Linguagens e Métodos Formais

José Carlos Bins Filho

12 de Dezembro de 2013

Modelos de Computação

- Modelos de computação são modelos que definem conjuntos de operações permitidas numa computação e so custos correspondente. São usados para provar teoremas sobre computabilidade e para medir a complexidade de algoritmos.
- Existem modelos computacionais simples e genéricos.
- Os modelos simples são normalmente usados em aplicações restritas. Exemplo:
  - Expressões Regulares: Especificam padrões de strings, usados principalmente em Linguagens de programação.
  - Autômatos Finitos: Usados especialmente em desenvolvimento de circuitos.
  - Linguagens Livres de contexto: Especificam a sintaxe de linguagens de programação.
  - Existem muitos outros modelos como: Autômato de pilha, diversos tipos de gramáticas, etc...

Modelos de Computação

- Hierarquia de Chomsky

Chomsky Hierarchy	Grammar	Language	Machine
Type-0	Unrestricted	Recursiely enumerable	Turing Machine
-	-	Recursive	-
Type-1	Context Sensitive	Context-Sensitive	Linear Bounded Autômaton
-	Indexed	Indexed	Nested Stack Autômaton
-	Tree-adjoining	Mildly Context-Sensitive	Enbedded Pushdown Autômaton
Type-2	Context-Free	Context-Free	PushDown Autômaton
-	Deterministic Context-free	Deterministic Context-Free	Determ. Pushdown Autômaton
-	Visibly Pushdown	Visibly Pushdown	Visibly Pushdown Autômaton
Type-3	Regular	Regular	Finite Autômaton
-	-	Star-free Language	Aperiodic Finite State Autômaton

Modelos de Computação

- Os modelos genéricos são capazes de representar qualquer tipo de computação (Tese de Church). O mais conhecido é a Máquina de Turing, mas existem outros. Os principais são:
  - Funções recursivas parciais (μ-functions). As funções primitivas recursivas estudadas são uma subclasse das μ-functions.
  - Cálculo Lambda
  - Lógica combinatória: similar ao cálculo lambda
  - Algoritmo de Markov: usa regras similares a gramáticas para operar sobre um string de símbolos
  - Máquina de Post: usa um conjunto de células e instruções para se deslocar e marcar/desmarcar as celulas
  - Máquina de Registradores: Idealização de um computador que usa registradores de tamanho ilimitado.
  - Sistemas de Produção: Basicamente Gramáticas de tipo-0.

Máquinas de Turing Universal

- Objetivo
  - Descrever uma máquina de Turing U, capaz de simular qualquer outra máquina de Turing M. Para isto a máquina deve conter na fita:
    - o conjunto de instruções sobre o comportamento da máquina a ser simulada;
    - o conteúdo da fita da máquina a ser simulada.
- Codificação: Para simplificar o processo se codifica uma máquina de Turing usando um alfabeto restrito $= \{0, 1\}$.
  $M =$
  $\{\{0, 1\}, Fita, \{0, 1, \epsilon\},$ Cabeçote, $\{q_1(estadoinicial), q_2(estadofinal), ...\},$

  - Codificando uma transição: $\delta(q_i, X_j) = (q_k, X_l, D_m)$
    Se assumirmos que:
    - $\{X_1, X_2, X_3\} = \{0, 1, \epsilon\}$ e
    - $\{D_1, D_2\} = \{Esquerda, Direita\}$

    então podemos codificar uma transição como: $0^i10^j10^k10^l10^m$
    e o conjunto P como: $111t_111t_211t_311...11t_n111$

Máquinas de Turing Universal

- Exemplo:
  - Dados $M = \{\{0,1\}, Fita, \{0,1,\epsilon\},$ Cabeçote, $\{q_1$(estado inicial), q_2(estado final), $q_3\}, P\}$ onde P é:

$$\begin{aligned} \delta(q_1,1) &= (q_3,0,D) \\ \delta(q_3,0) &= (q_1,1,D) \\ \delta(q_3,1) &= (q_2,0,D) \\ \delta(q_3,\epsilon) &= (q_3,1,E) \end{aligned}$$

    Máquina que inverte os bits de um string começando por 1.
  - O string que representa a máquina é dado por:

    11101001000101001100010101001001100010010100100110001000100100101111

  - Note que este não é o único string que satisfaz a codificação da máquina dada.

Máquinas de Turing Universal

- Exemplo (cont):
  - Se aplicarmos a sentença 1011 à máquina teremos o 1011 ao fim do string e portanto

$$< M, 1011 >= \quad 1110100100010100110001010010011 \\ 0001001001010011000100010010010111011$$

- Linguagem Universal: Defina L_u como a linguagem $\{< M, w > | M \text{ aceita } w\}$.

  L_u é chamada de universal porque o problema de um string $w \in \{0,1\}^*$ ser ou não aceito por uma máquina de Turing específica M é equivalente ao problema de w ser aceito por uma máquina M' que contém apenas simbolos $\{0, 1, \epsilon\}$ e construida usando a codificação vista anteriormente.

Máquinas de Turing Universal

- Simulador de MT: JFlap
  - O Jflap será o simulador usado para fazer o trabalho.

Problemas

- Uma Gramática Livre de Contexto dada é ambigua?
- A questão acima é um problema?
- Aqui nós estamos interessados em questões que envolvam apenas as resposta sim ou não.
- Informalmente, um problema é uma questão contendo um ou mais parâmetros e cuja resposta é sim ou não.
- Uma lista de valores (argumentos) para os parâmetros é chamado de instância do problema.
- No caso de ambiguidade acima a instância é a CFG específica.
- Restringindo apenas a problemas com respostas sim ou não, e codificando as instâncias dos problemas como strings sobre um alfabeto finito nós podemos transformar a resolução do problema na busca de um algoritmo.

Definição

- **Definição:** Um problema é decidível se existe um algoritmo que receba como entrada uma instância do problema e determine se a resposta para aquela instancia é sim ou não.
- Neste caso a linguagem que representa o problema é recursiva.
- Caso este algoritmo não exista o problema é dito indecidível.
- Alguns autores usam a noção de semi-decidibilidade. Um problema é semi-decidível se existe um algoritmo que pára quando a resposta do problema for sim, mas pode não parar caso a resposta seja não. Neste caso a linguagem que representa o problema é recursivamente enumerável.
- Uma consequência nada intuitiva é que se o problema só possui uma instância ele é trivialmente decidível.

Definição

- Exemplo:
  - Problema: Não existe inteiro positivo i, com $i \geq 3$, para o qual a equação $x^i + y^i = z^i$ é válida. (Conjectura de Fermat)
  - A conjectura de Fermat resistiu mais de 300 anos e finalmente foi provada em 1994 por Andrew Wiles, portanto agora é chamada de o Último Teorema de Fermat ou Teorema de Fermat-Wiles.
  - Portanto a resposta para o problema é um algoritmo que dê como resposta sim.
  - Note:
    - Este problema tem só uma instância, $\{x, y, z, i\}$ não são parâmetros do algoritmo mas variáveis.
    - Mesmo antes da conjectura ter sido provada o problema já era decidível, pois se a conjectura fosse falsa bastava um algoritmo que respondesse não.
    - De fato não importa se a conjectura é verdadeira ou não, para ele ser decidível ou não, porque existem algoritmos que decidem o problema em ambos os casos (verdadeira - responde sempre sim; falsa - responde sempre não)

Resumo de decidibilidade

Ling.	Ger.	Rec.	$\overline{D} = D$	Fech.	Decidível	Indecidível
Reg.	ER	AF	Sim	$L_1 \cup L_2$ $L_1 \cap L_2$ $L_1 L_2$ L^* $\overline{L}$	$L_1 = L_2$ $AF_1 = AF_2$ $L = \emptyset$ $\lvert L \rvert = \infty$ $w \in L$	
LC	GLC	AP	Não	$L_1 \cup L_2$ $L_1 L_2$ L^*	$L = \emptyset$ $\lvert L \rvert = \infty$ $w \in L$	$L_1 = L_2$ $GLC_1 = GLC_2$ $\overline{L}$ $L_1 \cap L_2$
Rec.	GT0	MT MPost	Sim	$L_1 \cup L_2$ $L_1 \cap L_2$ $L_1 L_2$ L^* $\overline{L}$		$L_1 = L_2$ $L = \emptyset$ $\lvert L \rvert = \infty$ $w \in L$
RE	GT0	MT MPost	Sim	$L_1 \cup L_2$ $L_1 \cap L_2$ $L_1 L_2$ L^*		$L_1 = L_2$ $L = \emptyset$ $\lvert L \rvert = \infty$ $w \in L$

Complexidade

- Taxa de crescimento:
  - Em análise de algoritmos é importante podermos estimar o uso de algum recurso (normalmente processamento e memória) em função do tamanho da entrada do algoritmo.
  - Um algoritmo pode ser eficiente para entradas pequenas mas ser irrealizável para entradas grandes.
  - Para descobrir se o algoritmo é factível temos que calcular a taxa de crescimento do algoritmo ou sua complexidade.
  - Isto é feito através da análise assintótica.

Análise Assintótica

- Na análise assintótica o que importa não é o custo exato de uso de um recurso, mas como este curso se comporta quando o tamanho do problema aumenta.
- Notação Big Oh.
  - Diz-se $f(n) = O(g(n))$, significando, $f(n)$ é limitada por $g(n)$, se existe um número real k e uma constante positiva c tal que:

$$\forall n > k : |f(n)| \leq c|g(n)|$$

  $g(n)$ é um dos possíveis limites superiores da função $f(n)$. Outras funções podem ser limites superiores de $f(n)$, mas em complexidade de algoritmos $g(n)$ deve ser a função com a menor taxa de crescimento que é limite superior de $f(n)$
  - Ex: $f(x) = 3x^2 + 2x + 7 \Rightarrow f(x) = O(n^2)$

$$3x^2 + 2x + 7 \leq 3x^2 + 2x^2 + 7x^2 \leq 12x^2$$
$$\text{para } x \geq 1 : \quad f(x) \leq 12x^2$$
$$\text{Logo:} \quad f(x) = O(n^2) \quad \text{com } k = 1 \text{ e } c = 12$$

Análise Assintótica

- O cálculo é uma aproximação, o que significa que nem todos os recursos são computados, mas todos os recursos importantes, principalmente os que são recursivos, devem ser levados em conta.
  - Exemplo: for (x=1; $x < n$; x++) f = f * x
    Quais eventos serão contados?
    Qual a complexidade do problema?
    E se o calculo a ser feito fosse $f = f * x * (x-1) * (x-2) * (x-3)$?

Classes de Complexidade

- **Definição:** Classe de complexidade é um conjunto de linguagens que possuem a mesma medida de complexidade para um determinado recurso.
- Note:
  - Para toda linguagem L pertencente a classe existe um algoritmo para o qual $\forall w, w \in L$ é decidível.
  - Os recursos mais usados para medida de complexidade são tempo de execução e memória.
  - O custo exato do recurso para os algoritmos L_1 e L_2, ambos pertencentes a mesma classe de complexidade, pode ser diferente, mas a medida de complexidade deles é o que determina a participação deles à classe.

Classes de Complexidade

- Classe P:
  - A primeira classe importante é a classe P (de polinomial).
  - **Definição:** $P = \{L \subseteq \{0,1\}^* |$ existe um algoritmo que decide L em tempo polinomial$\}$
    Por exemplo: $n^2, n^3, n^4, ...$ ou $O(n^k)$
  - Note:
    - Existe uma grande diferença entre o tempo de execução de um algoritmo com complexidade $O(n^2)$ e $O(n^4)$.
      Por exemplo de 10 para 100 entradas a diferença de crescimento de custo entre os dois algoritmos é de $\frac{O(n^4)}{O(n^2)} = 100 \rightarrow 10000$.
    - No entanto esta diferença é muito menor que a diferença entre algoritmos polinomiais e exponenciais, o que faz que todos os polinomiais sejam colocados em uma única classe.

Classes de Complexidade

- Classe NP:
  - Para alguns problemas não foram descobertos algoritmos polinomiais que os solucionem.
  - Duas possibilidades?
    - Não existem algoritmos para solucionar os problemas em tempo polinomial.
    - Existem algoritmos mas estes ainda não foram descobertos.
  - Verificabilidade Polinomial:
  - **Definição:** Um verificador V para uma linguagem L é um algoritmo tal que:
    $L = \{w | V \text{ aceita } < w, c > \text{ para algum string } c\}$
  - Exemplo:
    - L: conjunto de todos os grafos direcionados
    - w: um grafo direcionado que se quer testar se é Hamiltoniano
    - c: um ciclo hamiltoniano

    Note que caso c seja um ciclo Hamiltoniano então o grafo é hamiltoniano
  - Neste caso o tempo de execução da verificação é polinomial.

Classes de Complexidade

- Classe NP:
- Não ter sido descoberto um algoritmo polinomial para decidir um algoritmo não significa que o algoritmo não pertence a classe P.
- Na realidade para muitos problemas que se achava que não eram polinomiais foram descobertos algoritmos polinomiais que os decidem.
- Um problema recente (2002) foi o da composição de um número. Um número é dito composto se ele é o produto de dois outros números diferentes de 1.
- **Definição:** NP é a classe das linguagens que tem verificadores de tempo polinomiais.
- NP é uma classe importante pois possui muitos problemas de interesse prático.

Classes de Complexidade

- *PxNP*
  - *P* é a classe das linguagens que pode ser decidida em tempo polinomial
  - *NP* é a classe das linguagens que podem ser testadas em tempo polinomial
- Portanto ou $P \subset NP$ ou $P = NP$
- $P = NP$ é um dos maiores problemas matemáticos deste século.
- A maioria dos cientistas acredita que $P \neq NP$, mas não existe prova. Se $P \neq NP$ então não existe como substituir os algoritmos que usam força-bruta (teste de todas as possibilidades) para resolver um problema.

Classes de Complexidade

- NP-Completo
  - Em 1970 Cook & Levin descobriram problemas NP cuja complexidade está relacionada a da classe NP inteira.
  - Eles provaram que se existir uma solução polinomial para um destes problemas então todos os problemas NP tem solução polinomial.
  - Este tipo de problema é chamado de NP-completo
  - Portanto se for achado algum algoritmo que solucione um problema NP-completo em tempo polinomial isto significa que $P = NP$.
  - **Definição:** Uma linguagem B é NP-completa se satisfaz as condições abaixo:
    - $B \in NP$
    - $\forall A \in NP$ A é redutível a B em tempo polinomial
- Exemplo: Problema SAT (Satisfazibilidade Booleana)
  - Dada uma fórmula booleana usando apenas as operações E, OU e NÃO $(\wedge, \vee, \neg)$, a fórmula é dita satisfazível se alguma atribuição de valores para as variáveis faz o resultado da fórmula ser verdadeiro (V ou 1).
  - A fórmula $f = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$ é satisfazível.
    Para $x = 0$, $y = 1$ e $z = 0$.

Problemas Intratáveis e Algoritmos de Aproximação

- A Teoria da NP-Completude permite que o projetista concentre os seus esforços de uma forma mais produtiva ao revelar que a busca por um algoritmo eficiente para um certo problema é extremamente improvável, se não impossível
- Itens Importantes:
  - reduções são uma forma de mostrar que dois problemas são essencialmente idênticos. Um algoritmo rápido para um problema implica em um algoritmo rápido para o outro
  - Na prática um conjunto pequeno de problemas NP-Difíceis são suficientes para mostrar que outros problemas são difíceis (3-SAT, Vertex Cover, Hamiltonian Cycle)
  - Algoritmos de Aproximação garantem soluções que são próximas da solução ótima e provêem uma forma de lidarmos com problemas NP-Completos

Problemas Intratáveis e Algoritmos de Aproximação

- Exemplo de Problemas Intratáveis:
  - O Problema 3-SAT
    - Problema de satisfazibilidade de fórmulas booleanas na sua Forma Normal Conjuntiva (Produtório de Somatórios) onde cada somatório possui 3 variáveis.
      Observação: Embora k-Sat, $k \geq 3$ é NP-Completo, e Sat também é NP-Completo, o problema 2-Sat é P. Existem algoritmos (Even, Itai & Shamir (1976) e Aspvall, Plass & Tarjan (1979)) que resolvem o problema em tempo linear. (Cuidado o livro texto está incorreto pois diz que todos os algoritmos k-sat são NP-Completos)
  - O problema do caixeiro viajante (TSP)
    - Encontre uma rota que parta da cidade 1, visite as demais cidades exatamente uma vez e retorne à cidade 1, tal que a distância percorrida seja mínima
    - Em outras palavras, queremos encontrar uma permutação, dentre as n!, que tenha distância mínima.
  - Problema da Mochila (Knapsack)
    - Dada uma lista de itens com os respectivos tamanhos e valores e uma mochila com uma dada capacidade, ache o conjunto de itens de maior valor que cabem na mochila.