Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Árvores binárias de busca e a Conjectura da Otimalidade Dinâmica

Bruno Armond Braga

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisora: Prof.ª Dr.ª Cristina Gomes Fernandes

Durante o desenvolvimento deste trabalho, o autor recebeu auxílio financeiro da FAPESP – processo nº 2024/04708-2.

São Paulo
2024

Agradecimentos

Você vale tudo.
— Ednaldo Pereira

Primeiramente gostaria de agradecer aos meus amigos por estarem ao meu lado durante toda essa jornada. Um agradecimento especial à Tereza Cristina Lacerda que me acolheu de braços abertos quando ingressei no IME e à Fernanda Duarte dos Reis, minha melhor amiga que esteve presente em todos os momentos. Agradeço também Guilherme Vinicius Ferreira de Assis que virou inúmeras noites fazendo trabalhos comigo, ofereceu segundas opiniões indispensáveis em relação a esse trabalho e, acima de tudo, foi a pessoa mais importante para minha graduação, alguém que sempre pude contar nos momentos de sufoco.

Gostaria de agradecer à minha orientadora Cristina Gomes Fernandes que me apresentou ao tema desse trabalho, que sempre se dedicou de maneira excepcional para garantir que eu estava aprendendo e que principalmente potencializou o meu interesse por Teoria da Computação e, em especial, estruturas de dados.

Além da Cris, gostaria de agradecer aos professores Carlos Eduardo Ferreira, Jose Coelho de Pina Junior e Yoshiko Wakabayashi, que também foram grandes fontes de inspiração pra mim e me proporcionaram ensinamentos essenciais para a produção desse trabalho.

Por fim, gostaria de agradecer à comunidade imeana e às entidades das quais tive a honra de fazer parte. A sensação de pertencimento a esses espaços fez toda a diferença na minha graduação.

Resumo

Bruno Armond Braga. **Árvores binárias de busca e a Conjectura da Otimalidade Dinâmica**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

Árvores binárias de busca são estruturas elementares na ciência da computação e sua eficiente capacidade de responder perguntas sobre um conjunto faz com quem elas sejam muito utilizadas e estudadas. Apesar desse vasto estudo, há questões primordiais em aberto sobre o custo ótimo de algoritmos de busca em ABBs para sequências de acesso arbitrárias. Nesse trabalho cobriremos a conjectura da otimalidade dinâmica que diz que árvores splays são assintoticamente tão eficientes quanto qualquer outra árvore para qualquer sequência de acessos. Além disso, cobriremos uma visão geométrica de buscas que é muito útil para caracterizar o custo ótimo e veremos como conseguir delimitar inferiormente custos de sequências de acesso a partir das delimitações da alternância e do funil. Por fim, veremos uma redução que mostra que o problema de múltiplas buscas em árvores binárias de busca é NP-completo.

Palavras-chave: estrutura de dados. conjectura otimalidade dinâmica. árvores binárias de busca. árvores splay.

Abstract

Bruno Armond Braga. **Binary search trees and the Dynamic Optimality Conjecture**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

Binary search trees (BSTs) are fundamental structures in computer science, known for their efficient ability to answer queries about a set, making them widely used and studied. Despite extensive research, central questions still open about the optimal cost of search algorithms in BSTs for arbitrary access sequences. In this work, we will address the dynamic optimality conjecture, which states that splay trees are asymptotically as efficient as any other tree for any access sequence. Furthermore, we will explore a geometric approach to searches that proves useful for characterizing the optimal cost and examine how to derive lower bounds for the costs of access sequences based on alternation and funnel bounds. Finally, we will present a reduction demonstrating that the problem of multiple searches in binary search trees is NP-complete.

Keywords: data structures. dynamic optimality conjecture. binary search trees. splay trees.

Sumário

1 Introdução **1**

2 ABB estática ótima **5**
2.1 Custo de uma ABB 5
2.2 Natureza do problema 5
2.3 Algoritmo guloso 6
2.4 Algoritmo de Knuth 7

3 Árvores splay **11**
3.1 Introdução 11
3.2 Operação splay 12
3.2.1 Passos zig-zig e zag-zag 12
3.2.2 Passos zig-zag e zag-zig 12
3.2.3 Passos zig e zag 13
3.3 Análise da operação splay 15
3.4 Conjectura da Otimalidade Dinâmica 18

4 Interpretação geométrica de buscas em ABBs **21**
4.1 Conjuntos arboreamente satisfeitos 21
4.2 Visão geométrica de buscas 22

5 Algoritmo guloso **27**
5.1 Otimalidade 27
5.2 Guloso futurista 28

6 Conjuntos independentes de retângulos **33**
6.1 Orientação de retângulos 33
6.2 Guloso futurista orientado 34
6.3 Independência de retângulos 36

6.4 Delimitação dos retângulos independentes 39

7 Delimitações inferiores de Wilber **41**
7.1 Visão geral 41
7.2 Delimitação da alternância 42
7.3 Conjunto independente de retângulos a partir da alternância 45
7.4 Delimitação do funil 46
7.5 Conjunto independente de retângulos a partir do funil 49
7.6 Sequência bit-reversa 50

8 Relação entre as delimitações de Wilber **53**
8.1 Rotação e inversão de conjuntos de pontos 53
8.2 Relacionando a alternância com o funil e sua inversão temporal 54
8.3 Relacionando o funil e sua inversão temporal 58

9 Buscas múltiplas **63**

10 Conclusão **69**

Bibliografia **71**

Capítulo 1

Introdução

Árvores Binárias de Busca (ABBs) são estruturas de dados que armazenam um conjunto de chaves de um universo estático, que possui uma ordem total, e dão suporte a buscas neste conjunto. Denotaremos por n o número de elementos do conjunto armazenado na ABB considerada.

Estas estruturas são fundamentais na ciência da computação e possuem as mais diversas finalidades. Nesse projeto iremos estudar a chamada Conjectura da Otimalidade Dinâmica, interpretações geométricas de algoritmos de busca em ABBs e uma caracterização do custo ótimo. Também serão estudadas as delimitações de Wilber e como elas se relacionam.

Adaptando a definição dada por Sedgewick e Wayne [10], *árvore binária de busca* é uma árvore binária onde cada nó possui uma chave comparável e possivelmente um valor associado. Além disso, os nós satisfazem a restrição de que a chave em qualquer nó é maior do que as chaves em todos os nós na subárvore esquerda desse nó e menor do que as chaves em todos os nós na subárvore direita desse nó.

As ABBs também possuem um atributo *raiz* que aponta para o nó raiz ou possui valor *nulo*, caso a árvore esteja vazia.

Rotações são operações que trocam dois nós, pai-filho, entre si enquanto mantém a restrição vista acima. Veja a Figura 1.1. Essa operação é fundamental para garantir performance em alguns algoritmos de ABBs e é muito utilizada para controlar a altura das árvores.

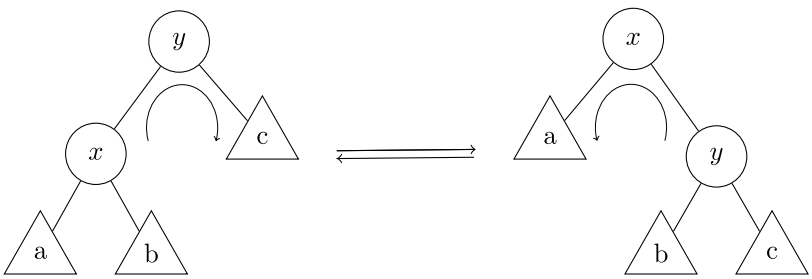


Figura 1.1: *Esquema de rotações.*

No contexto da Conjectura da Otimalidade Dinâmica, apenas serão consideradas buscas bem sucedidas, que chamaremos de *acessos*, e não são consideradas inserções ou remoções no conjunto armazenado. No modelo de computação adotado, um algoritmo de busca em uma ABB, para fazer o acesso a uma chave, mantém um único ponteiro. Chamamos de *nó corrente* o nó apontado por esse ponteiro.

No início da execução do acesso, o ponteiro aponta para a raiz da árvore e, após uma sequência de operações, deve encontrar o nó que armazena a chave procurada. As operações chamadas de *primitivas* são:

1. Mover o ponteiro para o filho esquerdo do nó corrente.
2. Mover o ponteiro para o filho direito do nó corrente.
3. Mover o ponteiro para o pai do nó corrente.
4. Fazer uma rotação que troca a posição do nó corrente e do seu pai.

O algoritmo tradicional de busca em ABB não executa a operação primitiva 4. Ele inicia a execução de um acesso na raiz e desce para o filho apropriado até alcançar a chave procurada. No modelo de computação adotado, todas essas operações possuem custo unitário.

Em geral, durante a execução de um acesso, uma série de nós distintos são apontados pelo ponteiro do algoritmo de busca. Ao fim da execução de uma busca bem sucedida, o nó corrente é o nó procurado. Assim, dizemos que o nó procurado foi *acessado* e que todos os nós que foram apontados pelo ponteiro durante a busca foram *visitados*. O nó acessado também é considerado um nó visitado.

O pior caso acontece quando a chave procurada está em um nó folha mais profundo. Nesse caso o algoritmo tem que passar por toda a altura da árvore até chegar a essa folha. Assim o custo do pior caso é proporcional à altura da árvore, que pode ser em princípio linear em n.

Com intuito de mitigar o custo do pior caso, algumas ABBs estrategicamente utilizam a operação primitiva 4 (rotação) para balancear o tamanho das subárvores e assim diminuir a altura da árvore. Um exemplo famoso, sobre o qual daremos mais detalhes à frente, é a árvore splay. Essa ABB executa rotações baseando-se na heurística “move to front” e assim balanceia a árvore durante as buscas. Essa árvore não utiliza armazenamento adicional para isso, ou seja, seus nós não têm campos extra.

O custo para executar um acesso nesse modelo está relacionado com o número de operações primitivas executadas até encontrar a chave procurada durante um acesso.

No artigo de Demaine et al. [12], é demonstrada a seguinte propriedade sobre ABBs. É possível transformar qualquer ABB com t nós em qualquer outra ABB com os mesmos t nós com no máximo $2t - 6$ rotações. O número preciso não é tão relevante para a análise desse texto. O importante é notar que o número de rotações necessárias para transformar uma ABB T com t nós em uma ABB T' com os mesmos t nós disposta de qualquer outra maneira é $O(t)$, ou seja, é linear no número de nós da ABB.

Por isso, podemos assumir que o número de rotações realizadas durante um acesso

é linear no número de nós visitados, assim definiremos o custo para executar um acesso nesse modelo como o número de nós visitados durante esse acesso.

Como o conjunto armazenado nas ABBs que consideraremos não sofre alterações, podemos assumir que esse conjunto é {1,2,...,n}.

ABBs *online* são ABBs que apenas possuem informações sobre os acessos passados. Ou seja, se $X = (x_1, \dots, x_m)$ é a sequência de acessos que serão feitos, onde cada $x_i \in \{1, 2, \dots, n\}$, ao acessar x_i, um algoritmo de busca online não tem conhecimento das chaves $x_{i+1}, \dots, x_m$, nem mesmo do valor de m. Em particular, muitas ABBs online possuem informações adicionais em cada nó que auxiliam o algoritmo de busca a decidir quando executar rotações durante a busca pelas chaves procuradas. Qualquer ABB online que usa O(1) palavras adicionais por nó possui tempo de execução dominado pelo número de operações primitivas.

Uma ABB é chamada de *offline* se tem conhecimento da sequência X de acessos antes de começar os acessos às chaves de X. Nesse contexto, o comportamento do algoritmo não depende unicamente do histórico de acessos passados, uma vez que o algoritmo possui conhecimento prévio de toda a sequência de chaves a serem acessadas.

Dada uma sequência X de acessos, uma ABB é considerada *ótima* se executa os acessos de X com o menor custo possível. Podemos definir $\mathrm{OPT}(X)$ como o número de nós visitados por uma ABB ótima para a sequência X. Em outras palavras, $\mathrm{OPT}(X)$ é o número mínimo necessário de visitas a nós para uma ABB concluir todos os acessos de X.

ABBs balanceadas estabelecem que $\mathrm{OPT}(X) = \mathrm{O}(m \log n)$, onde n é o número de chaves armazenadas na árvore e m é o comprimento de X. Como todo acesso tem custo maior ou igual a 1, então $\mathrm{OPT}(X) \geq m$. Wilber [13] provou que $\mathrm{OPT}(X) = \Theta(m \log n)$ para algumas classes de sequências X.

Uma ABB online é *dinamicamente ótima* se, para todas as sequências X, seu algoritmo de busca tem custo $\mathrm{O}(\mathrm{OPT}(X))$. De maneira mais geral, uma ABB online é *c-competitiva* se executa todos os acessos de sequências X suficientemente longas com custo no máximo $c\,\mathrm{OPT}(X)$.

Todo esse estudo foi feito para tentar responder à pergunta: "*Existe uma ABB online dinamicamente ótima?*".

Uma das tentativas de responder tal pergunta foi a árvore splay de Sleator e Tarjan [11]. Como já mencionamos, árvores splay são ABBs que seguem a heurística "move to front". Mais precisamente, após cada acesso, a árvore se reestrutura de uma maneira particular, trazendo o nó da chave que foi acessada para a raiz da árvore.

Apesar de existirem muitas ABBs muito bem documentadas com tempo por busca logarítmico em n, essas estruturas normalmente não conseguem alcançar uma eficiência superior a isso independentemente da entrada. Padrões de acesso do mundo real muitas vezes possuem estruturas repetitivas, como por exemplo bancos de dados que recebem solicitações frequentes para um pequeno número de elementos de alto tráfego. Em alguns desses casos, é possível ter uma performance melhor que $\Theta(\log n)$ por acesso utilizando uma árvore splay, uma vez que essa eventualmente ficaria com as chaves mais acessadas mais próximas à raiz da árvore.

Há uma conjectura não resolvida proposta por Sleator e Tarjan [11] que diz que as árvores splay são dinamicamente ótimas. Essa conjectura ficou conhecida como a Conjectura da Otimalidade Dinâmica.

Nesse texto, abordaremos uma série de aspectos diferentes em relação a delimitações de custo em algoritmos de busca em ABBs e implementações visando o menor custo possível. No Capítulo 2, será discutido o problema de encontrar uma ABB estática de custo mínimo para uma determinada sequência de acessos. No Capítulo 3, abordaremos o funcionamento e o custo de acessos das árvores splay e apresentaremos a Conjectura da Otimalidade Dinâmica. No Capítulo 4 e 5, examinaremos como tratar buscas em ABBs de maneira geométrica por meio de conjuntos arboreamente satisfeitos e será proposto um algoritmo online guloso para encontrar superconjuntos arboreamente satisfeitos a partir de conjuntos de pontos arbitrários. No Capítulo 6, ampliaremos a análise geométrica para desenvolver uma nova delimitação relacionada a retângulos independentes. O Capítulo 7 tratará das delimitações de Wilber e no Capítulo 8 explicaremos como elas se relacionam. Por fim, no Capítulo 9, apresentaremos uma redução de um problema NP-completo ao problema de buscas múltiplas em ABBs.

Capítulo 2

ABB estática ótima

Neste capítulo será apresentada a solução para o problema de encontrar uma árvore binária de busca estática com custo mínimo para uma determinada sequência de acessos, ou seja, uma ABB offline estática ótima. O algoritmo foi desenvolvido por [5].

2.1 Custo de uma ABB

O custo de acesso a uma chave em uma ABB estática é o número de nós visitados durante o algoritmo de busca. A *profundidade* de uma chave j em uma ABB T é definida pela distância do nó com chave j à raiz de T e é denotada por $d_T(j)$. Como rotações não são permitidas em ABBs estáticas, a estrutura da árvore não muda, logo o custo de um acesso à chave j numa tal ABB T sempre é 1 mais a profundidade de j em T, ou seja, é $1 + d_T(j)$.

O custo de executar uma sequência $X = (x_1, \ldots, x_m)$ de m acessos às chaves $1, 2, \ldots, n$ de uma ABB estática T é a somatória dos custos dos acessos executados, ou seja, é

$$c(T) = \sum_{i=1}^{m} (1 + d_T(x_i)).$$

Uma delimitação óbvia para este custo é que ele é no mínimo m, pois $d_T(j) \geq 0$.

Dada uma sequência X de acessos, uma ABB estática T é considerada ótima para X se $c(T)$ é mínimo, ou seja, se T executa os acessos de X com o menor custo possível.

2.2 Natureza do problema

Definimos o custo para uma sequência de acessos em uma ABB estática com base em cada elemento de X. Porém, nota-se que é possível definir o mesmo custo com base no número de ocorrências de cada elemento desta ABB em X. Denotaremos por $e(j)$ o número de ocorrências de j na sequência X. Cada acesso ao nó de chave j contribui com $d_T(j) + 1$ ao custo. Como o nó com chave j será acessado $e(j)$ vezes, então o nó de chave j

contribui com $(d_T(j) + 1) \cdot e(j)$ para o custo total, ou seja,

$$c(T) = \sum_{i=1}^{m}(1 + d_T(x_i)) = \sum_{j=1}^{n}(1 + d_T(j)) \cdot e(j),$$

onde a dependência do custo agora é no vetor com o número de ocorrências $e[1\,..\,n]$ de cada chave em X.

Essa abordagem também encapsula uma formulação alternativa do problema na qual $e[1\,..\,n]$ representa um vetor de probabilidades de acesso às chaves $[1\,..\,n]$ na sequência de entrada X. Nesse caso a ABB ótima é aquela que possui custo esperado mínimo. Essa é, inclusive, a abordagem utilizada no texto do [5]. Nesse artigo também são consideradas probabilidades de buscas mal sucedidas, o que não consideramos no nosso texto. Apesar de limitarmos a análise a buscas bem sucedidas, é possível utilizar a mesma abordagem para resolver problemas que consideram também buscas mal sucedidas.

Dado que a contribuição de cada chave para o custo é uma multiplicação entre a sua profundidade na ABB e seu número de ocorrências na sequência X, intuitivamente é esperado que os nós mais próximos da raiz de uma ABB ótima guardem as chaves com os maiores números de ocorrência na sequência, já que a ABB terá que acessar esses nós mais vezes. Analogamente, é esperado que os nós mais distantes da raiz de uma ABB ótima guardem as chaves com os menores números de ocorrência na sequência.

De maneira sucinta, é esperado que o custo de acessos mais caros, ou seja, acessos a nós mais profundos, sejam pagos menos vezes e é esperado que os custos de nós mais baratos, ou seja, acessos a nós mais superficiais, sejam pagos mais vezes.

2.3 Algoritmo guloso

É possível desenvolver um algoritmo guloso para a construção de uma ABB estática com base na intuição de priorizar que chaves com maior número de ocorrência estejam mais próximas da raiz. O Programa 2.1 organiza as chaves de maneira decrescente no número de ocorrências e adiciona iterativamente à árvore a chave de maior número de ocorrências que ainda não foi adicionada. Para isso, ele utiliza uma fila de prioridades de máximo que armazena pares $(j, e[j])$, considerando o segundo valor do par como prioridade.

Programa 2.1 Algoritmo guloso ABB.

Entrada: Número n de chaves e vetor $e[1\,..\,n]$ de ocorrências por chave.
Saída: ABB com as chaves 1 a n com respeito ao vetor e.

```
1  FUNÇÃO guloso_ABB(n, e)
2    Q ← fila_de_prioridade(e, n)        ▷ Cria uma fila de prioridade para o vetor e[1..n]
3    abb ← abb()                                                  ▷ Cria uma ABB vazia
4    enquanto Q ≠ ∅
5        (a,b) ← Q.remove_max()
6        abb.insere(a)                   ▷ Adiciona chave com maior número de ocorrências em Q
7  devolva abb
```

Apesar da abordagem acima ser bastante promissora, priorizar unicamente o número de ocorrências não garante adquirir uma ABB ótima. Isso acontece porque a posição de cada nó na árvore influencia a profundidade de todos os nós descendentes dele, principalmente levando em conta a propriedade de árvores binárias que exige que todos os nós da subárvore esquerda sejam menores que o nó e todos os nós da subárvore direita sejam maiores. Assim, em algumas situações, é mais vantajoso ter como pai um nó com menor número de ocorrências e ter nós com maior número de ocorrências mais profundos. Isto se torna ainda mais evidente quando pensamos em entradas com número de ocorrências por nó muito parecidos. Nestes casos, a estrutura de uma árvore binária mais balanceada tende a ser melhor. Veja o exemplo da Figura 2.1.

	Número de ocorrências
a	5
b	25
c	20
d	15

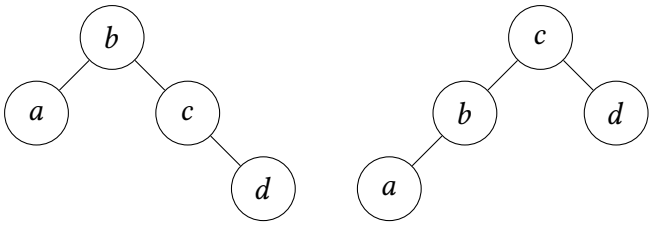


Figura 2.1: *Tabela com o número de ocorrências por chave em uma sequência de entrada. À esquerda, a árvore gerada pelo algoritmo guloso, com custo total 120, e à direita, a árvore ótima, com custo total 115.*

2.4 Algoritmo de Knuth

A análise da seção anterior nos leva a perceber que precisamos de uma conduta que considere tanto o número de ocorrências de cada nó quanto a estrutura da árvore em si. Um algoritmo para encontrar uma ABB estática ótima foi desenvolvido por [5].

Primeiramente é preciso entender como a estrutura de uma árvore ótima está disposta.

Lembremos que a profundidade de uma chave i em uma ABB T, é a distância, $d_T(i)$, entre o nó com chave i e a raiz de T. Assim sabemos que se a chave i está na subárvore T' de T, então a distância do nó com chave i à raiz de T é a distância do nó com chave i à raiz de T' adicionada à distância da raiz de T' à raiz de T. Suponha que a chave j seja raiz de T'. Então, $d_T(i) = d_{T'}(i) + d_T(j)$.

Seja T uma ABB estática ótima com n chaves para uma sequência de acessos X. Seja k a chave da raiz de T e T_e e T_d as subárvores esquerdas e direitas da raiz. Como k é a chave da raiz, T_e possui as chaves $\{1, \dots, k-1\}$ e T_d possui as chaves $\{k+1, \dots, n\}$.

Como T_e é a subárvore esquerda da raiz de T, a profundidade em T da raiz de T_e é 1. Logo, para qualquer chave $i \in T_e$, $d_T(i) = d_{T_e}(i) + 1$. O caso de T_d é análogo. Ademais, $d_T(k) = 0$, pois k é a chave da raiz.

Assim,

$$\begin{aligned}
c(T) &= \sum_{j=1}^{n}(d_T(j)+1)\cdot e(j) \\
&= \sum_{j=1}^{k-1}(d_T(j)+1)\cdot e(j) + \sum_{j=k+1}^{n}(d_T(j)+1)\cdot e(j) + (d_T(k)+1)\cdot e(k) \\
&= \sum_{j=1}^{k-1} d_T(j)\cdot e(j) + \sum_{j=k+1}^{n} d_T(j)\cdot e(j) + \sum_{j=1}^{n} e(j) \\
&= \sum_{j=1}^{k-1}(d_{T_e}(j)+1)\cdot e(j) + \sum_{j=k+1}^{n}(d_{T_d}(j)+1)\cdot e(j) + \sum_{j=1}^{n} e(j) \\
&= c(T_e) + c(T_d) + \sum_{j=1}^{n} e(j),
\end{aligned}$$

onde $c(T_e)$ considera apenas os acessos de X em $\{1,\ldots,k-1\}$, ou seja, considera $e[1\,..\,k-1]$, e $c(T_d)$ considera apenas os acessos de X em $\{k+1,\ldots,n\}$, ou seja, considera $e[k+1\,..\,n]$.

De acordo com a fórmula acima, é evidente que o custo total de uma ABB T depende do custo de suas duas subárvores T_e e T_d. Logo para uma árvore T ter custo mínimo então tanto a sua subárvore esquerda quanto a sua subárvore direita devem ter custo mínimo, ou seja, T_e e T_d devem ser ABBs ótimas para as buscas nas chaves que armazenam.

Expressaremos por $c[i,j]$ o custo mínimo de uma ABB que guarda as chaves de i a j em relação ao vetor de ocorrências $e[i\,..\,j]$ e por $s[i,j]$ a soma do número de ocorrências de todas as chaves entre i e j em X, ou seja, $s[i,j] = \sum_{h=i}^{j} e(h)$.

Chegamos na recorrência abaixo que resolve o problema:

$$c[i,j] = \begin{cases} 0 & \text{se } i > j. \\ \min_{i\le k\le j}\{c[i,k-1] + c[k+1,j]\} + s[i,j] & \text{se } i \le j. \end{cases}$$

O Programa 2.3 produz uma ABB estática com custo mínimo dado o vetor de ocorrências por chave.

Programa 2.2 Construção recursiva da ABB ótima.

Entrada: Dois inteiros i e j e a matriz de inteiros $r[1\,..\,n,1\,..\,n]$ construída pelo Programa 2.3.
Saída: ABB estática ótima com as chaves $[i\,..\,j]$.

```
1  FUNÇÃO construção_recursiva(i, j, r)
2    se i > j
3        então devolva NULL
4    k ← r[i, j]
5    nó ← novoNó(k)
6    nó.esq ← construção_recursiva(i, k − 1, r)
7    nó.dir ← construção_recursiva(k + 1, j, r)
8  devolva nó
```

Programa 2.3 Algoritmo de Knuth.

Entrada: Número n de chaves e vetor $e[1\,..\,n]$ de ocorrências por chave.
Saída: ABB estática ótima com chaves 1 a n e seu custo com respeito ao vetor e.

```
1   FUNÇÃO Knuth(n,e)
2     s[0] ← 0
3     para i de 1 até n faça
4         s[i] ← s[i − 1] + e[i]
5     para i de 1 até n + 1 faça
6         c[i, i − 1] ← 0
7     para ℓ de 1 até n faça
8         para i de 1 até n − ℓ + 1 faça
9             j ← i + ℓ − 1
10            c[i, j] ← c[i + 1, j]
11            r[i, j] ← i
12            para k de i + 1 até j faça
13                se c[i, k − 1] + c[k + 1, j] < c[i, j]
14                    então c[i, j] ← c[i, k − 1] + c[k + 1, j]
15                          r[i, j] ← k
16            c[i, j] ← c[i, j] + s[j] − s[i − 1]
17    abb ← construção_recursiva(1, n, r)
18  devolva abb, c[1,n]
```

Como o Programa 2.2 é utilizado pelo Programa 2.3, então é necessário analisar a construção antes do algoritmo de Knuth.

O Programa 2.2 recursivamente constrói a ABB. Isso é feito por meio de uma estratégia recursiva de baixo para cima. Assim, a função será executada uma vez para cada nó da ABB e mais uma vez para cada intervalo vazio. Como há exatamente n nós no intervalo de $[1\,..\,n]$, e há $n + 1$ intervalos vazios (que correspondem a chamadas da função com $i < j$) produzidos recursivamente, então o custo total do algoritmo é $O(n)$.

O Programa 2.3, por sua vez, adota uma estratégia de programação dinâmica. Esse tipo de estratégia se utiliza de espaço adicional de memória. Neste caso uma matriz, para armazenar computações já executadas. Dessa maneira, evita-se cálculos repetidos e maximiza-se a eficiência do programa.

Para preencher os elementos necessários da matriz para alcançar o custo final, o código itera por metade da matriz. Então serão necessárias $O(n^2)$ iterações. A iteração que preenche a posição $[i, j]$ das matrizes c e r representa o cálculo do custo mínimo para uma ABB que contenha todos os elementos do intervalo $[i\,..\,j]$. Esse cálculo verifica todas as possíveis raízes, o que é feito nas linhas 12-15 do Programa 2.3, então tem custo $O(n)$. Como há $O(n^2)$ iterações de custo $O(n)$, o custo total do algoritmo é $O(n^3)$.

Capítulo 3

Árvores splay

Neste capítulo apresentaremos as árvores splay, desenvolvidas por Sleator e Tarjan [11]. As árvores splay são ABBs que, além das rotinas usuais de busca, inserção e remoção, possuem uma rotina extra, chamada splay. Essa rotina deve ser acionada ao final de cada operação feita nesta árvore, aplicada ao nó mais profundo visitado durante a operação. Descreveremos o funcionamento da operação splay, analisaremos o seu custo amortizado e introduziremos a Conjectura da Otimalidade Dinâmica.

3.1 Introdução

A altura de uma árvore binária de busca é o comprimento de um caminho mais longo da raiz da árvore a uma de suas folhas. É sabido que a altura de uma ABB com n nós é um valor entre $\lg n$ e $n - 1$. O pior caso das operações de busca, inserção, remoção e outras em árvores binárias de busca é exatamente a altura da árvore. Por isso, surgiram na literatura várias propostas de implementações de ABBs que mantêm alguma propriedade que implica que a altura da árvore se mantém logarítmica no número de nós da ABB. Tais implementações, em geral, carregam informação extra nos nós da árvore e executam rotações durante as inserções e remoções porém não alteram a árvore durante as buscas.

Uma alternativa a essas estratégias é a árvore splay. Árvore Splay é uma árvore binária de busca balanceada proposta por Sleator e Tarjan [11]. Diferentemente das árvores binárias balanceadas citadas anteriormente, a árvore splay não utiliza armazenamento adicional e se reestrutura após toda operação, inclusive após as buscas.

A árvore splay segue a heurística “move to front”, assim a cada operação a árvore aciona a rotina splay para garantir que o nó mais profundo visitado seja movido para a raiz. Dessa maneira, buscas a chaves que foram recentemente buscadas têm seu custo reduzido porque seus nós estão mais próximos da raiz. Além disso, usualmente a rotina splay diminui a altura total da árvore e assim reduz o custo de o pior caso de operações futuras.

3.2 Operação splay

A essência da árvore splay está na operação splay. A operação splay é a responsável por mover um nó específico para a raiz por meio de sucessivos passos splay. Essa operação é fundamental para o funcionamento da estrutura e é utilizada por todas as outras operações.

A operação splay se utiliza de seis tipos distintos de passos splay para trazer um nó para a raiz: zig, zig-zig e zig-zag, e suas versões refletidas zag, zag-zag e zag-zig.

Seja x o nó que a operação splay está deslocando para a raiz.

3.2.1 Passos zig-zig e zag-zag

Os passos splay zig-zig e zag-zag são realizados quando x e seu pai são ambos filhos esquerdos ou ambos filhos direitos. Nesses casos, é necessário rotacionar o pai de x primeiro e em seguida rotacionar x.

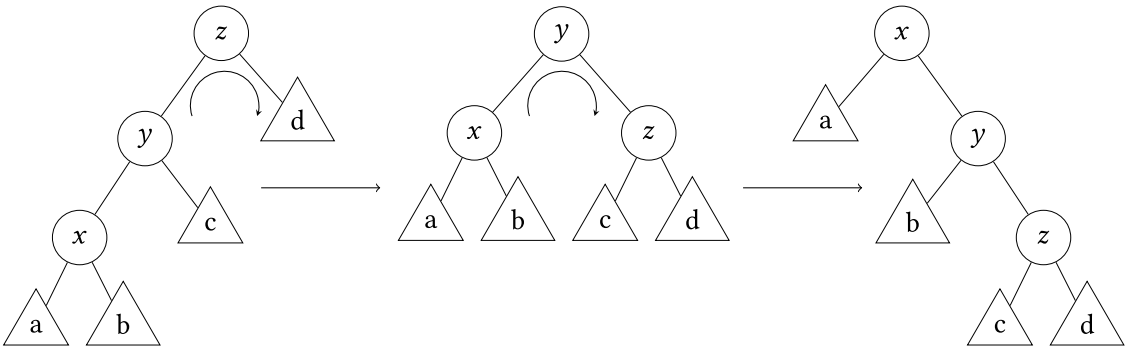


Figura 3.1: *Passo splay zig-zig onde x e y são ambos filhos esquerdos.*

3.2.2 Passos zig-zag e zag-zig

Os passos splay zig-zag e zag-zig são realizados quando x é filho esquerdo e o pai de x é filho direito ou x é filho direito e o pai de x é filho esquerdo. Nesses casos, é necessário rotacionar x duas vezes. Vale ressaltar que cada rotação será feita para um lado.

Note que esta rotação propositalmente diminui a altura da subárvore analisada.

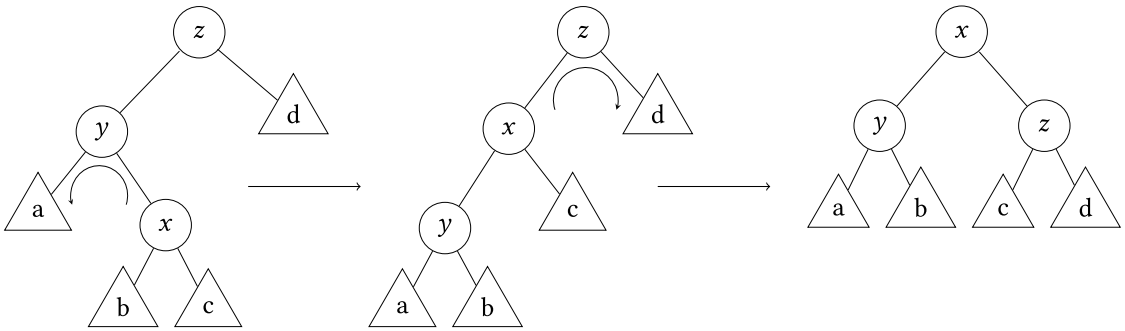


Figura 3.2: *Passo splay zag-zig onde x é filho direito e y é filho esquerdo.*

3.2.3 Passos zig e zag

Os passos splay zig e zag acontecem quando x é descendente direto do nó raiz. Nesses casos apenas é necessário realizar uma rotação no nó x e x se encontrará na raiz da árvore. Esses passos são os únicos de rotação única e são chamados de passos splay simples enquanto os demais são passos splay duplos.

Como a operação splay se encerra quando o nó analisado chega à raiz, estas rotações simples são executadas no máximo uma vez durante a execução de uma operação splay. Elas só acontece quando o caminho de x até a raiz da ABB tem comprimento ímpar.

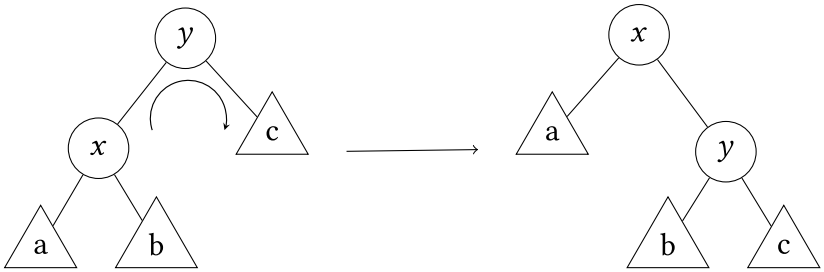


Figura 3.3: *Passo splay zig onde x é nó esquerdo da raiz.*

Em algumas estruturas são definidas rotações direitas e esquerdas. A operação primitiva de rotação do modelo de computação adotado rotaciona o nó corrente com seu pai. Veja o Programa 3.1, que consome O(1). Essencialmente rotações para a direita e esquerda são rotações de um nó com um de seus filhos que precisa ser definido enquanto que rotações de um nó com seu pai é uma operação bem definida.

Programa 3.1 Função de rotação.

Entrada: Recebe um vértice x de uma ABB que possui um vértice pai.
Saída: Executa uma rotação de x com seu pai.

```
1   FUNÇÃO rotação_com_pai(x)
2       y ← x.pai
3       z ← y.pai
4       se x = y.esq
5           então y.esq ← x.dir
6                 x.dir ← y
7                 se y.esq ≠ NULL
8                     então y.esq.pai ← y
9           senão y.dir ← x.esq
10                x.esq ← y
11                se y.dir ≠ NULL
12                    então y.dir.pai ← y
13      x.pai ← z
14      y.pai ← x
15      se z ≠ NULL
16          se z.esq = y
17              então z.esq ← x
18              senão z.dir ← x
```

A seguir veja o código da operação splay no Programa 3.2 que se utiliza das rotações dadas no Programa 3.1. Além disso, a Figura 3.4 ilustra a execução de uma operação splay onde há um passo splay zag-zag, um passo splay zig-zag e por fim um passo splay zig.

Programa 3.2 Função splay.

Entrada: Recebe um vértice x de uma ABB.
Saída: Reestrutura a ABB com passos splay até x ser raiz.

```
1   FUNÇÃO splay(x)
2       y ← x.pai
3       se y = NULL                                          ▷ x é raiz
4           retorne
5       se y.pai = NULL                                      ▷ zig/zag
6           então rotação_com_pai(x)
7       senão se y.esq = x e y.pai.esq = y ou y.dir = x e y.pai.dir = y   ▷ zig-zig/zag-zag
8           então rotação_com_pai(y)
9                 rotação_com_pai(x)
10      senão rotação_com_pai(x)                             ▷ zig-zag/zag-zig
11            rotação_com_pai(x)
12      splay(x)
```

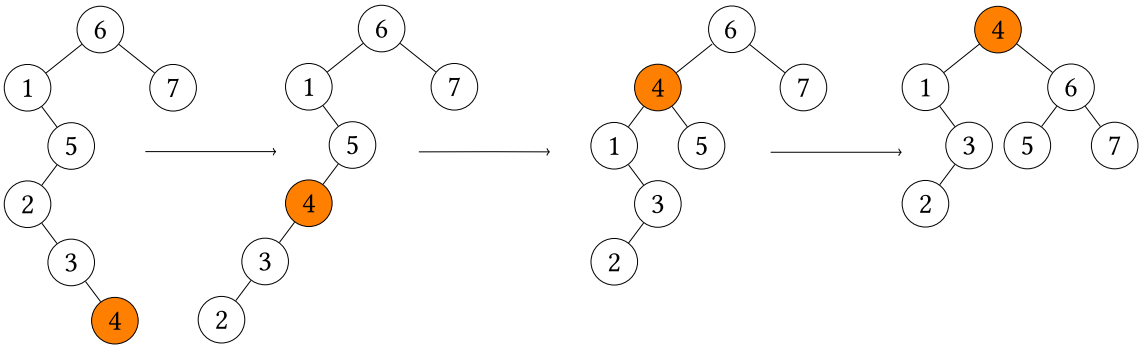


Figura 3.4: *Execução da operação splay após acesso à chave 4.*

Uma série de operações podem ser implementadas nessas estruturas como inserção, remoção, busca, junção e divisão. Todas essas operações chamam a operação splay no nó mais profundo visitado durante sua execução. É essencial entender que ao final de cada acesso, o nó com a chave acessada será levado para a raiz pela operação splay. O código da busca em árvores splay pode ser visto no Programa 3.3.

No escopo desse trabalho só são consideradas buscas bem-sucedidas, porém com poucas modificações nos códigos, é possível tratar buscas mal-sucedidas. A diferença é que o nó mais profundo visitado não será o nó com chave buscada, então o nó que a rotina splay levará para a raiz é uma das folhas que possui chave perto da chave buscada. O mesmo acontece para remoções mal-sucedidas que também não são consideradas nesse texto.

Programa 3.3 Função de busca em árvores splay.

Entrada: Recebe um inteiro z no intervalo $[1, n]$ e um nó x de uma árvore splay.
Saída: Acessa o nó com chave z e chama a rotina splay nele.

```
1  FUNÇÃO busca(z, x)
2      se x.val = z
3          então splay(x)
4          retorne
5      se x.val > z
6          então busca(z, x.esq)
7          senão busca(z, x.dir)
```

3.3 Análise da operação splay

Definimos o custo da operação splay como o número de rotações realizadas durante sua execução. Caso nenhuma rotação seja realizada, definimos o custo dessa operação como 1.

Faremos uma análise amortizada da estrutura pelo método do potencial. Seja S a árvore splay sendo analisada. Definimos $s^i(x)$ como o tamanho da subárvore enraizada no nó x de S depois do passo splay i de uma rotina splay. Definimos o potencial local do nó x logo depois do passo splay i como $r^i(x) = \lg(s^i(x))$. Por fim, definimos c como o custo real de uma operação splay, $\hat{c}$ como o custo amortizado de uma operação splay e $\Phi^i(S)$ como o potencial da árvore S depois do passo splay i, que é a somatória do potencial local de todos os nós de S nesse instante de tempo.

Para deduzir o custo amortizado de uma operação splay(x), vamos calcular o custo amortizado de cada passo splay realizado dentro da operação splay. Note que todos esses passos envolvem o nó x.

Lema 3.1. *Se o j-ésimo passo splay de uma rotina splay é um passo splay simples, então seu custo amortizado é menor que* $3(r^j(x) - r^{j-1}(x)) + 1$.

Demonstração. Denotemos por y o nó pai de x que é a raiz da árvore splay logo antes do último passo splay. Se o passo splay simples é um passo zig, então $c = 1$ e

$$\begin{aligned}
\hat{c} &= c + \Delta\Phi, \\
&= 1 + r^j(x) + r^j(y) - r^{j-1}(x) - r^{j-1}(y), \\
&< 1 + r^j(x) - r^{j-1}(x) && \text{pois } r^{j-1}(y) > r^j(y), \\
&< 1 + 3(r^j(x) - r^{j-1}(x)) && \text{pois } r^j(x) > r^{j-1}(x).
\end{aligned}$$

O mesmo vale simetricamente para o caso zag. □

Lema 3.2. *Se o j-ésimo passo splay de uma rotina splay é um passo splay duplo, então seu custo amortizado é menor que* $3(r^j(x) - r^{j-1}(x))$.

Demonstração. Para os próximos resultados será essencial entender uma propriedade da função logaritmo. A função logaritmo é côncava, ou seja, $\frac{\lg(a)+\lg(b)}{2} \leq \lg(\frac{a+b}{2})$. Isso pode ser evidenciado pela Figura 3.5.

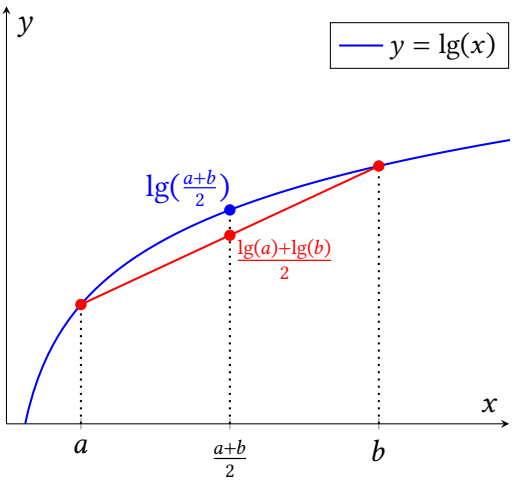


Figura 3.5: *Na imagem estão destacados pontos a e b arbitrários na função. Como a função é côncava, o ponto com* $y = \frac{\lg(a)+\lg(b)}{2}$ *nunca estará acima do ponto com* $y = \lg(\frac{a+b}{2})$.

Denotemos por y o nó pai de x antes do passo splay analisado e por z o avô de x no mesmo instante. Se o passo splay duplo é um passo zig-zig, então $c = 2$ e

$$\begin{aligned}
\hat{c} &= c + \Delta\Phi, \\
&= 2 + r^j(x) + r^j(y) + r^j(z) - r^{j-1}(x) - r^{j-1}(y) - r^{j-1}(z), \\
&= 2 + r^j(y) + r^j(z) - r^{j-1}(x) - r^{j-1}(y) && \text{pois } r^j(x) = r^{j-1}(z), \\
&< 2 + r^j(y) + r^j(z) - 2r^{j-1}(x) && \text{pois } r^{j-1}(y) > r^{j-1}(x), \\
&< 2 + r^j(x) + r^j(z) - 2r^{j-1}(x) && \text{pois } r^j(x) > r^j(y).
\end{aligned}$$

Por conta da concavidade da função lg vista acima, temos que:

$$\begin{aligned}
r^{j-1}(x) + r^j(z) &= \lg(s^{j-1}(x)) + \lg(s^j(z)), \\
&= 2 \cdot \frac{\lg(s^{j-1}(x)) + \lg(s^j(z))}{2}, \\
&\leq 2\lg(\frac{s^{j-1}(x) + s^j(z)}{2}) && \text{concavidade de lg}, \\
&< 2\lg(\frac{s^j(x)}{2}) && \text{pois } s^j(x) > s^{j-1}(x) + s^j(z), \\
&= 2(\lg(s^j(x)) - \lg(2)), \\
&= 2(r^j(x) - 1).
\end{aligned}$$

Assim, retornando ao problema original,

$$\begin{aligned}\hat{c} &< 2 + r^j(x) + r^j(z) - 2r^{j-1}(x),\\ &< 2 + r^j(x) + 2(r^j(x) - 1) - r^{j-1}(x) - 2r^{j-1}(x),\\ &= 3(r^j(x) - r^{j-1}(x)).\end{aligned}$$

O mesmo vale para o caso zag-zag por simetria.

Se o passo splay duplo é um passo zig-zag, então $c = 2$ e

$$\begin{aligned}\hat{c} &= c + \Delta\Phi,\\ &= 2 + r^j(x) + r^j(y) + r^j(z) - r^{j-1}(x) - r^{j-1}(y) - r^{j-1}(z),\\ &= 2 + r^j(y) + r^j(z) - r^{j-1}(x) - r^{j-1}(y) && \text{pois } r^j(x) = r^{j-1}(z),\\ &< 2 + r^j(y) + r^j(z) - 2r^{j-1}(x) && \text{pois } r^{j-1}(x) < r^{j-1}(y),\\ &< 2 + r^j(x) - 2r^{j-1}(x) && \text{pois } r^j(y) + r^j(z) < r^j(x),\\ &< 3(r^j(x) - r^{j-1}(x)).\end{aligned}$$

O mesmo vale para o caso zag-zig por simetria. □

Teorema 3.3. *O custo amortizado da operação splay é* $O(\lg n)$.

Demonstração. Caso a operação splay não faça rotações, o custo é 1 e a delimitação é imediata. Caso contrário, o custo amortizado da operação splay no nó x é a soma dos custos amortizados dos passos splay realizados durante a operação.

Denotemos por $\hat{c}_{splay}$ o custo amortizado da operação splay e por $\hat{c}_i$ o custo amortizado do i-ésimo passo splay realizado durante o Splay(x). Denotemos por t o número de passos splay realizados por essa operação. Então

$$\begin{aligned}\hat{c}_{splay} &= \sum_{i=1}^{t} \hat{c}_i,\\ &< \sum_{i=1}^{t} 3(r^i(x) - r^{i-1}(x)) + 1,\\ &= 3(r^t(x) - r^0(x)) + 1,\\ &\leq 3(r^t(x)) + 1 && \text{pois } r^0(x) \geq 0,\\ &= 3\lg n + 1 && \text{pois } r^t(x) = \lg(n),\\ &= O(\lg n).\end{aligned}$$

□

Teorema 3.4. *Se $m = \Omega(n)$, então uma sequência de m acessos em uma árvore splay possui custo* $O(m \lg n)$.

Demonstração. Note que $\Phi^0(S)$ é a soma do potencial local de todos os nós de S durante esse instante de tempo. Assim, sabemos que

$$\begin{aligned}
\Phi^0(S) &= \sum_{x=1}^{n} r^0(x) && \text{pela definição de } \Phi^0(S),\\
&= \sum_{x=1}^{n} \lg(s^0(x)) && \text{pela definição de } r^0(x),\\
&\leq \sum_{x=1}^{n} \lg n && \text{pois } s^0(x) \leq n,\\
&= n \lg n.
\end{aligned}$$

Essa propriedade será utilizada abaixo. Considere uma sequência de m acessos em uma árvore splay S. Então

$$\begin{aligned}
\sum_{i=1}^{m} c_i &= \sum_{i=1}^{m} (\hat{c}_i + \Phi^{i-1}(S) - \Phi^i(S)),\\
&= \sum_{i=1}^{m} \hat{c}_i + \Phi^0(S) - \Phi^m(S),\\
&\leq \sum_{i=1}^{m} \hat{c}_i + \Phi^0(S) && \text{pois } \Phi^m(S) \geq 0,\\
&\leq m(3 \lg n + 1) + n \lg n && \text{pois } \Phi^0(S) \leq n \lg n,\\
&= (3m + n)(\lg n) + m,\\
&= O(m \lg n) && \text{se } m = \Omega(n).
\end{aligned}$$

□

3.4 Conjectura da Otimalidade Dinâmica

Uma ABB online é *dinamicamente ótima* se, para todas as sequências X de acessos, seu algoritmo de busca tem custo $O(OPT(X))$. De maneira mais geral, uma ABB online é *c-competitiva* se executa todas as buscas para sequências X suficientemente longas com custo no máximo $c\,OPT(X)$.

Sleator e Tarjan [11] conjecturaram que a árvore splay é dinamicamente ótima. Essa conjectura é conhecida por Conjectura da Otimalidade Dinâmica. Os cientistas nas últimas décadas vêm buscando provar propriedades que $OPT(X)$ e as árvores splay possuem em comum. Uma série de características foram detectadas e o ramo de pesquisa se mantém ativo. É impressionante que em uma área tão amplamente pesquisada como a área de estrutura de dados, essa conjectura continue em aberto, intrigando cientistas pela sua dificuldade.

Pouco se sabe sobre o valor de OPT(X) para uma sequência X de acessos. No próximo capítulo abordaremos uma visão geométrica de buscas e em seguida buscaremos entender quais são as características de OPT(X) dentro dessa abordagem.

Capítulo 4

Interpretação geométrica de buscas em ABBs

Neste capítulo explicaremos a visão geométrica proposta por Demaine, Harmon, Iacono e Pătraşcu [3], o que são conjuntos de pontos arboreamente satisfeitos e como interpretar de maneira geométrica algoritmos de busca em ABBs.

4.1 Conjuntos arboreamente satisfeitos

Chamamos dois pontos a e b de *ortogonalmente colineares* se a e b estão na mesma linha horizontal ou vertical. Se a e b não são ortogonalmente colineares, denotamos por $\{a, b\}$-*retângulo* o retângulo ortogonal que tem a e b como vértices.

Um par de pontos $\{a, b\}$ de um conjunto de pontos P é *arboreamente satisfeito* se a e b são ortogonalmente colineares ou se há pelo menos um ponto do conjunto $P \setminus \{a, b\}$ que está dentro da região delimitada pelo $\{a, b\}$-retângulo, incluindo seu perímetro. Um conjunto P de pontos é *arboreamente satisfeito* se todos os pares de pontos do conjunto são arboreamente satisfeitos. Veja a Figura 4.1.

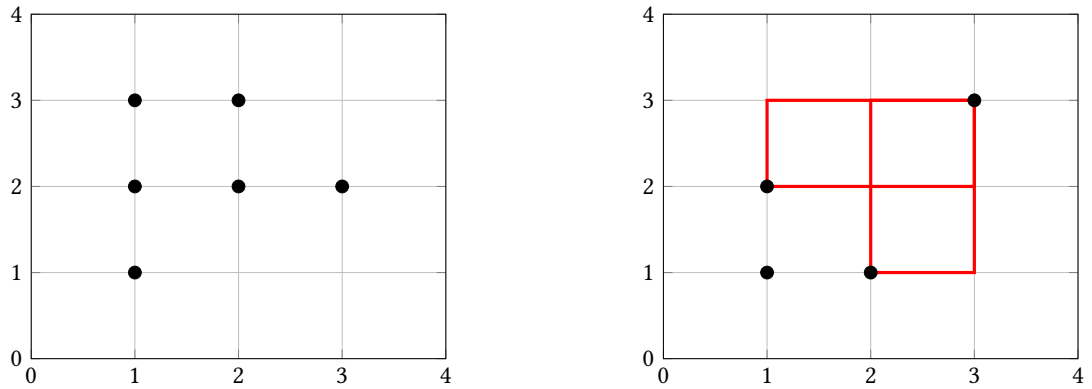


Figura 4.1: *À esquerda, um conjunto P de pontos arboreamente satisfeito. À direita, um conjunto P de pontos com dois pares de pontos arboreamente insatisfeitos com seus retângulos destacados.*

Lema 4.1. *Em um conjunto arboreamente satisfeito P, para todo par de pontos $\{a, b\}$ não ortogonalmente colineares, há sempre pelo menos um ponto de $P \setminus \{a, b\}$ que está em um dos lados do $\{a, b\}$-retângulo que incide em a, e há sempre também pelo menos um ponto de $P \setminus \{a, b\}$ que está em um dos lados do $\{a, b\}$-retângulo que incide em b.*

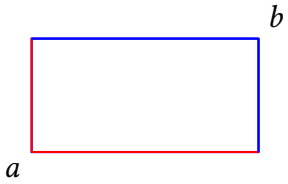


Figura 4.2: *O $\{a, b\}$-retângulo. Em vermelho estão destacados os lados do retângulo que incidem em a e em azul estão destacados os lados que incidem em b.*

Demonstração. Vamos provar, por indução em k, que todo $\{a, b\}$-retângulo com k pontos em seu interior satisfaz a propriedade do Lema.

Para $k = 0$, todo $\{a, b\}$-retângulo possui um ponto de $P \setminus \{a, b\}$ em um de seus vértices ou possui dois pontos de $P \setminus \{a, b\}$ em seu perímetro alinhados verticalmente ou horizontalmente. Caso o $\{a, b\}$-retângulo possua um ponto de $P \setminus \{a, b\}$ em um de seus vértices, então a propriedade do lema é válida. Caso o $\{a, b\}$-retângulo possua dois pontos de $P \setminus \{a, b\}$ em seu perímetro alinhados verticalmente ou horizontalmente, então um desses pontos está em um dos lados do $\{a, b\}$-retângulo que incide em a e o outro está em um dos lados do $\{a, b\}$-retângulo que incide em b, satisfazendo a propriedade do lema.

Suponha agora que $k > 0$ e que a propriedade vale para todo retângulo com menos que k pontos de P em seu interior. Seja e um ponto de P no interior do $\{a, b\}$-retângulo. O $\{a, e\}$-retângulo e o $\{e, b\}$-retângulo são retângulos de P com menos que k pontos em seu interior, logo a propriedade vale para esses retângulos. Como os lados do $\{a, b\}$-retângulo que incidem em a contêm os lados do $\{a, e\}$-retângulo que incidem a e os lados do $\{a, b\}$-retângulo que incidem em b contêm os lados do $\{e, b\}$-retângulo que incidem b, então a propriedade do lema vale também para o $\{a, b\}$-retângulo. □

4.2 Visão geométrica de buscas

No modelo de computação adotado, para realizar um acesso em uma ABB, o algoritmo de busca inicia o nó corrente na raiz da ABB. Em seguida, percorre a árvore descendo para o filho apropriado por meio de comparações até alcançar a chave procurada.

Dada uma sequência $X = (x_1, \ldots, x_m)$ de m acessos às chaves $1, 2, \ldots, n$, é possível ilustrar essa sequência X de maneira gráfica em um plano cartesiano da seguinte forma: o eixo x representará as chaves armazenadas na ABB e o eixo y representará os índices $i = 1, 2, \ldots, m$, que interpretamos como o tempo. Denotaremos por P_X esse conjunto de pontos que chamaremos de *visão geométrica da sequência de acessos*. Assim, um ponto de coordenada (x,i) representa a busca da chave x na ABB no instante de tempo i, ou seja, representa x_i. Veja o exemplo da Figura 4.3.

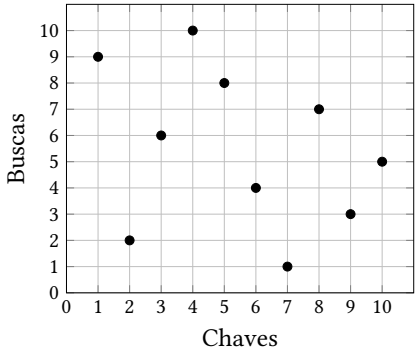


Figura 4.3: *Gráfico representando a sequência (7, 2, 9, 6, 10, 3, 8, 5, 1, 4) de acessos.*

A *visão geométrica da execução de um algoritmo de busca em ABB* para uma sequência $X = (x_1, \ldots, x_m)$ de buscas, de maneira similar, é o conjunto P de pontos na forma (x,i) tal que x é a chave de um dos nós visitados durante a busca pela chave x_i. Para cada $i \in \{1, \ldots, m\}$, os pontos de P pertencentes a reta $y = i$ representam as chaves dos nós visitados durante a busca por x_i. Veja a Figura 4.4.

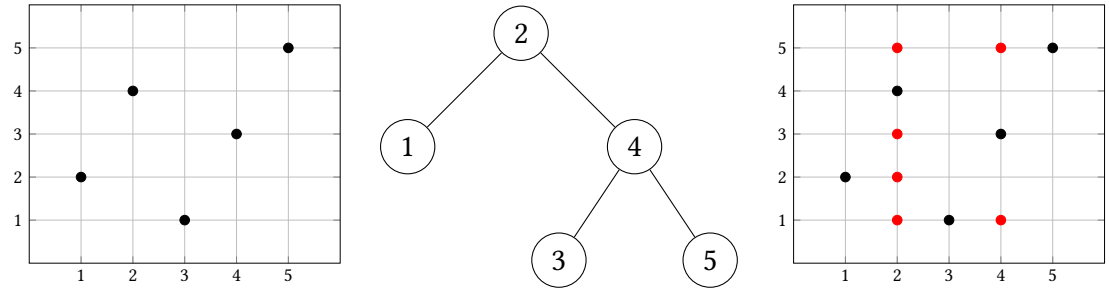


Figura 4.4: *À esquerda, o conjunto de pontos que representa a sequência $X = (3, 1, 4, 2, 5)$. No meio, uma possível ABB com as chaves {1,2,...,5}. À direita, a visão geométrica do algoritmo de busca que não efetua rotações na ABB do meio. Os pontos pretos são pontos de P_X relacionados à sequência de acessos e os pontos vermelhos são o restante dos nós visitados durante cada um desses acessos. Note que o conjunto de pontos à esquerda não é arboreamente satisfeito, mas o conjunto à direita é.*

Lema 4.2. *A visão geométrica de qualquer execução de um algoritmo de busca em ABB no modelo de computação adotado é um conjunto de pontos arboreamente satisfeito.*

Demonstração. Seja P a visão geométrica da execução de um algoritmo de busca em uma ABB T para alguma sequência X de buscas. Suponha, por contradição, que P não é um conjunto arboreamente satisfeito. Dessa maneira, há pelo menos um par $\{p, q\}$ de pontos de P que não é arboreamente satisfeito, onde $p = (a, i)$ e $q = (b, j)$. Assumiremos sem perda de generalidade que $i < j$ e $a < b$.

Dois lados do $\{p, q\}$-retângulo terão um papel a seguir na prova. São eles:

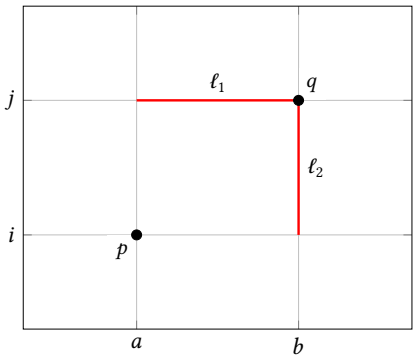
- $\ell_1 = \{(x, y) \mid a \leq x < b \text{ e } y = j\}$, e
- $\ell_2 = \{(x, y) \mid x = b \text{ e } i \leq y < j\}$,

onde, dentro do contexto da ABB T, ℓ_1 representa as visitas aos nós com chave no intervalo $[a, b)$ no instante j e ℓ_2 representa as visitas ao nó s entre os instantes de tempo i e $j - 1$.

Como $\{p, q\}$ é um par de pontos arboreamente insatisfeito, nota-se que não há nenhum ponto de $P \setminus \{p, q\}$ nas bordas ou no interior do $\{p, q\}$-retângulo. Logo, $(\ell_1 \cup \ell_2) \cap P = \emptyset$.

Sejam c_1 e c_3 respectivamente as chaves de dois nós n_1 e n_3 de uma ABB com $c_1 \leq c_3$. Se n_2 é o ancestral comum mais profundo dos nós n_1 e n_3 e sua chave é c_2, então sabemos que $c_1 \leq c_2 \leq c_3$ e, mais importante, n_2 é ancestral dos nós n_1 e n_3 nesta ABB, e assim os nós n_1 e n_3 têm profundidade maior ou igual a n_2. Logo, n_2 está no caminho da raiz desta ABB até qualquer nó com chave no intervalo $[c_1, c_3]$. Em outras palavras, para visitar qualquer nó da ABB com chave no intervalo $[c_1, c_3]$, é necessário visitar o ancestral comum mais profundo entre n_1 e n_3, que neste caso é n_2.

Seja r o nó de T com chave a e s o nó de T com chave b. Seja c a chave do ancestral comum mais profundo de r e s em T imediatamente antes da busca i. Seja d a chave do ancestral comum mais profundo de r e s em T imediatamente antes da busca j.
Como $\ell_1 \cap P = \emptyset$, então obrigatoriamente $d = b$. Por outro lado, como $\ell_2 \cap P = \emptyset$, temos que $c \neq b$ e, em algum instante h, com $i \leq h < j$, o nó s teve que ser visitado para ser rotacionado e se tornar o ancestral comum mais profundo entre r e s no instante imediatamente antes de j, logo $\ell_2 \cap P \neq \emptyset$, uma contradição.



□

A partir daqui, consideraremos apenas pontos (x, y) em que x e y são inteiros, $1 \leq x \leq n$ e $1 \leq y \leq m$.

Lema 4.3. *Qualquer conjunto de pontos arboreamente satisfeito representa a execução de um algoritmo de busca em ABB no modelo de computação adotado.*

Demonstração. Usaremos um tipo de árvore binária de busca denominada *treap*. Os nós de uma treap possuem dois campos, um chamado chave e outro chamado prioridade. A treap mantém duas propriedades: a propriedade de ordem de árvores binárias de busca em relação às chaves e a propriedade de heap em relação às prioridades. Assim, todo nó de uma treap possui chave maior que os nós da sua subárvore esquerda e chave menor que os nós da sua subárvore direita. Além disso, os nós da treap obedecem à propriedade de heap, que pode ser de dois tipos: se for um heap máximo, cada nó terá prioridade maior ou igual a de seus filhos; se for um heap mínimo, cada nó terá prioridade menor ou igual a de seus filhos.

Seja P um conjunto de pontos arboreamente satisfeito. Denotaremos por $N(h, i)$ a coordenada y do ponto mais baixo de P que possui coordenadas $x = h$ e $y > i$. Caso não exista tal ponto, definiremos $N(h, i) = \infty$. Assim,

$$N(h, i) = \min\{y : (h, y) \in P \text{ e } y > i\},$$

onde o mínimo sobre o conjunto vazio é ∞.

É possível criar uma treap com as chaves $1, \dots, n$ que é um heap mínimo nas prioridades a partir de P. Para cada $j \in \{1, \dots, n\}$, adicione um nó na forma $(j, N(j, 0))$, ou seja, adicione um nó com chave j e prioridade $N(j, 0)$. Essa é a nossa treap inicial.

Com essa treap criada, nota-se que já temos uma ABB que possui n nós. Se conseguirmos descrever uma maneira de, para cada instante de tempo $h \in \{1, \dots, m\}$, visitarmos todos os nós com chave z tais que $(z, h) \in P$ e reestruturarmos a treap de maneira a manter sua estrutura sem visitar nós extras, então descrevemos uma execução de um algoritmo offline de busca em uma ABB no modelo de computação adotado.

Primeiramente, note que os nós com prioridade mínima induzem uma subárvore da treap que contém a raiz. Seja i essa prioridade mínima. Os nós com prioridade i serão visitados no instante i e terão sua prioridade alterada apenas considerando essa subárvore. Queremos mostrar que, ao fazer tais alterações nas prioridades no instante i e reorganizar a treap dessa maneira, a treap toda continua satisfazendo a propriedade de heap.

Vamos supor, por contradição, que há um nó q cuja prioridade é menor que de seu pai p na treap após as modificações do instante i. Isso só pode ocorrer se p estiver dentro da subárvore dos nós visitados no instante i e q não estiver nessa subárvore. Seja a a chave de p e b a chave de q e assumiremos sem perda de generalidade que $a < b$. Assim, $N(a, i) > N(b, i)$. Veja a Figura 4.5.

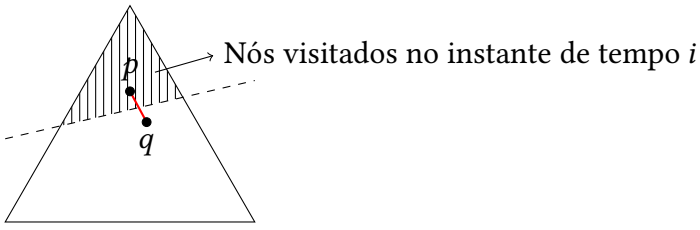


Figura 4.5: *Representação da situação descrita. O triângulo é uma representação simplificada de uma ABB. A área destacada representa todos os nós acessados no instante de tempo i. O nó q não foi visitado nesse instante de tempo.*

De maneira informal, os nós p e q são nós pai-filho ao término do instante i. Durante o instante de tempo i, p será visitado pelo algoritmo de busca e q não. Após o instante i, queremos visitar o nó q pela primeira vez no instante $j = N(b, i)$ antes de fazer qualquer outra visita ao nó p, porém isso é impossível. Vamos mostrar que, neste caso, o conjunto inicial P não era arboreamente satisfeito.

Os lados a seguir terão um papel na prova e estão representados na Figura 4.6. São eles:

- $\ell_1 = \{(x, y) \mid x = a \text{ e } i < y \leq j\}$, e
- $\ell_2 = \{(x, y) \mid a < x \leq b \text{ e } y = i\}$,

onde $\ell_1 \cap P = \emptyset$, pois o nó p não é visitado entre os instantes de tempo $i + 1$ e j já que $N(a, i) > j$, e $\ell_2 \cap P$ representa todas as visitas aos nós com chave no intervalo $(a, b]$ no instante de tempo i.

Denotemos por r o ponto com coordenadas (a, i) e por s o ponto com coordenadas (b, j). De acordo com a suposição feita, ambos estes pontos pertencem a P.

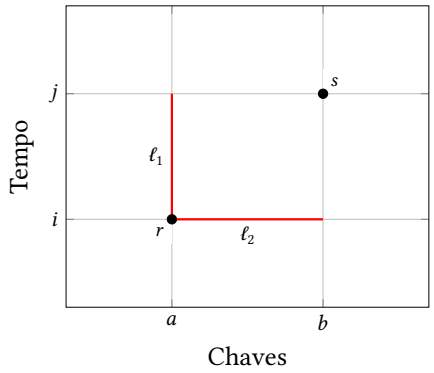


Figura 4.6: *Representação no conjunto P da situação descrita.*

Como $r, s \in P$, então $\{r, s\}$ representa um par de pontos arboreamente satisfeito. De acordo com o Lema 4.1, como r e s não são ortogonalmente colineares, há pelo menos um outro ponto de P em alguma das arestas do $\{r, s\}$-retângulo que incide em r. Como $\ell_1 \cap P = \emptyset$, então $\ell_2 \cap P \neq \emptyset$. Assim, há um ponto de P no lado ℓ_2 e, consequentemente, algum nó com chave no intervalo $(a, b]$ deve ter sido visitado durante o instante de tempo i.

Como q é filho direito de p, então qualquer nó de chave no intervalo (a, b) no instante de tempo i, pela propriedade da ordenação de ABBs, está na subárvore esquerda do nó q. Assim, para visitar qualquer nó com chave neste intervalo é necessário visitar ambos os nós p e q. Porém, pela hipótese, o nó q não é visitado em i, logo $\ell_2 \cap P = \emptyset$, uma contradição.

Assim, nota-se que a reestruturação da treap da maneira proposta é sempre válida em conjuntos de pontos arboreamente satisfeitos e tal reestruturação descreve um algoritmo offline que converte um conjunto de pontos arboreamente satisfeitos em uma execução de um algoritmo de busca em ABB dentro do modelo de computação adotado. □

Verificar se um conjunto de pontos é arboreamente satisfeito é demorado se feito à mão. Assim, com o intuito de automatizar e garantir que um conjunto de pontos é arboreamente satisfeito, implementamos a treap proposta na demonstração do Lema 4.3. Se a propriedade não for violada, o código imprime a árvore correspondente a cada instante de tempo. Caso a propriedade seja violada, o código imprime todas as árvores correspondentes aos instantes de tempo anteriores ao da violação e imprime o par de pontos que é arboreamente insatisfeito. A implementação pode ser vista aqui.

É importante notar que essa interpretação geométrica é muito útil para entender buscas em ABBs de maneira simplificada. Essa abordagem convenientemente oculta as rotações da ABB e leva em consideração apenas os nós visitados durante as buscas. É bastante impressionante que a disposição exata dos nós de uma ABB não é uma informação essencial para a análise e pode ser reconstruída a partir dos nós visitados.

Os resultados desse capítulo implicam que o tamanho do menor conjunto de pontos arboreamente satisfeito que contém P_X é exatamente $\mathrm{OPT}(X)$. No próximo capítulo apresentaremos um algoritmo guloso que, dado uma sequência X de acessos, produz um conjunto arboreamente satisfeito que contém P_X tentando minimizar o tamanho desse conjunto. Esse algoritmo é online e nem sempre obtém um conjunto de tamanhao $\mathrm{OPT}(X)$.

Capítulo 5

Algoritmo guloso

Neste capítulo entenderemos como a visão geométrica de um algoritmo de busca em ABB para uma sequência de acessos X se relaciona com o custo ótimo $OPT(X)$. Além disso, será apresentado um algoritmo guloso offline que transforma um conjunto arboreamente insatisfeito em um conjunto arboreamente satisfeito adicionando uma série de pontos, tentando adicionar o menor número possível de pontos ao conjunto. Por fim, argumentaremos como esse algoritmo pode ser adaptado para um algoritmo online em ABBs dentro do modelo de computação adotado.

5.1 Otimalidade

Revisemos a definição de custo nesse modelo de computação. O custo para realizar um acesso é o número de nós visitados durante essa operação. Assim, $OPT(X)$ é o menor custo necessário para um algoritmo de busca offline em ABB realizar todos os acessos de uma entrada $X = (x_1, \dots, x_m)$, ou seja, o número mínimo de visitas a nós necessárias para realizar todos os acessos.

Retornando à análise geométrica, seja P um conjunto de pontos. Denotaremos por $\text{minASS}(P)$ o tamanho do menor conjunto arboreamente satisfeito que contém P.

Seja P a visão geométrica de um algoritmo de busca em ABB que possui custo $OPT(X)$ para a sequência de acessos X. De acordo com o Lema 4.2, P é um conjunto arboreamente satisfeito. Além disso, $|P|$ é mínimo para a entrada X, pois P é a visão geométrica de um algoritmo de busca em ABB que possui o custo ótimo $OPT(X)$. Assim, nota-se que $OPT(X) = \text{minASS}(P_X) = |P|$.

Isso implica que o problema de encontrar o custo de um algoritmo ótimo de busca em ABB para uma sequência X de acessos pode ser reformulado como o problema de encontrar o menor superconjunto arboreamente satisfeito do conjunto P_X.

5.2 Guloso futurista

Apesar de não se saber se é possível encontrar o valor de $OPT(X)$ em tempo polinomial, e consequentemente encontrar o menor superconjunto de P_X arboreamente satisfeito, apresentaremos o algoritmo Guloso futurista — Greedy Future — que dado o conjunto P_X de pontos que representa a sequência X de acessos, produz um conjunto P de pontos pequeno que contém P_X e é arboreamente satisfeito. Esse algoritmo foi descoberto de maneira independente por Lucas [7] e por Munro [9].

Naturalmente $|P_X| \geq |X| = m$ e todos os pontos de P_X possuem coordenadas y distintas dentro do intervalo $[1, m]$.

O algoritmo funciona da seguinte maneira: inicialmente defina uma reta horizontal r em $y = 1$ e inicialize $P = P_X$. Seja $a \in P \cap r$ um ponto com coordenadas $(a.x, a.y)$. Para cada $\{a, b\}$-retângulo insatisfeito em P, com b um ponto com coordenadas $(b.x, b.y)$, com $b.y < a.y$, adicione a P um ponto em $(b.x, a.y)$. Após satisfazer todos os $\{a, b\}$-retângulos desse tipo, mova r uma unidade para cima e repita. O algoritmo termina após satisfazer todos os $\{a, b\}$-retângulos de P com r em $y = m$. Veja a Figura 5.1.

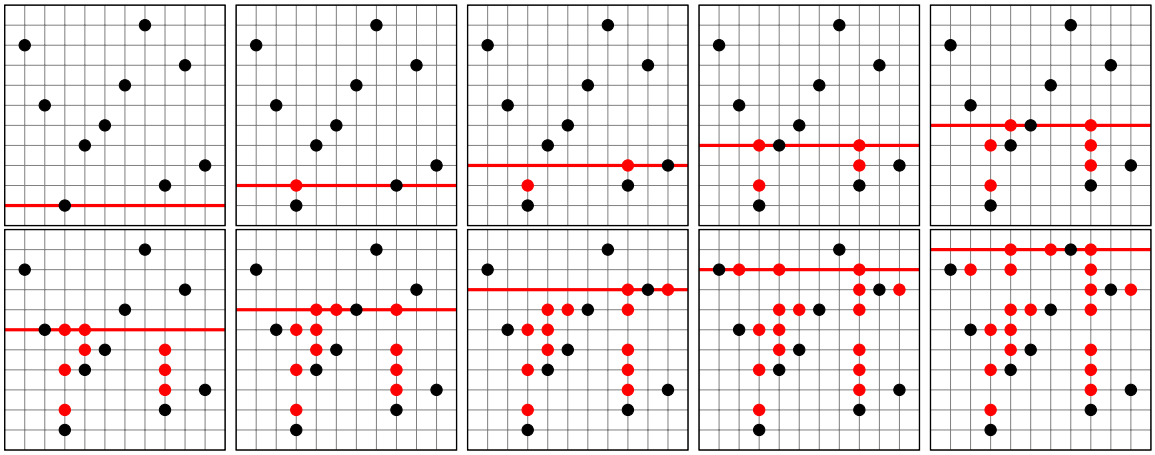


Figura 5.1: *Execução do Greedy Future para a sequência X = (3,8,10,4,5,2,6,9,1,7) de acessos.*

O algoritmo mantém o invariante que, ao final da iteração do algoritmo para a reta horizontal $y = l$, todos os pares de ponto $\{a, b\}$, com $a, b \in P$ e $a.y, b.y \leq l$ são arboreamente satisfeitos. Assim, por construção, o resultado do algoritmo é um conjunto arboreamente satisfeito.

O comportamento desse algoritmo é caracterizado por uma abordagem gulosa local. Esse algoritmo, quando refletido no contexto de ABBs, visita apenas os nós que estão no caminho da raiz até o nó com chave buscada e reorganiza todos os nós visitados de maneira a deixar mais perto da raiz, os nós que serão visitados mais cedo, e consequentemente, mais longe da raiz os nós que serão visitados mais tarde.

A Figura 5.2 evidencia este comportamento. A primeira coluna é uma cópia da Figura 5.1. A segunda coluna mostra o caminho utilizado pelo algoritmo de busca para executar o acesso da sequência X e a terceira coluna representa a disposição final da ABB após as rotações serem realizadas preparando-se para os próximos acessos. As ABBs da segunda e

terceira coluna estão propositalmente invertidas (com a raiz embaixo e as folhas em cima) para evidenciar que todos os nós visitados em cada acesso (nós em vermelho) são os nós representados pelos pontos na reta r daquele instante.

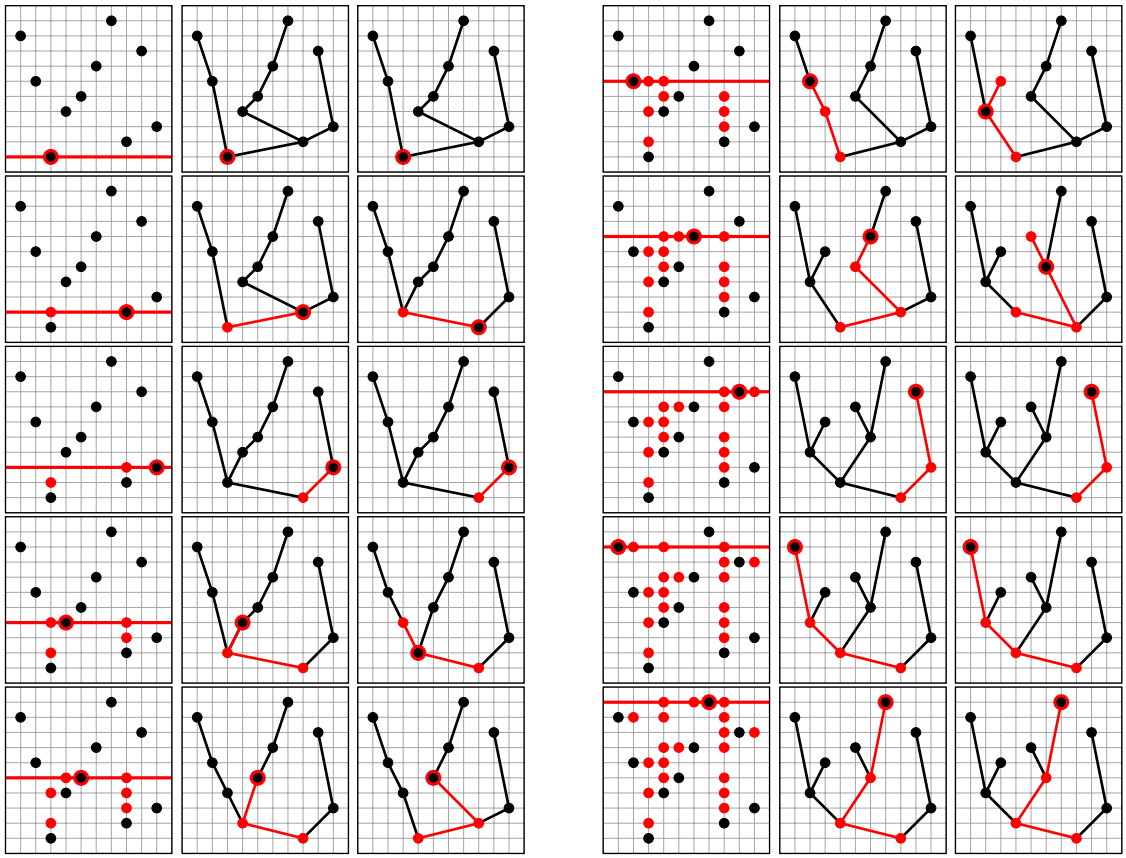


Figura 5.2: *Na primeira coluna, a execução do guloso futurista para* $X = (3, 8, 10, 4, 5, 2, 6, 9, 1, 7)$. *Na segunda coluna, a execução de cada acesso na ABB correspondente. Na terceira coluna a ABB final após as rotações executadas depois do acesso da coluna anterior. Note que os nós na linha* r *do algoritmo guloso da coluna da esquerda são exatamente os nós visitados na coluna do meio, e passíveis de reestruturação na última coluna.*

Como mostrado no Lema 4.3, e evidenciado pela Figura 5.2, é possível a partir unicamente do conjunto de pontos P resultante da execução do algoritmo guloso futurista para uma sequência X de acessos construir os passos que devem ser executados para um algoritmo de busca executar todos os acessos de X apenas visitando os nós representados pelos pontos de P.

Note que para isso foi necessário inicializar a árvore com os nós ordenados pelo primeiro acesso de cada nó e também foi necessário saber a próxima visita aos nós para ter a informação de como reestruturar a ABB durante os acessos de maneira a apenas visitar os nós propostos. Assim, apesar do guloso futurista ser um algoritmo online, a princípio a sua tradução para um algoritmo de busca em ABB necessita de mais informação e, portanto, é offline.

Demaine et al. [3] provaram que utilizando uma estrutura abstrata de dados chamada árvore split que implementa as operações maketree e split no modelo de computação

adotado é possível converter um algoritmo online do contexto de conjunto de pontos arboreamente satisfeitos em um algoritmo online do contexto de buscas em ABBs com custo extra limitado por um fator constante. A ideia central é evitar tomar decisões sobre como dispor os nós na ABB e armazenar esses nós em uma árvore split. Dessa maneira é possível utilizar a operação split no futuro quando for necessário visitar um dos nós presentes nesta árvore split. Essa prova é bastante enigmática e de difícil compreensão, envolvendo uma série de estruturas de dados como árvores treap, árvores rubro-negras e árvores 2-3-4. Para esse texto, basta perceber que é possível converter o algoritmo guloso futurista em um algoritmo online de busca em ABBs sem aumento significativo do seu custo.

Embora o algoritmo garanta encontrar um superconjunto P arboreamente satisfeito de P_X, o guloso futurista não garante que $|P| = \text{minASS}(P_X)$. Veja a Figura 5.3.

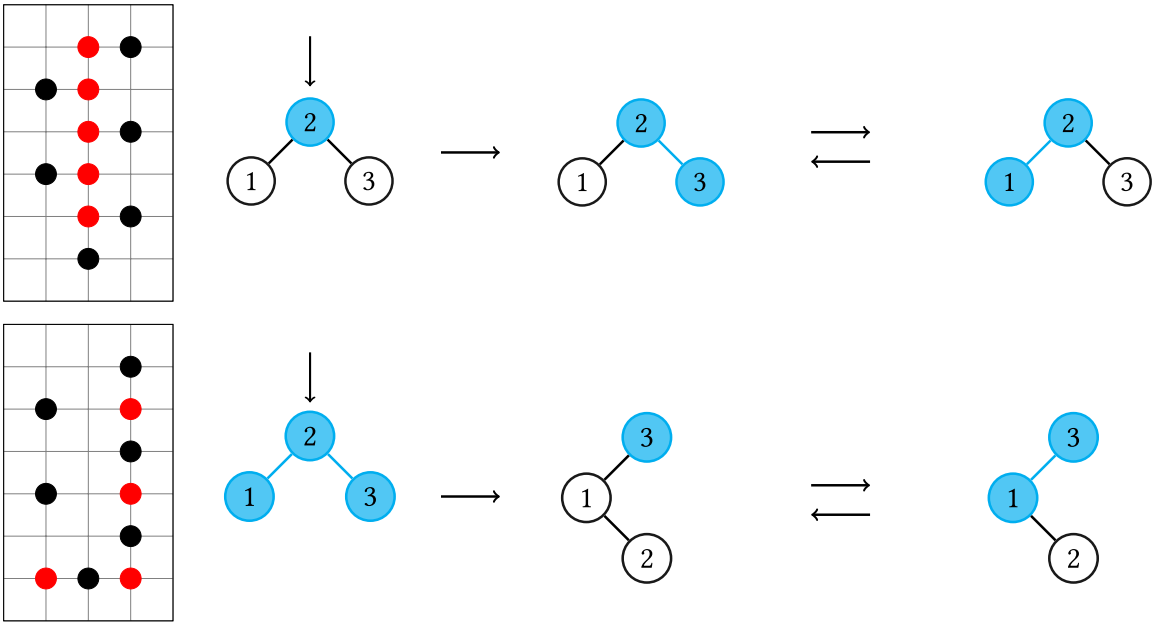


Figura 5.3: *Em cima, o conjunto de pontos P produzido pelo algoritmo guloso futurista para a sequência $X = (2, 3, 1, 3, 1, 3)$ de acessos. À direita desse conjunto de pontos está a ABB correspondente onde as buscas são feitas com os nós visitados em cada busca destacados. Abaixo, um conjunto de pontos P' com tamanho* $\text{minASS}(P_X)$*. À direita desse conjunto está a ABB correspondente onde os nós visitados em cada acessos estão destacados.*

Analisando o caso acima, perceba que o guloso futurista, por ser um algoritmo online, sempre assume que a primeira chave buscada é a chave da raiz da árvore inicial em sua execução refletida no contexto de ABBs. Após o acesso à chave 2, todas as buscas se alternam entre as chaves 1 e 3, porém a chave 2 não é mais buscada. Assim, fica óbvio perceber que o nó com chave 2 deveria ser retirado do caminho das duas folhas, porém pelo algoritmo fazer decisões gulosas locais, para ele rotacionar o nó com chave 2 para baixo é necessário que ele visite todos os nós da árvore e isso não pode ocorrer. No caso ótimo é exatamente isso que acontece: o algoritmo visita todos os nós no início e já rotaciona o nó com chave 2 para a folha, garantindo assim que esse nó não vai ser visitado de maneira desnecessária no futuro.

Para uma sequência X de acessos com tamanho m, formada por uma busca à chave 2 e

depois buscas alternadas entre as chaves 1 e 3, é possível notar que o custo do algoritmo guloso futurista é $OPT(X) + m/2$.

O pior caso encontrado do algoritmo guloso futurista foi proposto por Munro [9]. Esse caso acontece quando o algoritmo guloso futurista busca, em uma ABB completa, na ordem da sequência bit-reversa e em seguida busca apenas as chaves seguindo a mesma sequência. Munro conjecturou que o custo do guloso futurista é $OPT(X) + O(m)$ para toda sequência X de acessos. Essa sequência particular apresentada por Munro é de extrema importância para o estudo de delimitações de custo em algoritmos de busca em ABBs que permitem rotações e será explicada mais à frente no Capítulo 7.

Se provada a conjectura de Munro, que o custo do guloso futurista é $OPT(X) + O(m)$ para toda sequência X de acessos, como $OPT(X) \geq m$, então prova-se também que o algoritmo guloso futurista tem custo $O(OPT(X))$. Assim, o algoritmo de busca em ABBs resultante da conversão do algoritmo guloso futurista para o contexto de ABBs possuiria também custo $O(OPT(X))$ e seria dinamicamente ótimo. Ou seja, essa conjectura de Munro implica que a implementação online de Demaine et al. [3] do guloso futurista é dinamicamente ótima.

Capítulo 6

Conjuntos independentes de retângulos

Neste capítulo entenderemos como utilizar a visão geométrica da execução de um algoritmo de busca em ABB para delimitar inferiormente custos em sequências de acesso. Introduziremos as noções de orientação e independência de retângulos que serão fundamentais para a elaboração e análise dos algoritmos gulosos futuristas orientados. Explicaremos como funciona a delimitação dos retângulos independentes e por fim faremos um estudo da qualidade dessa delimitação inferior.

6.1 Orientação de retângulos

Para a construção de provas mais sofisticadas, é necessário entender melhor características da geometria dos retângulos formados por pontos de um conjunto. Lembre-se que todos os retângulos aqui considerados são ortogonais aos eixos cartesianos. Por conveniência das provas, todas as sequências de acessos desse capítulo possuem exatamente um acesso por chave, ou seja, $m = n$ e a sequência de busca é uma permutação de 1 a n.

Dividiremos os retângulos formados por um par de pontos de um conjunto P em dois grupos: os ◿-retângulos e os ◺-retângulos. Denotamos por ◿*-retângulo* um retângulo que possui o vértice de P à esquerda mais abaixo que o vértice de P à direita. Denotamos por ◺*-retângulo* um retângulo que possui o vértice de P à esquerda mais acima que o vértice de P à direita.

Assim, definimos que um conjunto de pontos P é ◿*-satisfeito* se todos os ◿-retângulos formados por dois pontos de P não ortogonalmente colineares são arboreamente satisfeitos. O conceito de ◺-satisfeito é definido simetricamente.

Também será útil definir quando ambos os tipos de retângulos são satisfeitos. Chamamos um superconjunto Z de P de ⊠*-satisfeito* em relação a P se existem subconjuntos $Z_{◿}$ e $Z_{◺}$ de Z tais que $Z_{◿} \cup P$ é ◿-satisfeito e $Z_{◺} \cup P$ é ◺-satisfeito.

Por fim, definimos $minASS_{◿}(P)$ como o tamanho de um menor superconjunto de P que é ◿-satisfeito. Analogamente, $minASS_{◺}(P)$ é o tamanho de um menor superconjunto

de P que é ⧅-satisfeito e $minASS_{⧆}(P)$ é o tamanho do menor superconjunto de P que é ⧆-satisfeito em relação a P.

Note que qualquer união entre dois conjuntos $Z_{◪}$ e $Z_{◹}$, onde $Z_{◪}$ é ◪-satisfeito, $Z_{◹}$ é ◹-satisfeito e ambos são superconjuntos de P, é ⧆-satisfeita em relação a P. Veja a Figura 6.1. Porém, $Z_{◪} \setminus P$ e $Z_{◹} \setminus P$ não necessariamente são disjuntos entre si. Levando em consideração ambas as propriedades, conclui-se então a Equação (6.1).

$$\text{minASS}_{⧆}(P) \leq \text{minASS}_{◪}(P) + \text{minASS}_{◹}(P). \tag{6.1}$$

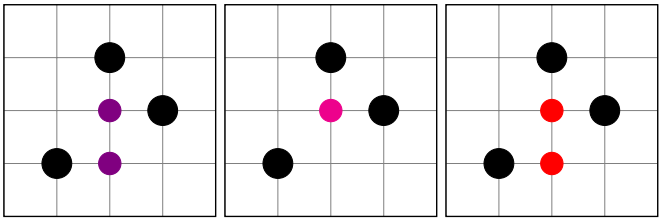


Figura 6.1: *Em preto, os pontos de P_X para a sequência de acessos $X = (1, 3, 2)$. À esquerda, em roxo, um conjunto de pontos $Z_{◪}$, onde $Z_{◪} \cup P_X$ é ◪-satisfeito e possui tamanho $minASS_{◪}$. Ao centro, em rosa, um conjunto de pontos $Z_{◹}$, onde $Z_{◹} \cup P_X$ é ◹-satisfeito e possui tamanho $minASS_{◹}$. À direita, em vermelho, um conjunto de pontos $Z_{⧆}$, onde $Z_{⧆} \cup P_X$ é ◪-satisfeito e ◹-satisfeito e possui tamanho $minASS_{⧆}$. Perceba que $Z_{⧆} = Z_{◪} \cup Z_{◹}$ e $Z_{◪} \cap Z_{◹} \neq \emptyset$.*

Outro ponto fundamental de ser compreendido é que todo conjunto Y, superconjunto de P, arboreamente satisfeito também é ⧆-satisfeito em relação a P. Isso acontece pois por conta do conjunto Y ser superconjunto de P e arboreamente satisfeito, então $Y \cup P = Y$ é ◪-satisfeito e ◹-satisfeito. Assim, temos que

$$\text{minASS}_{⧆}(P) \leq \text{minASS}(P). \tag{6.2}$$

6.2 Guloso futurista orientado

Definimos agora dois algoritmos muito parecidos com o guloso futurista visto no capítulo anterior, que nos ajudarão a descrever delimitações inferiores para sequências de acessos.

Chamaremos de guloso *futurista-◪* o seguinte algoritmo que recebe um conjunto P_X de pontos que representa uma sequência X de acessos. O algoritmo começa com $P = P_X$ e desliza uma reta horizontal r que começa em $y = 1$ e vai até $y = m$. O algoritmo mantém o invariante que todos os pares de pontos de P em r ou abaixo de r são ◪-satisfeitos. Esse algoritmo garante essa propriedade adicionando a P o menor número de pontos na linha r durante cada iteração. Ao fim da execução, o conjunto P de pontos é ◪-satisfeito. O algoritmo guloso futurista-◹ é definido de maneira simétrica considerando a ◹-satisfação.

Chamaremos de $\text{Fut}_{◪}(P)$ o conjunto de pontos adicionados durante a execução do algoritmo guloso futurista-◪ para um conjunto P de pontos. Todas essas definições valem simetricamente para o algoritmo guloso futurista-◹. Veja a Figura 6.2.

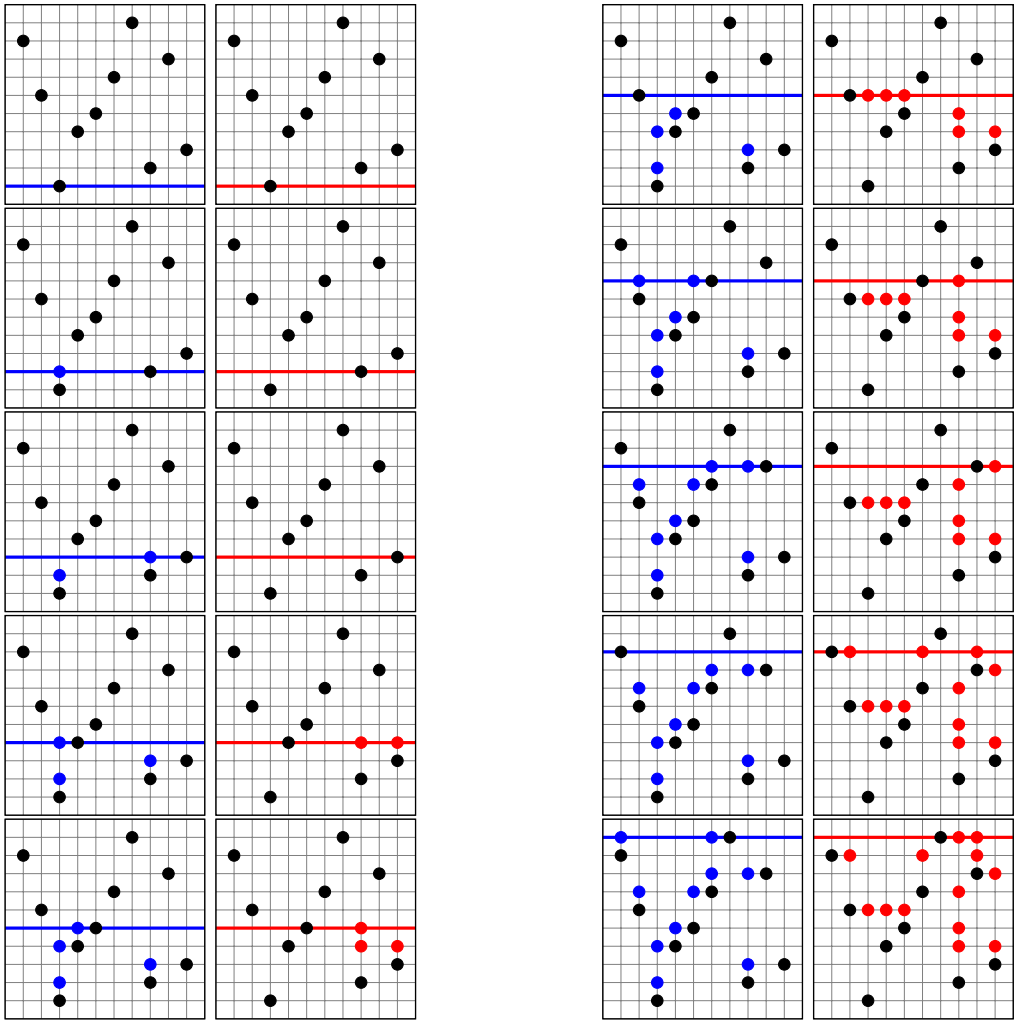


Figura 6.2: *Em preto, o conjunto P_X da sequência $X = (3, 8, 10, 4, 5, 2, 6, 9, 1, 7)$ de acessos. Em azul, o conjunto obtido pelo guloso futurista-◿para P_X e em vermelho, o do guloso futurista-◺.*

Definimos $R(q)$ como o par de pontos (a, b) do conjunto inicial tal que q foi um ponto adicionado durante algum dos algoritmos gulosos futuristas orientados e o $\{a, b\}$-retângulo possui q em um de seus vértices superiores. Esse par de pontos existe e é bem definido. Ele existe, pois os algoritmos gulosos futuristas orientados apenas adicionam o mínimo de pontos durante cada iteração para satisfazer arboreamente o conjunto de pontos que estão na linha descrita ou abaixo dela. Se não houvesse tal par de pontos, então o ponto q foi adicionado para satisfazer um outro par de pontos (c, d) onde c e/ou d são pontos que também foram adicionados pelo algoritmo. Porém, se c e/ou d foram adicionados pelo algoritmo então deve existir algum par de pontos mais ao extremo no mesmo sentido de c e d com pontos que fazem parte do conjunto inicial, ou seja, existe o par (a, b). Esse par de pontos é bem definido, pois assumimos que os conjuntos de pontos analisados nesse capítulo possuem as propriedades de x e y-coordenadas distintas.

Esses algoritmos terão um papel essencial para a última seção desse capítulo, onde buscamos delimitações inferiores de custo para sequências de acesso.

6.3 Independência de retângulos

Um conjunto I de retângulos é *independente* se todos os retângulos de I são arboreamente insatisfeitos e não há nenhum vértice de algum dos retângulos de I estritamente no interior de outro retângulo de I. Veja a Figura 6.3.

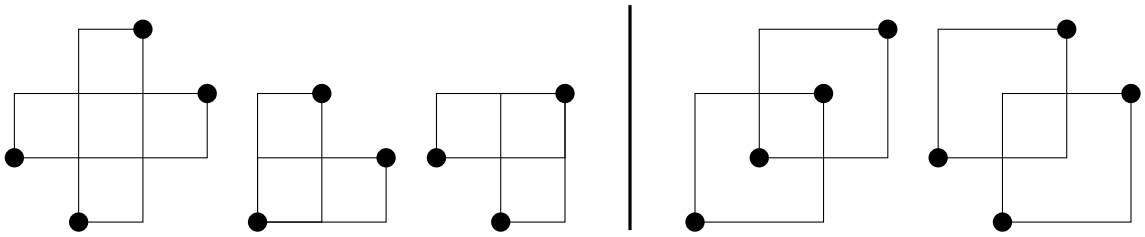


Figura 6.3: *Todas as combinações de dois ◪-retângulos que compartilham área no seu interior. À esquerda, pares de retângulos independentes e, à direita, pares de retângulos dependentes.*

Lema 6.1. *Para toda sequência X de acessos, existe um conjunto independente $IRB_{◪}(P_X)$ de ◪-retângulos tal que $|IRB_{◪}(P_X)| = |Fut_{◪}(P_X)|$.*

Demonstração. Para todo ponto q em $\text{Fut}_{◪}(P_X)$, sejam r e s o par de pontos de $R(q)$, sendo r o ponto de P_X abaixo de q e s o ponto de P_X à direita de q. Como $R(q)$ é bem definido, então $R(q) = R(t)$ se e somente se $q = t$. Seja $\text{IRB}_{◪}(P_X) = \{R(q) \mid q \in \text{Fut}_{◪}(P_X)\}$. Obviamente $|\text{IRB}_{◪}(P_X)| = |\text{Fut}_{◪}(P_X)|$ e por construção nenhum ponto de $\text{Fut}_{◪}(P_X) \cup P_X$ está estritamente dentro de algum retângulo de $\text{IRB}_{◪}$ e o vértice superior esquerdo de cada retângulos de $\text{IRB}_{◪}(P_X)$ está em $\text{Fut}_{◪}(P_X) \cup P_X$. □

Para visualizar a ideia da prova acima, veja a Figura 6.4. Note que analogamente à prova, o conjunto de ◪-retângulos independentes mostrado possui o vértice superior esquerdo localizado em algum ponto adicionado pelo algoritmo guloso futurista-◪.

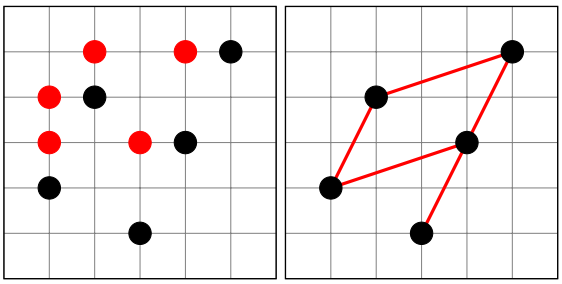


Figura 6.4: *Em preto, os pontos de P_X para a sequência $X = (3, 1, 4, 2, 5)$ de acessos. À esquerda, a execução do guloso futurista-◪. À direita, está destacado um conjunto independente de ◪-retângulos.*

Lema 6.2. *Considere um conjunto ◪-satisfeito Y de pontos com x-coordenadas inteiras, dois pontos a e b não ortogonalmente colineares tais que $a, b \in Y$, e uma linha vertical ℓ com x-coordenada não inteira estritamente entre $a.x$ e $b.x$. Então, existem dois pontos $p, q \in Y$ tais que $p.y = q.y$, p está à esquerda de ℓ e q está à direita de ℓ e não há nenhum ponto de Y no segmento de reta horizontal que conecta p e q.*

Demonstração. Assuma sem perda de generalidade que $a.x < b.x$. Seja p o ponto de Y à esquerda de ℓ mais à direita e, dentre estes, o mais acima do $\{a, b\}$-retângulo e seja q o ponto de Y à direita de ℓ mais abaixo e, dentre estes, o mais a esquerda que esteja na mesma altura ou acima de q no $\{a, b\}$-retângulo. Esses pontos existem pois a e b satisfazem as restrições de p e q respectivamente e Y é $\boxslash$-satisfeito. Por construção, p está à esquerda de ℓ e q à direita. Nota-se que $p.y = q.y$ obrigatoriamente, caso contrário o $\{p, q\}$-retângulo estaria insatisfeito, o que contradiz o que fato de Y ser $\boxslash$-satisfeito. Veja a Figura 6.5 para uma ilustração dessa contradição. □

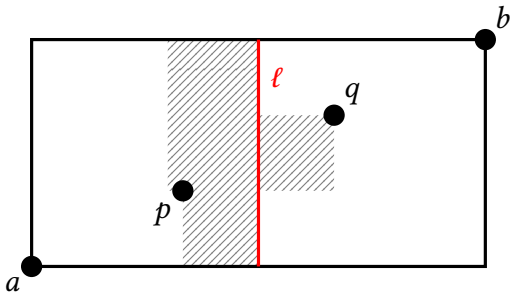


Figura 6.5: *Um $\{a, b\}$-retângulo de um conjunto $\boxslash$-satisfeito Y de pontos. O ponto p é o mais à direita e mais acima de Y à esquerda da reta ℓ e o ponto q é o mais à esquerda e mais abaixo que está à direita de ℓ e acima de p. A área destacada não pode conter pontos de Y por conta da escolha de p e q.*

Lema 6.3. *Para um conjunto independente I de retângulos de um conjunto P de pontos tal que todo ponto tem x-coordenada distinta e inteira, existe um $\{a, b\}$-retângulo com $(a, b) \in I$, e uma reta vertical ℓ que cruza o eixo x num valor não inteiro estritamente entre $a.x$ e $b.x$ tal que, dentro deste $\{a, b\}$-retângulo, ℓ não intersecta nenhum outro retângulo de I.*

Demonstração. Sejam a e b os dois vértice de um retângulo com largura máxima em I, ou seja, o retângulo com maior comprimento horizontal. Assuma por simetria que $a.x < b.x$. Como I é um conjunto independente de retângulos, então nenhum retângulo de I possui estritamente em seu interior algum vértice de outro retângulo de I.

Assim, existem três tipos possíveis de retângulos de I que compartilham alguma parte da área interna com o $\{a, b\}$-retângulo: o tipo-1 são os retângulos que possuem a como um de seus vértices, o tipo-2 são os retângulos que possuem b como um de seus vértices e o tipo-3 são os retângulos que não possuem nem a nem b como um de seus vértices. Note que os retângulos do tipo-3 não podem ser mais largos que o $\{a, b\}$-retângulo, pois o $\{a, b\}$-retângulo é o retângulo com largura máxima em I. Veja a Figura 6.6 para a visualização desses três tipos de retângulos.

Perceba que só é possível a criação de três blocos de retângulos. À esquerda, pode ser criado um bloco de retângulos do tipo-1 e do tipo-3. Ao meio, pode ser criado um bloco de retângulos do tipo-3. À direita, pode ser criado um bloco de retângulos do tipo-2 e do tipo-3. O bloco de retângulos à esquerda e o bloco de retângulos ao meio não podem ter áreas compartilhadas, pois o conjunto I é independente e o conjunto P tem x-coordenada distinta. O mesmo argumento vale para mostrar que o bloco de retângulos ao meio não pode ter área compartilhada com o bloco de retângulos à direita.

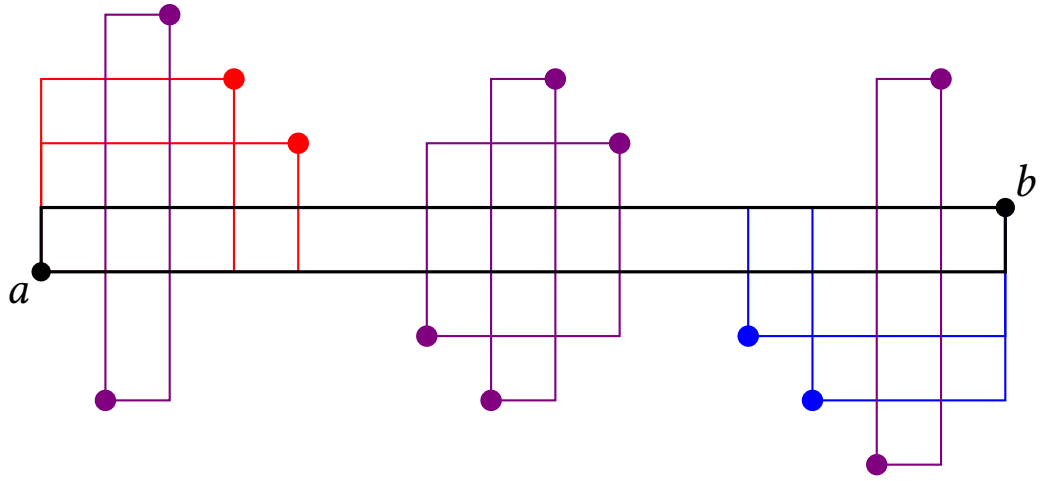


Figura 6.6: *Em preto, o $\{a, b\}$-retângulo sendo analisado. Em vermelho, os retângulos independentes que compartilham o vértice a. Em azul, os retângulos independentes que compartilham o vértice b. Em roxo, os retângulos independentes que não compartilham vértice em comum com o $\{a, b\}$-retângulo.*

Como os pontos de P possuem x-coordenada inteira, então é possível definir uma reta ℓ vertical que cruza o eixo x num valor não inteiro estritamente entre a x-coordenada do ponto mais à direita de P pertencente a um dos retângulos do primeiro bloco e a x-coordenada do ponto mais à esquerda de P pertencente a um dos retângulos do segundo bloco. O mesmo poderia ser feito para o segundo e terceiro bloco. A reta ℓ, dentro do $\{a, b\}$-retângulo, não intersecta nenhum retângulo além do $\{a, b\}$-retângulo. □

Lema 6.4. *Dado um conjunto independente I de $\boxslash$-retângulos de um conjunto P_X de pontos, qualquer superconjunto Y de P_X que seja $\boxslash$-satisfeito tem pelo menos cardinalidade $|I| + |P_X|$.*

Demonstração. Utilizando o conjunto I, é possível aplicar o Lema 6.3 para encontrar um $\{a, b\}$-retângulo em I e uma reta vertical ℓ, onde ℓ não intercepta nenhum outro retângulo dentro da área do $\{a, b\}$-retângulo. Utilizando esse $\{a, b\}$-retângulo e essa reta ℓ, podemos aplicar o Lema 6.2 para encontrar dois pontos p, q, com $p, q \in P$, tais que p está à esquerda de ℓ e q à direita. Assim, marcamos o par de pontos (p, q), removemos o $\{a, b\}$-retângulo de I e repetimos o processo até não sobrar nenhum retângulo em I.

Quando removemos um $\{a, b\}$-retângulo de I, por conta da propriedade de toda reta vertical ℓ não interceptar nenhum outro retângulo dentro da área do $\{a, b\}$-retângulo analisado, então nota-se que nenhum outro retângulo de I possui em seu interior simultaneamente tanto p quanto q. Assim, nota-se que cada par de pontos só será marcado uma única vez.

Como P_X possui apenas um ponto por coordenada y, então, para cada par de pontos marcado, no máximo um ponto pertence a P_X, então pelo menos um deles pertence apenas ao superconjunto Y. Logo, o número de pontos de $Y \setminus X$ é pelo menos o número de retângulos em I. □

Lema 6.5. *Para qualquer conjunto P_X de pontos, $minASS_{\boxslash}(P_X) = |Fut_{\boxslash}(P_X)| + |P_X|$.*

Demonstração. Após a execução do algoritmo $\boxslash$-futurista para um conjunto P_X de pontos

representando a sequência X de acessos, o conjunto de pontos P resultante desse algoritmo possui tamanho $|\text{Fut}_{\nearrow}(P_X)| + |P_X|$. Pelo Lema 6.1, existe um conjunto independente $\text{IRB}_{\nearrow}(P_X)$ de $\nearrow$-retângulos tal que $|\text{IRB}_{\nearrow}(P_X)| = |\text{Fut}_{\nearrow}(P_X)|$. Pelo Lema 6.4, como existe esse tal conjunto $\text{IRB}_{\nearrow}(P_X)$, então qualquer superconjunto Y de P_X que seja $\nearrow$-satisfeito possui cardinalidade maior ou igual a $|\text{IRB}_{\nearrow}(P_X)| + |P_X|$. Como o conjunto P de pontos é $\nearrow$-satisfeito e possui tamanho $|\text{Fut}_{\nearrow}(P_X)| + |P_X| = |\text{IRB}_{\nearrow}(P_X)| + |P_X|$, então esse é um superconjunto $\nearrow$-satisfeito minimal de P_X e $\text{minASS}_{\nearrow}(P_X) = |\text{Fut}_{\nearrow}(P_X)| + |P_X|$. □

Teorema 6.6. *Se um conjunto P_X de pontos contém um conjunto independente I de retângulos, então $\text{minASS}_{\boxtimes}(P_X) \geq |I|/2 + |P_X|$.*

Demonstração. Seja Z qualquer conjunto $\boxtimes$-satisfeito em relação ao P_X, onde $Z_{\nearrow} \cup P_X$ é $\nearrow$-satisfeito e $Z_{\searrow} \cup P_X$ é $\searrow$-satisfeito. Seja $I_{\nearrow}$ o conjunto independente de $\nearrow$-retângulos de I e analogamente seja $I_{\searrow}$ o conjunto independente de $\searrow$-retângulos de I. De acordo com o Lema 6.4, $|Z_{\nearrow} \cup P_X| \geq |I_{\nearrow}| + |P_X|$ e também $|Z_{\searrow} \cup P_X| \geq |I_{\searrow}| + |P_X|$. Suponha por simetria que $|I_{\nearrow}| \geq |I_{\searrow}|$, então $|I_{\nearrow}| \geq |I|/2$. Logo, $|Z| \geq |Z_{\nearrow} \cup P_X| \geq |I_{\nearrow}| + |P_X| \geq |I|/2 + |P_X|$. Como todos os conjuntos $\boxtimes$-satisfeitos em relação ao P_X possuem essa propriedade, então o conjunto de tamanho $\text{minASS}_{\boxtimes}(P_X)$ também deve ter. □

6.4 Delimitação dos retângulos independentes

De acordo com o Teorema 6.6, conjuntos independentes de retângulos estão atrelados intimamente com delimitações inferiores de custo. Assim, se conseguirmos encontrar um conjunto independente maior de retângulos, então encontraremos o melhor delimitante de custo, ou seja, o delimitante que mais se aproxima de $\text{OPT}(X)$.

Denotemos por $\text{maxIRB}(P_X)$ o maior conjunto independente de retângulos para o conjunto P_X de pontos que representa uma sequência X de acessos. Não se sabe se há uma maneira de encontrar o valor de $|\text{maxIRB}(P_X)|$ em tempo polinomial, muito menos se há uma maneira de encontrar o próprio conjunto independente de retângulos.

$$
\begin{aligned}
\frac{1}{2}|\text{maxIRB}(P_X)| + |P_X| &\leq \text{minASS}_{\boxtimes}(P_X) && \text{pelo Teorema 6.6,}\\
&\leq \text{minASS}_{\nearrow}(P_X) + \text{minASS}_{\searrow}(P_X) && \text{pela Equação (6.1),}\\
&= |Fut_{\nearrow}(P_X)| + |Fut_{\searrow}(P_X)| + 2|P_X| && \text{pelo Lema 6.5,}\\
&= |\text{IRB}_{\nearrow}(P_X)| + |\text{IRB}_{\searrow}(P_X)| + 2|P_X| && \text{pelo Lema 6.1,}\\
&\leq 2|\text{maxIRB}(P_X)| + 2|P_X|,\\
&< 2|\text{maxIRB}(P_X)| + 4|P_X|,\\
&\leq 4\text{minASS}_{\boxtimes}(P_X) && \text{pelo Teorema 6.6,}\\
&\leq 4\text{minASS}(P_X) && \text{pela Equação (6.2).}
\end{aligned}
$$

Uma maneira de encontrar um conjunto independente de retângulos é utilizando os algoritmos gulosos futuristas orientados. Isso será feito da seguinte maneira: rode os algoritmos $\nearrow$-futurista e $\searrow$-futurista e escolha o que colocar mais pontos. Seguindo os

mesmos passos da expressão acima, temos

$$\begin{aligned}\max\{|Fut_{\nearrow}(P_X)| + |Fut_{\searrow}(P_X)|\} + |P_X| &\geq \frac{1}{2}(|Fut_{\nearrow}(P_X)| + |Fut_{\searrow}(P_X)|) + |P_X|,\\ &\geq \frac{1}{4}\text{maxIRB}(P_X) + \frac{1}{2}|P_X|.\end{aligned}$$

Dessa maneira, o tamanho do conjunto independente de retângulos encontrado pelo melhor algoritmo guloso futurista orientado para um determinado conjunto de pontos P_X está em um fator constante do tamanho do maior conjunto independente de retângulos.

No próximo capítulo, veremos as delimitações inferiores de Wilber. Estas delimitações são algoritmos para delimitar custos em sequências de entradas e que conseguem encontrar conjuntos de retângulos independentes de outras maneiras.

Capítulo 7

Delimitações inferiores de Wilber

Nesse capítulo buscaremos entender o comportamento do custo $OPT(X)$ para diferentes sequências de acesso X. Compreenderemos o funcionamento das delimitações propostas por Wilber [13] e relacionaremos essas delimitações com a delimitação dos retângulos independentes do capítulo anterior.

7.1 Visão geral

Wilber foi o pioneiro na área de buscar delimitações inferiores de custo em algoritmos de busca em ABBs que permitem rotações. Durante sua pesquisa, Wilber desenvolveu duas delimitações que ficaram conhecidas como Wilber I e Wilber II. Posteriormente, essas delimitações foram denominadas, respectivamente, delimitação da alternância e delimitação do funil.

Essas delimitações foram propostas se utilizando de nomenclatura, definições e provas bastante complicadas. Nesse trabalho, em vez de mostrar tais provas, mostraremos que essas delimitações podem ser interpretadas como ferramentas para encontrar conjuntos de retângulos independentes. Assim, ficará evidente que a delimitação de retângulos independentes é assintoticamente pelo menos tão boa quanto ambas as delimitações propostas por Wilber.

Inicialmente, será necessário definir uma série de conceitos que serão usados posteriormente. Considere dois conjuntos E e D finitos e disjuntos de pontos. Denotamos por $mix(E, D)$ a string em {**E**,**D**}* obtida pela união $E \cup D$ em ordem crescente, substituindo cada elemento de E por **E** e cada elemento de D por **D**. Para os conjuntos $E = \{2, 3, 6\}$ e $D = \{1, 4, 5\}$, $mix(E, D)$ = **DEEDDE**. Essa string final é uma representação das posições relativas dos pontos em ordem crescente.

Dada uma string $s \in$ {**E**,**D**}*, definimos $blocos(s)$ como o número de blocos contíguos do mesmo símbolo em s. Assim,

$$blocos(s) = \begin{cases} 0, & \text{se s é vazia.} \\ 1 + \#\{i \mid s_i \neq s_{i+1}\}, & \text{caso contrário.} \end{cases}$$

Por exemplo, blocos(**DEEDDE**) = 4.

Para finalizar, definimos $intercala(E, D) = blocos(mix(E, D))$. Assim, compactamos as definições acima para produzir a função *intercala* que associa a dois conjuntos de pontos um inteiro que representa o número de blocos contíguos do mesmo símbolo da string resultante de $mix(E, D)$. Assim, para o exemplo acima $intercala(E, D) = 4$.

7.2 Delimitação da alternância

Seja P um conjunto não-vazio de pontos. Uma *árvore de referência* $\mathcal{T}$ em relação a P é uma árvore binária em que todo nó não-folha possui dois filhos e as folhas são rotuladas com os valores de $P.x$ em ordem crescente. Veja a Figura 7.1. Note que, para conjuntos com mais de dois pontos com x-coordenadas distintas, há múltiplas árvores de referência em relação a P.

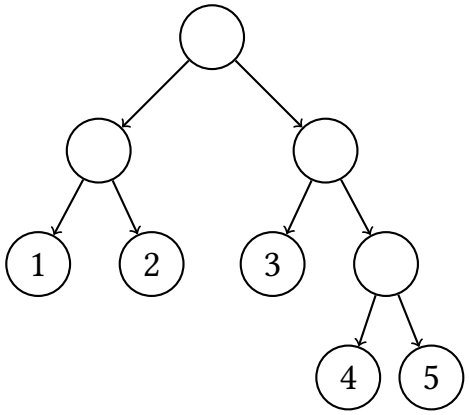


Figura 7.1: *Uma árvore de referência $\mathcal{T}$ para o conjunto P_X de pontos da sequência de acessos $X = (3, 1, 4, 2, 5)$.*

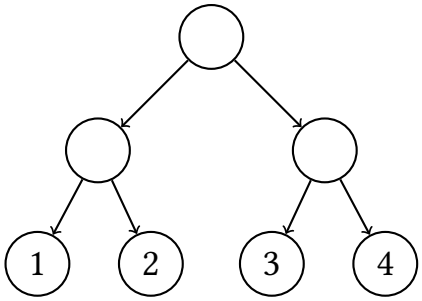
Sejam $\mathcal{T}_E$ e $\mathcal{T}_D$ respectivamente a subárvore esquerda e a subárvore direita da raiz de $\mathcal{T}$, e sejam $P_E = \{p \in P \mid p.x \in \mathcal{T}_E\}$ e $P_D = \{p \in P \mid p.x \in \mathcal{T}_D\}$. Definimos $a(P, \mathcal{T}) = intercala(P_E.y, P_D.y)$, que mostra quanto se intercalam os pontos em P_E e P_D de maneira cronológica. Por fim, definimos a *delimitação da alternância* para um conjunto P de pontos em relação à árvore de referência $\mathcal{T}$ como:

$$\text{Alt}_{\mathcal{T}}(P) = a(P, \mathcal{T}) + \text{Alt}_{\mathcal{T}_E}(P_E) + \text{Alt}_{\mathcal{T}_D}(P_D).$$

Veja um exemplo na Figura 7.2.

Note que a delimitação da alternância de um conjunto de pontos pode ter valores diferentes dependendo da escolha da árvore de referência. Veja a Figura 7.3. Assim, definimos a delimitação da alternância para uma sequência X de acessos como o valor da maior delimitação da alternância para a visão geométrica de X em relação a uma árvore de referência. Formalmente

$$\text{Alt}(X) = \max_{\mathcal{T}} \text{Alt}_{\mathcal{T}}(P_X).$$



Alt	$mix(E, D)$	*intercala*
$a(P, \mathcal{T})$	**DEEDED**	5
$\text{Alt}_{\mathcal{T}_E}(P_E)$	**DED**	3
$\text{Alt}_{\mathcal{T}_D}(P_D)$	**EDE**	3
$\text{Alt}_{\mathcal{T}}(P_X)$		11

Figura 7.2: *Árvore de referência $\mathcal{T}$ e $Alt_{\mathcal{T}}(P_X)$ para a sequência $X = (3, 2, 1, 4, 2, 3)$ de acessos.*

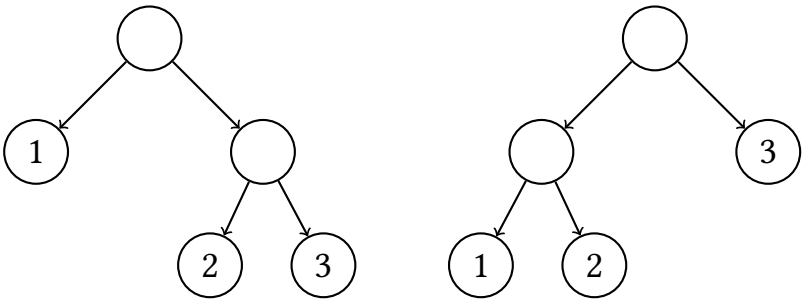


Figura 7.3: *À esquerda $\mathcal{T}_1$ e à direita $\mathcal{T}_2$. Para a sequência $X = (1, 3, 2)$ de acessos, $Alt_{\mathcal{T}_1}(P_X) = 4$, enquanto que $Alt_{\mathcal{T}_2}(P_X) = 5$.*

Para contribuir com a intuição do leitor, a seguir mostraremos uma maneira alternativa e bem mais informal de entender essa delimitação. Podemos pensar nas árvores de referência como árvores binárias de busca com as chaves armazenadas nas folhas e assim é possível contabilizar essa delimitação simulando buscas pelos rótulos das folhas dessa árvore. Para todo nó não-folha de $\mathcal{T}$, definimos como *filho preferido* deste nó o filho mais recentemente visitado numa busca, podendo ser o filho esquerdo, o filho direito ou, se nenhum dos dois filhos tiver sido visitado ainda, o valor nulo.

Dada uma sequência X de acessos, considere uma árvore de referência $\mathcal{T}$ em relação a P_X. Simule o algoritmo tradicional de busca em ABB que não faz rotações para buscar os valores de X em $\mathcal{T}$. Veja a Figura 7.4. Assim, $\text{Alt}_{\mathcal{T}}(X)$ é o somatório do número de vezes que o filho preferido de cada nó intermediário é alterado durante as buscas de X. Analogamente, $\text{Alt}(X) = \max_{\mathcal{T}} \text{Alt}_{\mathcal{T}}(X)$. Então, $\text{Alt}(X)$ é o somatório do número de vezes que o filho preferido de cada nó intermediário é alterado em uma árvore de referência que maximiza esse número.

Infelizmente essa delimitação não é justa se permitirmos repetições na sequência X. Uma maneira fácil de perceber que $OPT(X)$ pode ser muito maior que $\text{Alt}(X)$ é observando como a delimitação se comporta quando X possui repetições do mesmo acesso consecutivamente. Quando simulamos a mesma busca consecutivas vezes em uma árvore de referência, nenhum filho preferido é alterado, então a delimitação não aumenta. Sabemos que todo acesso tem custo maior ou igual a 1 e assim $OPT(X) \geq m$, mas devido a essas repetições, é possível que m seja arbitrariamente maior que $\text{Alt}(X)$. Para entender melhor o argumento acima, se atente às últimas duas simulações de busca na Figura 7.4.

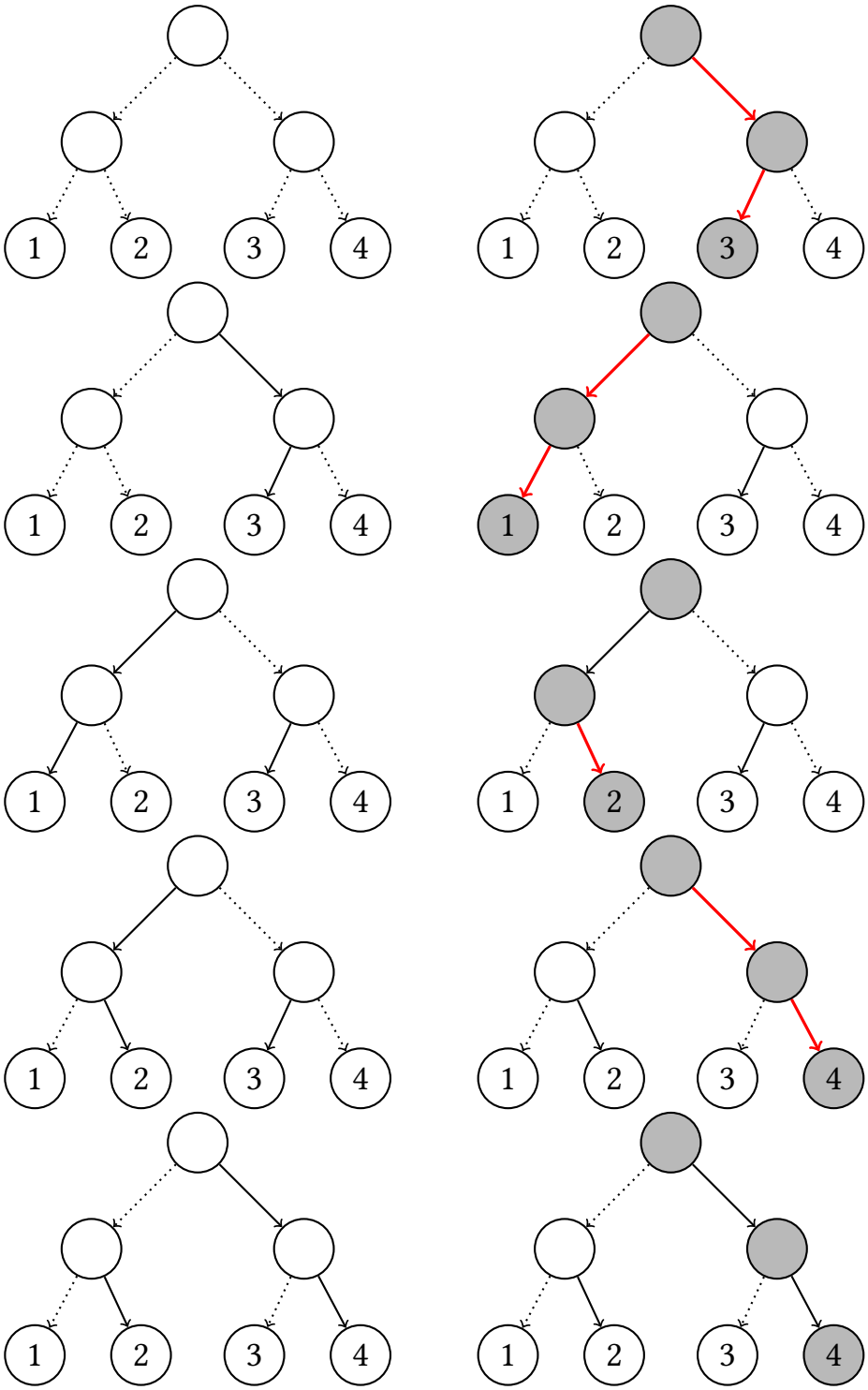


Figura 7.4: *Cálculo de $Alt_{\mathcal{T}}(X)$ para a sequência $X = (3, 1, 2, 4, 4)$ de acessos e a árvore de referência $\mathcal{T}$ ilustrada. À esquerda, as árvores antes da simulação de buscas. À direita, a simulação de cada busca. Os ponteiros que representam filhos preferidos estão por inteiro, enquanto que os ponteiros que representam filhos não-preferidos estão pontilhados. Durante as simulações de busca, estão em vermelho os filhos preferidos que foram alterados durante aquela busca. Note que $Alt_{\mathcal{T}}(X) = 7$, pois esse é o número de filhos preferidos alterados (ponteiros vermelhos) durante as buscas de X.*

7.3 Conjunto independente de retângulos a partir da alternância

Nessa seção mostraremos um algoritmo que encontra um conjunto independente de retângulos para P_X a partir de uma sequência X de acessos e uma árvore de referência. Mostraremos que o conjunto de retângulos produzido pelo algoritmo tem pelo menos $\text{Alt}(P_X) - (n-1)$ retângulos, onde n é o número de folhas na árvore de referência, ou seja, o número de chaves consideradas. Na prática, o algoritmo encontrará pares de pontos de P_X que representam retângulos.

Para melhorar o entendimento do leitor, aconselhamos acompanhar a descrição do algoritmo utilizando a Figura 7.5. Considere a visão geométrica P_X de uma sequência X de acessos em um plano 2D e uma árvore de referência $\mathcal{T}$ em relação a P_X com n folhas. Denotemos por I o conjunto de pares de pontos que o algoritmo retornará ao fim da execução. Para um nó x de $\mathcal{T}$, denotamos por $u(x)$ o maior rótulo de um nó na subárvore esquerda de x e por $v(x)$ o menor rótulo de um nó na subárvore direita de x. O algoritmo inicializa I como o conjunto vazio e verifica o nó r, raiz de $\mathcal{T}$, da seguinte forma: considere a reta vertical l no plano 2D com x-coordenada igual a $\frac{u(r)+v(r)}{2}$, adicione a I todos os pares $(a,b) \in P_X$ de pontos consecutivos, onde a está à esquerda de l e b está à direita de l. Em seguida, o algoritmo analisa recursivamente o nó filho esquerdo de r em $\mathcal{T}$ porém considerando apenas os pontos de P_X à esquerda de l e analisa o nó filho direito de r em $\mathcal{T}$ porém considerando apenas os pontos de P_X à direita de l.

Mostraremos que, para dois nós w e z de $\mathcal{T}$, o conjunto de pares de pontos encontrado durante a análise do algoritmo para o nó w é disjunto do conjunto de pares de pontos encontrado durante a análise do algoritmo para o nó z. Suponha que w e z não são descendentes um do outro em $\mathcal{T}$, então o subconjunto de P_X considerado durante a análise do algoritmo para cada um desses nós é disjunto, então o algoritmo não encontra nenhum par de pontos em comum. Suponha agora que w é descendente de z, então durante a análise do algoritmo para o nó z é encontrado pares de pontos, onde um está à esquerda de $\frac{u(z)+v(z)}{2}$ e outro está à direita. Porém, por w ser descendente de z, então o subconjunto de P_X considerado durante a análise de w possui apenas pontos de um lado da reta com x-coordenada $\frac{u(z)+v(z)}{2}$, logo o conjunto de pares de pontos encontrado pelo algoritmo para cada um desses nós é disjunto entre si. Como o conjunto de pares de pontos encontrado pelo algoritmo para dois nós diferentes é disjunto entre si, então o tamanho do conjunto R ao final da execução do algoritmo é igual a somatória do número de pares de pontos adicionados a R para cada um dos nós de $\mathcal{T}$.

Perceba que todo par (a,b) de pontos encontrado pelo algoritmo durante a análise do nó w de $\mathcal{T}$ representa uma alteração do filho preferido de w, uma vez que a e b estão em lados opostos da reta vertical com x-coordenada $\frac{u(w)+v(w)}{2}$ e são consecutivos dentro do subconjunto de P_X considerado. Nota-se que a alteração de filho preferido de valor nulo para outro valor é representada pelo ponto com menor y-coordenada dentro do subconjunto de P_X considerado, ou seja, esta alteração não é representada por nenhum retângulo de R. Assim, nota-se que o tamanho de R ao final da execução do algoritmo é, no pior caso, $\text{Alt}(P_X) - (n-1)$, pois há $n-1$ nós intermediários em $\mathcal{T}$. Se considerarmos $\text{Alt}(P_X) = \Omega(n)$, então o algoritmo encontra $\text{O}(\text{Alt}(P_X))$ retângulos e essa perda é assintoticamente

desprezível.

Por fim, mostraremos que o conjunto R ao final da execução do algoritmo representa um conjunto independente de retângulos. Assuma por contradição que R não representa um conjunto independente de retângulos, ou seja, existe um par de pontos $(a, b) \in R$ encontrado durante a análise do vértice i e outro par de pontos $(c, d) \in R$ encontrado durante a análise do vértice j tal que o vértice do retângulo representado por um desses pares de pontos está estritamente dentro da área delimitada pelo retângulos representado pelo outro par de pontos. É facil notar que se $i = j$ ou se i e j não forem descendentes entre si em $\mathcal{T}$, então é impossível (a, b) e (c, d) serem pares de pontos que representem retângulos dependentes em P_X. Assim, assuma sem perda de generalidade que j é descendente de i em $\mathcal{T}$. Se um vértice do $\{c, d\}$-retângulo estiver dentro da área do $\{a, b\}$-retângulo, então nota-se que existe um ponto entre c e d que possui x e y-coordenadas intermediárias em relação ao par (a, b), ou seja, a e b não são pontos consecutivos dentro do subconjunto de P_X considerado durante a análise do nó i que é uma contradição. Se um vértice do $\{a, b\}$-retângulo estiver dentro da área do $\{c, d\}$-retângulo e não existir nenhum vértice do $\{c, d\}$-retângulo dentro da área do $\{a, b\}$-retângulo, então nota-se que existe um ponto entre a e b que possui x e y-coordenadas intermediárias em relação ao par (c, d), ou seja, c e d não são pontos consecutivos dentro do subconjunto de P_X considerado durante a análise do nó j que é uma contradição.

7.4 Delimitação do funil

Seja P um conjunto de pontos. Definimos o *lado esquerdo do funil* de um ponto p de P como o conjunto de pontos de P que possuem as seguintes propriedades: estão à esquerda de p, abaixo de p e o retângulo ortogonal que possui p e algum ponto do funil esquerdo de p como vértices não possui nenhum outro ponto de P. Formalmente,

$$F_E(P, p) = \{q \in P \mid q.y < p.y \text{ e } q.x < p.x \text{ e } \{p, q\}\text{-retângulo é arboreamente insatisfeito}\}$$

De maneira simétrica definimos o *lado direito do funil* de p,

$$F_D(P, p) = \{q \in P \mid q.y > p.y \text{ e } q.x < p.x \text{ e } \{p, q\}\text{-retângulo é arboreamente insatisfeito}\}$$

Por fim, definimos o *funil* de p como a união do lado esquerdo do funil e do lado direito do funil de p, ou seja, $F_E(P, p) \cup F_D(P, p)$. Veja a Figura 7.6.

É importante notar que cronologicamente os pontos do lado esquerdo do funil de qualquer ponto p possuem x-coordenadas decrescentes. De maneira análoga, cronologicamente os pontos do lado direito do funil de qualquer ponto p possuem x-coordenadas crescentes. Isso acontece por conta da propriedade de insatisfação arbórea do ponto p com os pontos de seu funil.

A delimitação do funil para um ponto conta o número de intercalações entre o lado esquerdo e direito do funil. Assim, para cada ponto p de P, definimos $f(P, p) = \mathit{intercala}(F_E(P, p).y, F_D(P, p).y)$. A *delimitação do funil* para uma sequência X de acessos é a somatória desse valor para todos os pontos da visão geométrica de X, ou seja, $\text{Funil}(X) = \sum_{p \in P_X} f(P_X, p)$. Veja a Figura 7.7.

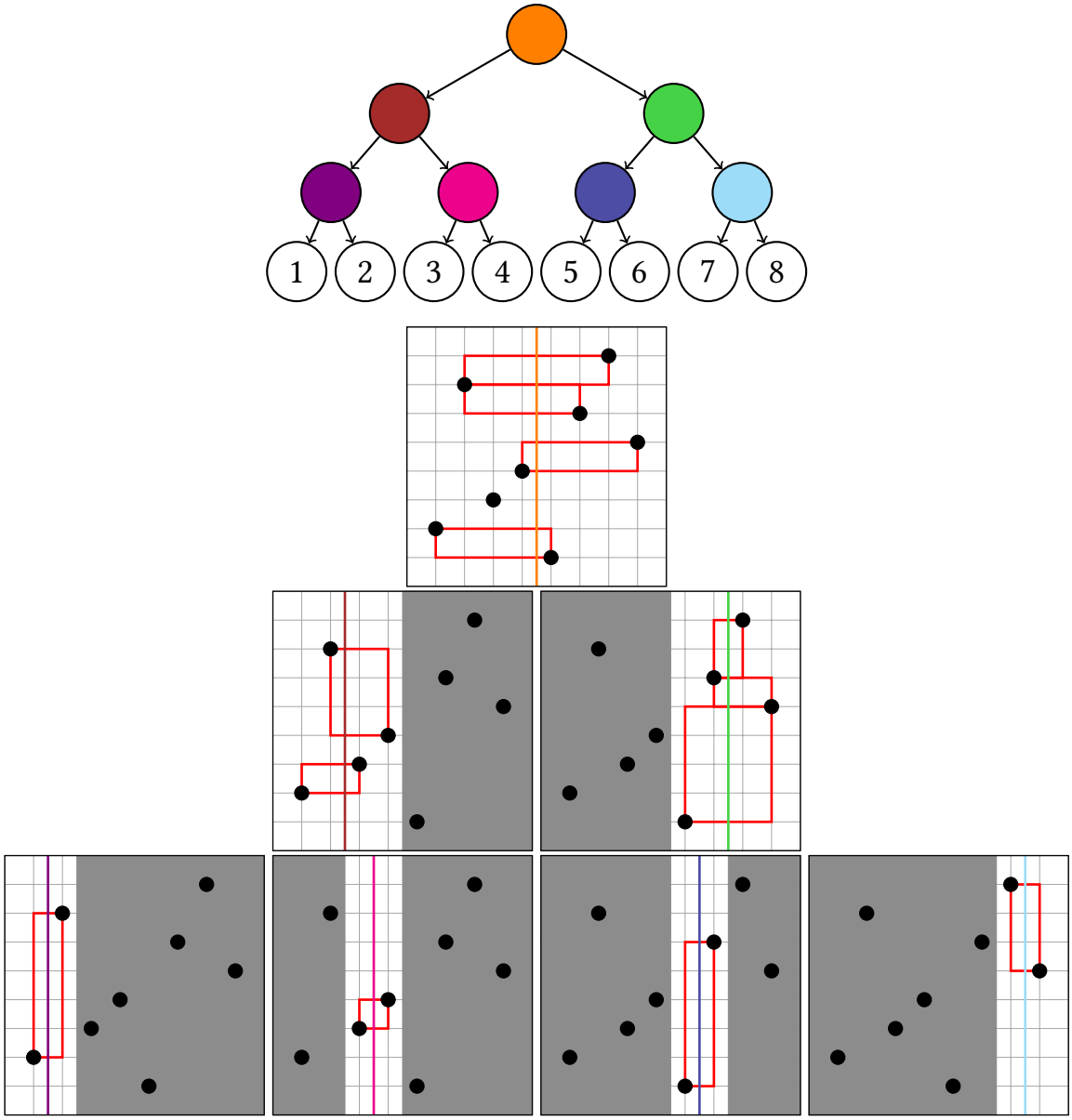


Figura 7.5: *Em cima, uma árvore de referência em relação à visão geométrica da sequência de acessos* $X = (5, 1, 3, 4, 8, 6, 2, 7)$. *Cada nó intermediário possui uma cor diferente. Abaixo, está descrita a execução do algoritmo recursivo que encontra os* $Alt_{\mathcal{T}}(P_X) - (n-1)$ *retângulos. Cada linha vertical possui a cor do nó intermediário correspondente.*

Grosseiramente, $f(P, p)$ é o número de intercalações direita-esquerda dos pontos de P abaixo de p, indo de cima para baixo, cujas x-coordenadas se aproximam da x-coordenada de p.

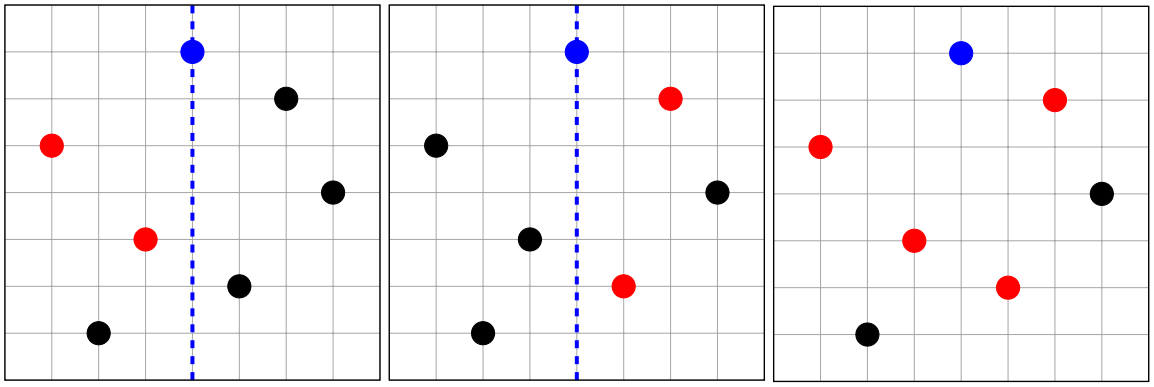


Figura 7.6: *O ponto (4,7) está destacado. À esquerda, os pontos vermelhos pertencem ao funil esquerdo do ponto azul. Ao meio, os pontos vermelhos pertencem ao funil direito do ponto azul. À direita, estão destacados de vermelho os pontos do funil do ponto azul, que são a união dos dois conjuntos anteriores.*

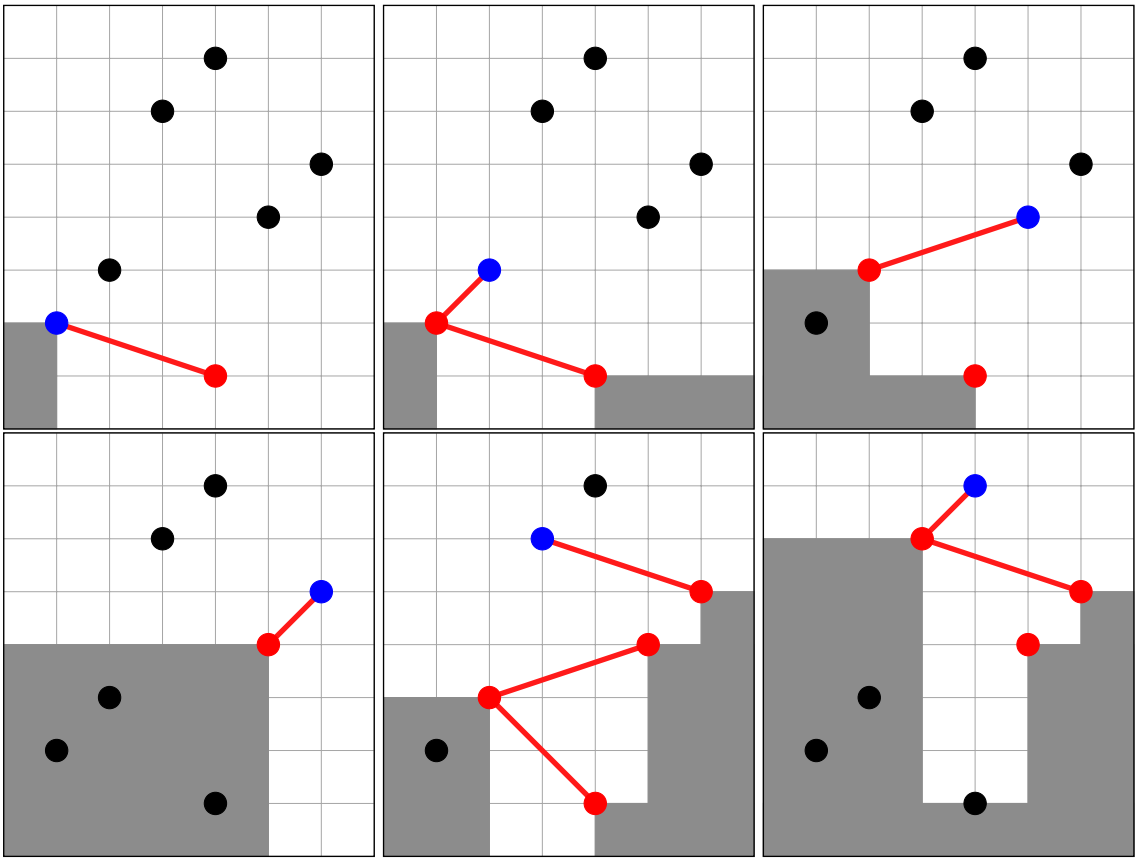


Figura 7.7: *Para a sequência $X = (4, 1, 2, 5, 6, 3, 4)$ de acessos, o cálculo de $f(P_X, p)$ para todo $p \in P_X$ é o número de linhas vermelhas. Cada imagem representa o cálculo de $f(P_X, p)$ para o ponto azul p. Os pontos vermelhos são os pontos pertencentes ao funil de p e cada linha vermelha representa uma intercalação dos funis esquerdo-direto desse ponto. Para essa sequência de acessos, $Funil(X) = 10$.*

7.5 Conjunto independente de retângulos a partir do funil

É possível encontrar um conjunto independente de retângulos associado a esta delimitação a partir do seguinte algoritmo: para cada ponto $p \in P_X$, trace um retângulo com vértices p e q para todo $q \in P_X$ associado à última ocorrência da string E ou D de cada bloco contíguo de $mix(F_E(P,p).y, F_D(P,p).y)$. Em outras palavaras, trace um retângulo com vértices em p e q para todo ponto q que for o ponto mais alto do funil esquerdo ou do funil direito antes de uma intercalação.

Note que todos os pares de ponto (p,q) encontrados possuem o ponto p como o ponto analisado e q como pontos abaixo de p. Assim, por construção, todos os retângulos encontrados são distintos. Além disso, por definição, todos os pontos q pertencentes ao funil do ponto p possuem o $\{p,q\}$-retângulo arboreamente insatisfeito, logo o conjunto de retângulos encontrado é independente.

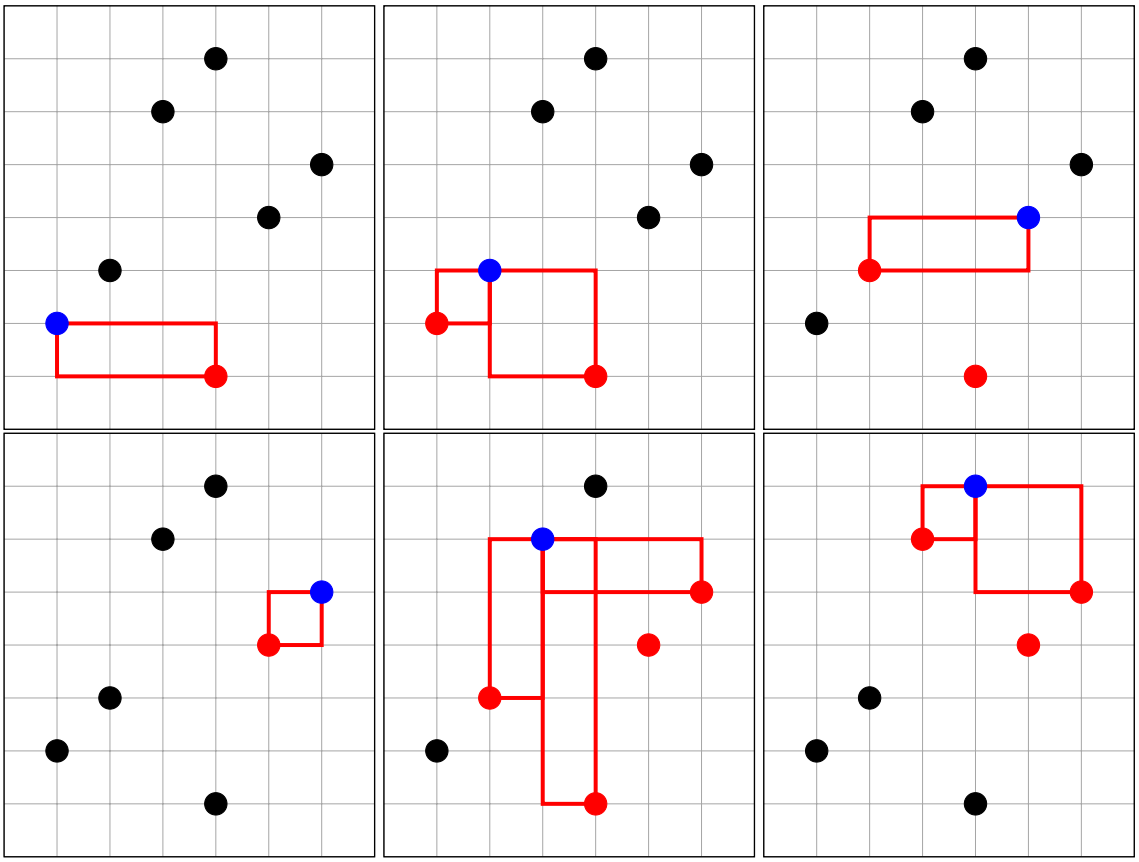


Figura 7.8: *Para a sequência de acessos $X = (4,1,2,5,6,3,4)$. Para cada ponto azul está destacado o conjunto independente de retângulos que será encontrado pelo algoritmo referente à delimitação do funil para aquele ponto. Perceba que nem todos os pontos do funil do ponto analisado são incluídos. Comparando com a Figura 7.7, é facil notar que o tamanho do conjunto de retângulos final é exatamente Funil(X).*

7.6 Sequência bit-reversa

Para evidenciar como as delimitações propostas por Wilber podem ser uma ferramenta poderosa para analisar sequências de acessos, mostraremos uma sequência que é sempre custosa, independente do algoritmo de busca.

Nesta seção, excepcionalmente, consideraremos que as ABBs contêm as chaves de 0 a $n-1$. Essa consideração será útil para maior simplificação das análises, porém não é necessária para o estudo abaixo.

Definimos a *sequência bit-reversa* de 0 a $n-1$ como a contagem de 0 a $n-1$ em binário e depois a sua leitura de trás pra frente, ou seja, leia os bits de cada número da direita para a esquerda. Veja a Figura 7.9.

$$
\begin{aligned}
0 = 000 &\longrightarrow 000 = 0\\
1 = 001 &\longrightarrow 100 = 4\\
2 = 010 &\longrightarrow 010 = 2\\
3 = 011 &\longrightarrow 110 = 6\\
4 = 100 &\longrightarrow 001 = 1\\
5 = 101 &\longrightarrow 101 = 5\\
6 = 110 &\longrightarrow 011 = 3\\
7 = 111 &\longrightarrow 111 = 7
\end{aligned}
$$

Figura 7.9: *Sequência bit-reversa para $n = 8$. A coluna da esquerda mostra a contagem de 0 a $n-1$ em binário e a coluna da direita mostra a leitura desses números em base 2 de trás para frente, ou seja, considerando os bits da direita para a esquerda.*

É bastante útil entender como um algoritmo de busca se comporta nas sequências de acessos em uma ABB completa fixa. Uma maneira de visualizar as buscas em binário dentro de uma ABB completa fixa é imaginar que os ponteiros representam um bit e a busca em binário dentro da ABB é a sequência de ponteiros, que com seus bits concatenados, resultam no número buscado. Veja a Figura 7.10.

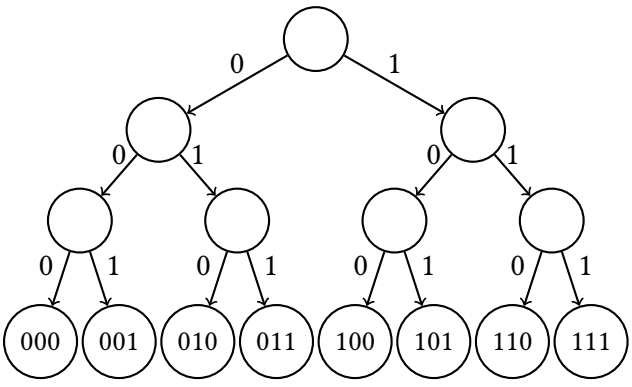


Figura 7.10: *ABB completa de 8 folhas com uma representação binária.*

Como estamos lendo os bits de trás para frente, sabemos que a característica do contador binário se mantém porém com os bits ao contrário. Assim, sabemos que essa sequência tem a propriedade que o maior bit se altera em todo incremento do número, o segundo

maior bit se altera em metade dos incrementos, o terceiro maior bit se altera em um quarto dos incrementos e o i-ésimo bit se altera em 2^{i-1} dos incrementos.

Voltando ao contexto de ABBs, as alterações dos bits descritas acima implicam que cada busca começa indo por uma subárvore diferente, tanto na raiz como nos nós intermediários. Assim, como sabemos que as alternâncias começam pela esquerda por esta ser a ordem na contagem, deduzimos que para qualquer nó intermediário, a j-ésima busca em um de seus descendentes entrará na sua subárvore esquerda se j for ímpar e na sua subárvore direita se j for par.

A princípio essa informação não parece tão poderosa, porém relembremos como funciona a delimitação da alternância. Para qualquer ABB fixa com as chaves nas folhas, a somatória do número de intercalações esquerda-direita de todos os nós intermediários dessa ABB é uma delimitação inferior no custo de executar essa sequência de acessos dentro do modelo de computação adotado. Assim, como notado anteriormente, nunca há duas buscas consecutivas na mesma subárvore: todas as buscas se alternam em todos os nós. Portanto, sabemos que nenhum ponteiro de filho preferido será percorrido durante as buscas dessa sequência. Assim, essa sequência maximiza a delimitação da alternância.

Na ABB completa com n folhas, cada folha tem altura $\lg n$. Então, sabemos que cada simulação de busca na ABB completa fixa acarretará em $\lg n$ alternâncias/alterações em ponteiros de filho preferido, e assim para uma sequência bit-reversa X de acessos com m buscas, $\text{Alt}_{\mathcal{T}}(X) = m \cdot \lg n$ para a árvore completa $\mathcal{T}$ com n folhas. Assim, sabemos que $OPT(X) \geq m \cdot \lg n$.

Esse resultado é particularmente especial, pois nos mostra que a melhor maneira de executar todos os acessos dessa sequência vai ter custo maior ou igual a $m \cdot \lg n$, ou em média, $\lg n$ por busca. Essa é a delimitação trivial de árvores binárias de busca balanceadas como a árvore rubro-negra ou a AVL. Em outras palavras, essa sequência é tão cara no modelo de computação que não existe nenhuma maneira de melhorar seu custo acima da delimitação trivial, e assim, qualquer ABB balanceada com custo $O(\lg n)$ por busca tem custo assintoticamente proporcional ao ótimo nessas sequências.

Por fim, vale a pena relembrar que essa sequência também é importante para as delimitações da eficiência do algoritmo guloso futurista. Como citado anteriormente, o pior caso conhecido do guloso futurista é buscar a sequência bit-reversa uma vez e depois seguir a sequência apenas buscando as folhas da ABB.

Note que esse caso é ruim, pois os nós intermediários se mantém no caminho das folhas e não permitem que o algoritmo traga as folhas para perto da raiz. Assim todas as buscas visitam uma série de nós desnecessários e acarretam em um custo de $OPT(X) + O(m)$. Veja a Figura 7.11.

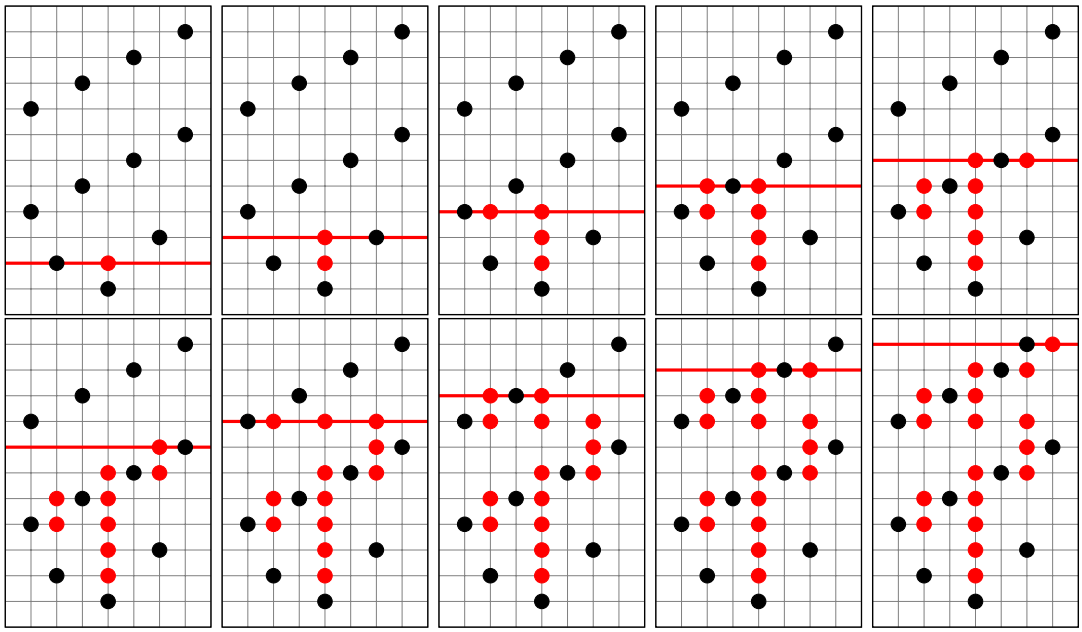


Figura 7.11: *Execução do algoritmo guloso futurista para a sequência* $X = (4, 2, 1, 3, 5, 7, 1, 3, 5, 7)$ *de acessos. As chaves* $4, 2, 1$ *representam as chaves dos nós intermediários e o restante são as folhas. Basta desenhar uma árvore completa e perceber o comportamento ineficiente.*

Capítulo 8

Relação entre as delimitações de Wilber

Nesta capítulo estabeleceremos uma relação entre a delimitação da alternância e a delimitação do funil. Faremos isso por meio de um argumento geométrico a partir de uma caracterização nova de z-retângulos proposta por Lecomte e Weistein [6].

8.1 Rotação e inversão de conjuntos de pontos

Pouco se sabe sobre a relação entre a delimitação da alternância e do funil para sequências de acessos sem restrições. Assim, neste capítulo consideraremos apenas sequências de acessos sem repetições.

Seja P um conjunto de pontos. Definimos a inversão temporal de um ponto $p \in P$ como $\overline{p} = (p.x, -p.y)$. Definimos a inversão temporal de um conjunto de pontos P como $\overline{P} = \{\overline{p} \mid p \in P\}$. Na prática, a inversão temporal de um conjunto de pontos é o conjunto resultante do espelhamento de cada um dos seus pontos em relação ao eixo das abscissas. Veja a Figura 8.1.

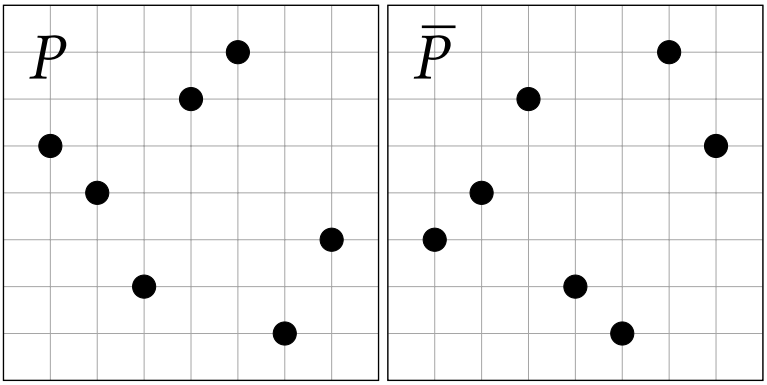


Figura 8.1: *À esquerda, um conjunto P de pontos. À direita, a sua inversão temporal $\overline{P}$.*

Analogamente, definimos a rotação 90°(em sentido anti-horário) de um ponto $p \in P$

como $p^{\perp} = (-p.y, p.x)$. A rotação 90° em sentido anti-horário de um conjunto de pontos P é definida por $P^{\perp} = \{p^{\perp} \mid p \in P\}$. Veja a figura 8.2.

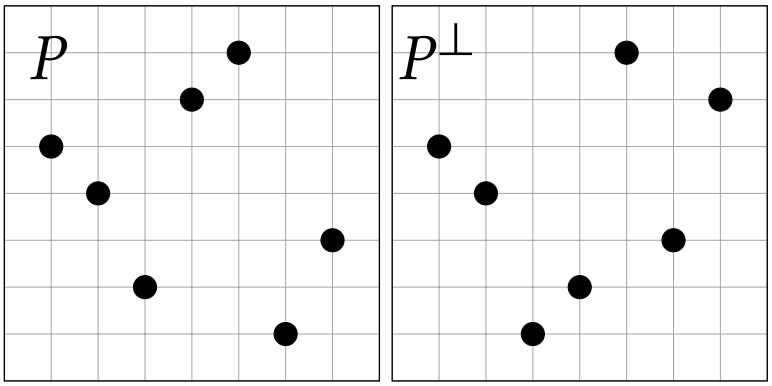


Figura 8.2: *À esquerda, o conjunto de pontos P_X relativo a sequência de acessos $X = (6, 3, 5, 4, 2, 1, 7)$. À direita, a inversão temporal de P_X.*

8.2 Relacionando a alternância com o funil e sua inversão temporal

Primeiramente, é necessário entender como a inversão temporal de um ponto influencia a sua delimitação do funil.

Lema 8.1. *Seja P_X a visão geométrica de uma sequência de acessos sem repetições e $\mathcal{T}$ uma árvore de referência em relação à P_X. Funil(P_X)+ Funil$(\overline{P_X}) \geq Alt_{\mathcal{T}}(P_X)$.*

Demonstração. Faremos uma prova por indução em k que, toda árvore $\mathcal{T}$ de k nós satisfaz a inequação do Lema.

Para $k = 1$, a árvore de referência $\mathcal{T}$ possui um único nó. Assim, $\text{Alt}_{\mathcal{T}}(P_X) = 0$ e a inequação é valida.

Suponha agora que $k > 1$ e que a propriedade vale para toda árvore com menos que k nós. Seja $\mathcal{T}_L$ e $\mathcal{T}_R$ respectivamente a subárvore esquerda e a subárvore direita da raiz de $\mathcal{T}$. Seja $P_L = \{p \in P_X \mid p.x \in \mathcal{T}_L\}$ e $P_R = \{p \in P_X \mid p.x \in \mathcal{T}_R\}$. Pela hipótese de indução, temos

$$\text{Funil}(P_E) + \text{Funil}(\overline{P_E}) \geq \text{Alt}_{\mathcal{T}_E}(P_E),$$
$$\text{Funil}(P_D) + \text{Funil}(\overline{P_D}) \geq \text{Alt}_{\mathcal{T}_D}(P_D).$$

Assim, temos

$$\begin{aligned} \text{Alt}_{\mathcal{T}}(P_X) &= a(P_X, \mathcal{T}) + \text{Alt}_{\mathcal{T}_E}(P_X) + \text{Alt}_{\mathcal{T}_D}(P_X), \\ &\leq a(P_X, \mathcal{T}) + \text{Funil}(P_E) + \text{Funil}(\overline{P_E}) + \text{Funil}(P_D) + \text{Funil}(\overline{P_D}). \end{aligned} \tag{8.1}$$

Agora, é necessário entender como a delimitação do funil é afetada pela inversão temporal. Seja p um ponto de P_X. Mostraremos que se $p \in P_E$, então $f(P_X, p) \geq f(P_E, p)$

e $f(\overline{P_X}, \overline{p}) \geq f(\overline{P_E}, \overline{p})$. De maneira simétrica, se $p \in P_D$ então $f(P_X, p) \geq f(P_D, p)$ e $f(\overline{P_X}, \overline{p}) \geq f(\overline{P_D}, \overline{p})$.

Suponha que $p \in P_E$. Como p pertence à P_E, então todos os pontos à esquerda de p também pertencem à P_E, ou seja, $F_E(P_E, p) = F_E(P_X, p)$. Além disso, todos os pontos que pertencem à P_E e estão à direita de p pertencem a tanto o funil direito de p em relação à P_E quanto em relação à P_X, ou seja, $F_D(P_E, p) \subseteq F_D(P_X, p)$. Veja a Figura 8.3. Assim, $mix(F_E(P_E, p).y, F_D(P_E, p).y)$ é uma subsequência de $mix(F_E(P_X, p).y, F_D(P_X, p).y)$, logo

$$blocos(mix(F_E(P_E, p), F_D(P_E, p))) \leq blocos(mix(F_E(P_X, p), F_D(P_X, p))),$$

$$f(P_E, p) \leq f(P_X, p).$$

O argumento simetricamente vale para os outros casos.

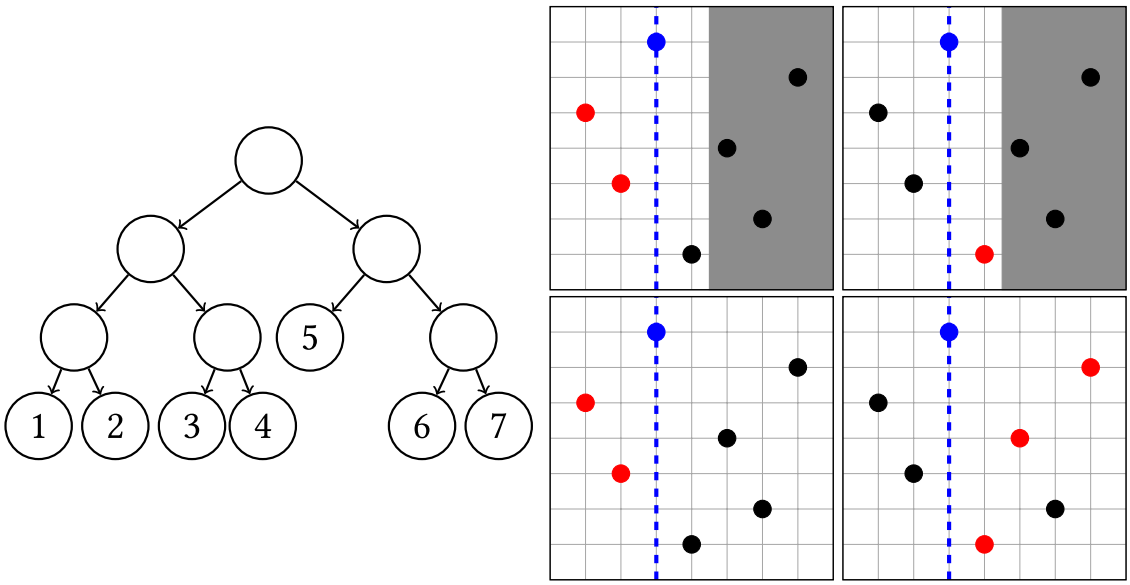


Figura 8.3: *À esquerda, uma árvore de referência $\mathcal{T}$ em relação à visão geométrica da sequência de acessos $X = (4, 6, 2, 5, 1, 7, 3)$. Note que $P_E = \{1, 2, 3, 4\}$. À direita, 4 conjuntos de pontos. Denotemos por p o ponto $(3, 7)$ destacado de azul. Os conjuntos de pontos vermelhos de cima representam respectivamente, $F_E(P_E, p)$ e $F_D(P_E, p)$. Os conjuntos de pontos vermelhos embaixo representam respectivamente, $F_E(P_X, p)$ e $F_D(P_X, p)$. Perceba que $F_E(P_E, p) = F_E(P_X, p)$ e $F_D(P_E, p) \subseteq F_D(P_X, p)$.*

Ao somarmos $f(P_X, p)$ para todo $p \in P_X$, temos

$$\begin{aligned} \text{Funil}(P_X) &= \sum_{p \in P_X} f(P_X, p), \\ &\geq \sum_{p \in P_E} f(P_E, p) + \sum_{p \in P_D} f(P_D, p), \\ &= \text{Funil}(P_E) + \text{Funil}(P_D). \end{aligned} \tag{8.2}$$

Analogamente para a inversão temporal, temos

$$\begin{aligned}\text{Funil}(\overline{P_X}) &= \sum_{p \in P_X} f(\overline{P_X}, \overline{p}), \\ &\geq \sum_{p \in P_E} f(\overline{P_E}, \overline{p}) + \sum_{p \in P_D} f(\overline{P_D}, \overline{p}), \\ &= \text{Funil}(\overline{P_E}) + \text{Funil}(\overline{P_D}). \end{aligned} \tag{8.3}$$

Combinando as Equações 8.1, 8.2 e 8.3, concluímos

$$\begin{aligned}\text{Funil}(P) + \text{Funil}(\overline{P}) &\geq \text{Funil}(P_E) + \text{Funil}(P_D) + \text{Funil}(\overline{P_E}) + \text{Funil}(\overline{P_D}), \\ &\geq \text{Alt}_{\mathcal{T}}(P_X) - a(P_X, \mathcal{T}). \end{aligned} \tag{8.4}$$

Para terminar a indução, basta retirar o termo $a(P_X, \mathcal{T})$ da Equação 8.4. Para isso, precisamos entender como a intercalação é afetada quando consideramos P_X e apenas um dos conjuntos P_E, P_D, $\overline{P_E}$ e $\overline{P_D}$.

Vamos definir quatro propriedades que serão utilizadas a seguir:

(a) $p \in \mathcal{T}_E$ e $f(P_X, p) \geq f(P_E, p) + 1$,

(b) $p \in \mathcal{T}_E$ e $f(\overline{P_X}, \overline{p}) \geq f(\overline{P_E}, \overline{p}) + 1$,

(c) $p \in \mathcal{T}_D$ e $f(P_X, p) \geq f(P_D, p) + 1$,

(d) $p \in \mathcal{T}_D$ e $f(\overline{P_X}, \overline{p}) \geq f(\overline{P_D}, \overline{p}) + 1$.

Vamos enumerar os pontos de P_X de maneira cronológica com y-coordenada crescente como $p_1, \ldots, p_m$ e dividiremos a análise em 6 casos. Como $a(P_X, \mathcal{T}) = intercala(P_E.y, P_D.y)$ e P_E e P_D não são vazios, então $a(P_X, \mathcal{T}) \geq 2$. Assim, quando analisamos cronologicamente os pontos de P_X, há $a(P_X, \mathcal{T}) - 1 \geq 1$ alternações entre pontos de P_E e pontos de P_D. Logo, há exatamente $a(P_X, \mathcal{T}) - 2$ pares de pontos consecutivos em relação ao seu funil orientado (funil esquerdo ou direito) que possuem diferença no eixo y de mais de 1 unidade. Formalmente, há exatamente $a(P_X, \mathcal{T}) - 2$ pares de índices (i, j) tal que $i + 1 < j$ que

$$\begin{aligned}&\text{caso 1: } p_i, p_j \in P_E \text{ e } p_{i+1}, \ldots, p_{j-1} \in P_D, \\ &\text{caso 2: } p_i, p_j \in P_D \text{ e } p_{i+1}, \ldots, p_{j-1} \in P_E.\end{aligned}$$

Além disso, há um índice $i^* > 1$ que representa o primeiro ponto do lado que começa a aparecer depois, ou seja,

$$\begin{aligned}&\text{caso 3: } p_{i^*} \in P_E \text{ e } p_1, \ldots, p_{i^*-1} \in P_D, \\ &\text{caso 4: } p_{i^*} \in P_D \text{ e } p_1, \ldots, p_{i^*-1} \in P_E.\end{aligned}$$

E por último, há um índice $j^* < m$ que representa o último ponto do lado que termina primeiro, ou seja,

$$\begin{aligned}&\text{caso 5: } p_{j^*} \in P_E \text{ e } p_{j^*+1}, \ldots, p_m \in P_D, \\ &\text{caso 6: } p_{j^*} \in P_D \text{ e } p_{j^*+1}, \ldots, p_m \in P_E.\end{aligned}$$

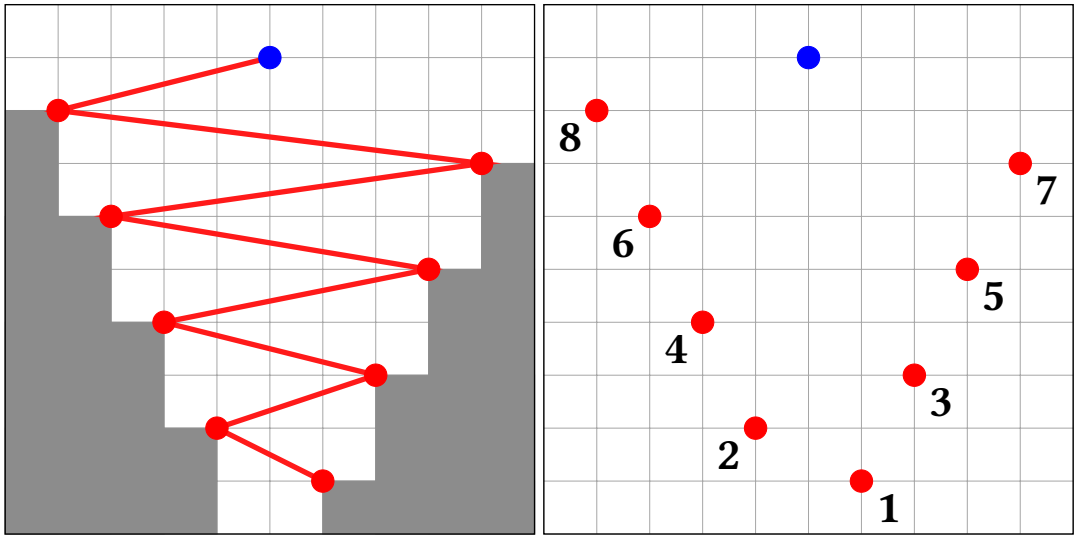


Figura 8.4: *À esquerda, está destacada a delimitação do funil para o ponto* $(5, 9)$ *da visão geométrica da sequência de acesso* $X = (6, 4, 7, 3, 8, 2, 9, 1, 5)$ *e, à direita, estão numerados de* 1 *a* 8 *os pontos desse funil. Perceba que os pares de índices* $(2, 4), (4, 6), (6, 8)$ *representam o caso 1. Os pares de índices* $(1, 3), (3, 5), (5, 7)$ *representam o caso 2. Além disso, o caso 3 acontece com índice* $i^* = 2$ *e também o caso 6 com índice* $j^* = 7$.

Assim, percebemos que os caso 3 e 4 são complementares assim como os casos 5 e 6. Logo, sempre acontece exatamente 1 caso de cada um desses pares. Como os primeiros dois casos acontecem $a(P_X, \mathcal{T}) - 2$ vezes, então somados temos $a(P_X, \mathcal{T})$ ocorrência de todos os 6 casos.

Cada caso desses implica em uma propriedade descrita anteriormente. A descrição exata é

caso 1 implica que p_j possui a propriedade (a) ou p_i possui a propriedade (b),
caso 2 implica que p_j possui a propriedade (c) ou p_i possui a propriedade (d),
caso 3 implica que p_{i^*} possui a propriedade (a),
caso 4 implica que p_{i^*} possui a propriedade (c),
caso 5 implica que p_{j^*} possui a propriedade (b),
caso 6 implica que p_{j^*} possui a propriedade (d).

Provaremos para o caso 1 e para o caso 3. O restante dos casos são análogos.

Para o caso 1, se $p_i.x < p_j.x$, então p_i está no funil esquerdo de p_j tanto em relação à P_X quanto em relação à P_E. Mas, em relação à P_X, p_{j-1} é um ponto adicional ao funil esquerdo quando comparado com o funil esquerdo em relação à P_E. Assim, $f(P_X, p_j) \geq f(P_E, p_j) + 1$. Se $p_i.x > p_j.x$, é possível usar o mesmo argumento para $\overline{P_X}$ e $\overline{P_E}$ analisando o ponto p_i. Assim, $f(\overline{P_X}, \overline{p_i}) \geq f(\overline{P_E}, \overline{p_i}) + 1$.

Para provar o caso 3, basta perceber que ambos os funis de p_{i^*} em relação à P_E são vazios por não existirem pontos abaixo de p_{i^*} que pertença à P_E. Já em relação à P_X, o funil direito de p_{i^*} possui pelo menos o ponto p_{i^*-1}. Logo, $f(P, p_{i^*}) \geq 1 = f(P_E, p_{i^*}) + 1$, já que $f(P_E, p_{i^*}) = 0$.

Assim, é possível recombinar as Equações 8.1, 8.2 e 8.3, se utilizando deste fato para concluir

$$\begin{aligned}\text{Funil}(P) + \text{Funil}(\overline{P}) &\geq \text{Funil}(P_E) + \text{Funil}(P_D) + \text{Funil}(\overline{P_E}) + \text{Funil}(\overline{P_D}) + a(P_X, \mathcal{T}),\\ &\geq \text{Alt}_{\mathcal{T}}(P_X).\end{aligned}$$

Assim, terminamos a prova indutiva. □

Apesar do Lema 8.1 relacionar as duas delimitações com a ajuda da inversão temporal de P_X, não é de grande utilidade isolada, pois não sabemos como Funil(P) e Funil($\overline{P}$) se relacionam.

8.3 Relacionando o funil e sua inversão temporal

Precisamos entender agora como transformações geométricas afetam a delimitação do funil. A delimitação não é afetada por espelhamentos no eixo Y. Isso acontece pois o número de intercalações entre os funis se mantém igual e assim a função *intercala* para cada ponto não tem seu valor alterado. Veja a Figura 8.5.

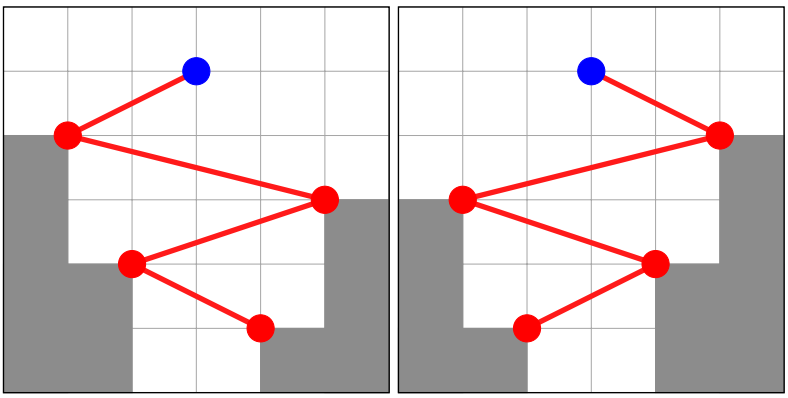


Figura 8.5: *À esquerda, está destacado o funil do ponto azul. À direita, está destacado o funil do mesmo ponto no conjunto inicial espelhado em relação ao eixo Y. Perceba que em ambos os casos há o mesmo número de intercalações entre os funis esquerdo e direito do ponto azul.*

Porém o valor da delimitação pode alterar com a inversão temporal. Veja a Figura 8.6.

Assim, a princípio Funil(P) e Funil($\overline{P}$) podem ter valores muito diferentes. Para entender essa relação precisamos caracterizar a delimitação do funil de maneira invariante em relação a rotações de 90°.

Para um conjunto de pontos P, chamamos de *z-retângulo* qualquer conjunto de 4 pontos $(p, q, r, s) \in P^4$ com as seguintes propriedades: $q.x < p.x < r.x < s.x$, $r.y < q.y < s.y < p.y$ e $P \cap [q.x, s.x] \times [r.y, p.y]$. Na prática, z-retângulo é um conjunto de 4 pontos de P que cronologicamente possuem x-coordenada relativa 3,1,4,2 e o retângulo que contém todos os 4 pontos em seu perímetro não possui nenhum outro ponto de P em sua área. Veja a Figura 8.7. Definimos por *zRet(P)* o número de z-retângulos de P.

Lema 8.2. *Para qualquer conjunto P de pontos, $zRet(P) = zRet(P^{\perp})$.*

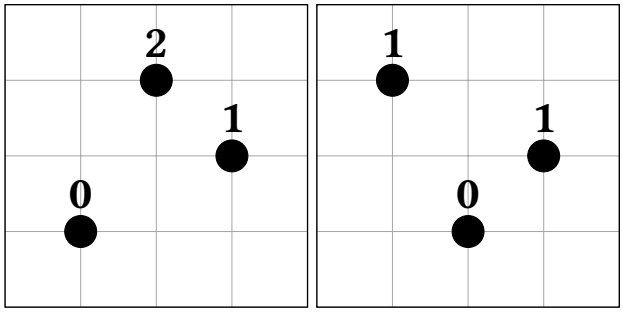


Figura 8.6: *À esquerda, a visão geométrica da sequência de acessos* $X = (1, 3, 2)$*. À direita, a inversão temporal de* P_X*. Acima de cada ponto está o valor de sua delimitação do funil. Assim,* $Funil(P_X) = 0 + 1 + 2 = 3$*, enquanto que* $Funil(\overline{P_X}) = 0 + 1 + 1 = 2$*.*

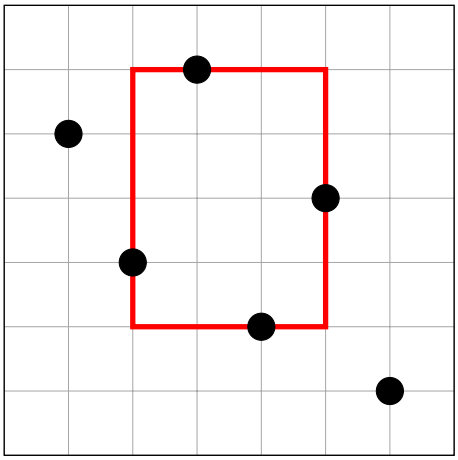


Figura 8.7: *O único z-retângulo da visão geométrica da sequência de acessos* $X = (6, 4, 2, 5, 1, 3)$*.*

Demonstração. Ao rotacionar P, o z-retângulo (p, q, r, s) de P se tornará o z-retângulo $(s^{\perp}, p^{\perp}, q^{\perp}, r^{\perp},)$ em $P^{\perp}$. Uma maneira simples de perceber esse fenômeno é rotacionar a Figura 8.7. Perceba que a x-coordenada relativa 3,1,4,2 é invariante em relação à rotações de 90°. □

Lema 8.3. *Para qualquer ponto* p *de um conjunto* P *de pontos, é possível encontrar* $\lfloor f(P, p)/2 \rfloor - 1$ *z-retângulos formados por* p *e três de pontos de seu funil.*

Demonstração. Para um ponto $p \in P$ com l pontos em seu funil, enumeremos os pontos de seu funil de maneira cronológica $(a_1, a_2, \ldots, a_l)$, onde $a_1.y < a_2.y < \ldots < a_l.y$. Suponha que $f(P, p) \geq 4$, pois caso contrário a delimitação é imediata.

Chamamos um par (i, j) de índices de *separadores esquerdos* se $i+1 < j, a_i.x > p.x, a_j.x > p.x$ e para todo k, com $i < k < j$, $a_k.x < p.x$. Em outras palavras, chamamos um par (i, j) de índices de separadores esquerdo se ambos os pontos a_i e a_j estão à direita de p e todos os pontos com índice k entre i e j estão à esquerda de p.

Sabemos que há $f(P, p)$ intercalações entre o funil esquerdo e o funil direito de p. Como os funis de p se alternam entre si $f(P, p) - 1$ vezes, então, no pior dos casos onde as intercalações se iniciam e terminam no funil esquerdo, existem $\lfloor f(P, p)/2 \rfloor - 1$ pares de

índices separadores de p. Veja a Figura 8.8.

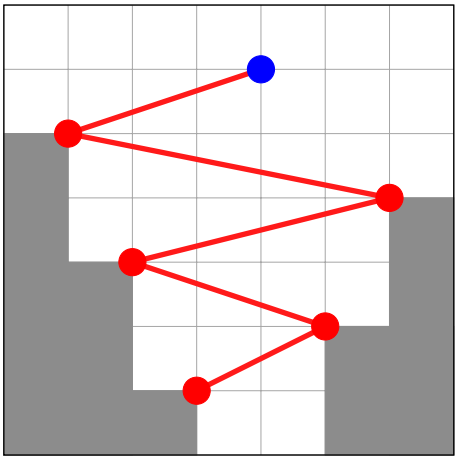


Figura 8.8: *Em azul, o ponto $p = (4, 6)$ do conjunto P de pontos. Em vermelho, os pontos de seu funil e suas intercalações. Perceba que $f(P, p) = 5$, o ponto mais alto e mais baixo do funil de p pertencem ao funil esquerdo e há apenas $\lfloor f(P, p)/2 \rfloor - 1 = 1$ par de índices separadores esquerdos.*

Assim, para cada par de índices (i, j) separadores esquerdos, note que os pontos (p, a_{i+1}, a_i, a_j) caracterizam um z-retângulo. Pela ordem dos índices, é fácil perceber que $p.y > a_j.y > a_{i+1}.y > a_i.y$. Como todos os pontos do funil direito possuem x-coordenada cronologicamente crescente, então sabemos que $a_j.x > a_i.x > p.x > a_{i+1}.x$. Assim, concluímos que as posições relativas estão corretas.

Por fim, é necessário garantir que $P \cap [a_{i+1}.x, a_j.x] \times [a_i.y, p.y]$. Por a_i e a_j serem pontos consecutivos do funil direito, sabemos que não há nenhum ponto no funil direito com y-coordenada entre $a_i.y$ e $a_j.y$. Como os pontos do funil esquerdo possuem x-coordenada cronologicamente decrescente e não existe nenhum ponto do funil de p com y-coordenada entre $a_i.y$ e $a_{i+1}.y$, então conclui-se que não há nenhum ponto q, $q \in P \setminus \{p, a_i, a_{i+1}, a_j\}$, na área descrita pelo $\{a_i, a_{i+1}\}$-retângulo, pelo $\{a_j, a_{i+1}\}$-retângulo e pelo $\{p, a_{i+1}\}$-retângulo. Por fim, pelos pontos do funil direito possuírem x-coordenada cronologicamente crescente, então também não há um ponto q, $q \in P \setminus \{p, a_j\}$, no $\{p, a_j\}$-retângulo. Veja a Figura 8.9.

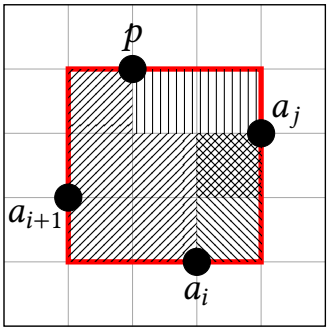


Figura 8.9: *Ilustração do argumento geométrico que mostra que os pontos (p, a_{i+1}, a_i, a_j) caracterizam um z-retângulo. Cada área hachurada representa uma parte diferente do argumento.*

□

Lema 8.4. *Para qualquer conjunto P de pontos, $zRet(P) \geq Funil(P)/2 - O(m)$.*

Demonstração. De acordo com o Lema 8.3, para cada ponto $p \in P$, é possível encontrar $\lfloor f(P,p)/2 \rfloor - 1$ z-retângulos distintos de P. Pelo Lema, para cada p, os z-retângulos encontrados referentes a esse ponto são formados por p e três pontos de seu funil, assim é fácil concluir que os z-retângulos encontrados para cada $p \in P$ são distintos entre si. Somando para todos os pontos, temos

$$\begin{aligned} zRet(P) &\geq \sum_{p \in P} (\lfloor f(P,p)/2 \rfloor - 1), \\ &\geq \sum_{p \in P} (f(P,p)/2 - 2), \\ &= \frac{1}{2} \sum_{p \in P} (f(P,p)) - 2m, \\ &= \text{Funil}(P)/2 - \text{O}(m). \end{aligned}$$

□

Lema 8.5. *Para qualquer conjunto P de pontos, $Funil(P) \geq 2 \cdot zRet(P)$.*

Demonstração. Fixe um ponto $p \in P$. Suponha que o número de z-retângulos formados por p e três pontos abaixo de p seja k. Rotule os pontos dos k z-retângulos desse formato em ordem crescente em relação a y-coordenada do ponto mais baixo por (p, q_1, r_1, s_1) até (p, q_k, r_k, s_k).

Considere dois z-retângulos consecutivos (p, q_i, r_i, s_i) e $(p, q_{i+1}, r_{i+1}, s_{i+1})$. Pela definição de z-retângulos sabemos que $r_i.y < q_i.y < s_i.y$ e $r_{i+1}.y < q_{i+1}.y < s_{i+1}.y$. Por conta da rotulação em ordem crescente na y-coordenada do ponto mais baixo dos z-retângulos, sabemos que $r_i.y < r_{i+1}.y$. Assim, $s_i.x \leq r_{i+1}.y$ pois caso contrário o ponto r_{i+1} está dentro do z-retângulo (p, q_i, r_i, s_i). Por consequência, $s_i.y \leq r_{i+1}.y$, pois caso contrário o ponto s_i está dentro do z-retângulo $(p, q_{i+1}, r_{i+1}, s_{i+1})$. Para auxiliar a visualização, veja a Figura 8.10. Agrupando essas propriedades, deduzimos que $r_i.y < q_i.y < s_i.y \leq r_{i+1}.y < q_{i+1}.y < s_{i+1}.y$. Assim, notamos que $q_1.y < s_1.y < q_2.y < s_2.y < \ldots < q_k.y < s_k.y$ para os z-retângulos considerados.

Pela definição de z-retângulos, sabemos que os pontos $q_1, s_1, \ldots, q_k, s_k$ pertencem ao funil de p. Assim, sabemos que há pelo menos $2k$ intercalações entre o funil esquerdo e o direito de p, ou seja, $f(P,p) = intercala(F_E(P,p).y, F_D(P,p).y) \geq 2k$. Sabemos que o conjunto de z-retângulos formados por um ponto $q \in P$ é disjunto do conjunto de z-retângulos formado pelo ponto $r \in P$, tal que $r \neq q$. Assim, ao somarmos para todos os pontos de P, temos que $\text{Funil}(P) \geq 2 \cdot zRet(P)$.

□

Corolário 8.5.1. *Para qualquer sequência X de acessos, $Funil(P_X) \geq Funil(\overline{P_X}) - \text{O}(m)$.*

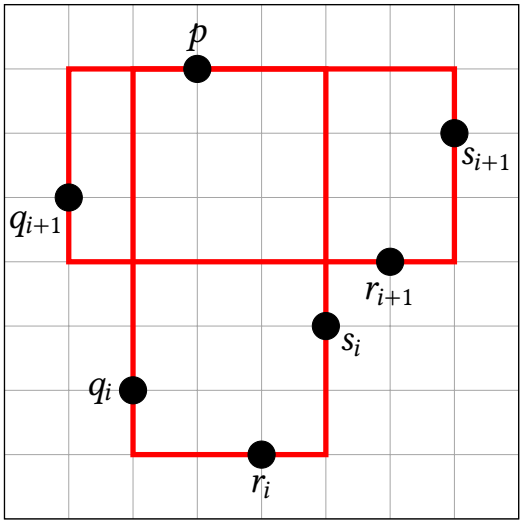


Figura 8.10: *Estão descritos dois z-retângulos consecutivos. Note que s_i e r_{i+1} podem ser rótulos do mesmo ponto.*

Demonstração.

$$\begin{aligned}
\text{Funil}(P_X) &\geq 2 \cdot zRet(P_X) && \text{pelo Lema 8.5,}\\
&= 2 \cdot zRet(P_X^{\perp}) && \text{pelo Lema 8.2,}\\
&= 2 \cdot zRet(P_X^{\perp\perp}) && \text{novamente pelo Lema 8.2,}\\
&= 2 \cdot zRet(\overline{P_X}), &&\\
&\geq \text{Funil}(\overline{P_X}) - \text{O}(m) && \text{pelo Lema 8.4.}
\end{aligned}$$

□

Teorema 8.6. *Para toda sequência X de acessos sem repetições e para toda árvore de referência $\mathcal{T}$ em relação à P_X, $Alt(P_X) \leq \text{O}(Funil(P_X) + m)$.*

Demonstração.

$$\begin{aligned}
\text{Alt}_{\mathcal{T}}(P_X) &\leq \text{Funil}(P_X) + \text{Funil}(\overline{P_X}) && \text{pelo Lema 8.1,}\\
&\leq \text{Funil}(P_X) + \text{Funil}(P_X) + \text{O}(m) && \text{pelo Corolário 8.5.1,}\\
&= \text{O}(\text{Funil}(P_X) + m).
\end{aligned}$$

□

Esses resultados são bastante impressionantes dado que delimitações construídas a partir de z-retângulos são invariantes em relação à rotações. Além disso, os z-retângulos nos mostram que rotações dos conjuntos de pontos de visões geométricas de sequência de acessos tem seu custo relacionado, ou seja, de grosso modo, fazer m buscas no conjunto de chaves de 1 a n tem seu custo relacionado com inverter os eixos e fazer as n buscas correspondentes no conjunto de chaves de 1 a m. Esse resultado é bastante impressionante.

Capítulo 9

Buscas múltiplas

Nesse capítulo mostraremos que ao retirar a propriedade da y-coordenada distinta do conjunto de pontos inicial, o problema de encontrar o menor superconjunto arboreamente satisfeito a partir de um conjunto de pontos é NP-difícil.

Usaremos a seguir uma série de conceitos básicos de complexidade computacional. Esses conceitos podem ser encontrados na seção 34 do livro Introduction to Algorithms de Cormen, Leiserson, Rivest e Stein [1].

Como evidenciado durante esse texto, ainda não se sabe se é possível computar minASS(P_X) para uma sequência X de acessos em tempo polinomial. Porém note que toda visão geométrica de sequências de acessos possui seus pontos com y-coordenada distinta. Essa propriedade é essencial para a esperança do desenvolvimento de um algoritmo com tempo polinomial para esse cálculo. Definimos o problema de buscas únicas em ABBs como encontrar o menor custo, dentro do modelo de computação adotado, para executar todos os acessos de uma sequência de acessos $X = (x_1, \dots, x_m)$, ou seja, em cada instante de tempo i, o algoritmo de busca em ABB precisa acessar o nó com chave x_i.

Ao retirarmos essa restrição, a estrutura se mantém com a diferença que permitimos buscas múltiplas em ABBs. Nesse novo problema estamos buscando encontrar o menor custo, dentro do modelo de computação adotado, para executar todos os acessos de uma sequência de conjuntos de acessos $Y = (C_1, C_2, \dots, C_m)$, onde C_i define um subconjunto do conjunto $C = \{1, 2, \dots, n\}$. Assim, no instante de tempo i, o algoritmo de busca em ABB precisa acessar todos os nós com chave $k \in C_i$ em ordem livre.

Mostraremos que para um conjunto P de pontos, em que seus pontos não possuem a restrição de y-coordenada distinta, o problema de encontrar o menor superconjunto de P que seja arboreamente satisfeito é NP-difícil.

Abaixo serão utilizados conceitos de planaridade da teoria dos grafos. Esses conceitos podem ser encontrados na seção 4 do livro Graph Theory de Diestel [4]. Exportamos também dessa seção o Teorema de Kuratowski:

Teorema de Kuratowski (1930). *Um grafo G é planar se e só se G não contém subdivisões nem do K_5 e nem do $K_{3,3}$.*

Para isso, utilizaremos um problema variante do 3SAT conhecido por Not-All-Equal 3SAT que é NP-completo para o caso não planar. Esse problema consiste em dadas k variáveis $x_1, x_2, \ldots, x_k$ e l cláusulas que possuem três dessas variáveis cada, encontrar uma valoração das k variáveis que faça com que cada cláusula possua pelo menos uma valoração verdadeira e uma valoração falsa.

Para descrever as variáveis faremos uso de linhas e fios entre pontos. Para um par (a, b) de pontos arboreamente insatisfeito, descrevemos o segmento de reta que liga a e b como uma *linha*. Um *fio* é um conjunto contíguo de linhas.

Todo fio indica um subconjunto de pontos onde há um número mínimo de pontos necessários a serem adicionados para torna-lo arboreamente satisfeito. É possível construir fios de maneira a ter apenas duas formas de tornar o subconjunto indicado por esse fio arboreamente satisfeito adicionando o número mínimo de pontos. Assim, denotamos uma dessas maneiras como a valoração verdadeira da variável x_i e a outra como a valoração falsa. Por padrão, nas imagens utilizaremos retângulos laranjas para a valoração verdadeira e retângulos azuis para a valoração falsa. Veja a Figura 9.1. A escolha de formas retangulares é para fins ilustrativos, retângulos representam possíveis conjuntos de pontos a serem adicionados.

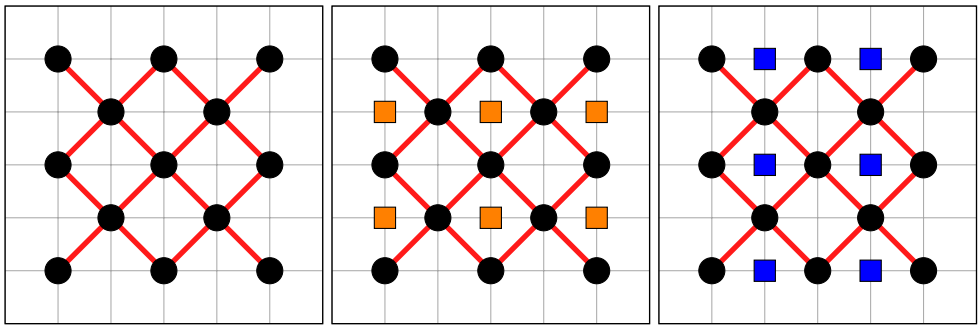


Figura 9.1: *À esquerda, um fio descrito pelas linhas vermelhas que representa uma variável. Ao meio, um superconjunto arboreamente satisfeito do conjunto de pontos da esquerda que representa a valoração verdadeira para a variável analisada. À direita, outro superconjunto arboreamente satisfeito que representa a valoração falsa para a mesma variável. Perceba que os conjuntos de pontos adicionados para a valoração verdadeira e para a valoração falsa são disjuntos e possuem mesmo tamanho.*

Para descrever as cláusulas, podemos nos utilizar de subestruturas que recebem três variáveis diferentes, ou seja fios diferentes, pelas pontas e organizam-as de maneira a garantir que o menor superconjunto arboreamente satisfeito dessa estrutura utiliza pelo menos uma valoração verdadeira e outra falsa. Veja a Figura 9.2.

Dependendo da geometria do problema e das cláusulas utilizadas, serão necessários ajustes adicionais. *Saltos* são espaçamentos adicionais que alternam a ordem das cores ao longo de um fio e podem ser usados para ajuste de paridade de maneira arbitrária. Além disso, podemos utilizar *adaptadores* que são espaçamentos para cruzamento de fios diferentes. Veja a Figura 9.3.

Assim, concluímos que é possível reduzir o problema 3SAT-Not-All-Equal no problema de busca do menor superconjunto arboreamente satisfeito. Foi provado por Moret [8] que o problema 3SAT-Not-All-Equal está em P se o grafo a seguir for planar: crie um vértice

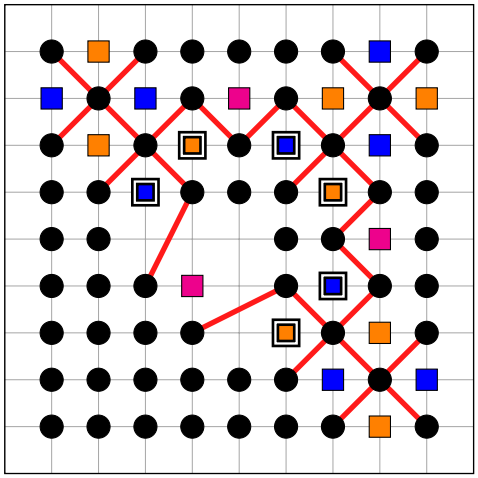


Figura 9.2: *Note que a única maneira de apenas adicionar dois dos três pontos indicados por retângulos rosas é com uma valoração verdadeira e outra falsa dentre as 3 valorações. Os retângulos laranjas/azuis destacados são aqueles relacionados aos retângulos rosa.*

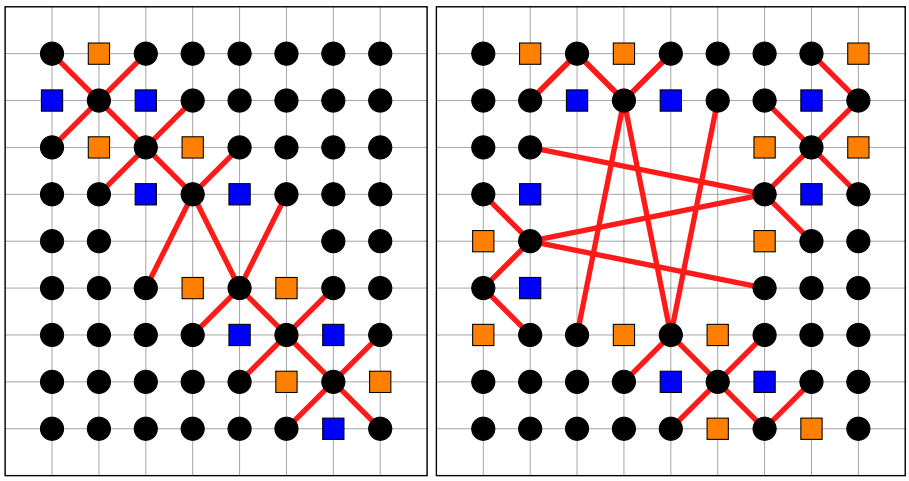


Figura 9.3: *À esquerda, um salto que tem como finalidade alternar a paridade ou fazer uma negação. À direita, um adaptador, onde dois fios de variáveis se cruzam sem interagir entre si.*

para cada variável, suas negações e para cada cláusula. Conecte os vértices de cada cláusula aos vértices das variáveis contidas em cada cláusula. Por fim, conecte todos os vértices relacionados à variáveis em um ciclo simples.

Faremos a redução do problema 3SAT-Not-All-Equal com 6 variáveis e com as cláusulas $C_1 = x_1 \wedge x_3 \wedge x_5$, $C_2 = x_1 \wedge x_4 \wedge x_5$, $C_3 = x_2 \wedge x_4 \wedge x_6$, $C_4 = \neg x_1 \wedge x_2 \wedge x_6$ e $C_5 = \neg x_1 \wedge x_3 \wedge x_6$ para o problema de buscas múltiplas. Veja que a Figura 9.4 mostra uma subdivisão do grafo K_5 em um subgrafo do problema, então pelo Teorema de Kuratowski concluímos que o grafo do problema não é planar e assim a instância é NP-completo.

Note que a Figura 9.5 mostra que é possível descrever o problema utilizando apenas as ferramentas descritas anteriormente. Assim, conclui-se que encontrar o menor super-conjunto do conjunto de pontos pretos da figura citada é uma solução para a instância do problema 3SAT-Not-All-Equal não planar descrito acima, ou seja, se soubermos resolver o problema de buscas múltiplas em tempo polinomial, então também sabemos resolver o

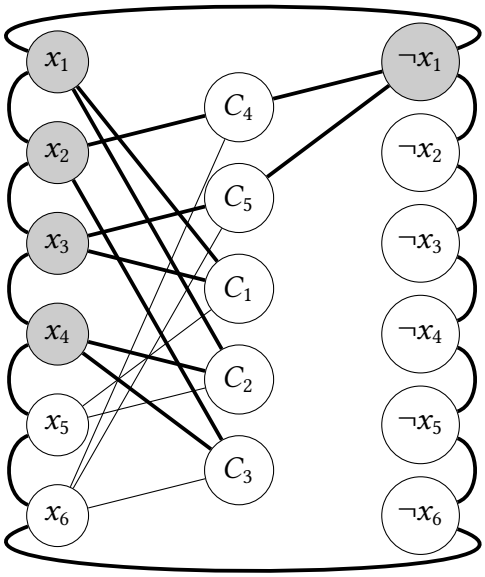


Figura 9.4: *Subdivisão* $\{x_1, x_2, x_3, x_4, \neg x_1\}$ *do* K_5 *do grafo descrito.*

problema 3SAT-Not-All-Equal não planar em tempo polinomial. Como 3SAT-Not-All-Equal não planar é um problema NP-completo, então o problema de múltiplas buscas também é.

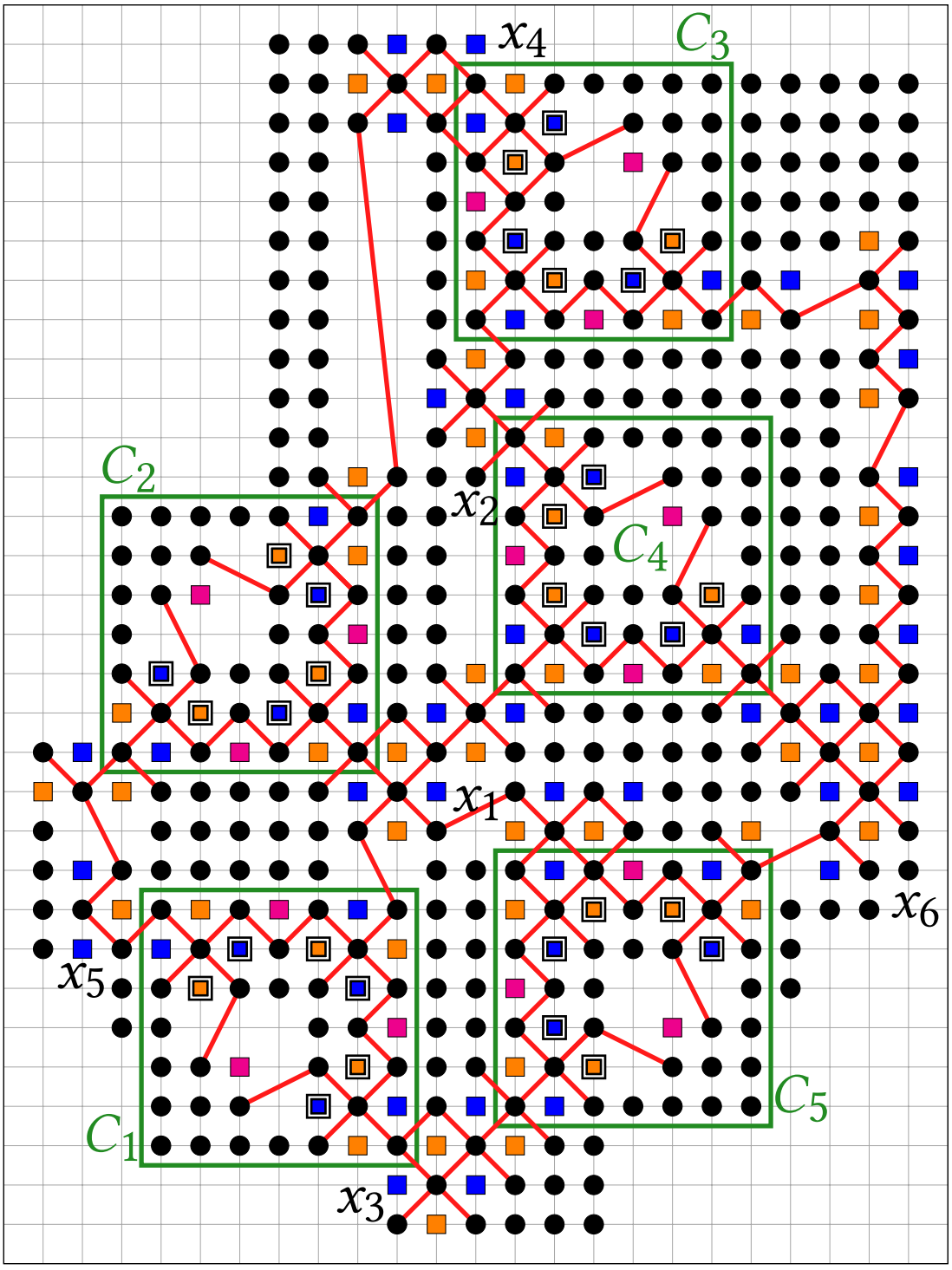


Figura 9.5: *Representação do problema dentro do contexto de buscas múltiplas. Perceba que todas as valorações possuem mesmo custo e o menor custo só se dá quando se respeita a restrição do problema 3SAT Not-All-Equal.*

Capítulo 10

Conclusão

Esse texto apresenta a Conjectura da Otimalidade Dinâmica assim como uma série de ferramentas que nos auxiliam a entender melhor o comportamento das árvores binárias de busca em contextos mais abstratos.

As árvores splay, cobertas nesse texto, são muito interessantes. A sua heurística simples e fácil de programar nos mostra que mesmo uma ABB que não gasta memória adicional pode ter uma performance assintoticamente eficiente.

É impressionante perceber que é possível traduzir o problema de encontrar o custo ótimo para um modelo de computação em um problema de encontrar superconjuntos com uma propriedade bastante simples da satisfação arbórea. Essa interpretação geométrica é especialmente útil por omitir as rotações sem desconsiderá-las, tornando assim o problema mais visual e intuitivo.

Além disso, essa abordagem permitiu o desenvolvimento de argumentos espaciais como o de conjuntos independentes de retângulos e o de z-retângulos. Esse foi o passo necessário para conseguir relacionar as duas delimitações propostas por Wilber, cuja conexão permaneceu indeterminada por mais de três décadas.

Por fim, para os leitores interessados, indico o estudo da árvore tango. Essa estrutura de dados, proposta por Demaine, Harmon, Iacono e Pătraşcu [2], possui para qualquer sequência X de acessos custo $O(\lg \lg n) \cdot OPT(X)$ e é a ABB conhecida com custo mais próximo do ótimo. Ela utiliza $O(\lg \lg n)$ bits a mais por nó e funciona simulando a delimitação da alternância e armazenando caminhos preferidos em ABBs balanceadas, onde *caminhos preferidos* são os caminhos maximais que passam apenas por filhos preferidos da ABB. Essa estrutura, é de muito interessante, mas é bastante desafiadora de entender, analisar e principalmente implementar. Dando continuidade a esse trabalho, o nosso plano é implementa-la durante o primeiro semestre de 2025 e complementar esse texto com a sua descrição.

Bibliografia

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. MIT Press, 2009 (ver p. 63).

[2] Erik D. Demaine et al. "Dynamic optimality — Almost". Em: *SIAM Journal on Computing* 37.1 (2007), pp. 240–251. URL: https://erikdemaine.org/papers/Tango_SICOMP/paper.pdf (ver p. 69).

[3] Erik D. Demaine et al. "The geometry of binary search trees". Em: *ACM-SIAM Symposium on Discrete Algorithms*. 2009. URL: https://api.semanticscholar.org/CorpusID:7609102 (ver pp. 21, 29, 31).

[4] Reinhard Diestel. *Graph Theory*. Springer Publishing Company, Incorporated, 2017. ISBN: 3662536218 (ver p. 63).

[5] Donald E. Knuth. "Optimum binary search trees". Em: *Acta Informatica* 1 (1971), pp. 14–25. URL: https://api.semanticscholar.org/CorpusID:5539760 (ver pp. 5–7).

[6] Victor Lecomte e Omri Weinstein. "Settling the relationship between Wilber's bounds for dynamic optimality". Em: *28th Annual European Symposium on Algorithms (ESA 2020)*. Ed. por Fabrizio Grandoni, Grzegorz Herman e Peter Sanders. Vol. 173. Leibniz International Proceedings in Informatics (LIPIcs). 2020, 68:1–68:21. ISBN: 978-3-95977-162-7. DOI: 10.4230/LIPIcs.ESA.2020.68. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2020.68 (ver p. 53).

[7] Joan M. Lucas. *Canonical Forms for Competitive Binary Search Tree Algorithms*. DCS-TR-. Rutgers University, Department of Computer Science, Laboratory for Computer Science Research, 1988. URL: https://books.google.com.br/books?id=b43HQwAACAAJ (ver p. 28).

[8] Bernard M. Moret. "Planar NAE3SAT is in P". Em: *SIGACT News* 19.2 (jun. de 1988), pp. 51–54. ISSN: 0163-5700. DOI: 10.1145/49097.49099. URL: https://doi.org/10.1145/49097.49099 (ver p. 64).

[9] James Munro. "On the competitiveness of linear search". Em: *8th Annual European Symposium on Algorithms (ESA 2000)*. Ed. por Mike S. Paterson. Vol. 1879, Lecture Notes in Computer Science (LNCS): Springer Berlin Heidelberg, 2000, pp. 338–345. ISBN: 978-3-540-45253-9 (ver pp. 28, 31).

[10] Robert Sedgewick e Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 2011 (ver p. 1).

[11] Daniel D. Sleator e Robert E. Tarjan. "Self-adjusting binary search trees". Em: *J. ACM* 32.3 (1985), pp. 652–686. ISSN: 0004-5411. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835 (ver pp. 3, 4, 11, 18).

[12] Daniel D. Sleator, Robert E. Tarjan e William P. Thurston. “Rotation distance, triangulations, and hyperbolic geometry”. English (US). Em: *Journal of the American Mathematical Society* 1.3 (jul. de 1988), pp. 647–681. ISSN: 0894-0347. DOI: 10.1090/S0894-0347-1988-0928904-4 (ver p. 2).

[13] Robert Wilber. “Lower bounds for accessing binary search trees with rotations”. Em: *SIAM Journal on Computing* 18.1 (1989), pp. 56–67. DOI: 10.1137/0218004 (ver pp. 3, 41).