Projeto e Análise de Algoritmos*

NP Complexidade Computacional

Segundo Semestre de 2019

*Criado por C. de Souza, C. da Silva, O. Lee, F. Miyazawa et al.

A maior parte deste conjunto de slides foi inicialmente preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para cursos de Análise de Algoritmos. Além desse material, diversos conteúdos foram adicionados ou incorporados por outros professores, em especial por Orlando Lee e por Flávio Keidi Miyazawa. Os slides usados nessa disciplina são uma junção dos materiais didáticos gentilmente cedidos por esses professores e contêm algumas modificações, que podem ter introduzido erros.

O conjunto de slides de cada unidade do curso será disponibilizado como guia de estudos e deve ser usado unicamente para revisar as aulas. Para estudar e praticar, leia o livro-texto indicado e resolva os exercícios sugeridos.

Lehilton

Agradecimentos (Cid e Cândida)

- Várias pessoas contribuíram direta ou indiretamente com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Uma lista destes "colaboradores" (em ordem alfabética) é dada abaixo:
  - Célia Picinin de Mello
  - Flávio Keidi Miyazawa
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias

Introdução à NP-Complexidade

Algoritmos eficientes

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

Como identificar algoritmos rápidos?

- Um algoritmo é polinomial tiver tempo $O(n^k)$ para k constante
- Diremos que um algoritmo é eficiente quando ele for polinomial

Algoritmos eficientes

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

Como identificar algoritmos rápidos?

- Um algoritmo é polinomial tiver tempo $O(n^k)$ para k constante
- Diremos que um algoritmo é eficiente quando ele for polinomial

Algoritmos eficientes

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

Como identificar algoritmos rápidos?

- Um algoritmo é polinomial tiver tempo $O(n^k)$ para k constante
- Diremos que um algoritmo é eficiente quando ele for polinomial

Algoritmos eficientes

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

Como identificar algoritmos rápidos?

- Um algoritmo é polinomial tiver tempo $O(n^k)$ para k constante
- Diremos que um algoritmo é eficiente quando ele for polinomial

Algoritmos eficientes

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

Como identificar algoritmos rápidos?

- Um algoritmo é polinomial tiver tempo $O(n^k)$ para k constante
- Diremos que um algoritmo é eficiente quando ele for polinomial

Algoritmos eficientes

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

Como identificar algoritmos rápidos?

- Um algoritmo é polinomial tiver tempo $O(n^k)$ para k constante
- Diremos que um algoritmo é eficiente quando ele for polinomial

Algoritmos eficientes

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

Como identificar algoritmos rápidos?

- Um algoritmo é polinomial tiver tempo $O(n^k)$ para k constante
- Diremos que um algoritmo é eficiente quando ele for polinomial

Organizando os problemas

Por que se preocupar com isso?

- Há milhares de problemas reais para os quais não conhecemos algoritmos polinomiais
- E acreditamos que não há algoritmos para muitos destes problemas

Queremos identificar que problemas são polinomiais!

Mas primeiro...

1. O que é e como representar problemas?
   R.: problemas de decisão formalizados como linguagens formais
2. Como comparar problemas?
   R.: usaremos um tipo particular de redução

Organizando os problemas

Por que se preocupar com isso?

- Há milhares de problemas reais para os quais não conhecemos algoritmos polinomiais
- E acreditamos que não há algoritmos para muitos destes problemas

Queremos identificar que problemas são polinomiais!

Mas primeiro...

1. O que é e como representar problemas?
   R.: problemas de decisão formalizados como linguagens formais
2. Como comparar problemas?
   R.: usaremos um tipo particular de redução

Organizando os problemas

Por que se preocupar com isso?

- Há milhares de problemas reais para os quais não conhecemos algoritmos polinomiais
- E acreditamos que não há algoritmos para muitos destes problemas

Queremos identificar que problemas são polinomiais!

Mas primeiro...

1. O que é e como representar problemas?
   R.: problemas de decisão formalizados como linguagens formais
2. Como comparar problemas?
   R.: usaremos um tipo particular de redução

Organizando os problemas

Por que se preocupar com isso?

- Há milhares de problemas reais para os quais não conhecemos algoritmos polinomiais
- E acreditamos que não há algoritmos para muitos destes problemas

Queremos identificar que problemas são polinomiais!

Mas primeiro...

1. O que é e como representar problemas?
   R.: problemas de decisão formalizados como linguagens formais
2. Como comparar problemas?
   R.: usaremos um tipo particular de redução

Organizando os problemas

Por que se preocupar com isso?

- Há milhares de problemas reais para os quais não conhecemos algoritmos polinomiais
- E acreditamos que não há algoritmos para muitos destes problemas

Queremos identificar que problemas são polinomiais!

Mas primeiro...

1. O que é e como representar problemas?
   R.: problemas de decisão formalizados como linguagens formais
2. Como comparar problemas?
   R.: usaremos um tipo particular de redução

Organizando os problemas

Por que se preocupar com isso?

- Há milhares de problemas reais para os quais não conhecemos algoritmos polinomiais
- E acreditamos que não há algoritmos para muitos destes problemas

Queremos identificar que problemas são polinomiais!

Mas primeiro...

1. O que é e como representar problemas?

   R.: problemas de decisão formalizados como linguagens formais

2. Como comparar problemas?

   R.: usaremos um tipo particular de redução

Organizando os problemas

Por que se preocupar com isso?

- Há milhares de problemas reais para os quais não conhecemos algoritmos polinomiais
- E acreditamos que não há algoritmos para muitos destes problemas

Queremos identificar que problemas são polinomiais!

Mas primeiro...

1. O que é e como representar problemas?
   R.: problemas de decisão formalizados como linguagens formais
2. Como comparar problemas?
   R.: usaremos um tipo particular de redução

Organizando os problemas

Por que se preocupar com isso?

- Há milhares de problemas reais para os quais não conhecemos algoritmos polinomiais
- E acreditamos que não há algoritmos para muitos destes problemas

Queremos identificar que problemas são polinomiais!

Mas primeiro...

1. O que é e como representar problemas?
   R.: problemas de decisão formalizados como linguagens formais
2. Como comparar problemas?
   R.: usaremos um tipo particular de redução

Problemas de Decisão

Um problema de decisão é um problema cuja a resposta é ***sim*** *ou* ***não****.*

Exemplos de problema de decisão

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

Não são de decisão: soma, ordenação, caminho mínimo etc.

Por que estudar problemas de decisão?

- são mais simples de estudar!
- às vezes saber se há solução é tão difícil quanto encontrá-la

Problemas de Decisão

Um problema de decisão é um problema cuja a resposta é ***sim*** *ou* ***não****.*

Exemplos de problema de decisão

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

Não são de decisão: soma, ordenação, caminho mínimo etc.

Por que estudar problemas de decisão?

- são mais simples de estudar!
- às vezes saber se há solução é tão difícil quanto encontrá-la

Problemas de Decisão

Um problema de decisão é um problema cuja a resposta é ***sim*** *ou* ***não****.*

Exemplos de problema de decisão

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

Não são de decisão: soma, ordenação, caminho mínimo etc.

Por que estudar problemas de decisão?

- são mais simples de estudar!
- às vezes saber se há solução é tão difícil quanto encontrá-la

Problemas de Decisão

Um problema de decisão é um problema cuja a resposta é ***sim*** *ou* ***não****.*

Exemplos de problema de decisão

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

Não são de decisão: soma, ordenação, caminho mínimo etc.

Por que estudar problemas de decisão?

- são mais simples de estudar!
- às vezes saber se há solução é tão difícil quanto encontrá-la

Problemas de Decisão

*Um problema de decisão é um problema cuja a resposta é **sim** ou **não**.*

Exemplos de problema de decisão

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

Não são de decisão: soma, ordenação, caminho mínimo etc.

Por que estudar problemas de decisão?

- são mais simples de estudar!
- às vezes saber se há solução é tão difícil quanto encontrá-la

Problemas de Decisão

Um problema de decisão é um problema cuja a resposta é ***sim*** *ou* ***não****.*

Exemplos de problema de decisão

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

Não são de decisão: soma, ordenação, caminho mínimo etc.

Por que estudar problemas de decisão?

- são mais simples de estudar!
- às vezes saber se há solução é tão difícil quanto encontrá-la

Versão de decisão para problemas de busca

Problema de Busca do Ciclo Hamiltoniano

Dado um grafo $G = (V, E)$, encontre um ciclo em G que passa por cada um dos vértices.

Problema de Decisão do Ciclo Hamiltoniano

Dado um grafo $G(V, E)$, existe um ciclo em G que passa por cada um dos vértices?

Exercício: Faça uma redução de Turing que gasta tempo polinomial do Problema de Busca do Ciclo Hamiltoniano para o Problema de Decisão do Ciclo Hamiltoniano. Dica: descubra uma aresta por vez

Versão de decisão para problemas de busca

Problema de Busca do Ciclo Hamiltoniano

Dado um grafo $G = (V, E)$, encontre um ciclo em G que passa por cada um dos vértices.

Problema de Decisão do Ciclo Hamiltoniano

Dado um grafo $G(V, E)$, existe um ciclo em G que passa por cada um dos vértices?

Exercício: Faça uma redução de Turing que gasta tempo polinomial do Problema de Busca do Ciclo Hamiltoniano para o Problema de Decisão do Ciclo Hamiltoniano. Dica: descubra uma aresta por vez

Versão de decisão para problemas de busca

Problema de Busca do Ciclo Hamiltoniano

Dado um grafo $G = (V, E)$, encontre um ciclo em G que passa por cada um dos vértices.

Problema de Decisão do Ciclo Hamiltoniano

Dado um grafo $G(V, E)$, existe um ciclo em G que passa por cada um dos vértices?

Exercício: Faça uma redução de Turing que gasta tempo polinomial do Problema de Busca do Ciclo Hamiltoniano para o Problema de Decisão do Ciclo Hamiltoniano. Dica: descubra uma aresta por vez

Versão de decisão para problemas de otimização

Problema do Caixeiro Viajante

Dado um grafo completo ponderado $G = (V, E)$, encontre um ciclo Hamiltoniano que passa por todos os vértices e que tenha o menor peso.

Problema de Decisão do Caixeiro Viajante

Dado um grafo completo ponderado $G = (V, E)$ e um parâmetro k, existe um ciclo Hamiltoniano que passa por todos os vértices e que tenha o peso no máximo k?

Exercício: Faça uma redução de Turing que executa em tempo polinomial do Problema do Caixeiro Viajante para o Problema de Decisão do Caixeiro Viajante. Dica: use uma busca binária.

Versão de decisão para problemas de otimização

Problema do Caixeiro Viajante

Dado um grafo completo ponderado $G = (V, E)$, encontre um ciclo Hamiltoniano que passa por todos os vértices e que tenha o menor peso.

Problema de Decisão do Caixeiro Viajante

Dado um grafo completo ponderado $G = (V, E)$ e um parâmetro k, existe um ciclo Hamiltoniano que passa por todos os vértices e que tenha o peso no máximo k?

Exercício: Faça uma redução de Turing que executa em tempo polinomial do Problema do Caixeiro Viajante para o Problema de Decisão do Caixeiro Viajante. Dica: use uma busca binária.

Versão de decisão para problemas de otimização

Problema do Caixeiro Viajante

Dado um grafo completo ponderado $G = (V, E)$, encontre um ciclo Hamiltoniano que passa por todos os vértices e que tenha o menor peso.

Problema de Decisão do Caixeiro Viajante

Dado um grafo completo ponderado $G = (V, E)$ e um parâmetro k, existe um ciclo Hamiltoniano que passa por todos os vértices e que tenha o peso no máximo k?

Exercício: Faça uma redução de Turing que executa em tempo polinomial do Problema do Caixeiro Viajante para o Problema de Decisão do Caixeiro Viajante. Dica: use uma busca binária.

Olhando para frente: algumas classes de complexidade

Informalmente, as classes P, NP, NP-Completo, NP-Difícil são conjuntos de problemas de decisão em que

- **P:** problemas que podem ser resolvidos em tempo polinomial
- **NP:** problemas que possuem um certificado curto que pode ser verificado em tempo polinomial
- **NP-Difícil:** problemas pelo menos tão difíceis quanto qualquer outro em NP: qualquer problema em NP pode ser redutível a um problema em NP-Difícil em tempo polinomial
- **NP-Completo:** problemas que estão tanto em NP como em NP-difícil

Exemplos de problemas em P

- Soma de elemento: Dado um conjunto de números S, existe $n \in S$ tal que $n = \sum_{m \in S \setminus \{n\}} m$?
- Conexidade: Dado grafo G, G é conexo ?
- Caminho Mínimo: Dado grafo completo $G = (V, E)$ com pesos nas arestas $w : E \to \mathbb{Z}^+$, vértices s e t e um inteiro positivo k, existe um caminho em G, de s para t, de peso no máximo k?
- 4-Coloração: Existe uma coloração de um mapa com 4 cores, de forma que duas regiões vizinhas não tenham a mesma cor?

Exemplos de problemas em NP-Completo

- Bipartição: Dado um conjunto de números S, existe $N \subseteq S$ tal que $\sum_{m \in N} m = \sum_{m \in S \setminus N} m$?
- Ciclo hamiltoniano: Dado grafo G, existe um ciclo hamiltoniano em G?
- Caixeiro viajante: Dado grafo completo $G = (V, E)$ com pesos nas arestas $c : E \to \mathbb{Z}^*$ e um inteiro positivo k, existe um ciclo hamiltoniano em G, de peso no máximo k?
- Caminho mais longo: Dado grafo completo $G = (V, E)$ com 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?
- 3-Coloração: Existe uma coloração de um mapa com 3 cores, de forma que duas regiões vizinhas não tenham a mesma cor?

Vários outros problemas NP-difíceis

Vários problemas importantes na indústria também são NP-Difícil!:

- 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
- Muitos outros...

Vários outros problemas NP-difíceis

Vários problemas importantes na indústria também são NP-Difícil!:

- 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
- Muitos outros...

Sutilizas?

- *Caminho mais curto* está em P
- *4-Coloração* está em P

Será que *Caminho mais longo* e *3-Coloração* também estão em P?

$$\text{Será } P = NP \text{ ?}$$

Conjecturamos que não!

- A classe NP contém muito problemas difíceis
- Se houver um algoritmo de tempo polinomial para um problema NP-completo, então também existe algoritmo polinomial para todo problema em NP
- Isso é, achamos que $P \cap NP\text{-Completo} = \emptyset$

Sutilizas?

- *Caminho mais curto* está em P
- *4-Coloração* está em P

Será que *Caminho mais longo* e *3-Coloração* também estão em P?

$$\text{Será } P = NP \text{ ?}$$

Conjecturamos que não!

- A classe NP contém muito problemas difíceis
- Se houver um algoritmo de tempo polinomial para um problema NP-completo, então também existe algoritmo polinomial para todo problema em NP
- Isso é, achamos que $P \cap NP\text{-Completo} = \emptyset$

Sutilizas?

- *Caminho mais curto* está em P
- *4-Coloração* está em P

Será que *Caminho mais longo* e *3-Coloração* também estão em P?

Será $P = NP$?

Conjecturamos que não!

- A classe NP contém muito problemas difíceis
- Se houver um algoritmo de tempo polinomial para um problema NP-completo, então também existe algoritmo polinomial para todo problema em NP
- Isso é, achamos que $P \cap NP\text{-Completo} = \emptyset$

Sutilizas?

- *Caminho mais curto* está em P
- *4-Coloração* está em P

Será que *Caminho mais longo* e *3-Coloração* também estão em P?

$$\text{Será } P = NP \text{ ?}$$

Conjecturamos que não!

- A classe NP contém muito problemas difíceis
- Se houver um algoritmo de tempo polinomial para um problema NP-completo, então também existe algoritmo polinomial para todo problema em NP
- Isso é, achamos que $P \cap NP\text{-Completo} = \emptyset$

Sutilizas?

- *Caminho mais curto* está em P
- *4-Coloração* está em P

Será que *Caminho mais longo* e *3-Coloração* também estão em P?

$$\text{Será } P = NP \text{ ?}$$

Conjecturamos que não!

- A classe NP contém muito problemas difíceis
- Se houver um algoritmo de tempo polinomial para um problema NP-completo, então também existe algoritmo polinomial para todo problema em NP
- Isso é, achamos que $P \cap NP\text{-Completo} = \emptyset$

Sutilizas?

- *Caminho mais curto* está em P
- *4-Coloração* está em P

Será que *Caminho mais longo* e *3-Coloração* também estão em P?

$$\text{Será } P = NP \text{ ?}$$

Conjecturamos que não!

- A classe NP contém muito problemas difíceis
- Se houver um algoritmo de tempo polinomial para um problema NP-completo, então também existe algoritmo polinomial para todo problema em NP
- Isso é, achamos que $P \cap NP\text{-Completo} = \emptyset$

Sutilizas?

- *Caminho mais curto* está em P
- *4-Coloração* está em P

Será que *Caminho mais longo* e *3-Coloração* também estão em P?

$$\text{Será } P = NP \text{ ?}$$

Conjecturamos que não!

- A classe NP contém muito problemas difíceis
- Se houver um algoritmo de tempo polinomial para um problema NP-completo, então também existe algoritmo polinomial para todo problema em NP
- Isso é, achamos que $P \cap NP\text{-Completo} = \emptyset$

Sutilizas?

- *Caminho mais curto* está em P
- *4-Coloração* está em P

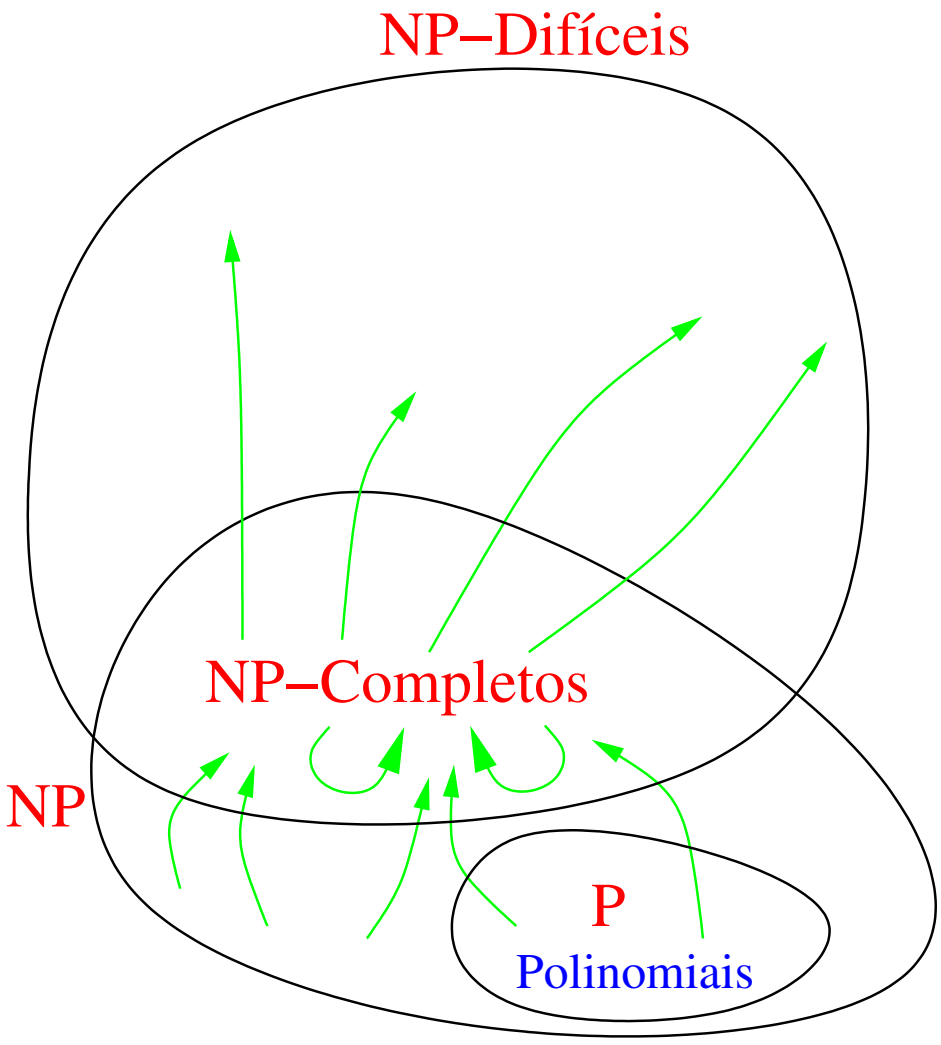
Será que *Caminho mais longo* e *3-Coloração* também estão em P?

$$\text{Será } P = NP \text{ ?}$$

Conjecturamos que não!

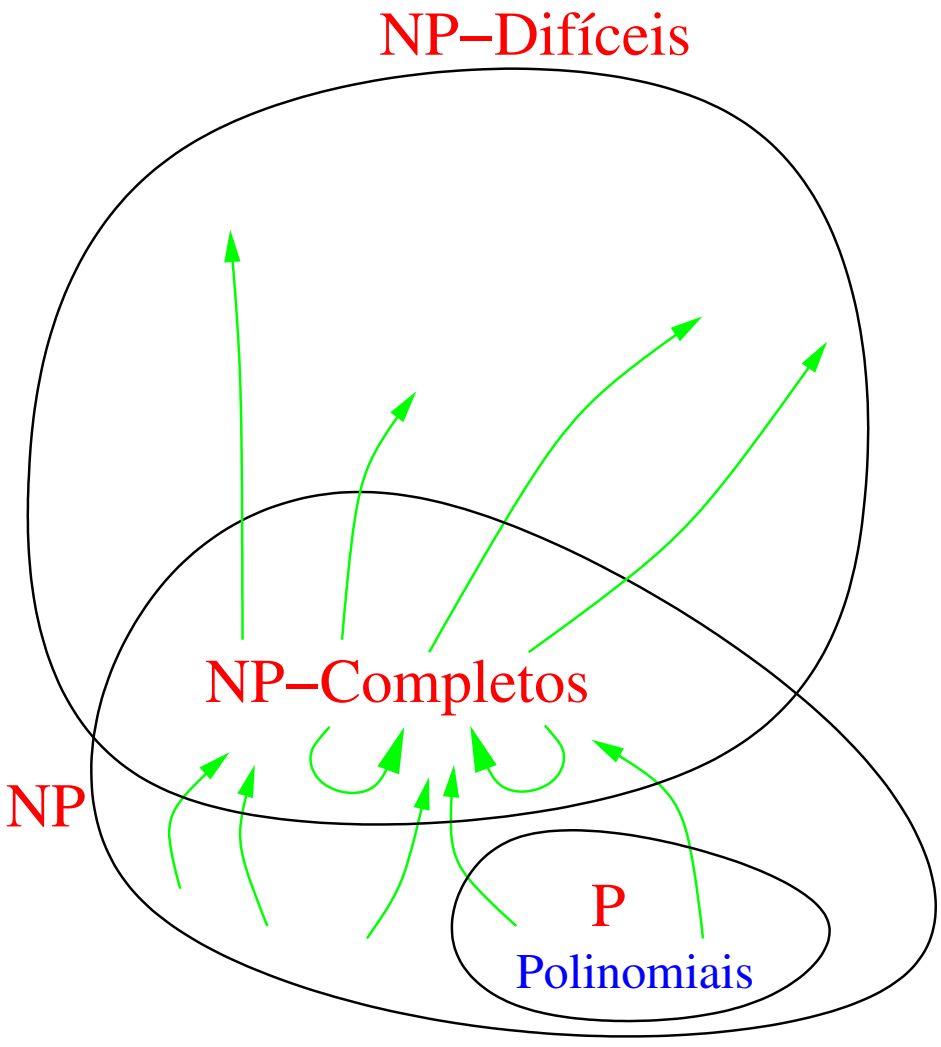
- A classe NP contém muito problemas difíceis
- Se houver um algoritmo de tempo polinomial para um problema NP-completo, então também existe algoritmo polinomial para todo problema em NP
- Isso é, achamos que $P \cap NP\text{-Completo} = \emptyset$

Possível configuração



No restante do curso, queremos entender essa figura!

Possível configuração



No restante do curso, queremos entender essa figura!

Por que saber se um problema é NP-Difícil?

- Acreditamos que não há algoritmo rápido para problemas NP-Difícil
- Não temos certeza, mas
  - o primeiro problema NP-completo conhecido data da década de 1970s (Cook-Levin)
  - bem antes disso, vários destes 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-completo preferidos, mas nenhum deles obteve algoritmo polinomial
  - basta que um problema NP-difícil tenha tempo polinomial para que **todos** problemas em NP tenham algoritmo de tempo polinomial
- Considerando tal dificuldade podemos nos concentrar em métodos adequados para tratá-los

Por que saber se um problema é NP-Difícil?

- Acreditamos que não há algoritmo rápido para problemas NP-Difícil
- Não temos certeza, mas
  - o primeiro problema NP-completo conhecido data da década de 1970s (Cook-Levin)
  - bem antes disso, vários destes 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-completo preferidos, mas nenhum deles obteve algoritmo polinomial
  - basta que um problema NP-difícil tenha tempo polinomial para que **todos** problemas em NP tenham algoritmo de tempo polinomial
- Considerando tal dificuldade podemos nos concentrar em métodos adequados para tratá-los

Por que saber se um problema é NP-Difícil?

- Acreditamos que não há algoritmo rápido para problemas NP-Difícil
- Não temos certeza, mas
  - o primeiro problema NP-completo conhecido data da década de 1970s (Cook-Levin)
  - bem antes disso, vários destes 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-completo preferidos, mas nenhum deles obteve algoritmo polinomial
  - basta que um problema NP-difícil tenha tempo polinomial para que **todos** problemas em NP tenham algoritmo de tempo polinomial
- Considerando tal dificuldade podemos nos concentrar em métodos adequados para tratá-los

Por que saber se um problema é NP-Difícil?

- Acreditamos que não há algoritmo rápido para problemas NP-Difícil
- Não temos certeza, mas
  - o primeiro problema NP-completo conhecido data da década de 1970s (Cook-Levin)
  - bem antes disso, vários destes 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-completo preferidos, mas nenhum deles obteve algoritmo polinomial
  - basta que um problema NP-difícil tenha tempo polinomial para que **todos** problemas em NP tenham algoritmo de tempo polinomial
- Considerando tal dificuldade podemos nos concentrar em métodos adequados para tratá-los

Por que saber se um problema é NP-Difícil?

- Acreditamos que não há algoritmo rápido para problemas NP-Difícil
- Não temos certeza, mas
  - o primeiro problema NP-completo conhecido data da década de 1970s (Cook-Levin)
  - bem antes disso, vários destes 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-completo preferidos, mas nenhum deles obteve algoritmo polinomial
  - basta que um problema NP-difícil tenha tempo polinomial para que **todos** problemas em NP tenham algoritmo de tempo polinomial
- Considerando tal dificuldade podemos nos concentrar em métodos adequados para tratá-los

Por que saber se um problema é NP-Difícil?

- Acreditamos que não há algoritmo rápido para problemas NP-Difícil
- Não temos certeza, mas
  - o primeiro problema NP-completo conhecido data da década de 1970s (Cook-Levin)
  - bem antes disso, vários destes 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-completo preferidos, mas nenhum deles obteve algoritmo polinomial
  - basta que um problema NP-difícil tenha tempo polinomial para que **todos** problemas em NP tenham algoritmo de tempo polinomial
- Considerando tal dificuldade podemos nos concentrar em métodos adequados para tratá-los

Por que saber se um problema é NP-Difícil?

- Acreditamos que não há algoritmo rápido para problemas NP-Difícil
- Não temos certeza, mas
  - o primeiro problema NP-completo conhecido data da década de 1970s (Cook-Levin)
  - bem antes disso, vários destes 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-completo preferidos, mas nenhum deles obteve algoritmo polinomial
  - basta que um problema NP-difícil tenha tempo polinomial para que **todos** problemas em NP tenham algoritmo de tempo polinomial
- Considerando tal dificuldade podemos nos concentrar em métodos adequados para tratá-los

Por que saber se um problema é NP-Difícil?

- Acreditamos que não há algoritmo rápido para problemas NP-Difícil
- Não temos certeza, mas
  - o primeiro problema NP-completo conhecido data da década de 1970s (Cook-Levin)
  - bem antes disso, vários destes 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-completo preferidos, mas nenhum deles obteve algoritmo polinomial
  - basta que um problema NP-difícil tenha tempo polinomial para que **todos** problemas em NP tenham algoritmo de tempo polinomial
- Considerando tal dificuldade podemos nos concentrar em métodos adequados para tratá-los

O que fazer se um problema for NP-Difícil?

Dado um problema NP-difícil, podemos concentrar os recursos em busca de

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

Antes, devemos identificar que problemas NP-difícil!

O que fazer se um problema for NP-Difícil?

Dado um problema NP-difícil, podemos concentrar os recursos em busca de

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

Antes, devemos identificar que problemas NP-difícil!

O que fazer se um problema for NP-Difícil?

Dado um problema NP-difícil, podemos concentrar os recursos em busca de

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

Antes, devemos identificar que problemas NP-difícil!

O que fazer se um problema for NP-Difícil?

Dado um problema NP-difícil, podemos concentrar os recursos em busca de

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

Antes, devemos identificar que problemas NP-difícil!

O que fazer se um problema for NP-Difícil?

Dado um problema NP-difícil, podemos concentrar os recursos em busca de

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

Antes, devemos identificar que problemas NP-difícil!

O que fazer se um problema for NP-Difícil?

Dado um problema NP-difícil, podemos concentrar os recursos em busca de

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

Antes, devemos identificar que problemas NP-difícil!

Linguagens Formais

Linguagens Formais

- 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 sobre Σ é denotado por Σ^*, que inclui a sequência vazia, ε.

Linguagem Formal

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

Exemplos:

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

Linguagens Formais

- 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 sobre Σ é denotado por Σ^*, que inclui a sequência vazia, ε.

Linguagem Formal

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

Exemplos:

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

Linguagens Formais

- 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 sobre Σ é denotado por Σ^*, que inclui a sequência vazia, ε.

Linguagem Formal

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

Exemplos:

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

Linguagens Formais

- 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 sobre Σ é denotado por Σ^*, que inclui a sequência vazia, ε.

Linguagem Formal

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

Exemplos:

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

Linguagens Formais

- 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 sobre Σ é denotado por Σ^*, que inclui a sequência vazia, ε.

Linguagem Formal

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

Exemplos:

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

Representando Problemas de Decisão

- Em um computador, o alfabeto é $\{0, 1\}$.
- A memória pode conter qualquer sequência de bits $I \in \{0, 1\}^*$
- Um problema de decisão é uma função $Q : \{0, 1\}^* \to \{0, 1\}$

Estamos interessados apenas nas sequências cuja resposta é sim!

Linguagem de um problema

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

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

Iremos identificar o problema Q com linguagem L correspondente!

Representando Problemas de Decisão

- Em um computador, o alfabeto é $\{0, 1\}$.
- A memória pode conter qualquer sequência de bits $I \in \{0, 1\}^*$
- Um problema de decisão é uma função $Q : \{0, 1\}^* \to \{0, 1\}$

Estamos interessados apenas nas sequências cuja resposta é sim!

Linguagem de um problema

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

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

Iremos identificar o problema Q com linguagem L correspondente!

Representando Problemas de Decisão

- Em um computador, o alfabeto é $\{0,1\}$.
- A memória pode conter qualquer sequência de bits $I \in \{0,1\}^*$
- Um problema de decisão é uma função $Q : \{0,1\}^* \to \{0,1\}$

Estamos interessados apenas nas sequências cuja resposta é sim!

Linguagem de um problema

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

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

Iremos identificar o problema Q com linguagem L correspondente!

Representando Problemas de Decisão

- Em um computador, o alfabeto é $\{0, 1\}$.
- A memória pode conter qualquer sequência de bits $I \in \{0, 1\}^*$
- Um problema de decisão é uma função $Q : \{0, 1\}^* \to \{0, 1\}$

Estamos interessados apenas nas sequências cuja resposta é sim!

Linguagem de um problema

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

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

Iremos identificar o problema Q com linguagem L correspondente!

Representando Problemas de Decisão

- Em um computador, o alfabeto é $\{0, 1\}$.
- A memória pode conter qualquer sequência de bits $I \in \{0, 1\}^*$
- Um problema de decisão é uma função $Q : \{0, 1\}^* \to \{0, 1\}$

Estamos interessados apenas nas sequências cuja resposta é sim!

Linguagem de um problema

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

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

Iremos identificar o problema Q com linguagem L correspondente!

Representando Problemas de Decisão

- Em um computador, o alfabeto é $\{0, 1\}$.
- A memória pode conter qualquer sequência de bits $I \in \{0, 1\}^*$
- Um problema de decisão é uma função $Q : \{0, 1\}^* \to \{0, 1\}$

Estamos interessados apenas nas sequências cuja resposta é sim!

Linguagem de um problema

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

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

Iremos identificar o problema Q com linguagem L correspondente!

Codificação

- Uma codificação é uma forma definida de representar um objeto como uma palavra de Σ^*
- Nos computadores, sempre usamos sequência de bits $\{0,1\}$
- Dado um objeto A, denotamos a codificação por $\langle A \rangle$

PATH

O problema de decidir se há um caminho de u a v com k arestas é

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

Exercício: Apresente uma codificação binária e compacta da linguagem PATH em de bits.

Codificação

- Uma codificação é uma forma definida de representar um objeto como uma palavra de Σ^*
- Nos computadores, sempre usamos sequência de bits $\{0,1\}$
- Dado um objeto A, denotamos a codificação por $\langle A \rangle$

PATH

O problema de decidir se há um caminho de u a v com k arestas é

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

Exercício: Apresente uma codificação binária e compacta da linguagem PATH em de bits.

Codificação

- Uma codificação é uma forma definida de representar um objeto como uma palavra de Σ^*
- Nos computadores, sempre usamos sequência de bits $\{0,1\}$
- Dado um objeto A, denotamos a codificação por $\langle A \rangle$

PATH

O problema de decidir se há um caminho de u a v com k arestas é

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

Exercício: Apresente uma codificação binária e compacta da linguagem PATH em de bits.

Codificação

- Uma codificação é uma forma definida de representar um objeto como uma palavra de Σ^*
- Nos computadores, sempre usamos sequência de bits $\{0, 1\}$
- Dado um objeto A, denotamos a codificação por $\langle A \rangle$

PATH

O problema de decidir se há um caminho de u a v com k arestas é

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

Exercício: Apresente uma codificação binária e compacta da linguagem PATH em de bits.

Codificação

- Uma codificação é uma forma definida de representar um objeto como uma palavra de Σ^*
- Nos computadores, sempre usamos sequência de bits $\{0, 1\}$
- Dado um objeto A, denotamos a codificação por $\langle A \rangle$

PATH

O problema de decidir se há um caminho de u a v com k arestas é

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

Exercício: Apresente uma codificação binária e compacta da linguagem PATH em de bits.

Codificação

- Uma codificação é uma forma definida de representar um objeto como uma palavra de Σ^*
- Nos computadores, sempre usamos sequência de bits $\{0, 1\}$
- Dado um objeto A, denotamos a codificação por $\langle A \rangle$

PATH

O problema de decidir se há um caminho de u a v com k arestas é

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

Exercício: Apresente uma codificação binária e compacta da linguagem PATH em de bits.

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Tamanho da instância

- O tamanho de uma instância $x \in \{0,1\}^*$, denotado por $n = |x|$, é o número de bits de x.
- Se referimos a uma instância como um objeto de alto nível A, então o tamanho é $n = |\langle A \rangle|$

Exemplo

Considere o problema PARES $= \{\langle k \rangle \mid k \text{ é par}\}$

- Em codificação **unária**, o número $k = 100$ seria $\langle 100 \rangle = \overbrace{111\ldots111}^{100\times}$
  - nesse caso, $n = |\langle 100 \rangle| = |111\ldots1111| = 100$
  - em geral, $n = |\langle k \rangle| = k = \Theta(k)$
- Em codificação **binária**, o número $k = 100$ seria $\langle 100 \rangle = 1100100$
  - nesse caso, $n = |\langle 100 \rangle| = |1100100| = 7$
  - em geral caso, $n = |\langle k \rangle| = \lceil \log_2 k \rceil = \Theta(\log k)$
- Em codificação ternária etc. também $n = |\langle k \rangle| = \Theta(\log k)$

Linguagem aceita em tempo polinomial

Linguagem aceita em tempo polinomial

- Linguagem aceita

Linguagem aceita

Linguagem aceita

A linguagem aceita por um algoritmo A é

$$L(A) = \{x \in \{0,1\}^* \mid A(x) \text{ termina e devolve } A(x) = 1\}.$$

- Um algoritmo A aceita um problema L se $L = L(A)$
- Um algoritmo A decide L se todo $x \in \{0,1\}^*$:
  - $x \in L$, então $A(x) = 1$ (A aceita x)
  - $x \notin L$, então $A(x) = 0$ (A rejeita x)

Linguagem aceita

Linguagem aceita

A linguagem aceita por um algoritmo A é

$$L(A) = \{x \in \{0,1\}^* \mid A(x) \text{ termina e devolve } A(x) = 1\}.$$

- Um algoritmo A aceita um problema L se $L = L(A)$
- Um algoritmo A decide L se todo $x \in \{0,1\}^*$:
  - $x \in L$, então $A(x) = 1$ (A aceita x)
  - $x \notin L$, então $A(x) = 0$ (A rejeita x)

Linguagem aceita

Linguagem aceita

A linguagem aceita por um algoritmo A é

$$L(A) = \{x \in \{0,1\}^* \mid A(x) \text{ termina e devolve } A(x) = 1\}.$$

- Um algoritmo A aceita um problema L se $L = L(A)$
- Um algoritmo A decide L se todo $x \in \{0,1\}^*$:
  - $x \in L$, então $A(x) = 1$ (A aceita x)
  - $x \notin L$, então $A(x) = 0$ (A rejeita x)

Linguagem aceita

Linguagem aceita

A linguagem aceita por um algoritmo A é

$$L(A) = \{x \in \{0,1\}^* \mid A(x) \text{ termina e devolve } A(x) = 1\}.$$

- Um algoritmo A aceita um problema L se $L = L(A)$
- Um algoritmo A decide L se todo $x \in \{0,1\}^*$:
  - $x \in L$, então $A(x) = 1$ (A aceita x)
  - $x \notin L$, então $A(x) = 0$ (A rejeita x)

Linguagem aceita

Linguagem aceita

A linguagem aceita por um algoritmo A é

$$L(A) = \{x \in \{0,1\}^* \mid A(x) \text{ termina e devolve } A(x) = 1\}.$$

- Um algoritmo A aceita um problema L se $L = L(A)$
- Um algoritmo A decide L se todo $x \in \{0,1\}^*$:
  - $x \in L$, então $A(x) = 1$ (A aceita x)
  - $x \notin L$, então $A(x) = 0$ (A rejeita x)

Linguagem aceita em tempo polinomial

- A classe P

Classe P

Classe P

$$P = \{L \subseteq \{0,1\}^* : \quad \text{existe um algoritmo } A \text{ que decide } L \text{ em tempo polinomial}\}$$

Em outras palavras, P é o conjunto de problemas que podem ser decididos por algoritmos que terminam em tempo polinomial.

Classe P

Classe P

$$P = \{L \subseteq \{0,1\}^* : \quad \text{existe um algoritmo } A \text{ que decide } L \text{ em tempo polinomial}\}$$

Em outras palavras, P é o conjunto de problemas que podem ser decididos por algoritmos que terminam em tempo polinomial.

Classe P

Teorema

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

Prova.

- $\subseteq$ Seja $L \in P$. Pela definição de P, existe um algoritmo que decide L, logo também existe um algoritmo que aceita L.
- $\supseteq$ Seja L uma linguagem ACEITA por um algoritmo A em tempo polinomial n^k, para uma constante k. Construa um algoritmo A' que DECIDE L em tempo polinomial: Dado $x \in L$ de tamanho n, sabemos que n^k é um tempo limite para a execução de A. O algoritmo A' simula o algoritmo A executando os mesmos passos que este. Após um tempo n^k, A' checa se A aceitou x. Caso positivo, A' aceita x. Caso contrário A' rejeita x.

□

Classe P

Teorema

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

Prova.

- $\subseteq$ Seja $L \in P$. Pela definição de P, existe um algoritmo que decide L, logo também existe um algoritmo que aceita L.
- $\supseteq$ Seja L uma linguagem ACEITA por um algoritmo A em tempo polinomial n^k, para uma constante k. Construa um algoritmo A' que DECIDE L em tempo polinomial: Dado $x \in L$ de tamanho n, sabemos que n^k é um tempo limite para a execução de A. O algoritmo A' simula o algoritmo A executando os mesmos passos que este. Após um tempo n^k, A' checa se A aceitou x. Caso positivo, A' aceita x. Caso contrário A' rejeita x.

□

Classe P

Teorema

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

Prova.

- $\subseteq$ Seja $L \in P$. Pela definição de P, existe um algoritmo que decide L, logo também existe um algoritmo que aceita L.
- $\supseteq$ Seja L uma linguagem ACEITA por um algoritmo A em tempo polinomial n^k, para uma constante k. Construa um algoritmo A' que DECIDE L em tempo polinomial: Dado $x \in L$ de tamanho n, sabemos que n^k é um tempo limite para a execução de A. O algoritmo A' simula o algoritmo A executando os mesmos passos que este. Após um tempo n^k, A' checa se A aceitou x. Caso positivo, A' aceita x. Caso contrário A' rejeita x.

□

Classe P

Teorema

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

Prova.

- $\subseteq$ Seja $L \in P$. Pela definição de P, existe um algoritmo que decide L, logo também existe um algoritmo que aceita L.
- $\supseteq$ Seja L uma linguagem ACEITA por um algoritmo A em tempo polinomial n^k, para uma constante k. Construa um algoritmo A' que DECIDE L em tempo polinomial: Dado $x \in L$ de tamanho n, sabemos que n^k é um tempo limite para a execução de A. O algoritmo A' simula o algoritmo A executando os mesmos passos que este. Após um tempo n^k, A' checa se A aceitou x. Caso positivo, A' aceita x. Caso contrário A' rejeita x.

□

Linguagem verificada em tempo polinomial

Linguagem verificada em tempo polinomial

- Verificação

Certificado

Queremos encontrar estruturas que possam certificar que uma instância tem resposta **sim**.

Certificado para PATH

- Considere uma instância $I = \langle G, u, v, k \rangle$ para PATH
  - Se a resposta é **sim**, então existe caminho P de u até v de tamanho no máximo k.
  - Se a resposta é **não**, então não existe caminho P de u até v de tamanho no máximo k.
- Neste caso, dizemos que P é um *certificado* para I.

O certificado deve ser curto, isso é, de tamanho polinomial.

Certificado

Queremos encontrar estruturas que possam certificar que uma instância tem resposta **sim**.

Certificado para PATH

- Considere uma instância $I = \langle G, u, v, k \rangle$ para PATH
  - Se a resposta é **sim**, então existe caminho P de u até v de tamanho no máximo k.
  - Se a resposta é **não**, então não existe caminho P de u até v de tamanho no máximo k.
- Neste caso, dizemos que P é um *certificado* para I.

O certificado deve ser curto, isso é, de tamanho polinomial.

Certificado

Queremos encontrar estruturas que possam certificar que uma instância tem resposta **sim**.

Certificado para PATH

- Considere uma instância $I = \langle G, u, v, k \rangle$ para PATH
  - Se a resposta é **sim**, então existe caminho P de u até v de tamanho no máximo k.
  - Se a resposta é **não**, então não existe caminho P de u até v de tamanho no máximo k.
- Neste caso, dizemos que P é um *certificado* para I.

O certificado deve ser curto, isso é, de tamanho polinomial.

Certificado

Queremos encontrar estruturas que possam certificar que uma instância tem resposta **sim**.

Certificado para PATH

- Considere uma instância $I = \langle G, u, v, k \rangle$ para PATH
  - Se a resposta é **sim**, então existe caminho P de u até v de tamanho no máximo k.
  - Se a resposta é **não**, então não existe caminho P de u até v de tamanho no máximo k.
- Neste caso, dizemos que P é um *certificado* para I.

O certificado deve ser curto, isso é, de tamanho polinomial.

Certificado

Queremos encontrar estruturas que possam certificar que uma instância tem resposta **sim**.

Certificado para PATH

- Considere uma instância $I = \langle G, u, v, k \rangle$ para PATH
  - Se a resposta é **sim**, então existe caminho P de u até v de tamanho no máximo k.
  - Se a resposta é **não**, então não existe caminho P de u até v de tamanho no máximo k.
- Neste caso, dizemos que P é um *certificado* para I.

O certificado deve ser curto, isso é, de tamanho polinomial.

Certificado

Queremos encontrar estruturas que possam certificar que uma instância tem resposta **sim**.

Certificado para PATH

- Considere uma instância $I = \langle G, u, v, k \rangle$ para PATH
  - Se a resposta é **sim**, então existe caminho P de u até v de tamanho no máximo k.
  - Se a resposta é **não**, então não existe caminho P de u até v de tamanho no máximo k.
- Neste caso, dizemos que P é um *certificado* para I.

O certificado deve ser curto, isso é, de tamanho polinomial.

Verificador

Também devemos verificar se um certificado é válido para uma instância rapidamente.

Verificador para PATH

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

1. $\ell \leftarrow 0$
2. Para cada aresta $(u, v) \in P$:
   - Se $(u, v) \notin E[G]$, devolva **não**
   - $\ell \leftarrow \ell + 1$
3. Se $\ell \leq k$, devolva **sim**
4. Senão devolva **não**

O verificador deve executar em tempo polinomial.

Verificador

Também devemos verificar se um certificado é válido para uma instância rapidamente.

Verificador para PATH

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

1. $\ell \leftarrow 0$
2. Para cada aresta $(u, v) \in P$:
   - Se $(u, v) \notin E[G]$, devolva **não**
   - $\ell \leftarrow \ell + 1$
3. Se $\ell \leq k$, devolva **sim**
4. Senão devolva **não**

O verificador deve executar em tempo polinomial.

Verificador

Também devemos verificar se um certificado é válido para uma instância rapidamente.

Verificador para PATH

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

1. $\ell \leftarrow 0$
2. Para cada aresta $(u, v) \in P$:
   - Se $(u, v) \notin E[G]$, devolva **não**
   - $\ell \leftarrow \ell + 1$
3. Se $\ell \leq k$, devolva **sim**
4. Senão devolva **não**

O verificador deve executar em tempo polinomial.

Caminho hamiltoniano

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

Problema do ciclo hamiltoniano

Dado grafo $G = (V, E)$, exite um ciclo hamiltoniano em G?

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

Certificado natural: uma sequência de vértices C de um ciclo

Caminho hamiltoniano

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

Problema do ciclo hamiltoniano

Dado grafo $G = (V, E)$, exite um ciclo hamiltoniano em G?

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

Certificado natural: uma sequência de vértices C de um ciclo

Caminho hamiltoniano

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

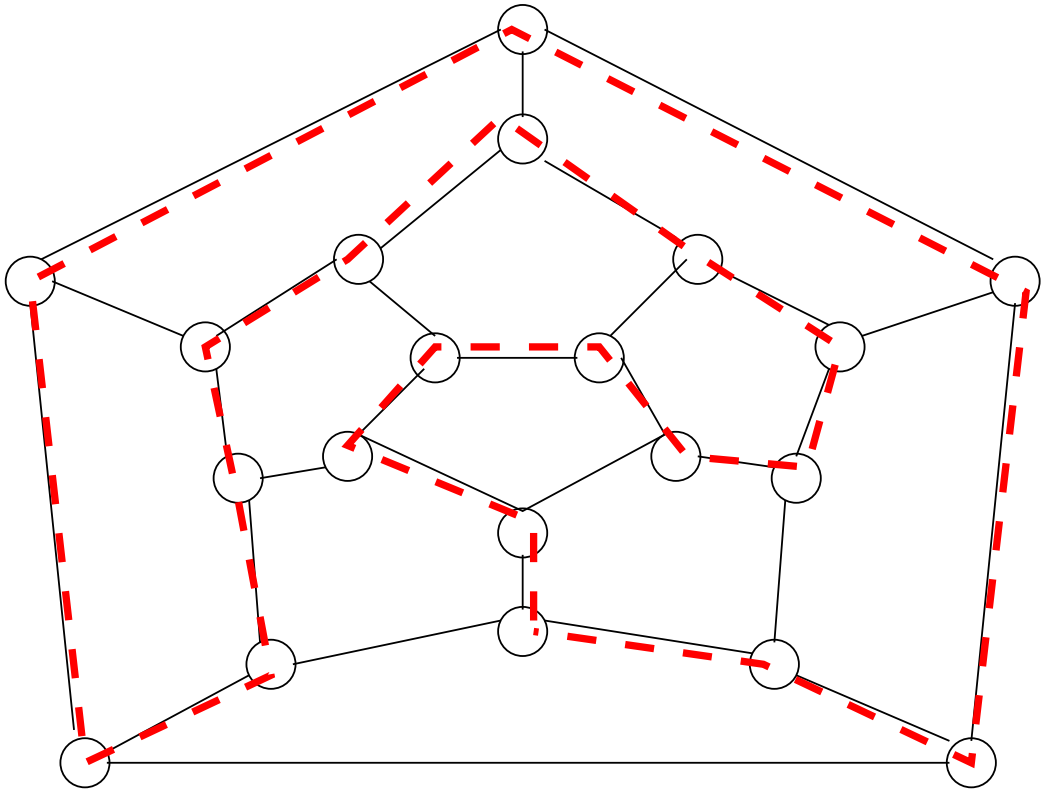
Problema do ciclo hamiltoniano

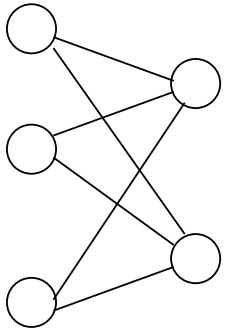
Dado grafo $G = (V, E)$, exite um ciclo hamiltoniano em G?

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

Certificado natural: uma sequência de vértices C de um ciclo

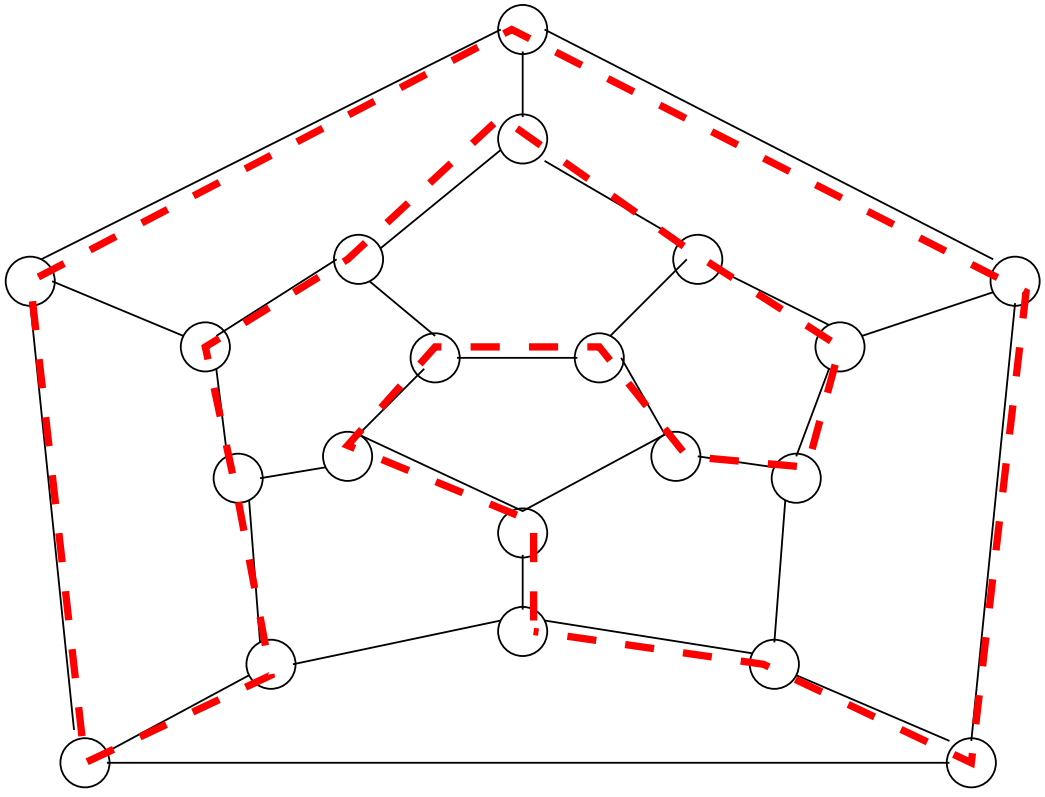
Exemplo de caminho hamiltoniano

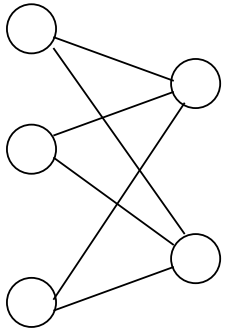




O grafo da direita não tem ciclo hamiltoniano!

Exemplo de caminho hamiltoniano





O grafo da direita não tem ciclo hamiltoniano!

Chutando um certificado

- Podemos descobrir deterministicamente se há um ciclo:
  - o algoritmo trivial gasta tempo $O(V!)$
  - os melhores algoritmos têm tempo $O(n^2 2^n)$

- Mas se alguém adivinhar um ciclo C, é fácil verificar se C é de fato hamiltoniano em tempo polinomial

Verifica um chute C

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

1. $S \leftarrow \emptyset$
2. Para cada aresta $(u, v) \in C$:
   - Se $(u, v) \notin E[G]$, devolva **não**
   - $S \leftarrow S \cup \{v\}$
3. Se $S = V$, devolva **sim**
4. Senão devolva **não**

Chutando um certificado

- Podemos descobrir deterministicamente se há um ciclo:
  - o algoritmo trivial gasta tempo $O(V!)$
  - os melhores algoritmos têm tempo $O(n^2 2^n)$
- Mas se alguém adivinhar um ciclo C, é fácil verificar se C é de fato hamiltoniano em tempo polinomial

Verifica um chute C

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

1. $S \leftarrow \emptyset$
2. Para cada aresta $(u, v) \in C$:
   - Se $(u, v) \notin E[G]$, devolva **não**
   - $S \leftarrow S \cup \{v\}$
3. Se $S = V$, devolva **sim**
4. Senão devolva **não**

Chutando um certificado

- Podemos descobrir deterministicamente se há um ciclo:
  - o algoritmo trivial gasta tempo $O(V!)$
  - os melhores algoritmos têm tempo $O(n^2 2^n)$

- Mas se alguém adivinhar um ciclo C, é fácil verificar se C é de fato hamiltoniano em tempo polinomial

Verifica um chute C

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

1. $S \leftarrow \emptyset$
2. Para cada aresta $(u, v) \in C$:
   - Se $(u, v) \notin E[G]$, devolva **não**
   - $S \leftarrow S \cup \{v\}$
3. Se $S = V$, devolva **sim**
4. Senão devolva **não**

Chutando um certificado

- Podemos descobrir deterministicamente se há um ciclo:
  - o algoritmo trivial gasta tempo $O(V!)$
  - os melhores algoritmos têm tempo $O(n^2 2^n)$
- Mas se alguém adivinhar um ciclo C, é fácil verificar se C é de fato hamiltoniano em tempo polinomial

Verifica um chute C

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

1. $S \leftarrow \emptyset$
2. Para cada aresta $(u, v) \in C$:
   - Se $(u, v) \notin E[G]$, devolva **não**
   - $S \leftarrow S \cup \{v\}$
3. Se $S = V$, devolva **sim**
4. Senão devolva **não**

Chutando um certificado

- Podemos descobrir deterministicamente se há um ciclo:
  - o algoritmo trivial gasta tempo $O(V!)$
  - os melhores algoritmos têm tempo $O(n^2 2^n)$
- Mas se alguém adivinhar um ciclo C, é fácil verificar se C é de fato hamiltoniano em tempo polinomial

Verifica um chute C

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

1. $S \leftarrow \emptyset$
2. Para cada aresta $(u, v) \in C$:
   - Se $(u, v) \notin E[G]$, devolva **não**
   - $S \leftarrow S \cup \{v\}$
3. Se $S = V$, devolva **sim**
4. Senão devolva **não**

Formalizando: certificado e verificador

Certificado

- Um certificado para um problema P é qualquer sequência $y \in \{0,1\}^*$ de tamanho no máximo $|y| \leq O(|I|^c)$, para alguma constante c
- A constante c depende somente do problema!

Verificador

- Um verificador é um algoritmo com dois argumentos $A(x, y)$ e termina devolvendo 0 ou 1, ou não termina.
- Os argumentos a serem passados são:
  - x, uma instância de um problema P
  - y, um certificado para a instância x

Formalizando: certificado e verificador

Certificado

- Um certificado para um problema P é qualquer sequência $y \in \{0,1\}^*$ de tamanho no máximo $|y| \leq O(|I|^c)$, para alguma constante c
- A constante c depende somente do problema!

Verificador

- Um verificador é um algoritmo com dois argumentos $A(x, y)$ e termina devolvendo 0 ou 1, ou não termina.
- Os argumentos a serem passados são:
  - x, uma instância de um problema P
  - y, um certificado para a instância x

Formalizando: certificado e verificador

Certificado

- Um certificado para um problema P é qualquer sequência $y \in \{0,1\}^*$ de tamanho no máximo $|y| \leq O(|I|^c)$, para alguma constante c
- A constante c depende somente do problema!

Verificador

- Um verificador é um algoritmo com dois argumentos $A(x,y)$ e termina devolvendo 0 ou 1, ou não termina.
- Os argumentos a serem passados são:
  - x, uma instância de um problema P
  - y, um certificado para a instância x

Formalizando: certificado e verificador

Certificado

- Um certificado para um problema P é qualquer sequência $y \in \{0,1\}^*$ de tamanho no máximo $|y| \leq O(|I|^c)$, para alguma constante c
- A constante c depende somente do problema!

Verificador

- Um verificador é um algoritmo com dois argumentos $A(x,y)$ e termina devolvendo 0 ou 1, ou não termina.
- Os argumentos a serem passados são:
  - x, uma instância de um problema P
  - y, um certificado para a instância x

Linguagem verificada

Linguagem verificada

A linguagem verificada por um algoritmo verificador A é

$$L(A) = \{x \in \{0,1\}^* \ : \ \begin{array}{l} \text{existe } y \in \{0,1\}^* \text{ com } |y| \leq O(|x|^c) \\ \text{tal que } A(x,y) \text{ termina} \\ \text{e devolve } A(x,y) = 1 \quad \} \end{array}$$

- Um algoritmo A verifica um problema L se $L = L(A)$
- Assim
  - se $x \in L$, **existe** certificado y tal que $A(x,y) = 1$
  - se $x \notin L$, **não existe** certificado y tal que $A(x,y) = 1$

Linguagem verificada

Linguagem verificada

A linguagem verificada por um algoritmo verificador A é

$$L(A) = \{x \in \{0,1\}^* : \begin{array}{l} \text{existe } y \in \{0,1\}^* \text{ com } |y| \leq O(|x|^c) \\ \text{tal que } A(x,y) \text{ termina} \\ \text{e devolve } A(x,y) = 1 \quad \} \end{array}$$

- Um algoritmo A verifica um problema L se $L = L(A)$
- Assim
  - se $x \in L$, **existe** certificado y tal que $A(x,y) = 1$
  - se $x \notin L$, **não existe** certificado y tal que $A(x,y) = 1$

Linguagem verificada

Linguagem verificada

A linguagem verificada por um algoritmo verificador A é

$$L(A) = \{x \in \{0,1\}^* : \begin{array}{l} \text{existe } y \in \{0,1\}^* \text{ com } |y| \leq O(|x|^c) \\ \text{tal que } A(x,y) \text{ termina} \\ \text{e devolve } A(x,y) = 1 \quad \} \end{array}$$

- Um algoritmo A verifica um problema L se $L = L(A)$
- Assim
  - se $x \in L$, **existe** certificado y tal que $A(x,y) = 1$
  - se $x \notin L$, **não existe** certificado y tal que $A(x,y) = 1$

A classe NP

A classe NP

Classe NP

Uma linguagem L pertence a NP se existe algoritmo verificador A que executa em tempo polinomial e $L = L(A)$, i.e.,

$$L = \{x \in \{0,1\}^* \ : \ \begin{array}{l} \text{existe } y \in \{0,1\}^* \text{ com } |y| \leq O(|x|^c) \\ \text{tal que } A(x,y) \text{ termina} \\ \text{e devolve } A(x,y) = 1 \quad \} \end{array}$$

O N em NP vem de não-determinístico!

A classe NP

Classe NP

Uma linguagem L pertence a NP se existe algoritmo verificador A que executa em tempo polinomial e $L = L(A)$, i.e.,

$$L = \{x \in \{0,1\}^* \ : \ \begin{array}{l} \text{existe } y \in \{0,1\}^* \text{ com } |y| \leq O(|x|^c) \\ \text{tal que } A(x,y) \text{ termina} \\ \text{e devolve } A(x,y) = 1 \quad \} \end{array}$$

O N em NP vem de não-determinístico!

Como saber se um problema está em NP

Dado problema L, devemos seguir esses passos:

1. Identificar um certificado de tamanho polinomial para L
2. Construir um algoritmo verificador $A(\cdot, \cdot)$ de tempo polinomial
3. Demonstrar que:
   - Se $x \in L$, então **existe** y tal que $A(x, y) = 1$
   - Se $y \notin L$, então para **qualquer** y, vale $A(x, y) = 0$

Exemplos:

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

Concluímos que PATH, HAM-CICLO $\in NP$

Como saber se um problema está em NP

Dado problema L, devemos seguir esses passos:

1. Identificar um certificado de tamanho polinomial para L
2. Construir um algoritmo verificador $A(\cdot, \cdot)$ de tempo polinomial
3. Demonstrar que:
   - Se $x \in L$, então **existe** y tal que $A(x, y) = 1$
   - Se $y \notin L$, então para **qualquer** y, vale $A(x, y) = 0$

Exemplos:

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

Concluímos que PATH, HAM-CICLO $\in NP$

Como saber se um problema está em NP

Dado problema L, devemos seguir esses passos:

1. Identificar um certificado de tamanho polinomial para L
2. Construir um algoritmo verificador $A(\cdot, \cdot)$ de tempo polinomial
3. Demonstrar que:
   - Se $x \in L$, então **existe** y tal que $A(x, y) = 1$
   - Se $y \notin L$, então para **qualquer** y, vale $A(x, y) = 0$

Exemplos:

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

Concluímos que PATH, HAM-CICLO $\in NP$

Como saber se um problema está em NP

Dado problema L, devemos seguir esses passos:

1. Identificar um certificado de tamanho polinomial para L
2. Construir um algoritmo verificador $A(\cdot, \cdot)$ de tempo polinomial
3. Demonstrar que:
   - Se $x \in L$, então **existe** y tal que $A(x, y) = 1$
   - Se $y \notin L$, então para **qualquer** y, vale $A(x, y) = 0$

Exemplos:

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

Concluímos que PATH, HAM-CICLO $\in NP$

Como saber se um problema está em NP

Dado problema L, devemos seguir esses passos:

1. Identificar um certificado de tamanho polinomial para L
2. Construir um algoritmo verificador $A(\cdot, \cdot)$ de tempo polinomial
3. Demonstrar que:
   - Se $x \in L$, então **existe** y tal que $A(x, y) = 1$
   - Se $y \notin L$, então para **qualquer** y, vale $A(x, y) = 0$

Exemplos:

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

Concluímos que PATH, HAM-CICLO $\in NP$

Como saber se um problema está em NP

Dado problema L, devemos seguir esses passos:

1. Identificar um certificado de tamanho polinomial para L
2. Construir um algoritmo verificador $A(\cdot, \cdot)$ de tempo polinomial
3. Demonstrar que:
   - Se $x \in L$, então **existe** y tal que $A(x, y) = 1$
   - Se $y \notin L$, então para **qualquer** y, vale $A(x, y) = 0$

Exemplos:

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

Concluímos que PATH, HAM-CICLO $\in NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

NP contém P

Seja $L \in P$

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

Verificador para L

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

- O verificador só precisa ignorar o certificado 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$(x, \cdot) = 0$
- Concluímos que $L \in NP$
- Portanto: $P \subseteq NP$

O problema complementar

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

- $\overline{L}$ é o conjunto de instâncias cuja resposta para L é **não**
- Exemplo:

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

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

Classe co-NP

A classe complementar de NP é o conjunto de linguagens

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

Exemplos:

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

O problema complementar

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

- $\overline{L}$ é o conjunto de instâncias cuja resposta para L é não
- Exemplo:

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

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

Classe co-NP

A classe complementar de NP é o conjunto de linguagens

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

Exemplos:

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

O problema complementar

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

- $\overline{L}$ é o conjunto de instâncias cuja resposta para L é **não**
- Exemplo:

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

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

Classe co-NP

A classe complementar de NP é o conjunto de linguagens

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

Exemplos:

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

O problema complementar

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

- $\overline{L}$ é o conjunto de instâncias cuja resposta para L é **não**
- Exemplo:

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

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

Classe co-NP

A classe complementar de NP é o conjunto de linguagens

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

Exemplos:

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

O problema complementar

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

- $\overline{L}$ é o conjunto de instâncias cuja resposta para L é **não**
- Exemplo:

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

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

Classe co-NP

A classe complementar de NP é o conjunto de linguagens

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

Exemplos:

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

O problema complementar

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

- $\overline{L}$ é o conjunto de instâncias cuja resposta para L é **não**
- Exemplo:

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

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

Classe co-NP

A classe complementar de NP é o conjunto de linguagens

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

Exemplos:

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

Um exemplo: tautologia

- Outra maneira de entender: co-NP é o conjunto de linguagens que possuem um certificado para a resposta **não**

Tautologia

Dada uma fórmula booleana com um conjunto de n variáveis e operadores $\wedge$, $\vee$, $\neg$ etc, ela é verdade 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 **não**: $x = 0, y = 1$
- Não conhecemos um certificado curto para instâncias sim

Um exemplo: tautologia

- Outra maneira de entender: co-NP é o conjunto de linguagens que possuem um certificado para a resposta **não**

Tautologia

Dada uma fórmula booleana com um conjunto de n variáveis e operadores $\wedge$, $\vee$, $\neg$ etc, ela é verdade 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 **não**: $x = 0, y = 1$
- Não conhecemos um certificado curto para instâncias sim

Um exemplo: tautologia

- Outra maneira de entender: co-NP é o conjunto de linguagens que possuem um certificado para a resposta **não**

Tautologia

Dada uma fórmula booleana com um conjunto de n variáveis e operadores $\wedge$, $\vee$, $\neg$ etc, ela é verdade 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 **não**: $x = 0, y = 1$
- Não conhecemos um certificado curto para instâncias sim

Um exemplo: tautologia

- Outra maneira de entender: co-NP é o conjunto de linguagens que possuem um certificado para a resposta **não**

Tautologia

Dada uma fórmula booleana com um conjunto de n variáveis e operadores $\wedge$, $\vee$, $\neg$ etc, ela é verdade 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 **não**: $x = 0, y = 1$
- Não conhecemos um certificado curto para instâncias sim

Um exemplo: tautologia

- Outra maneira de entender: co-NP é o conjunto de linguagens que possuem um certificado para a resposta **não**

Tautologia

Dada uma fórmula booleana com um conjunto de n variáveis e operadores $\wedge$, $\vee$, $\neg$ etc, ela é verdade 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 **não**: $x = 0, y = 1$
- Não conhecemos um certificado curto para instâncias sim

Um exemplo: tautologia

- Outra maneira de entender: co-NP é o conjunto de linguagens que possuem um certificado para a resposta **não**

Tautologia

Dada uma fórmula booleana com um conjunto de n variáveis e operadores $\wedge$, $\vee$, $\neg$ etc, ela é verdade 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 **não**: $x = 0, y = 1$
- Não conhecemos um certificado curto para instâncias sim

Classes de Complexidade

Resumo das Classes até agora

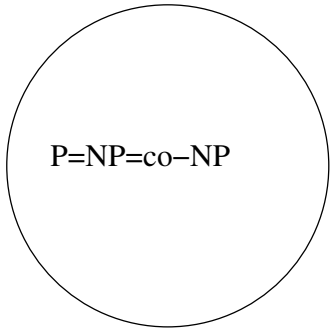
$$\mathrm{P} = \{L \subseteq \Sigma^* : \text{existe algoritmo polinomial } A \text{ que decide } L\ \}$$

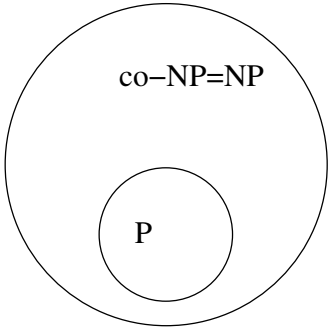
$$\mathrm{NP} = \{L \subseteq \Sigma^* : \text{existe verificador polinomial } A \text{ que verifica } L\text{:}$$
$$x \in L \text{ sse existe } y \in \Sigma^* \text{ polinomial em } |x|,$$
$$\text{tal que } A(x,y) = 1\ \}$$

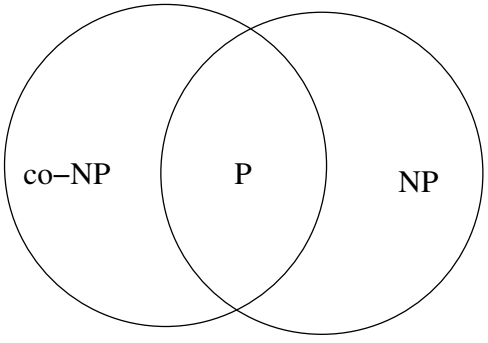
$$\text{co-NP} = \{L \subseteq \Sigma^* : \text{existe verificador polinomial } A \text{ que verifica } \overline{L}$$
$$x \notin L \text{ sse existe } y \in \Sigma^* \text{ polinomial em } |x|,$$
$$\text{tal que } A(x,y) = 1\ \}$$

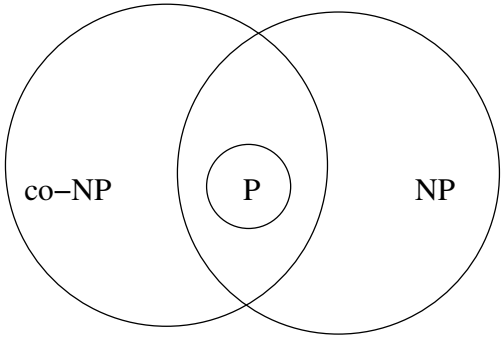
Classes de Complexidade

Possíveis configurações destas classes:









NP-Completude

NP-Completude

- Classe NP-Completo

Refletindo o que vimos

Coletando informações

Até agora vimos que há vários problemas:

- em P que podemos decidir em tempo polinomial
  - e.g.: PATH
- em NP que só sabemos verificar em tempo polinomial
  - e.g.: HAM-CICLO

Por que não achamos algoritmo rápido para HAM-CICLO?

- HAM-CICLO é mais difícil que PATH?
- HAM-CICLO é mais difícil que qualquer problema em P?

Queremos descobrir se HAM-CICLO é

NP-difícil $\approx$ mais difícil que qualquer problema polinomial

Refletindo o que vimos

Coletando informações

Até agora vimos que há vários problemas:

- em P que podemos decidir em tempo polinomial
  - e.g.: PATH
- em NP que só sabemos verificar em tempo polinomial
  - e.g.: HAM-CICLO

Por que não achamos algoritmo rápido para HAM-CICLO?

- HAM-CICLO é mais difícil que PATH?
- HAM-CICLO é mais difícil que qualquer problema em P?

Queremos descobrir se HAM-CICLO é

NP-difícil $\approx$ mais difícil que qualquer problema polinomial

Refletindo o que vimos

Coletando informações

Até agora vimos que há vários problemas:

- em P que podemos decidir em tempo polinomial
  - e.g.: PATH
- em NP que só sabemos verificar em tempo polinomial
  - e.g.: HAM-CICLO

Por que não achamos algoritmo rápido para HAM-CICLO?

- HAM-CICLO é mais difícil que PATH?
- HAM-CICLO é mais difícil que qualquer problema em P?

Queremos descobrir se HAM-CICLO é

NP-difícil $\approx$ mais difícil que qualquer problema polinomial

Refletindo o que vimos

Coletando informações

Até agora vimos que há vários problemas:

- em P que podemos decidir em tempo polinomial
  - e.g.: PATH
- em NP que só sabemos verificar em tempo polinomial
  - e.g.: HAM-CICLO

Por que não achamos algoritmo rápido para HAM-CICLO?

- HAM-CICLO é mais difícil que PATH?
- HAM-CICLO é mais difícil que qualquer problema em P?

Queremos descobrir se HAM-CICLO é

NP-difícil $\approx$ mais difícil que qualquer problema polinomial

Refletindo o que vimos

Coletando informações

Até agora vimos que há vários problemas:

- em P que podemos decidir em tempo polinomial
  - e.g.: PATH
- em NP que só sabemos verificar em tempo polinomial
  - e.g.: HAM-CICLO

Por que não achamos algoritmo rápido para HAM-CICLO?

- HAM-CICLO é mais difícil que PATH?
- HAM-CICLO é mais difícil que qualquer problema em P?

Queremos descobrir se HAM-CICLO é

NP-difícil $\approx$ mais difícil que qualquer problema polinomial

Refletindo o que vimos

Coletando informações

Até agora vimos que há vários problemas:

- em P que podemos decidir em tempo polinomial
  - e.g.: PATH
- em NP que só sabemos verificar em tempo polinomial
  - e.g.: HAM-CICLO

Por que não achamos algoritmo rápido para HAM-CICLO?

- HAM-CICLO é mais difícil que PATH?
- HAM-CICLO é mais difícil que qualquer problema em P?

Queremos descobrir se HAM-CICLO é

NP-difícil $\approx$ mais difícil que qualquer problema polinomial

Refletindo o que vimos

Coletando informações

Até agora vimos que há vários problemas:

- em P que podemos decidir em tempo polinomial
  - e.g.: PATH
- em NP que só sabemos verificar em tempo polinomial
  - e.g.: HAM-CICLO

Por que não achamos algoritmo rápido para HAM-CICLO?

- HAM-CICLO é mais difícil que PATH?
- HAM-CICLO é mais difícil que qualquer problema em P?

Queremos descobrir se HAM-CICLO é

NP-difícil $\approx$ mais difícil que qualquer problema polinomial

Refletindo o que vimos

Coletando informações

Até agora vimos que há vários problemas:

- em P que podemos decidir em tempo polinomial
  - e.g.: PATH
- em NP que só sabemos verificar em tempo polinomial
  - e.g.: HAM-CICLO

Por que não achamos algoritmo rápido para HAM-CICLO?

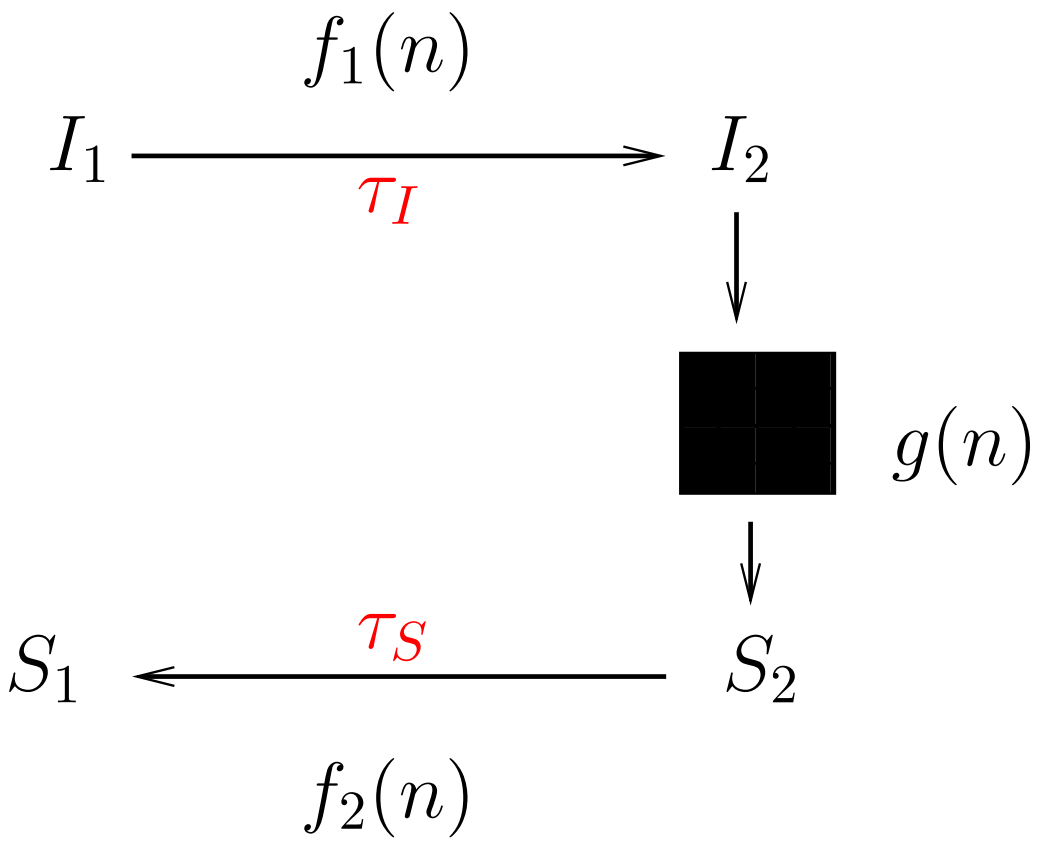
- HAM-CICLO é mais difícil que PATH?
- HAM-CICLO é mais difícil que qualquer problema em P?

Queremos descobrir se HAM-CICLO é

NP-difícil $\approx$ mais difícil que qualquer problema polinomial

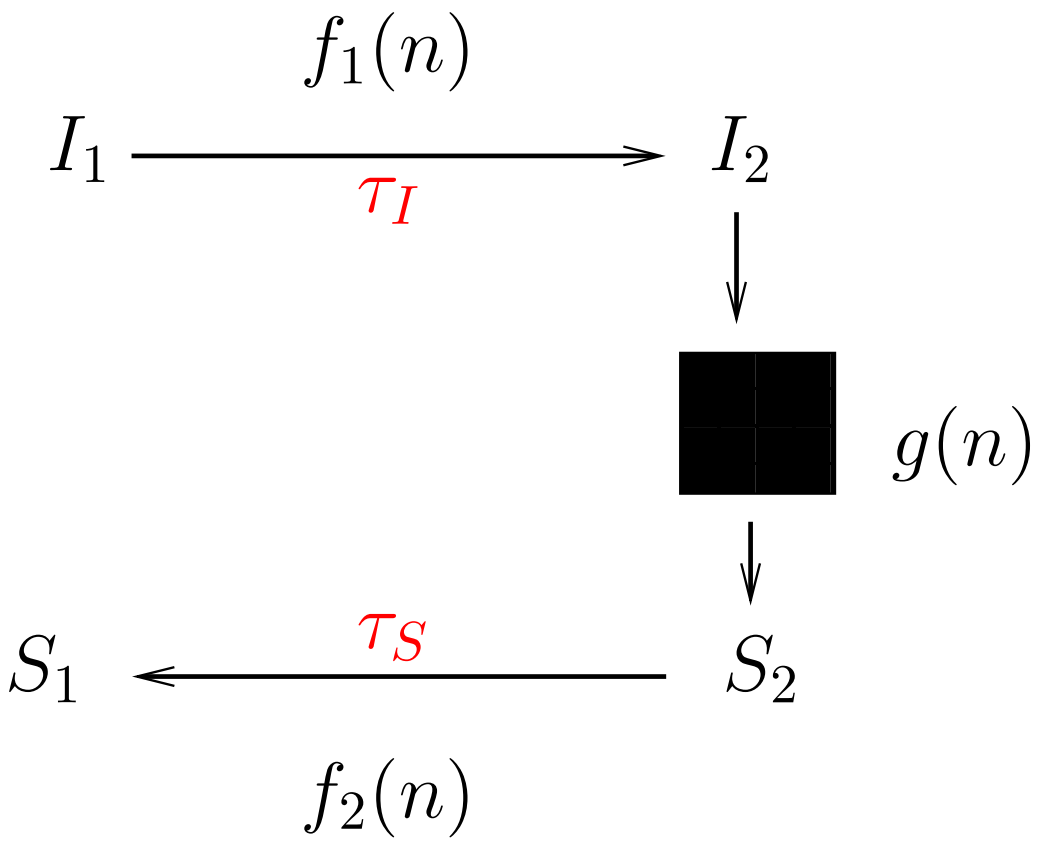
Como comparar problema?

A ferramenta adequada para comparar problemas são as reduções:



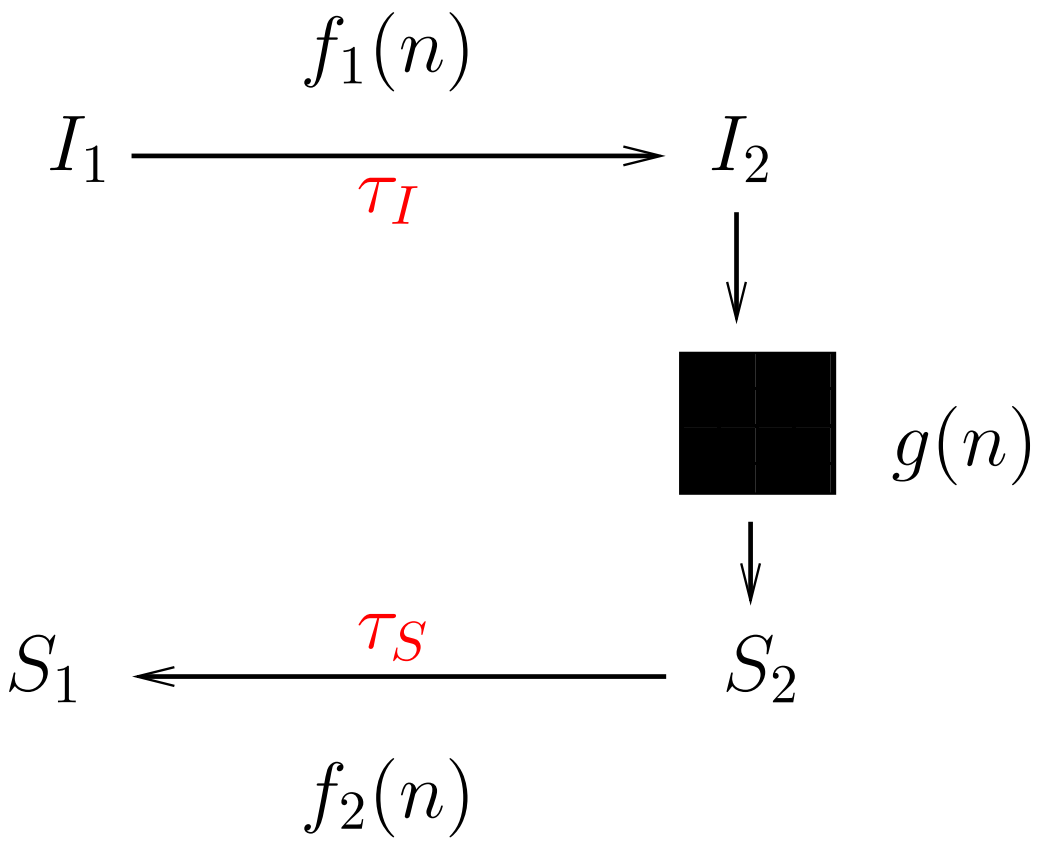
Como comparar problema?

A ferramenta adequada para comparar problemas são as reduções:



Como comparar problema?

A ferramenta adequada para comparar problemas são as reduções:



Reduções de Karp

Agora só estamos interessados em:

- saber se um problema é polinomialmente mais difícil
  - basta que $f_1(n)$ e $f_2(n)$ devem ter tempo polinomial
- lidar com problemas de decisão
  - a transformação de saída τ_S não faz nada!

Redução de Karp

Uma linguagem L_1 é redutível em tempo polinomial para outra linguagem L_2 ($L_1 \leqslant_p L_2$) se

- existe algoritmo $f : \{0,1\}^* \to \{0,1\}^*$
- $f(x)$ executa em tempo polinomial em $|x_1|^c$, para c constante
- $x_1 \in L_1$ se e somente se $x_2 \in L_2$.

Reduções de Karp

Agora só estamos interessados em:

- saber se um problema é polinomialmente mais difícil
  - basta que $f_1(n)$ e $f_2(n)$ devem ter tempo polinomial
- lidar com problemas de decisão
  - a transformação de saída τ_S não faz nada!

Redução de Karp

Uma linguagem L_1 é redutível em tempo polinomial para outra linguagem L_2 ($L_1 \leqslant_p L_2$) se

- existe algoritmo $f : \{0,1\}^* \to \{0,1\}^*$
- $f(x)$ executa em tempo polinomial em $|x_1|^c$, para c constante
- $x_1 \in L_1$ se e somente se $x_2 \in L_2$.

Reduções de Karp

Agora só estamos interessados em:

- saber se um problema é polinomialmente mais difícil
  - basta que $f_1(n)$ e $f_2(n)$ devem ter tempo polinomial
- lidar com problemas de decisão
  - a transformação de saída τ_S não faz nada!

Redução de Karp

Uma linguagem L_1 é redutível em tempo polinomial para outra linguagem L_2 ($L_1 \leqslant_p L_2$) se

- existe algoritmo $f : \{0,1\}^* \to \{0,1\}^*$
- $f(x)$ executa em tempo polinomial em $|x_1|^c$, para c constante
- $x_1 \in L_1$ se e somente se $x_2 \in L_2$.

Reduções de Karp

Agora só estamos interessados em:

- saber se um problema é polinomialmente mais difícil
  - basta que $f_1(n)$ e $f_2(n)$ devem ter tempo polinomial
- lidar com problemas de decisão
  - a transformação de saída τ_S não faz nada!

Redução de Karp

Uma linguagem L_1 é redutível em tempo polinomial para outra linguagem L_2 ($L_1 \leqslant_p L_2$) se

- existe algoritmo $f : \{0,1\}^* \to \{0,1\}^*$
- $f(x)$ executa em tempo polinomial em $|x_1|^c$, para c constante
- $x_1 \in L_1$ se e somente se $x_2 \in L_2$.

Reduções de Karp

Agora só estamos interessados em:

- saber se um problema é polinomialmente mais difícil
  - basta que $f_1(n)$ e $f_2(n)$ devem ter tempo polinomial
- lidar com problemas de decisão
  - a transformação de saída τ_S não faz nada!

Redução de Karp

Uma linguagem L_1 é redutível em tempo polinomial para outra linguagem L_2 ($L_1 \leqslant_p L_2$) se

- existe algoritmo $f : \{0,1\}^* \rightarrow \{0,1\}^*$
- $f(x)$ executa em tempo polinomial em $|x_1|^c$, para c constante
- $x_1 \in L_1$ se e somente se $x_2 \in L_2$.

Reduções de Karp

Agora só estamos interessados em:

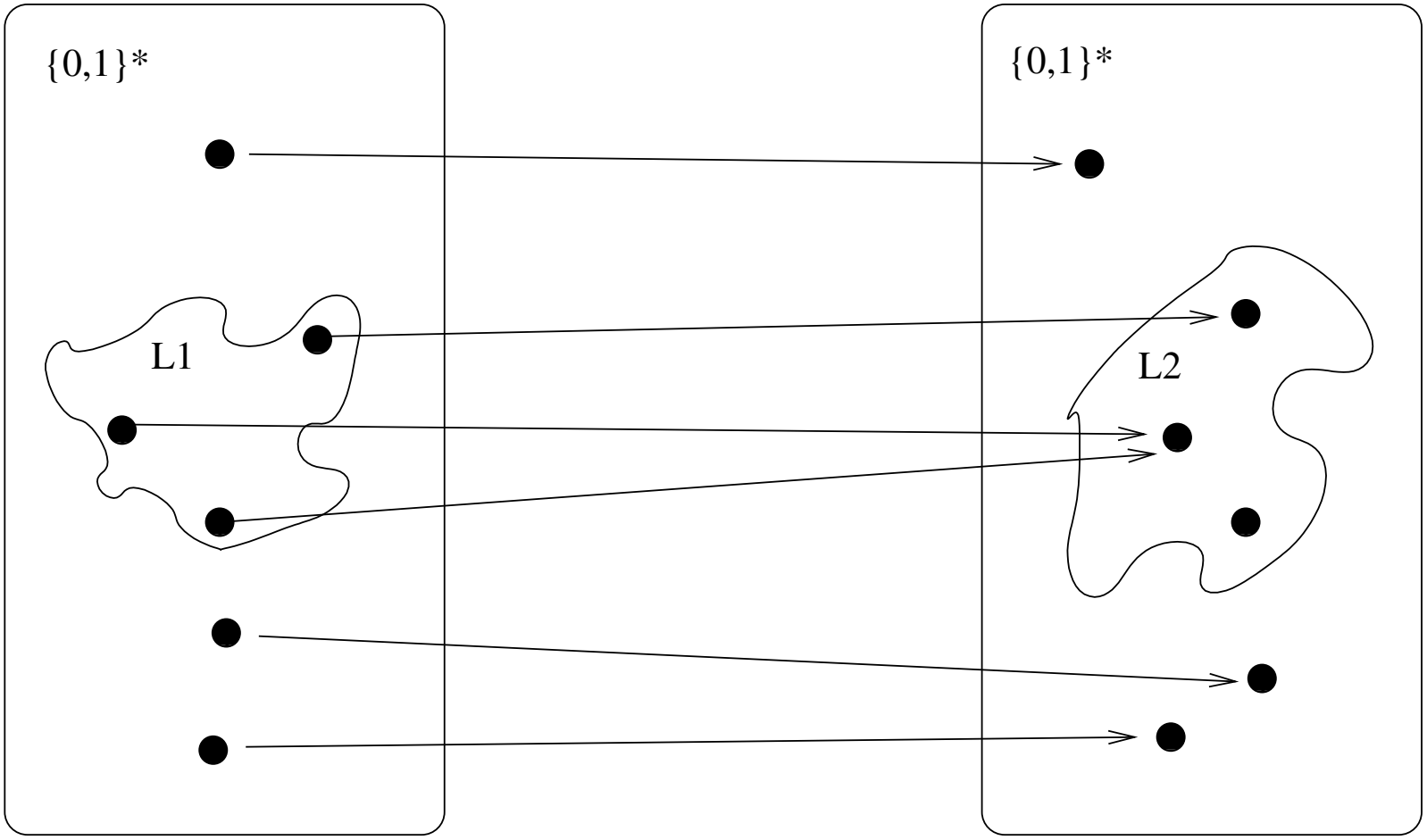
- saber se um problema é polinomialmente mais difícil
  - basta que $f_1(n)$ e $f_2(n)$ devem ter tempo polinomial
- lidar com problemas de decisão
  - a transformação de saída τ_S não faz nada!

Redução de Karp

Uma linguagem L_1 é redutível em tempo polinomial para outra linguagem L_2 ($L_1 \leqslant_p L_2$) se

- existe algoritmo $f : \{0,1\}^* \to \{0,1\}^*$
- $f(x)$ executa em tempo polinomial em $|x_1|^c$, para c constante
- $x_1 \in L_1$ se e somente se $x_2 \in L_2$.

Ilustração



Reduções de Tempo Polinomial

Teorema (1.1)

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

Prova.

- Suponha que $L_2 \in P$ e seja A_2 um algoritmo que decide L_2 em tempo polinomial.
- Seja f um algoritmo polinomial que reduz L_1 a L_2.
- Dado $x \in \{0,1\}^*$, defina $A_1(x) := A_2(f(x))$
- Pelo fato de f ser 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.

Reduções de Tempo Polinomial

Teorema (1.1)

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

Prova.

- Suponha que $L_2 \in P$ e seja A_2 um algoritmo que decide L_2 em tempo polinomial.
- Seja f um algoritmo polinomial que reduz L_1 a L_2.
- Dado $x \in \{0,1\}^*$, defina $A_1(x) := A_2(f(x))$
- Pelo fato de f ser 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.

Reduções de Tempo Polinomial

Teorema (1.1)

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

Prova.

- Suponha que $L_2 \in P$ e seja A_2 um algoritmo que decide L_2 em tempo polinomial.
- Seja f um algoritmo polinomial que reduz L_1 a L_2.
- Dado $x \in \{0,1\}^*$, defina $A_1(x) := A_2(f(x))$
- Pelo fato de f ser 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.

Reduções de Tempo Polinomial

Teorema (1.1)

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

Prova.

- Suponha que $L_2 \in P$ e seja A_2 um algoritmo que decide L_2 em tempo polinomial.
- Seja f um algoritmo polinomial que reduz L_1 a L_2.
- Dado $x \in \{0,1\}^*$, defina $A_1(x) := A_2(f(x))$
- Pelo fato de f ser 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.

Reduções de Tempo Polinomial

Teorema (1.1)

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

Prova.

- Suponha que $L_2 \in P$ e seja A_2 um algoritmo que decide L_2 em tempo polinomial.
- Seja f um algoritmo polinomial que reduz L_1 a L_2.
- Dado $x \in \{0,1\}^*$, defina $A_1(x) := A_2(f(x))$
- Pelo fato de f ser 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.

Reduções de Tempo Polinomial

Teorema (1.1)

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

Prova.

- Suponha que $L_2 \in P$ e seja A_2 um algoritmo que decide L_2 em tempo polinomial.
- Seja f um algoritmo polinomial que reduz L_1 a L_2.
- Dado $x \in \{0,1\}^*$, defina $A_1(x) := A_2(f(x))$
- Pelo fato de f ser 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 (NPC)

Definição

Um problema $L \subseteq \{0,1\}^*$ *é* NP*-completo se:*

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

- Se L satisfaz apenas a propriedade 2, então dizemos que L é NP-Difícil.

Classe NP-Completo (NPC)

Definição

Um problema $L \subseteq \{0,1\}^*$ é NP-completo se:

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

- Se L satisfaz apenas a propriedade 2, então dizemos que L é NP-Difícil.

Classe NP-Completo (NPC)

Teorema

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

Prova. Suponha que existe $L \in P \cap NPC$. Assim, para toda linguagem $L' \in NP$, temos $L' \preccurlyeq L$ pois $L \in NPC$. Mas como $L \in P$, pelo teorema2, L' pode ser decidida em tempo polinomial. Daí $L' \in P$. Portanto $NP \subseteq P$

Teorema

Se existe problema $L \in NP$ tal que $L \notin P$, então $NPC \cap P = \emptyset$.

Prova. Exercício

Classe NP-Completo (NPC)

Teorema

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

Prova. Suponha que existe $L \in P \cap NPC$. Assim, para toda linguagem $L' \in NP$, temos $L' \preccurlyeq L$ pois $L \in NPC$. Mas como $L \in P$, pelo teorema2, L' pode ser decidida em tempo polinomial. Daí $L' \in P$. Portanto $NP \subseteq P$

Teorema

Se existe problema $L \in NP$ tal que $L \notin P$, então $NPC \cap P = \emptyset$.

Prova. Exercício

Classe NP-Completo (NPC)

Teorema

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

Prova. Suponha que existe $L \in P \cap NPC$. Assim, para toda linguagem $L' \in NP$, temos $L' \preccurlyeq L$ pois $L \in NPC$. Mas como $L \in P$, pelo teorema2, L' pode ser decidida em tempo polinomial. Daí $L' \in P$. Portanto $NP \subseteq P$

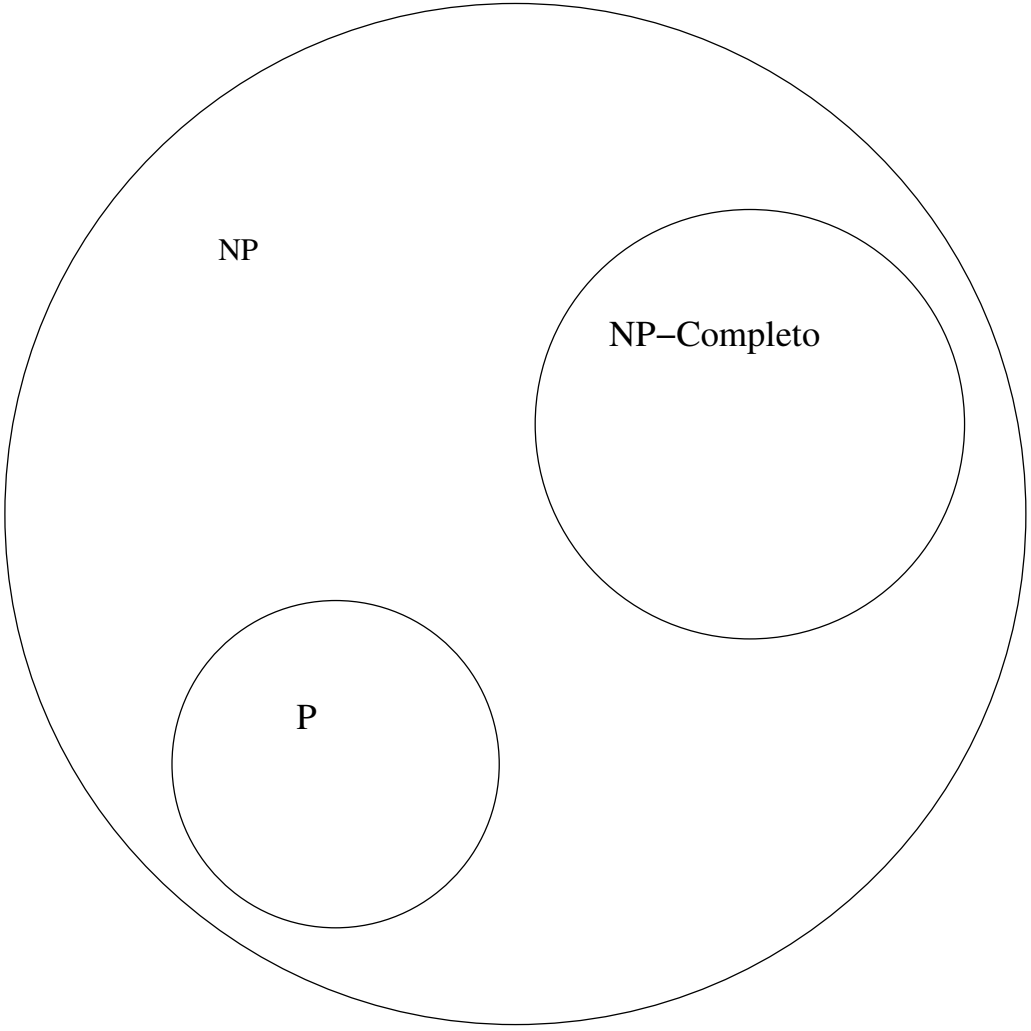
Teorema

Se existe problema $L \in NP$ tal que $L \notin P$, então $NPC \cap P = \emptyset$.

Prova. Exercício

Classe NP-Completo (NPC)

Como acreditamos que é a relação das classes:



NP-Completude

- Um primeiro problema NP-Completo

Satisfatibilidade (SAT)

Mas será que a classe NP-Completo é vazia?

Vamos ver que não!

- 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 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 (SAT)

Mas será que a classe NP-Completo é vazia?

Vamos ver que não!

- 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 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 (SAT)

Mas será que a classe NP-Completo é vazia?
Vamos ver que não!

- 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 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

$f = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
Atribuição: $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$

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

Satisfatibilidade

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

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

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 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

Teorema (Cook–Levin)

SAT é NP-Completo.

Prova. Vamos ver um resumo em seguida. □

Um esquema do Teorema de Cook-Levin

Um esquema do Teorema de Cook-Levin

- Circuit Satisfiability

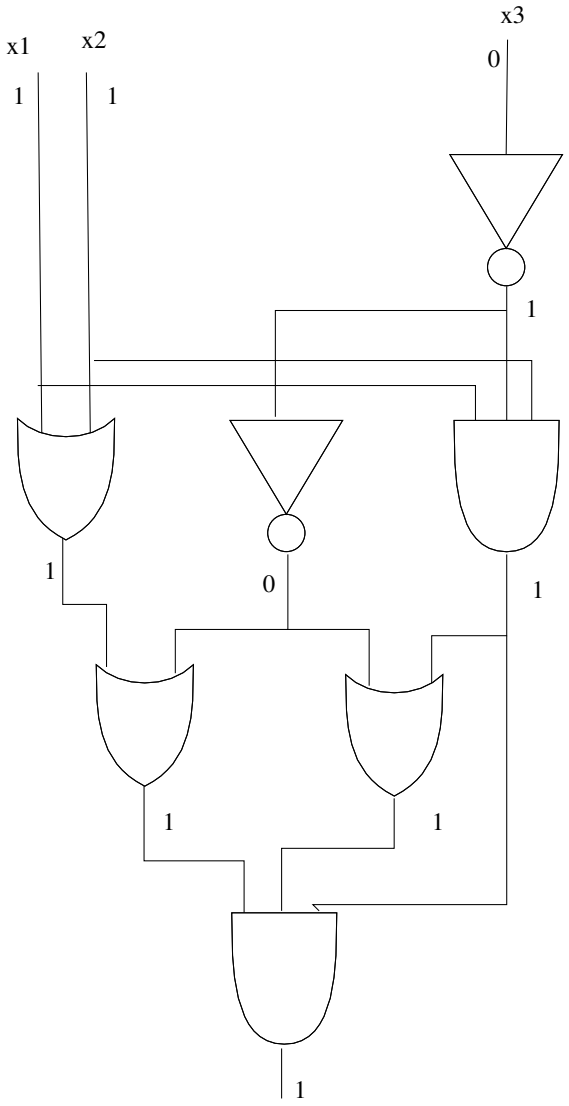
Circuit Satisfiability

- 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

- 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 o algoritmo verificador devolve 0.

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 o algoritmo verificador devolve 0.

□

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 ($Q \leqslant_p$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.

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 ($Q \leqslant_p$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.

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 ideia é montar um circuito que simula o algoritmo A executando em um computador de circuitos lógicos.

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 ideia é montar um circuito que simula o algoritmo A executando em um computador de circuitos lógicos.

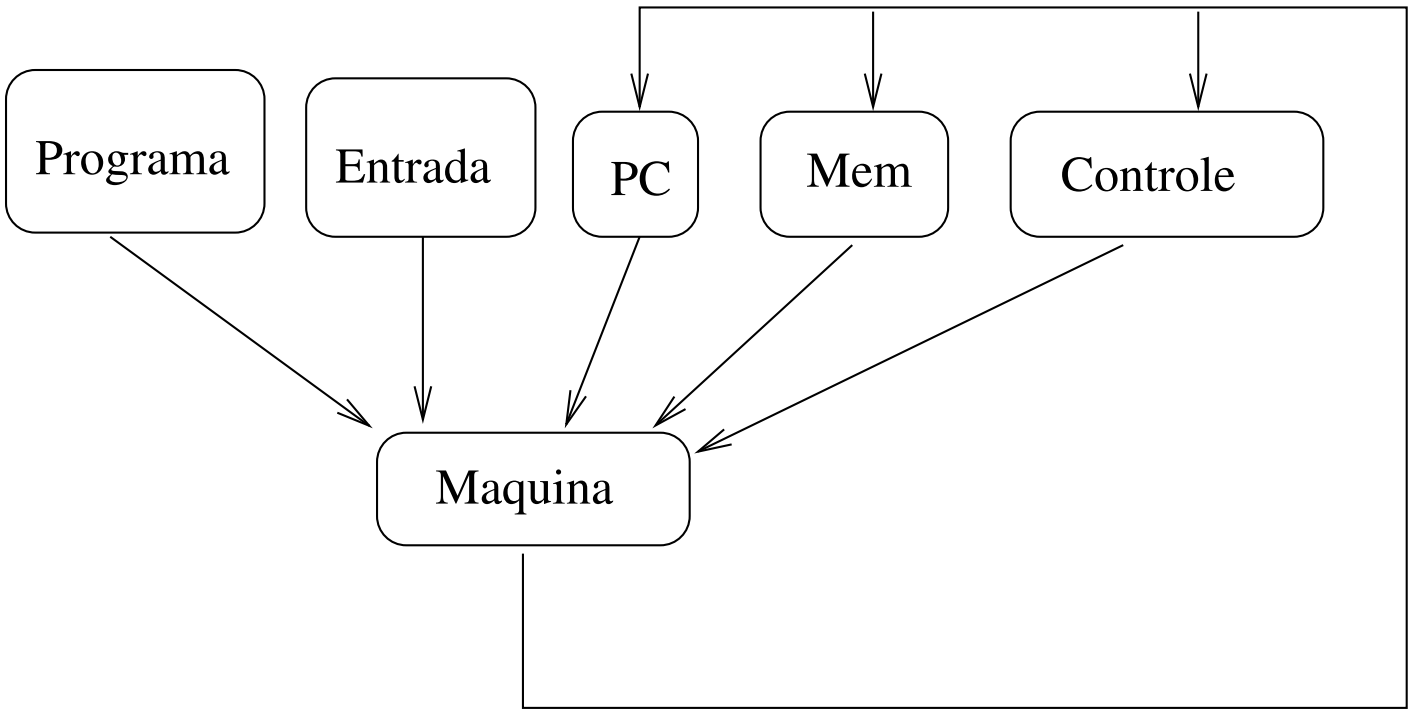
Circuit Satisfiability

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 ideia é montar um circuito que simula o algoritmo A executando em um computador de circuitos lógicos.

Continuação da prova



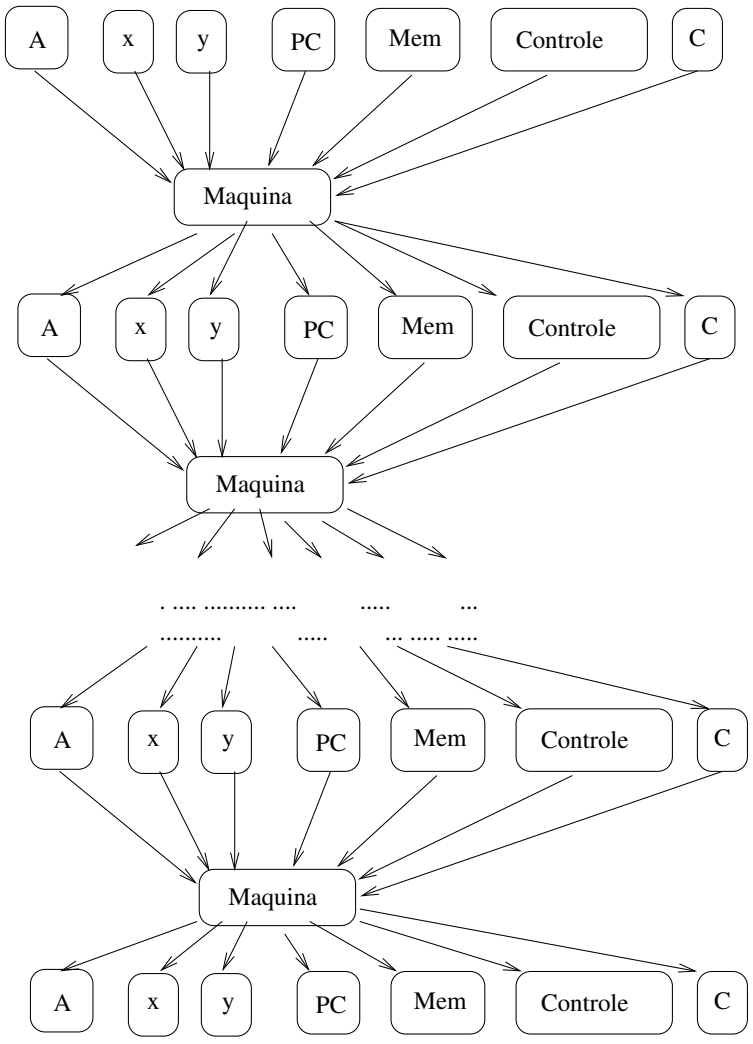
Continuação da prova

- 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

- 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 têm 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

- A máquina, o controle, o PC e A têm 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

- A máquina, o controle, o PC e A têm 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.

Demonstrando NP-completude

Que outros problemas são NP-difíceis?

- Sabemos que C-SAT $\in$ NPC
- Agora temos duas possibilidades descobrir se outro 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 C-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.

Que outros problemas são NP-difíceis?

- Sabemos que C-SAT $\in$ NPC
- Agora temos duas possibilidades descobrir se outro 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 C-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.

Que outros problemas são NP-difíceis?

- Sabemos que C-SAT $\in$ NPC
- Agora temos duas possibilidades descobrir se outro 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 C-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.

Que outros problemas são NP-difíceis?

- Sabemos que C-SAT $\in$ NPC
- Agora temos duas possibilidades descobrir se outro 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 C-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.

Que outros problemas são NP-difíceis?

- Sabemos que C-SAT $\in$ NPC
- Agora temos duas possibilidades descobrir se outro 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 C-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.

NP-Completude via redução

Teorema

Seja $L' \in$ NP-Completo. Se L é tal que $L' \leqslant_p L$ 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 $L'' \leqslant_p L'$.
- Como $L' \leqslant_p L$ então (exercício) $L'' \leqslant_p L$ para todo $L'' \in$ NP.
- Portanto L é NP-Difícil.

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

NP-Completude via redução

Teorema

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

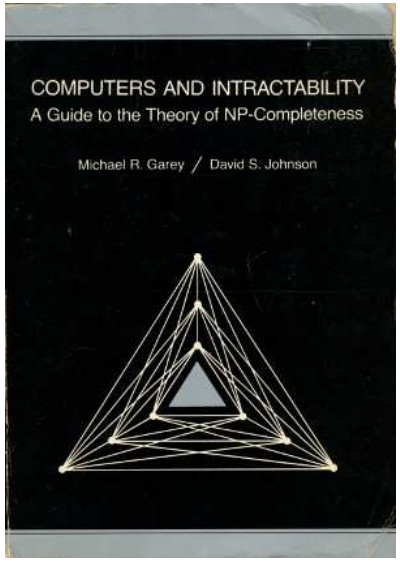
Prova. Como L' é NP-Completo, então

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

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

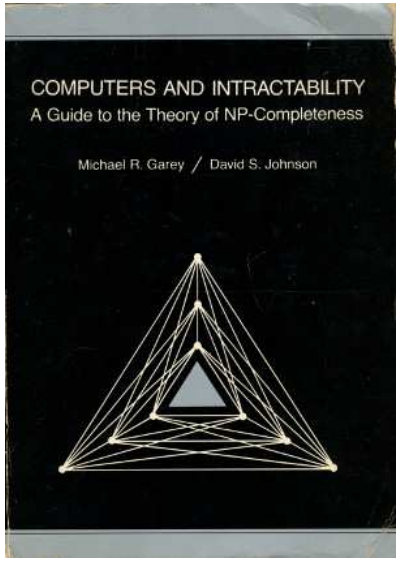
Literatura

- Karp mostrou em 1972 que uma lista de 21 problemas são NP-completos: veja a Wikipédia!
- Com sorte, seu problema foi estudado por Garey e Johnson (1979)



Literatura

- Karp mostrou em 1972 que uma lista de 21 problemas são NP-completos: veja a Wikipédia!
- Com sorte, seu problema foi estudado por Garey e Johnson (1979)



Demonstrando NP-completude

- SAT

Satisfatibilidade (SAT)

Relembrando:

- A fórmula pode ser escrita com os seguintes operadores:
  1. AND ($\wedge$).
  2. OR ($\vee$).
  3. NOT ($\neg$).
  4. Implicação ($\rightarrow$).
  5. Se e Somente Se ($\leftrightarrow$).

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

Satisfatibilidade é NP

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 é NP

Lema

SAT pertence a NP.

- Dado uma fórmula f para o problema, se $f \in \text{SAT}$ 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.

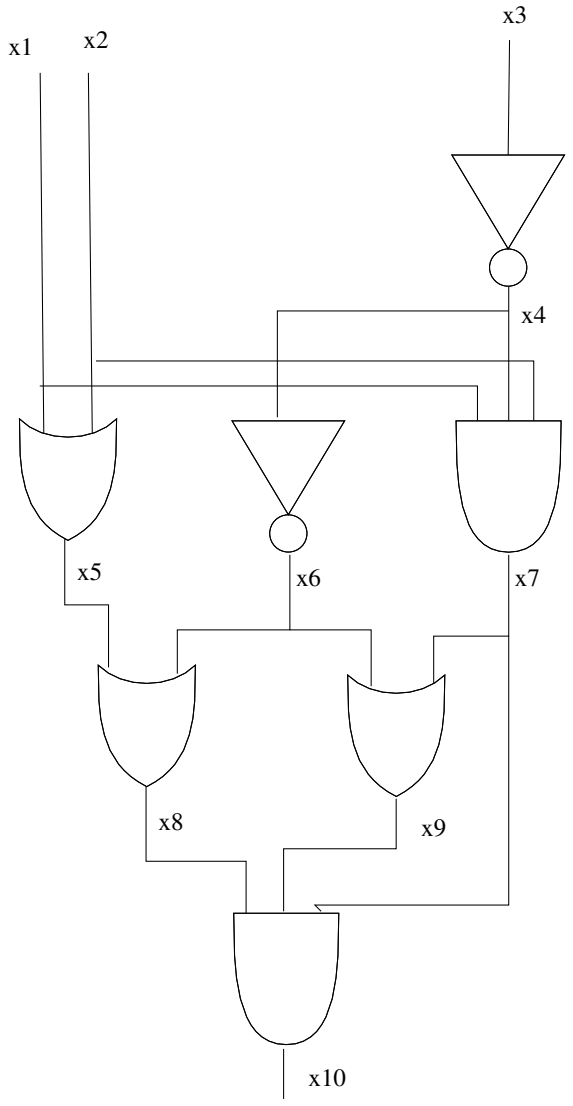
Satisfatibilidade é NP-Difícil

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.

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.

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 dos dois últimos lemas. □

Demonstrando NP-completude

- 3CNF-SAT

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).
- Ex: $(x_1) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$.
- 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-Difícil.

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

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

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.

3CNF-SAT

Lema

3CNF-SAT pertence a NP-Difícil.

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

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

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.

3CNF-SAT

Lema

3CNF-SAT pertence a NP-Difícil.

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

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

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.

3CNF-SAT

Lema

3CNF-SAT pertence a NP-Difícil.

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

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

Construiremos a fórmula f_2 em três etapas.

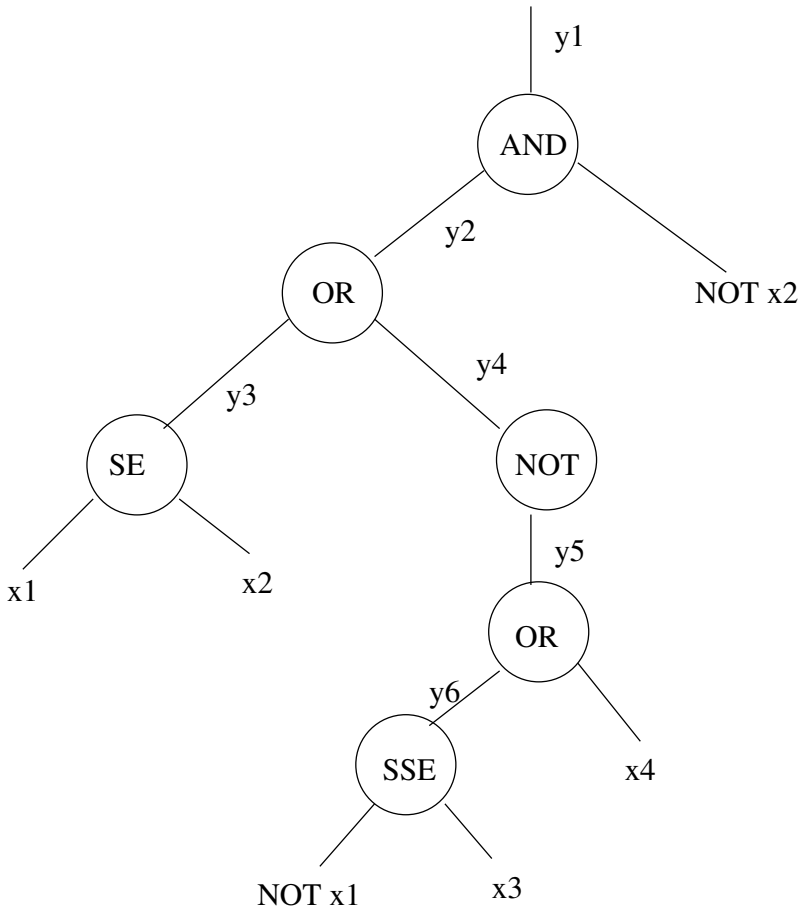
1. Primeiramente transformaremos f_1 em uma fórmula que é uma conjunção de cláusulas contendo até 3 variáveis.
2. A fórmula resultante então será transformada em outra que estará na CNF. Deveremos tirar os operadores $\leftrightarrow$ e $\rightarrow$.
3. Por último deixaremos cada cláusula com exatamente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

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. A ideia é parecida com a prova de NP-Dificuldade do SAT.
- 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 ys).
- 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

- Note que cada cláusula terá até 3 variáveis, no máximo duas como entrada do operador, mais uma nova (correspondente aos ys).
- 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

- Note que cada cláusula terá até 3 variáveis, no máximo duas como entrada do operador, mais uma nova (correspondente aos ys).
- 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

- 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.

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.

Demonstrando NP-completude

- Clique

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

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

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

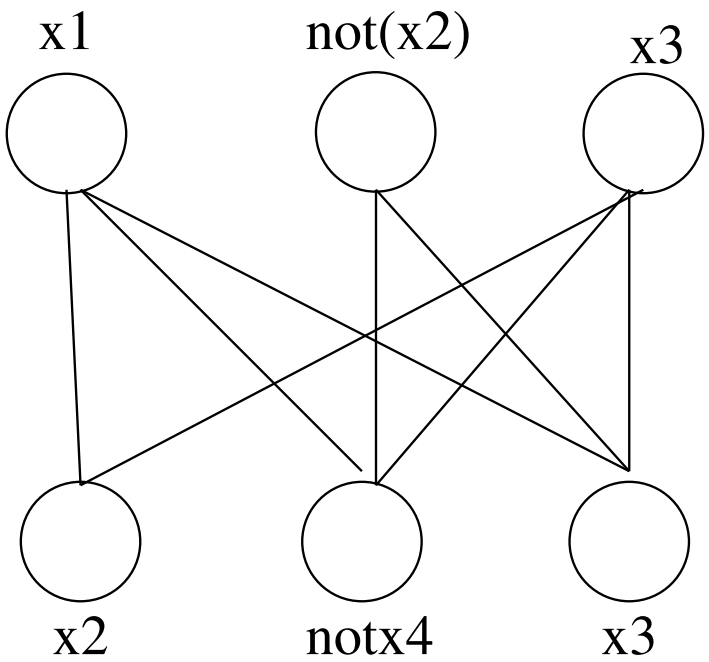
Lema

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.

Continuação da prova

(x1 or not(x2) or x3)



(x2 or not(x4) or x3)

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.

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.

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.

Demonstrando NP-completude

- Vertex-Cover

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 $|V| - k$.

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 $|V| - k$.

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

($\Rightarrow$)

- 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.

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.

Demonstrando NP-completude

- Subset-Sum

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'.

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

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 $x_1, x_2, \ldots, x_n, C_1, \ldots, C_k$.

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 $x_1, x_2, \ldots, x_n, C_1, \ldots, C_k$.

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

- 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

- 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

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

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.

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.

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.

Demonstrando NP-completude

- Ciclo Hamiltoniano

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 $\leqslant_p$ 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

Teorema

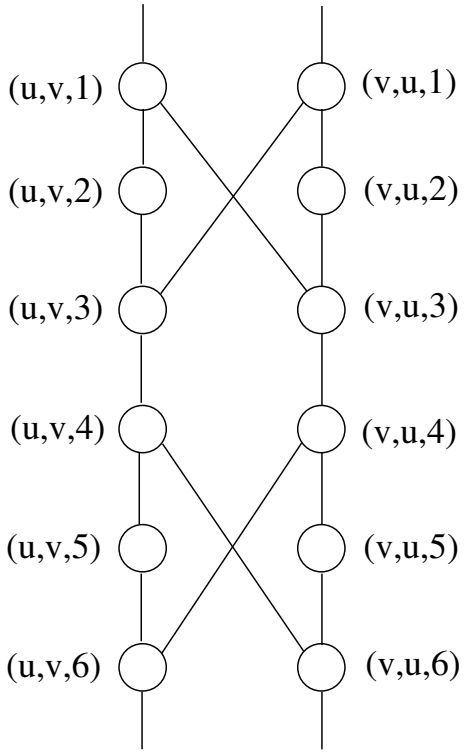
O problema HAM-CYCLE é NP-Difícil.

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

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

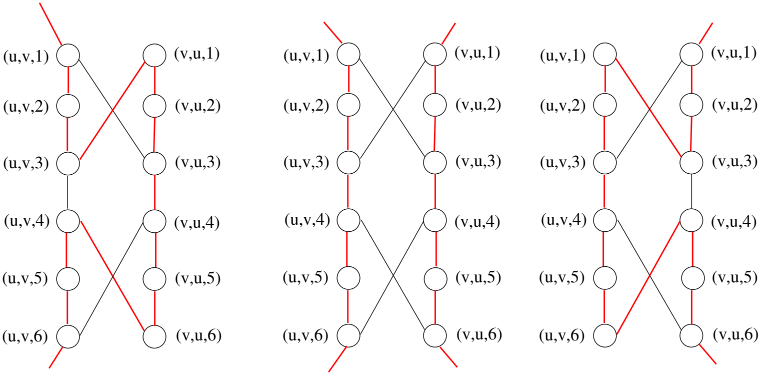
Ciclo Hamiltoniano

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



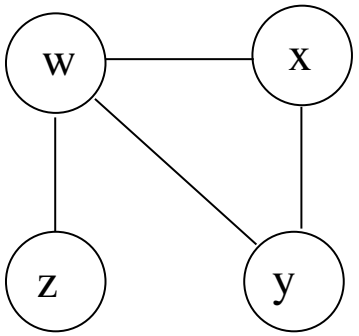
Ciclo Hamiltoniano

- 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:



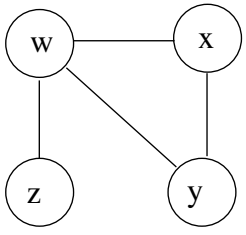
Ciclo Hamiltoniano

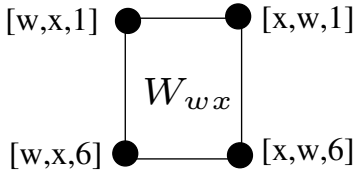
- Para cada aresta (u, v) de G teremos uma destas estruturas.
- Também teremos mais k vértices $s_1, \ldots, s_k$ chamados seletores.
- Para ligar as estruturas W_{uv} entre si e os vértices seletores iremos criar algumas arestas.
- Para deixar mais claro, vamos usar como exemplo o seguinte grafo G instância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

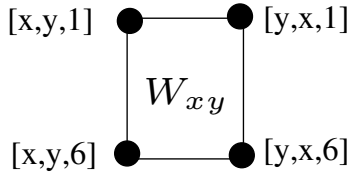


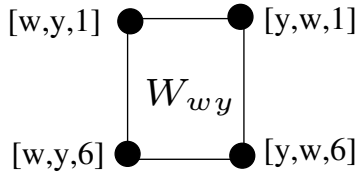
Ciclo Hamiltoniano

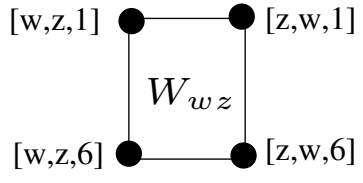
Com uma estrutura para cada aresta e seletores s_1 e s_1:









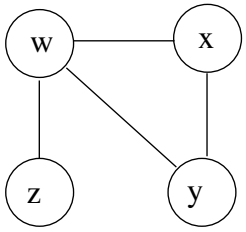


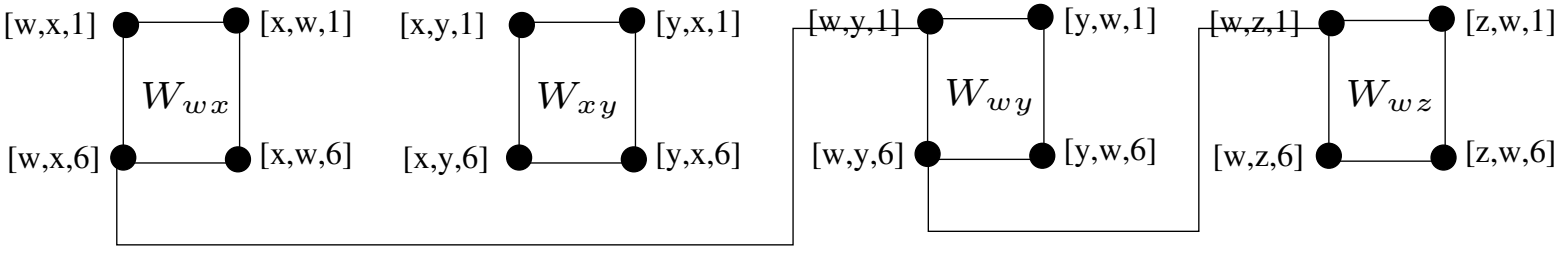
Ciclo Hamiltoniano

- 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

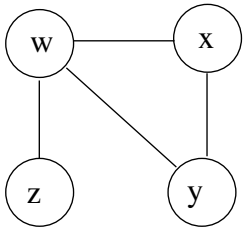
Para o vértice *w* acrescentamos arestas:

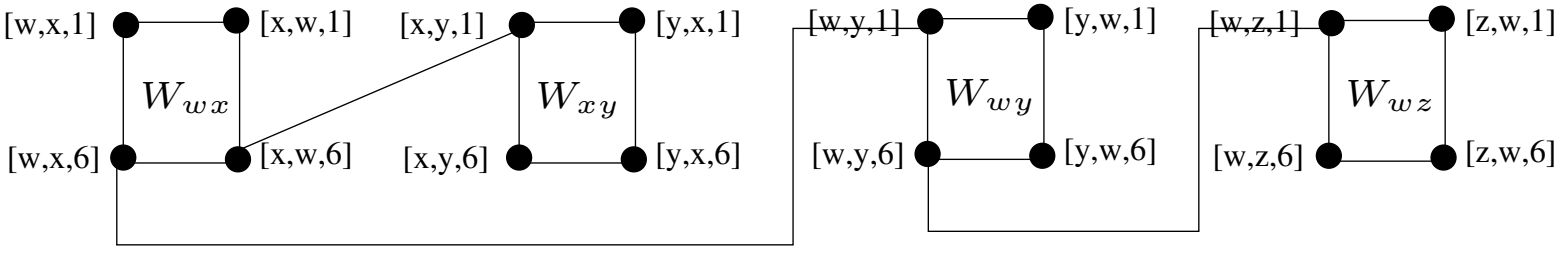




Ciclo Hamiltoniano

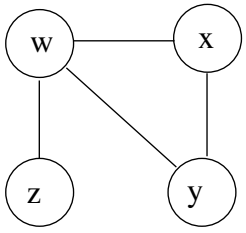
Para o vértice *x* acrescentamos arestas:

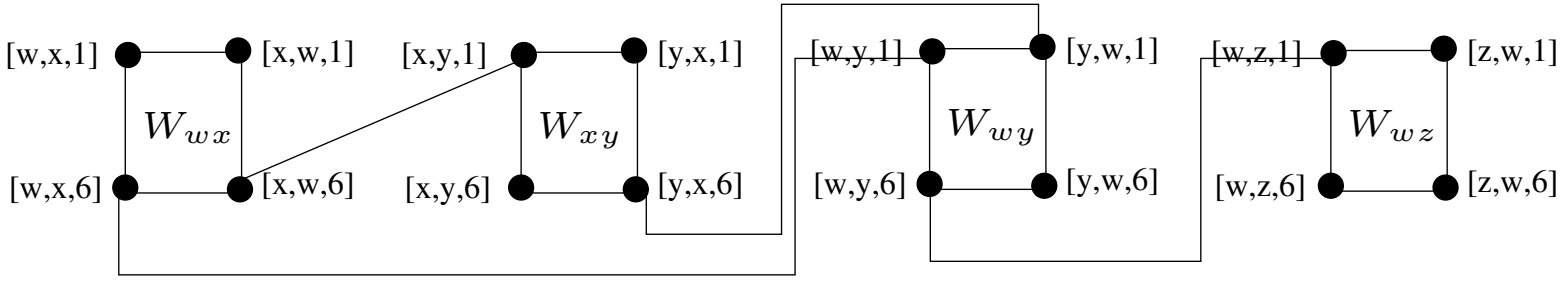




Ciclo Hamiltoniano

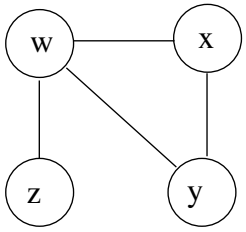
Para o vértice *y* acrescentamos arestas:





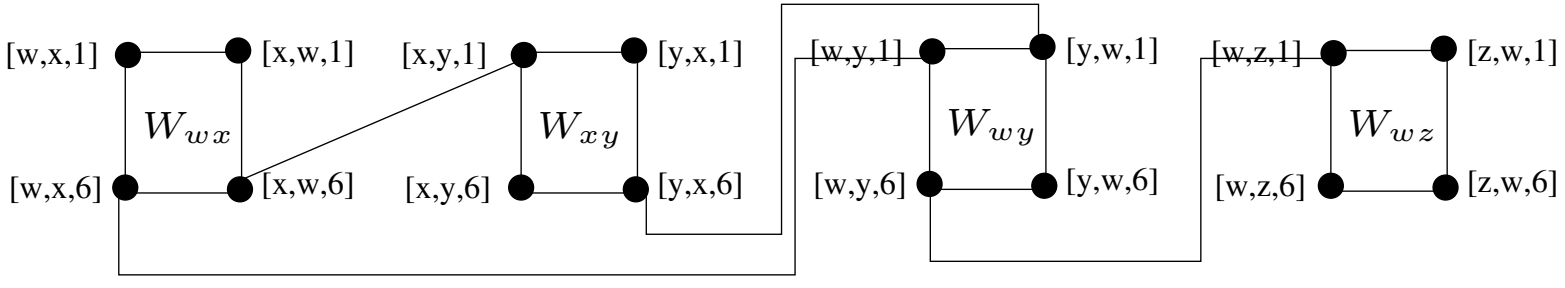
Ciclo Hamiltoniano

Para o vértice *z* não acrescentamos arestas:



s_1

s_2



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

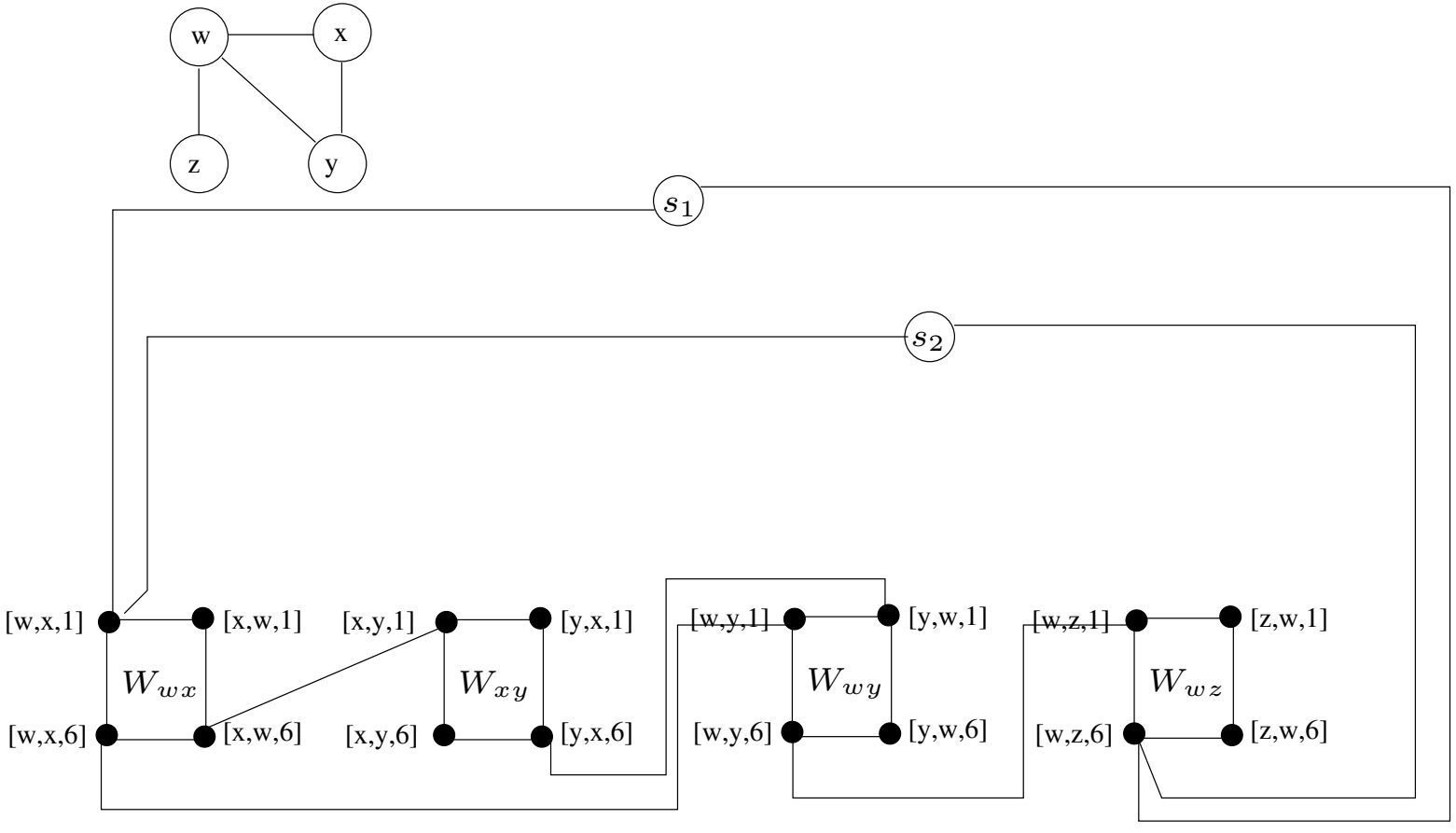
$$\{(s_j, [u, u^1, 1]) : u \in V \text{ e } 1 \leq j \leq k\}$$

e

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

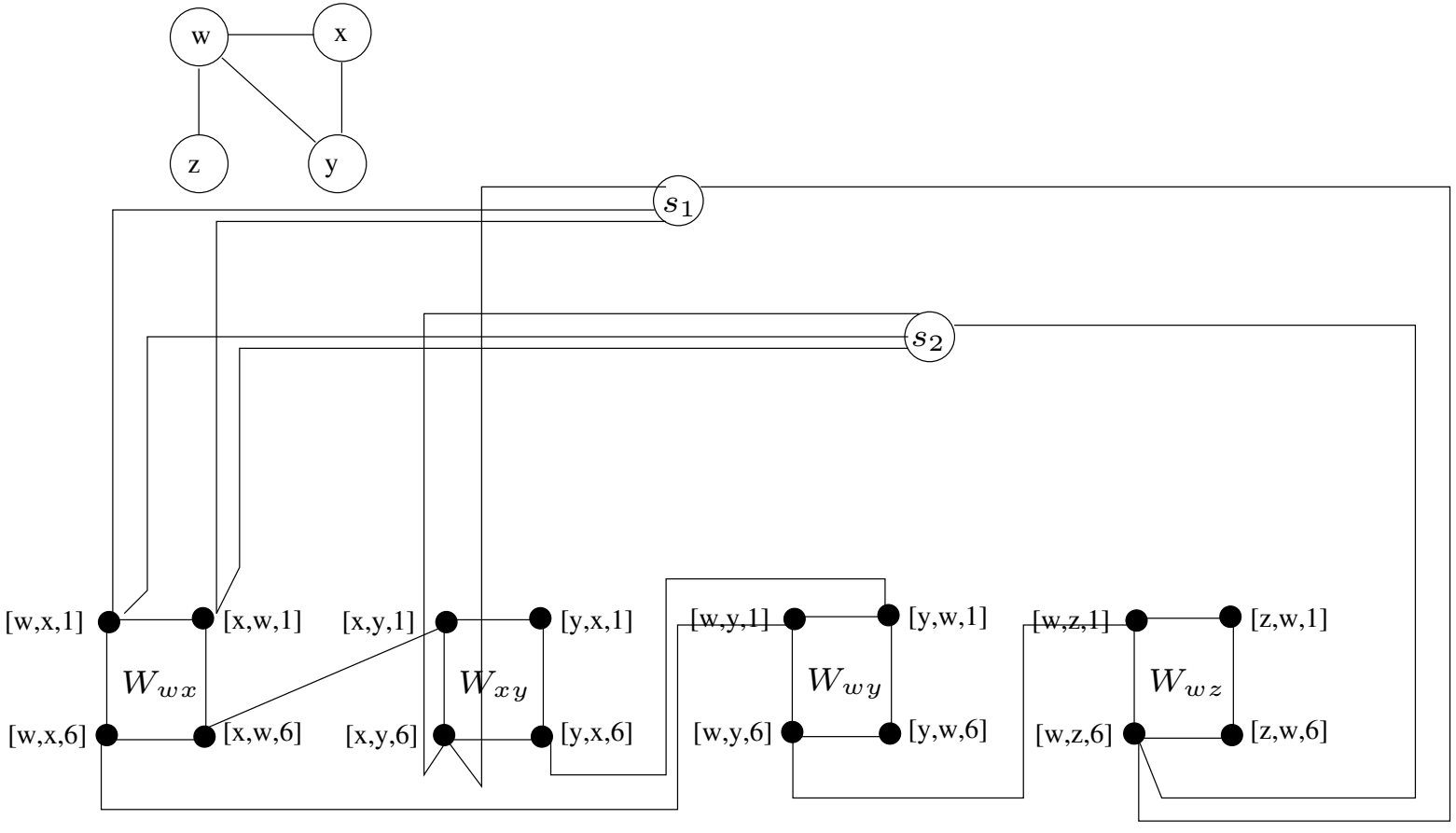
Ciclo Hamiltoniano

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



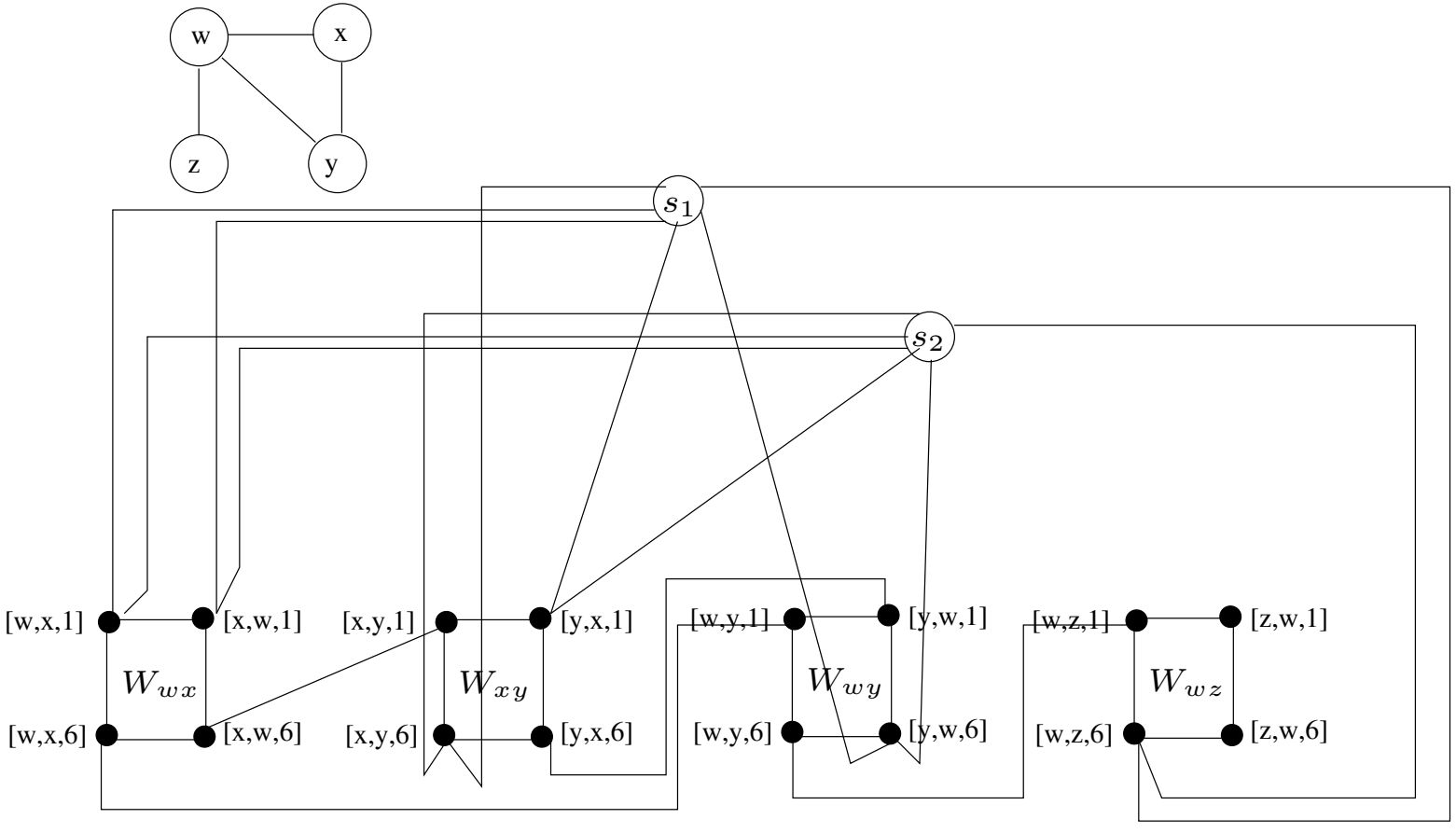
Ciclo Hamiltoniano

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



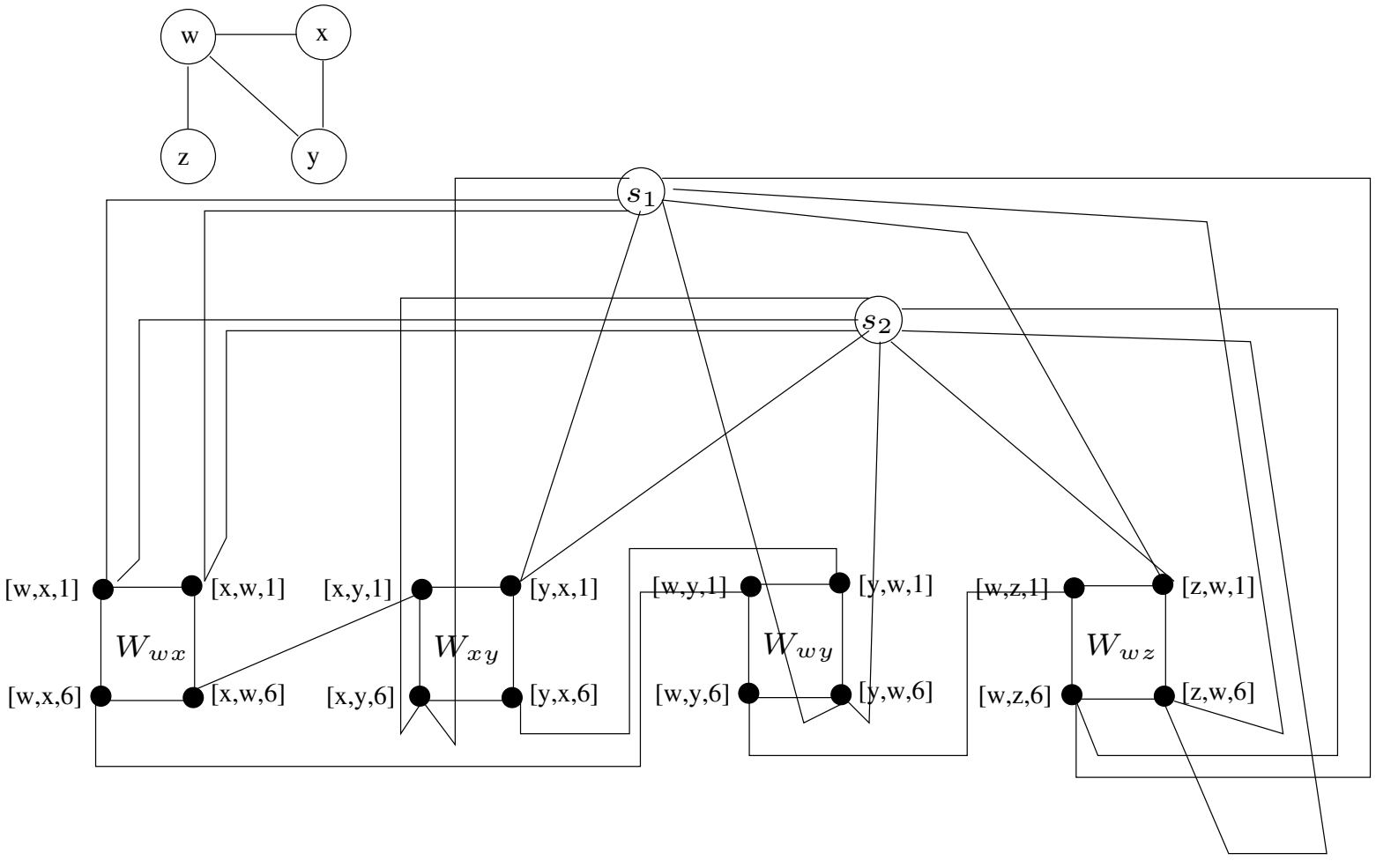
Ciclo Hamiltoniano

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

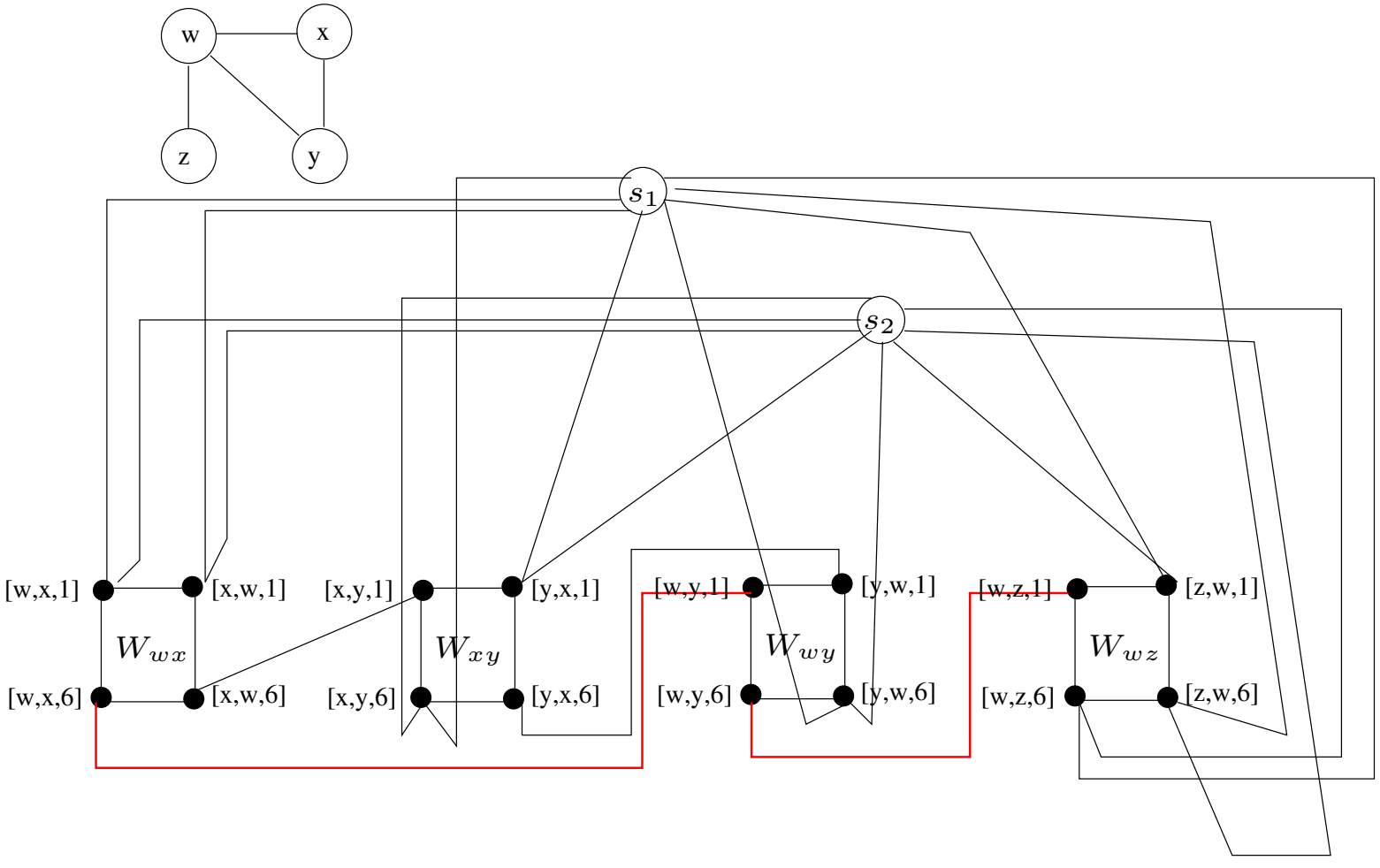
(2) $\{(s_j, [u_j, u_j^1, 1]) : 1 \leq j \leq k\}$

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

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

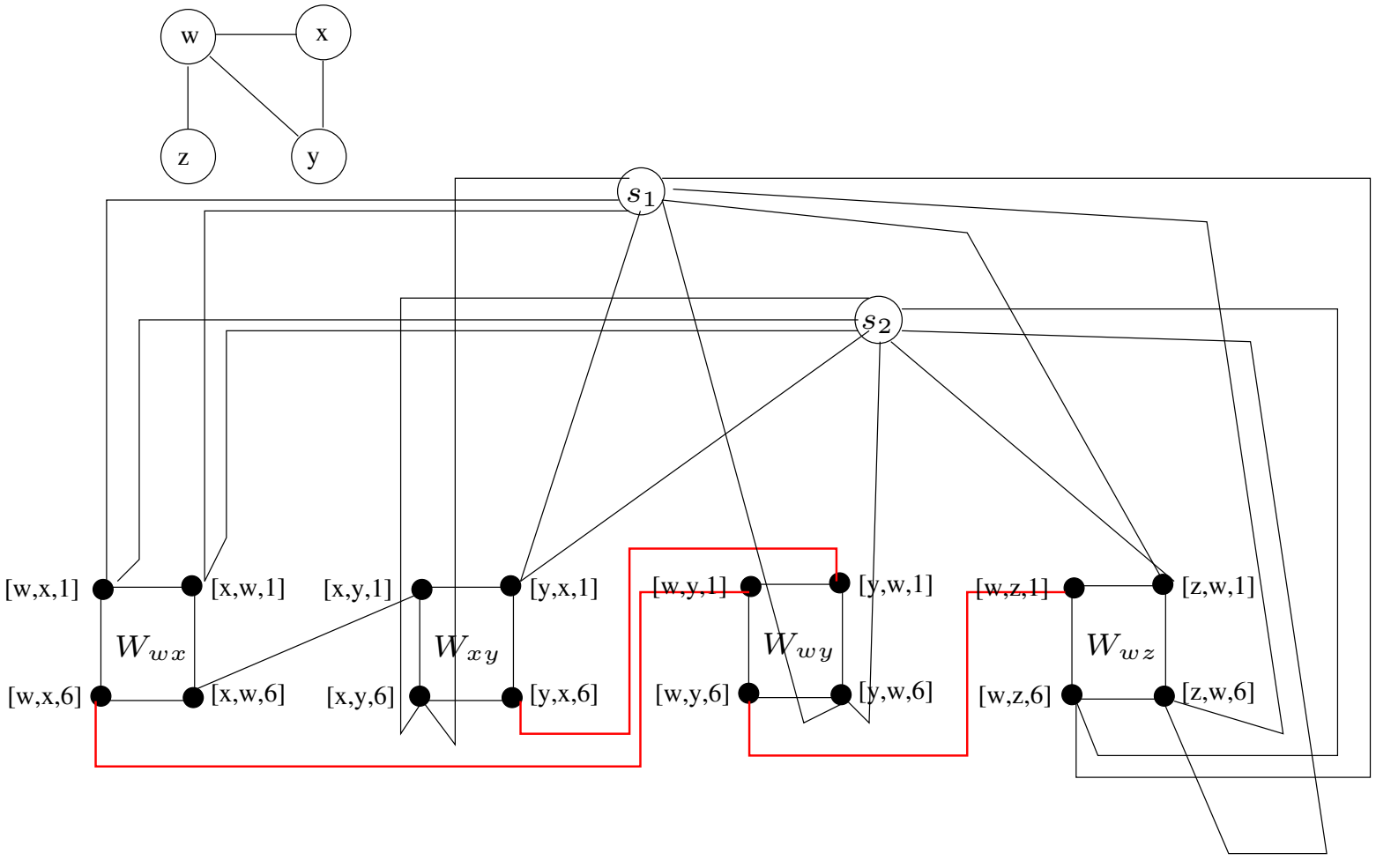
Ciclo Hamiltoniano

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



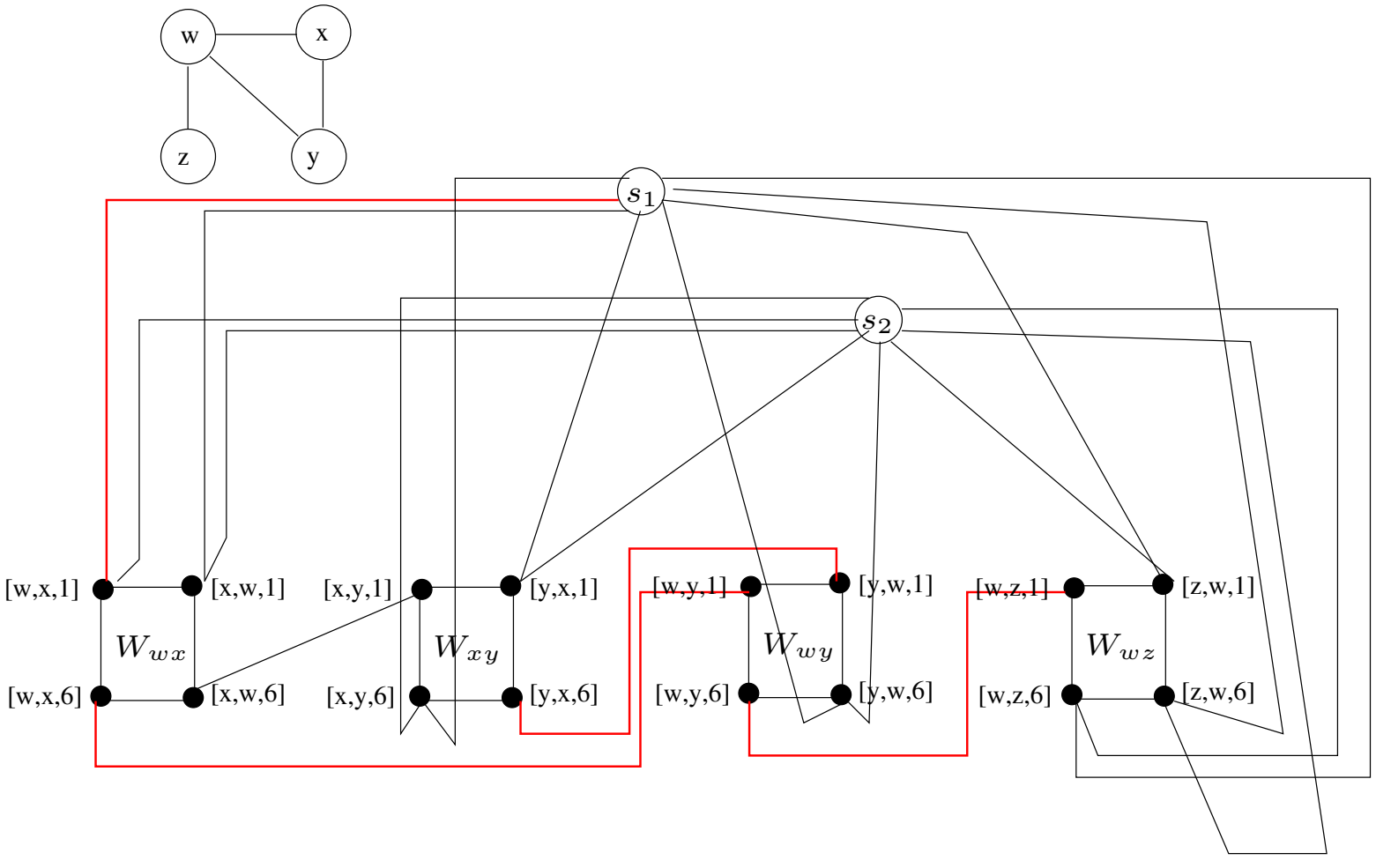
Ciclo Hamiltoniano

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



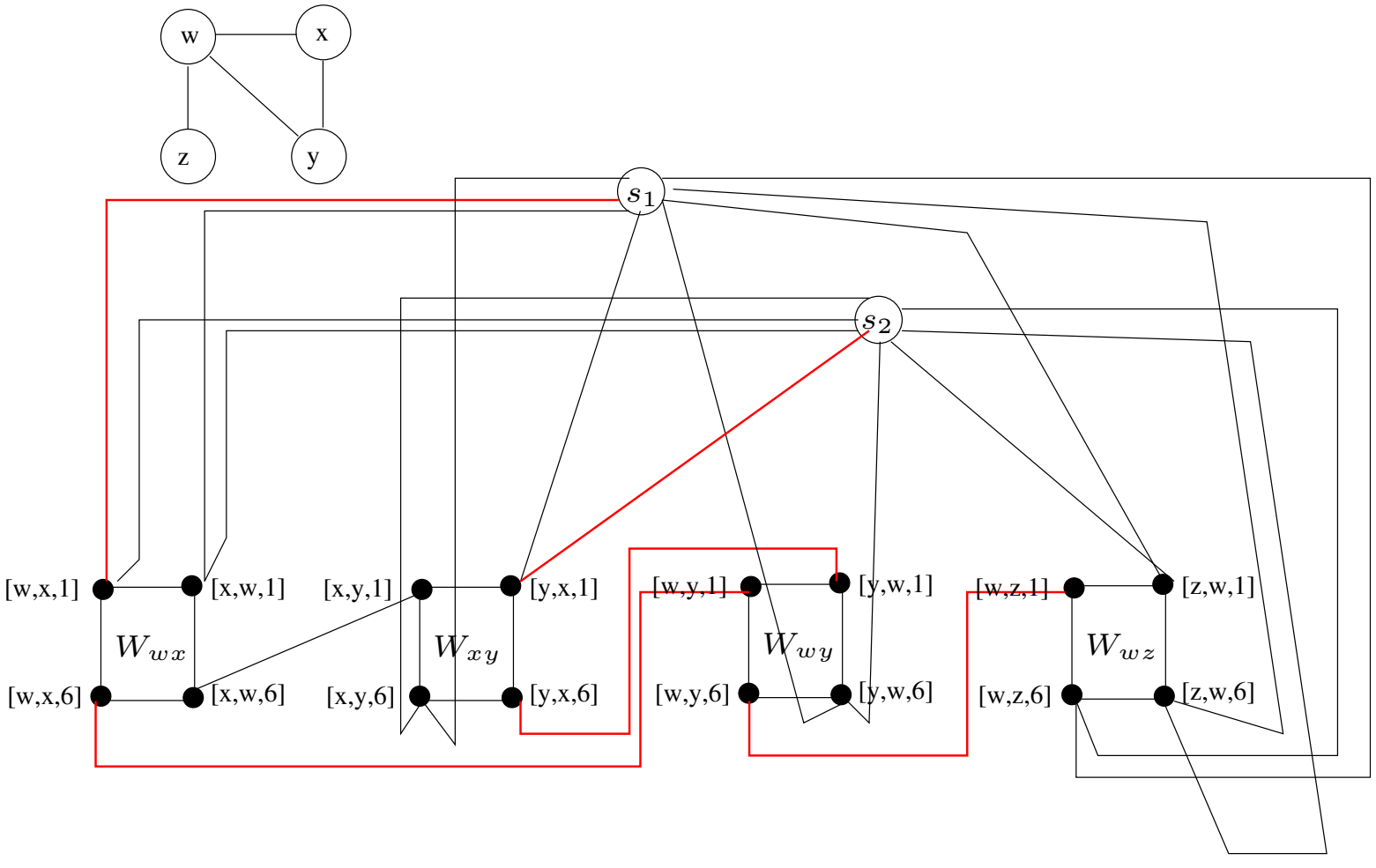
Ciclo Hamiltoniano

Incluir arestas (2) para *w*:



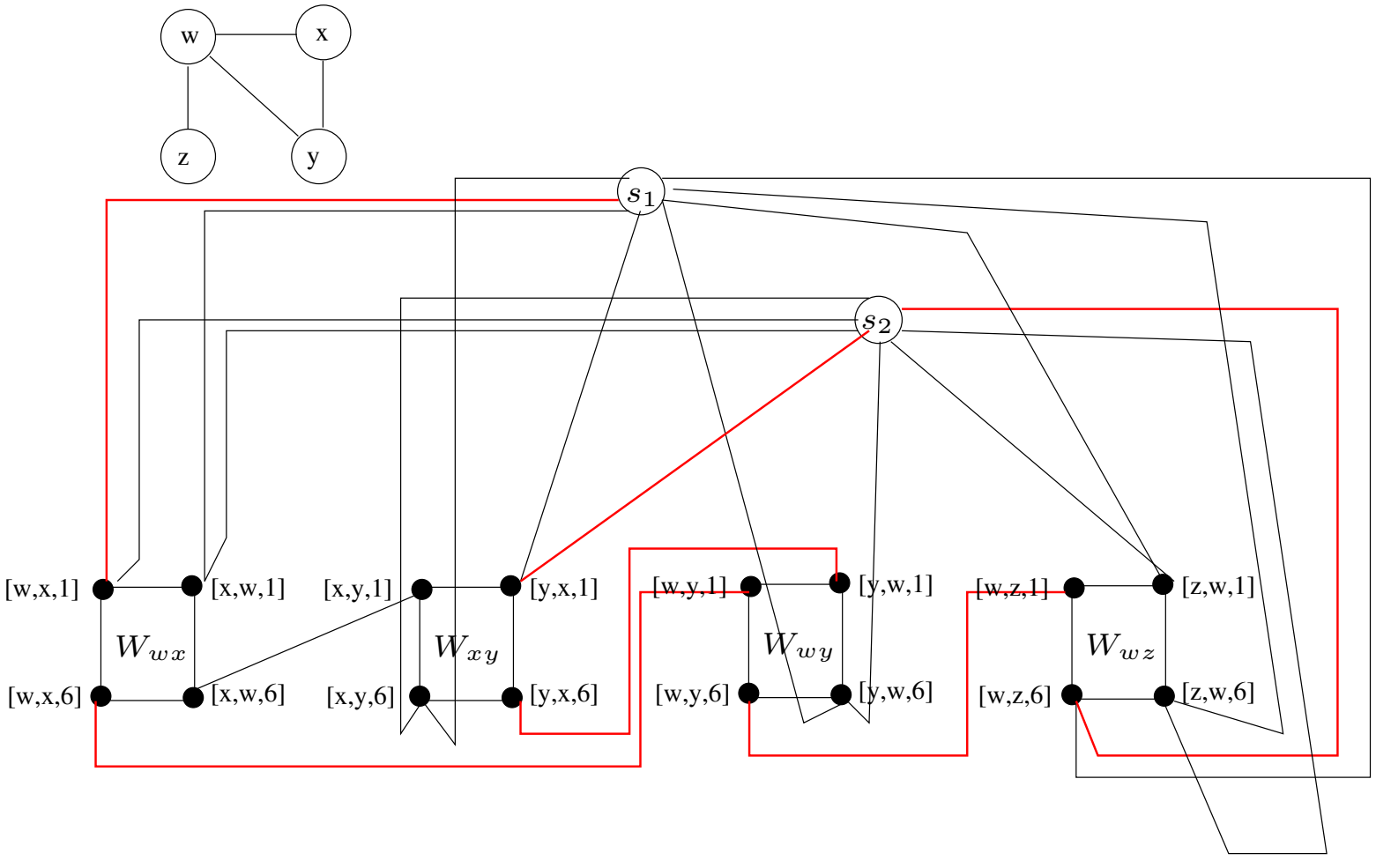
Ciclo Hamiltoniano

Incluir arestas (2) para *y*:



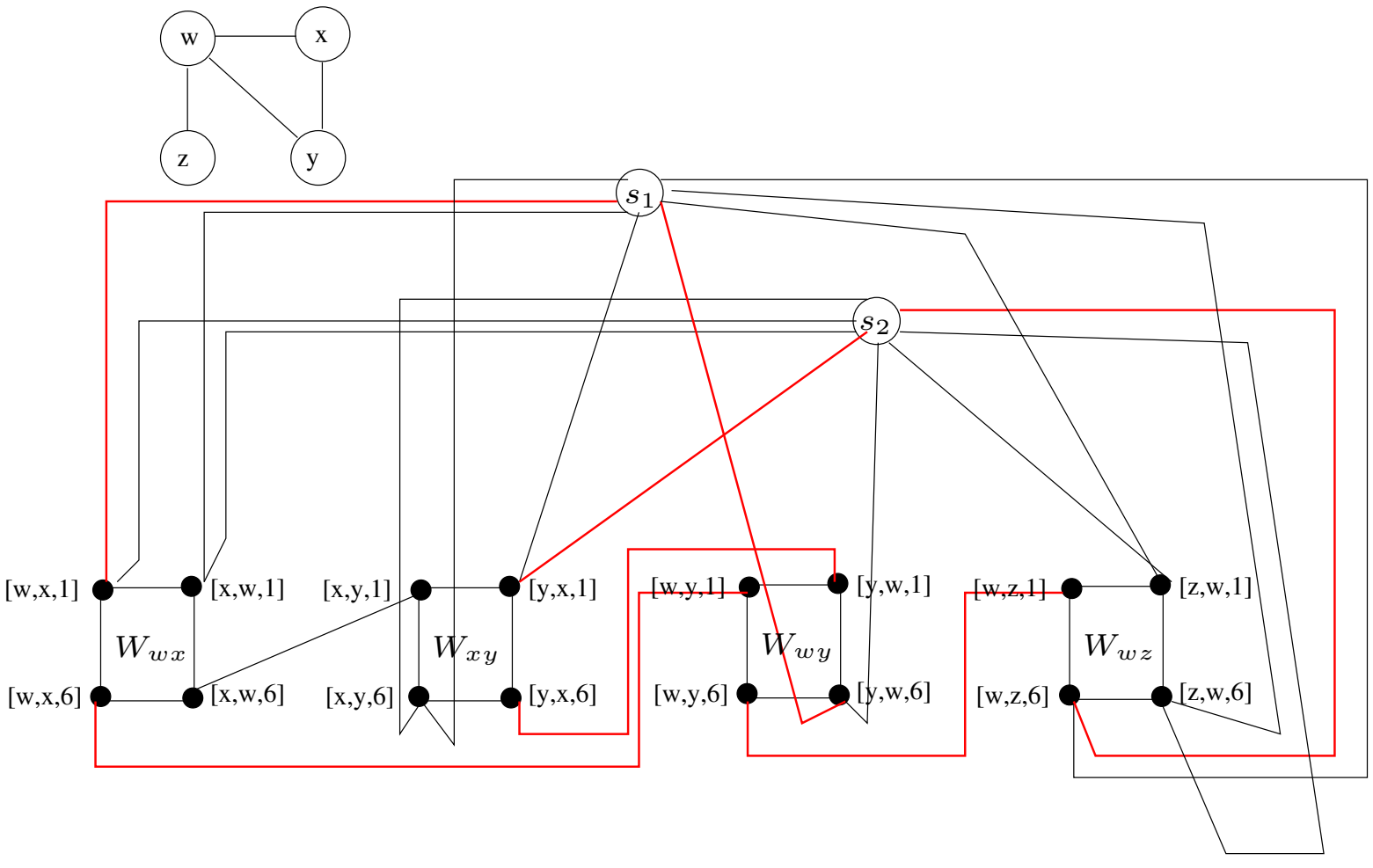
Ciclo Hamiltoniano

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

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.

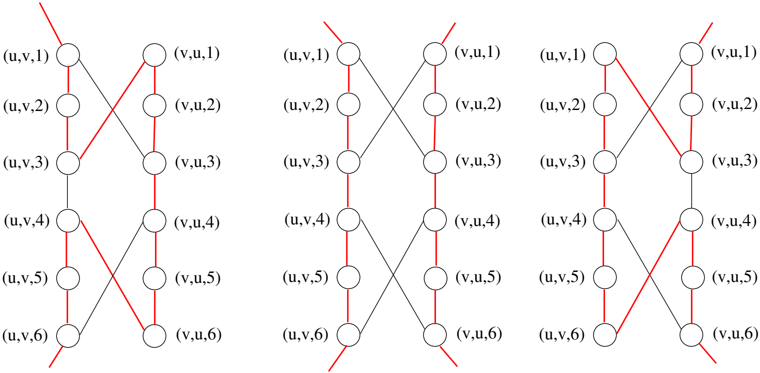
Ciclo Hamiltoniano

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 ideia 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.

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 ideia 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.

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 ideia 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.

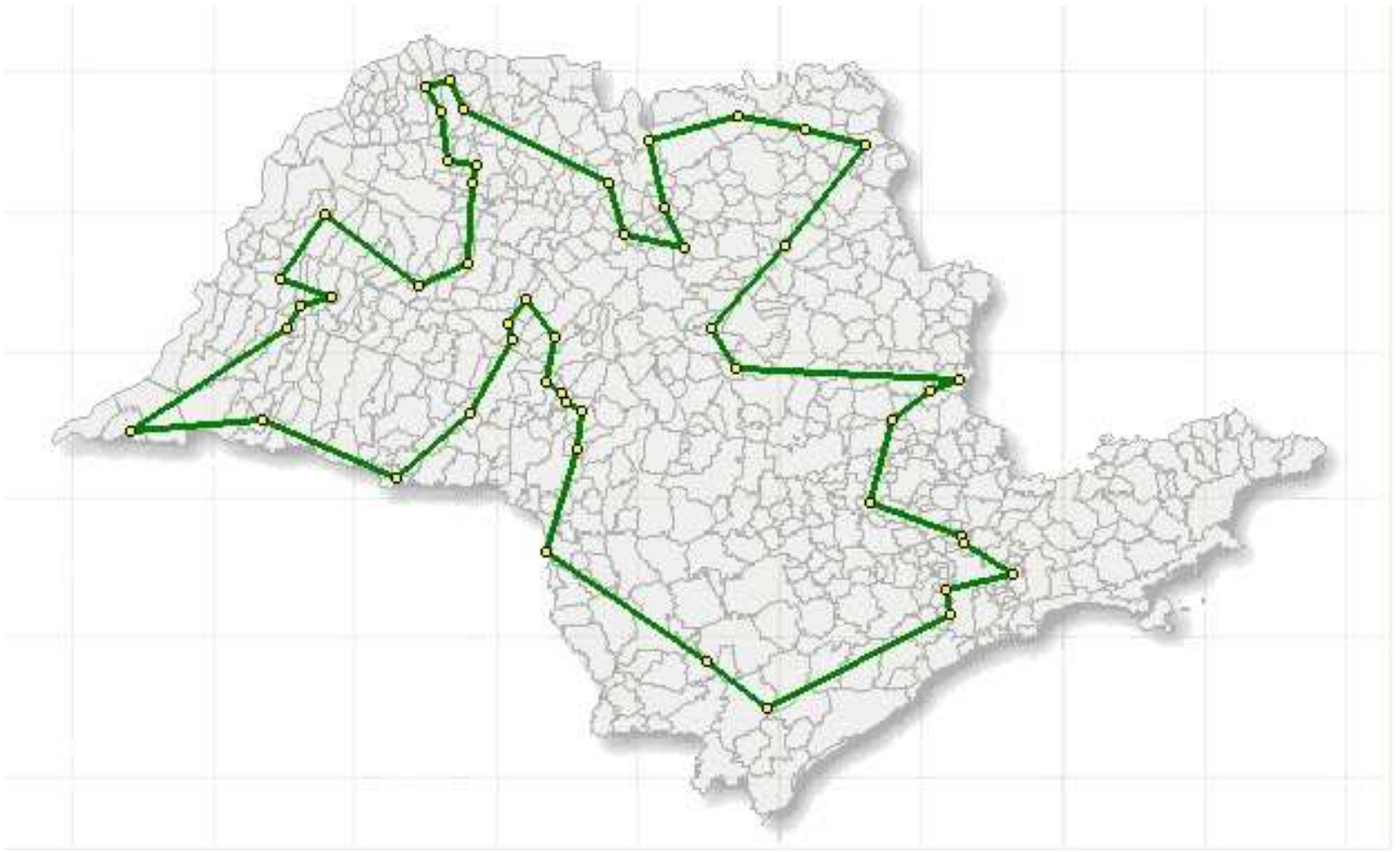
Demonstrando NP-completude

- Caixeiro Viajante (TSP)

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)



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.

- Sabemos que HAM-CICLO é NP-Completo
- 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.

Outros Tópicos de Complexidade

Outros Tópicos de Complexidade

- Complexidade de Espaço

Complexidade de Espaço

- Da mesma forma como avaliamos algoritmos em termos de tempo, podemos avalia-los em termos de espaço utilizado.

Definição

O espaço (memória) utilizado por um algoritmo determinístico corresponde ao número de células (bits) que este acessa durante sua execução.

Definição

O espaço utilizado por um algoritmo não-determinístico é o número de células acessadas em um ramo mais curto de execução da árvore até o estado aceita.

Complexidade de Espaço

Definição

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

Definição

$$NSPACE(f(n)) = \{L : L \text{ é uma linguagem decidida não-deterministicamente em espaço } O(f(n))\}$$

- Podemos mostrar por exemplo que linguagens em NP gastam espaço polinomial.

Complexidade de Espaço

Definição

PSPACE são as linguagens que podem ser decididas por algoritmos determinísticos que usam espaço polinomial:

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

Definição

NPSPACE são as linguagens que podem ser decididas por algoritmos não-determinísticos que usam espaço polinomial:

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

Complexidade de Espaço

Teorema

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

$$NSPACE(f(n)) \subseteq SPACE(f^2(n))$$

- O teorema nos diz que as linguagens decididas por algoritmos **não-determinísticos** com $f(n)$ de espaço podem ser decididas por algoritmos **determinísticos** com espaço $f^2(n)$.

Complexidade de Espaço

Teorema

PSPACE = NPSPACE.

Prova. É claro que PSPACE $\subseteq$ NPSPACE.

Se $L \in$ NPSPACE então ela é decidida não deterministicamente em espaço $O(n^k)$ para uma constante k.

Pelo teo. de Savitch L pode ser decidida por alg. determinístico com espaço $O(n^{2k})$ que é polinomial. $\square$

Complexidade de Espaço

- Sabemos que existe a seguinte relação entre as classes:

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

Outros Tópicos de Complexidade

- Indecibilidade

Indecibilidade

- Até então nos preocupamos com o "esforço" necessário que um algoritmo tem para resolver um problema.
- Mas existem problemas para os quais não há algoritmos!
- Tais problemas são ditos indecidíveis pois não há algoritmo que decide o problema.

Indecibilidade

Definição

Problema da Parada: Dada uma string s e um algoritmo A, o algoritmo A para quando recebe s como entrada?

- Uma primeira ideia:
  - Podemos tentar criar um algoritmo A' que simule/emule o funcionamento de A sobre a entrada s
  - Mas não sabemos se A para ou não, então não sabemos quanto parar a simulação
- De fato, vamos mostrar que este problema é indecidível!

Prova. No quadro.

Indecibilidade

Definição

Problema da Parada: Dada uma string s e um algoritmo A, o algoritmo A para quando recebe s como entrada?

- Uma primeira ideia:
  - Podemos tentar criar um algoritmo A' que simule/emule o funcionamento de A sobre a entrada s
  - Mas não sabemos se A para ou não, então não sabemos quanto parar a simulação
- De fato, vamos mostrar que este problema é indecidível!

Prova. No quadro.

Indecibilidade

Definição

Problema da Parada: Dada uma string s e um algoritmo A, o algoritmo A para quando recebe s como entrada?

- Uma primeira ideia:
  - Podemos tentar criar um algoritmo A' que simule/emule o funcionamento de A sobre a entrada s
  - Mas não sabemos se A para ou não, então não sabemos quanto parar a simulação
- De fato, vamos mostrar que este problema é indecidível!

Prova. No quadro.

Indecibilidade

Definição

Problema da Parada: Dada uma string s e um algoritmo A, o algoritmo A para quando recebe s como entrada?

- Uma primeira ideia:
  - Podemos tentar criar um algoritmo A' que simule/emule o funcionamento de A sobre a entrada s
  - Mas não sabemos se A para ou não, então não sabemos quanto parar a simulação
- De fato, vamos mostrar que este problema é indecidível!

Prova. No quadro.

Indecibilidade

Definição

Problema da Parada: Dada uma string s e um algoritmo A, o algoritmo A para quando recebe s como entrada?

- Uma primeira ideia:
  - Podemos tentar criar um algoritmo A' que simule/emule o funcionamento de A sobre a entrada s
  - Mas não sabemos se A para ou não, então não sabemos quanto parar a simulação
- De fato, vamos mostrar que este problema é indecidível!

Prova. No quadro.