4
Compressão de Grafos Web

O grafo web é um grafo direcionado e sem pesos tal que seus nós representam páginas web e cada aresta representa um *link* entre duas páginas web. Este grafo representa a estrutura da web.

Um dos fatos mais interessantes sobre grafos web é que qualquer um pode alterá-lo. É basicamente um artefato humano concebido. Qualquer pessoa pode criar uma página web apontando-a para várias outras páginas web e colocá-la *on-line*. Nenhuma regra nem restrições são impostas.

Contradizendo esta aparência caótica, o grafo web apresenta várias características e padrões interessantes. Uma surpreendente, descoberta por Albert *et al.* (Alb99), é que o comportamento global do grafo web obedece às leis de escala (*scale-law*) que são características apenas de sistemas auto-organizáveis altamente interativos e fenômenos críticos. Eles também descobriram que o grafo web apresenta propriedades dos grafos mundo-pequeno (*small-world*). Isto significa que:

1) O tamanho médio dos caminhos mais curtos entre dois nós é pequeno, ou seja, seu diâmetro é proporcional ao logaritmo da quantidade de nós;

2) Seu coeficiente de agrupamento é alto, ou seja, é localmente denso e globalmente esparsos (Wat98).

Outra importante descoberta de Albert *et al.* é que a distribuição do grau de entrada e saída dos nós seguem a lei das potências $P(x) = x^{-\alpha}$ tal que α é respectivamente $2,1$ e $2,45$. Outras características topológicas incluem a presença de uma grande quantidade de comunidades, nas quais são representadas por subgrafos bipartidos e densos, e o fato de que os nós e arestas do grafo web são inseridos e removidos com alta freqüência ao longo do tempo.

Algumas propriedades do grafo web são conseqüências dos padrões de construção das páginas web. Para fins de navegação, as páginas web do mesmo domínio tendem a apontar para outras páginas web do mesmo domínio. Esta *propriedade de localidade* representa uma quantidade de 75% do total de links (Sue01). Elas também tendem a apontar para páginas web de outros domínios que apresentam correlação semântica (Fla02). Esta *propriedade de semântica* é

freqüentemente usada por aplicações, tais como, *crawlers* focados. Para fins de usabilidade, páginas web do mesmo domínio freqüentemente são construídas usando modelos (*templates*). Em conseqüência, eles tendem a apontar para um subconjunto comum de páginas web (*propriedade de similaridade*). Outra propriedade relacionada à criação das páginas web é a tendência delas de terem links com identificadores únicos consecutivos em relação à ordem lexicográfica das URLs (*propriedade de consecutividade*).

A maioria dessas propriedades foram formalmente caracterizadas e justificadas por modelos matemáticos (Bon05). Finalmente, descobriu-se que cada aresta do grafo web pode ser representado usando $O(1)$ bits (Chi09). Esta é a importante *propriedade de compressibilidade*.

A popularidade dos grafos web tem crescido consideravelmente por causa do sucesso da sua principal aplicação: o *ranking* de páginas web (Pag98) (Kle99a). Desde então, várias outras aplicações foram consideradas para o grafo web como, por exemplo: estratégias de *crawlers* (Cho98) (Cha99) (Naj05); descoberta de comunidades ocultas (Kum99) (Dou07); detecção de *webspam* (Cas07); descoberta de páginas web relacionadas (Dea99); classificação e agrupamento de páginas web por tópico e; previsão da evolução da web usando modelos formais.

O grafo web tem sido um ótimo objeto de pesquisa por causa de suas interessantes características e de sua variedade de aplicações. No entanto, ele tem dois problemas relevantes relacionados com a usabilidade pelas aplicações. *O primeiro problema é que o grafo web é um objeto massivo.* Recentemente, foi encontrado cerca de um trilhão de nós na web (Goo08). O problema piora porque o grafo web está crescendo. *Isto liga ao segundo problema: o grafo web é um objeto altamente dinâmico.* Verificou-se que os *links* de uma página web mudam a uma taxa surpreendente de 25% a cada semana e 80% em um ano (Nto04). Portanto, o uso de *snapshots* estáticos do grafo web pode levar a dados obsoletos e resultados irrelevantes ou imprecisos, acarretando em má utilização dos recursos e perda de tempo dos usuários. Além disso, algumas aplicações exigem que o grafo web seja atualizado rapidamente como, por exemplo, *crawlers* e detectores de webspam.

Um ponto a favor do uso na prática dos grafos web é que eles são redundantes. Na verdade, qualquer esquema de compressão razoável comprimirá bastante o grafo web. Isso permitiu a proposta de várias representações compactas (Bha98) (Adl01) (Sue01) (Ran02) (Gui02) (Rag03) (Bol04a) (Bol04b) (Mah06) (Cla07) (Bue08) (Asa08) (Bol09).

O principal objetivo dessas representações é permitir a execução de consultas que retornam a lista de arestas incidentes de um determinado nó

abaixo de 1 microssegundo por aresta em média e para comprimir o máximo possível. No entanto, *eles foram projetados para serem executados apenas na memória principal (RAM)*. Eles alegaram que seria inviável executar a maioria dos algoritmos de grafos em memória externa (disco) com base em dois fatos:

1) Há uma discrepância entre o tempo de acesso de dados de memória principal e externa na ordem de aproximadamente 10^5;

2) Não é conhecido para vários algoritmos básicos, como DFS, como executá-los eficientemente em memória externa.

Conseqüentemente, isso tem limitado o tamanho do grafo web, que uma aplicação pode usar, para o tamanho da memória principal.

Não temos conhecimento de uma representação compacta *que foi projetada exclusivamente para memória externa*. Vale ressaltar que o autor tem conhecimento da representação *S-node* (Rag03) que permite a execução de algumas consultas avançadas em disco, porém, a *S-node* não foi especificamente projetada para a memória externa. Outra limitação é que a maioria das representações propostas só suporta um tipo de consulta eficientemente. Pelo conhecimento do autor, *não existe nenhuma representação compacta para grafos web que otimiza a execução de consultas avançadas*. Uma terceira limitação dessas representações é que elas representam um *snapshot* estático de um grafo web. Além disso, não se tem conhecimento de representações compactas *que oferecem suporte a operações de atualização, incluindo em memória principal*.

Neste capítulo, resolvemos os dois primeiros problemas acima ao propor uma representação compacta para grafos web específica para memória externa que suporta otimização de consultas avançadas, intitulada *árvore-w*. Propomos também um novo tipo de layout projetado especificamente para grafos web, intitulado *layout escalado*. Além disso, mostramos como construir um layout *cache-oblivious* para explorar a hierarquia de memória, sendo a primeira representação desse tipo para grafos web. Realizamos uma série de experimentos e comparamos com o *webgraph framework* (Bol04a) (Bol04b). Limitamos nossos resultados às páginas estáticas, ou seja, páginas que não são geradas dinamicamente com dados provindos de um banco de dados. Os resultados experimentais mostram que a árvore-w é competitiva com as representações compactas projetadas para memória principal em termos de taxa de compressão e de tempo de processamento. Além disso, demonstramos empiricamente que é viável utilizar uma representação compacta para memória externa na prática, contrariando a afirmação de vários autores (Sue01) (Bue08).

O restante do capítulo está organizado da seguinte forma: Seção 4.1 apresenta a árvore-w e a sua construção; Seção 4.2 apresenta outros possíveis

layouts para a árvore-w; Seção 4.3 mostra como executar as consultas e como otimizar consultas avançadas; Seção 4.4 apresenta os resultados experimentais de nossa implementação.

4.1 Árvore-w

Dado um grafo web $\mathcal{G}$, os nós de $\mathcal{G}$ são ordenados lexicograficamente pelas suas URLs e unicamente identificados por um número natural $\{1, 2, \dots\}$. A lista de adjacência de $\mathcal{G}$ é representada pela coleção $\mathcal{S} = \{s_1, s_2, \dots\}$ tal que todo $s \in \mathcal{S}$ é um conjunto de números naturais ordenado ascendentemente. Nesta seção, o objetivo é representar a coleção $\mathcal{S}$ em uma representação compacta projetada especificamente para a memória externa intitulada *Árvore-w*. Não exploramos estruturas de dados para armazenar a associação entre as URLs e os identificadores dos nós do grafo web uma vez que uma *prefix b-tree* (Bay77) poderia representar de forma compacta e eficiente as URLs e a associação com os identificadores.

A construção da árvore-w tem três estágios. Na primeira fase, construímos uma árvore binária $\mathcal{T}^f$ a partir da coleção $\mathcal{S}$. Na segunda fase, limitamos o tamanho dos nós de $\mathcal{T}^f$ obtendo uma nova árvore binária $\mathcal{T}$. Na terceira fase, agrupamos os nós de $\mathcal{T}$ em subárvores disjuntas de modo que coubessem é uma página (ou bloco) da memória externa obtendo, assim, a árvore-w $\mathcal{W}$. Finalmente, organizamos a árvore-w em uma memória externa de forma otimizada para a consulta de leitura completa (Seção 4.3.1). Devemos mencionar que, na prática, armazenamos todas as árvores intermediárias em memória externa durante a construção da árvore-w. Somente uma quantidade mínima de memória principal é usada.

A seguir, os três estágios da construção da árvore-w são apresentados em detalhes.

4.1.1 Primeiro estágio da construção

No primeiro estágio, a árvore $\mathcal{T}^f$ é construída a partir da coleção $\mathcal{S}$. Sem perda de generalidade, assumimos que a coleção $\mathcal{S}$ tem 2^k conjuntos, ou seja, $|\mathcal{S}| = 2^k$. A árvore $\mathcal{T}^f$ é uma árvore binária completa de profundidade k formada por nós-w (Figura 4.1), cujas arestas estão associadas a um conjunto.

Denotamos a aresta que conecta um nó ao seu pai por *aresta-pai* e a aresta que conecta o nó ao seu filho à esquerda e à direita por *aresta-filho à esquerda* e *aresta-filho à direita*, respectivamente. Dado um nó-w w, temos $C = A \cup B$, onde C é o conjunto associado com a aresta-pai de w e $A(B)$

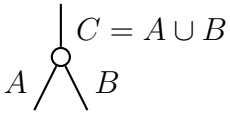


Figura 4.1: Nó-w – representado por círculos.

é o conjunto associado com a aresta-filho à esquerda(direita) de w. Nós-w realizam uma divisão dos elementos de C nos conjuntos A e B que podem ter elementos em comum. Tal operação sobre conjuntos pode ser representada por uma seqüência de operações chamada de *descrição* que é associada a cada nó-w.

Antes da apresentação da construção da árvore $\mathcal{T}^f$, detalhamos a seguir a geração e a codificação das descrições.

Primitiva merge de um nó-w

Dados três conjuntos A, B, C com $A \cup B \subseteq C$, a primitiva $merge(A, B, C)$ gera uma descrição D para (A, B, C) como descrito a seguir. Um item é um par de $\{o, n\}$, onde $o \in$ {LEFT, RIGHT, COPY, REDUNDANT} é uma operação e $n \in \mathbb{N}$ é o tamanho de uma carreira. A descrição D para (A, B, C) é a seqüência de itens recursivamente definido da seguinte forma. Se C é vazio então a descrição de (A, B, C) é uma seqüência vazia. Caso contrário, temos os seguintes casos:

1) O menor elemento de C pertence a $A \cap B$. Seja $R \geq 1$ o maior inteiro tal que os R menores elementos de C pertencem a $A \cap B$. Além disso, seja Z o conjunto que contém os R menores elementos de C. A descrição de (A, B, C) é dado pelo item {COPY, R} seguido da descrição para $(A-Z, B-Z, C-Z)$;

2) O menor elemento de C pertence a $A - B$. Seja $R \geq 1$ o maior inteiro tal que os R menores elementos de C pertencem a $A - B$. Além disso, seja Z o conjunto que contém os R menores elementos de C. A descrição de (A, B, C) é dado pelo item {LEFT, R} seguido da descrição para $(A - Z, B, C - Z)$;

3) O menor elemento de C pertence a $B - A$. Seja $R \geq 1$ o maior inteiro tal que os R menores elementos de C pertencem a $B - A$. Além disso, seja Z o conjunto que contém os R menores elementos de C. A descrição de (A, B, C) é dado pelo item {RIGHT, R} seguido da descrição para $(A, B - Z, C - Z)$;

4) O menor elemento de C não pertence a $A \cup B$. Seja $R \geq 1$ o maior inteiro tal que os R menores elementos de C não pertencem a $A \cup B$. Além disso, seja Z o conjunto que contém os R menores elementos de C. A descrição de

(A, B, C) é dado pelo item {REDUNDANT, R} seguido da descrição para $(A, B, C - Z)$.

Ilustramos a primitiva *merge* com o seguinte exemplo. Suponha que $A = \{1, 7, 8, 9\}$, $B = \{1, 2, 9\}$ e $C = \{1, 2, 3, 7, 8, 9\}$, a descrição é ({COPY, 1}, {RIGHT, 1}, {REDUNDANT, 1}, {LEFT, 2}, {COPY, 1}). Perceba que se $C = A \cup B$ então não há nenhuma operação REDUNDANT na descrição.

Codificação da descrição de um nó-w

A codificação de uma descrição associada a um nó-w é descrita a seguir. Todas as operações, exceto a primeira, pode ser representada com um bit. Isso porque não há itens adjacentes com a mesma operação e também porque a operação REDUNDANT não é codificada (essa operação é usada temporáriamente e apenas no segundo estágio da construção da árvore-w). Para o primeiro item, precisamos de dois bits pois há três possíveis operações. Usamos a codificação Gama (Seção 2.2.3) para representar o tamanho da carreira. O custo de codificação de um inteiro positivo r usando a codificação Gama é $2\lfloor \log_2 r \rfloor + 1$ bits. A codificação de uma descrição é simplesmente a concatenação dos seus itens codificados da esquerda para a direita.

Como exemplo, suponha que $A = \{1, 2, 7\}$, $B = \{3, 4, 5, 6, 7\}$ e $C = \{1, 2, 3, 4, 5, 6, 7\}$. A descrição D para (A, B, C) é dado por ({LEFT,2}, {RIGHT,4}, {COPY,1}). Cada item custa respectivamente 5, 6 e 2 bits. O custo de codificação de {LEFT, 2} é 5 bits porque usamos dois bits para a primeira operação. O custo de codificação de {RIGHT, 4} é 6 bits porque a operação precisa de um bit e o tamanho da carreira requer 5 bits usando a codificação Gama. O custo total de D é 13 bits.

Vale ressaltar que esse codificador *não* pertence à classe de codificadores de fonte de intercalação, definida no Capítulo 3, porque os resultados das operações de igualdade consecutivas são agrupadas em um único item {COPY, n} para algum $n > 1$.

Primitiva unmerge

Dado um conjunto C e uma descrição D, a primitiva $unmerge(C, D)$ recupera os conjuntos A e B tal que $A \cup B = C$ e D é uma descrição para (A, B, C).

Pelo fato de que não armazenamos o número de itens da descrição D, não é possível recuperar D pela decodificação de seus itens na memória e, em seguida, separar o conjunto C de acordo com D para reconstruir os conjuntos A e B. Ambas as tarefas devem ser executadas ao mesmo tempo. Em vez disso,

usamos a cardinalidade do conjunto C para determinar se devemos deixar de decodificar os itens de D da memória. No entanto, se já temos a descrição D então a tarefa de decodificá-la pode ser ignorada.

A primitiva *unmerge* funciona recursivamente da seguinte forma. Se o conjunto C é vazio então A e B também são conjuntos vazios; caso contrário, decodificamos o item $\{o, r\}$ na memória da esquerda para a direita como segue. Seja Z o conjunto que contém os r menores elementos do conjunto C. Temos os seguintes casos:

1) Suponha que a operação o é LEFT. Então, incluímos os elementos do conjunto Z no conjunto A, ou seja, $A = A \cup Z$;

2) Suponha que a operação o é RIGHT. Então, incluímos os elementos do conjunto Z no conjunto B, ou seja, $B = B \cup Z$;

3) Suponha que a operação o é COPY. Então, incluímos os elementos do conjunto Z no conjunto A e B, ou seja, $A = A \cup Z$ e $B = B \cup Z$.

Devemos lembrar que não há operações REDUNDANT na descrição D ao executar a primitiva *unmerge*. Finalmente, chamamos recursivamente o procedimento para o conjunto $C - Z$.

Ilustramos o procedimento com o seguinte exemplo. Seja o conjunto $C = \{1, 2, 3, 4, 5, 6, 7\}$ e a descrição D para (A, B, C) dado por ({LEFT, 2}, {RIGHT, 2}, {COPY, 1}, {RIGHT, 1}, {COPY, 1}). Na primeira iteração, temos o item {LEFT, 2}. Portanto, a primitiva remove os 2 menores elementos de C e coloca em A. Agora, temos $A = \{1, 2\}$ e $C = \{3, 4, 5, 6, 7\}$. Ao final, teremos os conjuntos $A = \{1, 2, 5, 7\}$ e $B = \{3, 4, 5, 6, 7\}$.

Construção

Neste ponto, descrevemos uma construção para a árvore binária $\mathcal{T}^f$. Para $i = 1, 2, \ldots, |\mathcal{S}|$, a i-ésima aresta-filho da esquerda para a direita localizada no último nível de $\mathcal{T}^f$ é associada ao conjunto $s_i \in \mathcal{S}$. Seja e uma aresta que não está localizada no último nível de $\mathcal{T}^f$. A aresta e é associada com a união de todos os conjuntos que são descendentes da aresta e em $\mathcal{T}^f$. A descrição do nó-w w é a descrição para (A, B, C), onde $A(B)$ é o conjunto associado com a aresta-filho à esquerda(direita) de w e C é o conjunto associado com a aresta-pai de w. A árvore binária $\mathcal{T}^f$ pode ser construída usando uma abordagem de baixo para cima. Como um exemplo, veja a Figura 4.2.

Por fim, perceba que a raiz de $\mathcal{T}^f$ é um nó-w cuja aresta-pai não se conecta a nenhum outro nó-w, assim como, as folhas possuem arestas-filho que não se conectam a nenhum outro nó-w. Apesar de não ser comum, isso ajuda

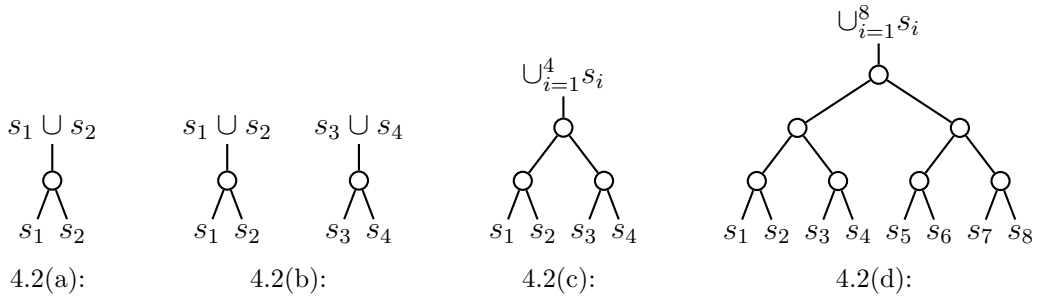


Figura 4.2: Primeiro estágio – (a) primeiro nó-w; (b) segundo nó-w; (c) conectado por um nó-w pai; (d) árvore binária $\mathcal{T}^f$ dos 8 conjuntos de $\mathcal{S}$.

a simplificar a apresentação da árvore-w e de suas árvores intermediárias. O autor acredita que isso não deve ser um problema.

Discussão

A principal propriedade do grafo web explorada para sua compressão pela árvore $\mathcal{T}^f$ é a propriedade de similaridade. A coleção $\mathcal{S}$ é ordenada lexicograficamente de acordo com a URL associada a cada conjunto de $\mathcal{S}$ permitindo que as páginas de mesmo domínio ficam próximas uma das outras. De acordo com a propriedade de similaridade, todas as páginas web de um domínio tendem a apontar para um mesmo subconjunto de páginas web do mesmo domínio e, portanto, existe uma alta probabilidade de dois conjuntos adjacentes de $\mathcal{S}$ terem elementos em comum.

Esta propriedade é explorada quando dois conjuntos de $\mathcal{S}$ são unidos por um nó-w. A descrição gerada tipicamente tem várias operações COPY. Perceba que esta operação codifica eficientemente os elementos em comum aos dois conjuntos. Na medida em que as subárvores inferiores são construídas e unidas por um nó-w, neste primeiro estágio, a descrição gerada ainda tende a ter operações COPY. Isso ocorre porque, de acordo com a propriedade de similaridade, todos os conjuntos de $\mathcal{S}$ correspondentes a um mesmo domínio tendem a ter um subconjunto em comum. Assim, o subconjunto em comum é codificado novamente na união entre dois nós-w de $\mathcal{T}^f$. Neste momento, também é naturalmente explorado a propriedade de localidade, na qual afirma que as páginas web tendem a apontar mais para as páginas web do mesmo domínio.

Por fim, perceba que podemos recuperar um conjunto da coleção $\mathcal{S}$ realizando uma sequência de *unmerges* da raiz de $\mathcal{T}^f$ até a folha desejada. No entanto, cada descrição codificada nessa sequência pode ter $O(|\mathcal{S}|)$ bits no pior caso. Isso acarreta na decodificação de $O(|\mathcal{S}| \log_2 |\mathcal{S}|)$ bits, no pior caso, para acessar um único conjunto de $\mathcal{S}$.

No próximo estágio, construímos a árvore binária $\mathcal{T}$ a partir de $\mathcal{T}^f$. Isso é feito de tal forma que seja necessário decodificar apenas $O(\log_2 |\mathcal{S}|+|\mathcal{S}|)$ bits, no pior caso, para acessar um conjunto de $\mathcal{S}$ sem afetar significativamente a taxa de compressão.

4.1.2 Segundo estágio da construção

No segundo estágio, a árvore $\mathcal{T}$ é construída a partir da árvore $\mathcal{T}^f$. Vale lembrar que ainda estamos assumindo que a coleção $\mathcal{S}$ tem 2^k conjuntos e que a árvore $\mathcal{T}^f$ tem profundidade k. A árvore $\mathcal{T}$, por sua vez, é uma árvore binária de profundidade $2k + 1$ formada por nós-w (Figura 4.1) e por nós-drenagem (Figura 4.3), que serão introduzidos mais adiante nesta seção. Em ambos os tipos de nós, as arestas estão associadas a um conjunto.

O objetivo é delimitar a descrição de cada nó da árvore $\mathcal{T}^f$ em até L bits e, assim, obter a árvore $\mathcal{T}$. Para isto, precisamos revisitar a primitiva *merge* e introduzir os nós-drenagem antes de descrevermos uma construção de $\mathcal{T}$. Denotamos por *nó delimitado* o nó cuja descrição codificada é limitada em até L bits e por *nó não-delimitado* o nó cuja descrição codificada não é limitada. Por exemplo, todos os nós-w da árvore $\mathcal{T}^f$ são não-delimitados.

Primitiva merge para nós-w revisitado

Para o segundo estágio, é necessário redefinir a primitiva *merge* apresentada na Seção 4.1.1 de forma que aceite o parâmetro de entrada L.

A primitiva $merge(A, B, C, L)$ para um nó-w w obtém uma descrição D_j ($j \geq 0$), tal que o custo de codificação não seja superior a L bits, a partir de uma descrição D_0 para (A, B, C), onde $A(B)$ é o conjunto associado com a aresta-filho à esquerda (direita) de w e C é o conjunto associado à aresta-pai de w. A descrição D_j não é uma descrição para (A, B, C) mas para a tripla (A', B', C) tais que $A' \supseteq A$, $B' \supseteq B$ e $A' \cup B' = C$.

A primitiva *merge* é apresentada no Algoritmo 6. Para simplificar a sua apresentação, separamos em três fases: a fase de pré-processamento, a fase principal e a fase de pós-processamento.

Na fase de pré-processamento (linhas 2-3), o algoritmo constrói uma descrição não-delimitada D_0 para (A, B, C) assim como descrito na Seção 4.1.1. Em seguida, transforma a descrição D_0 em uma nova descrição D_1 que não contem itens com a operação REDUNDANT. Para conseguir isso, o algoritmo examina a sequência D_0 da esquerda para a direita; sempre que encontra um item com operação REDUNDANT, digamos $d = \{\text{REDUNDANT}, r\}$, ele

Algoritmo 6 Primitiva *merge* para nós-w

```
1: function MERGE(Conj. A, Conj. B, Conj. C, Inteiro L) : Conj. A', Conj. B', Desc. D_j
2:     Construa uma descrição não-delimitada D_0 para (A, B, C)
3:     Combine os itens com operação REDUNDANT de D_0 com seus itens adjacentes
       obtendo uma nova descrição D_1
4:     Construa um min-heap de todos os pares de itens adjacentes de D_1 usando a função
       de custo f_w(·)
5:     Seja ℓ o custo de codificação de D_1
6:     Seja j um contador, começando no 1, para a j-ésima descrição D_j
7:     while ℓ > L do
8:         Extraia um item do min-heap e suponha que ele seja o i-ésimo item de D_j
9:         Decremente ℓ em f_w(i) unidades
10:        Seja d_i e d_{i+1} o i-ésimo e (i + 1)-ésimo item de D_j, respectivamente
11:        Seja r_i e r_{i+1} o tamanho da carreira de d_i e d_{i+1}, respectivamente
12:        Faça d_i igual a {COPY, r_i + r_{i+1}}
13:        Remova o item d_{i+1} e incremente j para obter uma nova descrição D_j
14:        Atualize o min-heap com os dois novos itens adjacentes de d_i
15:    end while
16:    A', B' ← UNMERGE(C, D_j)
17:    return A', B', D_j
18: end function
```

incrementa o tamanho da carreira de um item adjacente à d em r unidades e, em seguida, ele remove o item d da descrição D_0 (linha 3).

Na etapa principal (linhas 7-15), o algoritmo começa com a descrição D_1 e, na iteração $j-1$, combina dois itens adjacentes da descrição D_{j-1} para obter a descrição D_j. O laço termina quando o custo de codificação da descrição atual D_j é no máximo L bits. A etapa de união consiste em substituir dois itens adjacentes por um novo item tal que sua operação é COPY e seu tamanho da carreira é igual à soma dos tamanhos das carreiras de ambos os itens. A estratégia para escolher, em cada iteração, um par de itens adjacentes para ser combinado é selecionar aquele par que possui o menor valor em relação a uma certa função de custo f_w que é definido adiante nesta seção.

Na fase de pós-processamento (linha 16), a descrição D_j satisfaz o limite de custo necessário, mas não mais é uma descrição para (A, B, C). Assim, chamamos a primitiva *unmerge*, com parâmetros C e D_j, para obter os conjuntos A' e B' tais que $A' \cup B' = C$ e D_j é uma descrição para (A', B', C). Para construir uma descrição não-delimitada como descrito na Seção 4.1.1, o procedimento $\textsc{Merge}(A, B, A \cup B, \infty)$ é chamado e retorna os mesmos conjuntos A e B juntamente com sua descrição.

Resta definir $f_w(\cdot)$ (linha 4 do Algoritmo 6). Note que, depois de unir dois itens de uma descrição D_{j-1} para (A, B, C), obtemos uma nova descrição D_j que pode ser vista como uma descrição não-delimitada para (A', B', C) para algum conjunto $A' \supseteq A$ e $B' \supseteq B$ tais que $A' \cup B' = C$. Os novos elementos redundantes formados neste processo são os elementos dos conjuntos $A' - A$ e

$B' - B$. Podemos agora definir a função de custo f_w da seguinte forma.

Definição 4.1 *Dada uma descrição D, a função de custo $f_w(i)$ para nós-w é a razão entre o número de novos elementos redundantes e a economia de codificação devido à união do i-ésimo item e o $(i+1)$-ésimo item da descrição D. A economia é calculada da seguinte forma. Seja r_i e r_{i+1} os tamanhos da carreiras do i-ésimo e $(i+1)$-ésimo item e $r_i + r_{i+1}$ o tamanho da carreira do novo item. Portanto, a economia é $(2\lfloor \log_2 r_i \rfloor + 2\lfloor \log_2 r_{i+1} \rfloor + 4) - (2\lfloor \log_2 (r_i + r_{i+1}) \rfloor + 2)$ bits, na qual nunca é negativa. Se a economia for zero então $f_w(i) = \infty$.*

Como exemplo, suponha que $A = \{1, 2, 7\}$, $B = \{3, 4, 5, 6\}$ e $C = \{1, 2, 3, 4, 5, 6, 7\}$. A descrição D_0 para (A, B, C) é dado por ({LEFT,2}, {RIGHT,4}, {LEFT,1}). Vamos combinar os dois primeiros itens da descrição D_0. Isso acarretará em um novo item {COPY, 2+4} de uma nova descrição D_1 = ({COPY,6}, {LEFT,1}) para (A', B', C) tal que $A' = \{1, 2, 3, 4, 5, 6, 7\}$ e $B' = \{1, 2, 3, 4, 5, 6\}$. Perceba que os elementos redundantes são $A' - A = \{3, 4, 5, 6\}$, que foram copiados do B para o A, e $\{1, 2\}$ que foram copiados de A para B. O número total de novos elementos redundantes é 6, a economia é 4 bits e, portanto, $f_w(1) = 1,5$ elementos redundantes por bit economizado.

Ilustramos o Algoritmo 6 com o seguinte exemplo. Seja $A = \{1, 2, 5, 7\}$, $B = \{3, 4, 5, 6, 7\}$ e $C = \{1, 2, 3, 4, 5, 6, 7\}$. Assim, a descrição $D_0 = D_1$ para (A, B, C) é ({LEFT, 2}, {RIGHT, 2}, {COPY, 1}, {RIGHT, 1}, {COPY, 1}), que custa 15 bits. Se precisarmos delimitar D_1 então calculamos $f_w(1) = 2$, $f_w(2) = 1$, $f_w(3) = \infty$ e $f_w(4) = \infty$. Em seguida, selecione o menor deles, combine os itens adjacentes e obtenha uma descrição D_2 = ({LEFT, 2}, {COPY, 3}, {RIGHT, 1}, {COPY, 1}). A descrição D_2 já não descreve A, B e C, mas descreve $A' = \{1, 2, 3, 4, 5, 7\}$, B e C. Agora, ela custa 14 bits.

Nós-drenagem

A primitiva *merge* para nós-w tem a estratégia de substituir itens adjacentes da descrição, cujas operações não sejam COPY, por novos itens, tal que a operação é COPY e o comprimento da carreira é a soma do comprimento da carreira dos itens substituídos. Seja $A(B)$ o conjunto associado à aresta-filho à esquerda(direita) de um nó-w w. Ao executarmos a primitiva *merge* em w, um novo conjunto $A'(B')$ é obtido e associado à aresta-filho à esquerda(direita). Um problema emerge por causa desta abordagem: o conjunto $A'(B')$ pode ter mais elementos que o conjunto original $A(B)$. Os elementos do conjunto $A' - A$ $(B' - B)$ são chamados *Redundantes*.

Uma conseqüência deste problema é que os nós descendentes de w terão que descrever não apenas os elementos originais de seus conjuntos, assim como,

os elementos redundantes provindos de w. Suponha que o espaço reservado de L bits já não seja suficiente para descrever os elementos originais dos conjuntos de um nó-w u descendente de w. Portanto, já seriam gerados elementos redundantes, digamos x, ao aplicarmos a primitiva *merge* em u. Com adição dos elementos redundantes provindos da aplicação da primitiva *merge* em w, outros elementos redundantes serão gerados, diferentes dos elementos em x. Esse fenômeno é chamado *Efeito de Redundância em Cascata*.

Uma outra consequência mais grave desse problema é que os conjuntos associados às arestas-filho das folhas de $\mathcal{T}$ podem não mais pertencer à coleção $\mathcal{S}$. Isso ocorre por causa dos elementos redundantes que podem ter se espalhados até as folhas de $\mathcal{T}$.

Para evitar concomitantemente ambas as consequências, introduzimos um novo tipo de nó chamado *Nó-drenagem* (Figura 4.3). Denotamos por *aresta-pai* e *aresta-filho* as arestas que conectam o nó-drenagem ao seu pai e ao seu filho, respectivamente. Dado um nó-drenagem d, temos que $C \supseteq C'$, onde C é o conjunto de associado para a aresta-pai de d e C' é o conjunto de associado à aresta-filho de d.

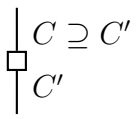


Figura 4.3: Nó-drenagem - representado por quadrados.

Seu objetivo principal é "drenar"a redundância gerada pela estratégia da primitiva *merge*. Para evitar que a redundância seja espalhada através da árvore, adotamos a política de colocar um nó-drenagem delimitado entre dois nós-w de forma que eles drenem para fora a redundância assim que seja gerada. Isso também ajuda a manter a taxa de compressão. E para garantir que os conjuntos associados às arestas-filho das folhas de $\mathcal{T}$ pertençam à coleção $\mathcal{S}$, temos que adicionar nós-drenagem não-delimitados às folhas de $\mathcal{T}$. Antes de construirmos $\mathcal{T}$, temos que mostrar como codificar e delimitar a descrição de um nó-drenagem.

Codificação da descrição de um nó-drenagem

A codificação de uma descrição associada a um nó-drenagem é descrita a seguir. A descrição tem somente itens com operação LEFT ou RIGHT. Se não houver nenhum item com operação RIGHT então emitimos para a saída um 0-bit-flag; caso contrário, emitimos para a saída 1-bit-flag e procedemos da seguinte forma. Se o primeiro item tem operação RIGHT então emitimos para a saída um 0-bit-flag; caso contrário, um 1-bit-flag é usado. Em seguida,

o tamanho da carreira de cada item é codificada usando a codificação Gama e concatenadas da esquerda para a direita. Não é necessário representar as outras operações.

Como exemplo, suponha que $C' = \{1, 2, 7\}$ e $C = \{1, 2, 3, 4, 5, 6, 7\}$. A descrição D para $(C', \emptyset, C)$ é dada por ({LEFT, 2}, {RIGHT, 4}, {LEFT, 1}). Cada item custa respectivamente 3, 5 e 1 bit. O custo total de D é 11 bits pois são 9 bits dos comprimentos das carreiras mais dois bits.

Primitiva merge para nós-drenagem

A primitiva $merge(C, C', L)$ para um nó-drenagem d obtém uma descrição D_j ($j \geq 0$) tal que o custo de codificação não seja superior a L bits, a partir da descrição D_0 para $(C', \emptyset, C)$, onde C' é o conjunto associado com a aresta-filho de d e C é o conjunto associado com a aresta-pai de d. A descrição D_j não é uma descrição para $(C', \emptyset, C)$ mas para (C'', B, C) tal que $C'' \supseteq C'$ e $C'' \cup B = C$.

A primitiva *merge* para nós-drenagem é apresentada no Algoritmo 7. Para simplificar a sua apresentação, novamente dividimo-la em três fases: a fase de pré-processamento, a fase principal e a fase de pós-processamento.

Algoritmo 7 Primitiva *merge* para nós-drenagem

```
 1: function MERGE(Conj. C, Conj. C', Inteiro L) : Conj. C'', Descrição D_j
 2:     Construa uma descrição não-delimitada D_0 para (C', ∅, C)
 3:     Construa um min-heap de triplas de itens adjacentes de D_0 usando f_d(·)
 4:     Seja ℓ o custo da codificação de D_0
 5:     Seja j um contador, começando com 2, para a j-ésima descrição D_j
 6:     while ℓ > L do
 7:         Extraia um item do min-heap e suponha que ele seja o i-ésimo item de D_{j-2}
 8:         Decremente ℓ por f_d(i) unidades
 9:         Seja d_i o i-ésimo item de D_{j-2}
10:         Seja r_{i-1}, r_i e r_{i+1} o tamanho da carreira de d_{i-1}, d_i e d_{i+1}, respectivamente
11:         Seja d_i igual a {LEFT, r_{i-1} + r_i + r_{i+1}}
12:         Remova os itens d_{i-1} e d_{i+1} e incremente j para obter uma nova descrição D_{j-1}
13:         Atualize o min-heap com dois novos itens adjacentes de d_i
14:     end while
15:     Substitua os itens {REDUNDANT, r} de D_{j-1} por {RIGHT, r} obtendo D_j
16:     C'', B ← UNMERGE(C, D_j)
17:     return C'', D_j
18: end function
```

Na fase de pré-processamento (linhas 2-3), o algoritmo simplesmente constrói a descrição não-delimitada D_0 para $(C', \emptyset, C)$ como apresentado na Seção 4.1.1. Na etapa principal (linhas 6-14), o algoritmo começa com a descrição D_0 e, na iteração $j - 2$, combina três itens adjacentes de D_{j-2} para obter uma descrição D_{j-1}. O laço termina quando o custo da codificação da descrição atual D_{j-1} é no máximo L bits. Em qualquer iteração $j - 1$, a

descrição D_{j-1} pode ter operações REDUNDANT. Assim, partimos do princípio que os itens com operação REDUNDANT são RIGHT apenas para calcular o custo da codificação de D_{j-1}. A etapa de união consiste em substituir três itens adjacentes d_{i-1}, d_i e d_{i+1}, tal que a operação de d_i é REDUNDANT, por um novo item tal que sua operação seja LEFT e o tamanho da carreira é a soma do tamanho das carreiras de todos os três itens. A estratégia escolhida para selecionar, em cada iteração, uma tripla de itens adjacentes a serem combinados é selecionar um entre todas as triplas de itens adjacentes, tal que a operação do item do meio seja REDUNDANT, que tem o menor valor com respeito à função de comprimento f_d que definimos mais adiante nesta seção.

Na fase de pós-processamento (linhas 15-17), a descrição D_{j-1} satisfaz o custo necessário mas não é mais uma descrição para $(C', \emptyset, C)$. Para se certificar de que não existem itens na descrição D_{j-1} da forma $\{\text{REDUNDANT}, r\}$, ele substitui cada um item desse tipo com o item $\{\text{RIGHT}, r\}$ obtendo a descrição D_j. Em seguida, ele chama a primitiva *unmerge*, com parâmetros C e D_j, para obter os conjuntos C' e B tal que $C'' \cup B = C$ e D_j é uma descrição para (C'', B, C). O conjunto B contém apenas os itens que foram drenados para fora do conjunto C. Portanto, ele é ignorado e só é retornado o conjunto C'' e a descrição D_j. Para a construção de uma descrição não-delimitada, chamamos $\text{MERGE}(C, C', \infty)$ que retorna o mesmo conjunto C' juntamente com a descrição.

Definição 4.2 *Dada uma descrição D, a função de custo $f_d(i)$ é a razão entre o tamanho da carreira do i-ésimo item d_i da descrição D e as economias de combinar d_{i-1}, d_i e d_{i+1}. A economia é calculada da seguinte forma. Seja r_{i-1}, r_i e r_{i+1} os tamanhos das carreiras de d_{i-1}, d_i e d_{i+1}, respectivamente. A economia é $(2\lfloor \log_2 r_{i-1} \rfloor + 2\lfloor \log_2 r_i \rfloor + 2\lfloor \log_2 r_{i+1} \rfloor + 3) - (2\lfloor \log_2 (r_{i-1} + r_i + r_{i+1}) \rfloor + 1)$ bits, na qual nunca é negativa. Se a economia for zero então $f_d(i) = \infty$.*

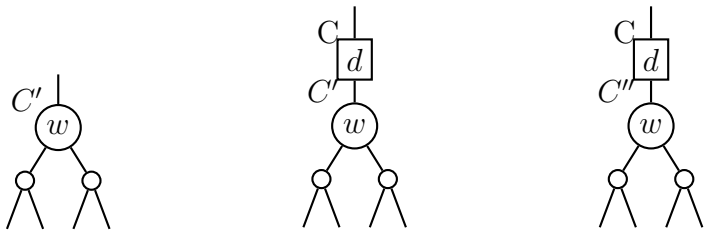
Como exemplo, suponha que D = ({LEFT,2}, {REDUNDANT,4}, {LEFT,1}) e queremos calcular $f_d(2)$. O tamanho da carreira do segundo item é 4 e a economia é 4, assim, $f_d(2) = 1$ elemento redundante por bit economizado.

Ilustramos o Algoritmo 7 com o seguinte exemplo. Suponha que $C = \{1, 2, 3, 4, 5, 6, 7\}$ e $C' = \{1, 3, 6, 7\}$. Assim, a descrição D_0 para $(C', \emptyset, C)$ é $(\{\text{LEFT}, 1\}, \{\text{REDUNDANT}, 1\}, \{\text{LEFT}, 1\}, \{\text{REDUNDANT}, 2\}, \{\text{LEFT}, 2\})$, que custa 11 bits. Partimos do princípio que os itens com operação REDUNDANT são RIGHT apenas para calcular os custos. Se precisarmos delimitar D_0 então calculamos $f_d(2) = \infty$ e $f_d(4) = 1$. Em seguida, escolhemos o menor deles, combinamos os itens adjacentes e obtemos a descrição D_1 = ({LEFT, 1}, {REDUNDANT, 1}, {LEFT, 5}). A descrição D_1 já não descreve C' e C, mas descreve $C'' = \{1, 3, 4, 5, 6, 7\}$ e C. Ela custa agora 7 bits.

Construção

Descrevemos um novo procedimento recursivo para delimitar as descrições dos nós-w da árvore $\mathcal{T}^f$ e, assim, obter a nova árvore $\mathcal{T}$.

O procedimento recebe como parâmetro de entrada uma árvore não-vazia T e um conjunto C e procede da seguinte forma. Seja w a raiz de T (Figura 4.4(a)) e seja A, B e C', respectivamente, o conjunto associado à aresta-filho à esquerda, à direita e à aresta-pai de w. Primeiro, criamos um nó-drenagem delimitado d e conectamos ao nó w de tal forma que d seja o pai de w (Figura 4.4(b)). Em seguida, associamos o conjunto C à aresta-pai de d. Perceba que o conjunto C' continua associado à aresta-pai de w. Agora, podemos executar a primitiva *merge* para o nó-drenagem d com os parâmetros C, C' e L e obtemos um conjunto C'' e uma descrição D. Este conjunto é associado à aresta-pai de w substituindo o conjunto C' (Figura 4.4(c)) e a descrição D é associada ao nó-drenagem d.



4.4(a): 4.4(b): 4.4(c):

Figura 4.4: Segundo estágio.

Procede-se para delimitar o nó-w w. Executamos a primitiva *merge* em w com os parâmetros A, B, C'' e L e obtemos os conjuntos A' e B' e a descrição D'. Esta descrição é associada ao nó-w w. Seja T_l e T_r as subárvores formadas se removermos d e w de T. O procedimento tem dois casos:

1) Suponha que $T_l(T_r)$ não é uma árvore vazia. Então, executamos recursivamente este procedimento com os parâmetros $T_l(T_r)$ e $A'(B')$;

2) Suponha que $T_l(T_r)$ é uma árvore vazia. Então, criamos um nó-drenagem não-delimitado $d'(d'')$ e conectamos ao nó-w w de tal forma que $d'(d'')$ seja o filho à esquerda (direita) de w. Associamos o conjunto $A(B)$ à aresta-filho de $d'(d'')$ e o conjunto $A'(B')$ à aresta-pai de $d'(d'')$. Agora, podemos executar a primitiva *merge* para o nó-drenagem $d'(d'')$ com os parâmetros $A'(B')$, $A(B)$ e $L = \infty$ e obtemos a descrição $D''(D''')$. Perceba que o conjunto resultante da primitiva é o mesmo conjunto $A(B)$ da entrada. Por fim, a descrição $D''(D''')$ é associada ao nó-drenagem $d'(d'')$.

Para construir a árvore $\mathcal{T}$, chamamos o novo procedimento com a árvore $\mathcal{T}^f$ e o conjunto $\{1, 2, \ldots, |\mathcal{S}|\}$ como parâmetros de entrada. O layout da árvore $\mathcal{T}$ é apresentado na Figura 4.5.

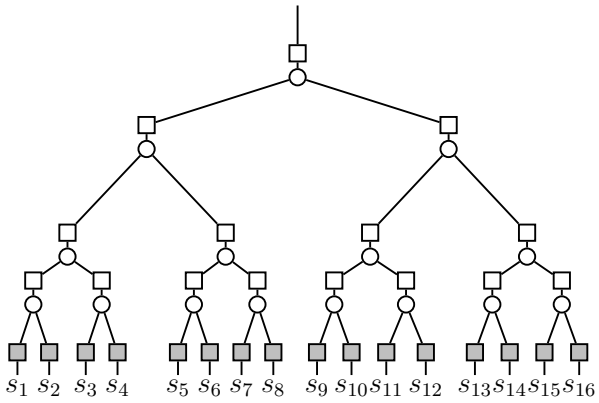


Figura 4.5: Árvore $\mathcal{T}$ – nós-w, nós-drenagem delimitados e não-delimitados são representados por círculos, quadrados não-preenchidos e preenchidos, respectivamente.

Discussão

Para o problema de delimitar uma descrição, propusemos um algoritmo guloso que usa a função de custo para selecionar, em cada iteração, itens adjacentes a serem combinados. A função de custo f_w é projetada para equilibrar o número de elementos redundantes gerados (n_r) e a quantidade de bits economizados (n_s). Ela é definida como a razão n_r/n_s para ambos os tipos de nós. Podemos reduzir o custo das descrições usando uma função ligeiramente diferente para os nós-drenagem dada por n_r/n_s^2. Essa função gera pequenas, porém, consistentes reduções do custo das descrições. Ela também gera reduções ligeiramente superiores quando o tamanho máximo L por nó é pequeno. É interessante notar que a escolha de uma função que é apenas proporcional à n_r aumenta o custo total das descrições na árvore.

Para acessar um conjunto da coleção $\mathcal{S}$ usando a árvore $\mathcal{T}$, temos que decodificar a descrição de $O(\log_2 |\mathcal{S}|)$ nós-w e nós-drenagem delimitados em L bits e a descrição do nó-drenagem não-delimitado na folha de $\mathcal{T}$ que pode ter $O(|\mathcal{S}|)$ bits no pior caso. Isso totaliza $O(\log_2 |\mathcal{S}| + |\mathcal{S}|)$ bits no pior caso, pois L é uma constante.

Agora, podemos construir a árvore-w a partir de $\mathcal{T}$ organizando-a para a memória externa.

4.1.3

Terceiro estágio da construção

No terceiro estágio, a árvore-w $\mathcal{W}$ é construída a partir da árvore $\mathcal{T}$. A árvore-w é formado por blocos de profundidade $2h$ como nós internos e por nós-drenagem não-delimitados como folhas. Para facilitar a apresentação, construímos a árvore-w usando um layout normal tal que cada bloco tem uma altura positiva fixa $2h$, ou seja, cada bloco tem 2^h filhos. Além disso, partimos do princípio que a coleção $\mathcal{S}$ tem 2^k conjuntos e que k é um inteiro positivo múltiplo de h (nos referimos à Seção 4.2 para generalizar para um grafo web de tamanho qualquer). Isto implica que a árvore-w tem profundidade $k/h + 1$ tal que a raiz está no nível 1 e as folhas estão no nível $k/h + 1$.

O objetivo é organizar a árvore $\mathcal{T}$ em uma memória externa e, assim, obter a árvore-w. Para isto, particionamos $\mathcal{T}$ em subárvores disjuntas e associamos cada uma a um bloco, de tal forma que o somatório das descrições codificadas do bloco seja delimitado pelo tamanho L_{block} de uma página da memória externa. Para isso, os nós internos de $\mathcal{T}$ devem ter sidos delimitados no segundo estágio por $L = L_{block}/(2^{h+1} - 2)$ bits. Os nós-drenagem não-delimitados, que são folhas de $\mathcal{T}$, não são associados a nenhum bloco e também são folhas da árvore-w. Para navegar entre os blocos é necessário associar a cada bloco um conjunto de ponteiros para o seus filhos.

Descrevemos um procedimento recursivo para a construção da árvore-w $\mathcal{W}$ a partir da árvore $\mathcal{T}$. O procedimento recebe como parâmetro de entrada uma árvore não-vazia T e funciona da seguinte forma. Seja d o nó-drenagem delimitado raiz de T. Seja T_d a subárvore de T com raiz em d e formada por todos os nós de T a uma distância $2h$ de d. Neste ponto, simplesmente associamos T_d a um bloco b. Seja $T_1, T_2, \ldots, T_{2^h}$ as subárvores formadas se removermos a subárvore T_d de T. Para $i = 1, 2, \ldots, 2^h$, o procedimento tem dois casos:

1) Suponha que a raiz de T_i é um nó-drenagem não-delimitado d. Então, fazemos d ser o i-ésimo filho de b;

2) Suponha que a raiz de T_i é um nó-drenagem delimitado d. Então, executamos o procedimento recursivamente com T_i como parâmetro de entrada.

Por fim, na volta da recursão de todas as subárvores T_i, resta-nos armazenar os ponteiros para os filhos do bloco b. Seja $\mathcal{W}_j$ a árvore com raiz no j-ésimo filho de b. Para $j = 1, 2, \ldots, 2^h - 1$, o valor do j-ésimo ponteiro de b é a soma do tamanho de todos os blocos de $\mathcal{W}_j$ mais o tamanho das descrições codificadas das folhas (nós-drenagem não-delimitados) de $\mathcal{W}_j$. O tamanho de um bloco é medido em bits e é igual à soma do tamanho de

suas descrições codificadas mais o custo dos ponteiros. Cada ponteiro pode ser codificado usando a codificação Gama.

Para construir a árvore-w $\mathcal{W}$, chamamos o procedimento com a árvore $\mathcal{T}$ como parâmetro de entrada.

Podemos ainda organizar fisicamente a árvore-w em uma memória externa de tal forma que a consulta de leitura completa possa ser executada com apenas uma passagem seqüencial e nenhum acesso aleatório. Para isso, basta serializarmos a árvore-w da seguinte forma. Armazenamos no início um cabeçalho que consiste no tamanho do conjunto de $\mathcal{S}$, ou seja, $|\mathcal{S}|$. Em seguida, adicionamos recursivamente ao final da memória externa, na ordem de uma DFS sobre $\mathcal{W}$ as descrições e ponteiros codificados de cada bloco e folha de $\mathcal{W}$.

Isso completa a construção da árvore-w. O layout completo de $\mathcal{W}$ é apresentado na Figura 4.6.

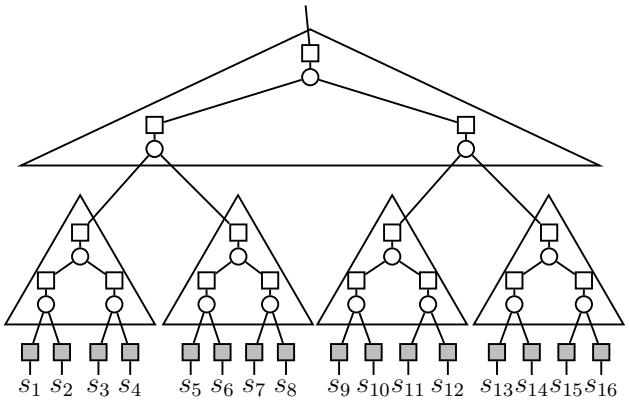


Figura 4.6: Árvore-w $\mathcal{W}$– blocos, nós-w, nós-drenagem delimitados e não-delimitados são representados por triângulos, círculos, quadrados não-preenchidos e preenchidos, respectivamente.

Discussão

Na construção da árvore-w, delimitamos em L_{block} bits apenas a soma dos tamanhos das descrições codificadas dos nós de um bloco. O custo dos ponteiros codificados, no entanto, não foi limitado e é proporcional à ramificação da árvore-w. Suponha que a memória externa é um disco com tamanho de página igual a L_{block} bits. Isso implica que, além de uma página de disco para armazenar as descrições codificadas, são necessárias mais páginas de disco para armazenar os ponteiros codificados.

Para ajustar as descrições e os ponteiros em um número constante de páginas de disco, precisamos reservar um espaço para os ponteiros ao construir um bloco no terceiro estágio da construção. No entanto, sabemos o valor dos ponteiros apenas após a construção de seus filhos. Portanto, podemos apenas

dar uma estimativa de seu custo. Se a estimativa não é precisa, poderemos reservar um espaço desnecessariamente grande deixando apenas um pequeno espaço para as descrições. Neste caso, aumentará o efeito de redundância em cascata e diminuirá a taxa de compressão.

Uma alternativa viável é limitar o custo dos ponteiros restringindo o número de ponteiros por bloco. Seja $|E|$ o número de arestas do grafo web. Como o custo de um ponteiro, no pior caso, é $O(\log_2 |E|)$ bits, então limitamos o número de ponteiros em $O(L_{block}/\log_2 |E|)$.

Podemos reduzir os custos dos ponteiros usando, em vez da codificação Gama, a codificação Golomb (Gol66) que requer um parâmetro δ. Um procedimento para codificar os ponteiros pela estimativa de δ é apresentado a seguir. O valor do primeiro ponteiro (P_1) é armazenado usando a codificação Gama e, em seguida, fazemos $p = P_1$ e $n = 1$. Para o próximo ponteiro (P_2), codificamo-lo com $\delta = \lfloor p/n \rfloor$. Incrementamos p e n em P_2 e 1 unidades, respectivamente. Repetimos este procedimento para todos os ponteiros do bloco. Este procedimento só é eficaz para os blocos que armazenam mais de um ponteiro. Ele gera um pequeno, porém, consistente aumento da taxa de compressão da árvore-w.

Podemos reduzir o custo dos ponteiros tratando o seguinte evento: o conjunto C associado com a aresta-pai das folhas da árvore-w não tem nenhum elemento redundante. Se este evento ocorre então utilizamos uma codificação diferente para os ponteiros dos blocos que são pais das folhas. Essa codificação é descrita a seguir. Se C não tiver elementos redundantes então o valor do ponteiro P_i é aumentado em 1 e não armazenamos o bit-flag da codificação do nó-drenagem (ver Seção 4.1.2); caso contrário, fazemos $P_i = 0$ e não armazenamos a descrição codificada da folha.

Quando decodificamos, interpretamos o valor do ponteiro $P_i = 0$ como indicador (*flag*) de que o conjunto associado à aresta-pai da folha não tem elementos redundantes e é igual ao conjunto associado à aresta-filho da folha. Portanto, não é preciso ler a descrição da folha. Quando $P_i > 0$, então subtraímos P_i por 1 e, assim, poderemos ter a posição inicial da descrição codificada da folha na memória.

Essa codificação justifica-se pelo fato de verificarmos, na prática, que o efeito de redundância em cascata é rapidamente contido nos nós-drenagem superiores da árvore $\mathcal{T}$. Portanto, as descrições das folhas (que são nós-drenagem não-delimitados) raramente refinam o conjunto associado à aresta-pai na prática.

4.2

Layouts

Neste seção, apresentamos outros layouts possíveis para a árvore-w. Dado a árvore binária $\mathcal{T}$ (ver Seção 4.1.2), podemos particioná-la em subárvores de diferentes formas se relaxarmos o fato da ramificação da árvore-w ser um valor fixo.

Iniciamos generalizando o layout normal apresentado na Seção 4.1.3 para representar um grafo web de qualquer tamanho. Observe que o layout normal é parecido com o da *b-tree* (Bay72). Em seguida, introduzimos um novo layout, chamado *Escalado*, para melhorar a taxa de compressão e tempo de processamento das consultas. Finalmente, apresentamos um layout baseado no trabalho de Bender *et al.* (Ben05) para explorar a hierarquia das memórias.

4.2.1 Generalização do layout normal

Para simplificar a apresentação da construção da árvore-w, forçamos que $|\mathcal{S}| = 2^k$ e que k fosse um múltiplo de um inteiro positivo h. Lembre-se que a altura da árvore-w é $k/h + 1$ sendo a altura de cada bloco $2h$.

Suponha que k não é necessariamente um múltiplo h e também que $|\mathcal{S}|$ não é necessariamente uma potência de 2. Então escolhemos um inteiro positivo k tal que $2^{k-1} < |\mathcal{S}| \leq 2^k$. Assim, os blocos do nível 1 até o nível $\lfloor (2k-1)/2h \rfloor$ tem altura $2h$.

No nível $\lfloor (2k-1)/2h \rfloor + 1$, os blocos podem ter altura distintas. Defina o operador $k \bmod h$ como o resto da divisão inteira entre k e h. Defina também $x = h$ se $k \bmod h = 0$, caso contrário, faça $x = k \bmod h$. Seja $r = \left\lfloor \left(2^k - |\mathcal{S}|\right) / 2^{x-1} \right\rfloor$ e $\ell = \lceil (|\mathcal{S}| - r \cdot 2^{x-1}) / 2^x \rceil$. Neste nível, os ℓ blocos à esquerda tem altura $2x$ e os r blocos mais à direita tem altura $2x - 2$.

Perceba ainda que ℓ-ésimo bloco mais à esquerda pode ter entre 2^{x-1} e 2^x (inclusive) filhos. Neste último caso, a subárvore associada ao ℓ-ésimo bloco mais à esquerda é balanceada.

Finalmente, as folhas estão no próximo nível (que não são associadas a qualquer bloco).

A seguir, um exemplo para ilustrar essa configuração da árvore-w. Se $|\mathcal{S}| = 59$ e $h = 3$ então $k = 6$ e $x = 3$. Assim, os blocos do nível 1 tem altura 6; no nível 2, os 7 blocos mais à esquerda tem altura 6 e o bloco mais à direita tem altura 4; perceba que o sétimo bloco mais à esquerda tem 7 filhos apenas, pois o bloco mais à direita tem 4 filhos e os 6 blocos mais à esquerda tem 8 filhos cada ($59 - 6 \cdot 8 - 1 \cdot 4 = 7$); finalmente, no nível último nível, encontram-se as 59 folhas.

4.2.2 Layout escalado

Outra maneira de reduzir o efeito de redundância em cascata, além dos nós-drenagem, é gradualmente alterar a altura de cada bloco da árvore-w da seguinte forma. A altura da raiz é fixada em $2h = 2$; incrementa-se h em 1 unidade e a altura do bloco do nível seguinte é $2h = 4$; incrementa-se novamente h em 1 unidade e a altura do bloco do nível seguinte é $2h = 6$ e; assim por diante. Esse layout de árvore é chamado de *Layout Escalado* (Figura 4.7). Esse layout pode ser estendido para aceitar um parâmetro inteiro positivo *scale* da seguinte forma. Os blocos dos níveis 1 ao *scale* tem altura fixada em $2h = 2$; incrementa-se h em 1 unidade e os blocos dos próximos *scale* níveis tem altura $2h = 4$ e; assim por diante.

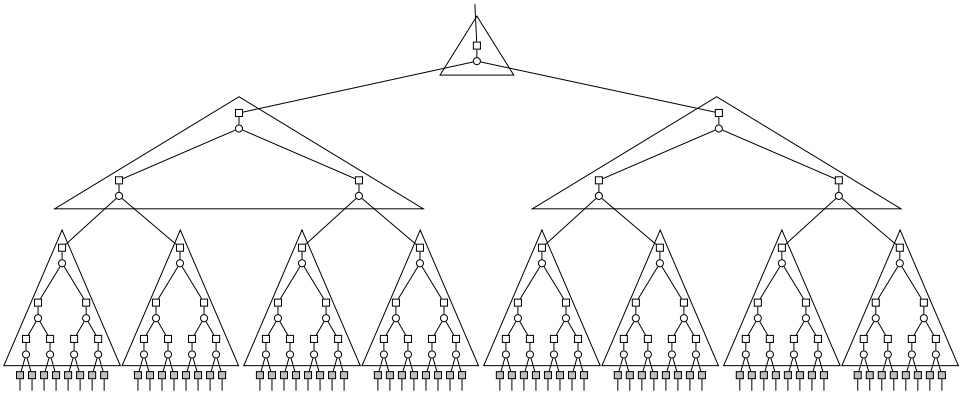


Figura 4.7: Árvore-w com layout escalado – a altura dos blocos são incrementados em 2 ($scale = 1$) a cada nível.

Este layout é justificado pelo comportamento do tamanho das descrições codificadas dos nós-w não-delimitados da árvore binária $\mathcal{T}^f$ do primeiro estágio da construção da árvore-w. Coletamos os dados de vários grafos web (arabic-2005, cnr-2000, eu-2005, in-2004, indochina-2004, it-2004, sk-2005, uk-2002, uk-2005 e webbase-2001) e apresentamos seu tamanho normalizado médio por nível na Figura 4.8. Observamos uma distribuição geométrica e, assim, realizamos uma regressão linear nos valores logarítmicos do tamanho médio normalizado ($length$) por nível (ℓ) da árvore binária $\mathcal{T}^f$. Obtivemos $length(\ell) = 1,1239 \cdot (0,6362)^{\ell}$ com uma coeficiente de correlação de 0.996884. Isso significa que nós-w dos níveis superiores têm tamanhos maiores do que dos níveis inferiores.

O número de acessos à memória externa para acessar uma folha da árvore-w, com layout escalado e parâmetro *scale*, é igual à altura da árvore-w denotado por H. Seja $2x$ a altura do bloco mais inferior desta árvore-w. Então,

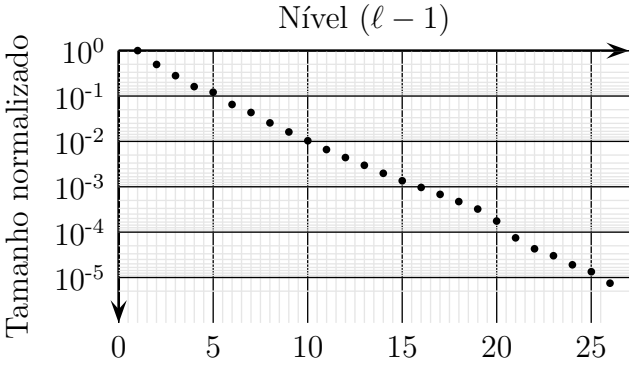


Figura 4.8: Regressão linear do tamanho médio das descrições codificadas dos nós-w por nível $(1, \ldots, \ell)$ da árvore binária $\mathcal{T}^f$.

x é determinado da seguinte forma:

$$scale \cdot (1 + 2 + \ldots + x) = \log_2 |\mathcal{S}| \quad \Rightarrow \quad x = O\left(\sqrt{\frac{\log_2 |\mathcal{S}|}{scale}}\right).$$

Assim, temos que a altura da árvore-w é dado por:

$$H = scale \cdot x + 1 = O\left(\sqrt{scale \cdot \log_2 |\mathcal{S}|}\right).$$

Perceba que a árvore-w com layout escalado e $scale = \log_2 |\mathcal{S}|$ é igual à árvore-w com layout normal e altura do bloco $2h = 2$.

Escolher o valor apropriado do parâmetro *scale* é essencial para permitir melhor um balanceamento entre o número de ponteiros e o tamanho máximo das descrições. O parâmetro *scale* deve ser inversamente proporcional ao tamanho máximo de um bloco L_{block}. Perceba que para $scale = 1$, por exemplo, o número de nós em um bloco rapidamente aumenta e o custo das descrições codificadas ultrapassam rapidamente o limite L_{block}. Assim, quanto maior L_{block} o parâmetro *scale* pode ser menor.

O parâmetro *scale* também deve ser diretamente proporcional à inclinação I da função $\log(length(\ell))$. Por exemplo, a inclinação da função $\log(length(\ell)) = -0,2\ell + 0,05$ da Figura 4.8 é $I = -0,2$. Uma inclinação próximo a 0 significa que o tamanho dos blocos não diminui significativamente à medida que o nível aumenta; se inclinação estiver distante de 0 então o tamanho dos blocos diminui significativamente à medida que o nível aumenta. Assim, quanto menor a inclinação, o parâmetro *scale* pode ser menor também.

Portanto, parece razoável determinar *scale* como sendo diretamente proporcional à c^I, onde $c > 1$ é uma constante, e inversamente proporcional à $\log_2(L_{block})$, ou seja, $scale = O\left(c^I / \log_2(L_{block})\right)$. Agora, a altura da árvore-w

é determinado por:

$$H = O\left(\sqrt{c^I \log_{L_{block}} |\mathcal{S}|}\right).$$

Comparado com a altura da árvore-w com layout normal, dado por $O\left(\log_{L_{block}} |\mathcal{S}|\right)$, notamos que a altura da árvore-w com layout escalado é assintoticamente menor. Isso implica na execução de consultas mais rápida na medida em que $|\mathcal{S}|$ cresce.

Como trabalho futuro, um procedimento pode ser feito para calcular automaticamente o parâmetro *scale* determinando-se o valor da função $length(\ell)$ para cada nível ℓ no primeiro estágio da construção da árvore-w. Em seguida, faz-se uma regressão linear na função $\log length(\ell)$ para determinar a inclinação I e, assim, o parâmetro *scale* pode ser usado no próximo estágio da construção da árvore-w.

Em resumo, o layout escalado reserva mais espaço para nós dos níveis superiores o que acarreta em uma descrição mais detalhada dos conjuntos e, conseqüentemente, evita o efeito de redundância em cascata. Outra vantagem deste layout é de reduzir a altura da árvore-w, implicando na redução do tempo de execução das consultas, sem adicionar complexidade na sua construção. Finalmente, seria interessante saber qual comportamento o layout escalado acarretaria em outros tipos de aplicações e de tipos de árvores como, por exemplo, as árvores RD (Hel94).

4.2.3 Layout Cache-Oblivious

Um aspecto importante para a construção de estruturas de dados é considerar a hierarquia de memória em que eles serão armazenados para a execução de suas operações como, por exemplo, o cache L1, L2, memória principal (RAM), memória externa (disco), etc. Muitos modelos computacionais tentam de capturar os efeitos da hierarquia de memória em tempo de execução dos algoritmos. "Com esses modelos, o programador deve prever qual o nível da hierarquia de memória é o gargalo. Por exemplo, uma *b-tree* que foi ajustada para executar em disco tem fraco desempenho na memória principal"(Ben05). Vários trabalhos exploram hierarquias de vários níveis de memória (modelo de memória *cache-oblivious*) enquanto outros se concentram em hierarquias de memória de dois níveis (modelo de memória externa). Introduzimos ambos os modelos a seguir.

O *modelo de memória externa* é um modelo de uma hierarquia de memória de dois níveis. Informalmente, ele mede o número de transferências entre o disco e a RAM realizada por um algoritmo. Este modelo consiste de uma memória interna de tamanho M e um disco para armazenar todos os dados

restantes. O algoritmo pode transferir blocos contíguos de dados de tamanho L_{block} para ou do disco a um custo unitário.

O *modelo de cache-oblivious*, no entanto, é um modelo de hierarquias de memória de vários níveis. Informalmente, ele mede o número de transferências entre cada par adjacente de memória. Este modelo analisa a estrutura de dados da mesma forma que no modelo de memória externa, mas a estrutura de dados não é explicitamente parametrizada por M ou L_{block} porque eles são desconhecidos. Assim, a análise é válida para M e L_{block} arbitrários, em particular, para todos os M's e L_{block}'s encontrados em cada nível da hierarquia de memória.

Para impor a localidade de dados para a árvore-w em uma hierarquia de memórias de vários níveis, podemos usar o layout *van Emde Boas*. Ele divide a árvore binária $\mathcal{T}$ em subárvores de altura de k (supondo k ser par). Isso divide a árvore binária $\mathcal{T}$ em uma subárvore superior $\mathcal{T}_w$, onde w é a raiz de $\mathcal{T}$ e vários filhos $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_r$. O layout é obtido recursivamente aplicando este procedimento para a subárvore $\mathcal{T}_w$ e para cada filho $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_r$. Referimos à (Ben05) para obter mais detalhes sobre a construção.

Vale ressaltar que as descrições codificadas de cada subárvore recursiva são armazenadas em um bloco de memória contíguo. No entanto, devemos acrescentar mais ponteiros para acessar aleatoriamente as subárvores recursivas. Finalmente, o layout *cache-oblivious* estático é ótimo para busca (acessar aleatoriamente uma folha), no pior caso.

4.3 Execução de Consultas

Nesta seção, mostramos como a árvore-w suporta a execução de consultas básicas (leitura completa, leitura aleatória e leitura aleatória em lote) e de consultas avançadas (superconjunto, subconjunto, igualdade, intersecção, *in-link*, aresta recíproca e *hub* e autoridade) usadas por várias aplicações.

Para facilitar a apresentação, executamos as consultas em uma árvore-w com layout normal tal que cada bloco tem uma altura positiva fixa $2h$, ou seja, cada bloco tem 2^h filhos. Além disso, partimos do princípio que a coleção $\mathcal{S}$ tem 2^k conjuntos e que k é um inteiro positivo múltiplo de h. Isto implica que a árvore-w tem profundidade $k/h + 1$ tal que a raiz está no nível 1 e as folhas estão no nível $k/h + 1$.

A árvore-w está armazenada em uma memória externa denotada por $\mathcal{M}$. Podemos imaginar a memória como um fluxo de bits em que as descrições e ponteiros da árvore-w são armazenados. Cada bloco b da árvore-w é armazenado em alguma região contígua $\mathcal{M}_b$ da memória $\mathcal{M}$. Perceba que a memória

$\mathcal{M}$ é simplesmente a serialização dos fluxos $\mathcal{M}_b$ de todos os nós b (blocos e folhas) da árvore-w.

Finalmente, é útil definirmos o conceito de caminho que será utilizado na apresentação da maioria das consultas suportadas pela árvore-w. Informalmente, um caminho é simplesmente a seqüência de nós da árvore-w ao percorrer de sua raiz até um determinado nó. O nó pode ser um bloco ou uma folha. Apresentamo-lo formalmente a seguir.

Definição 4.3 *Um caminho $\mathcal{P}_u$ é uma seqüência de nós $(u_1, u_2, \ldots, u_n)$, com $n \leq k/h$, da árvore-w que são visitados quando atravessamos de sua raiz até o nó u. Para um nó u_i, o índice i refere-se ao nível do nó na árvore-w. Assim, o primeiro nó u_1 do caminho é sempre a raiz da árvore-w e os próximos nós do caminho são os nós dos próximos níveis consecutivos. Para cada nível $\ell = 1, 2, \ldots, n-1$, se o nó $u_{\ell+1}$ é o j-ésimo filho de u_ℓ então definimos $\mathcal{P}_u^\ell = j$.*

A seguir, apresentamos a execução de consultas básicas (leitura completa, leitura aleatória e leitura aleatória em lote) na árvore-w.

4.3.1 Consultas básicas

Apresentamos aqui três consultas básicas: *leitura completa*, *leitura aleatória* e *leitura aleatória em lote*. Essas consultas são usadas pela maioria das aplicações de grafo web (Pag98) (Kle99a).

Recuperando um bloco simples Descrevemos o procedimento RecoverSingleBlock para ler, da memória $\mathcal{M}$, um único bloco da árvore-w. Esse procedimento é usado por todas as consultas desta seção.

No terceiro estágio da construção da árvore-w (Seção 4.1.3), construímos um bloco b a partir de uma subárvore T_d, cuja raiz é o nó-drenagem d da árvore binária T. As descrições codificadas dos nós de b são serializadas na ordem de uma DFS sobre T_d e armazenadas na memória $\mathcal{M}_b$. Os ponteiros para os filhos de b são codificados e armazenados no final de $\mathcal{M}_b$.

Primeiro, descrevemos os procedimentos RecoverCollection que retorna a coleção dos conjuntos associados às arestas-filho das folhas de T_d e, em seguida, descrevemos o procedimento RecoverPointers que retorna os ponteiros para os filhos do bloco.

RecoverCollection recebe como parâmetro de entrada o conjunto C e funciona da seguinte forma. Primeiro, criamos um nó-drenagem d e associamos o conjunto C à aresta-pai de d. Executamos a primitiva *unmerge* no nó-drenagem d, com o conjunto C como parâmetro de entrada, que retorna o

conjunto C'. Associamos C' à aresta-filho de d. Em seguida, criamos um nó-w w e o fazemos filho de d. Executamos a primitiva *unmerge* no nó-w w, com o conjunto C' como parâmetro de entrada, que retorna os conjuntos A e B. Associamos $A(B)$ à aresta-filho à esquerda(direita) de w, respectivamente. O procedimento atua de acordo com os seguintes casos:

1) Suponha que w é uma folha de T_d. Então, o procedimento apenas retorna a coleção $\{A, B\}$;

2) Caso contrário, o procedimento é chamado recursivamente para o conjunto $A(B)$ que retorna a coleção $\mathcal{S}'(\mathcal{S}'')$. Finalmente, o procedimento retorna a coleção $\mathcal{S}' \cup \mathcal{S}''$.

RecoverPointers é descrito a seguir. Para $i = 1, 2, \ldots, 2^h - 1$, o procedimento decodifica o ponteiro P_i que, na verdade, é o tamanho da subárvore $\mathcal{W}_i$ (definida na Seção 4.1.3). Seja p a posição inicial da memória $\mathcal{M}_b$ na memória $\mathcal{M}$. Para $i = 1, 2, \ldots, 2^h$, a posição inicial p_i da memória associada ao i-ésimo filho de b na memória $\mathcal{M}$ é dado por:

$$p_i = p + |\mathcal{M}_b| + \sum_{j=1}^{i-1} P_j, \tag{4-1}$$

onde $|\mathcal{M}_b|$ é o tamanho da memória $\mathcal{M}_b$ em bits. O procedimento retorna as posições iniciais $(p_1, p_2, \ldots, p_{2^h})$.

Finalmente, apresentamos o procedimento RecoverSingleBlock(C) que recebe como parâmetro de entrada o conjunto C e recupera o bloco b. Mantemos um ponteiro para indicar a posição atual na memória $\mathcal{M}_b$ que assumimos estar agora no início dele. Primeiro, o procedimento executa RecoverCollection com o conjunto C como entrada que retorna a coleção dos conjuntos associados com as arestas-filho das folhas da subárvore associada à b. O ponteiro da memória $\mathcal{M}_b$ está agora na posição inicial dos ponteiros codificados de b. Em seguida, o procedimento executa RecoverPointers que retorna os ponteiros para os filhos de b.

Consulta de leitura completa

A consulta de leitura completa consiste em retornar todos os conjuntos da coleção $\mathcal{S}$ na ordem do índice dos conjuntos $s_i \in \mathcal{S}$, ou seja, na ordem lexicográfica das URLs associadas a cada conjunto. Para isso, ele recupera todos os blocos e folhas da árvore-w na ordem de uma DFS sobre a árvore-w.

Primeiro, descrevemos o procedimento RecoverTree(C, W) que recebe como parâmetros de entrada um conjunto C e uma árvore 2^h-ária W. O

procedimento recupera os conjuntos de $\mathcal{S}$ associados às arestas-filho das folhas de W e funciona recursivamente da seguinte forma. Seja b a raiz de W. A memória $\mathcal{M}_b$ começa na posição atual da memória $\mathcal{M}$. Assim, chamamos o procedimento RecoverSingleBlock com conjunto C como parâmetro de entrada para ler o bloco b e recuperar a coleção $\{C_i \mid i = 1, 2, \ldots, 2^h\}$ e as posições $(p_1, p_2, \ldots, p_{2^h})$. No entanto, essas posições não são usadas aqui e ignoradas. O ponteiro da memória $\mathcal{M}$ é colocado na posição referente ao final de $\mathcal{M}_b$. Para $i = 1, 2, \ldots, 2^h$, o procedimento segue de acordo com os seguintes casos:

1) Suponha que b não é uma folha de W. Seja W_i a árvore com raiz no i-ésimo filho de b. Então, chamamos recursivamente esse procedimento com o conjunto C_i e a subárvore W_i como parâmetros de entrada;

2) Caso contrário, b é uma folha de W, ou seja, é um nó-drenagem não-delimitado. Então, associamos o conjunto C_i à aresta-pai de b. Executamos a primitiva *unmerge* em b que retorna o conjunto C'. Finalmente, acrescentamos C' à coleção $\mathcal{S}$.

Perceba que podemos ignorar os ponteiros porque, depois de recuperar um nó (bloco ou folha), o ponteiro na memória $\mathcal{M}$ já está na posição inicial da memória do próximo nó. Portanto, a organização física da árvore-w é otimizada para esta consulta de forma que é necessário apenas uma busca sequencial em $\mathcal{M}$ sem qualquer acesso aleatório. Cada nó é acessado e transferido à memória principal apenas uma vez. Além disso, mantemos no máximo k/h blocos e 1 folha na memória principal a cada instante.

A consulta de leitura completa obtém todos os conjuntos de $\mathcal{S}$ associados à aresta-filho das folhas da árvore-w $\mathcal{W}$. Para executar a consulta, colocamos o ponteiro na posição 0 de $\mathcal{M}$, lemos o cabeçalho que é a cardinalidade da coleção $\mathcal{S}$ e simplesmente executamos RecoverTree($\{1, 2, \ldots, |\mathcal{S}|\}, \mathcal{W}$).

Consulta de leitura aleatória

Dado $x \in \{1, 2, \ldots, |\mathcal{S}|\}$, a consulta de leitura aleatória retorna o conjunto s_x da coleção $\mathcal{S}$. Para isso, não precisamos recuperar todos os blocos da árvore-w mas apenas os nós do caminho $\mathcal{P}_d$ que começa na raiz da árvore-w e vai até o nó-drenagem não-delimitado d tal que s_x está associado à aresta-filho de d.

Primeiro, descrevemos o procedimento RecoverPath(C, W, x) que recebe um conjunto C, uma árvore W e um inteiro positivo x como parâmetros de entrada. O procedimento recupera o conjunto $s_x \in \mathcal{S}$ associado à aresta-filho

de uma das folhas de W e funciona recursivamente da seguinte forma. Seja b a raiz de W. A memória $\mathcal{M}_b$ começa na posição atual de $\mathcal{M}$. Assim, chamamos o procedimento RecoverSingleBlock com conjunto C como parâmetro de entrada para ler o bloco b e recuperar a coleção $\{C_i \mid i = 1, \ldots, 2^h\}$ e as posições $(p_1, p_2, \ldots, p_{2^h})$. Considere agora o caminho $\mathcal{P}_b$ para determinar o j-ésimo filho de b que devemos visitar. Seja ℓ o nível de b na árvore W. Pela Definição 4.3, o valor de j é exatamente $\mathcal{P}_b^\ell$ e é dado por:

$$j = \left\lceil \frac{x - \sum_{n=1}^{\ell-1} \left(2^h\right)^{k/h-n} \left(\mathcal{P}_b^n - 1\right)}{\left(2^h\right)^{k/h-\ell+1}} \right\rceil . \tag{4-2}$$

Assim, colocamos o ponteiro de $\mathcal{M}$ na posição p_j. Lembre-se que C_j é o conjunto associado à j-ésima aresta-filho de b. O procedimento atua de acordo com os seguintes casos:

1) Suponha que b não é uma folha de W. Seja W_j a árvore com raiz no j-ésimo filho de b. Então, chamamos recursivamente este procedimento com o conjunto C_j, a árvore W_j e o inteiro positivo x como parâmetros de entrada;

2) Caso contrário, b é uma folha de W, ou seja, um nó-drenagem não-delimitado. Então, associamos o conjunto C_j à aresta-pai de b. Executamos a primitiva *unmerge* em b que retorna o conjunto s_j da coleção $\mathcal{S}$. Neste caso, o valor de j é igual ao de x.

A consulta de leitura aleatória obtém o conjunto s_x da coleção $\mathcal{S}$ associado à aresta-filho de uma das folhas da árvore-w $\mathcal{W}$. Para executar a consulta, colocamos o ponteiro de $\mathcal{M}$ na posição 0, lemos o cabeçalho que é a cardinalidade da coleção $\mathcal{S}$ e simplesmente executamos RecoverPath$(\{1, 2, \ldots, |\mathcal{S}|\}, \mathcal{W}, x)$.

Vale a pena ressaltar que cada nó é acessado e transferido à memória principal apenas uma vez. Além disso, mantemos no máximo k/h blocos e 1 folha na memória principal a cada instante.

Consulta de leitura aleatória em lote

Operações em lote é uma estratégia comum para melhorar o tempo de execução sobre estruturas de dados em memória externa. Isso é especialmente verdadeiro para árvores porque os nós dos níveis superiores são mais acessados que os inferiores. Para a árvore-w, cada consulta de leitura aleatória recupera k/h blocos e 1 folha. A raiz é recuperada uma vez a cada consulta de leitura aleatória. Portanto, se soubermos antecipadamente os conjuntos que precisam ser acessados no lote então podemos evitar a recuperação repetida dos blocos superiores.

Primeiro, descrevemos o procedimento RecoverSeveralPaths(C, F, W) que recebe dois conjuntos C e F e uma árvore W como parâmetros de entrada. O procedimento recupera uma subcoleção $\mathcal{S}' \subseteq \mathcal{S}$. Os elementos do conjunto F são os índices em ordem ascendente dos conjuntos na coleção $\mathcal{S}$ a serem recuperados. O procedimento funciona recursivamente da seguinte forma. Se o conjunto F é vazio então o procedimento pára; caso contrário, ele procede da seguinte forma. Seja o r o menor elemento do conjunto F. Se o conjunto s_r não estiver associado à qualquer aresta-filho das folhas descendentes de W então o procedimento pára; caso contrário, ele funciona da seguinte forma. Seja b a raiz de W. A memória $\mathcal{M}_b$ começa na posição atual de $\mathcal{M}$. Assim, chamamos o procedimento RecoverSingleBlock com conjunto C como parâmetro de entrada para ler o bloco b e recuperar a coleção $\{C_i \mid i = 1, \ldots, 2^h\}$ e as posições $(p_1, p_2, \ldots, p_{2^h})$. Considere agora o caminho $\mathcal{P}_b$ para determinar o j-ésimo filho de b que devemos visitar. Seja ℓ o nível de b na árvore W. Por Definição 4.3, o valor de j é exatamente $\mathcal{P}_b^\ell$ e é dado pela Equação 4-2. Assim, colocamos o ponteiro de $\mathcal{M}$ na posição p_j. Lembre-se que C_j é o conjunto associado à j-ésima aresta-filho de b. O procedimento segue de acordo com os seguintes casos:

1) Suponha que b não é uma folha de W. Seja W_j a árvore com raiz no j-ésimo filho de b. Então, chamamos recursivamente este procedimento com os conjuntos C_j e F e a subárvore W_j como parâmetros de entrada;

2) Caso contrário, b é uma folha, ou seja, um nó-drenagem não-delimitado. Então, associamos o conjunto C_j à aresta-pai de b. Executamos a primitiva *unmerge* em b que retorna o conjunto $s_j \in \mathcal{S}$. Neste caso, o valor de j é igual ao de r. Em seguida, adicionamos o conjunto s_r na subcoleção $\mathcal{S}'$ e fazemos $F = F - \{r\}$. Finalmente, chamamos recursivamente este procedimento com o conjunto C e F e a subárvore W como parâmetros de entrada.

A consulta de leitura aleatória em lote obtém os conjuntos s_r da coleção $\mathcal{S}$, onde $r \in F$, associados às arestas-filho das folhas da árvore-w $\mathcal{W}$. Os elementos do conjunto F são os índices em ordem ascendente dos conjuntos na coleção $\mathcal{S}$ a serem recuperados. Para executar a consulta, colocamos o ponteiro de $\mathcal{M}$ na posição 0, lemos o cabeçalho que é a cardinalidade da coleção $\mathcal{S}$ e simplesmente executamos RecoverSeveralPaths$(\{1, 2, \ldots, |\mathcal{S}|\}, F, \mathcal{W})$.

Vale a pena ressaltar que cada nó é acessado e transferido à memória principal apenas uma vez. Além disso, mantemos no máximo k/h blocos e 1 folha na memória principal a cada instante.

4.3.2 Consultas avançadas

Apresentamos aqui a execução de consultas avançadas suportadas pela árvore-w. Para algumas dessas consultas, o processador de consulta pode otimizar bastante a sua execução usando as informações parciais disponíveis nos nós internos da árvore-w.

Normalmente, aplicações que usam grafos web não têm apenas a coleção $\mathcal{S}$, mas podem ter outras informações associadas a cada página web. Assim, precisamos estender a coleção $\mathcal{S}$ para ser parte de uma tabela, na qual é definida da seguinte maneira.

Definição 4.4 *Seja $\mathcal{D} = \{t_0, t_1, \ldots, t_n\}$ uma tabela de tuplas. Todas as tuplas $t_i \in \mathcal{D}$ tem os atributos $a, b, \ldots, m$, onde os atributos extraem valores dos domínios $D_a, D_b, \ldots, D_m$, respectivamente.*

Para ilustrar uma tabela típica $\mathcal{D}$ que poderia ser usado pelas aplicações de grafos web, usamos o exemplo a seguir. Apresentamos na Tabela 4.1 a tabela $\mathcal{D}$ com 8 tuplas. Cada tupla representa uma página web e eles têm 4 atributos: URL, Conjunto s_i, Conjunto s_i^T e o Conteúdo. A coleção de conjuntos da coluna 'Conjunto s_i' representa a lista de adjacência do grafo web $\mathcal{G}$ apresentado na Figura 4.9. A coleção de conjuntos da coluna 'Conjunto s_i^T' representa a lista de adjacência do grafo web transposto $\mathcal{G}^T$, ou seja, grafo com a mesma matriz de adjacência de $\mathcal{G}$, porém, transposta.

Tabela 4.1: Exemplo de uma tabela típica $\mathcal{D}$ – o grafo web $\mathcal{G}$ é estendido para a tabela $\mathcal{D}$, onde as tuplas são representadas por linhas e os atributos pelas colunas.

i	URL	Conjunto s_i	Conjunto s_i^T	Conteúdo
0	http://www.inf.puc-rio.br/	$\{1\}$	$\{1,4\}$	<html>...
1	http://www.inf.puc-rio.br/alunos/	$\{0,2,3,5\}$	$\{0,3\}$	<html>...
2	http://www.inf.puc-rio.br/prof/	$\{3,6\}$	$\{1,6\}$	<html>...
3	http://www.inf.puc-rio.br/w.html	$\{1,7\}$	$\{1,2,7\}$	<html>...
4	http://www.puc-rio.br/	$\{0\}$	$\{\}$	<html>...
5	http://www.puc-rio.br/grad.html	$\{\}$	$\{5\}$	<html>...
6	http://www.ufpe.br/	$\{2,7\}$	$\{2,7\}$	<html>...
7	http://www.ufpe.br/research/	$\{3,6\}$	$\{3,6\}$	<html>...

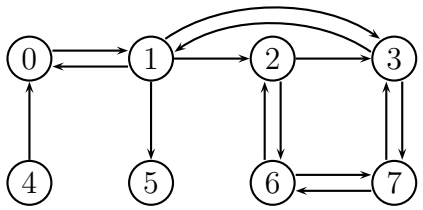


Figura 4.9: Grafo web $\mathcal{G}$ representado pela tabela $\mathcal{D}$ na Tabela 4.1.

Aplicações necessitam recuperar os dados e, por isso, elas precisam avaliar uma *consulta* na tabela para localizar todas as tuplas que satisfaçam uma determinada propriedade. Essa propriedade pode envolver vários atributos. Assim, as aplicações exigem uma linguagem para expressar essa propriedade. A consulta é melhor especificada por um *predicado* definido a seguir.

Definição 4.5 *Dada uma consulta q, o predicado de q define um mapeamento $\mathcal{Q} : D_a \times D_b \times \ldots \times D_m \rightarrow \{true, false\}$. A avaliação de q consiste em encontrar todas as tuplas que satisfazem o predicado e em retornar o conjunto $\{t_i \mid t_i \in \mathcal{D} \wedge q(t_i) = true\}$. O predicado é especificado por uma fórmula seletora f_s definida em termos dos atributos de $\mathcal{D}$.*

Por exemplo, se queremos a lista de adjacência da r-ésima página web da coleção $\mathcal{S}$ então avaliamos a consulta de leitura aleatória. Esta retorna o conjunto de tuplas $\{t_i \mid t_i \in \mathcal{D} \ \wedge \ i = r - 1\}$ com um único elemento t_{r-1} e a lista de adjacência é o valor do atributo 'Conjunto s_i' de t_{r-1}. A consulta de leitura completa, por outro lado, retorna o conjunto $\{t_i \mid t_i \in \mathcal{D}\}$.

Precisamos de algumas definições extras para o caso de um atributo ser valorado, ou seja, seu domínio é uma coleção de conjuntos. O segundo e o terceiro atributos da tabela $\mathcal{D}$, apresentado na Tabela 4.1, são valorados. Para avaliar uma consulta sobre atributos valorados, precisamos da seguinte definição.

Definição 4.6 *Suponha que um dos atributos, digamos a, da tabela $\mathcal{D}$ é valorado com domínio D_a. Seja F o conjunto filtro cujos elementos pertencem ao domínio D_a. Denotamos por um operador de consulta O um elemento do conjunto $\{\subseteq, \supseteq, =, \cap, \cup, \approx_s, |\cdot|\}$, onde $\approx_s$ é o operador de similaridade e $|A|$ é a cardinalidade de um conjunto arbitrário A. Dada uma consulta q, um predicado valorado de q é especificado por uma fórmula seletora f_s definida em termos do atributo valorado a, do conjunto filtro F e do operador O.*

Suponha que uma aplicação precisa encontrar todas as páginas web na tabela $\mathcal{D}$ que apontam para pelo menos 2 páginas web da Universidade PUC-Rio. Assim, precisamos avaliar uma consulta que retorna o conjunto $\{t_i \mid t_i \in \mathcal{D} \wedge |s_i \cap F| \geq 2\}$, onde s_i é o segundo atributo da tupla t_i e o conjunto filtro $F = \{0, 1, 2, 3, 4, 5\}$. A resposta é o conjunto $\{t_1\}$.

Podemos agora fazer uma conexão: se o domínio dos atributos 'Conjunto s_i' e 'Conjunto s_i^T' são as coleções $\mathcal{S}$ e $\mathcal{S}^T$, respectivamente, do grafo web $\mathcal{G}$ e seu transposto $\mathcal{G}^T$ então podemos usar a árvore-w como a estrutura de dados para executar consultas avançadas. Essas consultas retornam um subconjunto de páginas web que tem uma propriedade em comum. Inicialmente, não sabemos quais páginas web tem essa propriedade. Assim, é necessário executar uma

consulta de leitura completa na coleção $\mathcal{S}$ para encontrá-los. Para grafos web muito grandes, esta não é uma solução razoável. A árvore-w, no entanto, pode otimizar a execução de tais consultas. A idéia chave é utilizar as informações parciais sobre os conjuntos da coleção $\mathcal{S}$ disponíveis em cada nó interno da árvore-w.

A seguir, apresentamos o processador de consultas da árvore-w e alguns tipos de consultas que podem ser otimizadas como, por exemplo, consultas orientadas à conjuntos, consultas de arestas recíprocas e consultas de *hubs* e autoridades.

Processador de consultas

Dada uma árvore-w que representa a coleção $\mathcal{S} = \{s_i \mid i = 1, 2, \ldots, |\mathcal{S}|\}$, a avaliação de um predicado de consulta q, especificado por uma função seletora f_s definida sobre conjuntos que representam listas de adjacência, é realizada adaptando-se o procedimento da consulta de leitura completa RecoverTree (Seção 4.3.1) de tal forma a encontrar conjuntos s_i que satisfaçam f_s, ou seja, $f_s(s_i) = true$. Esse procedimento, intitulado ProcessQuery, utiliza as informações parciais disponíveis nos nós da árvore-w sobre os conjuntos de coleção $\mathcal{S}$ de forma a evitar a recuperação de todos eles.

Antes de detalhar esse procedimento, precisamos mostrar como construir uma fórmula f_a (diferente da fórmula f_s) que é utilizada para acelerar o processamento das consultas. Em seguida, descrevemos uma caracterização da árvore-w, apresentada sob forma de teorema, para derivar a forma de como acelerar o processamento das consultas. Por fim, detalharemos o procedimento ProcessQuery com suporte a essa aceleração.

Na proposição a seguir, apresentamos como definir a fórmula f_a.

Definição 4.7 *Dado uma fórmula seletora f_s, dizemos que uma formula f_a é aceleradora em relação à f_s se e somente se para todo par de conjuntos C e C', com $C \supseteq C'$, se $f_s(C')$ então $f_a(C)$.*

Qualquer fórmula f_a que satisfaça as condições acima pode ser usada pelo processador de consultas para acelerá-lo. Vamos dar um exemplo para ilustrar isso. Suponha que a fórmula seletora $f_s(C)$ é dada por $C \subseteq F$, ou seja, o conjunto F tem todos os elementos de C. Assim, a fórmula aceleradora $f_a(C)$ é dada por $|C \cap F| > 0$, ou seja, o conjunto F ter pelo menos um elemento de C. Seja $F = \{1, 2, 3\}$. Se $C' = \{1, 3\}$ então a fórmula seletora $f_s(C')$ é satisfeita e, conseqüentemente, a fórmula aceleradora $f_a(C)$ também é satisfeita para qualquer C, porque $C \supseteq C'$. No entanto, se $C = \{4, 5\}$ então a fórmula aceleradora $f_a(C)$ não é satisfeita e, conseqüentemente, a fórmula seletora $f_s(C')$ também não é satisfeita para qualquer C', porque $C \supseteq C'$.

Agora estamos prontos para apresentar a caracterização da árvore-w, que é apresentado sob forma de teorema, a seguir.

Teorema 4.8 *Seja d_i a folha da árvore-w tal que o conjunto $s_i \in \mathcal{S}$ é associado à sua aresta-filho e seja C_ℓ o conjunto associado à aresta-pai do ℓ-ésimo nó no caminho $\mathcal{P}_{d_i}$, para todos os níveis $\ell = 1, 2, \ldots, k/h + 1$. Dadas as fórmulas seletora f_s e aceleradora f_a, a árvore-w tem as seguintes propriedades:*

($\rightarrow$) Suponha que o conjunto s_i satisfaça a fórmula f_s. Então, o conjunto C_ℓ satisfaz a fórmula f_a, para todos os níveis $\ell = 1, 2, \ldots, k/h + 1$;

($\leftarrow$) Suponha que o conjunto C_ℓ para algum nível ℓ, $1 \leq \ell \leq k/h + 1$, não satisfaça a fórmula f_a. Seja e a aresta associada ao conjunto C_ℓ. Então, nenhum conjunto s de $\mathcal{S}$, tal que a aresta-filho associada à s é descendente de e, satisfaz a fórmula f_s.

Prova. A prova é direta pois temos que $s_i \subseteq C_{k/h+1} \subseteq C_{k/h} \subseteq \ldots \subseteq C_2 \subseteq C_1$ garantido pela construção da árvore-w descrito na Seção 4.1.3. ■

A aceleração é derivada da propriedade ($\leftarrow$) do Teorema 4.8: se o conjunto C_ℓ não satisfaz a fórmula aceleradora f_a então não precisamos recuperar qualquer nó da árvore-w descendente da aresta e, associada ao conjunto C_ℓ.

Agora podemos descrever o procedimento ProcessQuery$(C, W, \mathcal{D}, q)$. Ele recebe um conjunto C, uma árvore W, uma tabela $\mathcal{D}$ e uma consulta q como parâmetros de entrada e retorna um conjunto de tuplas T da tabela $\mathcal{D}$, cujos conjuntos $s_i \in \mathcal{S}$ satisfazem a fórmula seletora $f_s(s_i)$ da consulta q. O procedimento é adaptado do procedimento do RecoverTree (Seção 4.3.1) da seguinte forma:

1) Adicionamos a seguinte condição antes de chamarmos o procedimento RecoverSingleBlock. Se o conjunto C não satisfaz a fórmula aceleradora f_a então o procedimento pára; caso contrário, ele procede para executar RecoverSingleBlock;

2) Reescrevemos o caso 2 da seguinte forma. Caso contrário, b é um nó-drenagem não-delimitado. Se o conjunto C_i satisfaz a fórmula aceleradora f_a então associamos o conjunto C_i à aresta-pai de b. E, em seguida, executamos a primitiva *unmerge* em b que retorna o conjunto C'. Finalmente, se o conjunto C' satisfaz a fórmula seletora f_s então a tupla, cujo conjunto C' pertence a um de seus atributos, da tabela $\mathcal{D}$ é incluída no conjunto de retorno T.

Finalmente, dado uma consulta q e uma tabela $\mathcal{D}$, tal que a árvore-w $\mathcal{W}$ represente os conjuntos $s_i \in \mathcal{S}$ de um dos atributos de $\mathcal{D}$, o processamento da consulta retorna todas as tuplas de $\mathcal{D}$ tal que satisfaçam a fórmula seletora $f_s(s_i)$ da consulta q. Para executar a consulta, colocamos o ponteiro de $\mathcal{M}$ na posição 0, lemos o cabeçalho, que é a cardinalidade da coleção $\mathcal{S}$, e simplesmente executamos ProcessQuery$(\{1, 2, \ldots, |\mathcal{S}|\}, \mathcal{W}, \mathcal{D}, q)$.

Consulta orientada à conjuntos

Apesar de muitos tipos de consultas poderem ser avaliadas na árvore-w, apenas algumas delas podem ser aceleradas. Isso porque podemos não encontrar uma fórmula aceleradora f_a em relação a uma determinada fórmula seletora f_s. Apresentamos nesta seção as fórmulas aceleradoras f_a em relação às fórmulas seletoras f_s expressas pelo operador de consulta $O \in \{\subseteq, \supseteq, =, \cap, |\cdot|\}$ e pelo conjunto filtro F. Denotamos por C_ℓ, para algum nível $\ell = 1, 2, \ldots, k/h + 1$, o conjunto associado à aresta-pai de um dos nós do caminho $\mathcal{P}_{d_i}$ usado no Teorema 4.8.

A consulta de *superconjunto* pergunta por páginas web que apontam para todas as páginas web do conjunto F. Isso significa que o conjunto s_i satisfaz f_s se e somente se $s_i \supseteq F$. Por conseguinte, a fórmula aceleradora f_a em relação a f_s é dada por $C_\ell \supseteq F$.

A consulta de *subconjunto* pergunta por páginas web que apontam para um subconjunto de páginas web do conjunto F. Isso significa que o conjunto s_i satisfaz f_s se e somente se $s_i \subseteq F$. Por conseguinte, a fórmula aceleradora f_a em relação a f_s é dada por $|C_\ell \cap F| > 0$.

A consulta de *igualdade* pergunta por páginas web que apontam para exatamente as páginas web do conjunto F. Isso significa que o conjunto s_i satisfaz f_s se e somente se $s_i = F$. Por conseguinte, a fórmula aceleradora f_a em relação a f_s é dada por $C_\ell \supseteq F$.

A consulta de *intersecção* pergunta por páginas web que apontam para pelo menos k_0 e para no máximo k_1 páginas web do conjunto F. Isso significa que o conjunto s_i satisfaz f_s se e somente se $k_0 \leq |s_i \cap F| \leq k_1$. Por conseguinte, a fórmula aceleradora f_a em relação a f_s é dada por $|C_\ell \cap F| \geq k_0$.

Dado um operador O, um filtro F, a consulta q consistindo da fórmula $f_s(s_i, O, F)$ e uma tabela $\mathcal{D}$, tal que a árvore-w $\mathcal{W}$ represente os conjuntos $s_i \in \mathcal{S}$ de um dos atributos de $\mathcal{D}$, o processamento da consulta retorna o conjunto de tuplas $\{t_i \mid t_i \in \mathcal{D} \wedge f_s(s_i, O, F) = true\}$. Para executar a consulta, colocamos o ponteiro de $\mathcal{M}$ na posição 0, lemos o cabeçalho, que é a cardinalidade da coleção $\mathcal{S}$, e simplesmente executamos ProcessQuery$(\{1, 2, \ldots, |\mathcal{S}|\}, \mathcal{W}, \mathcal{D}, q)$.

Essas consultas são suficientes para executar quaisquer operações de grafo

apresentadas em (Rag03). Por exemplo, a consulta *in-link* consiste em retornar um conjunto de páginas web que aponta para uma determinada página web t_i, supondo que o grafo web transposto não está disponível. Seja s_i o conjunto de *out-links* de t_i. Assim, podemos simplesmente avaliar a consulta de subconjunto com $F = \{i\}$.

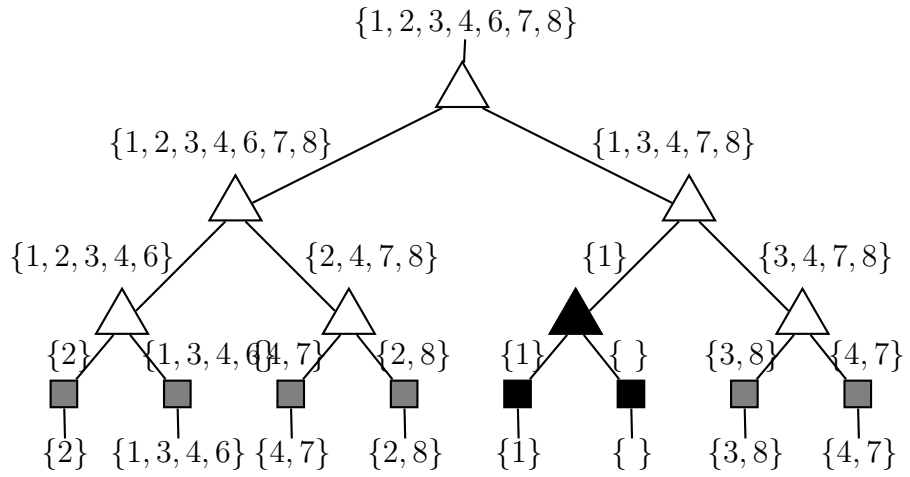
Consulta de arestas recíprocas

A consulta de arestas recíprocas consiste em retornar um conjunto de pares de páginas web (a, b) tal que a página web a aponta para b e vice-versa. Essa consulta é usada por aplicações de detecção de webspam e foi usada em (Cas07). No entanto, eles não descreveram seu processador de consulta. Assim, apresentamos um possível processamento de consulta para representações de grafos web que só permitem consultas básicas como segue.

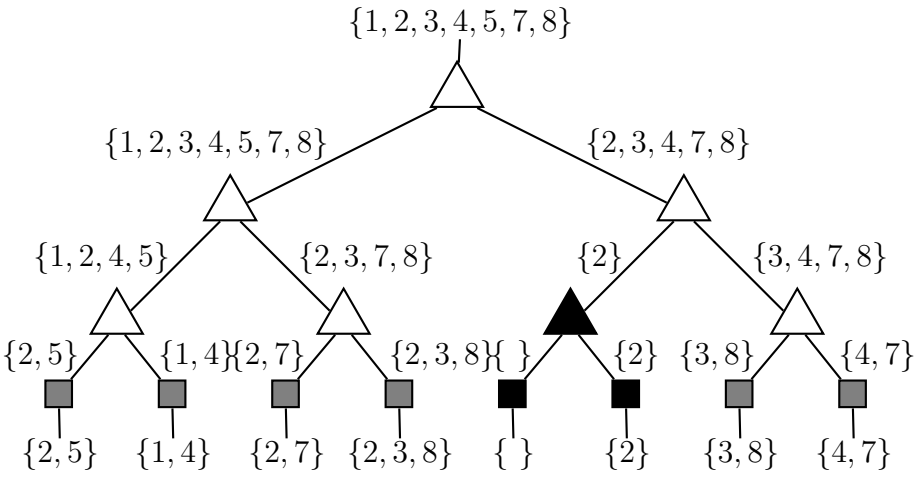
Seja $\mathcal{S} = \{s_i \mid i = 1, 2, \ldots, |\mathcal{S}|\}$ e $\mathcal{S}^T = \{s_i^T \mid i = 1, 2, \ldots, |\mathcal{S}^T|\}$ as coleções de conjuntos que representam a lista de adjacência do grafo web $\mathcal{G}$ e de seu transposto $\mathcal{G}^T$, respectivamente. Seja $\mathcal{W}$ e $\mathcal{W}^T$ as árvores-w que representam as coleções $\mathcal{S}$ e $\mathcal{S}^T$, respectivamente. Realizamos uma consulta de leitura completa em $\mathcal{W}$ e outra em $\mathcal{W}^T$ ao mesmo tempo. Para cada elemento j do conjunto $s_i \cap s_i^T$, a consulta retorna o par (t_i, t_j) se $i < j$ (para evitar o par (t_j, t_i)). Em outras palavras, a consulta retorna o conjunto $\{(t_i, t_j) \mid t_i, t_j \in \mathcal{D} \wedge j \in s_i \cap s_i^T \wedge i < j\}$.

Mais uma vez, podemos usar a árvore-w para otimizar a execução da consulta. Denotamos por C_ℓ, para algum nível $\ell = 1, 2, \ldots, k/h+1$, o conjunto associado à aresta-pai de um dos nós do caminho $\mathcal{P}_{d_i}$ usado no Teorema 4.8. Seja C_ℓ^T o conjunto correspondente de C_ℓ na árvore-w $\mathcal{W}^T$. A fórmula seletora f_s é dada por $j \in s_i \cap s_i^T$ e a fórmula aceleradora f_a em relação a f_s é dada por $|C_\ell \cap C_\ell^T| > 0$. Usamos essa fórmula para acelerar as duas consultas de leitura completa acima. Finalmente, o procedimento ProcessQuery tem que ser adaptado para processar os grafos $\mathcal{G}$ e $\mathcal{G}^T$ ao mesmo tempo. Porém, é uma adaptação simples e os detalhes não são apresentados aqui.

Para ilustrar a execução da consulta de aresta recíproca, apresentamos a árvore-w $\mathcal{W}$ e $\mathcal{W}^T$ na Figura 4.10 para o grafo web apresentado na Figura 4.9. Os blocos pretos não são recuperados, assim como, nenhum de seus descendentes. Essa otimização representa economia de acesso à memória externa durante a execução da consulta. O conjunto de pares retornado pela consulta é $\{(1, 2), (2, 4), (3, 7), (4, 8), (7, 8)\}$.



4.10(a): Árvore-w $\mathcal{W}$ do grafo web $\mathcal{G}$ apresentado na Figura 4.9.



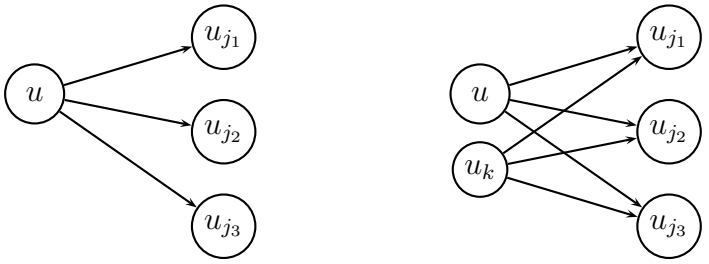
4.10(b): Árvore-w $\mathcal{W}^T$ do grafo web transposto $\mathcal{G}^T$ apresentado na Figura 4.9.

Figura 4.10: Consulta de aresta recíproca – blocos e folhas são representados por triângulos e quadrados cinza, respectivamente. Os nós pretos não são recuperados pela consulta.

Consulta de hubs e autoridades

A consulta de *hubs* e autoridades consiste em retornar um conjunto de subgrafos bipartidos e completos (ou densos) de um grafo web. Essa consulta é usada por algoritmos de *ranking* (Kle99a) e para encontrar comunidades ocultas (Kum99). Os nós que apontam são os *hubs* e os nós que são apontados são as *autoridades* (Figura 4.11).

Apresentamos dois procedimentos para encontrar subgrafos bipartidos completos em um grafo web usando a árvore-w $\mathcal{W}$. A estratégia adotada é extrair um subgrafo $\mathcal{G}_u$, que inclui o nó $u \in \mathcal{G}$, para todo nó u do grafo web $\mathcal{G}$. Espera-se que o subgrafo $\mathcal{G}_u$ seja muito menor que o grafo $\mathcal{G}$ e que caiba na memória principal. Assim, podemos executar um procedimento, que pode ser encontrado na literatura, para encontrar subgrafos bipartidos em $\mathcal{G}_u$. Todos os



4.11(a): Segundo passo. 4.11(b): Quarto passo.

Figura 4.11: Subgrafo $\mathcal{G}_u$ – os nós u e u_k podem vir a ser os *hubs* e os nós $u_{j_1}, u_{j_2}, \ldots, u_{j_{|s_i|}}$ as autoridades.

subgrafos bipartidos extraídos devem ter o nó u.

Primeiro, assumimos que a coleção $\mathcal{S}^T$ do grafo web transposto $\mathcal{G}^T$ não está disponível. Lembre-se que a coleção $\mathcal{S}$ é parte de uma tabela $\mathcal{D}$. O procedimento funciona da seguinte maneira. Executamos o procedimento RecoverTree($\{1, 2, \ldots, |\mathcal{S}|\}, \mathcal{W}$) para retornar todos os conjuntos $s_i \in \mathcal{S}$. Para cada conjunto $s_i \in \mathcal{S}$, execute os seguintes passos:

1) Seja u o nó de $\mathcal{G}$ que tem o conjunto s_i representando sua lista de adjacência. Então, adicionamos o nó u no subgrafo $\mathcal{G}_u$;

2) Para todo $j \in s_i$, adicionamos o nó u_j ao subgrafo $\mathcal{G}_u$ e adicionamos também uma aresta direcionada do nó u para o nó u_j. Usamos um índice nos nós adicionados u_j apenas para distinguir entre eles $(u_{j_1}, u_{j_2}, \ldots, u_{j_{|s_i|}})$, como apresentado na Figura 4.11(a).

3) Em seguida, executamos uma consulta de intersecção com $F = \{j \mid j \in s_i\}$ e $k_0 = 1$ como parâmetros de entrada na árvore-w $\mathcal{W}$. A consulta retorna o conjunto de tuplas $T = \{t_k \mid t_k \in \mathcal{D} \wedge |s_k \cap F| \geq 1\}$;

4) Para cada tupla $t_k \in T$, adicionamos o nó u_k, cujo conjunto s_k da tupla t_k representa a lista de adjacência do nó u_k, no subgrafo $\mathcal{G}_u$. Adicionamos também uma aresta direcionada do nó u_k para cada nó u_j tal que $j \in s_k$ (Figura 4.11(b));

5) Finalmente, executamos um procedimento em memória principal para encontrar subgrafos bipartidos completos em $\mathcal{G}_u$ que tem o nó u como *hub*. Note que nem todos os nós $u_k(u_j)$ serão *hubs*(autoridades).

Vale a pena ressaltar que o quinto passo pode ser exponencial se o procedimento em memória principal enumerar *todos* os subgrafos bipartidos completos em $\mathcal{G}_u$. No entanto, uma melhor caracterização de $\mathcal{G}_u$ ou dos

subgrafos bipartidos a serem extraídos pode levar a algoritmos mais eficientes como em (Epp1994) e (Mub2010).

Agora, suponha que a coleção $\mathcal{S}^T$ está disponível. Usamos um procedimento igual ao anterior, exceto que a consulta de intersecção no passo 3 acima pode ser substituído por uma consulta de leitura aleatória em lotes com o conjunto filtro F como parâmetro de entrada na árvore-w $\mathcal{W}^T$. A consulta retorna uma subcoleção do $\mathcal{S}^T$ correspondente para o conjunto T do passo 3.

4.4 Resultados Experimentais

Nesta seção, apresentamos os resultados de uma série de experimentos realizados na árvore-w que duraram cerca de nove meses para executar, incluindo a re-execução de alguns deles e a correção de erros de implementação.

O modelo computacional utilizado para a execução da árvore-w foi o modelo de memória externa (Seção 4.2). Portanto, realizamos cada experimento com a árvore-w armazenada no disco. Somente uma quantidade mínima de memória principal foi usada para as estruturas de dados intermediárias, tais como os nós recuperados de um caminho ou ainda os nós utilizados para cache. Por outro lado, a maioria das representações de grafos web são projetadas para executar suas operações na memória principal. Portanto, *estamos principalmente interessados em avaliar o comportamento da árvore-w no disco para determinar até onde se distancia de uma representação de memória principal.*

Vários autores afirmaram que representações de grafos web precisam ser armazenados na memória principal pois poderia ser proibitivo executar algoritmos para grafos em memória externa (Sue01) (Bue08), uma vez que esses algoritmos não são *disk-friendly* (Cla07) e por causa do alto custo das operações em disco. Portanto, *estamos também interessados em verificar a viabilidade de representações em memória externa.*

No entanto, dada a quantidade de parâmetros das representações, incluindo as variáveis tecnológicas, linguagem de desenvolvimento e humanas, uma comparação justa parece difícil de fornecer. Assim, *nossa abordagem é mostrar que a árvore-w é competitiva o suficiente para ser considerado entre as outras representações, incluindo às destinadas à memória principal.* Somente com uma combinação de implementação eficiente, estratégias de cache, hardware, ajuste de parâmetro, etc, seria possível determinar a melhor representação para cada aplicação.

Os experimentos foram realizados em um computador com processador Intel i7 de 2,66 GHz (usamos apenas um núcleo), uma memória RAM DDR3-1.333 MHz de 8 GB e um disco rígido SATA-2 de 7200 RPM e 1 TB de

capacidade e executados no Windows 7 Professional 64 bits. O sistema de arquivo é NTFS e formatado usando páginas de disco com tamanho 4 KB. O conjunto de dados usado nos experimentos está disponível em (Web10) e consiste de 24 grafos web reais (Tabela 4.2) coletados da Web.

Tabela 4.2: Base de dados de grafos web reais usados nos experimentos.

Grafo web	**No. Nós**	**No. Arestas**	**No. Arestas por Nó**
Arabic 2005	22,744,080	639,999,458	28.1
CNR 2000	325,557	3,216,152	9.8
EU 2005	862,664	19,235,140	22.2
IN 2004	1,382,908	16,917,053	12.2
Indochina 2004	7,414,866	194,109,311	26.1
IT 2004	41,291,594	1,150,725,436	27.8
SK 2005	50,636,154	1,949,412,601	38.4
UK 2002	18,520,486	298,113,762	16.0
UK 2005	39,459,925	936,364,282	23.7
UK 2006 5	77,741,046	2,965,197,340	38.1
UK 2006 6	80,644,902	2,481,281,617	30.7
UK 2006 7	96,395,298	3,030,665,444	31.4
UK 2006 8	100,751,978	3,250,153,746	32.2
UK 2006 9	106,288,541	3,871,625,613	36.4
UK 2006 10	93,463,772	3,130,910,405	33.4
UK 2006 11	106,783,458	3,479,400,938	32.5
UK 2006 12	103,098,631	3,768,836,665	36.5
UK 2007 1	108,563,230	3,929,837,236	36.1
UK 2007 2	110,123,614	3,944,932,566	35.8
UK 2007 3	107,565,084	3,642,701,825	33.8
UK 2007 4	106,867,191	3,790,305,474	35.4
UK 2007 5	105,896,555	3,738,733,648	35.3
UK union	133,633,040	5,507,679,822	41.2
Webbase 2001	118,142,155	1,019,903,190	8.6
Média	**72,335,110**	**2,359,545,269**	**29.4**

A árvore-w foi implementada usando a linguagem C++ compilada com Microsoft Visual Studio 2010 Ultimate 64 bits. Usamos a tecnologia de memória mapeada (*Memory Mapped*) disponível na maioria dos sistemas operacionais para gerenciar os acessos ao disco. Usamos as seguintes opções da memória mapeada: `FILE_FLAG_NO_BUFFERING` e `FILE_FLAG_WRITE_THROUGH`, que forçam o sistema operacional a não usar *buffering* nem cache; e a opção `SEC_NOCACHE`, que força a memória mapeada para não usar cache. No entanto, o *buffering* e cache do disco rígido foram usados.

Os parâmetros que podemos ajustar para a árvore-w são: codificadores dos ponteiros (Gama ou Golomb); as funções de custo f_w e f_d para nós-w e nós-drenagem, respectivamente, como definidas na Seção 4.1.2; tamanho do bloco ($L_{block} \in \{2\text{ KB}, 4\text{ KB}, 8\text{ KB}\}$); layout da árvore (normal ou escalado); e parâmetro do layout $(1, 2, \ldots, 6)$, para o layout normal o parâmetro é a altura de bloco (h) e para o layout escalado o parâmetro é *scale* (definido na Seção 4.2). Fixamos alguns parâmetros usando valores que fornecem resultados

melhores e consistentes. Usamos a codificação de Golomb como o codificador dos ponteiros e a função de custo para nós-w é $f_w(\cdot) = n_r/n_s$ e $f_d(\cdot) = n_r/n_s^2$ para nós-drenagem, conforme discutido na Seção 4.2. Resumindo, realizamos os experimentos para as $36(= 2 \cdot 3 \cdot 6)$ combinações dos parâmetros restantes em todos os 24 grafos web e seus transpostos.

Comparamos a árvore-w com o *webgraph framework*, proposto em (Bol04a) e disponível em (Frw10). Ele foi implementado usando a linguagem Java e executado usando o *Java Runtime Environment 6*.

A principal técnica utilizada pelo *webgraph framework* é a compressão por referência. Os nós do grafo web são ordenados lexicograficamente de acordo com a URL associada a cada um e atribuído um identificador único a cada nó. Dizemos que a distância entre dois nós é a diferença entre os valores dos identificadores deles. Na compressão por referência, a representação das arestas incidentes de um nó u é baseada nas arestas incidentes de um outro nó v distante a, no máximo, uma janela de tamanho w do nó u. Um segundo parâmetro do *webgraph framework* é a contagem de referência usado para controlar a quantidade de referências que o nó u precisa para ser decodificado. Por exemplo, suponha que o nó u foi codificado com referência ao nó v. O nó v, por sua vez, pode ter sido codificado por referência a um outro nó, digamos t, que pode estar distante de u em $2w$ unidades. O parâmetro de contagem de referência serve para limitar o tamanho dessa cadeia.

Os parâmetros que ajustamos para o *webgraph framework* são: tamanho da janela (7, 15, 31, 63); e a contagem de referência (3, 256, ∞). Para os outros parâmetros, usamos seus valores padrões. Consulte (Bol04a) para obter mais explicações sobre outros parâmetros. Não testamos todas as combinações possíveis dos parâmetros mas, para diferentes experimentos, fixamos um parâmetro (digamos tamanho da janela) e variamos o outro (digamos contagem de referência). Resumindo, conduzimos experimentos para 4 combinações, no máximo (4 tamanhos da janela para 1 contagem de referência fixa), e apenas nos 24 grafos web. Os grafos transpostos não foram incluídos nos experimentos para o *webgraph framework*.

Realizamos experimentos de compressão em grafos web para analisar o tempo de compressão e o custo de armazenamento. Também realizamos experimentos de consultas básicas e avançadas para analisar o tempo de execução e o número de acessos no disco. Realizamos experimentos de escalabilidade para analisar o comportamento quando o grafo web aumenta o número de nós e de arestas. Finalmente, analisamos os resultados de cada experimento para o grafo web *uk-union*, já que é o maior grafo em termos de número de nós e de arestas entre 24 grafos web da Tabela 4.2. Os resultados são médias

e/ou desvios-padrões calculados fixando-se um parâmetro (digamos tamanho do bloco) e variando os outros (digamos altura do bloco h e *scale*). A seguir, descrevemos os experimentos e os seus resultados em detalhes.

4.4.1 Experimentos de compressão

Um experimento de compressão consiste em construir uma representação compacta de um grafo web, que está armazenado em um disco usando uma representação não-compacta, e armazená-lo no mesmo disco.

Realizamos experimentos de compressão para medir o tempo de compressão e custo de armazenamento dos 24 grafos web e seus transpostos em todas as 36 combinações de parâmetros. O processo de construção da árvore-w assume que a representação final não caberá na memória principal. Portanto, todas as operações foram realizadas no disco com todas as estruturas de dados armazenadas em disco, bem como as árvores intermediárias usadas para construí-la (Seção 4.1). Os resultados do tempo de compressão, medidos em nanossegundos por aresta, e os resultados do custo de armazenamento, medidos em bits por aresta, da árvore-w são apresentados na Tabela 4.3. Os resultados do tempo e do custo incluem o grafo e os ponteiros usados para permitir acesso aleatório.

Tabela 4.3: Resultados do tempo de compressão e do custo de armazenamento da árvore-w.

Parâmetro	**Valor**	**Tempo (ns/aresta)**		**Custo (bits/aresta)**	
		Média	**Des. Pad.**	**Média**	**Des. Pad.**
Tamanho do bloco	2 KB	617	52	2,62	0,38
	4 KB	586	43	2,41	0,17
	8 KB	556	38	2,30	0,08
Layout da árvore	Normal	596	42	2,51	0,34
	Escalado	576	57	2,44	0,31
Altura do bloco (Layout Normal)	1	621	36	2,33	0,01
	2	586	30	2,29	0,02
	3	554	33	2,31	0,05
	4	574	31	2,40	0,10
	5	604	29	2,59	0,20
	6	641	43	2,81	0,30
Scale (Layout Escalado)	1	656	76	2,94	0,47
	2	600	38	2,46	0,14
	3	582	38	2,36	0,09
	4	541	26	2,30	0,05
	5	538	24	2,29	0,05
	6	536	26	2,27	0,04

O tempo médio global de compressão é de 586 ns/aresta e o desvio-padrão é 50 ns/aresta. Ambas as medidas foram calculadas a partir de todos os 24

grafos web e todas as 36 combinações de parâmetros. Observamos os seguintes fatos:

1) O tempo médio diminui à medida que o tamanho do bloco aumenta;

2) Não há diferença significativa entre os resultados para diferentes layouts;

3) O tempo médio para a árvore com layout normal é menor quando a altura do bloco é $h = 3$, que representa o ponto mínimo do balanceamento entre o tempo de processamento das descrições e dos ponteiros;

4) O tempo médio da árvore com layout escalado diminui na medida em que o parâmetro *scale* aumenta.

Para os grafos web transpostos, o tempo médio global de compressão é 560 ns/aresta e o desvio padrão é 53 ns/aresta. Além disso, eles apresentam o mesmo comportamento de seus grafos web não-transpostos correspondentes.

O custo médio global de armazenamento é de 2,45 bits/aresta e desvio padrão de 0,28 bits/aresta. Ambas as medidas foram calculadas a partir de todos os 24 grafos web e todas as 36 combinações de parâmetros. Observamos os seguintes fatos:

1) O custo médio diminui conforme o tamanho do bloco aumenta, porque, na medida em que o tamanho do bloco aumenta, permite-se armazenar uma descrição codificada maior nos blocos reduzindo o efeito de redundância em cascata;

2) Não há diferença significativa entre os resultados para diferentes layouts;

3) O custo médio para a árvore com layout normal é menor quando $h = 2$, que representa o ponto mínimo do balanceamento entre o custo das descrições e dos ponteiros. A partir de $h = 2$, o custo médio aumenta à medida que a altura do bloco aumenta, porque o número máximo de bits L permitido para cada descrição codificada é reduzida. Lembre-se que L é igual à L_{block} dividido pelo número de nós no bloco. Isto leva a um aumento do efeito de redundância em cascata;

4) O custo médio diminui à medida que o parâmetro *scale* aumenta. Observamos que o layout escalado realmente funciona como esperado uma vez que foi projetado para se aproveitar do comportamento dos grafos web na árvore-w (Seção 4.2). Ou seja, ele aumenta o espaço permitido para os nós superiores e diminui o número de ponteiros nos blocos inferiores.

Para os grafos web transpostos, o custo médio global é 2,18 bits/aresta e o desvio-padrão é 0,11 bits/aresta. Eles apresentam o mesmo comportamento que os grafos não-transpostos.

Observamos uma forte correlação: *à medida que o custo de armazenamento diminui, o tempo de compressão diminui*. Essa correlação não é algo típico em algoritmos de compressão. Para obter mais compressão, os algoritmos tem que aumentar o número de padrões detectados acarretando em um aumento no tempo de processamento. No entanto, para a árvore-w, o tempo de processamento e o custo de armazenamento diminui na medida em que encontramos melhores pontos de balanceamento entre o número de ponteiros e o espaço permitido para as descrições codificadas. Perceba que quanto melhor o balanceamento, neste caso, menor a quantidade de dados (ponteiros, descrições e elementos redundantes) para processar e armazenar em disco. Os padrões detectados, porém, pelos esquemas das codificações das descrições e dos ponteiros permanecem os mesmos. Finalmente, observamos que a árvore com layout escalado é mais previsível.

Medimos o tempo de compressão do *webgraph framework* usando o parâmetro de contagem de referência fixo em ∞ e variando o parâmetro de tamanho da janela. Variamos apenas o parâmetro de tamanho de janela porque é o que mais afeta o tempo de compressão. Observamos que, apesar do *webgraph framework* ser uma representação em memória principal, parece que os dados compactados são armazenados em um buffer na memória principal. Quando o buffer está cheio, ele armazena-o seqüencialmente no disco. Os resultados do tempo de compressão do *webgraph framework* são apresentados na Tabela 4.4. Observamos que o tempo degrada-se muito com o aumento do tamanho da janela.

Tabela 4.4: Tempo de compressão do *webgraph framework*, em ns/aresta, com a contagem de referência fixado em ∞.

Tamanho da janela	**Média**	**Des. Pad.**
7	237	39
15	398	71
31	722	190
63	1.556	219

Os resultados do custo de armazenamento do *webgraph framework* são apresentados na Tabela 4.5. Os resultados incluem o grafo e os ponteiros (*offset vector*) usados para permitir acesso aleatório. Para esse critério, ambos os parâmetros afetam o custo de armazenamento, mas fixamos o tamanho da janela em 7 e variamos a contagem de referência. Observamos que o custo diminui com o aumento da contagem de referência.

Aumentamos o tamanho da janela para reduzir ainda mais o custo do *webgraph framework*. Testamo-lo com o tamanho da janela fixado em 63 e a contagem de referência fixado em ∞. Para este experimento, o custo médio é 2,29 bits/aresta e o desvio-padrão é 0,88 bits/aresta. Observamos os seguintes fatos:

1) A maioria dos 24 grafos web teve seus tamanhos finais reduzidos quando o tamanho da janela aumentou, exceto para o grafo web *eu-2005*, na qual aumentou de 4,86 para 5,66 bits/aresta;

2) Além disso, observamos que os desvios padrões dos custos do *webgraph framework* de cada combinação de parâmetros testada são maiores do que os da árvore-w (apresentados na Tabela 4.3).

Comparando a árvore-w e o *webgraph framework* em relação ao tempo de compressão e o custo de armazenamento, notamos que a árvore-w apresenta um bom equilíbrio entre ambos os critérios. De fato, quanto menor o custo de armazenamento menor é o tempo de compressão. O *webgraph framework*, no entanto, tem que sacrificar o tempo de compressão para diminuir o custo e vice-versa. Como veremos mais à frente, ele também tem que sacrificar o tempo de consulta de leitura aleatória para atingir os mesmos custos de armazenamento da árvore-w. De fato, para o *webgraph framework* atingir 2,29 bits/aresta na média, o tempo de compressão fica 2,89 vezes mais lento e o tempo da consulta de leitura aleatória fica 1.663 vezes mais lento em relação aos tempos da árvore-w com layout escalado e $scale = 5$.

Tabela 4.5: Custo de armazenamento do *webgraph framework*, em bits/aresta.

Tamanho da Janela	**Contagem de Referência**	**Média**	**Des. Pad.**
7	3	3,40	0,68
	256	2,64	0,66
	∞	2,63	0,66
63	∞	2,29	0,88

Os resultados do grafo web *uk-union* são resumidos como segue. A construção da árvore-w demorou, em média, 52 minutos na qual o menor tempo foi 43 minutos para o layout escalado 6 com bloco 64 KB e o maior tempo foi de 1 hora e 10 minutos para o layout escalado 1 com bloco 16 KB. A construção da representação *webgraph framework* levou 18, 31 e 51 minutos com o tamanho da janela de 7, 15 e 31, respectivamente. Além disso, o *Virtual Node Miner* (Bue08), que também é uma representação de memória principal, relatou 2 horas e 30 minutos para comprimir o grafo web *uk-union* usando um computador semelhante. O custo médio da árvore-w é 2,19 bits/aresta na

qual o menor é 1,82 bits/aresta para o layout escalado 6 com bloco 64 KB e o maior 4,66 bits/aresta para o layout escalado 1 com bloco 16 KB. O custo do *webgraph framework* é de 2,89, 2,13 e 2,11 bits/aresta com a contagem de referência 3, 256 e ∞, respectivamente, e com tamanho de janela fixado em 7.

4.4.2 Experimentos de consultas básicas

Um experimento de consulta básica consiste em recuperar, do grafo web, todos os nós e arestas seqüencialmente (consulta de leitura completa) e em recuperar 100.000 nós escolhidos aleatoriamente (consulta de leitura aleatória).

Realizamos experimentos de consulta básica nos 24 grafos web e seus transpostos em todas as 36 combinações de parâmetros. Todas as estruturas de dados foram armazenadas em disco. Especificamente para a consulta de leitura aleatória, pré-processamos e armazenamos em memória principal os 2^{15} nós-w e nós-drenagem mais superiores da árvore-w. O resto dos dados foi armazenado em disco. Os resultados das consultas básicas, medidos em nanossegundos por aresta, são apresentados na Tabela 4.6.

Tabela 4.6: Resultados das consultas básicas na árvore-w.

Parâmetro	**Valor**	**Leit. comp. (ns/aresta)**		**Leit. aleat. (ns/aresta)**	
		Média	**Des. Pad.**	**Média**	**Des. Pad.**
Tamanho do bloco	2 KB	132	5	1.655	467
	4 KB	133	4	1.997	869
	8 KB	133	3	2.290	1.497
Layout	Normal	132	4	2.036	1.346
	Escalado	134	5	1.929	599
Altura do bloco (Layout normal)	1	132	1	805	23
	2	130	1	1.042	45
	3	132	1	1.946	121
	4	131	1	1.554	113
	5	131	4	2.408	457
	6	129	4	4.440	1.606
Scale (Layout escalado)	1	144	2	2.909	645
	2	136	1	1.671	101
	3	134	2	2.095	122
	4	130	1	1.972	92
	5	131	1	1.789	21
	6	130	1	1.138	40

O tempo médio geral da consulta completa é de 133 ns/aresta e o desvio padrão é 4 ns/aresta. Ambas as medidas foram calculadas a partir de todos os 24 grafos web e todas as 36 combinações de parâmetros. Lembre-se que a consulta de leitura completa executa apenas uma leitura seqüencial no disco e nenhum *disk seek*. Observamos os seguintes fatos:

1) O custo de processamento dos blocos superiores são amortizados entre os inferiores e não adicionam um valor significativo para o tempo médio final;

2) Não há diferença significativa entre os resultados para diferentes tamanhos de blocos e de layout;

3) Para o layout normal, o tempo médio é praticamente o mesmo para diferentes altura de blocos; era esperado que o tempo médio fosse correlacionado com o custo médio, mas essa homogeneidade pode ser explicada por detalhes de implementação (como custo de recursão) e pelo alto tempo de processamento dos ponteiros;

4) Para o layout escalado, o tempo médio diminui ligeiramente à medida que o parâmetro *scale* aumenta. Isto ocorre porque, à medida que o balanceamento entre o número de ponteiros e tamanho das descrições melhora, o tempo de processamento e o custo de armazenamento diminui.

Para os grafos web transpostos, o tempo médio global é 107 ns/aresta e o desvio padrão é 4 ns/aresta. Eles apresentam comportamentos semelhantes aos seus grafos não-transpostos.

O tempo médio da consulta de leitura aleatória é 1.981 ns/aresta e o desvio padrão é 1.067 ns/aresta. Observamos os seguintes fatos:

1) Mesmo com os blocos dos níveis superiores armazenados em cache e pré-processados, o tempo médio é uma ordem de magnitude maior que o da consulta de leitura completa. Isto ocorre porque a consulta de leitura aleatória tem que acessar um número de blocos igual à altura da árvore-w e ela realiza uma leitura de disco e um *disk seek* por bloco;

2) O tempo médio da consulta aumenta à medida que o tamanho do bloco aumenta. Isto ocorre porque as alturas das árvores permanecem as mesmas, enquanto que as descrições codificadas por bloco são maiores (limitadas à L_{block}). Portanto, consome-se mais tempo para decodificá-las e processá-las. Isto é especialmente verdade para o layout escalado cujas descrições codificadas tendem a atingir o valor de L_{block} na maioria dos blocos por causa do melhor ajuste do número de nós por bloco de cada nível;

3) O tempo médio da árvore-w com layout escalado é marginalmente menor que o com layout normal, aproximadamente 5%;

4) O tempo médio aumenta à medida que a altura do bloco aumenta;

5) O tempo médio diminui à medida que *scale* aumenta.

Para os grafos web transpostos, o tempo médio global é 1.896 ns/aresta e o desvio padrão é 1.100 ns/aresta. Eles apresentam o mesmo comportamento de seus grafos web não-transpostos.

A altura da árvore-w com layout escalado é assintoticamente menor que com layout normal. No entanto, perceba que a árvore-w com layout normal e altura 1 tem o menor tempo de consulta de leitura aleatória. Além dessa observação, os itens 4 e 5 acima podem ser explicados pelos seguintes fatos que otimizam a execução da consulta na árvore-w:

1) Vários discos lêem à frente mais páginas de disco do que foi solicitado. Assim, se os blocos a serem lidos estiverem próximos uns dos outros (quando serializados na memória) então nenhuma operação extra de disco será necessária;

2) A soma dos tamanhos das descrições codificadas de um bloco nos níveis inferiores é normalmente menor que o tamanho máximo do bloco (Figura 4.8) e, assim, uma operação de leitura no disco pode retornar vários blocos de uma vez. Quando o número de nós em um bloco é grande, a soma dos tamanhos das descrições codificadas é tipicamente o tamanho máximo do bloco. Isso significa que a posição de início de um bloco na memória pode ser distante da posição de início de seus descendentes e, portanto, necessita de mais operações de *disk seek*;

3) Finalmente, observamos uma forte correlação do tempo de processamento com o custo de armazenamento nos resultados da consulta de leitura aleatória. Em adição aos dois fatos acima, quanto menor a quantidade de acessos a disco e a quantidade de bits para decodificar, menor é o tempo de processamento.

Comparamos os tempos das consultas básicas da árvore-w com *webgraph framework*. Os resultados são apresentados na Tabela 4.7. Observamos os seguintes fatos:

1) A consulta de leitura completa é muito rápida (cerca de 6 vezes) em relação à árvore-w e a representação parece também ter sido otimizada para essa consulta;

2) Para ambas as consultas, o parâmetro tamanho da janela não afeta significativamente o tempo médio. No entanto, o parâmetro contagem de referência tem um enorme impacto na consulta de leitura aleatória. Por exemplo, quando fixado em ∞, o tempo médio cresce cerca de 904 vezes em relação ao tempo médio de quando é fixado em 256. Isto ocorre porque o tempo

de execução é diretamente proporcional a esse parâmetro e que, quando fixado em ∞, o pior caso de cada consulta de leitura aleatória é linearmente proporcional ao número de nós do grafo web.

Embora o *webgraph framework* seja uma representação de memória principal, existem conjuntos de parâmetros que o tornam *inviável* na prática. Em particular, não é viável usar um conjunto de parâmetros que atinja a compressão máxima permitida.

Observe que o *webgraph framework* apresenta um *tradeoff* entre compressão e tempo de acesso aleatório. Uma aplicação deve levar isso em conta ajustando o parâmetro de contagem de referência apropriadamente. Para ilustrar isso, perceba que, com um custo de 3,40 bits/aresta, o *webgraph framework* tem o tempo de acesso aleatório de 145 ns/aresta, enquanto que, com um custo de 2,64 bits/aresta, ele tem o tempo de 3.291 ns/aresta. Ao ajustarmos o parâmetro de janela, *webgraph framework* apresenta o *tradeoff* entre custo e tempo de compressão. Resumindo, podemos usar o *webgraph framework* com parâmetro janela fixado em 7 e contagem de referência fixado em 256 para obter valores balanceados entre as operações: custo de 2,64 bits/aresta, tempo de compressão 237 ns/aresta, de leitura completa 21 ns/aresta e aleatória 3.291 ns/aresta.

A árvore-w não apresenta esses *tradeoffs*, ou seja, os parâmetros que apresentam as melhores taxas de compressão também apresentam os (quase) melhores tempos de construção, de leitura completa e aleatória. Nesse caso, a árvore-w com layout escalado e *scale* fixado em 6 apresenta custo de 2,27 bits/aresta, tempo de compressão 536 ns/aresta, de leitura completa 130 ns/aresta e aleatória 1.138 ns/aresta. Com esses dados, *a árvore-w pode ser considerada competitiva em relação ao webgraph framework.*

Tabela 4.7: Resultados das consultas básicas do *webgraph framework* com tamanho da janela fixado em 7.

Parâmetro	**Valor**	**Leit. Comp. (ns/aresta)**		**Leit. Aleat. (ns/aresta)**	
		Média	**Des. Pad.**	**Média**	**Des. Pad.**
Contagem de referência	3	22	3	145	42
	256	21	3	3,291	725
	∞	22	5	2,975,220	3,727,223

A árvore-w permite o uso de estratégias simples de cache que nos permitiu carregar e pré-processar os blocos superiores que são comuns às consultas de leitura aleatória. Observe que nem todas as representações na literatura permitem usar estratégias simples de cache. No entanto, *o autor reconhece a importância do cache para viabilizar as estruturas de dados externas na prática.*

E que a utilização de uma estratégia de cache mais sofisticada pode levar a melhorias de desempenho.

Finalmente, os resultados do *uk-union* são resumidos a seguir. O tempo médio de leitura completa é 112 ns/aresta (tempo total é 10 minutos), na qual o menor tempo é 99 ns/aresta (9 minutos) para o layout normal com altura 6 e bloco 16 KB e o maior deles é 123 ns/aresta (11 minutos) para o layout escalado com $scale = 1$ e bloco 16 KB. O tempo médio de leitura aleatória é 2.064 ns/edge, na qual o menor tempo é 683 ns/aresta para o layout normal com altura 1 e bloco 32 kb e o maior deles é 6.320 ns/aresta para o layout normal com altura 6 e bloco 64 KB. O tempo da consulta de leitura completa do *webgraph framework* com o tamanho da janela fixado em 7 é 22, 19 e 16 ns/aresta (2, 1,77 e 1,45 minutos, respectivamente) para a contagem de referência fixada em 3, 256 e ∞, respectivamente. O tempo médio para a leitura aleatória é 112 e 2.998 ns/aresta para a contagem de referência fixada em 3 e 256, respectivamente. Quando a fixamos em ∞, fomos obrigados a terminar o experimento antes do final porque já estava durando muito tempo.

4.4.3 Experimentos de consultas orientadas à conjuntos

Um experimento de consulta orientada à conjuntos consiste em encontrar todos os conjuntos s_k da coleção $\mathcal{S}$ de um grafo web que satisfaçam as consultas com os seguintes operadores: subconjunto, superconjunto, igualdade e intersecção. O conjunto da consulta F é criado com no máximo 5 intervalos da forma $[a, b]$ tal que $a, b \geq 0$, $b - a + 1 \leq 2$ e o valor de a é escolhido aleatoriamente. Cada experimento é repetido 20 vezes para diferentes conjuntos de filtro F e o resultado é a média dessas 20 repetições. Para a consulta de intersecção, estamos interessados nos conjuntos s_k que tem pelo menos $k_0 = 2$ e no máximo $k_1 = 5$ elementos em comum com o conjunto filtro F.

Realizamos experimentos de consulta orientada à conjuntos nos 24 grafos web e em seus transpostos em todas as 36 combinações de parâmetros e para os operadores de subconjunto, superconjunto, igualdade e intersecção. Devemos lembrar que todas as estruturas de dados foram armazenadas no disco e o cache não foi usado. Os resultados das consultas orientada à conjuntos, medidos em porcentagem de blocos recuperados pelo total de blocos, são apresentados na Tabela 4.8. Os resultados são resumidos separadamente para cada operador: subconjunto, superconjunto, intersecção e igualdade.

A média geral de blocos recuperados é, respectivamente, 0,07%, 0,04%, 0,06% e 0,04% e o desvio padrão é, respectivamente, 0,13%, 0,10%, 0,12% e 0,10%. Em cada experimento, o número total de blocos recuperados não

Tabela 4.8: Resultados das consultas orientada à conjuntos na árvore-w, medidos em blocos recuperados.

Parâmetro	**Valor**	**Subconjunto**		**Superconjunto**		**Intersecção**		**Igualdade**	
		Média	**Des. Pad.**	**Média**	**Des. Pad.**	**Média**	**Des. Pad.**	**Média**	**Des. Pad.**
Tamanho do bloco	2 kb	0,132	0,199	0,095	0,163	0,119	0,190	0,095	0,164
	4 kb	0,051	0,078	0,030	0,054	0,043	0,072	0,029	0,053
	8 kb	0,026	0,038	0,009	0,017	0,019	0,031	0,009	0,016
Layout	Normal	0,062	0,111	0,042	0,090	0,053	0,103	0,042	0,090
	Escalado	0,078	0,152	0,047	0,120	0,067	0,145	0,047	0,120
Altura do bloco (Layout normal)	1	0,006	0,003	0,004	0,001	0,002	0,001	0,001	0,001
	2	0,008	0,002	0,001	0,001	0,004	0,001	0,001	0,001
	3	0,019	0,010	0,007	0,005	0,013	0,006	0,007	0,005
	4	0,033	0,017	0,019	0,013	0,026	0,016	0,019	0,013
	5	0,088	0,055	0,054	0,042	0,075	0,050	0,053	0,040
	6	0,216	0,193	0,169	0,159	0,198	0,179	0,170	0,160
scale (Layout escalado)	1	0,359	0,206	0,255	0,188	0,331	0,207	0,252	0,191
	2	0,042	0,026	0,015	0,014	0,031	0,022	0,014	0,013
	3	0,024	0,013	0,006	0,005	0,017	0,010	0,006	0,005
	4	0,016	0,006	0,004	0,003	0,010	0,005	0,004	0,003
	5	0,014	0,006	0,003	0,002	0,008	0,005	0,003	0,002
	6	0,011	0,004	0,002	0,002	0,007	0,004	0,002	0,002

ultrapassa de 1,71% do número total de blocos em cada árvore-w. Os resultados apresentam um comportamento consistente para todo os operadores:

1) À medida que o tamanho do bloco aumenta, o número e a porcentagem de blocos recuperados diminui;

2) O layout escalado recupera menos blocos do que o layout normal, aproximadamente 50%, mas é maior em termos de blocos recuperados pelo total de blocos;

3) À medida que a altura do bloco (*scale*) aumenta, o número e a porcentagem de blocos recuperados aumenta (diminui).

Para os grafos web transpostos, a média geral de blocos recuperados é 0,09%, 0,06%, 0,08% e 0,06% para as consultas de subconjunto, superconjunto, intersecção e igualdade, respectivamente, e o desvio padrão é 0,13%, 0,12%, 0,13% e 0,11%, respectivamente. Embora superior, apresentam o mesmo comportamento de seus grafos web não-transpostos.

Este comportamento tem uma explicação simples: na medida em que os nós dos blocos superiores têm mais espaço para "descrever"os conjuntos associados às arestas incidentes aos nós, mais cedo o processador de consulta pode determinar se há conjuntos s_k da coleção $\mathcal{S}$ que satisfará o predicado de consulta. Quanto mais cedo ele consegue determinar que não haja nenhum conjunto então nenhum nó descendente de uma subárvore mais alta precisa ser recuperado (Teorema 4.8). A árvore com layout escalado foi construída para permitir mais espaço para as descrições dos nós dos blocos superiores e vemos que ele funciona como esperado. Por causa disso, observamos uma forte correlação: *na medida em que o custo de armazenamento aumenta, o número de blocos recuperados também aumenta.*

Lembre-se de que os grafos web tem a propriedade de localidade quando sua lista de adjacência é representada na ordem lexicográfica de suas URLs. O processador de consulta toma vantagem dessa propriedade sem qualquer esforço adicional ao avaliar uma consulta avançada. Porque é mais provável que a maioria dos conjuntos que o processador de consulta precisa avaliar sejam próximos um dos outros o que diminui o número de blocos a serem recuperados. No entanto, o pior caso de uma consulta orientada à conjuntos exige a recuperação de todos os nós da árvore-w. Normalmente, este não é o caso para um grafo web representado pela árvore-w, como podemos verificar na Tabela 4.8. A maioria das representações na literatura requer uma varredura completa sobre os dados. Este fato limita os tipos de consultas que podem ser eficientemente avaliadas. Além disso, existem cenários (ex. grafo muito

grande) que são simplesmente *inviáveis* para executar essas consultas sobre essas representações.

Embora o *webgraph framework* não otimize as consultas orientadas à conjuntos, é ainda útil apresentar os resultados da árvore-w para o grafo web *uk-union*. O número médio de blocos recuperados é 0,05%, 0,04%, 0,05% e 0,04% para os operadores subconjunto, superconjunto, intersecção e igualdade, respectivamente. Os menores resultados são 0,0002%, 0,0001%, 0,0002% e 0,0001%, respectivamente, e os maiores são 0,58%, 0,56%, 0,57% e 0,56%, respectivamente, para os mesmos operadores. Esses resultados mostram que a escolha dos parâmetros da árvore-w afeta significativamente o desempenho das consultas orientada à conjuntos:

1) Para a árvore-w com layout escalado, $scale = 1$ e bloco 2 KB, as consultas são executados em 18.948, 16.855, 17.501 e 16.864 ms, em média, para os operadores subconjunto, superconjunto, intersecção e igualdade, respectivamente;

2) Para a árvore-w com layout escalado, $scale = 6$ e bloco 2 KB, o tempo médio é de 1.348, 42, 404 e 42 ms, respectivamente;

3) Para a árvore-w com layout escalado com $scale = 6$, ela apresenta o menor número médio, entre todos os operadores e tamanhos de bloco, de 126 blocos recuperados e um tempo médio de 412 ms;

4) Para a árvore-w com layout normal e altura 1, ela atinge o menor tempo médio, entre todos os operadores e tamanhos de bloco, de 225 ms e recupera 293 blocos.

Finalmente, observamos que existe uma correlação com o tempo de processamento e o número de blocos recuperados. Leva-se aproximadamente 0,48 ms para recuperar um bloco, em média.

4.4.4 Experimentos de consultas de arestas recíprocas

Um experimento de consulta de arestas recíprocas consiste em encontrar todos os conjuntos s_k da coleção $\mathcal{S}$ que tem elementos comuns com seu conjunto correspondente s_k^T da coleção $\mathcal{S}^T$.

Realizamos experimentos nos 24 grafos web sobre todas as 36 combinações de parâmetros. Todas as estruturas de dados foram armazenadas em disco e nenhum dado foi armazenado em cache. Os resultados, medidos em porcentagem do número de blocos recuperados pelo número total de blocos na árvore-w, são apresentados na Tabela 4.9.

Tabela 4.9: Os resultados das consultas de arestas recíprocas, medidas em porcentagem do número de blocos recuperados pelo número total de blocos.

Parâmetro	**Valor**	**Média**	**Des. Pad.**
Tamanho do bloco	2 KB	72,3	4,2
	4 KB	71,9	4,0
	8 kB	71,6	3,4
Layout	Normal	70,7	1,5
	Escalado	73,2	5,0
Altura do bloco (Layout normal)	1	69,3	0,0
	2	69,1	0,0
	3	72,4	0,1
	4	71,9	0,2
	5	69,3	0,2
	6	72,3	1,0
scale (Layout escalado)	1	83,8	1,4
	2	69,3	0,2
	3	68,6	0,1
	4	72,4	0,1
	5	72,6	0,3
	6	72,4	0,1

A porcentagem média global é de 71,9% e o desvio padrão é de 3,8%. Isso significa que cerca de 28% dos blocos não precisaram ser recuperados. Isso ainda é melhor do que outras representações que têm de ler todos os dados. Observamos que os resultados são homogêneos e não variam bastante de acordo com os parâmetros.

Os resultados do grafo web *uk-union* mostram que o número médio de blocos recuperados é 79,4% do número médio de blocos, na qual a menor porcentagem é 73,2% para o layout escalado com $scale = 3$ e bloco 32 KB e a maior delas é de 87,1% para o layout escalado com $scale = 1$ e bloco 16 KB.

4.4.5 Experimentos de escalabilidade

Um experimento de escalabilidade consiste em executar os experimentos de compressão, de consultas básicas, orientadas à conjuntos e de arestas recíprocas para o mesmo grafo web, mas com diferente número de nós.

Realizamos experimentos sobre o grafo web *uk-union* e seu transposto com 20%, 40%, 60%, 80% e 100% de seus 133,633,040 nós para todas as 36 combinações de parâmetros. Os resultados de escalabilidade para as árvores-w $\mathcal{W}$ e $\mathcal{W}^T$ são apresentados na Tabela 4.10 e 4.11, respectivamente.

Observamos os seguintes fatos:

1) O custo de armazenamento é pior para a árvore-w com 20%. Ele cai bastante para a árvore-w com 40% nós e mantém-se estável até 80%, mas continua a cair para a árvore-w com 100%. Esses números mostram que o custo de

Tabela 4.10: Resultados de escalabilidade da árvore-w $\mathcal{W}$.

Medida	**Unidade**	**20%**	**40%**	**60%**	**80%**	**100%**
Custo de armazenamento	bits/aresta	2,87	2,24	2,25	2,25	2,19
Tempo de compressão	ns/aresta	649	539	555	557	562
Cons. de leitura completa	ns/aresta	156	120	121	119	112
Cons. de leitura aleatória	ns/aresta	1.116	1.523	2.086	2.139	2.064
Cons. de subconjunto	% nós recuperados	0,14	0,06	0,06	0,06	0,05
Cons. de superconjunto	% nós recuperados	0,13	0,05	0,06	0,05	0,04
Cons. de intersecção	% nós recuperados	0,14	0,06	0,06	0,05	0,05
Cons. de igualdade	% nós recuperados	0,13	0,05	0,06	0,05	0,04
Cons. de aresta recíproca	% nós recuperados	81	78	78	78	79

Tabela 4.11: Resultados de escalabilidade da árvore-w $\mathcal{W}^T$.

Medida	**Unidade**	**20%**	**40%**	**60%**	**80%**	**100%**
Custo de armazenamento	bits/aresta	2,55	1,94	1,94	1,92	1,84
Tempo de compressão	ns/aresta	765	591	604	598	639
Cons. de leitura completa	ns/aresta	116	86	85	82	80
Cons. de leitura aleatória	ns/aresta	880	1.204	1.682	1.748	1.675
Cons. de subconjunto	% nós recuperados	0,07	0,07	0,08	0,10	0,11
Cons. de superconjunto	% nós recuperados	0,06	0,07	0,08	0,09	0,10
Cons. de intersecção	% nós recuperados	0,06	0,07	0,08	0,10	0,11
Cons. de igualdade	% nós recuperados	0,06	0,07	0,08	0,09	0,10

armazenamento não cresce à medida que ele escala para cima. Uma possível explicação para isso é que, com a adição de novos nós do grafo web, o custo dos blocos superiores são amortizados entre os inferiores e as folhas;

2) O mesmo comportamento pode ser observado para o tempo de compressão. Isto ocorre porque existe uma correlação entre o tempo de compressão e o custo de armazenamento;

3) O tempo médio da consulta de leitura completa apresenta uma diminuição consistente, porém, pequena na medida em que a árvore-w escala para cima. Isto ocorre pelo mesmo motivo acima, o tempo de processamento dos blocos superiores é amortizado entre os inferiores e folhas;

4) Observe que o grafo web dobra o número de nós da árvore-w com 20%, 40% e 80%. Para essas árvores-w, vemos um comportamento consistente no tempo médio que cresce em 512 ns/aresta, em média, cada vez que a árvore-w dobra de número de folhas. É um comportamento esperado já que a altura da árvore-w cresce com a adição de nós do grafo web;

5) Os resultados das consultas orientadas à conjuntos mostram que o número de blocos recuperados cresce lentamente na medida em que a árvore-w escala para cima, mas ele diminui lentamente em porcentagem de blocos recuperados pelo número total de blocos da árvore-w. Também é um

comportamento esperado já que a altura da árvore-w cresce com a adição de nós do grafo web;

6) Para a árvore-w transposta $\mathcal{W}^T$, observamos um comportamento semelhante da árvore-w $\mathcal{W}$ no número total de blocos, mas cresce em termos de porcentagem de blocos recuperados na medida em que a árvore-w escala para cima;

7) Lembre-se de que a consulta de arestas recíprocas usa ambas as árvores-w $\mathcal{W}$ e $\mathcal{W}^T$ para avaliá-la. Para esse experimento, os resultados mostram números semelhantes à medida que a árvore-w cresce.

Estamos interessados em analisar mais detalhadamente o custo de armazenamento na medida em que a árvore-w escala para cima. Realizamos experimentos sobre o grafo web *uk-union* e seu transposto com 20%, 40%, 60%, 80% e 100% de seus 133,633,040 nós para todas as 36 combinações de parâmetros. Os resultados de escalabilidade para as árvores-w $\mathcal{W}$ são apresentados na Tabela 4.12.

Tabela 4.12: Resultados de escalabilidade do custo de compressão da árvore-w.

Parâmetro	**Valor**	**Custo médio (bits/aresta)**				
		20%	**40%**	**60%**	**80%**	**100%**
Tamanho do bloco	2 KB	3,24	2,50	2,53	2,53	2,47
	4 KB	2,78	2,18	2,20	2,20	2,13
	8 KB	2,58	2,03	2,04	2,03	1,97
Layout	Normal	3,01	2,27	2,24	2,21	2,13
	Escalado	2,73	2,20	2,27	2,30	2,25
Altura do bloco (Layout Normal)	1	2,56	1,98	1,98	1,96	1,90
	2	2,54	1,96	1,97	1,97	1,91
	3	2,61	2,04	2,04	2,02	1,96
	4	2,78	2,19	2,18	2,17	2,11
	5	3,21	2,58	2,52	2,46	2,36
	6	4,33	2,90	2,75	2,65	2,55
Scale (Layout Escalado)	1	3,24	2,76	3,12	3,35	3,38
	2	2,82	2,33	2,33	2,31	2,22
	3	2,61	2,08	2,13	2,17	2,11
	4	2,59	2,03	2,03	2,02	1,95
	5	2,56	2,01	2,01	2,00	1,93
	6	2,53	1,98	1,98	1,97	1,91

Observamos os seguintes fatos:

1) Para o parâmetro tamanho do bloco, observamos o mesmo comportamento nas árvores-w com 20%, 40% e 80%: o maior custo ocorre na árvore-w com 20%; o custo diminui bastante para 40% e permanece estável até 80%; finalmente, o custo cai para a árvore-w com 100%;

2) Para todas as alturas de bloco $(1, 2, \ldots, 6)$ do layout normal, a árvore-w consistentemente diminui à medida que ela escala para cima;

3) O layout escalado tem o mesmo comportamento do layout normal; exceto para $scale = 1$, na qual se degrada na medida em que a árvore-w escala para cima.

Novamente, esses fatos ocorrem porque o custo dos blocos superiores são amortizados com o custo dos nós inferiores. O custo de armazenamento da árvore-w com layout escalado e $scale = 1$, no entanto, cresce porque não o valor do parâmetro não representa um balanceamento apropriado entre o número de ponteiros e o tamanho máximo das descrições.

Finalmente, os resultados experimentais apresentados aqui não são suficientes para analisar a escalabilidade da árvore-w. Mais experimentos em outros grafos web maiores devem ser realizados para consolidar o comportamento apresentado aqui.

4.4.6 Discussão

De acordo com os experimentos, a melhor combinação de parâmetros é o layout normal(escalado) com o parâmetro altura do bloco (*scale*) fixado em 1(6) e o tamanho do bloco fixado em 64 KB. Resultando nos seguintes números: tempo de compressão 621(536) ns/aresta, custo de armazenamento 2,33(2,27) bits/aresta, tempo de leitura completa 132(130) ns/aresta, tempo de leitura aleatória 805(1.138) ns/aresta, consulta de subconjunto 787(1.130) blocos recuperados, consulta de superconjunto 163(369) blocos recuperados, consulta de intersecção 554(892) blocos recuperados, consulta de igualdade 158(356) blocos recuperados e a consulta de arestas recíprocas evita de recuperar 30,7% (27,6%) dos blocos.

Espera-se que, na medida em que a árvore-w escale para cima, a melhor combinação de parâmetros seja a árvore-w com layout escalado. Além disso, quanto maior o tamanho do bloco melhor os resultados de todos os experimentos. A árvore-w transposta parece ter melhores resultados do que a árvore-w não-transposto. Em vista de todos os experimentos realizados aqui, *o autor acredita que são suficientes para demonstrar a viabilidade da árvore-w em disco.*

Para dar uma melhor noção dos resultados experimentais resumidos no primeiro parágrafo desta seção em um grafo web maior, fornecemos a seguinte situação. Google informou recentemente que encontrou cerca de 2^{40} páginas web únicas e queremos representar esse grafo web usando a árvore-w. Deste modo, se usarmos a árvore-w com layout normal(escalado) com a melhor configuração e supondo que esses valores escalem para um grafo web com 2^{40} nós e 2^5 arestas por nó então precisaríamos de 252(218) dias para construí-la

resultando em uma árvore-w com altura de 40(19) blocos, 9.544(9.298) GB de espaço para armazená-lo (observe que não inclui o custo do grafo transposto) e 53(53) dias para realizar uma consulta de leitura completa. *Lembre-se que o modelo computacional usado tem apenas um processador e um disco.*

Em vista desses números, vale a pena discutir a possibilidade de paralelizar as operações da árvore-w em p processadores e p discos. Para isso, simplesmente particionamos a árvore-w em p subárvores e associamos cada uma delas a um par de processador e disco podendo facilmente obter um fator de melhoria de $O(p)$. Além disso, em vista dos resultados do perfil de execução da implementação (não apresentados aqui) e da configuração do computador usado nos experimentos, qualquer otimização no hardware ou na implementação da árvore-w melhoraria os resultados apresentados. Portanto, o autor acredita (conjectura) que não há necessidade de usar um modelo computacional distribuído para um grafo web de até 1 trilhão de nós uma vez que apenas um computador com vários processadores e discos seria suficiente para processá-lo. Apesar de que seria também fácil de distribuí-lo.

Observe que *nenhuma representação na literatura seria adequada para um grafo web deste tamanho uma vez que não caberia na memória principal.* É verdade que definindo os parâmetros janela e contagem de referência do *webgraph framework* com valores fixos e não dependentes do tamanho de nós ou arestas do grafo web, o número de acessos ao disco para cada consulta de leitura completa poderia ser linear e o de acesso aleatório poderia ser constante. No entanto, a implementação atual dessa representação não foi feita para ser executada em disco e não é apropriada nem justo utilizá-la nos experimentos em disco. Em vista dos experimentos realizados, *a árvore-w pode ser considerada competitiva em relação ao webgraph framework.*

Ao considerarmos o objetivo de desenvolver uma estrutura de dados para enriquecer as aplicações de grafos web permitindo-as que executem novos tipos de consultas de forma eficiente, a árvore-w apresenta uma vantagem em relação ao *webgraph framework*. Este último teria que percorrer toda a representação para retornar a resposta.

Uma outra vantagem da representação proposta é a possibilidade de suportar as operações de adição e remoção de nós e arestas do grafo web de forma eficiente. No *webgraph framework*, a adição de um nó ou aresta poderia acarretar na reconstrução completa da representação.

Finalmente, reconhece-se que outros estudos são necessários com diferentes configurações, implementações e dados para que a nova representação venha a ser realmente utilizada na prática.