TEORIA DE COMPLEXIDADE

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO 1º SEMESTRE DE 2008

Antonio Alfredo Ferreira Loureiro

loureiro@dcc.ufmg.br

http://www.dcc.ufmg.br/~loureiro

Introdução

- Problemas intratáveis ou difíceis são comuns na natureza e nas áreas do conhecimento.
- Problemas podem ser classificados em:
  - "fáceis": resolvidos por algoritmos polinomiais.
  - "difíceis": os algoritmos conhecidos para resolvê-los são exponenciais.
- Complexidade de tempo da maioria dos problemas é polinomial ou exponencial.
- **Polinomial**: função de complexidade é $O(p(n))$, onde $p(n)$ é um polinômio.
  - Exemplos: pesquisa binária ($O(\log n)$), pesquisa seqüencial ($O(n)$), ordenação por inserção ($O(n^2)$), e multiplicação de matrizes ($O(n^3)$).
- **Exponencial**: função de complexidade é $O(c^n)$, $c > 1$.
  - Exemplo: **Problema do Caixeiro Viajante** (PCV) ($O(n!)$).
  - Mesmo problemas de tamanho pequeno a moderado não podem ser resolvidos por algoritmos não-polinomiais.

Problemas $\mathcal{NP}$-Completo

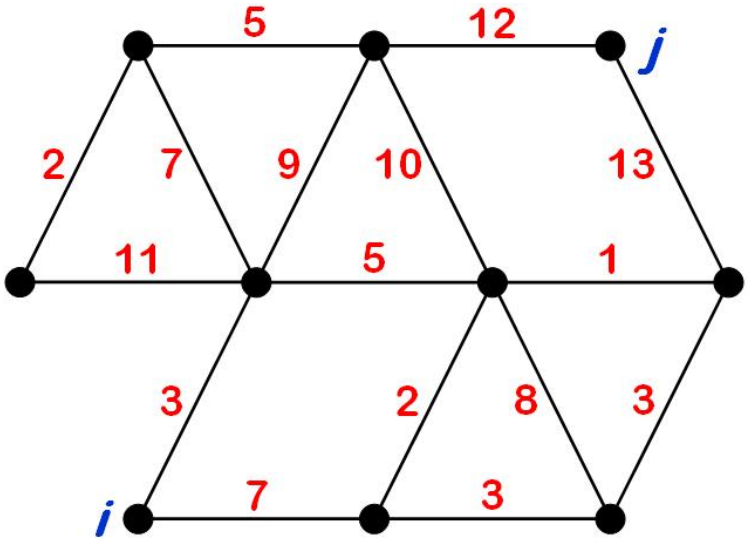
- A teoria de complexidade a ser apresentada não mostra como obter algoritmos polinomiais para problemas que demandam algoritmos exponenciais, nem afirma que não existem.
- É possível mostrar que os problemas para os quais não há algoritmo polinomial conhecido são computacionalmente relacionados.
- Formam a classe conhecida como $\mathcal{NP}$.
- Propriedade: um problema da classe $\mathcal{NP}$ poderá ser resolvido em tempo polinomial se e somente se todos os outros problemas em $\mathcal{NP}$ também puderem.
- Este fato é um forte indício de que dificilmente alguém será capaz de encontrar um algoritmo eficiente para um problema da classe $\mathcal{NP}$.

Classe $\mathcal{NP}$: Problemas "Sim/Não"

- Para o estudo teórico da complexidade de algoritmos consideram-se problemas cujo resultado da computação seja "sim" ou "não".
- Versão do Problema do Caixeiro Viajante (PCV) cujo resultado é do tipo "sim/não":
  - Dados: uma constante k, um conjunto de cidades $C = \{c_1, c_2, \cdots, c_n\}$ e uma distância $d(c_i, c_j)$ para cada par de cidades $c_i, c_j \in C$.
  - Questão: Existe um "roteiro" para todas as cidades em C cujo comprimento total seja menor ou igual a k?
- Característica da classe $\mathcal{NP}$: problemas "sim/não" para os quais uma dada solução pode ser verificada facilmente.
- A solução pode ser muito difícil ou impossível de ser obtida, mas uma vez conhecida ela pode ser verificada em tempo polinomial.

Caminho em um Grafo

- Considere um grafo com peso, dois vértices i, j e um inteiro $k > 0$.



- *Fácil*: Existe um caminho de i até j com peso $\leq k$?.
  - Há um algoritmo eficiente com complexidade de tempo $O(E \log V)$, sendo E o número de arestas e V o número de vértices (algoritmo de Dijkstra).

- *Difícil*: Existe um caminho de i até j com peso $\geq k$?
  - Não se conhece algoritmo eficiente. É equivalente ao PCV em termos de complexidade.

Coloração de um Grafo

- Em um grafo $G = (V, A)$, mapear $C : V \rightarrow S$, sendo S um conjunto finito de cores tal que se $\overline{vw} \in A$ então $c(v) \neq c(w)$ (vértices adjacentes possuem cores distintas).
- O número cromático $X(G)$ de G é o menor número de cores necessário para colorir G, isto é, o menor k para o qual existe uma coloração C para G e $|C(V)| = k$.
- O problema é produzir uma coloração ótima, que é a que usa apenas $X(G)$ cores.
- Formulação do tipo "sim/não": dados G e um inteiro positivo k, existe uma coloração de G usando k cores?
  - *Fácil*: $k = 2$.
  - *Difícil*: $k > 2$.
- Aplicação: modelar problemas de agrupamento (*clustering*) e de horário (*scheduling*).

Coloração de um grafo: Otimização de compiladores

- Escalonar o uso de um número finito de registradores (idealmente com o número mínimo).
- No trecho de programa a ser otimizado, cada variável tem intervalos de tempo em que seu valor tem de permanecer inalterado, como depois de inicializada e antes do uso final.
- Variáveis com interseção nos tempos de vida útil não podem ocupar o mesmo registrador.
- Modelagem por grafo: vértices representam variáveis e cada aresta liga duas variáveis que possuem interseção nos tempos de vida.
- Coloração dos vértices: atribui cada variável a um agrupamento (ou classe). Duas variáveis com a mesma cor não colidem, podendo assim ser atribuídas ao mesmo registrador.

Coloração de um grafo: Otimização de compiladores

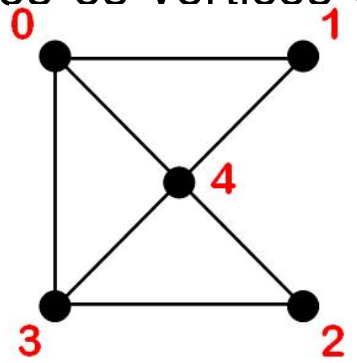
- Evidentemente, não existe conflito se cada vértice for colorido com uma cor distinta.
- O objetivo porém é encontrar uma coloração usando o mínimo de cores (computadores têm um número limitado de registradores).
- **Número cromático**: menor número de cores suficientes para colorir um grafo.

Coloração de um grafo: Problema do horário

- Suponha que os exames finais de um curso tenham de ser realizados em uma única semana.
- Disciplinas com alunos de cursos diferentes devem ter seus exames marcados em horários diferentes.
- Dadas uma lista de todos os cursos e outra lista de todas as disciplinas cujos exames não podem ser marcados no mesmo horário, o problema em questão pode ser modelado como um problema de coloração de grafos.

Circuito Hamiltoniano

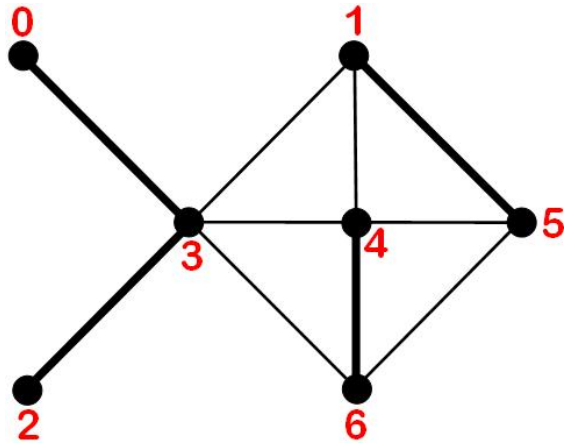
- **Circuito Hamiltoniano**: passa por todos os vértices uma única vez e volta ao vértice inicial (ciclo simples).
- **Caminho Hamiltoniano**: passa por todos os vértices uma única vez (ciclo simples).



- Exemplos:
  Circuito Hamiltoniano: 0 1 4 2 3 0
  Caminho Hamiltoniano: 0 1 4 2 3
- Existe um ciclo de Hamilton no grafo G?
  - *Fácil*: Grafos com grau máximo = 2.
  - *Difícil*: Grafos com grau > 2.
- É um caso especial do PCV.
  - Pares de vértices com uma aresta entre eles tem distância 1; e
  - Pares de vértices sem aresta entre eles têm distância infinita.

Cobertura de arestas

- Uma **cobertura de arestas** de um grafo $G = (V, A)$ é um subconjunto $A' \subset A$ de k arestas tal que todo $v \in V$ é parte de pelo menos uma aresta de A'.
- O conjunto resposta para $k = 4$ é $A' = \{(0,3), (2,3), (4,6), (1,5)\}$.



- Uma **cobertura de vértices** é um subconjunto $V' \subset V$ tal que se $(u, v) \in A$ então $u \in V'$ ou $v \in V'$, isto é, cada aresta do grafo é incidente em um dos vértices de V'.
- Na figura, o conjunto resposta é $V' = \{3, 4, 5\}$, para $k = 3$.
- Dados um grafo e um inteiro $k > 0$
  - *Fácil*: há uma cobertura de arestas $\leq k$?.
  - *Difícil*: há uma cobertura de vértices $\leq k$?

Algoritmos não-determinísticos

- **Algoritmos determinísticos**: o resultado de cada operação é definido de forma única.
- Em um arcabouço teórico, é possível remover essa restrição.
- Apesar de parecer irreal, este é um conceito importante e geralmente utilizado para definir a classe $\mathcal{NP}$.
- Neste caso, os algoritmos podem conter operações cujo resultado não é definido de forma única.
- **Algoritmo não-determinístico**: capaz de escolher uma dentre as várias alternativas possíveis a cada passo.
- Algoritmos não-determinísticos contêm operações cujo resultado não é unicamente definido, ainda que limitado a um conjunto especificado de possibilidades.

Função *escolhe(C)*

- Algoritmos não-determinísticos utilizam uma função *escolhe(C)*, que escolhe um dos elementos do conjunto C de forma arbitrária.
- O comando de atribuição X $\leftarrow$ *escolhe* (1:n) pode resultar na atribuição a X de qualquer dos inteiros no intervalo $[1, n]$.
- A complexidade de tempo para cada chamada da função *escolhe* é $O(1)$.
- Neste caso, não existe nenhuma regra especificando como a escolha é realizada.
- Se um conjunto de possibilidades levam a uma resposta, este conjunto é escolhido sempre e o algoritmo terminará com sucesso.
- Por outro lado, um algoritmo não-determinístico termina sem sucesso se e somente se não há um conjunto de escolhas que indica sucesso.

Comandos *sucesso* e *insucesso*

- Algoritmos não-determinísticos utilizam também dois comandos, a saber:
  - *insucesso*: indica término sem sucesso.
  - *sucesso*: indica término com sucesso.
- Os comandos *insucesso* e *sucesso* são usados para definir uma execução do algoritmo.
- Esses comandos são equivalentes a um comando de parada de um algoritmo determinístico.
- Os comandos *insucesso* e *sucesso* também têm complexidade de tempo $O(1)$.

Máquina Não-Determinística

- Uma máquina capaz de executar a função *escolhe* admite a capacidade de **computação não-determinística**.
- Uma máquina não-determinística é capaz de produzir cópias de si mesma quando diante de duas ou mais alternativas, e continuar a computação independentemente para cada alternativa.
- A máquina não-determinística que acabamos de definir não existe na prática, mas ainda assim fornece fortes evidências de que certos problemas não podem ser resolvidos por algoritmos determinísticos em tempo polinomial, conforme mostrado na definição da classe $\mathcal{NP}$-completo à frente.

Pesquisa Não-Determinística

- Pesquisa o elemento x em um conjunto de elementos $A[1:n]$, $n \geq 1$.

```
PESQUISAND(A, 1, n)
1  j ← ESCOLHE(A, 1, n)
2  if A[j] = x
3     then sucesso
4     else insucesso
```

- Determina um índice j tal que $A[j] = x$ para um término com sucesso ou então insucesso quando x não está presente em A.
- O algoritmo tem complexidade não-determinística $O(1)$.
- Para um algoritmo determinístico a complexidade é $\Omega(n)$.

Ordenação Não-Determinística

- Ordena um conjunto $A[1:n]$ contendo n inteiros positivos, $n \geq 1$.

```
ORDENAND(A, 1, n)
1  for i ← 1 to n do B[i] ← 0
2  for i ← 1 to n
3  do j ← ESCOLHE(A, 1, n)
4      if B[j] = 0
5        then B[j] ← A[i]
6        else insucesso
```

- Um vetor auxiliar B[1:n] é utilizado. Ao final, B contém o conjunto em ordem crescente.
- A posição correta em B de cada inteiro de A é obtida de forma não-determinística pela função escolhe.
- Em seguida, o comando de decisão verifica se a posição $B[j]$ ainda não foi utilizada.
- A complexidade é $O(n)$. (Para um algoritmo determinístico a complexidade é $\Omega(n \log n)$)

Problema da Satisfabilidade

- Considere um conjunto de **variáveis booleanas** $x_1, x_2, \cdots, x_n$, que podem assumir valores lógicos *verdadeiro* ou *falso*.
- A negação de x_i é representada por $\overline{x_i}$.
- Expressão booleana: variáveis booleanas e operações **ou** ($\vee$) e **e** ($\wedge$) (também chamadas respectivamente de adição e multiplicação).
- Uma expressão booleana E contendo um produto de adições de variáveis booleanas é dita estar na **forma normal conjuntiva**.
- Dada E na forma normal conjuntiva, com variáveis x_i, $1 \leq i \leq n$, existe uma atribuição de valores verdadeiro ou falso às variáveis que torne E verdadeira ("satisfaça")?
- $E_1 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3} \vee x_2) \wedge (x_3)$ é *satisfatível* ($x_1 = F$, $x_2 = V$, $x_3 = V$).
- A expressão $E_2 = x_1 \wedge \overline{x_1}$ não é *satisfatível*.

Problema da Satisfabilidade

- O algoritmo AvalND(E,n) verifica se uma expressão E na forma normal conjuntiva, com variáveis x_i, $1 \leq i \leq n$, é *satisfatível*.

```
AVALND(E, n)
1  for i ← 1 to n
2  do x_i ← ESCOLHE(true, false)
3      if E(x_1, x_2, ··· , x_n)
4        then sucesso
5        else insucesso
```

- O algoritmo obtém uma das 2^n atribuições possíveis de forma não-determinística em $O(n)$.
- Melhor algoritmo determinístico: $O(2^n)$.
- Aplicação: definição de circuitos elétricos combinatórios que produzam valores lógicos como saída e sejam constituídos de portas lógicas **e**, **ou** e **não**.
- Neste caso, o mapeamento é direto, pois o circuito pode ser descrito por uma expressão lógica na forma normal conjuntiva.

Caracterização das Classes $\mathcal{P}$ e $\mathcal{NP}$

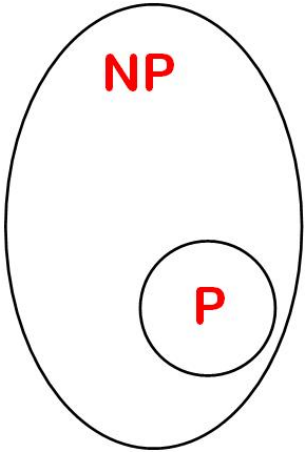
- $\mathcal{P}$: conjunto de todos os problemas que podem ser resolvidos por *algoritmos determinísticos* em tempo *polinomial*.
- $\mathcal{NP}$: conjunto de todos os problemas que podem ser resolvidos por *algoritmos não-determinísticos* em tempo *polinomial*.
- Para mostrar que um determinado problema está em $\mathcal{NP}$, basta apresentar um algoritmo não-determinístico que execute em tempo polinomial para resolver o problema.
- Outra maneira é encontrar um algoritmo determinístico polinomial para verificar que uma dada solução é válida.

Existe Diferença entre $\mathcal{P}$ e $\mathcal{NP}$?

- $\mathcal{P} \subseteq \mathcal{NP}$, pois algoritmos determinísticos são um caso especial dos não-determinísticos.
- A questão é se $\mathcal{P} = \mathcal{NP}$ ou $\mathcal{P} \neq \mathcal{NP}$.
- Esse é o problema não resolvido mais famoso que existe na área de ciência da computação.
- Se existem algoritmos polinomiais determinísticos para todos os problemas em $\mathcal{NP}$, então $\mathcal{P} = \mathcal{NP}$.
- Por outro lado, a prova de que $\mathcal{P} \neq \mathcal{NP}$ parece exigir técnicas ainda desconhecidas.

Existe Diferença entre $\mathcal{P}$ e $\mathcal{NP}$?

- Descrição tentativa do mundo $\mathcal{NP}$, em que a classe $\mathcal{P}$ está contida na classe $\mathcal{NP}$.



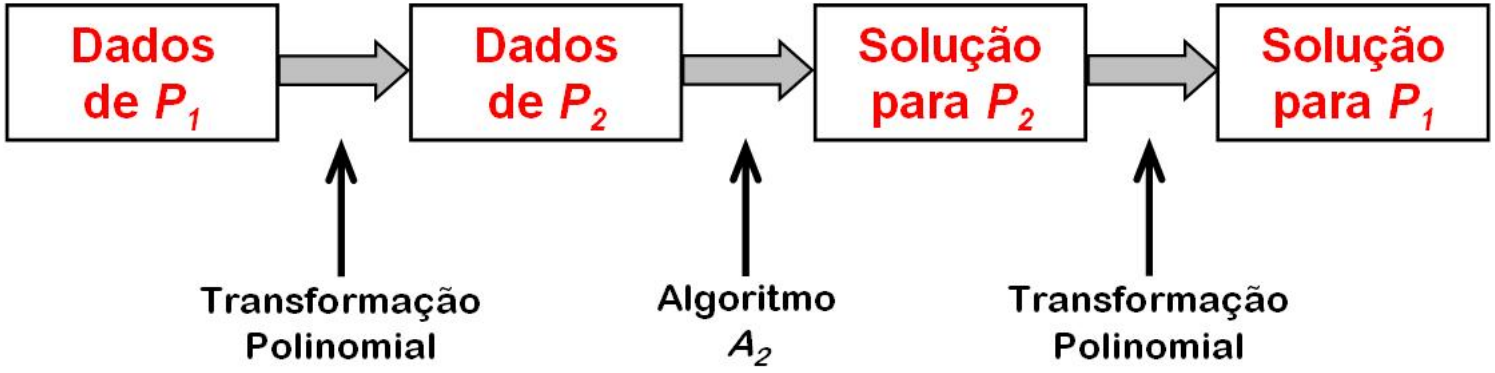
- Acredita-se que $\mathcal{NP} \gg \mathcal{P}$, pois para muitos problemas em $\mathcal{NP}$, não existem algoritmos polinomiais conhecidos, nem um **limite inferior não-polinomial** provado.

$\mathcal{NP} \supset \mathcal{P}$ ou $\mathcal{NP} = \mathcal{P}$?
Conseqüências

- Muitos problemas práticos em $\mathcal{NP}$ podem ou não pertencer a $\mathcal{P}$ (não conhecemos nenhum algoritmo determinístico eficiente para eles).
- Se conseguirmos provar que um problema não pertence a $\mathcal{P}$, então não precisamos procurar por uma solução eficiente para ele.
- Como não existe tal prova, sempre há esperança de que alguém descubra um algoritmo eficiente.
- Quase ninguém acredita que $\mathcal{NP} = \mathcal{P}$.
- Existe um esforço considerável para provar o contrário, mas a questão continua em aberto!

Transformação Polinomial

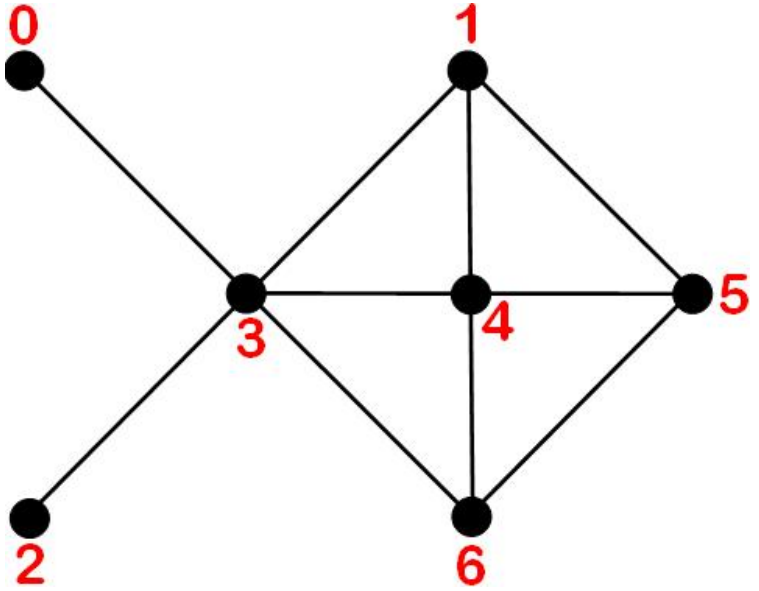
- Sejam P_1 e P_2 dois problemas "sim/não".
- Suponha que um algoritmo A_2 resolva P_2.
- Se for possível transformar P_1 em P_2 e a solução de P_2 em solução de P_1, então A_2 pode ser utilizado para resolver P_1.
- Se pudermos realizar as transformações nos dois sentidos em tempo polinomial, então P_1 é *polinomialmente transformável* em P_2.



- Esse conceito é importante para definir a classe $\mathcal{NP}$-completo.
- Para mostrar um exemplo de transformação polinomial, definiremos clique de um grafo e conjunto independente de vértices de um grafo.

Conjunto Independente de Vértices de um Grafo

- O conjunto independente de vértices de um grafo $G = (V, A)$ é constituído do subconjunto $V' \subseteq V$, tal que $v, w \in V' \Rightarrow (v, w) \notin A$.
- Todo par de vértices de V' é não adjacente (V' é um subgrafo totalmente desconectado).
- Exemplo de cardinalidade 4: $V' = \{0, 2, 1, 6\}$.

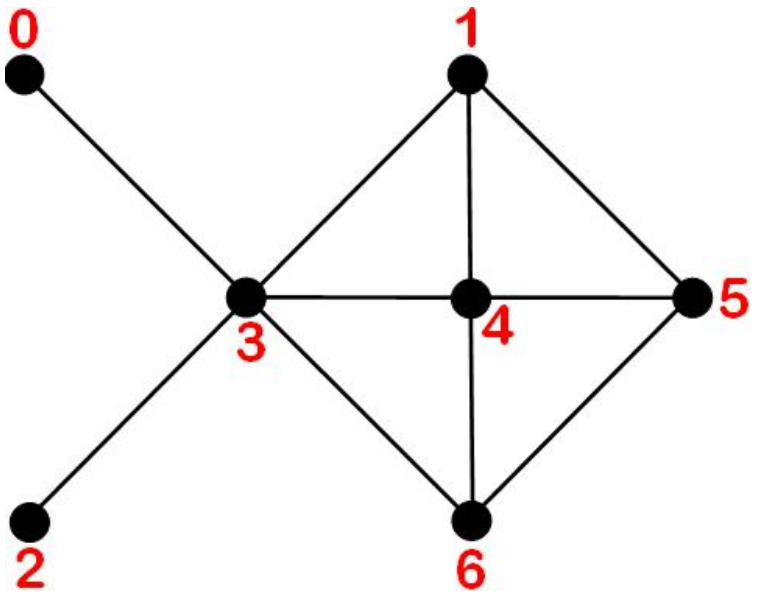


Conjunto Independente de Vértices: Aplicação

- Em problemas de dispersão é necessário encontrar grandes conjuntos independentes de vértices. Procura-se um conjunto de pontos mutuamente separados.
- Exemplo: identificar localizações para instalação de franquias.
- Duas localizações não podem estar perto o suficiente para competirem entre si.
- Solução: construir um grafo em que possíveis localizações são representadas por vértices, e arestas são criadas entre duas localizações que estão próximas o suficiente para interferir.
- O maior conjunto independente fornece o maior número de franquias que podem ser concedidas sem prejudicar as vendas.
- Em geral, conjuntos independentes evitam conflitos entre elementos.

Clique de um grafo

- **Clique** de um grafo $G = (V, A)$ é constituído do subconjunto $V' \subseteq V$, tal que $v, w \in V' \Rightarrow (v, w) \in A$.
- Todo par de vértices de V' é adjacente (V' é um subgrafo completo).
- Exemplo de cardinalidade 3: $V' = \{3, 1, 4\}$.



Clique de um grafo: Aplicação

- O problema de identificar agrupamentos de objetos relacionados freqüentemente se reduz a encontrar grandes cliques em grafos.
- Exemplo: empresa de fabricação de peças por meio de injeção plástica que fornece para diversas outras empresas montadoras.
- Para reduzir o custo relativo ao tempo de preparação das máquinas injetoras, pode-se aumentar o tamanho dos lotes produzidos para cada peça encomendada.
- É preciso identificar os clientes que adquirem os mesmos produtos, para negociar prazos de entrega comuns e assim aumentar o tamanho dos lotes produzidos.
- Solução: construir um grafo com cada vértice representando um cliente e ligar com uma aresta os que adquirem os mesmos produtos.
- Um clique no grafo representa o conjunto de clientes que adquirem os mesmos produtos.

Transformação Polinomial

- Considere P_1 o problema clique e P_2 o problema conjunto independente de vértices.
- A instância I de clique consiste de um grafo $G = (V, A)$ e um inteiro $k > 0$.
- A instância $f(I)$ de conjunto independente pode ser obtida considerando-se o grafo complementar $\overline{G}$ de G e o mesmo inteiro k.
- $f(I)$ é uma transformação polinomial:
  1. $\overline{G}$ pode ser obtido a partir de G em tempo polinomial.
  2. G possui clique de tamanho $\geq k$ se e somente se $\overline{G}$ possui conjunto independente de vértices de tamanho $\geq k$.

Transformação Polinomial

- Se existe um algoritmo que resolve o conjunto independente em tempo polinomial, ele pode ser utilizado para resolver clique também em tempo polinomial.
- Diz-se que clique $\propto$ conjunto independente.
- Denota-se $P_1 \propto P_2$ para indicar que P_1 é polinomialmente transformável em P_2.
- A relação $\propto$ é transitiva:

$$((P_1 \propto P_2) \wedge (P_2 \propto P_3)) \rightarrow P_1 \propto P_3.$$

Problemas $\mathcal{NP}$-Completo e $\mathcal{NP}$-Difícil

- Dois problemas P_1 e P_2 são **polinomialmente equivalentes** se e somente se $P_1 \propto P_2$ e $P_2 \propto P_1$.
- Exemplo: **problema da *satisfabilidade*** (SAT). Se $SAT \propto P_1$ e $P_1 \propto P_2$, então $SAT \propto P_2$.
- Um problema P é $\mathcal{NP}$**-difícil** se e somente se $SAT \propto P$ (*satisfabilidade* é redutível a P).
- Um problema de decisão P é denominado $\mathcal{NP}$**-completo** quando:
  1. $\mathcal{P} \in \mathcal{NP}$.
  2. Todo problema de decisão $P' \in \mathcal{NP}$-completo satisfaz $P' \propto P$.

Problemas $\mathcal{NP}$-Completo e $\mathcal{NP}$-Difícil

- Um problema de decisão P que seja $\mathcal{NP}$-difícil pode ser mostrado ser $\mathcal{NP}$-completo exibindo um algoritmo não-determinístico polinomial para P.
- Apenas problemas de decisão ("sim/não") podem ser $\mathcal{NP}$-completo.
- Problemas de otimização podem ser $\mathcal{NP}$-difícil, mas geralmente, se P_1 é um problema de decisão e P_2 um problema de otimização, é bem possível que $P_1 \propto P_2$.
- A dificuldade de um problema $\mathcal{NP}$-difícil não é menor do que a dificuldade de um problema $\mathcal{NP}$-completo.

Exemplo: Problema da Parada

- É um exemplo de problema $\mathcal{NP}$-difícil que não é $\mathcal{NP}$-completo.
- Consiste em determinar, para um algoritmo determinístico qualquer A com entrada de dados E, se o algoritmo A termina (ou entra em um *loop* infinito).
- É um problema **indecidível**. Não há algoritmo de qualquer complexidade para resolvê-lo.
- Mostrando que $SAT \propto$ problema da parada:
  - Considere o algoritmo A cuja entrada é uma expressão booleana na forma normal conjuntiva com n variáveis.
  - Basta tentar 2^n possibilidades e verificar se E é *satisfatível*.
  - Se for, A pára; senão, entra em *loop*.
  - Logo, o problema da parada é $\mathcal{NP}$-difícil, mas não é $\mathcal{NP}$-completo.

Teorema de Cook

- Existe algum problema em $\mathcal{NP}$ tal que se ele for mostrado estar em $\mathcal{P}$, implicaria $\mathcal{P} = \mathcal{NP}$?
- **Teorema de Cook**: Satisfabilidade (SAT) está em $\mathcal{P}$ se e somente se $\mathcal{P} = \mathcal{NP}$.
- Ou seja, se existisse um algoritmo polinomial determinístico para *satisfabilidade*, então todos os problemas em $\mathcal{NP}$ poderiam ser resolvidos em tempo polinomial.
- A prova considera os dois sentidos:
  1. SAT está em $\mathcal{NP}$ (basta apresentar um algoritmo não-determinístico que execute em tempo polinomial). Logo, se $\mathcal{P} = \mathcal{NP}$, então SAT está em $\mathcal{P}$.
  2. Se SAT está em $\mathcal{P}$, então $\mathcal{P} = \mathcal{NP}$. A prova descreve como obter de qualquer algoritmo polinomial não determinístico de decisão A, com entrada E, uma fórmula $Q(A, E)$ de modo que Q é *satisfatível* se e somente se A termina com sucesso para E. O comprimento e tempo para construir Q é $O(p^3(n)\log(n))$, onde n é o tamanho de E e $p(n)$ é a complexidade de A.

Prova do Teorema de Cook

- A prova, bastante longa, mostra como construir Q a partir de A e E.
- A expressão booleana Q é longa, mas pode ser construída em tempo polinomial no tamanho de E.
- Prova usa definição matemática da **Máquina de Turing não-determinística** (MTND), capaz de resolver qualquer problema em $\mathcal{NP}$.
  - incluindo uma descrição da máquina e de como instruções são executadas em termos de fórmulas booleanas.
- Estabelece uma correspondência entre todo problema em $\mathcal{NP}$ (expresso por um programa na MTnd) e alguma instância de SAT.
- Uma instância de SAT corresponde à tradução do programa em uma fórmula booleana.
- A solução de SAT corresponde à simulação da máquina executando o programa em cima da fórmula obtida, o que produz uma solução para uma instância do problema inicial dado.

Prova de que um Problema é $\mathcal{NP}$-Completo

- São necessários os seguintes passos:
  1. Mostre que o problema está em $\mathcal{NP}$.
  2. Mostre que um problema $\mathcal{NP}$-completo conhecido pode ser polinomialmente transformado para ele.
- É possível porque Cook apresentou uma prova direta de que SAT é $\mathcal{NP}$-completo, além do fato de a redução polinomial ser transitiva

$$((SAT \propto P_1) \wedge (P_1 \propto P_2)) \rightarrow SAT \propto P_2.$$

- Para ilustrar como um problema P pode ser provado ser $\mathcal{NP}$-completo, basta considerar um problema já provado ser $\mathcal{NP}$-completo e apresentar uma redução polinomial desse problema para P.

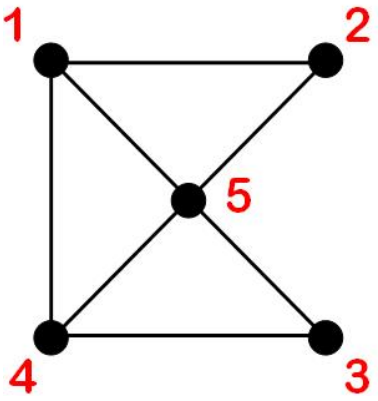
PCV é $\mathcal{NP}$-completo: Parte 1 da Prova

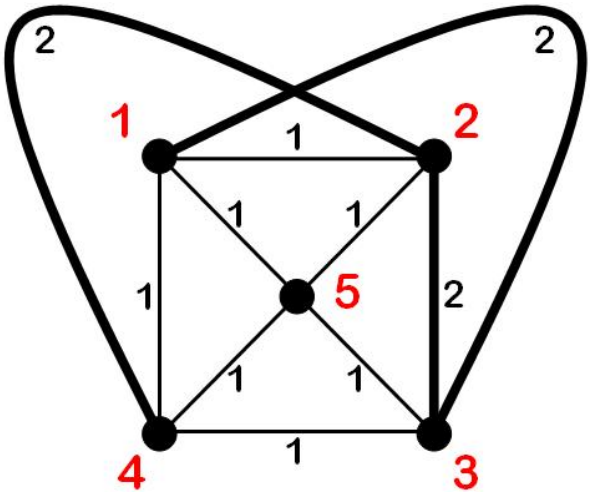
- Mostrar que o Problema do Caixeiro Viajante (PCV) está em $\mathcal{NP}$.
- Prova a partir do problema **Circuito Hamiltoniano**, um dos primeiros que se provou ser $\mathcal{NP}$-completo.
- Isso pode ser feito:
  - apresentando (como abaixo) um algoritmo não-determinístico polinomial para o PCV ou
  - mostrando que, a partir de uma dada solução para o PCV, esta pode ser verificada em tempo polinomial.

```
procedure PCVND;
begin
  i := 1;
  for t := 1 to v do
    begin
      j := escolhe(i, lista-adj(i));
      antecessor[j] := i;
      i := j;
    end;
end;
```

PCV é $\mathcal{NP}$-completo: Parte 2 da Prova

- Apresentar uma redução polinomial do Circuito Hamiltoniano para o PCV.
- Pode ser feita conforme o exemplo abaixo.

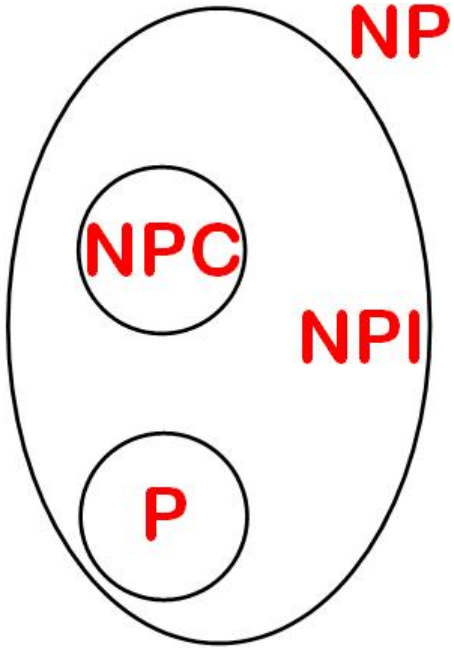




- Dado um grafo representando uma instância do Circuito Hamiltoniano, construa uma instância do PCV como se segue:
  1. Para cidades use os vértices.
  2. Para distâncias use 1 se existir uma aresta no grafo original e 2 se não existir.
- A seguir, use o PCV para achar um roteiro menor ou igual a V.
- O roteiro é o Circuito Hamiltoniano.

Classe $\mathcal{NP}$-Intermediária

- Segunda descrição tentativa do mundo $\mathcal{NP}$, assumindo $\mathcal{P} \neq \mathcal{NP}$.



- Existe uma classe intermediária entre $\mathcal{P}$ e $\mathcal{NP}$ chamada $\mathcal{NPI}$.
- $\mathcal{NPI}$ seria constituída por problemas que ninguém conseguiu uma redução polinomial de um problema $\mathcal{NP}$-completo para eles, onde $\mathcal{NPI} = \mathcal{NP}$ - ($\mathcal{P} \cup \mathcal{NP}$-completo).

Membros Potenciais de $\mathcal{NPI}$

- **Isomorfismo de grafos**: Dados $G = (V, E)$ e $G' = (V, E')$, existe uma função $f : V \rightarrow V$, tal que $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$?
  - Isomorfismo é o problema de testar se dois grafos são o mesmo.
  - Suponha que seja dado um conjunto de grafos e que alguma operação tenha de ser realizada sobre cada grafo.
  - Se pudermos identificar quais grafos são duplicatas, eles poderiam ser descartados para evitar trabalho redundante.
- **Números compostos**: Dado um inteiro positivo k, existem inteiros $m, n > 1$ tais que $k = mn$?
  - Princípio da criptografia RSA: é fácil encontrar números primos grandes, mas difícil fatorar o produto de dois deles.

Classe $\mathcal{NP}$-Completo: Resumo

- Problemas que pertencem a $\mathcal{NP}$, mas que podem ou não pertencer a $\mathcal{P}$.
- Propriedade: se qualquer problema $\mathcal{NP}$-completo puder ser resolvido em tempo polinomial por uma máquina determinística, então todos os problemas da classe podem, isto é, $\mathcal{P} = \mathcal{NP}$.
- A falha coletiva de todos os pesquisadores para encontrar algoritmos eficientes para estes problemas pode ser vista como uma dificuldade para provar que $\mathcal{P} = \mathcal{NP}$.
- Contribuição prática da teoria: fornece um mecanismo que permite descobrir se um novo problema é "fácil" ou "difícil".
- Se encontrarmos um algoritmo eficiente para o problema, então não há dificuldade. Senão, uma prova de que o problema é $\mathcal{NP}$-completo nos diz que o problema é tão "difícil" quanto todos os outros problemas "difíceis" da classe $\mathcal{NP}$-completo.