Conceitos Básicos sobre Linguagens e Operações com Linguagens

Alfabeto é um conjunto finito de simbolos. Σ = {a1, . . . ,an}

Palavra é uma sequencia finita de n simbolos de um alfabeto. w = w1 . . . wn

Comprimento de uma palavra é a quantidade de simbolos que ela tem. |w|

Linguagem é um conjunto de palavras. L = {w, x, y}

Com qualquer alfabeto so é possivel construir uma palavra de comprimento zero, denotada ε.

Σ\* é um conjunto infinito de todas as palavras possiveis que podem ser formadas com os simbolos de Σ. Toda linguagem é subconjunto de Σ\*.

Concatenação entre palavras é formar uma palavra nova onde a primeiro aparecem todos os simbolos da primeira palavra da operaçao e logo em seguida todos os simbolos da segunda palavra (na mesma ordem em que aparecem na palavra). w1 = α1α2 . . . αn ∈ Σ\* (|w1| = n) e w2 = β1β2 . . . βt ∈ Σ\* (|w2| = t), então w1w2 ∈ Σ\* e w1w2 = α1α2 . . . αnβ1β2 . . . βt, com |w1w2| = n + t.

Uniao: resulta em um conjunto que tem todos os elementos de cada uma das linguagens. L1 ∪ L2

Interseção: resulta em um conjunto que tem apenas os elemento que estao contidos em ambas ao mesmo tempo. L1 ∩ L2

Diferença: resulta em um conjunto que tem todos os elementos da primeira linguagem da operação excluindo os que tambem estao contidos na segunda. L1 − L2

Complemento: conjunto de todos os elementos de Σ\* excluindo os que estao contidos na linguagem. /L

Concatenaçao: sao todas as concatenaçoes entre cada elemento das duas linguagens. L1.L2

Estrela de Kleene: conjunto infinito de todas as concatenaçoes possiveis entre palavras da propria linguagem. L\*

Autômatos Finitos Determinísticos e Expressões Regulares e sua equivalência

Um AFD é uma 5-upla A = (Σ, Q, q0, F, δ). Onde Σ é o alfabeto; Q conjunto finito de estados; q0 é o estado inicial; F é conjunto finito de estados finais e δ é a função de transição δ : Q × Σ → Q, ou seja, para cada par formado por um estado de Q e um simbolo de Σ, a funçao de transição fornece um estado de Q.

Um AFD tambem pode ser representado por um grafo direcionado. Onde os verticies representam os estados. Cada verticie é representado por um circulo rotulado pelo respectivo estado. O estado inicial é marcado po uma setinha com um I apontando para ele e o estado final é marcado por um circulo duplo. As arestas do grafo representam as funções de transição, que sao rotuladas pelo simbolo de Σ. Por exemplo se temos δ(q, σ) = q′, entao teremos um verticie q que aponta para o veticie q’, essa aresta é rotulada por σ.

O autômato se chama determinístico porque, para cada par de estado de Q e símbolo de Σ, a sua função de transição δ oferece exatamente um estado como resposta.

Uma configuração de um AFD é um par formado por um estado do automato e uma palavra de Σ\*. O primeiro representa o estado atual do automato e o segundo o trecho da palavra que ainda nao foi lida pelo automato. (q, w) ⊢ (q′, w′).

Dizemos que uma palavra é aceita pelo automato quando apos o automato ler todos os seus simbolos ele termina em um estado final.

Seja L uma linguagem. Dizemos que L é uma linguagem regular se existe um AFD A tal que L = L(A), isto é, se é possível construir um AFD tal que as palavras aceitas por ele sejam exatamente as palavras que pertencem à linguagem L.

Uma ER é uma palavra no alfabeto que tambem pode usar os simbolos ∪, ·, \*,(,), ∅, ε. Temos as regras:

(1) se σ ∈ Σ então σ é uma expressão regular;

(2) ∅ e ε são expressões regulares;

(3) se r1 e r2 são expressões regulares, então (r1 ∪ r2) e (r1 · r2) também

são;

(4) se r é uma expressão regular, então r\* também é;

(5) Nada mais é considerado expressão regular

E podemos deﬁnir a linguagem L(r) recursivamente a partir das seguintes regras:

(1) Se σ ∈ Σ, então L(σ) = {σ};

(2) L(∅) = ∅;

(3) L(ε) = {ε};

(4) L(r1 ∪ r2) = L(r1) ∪ L(r2);

(5) L(r1.r2) = L(r1).L(r2);

(6) L(r\*) = L(r)\*

Dado um autômato ﬁnito M, determinar a linguagem L(M) que ele aceita:

1) Tira equaçoes do grafo. Por exemplo se do q1 com 0 ele vai pro q2 e com 1 ele vai por q3, a equaçao seria L1=0L2 U 1L3. Faça isso para todos os estados. Sendo que o estado final sempre tem U {ε}.

2) Resolver os sistemas. Se vc perceber que o automato tem um estado morto, ou seja, desse estado nao sai nenhuma setinha, entao a equaçao desse estado é igual a ∅. Caso nao tenha nenhum, resolvemos por substituiçao.

Na verdade, ao aplicar o método de substituição para resolver sistemas de equações e achar a linguagem aceita por um autômato vamos nos deparar muitas vezes com equações em que uma linguagem é escrita em termos dela própria.

Lema de Ardem:

Sejam A e B linguagens em um alfabeto Σ. Se ε /∈ A então o maior subconjunto de Σ\* que satisfaz X = AX ∪ B é X = A\*B.

Autômatos Finitos Não-Determinísticos e Expressões Regulares

e sua equivalência

Um AFND A é uma 5-upla A = (Σ, Q, q0, F, ∆), onde Σ, Q, q0 e F são deﬁnidos da mesma forma que em um AFD e ∆ é a função de transição não-determinística. Esta função tem o formato

∆ : Q × (Σ ∪ {ε}) → P(Q).

A resposta da função δ é um elemento do conjunto Q, ou seja, é um estado do AFD para onde o autômato irá ao executar a transição. Já a resposta da função ∆ é um elemento do conjunto P(Q), ou seja, é um conjunto de estados do AFND. Este conjunto de estados representa as possibilidades de estados para onde o autômato poderá ir ao executar a transição. Isto signiﬁca que, quando um AFND executa uma transição, o estado para onde ele vai é escolhido aleatoriamente dentre o conjunto de possibilidades oferecido pela resposta da função de transição. Esta é uma das fontes de não-determinismo do autômato.

Outra diferença entre a função de transição determinística e a função de transi-

ção não-determinística é que a não-deterministica pode trocar de estado sem consumir nenhum simbolo da palavra, computando o ε.

Como uma conﬁguração pode possuir várias conﬁgurações seguintes, o autômato pode ter a possibilidade de realizar diversas computações distintas com uma mesma palavra w. Devido às ε-transições, é possível também a existência de computações inﬁnitas em um AFND, o que nunca pode acontecer em um AFD. Finalmente, devido à possibilidade de que a resposta para algumas transições seja o conjunto vazio, é possível a existência de computações em um

AFND que travam sem conseguir ler a palavra da entrada completamente, o que também nunca pode acontecer em um AFD.

Autômatos Finitos Determinísticos e Não-Determinísticos e sua

equivalência

Um AFD é um caso particular de um AFND. Considere um AFND em que a função de transição ∆ satisfaz as seguintes propriedades:

(1) ∆(q, ε) = ∅, para todo q ∈ Q E

(2) ∆(q, σ) é um conjunto unitário para todo par (q, σ), onde q ∈ Q e σ ∈ Σ

Neste caso, o que temos na prática é um AFD escrito no “formato” de um AFND, uma vez que retiramos todas as possibilidades de não-determinismo.

Dado um AFND para construirmos um AFD fazemos uma tabela de transição de função da seguinte forma: de um lado os estados e do outro para onde ele vai quando le cada simbolo. Sendo que tanto o “estado” quanto pra onde ele vai, sao na verdade conjuntos de estados. Entao fazemos isso para cada estado do AFND. Quando acabarem estes, faremos a mesma coisa, agora, para cada conjunto de estado que apareceu do lado direito da tabela. Ao final cada conjunto que estiver do lado esquerdo, sera um estado do AFD, o estado inicial é o mesmo que o do AFND desde que ele nao tem nenhuma transiçao ε e os finais sera todos aqueles cujo conjuntos contem o estado final do AFND.

Gramáticas Regulares e Autômatos Finitos e sua equivalência

Uma gramática G é uma quádrupla G = (T, V, S, R), onde T é conjunto de simbolos terminais; V é conjunto de variaveis; S é simbolo inicial; R conjunto de regras.

Uma regra de uma gramática (um elemento do conjunto R da gramática) tem o formato u → v, onde u e v sao palavras do alfabeto e u contem pelo menos um simbolo de V.

Uma derivação a um passo de um gramatica é como a função de transição de um automato, ela substitui uma subpalavra que ocorre do lado esquedo da regra pelo lado direito desta mesma regra. x ⇒ y.

Uma gramática regular é uma gramática G = (T, V, S, R) onde todo elemento de R tem um dos seguintes formatos:

(1) X → aY ;

(2) X → a OU

(3) X → ε,

onde X, Y ∈ V e a ∈ T

Toda liguagem gerada por uma gramatica regular é regular. Para provar isso, mostramos que é possivel construir um automato de uma gramatica. De forma:

• Σ = T;

• Q = V ∪ {qf};

• q0 = S;

• F = {qf};

• ∆:

(1) Para cada regra no formato X → aY , adiciono Y ao conjunto ∆(X, a); (ou seja, estando no estado X, se ler a, vai para o estado Y)

(2) Para cada regra no formato X → a, adiciono qf ao conjunto ∆(X, a); (estando no X, se ler a, vai para qf)

(3) Para cada regra no formato X → ε, adiciono qf ao conjunto ∆(X, ε). (estando no X, vai para qf sem precisar ler nenhum simbolo)

Propriedades de Fechamento das Linguagens Regulares

Uma linguagem é provada regular se for possivel contruir uma ER, um AFD ou um AFND para ela.

Dizemos que a operação é fechada para uma linguagem regular se tomarmos duas linguagens quaisquer e a linguagem resultante for tambem regular.

União:

Se conseguirmos montar o automato L1 U L2, entao provamos que a linguagem L(L1UL2) é regular. Para isso basta montar um automato para cada uma das linguagens. Ai criamos mais um estado, que vai ser o estado inicial do nosso AFND, desse estado fazemos duas transições ε, uma para cada estado “inicial”(pq ele deixa de ser inicial no AFND) dos automatos ja construidos. O estados finais sao os finais de cada um.

Concatenação:

Vale a mesma regra, montamos um AFND. Um automato para cada linguagem, so que o inicial do AFND vai ser o inicial da primeira linguagem, e o final desta, deixa de ser final. e dele sai um transição ε para o estado inicial do automato da segunda linguagem. Os estados finais sao os estados finais da segunda linguagem.

Estrela:

Da mesma forma. Um automato pra linguagem. E tambem criamos um estado inicial novo, que vai ser ligado ao “inicial” da linguagem por transição ε. Alem disso, botamos tambem uma transição ε do estado final para o “inicial” da linguagem, para criar um loop infinito, ja que a estrela gera um conjunto infinito.

Interseção:

A ideia para a construção deste autômato é simular em paralelo a computação dos automatos B1 e B2 com uma mesma palavra, executando as transições nos dois automatos de forma sincronizada. O que muda em relaçao a operaçao de uniao é a condiçao de aceitaçao do novo automato. No caso da interseção, se ambos os automatos aceitarem a palavra, ela esta na interseçao das linguagens e o nosso automato deverá aceitar. Se pelo menos um dos automatos rejeitar a palavra, entao o nosso novo automato tambem devera rejeita-la.

Complemento:

Basta inverter todos os automatos finais do automato da linguagem, ou seja, tudo que era final deixa de ser, e tudos que nao eram passam a ser.

Diferença:

Temos que L1 − L2 = L1 ∩ /L2, ou seja a diferença nada mais é que uma interseçao com um complemento. Ja que sabemos que ambas sao operaçoes fechadas, entao a diferença tambem é.

Propriedades de Fechamento das Linguagens Livres de Contexto

Uniao:

Se conseguirmos construir uma gramatica livre de contexto para a uniao de duas linguagens livres de contexto, entao esta provado que a operaçao é fechada. Basta criar cada gramatica para as linguagens e mais uma nova variavels e duas regras. Da forma: S → S1 e S → S2, onde S1 e S2 sao os simbolos iniciais das linguagens envolvidas na operaçao.

Concatenação:

Nesse caso o que fazemos é criar uma variavel e uma regra: S → S1S2

Estrela de Kleene:

Fazemos: S → S1S e S → ε

Interseção:

Para provar que uma operação nao é fechada basta mostrar um exemplo que nao de uma gramatica livre de contexto.

Sejam

L1 = {a^nb^nc^m : m, n ≥ 0} e

L2 = {a^mb^nc^n : m, n ≥ 0}. que sao livres de contexto.

Uma gramática livre de contexto G = (T, V, S, R) que gera L1 é a gramática com T = {a, b, c}, V = {X, Y, Z}, S = X e R = {X → Y Z, Y → aY b, Y → ε, Z → cZ, Z → ε}. Uma gramática livre de contexto análogo gera L2.

L1 ∩ L2 = {a^nb^nc^n : n ≥ 0}. que nao é livre de contexto.

Complemento:

Começamos relembrando uma igualdade entre conjuntos conhecida como Lei de De Morgan:

L1 ∩ L2 = /(/L1 ∪ /L2). Supondo que /L1 e /L2 sao fechadas em relaçao a gramaticas livres de contextos. Como ja sabemos que uniao tambem é, entao: /L1 ∪ /L2 é fechada, e como estamos supondo que o complemento é fechado entao /(/L1 ∪ /L2) tambem deve ser. Mas /(/L1 ∪ /L2) = L1 ∩ L2, que ja sabemos que nao é livre de contexto. Entao o complemento tambem nao é fechada.

Diferença:

Temos que /L = Σ\* − L. A diferença nao pode ser fechada porque a estrela é fechada, mas o complemento nao é fechado, entao essa igualdade é falsa.

O Lema do Bombeamento para Linguagens Regulares e seu uso

É conveniente estabelecer a noção de bombeabilidade em um contexto mais geral que o das linguagens regulares. Seja L uma linguagem em um alfabeto Σ e seja w ∈ L. Dizemos que y ∈ Σ\* é uma subpalavra de w bombeável em L se

(1) y != ε;

(2) existem x, z ∈ Σ\* tais que w = xyz;

(3) xy^kz ∈ L para todo k ≥ 0.

Seja M um autômato ﬁnito determinístico com n estados e seja L a linguagem aceita por M. Se w é uma palavra de L com comprimento maior ou igual a n (numero de estados) então existe uma decomposição de w na forma w = xyz, onde

(1) y != ε(|y| > 0);

(2) |xy| ≤ n;

(3) xy^kz ∈ L para todo k ≥ 0.

Aplicação do Lema do Bombeamento: Provar que uma linguagem NÃO é regular.

Suponha, então, por contradição, que L seja aceita por algum autômato ﬁnito determinístico com n estados. De acordo com o lema do bombeamento qualquer palavra w ∈ L de comprimento maior ou igual a n terá que admitir uma subpalavra bombeável. Assim, para obter uma contradição, basta achar uma palavra satisfazendo essa condição de comprimento L. E depois disso vc tem que se virar nos 30 pra conseguir provar alguma coisa. Veja exemplos da apostila (Pag.96).

Gramáticas Livres de Contexto e Derivações

Seja G uma gramática com conjunto de terminais T, conjunto de variaveis V e simbolo S. Dizemos que G é livre de contexto se todas as regras sao do tipo X → w, onde X ∈ V e w ∈ (T ∪ V )\*.

Observe que as regras de uma gramática regular se encaixam neste formato.

Portanto, toda gramática regular é livre de contexto.

Seja G uma gramática livre de contexto com terminais T,variáveis V , símbolo inicial S e conjunto R de regras, e sejam w, w′ ∈ (T ∪ V )\*. Dizemos que w′ pode ser derivada em um passo a partir de w em G, notação w ⇒G w′, se w′ pode ser obtida a partir de w substituindo-se uma variável que aparece em w pelo lado direito de alguma regra da gramática G que possua esta mesma variável no seu lado esquerdo. Em outras palavras, para que w ⇒G w′ é preciso que:

(1) exista uma decomposição da forma w = uXv, onde u, v ∈ (T ∪ V )\* e X ∈ V ;

(2) exista uma regra X → α em R E

(3) w′ = uαv.

Dizemos que w′ pode ser derivada a partir de w em G se existem palavras w1, . . . , wn−1 ∈ (T ∪ V )\* tais que w = w0 ⇒ w1 ⇒ · · · ⇒ wn−1 ⇒ wn = w′. Chamamos a isto uma derivação de w′ a partir de w em G e escrevemos w ⇒\* w′.

O conjunto de todas as palavras de T\* que podem ser derivadas a partir do simbolo inicial S na gramatica G é a linguagem gerada por G. Denotado por L(G) a linguagem gerada por G e usando a notação definida acima, temos que: L(G) = {w ∈ T\* : existe uma derivação S ⇒∗ w em G}.

Dizemos que L é uma linguagem livre de contexto se existe uma gramática livre de contexto G tal que L = L(G), isto é, se é possível construir uma gramática livre de contexto que gere a linguagem L.

O que diferencia uma gramatica livre de contexto de uma não livre de contexto sao as regras que podemos escrever. Para uma gramática não ser livre de contexto basta que, do lado esquerdo da seta de alguma de suas regras, apareça algo mais complicado que uma variável

isolada.

Arvores de Análise Sintática e Ambiguidade

As árvores de análise sintática são deﬁnidas recursivamente da seguinte maneira:

Árvores básicas: se σ ∈ T, X ∈ V e X → ε é uma regra de G, então

σ • e X--ε

são árvores gramaticais.

Se o vértice v de uma árvore gramatical T está rotulado por uma variável X, e seus ﬁlhos por α1, . . . , αn ∈ T ∪ V então X → α1 · · · αn tem que ser uma regra da gramática G. Diremos que esta é a regra associada ao vértice v. No caso de v ser a raiz, T é uma X-árvore. Caso a árvore gramatical consista apenas de uma folha rotulada por um terminal σ, diremos que se trata de

uma σ-árvore.

Uma vez tendo introduzido árvores gramaticais, temos uma outra maneira de gerar palavras a partir de uma gramática livre de contexto. Para isso, deﬁnimos a colheita c(T ) de uma árvore gramatical T (também chamada de resultado da árvore gramatical). Como a deﬁnição de árvore é recursiva, assim será a deﬁnição de colheita. As colheitas das árvores básicas são c(T) = c(T1) · c(T2)· · · c(Tn).

Digamos que w é uma palavra que pode ser derivada em uma gramática livre de contexto G com símbolo inicial S. Uma árvore de derivação para w é uma S-árvore de G cuja colheita é w. Note que reservamos o nome de árvore de derivação para o caso especial das S-árvores.

Seja G uma gramática livre de contexto e w ∈ L(G). Então:

(1) existe uma árvore de derivação cuja colheita é w;

(2) a cada árvore de derivação cuja colheita é w corresponde uma única

derivação mais à esquerda de w;

(3) a cada árvore de derivação cuja colheita é w corresponde uma única

derivação mais à direita de w.

A gramática G é ambígua se existe uma palavra w ∈ L(G)

que admite duas árvores de derivação distintas em G.

-----------------------------------------------------------------------------------------------------------------

As seguintes aﬁrmações são equivalentes entre si:

(1) L é uma linguagem regular.

(2) L é aceita por algum AFD.

(3) L é aceita por algum AFND.

(4) L pode ser gerada por uma expressão regular.

(5) L pode ser gerada por uma gramática regular.

PS: Sorvete é bom e deve ser comido de colher, nao garfo.

Gramaticas Livres de Contexto e Automatos de Pilha e sua equivalencia

No caso de gramáticas regulares, as regras são extremamente rígidas: uma variável só pode ser levada na concatenação de algum terminal com alguma variável, sendo que a variável tem que estar à direita do terminal, ou então levada em um terminal isolado ou em ε. Ja as gramaticas livres de contexto, tem regras mais flexiveis. A unica restrição é que a esquerda da seta deve ter uma variavel isolada.

Dizemos que w′ pode ser derivada em um passo a partir de w em G, notação w ⇒G w′, se w′ pode ser obtida a partir de w substituindo-se uma variável que aparece em w pelo lado direito de alguma regra da gramática G que possua esta seta so pode ter uma variavel isolada.

mesma variável no seu lado esquerdo. Em outras palavras, para que w ⇒G w′ é preciso que:

(1) exista uma decomposição da forma w = uXv, onde u, v ∈ (T ∪ V )\* e X ∈ V ;

(2) exista uma regra X → α em R E

(3) w′ = uαv.

Dizemos que L é uma linguagem livre de contexto se existe

uma gramática livre de contexto G tal que L = L(G), isto é, se é possível construir

uma gramática livre de contexto que gere a linguagem L.

Um autômato de pilha não-determinístico (abreviado como

AP) A é uma 6-upla A = (Σ, Γ, Q, q0, F, ∆), onde:

• Σ é o alfabeto da entrada;

• Γ é o alfabeto da pilha;

• Q é um conjunto ﬁnito de estados;

• q0 ∈ Q é o estado inicial;

• F ⊆ Q é o conjunto de estados ﬁnais e

• ∆ é a função de transição não-determinística. Esta função tem o formato

∆ : Q × (Σ ∪ {ε}) × (Γ ∪ {ε}) → Pf(Q × Γ\*).

Os automatos de pilha nao-deterministico sao a classe de automatos que aceitam linguagens livres de contexto. Ao contrario dos automatos finitos, os automatos de pilha tem memoria infinita, porem o acesso a ela é restrito, ja que o último item que foi posto na memória é brigatoriamente o primeiro a ser consultado.

O procedimento para determinar se uma palavra w pertence a uma linguagem L consistem em, utilizando uma memoria infinita em forma de pilha, ir empilhando ou desempilhando simbolos da palavra ou nao, quando necessario, como uma maneira de lembrar o que ja foi lido e mudando o automato de estado.

A função de transição pode ser escrita da forma definida acima ou em uma tabela, onde na primeira coluna ficam os estados (o estado atual do automato); na segunda a entrada (o simbolo da palavra que esta sendo lido); na terceira o topo da pilha (o simbolo a ser lido no topo da pilha); e na quarta a transiçao em si, da forma (p, u) onde p é um estado e u uma palavra.

O que acontece na transiçao é: quando as tres primeira colunas forem verdade, o automato vai trocar do estado que aparece na primeira coluna pelo estado p, e trocar o simbolo que esta no topo da pilha por u, de forma que o ultimo simbolo de u fique no topo da pilha.

Outros casos que podem acontecer sao:

na segunda coluna aparece ε; entao a entrada nao é consultada;

na terceira coluna aparece ε; entao o topo da pilha nao é consultado;

na terceira coluna tem algo diferente de ε e u=ε; remove o simbolo no topo da pilha;

na terceira coluna aparece ε e u!=ε; apenas empilha u;

na terceira coluna aparece ε e u=ε; nao altera a pilha.

A tabela de transiçao, tambem pode ter uma quinta coluna contendo comentarios que descrevam a transiçao.

M é um autômato de pilha cujos elementos são dados pelo vetor (Σ, Γ, Q, q0, F, ∆), diremos que uma palavra w ∈ Σ\* é aceita por M se existe uma computação

(q1, w, ε) ⊢\* (p, ε, ε),

onde p é um estado ﬁnal de M.

Em muitos casos é util que exista uma transiçao que troque de estado sem ler nenhuma entrada nem alterar nada na pilha, assim o automato troca de estado “aleatoriamente”.

Outra tecnica util é inventar outro simbolo para o alfabeto de pilha e acrescenta-lo a pilha quando ainda vazia, dessa forma ele estara marcando o final da pilha, porem pela definiçao de aceitaçao de uma palavra a pilha deve estar vazia para ela ser aceita, entao, tem que ter uma transiçao no final que retire essa simbolo do fim da pilha quando terminar a computaçao.

O Lema do Bombeamento para Linguagens Livres de Contexto e seu uso

A estratégia é a mesma adotada para o caso de linguagens regulares. Isto é, provaremos que toda linguagem livre de contexto satisfaz uma propriedade de bombeamento. Portanto, uma linguagem que não satisﬁzer esta propriedade não pode ser livre de contexto. Começamos com um resultado relativo a árvores que será necessário na demonstração do lema do bombeamento.

Seja f(h) o número de folhas de uma árvore m-ária completa de altura h. Como a árvore m-ária completa é aquela que tem o maior número possível de folhas, o problema estará resolvido se formos capazes de encontrar uma fórmula para f(h) em função de h e m. Faremos isto determinando uma relação de recorrência para f(h) e resolvendo-a. Para começar, se a árvore tem altura zero, então consiste apenas de um vértice. Neste caso há apenas uma folha, de modo que f(0) = 1. Para estabelecer a relação de recorrência podemos imaginar que T é uma árvore m-ária completa de altura h. É claro que, removendo todas as folhas de T , obtemos uma árvore m-ária completa T′ de altura h − 1. Para reconstruir T a partir de T′ precisamos repor as folhas.Fazemos isso dando m-ﬁlhos a cada folha de T′. Como T′ tem f(h − 1) folhas, obtemos f(h) = mf(h − 1).

Assim,

f(h) = mf(h − 1) = m²f(h − 2) = · · · = m^hf(0) = m^h

Portanto, f(h) = m^h é a fórmula desejada.

Para estabelecer uma relaça entre esse resultado e o lema do bombeamento precisamos de uma definiçao. Seja G uma linguagem livre de contexo, a amplitude α(G) de G é o comprimento maximo das palavras que aparecem a direita da seta em uma regra de G.

Se uma gramatica livre de contexto tem amplitude α, entao todas as suas arvores gramaticais sao α-arias.

Seja G uma linguagem livre de contexto. Se X é uma variável de G e se w é colheita de uma X-árvore de G de altura h então

|w| ≤ α(G)^h

Seja G uma gramática livre de contexto. Existe um número inteiro ρ, que depende de G, tal que, se w ∈ L(G) e |w| ≥ ρ, então existe uma decomposição de w na forma w = uvxyz, onde

(1) vy != ε;

(2) |vxy| ≤ ρ;

(3) uv^nxy^nz ∈ L(G), para todo n ≥ 0.

Suponhamos que G tem k variáveis e que α(G) ≥ 2; neste caso escolheremos ρ = α(G)

k+1 (no caso em que α(G) = 1, o lema funcionará com a escolha de ρ = 2). Seja w ∈ L(G) uma palavra com comprimento maior ou igual a ρ. Já sabemos, pelo teorema do capítulo anterior, que deve existir ao menos uma árvore de derivação com colheita w. Entre todas as árvores com esta colheita, escolha uma, que chamaremos de T , que satisfaça a seguinte propriedade:

Hipótese 1: T tem o menor número possível de folhas entre todas as árvores de derivação de colheita w em G. Como a colheita de T tem comprimento maior ou igual a ρ = α(G)^k+1, que é maior do que α(G)^k, segue do lema da seção 1 que T tem altura pelo menos k + 1. Chamando de C o mais longo caminho em T que vai de sua raiz a uma folha, concluímos que C tem, pelo menos, k + 1 arestas. Logo C tem, no mínimo, k+ 2 vértices. Como só pode haver uma folha num tal caminho, então C tem k+ 1 vértices interiores. Contudo, k é o número de variáveis da gramática, e cada vértice de C está rotulado por uma variável. Portanto, pelo princípio da casa do pombo, há dois vértices diferentes em C rotulados pela mesma variável. Entre todos os pares de vértices de C rotulados pela mesma variável, escolha aquele que satisfaz a seguinte propriedade:

Hipótese 2: o vértice ν1 precede o vértice ν2 e todos os vértices de C entre ν1 e a folha são rotulados por variáveis distintas.

Lembre-se que as árvores que estamos considerando são grafos orientados. Portanto, C é um caminho orientado; logo faz sentido dizer, de dois vértices de C, que um precede o outro. Seja A a variável que rotula ν1 e ν2. Temos então duas A-árvores: T1, com raiz em ν1, e T2 com raiz em ν2. Denotaremos por x a colheita de T2. Observe que, como ν1 precede ν2 ao longo de C, então x é uma subpalavra da colheita de T1. Assim, podemos decompor a colheita de T1 na forma vxy. Entretanto, pela hipótese 2, o caminho (ao longo de C) que vai de ν1 à folha tem, no máximo, k + 2 vértices (contando com a folha, que é rotulada por um terminal!). Além disso, como C é o mais longo caminho que vai da raiz de T a uma folha, o trecho de C que começa em ν1 é o mais longo caminho em T1 entre sua raiz (que é ν1) e uma folha. Portanto, T1 tem altura no máximo k + 1.

Concluímos, utilizando novamente o lema da seção 1 que, como vxy é a colheita de T1, então

|vxy| ≤ α(G)^k+1 = ρ

Isto prova (2) do enunciado do lema.

Considere agora o que acontece na construção de T quando chegamos a ν2. Este vértice é rotulado pela variável A e a ele está associada uma regra que tem A do lado esquerdo da seta. Mas suponha que, chegados a ν2, decidimos aplicar a mesma regra que aplicamos quando chegamos a ν1. Podemos fazer isto porque esta também é uma regra que tem A do lado esquerdo. Se continuarmos, vértice a vértice, teremos uma nova árvore gramatical em G, cuja colheita é uv2xy2 z. Se repetirmos este procedimento n vezes, obteremos uma árvore cuja colheita é uvnxyn z. Isto prova (3) quando n > 0.

Por outro lado, ao chegar ao vértice ν1, também podemos usar a regra associada a ν2 e continuar a construir a árvore como se fosse T2. Neste caso obteremos uma árvore T0 com menos vértices que T e com colheita uxz, que corresponde a tomar n = 0 em (3).

Só nos resta mostrar que vy != ε. Mas se vy fosse igual a ε então a árvore T0, construída no parágrafo anterior, teria colheita igual a w, e menos folhas que T , o que contradiz a hipótese 1. Portanto, vy != ε e provamos (1), concluindo assim a demonstração do lema do bombeamento.

Maquinas de Turing, Linguagens Recursivas e Recursivamente Enumeraveis

A maquina de Turing é semelhante a um automato finito, mas com memoria infinita e irrestrita, é um modelo muito mais acurado de um computador de proposito geral. Uma maquina de Turing pode fazer tudo que um computador real pode fazer, porem, ela nao pode resolver certos problemas, problemas que estao alem dos limites teoricos da computaçao.

O modelo da maquina de Turing usa uma fita infinita como sua memoria ilimitada. Ela tem uma cabeça de fita que pode ler e escrever simbolos e mover-se sobre a fita. Inicialmente a fita contem apenas a cadeia de entrada e esta em branco em todo o resto. Se a maquina precisa armazenar informaçao, ela pode escreve-la sobre a fita. Para ler a informaçao escrita, a maquina pode mover sua cabeça de volta para a posiçao onde a informaçao for escrita. A maquina continua a computar ate que ela decida produzir uma saida. As saidas aceite e rejeite sao obtidas entrando em estados de aceitaçao ou rejeiçao. Se nao entrar em nenhum dos dois estados ela continuara para sempre.

Uma maquina de Turing (deterministica) é uma 7-upla M = (Q, Σ, Γ, q0, qaceita, qrejeita, δ ), onde:

• Q é o conjunto finito de estados;

• Σ é o alfabeto de entrada sem o simbolo em branco;

• Γ é o alfabeto de fita, onde simbolo em branco ∈ Γ e Σ ⊆ Γ;

• δ: Q x Γ → Q x Γ x {E, D} é a funçao de transiçao;

• q0 ∈ Q é o estado inicial;

• qaceita ∈ Q é o estado de aceitaçao;

• qrejeita ∈ Q é o estado de rejeiçao, onde qaceita != qrejeita.

Uma maquina de Turing M aceita uma entrada w se uma sequencia de configuraçoes C1, C2, …, Ck existe onde:

1. C1 é a configuraçao inicial de M sobre a entrada w;

2. cada Ci origina Ci+1 e

3. Ck é um configuraçao de aceitaçao.

Uma linguagem é recursivamente enumerável (ou Turing-reconhecivel) se alguma maquina de Turing a reconhece.

Uma maquina de Turing, ao receber uma entrada, pode aceitar, rejeitar ou entrar em loop. A maquina falha ao aceitar uma entrada quando passa ao estado qrejeita ou entra em loop. Maquinas que nunca entram em loop sao chamadas decisores, porque elas sempre tomam uma decisao de aceitar ou rejeitar. Um decisor que reconhece uma linguagem tambem é dito que ele decide essa linguagem.

Um linguagem é recursiva (ou Turing-decidivel) se alguma maquina de Turing a decide.

Maquinas de Turing de Multiplas Fitas e sua equivalencia com as maquinas tradicionais

Uma maquina de Turing multifita é como uma maquina de Turing comum com varias fitas. Cada fita tem sua propria cabeça para leitura e escrita. Inicialmente a entrada aparece sobre a fita 1, e as outras iniciam em branco. A funçao de transiçao é modificada para permitir ler, escrever e mover as cabeças em algumas ou todas as fitas simultaneamente:

δ: Q x Γ^k → Q x Γ^k x {E, D, P}^k,

onde k é o numero de fitas.

Maquinas de Turing multifita sao equivalentes a maquinas de Turing comuns. O que quer dizer que reconhecem a mesma linguagem.

É possivel converter uma maquina multifita M em uma de unica fita S.Digamos que M te k fitas. Entao S simula o efeito de k fitas armazenando sua informaçao na sua unica fita. Ela usa o novo simbolo # como um delimitador para separar o conteudo das diferentes fitas. Alem do conteudo dessas fitas, S tem de manter registro das posiçoes das cabeças. Ela faz isso escrevendo um simbolo de fita com um ponto acima dele para marcar o local onde a cabeça estaria naquela fita.

Maquinas de Turing Nao-Deterministicas e sua equivalencia com as maquinas tradicionais

Em qualquer ponto de uma computaçao, a maquina de Turing nao-deterministica pode proceder de acordo com varias possibilidades. A computaçao de uma maquina de Turing nao-deterministica é uma arvore cujos ramos correspondem a diferentes possibilidades para a maquina. Se algum ramo da computaçao leva ao estado de aceitaçao, a maquina aceita sua entrada.

Toda maquina de Turing nao-deterministica tem uma maquina de Turing deterministica que lhe é equivalente.

Podemos simular uma maquina nao-determistica N com uma deterministica D: A maquina D tem 3 fitas, o que é equivalente a ter uma fita. A fita 1 sempre contem a cadeia de entrada e nunca é alterada. A fita 2 mantem uma copia da fita de N em algum ramo de sua computaçao nao-deterministica. A fita 3 mantem o registro da posiçao D na arvore de computaçao nao-deterministica de N.

Propriedades de Fechamento das Linguagens Recursivas e Recursivamente Enumeraveis

A classe recursivamente enumerável contém a classe das recursivas. Por esse motivo, se provarmos o fechamento de alguma operação para recursivamente enumeráveis, isso significa que as recursivas também podem se utilizar dessa prova.

A união é fechada para máquinas de Turing. Podemos criar uma máquina de Turing capaz de simular duas outras máquinas ao mesmo tempo. Teremos,duas fitas e o conjunto de estados será um conjunto de combinações dois a dois dos estados das máquinas anteriores. O conjunto de estados finais erá formado por qualquer estado que possua um ou dois estados finais das máquinas originais. Ou seja, a máquina aceita uma palavra se A ou B aceitam. Logo, ela equivale a A U B, e a união é fechada.

A interseção pode ser verificada de forma análoga, porém só entramos em um estado final caso ambos os estados que fazem parte da combinação estado de A x estado de B forem estados finais nas máquinas originais A e B. Dessa forma, podemos dizer que a interseção também é fechada.

Para uma linguagem ser recursiva, basta que ela seja decidida por um decisor; isto é, existe uma máquina de Turing M que decide a linguagem, sendo que dada uma entrada qualquer, sempre teremos uma saída, que dirá se a entrada pertence ou não à linguagem. Podemos perceber que é intuitivo descobrir o complemento dessa linguagem: basta trocar de lugar os estados de aceitação e rejeição. Com isso, provamos que linguagens recursivas são fechadas no complemento.

Uma vez que as linguagens recursivas são fechadas tanto na interseção quanto no complemento, e sabendo que a diferença de A e B nada mais é do que A interseção /B, podemos afirmar, também, que a diferença é fechada para linguagens recursivas.

Por outro lado, uma linguagem L recursivamente enumerável tem complemento também recursivamente enumerável, isso garante o caminho contrário. Então, sempre vamos ter uma resposta para uma entrada qualquer, e portanto tanto L quanto seu complemento são decidíveis, ou seja, recursivas. Mas isso significaria que todas as linguagens recursivamente enumeráveis são também recursivas, o que sabemos que não é verdade. Portanto, linguagens recursivamente enumeráveis não são fechadas no complemento.

Com isso, temos também a implicação que a diferença não é fechada para linguagens recursivamente enumeráveis, apesar de ser para linguagens recursivas.

Tese de Church-Turing e Maquina de Turing Universal

Geralmente assume-se que um algoritmo deve satisfazer os seguintes requisitos:

1. O algoritmo consiste de um conjunto finito de instruções simples e precisas, que são descritas com um número finito de símbolos.
2. O algoritmo sempre produz resultado em um número finito de passos.
3. O algoritmo pode, a princípio, ser executado por um ser humano com apenas papel e lápis.
4. A execução do algoritmo não requer inteligência do ser humano além do necessário para entender e executar as instruções.

Tudo aquilo que é comumente considerado como “algoritmicamente computavel” pode ser

computado por uma Maquina de Turing. Um problema e decidvel se e somente

se ele e decidivel por uma Maquina de Turing.

Argumentos a favor da tese:

- Maquinas de Turing “anabolizadas” nao possuem maior poder computacional

- λ**-**Calculo de Alonzo Church

- Funçoes μ-recursivas

- Maquinas de Registradores

Uma maquina de Turing Universal U é capaz de simular qualquer outra maquina de Turing. Para isso ela deve conter na fita o conjunto de instruçoes sobre o comportamento da maquina a ser simulada e o conteudo da fita da maquina a ser simulada.

Σ= {[simbolo de inicio da fita], 0, 1, σ, q, X, Y, Z, #, [simbolo espaço vazio], a, b}

Este alfabeto e usado para descrever tanto os estados, quanto os simbolos da maquina M. Para isso:

1. Enumeramos, separadamente, os simbolos e os estados de M;

2. Para distinguir simbolos de estados, adicionamos aos simbolos o prefixo σ e aos estados o prexo q;

3. Representamos as setas → e ← como se fossem simbolos de M.

Digamos que a maquina M a ser simulada tem n estados e m simbolos. Os estados e os simbolos de M serao enumerados em unario na fita de U. Isto e, o numero n sera representado por 000...0 = 0^n; uma sequencia de n zeros.

A representaçao de um simbolo ocupa no maximo m+3 casas. Se menos de m+3 casas estiverem sendo ocupadas, as restantes serao preenchidas por 1's. Por exemplo, o k-esimo simbolo aparece como σ0^k1^(m+2-k).

E o k-esimo estado aparece como q0^k1^(n-k).  
A transiçao δ(qi, j) = (qr, s) de M e codicada na ta de U na forma:

| X | q0^i1^(n-i) | σ0^j1^(m+2-j) | q0^r1^(n-r) | σ0^s1^(m+2-s) | X |

Na descriçao completa de M com entrada w na fita de U abaixo, Si denota um segmento de transiçao, e sempre aparece entre dois X's.

| [simbolo inicio de fita] | X | S1 | X |...| St | Y | Q | Z | σ | u1 | # | u2 |...|

Ate Y: Comportamento (transiçoes) de M

De Y ate Z: Q é o estado atual de M

De Z ate final: w

Decidibilidade e o Problema da Parada

Suponha que exista um algoritmo P que receba como entrada o codigo de um outro algoritmo A e uma entrada e para A e responda sempre corretamente se o algoritmo A para ou nao quando é executado com entrada e. Isto é, P responde “Sim” se o algoritmo A para com a entrada e e responde “Nao” se o algoritmo P nao para (entra em loop) com a entrada e. Dizemos que este algoritmo hipotetico P resolveria o Problema da Parada. Vamos mostrar que o algoritmo P nao pode existir. Em outras palavras, vamos mostrar que o Problema da Parada nao é algoritmicamente soluvel. Pela tese de Church-Turing, tal algoritmo existe se e somente se a condiçao que ele testa pode ser expressa de forma que possa ser decidida por uma Maquina de Turing. A representaçao de P no formalismo das Maquinas de Turing seria uma Maquina de Turing MH que decida a linguagem H = {c(M)Zc(w) : a maquina de Turing M para com entrada w}, onde c(M) e c(w) representam, respectivamente, codificaçoes apropriadas de M e de w no alfabeto da fita de MH. Para c(M), podemos utilizar, por exemplo, a mesma codificaçao que é utilizada na fita da Maquina de Turing Universal U. Primeiramente, notamos que esta é uma linguagem recursivamente enumeravel, pois a maquina de Turing universal U aceita precisamente esta linguagem, como vimos anteriormente. Vamos mostrar, no entanto, que esta nao é uma linguagem recursiva, isto é, que nao pode existir uma maquina MH que decida esta linguagem. Se H for recursiva (respectivamente, recursivamente enumeravel), entao a

linguagem H′ = {c(M) : a maquina de Turing M para com entrada c(M)} tambem é recursiva (respectivamente, recursivamente enumeravel). Se existe uma maquina de Turing MH que decide (aceita) H, entao uma maquina de Turing MH′ que transforma a entrada ⊲c(M) em ⊲c(M)Zc(M) e entao simula MH nesta entrada decidiria (aceitaria) H′. Concluimos entao que

H′, assim como H, é recursivamente enumeravel. tambem é recursiva (respectivamente, recursivamente enumeravel). Se existe uma maquina de Turing MH que decide (aceita) H, entao uma maquina de Turing MH′ que transforma a entrada ⊲c(M) em ⊲c(M)Zc(M) e entao simula MH nesta entrada decidiria (aceitaria) H′. Concluimos entao que H′, assim como H, é recursivamente enumeravel. Com o resultado acima, para mostrarmos que H nao é recursiva, basta mostrarmos que H′ nao é recursiva. Entretanto, se H′ fosse recursiva, como a classe de linguagens recursivas é fechada por complemento, /H′ tambem seria recursiva. Entretanto, podemos mostrar que /H′ nao é sequer recursivamente enumeravel. Suponhamos, por contradiçao, que /H′ é recursivamente enumeravel. Entao existe uma maquina de Turing N que aceita /H′. Suponha que c(N) ∈ /H′. Isto implica, pela definiçao de /H′, que

N nao para com entrada c(N). (1)

Por outro lado, /H′ é a linguagem aceita por N. Logo, se c(N) pertence a linguagem aceita por N, entao, pela definiçao de aceitaçao por uma Maquina de Turing,

a maquina N para com entrada c(N). (2)

As afirmativas (1) e (2) formam uma contradiçao. Logo, nao é possivel que c(N) ∈ /H′. Assim, vamos supor entao que c(N) !∈ /H′, ou seja, que c(N) ∈ H′. Isto implica, pela definiçao de H′, que

N para com entrada c(N). (3)

Por outro lado, recapitulando, /H′ é a linguagem aceita por N. Assim, se c(N) nao pertence a linguagem aceita por N, entao, pela definiçao de aceitaçao por uma Maquina de Turing,

a maquina N nao para com entrada c(N). (4)

As afirmativas (3) e (4) formam novamente uma contradiçao. Logo, nao é possivel que c(N) !∈ /H′. Se nao é possivel que c(N) ∈ /H′ e tambem nao é possivel que c(N) !∈ /H′, concluimos entao que a maquina N que aceitaria /H′ nao pode existir. Isto significa que /H′ nao é recursivamente enumeravel, logo H′ tambem nao é recursiva. Consequentemente, H tambem nao é recursiva. Desta forma, existem linguagens que sao recursivamente enumeraveis mas nao sao recursivas (H e H′ exemplificam isto), a classe das linguagens recursivamente enumeraveis nao é fechada por complemento (H′ e H′ exemplificam isto) e existem problemas que nao podem ser algoritmicamente resolvidos, como o algoritmo P exemplifica, pois a existencia de P é equivalente a H ser uma linguagem recursiva. Portanto, é impossivel determinar algoritmicamente se um algoritmo ira ou nao parar com uma dada entrada. Assim, o Problema da Parada, definido acima, é um problema indecidivel.

Classes de Complexidade e a Hierarquia das Classes

Uma maquina de Turing é dita polinomialmente limitada se ha uma funçao polinomial p(n) tal que, para toda entrada w ∈ Σ\*0, qualquer computaçao a partir do estado inicial da maquina sempre alcança um estado de parada em, no maximo, p(|w|) passos.

A classe P é definida como a classe de todas as linguagens que podem ser decididas por uma maquina de Turing deterministica polinomialmente limitada.

Analogamente, a classe NP é definida como a classe de todas as linguagens que podem ser decididas por uma maquina de Turing nao-deterministica polinomialmente limitada.

P ⊆ NP.

Este resultado segue diretamente do fato de que toda Maquina de Turing deterministica pode ser considerada como um caso particular de uma Maquina de Turing nao-deterministica.

Uma maquina de Turing é dita exponencialmente limitada se ha uma constante inteira c > 1 e uma funçao polinomial p(n) tal que, para toda entrada w ∈ Σ\*0, qualquer computaçao a partir do estado inicial da maquina sempre alcança um estado de parada em, no maximo, cp(|w|) passos.

A classe EXPTIME é definida como a classe de todas as linguagens que podem ser decididas por uma maquina de Turing deterministica exponencialmente limitada.

A classe NEXPTIME é definida como a classe de todas as linguagens que podem ser decididas por uma maquina de Turing nao-deterministica exponencialmente limitada.

EXPTIME ⊆ NEXPTIME.

A prova deste resultado é inteiramente analoga a prova de que P ⊆ NP.

NP ⊆ EXPTIME.

Como vimos anteriormente, uma Maquina de Turing nao-deterministica pode ser transformada em uma Maquina de Turing deterministica. Entretanto, esta transformaçao tem um preço. A Maquina de Turing deterministica pode precisar executar um numero exponencialmente maior de passos do que a Maquina de Turing nao-deterministica original. Assim, ao transformarmos uma Maquina de Turing nao-deterministica polinomialmente limitada em uma Maquina de Turing deterministica, devido a este aumento exponencial do numero de passos executado pela maquina, poderemos obter uma Maquina de Turing deterministica exponencialmente limitada. Logo, NP ⊆ EXPTIME.

Temos entao a seguinte hierarquia entre as classes de complexidade estudadas

ate o momento:

P ⊆ NP ⊆ EXPTIME ⊆ NEXPTIME.

Uma maquina de Turing é dita espacialmente polinomialmente limitada se ha um polinomio p(n) tal que, para toda entrada w ∈ Σ\*0, qualquer computaçao a partir do estado inicial da maquina sempre alcança um estado de parada tendo visitado, no maximo, p(|w|) casas distintas da fita da maquina.

A classe PSPACE é definida como a classe de todas as linguagens que podem ser decididas por uma maquina de Turing deterministica espacialmente polinomialmente limitada.

A classe NPSPACE é definida como a classe de todas as linguagens que podem ser decididas por uma maquina de Turing nao-deterministica espacialmente polinomialmente limitada.

Da mesma maneira que nos casos anteriores em que P ⊆ NP e EXPTIME ⊆ NEXPTIME, temos que PSPACE ⊆ NPSPACE. Entretanto, enquanto nos casos de P e NP e de EXPTIME e NEXPTIME nao ha nenhum resultado conclusivo que nos diga se as inclusoes de P em NP e de EXPTIME em NEXPTIME sao proprias ou nao, conforme discutiremos mais adiante, no

caso das classes PSPACE e NPSPACE tal resultado conclusivo existe. Ele é conhecido como Teorema de Savitch:

PSPACE = NPSPACE.

O que o teorema acima nos diz é que, enquanto a transformaçao de uma Maquina de Turing nao-deterministica em uma Maquina de Turing deterministica pode acarretar em um aumento exponencial do numero de passos que a maquina precisara executar, este aumento exponencial nao ocorre com relaçao a quantidade de memoria (quantidade de fita) que a maquina ira utilizar. Assim, a quantidade de fita que a maquina deterministica ira utilizar é apenas polinomialmente maior do que a quantidade de fita utilizada pela maquina nao-deterministica original.

P ⊆ PSPACE.

A cada passo da computaçao, uma Maquina de Turing visita no maximo uma casa nova (uma casa que ainda nao foi visitada) da fita. Entao, se a maquina é polinomialmente limitada e executa no maximo f(|w|) passos, entao ela visita no maximo f(|w|) + 1 casas distintas da fita, o que significa que ela tambem é espacialmente polinomialmente limitada. A partir desta implicaçao, temos que P ⊆ PSPACE.

Com o mesmo argumento acima, podemos concluir tambem que NP ⊆ NPSPACE. Mas como NPSPACE = PSPACE, pelo Teorema de Savitch, temos entao o resultado a seguir.

NP ⊆ PSPACE.

PSPACE ⊆ EXPTIME.

Seja L ∈ PSPACE. Entao, existe uma Maquina de Turing deterministica M = (Σ0, Σ,Q, q0, F, δ) que decide L visitando no maximo f(|w|) casas distintas da fita, para todo w ∈ Σ\*0 (maquina espacialmente polinomialmente limitada). Vamos analisar quantas configuraçoes diferentes esta maquina pode apresentar durante a computaçao com uma dada entrada w ∈ Σ\*0. Suponha que |Q| = n e || = m. Como a maquina é espacialmente polinomialmente limitado, o cabeçote esta restrito a uma regiao da fita formada por f(|w|) casas. Temos entao que, em um dado momento, a maquina pode estar em um de seus n estados, com o seu cabeçote posicionado em uma das f(|w|) casas da fita e com um de m possiveis simbolos do alfabeto em cada uma das f(|w|) casas da fita. Assim, existe um total de n × f(|w|) × mf(|w|) possiveis configuraçoes em que a maquina pode estar ao longo da computaçao.A maquina nao pode repetir uma configuraçao durante a sua computaçao, pois isto a faria entrar em um loop, ja que ela é deterministica. Mas a maquina nao pode entrar em loop, ja que ela é um decisor. Assim, esta maquina executa no maximo n × f(|w|) × mf(|w|) passos durante a computaçao com a entrada w. Portanto, a maquina é exponencialmente limitada, o que significa que L ∈ EXPTIME.

Novamente, apresentamos a hierarquia entre as classes de complexidade

estudadas ate o momento:

P ⊆ NP ⊆ PSPACE ⊆ EXPTIME ⊆ NEXPTIME.

Reduçao Polinomial, Diﬁculdade e Completude e o Problema P vs NP

Dizemos que uma funçao f : Σ\*0 → Σ\*0 é polinomialmente computavel se existe uma maquina de Turing deterministica polinomialmente limitada que a computa.

Considere duas linguagens L1,L2 ⊆ Σ\*0. Dizemos que existe uma reduçao polinomial de L1 para L2, denotada por L1 ≤P L2, se existe uma funçao polinomialmente computavel f : Σ\* → Σ\* tal que w ∈ L1 se e somente se f(w) ∈ L2.

Seja C uma classe de complexidade e L uma linguagem. Dizemos que L é C-Dificil se, para toda linguagem L′ ∈ C, existe uma reduçao polinomial de L′ para L. Desta forma, L ser C-Dificil significa que L é ao menos tao complexa quanto qualquer linguagem de C.

Seja C uma classe de complexidade e L uma linguagem.Dizemos que L é C-Completa se:

1. L é C-Dificil e

2. L ∈ C.

Desta forma, uma linguagem C-Completa é uma linguagem de C que é ao menos tao complexa quanto qualquer linguagem de C, o que nos permite pensar nas linguagens C-Completas como as linguagens mais complexas dentro da classe C, ou o “limite superior” de complexidade dentro da classe.

Acredita-se fortemente que P != NP, mas isto nao foi provado. Um indicio forte da validade desta desigualdade vem do estudo dos problemas NP-Completos.

P = NP se e somente se existe uma linguagem L NPCompleta tal que L ∈ P.

Suponha que P = NP e que L é uma linguagem NPCompleta.

Queremos mostrar que L ∈ P. Pela condiçao (2) da definiçao 23, se L é NP-Completa, entao L ∈ NP. Mas como estamos assumindo a hipotese de que NP = P, entao L ∈ P, como queriamos mostrar.

Seja L uma linguagem NP-Completa tal que L ∈ P. Queremos mostrar que P = NP. Como L ∈ P, L pode ser decidida por uma maquina de Turing deterministica polinomialmente limitada M. Por outro lado, pela condiçao (1) da definiçao 23, se L é NP-Completa, entao L é NP-Dificil.

Por sua vez, se L é NP-Dificil, entao, para toda linguagem L′ ∈ NP, existe uma reduçao polinomial de L′ para L, que pode ser computada por uma maquina de Turing deterministica polinomialmente limitada M′. Mas entao a composiçao das maquinas de Turing M′ e M tambem resulta em uma maquina de Turing deterministica polinomialmente limitada. Esta maquina

decide L′, o que significa que L′ ∈ P. Como esta argumentaçao vale para qualquer L′ ∈ NP, temos entao que P = NP.

O primeiro problema que foi determinado como sendo NP-Completo foi o problema de determinar de uma dada formula da logica proposicional é satisfativel, no teorema de Cook. A partir de entao, atraves do uso de reduçoes polinomiais, muitos outros problemas foram caracterizados como NP-Completos. Ate hoje, nenhum algoritmo polinomial foi descoberto para

nenhum dos problemas NP-Completos, o que nos faz acreditar fortemente que os problemas NP-Completos nao estao em P, o que é equivalente a P != NP.

Comentarios adicionais:

Sim, a partir de problema da parada foi so crtl+c ctrl+v

Eu nao gosto muito de acentos, deal with it

e Menasche, vai tomar no cu!

Bjs e boa prova pra seja la quem estiver fudido o suficiente pra estar lendo um resumo meu.