UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
Departamento de Ciências de Computação

SCC-205 - Capítulo 5
Computabilidade e Complexidade

João Luís Garcia Rosa[1]

[1]Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
http://www.icmc.usp.br/~joaoluis

2012

Sumário

Sumário

Máquinas de Turing com Múltiplas Cabeças/Fitas

- Diz-se que uma cadeia w é aceita por uma máquina de Turing de 2 cabeças e 2 fitas se, quando w é escrita na fita 1 e a cabeça 1 percorre o símbolo mais a esquerda de w no estado q_0, e a fita 2 está inicialmente em branco, M finalmente para em seu estado de aceitação q_a.

Máquinas de Turing Não Determinísticas

- **Teorema**: Seja L uma linguagem que pode ser reconhecida por uma máquina de Turing de k cabeças e k fitas. Então $L = T(M)$ para alguma máquina de Turing M de 1 cabeça e 1 fita.
- **Definição**: Uma máquina M é chamada de máquina de Turing **não determinística** se ela incluir quíntuplas que especifiquem movimentos múltiplos para um dado par estado/símbolo, isto é, se ela é definida como na Definição de Máquina de Turing do capítulo 3, exceto que agora δ mapeia $Q \times \Sigma'$ a subconjuntos de $Q \times \Sigma' \times \{L, R, S\}$.

Máquinas de Turing Não Determinísticas

- Diz-se que uma máquina de Turing não determinística M aceita uma cadeia w se existir alguma cadeia de transições da máquina na entrada w que alcança uma DI de parada que inclui o estado de aceitação q_a.
- A definição de aceitação está de acordo com a definição de aceitação não determinística dada para aceitadores de estados finitos não determinísticos.
- O próximo resultado mostra que, assim como com os AFNs, o não determinismo não adiciona potência computacional nova ao modelo determinístico original.

Máquinas de Turing Não Determinísticas

- **Teorema**: Seja M uma máquina de Turing não determinística que aceita L. Então existe uma máquina determinística M' que também aceita L.
- **PROVA**: Vai-se construir uma máquina determinística M' de 2 fitas e 2 cabeças que age como um interpretador para M. Em sua segunda fita M' mantém um registro das possíveis DIs ativas de M em qualquer instante de sua computação. Por exemplo, se M inclui as instruções q_0

$$(q_0\ a\ q_i\ b\ R)$$
$$(q_0\ a\ q_j\ c\ L)$$

então depois de uma execução de instrução na entrada aa, a fita 2 de M' se parece com

$$\#\ b\ q_i\ a\ \#\ q_j\ B\ c\ a\#$$

Máquinas de Turing Não Determinísticas

- Usa-se o símbolo # para separar as codificações das DIs. Suponha que em algum instante a fita 2 se pareça com

$$\# \, DI_1 \, \# \, DI_2 \, \# \ldots \# \, DI_n \#$$

Então M' opera para formar o próximo conjunto de DIs processando a cadeia da fita 2 da esquerda para a direita. Quando processar o j-ésimo bloco, M' pode

1. apagar o bloco se nenhuma transição for possível; ou
2. substituir o bloco por um novo conjunto finito de blocos de DIs que representam as possíveis novas transições de M na DI dada. ♦

Sumário

Uma Linguagem que não é RE

- No capítulo anterior estudou-se um procedimento para enumerar as máquinas de Turing.
- Desta forma, é possível achar na enumeração a máquina de Turing M_i, a "i-ésima máquina de Turing".
- Imagine que seu código binário seja w_i.
- Muitos inteiros não correspondem a nenhuma máquina de Turing.
- Se w_i não é um código de máquina de Turing válido, M_i será a máquina de Turing com um estado e nenhuma transição.
- Ou seja, M_i é uma máquina de Turing que para para qualquer entrada.
- Portanto, $T(M_i)$ é $\emptyset$ se w_i falha ao ser um código de uma máquina de Turing válida.

Uma Linguagem que não é RE

- **Definição**: A linguagem L_d, a **linguagem de diagonalização**, é o conjunto de cadeias w_i tal que w_i não está em $T(M_i)$.
- Ou seja, L_d consiste de todas as cadeias w tal que a máquina de Turing M cujo código é w não aceita w.
- A razão do nome linguagem de "diagonalização" pode ser entendida através da figura.
- A tabela informa, para todo i (linha) e j (coluna), se a máquina de Turing M_i aceita a cadeia de entrada w_j: 1 significa "sim" e 0 significa "não".
- Pode-se pensar na i-ésima linha como o *vetor característico* para a linguagem $T(M_i)$; ou seja, os 1's nesta linha indicam as cadeias que são elementos desta linguagem.

Uma Linguagem que não é RE

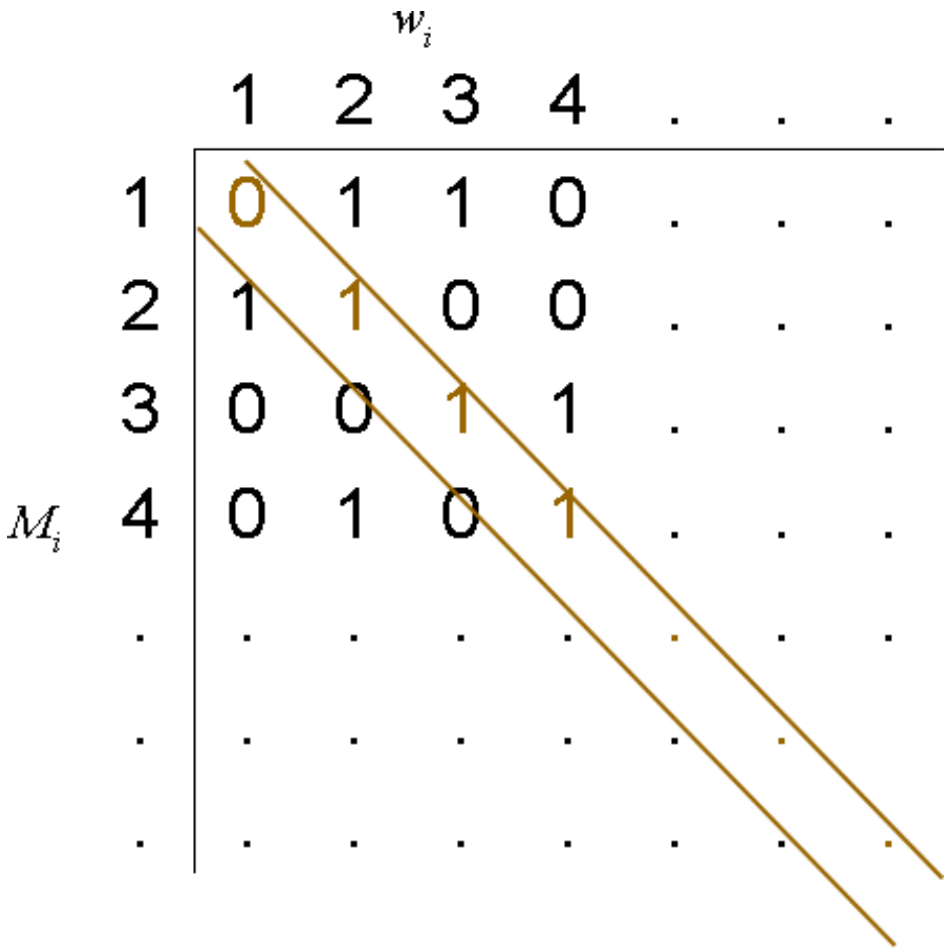


Figure : Tabela que representa aceitação de cadeias por máquinas de Turing.

Uma Linguagem que não é RE

- Os valores na diagonal dizem se M_i aceita w_i.
- Para construir a L_d, complementa-se a diagonal.
- Por exemplo, assumindo figura como correta, a diagonal complementada corresponde a 1, 0, 0, 0, ...
- Portanto, L_d contém $w_1 = \lambda$, não contém w_2 a w_4, que são 0, 1 e 00, e assim por diante (supondo que $\Sigma = \{0, 1\}$).
- O truque de complementar a diagonal para construir o vetor característico de uma linguagem que não pode ser a linguagem que aparece em nenhuma linha é chamado de **diagonalização**.

Uma Linguagem que não é RE

- Isto funciona porque o complemento da diagonal é ele próprio um vetor característico que descreve a pertinência em alguma linguagem, a L_d.
- Este vetor característico discorda em alguma coluna com toda linha da tabela.
- Portanto, o complemento da diagonal não pode ser o vetor característico de nenhuma máquina de Turing.
- **Teorema**: L_d não é uma linguagem recursivamente enumerável (RE). Ou seja, não há nenhuma máquina de Turing que aceite L_d.

Uma Linguagem que não é RE

- **PROVA**: Suponha que L_d seja $T(M)$ para alguma máquina de Turing M. Como L_d é uma linguagem sobre o alfabeto $\{0,1\}$, M estaria na lista das máquinas de Turing, já que esta lista inclui todas as máquinas de Turing com alfabeto de entrada $\{0,1\}$. Portanto, há pelo menos um código para M, por exemplo, $M = M_i$. Agora será que w_i está em L_d?
  - Se w_i está em L_d, então M_i aceita w_i. Mas então, pela definição de L_d, w_i não está em L_d, porque L_d contém apenas aqueles w_j tal que M_j **não** aceita w_j.
  - Similarmente, se w_i não está em L_d, então M_i não aceita w_i. Portanto, por definição de L_d, w_i **está** em L_d.

  Como w_i não pode estar e não estar em L_d ao mesmo tempo, há uma contradição na suposição de que M existe. Ou seja, L_d não é uma linguagem recursivamente enumerável. ♦

Sumário

Algoritmo

- Um algoritmo[1] é um conjunto finito de instruções que, se seguido, realiza uma determinada tarefa. Deve satisfazer os seguintes critérios [6]:
  1. **Entrada**: Zero ou mais quantidades são supridas;
  2. **Saída**: No mínimo uma quantidade é produzida;
  3. **Clareza**: Cada instrução é clara e não ambígua;
  4. **Finitude**: Se o algoritmo for percorrido passo a passo (*trace*), então em todos os casos, o algoritmo termina depois de um número finito de passos;
  5. **Efetividade**: Toda instrução deve ser muito básica, de tal forma que possa ser realizada, em princípio, por uma pessoa usando apenas lápis e papel. Não é suficiente que cada operação seja definida como no critério 3; ela também tem de ser factível.

[1] A palavra "algoritmo" vem do nome de um matemático persa (825 d.C.), Abu Ja'far Mohammed ibn Musa al Khowarizmi.

O Problema da Parada

- Limites da computação: máquinas de Turing são "potentes" e "flexíveis", mas têm limites.
- Uma questão prática para o cientista de computação: este programa processará dados da forma pretendida?
- Mesmo mais fundamental do que determinar **se** ou não um programa está **correto** é dizer **se** ou não ele **terminará**, fornecendo a resposta correta.
- Reescrevendo a última questão em termos das máquinas de Turing, então, tem-se o seguinte:

O Problema da Parada

Definição

O Problema da Parada: Dada uma máquina de Turing M_n com semântica φ_n e uma cadeia de dados w, $\varphi_n(w)$ é definida? Isto é, M_n sempre para se iniciada no estado q_0 percorrendo o quadrado mais a esquerda de w?

O Problema da Parada

- Seria muito bom se fosse possível achar algum testador de parada universal que, quando dado um número n e uma cadeia w, poderia efetivamente dizer se ou não a máquina M_n **sempre** parará se sua entrada de dados for w.
- Entretanto, o teorema dirá que tal máquina **não** existe.
- Antes de mostrar que nenhum procedimento efetivo pode resolver o problema da parada, veja por que o procedimento óbvio falha.
- Vai-se pegar a máquina universal U e "rodá-la" na fita $(e(M_n), w)$.
- Quando ela para, o controle é transferido para uma sub-rotina que imprime um 1 para significar que a computação de M_n em w parou realmente.

O Problema da Parada

- Mas quando o controle pode ser transferido para uma sub-rotina que imprime um 0 para significar que M_n **nunca para** com w?
- Depois de muitas simulações por U, é possível concluir que M_n **nunca** parará em w, ou que, depois de muito tempo, as computações pararão?
- Esta abordagem claramente falha.
- Mas para provar que todas as abordagens que tentam decidir a terminação algoritmicamente necessariamente falharão é tarefa bem menos óbvia, e para isso deve-se se basear no **argumento diagonal** usado por Cantor para mostrar que nenhuma enumeração poderia incluir todos os números reais.

Argumento diagonal de Cantor

$$
\begin{array}{lcccccccccccc}
E_0 = & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \cdots \\
E_1 = & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \mathrm{w} & \cdots \\
E_2 = & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \cdots \\
E_3 = & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{m} & \mathrm{w} & \cdots \\
E_4 = & \mathrm{w} & \mathrm{m} & \mathrm{m} & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \cdots \\
E_5 = & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \cdots \\
E_6 = & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \cdots \\
E_7 = & \mathrm{w} & \mathrm{m} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \cdots \\
E_8 = & \mathrm{m} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \cdots \\
E_9 = & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{m} & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \cdots \\
E_{10} = & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \cdots \\
E_{11} = & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{m} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\
E_u \neq & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{w} & \mathrm{m} & \mathrm{w} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{m} & \mathrm{w} & \cdots
\end{array}
$$

Figure : Uma ilustração do argumento diagonal de Cantor para a existência de conjuntos incontáveis. A sequência final (E_u) não pode ocorrer em nenhum outro lugar na listagem acima [10].

O Problema da Parada

Cantor

O matemático russo Georg Ferdinand Ludwig Philipp Cantor provou que os números reais não são contáveis em 1874. Ele produziu seu famoso "argumento diagonal" em 1890, que deu uma segunda prova, mais enfática e interessante, de que os números reais não são contáveis.

- Para antever o estudo do problema da parada, primeiro será fornecido um resultado que relaciona a enumeração das máquinas de Turing com a discussão de funções numéricas.
- Lembre-se que uma função Turing-computável ψ é **total** se ela retorna uma saída para toda entrada.

O Problema da Parada

- **Teorema**: Não existe nenhuma função Turing-computável total g que enumera as funções Turing-computáveis totais no seguinte sentido: a função Turing-computável ψ é total se e somente se ψ for igual à função $\varphi_{g(n)}$ computada por $M_{g(n)}$ para algum n.

O Problema da Parada

- **Teorema**: (**A Insolubilidade do Problema da Parada**). Considere a função *pare* : $\mathbb{N} \rightarrow \mathbb{N}$ definida como:

$$pare(x) = \begin{cases} 1 & \text{se } M_x \text{ para na entrada } x \\ 0 & \text{se } M_x \text{ nunca para em } x \end{cases}$$

Então *pare* **não é Turing-computável**.

O Problema da Parada

- Antes de provar este resultado, far-se-á algumas observações.
  - Primeiro, note que *pare* é uma função total com certeza: tem um valor definido para toda entrada.
  - O que se está tentando mostrar é que *pare* **não** é computável: não existe nenhum método (formalmente expresso como uma máquina de Turing) para calcular os valores de *pare*.
  - Note também que o teorema é muito conservador, pois considera apenas a não computabilidade de Turing.
  - De fato, se a tese de Church estiver subscrita, *pare* não é computável por nenhum algoritmo especificado em qualquer linguagem de programação.

O Problema da Parada

- **PROVA DO TEOREMA**: Suponha que se construa uma matriz $\mathbb{N} \times \mathbb{N}$, estabelecendo a entrada (i, j) para $\downarrow$ se a computação de $\varphi_i(j)$ termina, enquanto estabelece-se a $\uparrow$ caso contrário. Para construir uma função φ não no conjunto de φ's adota-se o argumento diagonal de Cantor: percorrendo a diagonal de cima para baixo, "inverte-se as setas," dando a $\varphi(x)$ o comportamento da parada oposto a $\varphi_x(x)$:

$$\varphi(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \text{ não é definido} \\ \perp & \text{se } \varphi_x(x) \text{ é definido} \end{cases}$$

Mas isto apenas diz que

$$\varphi(x) = \begin{cases} 1 & \text{se } \mathit{pare}(x) = 0 \\ \perp & \text{se } \mathit{pare}(x) = 1 \end{cases}$$

O Problema da Parada

- Agora se *pare* for computável, existe certamente uma máquina de Turing que computa φ.
- Entretanto, o argumento diagonal garante que φ **não é computável**, se φ for φ_j para algum j ter-se-ia $\varphi_j(j) = 1$ se $\varphi_j(j) = \perp$ enquanto $\varphi_j(j) = \perp$ se $\varphi_j(j) = 1$ - uma contradição.
- Assim nenhum método de cálculo para *pare* pode existir. ♦

- Portanto, exibiu-se um problema geral em ciência da computação, natural, interessante e valioso, que não tem solução algorítmica. (Este resultado é extremamente fundamental e está fortemente relacionado com o resultado famoso do lógico austríaco Kurt Gödel da incompletude das teorias formais da aritmética.)

Teorema da Incompletude de Gödel

Teorema da Incompletude de Gödel

O matemático alemão David Hilbert estabeleceu em 1900 que os matemáticos deveriam buscar expressar a matemática na forma de um sistema formal, consistente, completo e decidível. Em 1931, Gödel provou que o ideal de Hilbert era impossível de satisfazer, mesmo no caso da aritmética simples. Este resultado é conhecido como primeiro teorema da incompletude de Gödel. Mais tarde, em 1939, Turing e Church mostraram independentemente que nenhum sistema formal consistente da aritmética é decidível, nem mesmo a lógica de predicados de primeira ordem, considerado mais fraco, é decidível [3].

Problemas Indecidíveis

- **Definição**: Um conjunto $S \subset \mathbb{N}$ é **decidível** se a pertinência em S pode ser determinada algoritmicamente. Isto é, S é decidível (ou recursivo ou solúvel) se a função característica de S,

$$\chi_S(x) = \begin{cases} 1 & \text{se } x \in S \\ 0 & \text{se } x \notin S \end{cases}$$

  é Turing-computável.
- Se S não é decidível, diz-se que S é **indecidível**, ou insolúvel ou não recursivo ou, em casos que requer ênfase considerável, recursivamente insolúvel.

Problemas Indecidíveis

Algumas linguagens não são decidíveis

Lembre-se da enumeração feita com as máquinas de Turing. Como existem incontáveis linguagens e somente um número contável de máquinas de Turing (enumeração), conclui-se que algumas linguagens não são decidíveis por máquinas de Turing, nem mesmo reconhecidas por máquinas de Turing.

Problemas Indecidíveis

- Tem-se um exemplo de um conjunto indecidível: o conjunto

$$K = \{ n \mid \varphi_n(n) \text{ retorna um valor } \}$$

é recursivamente insolúvel. O próximo resultado mostra como mapear a indecidibilidade de K na teoria da linguagem.

- **Definição**: Seja G uma gramática do tipo 0 sobre algum alfabeto $V \cup \Sigma$. O **problema de dedução** para G é o problema de determinar, dadas as cadeias arbitrárias x, y em $(V \cup \Sigma)^*$, se $x \Rightarrow^* y$ em G.
- **Teorema**: O problema de dedução para gramáticas do tipo 0 é indecidível.

Problemas Indecidíveis

- Contrastando com o resultado anterior, tem-se o seguinte:
- **Teorema**: O problema de dedução para gramáticas sensíveis ao contexto é decidível.
- **Corolário**: O problema de dedução para gramáticas livres de contexto é decidível.
- **Definição**: Dada uma classe de gramáticas C, o **problema de esvaziamento** para C é o problema de determinar para G arbitrária em C, se a linguagem $L(G)$ é vazia.
- **Teorema**: O problema de esvaziamento para gramáticas livres de contexto é decidível.

Linguagens Recursivas

- Linguagens recursivamente enumeráveis (RE) são aceitas (reconhecidas) por máquinas de Turing.
- Linguagens RE podem ser agrupadas em duas classes:
  1. Classe 1 (**linguagens recursivas**): cada linguagem L nesta classe tem uma máquina de Turing (pensada como um algoritmo) que não apenas aceita cadeias de L, como também indica quais cadeias não estão em L através da **parada**.
  2. Classe 2 (**RE mas não recursivas**): cada linguagem L nesta classe tem uma máquina de Turing (não pensada como um algoritmo) que aceita cadeias de L, mas pode **não parar** quando uma cadeia de entrada não está em L.

Linguagens Recursivas

- **Definição**: Formalmente, uma linguagem L é **recursiva** se $L = T(M)$ para alguma máquina de Turing M tal que:
  1. Se $w \in L$, então M aceita (e portanto para no estado de aceitação).
  2. Se $w \notin L$, então M rejeita (para num estado de não aceitação).
- Uma máquina de Turing deste tipo corresponde à noção formal de algoritmo.
- Uma dada linguagem L, pensada como um problema, é chamada de **decidível** se L é uma linguagem recursiva; e **indecidível** caso contrário.
- A existência ou não existência de um algoritmo para resolver o problema (isto é, o problema é decidível ou indecidível) é mais importante do que a existência de uma máquina de Turing para resolver o problema.

Linguagens Recursivas

Problemas decidíveis

A classe de **problemas decidíveis** é equivalente à classe das linguagens recursivas.

Problemas parcialmente decidíveis

A classe dos **problemas parcialmente decidíveis** é equivalente à classe das linguagens recursivamente enumeráveis.

Decidibilidade e Aceitabilidade

Decidibilidade e Aceitabilidade

A noção de decidibilidade é mais restrita que a de aceitabilidade (ser reconhecível), uma vez que neste último caso, é permitido que a máquina de Turing nunca pare.

Decidibilidade = Algoritmo.
Aceitabilidade = Procedimento.

Reconhecedores são mais poderosos que decididores.

A Linguagem Universal

- **Definição**: A **linguagem universal** L_u é o conjunto de cadeias binárias, que codificam um par (M, w), onde M é uma máquina de Turing com o alfabeto de entrada binário e w é uma cadeia em $(0 + 1)^*$, tal que w está em $T(M)$. Ou seja, L_u é o conjunto de cadeias que representam uma máquina de Turing e uma entrada aceita por esta máquina de Turing. A **máquina de Turing universal** U processa L_u, ou seja, $L_u = T(U)$.
- L_u não é recursiva, apesar de ser uma linguagem RE.

Linguagens Recursivas, RE e não-RE

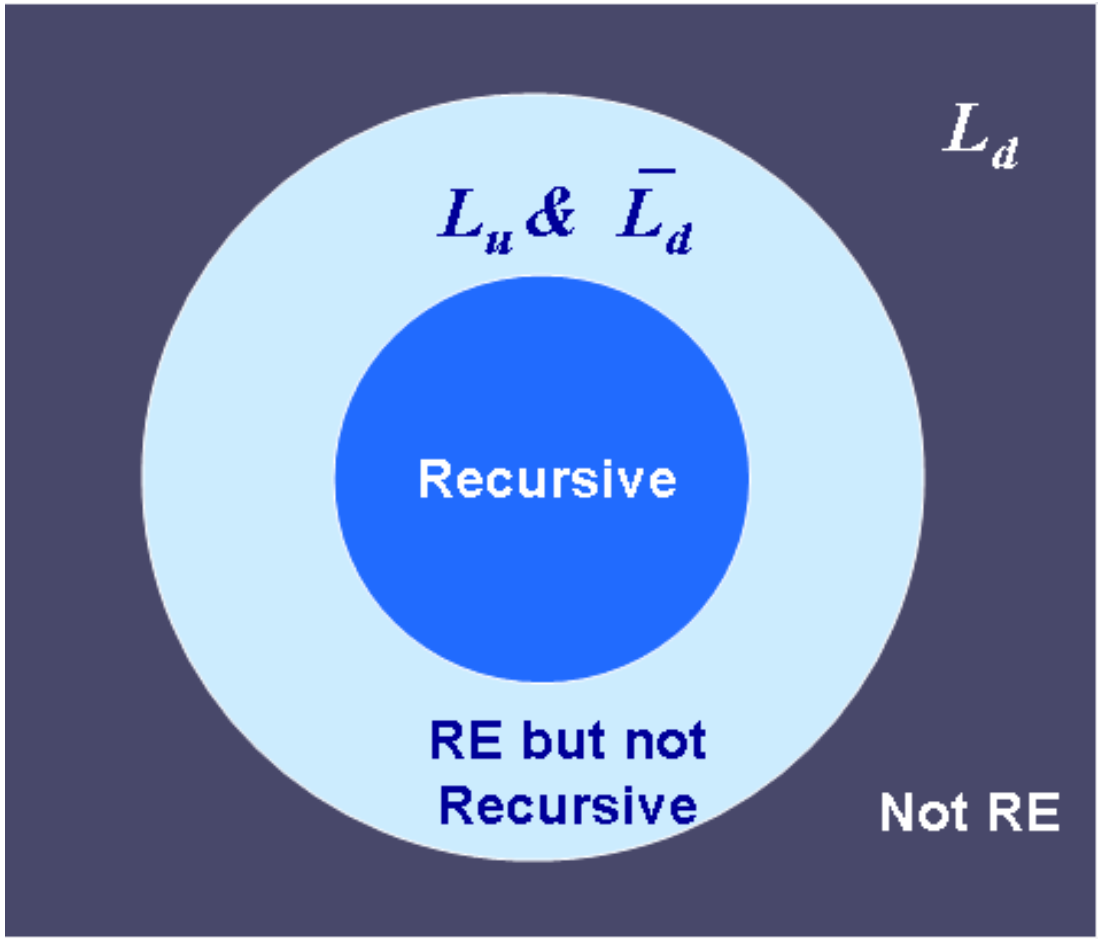


Figure : Relacionamento entre linguagens recursivas, RE e não-RE.

Problema de Correspondência de Post

- Vai-se agora discutir o **problema de correspondência de Post**, um resultado de indecidibilidade clássico sobre equações de cadeia, descoberto pelo matemático polonês Emil L. Post, com aplicações diretas à teoria da linguagem.
- **Definição**: Um **sistema de Post** P sobre um alfabeto finito Σ é um conjunto de pares ordenados (y_i, z_i), $1 \leq i \leq n$, onde y_i, z_i são cadeias em Σ^*. Um par (y_i, z_i) é algumas vezes chamado de uma equação de Post. O problema da correspondência de Post (PCP) é o problema de determinar, para um sistema de Post arbitrário P, se existem inteiros $i_1, ..., i_k$ tais que

$$y_{i_1} y_{i_2} ... y_{i_k} = z_{i_1} z_{i_2} ... z_{i_k}$$

Os i_j's não precisam ser distintos. Para um dado PCP, uma cadeia solução é uma cadeia de Post.

Problema de Correspondência de Post

- **Exemplo**: Considere o sistema de Post sobre $\Sigma = \{0, 1\}$:

  $$P = \{(1, 01), (\lambda, 11), (0101, 1), (111, \lambda)\}$$

  Este sistema tem uma solução, já que

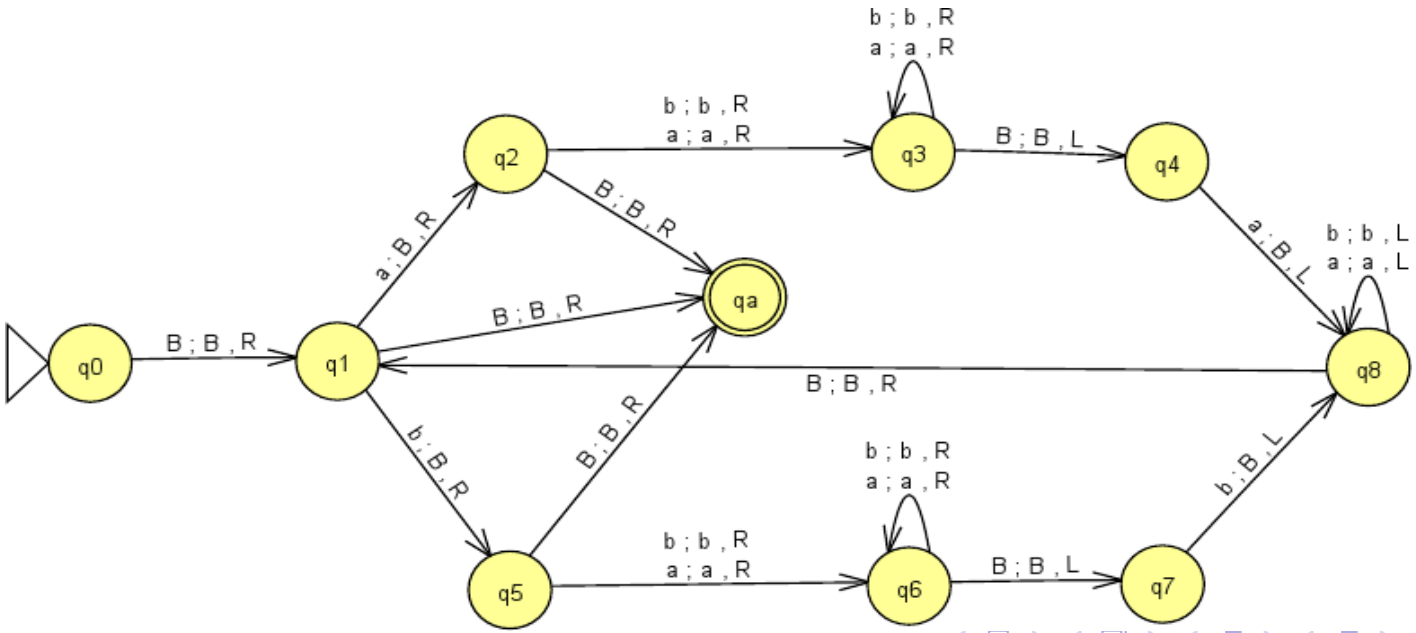
  $$y_2 y_1 y_1 y_3 y_2 y_4 = 110101111 = z_2 z_1 z_1 z_3 z_2 z_4.$$

- O seguinte teorema é um resultado fundamental.
- **Teorema**: O problema de correspondência de Post é insolúvel.
- Porque é orientado a equação, o PCP é particularmente bem adaptado para aplicações na teoria da linguagem.
- **Teorema**: O problema de decidir se uma gramática livre de contexto arbitrária é ambígua é indecidível.

Sumário

Complexidade de Tempo de uma Máquina de Turing

- A complexidade de tempo de uma computação mede a quantidade de trabalho gasto pela computação.
- O tempo de uma computação de uma máquina de Turing é quantificado pelo número de transições processadas.
- Seja uma máquina de Turing M que aceita palíndromos sobre o alfabeto $\Sigma = \{a, b\}$ [9].



Complexidade de Tempo de uma Máquina de Turing

$\lvert w\rvert = 0$	$\lvert w\rvert = 1$	$\lvert w\rvert = 2$		$\lvert w\rvert = 3$	
q_0BB	q_0BaB	q_0BaaB	q_0BabB	q_0BabaB	q_0BaabB
Bq_1B	Bq_1aB	Bq_1aaB	Bq_1abB	Bq_1abaB	Bq_1aabB
BBq_a	BBq_2B	BBq_2aB	BBq_2bB	BBq_2baB	BBq_2abB
	$BBBq_a$	$BBaq_3B$	$BBbq_3B$	$BBbq_3aB$	$BBaq_3bB$
		BBq_4aB	BBq_4bB	$BBbaq_3B$	$BBabq_3B$
		Bq_8BBB		$BBbq_4aB$	$BBaq_4bB$
		BBq_1BB		BBq_8bBB	
		$BBBq_aB$		Bq_8BbBB	
				BBq_1bBB	
				$BBBq_5BB$	
				$BBBBq_aB$	

Complexidade de Tempo de uma Máquina de Turing

- Como esperado, o número de transições em uma computação depende da cadeia de entrada.
- De fato, a quantidade de trabalho difere para cadeias de mesmo comprimento.
- Ao invés de tentar determinar o número exato de transições para cada cadeia de entrada, a complexidade de tempo de uma máquina de Turing é medida pelo trabalho necessário para cadeias de um comprimento fixo.
- **Definição**: Seja M uma máquina de Turing. A **complexidade de tempo** ("tempo de execução") de M é a função $ct_M : \mathbb{N} \rightarrow \mathbb{N}$ tal que $ct_M(n)$ é o número máximo de transições processadas por uma computação de M quando iniciada com uma cadeia de entrada de comprimento n, independente de M aceitar ou não.

Complexidade de Tempo de uma Máquina de Turing

- Quando se avalia a complexidade de tempo de uma máquina de Turing, assume-se que a computação termina para toda cadeia de entrada.
- Não faz sentido discutir a eficiência de uma computação que continua indefinidamente.
- A definição serve para máquinas que aceitam linguagens e computam funções.
- A complexidade de tempo de máquinas determinísticas multi-trilhas e multi-fitas é definida de maneira similar.

Complexidade de Tempo de uma Máquina de Turing

- A complexidade de tempo dá a performance de pior caso da máquina de Turing.
- Analisando um algoritmo, escolhe-se a performance do pior caso por duas razões:
  1. Considera-se as limitações da computação algorítmica. O valor $ct_M(n)$ especifica os recursos mínimos necessários para garantir que a computação de M termina quando iniciada com uma cadeia de entrada de comprimento n.
  2. A performance do pior caso é frequentemente mais fácil de avaliar do que a performance média.

Complexidade de Tempo de uma Máquina de Turing

- A máquina M que aceita os palíndromos sobre $\Sigma = \{a, b\}$ é usada para demonstrar o processo de determinar a complexidade de tempo de uma máquina de Turing.
- Uma computação de M termina quando toda a cadeia de entrada foi substituída por brancos ou o primeiro par de símbolos não previsto é descoberto.
- Como a complexidade de tempo mede a performance do pior caso, há necessidade de se preocupar apenas com as cadeias cujas computações fazem com que a máquina faça o maior número possível de ciclos acha-e-apaga.
- Esta condição é satisfeita quando a entrada é aceita por M.

Complexidade de Tempo de uma Máquina de Turing

- Usando estas observações, pode-se obter os valores iniciais da função ct_M da computações da tabela.

$$ct_M(0) = 2$$
$$ct_M(1) = 3$$
$$ct_M(2) = 7$$
$$ct_M(3) = 10$$

- Quando M processa uma cadeia de comprimento par, a computação alterna entre sequências de movimentos a direita e a esquerda da máquina.

Complexidade de Tempo de uma Máquina de Turing

- Inicialmente a cabeça é posicionada no símbolo mais a esquerda da porção não branca da fita:
  - *Movimentos à direita*: a máquina se move à direita, apagando o símbolo não-branco mais à esquerda. O resto da cadeia é lida e a máquina entra no estado q_4 ou q_7. Isto requer $k + 1$ transições, onde k é o comprimento da porção não-branca da cadeia.
  - *Movimentos à esquerda*: M se move para a esquerda, apagando o símbolo correspondente, e continua através da porção não-branca da fita. Isto requer k transições.
- As ações acima reduzem o comprimento da porção não-branca da fita por dois.
- O ciclo de comparações e apagamentos é repetido até que a fita esteja completamente em branco.

Complexidade de Tempo de uma Máquina de Turing

- Como observado, a performance do pior caso para uma cadeia de comprimento par ocorre quando M aceita a entrada.
- A computação que aceita uma cadeia de comprimento n requer $n/2$ iterações do *loop* anterior.

Iteração	**Direção**	**Transições**
1	direita	$n+1$
	esquerda	n
2	direita	$n-1$
	esquerda	$n-2$
3	direita	$n-3$
	esquerda	$n-4$
...	...	...
$n/2$	direita	1

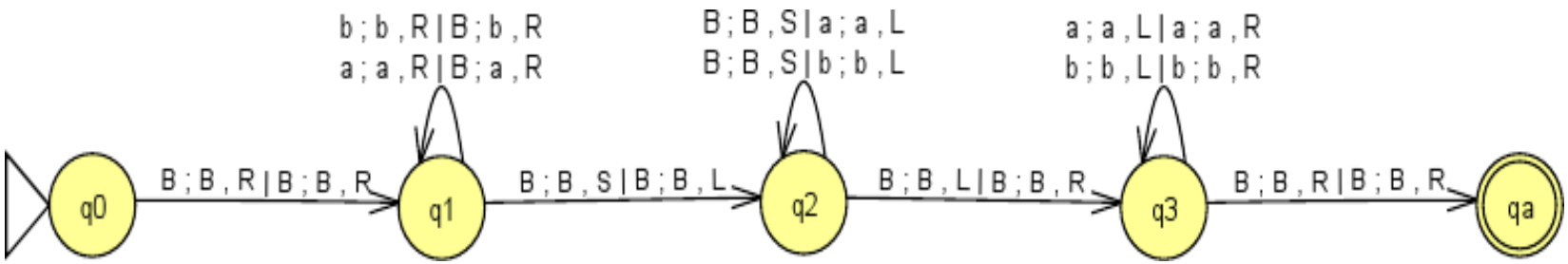
Complexidade de Tempo de uma Máquina de Turing

- O número máximo de transições em uma computação de uma cadeia de comprimento par n é a soma dos primeiros $n+1$ números naturais.
- Uma análise de cadeias de comprimento ímpar produz o mesmo resultado.
- Consequentemente, a complexidade de tempo de M é dada pela função:

$$ct_M(n) = \sum_{i=1}^{n+1} i = \frac{(n+2)(n+1)}{2}$$

Complexidade de Tempo de uma Máquina de Turing

- A máquina de duas fitas M'



também aceita o conjunto de palíndromos sobre $\Sigma = \{a, b\}$.

- Uma computação de M' percorre a entrada fazendo uma cópia na fita 2.
- A cabeça da fita 2 é depois movida à posição inicial.
- Então, as cabeças se movem ao longo da entrada, fita 1 para a esquerda e fita 2 para a direita, comparando os símbolos das fitas 1 e 2.

Complexidade de Tempo de uma Máquina de Turing

- Se as cabeças encontrarem símbolos diferentes, a entrada **não** é um palíndromo e a computação para num estado de não-aceitação.
- Quando a entrada é um palíndromo, a computação para e aceita quando brancos são simultaneamente lidos nas fitas 1 e 2.
- Para uma entrada de comprimento n, o número máximo de transições ocorre quando a cadeia é um palíndromo.
- Uma aceitação requer três passos completos:
  1. a cópia,
  2. a volta e
  3. a comparação.
- Contando o número de transições em cada passo, vê-se que a complexidade de tempo de M' é $ct_{M'}(n) = 3(n+1)+1$.

Complexidade de Tempo de uma Máquina de Turing

- A definição da complexidade de tempo ct_M é baseada nas computações da máquina M e não na linguagem aceita pela máquina.
- Sabe-se que muitas máquinas diferentes podem ser construídas para aceitar a mesma linguagem, cada qual com diferentes complexidades de tempo.
- Diz-se que uma linguagem L é aceita em tempo determinístico $f(n)$ se houver uma máquina de Turing determinística M qualquer com $ct_M(n) \leq f(n)$ para todo $n \in \mathbb{N}$.
- A máquina M mostrou que o conjunto de palíndromos sobre $\Sigma = \{a, b\}$ é aceito em tempo $\frac{(n^2+3n+2)}{2}$ enquanto que a máquina de duas fitas M' exibiu aceitação no tempo $3n + 4$.

Complexidade de Tempo de uma Máquina de Turing

- Uma transição de uma máquina de duas fitas utiliza mais informação e realiza uma operação mais complicada do que a máquina de uma fita.
- A estrutura adicional da transição de duas fitas permitiu a redução no número de transições de M' necessárias para processar uma cadeia em relação à máquina de uma fita M.
- Potanto, vê-se um equilíbrio entre a complexidade das transições e o número que deve ser processado.
- Há formas de "acelerar" uma máquina que aceita uma linguagem L para produzir uma nova máquina que aceita L num tempo menor.

Taxas de Crescimento

- Como sempre é possível "melhorar" uma máquina para que aceite a linguagem num tempo menor, é interessante representar a complexidade de tempo por uma **taxa de crescimento** ao invés de uma função.

Table : Crescimento de funções [9]

n	0	5	10	25	50	100	1.000
$20n + 500$	500	600	700	1.000	1.500	2.500	20.500
n^2	0	25	100	625	2.500	10.000	1.000.000
$n^2 + 2n + 5$	5	40	125	680	2.605	10.205	1.002.005
$\frac{n^2}{(n^2+2n+5)}$	0	0,625	0,800	0,919	0,960	0,980	0,998

Taxas de Crescimento

- A taxa de crescimento de uma função mede o aumento dos valores da função quando a entrada se torna arbitrariamente grande.
- Intuitivamente, a taxa de crescimento é determinada pelo contribuinte mais significativo ao crescimento da função.
- A contribuição de termos individuais aos valores de uma função pode ser vista examinando o crescimento das funções n^2 e $n^2 + 2n + 5$ na tabela.
- A contribuição de n^2 à $n^2 + 2n + 5$ é medida pela razão dos valores da função na última linha.
- Os termos linear e constante da função $n^2 + 2n + 5$ são chamados de *termos de ordem mais baixa*.

Considerações sobre a Eficiência

- Estes termos influenciam os valores iniciais da função, mas à medida em que n aumenta, estes termos não contribuem significativamente para o crescimento dos valores da função.
- Para n pequeno (até 25), o termo $20n + 500$ produz um resultado maior que n^2, pois a diferença entre n^2 e n é pequena. A medida em que n cresce, o termo n^2 domina o termo $20n$. Dessa forma, para n grande, o tempo necessário para a aceitação é quase proporcional à n^2.
- Para essa tendência de uma função tornar-se proporcional a outra à medida em que aumenta, é dito ser “da ordem” da função.

Considerações sobre a Eficiência: Notação $\mathcal{O}$

- **Definição**: Dadas duas funções, $f(n)$ e $g(n)$, diz-se que $f(n)$ é **da ordem de** $g(n)$ ou que $f(n)$ é $\mathcal{O}(g(n))$, se existirem inteiros positivos a e b tais que $f(n) \leq a * g(n)$ para todo $n \geq b$.
- Por exemplo, se $f(n) = n^2 + 100n$ e $g(n) = n^2$, $f(n)$ é $\mathcal{O}(g(n))$, uma vez que $n^2 + 100n$ é menor ou igual a $2n^2$ para todo $n \geq 100$. Nesse caso, a é igual a 2 e b é igual a 100.
- Essa mesma $f(n)$ é também $\mathcal{O}(n^3)$, já que $n^2 + 100n$ é menor ou igual a $2n^3$ para todo n maior ou igual a 8.
- Dada uma função $f(n)$, podem existir várias funções $g(n)$ tais que $f(n)$ seja $\mathcal{O}(g(n))$.

Considerações sobre a Eficiência: Notação $\mathcal{O}$

Table : $f(n) = \mathcal{O}(g(n))$ para $a = 2$ e $b = 100$, pois $f(n) \leq a * g(n)$ para todo $n \geq b$

n	$g(n) = n^2$	$100n$	$f(n) = n^2 + 100n$	$2n^2$
10	100	1000	**1100**	200
50	2500	5000	**7500**	5000
100	10000	10000	**20000**	**20000**
101	10201	10100	20301	**20402**
150	22500	15000	37500	**45000**
200	40000	20000	60000	**80000**
250	62500	25000	87500	**125000**
300	90000	30000	120000	**180000**

Considerações sobre a Eficiência: Notação $\mathcal{O}$

Table : $f(n) = \mathcal{O}(g(n))$ para $a = 2$ e $b = 8$, pois $f(n) \leq a * g(n)$ para todo $n \geq b$

n	n^2	$100n$	$f(n) = n^2 + 100n$	$g(n) = n^3$	$2n^3$
1	1	100	**101**	1	2
2	4	200	**204**	8	16
3	9	300	**309**	27	54
4	16	400	**416**	64	128
5	25	500	**525**	125	250
6	36	600	**636**	216	432
7	49	700	**749**	343	686
8	64	800	864	512	**1024**
9	81	900	981	729	**1458**
10	100	1000	1100	1000	**2000**

Considerações sobre a Eficiência: Notação $\mathcal{O}$

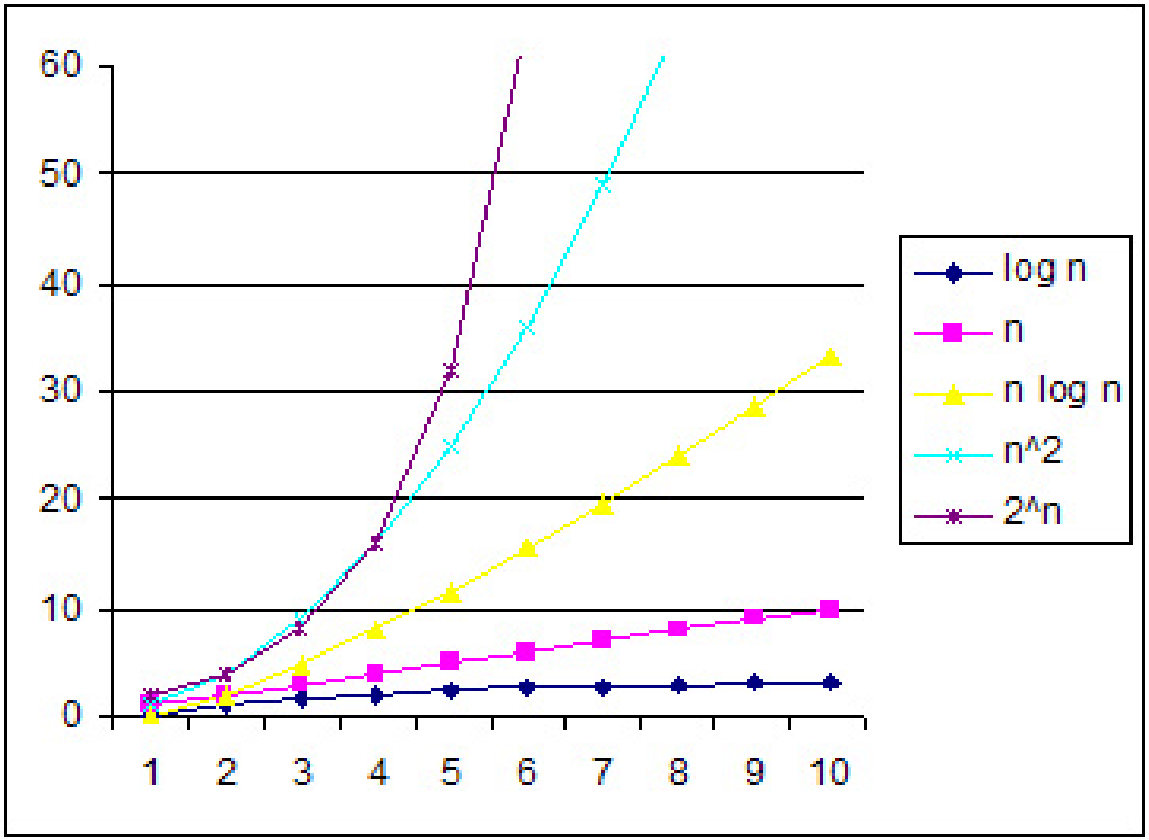


Figure : Crescimentos de algumas funções.

Polinômio com coeficientes integrais

- **Definição**: Um **polinômio com coeficientes integrais** é uma função da forma

$$f(n) = c_r \cdot n^r + c_{r-1} \cdot n^{r-1} + ... + c_1 \cdot n + c_0$$

onde $c_0, c_1, ..., c_{r-1}$ são inteiros quaisquer, c_r é um inteiro não nulo, e r é um inteiro positivo. As constantes c_i são os **coeficientes** de f e r é o **grau** do polinômio. Um polinômio com coeficientes integrais define uma função de números naturais para inteiros.

- A presença de coeficientes negativos pode produzir valores negativos. Por exemplo, se $f(n) = n^2 - 3n - 4$, então $f(0) = -4$, $f(1) = -6$, $f(2) = -6$ e $f(3) = -4$. Os valores do polinômio $g(n) = -n^2 - 1$ são negativos para todo número natural n.

Polinômio com coeficientes integrais

- **Teorema**: Seja f um polinômio de grau r. Então
  1. $f = \mathcal{O}(n^r)$
  2. $n^r = \mathcal{O}(f)$
  3. $f = \mathcal{O}(n^k)$, para todo $k > r$
  4. $f \neq \mathcal{O}(n^k)$, para todo $k < r$.
- Uma das consequências deste Teorema é que a taxa de crescimento de qualquer polinômio pode ser caracterizada por uma função da forma n^r.
- Condições (1) e (2) mostram que um polinômio de grau r tem a mesma taxa de crescimento que n^r.
- No entanto, pelas condições (3) e (4), seu crescimento não é equivalente ao de n^k, para qualquer k diferente de r.

Funções Polinomialmente Limitadas

- **Teorema**: Seja r um número natural e sejam a e b números reais maiores que 1. Então
  1. $log_a(n) = \mathcal{O}(n)$
  2. $n \neq \mathcal{O}(log_a(n))$
  3. $n^r = \mathcal{O}(b^n)$
  4. $b^n \neq \mathcal{O}(n^r)$
  5. $b^n = \mathcal{O}(n!)$
  6. $n! \neq \mathcal{O}(b^n)$.
- **Definição**: Uma função f é **polinomialmente limitada** se $f = \mathcal{O}(n^r)$ para algum número natural r.
- Ainda que não seja um polinômio, segue de (1) que $n\ log_2(n)$ é limitado pelo polinômio n^2.

Funções Polinomialmente Limitadas

- As funções polinomialmente limitadas, que incluem os polinômios, constituem uma importante família de funções que serão associadas com a complexidade de tempo de algoritmos solúveis eficientemente.
- As condições (4) e (6) mostram que as funções exponencial e fatorial não são polinomialmente limitadas.
- Hierarquia de $\mathcal{O}$:
  - $\mathcal{O}(1)$: ordem constante
  - $\mathcal{O}(log_a\ n)$: ordem logarítmica
  - $\mathcal{O}(n)$: ordem linear
  - $\mathcal{O}(n\ log_a\ n)$: $n\ log\ n$
  - $\mathcal{O}(n^2)$: ordem quadrática
  - $\mathcal{O}(n^3)$: ordem cúbica
  - $\mathcal{O}(n^r)$: ordem polinomial $r \geq 0$
  - $\mathcal{O}(b^n)$: ordem exponencial $b > 1$
  - $\mathcal{O}(n!)$: ordem fatorial.

Algoritmos Polinomiais e Não Polinomiais

- **Algoritmos exponenciais** são, em geral, simples: variações de pesquisa exaustiva no espaço de soluções (força bruta).
- **Algoritmos polinomiais** são obtidos através de um entendimento mais profundo da estrutura do problema. Veja como exemplo a descoberta em agosto de 2002 de um algoritmo polinomial para o "Problema de Verificar se um número é Primo."
- Um problema é considerado **intratável** se não existe um algoritmo polinomial para resolvê-lo.
- Um problema é considerado bem resolvido/**tratável** se existe um algoritmo polinomial para o problema. Tais problemas são considerados eficientes.

Algoritmos Polinomiais e Não Polinomiais

- A eficiência de um algoritmo é caracterizada por sua taxa de crescimento.
- Um algoritmo em tempo polinomial, ou simplesmente um algoritmo polinomial, tem sua complexidade de tempo polinomialmente limitada.
- Portanto, $ct_M = \mathcal{O}(n^r)$ para algum $r \in \mathbb{N}$, onde M é uma máquina de Turing padrão que computa o algoritmo.
- A distinção entre algoritmos polinomiais e não polinomiais é aparente quando se considera o número de transições de uma computação quando o comprimento da entrada aumenta.
- A tabela ilustra os enormes recursos necessários para um algoritmo cujo complexidade de tempo não é polinomial.

Algoritmos Polinomiais e Não Polinomiais

Table : Número de transições de máquina com complexidade de tempo ct_M com entrada de comprimento n. [9]

n	ct_M					
	$log_2(n)$	n	n^2	n^3	2^n	$n!$
5	2	5	25	125	32	120
10	3	10	100	1.000	1.024	3.628.800
20	4	20	400	8.000	1.048.576	$2{,}4 \cdot 10^{18}$
30	4	30	900	27.000	$1{,}0 \cdot 10^{9}$	$2{,}6 \cdot 10^{32}$
40	5	40	1.600	64.000	$1{,}1 \cdot 10^{12}$	$8{,}1 \cdot 10^{47}$
50	5	50	2.500	125.000	$1{,}1 \cdot 10^{15}$	$3{,}0 \cdot 10^{64}$
100	6	100	10.000	1.000.000	$1{,}2 \cdot 10^{30}$	$> 10^{157}$
200	7	200	40.000	8.000.000	$1{,}6 \cdot 10^{60}$	$> 10^{374}$

Problema de Decisão

- **Definição**: Um **problema de decisão** P é um conjunto de questões, cada uma com uma resposta *sim* ou *não*.
- Uma solução para um problema de decisão P é um algoritmo que determina a resposta apropriada a toda questão $p \in P$.
- Um algoritmo que resolve um problema de decisão deve ser:
  - *Completo*: produz uma resposta, positiva ou negativa, a cada questão no domínio do problema.
  - *Mecânico*: consiste de uma sequência finita de instruções, executada sem requerer *insight*, imaginação ou advinhação.
  - *Determinístico*: entradas idênticas produzem o mesmo resultado.
- Um procedimento que satisfaz as propriedades anteriores é chamado de *efetivo*.

Problema de Decisão

- As computações de uma máquina de Turing padrão são claramente mecânicas e determinísticas.
- Uma máquina de Turing que para para toda cadeia de entrada também é completa.
- Por causa da efetividade intuitiva de suas computações, máquinas de Turing provêem um arcabouço formal que pode ser usado para construir soluções para os problemas de decisão.
- Um problema é respondido afirmativamente se a entrada é aceita por uma máquina de Turing e negativamente se é rejeitada.

Máquinas de Turing e Não-Determinismo

- **Teorema**: Seja L uma linguagem aceita por uma máquina de Turing determinística de k fitas M com complexidade de tempo $ct_M(n) = f(n)$. Então L é aceita por uma máquina de Turing padrão N com complexidade de tempo $ct_N(n) = \mathcal{O}(f(n)^2)$.
- As computações não-determinísticas são muito diferentes das determinísticas.
- Uma máquina determinística resolve um problema de decisão gerando uma solução.
- Uma máquina não-determinística precisa determinar se uma das possibilidades é uma solução.

Máquinas de Turing e Não-Determinismo

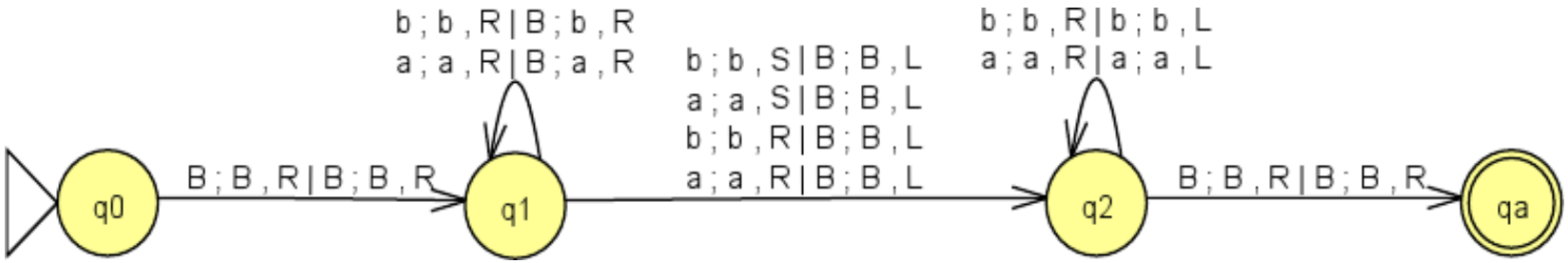
- Uma cadeia é aceita por uma máquina não-determinística se pelo menos uma computação termina em um estado de aceitação.
- A aceitação de uma cadeia não é afetada pela existência de outras computações que param em estados de não aceitação ou que não param em estado nenhum.
- A performance de pior caso do algoritmo, entretanto, mede a eficiência sobre todas as computações.
- **Definição**: Seja M uma máquina de Turing não-determinística. A complexidade de tempo de M é a função $ct_M : \mathbb{N} \rightarrow \mathbb{N}$ tal que $ct_M(n)$ é o número máximo de transições processadas por uma computação, empregando qualquer escolha de transições, de uma cadeia de entrada de comprimento n.

Máquinas de Turing e Não-Determinismo

- A definição é idêntica àquela da complexidade de tempo de uma máquina determinística.
- A ênfase está em que a análise não-determinística deve considerar todas as computações possíveis para uma cadeia de entrada.
- Como antes, assume-se que toda computação de M termina.
- Computações não-determinísticas que usam a estratégia "adivinha-e-checa" são geralmente mais simples que as determinísticas.
- A simplicidade reduz o número de transições necessárias para uma única computação.
- Empregando esta estratégia, pode-se construir uma máquina não-determinística para aceitar os palíndromos sobre $\Sigma = \{a, b\}$.

Máquinas de Turing e Não-Determinismo

- A máquina não-determinística de duas fitas M



  aceita os palíndromos sobre $\Sigma = \{a, b\}$ no tempo $ct_M(n) = n + 2$.
- Ambas as cabeças se movem para a direita com a entrada sendo copiada na fita 2.
- A transição a partir de q_1 "adivinha" o centro da cadeia.

Máquinas de Turing e Não-Determinismo

- Uma transição de q_1 que move a cabeça da fita 1 para a direita a da fita 2 para a esquerda procura por um palíndromo de comprimento ímpar, enquanto que uma transição que deixa a cabeça na fita 2 na mesma posição checa um palíndromo de comprimento par.
- O número máximo de transições para uma computação com cadeia de entrada de comprimento n é $n + 2$, o número de transições necessárias para ler a cadeia de entrada inteira na fita 1.
- As máquinas dos palíndromos ilustram a redução da complexidade de tempo que pode ser obtida usando não-determinismo.

Sumário

Complexidade de Espaço

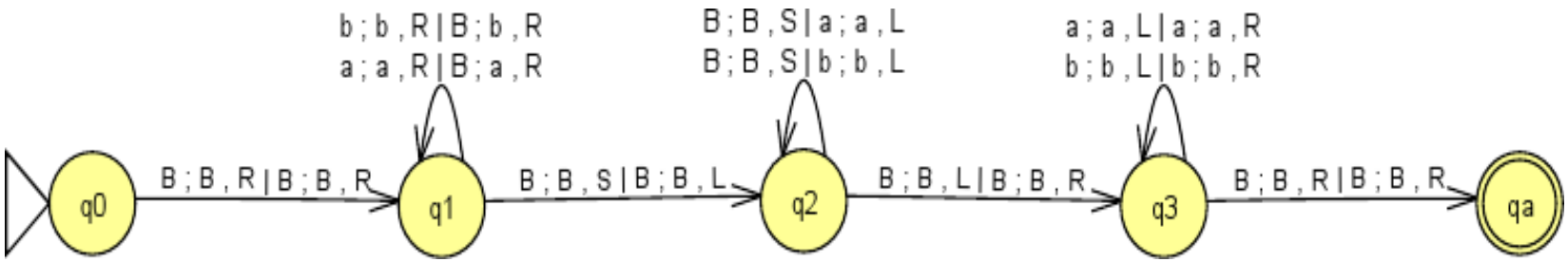
- Suponha uma arquitetura de máquina de Turing com k fitas. A fita 1, de apenas leitura, é usada como fita de entrada. As demais $k - 1$ fitas são fitas de trabalho.
- Com uma cadeia de entrada de comprimento n, a cabeça na fita de entrada deve permanecer dentro das posições 0 até $n + 1$.
- A máquina de Turing lê a fita de entrada mas realiza seu trabalho nas fitas restantes e o espaço ocupado pela entrada não é incluído na análise da complexidade.
- A existência de uma fita de entrada de apenas leitura separa a quantidade de espaço necessária para a entrada do espaço de trabalho necessário para a computação.
- A complexidade de espaço mede a quantidade de espaço nas fitas de trabalho usadas por uma computação.

Complexidade de Espaço

- **Definição**: Seja M uma máquina de Turing de k fitas projetada para análise de complexidade de espaço. A **complexidade de espaço** de M é a função $ce_M : \mathbb{N} \rightarrow \mathbb{N}$ tal que $ce_M(n)$ é o número máximo de quadrados de fita lidos nas fitas 2, ... k por uma computação de M quando iniciada com uma cadeia de entrada de comprimento n.
- Como a complexidade de espaço mede apenas as fitas de trabalho, é possível que $ce_M(n) < n$.
- Isto é, o espaço necessário para a computação pode ser menor que o comprimento da entrada.
- Assim como com a complexidade de tempo, assume-se que as computações de uma máquina de Turing terminam para toda cadeia de entrada.

Complexidade de Espaço

- **Exemplo**: Seja, de novo, a máquina determinística de duas fitas M', que aceita os palíndromos sobre $\Sigma = \{a, b\}$:



- Esta máquina obedece às especificações de uma máquina projetada para a análise de complexidade de espaço: a fita de entrada é de leitura apenas e a cabeça da fita lê apenas a cadeia de entrada e os brancos em ambas as extremidades.

Complexidade de Espaço

- A complexidade de espaço de M' é $n + 2$; uma computação reproduz a entrada na fita 2 e compara as cadeias nas fitas 1 e 2 lendo as cadeias em direções opostas.
- Projeta-se agora uma máquina de três cabeças M que aceita os palíndromos com $ce_M(n) = \mathcal{O}(log(n))$.
- As fitas de trabalho, que armazenam a representação binária de inteiros, são usadas como contadores.
- Uma computação de M com entrada u de comprimento n consiste dos seguintes passos (seja i o inteiro cuja representação binária está nas fitas 2 e 3):

Complexidade de Espaço

- Computação de M:
  1. Um único *1* é escrito na fita 3.
  2. A fita 3 é copiada na fita 2.
  3. A cabeça da fita de entrada é posicionada no quadrado mais a esquerda.
  4. Enquanto o número na fita 2 não é 0,
     1. Mova a cabeça da fita de entrada um quadrado para a direita.
     2. Decremente o valor na fita 2.
  5. Se o símbolo lido na fita de entrada é um branco, pare e aceite.
  6. O i-ésimo símbolo da entrada é gravado usando estados da máquina.
  7. A cabeça da fita de entrada é movida para a extrema direita da entrada (posição da fita $n+1$).
  8. A fita 3 é copiada na fita 2.
  9. Enquanto o número na fita 2 não é 0,
     1. Mova a cabeça da fita de entrada um quadrado para a esquerda.
     2. Decremente o valor na fita 2.
  10. Se o $(n-i+1)$-ésimo símbolo combina com o i-ésimo símbolo, então incremente o valor na fita 3 e vá para o passo 2. Caso contrário pare e rejeite.

Complexidade de Espaço

- As operações nas fitas 2 e 3 incrementam e decrementam a representação binária de inteiros.
- Como $n+1$ é o maior número escrito em qualquer destas fitas, cada fita usa no máximo $log(n+1)+2$ quadrados de fita.

- **Teorema**: Seja M uma máquina de Turing de k fitas com complexidade de tempo $ct_M(n) = f(n)$. Então $ce_M(n) \leq (k-1) \cdot f(n)$.
- **PROVA**: A quantidade máxima de fita é lida quando cada transição de M move as cabeças das fitas 2, ..., k à direita. Neste caso, cada cabeça em uma fita de trabalho lê no máximo $f(n)$ quadrados diferentes. ♦

Sumário

Problemas Tratáveis

- A complexidade computacional provê uma medida dos recursos exigidos por um algoritmo.
- A complexidade de um problema de decisão é determinada pela complexidade do algoritmo que resolve o problema.
- A análise da complexidade provê critérios para classificar os problemas baseado na praticabilidade de suas soluções.
- Um problema que é teoricamente solúvel pode não ter uma solução prática; pode não haver algoritmo que resolva o problema sem precisar de uma quantidade extraordinária de tempo e memória.
- Há os problemas tratáveis e intratáveis.

Problemas Tratáveis

- Problemas são considerados **tratáveis** se houver uma máquina de Turing que resolva o problema cujas computações são polinomialmente limitadas.
- Há vários problemas famosos para os quais não se sabe se há soluções determinísticas em tempo polinomial.
- Entre esses, há o problema do ciclo hamiltoniano[2], que tem uma solução em tempo polinomial não determinística.
- A questão de saber se todo problema que pode ser resolvido em tempo polinomial por um algoritmo não-determinístico pode também ser resolvido deterministicamente em tempo polinomial é **a** questão da ciência de computação teórica.

[2] Um ciclo é um caminho onde o último vértice é adjacente ao primeiro. Um ciclo no qual nenhum vértice é repetido é chamado de ciclo simples. Dado um grafo não dirigido G, um *ciclo hamiltoniano* é um ciclo simples que contém todos os vértices de G.

Problemas de Decisão Tratáveis e Intratáveis

- Uma linguagem L sobre Σ é *decidível em tempo polinomial*, ou simplesmente *polinomial*, se há um algoritmo que determina a pertinência em L para o qual o tempo requerido por uma computação cresce polinomialmente com o comprimento da cadeia de entrada.
- Como pode haver muitos algoritmos que decidem a pertinência em uma linguagem L, a complexidade de tempo de L é caracterizada pela taxa de crescimento da solução mais eficiente.
- A noção de decidibilidade em tempo polinomial é formalmente definida usando transições de uma máquina de Turing padrão como o arcabouço computacional para medir o tempo de uma computação.

Problemas de Decisão Tratáveis e Intratáveis

- A linguagem dos palíndromos sobre $\Sigma = \{a, b\}$ é polinomial pois a máquina M aceita a linguagem em tempo $\mathcal{O}(n^2)$.
- **Definição**: Uma linguagem L é **decidível em tempo polinomial** se houver uma máquina de Turing padrão M que aceita L com $ct_M = \mathcal{O}(n^r)$, onde r é um número natural independente de n. A família de linguagens decidíveis em tempo polinomial é denotada $\mathcal{P}$.
- A teoria da computabilidade está preocupada em estabelecer se os problemas de decisão são teoricamente decidíveis.
- A teoria da complexidade tenta distinguir problemas que são solúveis na prática daqueles que são solúveis apenas em princípio.

A classe $\mathcal{P}$ [1]

- $\mathcal{P}$ é a classe de problemas que podem ser resolvidos por algoritmos determinísticos em tempo polinomial.
- Simplificação: "estar em $\mathcal{P}$" significa "fácil" e "não estar em $\mathcal{P}$" significa "difícil."
- Na teoria esta suposição é válida, mas na prática nem sempre é verdadeira por várias razões:

A classe $\mathcal{P}$

1. A teoria ignora fatores constantes. Um problema que tem tempo $10^{1000}n$ está em $\mathcal{P}$ (tem tempo linear), mas é intratável na prática. Um problema que tem tempo $10^{-10000}2^n$ não está em $\mathcal{P}$ (tem tempo exponencial), mas é tratável para valores de n acima de 1000.
2. A teoria ignora o tamanho dos expoentes. Um problema que tem tempo n^{1000} está em $\mathcal{P}$, mas é intratável. Um problema com tempo $2^{\frac{n}{1000}}$ não está em $\mathcal{P}$, mas é tratável com n acima de 1000. Embora, em geral, graus elevados e coeficientes muito grandes não ocorram na prática.
3. A teoria só considera a análise de pior tempo.
4. A teoria não considera soluções probabilísticas mesmo aquelas que admitem uma pequena probabilidade de erro. Um algoritmo 2^n é muito mais rápido que um algoritmo n^5 para $n \leq 22$. ($n = 22 \rightarrow 4.194.304 \times 5.153.632$).

Problemas de Decisão Tratáveis e Intratáveis

- Uma solução para um problema de decisão pode ser impraticável por causa dos recursos necessários às computações.
- Problemas para os quais não há algoritmo eficiente são chamados de **intratáveis**.
- Por causa da taxa de crescimento da complexidade de tempo, algoritmos não-polinomiais são considerados inviáveis para quase todos os casos do problema, exceto os mais simples.
- A divisão da classe de problemas de decisão solúveis em problemas polinomiais e não-polinomiais é usada para distinguir os problemas solúveis eficientemente dos problemas intratáveis.

Problemas de Decisão Tratáveis e Intratáveis

- A classe $\mathcal{P}$ foi definida em termos da complexidade de tempo de uma implementação de um algoritmo em uma máquina de Turing padrão.
- A máquina de Turing pode ser também multi-trilhas ou multi-fitas que a classe $\mathcal{P}$ de problemas de decisão polinomial é invariante.
- Pode-se mostrar que uma linguagem aceita por uma máquina multi-trilhas no tempo $\mathcal{O}(n^r)$ é também aceita para uma máquina de Turing padrão no mesmo tempo.
- A transição da máquina multi-fitas para padrão também preserva as soluções polinomiais.
- Uma linguagem aceita em tempo $\mathcal{O}(n^r)$ por uma máquina multi-fitas é aceita no tempo $\mathcal{O}(n^{2r})$ por uma máquina padrão.

Problemas de Tempo Polinomial

- Serão abordados conceitos da teoria da intratabilidade:
  - As classes $\mathcal{P}$ e $\mathcal{NP}$ de problemas que podem ser resolvidos em tempo polinomial por máquinas de Turing determinísticas e não-determinísticas, respectivamente,
  - A técnica de redução polinomial de tempo,
  - A noção $\mathcal{NP}$-completo.
- A maioria dos problemas com soluções eficientes está em $\mathcal{P}$.
- Um problema de estruturas de dados e algoritmos: encontrar uma árvore geradora mínima (MWST = "minimum-weight spanning tree") para um grafo [5].

O algoritmo de Kruskal [10]

- O algoritmo de Kruskal é um algoritmo em teoria dos grafos que busca uma árvore geradora mínima para um grafo conexo com pesos.
- Isto significa que ele encontra um subconjunto das arestas que forma uma árvore que inclui todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore, é minimizado.
- Se o grafo não for conexo, então ele encontra uma floresta geradora mínima (uma árvore geradora mínima para cada componente conexo do grafo).
- O algoritmo de Kruskal é um exemplo de um algoritmo **guloso** (também conhecido como ganancioso ou “greedy”).

O algoritmo de Kruskal

- Seu funcionamento é mostrado a seguir:
  1. crie uma floresta F (um conjunto de árvores), onde cada vértice no grafo é uma árvore separada,
  2. crie um conjunto S contendo todas as arestas do grafo,
  3. enquanto S for não-vazio, faça:
     1. remova uma aresta com peso mínimo de S,
     2. se essa aresta conecta duas árvores diferentes, adicione-a à floresta, combinando duas árvores numa única árvore parcial,
     3. caso contrário, descarte a aresta.
- Ao fim do algoritmo, a floresta tem apenas um componente e forma uma árvore geradora mínima do grafo.

O algoritmo de Kruskal

- Para um grafo com n vértices e m arestas, o algoritmo de Kruskal leva $\mathcal{O}(m\ log\ m)$ passos para ordenar as arestas pelo custo, $\mathcal{O}(m)$ passos para decidir se a aresta deve ser adicionada ($\mathcal{O}(1)$ para cada aresta) e tempo $\mathcal{O}(n^2)$ para manter uma tabela com as conexões vértice a vértice. Tempo total $\mathcal{O}(n^2 + m\ log\ m)$, que para grafos esparsos é $\mathcal{O}(n^2)$.
- Esse tempo de execução é polinomial no "tamanho" da entrada, que corresponde à soma de n e m.

O MWST na máquina de Turing

- Quando se estuda algoritmos, encontra-se “problemas” com saídas de diversas formas (como a lista de arestas em uma MWST).
- Quando se lida com máquinas de Turing, pensa-se em problemas de linguagens e a única saída é *sim* (aceita) ou *não* (rejeita).
- Por exemplo, o problema da MWST pode ser: “dado o grafo G e o limite W, G tem uma árvore geradora de peso W ou menor?”
- Desta forma, o problema pode parecer mais fácil.
- Mas na teoria da intratabilidade, quer-se demonstrar que um problema é difícil e o fato de uma versão sim/não ser difícil implica que uma versão com uma resposta mais completa também é difícil.

O MWST na máquina de Turing

- Embora seja possível pensar no "tamanho" de um grafo como $n + m$, a entrada para uma máquina de Turing é uma cadeia sobre um alfabeto finito (Σ).
- Ou seja, deve-se codificar vértices e arestas de maneira apropriada.
- Portanto, as entradas para máquinas de Turing são "mais longas" que o tamanho intuitivo da entrada.
- Mas essa diferença não é significativa:
  1. A diferença entre o tamanho, como uma cadeia de entrada de uma máquina de Turing e como uma entrada informal de um problema, nunca é maior que um pequeno fator, em geral o logaritmo do tamanho da entrada. Desse modo, o que pode ser feito em tempo polinomial não é alterado.
  2. O comprimento de uma cadeia é uma medida mais precisa do número de bytes que um computador real tem de ler para obter sua entrada.

O MWST na máquina de Turing

- Seja, a seguir, um código possível para os grafos e limites de peso como entrada para o problema da MWST:
  1. Atribua inteiros de 1 a n aos vértices.
  2. Inicie o código com o valor de n em binário e o limite de peso W em binário, separados por uma vírgula.
  3. Se houver uma aresta entre os vértices i e j com peso w, insira (i, j, w) no código. i, j e w estão em binário.
- Desta forma, um dos códigos possíveis para o grafo da figura, com o limite $W = 20$ é

111,10100(1,10,111)(1,100,101)(10,11,1000)(10,100,1001)(10,101,111)(11,101,101)
(100,101,1111)(100,110,110)(101,110,1000)(101,111,1001)(110,111,1011)

- Em $\mathcal{O}(n^2)$ etapas pode-se implementar a versão do algoritmo de Kruskal em uma máquina de Turing de várias fitas.

O MWST na máquina de Turing

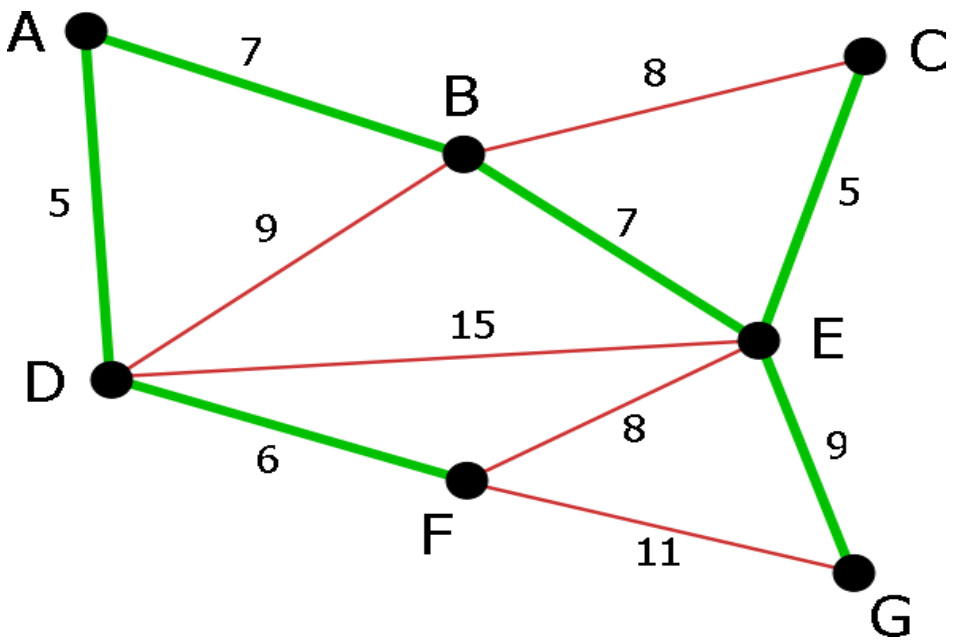


Figure : Ordem de seleção das arestas: AD, CE, DF, AB, BE, EG.

O MWST na máquina de Turing

- As fitas extras são usadas para:
  1. Uma fita é usada para armazenar os vértices e seus números de componentes atuais. Comprimento: $\mathcal{O}(p)$, onde p é o comprimento da entrada.
  2. Uma fita é usada para guardar a aresta de menor peso, ainda não marcada, à medida em que se varre as arestas na fita de entrada. Tempo: $\mathcal{O}(p)$.
  3. Quando uma aresta for selecionada, coloque seus dois vértices em uma fita. Busque na tabela de vértices e componentes os componentes desses dois vértices. Tempo: $\mathcal{O}(p)$.
  4. Uma fita pode ser usada para guardar os dois componentes i e j, intercalados quando se encontra uma aresta para conectar dois vértices desconexos. Depois varre-se a tabela de vértices e componentes, e cada vértice encontrado no componente i tem seu número de componente mudado para j. Tempo: $\mathcal{O}(p)$.

O MWST na máquina de Turing

- Conclusão: uma rodada pode ser executada no tempo $\mathcal{O}(p)$ em uma máquina de Turing de várias fitas.
- Como o número de rodadas é no máximo p, conclui-se que o tempo $\mathcal{O}(p^2)$ é suficiente em uma máquina de Turing de várias fitas.
- Há um teorema que diz que qualquer ação que uma máquina de Turing de várias fitas pode realizar em s etapas, uma máquina de Turing de fita única pode realizar em $\mathcal{O}(s^2)$ etapas.
- Assim, se uma máquina de Turing de várias fitas demora $\mathcal{O}(p^2)$ etapas, pode-se construir uma máquina de Turing de uma única fita para fazer o mesmo em $\mathcal{O}((p^2)^2) = \mathcal{O}(p^4)$ etapas.
- Ou seja, a versão "sim/não" do problema da MWST está em $\mathcal{P}$.

Sumário

A Classe $\mathcal{NP}$ [1]

- $\mathcal{NP}$ é a classe de problemas que podem ser resolvidos por algoritmos não-determinísticos em tempo polinomial.
- Ou, alternativamente, $\mathcal{NP}$ é a classe de problemas de decisão para os quais uma dada solução pode ser verificada em tempo polinomial.
- Assim, para mostrar que um problema está em $\mathcal{NP}$:
  - apresenta-se um algoritmo não-determinístico polinomial para RESOLVER o problema ou
  - apresenta-se um algoritmo determinístico polinomial para VERIFICAR que uma dada solução (a solução adivinhada) é válida.

A Classe $\mathcal{NP}$

- A computação de uma máquina não-determinística que resolve um problema de decisão examina uma das soluções possíveis ao problema.
- A habilidade em selecionar não-deterministicamente uma única solução potencial, ao invés de sistematicamente examinar todas as possíveis soluções, reduz a complexidade da computação da máquina não-determinística.
- **Definição**: Uma linguagem L é aceita em **tempo polinomial não-determinístico** se houver uma máquina de Turing não determinística M que aceita L com $ct_M = \mathcal{O}(n^r)$, onde r é um número natural independente de n. A família de linguagens aceitas em tempo polinomial não-determinístico é chamada de $\mathcal{NP}$.

A Classe $\mathcal{NP}$

- Como toda máquina determinística é também não-determinística, $\mathcal{P} \subseteq \mathcal{NP}$.
- Uma linguagem aceita em tempo polinomial por máquinas multi-trilhas ou multi-fitas determinísticas está em $\mathcal{P}$.
- A construção de uma máquina de Turing padrão equivalente a partir de uma dessas alternativas preserva a complexidade de tempo polinomial.
- Há uma técnica para construir uma máquina determinística equivalente a partir das transições de uma máquina não-determinística.
- Infelizmente, esta construção não preserva a complexidade de tempo polinomial.

A Classe $\mathcal{NP}$

- **Definição**: Sejam L_1 e L_2 linguagens sobre os alfabetos Σ_1 e Σ_2, respectivamente. Diz-se que L_1 é **redutível** a L_2 em tempo polinomial se houver uma função computável em tempo polinomial $f : \Sigma_1^* \rightarrow \Sigma_2^*$ tal que $u \in L_1$ sse $f(u) \in L_2$.
- Uma redução de L_1 para L_2 transforma o problema de aceitação de L_1 ao problema de aceitação de L_2. Seja F uma máquina de Turing que computa a função f. Se L_2 é aceita por uma máquina M, então L_1 é aceita por uma máquina que
  1. "roda" F em uma cadeia de entrada $u \in \Sigma_1^*$
  2. "roda" M em $f(u)$.
- A cadeia resultante $f(u)$ é aceita por M sse $u \in L_1$.
- A complexidade de tempo da máquina composta pode ser obtida das de F e M.

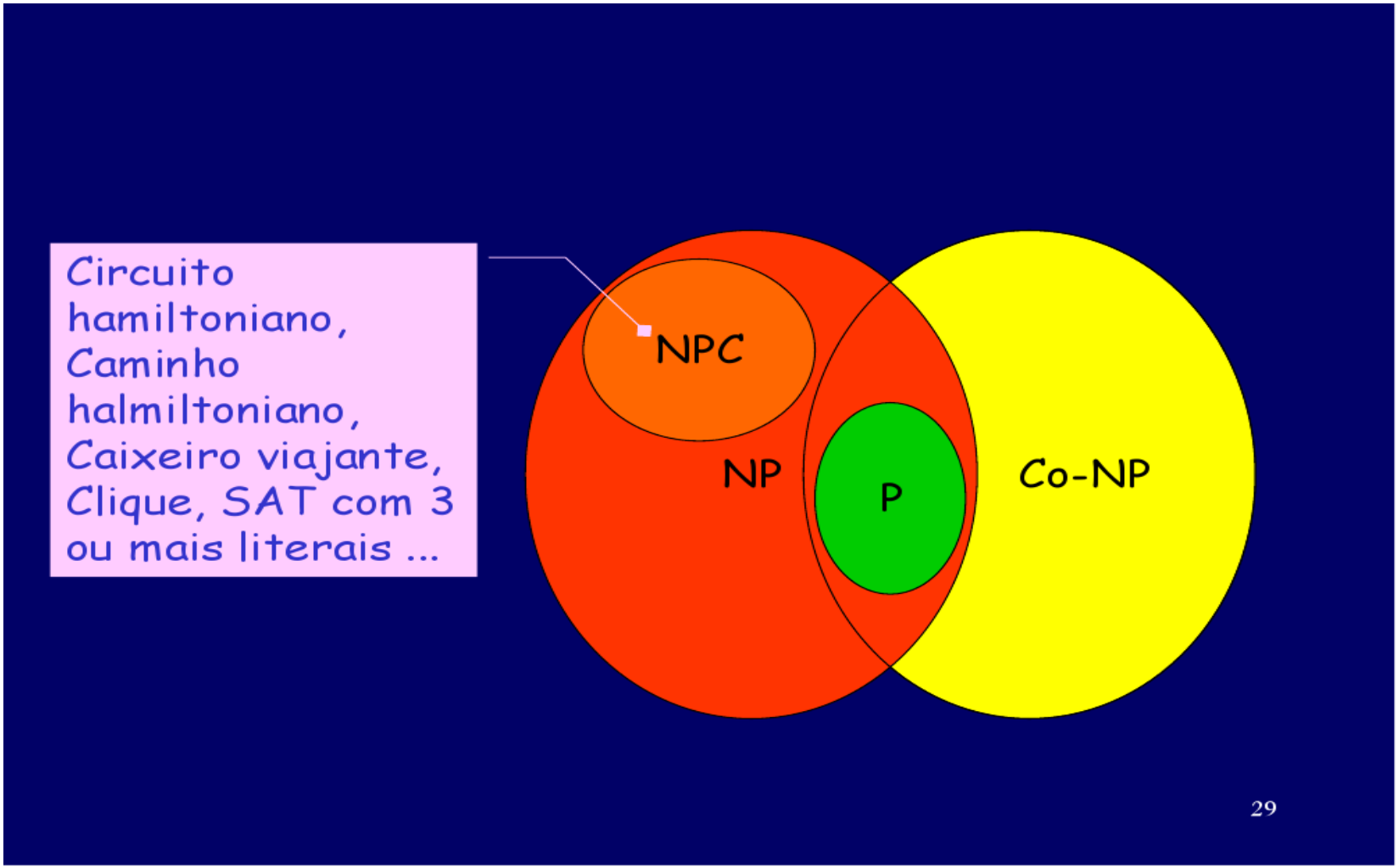
A Classe $\mathcal{NP}$

- **Teorema**: Seja L_1 redutível a L_2 em tempo polinomial e seja $L_2 \in \mathcal{P}$. Então $L_1 \in \mathcal{P}$.
- **Definição**: Uma linguagem L_2 é chamada de $\mathcal{NP}$**-difícil** se para todo $L_1 \in \mathcal{NP}$, L_1 é redutível a L_2 em tempo polinomial. Uma linguagem $\mathcal{NP}$-difícil que também está em $\mathcal{NP}$ é chamada de $\mathcal{NP}$**-completo**.
- Considera-se uma linguagem $\mathcal{NP}$-completo como uma linguagem universal na classe $\mathcal{NP}$.
- A descoberta de uma máquina em tempo polinomial que aceita uma linguagem $\mathcal{NP}$-completo pode ser usada para construir máquinas que aceitam toda linguagem em $\mathcal{NP}$ em tempo polinomial determinístico.
- **Teorema**: Se houver uma linguagem $\mathcal{NP}$-completo que também está em $\mathcal{P}$, então $\mathcal{P} = \mathcal{NP}$.

$\mathcal{NP}$-Completude [1]

- No início dos anos 1970, Cook [2] descobriu certos problemas em $\mathcal{NP}$ cuja complexidade individual era relacionada com a da classe inteira.
- Estes problemas são chamados $\mathcal{NP}$-completos.
- Eles são os mais difíceis da sua própria classe e assim pode-se escolher qualquer um deles para avançar técnicas de resolução para a classe inteira.
- Se um algoritmo de tempo polinomial existir para qualquer um destes problemas, todos os problemas em $\mathcal{NP}$ seriam resolvidos em tempo polinomial.
- Talvez seja esta a razão de se acreditar que $\mathcal{P} \neq \mathcal{NP}$.
- Assim, a classe $\mathcal{NP}$-completo tem a propriedade de que, se um problema $\mathcal{NP}$-completo puder ser resolvido em tempo polinomial todos os problemas em $\mathcal{NP}$ tem solução polinomial e $\mathcal{P} = \mathcal{NP}$.

$\mathcal{NP}$-Completude



Completude

- **Definição**: Seja $\mathcal{C}$ uma classe de complexidade e L uma linguagem/problema. Diz-se que L é $\mathcal{C}$-completo se:
  1. L está em $\mathcal{C}$
  2. Para toda linguagem $A \in \mathcal{C}$, A é redutível a L.
- Se uma linguagem L satisfaz a propriedade 2 mas não necessariamente a 1, diz-se que L é $\mathcal{NP}$-difícil.
- Assim, $\mathcal{NP}$-completo está no centro do problema de decidir se $\mathcal{P} = \mathcal{NP}$.

$\mathcal{NP}$-difíceis

- Apenas problemas de decisão podem ser $\mathcal{NP}$-completos.
- Problemas de otimização podem ser $\mathcal{NP}$-difíceis.
- E problemas indecidíveis?? Podem ser $\mathcal{NP}$-difíceis?
- *Problema da Parada*:
  - É um exemplo de $\mathcal{NP}$-difícil que não é $\mathcal{NP}$-completo.
  - Este problema, como se sabe, é indecidível pois não há nenhum algoritmo de nenhuma complexidade que possa resolvê-lo.

Um Exemplo $\mathcal{NP}$: PCV

- *Problema do Caixeiro Viajante (PCV)*:
  - *Dados*: um conjunto de cidades $C = \{c_1, c_2, \cdots, c_m\}$, uma distância $d(c_i, c_j)$ para cada par de cidades $c_i, c_j \in C$, e uma constante k.
  - *Problema*: Existe um roteiro para todas as cidades em C cujo comprimento total seja menor ou igual a k?
- Obs: pode também ser refraseado em termos de caminho hamiltoniano.
  - *Dados*: um grafo G não dirigido com peso nas arestas e um número k.
  - *Problema*: encontrar um caminho hamiltoniano cujo custo seja no máximo k.

Um Exemplo $\mathcal{NP}$: PCV

- Um problema que parece estar em $\mathcal{NP}$ mas não em $\mathcal{P}$: o PCV.
- A entrada ao PCV é a mesma da MWST: um grafo com pesos inteiros nas arestas tal como na figura e um limite de peso W.
- A questão é se o grafo tem um "ciclo hamiltoniano" de peso total no máximo W.

Um Exemplo $\mathcal{NP}$: PCV

- **Exemplo**: O grafo da figura abaixo tem apenas um ciclo hamiltoniano: o ciclo (1, 2, 4, 3, 1). O peso total deste ciclo é 15+20+18+10 = 63. Então, se W é 63 ou mais, a resposta é "sim", e se $W < 63$ a resposta é "não".

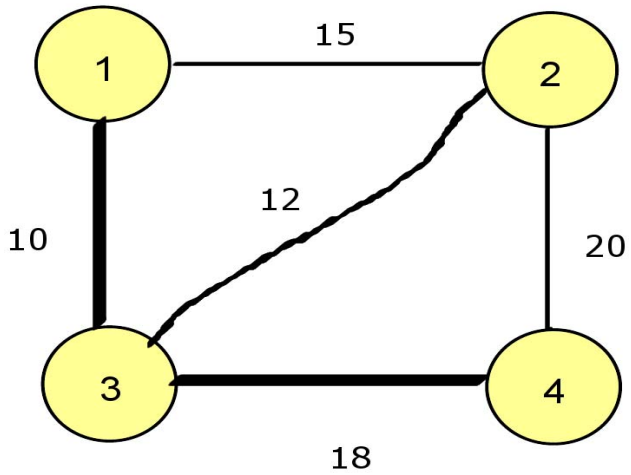


Figure : Grafo onde sua MWST é indicada pelas linhas grossas.

Um Exemplo $\mathcal{NP}$: PCV

- Entretanto, o PCV em grafos de quatro vértices é muito simples.
- Neste caso, não se tem mais de dois ciclos hamiltonianos.
- Em grafos de n vértices, o número de ciclos distintos cresce como $\mathcal{O}(n!)$, que é maior que 2^{cn} para qualquer constante c.
- Parece que para resolver o PCV precisa-se tentar todos os ciclos e computar seu peso total.
- Precisa-se examinar o número exponencial de ciclos antes de se concluir que não há nenhum com o limite de peso W ou achar um no final da exploração, na pior das hipóteses.

Um Exemplo $\mathcal{NP}$: PCV

- Por outro lado, se houver uma máquina não-determinística, pode-se advinhar uma permutação dos vértices, e computar o peso total para o ciclo de vértices naquela ordem.
- Nenhum caminho usará mais de $\mathcal{O}(m)$ passos, para uma entrada de comprimento m.
- Em uma máquina de Turing não determinística multi-fitas, pode-se advinhar uma permutação em $\mathcal{O}(m^2)$ passos e checar seu peso total em um tempo similar.
- Portanto, uma máquina de Turing não determinística de única fita pode resolver o PCV no tempo $\mathcal{O}(m^4)$ no máximo.
- Logo, o PCV está em $\mathcal{NP}$.

Um Exemplo $\mathcal{NP}$-Completo: Satisfazibilidade

- *SAT*: Checar se uma expressão booleana na forma normal conjuntiva com literais x_i ou $\neg x_i$, $1 \leq i \leq n$ é satisfazível, isto é, se existe uma atribuição de valores lógicos (V ou F) que torne a expressão verdadeira.
- Satisfazível:

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3 \vee x_2) \wedge (x_3)$$
$$x_1 = F; x_2 = V; x_3 = V$$

- Não Satisfazível:

$$(x_1) \wedge (\neg x_1)$$

Um Exemplo $\mathcal{NP}$-Completo: Satisfazibilidade

- Algoritmo:

```
Procedure Aval(E,n)
Begin
For i <- 1 to n do
  xi <- Escolhe(true,false)
if E(x1,x2,...xn) = true then sucesso
else insucesso
end
```

- O algoritmo obtém uma das 2^n atribuições de forma não-determinística em $\mathcal{O}(n)$.

SAT e o Problema da Parada

- Considere o algoritmo *A* cuja entrada é uma expressão booleana *E* na forma normal conjuntiva com *n* variáveis
- Basta tentar 2^n possibilidades e verificar se é satisfazível.
- Se for para, senão entra em *loop*.
- Logo, o problema da parada é $\mathcal{NP}$-difícil mas não é $\mathcal{NP}$-completo.

Teorema de Cook-Levin

- *SAT* é $\mathcal{NP}$-completo (**Teorema de Cook-Levin**).
- Ideia da Prova: mostrar que *SAT* está em $\mathcal{NP}$ é fácil. A parte difícil é mostrar que toda linguagem em $\mathcal{NP}$ é polinomialmente redutível a *SAT*.
- *SAT* está em $\mathcal{NP}$, desde que se possa checar o resultado de uma atribuição de valores verdade para os literais em tempo polinomial.
- Uma vez que se tem um problema $\mathcal{NP}$-completo, pode-se obter outros por redução polinomial a partir dele.
- Assim, estabelecer o primeiro $\mathcal{NP}$-completo é mais difícil.

A Questão $\mathcal{P} =^{?} \mathcal{NP}$

- É uma questão aberta se $\mathcal{P} = \mathcal{NP}$, pois a prova parece exigir técnicas ainda desconhecidas.
- Mas acredita-se que **não são** a mesma classe.

Sumário

Classes co-$\mathcal{NP}$ e $\mathcal{PS}$

- **A classe co-$\mathcal{NP}$**: Uma linguagem está em co-$\mathcal{NP}$ se seu complemento está em $\mathcal{NP}$. Todas as linguagem em $\mathcal{P}$ estão em co-$\mathcal{NP}$, mas é provável que algumas linguagens em $\mathcal{NP}$ não estejam em co-$\mathcal{NP}$ e vice-versa. Em particular, os problemas $\mathcal{NP}$-completos não parecem estar em co-$\mathcal{NP}$.
- **A classe** $\mathcal{PS}$: Uma linguagem está em $\mathcal{PS}$ (espaço polinomial) se for aceita por uma máquina de Turing determinística para a qual há um polinômio $p(n)$ tal que na entrada de comprimento n a máquina de Turing nunca usa mais que $p(n)$ quadrados de sua fita.

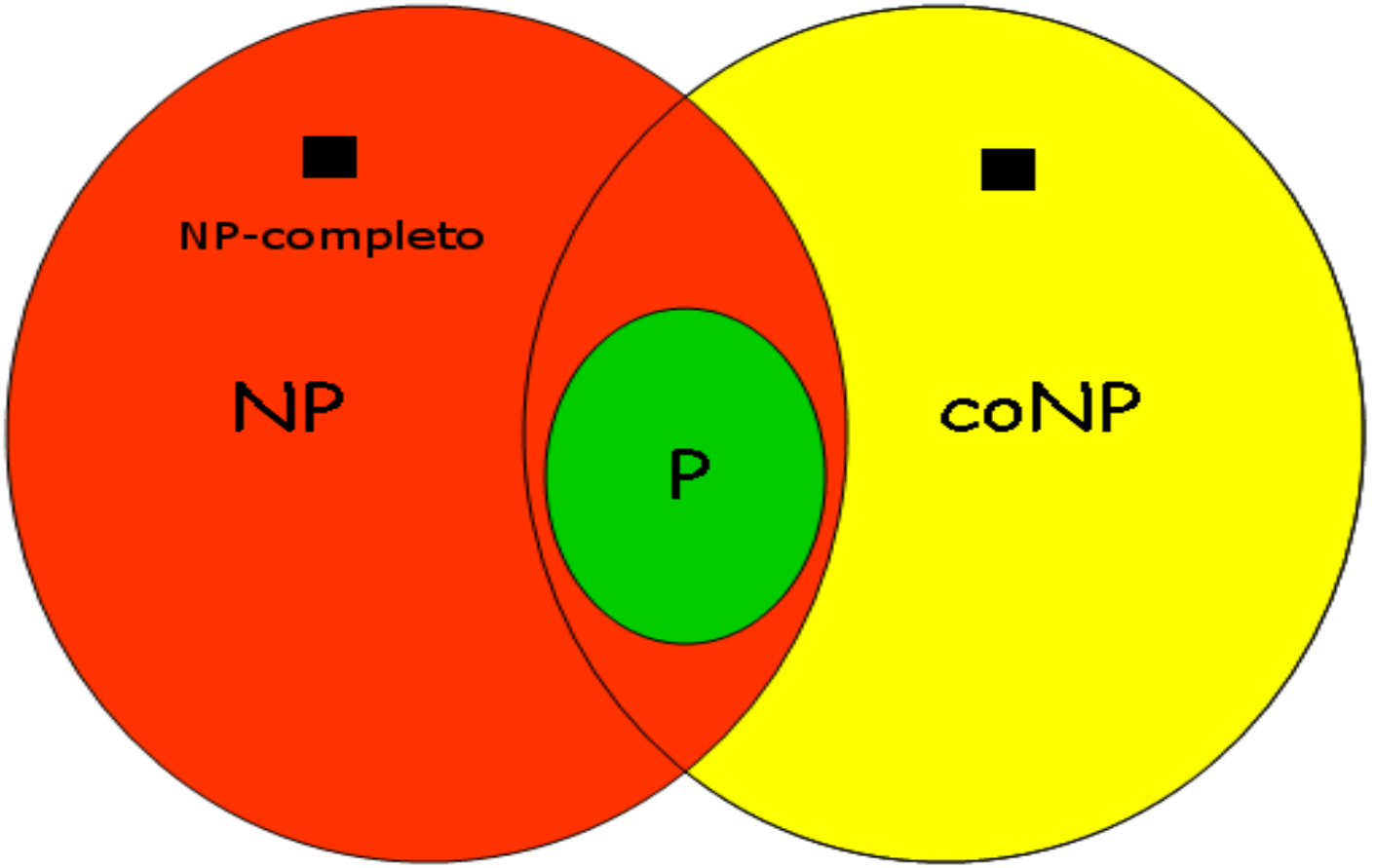
A Classe co-$\mathcal{NP}$

- Classe co-$\mathcal{NP}$ de problemas: problemas de decisão cuja solução **negativa** admite uma certificado/verificação polinomial.
- Exemplo: Validade
  - *Dado*: uma expressão booleana
  - *Problema*: Decidir se a expressão é **válida** (i.e. satisfazível para **todas** as atribuições de valores lógicos)
- Expressão Booleana Válida: $x \vee \neg x$
- Expressão Booleana Inválida: x

A Classe co-$\mathcal{NP}$

- Validade está em co-$\mathcal{NP}$?
  1. Escolha/Adivinhe uma atribuição de valores lógicos
  2. Cheque se ela **não** satisfaz a expressão.
- Por definição, o complemento de toda linguagem $\mathcal{NP}$ está em co-$\mathcal{NP}$.
- O complemento de uma linguagem co-$\mathcal{NP}$ está em $\mathcal{NP}$.
- VALIDADE está em co-$\mathcal{NP}$!
- Desde que *SAT* está em $\mathcal{NP}$... $\mathcal{P} \subseteq$ co-$\mathcal{NP}$: pois co-$\mathcal{P}$ = $\mathcal{P}$ e $\mathcal{P} \subseteq \mathcal{NP}$.

Classes $\mathcal{NP}$, $\mathcal{P}$ e co-$\mathcal{NP}$



Classes $\mathcal{NPS}$ e Algoritmos Aleatórios

- **A classe** $\mathcal{NPS}$: Pode-se também definir aceitação por uma máquina de Turing não-determinística cujo uso da fita é limitado por uma função polinomial de seu comprimento da entrada. A classe destas linguagens é referida como $\mathcal{NPS}$. Há um teorema que assegura que $\mathcal{PS} = \mathcal{NPS}$. Sabe-se que uma máquina de Turing não-determinística com limite de espaço $p(n)$ pode ser simulada por uma máquina de Turing determinística usando espaço $p^2(n)$.
- **Algoritmos Aleatórios e Máquinas de Turing**: Muitos algoritmos são produtivos de forma aleatória. Em um computador real, um gerador de números aleatórios é usado para simular o "cara-ou-coroa." Uma máquina de Turing aleatória pode alcançar o mesmo comportamento aleatório se tiver uma fita adicional na qual uma sequência de bits aleatórios é escrita.

Classes $\mathcal{RP}$ e $\mathcal{ZPP}$

- **A classe** $\mathcal{RP}$: Uma linguagem é aceita em tempo polinomial aleatório se houver uma máquina de Turing aleatória de tempo polinomial que tem no mínimo 50% de chance de aceitar sua entrada se a entrada estiver na linguagem. Se a entrada não estiver na linguagem, então esta máquina de Turing nunca aceitará. Tal máquina de Turing ou algoritmo é chamado “Monte-Carlo.”
- **A classe** $\mathcal{ZPP}$: Uma linguagem está na classe de tempo polinomial probabilística de erro zero se for aceita por uma máquina de Turing aleatória que sempre toma a decisão correta a respeito da pertinência à linguagem; esta máquina de Turing deve rodar no tempo polinomial esperado, ainda que o pior caso possa ser maior que qualquer polinômio. Tal máquina de Turing ou algoritmo é chamado “Las Vegas.”

Bibliografia I

[1] Aluisio, Sandra Maria
SCE-0185 - Teoria da Computação e Linguagens Formais.
Notas de Aula. Ciências de Computação. Instituto de Ciências Matemáticas e de Computação. Universidade de São Paulo, 2007.

[2] Cook, S. A.
The complexity of theorem proving procedures.
Proceedings of the Third Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, pp. 151–158, 1971.

Bibliografia II

[3] Copeland, J.
Biography of Turing, 2000.
http://www.alanturing.net/turing_archive/pages/Reference%20Articles/Bio%20of%20Alan%20Turing.html.

[4] Hopcroft, J. E., Ullman, J. D.
Formal Languages and Their Relation to Automata.
Addison-Wesley Publishing Company, 1969.

[5] Hopcroft, J. E., Motwani, R., and Ullman, J. D.
Introduction to Automata Theory, Languages, and Computation.
Second Edition. Addison-Wesley, 2001.

Bibliografia III

[6] Horowitz, E., Sahni, S., and Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.

[7] Moll, R. N., Arbib, M. A., and Kfoury, A. J.
An Introduction to Formal Language Theory.
Springer-Verlag, 1988.

[8] Rosa, J. L. G.
Linguagens Formais e Autômatos.
Editora LTC. Rio de Janeiro, 2010.

Bibliografia IV

[9] Sudkamp, T. A.
Languages and Machines - An Introduction to the Theory of Computer Science.
Second Edition. Addison-Wesley, 1998.

[10] Wikipedia
`http://pt.wikipedia.org/wiki/Algoritmo_de_Kruskal`.