Complexidade de Algoritmos I

Prof. Pedro J. de Rezende

2º Semestre de 2002

Relações de Recorrência*

1 Introdução

Intuitivamente, a partir de uma demonstração por indução pode-se extrair um algoritmo recursivo e, da descrição deste assim obtida, pode-se facilmente imaginar que nasce imediatamente uma expressão recursiva para a função de complexidade do algoritmo. O objeto de estudo dentro do corrente tópico é justamente tais funções.

Chamamos de *relação de recorrência* a uma expressão recursiva para definição uma função. Estaremos especialmente estudando formas de se encontrarem soluções (i.é., fórmulas fechadas) para relações de recorrência.

O exemplo clássico de relação de recorrência, que aprendemos desde o segundo grau, é a fórmula de Fibonacci:

$$F(n) = F(n-1) + F(n-2), F(1) = 1, F(0) = 0.$$

Para qualquer valor de $n \geq 1$, podemos calcular a partir desta expressão, o valor da função $F(n)$. Por exemplo, $F(3) = F(2) + F(1) = (F(1) + F(0)) + F(1) = (1 + 0) + 1 = 2$, $F(4) = F(3) + F(2) = 2 + 1 = 3$ e assim por diante.

Dada sua natureza recursiva, toda relação de recorrência tem uma ou mais **condições de parada** (da recorrência) sem as quais não é possível calcular seus valores. Para a fórmula de Fibonacci, as condições de parada são $F(1) = 1, F(0) = 0$.

Uma fórmula que pode ser usada para definir uma relação de recorrência $T(n)$ qualquer é a seguinte:

$$T(n) = f((T(1), T(2), \ldots, T(n-1), n),$$

onde f é uma função de n parâmetros onde, para calcular $T(n)$, é necessário conhecer o valor de $T(n')$ para todos os $n' < n$. Daqui por diante, denotaremos por T, com freqüência, a função que descreve o tempo despendido por um algoritmo.

2 Revisão de Somatórios

Quando um algoritmo apresenta construções iterativas (como laços) ou chamadas recursivas, sua complexidade pode ser expressa como a soma dos

*Escriba: Fernando José Castor de Lima Filho.

tempos gastos em cada iteração ou em cada execução do procedimento recursivo. Por exemplo: o laço principal do algoritmo *MergeSort* é responsável por combinar duas seqüências ordenadas em uma só seqüência ordenada. Esse laço realiza, no pior caso, n operações de comparação. Além disso, ele é executado cada vez que o procedimento recursivo *mergeSort* é chamado, o que ocorre $\log_2 n$ vezes, para uma entrada de tamanho n. O número de comparações realizadas por este algoritmo é dado pelo seguinte somatório:

$$\sum_{i=1}^{\log_2 n} n.$$

Não é possível, porém, comparar este resultado com outras funções na forma em que ele está. Para tanto, é necessário torná-lo mais parecido com as funções com as quais estamos habituados a trabalhar. Para este somatório extremamente simples, o resultado ou *forma fechada* é a expressão $n \log_2 n$. Nesta seção apresentamos alguns exemplos mais interessantes de somatórios e suas formas fechadas.

2.1 Propriedades dos Somatórios

Dada uma seqüência $a_1, a_2, ...$ de números, a soma finita $a_1 + a_2 + ... + a_n$ pode ser escrita como

$$\sum_{k=1}^{n} a_k.$$

Se $n = 0$, o valor da soma é definido como 0. Se n não é um inteiro, supomos que o limite superior é $\lfloor n \rfloor$. O valor da série finita é sempre bem definido e seus termos podem ser somados em qualquer ordem.

Dada uma seqüência $a_1, a_2, ...$ de números, a soma infinita $a_1 + a_2 + ...$ pode ser escrita como

$$\sum_{k=1}^{\infty} a_k$$

e é interpretada como

$$\lim_{n \to \infty} \sum_{k=1}^{\infty} a_k.$$

Se o limite não existe, dizemos que a série diverge; caso contrário, que ela converge.

Somatórios apresentam a propriedade da linearidade, ou seja, para qualquer número real c e quaisquer seqüências finitas $a_1, a_2, ..., a_n$ e $b_1, b_2, ..., b_n$,

$$\sum_{k=1}^{n} (ca_k + b_k) = c \sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k.$$

Esta propriedade também é válida para séries infinitas convergentes.

Em decorrência da propriedade de linearidade, é possível colocar em evidência termos multiplicativos que sejam independentes do somatório. Por exemplo:

$$\sum_{k=1}^{n}(f(x)a_k) = f(x)\sum_{k=1}^{n}(a_k),$$

onde $f(x)$ é uma função independente de k. Manipulação similar pode ser feita com somatórios que incorporam notação assintótica. Por exemplo:

$$\sum_{k=1}^{n}\Theta(f(k)) = \Theta(\sum_{k=1}^{n}f(k)).$$

Nesta equação, a notação Θ no lado esquerdo se aplica à variável k. No lado direito, porém, ela se aplica a n. Manipulações desta natureza também podem ser aplicadas a séries infinitas convergentes.

2.2 Exemplos de Somatórios

Saber encontrar a forma fechada de um somatório é muito importante para a resolução de relações de recorrência. Uma das técnicas que apresentamos a seguir faz uso do fato de que relações de recorrência podem ser reescritas como somatórios. Nesta seção, apresentamos alguns somatórios encontrados com freqüência na análise de algoritmos, junto com suas respectivas formas fechadas. O leitor interessado poderá encontrar uma rica discussão das técnicas em [2, 1].

2.2.1 Série Aritmética

Uma seqüência $a_1 + a_2 + a_3 + ... + a_n$ tal que $a_k - a_{k-1} = c$ para $1 < k \leq n$ e c constante é chamada de *progressão aritmética*. Sua soma $\sum_{k=1}^{n} a_k$ é chamada de *série aritmética* e sua forma fechada é dada por:

$$\sum_{k=1}^{n} a_k = \frac{n(a_n + a_1)}{2}.$$

2.2.2 Série Geométrica

Uma seqüência $a_1 + a_2 + a_3 + ... + a_n$ tal que $a_k/a_{k-1} = c$ para $1 < k \leq n$ e c constante é chamada de *progressão geométrica*. Sua soma $\sum_{k=1}^{n} a_k$ é chamada de *série geométrica* e sua forma fechada é dada por:

$$\sum_{k=1}^{n} a_k = a_1\frac{c^{n+1} - 1}{c - 1}.$$

Quando a série é infinita e $|c| < 1$, temos uma série geométrica infinita decrescente cuja soma é dada por:

$$\sum_{k=1}^{\infty} a_k = \frac{a_1}{1 - c}.$$

2.2.3 Série Harmônica

$$\sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n),$$

onde $\gamma = 0,577...$ é a *constante de Euler*.

2.2.4 Soma de Logaritmos

$$\sum_{k=1}^{n} \lfloor \log_2 k \rfloor = (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 \in \Theta(n \log n).$$

2.2.5 Produtos

O produto finito $a_1 \cdot a_2 \cdot \ldots \cdot a_n$ é denotado por $\prod_{k=1}^{n} a_k$. Se $n = 0$, o valor do produto é definido como 1. Às vezes é conveniente converter uma fórmula contendo um produto em uma fórmula com um somatório usando a identidade: $\log\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \log a_k$.

2.2.6 Somatórios Interessantes

Exemplo 1

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}.$$

Exemplo 2

$$\sum_{k=1}^{n} k2^k = (n-1)2^{n+1} + 2.$$

Exemplo 3

$$\sum_{k=1}^{n} k2^{n-k} = 2^{n+1} - 2 - n.$$

3 Técnicas para a resolução de relações de recorrência

3.1 Expansão Telescópica

Como exemplo de um algoritmo cuja complexidade de tempo pode facilmente ser expressa através de uma relação de recorrência, considere o que realiza busca binária em um vetor ordenado. Esse algoritmo usa o seguinte procedimento recursivo: sejam V um vetor com n elementos ordenado crescentemente e x um valor que desejamos encontrar em V. Começamos por comparar x com o elemento $V(m)$ que se encontra na posição do meio ($m = (n+1)/2$) do vetor.

Se x for igual a $V(m)$, então m é o índice procurado e a busca termina. Se x for menor que $V(m)$, o elemento procurado só pode se encontrar à esquerda de $V(m)$, i.é., na primeira metade do vetor. Podemos considerar, então, que o vetor no qual estamos realizando a busca passa a ser aquele que vai do primeiro elemento de V ao elemento imediatamente anterior a m e podemos repetir o mesmo procedimento recursivamente para esse vetor menor. Similarmente, para o caso de x ser maior que $V(m)$.

Analisando o comportamento deste algoritmo, percebemos que ele realiza uma comparação e, logo em seguida, passa a agir sobre um vetor que tem metade do tamanho de V. O resultado da comparação determina se a metade de V na qual a busca prosseguirá será a primeira ou a segunda. A complexidade de comparações deste algoritmo é descrita, então, pela seguinte relação de recorrência:

$$T(n) = T\left(\lfloor n/2 \rfloor\right) + 1. \tag{1}$$

De um modo geral, se um algoritmo é definido de maneira recursiva, então podemos expressar sua complexidade em termos de uma relação de recorrência.

Quando comparamos a eficiência de algoritmos, normalmente estamos interessados em comparar as funções que descrevem suas complexidades analisando-as de maneira assintótica (utilizando, portanto, as classes o, O, Θ, Ω e ω). Se uma função $T(n)$ se encontra, porém, na forma de uma relação de recorrência, não é possível que se a compare com outras funções conhecidas.

Para resolver esta dificuldade, temos que encontrar uma forma fechada para $T(n)$, de modo que possamos compará-la com outras funções conhecidas. A forma fechada de uma relação de recorrência também é chamada de *solução* da recorrência.

Voltemos ao exemplo anterior. Ignorando, por enquanto, pisos e tetos, isto é, assumindo inicialmente que n é uma potência de 2, $n = 2^k$, vamos considerar que $T(n/2^k) = T(1) = 1$ e façamos a expansão da relação de recorrência:

$$\begin{aligned}
T(n) &= T(\tfrac{n}{2}) + 1 \\
&= (T(\tfrac{n}{4}) + 1) + 1 \\
&= ((T(\tfrac{n}{8}) + 1) + 1) + 1 \\
&= \ldots \\
&= (\ldots((T\left(\tfrac{n}{2^k}\right) \underbrace{+1) + 1) + \ldots + 1) + 1}_{k-1 \text{ parcelas}} \\
&= \sum_{i=1}^{k} 1 \\
&= k = \log_2 n \in O(\log n)
\end{aligned}$$

A idéia básica da técnica apresentada é expandir a relação de recorrência até que possa ser detectado o seu comportamento no caso geral. Esta técnica para fechar relações de recorrência é chamada de *Expansão Telescópica*. A rigor,

deveríamos fazer uma demonstração indutiva da validade da 5ª igualdade acima, mas a omitiremos.

Convém frisar que trocar o termo '1' por uma constante c qualquer na relação de recorrência (1) acarretaria apenas na multiplicação do resultado por essa constante. Neste caso, a solução da recorrência passaria a ser $O(c \log_2 n)$.

Apesar de interessante como um primeiro exemplo, a relação de recorrência apresentada em (1) é demasiado simples. Sua resolução se resume em achar o número de parcelas da soma resultante da expansão telescópica. São comuns, porém, casos em que há expressões mais complexas na expansão da relação de recorrência, ao invés de apenas termos simples. Nessas situações, fechar o somatório pode ser bem mais difícil. Tomemos como exemplo a seguinte recorrência:

$$T(n) = T(n/2) + n,\ T(1) = ?. \tag{2}$$

Na expressão acima, o valor da base da recorrência foi deixado em aberto temporariamente. Fizemos isso porque a fixação posterior do valor da condição de parada (ou base) pode nos ajudar a fechar mais facilmente o somatório resultante da expansão telescópica. Esse tipo de artifício pode ser usado sem maiores preocupações, já que alterar a base de uma relação de recorrência modifica sua solução em apenas uma constante aditiva.

Podemos expandir a relação de recorrência (2) da seguinte maneira:

$$\begin{aligned}
T(n) &= T(n/2) + n \\
&= (T(n/4) + n/2) + n \\
&= ((T(n/8) + n/4) + n/2) + n \\
&= \ldots \\
&= (\ldots((T(n/2^k) + n/2^{k-1}) + n/2^{k-2}) + \ldots + n/2) + n \\
&= T(1) + 2 + 4 + 8 + 16 + \ldots + 2^k
\end{aligned}$$

Pela série gerada a partir da expansão da recorrência, chegamos à conclusão de que um bom valor para $T(1)$ é 1, já que isso facilita o fechamento do somatório.

$$\begin{aligned}
T(n) &= 1 + 2 + 4 + 8 + 16 + \ldots + 2^k \\
&= \textstyle\sum_{i=0}^{k} 2^i \\
&= \textstyle\sum_{i=0}^{\log_2 n} 2^i \\
&= 2^{(\log_2 n)+1} - 1 \\
&= 2n - 1
\end{aligned}$$

Em princípio, todas as relações de recorrência podem ser resolvidas através de expansão telescópica. Tudo o que é preciso é a devida criatividade para transformar a recorrência em um somatório conveniente.

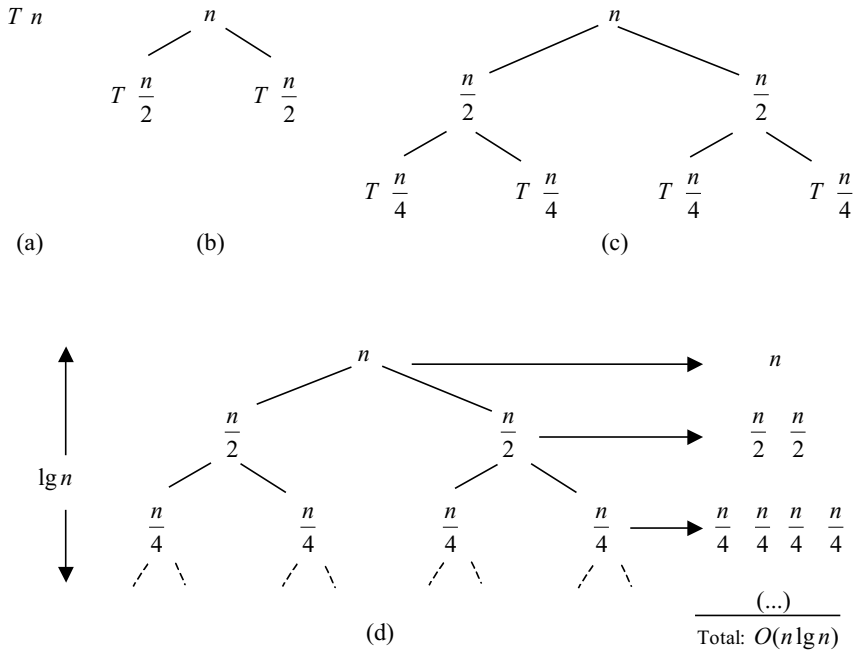

Figura 1: Árvore de Recorrência para a expressão $T(n) = 2T(\frac{n}{2}) + n$

Vamos tentar encontrar a solução para a relação de recorrência que descreve o comportamento do algoritmo *MergeSort*[2]:

$$T(n) = 2T(n/2) + n,\ T(1) =?. \tag{3}$$

Dessa vez, porém, usaremos uma técnica um pouco diferente que consiste de uma representação gráfica para melhor visualizar as etapas da expansão. Essa representação é conhecida como *Árvore de Recorrência*. A árvore de recorrência para a expressão 3 é apresentada na figura 1.

Através desta representação, torna-se muito fácil ver o que foi gerado pela expansão a cada passo. Na figura 1, por exemplo, podemos perceber que cada expansão da recorrência acrescenta n ao resultado final. Conseqüentemente, para resolvermos a recorrência, é necessário apenas achar a altura da árvore, que neste caso é $\log_2 n$. Portanto, a solução da recorrência é dada pelo somatório $\sum_{i=1}^{\log n} n = n\log n$.

3.2 Método da Substituição

Conforme dissemos na seção anterior, o método da expansão telescópica permite que resolvamos qualquer relação de recorrência. Apesar disso, nem

sempre ele leva a uma solução simples, pois podem ser muito difíceis as manipulações algébricas necessárias para se obter uma forma fechada para algumas recorrências. Nesta seção, apresentamos um método que nos permite achar a solução de uma recorrência de maneira mais fácil, exigindo apenas um conhecimento básico de indução matemática.

Indução matemática é uma técnica de demonstração bastante poderosa, como já foi visto em [3]. Façamos uma rápida revisão da idéia básica da indução forte.

Seja T um teorema que se quer provar. Suponha que T inclui um parâmetro n que pode ter como valor qualquer número natural. Ao invés de provar diretamente que T é verdade para todos os valores de n, provamos as seguintes condições:

1. T é verdade para $n = n_0$.
2. Para $n > n_0$, se T é verdade para todo $n' < n$, então T é verdade para n.

Pelo princípio da indução forte, estas condições acarretam na veracidade do teorema T para todos os valores naturais do parâmetro n, a partir de n_0.

Relações de recorrência e provas por indução estão fortemente ligadas. Nas duas afirmações acima, percebemos a estrutura de uma relação de recorrência: a primeira pode ser vista como a condição de parada da recorrência enquanto a segunda está bastante ligada à idéia de uma função que é definida em termos de seu próprio valor para entradas menores. Dada essa semelhança, é natural supor que provas por indução sejam uma bom meio para se a validade de soluções de relações de recorrência.

Vamos, então, provar por indução que a recorrência:

$$T(n) = 2T(n/2) + n,\ T(1) =? \tag{4}$$

pertence a $O(n \log n)$, ou seja, vamos mostrar que existem constantes $c \in \mathbb{N}$ e $n_0 \in \mathbb{N}$ tais que $T(n) \leq cn \log n$, para todo $n \geq n_0$.

A única diferença da expressão (4) e da expressão (3) é o fato de estarmos usando o piso de $n/2$, ao invés da fração pura.[2]

Para realizar a prova, usaremos como hipótese o caso $\lfloor n/2 \rfloor$. A hipótese nos diz que $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$. Por enquanto, deixaremos a base em aberto, pois esta será a condição de parada da recorrência. A partir da hipótese, podemos afirmar que

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \leq 2c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor + n$$

Como, pela hipótese de indução, temos que $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$, então $T(\lfloor n/2 \rfloor)$ também é menor que $c(n/2)\log(n/2)$. Conseqüentemente, podemos

[2] O fato (verdadeiro) de que na maioria dos casos, pisos e tetos afetam em muito pouco o resultado de uma relação de recorrência (e podem ser ignorados) pode ser demonstrado por indução, mas não faremos isso nestas notas.

remover os pisos da expressão acima, o que nos leva à seguinte desigualdade:

$$\begin{aligned} T(n) &\leq 2c(n/2)\log(n/2) + n \\ &= cn(\log n - \log 2) + n \\ &= cn\log n - cn + n \\ &\leq cn\log n \end{aligned}$$

A última linha do desenvolvimento acima é verdade para todo $c \geq 1$.

Para completar nossa prova, falta apenas encontrarmos a base da indução. A intuição comum é considerar $T(1) = 1$. Se usássemos essa idéia, porém, teríamos a desigualdade

$$\begin{aligned} T(1) &\leq c1\log 1 \\ 1 &\leq 0 \end{aligned}$$

que obviamente é falsa. Esta dificuldade pode ser facilmente contornada graças ao fato de a notação assintótica exigir que $T(n)$ seja menor ou igual a $cn\log n$ apenas para $n \geq n_0$, onde n_0 é uma constante. Se usarmos $T(2)$ e $T(3)$ como bases, a recorrência não dependerá mais do valor de $T(1)$ diretamente, para $n > 3$. A partir da expressão (4), calculamos que $T(2) = 4$ e $T(3) = 5$. Para terminar a prova, falta apenas escolher um c de valor grande o suficiente para que $T(2) \leq 2c\log 2$ e $T(3) \leq 3c\log 3$. É simples verificar que qualquer escolha de $c \geq 2$ satisfaz as duas inequações.

A prova que acabamos de realizar só foi possível porque já tínhamos uma idéia sobre a solução da recorrência. Esta é exatamente a desvantagem do método da substituição. Só vale a pena usá-lo quando se tem uma idéia razoável sobre a solução da relação de recorrência em questão. Para o exemplo que acabamos de apresentar, um palpite muito baixo nos impediria de completar a prova enquanto um palpite muito alto nos daria um limite correto, mas muito distante da solução da recorrência. Felizmente, há algumas heurísticas que podem nos ajudar, na hora de tentar um palpite.

Se uma recorrência é muito parecida com outra que já foi vista, é razoável supor que sua solução também seja parecida. Por exemplo, as recorrências (3) e (4) são ambas pertencentes a $\Theta(n\log n)$ (deixamos a prova de que ambas são pertencentes a $\Omega(n\log n)$ como exercício para o leitor). O mesmo vale para a recorrência

$$T(n) = 2T(\lfloor n/2 \rfloor + 37) + n. \tag{5}$$

Intuitivamente, isso é verdade porque 37 é uma constante e se fizermos n suficientemente grande, $\lfloor n/2 \rfloor + 37$ se torna muito próximo de $\lfloor n/2 \rfloor$.

Similarmente, a recorrência

$$T(n) = 2T(\lfloor n/2 \rfloor + 37) + 41n + 101 \tag{6}$$

também tem solução pertencente a $\Theta(n\log n)$. A constante 41 multiplicando o termo n acrescenta apenas uma constante multiplicativa à solução. O termo 101 resulta na adição de uma parcela da ordem de n ao resultado da recorrência

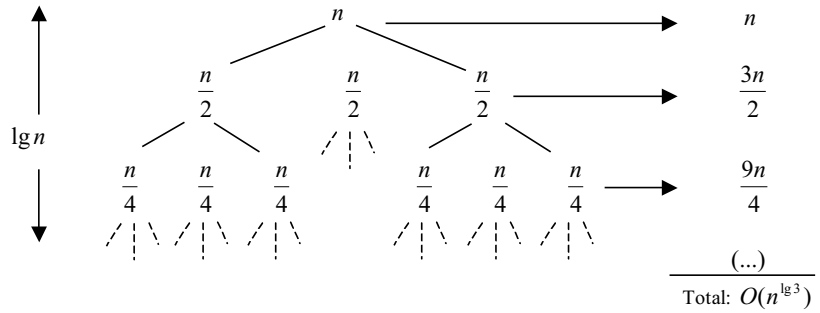

Figura 2: Árvore de Recorrência para a expressão $T(n) = 3T(n/2) + n$

e também não influi na classe da solução, dado que esta pertence a $\Theta(n \log n)$ e $n \in o(n \log n)$. De maneira mais abrangente, podemos afirmar que acrescentar termos de menor ordem aditivamente não deve mudar a intuição quanto ao resultado de uma relação de recorrência.

A situação muda, entretanto, quando acrescentamos termos multiplicativamente. A relação de recorrência

$$T(n) = 3T(\lfloor n/2 \rfloor) + n, \tag{7}$$

apesar de muito similar às últimas apresentadas, não tem solução pertencente a $\Theta(n \log n)$. Olhando para a árvore de recorrência correspondente a essa relação, na figura 2, podemos perceber que a soma dos termos de cada passo resulta em um valor maior que n por um fator multiplicativo que não é constante. De fato, a expansão telescópica da relação de recorrência (7) tem solução pertencente a $O(n^{\log_2 3})$.

Semelhantemente, alterar o termo pelo qual n é dividido a cada passo também modifica a solução da recorrência. Uma árvore apresentando a expansão da relação de recorrência

$$T(n) = 2T(\lfloor n/3 \rfloor) + n \tag{8}$$

é apresentada na figura 3. Convidamos o leitor a provar que a solução dessa relação de recorrência pertence a $O(n)$.

Para uma recorrência com a forma $T(n) = aT(\lfloor n/b \rfloor) + n$, se a for igual a b, ocorrerá que no i-ésimo passo da expansão da recorrência (com $i \geq 0$), teremos a soma de a^i termos iguais a n/b^i. Como $a = b$, a expressão resultante do i-ésimo passo será $a^i n/a^i = n$. A árvore resultante de uma expansão como essa tem altura $\log_b n$ e, conseqüentemente, a solução da recorrência pertence a $O(n \log n)$. Na próxima seção, apresentaremos um resultado mais geral que pode ser usado para resolver diversos tipos de relação de recorrência.

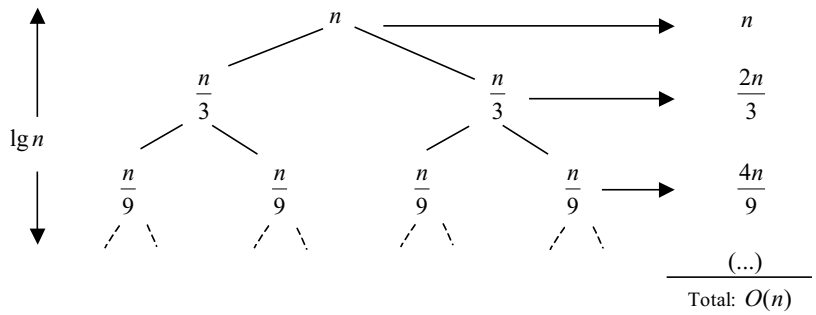

Figura 3: Árvore de Recorrência para a expressão $T(n) = 2T(n/3) + n$

Há situações em que, apesar de se ter uma conjectura sobre o limite assintótico, a demonstração de sua correção para o caso geral pode ser matematicamente laboriosa pois faz-se necessário que a expressão conjecturada seja precisa ao detalhe. Normalmente, nestes casos, o problema é que a suposição indutiva não é forte o suficiente para provar uma solução mais detalhada e torna-se necessário ajustar o palpite através da subtração de algum termo de menor ordem.

Considere a recorrência

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1. \tag{9}$$

Podemos ter o palpite de que a solução pertence a $O(n)$ e podemos tentar mostrar que $T(n) \leq cn$, para uma escolha apropriada da constante c. Substituindo nosso palpite na expressão (9), obtemos

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1, \end{aligned}$$

que não implica que $T(n) \leq cn$, qualquer que seja a escolha de c. A idéia que surge imediatamente neste caso é tentar um palpite maior, por exemplo, $T(n) \in O(n^2)$. Um palpite como esse funcionaria, mas não seria necessário pois o nosso palpite inicial está quase correto, necessitando apenas de um pequeno ajuste.

Nosso palpite se mostrou grande demais por um termo de menor ordem (nesse caso, uma constante). Resolvemos o problema então subtraindo de nosso palpite um termo da mesma ordem de grandeza do termo excedente. Nosso novo palpite é, então, $T(n) \leq cn - b$, onde $b \geq 0$ é constante. Agora temos:

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b, \end{aligned}$$

para $b \geq 1$.

Pode ser inicialmente contra-intuitivo considerar que se deve *subtrair* um termo (ainda que de menor ordem). Afinal de contas, se o algebrismo não funciona, não deveríamos tentar aumentar nosso palpite? A chave para entender este passo é lembrar que estamos usando indução matemática: podemos provar algo mais apertado para um dado valor através da suposição de algo mais apertado para valores menores.

É fácil errar quando usamos notação assintótica. Por exemplo, podemos tentar provar que a relação de recorrência (4) pertence a $O(n)$ usando o palpite de que $T(n) \leq cn$ e argumentando

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \in O(n), \end{aligned}$$

uma vez que c é uma constante. O erro neste caso é que não provamos a forma exata da hipótese indutiva, isto é, não provamos que $T(n) \leq cn$. Começamos tentando provar uma coisa (que $T(n) \leq cn$) mas saltamos para uma conclusão alternativa que não corresponde ao que pretendíamos demonstrar. Note que $T(n) \leq cn$ implica que $T(n) \in O(n)$, mas, por outro lado, $T(n) \leq cn + n$ e $cn + n \in O(n)$ *não* implicam que $T(n) \in O(n)$.

Algumas vezes, um pouco de manipulação algébrica pode fazer uma recorrência aparentemente desconhecida tomar um aspecto mais familiar. Como exemplo, considere a seguinte relação de recorrência:

$$T(n) = T(\lfloor \sqrt{n} \rfloor) + \log_2 n.$$

Embora pareça complicada, essa recorrência pode ser simplificada através de uma troca de variáveis. Por conveniência, não vamos nos preocupar em arredondar valores. Renomeando $\log_2 n$ para m, obtemos

$$T(2^m) = 2T(2^{m/2}) + m.$$

Podemos então substituir $T(2^m)$ por $S(m)$ e produzir a nova recorrência

$$S(m) = 2S(m/2) + m,$$

que é muito parecida com a recorrência (4) e tem a mesma solução: $S(m) \in O(m \log m)$. Mudando de volta de $S(m)$ para $T(2^m) = T(n)$, obtemos que $T(n) \in O(\log n \log \log n)$.

3.3 O Teorema Master

Até aqui, muitas das relações de recorrência que vimos foram da forma $T(n) = aT(n/b) + f(n)$ e obtivemos soluções caso a caso. Seria interessante termos um teorema que nos permita obter, de modo simples, soluções para todas as relações de recorrência dessa forma. Isto é exatamente o que nos dá o seguinte Teorema Master:

Teorema 1 (Teorema Master[1])
Sejam $a \geq 1$ e $b \geq 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida para os inteiros não-negativos pela relação de recorrência

$$T(n) = aT(n/b) + f(n),$$

onde interpretamos que n/b significa $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Então $T(n)$ pode ser limitada assintoticamente da seguinte maneira:

1. *Se $f(n) \in O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) \in \Theta(n^{\log_b a})$*
2. *Se $f(n) \in \Theta(n^{\log_b a})$, então $T(n) \in \Theta(n^{\log_b a} \log n)$*
3. *Se $f(n) \in \Omega(n^{\log_b a + \epsilon})$, para algum $\epsilon > 0$ e se $af(n/b) \leq cf(n)$, para alguma constante $c < 1$ e para n suficientemente grande, então $T(n) \in \Theta(f(n))$*

O teorema é um resultado suficientemente poderoso para resolver grande parte das relações de recorrência vistas no decorrer desta disciplina. Convidamos o leitor sagaz a consultar em [1] a prova deste teorema. Em especial, a última parte daquela prova é interessante para que se possa melhor intuir quando pisos e tetos fazem ou não diferença no resultado de uma relação de recorrência.

Por questões didáticas, apresentaremos aqui a prova para um caso particular do teorema acima, que define soluções para um sub-conjunto expressivo das relações de recorrência resolvidas pelo Teorema Master.

Teorema 2 (Teorema 3.4 [2])
Dada uma relação de recorrência da forma $T(n) = aT(n/b) + cn^k$, onde $a, b \in \mathbb{N}, a > 1, b \geq 2$ e $c, k \in \mathbb{R}^+$,

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}), & se\ a > b^k \\ \Theta(n^k \log n), & se\ a = b^k \\ \Theta(n^k), & se\ a < b^k \end{cases}$$

Prova: Por simplicidade, assumimos[3] que $n = b^m$ de modo que n/b é sempre um inteiro.

Vamos começar expandindo a relação de recorrência:

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ &= a(aT(n/b^2) + c(n/b)^k) + cn^k \\ &= a(a(aT(n/b^3) + c(n/b^2)^k) + c(n/b)^k) + cn^k \\ &= ... \\ &= a(a(...aT(n/b^m) + c(n/b^{m-1})^k) + ...) + cn^k. \end{aligned}$$

[3]Esta prova pode também ser feita sem esta suposição, mas, com ela, evitamos inúmeros detalhes cuja inclusão aqui obscureceria aspectos interessantes da prova.

Assumindo que $T(1) = c$, ficamos com:

$$\begin{aligned} T(n) &= ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + ... + cb^{mk} \\ T(n) &= c\sum_{i=0}^{m} a^{m-i}b^{ik} \\ T(n) &= ca^m \sum_{i=0}^{m} (b^k/a)^i. \end{aligned}$$

Na última linha do desenvolvimento acima, já podemos ver os casos do enunciado do teorema, com base em como séries geométricas se comportam quando o termo b^k/a é maior, menor ou igual a zero.

Caso 1: $a > b^k$
Neste caso, o somatório $\sum_{i=0}^{m}(b^k/a)^i$ converge para uma constante. Daí temos que $T(n) \in \Theta(ca^m)$. Como $n = b^m$, então $m = \log_b n$, conseqüentemente, $T(n) \in \Theta(n^{\log_b a})$

Caso 2: $a = b^k$
Como $b^k/a = 1$, $\sum_{i=0}^{m}(b^k/a)^i = m+1$. Daí temos que $T(n) \in \Theta(ca^m m)$. Como $m = \log_b n$ e $a = b^k$, então $ca^m m = n^{\log_b a} \log_b n = n^k \log_b n$, o que nos leva à conclusão de que $T(n) \in \Theta(n^k \log_b n)$.

Caso 3: $a < b^k$
Neste caso, a série não converge quando m vai para infinito, mas é possível calcular sua soma para um número finito de termos.

$$\begin{aligned} T(n) &= ca^m \sum_{i=0}^{m} \left(b^k/a\right)^i \\ &= ca^m \left(\frac{(b^k/a)^{m+1} - 1}{(b^k/a) - 1}\right) \end{aligned}$$

Desprezando as constantes na última linha da expressão acima e sabendo que $a^m \left(\frac{(b^k/a)^{m+1}-1}{(b^k/a)-1}\right) = b^{km}$ e $b^m = n$, concluímos que $T(n) \in \Theta(n^k)$. CQD

Os resultados dos teoremas 1 e 2 se aplicam a uma grande quantidade de relações de recorrência e são muito úteis na análise de eficiência de algoritmos.

4 Exemplos de Relações de Recorrência

Nesta seção, apresentaremos as relações de recorrência que definem a complexidade de tempo de alguns algoritmos. Para manter o leitor bem situado, os exemplos serão baseados em alguns algoritmos apresentados nas referências (capítulos 4 e 5 de[2] e Seção 21.3 de [1]).

4.1 Avaliação de Polinômios

Dada uma seqüência de números reais $a_n, a_{n-1}, ..., a_1, a_0$ e um número real x, estamos interessados em avaliar o polinômio $P_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$.

O primeiro algoritmo que analisaremos parte do pressuposto de que sabemos calcular o valor do polinômio $P_{n-1}(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0$, o que nos deixa com o trabalho de calcular apenas $P_n(x) = a_n \cdot x^n + P_{n-1}(x)$. Este último passo requer n multiplicações e uma adição. Como o número de adições é pequeno comparado ao de multiplicações, nós o ignoraremos quando calcularmos a complexidade de tempo do algoritmo. Neste caso, a relação de recorrência resultante é a seguinte:

$$T(n) = T(n-1) + n.$$

A solução dessa recorrência está em $\Theta(n^2)$.

Embora esse algoritmo resolva o problema proposto, ele é muito ineficiente. É possível projetar outro mais rápido simplesmente fortalecendo nossa pré-suposição.[4] Suponhamos, então, que sabemos calcular o valor do polinômio $P_{n-1}(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0$ e que também sabemos calcular x^{n-1}. Neste caso, o trabalho que sobra é o de calcular $P_n(x) = a_n \cdot x \cdot x^{n-1} + P_{n-1}(x)$. Esta computação pode ser feita com apenas duas multiplicações e uma adição, o que já é muito melhor que as n multiplicações do algoritmo anterior. A relação de recorrência que descreve a complexidade de tempo deste segundo algoritmo é a seguinte:

$$T(n) = T(n-1) + 2.$$

A solução dessa recorrência é igual a $2n$.

É possível obter um algoritmo ainda mais rápido se, ao invés de removermos o último coeficiente do polinômio, removemos o primeiro. O problema menor se torna então o de avaliar o polinômio representado pelos coeficientes $a_n, a_{n-1}, ..., a_1, a_0$ e nossa suposição será a de que sabemos calcular $P'_{n-1}(x) = a_nx^{n-1} + a_{n-1}x^{n-2} + ... + a_1$. Neste caso, $P_n(x) = x \cdot'_{n-1}(x) + a_0$ e, conseqüentemente, temos que realizar apenas uma multiplicação e uma adição, o que nos leva à recorrência:

$$T(n) = T(n-1) + 1,$$

que tem solução igual a n.

4.2 Subgrafo Induzido Maximal

Este problema consiste em, dados um grafo não-orientado $G = (V, E)$ e um inteiro k, encontrar um subgrafo induzido $H = (U, F)$ de G de tamanho máximo tal que todos os vértices de H tenham grau $\geq k$ (em H), ou concluir que nenhum tal subgrafo induzido existe.

Para resolver este problema, pode ser usado o seguinte algoritmo indutivo baseado no parâmetro $n = |V|$, o número de vértices de G:

[4] O leitor familiarizado com demonstrações por indução já deve ter se dado conta que estamos (disfarçadamente) fortalecendo a hipótese de indução da prova (por indução) que estabelece a corretude do algoritmo aqui tratado.

Base: para $n = 1$, o grafo G tem um subgrafo induzido que satisfaz o enunciado (o próprio G) somente no caso em que $k = 0$.

Hipótese: Dados um grafo não-orientado $G = (V, E)$ e um inteiro k, sabemos encontrar um subgrafo induzido $H = (U, F)$ de G de tamanho máximo tal que todos os vértices de H tenham grau $\geq k$ (em H), ou concluir que nenhum tal subgrafo induzido existe, desde que o número de vértices de G seja $< n$.

Passo: Sejam $G = (V, E)$ um grafo não-orientado com $|V| = n$ e k um inteiro. Se $|V| < k + 1$, nenhum vértice pode ter grau $\geq$ k, conseqüentemente, não há nenhum subgrafo induzido com a propriedade desejada.

Assumamos, então, que $n \geq k + 1$. Se todos os vértices de G têm grau $\geq k$, então, basta tomarmos $H = G$. Caso contrário, selecionamos um vértice v cujo grau seja $< k$ e o removemos de G, junto com todas as suas arestas incidentes. O grafo resultante tem $n - 1$ vértices e, por hipótese de indução, sabemos resolver esse problema. CQD

Complexidade: Para analisar a complexidade deste algoritmo, é necessário formalizar um pouco mais a representação do grafo. Vamos levar em consideração apenas as duas representações mais usuais: lista de adjacências e matriz de adjacências.

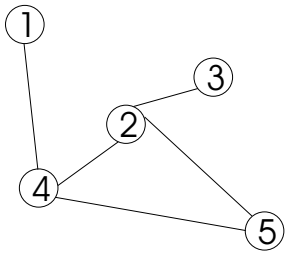

Figura 4: Exemplo simples de um grafo não-orientado.

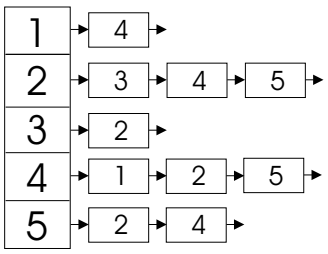

Figura 5: Representação de lista de adjacências para o grafo G da figura 4.

Na representação por lista de adjacências, é usado um vetor onde cada posição corresponde a um vértice do grafo G e, para cada vértice v de G,

temos uma lista encadeada onde cada nó da lista corresponde a um vértice de G adjacente a v. Veja um exemplo na figura 4 cuja representação por lista de adjacências é apresentada na figura 5.

	1	2	3	4	5
1	0	0	0	**1**	0
2	0	0	**1**	**1**	**1**
3	0	**1**	0	0	0
4	**1**	**1**	0	0	**1**
5	0	**1**	0	**1**	0

Figura 6: Representação por matriz de adjacências para o grafo da figura 4.

Na representação por matriz de adjacências, é construída uma matriz $n \times n$, M, onde as posições $M_{i,j}$ $(1 \leq i, j \leq n)$ são preenchidas de modo que $M_{i,j} = 0$ se não há aresta entre os vértices i e j, $M_{i,j} = 1$ se há tal aresta. Valores diferentes de 1 podem ter significados diferentes como a existência de múltiplas arestas ou de arestas com pesos (ou custos), dependendo da aplicação. A matriz de adjacências para o grafo da figura 4 é apresentada na figura 6.

É razoável supor que, durante a construção de qualquer das duas representações, podemos armazenar em cada vértice, seu grau. Assim, a questão de se determinar se um dado vértice tem grau menor que k se torna uma operação de tempo constante.

Usando a representação por lista de adjacências, a cada passo é necessário encontrar um vértice v cujo grau seja menor que k (tempo n, no pior caso). Uma vez que v seja encontrado, este deve ser removido e devem ser removidas as entradas correspondentes a ele nas listas de adjacências de todos os vértices adjacentes a v. Essa segunda operação depende diretamente do grau de cada vértice da lista de adjacências de v e requer um total de $\sum_{i=1}^{k-1} g(v_i)$ onde, no pior caso, $g(v_i) = n-1$. A recorrência que expressa a complexidade do algoritmo para a representação por lista de adjacências é, portanto, a seguinte:

$$T(n) = T(n-1) + n + (k-1)(n-1).$$

Como estamos considerando que k é uma entrada do problema, a solução desta recorrência pertence a $O(kn^2)$.

Exercício: O algoritmo acima descrito tem complexidade linear, quadrática ou exponencial? Justifique sua resposta.

Se usarmos a representação por matriz de adjacências, a tarefa do algoritmo se torna um pouco mais simples. Neste caso, quando encontramos um vértice v

de grau $< k$, precisamos percorrer apenas uma linha (correspondente ao vértice v) e uma coluna (correspondente aos vértices aos quais v é adjacente) da matriz. A complexidade para esta representação é descrita pela seguinte relação de recorrência:

$$T(n) = T(n-1) + n + n.$$

A solução para esta recorrência pertence a $O(n^2)$, ou seja, a representação da informação aumentou a velocidade do algoritmo.

É possível reduzir ainda mais o tempo necessário para se encontrar os vértices de grau menor que k através do uso de uma estrutura auxiliar como, por exemplo, um *heap* [2] com a chave de acesso sendo o grau de cada vértice. O tempo necessário para construir o *heap* inicial é $O(n \log n)$ e a remoção do i-ésimo elemento do *heap* toma tempo $O(\log(n - i + 1))$.

Exercício: Com o uso da estrutura auxiliar de *heap* como descrito acima, qual a complexidade final do algoritmo?

Uma importante lição que podemos tirar deste exemplo é que a eficiência de um algoritmo é freqüentemente melhorada se se escolhe uma estrutura de dados adequada para representação da informação de modo a facilitar os acessos requeridos pelo algoritmo.

4.3 O Problema da União e Busca de Conjuntos

4.3.1 A Função Logaritmo Iterado

Antes de entrarmos no Problema propriamente dito, vamos apresentar a *Função Logaritmo Iterado*.

Inicialmente, definimos a função[5] $\log^{(i)} n$ recursivamente para inteiros não-negativos por:

$$\log^{(i)} n = \begin{cases} n, & \text{se } i = 0 \\ \log(\log^{(i-1)} n), & \text{se } i > 0 \text{ e } \log^{(i-1)} n > 0 \\ \text{indefinida}, & \text{se } i > 0 \text{ e } \log^{(i-1)} n \leq 0 \text{ ou } \log^{(i-1)} \text{ é indefinida} \end{cases}$$

O leitor deve tomar o cuidado de não confundir as notações $\log^{(i)} n$ (i-ésima iterada da função logaritmo) e $\log^i n$ (i-ésima potência do logaritmo de n); esta última é o mesmo que $(\log n)^i$.

Definimos a *função logaritmo iterado*, denotada por $\log^* n$, lê-se "log estrela de n"como:

$$\log^* n = \min \left\{ i \geq 0 : \log^{(i)} n \leq 1 \right\}$$

Vejamos como é lento o crescimento da função logaritmo iterado $\log_2^* n$:

[5] Omitimos aqui a base do logaritmo, mas, quando for conveniente, esta pode ser explicitada.

$$
\begin{aligned}
\log_2^* 2 &= 1 \\
\log_2^* 2^2 = \log_2^* 4 &= 2 \\
\log_2^* 2^{2^2} = \log_2^* 2^4 = \log_2^* 16 &= 3 \\
\log_2^* 2^{2^{2^2}} = \log_2^* 2^{16} = \log_2^* 65536 &= 4 \\
\log_2^* 2^{2^{2^{2^2}}} = \log_2^* 2^{65536} &= 5
\end{aligned}
$$

Exercício: Na representação decimal, quantos dígitos tem o número $2^{2^{2^{2^2}}}$ (que é o *menor* inteiro N para o qual $\log_2^* N$ é maior que 4)?

4.3.2 Coleção de Conjuntos Disjuntos

Estudaremos agora o problema de especificar uma estrutura eficiente para representação de uma coleção de conjuntos disjuntos de modo a podermos processar rapidamente duas operações: união de pares de conjuntos da coleção, e busca pelo conjunto ao qual um dado elemento pertence.

Inicialmente, considere que temos uma coleção de n objetos distintos, na qual cada um destes é univocamente identificado. Podemos pensar neste ponto que temos uma coleção de n conjuntos unitários, cada um dos quais pode ser identificado pelo (identificador correspondente ao) seu único elemento. Após uma seqüência de operações de UNIÃO são formados conjuntos de múltiplos elementos cada um dos quais é identificado por algum de seus elementos de modo que operações de BUSCA podem ser processadas para quaisquer elementos.

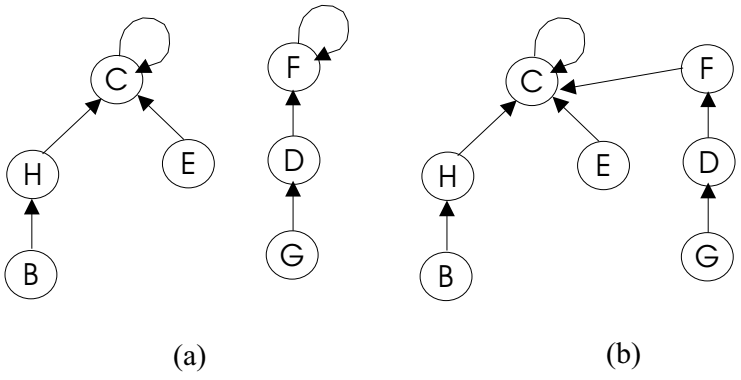

Figura 7: (a) Conjuntos disjuntos $\{B, C, E, H\}$ e $\{D, F, G\}$ e suas representações (b) Resultado da aplicação da operação UNIÃO (C, F)

A estrutura de representação que escolhermos deve dar suporte à operação de união de conjuntos mantendo um identificador para cada conjunto formado,

e à de busca pelo conjunto a que pertence um dado elemento retornando o identificador do conjunto correspondente.

Uma representação que permite que realizemos eficientemente uma seqüência destas operações (UNIÃO e BUSCA) utiliza o que chamamos de árvores enraizadas. Cada árvore representa um conjunto e cada um de seus nós um elemento do respectivo conjunto.

Cada nó da árvore é um registro contendo o identificador do elemento e um apontador para seu nó pai na árvore. Ao elemento representado pelo nó raiz designamos como representante do conjunto (e usamos seu identificador para identificar o próprio conjunto). (O apontador do nó raiz aponta para si mesmo.) A figura 7(a) apresenta dois conjuntos representados desta maneira e a figura 7b) mostra o conjunto resultante da aplicação de uma operação de união.

Para o Problema da UNIÃO e BUSCA de conjuntos, nosso objetivo é fazer com que as operações de UNIÃO e de BUSCA possam ser rapidamente executadas. Para alcançarmos esse objetivo, lançamos mão de duas heurísticas que diminuem substancialmente o tempo de execução dessas duas operações.

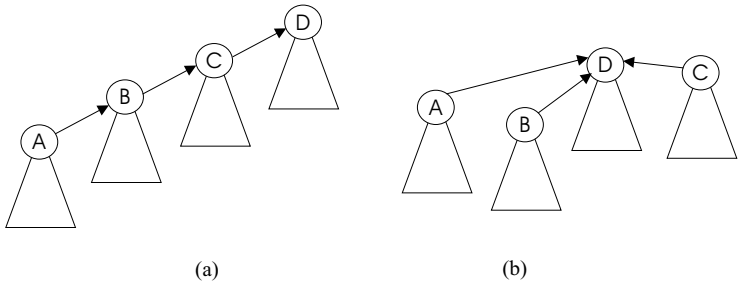

Figura 8: (a) Uma árvore de conjuntos disjuntos. Os triângulos correspondem às sub-árvores cujas raízes são os vértices desta árvore. (b) A árvore formada pelos mesmos vértices, depois de realizada uma operação FIND(A).

A primeira heurística, aplicada à operação de UNIÃO, visa que a árvore resultante tenha a menor altura possível. Para isso, fazemos com que a raiz da árvore com menor altura aponte para a raiz da árvore com maior altura. Deste modo, a altura da árvore resultante é no máximo apenas uma unidade a mais que a altura da mais alta das duas árvores unidas.

A segunda heurística, chamada de compressão de caminho, se aplica durante cada execução de uma operação de BUSCA e visa encurtar o caminho até a raiz de todos os nós intermediários do caminho percorrido. Isso se dá através da ligação direta de cada um destes nós intermediários até a raiz. A figura 8 ilustra a operação desta heurística.

Usando estas duas heurísticas apresentadas, pode-se provar que a realização de uma seqüência de m operações de UNIÃO e BUSCA em uma coleção de n elementos pode ser feita em tempo $O(m \log^* n)$. Veja a demonstração deste fato

em [1]. Uma análise mais acurada (veja [4, 5]) mostra que uma quota superior ainda melhor é possível, explicitamente, $O(m\ \alpha(m,n))$, onde $\alpha(m,n)$ é a inversa da *função de Ackermann* (descrita em [1]). A função $\alpha(m,n)$ cresce ainda mais lentamente que a função $\log^* n$.

5 Exercícios Suplementares

1. Encontre um limite superior para a solução da relação de recorrência $T(n) = 168T(\lfloor n/672 \rfloor + 135) + 168T(\lfloor n/672 \rfloor + 135) + 168T(\lfloor n/672 \rfloor + 189) + 168T(\lfloor n/672 \rfloor + 189) + 456n + 1$.

Referências

[1] T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, segunda edição, MIT Press, 2001.

[2] U. Manber, *Algorithms: A Creative Approach*, Addison-Wesley, 1989.

[3] J. Porto, P. Rezende, *Provas por Indução. Notas de Aula para a Disciplina de Complexidade de Algoritmos*, Instituto de Computação, UNICAMP, 2002.

[4] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm", *Journal of the ACM*, 22(2):215-225, 1975.

[5] R. E. Tarjan, *Data Structures and Network Algoritms*. SIAM, 1983.