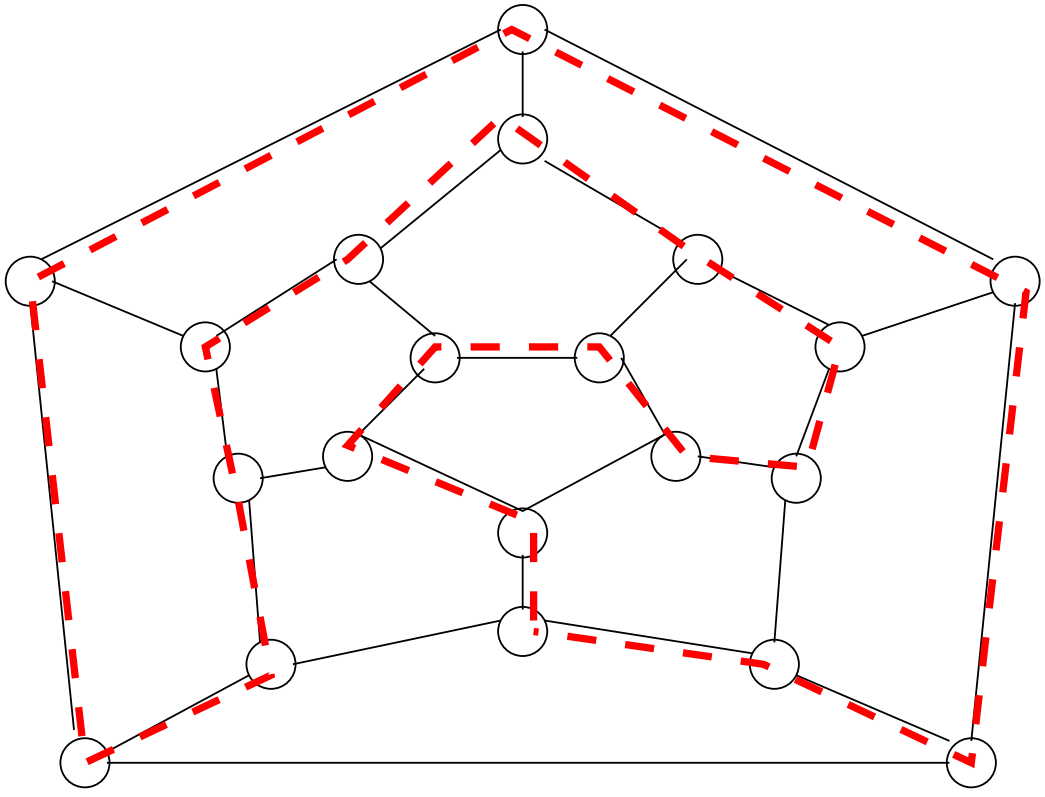
Problema do ciclo hamiltoniano

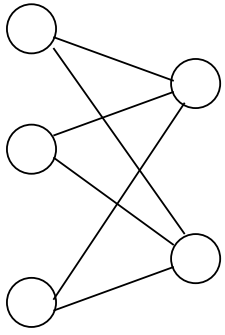
$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ é um grafo hamiltoniano } \}$$

Escolhas para certificado

- um ciclo hamiltoniano de G ou
- uma sequência de vértices de G

Exemplo de ciclo hamiltoniano





O grafo da direita não tem ciclo hamiltoniano!

Chutando um ciclo hamiltoniano

Podemos decidir se há um ciclo hamiltoniano:

- o algoritmo trivial gasta tempo $O(V!)$
- os melhores algoritmos têm tempo $O(n^2 2^n)$
- esses algoritmos são determinísticos

Mas se sortearmos um ciclo C:

- é fácil verificar se ele é hamiltoniano em tempo linear
- basta sortear uma sequência S de $|V|$ vértices
- o sorteio do ciclo é um processo não determinístico

Verificando um ciclo

VERIFICA-HAM-CICLO($\langle G \rangle, \langle S \rangle$)

1. se S não contém todos os vértices de G, devolva **não**
2. faça $n \leftarrow |S|$ e $S[n+1] \leftarrow S[1]$
3. para cada $i = 1, 2, \ldots, |S|$:
   - seja $u \leftarrow S[i]$ e $v \leftarrow S[i+1]$
   - se (u, v) não é aresta de G, devolva **não**
4. devolva **sim**

Linguagem verificada

Um algoritmo V verifica uma linguagem L se

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

Em outras palavras,

1. se $x \in L$, então existe certificado y tal que $V(x,y) = 1$
2. se $x \notin L$, então não existe certificado y tal que $V(x,y) = 1$

A classe NP

Definição

A classe NP é o conjunto de linguagens $L \subseteq \{0,1\}^*$ para as quais existe algoritmo V que verifica L em tempo polinomial.

Em outras palavras, se $L \in \text{NP}$, então

1. existe algoritmo $V(x, y)$ que verifica L
2. esse algoritmo executa em tempo polinomial em $|x|$ e $|y|$
3. para cada $x \in L$, existe certificado y polinomial em $|x|$

Determinando se problema é NP

Dado problema L, devemos seguir esses passos:

1. identifique um certificado de tamanho polinomial para L
2. construa um algoritmo verificador $V(x, y)$ polinomial
3. demonstre que:
   - se $x \in L$, então **existe** y tal que $V(x, y) = 1$
   - xe $x \notin L$, então para **qualquer** y, vale $V(x, y) = 0$

Exemplos:

- VERIFICA-PATH é um verificador para PATH
- VERIFICA-HAM-CICLO é um verificador para HAM-CYCLE

Portanto, PATH e HAM-CYCLE estão em NP.

NP contém P

Seja $L \in \mathrm{P}$

- exite algoritmo A que decide L
- vamos construir um verificador para L

VERIFICAL(x, y):
1. devolva $A(x)$

- o verificador só precisa ignorar o argumento y
- podemos escolher qualquer certificado (e.g. $y = \varepsilon$)
- analisamos:
  - se $x \in L$, então $A(x) = 1$, daí VERIFICAL$(x, \varepsilon) = 1$
  - se $x \notin L$, então $A(x) = 0$, daí VERIFICAL$(x, y) = 0$ para todo y
- assim $L \in \mathrm{NP}$ e concluímos que $\mathrm{P} \subseteq \mathrm{NP}$

Classes de problemas

- A classe co-NP

O problema complementar

Dada uma linguagem L, o seu complementar é $\overline{L} = \{0,1\}^* \setminus L$.

- $\overline{L}$ é o conjunto de instâncias de L com resposta **não**
- exemplo:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ possui ciclo hamiltoniano}\}$$

$$\overline{\text{HAM-CYCLE}} = \{\langle G \rangle : G \text{ não possui ciclo hamiltoniano}\}$$

Definição

A classe complementar de NP é o conjunto de linguagens

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

Exemplos:

- $\overline{\text{HAM-CYCLE}} \in \text{co-NP}$
- $\text{P} \subseteq \text{co-NP}$

Outro exemplo: tautologia

O conjunto co-NP é contém as linguagens que possuem um certificado curto para a resposta **não**.

Tautologia

Dada uma fórmula booleana com um conjunto de n variáveis e operadores $\wedge$, $\vee$, $\neg$ etc, ela é verdadeira para toda atribuição de variáveis?

- $p \vee \neg p$, $((z \wedge y) \vee \neg x \vee (x \wedge \neg y)) \vee (\neg z \wedge y)$ são tautologias
- p, $(\neg y \vee x) \wedge (y \vee \neg x)$ não são tautologias

- certificado curto para instâncias **não**: $x = 0, y = 1$
- não conhecemos certificado curto para instâncias **sim**

NP-completude

Resumo das classes até agora

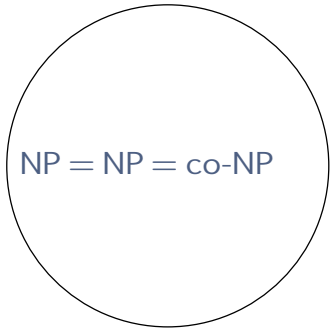
$$\mathsf{P} = \{L \subseteq \Sigma^* : \text{há algoritmo que decide } L \text{ em tempo polinomial}\}$$

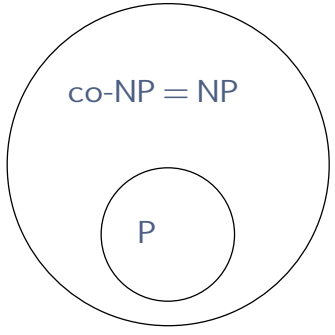
$$\mathsf{NP} = \{L \subseteq \Sigma^* : \text{há algoritmo que verifica } L \text{ em tempo polinomial}\}$$

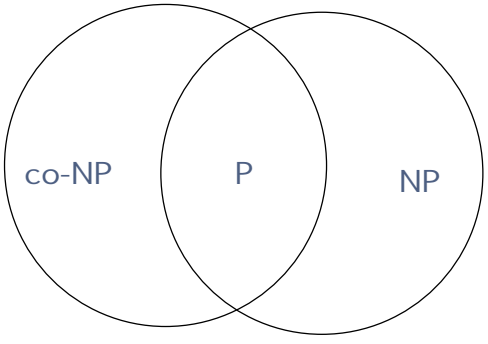
$$\text{co-NP} = \{L \subseteq \Sigma^* : \text{há algoritmo que verifica } \overline{L} \text{ em tempo polinomial}\}$$

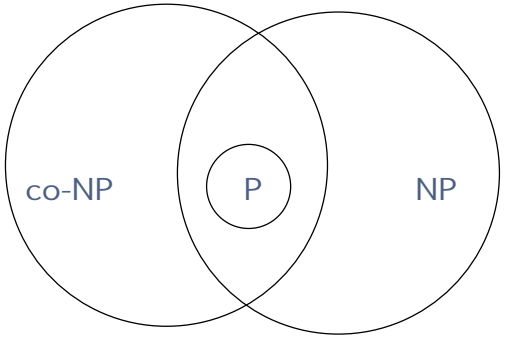
Classes de complexidade

Possíveis configurações dessas classes:









NP-completude

- Classe NP-completo

Refletindo sobre o que vimos

Vimos alguns exemplos de problemas que

- podemos decidir em tempo polinomial: PATH
- só sabemos verificar em tempo polinomial: HAM-CYCLE

Por que não conhecemos algoritmo rápido para HAM-CYCLE?

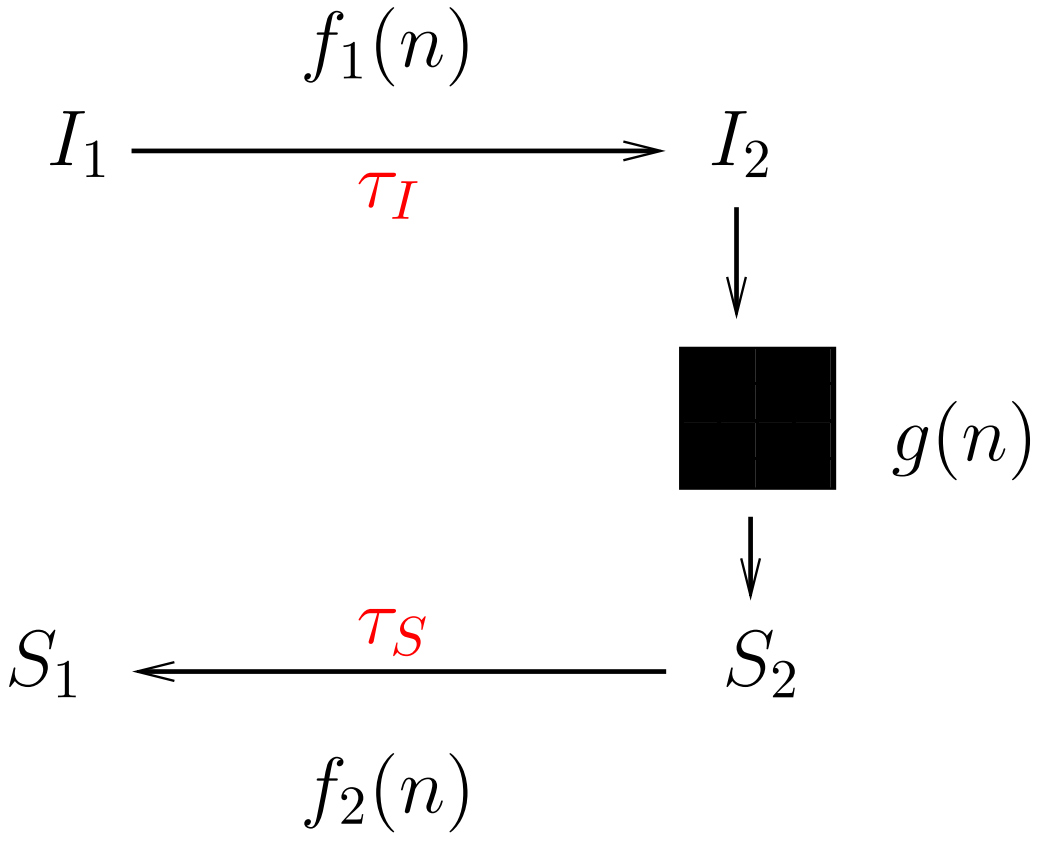
- será que HAM-CYCLE é mais difícil do que PATH?
- ou que HAM-CYCLE é mais difícil que qualquer um em P?

Vamos mostrar que HAM-CYCLE é NP-difícil

- não sabemos se é mais difícil do que algum problema P
- mas é pelo menos tão difícil quanto qualquer problema NP

Como comparar problemas?

A ferramenta adequada para comparar problemas são as reduções



Reduções de Karp

Estamos interessados em reduções em que:

1. a transformação de entrada τ_I leva tempo polinomial
2. não há transformação de saída τ_S

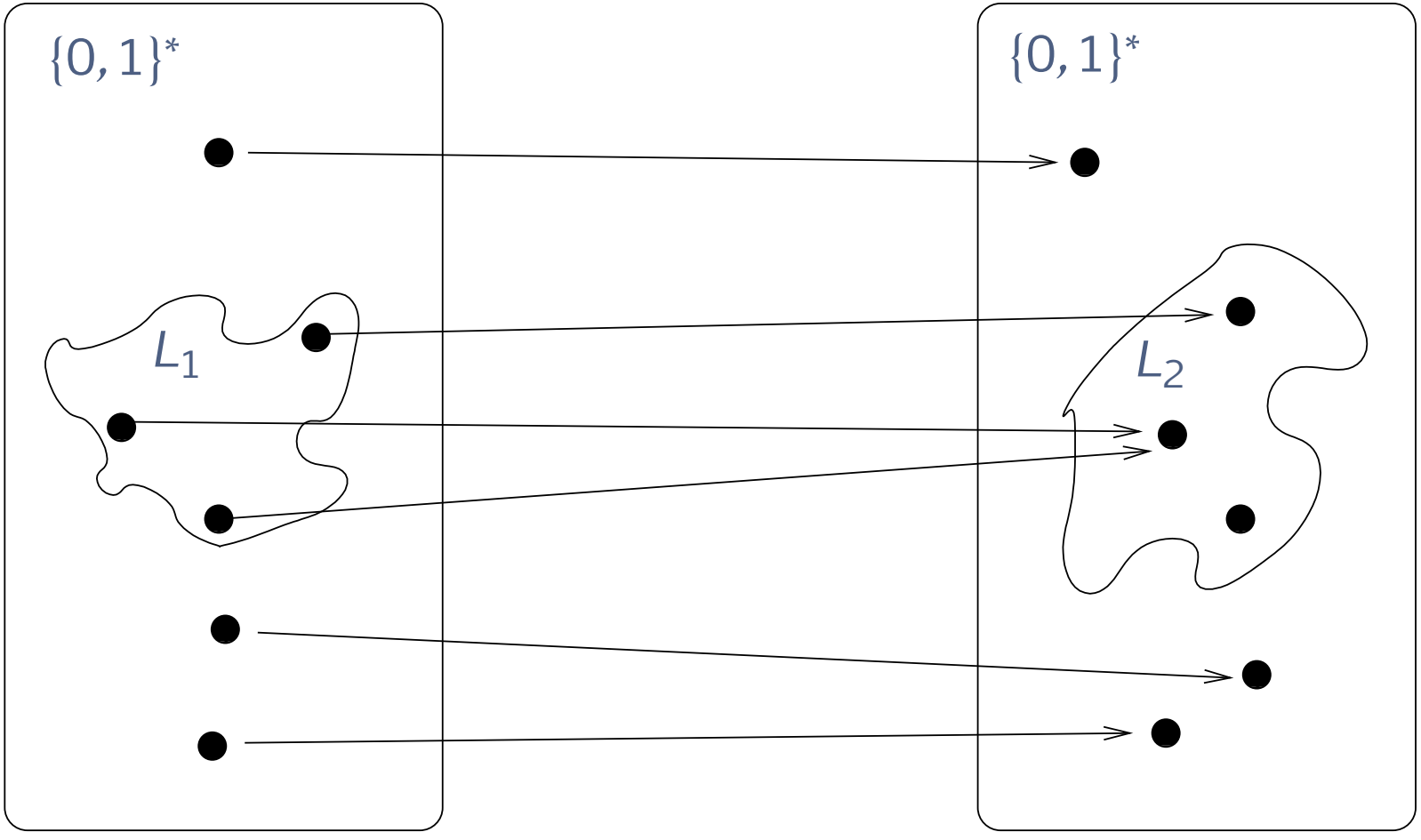
Redução de Karp

A linguagem L_1 é redutível em tempo polinomial para L_2 se

- existe algoritmo $f : \{0,1\}^* \to \{0,1\}^*$,
- $f(x)$ leva tempo polinomial em $|x|$,
- $x \in L_1$ se e somente se $f(x) \in L_2$.

- escrevemos $L_1 \leqslant_p L_2$
- dizemos que f reduz L_1 para L_2

Ilustração



Reduções de tempo polinomial

Teorema

Considere linguagens $L_1, L_2 \subseteq \{0,1\}^*$ tais que $L_1 \leqslant_p L_2$.
Se $L_2 \in \mathrm{P}$, então $L_1 \in \mathrm{P}$.

Demonstração:

- suponha que $L_2 \in \mathrm{P}$
- seja A_2 um algoritmo que decide L_2 em tempo polinomial
- seja f um algoritmo polinomial que reduz L_1 a L_2
- crie um algoritmo A_1 fazendo $A_1(x) = A_2(f(x))$
- já que f é uma redução, obtemos:
  - se $x \in L_1$, então $f(x) \in L_2$ e $A_2(f(x)) = 1$, daí $A_1(x) = 1$
  - se $x \notin L_1$, então $f(x) \notin L_2$ e $A_2(f(x)) = 0$, daí $A_1(x) = 0$
- assim, A_1 decide L_1 em tempo polinomial

Classe NP-completo

Definição

A classe NP-completo é o conjunto de linguagens $L \subseteq \{0,1\}^*$ tais que:

1. $L \in \text{NP}$
2. $L' \leqslant_p L$ para todo $L' \in \text{NP}$

- se apenas 2 for satisfeita, dizemos que L é NP-difícil

Condição para NP = NP

Teorema

Se existe algoritmo que decide $L \in$ NP-completo em tempo polinomial, então P = NP.

Demonstração:

- suponha que existe $L \in \text{P} \cap \text{NP-completo}$
- como $L \in$ NP-completo, para toda $L' \in \text{NP}$, temos $L' \leqslant_p L$
- mas como $L \in \text{P}$, isso implica $L' \in \text{P}$ pelo teorema anterior
- então $\text{NP} \subseteq \text{P}$ e, portanto, $\text{NP} = \text{P}$

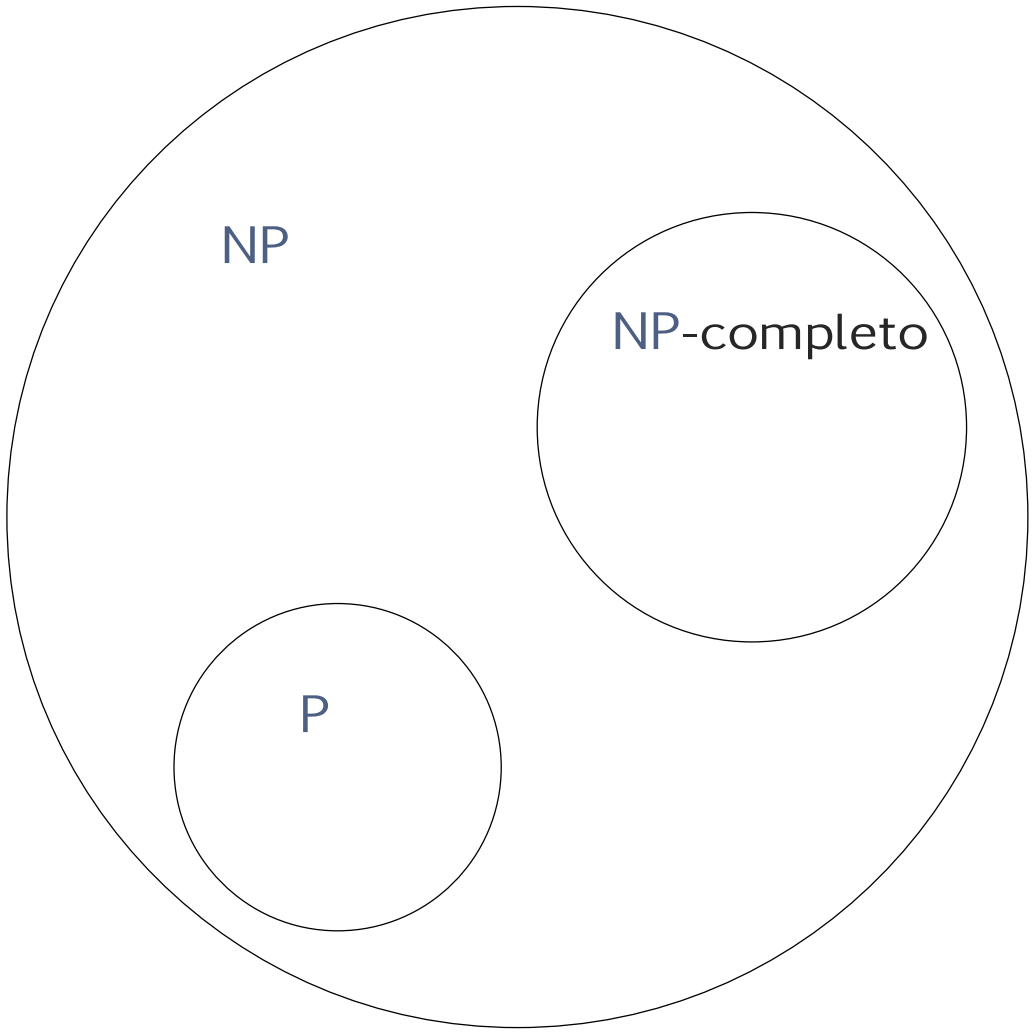
Teorema

Se existe problema $L \in \text{NP}$ tal que $L \notin \text{P}$, então para todo $\text{NP-completo} \cap \text{P} = \emptyset$.

- exercício

Possível configuração de NP-completo

Como acreditamos que é a relação das classes:



NP-completude

- Um primeiro problema NP-completo

A classe NP-completo não é vazia

Mas será que a classe NP-completo é vazia?

- Cook e Levin responderam que não (independentemente)
- eles mostraram que SAT $\in$ NP-completo

Teorema de Cook–Levin

SAT é NP-completo.

- SAT é o problema da satisfatibilidade
- não vamos demonstrar esse teorema
- mas veremos um rascunho para um problema parecido

Satisfatibilidade

Considere uma fórmula booleana

- contém um conjuntos de variáveis booleanas
- é escrita usando os seguintes operadores
  1. negação ($\neg$)
  2. conjunção ($\wedge$)
  3. disjunção ($\vee$)
  4. implicação ($\rightarrow$)
  5. equivalência ($\leftrightarrow$)

Problema da satisfatibilidade (SAT)

- **Entrada:** uma fórmula booleana
- **Pergunta:** existe atribuição de variáveis booleana para a qual a avaliação da fórmula é verdadeira?

Tabela de operadores

Tabela verdade dos operadores

a	b	$\neg a$	$a \wedge b$	$a \vee b$	$a \rightarrow b$	$a \leftrightarrow b$
F	F	V	F	F	V	V
F	V	V	F	V	V	F
V	F	F	F	V	F	F
V	V	F	V	V	V	V

Fórmula satisfazível

Uma formula é satisfazível se houver atribuição das variáveis para a qual a avaliação é verdadeira.

Exemplo:

- fórmula: $f = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
- atribuição: $x_1 = 0,\ x_2 = 0,\ x_3 = 1,\ x_4 = 1$
- avaliação:

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

Linguagem correspondente

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

Satisfatibilidade

$$\text{SAT} = \{\langle f \rangle : f \text{ é uma fórmula booleana satisfazível}\}$$

É fácil verificar

Lema

SAT é NP.

Escrevemos um algoritmo verificador:

VERIFICA-SAT$(\langle f \rangle, \langle y \rangle)$
1. se $f(y) = 1$, devolva **sim**
2. senão, devolva **não**

Observe que o algoritmo verifica SAT em tempo polinomial:

1.
   - se $\langle f \rangle \in$ SAT, então f é satisfazível
   - daí, existe atribuição y das variáveis que faz f verdadeira
   - e VERIFICA-SAT$(\langle f \rangle, \langle y \rangle)$ devolve **sim**
2.
   - suponha VERIFICA-SAT$(\langle f \rangle, \langle y \rangle)$ devolve **sim**
   - então y é uma atribuição verdadeira para a fórmula
   - daí f é satisfazível e $\langle f \rangle \in$ SAT

NP-completude

- Um esquema do teorema de Cook-Levin

Circuito lógico

Considere um circuito lógico

- contém n fios de entrada
- é formado combinando portas lógicas
  - NOT ($\neg$)
  - AND ($\wedge$)
  - OR ($\vee$)
- a saída do circuito é dada por um fio

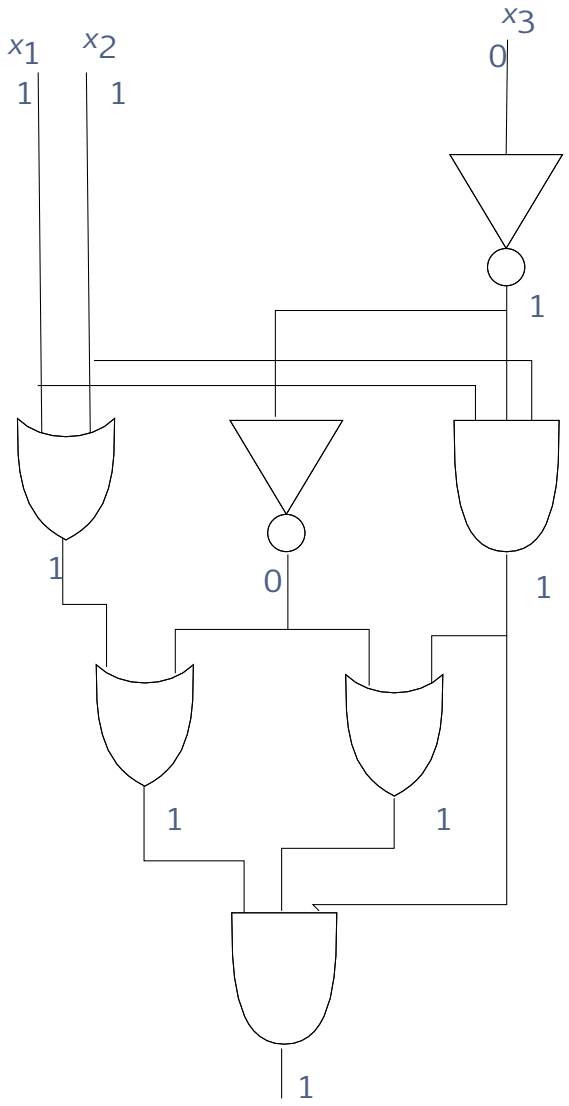
Satisfatibilidade de circuito

- Uma atribuição de um circuito é uma atribuição de um bit 0 ou 1 para cada fio de entrada.
- Um circuito é satisfazível se ele possuir uma atribuição que resulta em bit 1 no fio de saída.

Problema da satisfatibilidade de circuito (C-SAT)

- **Entrada:** um circuito lógico
- **Pergunta:** o circuito é satisfazível?

Exemplo de circuito



Linguagem correspondente

A linguagem C-SAT contém circuitos lógicos satisfatíveis.

Satisfatibilidade de circuito

$$\text{C-SAT} = \{\langle C \rangle : C \text{ é um circuito lógico satisfazível}\}$$

Há um algoritmo simples de tempo $O(2^n(n+m))$

- n é o número de fios de entrada
- m é o número de ligações entre portas

Lema

C-SAT é NP.

- análogo à demonstração de que SAT $\in$ NP (exercício)

Redução de NP para circuitos

Queremos mostrar que C-SAT é NP-completo.

- falta mostrar então que C-SAT é NP-difícil
- queremos que para todo $Q \in \text{NP}$, tenhamos $Q \leqslant_p \text{C-SAT}$

Nosso objetivo é projetar uma redução polinomial F:

- **Entrada:** instância $x \in \{0,1\}^*$ de Q
- **Saída:** circuito $F(x)$ tal que

$$x \in Q \quad \text{se e somente se} \quad F(x) \in \text{C-SAT}$$

NP-dificuldade

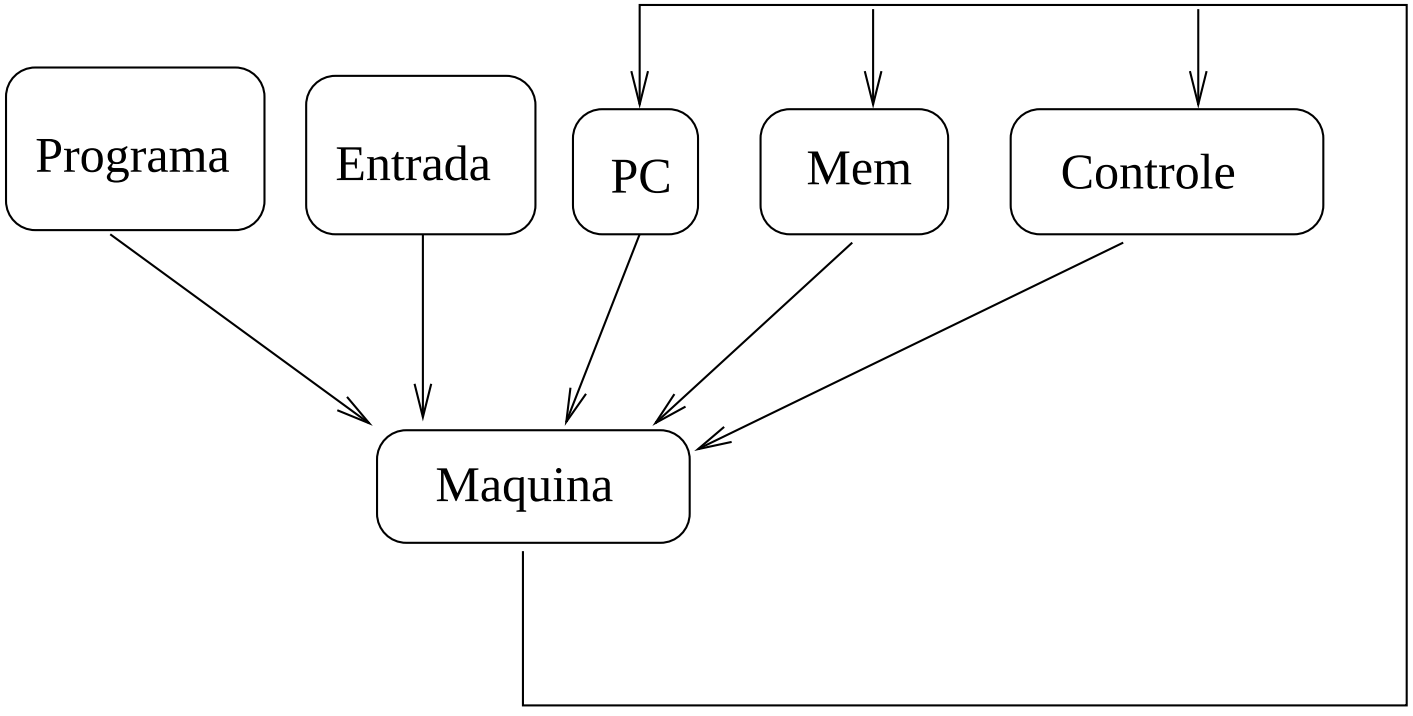
Lema

C-SAT é NP-difícil.

- considere um problema arbitrário $Q \in \text{NP}$
- existe um algoritmo verificador polinomial A para Q
- dada instância x de Q, criaremos instância $F(x)$ de C-SAT
- o algoritmo verificador de Q recebe duas strings, x e y:
  - a entrada x tem tamanho $|x| = n$
  - o certificado y tem tamanho $|y| = n^{k'}$, para k' constante
- relembre que A leva tempo polinomial em $|x|$ e $|y|$
- assim, ele executa até n^k passos, para k constante
- a ideia é montar um circuito que simula n^k passos de A

Continuação da prova

O algoritmo A pode ser implementado por um computador de circuitos lógicos.

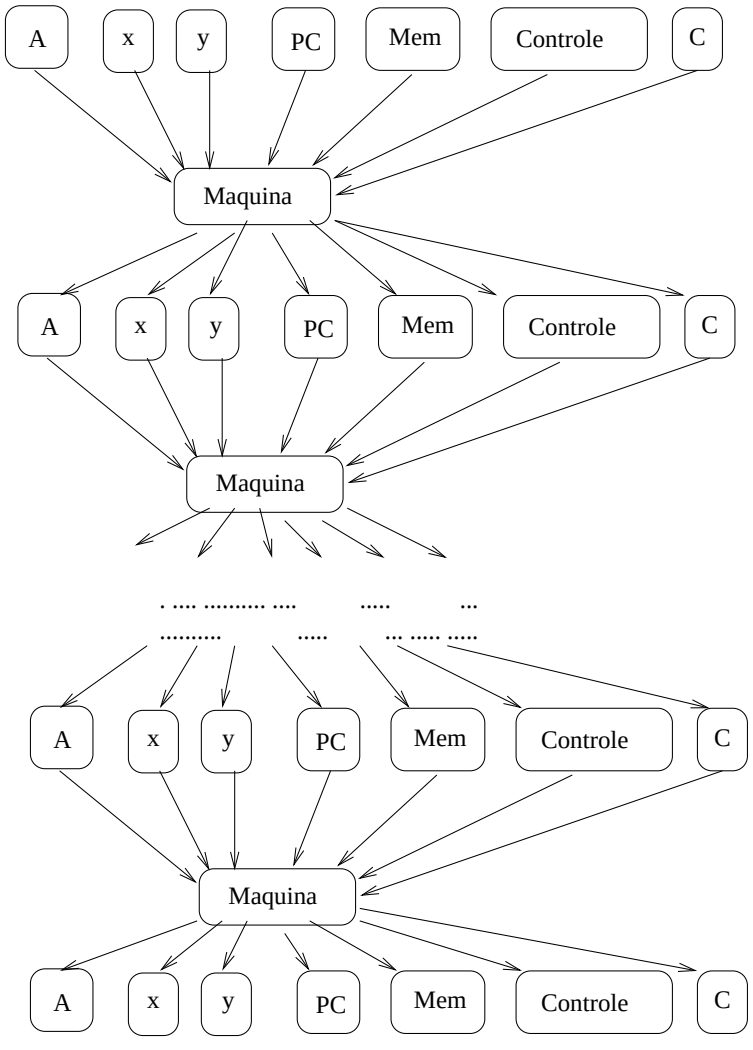


- mas os circuitos têm retroalimentação
- após executar uma instrução, a memória é modificada

Continuação da prova

- ajustamos o algoritmo A para que a saída seja escrita em um bit específico da memória denotado por C
- cada instrução executada pelo algoritmo A
  - começa em um estado de memória, PC e controle
  - e modifica esse estado de memória, PC e controle
- criamos n^k cópias da máquina, uma para cada instrução:
  1. o estado de entrada da primeira máquina corresponde ao estado inicial da execução do algoritmo
  2. a saída de uma instrução correspondente a uma cópia modifica o estado de entrada da cópia seguinte
  3. a saída do circuito é a saída de uma conjunção (porta $\vee$) ligando os bits correspondentes a C

Continuação da prova



Continuação da prova

O circuito construído tem tamanho polinomial.

- o tamanho da máquina, controle, PC e A independem de x
- a memória, y e x têm tamanho polinomial em $|x|$
- fazemos apenas n^k cópias da máquina

Fios de entrada e de saída

- todo circuito é fixo e pode ser construído a partir de x
- há um fio de entrada para cada bit do certificado y
- o fio de saída vale 1 se algum campo C foi mudado para 1

Como a redução levou tempo polinomial, falta mostrar apenas que $x \in Q$ se e somente se $F(x) \in$ C-SAT.

Continuação da prova

1. Suponha que $x \in Q$
   - então há certificado y tal que $A(x, y) = 1$
   - forneça y como entrada para o circuito $F(x)$
   - então alguma cópia da máquina muda C para 1
   - a saída do circuito será 1
2. Suponha que $x \notin Q$
   - então $A(x, y) = 0$ para todo y
   - forneça entrada y arbitrária para o circuito $F(x)$
   - assim o campo C de cada cópia nunca muda para 1
   - portanto a saída do circuito será 0

Teorema

C-SAT é NP-completo.

Demonstrando NP-completude

Que outros problemas são NP-difíceis?

Já sabemos que C-SAT é NP-completo

- como descobrir se outro problema Q é NP-difícil?
- temos duas possibilidades:
  1. mostrar que $L \leqslant_p Q$ para todo problema $L \in \text{NP}$
  2. mostrar que $L \leqslant_p Q$ para algum problema $L \in \text{NP-difícil}$

Normalmente usamos a segunda opção

- basta reduzir um problema NP-difícil para o problema Q
- ou seja, mostramos que Q é tão difícil quanto um NP-difícil
- esse problema NP-difícil pode ser C-SAT, por exemplo

NP-completude via redução

Teorema

Considere uma linguagem Q e seja $L \in$ NP-difícil.
Se $L \leqslant_p Q$, então L é NP-difícil.

Demonstração:

- como L é NP-difícil, para todo $L' \in$ NP temos $L' \leqslant_p L$
- assim $L' \leqslant_p L$ e $L \leqslant_p Q$, o que implica $L' \leqslant_p Q$
- portanto L é NP-difícil

Se além disso $Q \in$ NP, então Q é NP-completo.

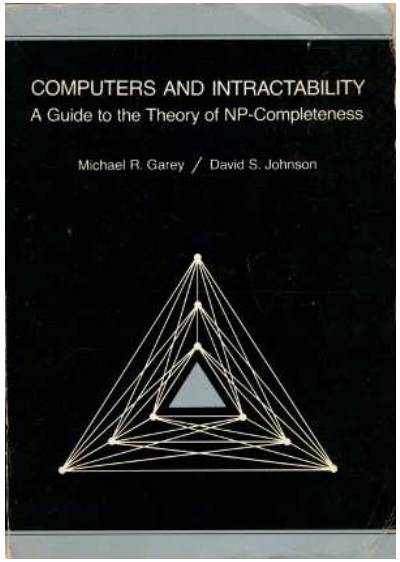
Literatura

Karp mostrou em 1972 que 21 problemas são NP-completos

- listou várias reduções de problemas NP-completos
- as reduções induzem uma árvore com raiz em SAT
- veja a Wikipédia!

Com sorte, seu problema foi estudado por Garey e Johnson

- livro publicado em 1979
- estuda e classifica dezenas de problemas
- entre os mais citados em Ciência da Computação



Demonstrando NP-completude

- Satisfatibilidade

Satisfatibilidade

Relembrando: considere uma fórmula booleana

- contém um conjuntos de variáveis booleanas
- é escrita usando os seguintes operadores
  1. negação ($\neg$)
  2. conjunção ($\wedge$)
  3. disjunção ($\vee$)
  4. implicação ($\rightarrow$)
  5. equivalência ($\leftrightarrow$)

Satisfatibilidade

$$\text{SAT} = \{\langle f \rangle : f \text{ é uma fórmula booleana satisfazível}\}$$

Satisfatibilidade é NP-difícil

Lema

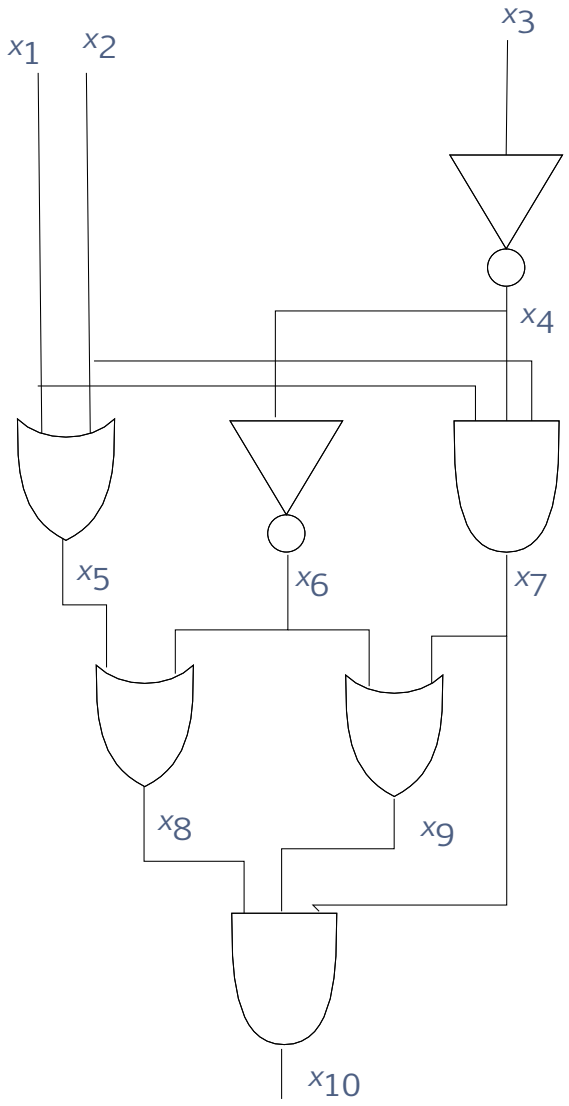
SAT é NP-difícil.

- vamos reduzir C-SAT para SAT
- dado circuito c, vamos criar uma fórmula f tal que:

$$\langle c \rangle \in \text{C-SAT} \text{ se e somente se } \langle f \rangle \in \text{SAT}$$

- iremos criar uma variável x_i para cada fio do circuito

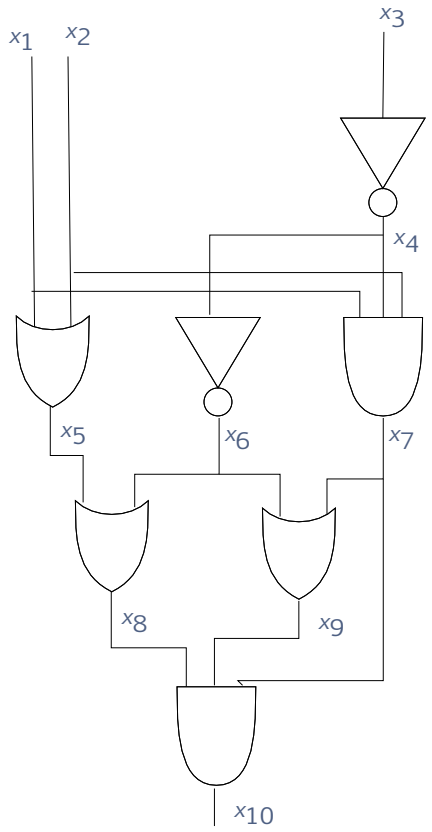
Exemplo de circuito



Criando uma fórmula

Para cada fio do circuito

- crie equivalência entre a fórmula da porta e a variável
- faça a conjunção das equivalências e da variável de saída



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

Demonstração do lema

Primeiro, criamos a instância reduzida

- dado um circuito c, crie a fórmula f correspondente
- observe que isso leva tempo polinomial no tamanho de c

Depois, mostramos que instâncias **sim** são equivalentes

1. Suponha que c é satisfazível
   - valoramos as variáveis x_i de acordo com os fios do circuito
   - como cada porta é consistente, as equivalências valem 1
   - como o fio de saída tem valor 1, a formúla f também vale 1
   - portanto, f é satisfazível
2. Suponha que f é satisfazível
   - fornece os valores de x_i para os fios de entrada
   - como f é uma conjunção, todas equivalências valem 1
   - assim, cada porta lógica é consistente com a equivalência
   - então, os fios de ligação são consistentes com as variáveis
   - em particular, o fio de saída deve valer 1
   - portanto, c é satisfazível

Satisfatibilidade é NP-completo

Teorema

SAT é NP-completo.

Demonstração:

- já sabemos que $\text{SAT} \in \text{NP}$
- e o lema anterior diz que $\text{SAT} \in \text{NP-difícil}$

Demonstrando NP-completude

- Satisfatibilidade 3CNF

Forma normal conjuntiva

Uma fórmula está na forma normal conjuntiva (CNF) se:

1. ela é uma conjunção ($\wedge$) de cláusulas
2. cada cláusula é uma disjunção ($\vee$) de literais
3. um literal é uma variável ou sua negação ($\neg$)

Se cada cláusula tiver exatamente três literais, ela é dita 3CNF.

Exemplos:

- a fórmula $(x_1 \vee x_3) \wedge (\neg x_1) \vee (x_2 \wedge \neg x_3)$ não é CNF
- a fórmula $(x_1) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$ é CNF
- a fórmula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4)$ é 3CNF

Satisfatibilidade 3CNF

Satisfatibilidade 3CNF

$$\text{3CNF-SAT} = \{\langle f \rangle : f \text{ é uma fórmula 3CNF satisfazível}\}$$

- observe que 3CNF-SAT é um subconjunto de SAT
- dizemos que 3CNF-SAT é um caso particular

Lema

3CNF-SAT é NP.

- verificamos se a entrada é uma fórmula 3CNF
- depois utilizamos um verificador para SAT

Satisfatibilidade 3CNF é NP-difícil

Lema

3CNF-SAT é NP-difícil.

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

- considere uma fórmula booleana arbitrária f_1
- vamos criar em tempo polinomial uma fórmula 3CNF f_2
- queremos que

$$\langle f_1 \rangle \in \text{SAT se e somente se } \langle f_2 \rangle \in \text{3CNF-SAT}$$

Faremos isso em três etapas

1. mudar f_1 para conjunção de cláusulas com até 3 variáveis
2. transformar cada cláusula em disjunção de literais
3. garantir que cada cláusula tenha exatamente 3 literais

Construindo cláusulas

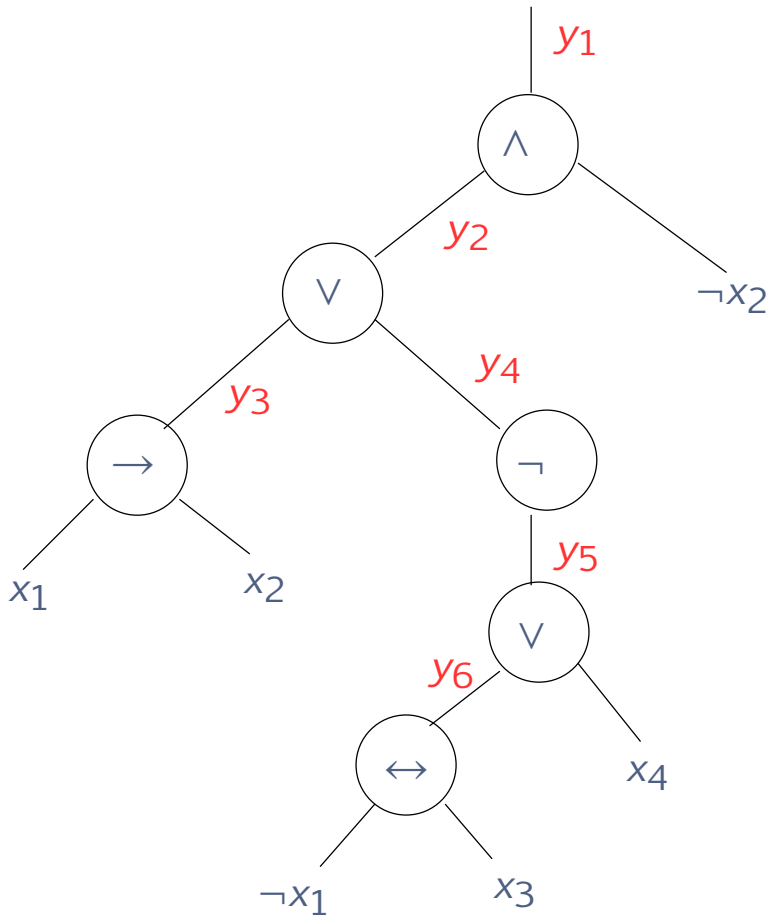
Como separar a fórmula f_1 em cláusulas?

- note que cada operação tem no máximo dois operandos
- vamos construir uma árvore binária de avaliação:
  1. cada operador da fórmula corresponde a um nó da árvore
  2. os filhos de um nó correspondem aos operando
  3. cada nó é associado a uma nova variável booleana

Árvore de avaliação

Exemplo:

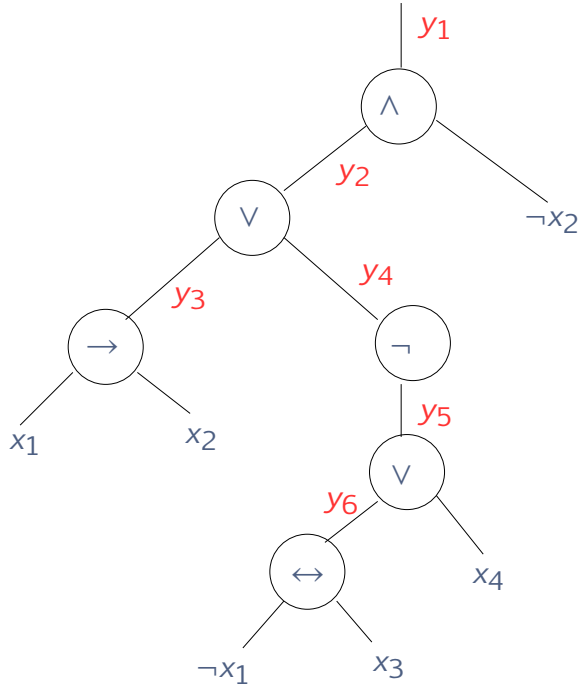
$$f_1 = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$



Construindo cláusulas (cont)

Podemos interpretar a árvore como um circuito

- análogo à redução de C-SAT para SAT
- cada "fio" corresponde a uma variável booleana
- a variável de um nó corresponde a avaliação do nó
  - primeiro avaliamos os nós filhos
  - depois aplicamos a operação
- o saída corresponde à variável do nó raiz



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

Construindo cláusulas (cont)

A transformação é polinomial

- cada operador é binário ou unário
- assim, cada cláusula terá 2 ou 3 variáveis
- o número de operadores é linear no tamanho de f
- assim f' tem tamanho polinomial em $|\langle f_1 \rangle|$
- e também pode ser criada em tempo polinomial

A nova fórmula é equivalente

- note que f' é satisfazível sse f é satisfazível
- o argumento é análogo ao da redução de C-SAT

Transformando em CNF

Como transformar f' em CNF?

- a fórmula f' já é uma conjunção
- falta transformar cada cláusula em uma disjunção

Transformando uma cláusula em disjunção

1. tome cada cláusula c de f'
2. construa a tabela verdade de c
3. construa uma disjunção correspondente aos valores 0
4. negue e aplique a regra de De Morgan

Exemplo de transformação de cláusula

1. considere uma cláusula $(y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
2. construa a tabela verdade

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

3. construa uma disjunção correspondente aos valores 0:

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

4. negue e aplique a regra de De Morgan:

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

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

Transformando em CNF (cont)

A transformação é polinomial

- repetimos esse processo para cada cláusula de f'
- isso resulta em uma nova fórmula f''
- relembre que cláusula tem no máximo 3 variáveis
- daí a tabela verdade tem no máximo 8 linhas
- portanto f'' tem tamanho polinomial no tamanho de f'

A nova fórmula é equivalente

- cada cláusula é trocada por uma subfórmula equivalente
- então a fórmula completa f'' é equivalente à formula f'

Transformando em 3CNF

Como transformar f'' em 3CNF?

- a fórmula f'' já é CNF
- cada cláusula tem um, dois ou três literais
- falta completar cláusulas pequenas

Analisamos cada cláusula

1. trocamos cláusula com dois literais $(l_1 \vee l_2)$ por

$$(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$$

2. trocamos cláusulas com um literal (l_1) por

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

Os valores das novas fórmulas independem de p e q.

Transformando em 3CNF (cont)

A transformação é polinomial

- após a transformação, obtemos uma fórmula 3CNF f_2
- adicionamos no máximo mais duas novas variáveis
- e para cada cláusula, adicionamos no máximo outras três
- portanto f_2 tem tamanho polinomial no tamanho de f''

A nova fórmula é equivalente

- cada cláusula é trocada por uma subfórmula equivalente
- então a fórmula completa f_2 é equivalente à formula f''

Concluindo

Redução

- de uma fórmula f_1 obtivemos uma fórmula f_2
- as transformações são polinomiais no tamanho de f_1
- f_1 é satisfazível se e somente se f_2 é satisfazível
- concluímos que SAT $\leqslant_p$ 3CNF-SAT

Portanto, 3CNF-SAT é NP-difícil.

Demonstrando NP-completude

- Clique

Clique

Uma clique de um grafo não direcionado $G = (V, E)$ é um conjunto de vértices $V' \subseteq V$ de forma que quaisquer dois vértices nesse conjunto estejam conectados.

Problema da clique máxima

- é a tarefa de achar uma clique de maior cardinalidade
- a versão de decisão é saber se há clique de tamanho k

Clique máxima

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ é um grafo com clique de tamanho } k\}$$

Clique é NP

Lema

CLIQUE é NP.

Demonstração

- exercício

Clique é NP-difícil

Lema

CLIQUE é NP-difícil.

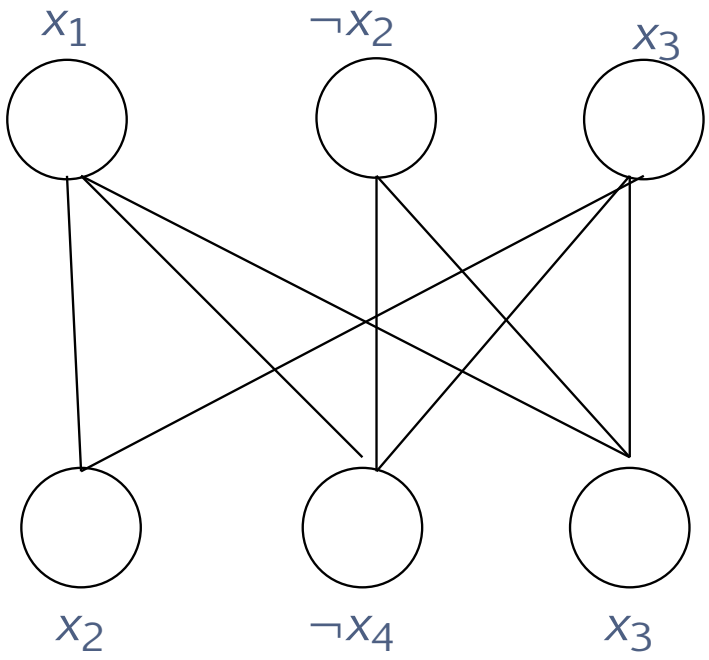
- reduziremos 3CNF-SAT para CLIQUE
- dada fórmula f com k cláusulas, criamos um grafo G
- G terá clique de tamanho k sse f for satisfazível

Redução

- seja $f = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ uma fórmula 3CNF
- cada cláusula C_i de f possui três literais l_1^i, l_2^i e l_3^i
- para cada cláusula C_i, construímos 3 vértices v_1^i, v_2^i e v_3^i
- consideramos outra cláusula C_j com vértices v_1^j, v_2^j e v_3^j
- haverá aresta entre v_x^i e v_y^j se os literais são consistentes
- ou seja, se o literal l_x^i não é negação do literal l_x^j

Exemplo de redução

$x_1 \vee \neg x_2 \vee x_3$



$x_2 \vee \neg x_4 \vee x_3$

Demonstração

A redução leva tempo polinomial

- para cada cláusula, construímos três vértices
- o número de arestas o polinomial no número de vértices
- adicionar cada aresta leva tempo constante

A instância reduzida é equivalente

1. Suponha que f é satisfazível
   - assim cada cláusula tem valor 1
   - para cada cláusula, escolha um literal com valor 1
   - cada dois vértices escolhidos têm literais consistentes
   - do contrário, algum deles teria valor 0
   - então, os vértices escolhidos são uma clique de tamanho k

Demonstração (cont)

2. Suponha que o grafo tem uma clique de tamanho k
   - cada vértice na clique vem de uma cláusula distinta
   - cada par de vértices têm literais consistentes
   - então cada variável correspondente a um vértice:
     - 2.1 ou só aparece como literal não negado
     - 2.2 ou só aparece como literal negado
   - isso induz uma atribuição de variáveis
     - 2.1 $x_i = 1$ se x_i não aparece negado
     - 2.2 $x_i = 0$ se x_i aparece negado
   - essa atribuição torna cada cláusula verdadeira
   - portanto, f é satisfazível

Demonstrando NP-completude

- Cobertura por vértices

Cobertura por vértices

Uma cobertura por vértice (*vertex cover*) de 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'.

Problema da cobertura por vértices mínima

- é a tarefa de achar uma cobertura de menor cardinalidade
- a versão de decisão é saber se há cobertura de tamanho k

Cobertura por vértices máxima

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

Cobertura por vértices é NP

Lema

VERTEX-COVER é NP.

Demonstração

- exercício

Cobertura por vértices é NP-difícil

Lema

VERTEX-COVER é NP-difícil.

- reduziremos CLIQUE para VERTEX-COVER
- dado grafo $G = (V, E)$, criamos o complemento $\overline{G} = (V, \overline{E})$
- vamos mostrar que G terá uma clique de tamanho k sse $\overline{G}$ tiver uma cobertura por vértices de tamanho $|V| - k$.

Demonstração

1. suponha que G tem uma clique C de tamanho k
   - defina o conjunto $V' = V \setminus C$
   - vamos mostrar que V' é uma cobertura por vértices de $\overline{G}$
   - considere dois vértices $u, v \in C$
   - como C é clique de G, (u, v) é uma aresta de G
   - então, (u, v) não é uma aresta de $\overline{G}$
   - isso significa que $\overline{G}[C] = \overline{G} - V'$ não tem arestas
   - portanto, V' é uma cobertura por vértices de $\overline{G}$

Demonstração (cont)

2. Suponha que $\overline{G}$ tem cobertura V' de tamanho $|V| - k$
   - defina o conjunto $C = V \setminus V'$
   - vamos mostrar que C é uma clique em G
   - considere dois vértices $u, v \in C$
   - como V' é cobertura de $\overline{G}$, (u, v) não é uma aresta de G
   - então (u, v) é uma aresta em G
   - portanto C é uma clique em G de tamanho k

Demonstrando NP-completude

- Ciclo hamiltoniano

Ciclo hamiltoniano

Relembrando: considere um grafo não direcionado $G = (V, E)$

- um ciclo hamiltoniano é um ciclo que cobre V
- se G tiver ciclo hamiltoniano, então G é hamiltoniano

Exemplo

Problema do ciclo hamiltoniano

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ é um grafo hamiltoniano } \}$$

Ciclo hamiltoniano é NP-difícil

Teorema

HAM-CYCLE é NP-difícil.

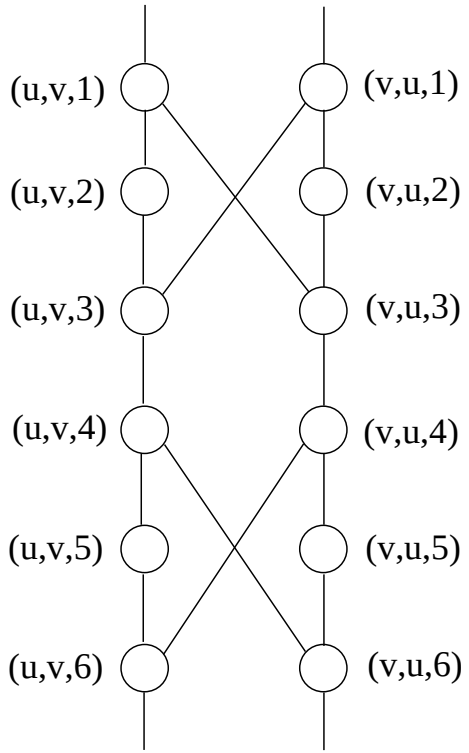
- reduziremos VERTEX-COVER para HAM-CYCLE
- dado grafo G e parâmetro k, criaremos um grafo G'
- queremos que

$\langle G, k \rangle \in$ VERTEX-COVER se e somente se $\langle G' \rangle \in$ HAM-CYCLE

Widget

Criaremos uma estrutura especial

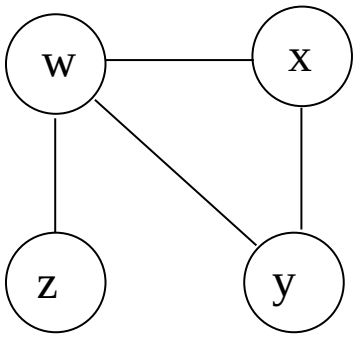
- chamamos esses tipos de estruturas de widgets
- para cada aresta (u, v) de G, criamos W_{uv}



Exemplo de instância

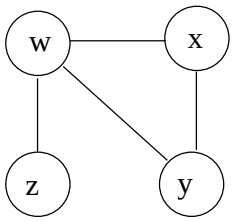
Iremos considerar uma instância de exemplo

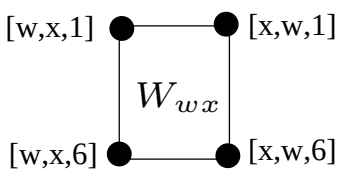
- um grafo $G = (V, E)$ abaixo
- um valor $k = 2$

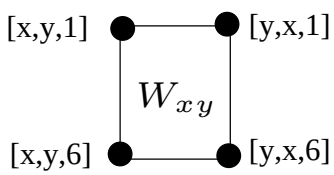


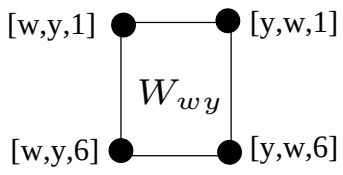
Exemplo

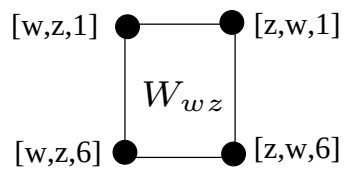
Adicionamos um widget para cada aresta





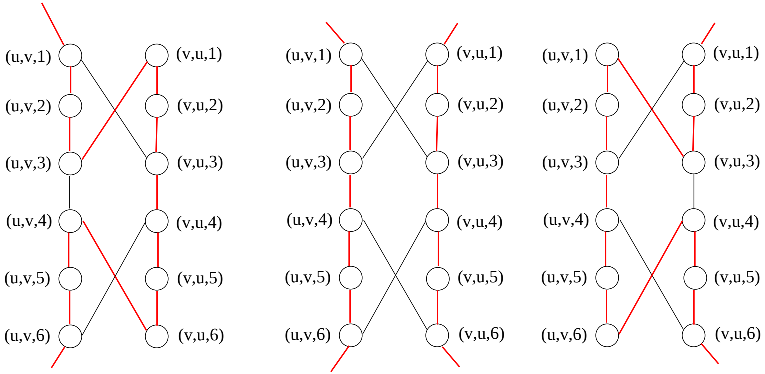






Propriedade do widget

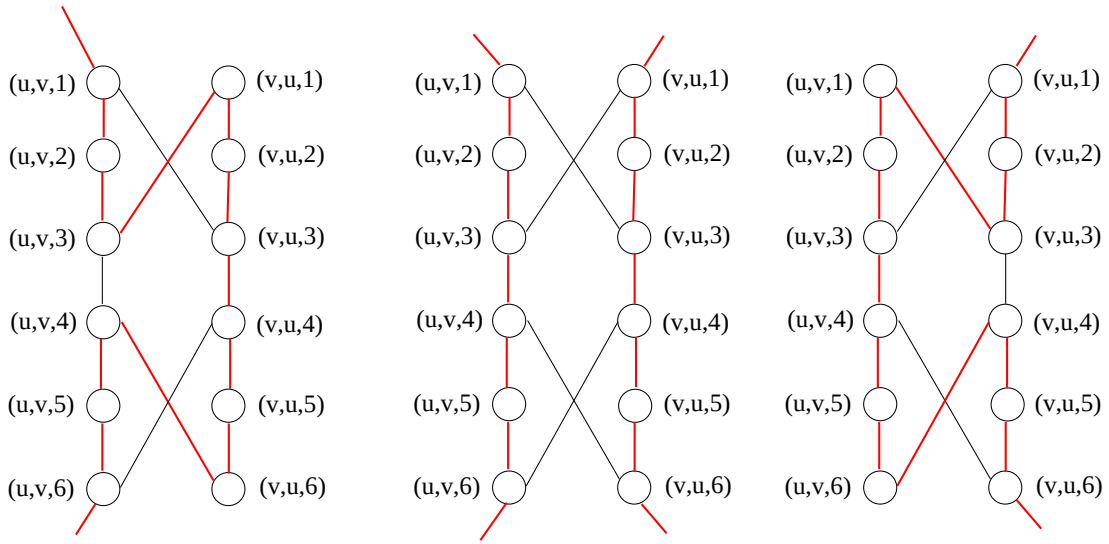
- apenas quatro vértices têm ligação externa
- o importante é que para percorrer todos os vértices dessa estrutura temos apenas 3 opções:



Interpretando o widget

Como utilizar o widget?

- queremos cobrir todas arestas usando vértices
- então vamos passar por todos widgets com um ciclo



Interpretando as opções do ciclo hamiltoniano

1. a aresta é coberta pelo vértice *u* somente
2. a aresta é coberta pelo vértices *u* e *v*
3. a aresta é coberta pelo vértice *v* somente

Escolhendo um vértice

Cobrindo arestas com um vértice

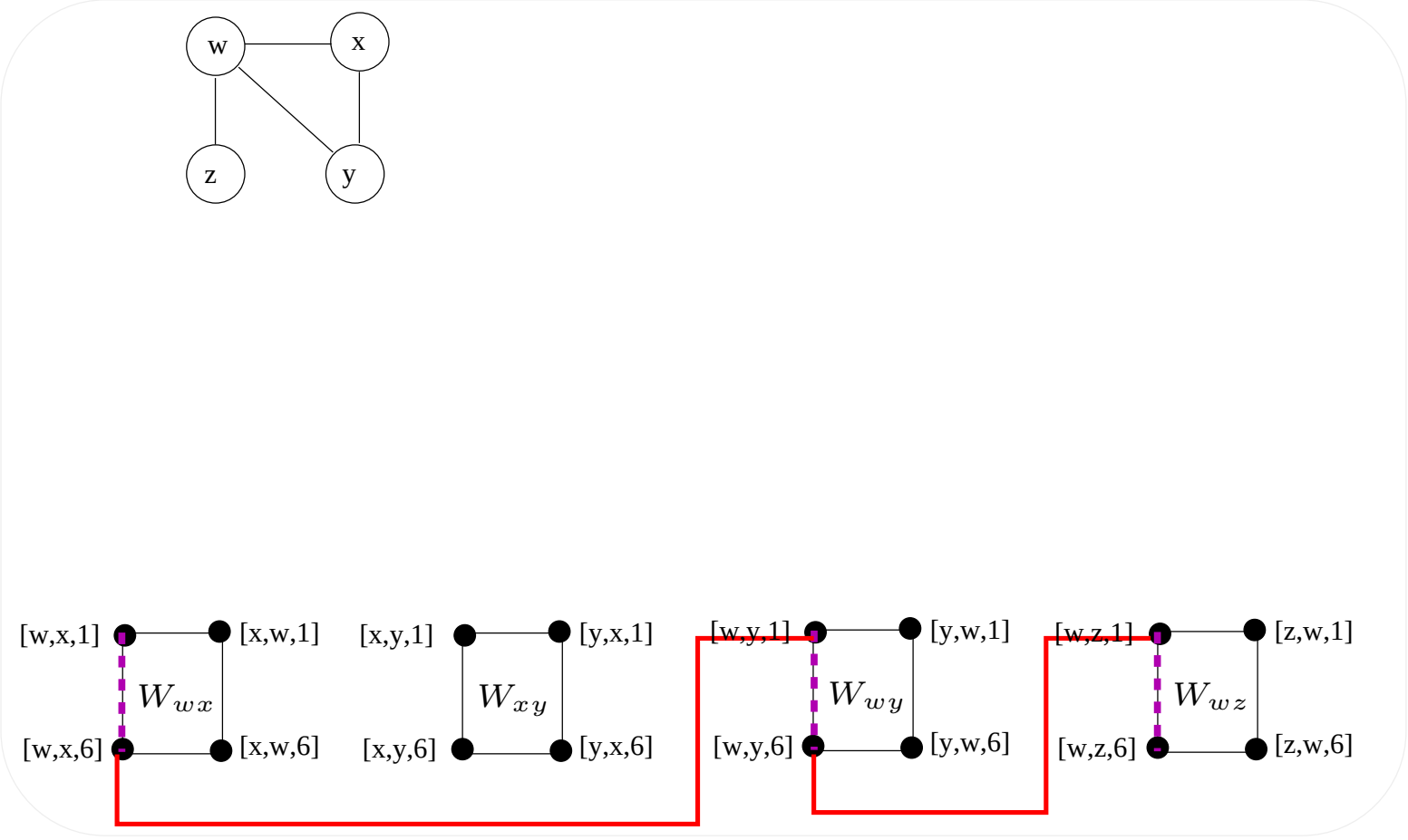
- suponha que u está em uma cobertura por vértices
- cobrimos todas as arestas incidentes em u
- então o ciclo deve passar pelos widgets correspondentes

Como cobrir os widgets correspondentes?

- sejam $u^{(1)}, \ldots, u^{(d(u))}$ os vizinhos de u
- ligamos os widgets das arestas $(u, u^{(1)}), \ldots, (u, u^{(d(u))})$
- adicionamos $\{([u, u^i, 6], [u, u^{(i+1)}, 1) : 1 \leq i \leq d(u) - 1\}$
- assim, existe caminho de $[u, u^{(1)}, 1]$ até $[u, u^{(d(u))}, 6]$
- esse caminho cobre todos esses widgets

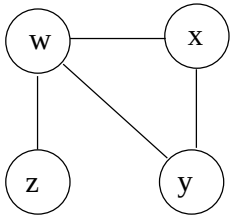
Caminho de um vértice

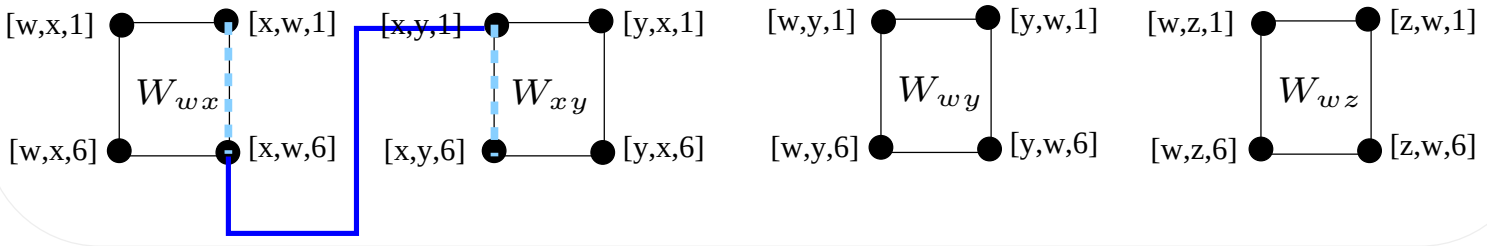
Para o vértice *w* acrescentamos arestas:



Caminho de um vértice

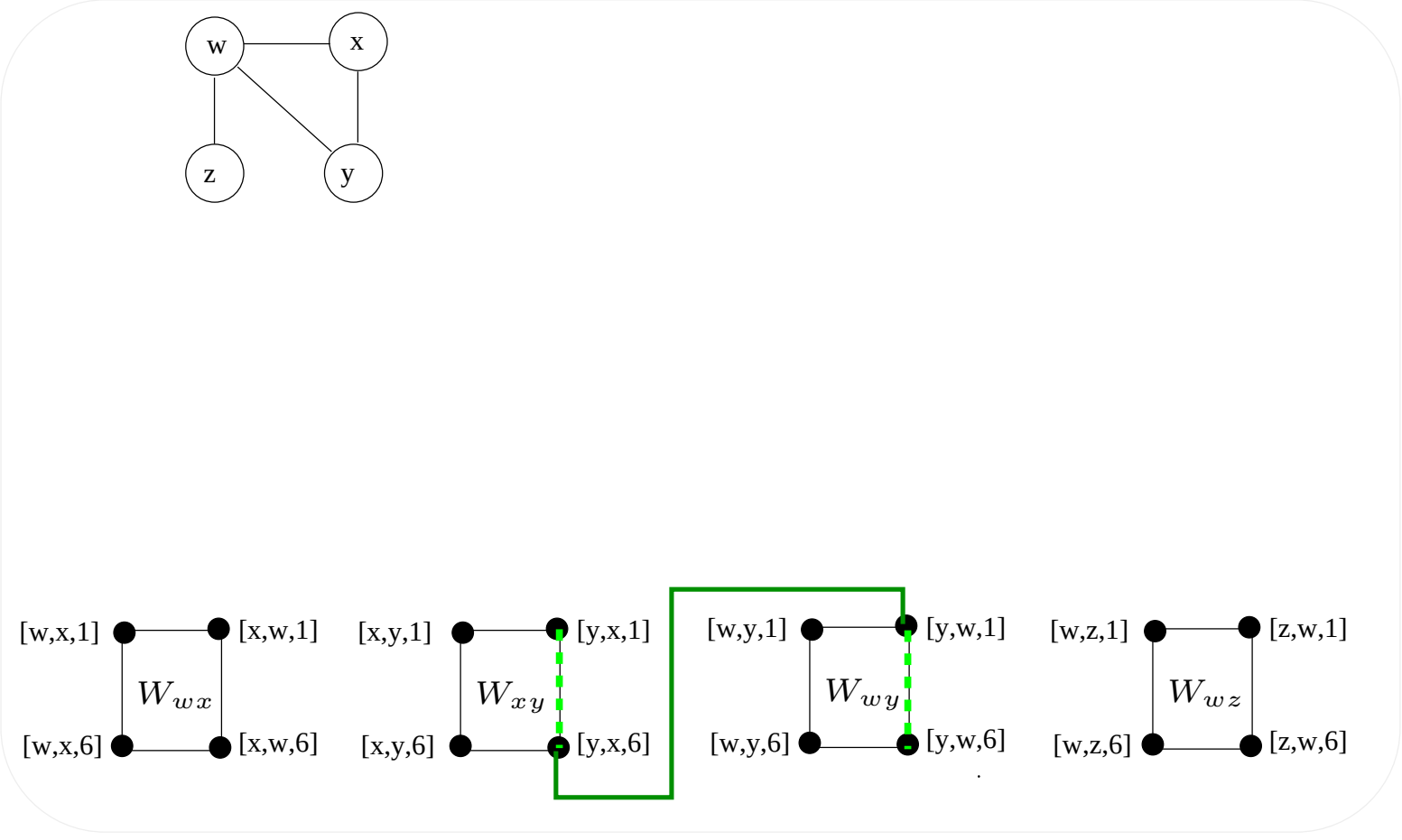
Para o vértice x acrescentamos arestas:





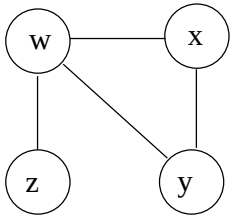
Caminho de um vértice

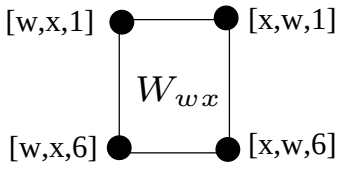
Para o vértice *y* acrescentamos arestas:

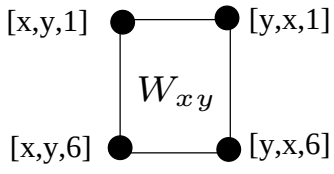


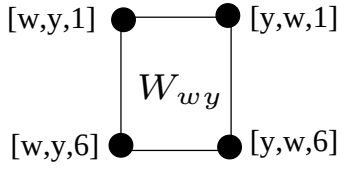
Caminho de um vértice

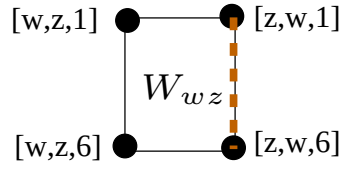
Para o vértice z não acrescentamos arestas:











Vértices seletores

Como selecionar os vértices a partir do ciclo hamiltoniano?

- criaremos mais k vértices $s_1, \ldots, s_k$ chamados seletores
- eles formarão um ciclo com os caminhos:
  - um seletor liga-se ao início de um caminho de vértice
  - um caminho de vértice retorna ao próximo seletor

Assim, precisamos adicionar arestas

1. de cada seletor s_j para cada início de caminho $[u, u^{(1)}, 1]$:

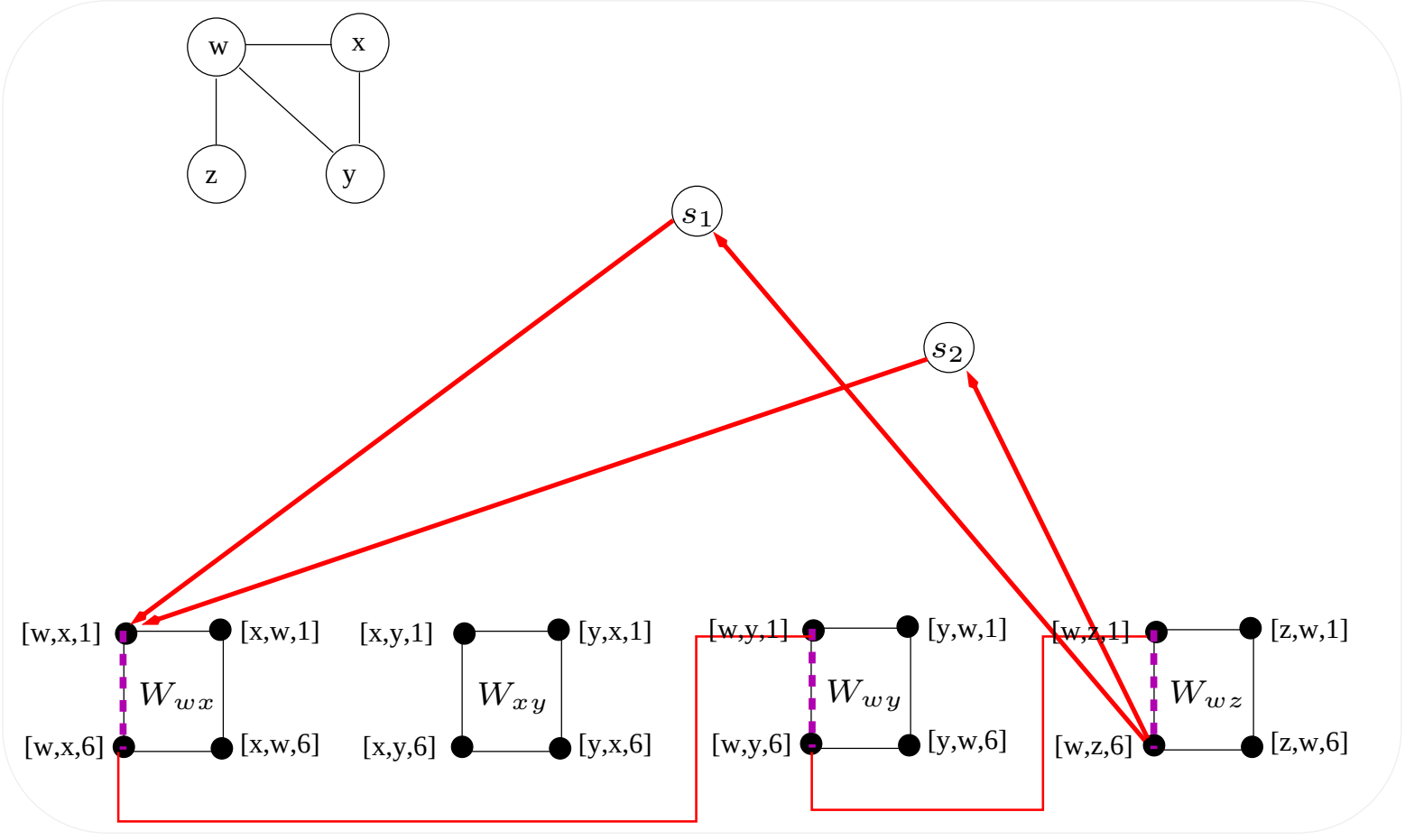
$$\{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ e } 1 \leq j \leq k\}$$

2. de cada fim de caminho $[u, u^{d(u)}, 6]$ para cada seletor s_j e

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

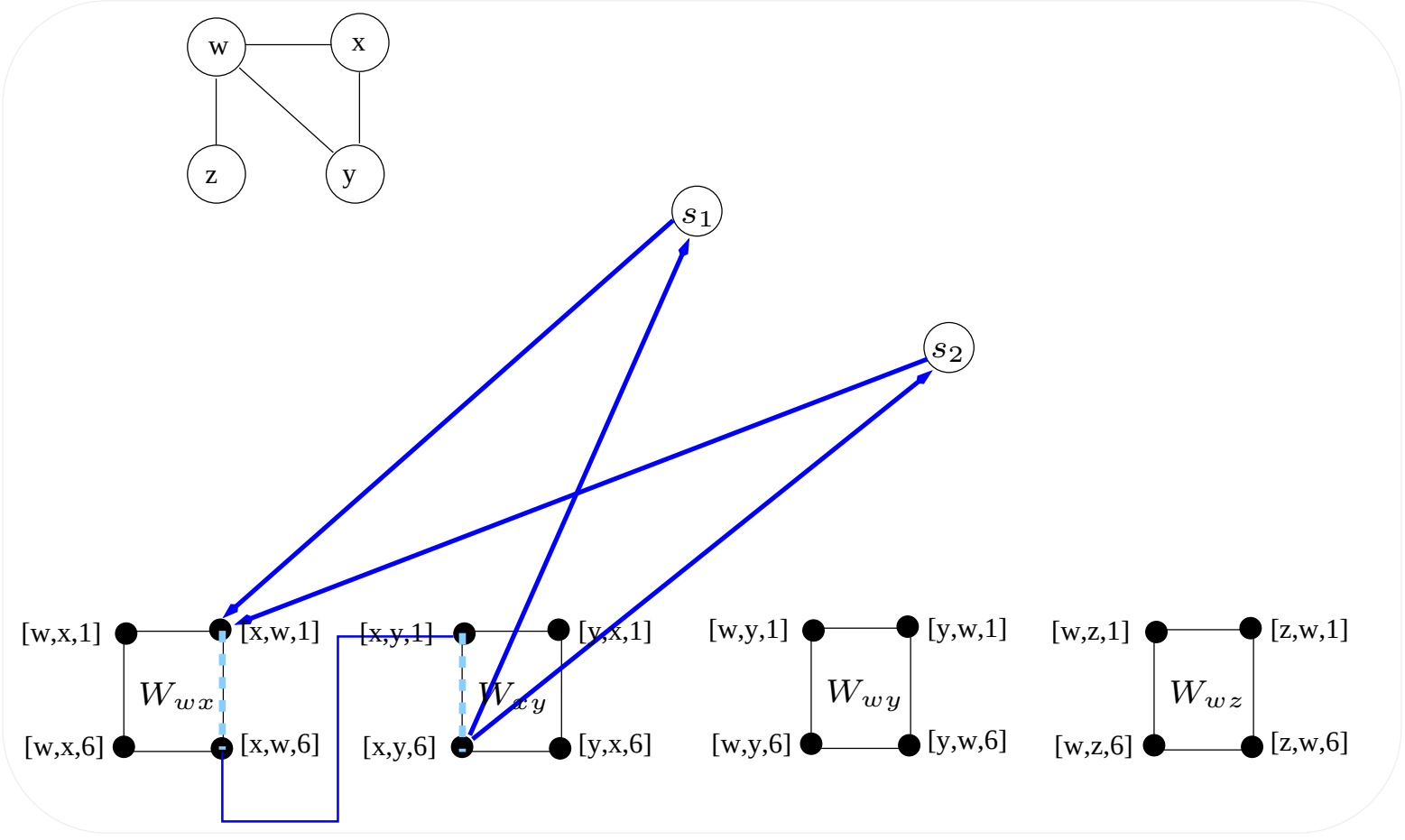
Caminho de um vértice

Ligamos os seletores ao caminho de *w*:



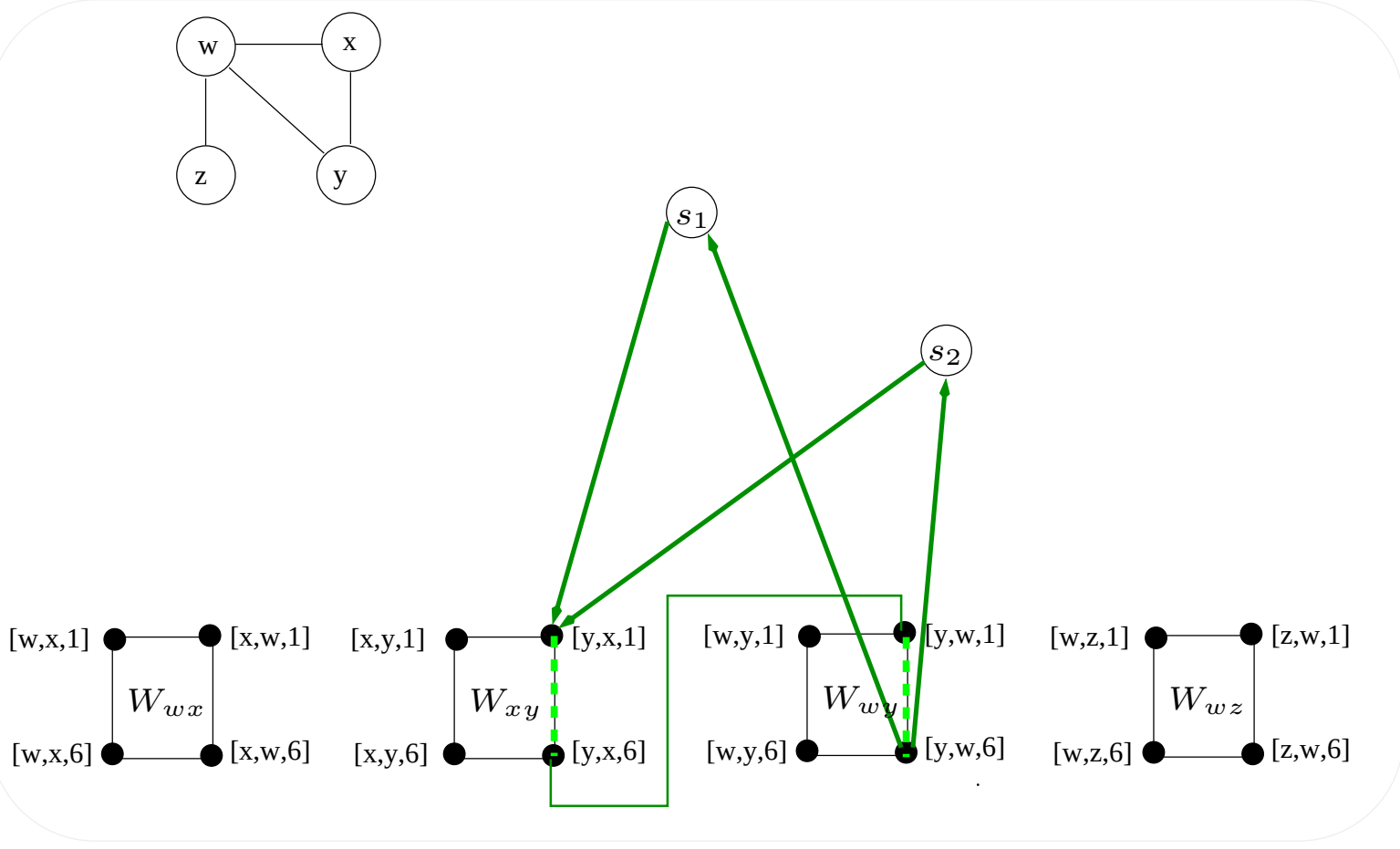
Caminho de um vértice

Ligamos os seletores ao caminho de x:



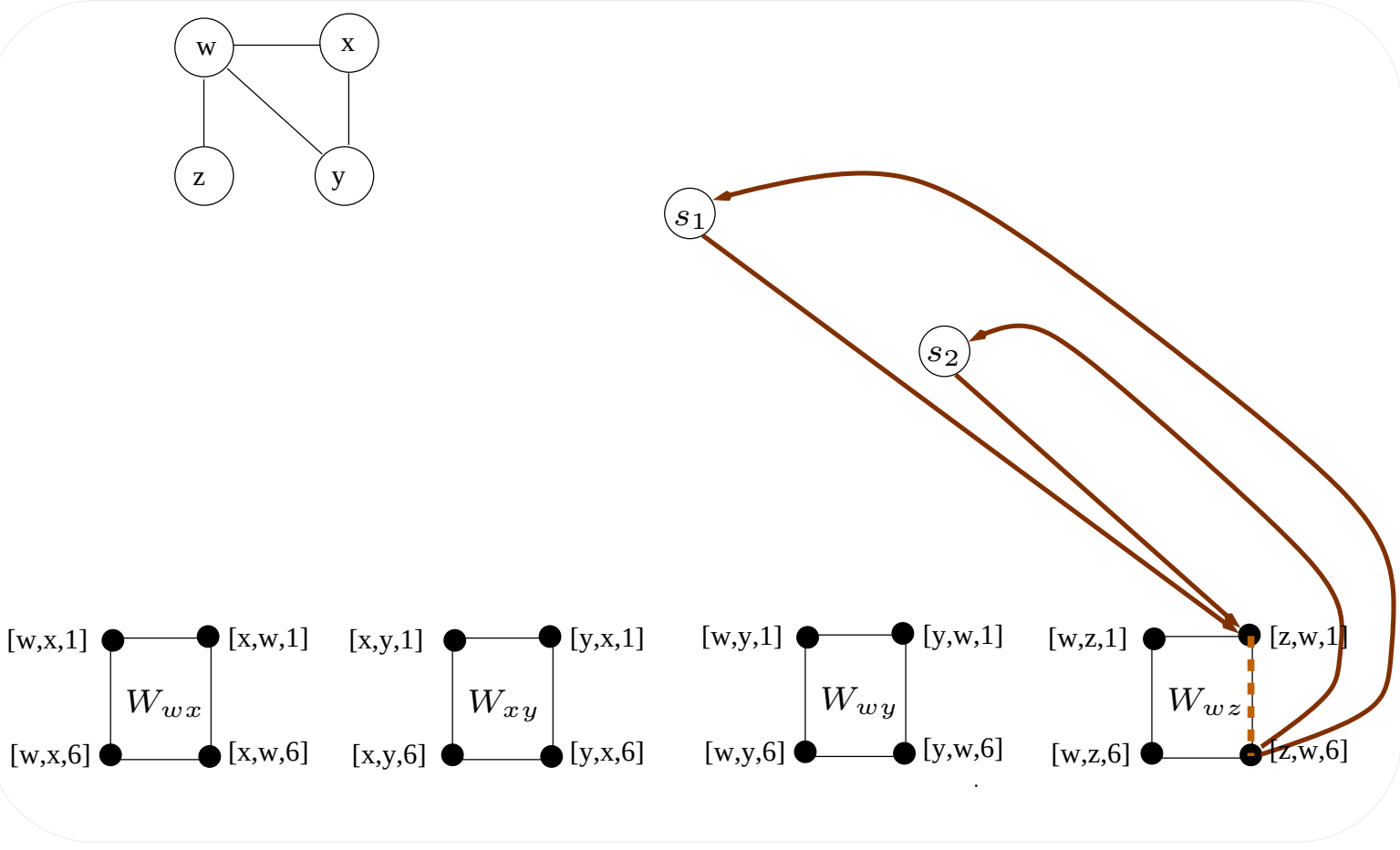
Caminho de um vértice

Ligamos os seletores ao caminho de y:



Caminho de um vértice

Ligamos os seletores ao caminho de z:



Redução é polinomial

Quanto tempo gastamos com a redução?

- começamos com um grafo $G = (V, E)$ e parâmetro k
- construímos outro grafo $G' = (V', E')$ com $|V'| = k + 12 \cdot |E|$
- como $k \leq |V|$, o número de vértices é $|V'| \leq |V| + 12 \cdot |E|$
- também, o número de arestas é $|E'| \leq |V'|(|V'| - 1)/2$
- portanto a construção é polinomial

Equivalência entre instâncias

Queremos mostrar que a redução é segura

1. se G tem cobertura com k vértices, então G' é hamiltoniano
2. se G' é hamiltoniano, então G tem cobertura com k vértices

Demonstração

- em seguida, analisamos cada afirmação
- damos apenas um esboço das ideias
- detalhes formais estão em CLRS

Demonstração da ida

Suponha que G tem uma cobertura com k vértices.

- seja $V^* = \{u_1, \ldots, u_k\}$ uma tal cobertura por vértices
- queremos encontrar um ciclo hamiltoniano de G'
- basta dizer quais arestas estão no ciclo
- e mostrar que todos os vértices são cobertos

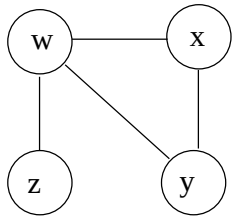
Construímos um subcaminho para cada vértice de V^*

1. ligamos cada seletor s_i ao início do caminho de u_i
2. adicionamos o caminho de cada u_i pelos widgets
3. ligamos a o final de cada caminho de u_i ao próximo seletor

Adicionando subcaminhos

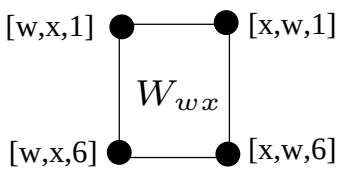
Formando um ciclo hamiltoniano

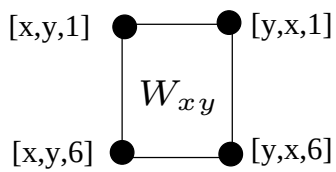
1. observe que $V^* = \{w, y\}$ é uma cobertura com 2 vértices
2. primeiro, adicionamos um subcaminho para w
3. depois, adicionamos um subcaminho para y

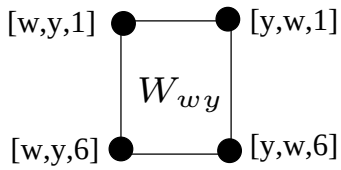


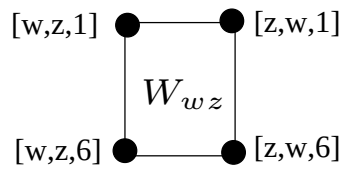
s_1

s_2





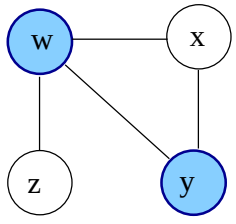




Adicionando subcaminhos

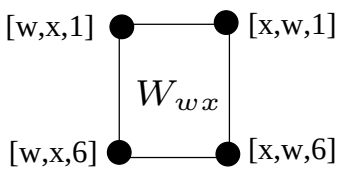
Formando um ciclo hamiltoniano

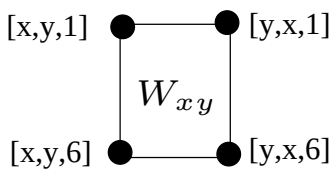
1. observe que $V^* = \{w, y\}$ é uma cobertura com 2 vértices
2. primeiro, adicionamos um subcaminho para w
3. depois, adicionamos um subcaminho para y

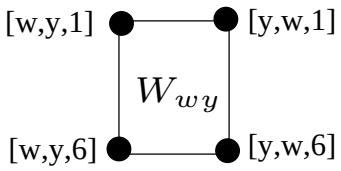


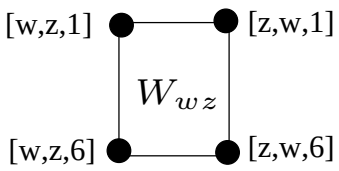
s_1

s_2





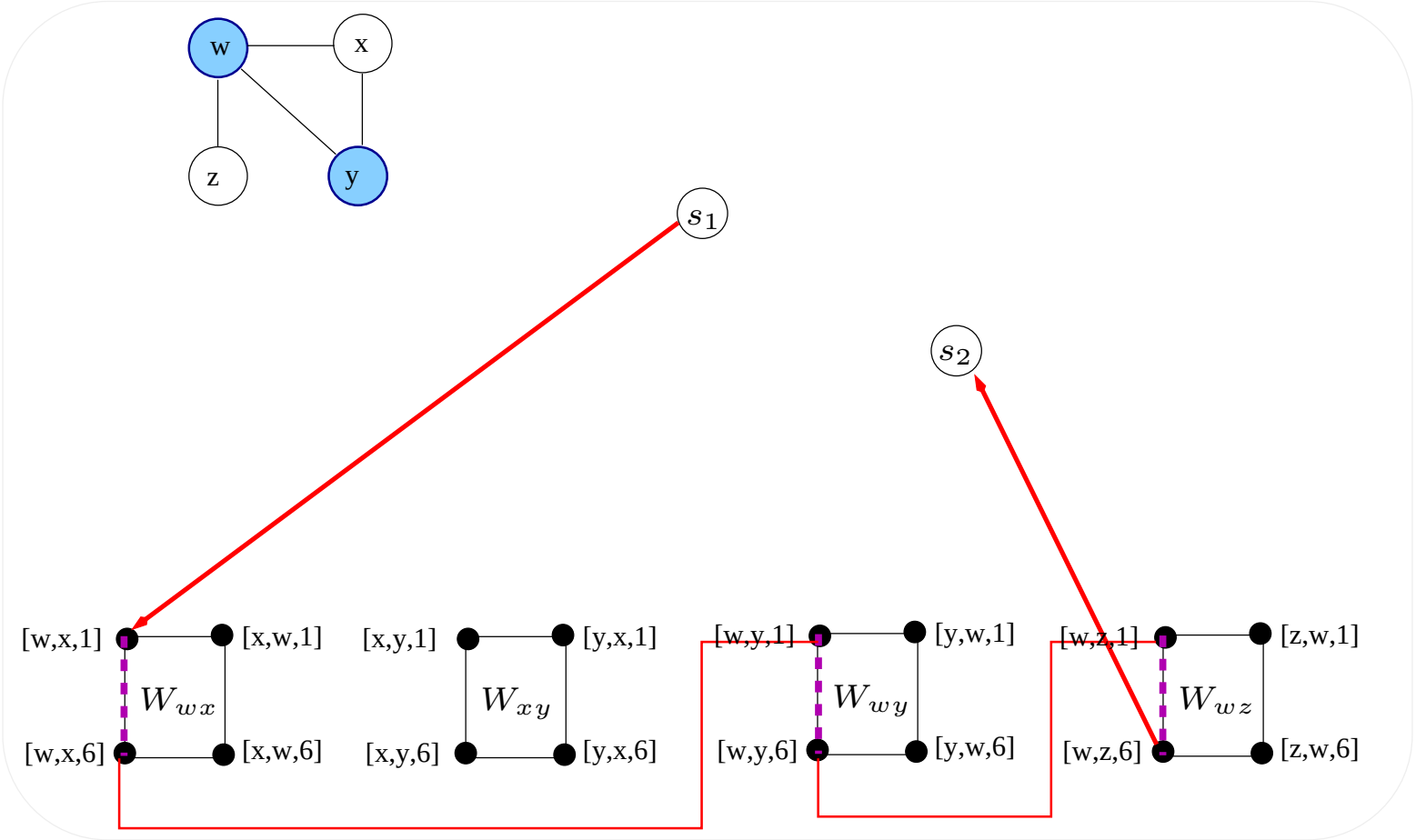




Adicionando subcaminhos

Formando um ciclo hamiltoniano

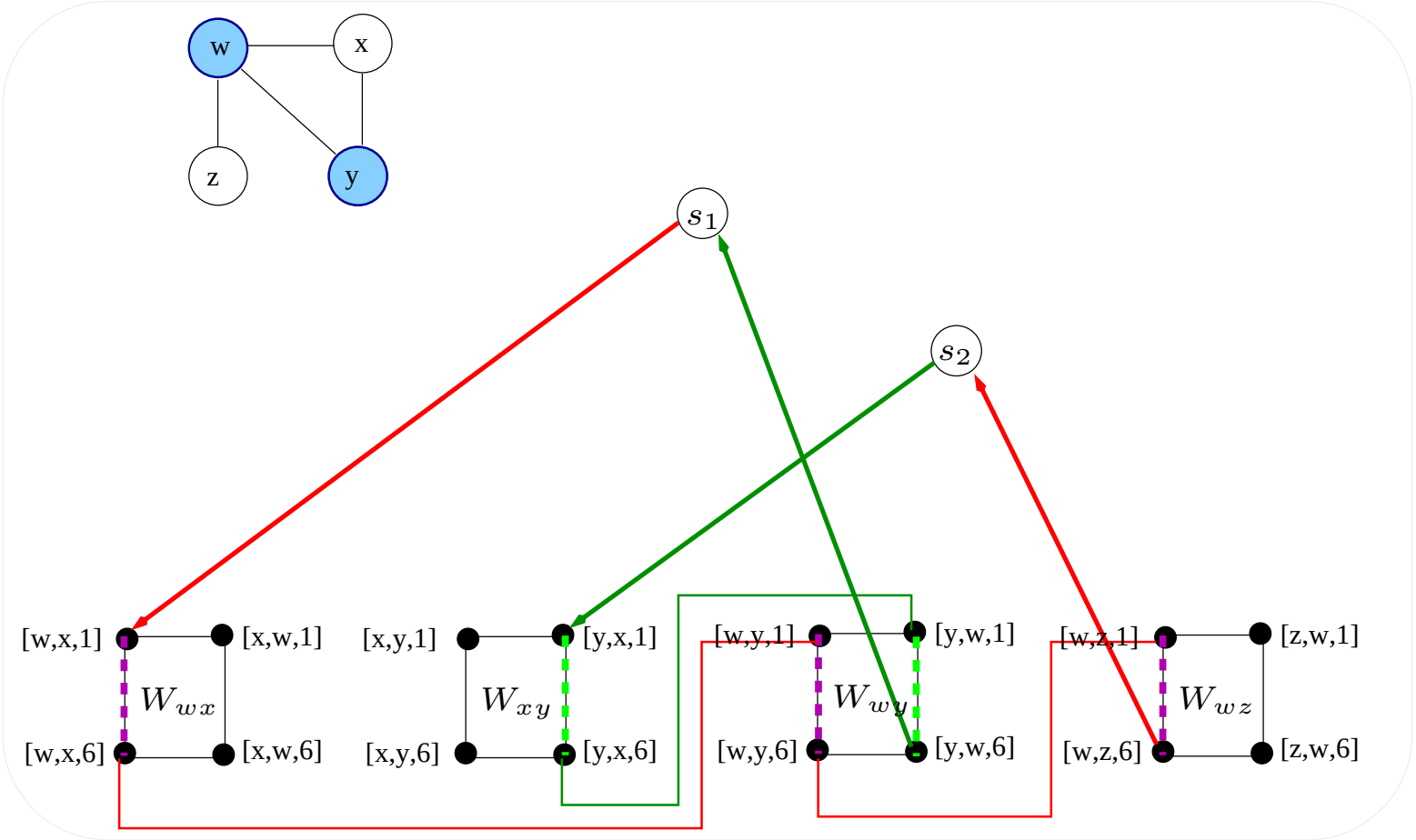
1. observe que $V^* = \{w, y\}$ é uma cobertura com 2 vértices
2. primeiro, adicionamos um subcaminho para w
3. depois, adicionamos um subcaminho para y



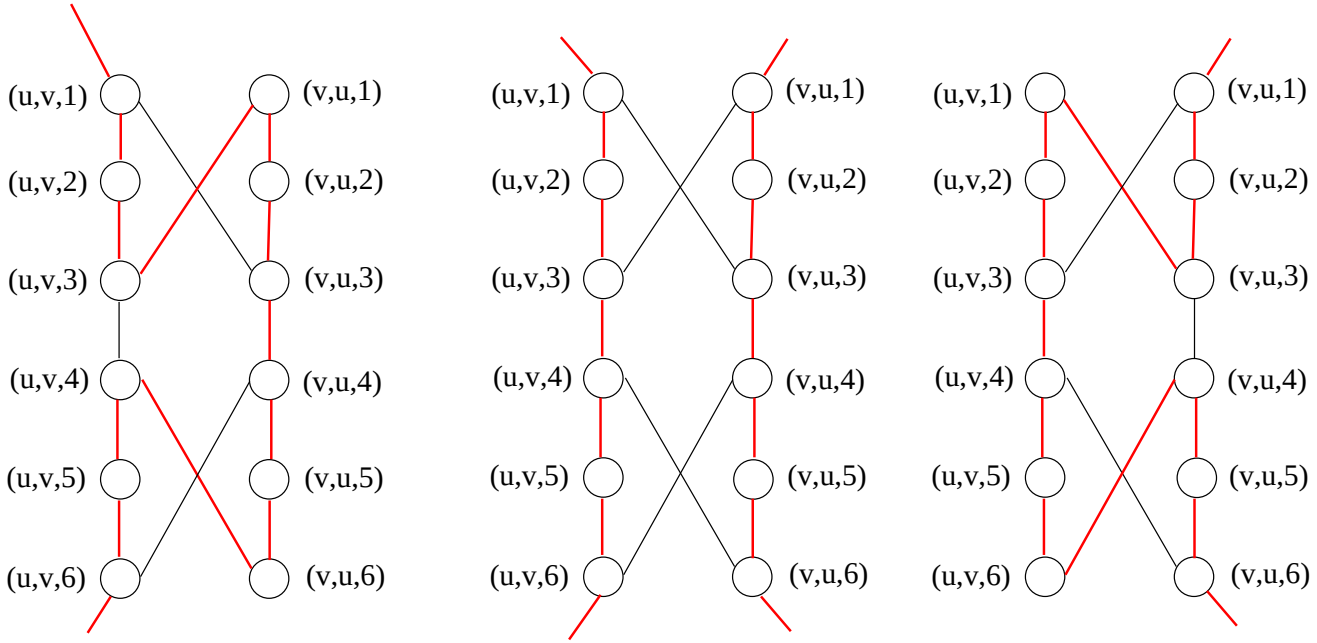
Adicionando subcaminhos

Formando um ciclo hamiltoniano

1. observe que $V^* = \{w, y\}$ é uma cobertura com 2 vértices
2. primeiro, adicionamos um subcaminho para w
3. depois, adicionamos um subcaminho para y



Demonstração da ida (cont)



Falta detalhar as arestas dos widgets

- todo widget W_{uv} é percorrido por um ou dois subcaminhos
- os subcaminhos correspondem a u, a v ou a ambos
  1. se corresponde a u, escolha a primeira opção
  2. se corresponde a ambos, escolha a opção do meio
  3. se corresponde a v, escolha a última opção

Demonstração da ida (cont)

As arestas adicionas formam um ciclo

- por construção, as arestas formam um caminho fechado
- observe que nenhum vértice é percorrido duas vezes
- então as arestas formam um ciclo

O ciclo é hamiltoniano

- por construção, todo vértice seletor é percorrido
- considere vértices de W_{uv}
- como V^* é cobertura, u ou v foi selecionado
- então W_{uv} foi percorrido por algum subcaminho
- assim todo vértice foi percorrido
- portanto o ciclo é hamiltoniano

Demonstração da volta

Suponha que G' tem uma ciclo hamiltoniano.

- seja $C \subseteq E'$ arestas desse ciclo
- como C é hamiltoniano, cada seletor é percorrido por C
- defina

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

- ou seja, os "vértices" adjacentes a algum seletor em C
- mostraremos que V^* é uma cobertura de G com k vértices

Demonstração da volta (cont)

- se um caminho entra em $[u, u^{(1)}, 1]$, ele sai por $[u, u^{(1)}, 6]$
- mas se ele sai de $[u, u^{(1)}, 6]$, ele entra em $[u, u^{(2)}, 1]$
- mas se ele entra de $[u, u^{(2)}, 1]$, ele sai por $[u, u^{(2)}, 6]$
- e assim por diante até o próximo vértice seletor
- daí, o ciclo é formado por subcaminhos de vértices de V^*
- então todo widget é percorrido por algum subcaminho
- portanto, cada aresta (u, v) tem um vértice incidente em V^* correspondente ao subcaminho que percorre W_{uv}
- concluímos que V^* é uma cobertura com k vértices

Demonstrando NP-completude

- Problema do caixeiro viajante

Problem do caixeiro viajante

Problema do caixeiro viajante

Entrada:

- um grafo completo $G = (V, E)$
- uma função de custo $c : V \times V \to \mathbb{Z}$

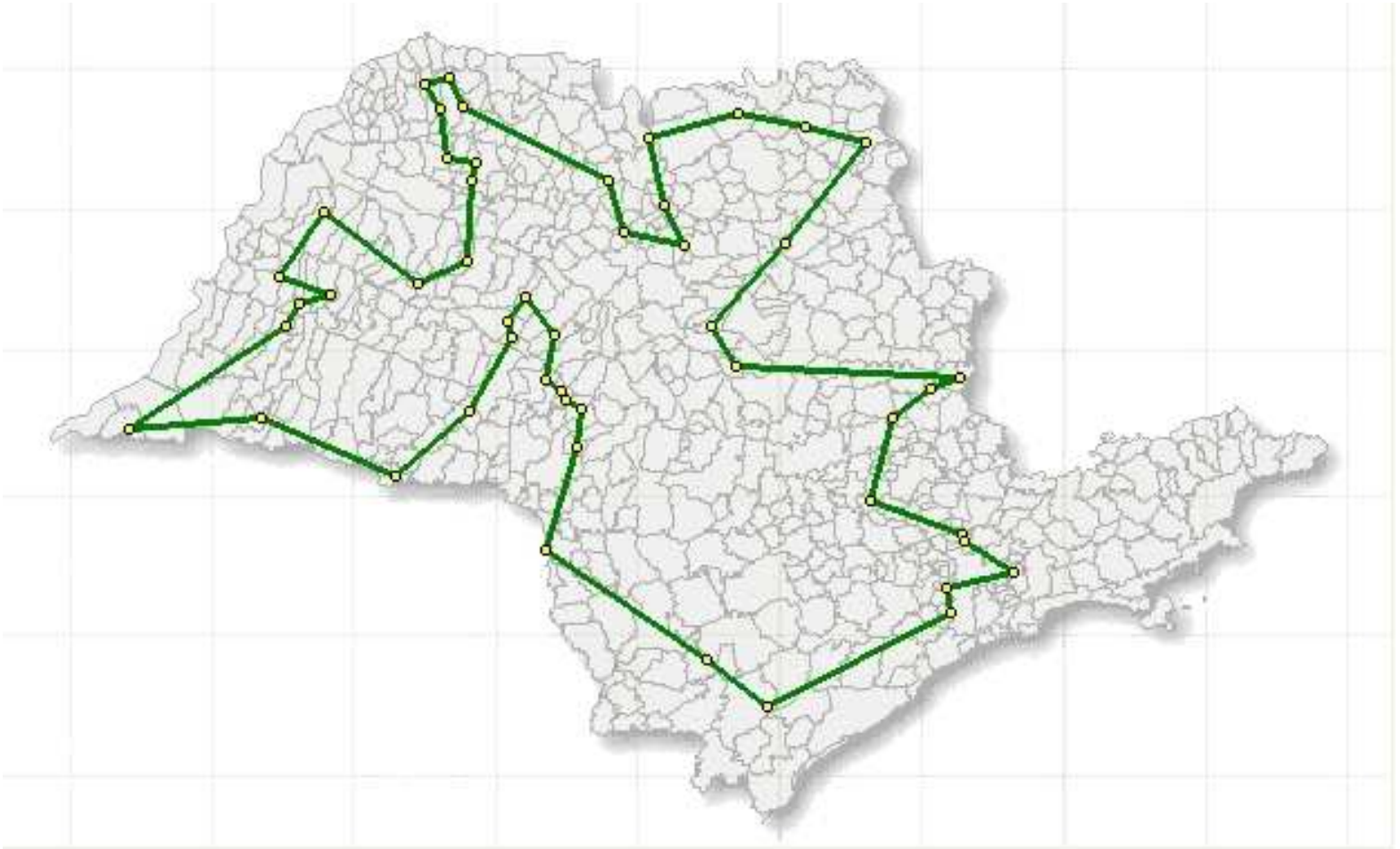
Solução:

- um ciclo hamiltoniano C de G

Objetivo:

- minimizar $\sum_{(u,v) \in E[C]} c(u, v)$

Exemplo



Caixeiro viajante é NP

Refletindo

- a entrada contém um grafo completo
- então é claro que ele tem ciclo hamiltoniano
- a difículdade é encontrar o de custo mínimo

Caixeiro viajante

$$\text{TSP} = \{\langle G, c, k\rangle : G \text{ tem ciclo hamiltoniano de custo } k\}$$

- a sigla TSP vem de *traveling-salesman problem*

Caixeiro viajante é NP-difícil

Teorema

TSP é NP-difícil.

- reduziremos HAM-CYCLE para TSP
- dado um grafo G, vamos construir um grafo completo G'
- vamos mostrar que G é hamiltoniano sse G' tiver um ciclo hamiltoniano de custo 0

Redução

Dada uma instância $G = (V, E)$ de HAM-CYCLE

- construa um grafo completo G'
- para cada par $(u, v) \in V$, defina

$$c(u, v) = \begin{cases} 0 & \text{se } (u, v) \text{ é aresta de } G \\ 1 & \text{se } (u, v) \text{ não é arestas de } G \end{cases}$$

- defina $k = 0$

Demonstração

A redução é polinomial

- o grafo G tem $|V|$ vértices
- como G' é completo, ele tem $|V|(|V|-1)/2$ arestas
- então podemos construí-lo em tempo polinomial

As instâncias são equivalentes

$\Rightarrow$
- suponha que G possui um ciclo hamiltoniano
- então esse ciclo tem custo 0 em G'

$\Leftarrow$
- suponha que G' possui um ciclo hamiltoniano de custo 0
- então esse ciclo não usa arestas de custo 1
- portanto ele é um ciclo hamiltoniano de G

Demonstrando NP-completude

- Soma de subconjunto

Soma de subconjunto

Problema da soma de subconjunto

- **Entrada:** um conjunto de naturais S e um natural t
- **Pergunta:** existe subconjunto $S' \subseteq S$ tal que $\sum_{s \in S'} s = t$?

Soma de subconjunto

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

Exemplo

Exemplo de instância

- sejam $S = \{1, 4, 10, 14, 54, 100, 1004, 1003\}$ e $t = 1027$
- nesse caso, $S' = \{10, 14, 1003\}$ é uma solução
- então $\langle S, t \rangle$ é uma instância **sim** do problema

Vamos mostrar que SUBSET-SUM é NP-completo.

Soma de subconjunto é NP

Lema

SUBSET-SUM é NP.

Demonstração

- exercício

Soma de subconjunto é NP-difícil

Teorema

SUBSET-SUM é NP-difícil.

- reduziremos 3CNF-SAT para SUBSET-SUM
- recebemos uma fórmula f com variáveis $x_1, \ldots, x_n$ e cláusulas $C_1, \ldots, C_k$ com 3 literais cada
- sem perda de generalidade, supomos que uma variável e sua negação não aparecem na mesma cláusula

Redução

Vamos criar uma série de números naturais

- cada um tem $n + k$ dígitos decimais
- cada dígito corresponde a uma variável ou uma cláusula
- os dígitos estão em ordem $x_1, x_2, \ldots, x_n, C_1, \ldots, C_k$

Conjunto de números S

- criamos dois números para cada variável e cada cláusula

Número alvo t

- criamos um número especial

Exemplo de instância reduzida

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

	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

Redução (cont)

Para uma variável x_i

1. 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 em C_j
2. também 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 cláusula C_i

1. criamos um número s_i com um 1 no dígito C_i
2. também um número s_i' com um 2 no dígito C_i

Para o número t

1. criamos um número com um 1 em cada dígito x_i e um 4 em cada dígito C_i

Exemplo de instância reduzida

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

	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

Demonstração

Propriedades da instância reduzida

- todos os números são distintos, pois uma cláusula não tem uma variável e sua negação
- os dígitos $x_1, \ldots, x_n$ são diferentes para todos os números correspondentes a variáveis ou cláusulas

Se somarmos todos os números de S

- o maior valor que um dígito pode atingir é 6, pois um dígito C_j corresponde a um cláusula com 3 literais
- então a soma em um dígito não interfere na soma de outro dígito mais significativo

Demonstração (cont)

A redução leva tempo polinomial.

- além de t, o conjunto S tem $2(n+k)$ números decimais
- cada um dos números tem $n+k$ dígitos

Vamos mostrar a equivalência das instâncias:

$$f \text{ é satisfazível sse existe } S' \subseteq S \text{ tal que } \sum_{s \in S'} s = t$$

Demonstração (cont)

1. Suponha que f é satisfazível
   - seja $x_1, \ldots, x_n$ uma atribuição de variáveis que satisfaça f
   - vamos criar um subconjunto $S' \subseteq S$
   - para cada variável x_i
     - se $x_i = 1$, adicione v_i ao conjunto S'
     - se $x_i = 0$, adicione v_i' ao conjunto S'
   - para cada cláusula C_j
     - se todos três literais de C_j forem satisfeitos, adicione s_j a S'
     - se apenas dois literais forem satisfeitos, adicione s_j' a S'
     - se apenas um literal for satisfeito, adicione s_j e s_j' a S'
   - defina $t' = \sum_{s \in S'} s$
   - cada dígito x_i de t' vale 1, pois v_i ou v_i' está em S'
   - cada dígito C_j de t' vale 4 pela forma que inserimos s_i e s_i'
   - concluímos que $t' = t$ e a instância reduzida é sim

Demonstração (cont)

2. Suponha que existe $S' \subseteq S$ tal que $\sum_{s \in S'} s = t$
   - relembre que temos 1 no dígito x_i de t
   - então exatamente um número de v_i e v_i' está em S'
   - isso induz uma atribuição de cada variável x_i
     - se $v_i \in S'$, então faça $x_i = 1$
     - se $v_i' \in S'$, então faça $x_i = 0$
   - também, temos 4 no dígito C_j de t
   - assim há pelo menos um número em S' que tem 1 em C_j
   - esse número corresponde a um literal de C_j
   - então a atribuição induzida satisfaz cada cláusula
   - portanto f é satisfazível

Outros tópicos de complexidade

Outros tópicos de complexidade

- Complexidade de espaço

Espaço utilizado

Podemos analisar algoritmos em termos de espaço utilizado

Definição

O espaço utilizado por um algoritmo corresponde ao número de bits que ele acessa durante sua execução.

- um mesmo bit pode ser acessado várias vezes
- estamos interessados na memória máxima utilizada

Complexidade de espaço

Também classificamos problemas em termos de espaço

Definição

$$\text{SPACE}(f(n)) = \{L : L \text{ é uma linguagem decidida utilizando espaço } O(f(n))\}$$

Definição

$$\text{NSPACE}(f(n)) = \{L : L \text{ é uma linguagem verificada utilizando espaço } O(f(n))\}$$

- por exemplo, linguagens em NP usam espaço polinomial

Espaço polinomial

Definição

PSPACE é o conjunto de linguagens decididas por algoritmos que usam espaço polinomial:

$$\text{PSPACE} = \bigcup_{k \geq 1} \text{SPACE}(n^k)$$

Definição

NPSPACE é o conjunto de linguagens verificadas por algoritmos que usam espaço polinomial:

$$\text{NPSPACE} = \bigcup_{k \geq 1} \text{NSPACE}(n^k)$$

Espaço polinomial

Será qeu PSPACE $\neq$ NPSPACE?

Teorema de Savitch

Para qualquer função $f : \mathbb{N} \to \mathbb{R}$

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

- em outras palavras, se podemos verificar usando $f(n)$ bits
- então também podemos decidir usando $f^2(n)$ bits

Espaço polinomial

Teorema

PSPACE = NPSPACE.

Demonstração:

- claro que PSPACE $\subseteq$ NPSPACE
- seja L uma linguagem em NPSPACE
- ela é verificada usando $O(n^k)$ bits alguma constante k
- por Savitch, podemos decidir L com $O(n^{2k})$ bits
- portanto $L \in$ PSPACE e NPSPACE $\subseteq$ PSPACE

Relações entre classes

Conhecemos as seguintes relações

$$P \subseteq NP \subseteq PSPACE = NPSPACE$$

- é um problema em aberto determinar se $P \neq NP$
- assim como determinar se $NP \neq PSPACE$

Outros tópicos de complexidade

- Indecibilidade

Indecibilidade

Até agora, estudamos algoritmos

- tentamos estimar os recursos necessários
- comparamos diferentes algoritmos
- e estabelecemos limitantes para os problema

Mas existem problemas para os quais não existem algoritmos!

- ou seja, a relação de entrada e saída é bem definida
- mas não há algoritmo que compute a saída
- esses problemas são chamados de indecidíveis

Um exemplo

Problema da parada

- **Entrada:** uma string s que codifica um algoritmo A
- **Pergunta:** o algoritmo A para se receber s como entrada?

Uma ideia ineficaz:

- criamos um algoritmo A' que simula A sobre entrada s
- se a simulação de A terminar, devolva **sim**
- mas se A nunca parar?

Não é difícil mostrar que esse problema é indecidível!