TEORIA DE COMPLEXIDADE

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO 1º SEMESTRE DE 2009

Antonio Alfredo Ferreira Loureiro

loureiro@dcc.ufmg.br

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

Gödel e o problema da Incompletude

- Em 1931, Gödel publica o trabalho

> Gödel, K. (1931). "Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I" (On formally undecidable propositions of *Principia Mathematica* and related systems I), *Monatshefte für Mathematik und Physik*, 38:173–198. [Reimpresso e traduzido em Gödel (1986, pp. 144–195)]
>
> Gödel, K. (1986). *Collected Works*, vol. 1, S. Feferman et al., eds., Oxford: Oxford University Press.

- Gödel prova que em um sistema lógico formal existem afirmações verdadeiras que não podem ser provadas.
- Mesmo lidando com "aritmética", o sistema axiomático que a inclui não pode ser simultaneamente completo e consistente.
- Isto significa que, se o sistema é consistente, então existirão proposições que não poderão ser nem comprovadas nem negadas por este sistema axiomático.
- Se o sistema for completo, então ele não se poderá validar a si mesmo sendo, portanto, inconsistente.

Turing e sua máquina (1)

- Em 1937, Turing publica o trabalho

> Turing, A.M. (1937). "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, series 2 42:230–265, doi:10.1112/plms/s2-42.1.230 (Turing, A.M. (1938). "On Computable Numbers, with an Application to the Entscheidungsproblem: A correction", *Proceedings of the London Mathematical Society*, series 2 43: 544–546, 1937, doi:10.1112/plms/s2-43.6.544)
>
> Entscheidungsproblem: termo em alemão para "problema de decisão". Em 1928, na conferência international de matemática, David Hilbert propôs três questões sendo que a terceira delas ficou conhecida como "O Problema de Decisão de Hilbert" (*Hilbert's Entscheidungsproblem*).

- Turing propõe uma máquina, que passaria a ser chamada na literatura de máquina de Turing (computador abstrato).

- Até aquele momento, é a formalização mais convincente da noção de uma função computável algoritmicamente.

Turing e sua máquina (2)

- Conseqüência: provas de impossibilidade relacionadas à computação de determinadas tarefas.
- Exemplo: Turing prova que nenhum algoritmo (i.e., Máquina de Turing) pode decidir num número finito de passos se uma formula arbitrária do cálculo de predicados é satisfazível.
- Turing reformula os resultados obtidos por Gödel no célebre *Entscheidungsproblem*: qualquer cálculo que possa ser feito por um ser humano poderá ser feito por este tipo de máquina.
- Além disso, Turing prova que o "Problema da Parada" é indecidível, ou seja, não é possível decidir algoritmicamente se a máquina de Turing irá parar ou não e, assim, provou-se que não há solução para o *Entscheidungsproblem*.

Indecidibilidade definida. Qual é próxima questão?

- Dada a máquina de Turing, com um modelo computacional muito bem definido, e uma teoria associada para explicar que problemas podem ou não serem resolvidos por ela, surge naturalmente a próxima questão:

  Qual é a dificuldade computacional de computar funções?

➜ Tema principal da complexidade computacional.

Rabin e os primórdios da complexidade computacional

- Em 1959/1960, Rabin publica os trabalhos

  Rabin, M.0. (1959). Speed of Computation and Classification of Recursive Sets. *Third Convention of Scientific Societies*, Israel, 1959, 1–2.

  Rabin, M.0. (1960). Degree of Difficulty of Computing a Function and a Partial Ordering of Recursive Sets, *Technical Report O.N.R. Contract*, pp. 341–360.

- É um dos primeiros cientistas a tratar desta questão explicitamente: o que significa dizer que computar a função f é mais difícil que computar a função g?

- Rabin sugere um arcabouço axiomático que define a base para a teoria de complexidade abstrata desenvolvida por Blum e outros.

  Blum, M. (1967). A Machine Independent Theory of the Complexity of Recursive Functions. *Journal of the ACM*, 14(2):322-336, April.

Hartmanis & Stearns e o nome do jogo

- Em 1965, Hartmanis & Stearns publicam o trabalho

  Hartmanis, J. and Stearns, R.E. (1965). On the Computational Complexity of Algorithms. *Transactions of AMS*, 117:285–306.

- Trabalho de referência que dá o nome à área.

- Propõem como medida de complexidade o tempo de computação em máquinas de Turing multi-fitas.

Cobham e complexidade intrínseca

- Em 1965, Cobham publica o trabalho

> Cobham. A. (1965). The Intrinsic Computational Difficulty of Functions. *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Sciences*, Y. Bar-Hellel, ed., North Holland, Amsterdam, 1965, 24–30.

- Cobham enfatiza o termo "intrínseco", i.e., ele estava interessado numa teoria que fosse independente de uma máquina.
- Ele pergunta se a multiplicação é mais difícil que a adição e acreditava que essa questão não poderia ser resolvida até que uma teoria fosse adequadamente proposta.
- Cobham também definiu e caracterizou uma classe importante de funções que ele chamou de $\mathcal{I}$, i.e., funções nos números naturais computáveis em tempo limitado por um polinômio no tamanho da entrada.

Questão que passa a dominar

- Qual é a medida de complexidade computacional correta/adequada?
  - Escolhas naturais: tempo e espaço.
- Não há um consenso que essas métricas sejam as corretas ou as únicas:
  - Cobham sugeriu que "*. . . some measure related to the physical notion of work [may] lead to the most satisfactory analysis*."
  - Rabin propôs axiomas que uma medida de complexidade deveria satisfazer.
- Após quase meio século dos primeiros trabalhos da área, está claro que *tempo* e *espaço* estão certamente entre as métricas mais importantes.
- No entanto, existem outras importantes:
  - Tempo e espaço para modelos computacionais paralelos.
  - Tamanho do hardware (complexidade combinatória ou de circuitos booleanos): suponha que uma função f receba como entrada strings finitos de bits e gere também como saída strings finitos de bits e a complexidade $C(n)$ de f seja o tamanho do menor circuito booleano que computa f para todas as entradas de tamanho n.

A importância do "tempo"

- Em 1965, Cobham formaliza o conceito de classe de problemas que podem ser resolvidos em uma quantidade de tempo limitada por um polinômio do tamanho da entrada.
  - Em 1953, von Neumann distingue algoritmos polinomiais de exponenciais no tempo mas não apresenta uma formalização.
- Em 1972, Karp propõe a notação usada em complexidade computacional.
  - Em particular, a classe de problemas P que podem ser resolvidos em tempo polinomial.
- Uma classe de complexidade é um conjunto de funções que podem ser computadas dentro de certos limites de recursos.
- A classe P passa a ser identificada com problemas tratáveis.
  - Na prática, não encontramos algoritmos polinomiais da ordem de n^{1000} ou de algoritmos exponenciais da ordem de $2^{0.001n}$

A era atual (1)

- A partir de meados da década de 1960, vários pesquisadores publicam trabalhos relacionados à complexidade computacional.

- Em 1971, Cook publica o primeiro problema NP-completo (o problema da satisfabilidade booleana – SAT):

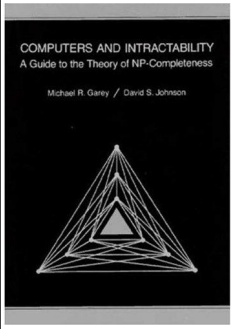
  Cook, S.A. (1971). The Complexity of Theorem Proving Procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151–158.

- Em 1972, Karp publica uma lista de 21 problemas NP-completo, mostrando a importância da área:

  Karp, R.M. Karp (1972). "Reducibility Among Combinatorial Problems", in R.E. Miller and J.W. Thatcher (editors). *Complexity of Computer Computations*. New York: Plenum, pp. 85–103.

A era atual (2)

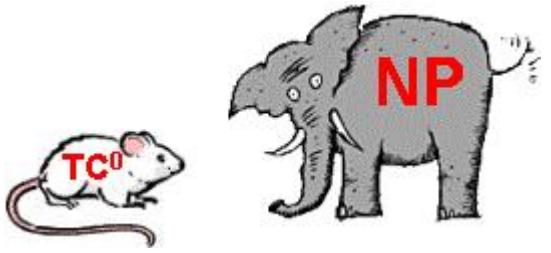
- Estima-se que existam hoje mais de 3000 problemas NP-completo.



Livro clássico da área:
Garey, M.R. & Johnson, D.S. (1979). "Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H. Freeman, 340 pages.

- Hoje, existem quase 500 classes de complexidade catalogadas

(Veja *Complexity Zoo* em http://qwiki.stanford.edu/wiki/Complexity_Zoo).



TC^0: The class of decision problems solvable by polynomial-size, constant-depth circuits with unbounded fanin, which can use AND, OR, and NOT gates as well as threshold gates. A threshold gate returns 1 if at least half of its inputs are 1, and 0 otherwise.

ACM Turing Award e Complexidade computacional (1)

Sobre o ACM A.M. Turing Award

The A.M. Turing Award was named for Alan M. Turing, the British mathematician who articulated the mathematical foundation and limits of computing, and who was a key contributor to the Allied cryptanalysis of the German Enigma cipher during World War II. Since its inception in 1966, the Turing Award has honored the computer scientists and engineers who created the systems and underlying theoretical foundations that have propelled the information technology industry.

ACM's most prestigious technical award is accompanied by a prize of $250,000. It is given to an individual selected for contributions of a technical nature made to the computing community. The contributions should be of lasting and major technical importance to the computer field. Financial support of the Turing Award is provided by the Intel Corporation and Google Inc.

Fonte: http://awards.acm.org/turing

ACM Turing Award e Complexidade computacional (2)

Cientistas que receberam o ACM Turing Award relacionados à complexidade computacional:

1976: Michael O. Rabin e Dana Scott: "For their joint paper 'Finite Automata and Their Decision Problem,' which introduced the idea of nondeterministic machines, which has proved to be an enormously valuable concept. Their (Scott & Rabin, 1959) classic paper has been a continuous source of inspiration for subsequent work in this field."

1982: Stephen A. Cook: "For his advancement of our understanding of the complexity of computation in a significant and profound way. His seminal paper, 'The Complexity of Theorem Proving Procedures,' presented at the 1971 ACM SIGACT Symposium on the Theory of Computing, laid the foundations for the theory of NP-Completeness. The ensuing exploration of the boundaries and nature of NP-complete class of problems has been one of the most active and important research activities in computer science for the last decade."

1985: Richard M. Karp: "For his continuing contributions to the theory of algorithms including the development of efficient algorithms for network flow and other combinatorial optimization problems, the identification of polynomial-time computability with the intuitive notion of algorithmic efficiency, and, most notably, contributions to the theory of NP-completeness. Karp introduced the now standard methodology for proving problems to be NP-complete which has led to the identification of many theoretical and practical problems as being computationally difficult."

ACM Turing Award e Complexidade computacional (3)

1993: Juris Hartmanis e Richard E. Stearns: “In recognition of their seminal joint research which established the foundations for the field of computational complexity theory. In their paper ‘On the Computational Complexity of Algorithms (Transactions of the American Mathematical Society, 117(5):285–306, May 1965’) they provided a precise definition of the complexity measure defined by computation time on Turing machines and developed a theory of complexity classes. The paper sparked the imagination of many computer scientists and led to the establishment of complexity theory as a fundamental part of the discipline.”

1995: Manuel Blum: “In recognition of his contributions to the foundations of computational complexity theory and its application to cryptography and program checking.”

2000: Andrew Chi-Chih Yao: “In recognition of his fundamental contributions to the theory of computation, including the complexity-based theory of pseudorandom number generation, cryptography, and communication complexity.”

Complexidade de problemas

- Problemas intratáveis ou difíceis são comuns na natureza e nas áreas do conhecimento.
- Problemas podem ser classificados em:
  - "fáceis" (tratáveis): resolvidos por algoritmos polinomiais.
  - "difíceis" (intratáveis): os algoritmos conhecidos para resolvê-los são exponenciais.
- Complexidade de tempo da maioria dos problemas é polinomial ou exponencial.

Teoria da complexidade

- Problema:
  - Conjunto de parâmetros (definição das instâncias).
  - Conjunto de propriedades (restrições do problema).

- Tamanho das instâncias:
  - Quantidade de bits necessária para representar as instâncias em computadores digitais.

- A questão é . . .
  - ➜ Qual o número de computações (operações em bits) necessárias para se obter a melhor solução (solução ótima)?

Classes de complexidade associadas a problemas fáceis $\times$ difíceis

- **Polinomial**: complexidade é $O(p(n))$, onde $p(n)$ é um polinômio.
  - Pesquisa binária ($O(\log n)$)
  - Pesquisa seqüencial ($O(n)$)
  - Ordenação por inserção ($O(n^2)$)
  - Multiplicação de matrizes ($O(n^3)$)

  ➜ **P** (*Polinomial Time*): problemas onde o número de computações cresce polinomialmente em função do tamanho da instância.

- **Exponencial**: complexidade é $O(c^n)$, $c > 1$.
  - Exemplo: Problema do Caixeiro Viajante (PCV), que é $O(n!)$.
  - Mesmo problemas de tamanho pequeno a moderado não podem ser resolvidos por algoritmos não-polinomiais, i.e., algoritmos força bruta que são exponenciais.

  ➜ **NP** (*Nondeterministic Polinomial Time*): problemas onde o número de computações cresce exponencialmente em função do tamanho da instância e não existe garantia da existência de algoritmos melhores.

Tipos de problemas (1)

A partir de um problema "complexo" computacionalmente, podemos formular três tipos de problemas (versões):

1. Problema de decisão:
   - Consiste na verificação (decisão) da veracidade ou não de determinada questão para o problema (resposta SIM ou NÃO).
   - Verificação da existência de determinada solução.

2. Problema de localização:
   - Consiste na verificação da existência e identificação (localização) de uma solução (segundo algum critério) para o problema.
   - Deve-se encontrar uma estrutura que satisfaça uma ou mais propriedades.

3. Problema de otimização:
   - Consiste na verificação da existência e identificação da melhor (otimização) solução possível, dentro as soluções factíveis para o problema.
   - Deve-se encontrar uma estrutura que satisfaça um critério de otimização.

Tipos de problemas (2)

- A definição desses três tipos de problemas foi feita de tal forma a ser bem caracterizada a relação entre os mesmos:
  - Um problema de otimização possui embutido um problema de localização que por sua vez possui embutido um problema de decisão.

- Ordem de dificuldade desses problemas:

$$\textsc{Decisão} \preceq \textsc{Localização} \preceq \textsc{Otimização}$$

- Se for comprovado que um problema de decisão é intratável, também serão suas versões de localização e de otimização.
  - Daí o fato de que toda a teoria NP se basear na versão de decisão dos problemas, sem perda de generalidade.

Tipos de problemas (3)
Exemplo

DECISÃO: Seja um grafo G e um inteiro $k > 0$.

$\triangleright$ Existe em G um clique de tamanho $\geq k$?

LOCALIZAÇÃO: Seja um grafo G e um inteiro $k > 0$.

$\triangleright$ Encontre em G um clique de tamanho $\geq k$.

OTIMIZAÇÃO: Seja um grafo G.

$\triangleright$ Encontre em G um clique de tamanho máximo.

Problemas NP

- A teoria de complexidade 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 NP.

- Propriedade: um problema da classe NP poderá ser resolvido em tempo polinomial se, e somente se, todos os outros problemas em 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 NP.

Classe NP: Problemas SIM/NÃO

- Para o estudo teórico da complexidade de algoritmos consideram-se problemas cujo resultado da computação seja SIM ou NÃO.
  ➜ Problemas de decisão.

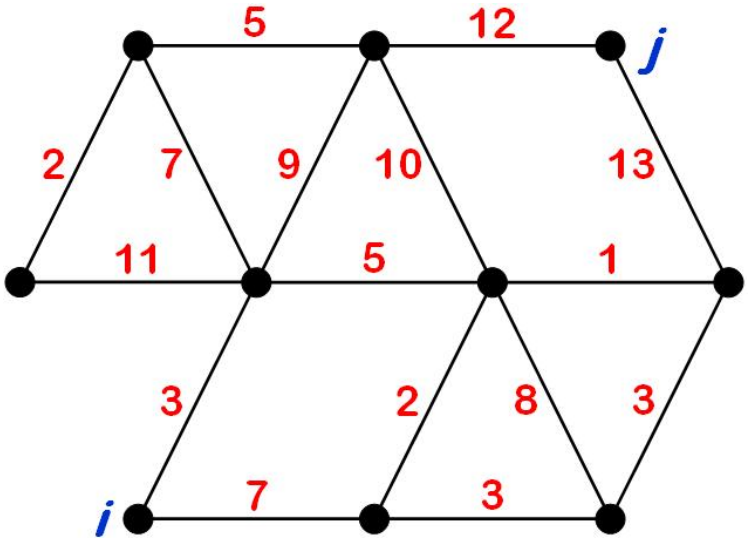
- Versão do PCV cujo resultado é do tipo SIM/NÃO:
  - Dados: constante k, conjunto de cidades $C = \{c_1, c_2, \cdots, c_n\}$ e distâncias $d(c_i, c_j)$ para cada par de cidades $c_i, c_j \in C$.
  - Questão: Existe um "roteiro" para todas as cidades em C cujo comprimento total seja menor ou igual a k?

- Característica da classe 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 $G = (V, E)$, mapear $C : V \to S$, sendo S um conjunto finito de cores tal que se a aresta $vw \in E$ então $c(v) \neq c(w)$.
  ➜ Vértices adjacentes possuem cores distintas.
- O número cromático $\chi(G)$ é o menor nº de cores necessário para colorir G, i.e., o menor k para o qual existe uma coloração C para G e $|C(V)| = k$.
- Problema: produzir uma coloração ótima que usa apenas $\chi(G)$ 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 (1)

- 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 (2)

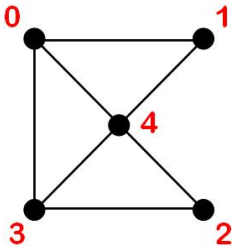
- Evidentemente, não existe conflito se cada vértice for colorido com uma cor distinta.
- O objetivo porém é encontrar uma coloração usando o mínimo de cores (computadores têm um número limitado de registradores).
- **Número cromático**: menor número de cores suficientes para colorir um grafo.

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.

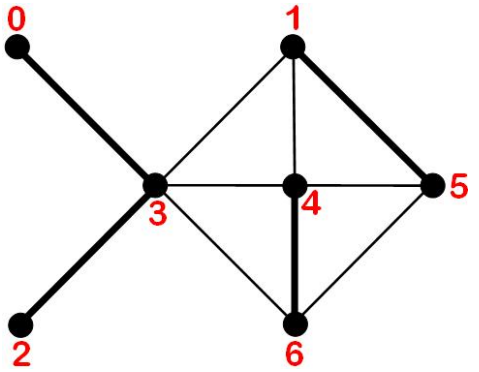
Circuito Hamiltoniano

- **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.
- Exemplos:
  Circuito Hamiltoniano: 0 1 4 2 3 0
  Caminho Hamiltoniano: 0 1 4 2 3

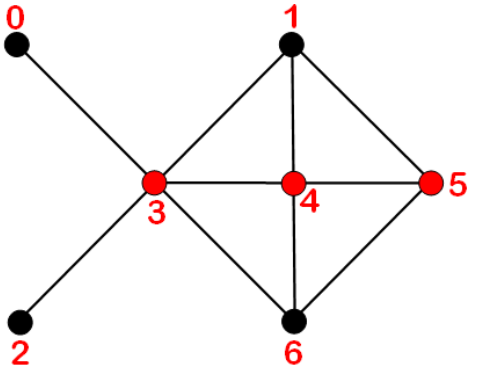


- Existe circuito Hamiltoniano no grafo G?
  - *Fácil*: Grafos com grau máximo = 2.
  - *Difícil*: Grafos com grau > 2.
- É um caso especial do PCV.
  - Pares de vértices com uma aresta entre eles tem distância 1; e
  - Pares de vértices sem aresta entre eles têm distância infinita.

Cobertura de arestas e vértices



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



- Uma **cobertura de vértices** é um subconjunto $V' \subset V$ tal que se $\{u, v\} \in E$ então $u \in V'$ ou $v \in V'$, isto é, cada aresta do grafo é incidente a um dos vértices de V'.
- $V' = \{3, 4, 5\}$, 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 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 sse não há um conjunto de escolhas que indica sucesso.

Comandos *sucesso* e *insucesso*

- Algoritmos não-determinísticos utilizam também dois comandos:
  - ***sucesso***: indica término com sucesso.
  - ***insucesso***: indica término sem sucesso.

- Os comandos ***sucesso*** e ***insucesso*** são usados para definir uma execução do algoritmo.

- Esses comandos são equivalentes a um comando de parada de um algoritmo determinístico.

- Os comandos ***sucesso*** e ***insucesso*** 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.

Pesquisa não-determinística

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

```
PESQUISAND(A, x)                          ▷ Pesquisa x no vetor A de forma ND
1  j ← ESCOLHE(A, x)                      ▷ Determina o índice j de x em A, se existir
2  if A[j] = x                            ▷ Pesquisa com sucesso?
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 $O(n)$.

Ordenação não-determinística

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

```
ORDENAND(A)                                  ▷ Ordena vetor A de forma ND
1  for i ← 1 to n
2      do B[i] ← 0                           ▷ Inicializa vetor auxiliar B com zero
3  for i ← 1 to n
4      do j ← ESCOLHE(A, i)                  ▷ Determina o índice j do elemento A[i]
5          if B[j] = 0                       ▷ É possível ordenar o elemento A[i]?
6            then B[j] ← A[i]                ▷ Coloca o elemento A[i] na posição final B[j]
7            else insucesso
```

- Usa um vetor auxiliar $B[1 : n]$, que irá conter o conjunto em ordem crescente.
- A posição correta em B de cada elemento i de A é obtida de forma não-determinística pela função *escolhe*.
- O comando de decisão verifica se a posição $B[j]$ ainda não foi utilizada.
- Complexidade: $O(n)$.
  - ➜ Para um algoritmo determinístico a complexidade é $\Omega(n \log n)$.

Problema da Satisfabilidade (1)

- 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 $\overline{x_i}$.
- Seja uma expressão booleana com operações de disjunção (ou adição — $\vee$) e conjunção (ou multiplicação — $\wedge$).
- Uma expressão booleana E contendo um produto de adições de variáveis booleanas é dita estar na **forma normal conjuntiva** (FNC).
- Dada E na forma normal conjuntiva, com variáveis x_i, $1 \leq i \leq n$, existe uma atribuição de valores V ou F às variáveis que torne E verdadeira ("satisfaça")?

Problema da Satisfabilidade (2)

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

```
AVALND(E, n)                                  ▷ Avalia E na FNC de forma ND
1  for i ← 1 to n
2      do x_i ← ESCOLHE(true, false)          ▷ Escolhe para cada variável x_i o valor V ou F
3  if E(x_1, x_2, ··· , x_n)                  ▷ Avalia E para os valores atribuídos a x_i
4    then sucesso
5    else insucesso
```

- Obtém uma das 2^n atribuições de forma não-determinística em $O(n)$.
- Melhor algoritmo determinístico: $O(2^n)$.
- Aplicação: definição de circuitos 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.

Problema da Satisfabilidade (3)

Exemplos:

- $E_1 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3} \vee x_2) \wedge (x_3)$ é *satisfazível*

x_1	x_2	x_3	E_1
V	V	V	V
V	V	F	F
V	F	V	V
V	F	F	F
F	V	V	V
F	V	F	F
F	F	V	F
F	F	F	F

  - ➜ Existem três conjuntos de valores que satisfazem E_1.
  - ➜ A função *escolhe* seleciona um deles.

- $E_2 = x_1 \wedge \overline{x_1}$ não é *satisfazível*.

Caracterização das classes P e NP

P: conjunto de todos os problemas que podem ser resolvidos por *algoritmos determinísticos* em tempo *polinomial*.

NP: conjunto de todos os problemas que podem ser resolvidos por *algoritmos não-determinísticos* em tempo *polinomial*.

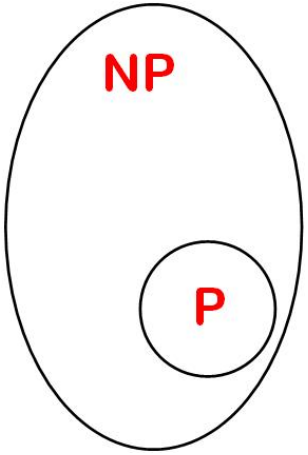
- Para mostrar que um determinado problema está em NP, basta apresentar um algoritmo não-determinístico que execute em tempo polinomial para resolver o problema.
- Outra maneira é encontrar um algoritmo determinístico polinomial para verificar que uma dada solução é válida.

Existe diferença entre P e NP?

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

Existe diferença entre P e NP?

- Descrição tentativa do mundo NP, incluindo a classe P.



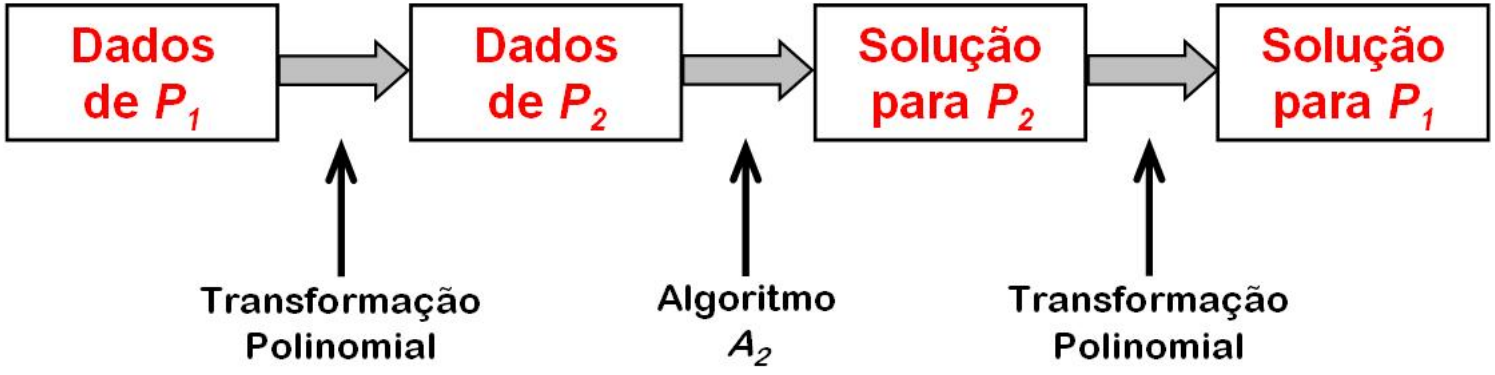
- Acredita-se que NP $\gg$ P, pois para muitos problemas em NP, não existem algoritmos polinomiais conhecidos, nem um **limite inferior não-polinomial** provado.

NP $\supset$ P ou NP = P? Conseqüências

- Muitos problemas práticos em NP podem ou não pertencer a P (não conhecemos nenhum algoritmo determinístico eficiente para eles).
- Se conseguirmos provar que um problema não pertence a 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 NP = P.
- Existe um esforço considerável para provar o contrário, mas a questão continua em aberto!

Transformação polinomial

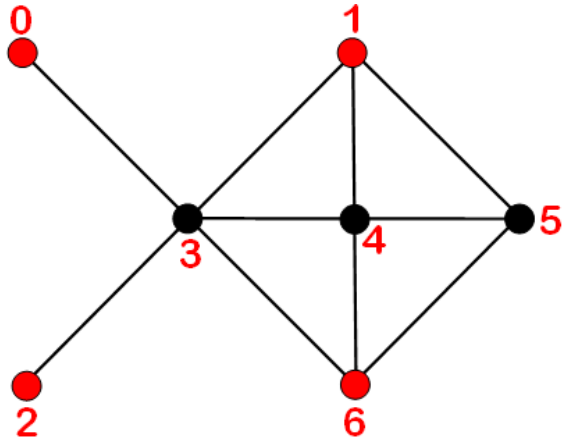
- Sejam P_1 e P_2 dois problemas SIM/NÃO.
- Suponha que um algoritmo A_2 resolva P_2.
- Se for possível transformar P_1 em P_2 e a solução de P_2 em solução de P_1, então A_2 pode ser utilizado para resolver P_1.
- Se pudermos realizar as transformações nos dois sentidos em tempo polinomial, então P_1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



- Esse conceito é importante para definir a classe 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 $G = (V, E)$ é constituído do subconjunto $V' \subseteq V$, tal que $v, w \in V' \Rightarrow \{v, w\} \notin E$.
- Todo par de vértices de V' é não adjacente.
  ➜ V' é um subgrafo totalmente desconexo.
- Exemplo:



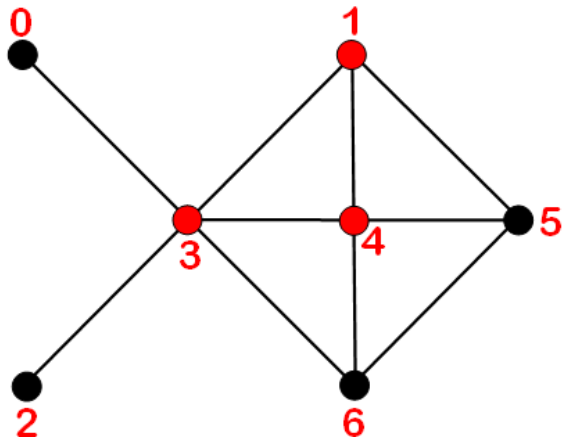
Para cardinalidade 4, temos $V' = \{0, 1, 2, 6\}$.

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



➜ Note que outras possibilidades para V' seriam $\{1, 4, 5\}$, $\{3, 4, 6\}$ e $\{4, 5, 6\}$.

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 $G = (V, E)$ 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 NP-Completo e 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 NP-difícil se e somente se $SAT \propto P$ (*satisfabilidade* é redutível a P).
- Um problema de decisão P é denominado NP-completo quando:
  1. $P \in NP$;.
  2. Todo problema de decisão $P' \in$ NP-completo satisfaz $P' \propto P$.

Problemas NP-Completo e NP-Difícil

- Um problema de decisão P que seja NP-difícil pode ser mostrado ser NP-completo exibindo um algoritmo não-determinístico polinomial para P.
- Apenas problemas de decisão podem ser NP-completo.
- Problemas de otimização podem ser 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 NP-difícil não é menor do que a dificuldade de um problema NP-completo.

Exemplo: Problema da Parada

- É um exemplo de problema NP-difícil que não é 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 *satisfazível*.
  - Se for, A pára; senão, entra em *loop*.
  - Logo, o problema da parada é NP-difícil, mas não é NP-completo.

Teorema de Cook

- Existe algum problema em NP tal que se ele for mostrado estar em P, implicaria P = NP?
- **Teorema de Cook**: Satisfabilidade (SAT) está em P se e somente se P = NP.
- Ou seja, se existisse um algoritmo polinomial determinístico para *satisfabilidade*, então todos os problemas em NP poderiam ser resolvidos em tempo polinomial.
- A prova considera os dois sentidos:
  1. SAT está em NP (basta apresentar um algoritmo não-determinístico que execute em tempo polinomial). Logo, se P = NP então SAT está em P.
  2. Se SAT está em P então P = 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 *satisfazível* 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 NP.
  - incluindo uma descrição da máquina e de como instruções são executadas em termos de fórmulas booleanas.
- Estabelece uma correspondência entre todo problema em 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 NP-Completo

- São necessários os seguintes passos:
  1. Mostre que o problema está em NP.
  2. Mostre que um problema NP-completo conhecido pode ser polinomialmente transformado para ele.
- É possível porque Cook apresentou uma prova direta de que SAT é 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 NP-completo, basta considerar um problema já provado ser NP-completo e apresentar uma redução polinomial desse problema para P.

PCV é NP-completo: Parte 1 da Prova

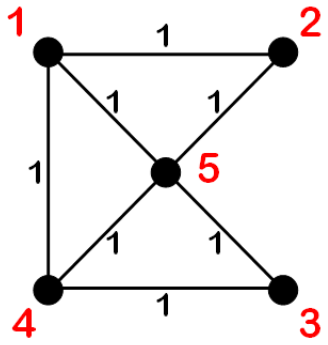
- Mostrar que o Problema do Caixeiro Viajante (PCV) está em NP.
- Prova a partir do problema **Circuito Hamiltoniano**, um dos primeiros que se provou ser NP-completo.
- Isso pode ser feito:
  - Apresentando (como abaixo) um algoritmo não-determinístico polinomial para o PCV ou
  - Mostrando que, a partir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

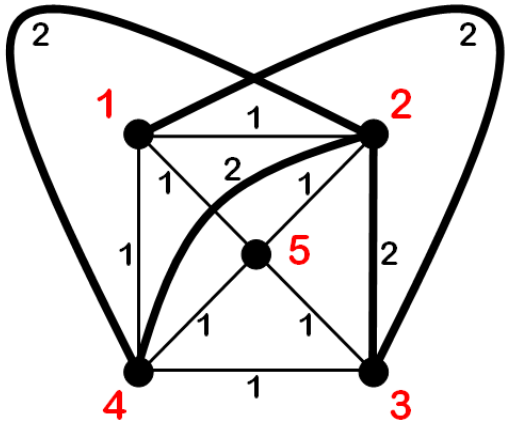
PCVND

```
1  for i ← 1 to v
2      do j ← ESCOLHE(i, lista-adj(i))
3         antecessor[j] ← i
```

PCV é NP-completo: Parte 2 da Prova

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

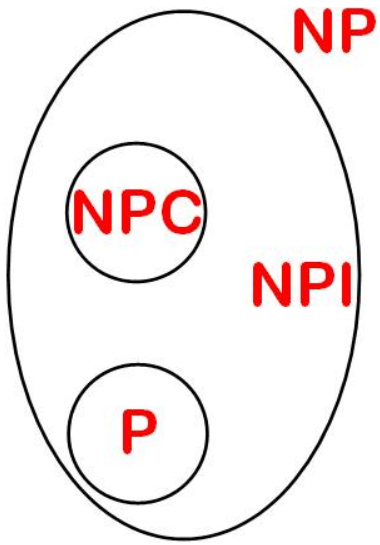




- Dado um grafo representando uma instância do ciclo de Hamilton, construa uma instância do PCV como se segue:
  1. Cidades: use os vértices.
  2. Distâncias: use 1 se existir uma aresta no grafo original e 2 se não existir.
- A seguir, use o PCV para achar um roteiro menor ou igual a V.
- O roteiro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

Classe NP-Intermediária

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



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

Membros potenciais de NPI

- **Isomorfismo de grafos**: Dados $G = (V, E)$ e $G' = (V, E')$, existe uma função $f : V \to V$, 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 NP-Completo: Resumo

- Problemas que pertencem a NP, mas que podem ou não pertencer a P.
- Propriedade: se qualquer problema NP-completo puder ser resolvido em tempo polinomial por uma máquina determinística, então todos os problemas da classe podem, isto é, P = NP.
- A falha coletiva de todos os pesquisadores para encontrar algoritmos eficientes para estes problemas pode ser vista como uma dificuldade para provar que P = 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 NP-completo nos diz que o problema é tão “difícil” quanto todos os outros problemas “difíceis” da classe NP-completo.

Problemas exponenciais

- É desejável resolver instâncias grandes de problemas de otimização em tempo razoável.
- Os melhores algoritmos para problemas NP-completo têm comportamento de pior caso exponencial no tamanho da entrada.
- Para um algoritmo que execute em tempo proporcional a 2^N, não é garantido obter resposta para todos os problemas de tamanho $N \geq 100$.
- Independente da velocidade do computador, ninguém poderia esperar por um algoritmo que leva 2^{100} passos para terminar sua tarefa.
- Um supercomputador poderia resolver um problema de tamanho $N = 50$ em 1 hora, ou $N = 51$ em 2 horas, ou $N = 59$ em um ano.
- Nem um computador paralelo com um milhão de processadores, (cada um sendo um milhão de vezes mais rápido que o mais rápido existente) é suficiente para $N = 100$.

O que fazer para resolver problemas exponenciais?

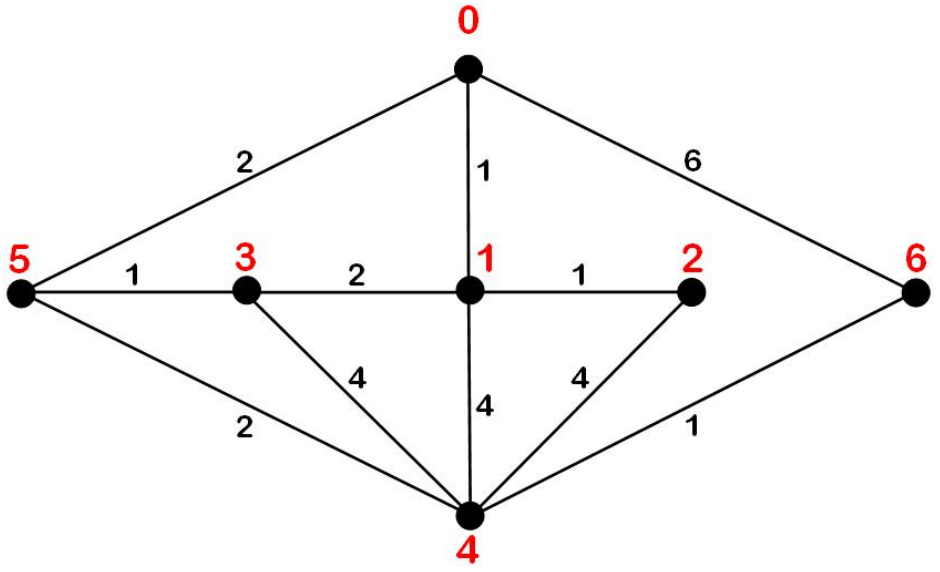
- Usar algoritmos exponenciais "eficientes" aplicando técnicas de tentativa e erro.
- Usar algoritmos aproximados. Acham uma resposta que pode não ser a solução ótima, mas é garantido ser próxima dela.
- Concentrar no caso médio. Buscar algoritmos melhores que outros neste quesito e que funcionem bem para as entradas de dados que ocorrem usualmente na prática.
  - Existem poucos algoritmos exponenciais que são muito úteis na prática.
  - Exemplo: Simplex (programação linear). Complexidade de tempo exponencial no pior caso, mas muito rápido na prática.
  - Tais exemplos são raros. A grande maioria dos algoritmos exponenciais conhecidos não é muito útil.

Circuito Hamiltoniano: Tentativa e erro

- Exemplo: encontrar um **Circuito Hamiltoniano** em um grafo.
- Obter algoritmo tentativa e erro a partir de algoritmo para caminhamento em um grafo.
- DFS faz uma busca em profundidade no grafo em tempo $O(V + E)$.

Circuito Hamiltoniano: Tentativa e erro

- Aplicando o DFS ao grafo da figura abaixo a partir do vértice 0, o procedimento Visita obtém o caminho 0 1 2 4 3 5 6, o que não é um ciclo simples.



	1	2	3	4	5	6
0	1				2	6
1		1	2	4		
2				4		
3				2	1	
4					2	1
5						

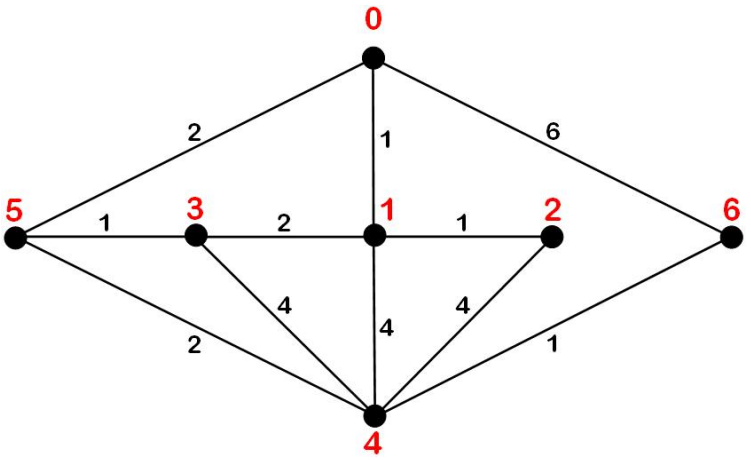
- Para encontrar um ciclo de Hamilton, caso exista, devemos visitar os vértices do grafo de outras maneiras.
- A rigor, o melhor algoritmo conhecido resolve o problema tentando todos os caminhos possíveis.

Circuito Hamiltoniano: Tentando todas as possibilidades

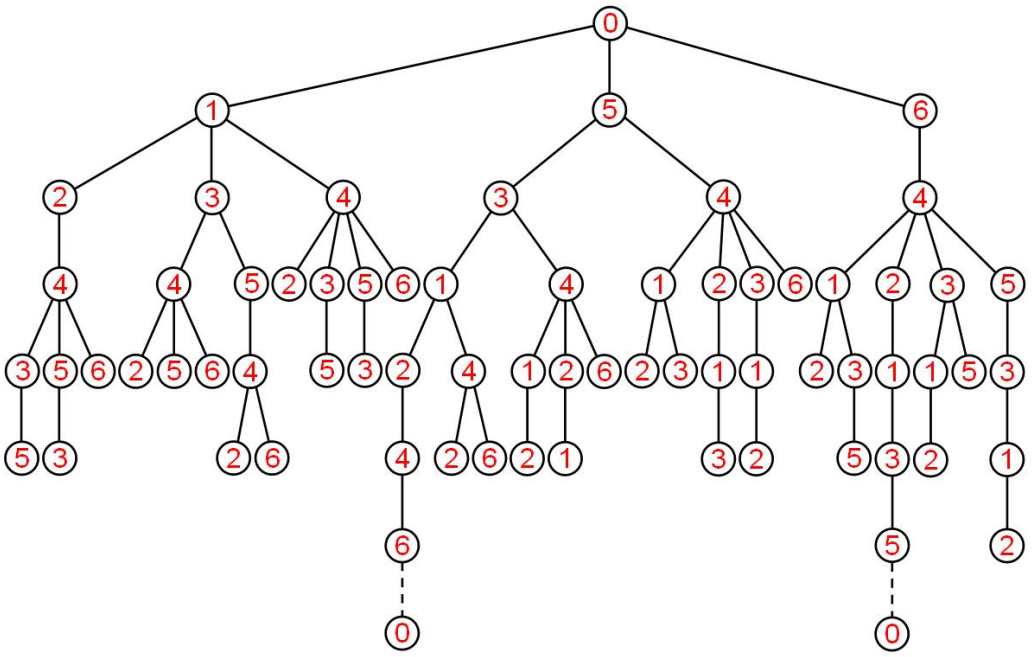
- Para tentar todas as possibilidades, vamos alterar o procedimento Visita.
- Desmarca o vértice já visitado no caminho anterior e permite que seja visitado novamente em outra tentativa.
- O custo é proporcional ao número de chamadas para o procedimento Visita.
- Para um grafo completo, (arestas ligando todos os pares de nós) existem $N!$ ciclos simples. Custo é proibitivo.

Circuito Hamiltoniano: Tentando todas as possibilidades

Para o grafo



A árvore de caminhamento é:



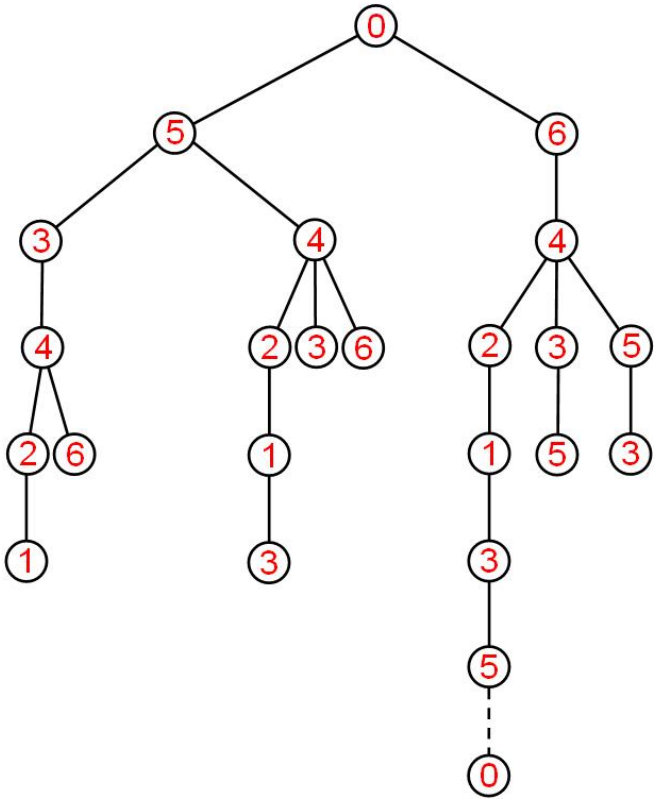
Existem duas respostas:

1. 0 5 3 1 2 4 6 0 e
2. 0 6 4 2 1 3 5 0

Circuito Hamiltoniano: Tentativa e erro com poda

- Diminuir número de chamadas a Visita fazendo "**poda**" na árvore de caminhamento.
- No exemplo anterior, cada ciclo é obtido duas vezes, caminhando em ambas as direções.
- Insistindo que o nó 2 apareça antes do 0 e do 1, não precisamos chamar Visita para o nó 1 a não ser que o nó 2 já esteja no caminho.

Árvore de caminhamento obtida:



Circuito Hamiltoniano: Tentativa e erro com poda

- Entretanto, esta técnica não é sempre possível de ser aplicada.
- Suponha que se queira um caminho de custo mínimo que não seja um ciclo e passe por todos os vértices: 0 6 4 5 3 1 2 é solução.
- Neste caso, a técnica de eliminar simetrias não funciona porque não sabemos *a priori* se um caminho leva a um ciclo ou não.

Circuito Hamiltoniano: *Branch-and-Bound*

- Outra saída para tentar diminuir o número de chamadas a Visita é por meio da técnica de ***branch-and-bound***.
- A idéia é cortar a pesquisa tão logo se saiba que não levará a uma solução.
- Corta chamadas a Visita tão logo se chegue a um custo para qualquer caminho que seja maior que um caminho solução já obtido.
- Exemplo: encontrando 0 5 3 1 2 4 6, de custo 11, não faz sentido continuar no caminho 0 6 4 1, de custo 11 também.
- Neste caso, podemos evitar chamadas a Visita se o custo do caminho corrente for maior ou igual ao melhor caminho obtido até o momento.

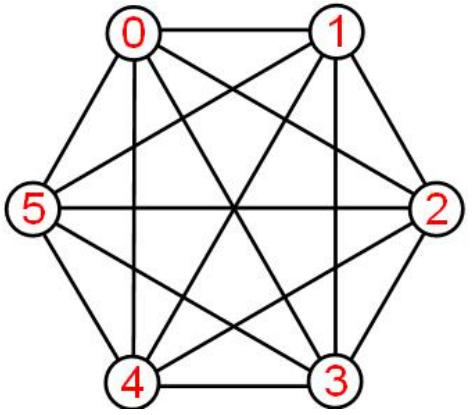
Heurísticas para problemas NP-Completo

- **Heurística**: algoritmo que pode produzir um bom resultado (ou até a solução ótima), mas pode também não obter solução ou obter uma distante da ótima.
- Uma heurística pode ser determinística ou probabilística.
- Pode haver instâncias em que uma heurística (probabilística ou não) nunca vai encontrar uma solução.
- A principal diferença entre uma heurística probabilística e um **algoritmo Monte Carlo** é que o algoritmo Monte Carlo tem que encontrar uma solução correta com uma certa probabilidade (de preferência alta) para qualquer instância do problema.

Heurística para o PCV

- Algoritmo do vizinho mais próximo, heurística gulosa simples:
  1. Inicie com um vértice arbitrário.
  2. Procure o vértice mais próximo do último vértice adicionado que não esteja no caminho e adicione ao caminho a aresta que liga esses dois vértices.
  3. Quando todos os vértices estiverem no caminho, adicione uma aresta conectando o vértice inicial e o último vértice adicionado.
- Complexidade: $O(n^2)$, sendo n o número de cidades, ou $O(d)$, sendo d o conjunto de distâncias entre cidades.
- Aspecto negativo: embora todas as arestas escolhidas sejam localmente mínimas, a aresta final pode ser bastante longa.

Heurística para o PCV



	1	2	3	4	5
0	3	10	11	7	25
1		8	12	9	26
2			9	4	20
3				5	15
4					18

- Caminho ótimo para esta instância: 0 1 2 5 3 4 0 (comprimento 58).
- Para a heurística do vizinho mais próximo, se iniciarmos pelo vértice 0, o vértice mais próximo é o 1 com distância 3.
- A partir do 1, o mais próximo é o 2, a partir do 2 o mais próximo é o 4, a partir do 4 o mais próximo é o 3, a partir do 3 restam o 5 e o 0.
- O comprimento do caminho 0 1 2 4 3 5 0 é 60.

Heurística para o PCV

- Embora o algoritmo do vizinho mais próximo não encontre a solução ótima, a obtida está bem próxima do ótimo.
- Entretanto, é possível encontrar instâncias em que a solução obtida pode ser muito ruim.
- Pode mesmo ser arbitrariamente ruim, uma vez que a aresta final pode ser muito longa.
- É possível achar um algoritmo que garanta encontrar uma solução que seja razoavelmente boa no pior caso, desde que a classe de instâncias consideradas seja restrita.

Algoritmos aproximados para problemas NP-Completo

- Para projetar algoritmos polinomiais para "resolver" um problema de otimização NP-completo é necessário relaxar o significado de resolver.
- Removemos a exigência de que o algoritmo tenha sempre de obter a solução ótima.
- Procuramos algoritmos eficientes que não garantem obter a solução ótima, mas sempre obtêm uma próxima da ótima.
- Tal solução, com valor próximo da ótima, é chamada de solução aproximada.
- Um **algoritmo aproximado** para um problema P é um algoritmo que gera **soluções aproximadas** para P.
- Para ser útil, é importante obter um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado.

Medindo a qualidade da aproximação

- O comportamento de algoritmos aproximados quanto à qualidade dos resultados (não o tempo para obtê-los) tem de ser monitorado.
- Seja I uma instância de um problema P e seja $S^*(I)$ o valor da solução ótima para I.
- Um algoritmo aproximado gera uma solução possível para I cujo valor $S(I)$ é maior (pior) do que o valor ótimo $S^*(I)$.
- Dependendo do problema, a solução a ser obtida pode minimizar ou maximizar $S(I)$.
- Para o PCV, podemos estar interessados em um algoritmo aproximado que minimize $S(I)$: obtém o valor mais próximo possível de $S^*(I)$.
- No caso de o algoritmo aproximado obter a solução ótima, então $S(I) = S^*(I)$.

Algoritmos aproximados: Definição

- Um algoritmo aproximado para um problema P é um algoritmo polinomial que produz uma solução $S(I)$ para uma instância I de P.
- O comportamento do algoritmo A é descrito pela **razão de aproximação**

$$R_A(I) = \frac{S(I)}{S^*(I)},$$

que representa um problema de minimização

- No caso de um problema de maximização, a razão é invertida.
- Em ambos os casos, $R_A(I) \geq 1$.

Algoritmos aproximados para o PCV

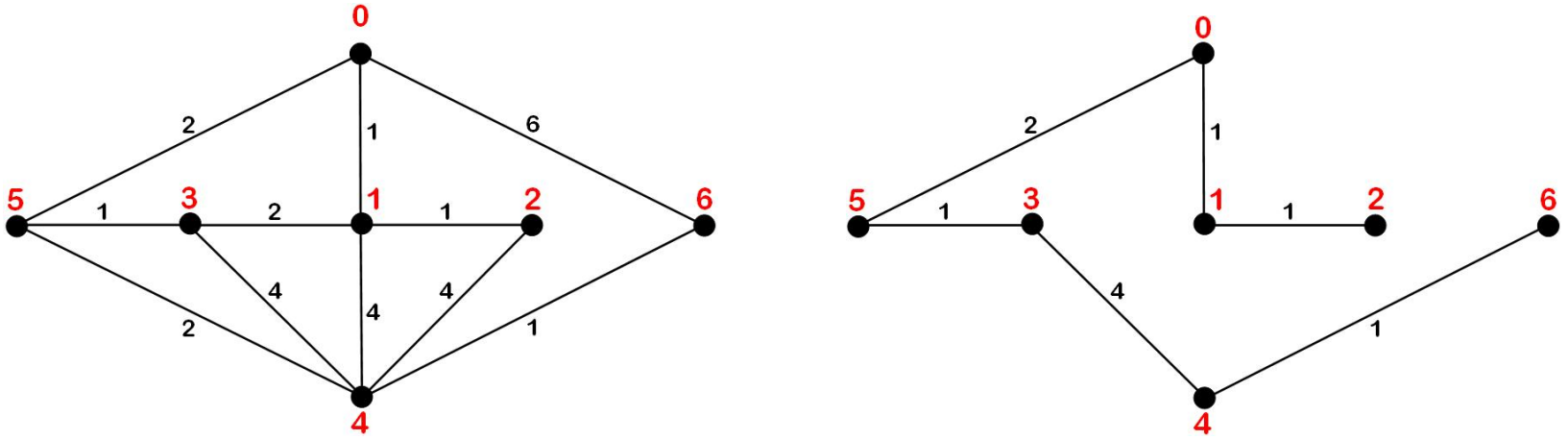
- Seja $G = (V, E)$ um grafo não direcionado, completo, especificado por um par (N, d).
- N é o conjunto de vértices do grafo (cidades), e d é uma função distância que mapeia as arestas em números reais, onde d satisfaz:
  1. $d(i, j) = d(j, i)\ \forall i, j \in N$,
  2. $d(i, j) > 0\ \forall i, j \in N$,
  3. $d(i, j) + d(j, k) \geq d(i, k)\ \forall i, j, k \in N$
- 1ª propriedade: a distância da cidade i até outra j é igual à de j até i.
  - Quando isso não acontece, temos o problema conhecido como **PCV Assimétrico**.
- 2ª propriedade: apenas distâncias positivas.
- 3ª propriedade: **desigualdade triangular**. A distância de i até j somada com a de j até k deve ser maior do que a distância de i até k.

Algoritmos aproximados para o PCV

- Quando o problema exige distâncias não restritas à desigualdade triangular, basta adicionar uma constante k a cada distância.
- Exemplo: as três distâncias envolvidas são 2, 3 e 10, que não obedecem à desigualdade triangular pois $2 + 3 < 10$. Adicionando $k = 10$ às três distâncias obtendo 12, 13 e 20, que agora satisfazem a desigualdade triangular.
- O problema alterado terá a mesma solução ótima que o problema anterior, apenas com o comprimento da rota ótima diferindo de $n \times k$.
- Cabe observar que o PCV equivale a encontrar no grafo $G = (V, E)$ um **circuito Hamiltoniano** de custo mínimo.

Árvore geradora mínima (AGM)

- Considere um grafo $G = (V, E)$, sendo V as n cidades e E as distâncias entre cidades.
- Uma árvore geradora é uma coleção de $n - 1$ arestas que ligam todas as cidades por meio de um subgrafo conectado único.
- A **árvore geradora mínima** é a árvore geradora de custo mínimo.
- Existem algoritmos polinomiais de custo $O(E \log V)$ para obter a árvore geradora mínima quando o grafo de entrada é dado na forma de uma matriz de adjacência.
- Grafo e árvore geradora mínima correspondente:



Limite inferior para a solução do PCV a partir da AGM

- A partir da AGM, podemos derivar o limite inferior para o PCV.
- Considere uma aresta (x_1, x_2) do caminho ótimo do PCV. Remova a aresta e ache um caminho iniciando em x_1 e terminando em x_2.
- Ao retirar uma aresta do caminho ótimo, temos uma árvore geradora que consiste de um caminho que visita todas as cidades.
- Logo, o caminho ótimo para o PCV é necessariamente maior do que o comprimento da AGM.
- O **limite inferior** para o custo deste caminho é a AGM.
- Logo, $\acute{O}timo_{PCV} >$ AGM.

Limite superior de aproximação para o PCV

- A desigualdade triangular permite utilizar a AGM para obter um **limite superior** para a razão de aproximação com relação ao comprimento do caminho ótimo.
- Vamos considerar um algoritmo que visita todas as cidades, mas pode usar somente as arestas da AGM.
- Uma possibilidade é iniciar em um vértice folha e usar a seguinte estratégia:
  - Se houver aresta ainda não visitada saindo do vértice corrente, siga aquela aresta para um novo vértice.
  - Se todas as arestas a partir do vértice corrente tiverem sido visitadas, volte para o vértice adjacente pela aresta pela qual o vértice corrente foi inicialmente alcançado.
  - Termine quando retornar ao vértice inicial.

Limite superior de aproximação para o PCV: Busca em profundidade

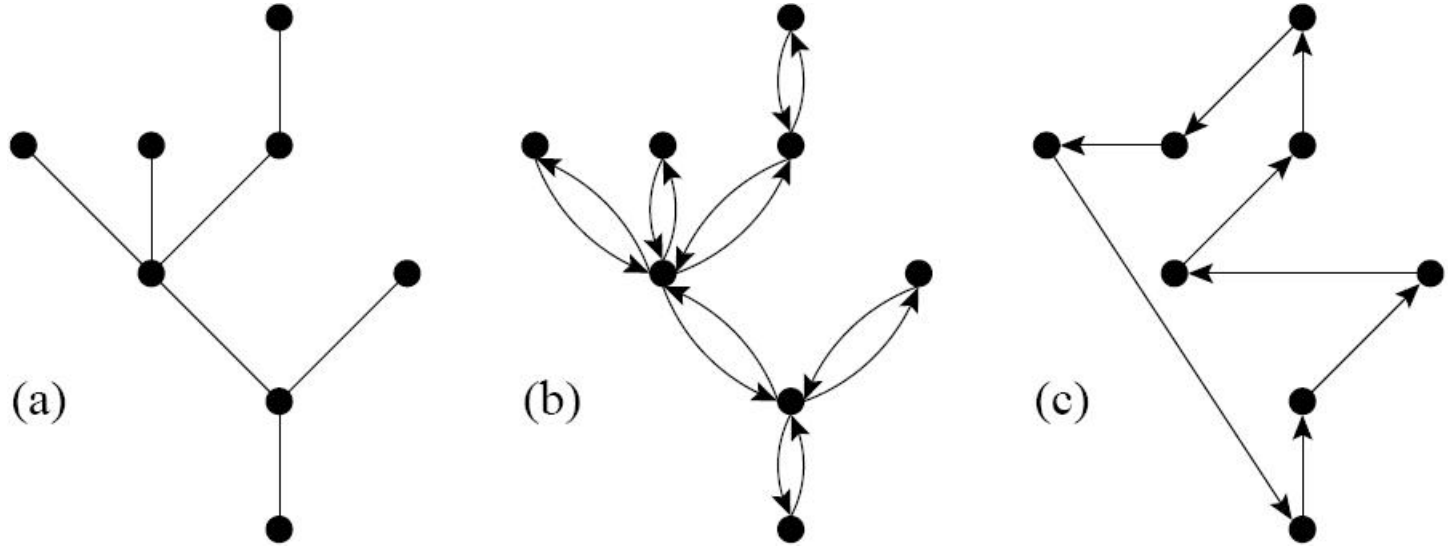
- O algoritmo descrito anteriormente é a DFS aplicada à AGM.
- Verifica-se que:
  - o algoritmo visita todos os vértices.
  - nenhuma aresta é visitada mais do que duas vezes.
- Obtém um caminho que visita todas as cidades cujo custo é menor ou igual a duas vezes o custo da árvore geradora mínima.
- Como o caminho ótimo é maior do que o custo da AGM, então o caminho obtido é no máximo duas vezes o custo do caminho ótimo:

$$Caminho_{PCV} < 2Ótimo_{PCV}.$$

- Restrição: algumas cidades são visitadas mais de uma vez.
- Para contornar o problema, usamos a desigualdade triangular.

Limite superior de aproximação para o PCV: Desigualdade triangular

- Introduzimos curto-circuitos que nunca aumentam o comprimento total do caminho.
- Inicie em uma folha da AGM, mas sempre que a busca em profundidade for voltar para uma cidade já visitada, salte para a próxima ainda não visitada.
- A rota direta não é maior do que a anterior indireta, em razão da desigualdade triangular.
- Se todas as cidades tiverem sido visitadas, volte para o ponto de partida.



- O algoritmo constrói um caminho solução para o PCV porque cada cidade é visitada apenas uma vez, exceto a cidade de partida.

Limite superior de aproximação para o PCV: Desigualdade triangular

- O caminho obtido não é maior que o caminho obtido em uma busca em profundidade, cujo comprimento é no máximo duas vezes o do caminho ótimo.
- Os principais passos do algoritmo são:
  1. Obtenha a árvore geradora mínima para o conjunto de n cidades, com custo $O(n^2)$.
  2. Aplique a busca em profundidade na AGM obtida com custo $O(n)$, a saber:
     - Inicie em uma folha (grau 1).
     - Siga uma aresta não utilizada.
     - Se for retornar para uma cidade já visitada, salte para a próxima ainda não visitada (rota direta menor que a indireta pela desigualdade triangular).
     - Se todas as cidades tiverem sido visitadas, volte à cidade de origem.
- Assim, obtivemos um algoritmo polinomial de custo $O(n^2)$, com uma razão de aproximação garantida para o pior caso de $R_A \leq 2$.

Como melhorar o limite superior a partir da AGM

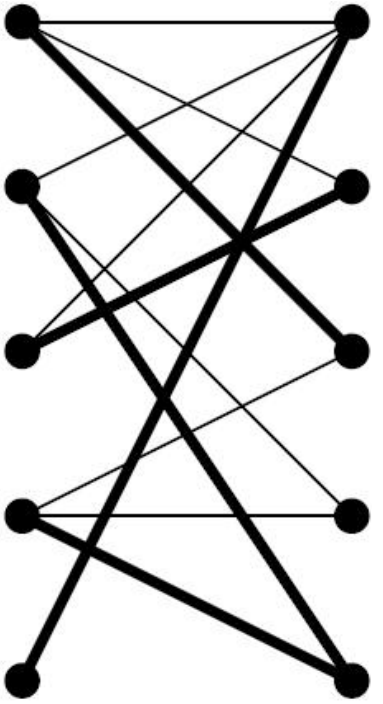
- No algoritmo anterior um caminho para o caixeiro viajante pode ser obtido dobrando os arcos da AGM, o que leva a um pior caso para a razão de aproximação no máximo igual a 2.
- Melhora-se a garantia de um fator 2 para o pior caso, utilizando o conceito de grafo Euleriano.
- Um **grafo Euleriano** é um grafo conectado no qual todo vértice tem grau par.
- Um grafo Euleriano possui um **caminho Euleriano**, um ciclo que passa por todas as arestas exatamente uma vez.
- O caminho Euleriano em um grafo Euleriano, pode ser obtido em tempo $O(n)$, usando a busca em profundidade.
- Podemos obter um caminho para o PCV a partir de uma AGM, usando o caminho Euleriano e a técnica de curto-circuito.

Como melhorar o limite superior a partir da AGM

- Passos do algoritmo:
  - Suponha uma AGM que tenha cidades do PCV como vértices.
  - Dobre suas arestas para obter um grafo Euleriano.
  - Encontre um caminho Euleriano para esse grafo.
  - Converta-o em um caminho do caixeiro viajante usando curto-circuitos.
- Pela desigualdade triangular, o caminho do caixeiro viajante não pode ser mais longo do que o caminho Euleriano e, conseqüentemente, de comprimento no máximo duas vezes o comprimento da AGM.

Casamento mínimo com pesos

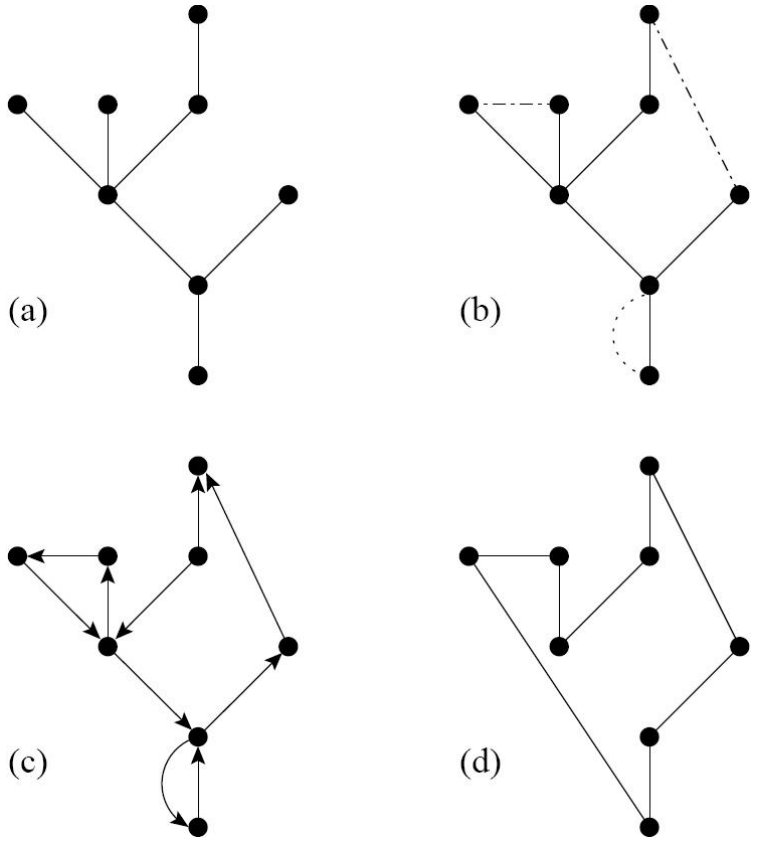
- Christophides propôs uma melhoria no algoritmo anterior utilizando o conceito de **casamento mínimo com pesos** em grafos.
- Dado um conjunto contendo um número par de cidades, um casamento é uma coleção de arestas M tal que cada cidade é a extremidade de exatamente um arco em M.
- Um casamento mínimo é aquele para o qual o comprimento total das arestas é mínimo.
- Todo vértice é parte de exatamente uma aresta do conjunto M.
- Pode ser encontrado com custo $O(n^3)$.



Casamento mínimo com pesos

- Considere a AGM T de um grafo.
- Alguns vértices em T já possuem grau par, assim não precisariam receber mais arestas se quisermos transformar a árvore em um grafo Euleriano.
- Os únicos vértices com que temos de nos preocupar são os vértices de grau ímpar.
- Existe sempre um número par de vértices de grau ímpar, desde que a soma dos graus de todos os vértices tenha de ser par porque cada aresta é contada exatamente uma vez.
- Uma maneira de construir um grafo Euleriano que inclua T é simplesmente obter um casamento para os vértices de grau ímpar.
- Isto aumenta de um o grau de cada vértice de grau ímpar. Os de de grau par não mudam.
- Se adicionamos em T um casamento mínimo para os vértices de grau ímpar, obtemos um grafo Euleriano que tem comprimento mínimo dentre aqueles que contêm T.

Casamento mínimo com pesos: Exemplo

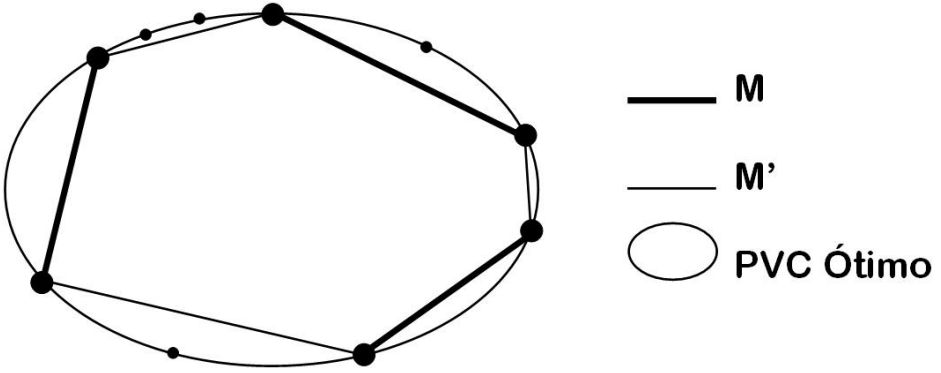


(a) Uma árvore geradora mínima T.
(b) T mais um casamento mínimo dos vértices de grau ímpar.
(c) Caminho de Euler em (b).
(d) Busca em profundidade com curto-circuito.

Casamento mínimo com pesos

- Basta agora determinar o comprimento do grafo de Euler.

- Caminho do caixeiro viajante em que podem ser vistas seis cidades correspondentes aos vértices de grau ímpar enfatizadas.



- O caminho determina os casamentos M e M'.

Casamento mínimo com pesos

- Seja I uma instância do PCV, e $Comp(T)$, $Comp(M)$ e $Comp(M')$, respectivamente, a soma dos comprimentos de T, M e M'.
- Pela desigualdade triangular devemos ter que:

$$Comp(M) + Comp(M') \leq \acute{O}timo(I).$$

- Assim, ou M ou M' tem de ter comprimento menor ou igual a $\acute{O}timo(I)/2$.
- Logo, o comprimento de um casamento mínimo para os vértices de grau ímpar de T tem também de ter comprimento no máximo $\acute{O}timo(I)/2$.
- Desde que o comprimento de M é menor do que o caminho do caixeiro viajante ótimo, podemos concluir que o comprimento o grafo Euleriano construído é:

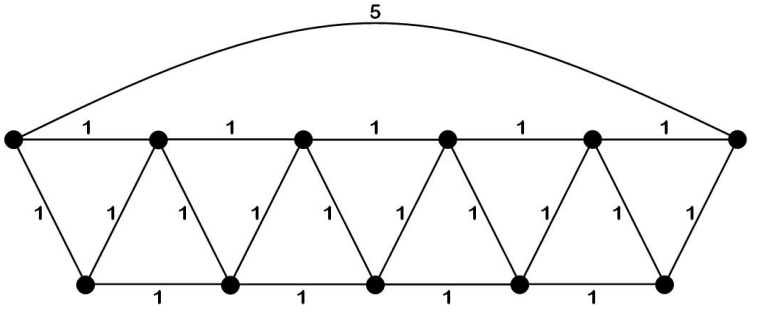
$$Comp(I) < \frac{3}{2}\acute{O}timo(I).$$

Casamento mínimo com pesos: Algoritmo de Christophides

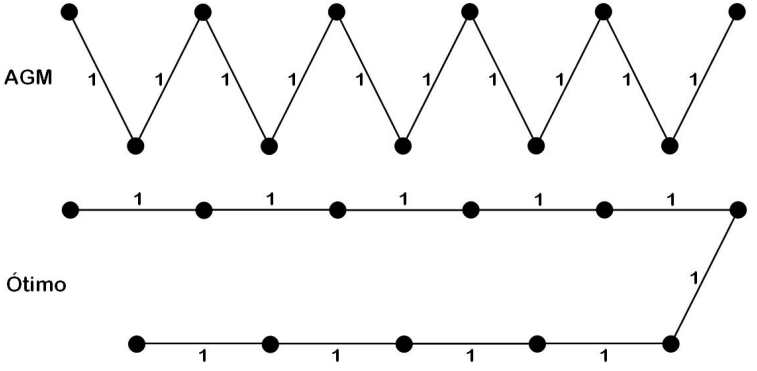
- Os principais passos do algoritmo de Christophides são:
  1. Obtenha a AGM T para o conjunto de n cidades, com custo $O(n^2)$.
  2. Construa um casamento mínimo M para o conjunto de vértices de grau ímpar em T, com custo $O(n^3)$.
  3. Encontre um caminho de Euler para o grafo Euleriano obtido com a união de T e M, e converta o caminho de Euler em um caminho do caixeiro viajante usando curto-circuitos, com um custo de $O(N)$.
- Assim obtivemos um algoritmo polinomial de custo $O(n^3)$, com uma razão de aproximação garantida para o pior caso de $R_A < \frac{3}{2}$.

Algoritmo de Christophides: Pior caso

- Exemplo de pior caso do algoritmo de Christofides:



- A AGM e o caminho ótimo são:



- Neste caso, para uma instância I:

$$C(I) = \frac{3}{2}[\acute{O}timo(I) - 1],$$

onde o $\acute{O}timo(I) = 11$, $C(I) = 15$, e $AGM = 10$.