Teoria da Complexidade

Rômulo César Silva

Unioeste

Julho de 2020

unioeste
Universidade Estadual do Oeste do Paraná

Sumário

Teoria da Complexidade

O estudo da complexidade tem 2 abordagens principais:

1. complexidade de **algoritmos**: foca em técnicas para o cálculo da complexidade de tempo e espaço de algoritmos
2. complexidade de **problemas**: procura agrupar os problemas em classes de complexidade, que de alguma forma, traduzem a dificuldade de obter algoritmos eficientes que os resolvam.

Aqui, o enfoque é o da segunda abordagem.

Teoria da Complexidade

Problema

Um **problema** é uma questão geral que pode ser descrita em termos de seus dados ou parâmetros de entrada e da caracterização de sua solução (saída).

Ex.: ordenação de números inteiros.
Dados de entrada: N números inteiros.
Saída: Os N números inteiros ordenados crescentemente.

Categorias de Problemas

Do ponto de vista da estrutura da solução, os problemas podem ser divididos em 3 categorias:

- **Problemas de Decisão**: a solução consiste em decidir se uma instância do problema satisfaz uma determinada propriedade. Ou seja, a resposta é simplesmente SIM ou NÃO.
- **Problemas de Localização**: a solução corresponde a encontrar uma estrutura que satisfaz uma determinada propriedade.
- **Problemas de Otimização**: a solução corresponde a encontrar uma estrutura que satisfaz uma determinada propriedade que atenda algum critério de otimização através da comparação de soluções.

Categorias de Problemas

A questão central de um mesmo problema geralmente pode ser formulada de diferentes maneiras, tal que o problema possa ser categorizado como de **decisão**, ou **localização** ou ainda **otimização**.

Exemplo:

- **Decisão**: Dado um grafo G, G contém clique de tamanho k?
- **Localização**: Dado um grafo G, encontre clique de tamanho k.
- **Otimização**: Dado um grafo G, qual a maior clique?

Eficiência de Algoritmos

Quanto à existência de algoritmos eficientes, um problema é considerado:

1. **Tratável**: se existe algoritmo eficiente para resolvê-lo, ou seja, se pode ser solucionado em **tempo polinomial**
2. **Intratável**: se não se conhece algoritmo eficiente para resolvê-lo.

Cota Superior de Complexidade

A **cota superior** de complexidade de um problema é a complexidade de tempo do algoritmo mais eficiente conhecido para resolvê-lo.

- a **cota superior** de complexidade de um problema pode ser melhorada caso seja encontrado outro algoritmo mais eficiente.

Ex.: multiplicação de matrizes pode ser feita em $O(n^3)$. Essa era a cota superior para multiplicação de matrizes até que Strassen desenvolveu um algoritmo $O(n^{\log 7})$. Atualmente (ano base: 2020), a cota superior é de $O(n^{2.376})$ devido ao algoritmo desenvolvido por Coppersmith e Winograd.

Cota Inferior de Complexidade

A **cota inferior** de complexidade de um problema corresponde à complexidade de tempo mínima possível para resolvê-lo.

- nem sempre a cota inferior de um problema é conhecida
- em alguns casos pode-se demonstrar matematicamente ser impossível conseguir um algoritmo assintoticamente mais eficiente. Ex.: o problema da ordenação *usando comparação* tem cota inferior de $\Omega(n\,lg\;n)$
- quando um algoritmo tem complexidade exatamente igual à cota inferior do problema, diz-se que o algoritmo é **ótimo**. Ex.: o algoritmo *Heapsort* para ordenação, que usa comparação, cuja complexidade é $O(n\,lg\;n)$ para o pior caso.

Cotas Inferior e Superior de Complexidade

- Tanto no caso de **cota inferior** quanto de **cota superior** é considerado o **pior caso**.
- Observe que enquanto a **cota superior** é dependente de um algoritmo específico, a **cota inferior** é dependente do problema em si.

Redução entre Problemas

Reduzir um problema a outro significa que de alguma maneira o primeiro pode ser convertido no segundo. Ou seja, o primeiro pode ser visto como um caso particular do segundo.

Essa ideia de redução é extramente útil porque na prática permite que se aplique um algoritmo já conhecido para resolver outro problema de natureza aparentemente diferente para o qual ele foi inicialmente projetado.

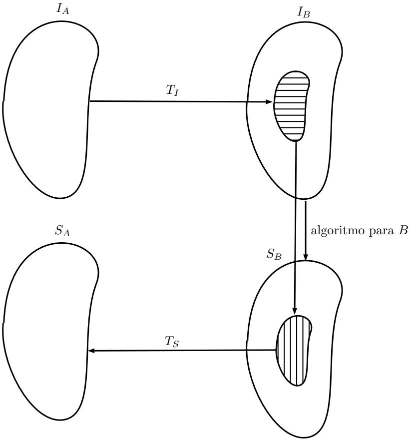
Redução entre Problemas

Uma **redução** do problema A para o problema B, denotada por $A \propto B$, é formalizada da seguinte maneira:

- Problema A:
  - I_A: instância genérica de entrada do problema A
  - S_A: solução do problema A para I_A
- Problema B:
  - I_B: instância genérica de entrada do problema B
  - S_B: solução do problema B para I_B
- existe uma transformação que converte I_A em I_B
- existe uma transformação que converte S_B em S_A

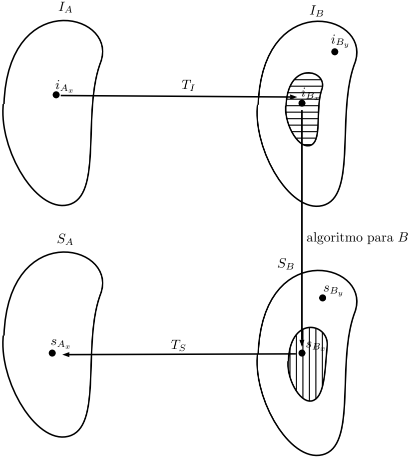
Observe que se conhece-se algoritmo que resolve o problema B, automaticamente pode-se utilizá-lo para resolver o problema A: primeiro usa-se a transformação que converte a entrada de A em entrada de B, depois aplica-se o algoritmo para B, e em seguida usa-se a transformação que converte a solução de B em solução de A.

Redução entre Problemas



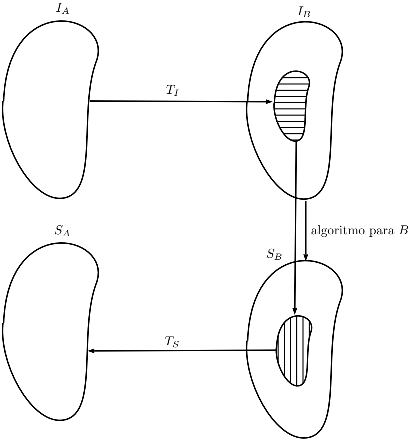
- deve ser possível converter todo o conjunto de instâncias do problema A em um subconjunto de instâncias do problema B
- as soluções correspondentes a esse subconjunto de instâncias do problema B devem corresponder às soluções do problema A

Redução entre Problemas



- a instância i_{A_x} é convertida em i_{B_x} pela transformação T_I
- o algoritmo para o problema B aplicado à instância i_{B_x} leva à solução s_{B_x}, que por sua vez é convertida na solução s_{A_x} pela transformação T_S.
- podem existir instâncias como i_{B_y} que não tem instância correspondente no problema A

Redução entre Problemas



$A \propto B$:

- problema A pode ser visto como um caso particular do problema B. Logo resolver B é "pelo menos tão difícil" quanto resolver A
- ter algoritmo para resolver A não significa ter algoritmo para resolver B

Redução entre Problemas

A redução $A \propto B$ é útil quando:

- já tem-se algoritmo que resolve B e deseja-se obter algoritmo que resolve A. Consequentemente fica estabelecida uma *cota superior* de complexidade para A
- conhece-se *cota inferior* de complexidade para A e deseja-se estabelecer uma *cota inferior* para B

Redução entre Problemas - exemplo

Problema ORD: Ordenação

- **entrada**: sequência de n números: $x_0, x_1, \ldots, x_{n-1}$
- **saída**: uma permutação $y_0, y_1, \ldots, y_{n-1}$ da sequência de entrada tal que $y_i \leq y_j$ sempre que $i < j$

Problema EMP: Emparelhamento

- **entrada**: duas sequências de inteiros $X = (x_0, x_1, \ldots, x_{n1})$ e $Y = (y_0, y_1, \ldots, y_{n1})$
- **saída**: um emparelhamento dos elementos nas duas sequências, de modo que o menor valor em X seja emparelhado com o menor valor em Y, o próximo valor menor em X seja emparelhado com o próximo valor menor em Y e assim por diante.

Redução entre Problemas - exemplo

ORD:

- entrada: $11, 2, 7, 9, 5, 17, 8$
- saída: $2, 5, 7, 8, 9, 11, 17$

EMP:

- entrada: $X = (23, 42, 17, 93, 88, 12, 57, 90)$ e
  $Y = (48, 59, 11, 89, 12, 91, 64, 34)$
- saída: $((12, 11), (17, 12), (23, 34), (42, 48), (57, 59), (88, 64), (90, 89), (93, 91))$

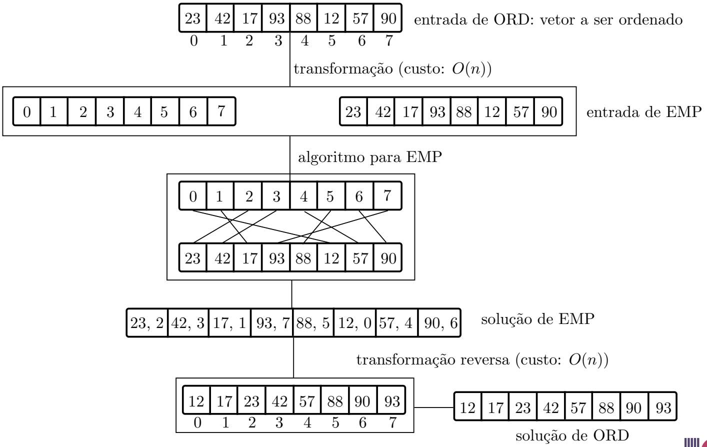
Redução entre Problemas - exemplo

EMP $\propto$ ORD:

- é fácil observar que EMP pode ser reduzido a ORD: basta usar algum algoritmo de ordenação para ordenar a sequência X e a sequência Y separadamente, e em seguida o emparelhamento é feito com base nas posições das sequências ordenadas.
- uma entrada de EMP é convertida em duas entradas de ORD.
- duas soluções de ORD são combinadas para formar uma solução de EMP.
- note que EMP não requer que os elementos das duas sequências sejam ordenados. Isto é meramente uma maneira possível de solucionar o problema.

Redução entre Problemas - exemplo

Um exemplo da redução contrária: ORD $\propto$ EMP



Redução entre Problemas

Observação!: O fato de $A \propto B$ não garante que $B \propto A$.

Redução entre Problemas

Definição. Um problema A é redutível a B em tempo $f(n)$ se n é o tamanho da entrada do problema A e o custo das transformações $I_A \rightarrow I_B$ e $S_B \rightarrow S_A$ é $O(f(n))$.

Notação: $A \propto_{f(n)} B$

Se $f(n) \in O(n^k)$ para k real, dizemos que a redução é polinomial.

Redução entre Problemas

- Se $A \propto_{f(n)} B$ é redução polinomial, então significa que se o algoritmo para o problema B tem complexidade polinomial, então A também tem complexidade polinomial.
- $\propto$ é relação transitiva: se $A \propto B$ e $B \propto C$ então existe redução $A \propto C$.
- Se não existe algoritmo polinomial para A então NÃO existe algoritmo polinomial para B
- Caso $A \propto_{f(n)} B$ e $B \propto_{g(n)} A$, em que $f(n) \in O(n^r)$ e $g(n) \in O(n^s)$, então A e B são polinomialmente equivalentes.

Redução entre Problemas - outro exemplo

Problema MULTMAT: Multiplicação de Matrizes Arbitrárias

- entrada: 2 matrizes $A_{n \times k}$ e $B_{k \times m}$
- saída: matriz $C_{n \times m} = A.B$

Problema MULTMATSIM: Multiplicação de Matrizes Simétricas

- entrada: 2 matrizes simétricas $A_{n \times n}$ e $B_{n \times n}$
- saída: matriz $C_{n \times n} = A.B$

Relembrando: uma matriz A é dita simétrica se $A = A^T$, ou seja, se ela é igual a sua transposta.

Redução entre Problemas - outro exemplo

Pode-se mostrar que existe a redução MULTMAT $\propto$ MULTMATSIM, ou seja, a multiplicação de matrizes arbitrárias é uma caso particular de multiplicação de matrizes simétricas.

- dadas matrizes A e B de entrada do problema MULTMAT transforme-as na entrada para MULTMATSIM construindo as matrizes de ordem $2n$:
  $A' = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$ e $B' = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}$
- aplicando o algoritmo para MULTMATSIM obtém-se como solução: $C' = A'.B' = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}$
- a conversão da solução C' de MULTMATSIM para a solução C de MULTMAT corresponde a copiar os elementos das n primeiras linhas e n primeiras colunas de C'.

Redução entre Problemas - outro exemplo

- o custo da transformação $I_{MULTMAT} \rightarrow I_{MULTMATSIM}$ é $O(n^2)$
- o custo da transformação $S_{MULTMATSIM} \rightarrow S_{MULTMAT}$ é $O(n^2)$
- considerando C_{ALG} o custo do algoritmo para o problema MULTMATSIM, o custo total da redução é: $C_{I_{MULTMAT} \rightarrow I_{MULTMATSIM}} + C_{ALG} + C_{S_{MULTMATSIM} \rightarrow S_{MULTMAT}}$

unioeste
Universidade Estadual do Oeste do Paraná

Classes de Problemas

A ideia é agrupar os problemas em classes de complexidade.

Nesta classificação, são considerados apenas **problemas de decisão**, isto é, aqueles cuja solução é SIM ou NÃO.

Embora pareça isso ser uma limitação, é fácil mostrar que problemas de localização e problemas de otimização podem ser convertidos em problemas de decisão, dependendo de como é feita sua formulação.

Classe **P**

P corresponde ao conjunto de problemas que podem ser resolvidos por algum algoritmo *determinístico polinomial*.
Ou seja, a cota superior seja limitada por $O(n^k)$.

Exemplos:

- Multiplicação de Matrizes: a existência do algoritmo tradicional, que é $0(n^3)$ já garante que o problema seja da classe **P**
- Ordenação de vetores: a existência do *Heapsort*, cuja complexidade é $O(n\,lg\,n)$ para o pior caso.
- Menor caminho em grafos: para grafos sem pesos negativos, o algoritmo de Dijkstra é $O(n^2)$. E para grafo com pesos negativos, o algoritmo de Floyd-Warshall é $O(n^3)$

Classe **NP**

NP é a classe de problemas que podem ser resolvidos por um algoritmo *não-determinístico polinomial*.

Observação: um algoritmo não-determinístico é um algoritmo executado em máquina não-determinística. Logo é apenas uma abstração.

Algoritmo Não-determinístico

Um algoritmo não-determinístico tem as seguintes características:

- é dividido em 2 fases:
  1. **fase de construção**: fase em que se constrói uma proposta de solução do problema
  2. **fase de verificação**: fase em que se certifica se a proposta de solução construída é de fato solução.
- a fase de verificação é determinística e polinomial, e retorna SIM caso a proposta de solução construída seja de fato solução ou NÃO caso contrário

Algoritmo Não-determinístico

- a fase de construção é não-determinística, em que geralmente é a feita a escolha dos elementos que irão compor a solução.
  - a questão é: que elementos escolher? Em que ordem?
  - essas perguntas podem ser abstraídas supondo a existência de um comando `guess` de complexidade $O(1)$ que "adivinha" qual o elemento deve ser escolhido dentro de um conjunto de n elementos.
- A parte correspondente à fase de verificação é também chamada de **certificado** porque ela certifica *deterministicamente* se a fase de construção obteve uma solução de fato.

Algoritmo Não-determinístico - exemplo

Considere o seguinte problema de decisão: Dado um grafo $G = (V, E)$ e um inteiro $1 \leq k \leq n$ onde $|V| = n$, determine se G tem clique de tamanho k:

- entrada: grafo G e inteiro K
- saída: SIM caso G tenha clique de tamanho k, NÃO caso contrário

Algoritmo Não-determinístico - exemplo

Algoritmo 1 CliqueNaoDeterministico

```
Input: grafo G = (V, E) e inteiro k
Output: SIM ou NÃO
/* Fase de Construção                                    */
S ← V
C ← {}
for i ← 1 to k do
    u ← Guess(S)
    S ← S − {u}
    C ← C ∪ u
end
/* Fase de Verificação                                   */
for ∀u, v distintos em C do
    if aresta (u, v) ∉ E then
        return NÃO
    end
    return SIM
end
```

Classe **NP**

- a classe **NP** também pode ser cararcterizada como sendo o conjunto de todos os problemas que admitem um *certificado* polinomial.
- Como todo algoritmo determinístico é um caso particular de algoritmo não-determinístico, então $\mathbf{P} \subseteq \mathbf{NP}$.

Questão Fundamental da Teoria da Computação: $\mathbf{P} = \mathbf{NP}$? Ou seja, $\mathbf{NP} \subseteq \mathbf{P}$?

Esta questão está em aberto. Há um prêmio de U$1 mi para quem solucioná-la. Veja:

https://www.claymath.org/millennium-problems

Classes **NP-difícil** e **NP-completo**

Definição. Um problema de decisão A é **NP-difícil** se todos os problemas da classe **NP** são polinomialmente redutíveis a A.

Ou seja, um problema é **NP-difícil** se for pelo menos tão difícil quanto qualquer problema em NP.

Um problema é **NP-completo** se for **NP-difícil** e estiver em **NP**.

Classe **NP-completo**

Stephen Cook demonstrou a existência de um problema **NP-completo**: o problema SAT (Problema da Satisfatibilidade)

SAT: dada uma fórmula lógica $\mathcal{F}$ na forma normal conjuntiva, existe atribuição de valores às variáveis lógicas para a qual $\mathcal{F}$ é verdadeira?

- variáveis lógicas: $x_1, ..., x_n$ e suas negações $(\overline{x_1}, ..., \overline{x_n})$
- operadores lógicos: $+$ (or) , . (and)
- cláusulas: $C_1, C_2, ..., C_m$ da forma $C_i = (x_{i1} + x_{i2} + ...)$
- $\mathcal{F} = C_1.C_2...C_m$

Ex.: $\mathcal{F} = (x_1 + x_2 + \overline{x_3}).(\overline{x_1} + \overline{x_2} + x_3).(x_1 + \overline{x_3})$ é verdadeira para $x_1 = 1$ e $x_2 = x_3 = 0$.

Universidade Estadual do Oeste do Paraná

Classe **NP-completo**

Com a descoberta de que SAT é **NP-completo** foi possível demonstrar que vários outros problemas também são **NP-completo** usando reduções polinomiais.

Assim para demonstrar que um problema A é **NP-completo** é preciso:

1. mostrar que A está em **NP**. Isto é feito escrevendo algoritmo não-determinístico polinomial para A, geralmente uma tarefa trivial
2. encontrar uma redução polinomial de um problema B **NP-completo** já conhecido para o problema A, ou seja, $B \propto_{poli} A$.

Classe **NP-completo** - exemplos

- **Clique**: Dado um grafo não orientado G e um número k, G tem uma clique com mais que k vértices?
- **Mochila Binária**: Dados conjunto U de n objetos , dois valores inteiros w_i (peso) e c_i (custo) associados a cada objeto, existe subconjunto $Z \subseteq U$ tal que $\Sigma_{u_i \in Z} w_i \leq W$ e $\Sigma_{u_i \in Z} c_i \geq C$?
- **Ciclo Hamiltoniano**: Dado um grafo G, existe um ciclo simples que passe por todos os vértices de G?
- **Campo Minado**: Dado um tabuleiro parcialmente preenchido do jogo *Minesweeper*, existe uma disposição de minas que seja consistente com o tabuleiro?
- **Cobertura de Vértices**: Dado um grafo G não orientado e um inteiro k, G tem cobertura com menos que k vértices?

Classe **NP-completo** - exemplos

- **3-SAT**: versão de SAT em que cada cláusula contém exatamente 3 literais
- **Conjunto Independente**: dado um grafo não orientado G e um inteiro k, existe conjunto independente em G com k vértices?
- **3-Coloração**: dado um grafo G, G pode ser colorido (coloração de vértices) com 3 cores?
- **Partição**: dado um conjunto V de n elementos e valor inteiro f_i associado a cada elemento, existe subconjunto $X \subset V$ tal que $\Sigma_{v_i \in X} f_i = \Sigma_{v_i \in V - X} f_i$?

unioeste
Universidade Estadual do Oeste do Paraná

Classe **NP-completo** - exemplos

- **Caixeiro Viajante** ou *Traveling Salesman Problem* (TSP): Um *tour* em um conjunto de cidades é uma viagem que começa e termina na mesma cidade e passa por todas as outras cidades exatamente uma única vez.
  TSP: Dado um conjunto de cidades V e distâncias entre todos os pares de cidades em V, e um inteiro positivo D, existe um *tour* das cidades em V cuja distância total é menor que D?
- **Conjunto Dominante**: Dado um grafo $G = (V, E)$, um conjunto dominante em G é um subconjunto de vértices X tal que para todo vértice $v \in V$, $v \in X$ ou existe vértice $x \in X$ tal que existe aresta $(x, v) \in E$.
  Dado um grafo não orientado G e um inteiro k, existe conjunto dominante com k vértices?
- ... a lista é grande! Lista da Wikipedia:

https://en.wikipedia.org/wiki/List_of_NP-complete_problems

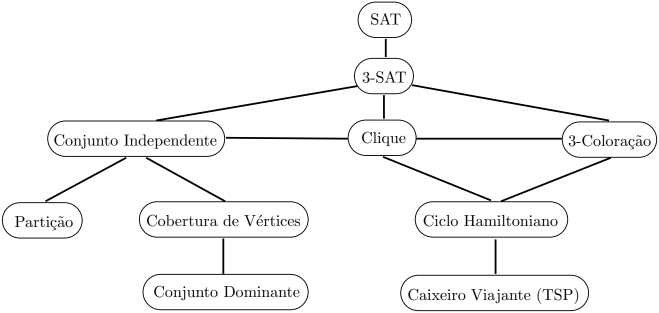
Teoria da Complexidade

Alguns resultados importantes:

- **Teorema de Cook**: SAT é **NP-completo**
- Todos os problemas em **NP-completo** são polinomialmente equivalentes
- se existir algoritmo determinístico polinomial para algum problema em **NP-completo** então existe algoritmo determinístico polinomial para todos eles.
- **P**=**NP** se e somente se existir algoritmo determinístico polinomial para algum problema em **NP-completo**

Reduções entre Problemas **NP-completo**

A figura abaixo mostra algumas reduções além de outras possíveis.



Exemplo de Prova de **NP-completude**

Vamos provar que **SAT** $\propto_{poli}$ **Clique**.

Seja $\mathcal{F} = C_1 C_2 ... C_m$ uma expressão booleana arbitrária com variáveis $x_1, x_2, ..., x_n$. Construa um grafo da seguinte maneira:

1. crie um vértice para cada literal que aparece em uma cláusula
2. adicione arestas ligando os vértices de diferentes cláusulas a menos que seja uma variável e seu complemento
3. vértices da mesma cláusula não são conectados

O grafo assim construído tem uma clique de tamanho $\geq m$ se e somente se $\mathcal{F}$ é satisfazível.

- custo da transformação: número de vértices do grafo é $O(nm)$ e o número de arestas é $O(n^2 m^2)$

Exemplo de Prova de **NP-completude**

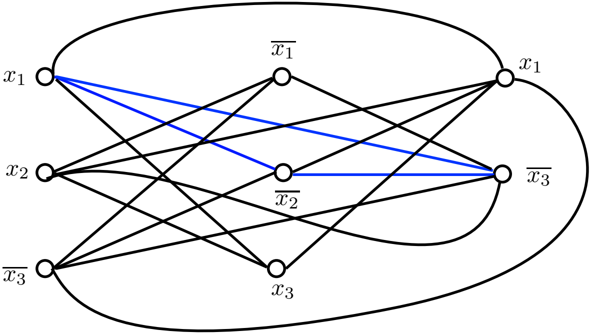
A construção garante que o tamanho da clique maximal não é maior que m.

Suponha que $\mathcal{F}$ seja satisfazível. Então existe uma atribuição de valores-verdade tal que cada cláusula tenha pelo menos uma variável com valor-verdade *true*.

É possível afirmar que os vértices correspondendo a *true* em G estão conectados porque os vértices escolhidos não podem ter aresta ligando a seu complemento.
Isto significa que o subgrafo resultante é uma clique.

Exemplo de Prova de **NP-completude**

Grafo equivalente à fórmula $(x_1 + x_2 + \overline{x_3}).(\overline{x_1} + \overline{x_2} + x_3).(x_1 + \overline{x_3})$



Exemplo de Prova de **NP-completude**

Inversamente, assuma que G contém uma clique de tamanho $\geq m$.

A clique deve consistir de m vértices de "colunas" de cláusulas distintas. Podemos atribuir à variável correspondente um valor-verdade *true* e a seus complementos *false*, e às demais variáveis valores arbitrários.

Desde que todos os vértices na clique estão conectados, e sabemos que um par de vértices representando uma variável e seu complemento nunca está conectado, esta atribuição de valores é consistente e portanto satisfazível.

Outras Classes de Complexidade

PSPACE é a classe de problemas que admitem algoritmos determinísticos que usam espaço polinomial no tamanho da entrada.

NPSPACE é a classe de problemas que admitem algoritmos não-determinísticos que usam espaço polinomial no tamanho da entrada.

Já foi demonstrado que **PSPACE**=**NPSPACE**.

Outra Classes de Complexidade

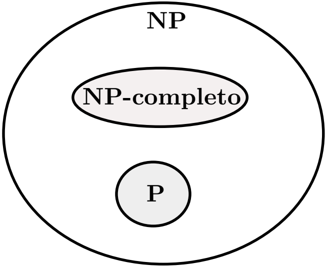
EXPTIME é a classe de problemas solúveis deterministicamente em tempo exponencial.

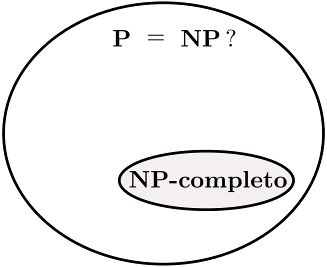
Já se sabe que:

- **P**$\neq$**EXPTIME**
- **P**$\subseteq$**PSPACE**
- **NP**$\subseteq$**PSPACE**
- **PSPACE**$\subseteq$**EXPTIME**

Relações entre Classes de Complexidade

Uma das possibilidades abaixo ocorre. A da direita se **P**=**NP**, que está ainda em aberto, for verdade.



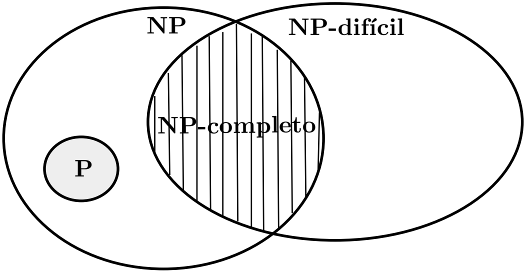


Relações entre Classes de Complexidade

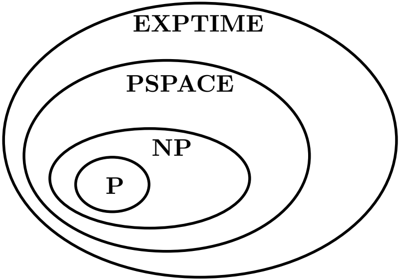
A classe **NP-difícil**$\neq$**NP**, pois todos os problemas de otimização cuja versão de decisão estão em **NP-completo** estão em **NP-difícil**.

Além disso, já foi demonstrado que o problema SAT se reduz polinomialmente ao problema da Parada da Máquina de Turing (HALT). Ou seja, HALT é um problema **NP-difícil**, mas não é **NP-completo**, pois é um problema indecidível.

Relações entre Classes de Complexidade



Relações entre Classes de Complexidade



Teoria da Complexidade

- Dizer que um problema está em **NP** (**P**, **PSPACE**) nos dá um limitante superior sobre sua dificuldade (complexidade)
- Dizer que um problema está em **NP-difícil** nos dá um limitante inferior sobre sua dificuldade, significando que ele é pelo menos tão difícil de se resolver quanto qualquer outro problema em **NP**
- Dizer que um problema é **NP-completo** significa que se tem um limitante superior e um limitante inferior sobre sua complexidade.

O que fazer com um problema **NP-completo**?

Utilizar de outras estratégias para resolvê-lo:

- algoritmos aproximados: ao invés de procurar por uma solução exata, procurar uma solução aproximada que seja admissível dentro do contexto.
- técnicas de IA:
  - Métodos Heurísticos
  - *Simulated Annealing* (Têmpera Simulada)
  - Algoritmos Genéticos
  - Colônia de Formigas
  - Enxame de Partículas
  - ...

Bibliografia I

[OpenDSA 2020] OpenDSA Project Contributors.
Reductions. MIT, disponível em:
https://opendsa-server.cs.vt.edu/ODSA/Books/CS4104/html/Reduction.html#reductions, Acesso em 24/07/2020.

[Kleinberg 2014] Kleinberg, J.; Tardos, E.
Algorithm Design. Pearson, Londres:2014.

[Garey 1979] Garey, M. R.; Johnson, D. S.
Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman & Co., 1979.