UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

GEORGE EDSON ALBUQUERQUE PINTO

OTIMALIDADE DINÂMICA: UM *SURVEY*

FORTALEZA

2020

GEORGE EDSON ALBUQUERQUE PINTO

OTIMALIDADE DINÂMICA: UM *SURVEY*

Proposta de qualificação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação da Universidade Federal do Ceará (UFC) como requisito parcial para a defesa de dissertação.
Orientador: Prof. Dr. Victor Almeida Campos

FORTALEZA

2020

GEORGE EDSON ALBUQUERQUE PINTO

OTIMALIDADE DINÂMICA: UM *SURVEY*

Proposta de qualificação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação da Universidade Federal do Ceará (UFC) como requisito parcial para a defesa de dissertação.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Victor Almeida Campos (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Manoel Bezerra Campêlo Neto
Universidade Federal do Ceará (UFC)

Prof. Dr. Carlos Vinícius Gomes Costa Lima
Universidade Federal do Ceará (UFC)

Prof. Dr. Francicleber Martins Ferreira
Universidade Federal do Ceará (UFC)

Prof. Dr. Nicolas de Almeida Martins
Universidade da Integração Internacional da Lusofonia Afro-Brasileira (UNILAB)

RESUMO

As Árvores Binária de Busca (*BSTs*) são uma das estruturas de dados clássica para a Ciência da Computação e que, apesar de ser uma estrutura de dados simples, possui muitas questões em aberto depois de décadas de pesquisa. A principal questão em aberto sobre *BSTs* ainda permanece sem solução: "Qual é a melhor Árvore Binária de Busca?". Em 1983 Sleator e Tarjan [1] propuseram a Árvore *Splay* e conjecturaram que o custo para realizar qualquer sequência de buscas S nesta *BST* é tão bom, assintoticamente, quanto o menor custo possível, denotado por $\mathrm{OPT}(S)$. Esta é a famosa conjectura da otimalidade dinâmica, onde qualquer *BST* que realiza qualquer sequência de buscas S com custo $O(\mathrm{OPT}(S))$ é dita dinamicamente ótima. Desde que a conjectura foi proposta houveram muitas tentativas de prová-la, mas ainda não foi provado que a Árvore *Splay*, ou qualquer outra *BST*, é dinamicamente ótima. O melhor custo conhecido é $\mathcal{O}(\log\log n\,\mathrm{OPT}(S))$ da Árvore Tango proposta por Demaine *et al.* [2], e das árvores *Multi-Splay Tree* [3] e *Chain-Splay Tree* [4]. Através do estudo de *BSTs*, Demaine *et al.* [5] propuseram um problema de pontos no plano que equivale ao problema de busca em uma *BST*. Este é resultado surpreendente, pois é uma maneira mais fácil de visualizar com uma execução *BST*. Finalmente, esta dissertação trata-se de um *survey* da literatura sobre a otimalidade dinâmica, onde reunimos os principais resultado relacionados ao problema.

Palavras-chave: Árvore Binária de Busca. Otimalidade dinâmica. Estrutura de dados. Limitantes.

ABSTRACT

Binary Search Trees (*BSTs*) are one of the classic data structures for Computer Science and which, despite being a simple data structure, has many open questions after decades of research. The main open question about *BSTs* still remains unsolved: "What is the best Binary Search Tree?". In 1983 Sleator e Tarjan [1] proposed the *Splay* Tree and conjectured that the cost of performing any search sequence S in this *BST* is as good, asymptotically, as the lowest possible cost, denoted by $\mathrm{OPT}(S)$. This is the famous conjecture of dynamic optimality, where any *BST* that performs any search sequence S with cost $O(\mathrm{OPT}(S))$ is said to be dynamic optimal. Since the conjecture was proposed, there have been many attempts to prove it, but it has not yet been proven that the *Splay* Tree, or any other *BST*, is dynamic optimal. The best known cost is $\mathcal{O}(\log\log n\,\mathrm{OPT}(S))$ from the Tango Tree proposed by Demaine *et al.* [2], and from the *Multi-Splay Tree* [3] and *Chain-Splay Tree* [4]. Through the study of *BSTs*, Demaine *et al.* [5] proposed a problem of points in the plane that is equivalent to the search problem in a *BST*. This is a surprising result, beacause it is an easier way to visualize a *BST* execution. Finally, this dissertation is a survey of the literature on dynamic optimality, where we bring together the main results related to the problem.

Keywords: Binary Search Tree. Dynamic Optimality. Data structure. Bounds.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE SÍMBOLOS

$\square ab$	Retângulo alinhado ao eixos definido pelos pontos a e b
$\text{chave}(x)$	Chave do nó x de uma *BST*
$\text{d}_\text{T}(x)$	Profundidade do nó x de uma *BST* T
$\text{dir}(x)$	Filho direito do nó x de uma *BST*
$\text{esq}(x)$	Filho esquerdo do nó x de uma *BST*
$\text{LCA}_T(x,y)$	Menor ancestral comum dos nós x e y de uma *BST* T
$\text{pai}_T(x)$	Pai do nó x de uma *BST* T
$\text{raiz}(T)$	Nó raiz da uma *BST* T
$\text{Rot}(x)$	Rotação do nó x em uma *BST*
$T_1 \xrightarrow{Q} T_2$	Rearranjo de uma *BST* T_1 para uma *BST* T_2
$\text{T}_\text{L}(x)$	Subárvore esquerda do nó x de uma *BST*
$\text{T}_\text{R}(x)$	Subárvore direita do nó x de uma *BST*

SUMÁRIO

1 INTRODUÇÃO

O problema de busca é um dos problemas clássicos de Ciência da Computação. Suponha que precisamos armazenar um conjunto S de n elementos de algum universo ordenado $\mathcal{U}$ e queremos realizar consultas do tipo: um certo elemento $x \in \mathcal{U}$ está em S? E em caso de resposta "SIM", queremos recuperar dados adicionais associados a x. Para verificar se uma chave x está em S, podemos comparar x com cada elemento de S. Esta estratégia nos leva a resposta, mas pode gastar muito tempo, se S possuir muitos elementos. Uma outra estratégia é fazer comparações, dos tipos $<$, $>$ e $=$, entre x e os elementos de S, pois como $\mathcal{U}$ é um universo ordenado, quando comparamos x com $y \in S$ e $x < y$, então não precisamos mais comparar x com elementos de S maiores do que y. E de forma simétrica, esta observação também é válida quando $x > y$. Verificar se $x \in S$ com esta nova estratégia pode, possivelmente, ser feita com menos comparações do que a primeira estratégia e, consequentemente, pode gastar menos tempo.

Para este problema podemos utilizar uma Árvore Binária de Busca (do inglês, *Binary Search Tree - BST*), pois uma *BST* pode armazenar um conjunto ordenado de chaves e, além disto, podemos realizar buscas por chaves. As Árvores Binária de Busca é uma estruturas de dados clássica, e foram apresentadas inicialmente por Hibbard [6] em 1962. Uma *BST* é uma estrutura de dados simples, mas que apesar de décadas de pesquisa ainda possui muitas questões em aberto. A principal questão sobre *BSTs*, apresentada por Sleator e Tarjan [1] em 1983, ainda permanece sem solução: "Qual é a melhor Árvore Binária de Busca?". *BSTs* são utilizadas em *kernels* de sistemas operacionais e em memória *cache* em redes de computadores [7].

O problema de busca em *BST* possui duas versões, a estática e a dinâmica, que é quando a *BST* tem sua estrutura fixada e quando a estrutura da *BST* pode ser modificada durante as buscas, respectivamente. Estes problemas são descritos com mais precisão no Capítulo 3. Nesta dissertação estamos mais interessados na versão dinâmica, pois esta versão ainda possui muitas questões em aberto. Para a versão estática já é conhecido um algoritmo de custo $\mathcal{O}(n^2)$, proposto por Knuth [8] em 1971, para obter uma *BST* que pode atingir o menor custo possível.

Uma busca em uma *BST* é realizada por um algoritmo $\mathcal{A}$, chamado de algoritmo *BST*, que pode mover-se entre as chaves da árvore e deve visitar a chave buscada. O custo de uma busca é a quantidade de chaves visitadas por $\mathcal{A}$ durante a busca. No Capítulo 2 descrevemos o modelo *BST* formalmente. Uma busca em uma *BST* utiliza a segunda estratégia descrita anteriormente, na tentativa de reduzir o custo da busca. O menor custo possível para $\mathcal{A}$ realizar uma sequência de buscas S é chamado de custo ótimo, e denotado por $\mathrm{OPT}(S)$.

Dizemos que $\mathcal{A}$ é dinamicamente ótimo se, para qualquer sequência de buscas S, $\mathcal{A}$ pode realizar a sequência de buscas com custo $\mathcal{O}(\mathrm{OPT}(S))$. Ou seja, um algoritmo *BST* dinamicamente ótimo é assintoticamente tão bom quanto o melhor algoritmo *BST* para qualquer sequência de buscas. Sleator e Tarjan [9] apresentaram a *Árvore Splay*, uma *BST*, e conjecturaram que esta árvore é dinamicamente ótima, ou seja, o custo para realizar qualquer sequência de buscas S em uma Árvore *Splay* é $\mathcal{O}(\mathrm{OPT}(S))$. Lucas [10] também propôs um algoritmo *BST* guloso e conjeturou que seu algoritmo fornece uma aproximação de fator constante para o custo ótimo. Munro [11] propôs um algoritmo semelhante ao de Lucas independentemente mais de uma década depois. No entanto, ainda não foi provado que a Árvore *Splay*, ou qualquer outro algoritmo *BST*, é dinamicamente ótimo.

Por muito tempo o melhor custo provado para realizar uma sequência de buscas S em um *BST* era $\mathcal{O}(\log n\,\mathrm{OPT}(S))$. Este custo pode ser obtido buscando por S em uma *BST* balanceada (*BST* de altura $\log n$). Recentemente Demaine *et al.* [2] apresentaram a Árvore Tango, que tem custo $\mathcal{O}(\log\log n\,\mathrm{OPT}(S))$ para buscar por S. Este é o melhor resultado conhecido atualmente. Posteriormente Wang, Derryberry e Sleator [3, 12] propuseram a *Multi-Splay Tree* e Georgakopoulos [4] propôs a *Chain-Splay Tree*, que também possuem custo $\mathcal{O}(\log\log n\,\mathrm{OPT}(S))$.

Recentemente Demaine *et al.* [5] apresentaram uma representação de uma sequência de buscas em *BST* através de um conjunto de pontos no plano, chamada de Visão Geométrica, para visualizar as buscas em uma *BST* de forma mais fácil. Os autores definiram um problema através desta representação equivalente ao problema dinâmico de busca em *BST*. Na tentativa de provar a otimalidade dinâmica através desta representação, Demaine *et al.* [13] propuseram dois modelo de programação linear inteira, mas não obtiveram bons resultados. Derryberry, Sleator e Wang [14] também apresentaram uma representação semelhante independentemente.

Pelo motivo de não ter sido provado que nenhum algoritmo *BST* é $\mathcal{O}(\mathrm{OPT}(S))$, alguns limitantes, superiores e inferiores, são mostrados para determinar a distância entre o custo de um algoritmo *BST* e o valor da solução ótima. O custo de qualquer algoritmo *BST* é um limitante superior, mas, como o objetivo é aproximar este custo o máximo possível do custo ótimo, nem todo algoritmo dá um bom limitante superior. O melhor limitante superior conhecido é $\mathcal{O}(\log\log n\,\mathrm{OPT}(S))$, obtido nas Árvores Tango [2], *Multi-Splay Tree* [3] e *Chain-Splay Tree* [4]. Por outro lado, um limitante inferior mostra que a solução ótima é pelo menos o valor deste limitante. Um limitante inferior trivial é o tamanho da sequência de buscas, pois em cada busca pelo menos um nó é visitado, mas este limitante pode ser muito distante da solução ótima.

Wilber [15] propôs dois limitantes inferiores, descritos no Capítulo 5. Recentemente, Lecomte e Weinstein [16] responderam a conjectura apresentada por Wilber de que o seu segundo limitante domina o primeiro. Outros limitantes inferiores foram propostos usando sua visão geométrica. Demaine *et al.* [5] e Derryberry, Sleator e Wang [14] propuseram independentemente limitantes inferiores baseados em suas visões geométricas. Outra forma de tentar atingir otimalidade dinâmica é caracterizar algumas sequências S de buscas nas quais um determinado um algoritmo *BST* tem custo $\mathcal{O}(\mathrm{OPT}(S))$.

Esta dissertação trata-se de um *survey* sobre o problema de busca em Árvores Binária de Busca. Artigos dos principais autores da área em estudo foram utilizados como fonte de pesquisa, possibilitando que este tomasse forma para ser fundamentado.

Este *survey* é organizado em sete capítulos como segue. No Capítulo 2 formalizamos os conceitos e definições de Árvore Binária de Busca necessárias para a compreensão do problema. No Capítulo 3 apresentamos a definição do problema de busca em *BST* nas versões estática e dinâmica. No Capítulo 4 apresentamos a definição da visão geométrica proposta por Demaine *et al.* [5] e mostramos a equivalência com o problema de busca em *BST*. No Capítulo 5 apresentamos limitantes superiores e inferiores para o custo ótimo propostos na literatura. No Capítulo 6 descrevemos algumas tentativas recentes de provar a otimalidade dinâmica. Finalmente, no Capítulo 7 apresentaremos as considerações finais.

2 ÁRVORES BINÁRIA DE BUSCA E SEUS MODELOS

Neste capítulo, formalizamos os conceitos e definições de Árvore Binária de Busca necessárias para a compreensão do problema estudado (Seção 2.1). Além disso, apresentamos alguns modelos *BST* propostos na literatura e definimos o modelo usado nesta dissertação (Seção 2.2). Os conceitos e terminologias aqui apresentados seguem os utilizados por Szwarcfiter e Markenzon [17] e Cormen *et al.* [18]. Alguns termos e abreviações usadas neste capítulo são mantidas como no inglês por ser de uso comum na literatura.

2.1 Árvore Binária de Busca

Suponha que precisamos armazenar um conjunto S de n elementos de algum universo ordenado $\mathcal{U}$ e queremos realizar consultas do tipo: um certo elemento $x \in \mathcal{U}$ está em S? E em caso de resposta "SIM", queremos recuperar dados adicionais associados a x. Para este problema podemos utilizar uma *Árvore Binária de Busca*.

Uma *Árvore Binária* T é um conjunto finito de elementos denominados nós, tal que: $T = \emptyset$ e a árvore é dita *vazia*; ou existe um nó especial r, denominado *raiz* de T e denotado por raiz(T), e os nós restantes podem ser particionados em dois subconjuntos $\mathrm{T_L}(r)$ e $\mathrm{T_R}(r)$, as *subárvores esquerda* e *direita* de r, respectivamente, as quais são árvores binárias. A raiz da subárvore esquerda (direita) de um nó x, se existir, é denominada *filho esquerdo* (*direito*) de x, denotados por esq(x) e dir(x), respectivamente, enquanto x é o nó *pai* de ambos. O nó pai de x é denotado por $\mathrm{pai}_T(x)$, ou simplesmente pai(x) quando está claro a árvore referenciada. O nó raiz é o único nó que não possui pai. Se o filho esquerdo (direito) de um nó x não existir, então a subárvore esquerda (direita) de x é vazia. Um nó que não possui ambos os filhos é chamado de *folha*.

Uma *subárvore* T' de T é uma árvore binária, tal que os nós de T' são um subconjunto dos nós de T e para todo nó x, que não é raiz de T', $\mathrm{pai}_{T'}(x) = \mathrm{pai}_T(x)$. A subárvore contendo um nó x e todos os nós das suas subárvores esquerda e direita é chamada de *subárvore enraizada* em x. O *tamanho da árvore*, denotado por $|T|$, é a quantidade de nós contidos na árvore. Dizemos que um conjunto de nós Q *induz* uma árvore binária T' de uma árvore binária T, se T' é subárvore de T com conjunto de nós Q. Além disso, dizemos que um subconjunto Q dos nós de T é *conexo*, se para quaisquer dois nós $x, y \in Q$ existe um caminho iniciando em x e terminando em y em T contendo apenas nós de Q.

Um *caminho* em uma árvore binária é uma sequência de nós $P = (x_1, x_2, \ldots, x_p)$ sem repetições, tal que $\text{pai}(x_i) = x_{i+1}$ ou $\text{pai}(x_{i+1}) = x_i$. A *profundidade* de x (*depth*, do inglês) em uma árvore T é a quantidade de arestas no caminho entre x e $\text{raiz}(T)$, denotada por $\text{d}_\text{T}(x)$. A *altura de T* é a maior profundidade de um nó em T. Um *nível* em uma árvore binária consiste de todos os nós com a mesma profundidade. Ou seja, um nó x está no nível d se $\text{d}_\text{T}(x) = d$.

A *espinha esquerda* (*direita*) de uma árvore binária é o maior caminho iniciando na raiz da árvore contendo apenas filhos esquerdo (direito). Uma árvore binária é *enviesada à esquerda* (*direita*), se todos os nós, exceto a raiz, são filhos esquerdo (direito). Os *ancestrais* de um nó x em uma árvore binária T são os nós no caminho de x a $\text{raiz}(T)$. Um nó y é *descendente* de um nó x se x é ancestral de y. Todo nó é um ancestral e um descente dele mesmo. O *menor ancestral comum* (*Lowest Common Ancestor* - LCA, do inglês) de x e y em uma árvore T é o ancestral mais profundo compartilhado por x e por y. Denotamos o menor ancestral comum por $\text{LCA}_T(x,y)$. Quando está claro sobre qual árvore está sendo referenciada, representamos simplesmente por $\text{LCA}(x,y)$.

Uma árvore binária é *completa*, se x é um nó que possui alguma de suas subárvores vazia, então x está no último ou penúltimo nível da árvore. E uma árvore binária é *cheia* se todos os nós que possuem alguma de suas subárvores vazia está no último nível. O Lema 2.1.1 mostra uma relação entre a quantidade de nós e altura de uma árvore binária completa e o Lema 2.1.2 mostra que as árvores binárias completas são as que possuem altura mínima. Estes dois lemas são adaptados de Szwarcfiter e Markenzon [17].

Lema 2.1.1 ([17])**.** *Se uma árvore binária T com n nós é completa, então sua altura é $h = \lfloor \log_2 n \rfloor$.*

Demonstração. Antes mostrar que $h = \lfloor \log_2 n \rfloor$ precisamos fazer algumas observações. Primeiro, podemos observar que em cada nível k de uma árvore binária tem pelo menos 1 nó e no máximo 2^k nós. A partir disso, podemos concluir que uma árvore binária cheia de altura k possui $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$ nós.

Agora vamos mostrar que $h = \lfloor \log_2 n \rfloor$. Seja T' a árvore binária obtida de T pela remoção dos k nós do último nível. Como T é completa, T' é cheia com altura $h-1$ e, portanto, $n' = 2^{h-1+1} - 1 = 2^h - 1$ nós. Pela observação feita anteriormente, $1 \leq k \leq 2^h$. Assim, temos que:

$$\begin{array}{rcccl} n'+1 & \leq & n & \leq & n'+2^h \\ 2^h-1+1 & \leq & n & \leq & 2^h-1+2^h \\ 2^h & \leq & n & \leq & 2^{h+1}-1 \\ 2^h & \leq & n & < & 2^{h+1} \\ h & \leq & \log_2 n & < & h+1 \\ h & = & \lfloor \log_2 n \rfloor & & \end{array}$$

□

Lema 2.1.2 ([17]). *Se T é uma árvore binária completa, então T possui altura mínima.*

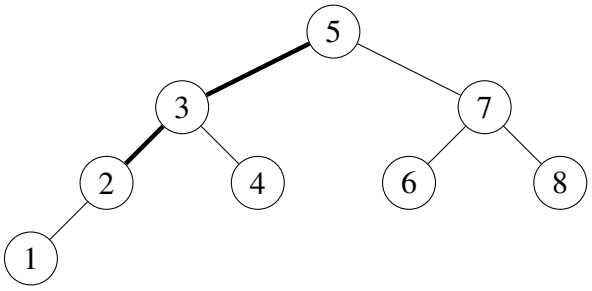
Demonstração. Seja T_1 uma árvore binária de altura mínima com n nós. Se T_1 é completa, então, pelo Lema 2.1.1, T e T_1 possuem a mesma altura, isto é, T possui altura mínima. Caso contrário, efetua-se a seguinte operação em T_1: retirar uma folha w de seu último nível e tornar w filho de algum nó v, localizado em algum nível acima do penúltimo, que possui alguma de suas subárvores vazia. Repete-se esta operação enquanto for possível. Ou seja, até que a árvore T_2, resultante da transformação, seja completa. Podemos observar que a altura de T_2 não pode ser menor do que a de T_1, pois T_1 possui altura mínima, nem maior, pois nenhum nó foi movido para um nível abaixo. Então as alturas de T_1 e T_2 são iguais. Como T_2 é completa, novamente pelo Lema 2.1.1, as alturas de T e T_2 são iguais. Portanto, T possui altura mínima. □

Uma *Árvore Binária de Busca* (*Binary Search Tree* - BST, do inglês) T é uma árvore binária, tal que cada nó x de T tem uma *chave* em um conjunto ordenado $\mathcal{U}$, denotada por chave(x). Além disso, as chaves possuem a seguinte *propriedade de ordenação*: se x, y_1 e y_2 são nós de uma *BST*, tal que $y_1 \in \mathrm{T_L}(x)$ e $y_2 \in \mathrm{T_R}(x)$, então chave$(y_1) \leq$ chave(x) e chave$(y_2) \geq$ chave(x). O (único) caminho entre a raiz de uma *BST* e um de seus nós x é chamado de *caminho de busca* por x. Para simplificar a notação, considere que os nós da *BST* são elementos do conjunto $\mathcal{U}$. Assim, para dois nós x e y em uma *BST*, podemos comparar seus valores escrevendo simplesmente $x \leq y$ ao invés de chave$(x) \leq$ chave(y), por exemplo. Observe que em uma *BST* T $\min\{x,y\} \leq \mathrm{LCA}_T(x,y) \leq \max\{x,y\}$. Dizemos que duas *BSTs* T_1 e T_2 são *idênticas* se ambas possuem o mesmo conjunto de nós e os descendentes de todo nó x são os mesmos em ambas as árvores.

Na Figura 1 temos uma representação visual de uma *BST* com o conjunto de chaves $\{1,\ldots,8\}$. O nó raiz desta *BST* é 5. Os descendentes de 3 são $\{1,2,3,4\}$, e os ancestrais são

$\{3,5\}$. Os nós $\{2,4,6,8\}$ formam o seu nível 2. Também temos que $\text{LCA}(1,4) = 3$. O caminho em destaque $(5,3,2)$, é o caminho de busca por 2.

Figura 1 – Árvore Binária de Busca (*BST*)



Fonte: Elaborada pelo autor

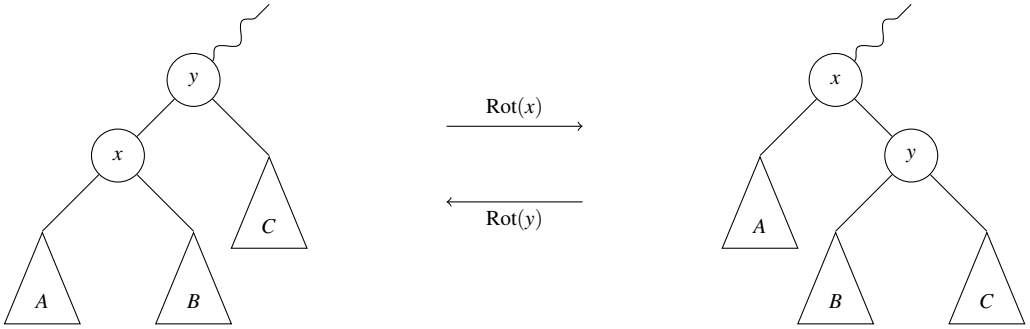
Dizemos que uma classe de árvores binárias é *balanceada* se suas árvores com n nós tem altura $\mathcal{O}(\log n)$. As *BSTs AVL Tree* e *Red-Black Tree*, apresentadas por Adelson-Velskiĭ e Landis [19] e por Guibas e Sedgewick [20], respectivamente, são exemplos de *BSTs* balanceadas.

A estrutura de uma *BST* pode ser modificada para outra *BST* através de uma *rotação* de um nó e seu pai. Na Definição 2.1.1 apresentamos a definição de rotação.

Definição 2.1.1 (Rotação)**.** *Se T_1 e T_2 são* BSTs*, tal que T_2 é resultante da rotação do nó x e seu pai y em T_1, então* $\text{pai}_{T_2}(y) = x$ *e* $\text{pai}_{T_2}(x) = \text{pai}_{T_1}(y)$*. Além disso, se x é filho esquerdo (direito) de y e z o filho direito (esquerdo) de x, então* $\text{pai}_{T_2}(z) = y$*, se z existir, e para todo nó v que não é x, y ou z,* $\text{pai}_{T_2}(v) = \text{pai}_{T_1}(v)$.

Denotamos uma rotação do nó x por $\text{Rot}(x)$. Uma rotação preserva a propriedade de ordenação da *BST*. Se o nó rotacionado x é filho esquerdo (direito), então esta rotação é chamada de *rotação direita* (*esquerda*). Segundo Kozma [21], as rotações foram usadas primeiramente por Adelson-Velskiĭ e Landis [19] na *AVL Tree*. Na Figura 2 temos exemplos dos dois tipos de rotações, direita e esquerda.

Figura 2 – Rotação em *BST*



Fonte: Elaborada pelo autor

2.2 Modelos *BST*

Para consultar se um certo elemento $x \in \mathcal{U}$ está em um conjunto S, podemos utilizar uma *BST* cujas chaves correspondem aos elementos de S. Esta consulta é chamada de *busca*. Para realizar uma busca em uma *BST* é utilizado um *ponteiro* z apontando para um nó da árvore, tal que z pode realizar as seguintes operações *BST*: mover-se ou para o pai ou para o filho esquerdo ou para o filho direito do nó apontado ou rotacionar o nó apontado. Estas operações são denotadas por $z \leftarrow \text{pai}(z)$, $z \leftarrow \text{esq}(z)$, $z \leftarrow \text{dir}(z)$ e $\text{Rot}(z)$, respectivamente. Quando um nó é apontado por z, dizemos que o nó foi *visitado* nesta busca.

Uma busca por uma chave x em uma *BST* T consiste em fazer z apontar para a raiz de T, onde z pode realizar uma sequência de operações *BST* em qualquer ordem; no entanto, z deve visitar o nó com chave x em algum momento da busca. Nesta dissertação consideramos que os valores das chaves em uma *BST* são os inteiros $1,\ldots,n$ e que todas as chaves buscadas estão na árvore.

Alguns modelos *BST* foram propostos. Wilber [15] definiu um modelo em que as operações "$z \leftarrow \text{pai}(z)$", "$z \leftarrow \text{esq}(z)$", "$z \leftarrow \text{dir}(z)$" e "$\text{Rot}(z)$" tem custo 1. Desta forma, o custo para buscar uma chave é quantidade de movimentos do ponteiro mais a quantidade de rotações. Outro modelo *BST* é definido por Demaine *et al.* [5], onde o custo de uma busca é a quantidade de nós visitados pelo ponteiro durante a busca.

O modelo de Demaine *et al.* é equivalente ao modelo de Wilber por um fator constante. Mostramos esta equivalência no Lema 2.2.3, mas antes disso mostramos o algoritmo TRANSFORMAR, que transforma uma *BST* em uma árvore enviesada a direita, e o Lema 2.2.1, que mostra que uma *BST* pode ser transformada em outra *BST* com o mesmo conjunto de nós com custo linear. Este algoritmo e este lema são ferramentas para o Lema 2.2.3.

O algoritmo TRANSFORMAR usa o algoritmo LIMPAESQUERDA, para transformar a *BST* de forma que a subárvore esquerda do nó apontado seja vazia, para isso o ponteiro move-se para o filho esquerdo, se existir, e o rotaciona. Este algoritmo recebe uma pilha P para empilhar operações *BST*, onde armazena em P as operações *BST* reversas às realizadas. Além disso, o algoritmo tem acesso ao ponteiro z do TRANSFORMAR. Para cada movimento de z para o filho esquerdo e rotação realizada são empilhadas as operações $z \leftarrow \text{dir}(z)$ e $\text{Rot}(z)$. As operações são empilhadas pelo comando $\text{Empilha}(P, op_1, \ldots, op_n)$, onde a sequência de operações $(op_1, \ldots, op_n)$ são empilhadas em P na ordem $op_n, \ldots, op_1$ para que a ordem de remoção da pilha seja op_1 para op_n.

Algoritmo 1: LIMPAESQUERDA

```
   Entrada: Pilha de operações BST P
1  início
2      enquanto T_L(z) ≠ ∅ faça
3          z ← esq(z)
4          Rot(z)
5          Empilha(P, z ← dir(z), Rot(z))
6      fim
7  fim
```

O algoritmo TRANSFORMAR tem como entrada uma *BST T* e transforma T em uma *BST* enviesada a direita. O algoritmo inicia com um ponteiro z apontado para a raiz de T. O LIMPAESQUERDA é usado para deixar a subárvore esquerda do nó apontado vazia, e quando o LIMPAESQUERDA termina, ou seja, quando a subárvore esquerdo de z é vazia, z move-se para seu filho direito, se existir. O algoritmo termina quando as subárvores esquerda e direita de z são vazias, ou seja, quando a árvore é enviesada a direita. Além disso, o algoritmo armazena em um pilha P as operações *BST* reversas às realizadas. Para cada movimento de z para o filho direito é empilhada a operação $z \leftarrow \text{pai}(z)$. Cada vez que LIMPAESQUERDA é executado P é passada. Ao final da execução o TRANSFORMAR retorna a pilha de operações gerada.

Algoritmo 2: TRANSFORMAR

```
   Entrada: BST T
   Saída : Pilha de operações BST P
1  início
2      z ← raiz(T)
3      Pilha ← ∅
4      LIMPAESQUERDA(P)
5      enquanto T_R(z) ≠ ∅ faça
6          z ← dir(z)
7          Empilha(P, z ← pai(z))
8          LIMPAESQUERDA(P)
9      fim
10     retorne P
11 fim
```

Observação 2.2.1. *No algoritmo* TRANSFORMAR *como as operações na pilha são inversas às operações realizadas sobre z, desempilhar as operações uma a uma e realizá-las na ordem de remoção com z na mesma posição que no final do algoritmo (ao final do algoritmo z está no único nó folha de R), R é transformada de volta em T.*

No Lema 2.2.1 mostramos que uma *BST* com n nós pode ser transformada em outra *BST* com o mesmo conjunto de nós usando uma quantidade linear de operações *BST*. Culik e Wood [22] mostraram que esta transformação pode ser feita utilizando no máximo $2n-2$ rotações.

Lema 2.2.1. *Uma* BST T_1*, onde* $|T_1| = n$*, pode ser transformada em uma* BST T_2 *com o mesmo conjunto de nós com máximo* $6n-6$ *operações* BST*. Além disso, esta transformação pode ser feita com no máximo* $2n-2$ *rotações.*

Demonstração. Sejam T_1 e T_2 *BSTs* com o mesmo conjunto de n nós. Para transformar T_1 em T_2, podemos primeiramente transformar T_1 em uma *BST* enviesada à direita R com o algoritmo TRANSFORMAR. Podemos ver que as rotações realizadas pelo algoritmo são feitas sempre em filhos esquerdos e o nó rotacionado passa a fazer parte da espinha direita da árvore após a rotação. Assim, nenhum nó da espinha direita é rotacionado e, portanto, nenhum nó é rotacionado mais do que uma vez. Como existem no máximo $n-1$ nós que não estão na espinha direita de T_1, então T_1 pode ser transformada em R com no máximo $n-1$ rotações. Para cada rotação o ponteiro move-se para o filho esquerdo (nó rotacionado) e, portanto, são realizados no máximo $n-1$ movimentos para o filho esquerdo. Além disso, quando a subárvore esquerda do nó apontado é vazia o ponteiro move-se para o filho direito, se existir. Assim, são realizados exatamente $n-1$ movimentos para o filho direito, pois em R todos os nós possuem filho direito, exceto seu único nó folha. Portanto, a transformação de T_1 em R pode ser feita com no máximo $3n-3$ operações *BST*.

Agora para transformar R em T_2, pela Observação 2.2.1, se aplicarmos o algoritmo TRANSFORMAR com T_2 como entrada, será retornada uma pilha de operações *BST* que transformam R em T_2. Além disso, como as operações *BST* da pilha retornada são inversas às realizadas pelo algoritmo, então esta pilha contem no máximo $3n-3$ operações *BST*. Assim, T_1 pode ser transformada em T_2 com no máximo $6n-6$ operações *BST*. Ainda pela Observação 2.2.1, observe que para cada rotação realizada pelo algoritmo uma rotação é empilhada. Assim, como T_1 é transformada em R com no máximo $n-1$ rotações, R pode ser transformada em T_2 com no máximo $n-1$ rotações. Portanto, T_1 pode ser transformada em T_2 com no máximo $2n-2$ rotações. □

Este resultado de Culik e Wood foi melhorado por Sleator, Tarjan e Thurston para *BSTs* de tamanho pelo menos 11 ($n \geq 11$), para $2n-6$ rotações como descrito no Lema 2.2.2.

Lema 2.2.2 ([23])**.** *Uma* BST T_1*, onde* $|T_1| = n$ *e* $n \geq 11$*, pode ser transformada em uma* BST T_2 *com o mesmo conjunto de nós com no máximo* $2n-6$ *rotações.*

Para ver a equivalência entre os modelos *BST* propostos por Wilber [15] e por Demaine *et al.* [5] (Demaine *et al.* enunciaram este resultado sem prova) fazemos a seguinte observação sobre o custo mínimo de uma busca.

Observação 2.2.2. *Sejam* $\mathcal{A}$ *um algoritmo no modelo* BST *que busca por uma chave* s *em uma* BST T_1*,* T_2 *a* BST *resultante depois da busca por* s *e* Q *o conjunto de nós de* T_1 *visitados por* $\mathcal{A}$ *durante esta busca. O custo de* $\mathcal{A}$ *no modelo de Wilber é pelo menos* $|Q|-1$*, que é a quantidade mínima de movimentos do ponteiro para visitar os nós de* Q*, e o custo de* $\mathcal{A}$ *no modelo de Demaine et al. é* $|Q|$ *por definição.*

Por outro lado, não é necessário fazer muito mais que isto. Como mostramos no Lema 2.2.3, o algoritmo $\mathcal{A}$ pode ser simulado por um algoritmo que realiza $\mathcal{O}(|Q|)$ operações *BST*, o que indica que realizar mais operações que $\mathcal{O}(|Q|)$ seria ineficiente.

Lema 2.2.3. *Existe um algoritmo* $\mathcal{A}'$ *que transforma* T_1 *em* T_2*, busca por* s *e tem custo no máximo* $6|Q|-6$ *no modelo de Wilber.*

Demonstração. Como uma busca inicia na raiz da árvore e as operações *BST* são realizadas entre nós conectados (pai e filho), então Q induz uma subárvore conexa Q_1 de T_1 contendo a raiz de T_1 e uma subárvore conexa Q_2 de T_2 contendo a raiz de T_2. Além disto, $s \in Q$. Agora podemos observar que T_1 pode ser transformada em T_2 com no máximo $6|Q|-6$ operações *BST* ao transformar Q_1 em Q_2, pelo processo descrito no Lema 2.2.1. Portanto, é possível fazer estas operações com custo $\mathcal{O}(|Q|)$. □

Pela Observação 2.2.2 e pelo Lema 2.2.3 podemos concluir que é possível simular o algoritmo $\mathcal{A}$ com custo $\Theta(|Q|)$. Assim, dado o resultado anterior, no restante do texto, sempre que mencionado o modelo *BST*, nos referimos ao modelo *BST* proposto por Demaine *et al.*.

3 ESTUDO DO PROBLEMA

O *problema de busca em BST* é definido da seguinte forma: dada uma *BST* inicial T, realizamos uma sequência de buscas em T no modelo *BST*. Neste capítulo, apresentamos as versões deste problema: a versão estática (Seção 3.1) e a versão dinâmica (Seção 3.2), com suas variações *offline* (Subseção 3.2.1) e *online* (Subseção 3.2.2).

Além disso, neste capítulo também descrevemos a análise competitiva, que é uma maneira de analisar algoritmos fazendo comparações entre algoritmos *online* e *offline* (Subseção 3.2.2), e a conjectura da Otimalidade Dinâmica, que foi apresentada por Sleator e Tarjan [9] e permanece em aberto há mais de 30 anos (Seção 3.3).

3.1 Problema estático

A versão estática do problema tem como objetivo projetar um algoritmo que, dada uma sequência de buscas S, construa uma árvore binária de busca T que minimize o custo total para buscar S em T. Nesta versão do problema, a *BST* construída pelo algoritmo tem sua estrutura estática, isto é, não pode ser reestruturada por rotações durante as buscas.

Podemos observar que o custo mínimo para buscar uma chave s em uma *BST* estática T é visitar apenas os nós do caminho de busca de s, ou seja, este custo é $d_T(s)+1$. Assim, o custo mínimo para realizar uma sequência de buscas $S = (s_1, s_2, \ldots, s_m)$ em T é $\sum_{i=1}^{m}(d_T(s_i)+1) = m+\sum_{i=1}^{m} d_T(s_i)$. Denotamos por $\text{cost}_T(S)$ o custo para realizar uma sequência de buscas S em uma *BST* T. Também podemos observar que quando uma *BST* estática é balanceada, o custo para realizar uma sequência de buscas S de tamanho m é $\mathcal{O}(m \log n)$, pois cada nó possui profundidade $\mathcal{O}(\log n)$.

O menor custo possível para buscar uma sequência S em uma *BST* estática é chamado de *ótimo estático* de S e denotado por $\text{OPT}^{stat}(S)$, ou seja, $\text{OPT}^{stat}(S) = \min_T \text{cost}_T(S)$. Dizemos que uma *BST* estática T é *ótima* para uma sequência de buscas S, se $\text{cost}_T(S) = \text{OPT}^{stat}(S)$. Note que $\text{OPT}^{stat}(S)$ depende apenas da frequência em que as chaves aparecem em S.

Para este problema Gilbert e Moore [24] apresentaram um algoritmo de programação dinâmica de tempo $\mathcal{O}(n^3)$ e espaço $\mathcal{O}(n^2)$. Knuth [8] também apresentou um algoritmo de programação dinâmica capaz de encontrar uma *BST* estática ótima com custo $\mathcal{O}(n^2)$. Este algoritmo de Knuth precisa conhecer a frequência que as chaves aparecem na sequência de buscas. Yao [25] também apresentou um algoritmo de programação dinâmica com custo $\mathcal{O}(n^2)$.

Mehlhorn [26] apresentou um algoritmo de custo $\mathcal{O}(n)$ que, dada uma sequência de buscas S, constrói uma *BST* estática T, tal que $\text{cost}_T(S) = \mathcal{O}(\text{OPT}^{stat}(S))$. Levcopoulos, Lingas e Sack [27] também apresentaram um algoritmo de custo linear que, dada uma sequência de buscas, constrói uma *BST* estática em que o custo para buscar esta sequência nesta árvore está dentro de um fator de $1+\varepsilon$ do custo ótimo, para uma constante $\varepsilon > 0$. Estas árvores de custo $\mathcal{O}(\text{OPT}^{stat}(S))$ são chamadas árvores *quase ótimas* para a sequência de buscas S.

Uma versão deste problema é quando as chaves são armazenadas nas folhas da *BST*. Hu e Tucker [28] propuseram um algoritmo para este problema com tempo $\mathcal{O}(n^2)$ e com espaço $\mathcal{O}(n)$. Mais tarde Garsia e Wachs [29] também propuseram um algoritmo que melhora este tempo para $\mathcal{O}(n \log n)$. Outra versão é quando a altura da árvore é limitada por uma constante L. Para esta versão, Garey [30] propôs um algoritmo de custo $\mathcal{O}(Ln^2)$, onde as chaves são armazenadas nos nós folha. Wessner [31] e Itai [32] também propuseram algoritmos de custos $\mathcal{O}(Ln^2)$, mas com as chaves armazenadas não apenas nas folhas. Becker [33] mostrou que para qualquer Δ fixo uma *BST* estática ótima de altura $h \leq h_{\min}(n) + \Delta$, onde $h_{\min}(n)$ é a altura mínima de uma *BST* com n nós, pode ser construída em tempo $\mathcal{O}(n^2)$. Este algoritmo de Becker melhora os resultados de Wessner e Itai, pois pelos lemas 2.1.1 e 2.1.2 temos que $L \geq \lfloor \log_2 n \rfloor$ e, portanto, os algoritmos de Wessner e Itai têm custo $\mathcal{O}(n^2 \log_2 n)$ para construir uma *BST* estática ótima de altura limitada por $h_{\min}(n) + \Delta$.

3.2 Problema dinâmico

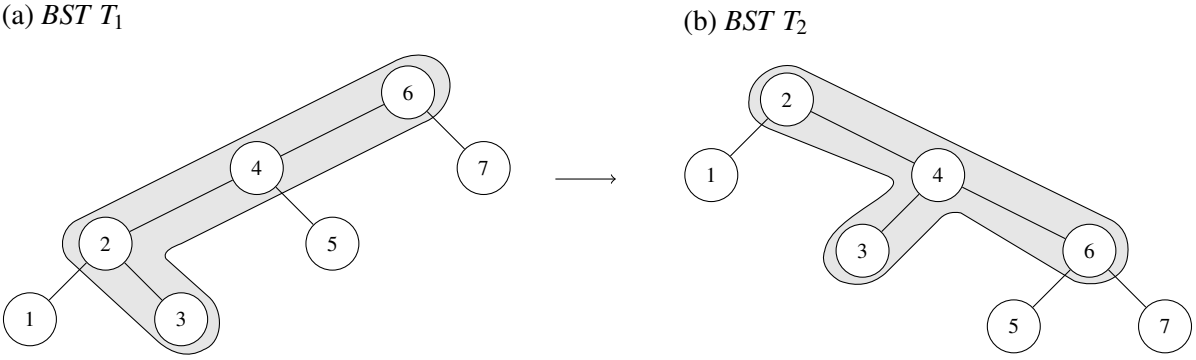
Para a versão do problema dinâmico, o objetivo é executar uma sequência de buscas $S = (s_1, s_2, \ldots, s_m)$ em uma *BST* dada (chamada de *BST inicial*), onde esta árvore pode ser reestruturada por rotações. A instância para este problema é uma sequência de buscas S e uma *BST* T. Este problema pode ser *offline* ou *online*, que é quando a sequência de buscas é conhecida antes de iniciar as buscas e quando as buscas são conhecidas uma a uma, respectivamente.

Como durante uma busca em uma *BST* T_1 podem ser realizadas rotações, ao final da busca uma outra *BST* T_2 resultante da reestruturação feita por estas rotações é obtida. Dizemos que esta reestruturação é feita por um *rearranjo* de T_1 para T_2, como definido abaixo:

Definição 3.2.1 (Rearranjo)**.** *Sejam T_1 e T_2* BSTs *com o mesmo conjunto de nós e Q um subconjunto conexo dos nós de T_1 contendo* raiz(T_1)*. Dizemos que T_1 pode ser rearranjada para T_2, se toda subárvore enraizada em x de T_1, tal que $x \notin Q$, é idêntica a subárvore enraizada em x de T_2.*

Denotamos um rearranjo de T_1 para T_2 por $T_1 \xrightarrow{Q} T_2$. Se Q contem todos os nós de T_1, denotamos simplesmente por $T_1 \to T_2$. O custo desta operação é $|Q|$. Na Figura 3a temos um exemplo de uma *BST* com o conjunto de nós $\{1,\ldots,7\}$ e na Figura 3b uma *BST* resultante do rearranjo $T_1 \xrightarrow{Q} T_2$, onde $Q = \{2,3,4,6\}$ (os nós 4 e 2 são rotacionados, nesta ordem, em T_1 para obter T_2).

Figura 3 – Rearranjo de T_1 para T_2 por $Q \to Q'$



Fonte: Elaborada pelo autor

Um *algoritmo BST* $\mathcal{A}$ é um algoritmo que, dada uma instância (T_0, S) do problema dinâmico, executa a sequência de buscas $S = (s_1, s_2, \ldots, s_m)$ na *BST* T_0. Dizemos que $\mathcal{A}$ executa a instância (T_0, S) por uma *execução* $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$, se todos os rearranjos $T_{i-1} \xrightarrow{Q_i} T_i$ transformam T_{i-1} em T_i e $s_i \in Q_i$ para todo $i \in \{1,\ldots,m\}$. O *custo da execução* E por $\mathcal{A}$ é $\sum_{i=1}^{m} |Q_i|$, denotado por $\text{cost}_{\mathcal{A}}(T_0, S)$.

3.2.1 Problema offline

O problema dinâmico *offline* conhece toda a sequência de buscas dada na instância antes de iniciar as buscas. Assim, por conhecer as próximas buscas, após cada busca o algoritmo *BST* pode reestruturar a árvore na tentativa de reduzir o custo das buscas futuras e, consequentemente, reduzir o custo total da sequência de buscas.

O *custo offline ótimo* para uma sequência de buscas S, denotado por $\text{OPT}(S)$, é definido como $\text{OPT}(S) = \min_{\mathcal{A},T} \text{cost}_{\mathcal{A}}(T, S)$. Ou seja, $\text{OPT}(S)$ é o custo mínimo para executar S dentre todos os algoritmos *BST offline* e todas *BSTs* iniciais com n nós. Podemos observar que dadas duas instâncias (S, T_1) e (S, T_2), onde T_1 e T_2 são *BSTs* com o mesmo conjunto de n nós e S é uma sequência de buscas, temos que $\min_{\mathcal{A}} \text{cost}_{\mathcal{A}}(T_1, S) \leq \min_{\mathcal{A}} \text{cost}_{\mathcal{A}}(T_2, S) + \mathcal{O}(n)$, pois antes de iniciar a execução T_2 pode ser reestruturada com custo linear para ser idêntica a T_1 e, portanto, acrescenta no máximo $\mathcal{O}(n)$ a $\text{cost}_{\mathcal{A}}(T_2, S)$ [22, 23].

Podemos ver no Teorema 3.2.1 que o custo ótimo do problema dinâmico *offline* é no máximo o ótimo estático.

Teorema 3.2.1 ([21]). *Para toda sequência de buscas S, temos que* $\mathrm{OPT}(S) \leq \mathrm{OPT}^{stat}(S)$.

Demonstração. No problema dinâmico podemos simular a execução de uma sequência de buscas S em uma *BST* T_0 do problema estático. Definimos uma execução $E = \langle T_0, Q_1, T_1, \ldots, Q_m, T_m \rangle$ para o problema dinâmico, onde cada Q_i $(1 \leq i \leq m)$ contem apenas os nós do caminho de busca de s_i e cada T_i é idêntica a T_0. □

Para este problema, Lucas [10] propôs um algoritmo *BST offline* guloso (Demaine *et al.* [5] chamaram este algoritmo de *Greedy Future*) que segue dois princípios: (1) visitar apenas os nós do caminho de busca da chave buscada; e (2) rearranjar o caminho de busca movendo a subárvore contendo a próxima chave a ser buscada o mais próximo possível da raiz e aplicando recursivamente estes rearranjos para as buscas seguintes.

O *Greedy Future* executa uma instância (T_0, S), onde $S = (s_1, s_2, \ldots, s_m)$, por uma execução $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$, onde cada Q_i contem apenas os nós do caminho de busca por s_i em T_{i-1}. Seja S' a subsequência $(s_{i+1}, \ldots, s_m)$ e seja τ_i a subárvore induzida por Q_i em T_{i-1}. Definimos uma *BST* τ_i' sobre os nós de Q_i como a descreveremos seguir, e construímos T_i ao fazer o rearranjo $\tau_i \to \tau_i'$. τ_i' é definida recursivamente, a partir de Q_i e S', como segue: se $S' = \emptyset$, então escolha τ_i' como uma *BST* qualquer com os nós de Q_i; se $S' \neq \emptyset$, seja s o primeiro elemento de S', e defina $R = \{s\}$ se $s \in Q_i$; caso contrário, R contem precisamente o predecessor e o sucessor de s em Q_i, se existirem. Seja S'_+ a subsequência de S' restrita aos valores estritamente maiores que os valores de R e seja S'_- a subsequência de S' restrita aos valores estritamente menores que os valores de R. Como R possui no máximo dois nós, então $\mathrm{raiz}(\tau_i')$ é o nó de menor valor em R, e se R possui um segundo nó, então $\mathrm{dir}(\mathrm{raiz}(\tau_i'))$ é este nó. Em seguida, as subárvores esquerda e direita dos nós fixados em τ_i', respectivamente, são definidas repetindo este processo recursivamente para o subconjunto de nós $Q'_+ = \{x \mid x \in Q_i \text{ e } x > \max(R)\}$ com a subsequência S'_+ e para o subconjunto de nós $Q'_- = \{x \mid x \in Q_i \text{ e } x < \min(R)\}$ com a subsequência S'_-. Note que, no caso em que R possui dois nós, o filho esquerdo de $\mathrm{dir}(\mathrm{raiz}(\tau_i'))$ não está em Q_i.

Os principais aspectos gulosos do *Greedy Future* é visitar apenas os nós do caminho da chave buscada e a forma em que o caminho de busca é rearranjado, movendo as próximas chaves a serem buscadas o mais próximo possível da raiz. Um algoritmo semelhante ao de Lucas foi proposto por Munro [11] independentemente mais de uma década depois.

3.2.2 *Problema* online

A versão *online* do problema dinâmico recebe uma busca por vez da sequência de buscas da instância. Pelo fato de que o algoritmo não conhece as buscas seguintes a uma busca s_i de uma sequência de buscas $S = (s_1, \ldots, s_m)$, podemos observar que, nesta versão, as rotações durante esta busca não dependem das buscas seguintes $(s_{i+1}, \ldots, s_m)$.

Uma forma de analisar um algoritmo *BST online* é através da *análise competitiva*, onde o custo de um algoritmo *BST online* é comparando com o valor ótimo *offline*. O termo análise competitiva foi apresentado primeiramente por Karlin *et al.* [34], mas Sleator e Tarjan já haviam comparado algoritmos *offline* e *online* antes [9, 35]. Dizemos que um algoritmo *BST online* $\mathcal{A}$ é $f(n)$-competitivo se seu custo para qualquer sequência de buscas S é limitado por uma função $f(n)$ do valor ótimo *offline* para aquela sequência, ou seja, $\text{cost}_{\mathcal{A}}(T,S) \leq f(n)\,\text{OPT}(S) + O(n)$. Chamamos $f(n)$ de *fator competitivo*.

Para este problema, Allen e Munro [36] apresentaram um algoritmo simples chamado *Simple Exchange* (também chamado de *Rotate Once*, por Kozma [21]). Este algoritmo visita os nós do caminho de busca da chave buscada s e rotaciona s uma única vez, se seu nó pai(s) existir. O *Simple Exchange* é baseado na heurística *Transposition* proposta por Rivest [37] para listas lineares. Allen e Munro mostraram que o custo deste algoritmo é $\Theta(\sqrt{n})$, se as chaves são buscadas com probabilidades iguais e independentes.

Allen e Munro também apresentaram o algoritmo *Move to Root*, onde a chave buscada é rotacionada até a raiz da árvore. A chave buscada é rotacionada enquanto possuir nó pai, ou seja, enquanto não é a raiz da árvore. Este algoritmo também é baseado em uma heurística para listas lineares, a *Move to Front* [37, 38]. Allen e Munro mostraram que o custo do seu algoritmo é no máximo $2\log 2$ vezes o custo ótimo estático, quando os acessos possuem distribuição de frequências independentes.

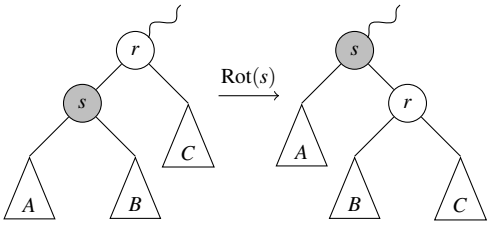
Sleator e Tarjan [1] apresentaram a Árvore *Splay* que é uma *BST* e, assim como o *Move to Root*, move a chave buscada s para a raiz da árvore, mas de forma diferente. Enquanto no *Move to Root* as rotações são realizadas sempre em s, a Árvore *Splay* também rotaciona o nó pai(s), se ou s ou pai(s) não é a raiz da árvore. Em uma busca por uma chave s em uma Árvore *Splay* T, os nós do caminho de busca de s são visitados e o algoritmo SPLAY, descrito abaixo, é executado. As operações (1), (2) e (3) destacadas no algoritmo SPLAY são ilustradas nas figuras 4a, 4b e 4c, respectivamente, onde a chave buscada sempre é s. Sleator e Tarjan denominaram estas operações por *Zig*, *Zig-Zig* e *Zig-Zag*, respectivamente.

Algoritmo 3: SPLAY

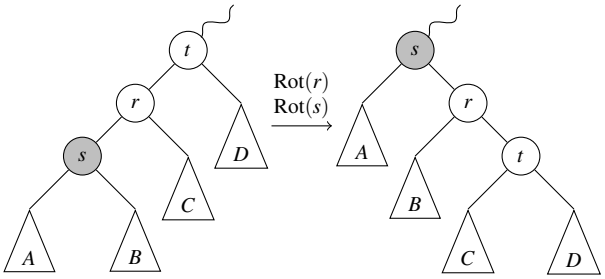
```
   Entrada: BST T e chave buscada s de T
1  início
2      enquanto s ≠ raiz(T) faça
3          se pai(s) = raiz(T) então                                          (1)
4              Rot(s)
5          senão
6              se pai(s) e s são ou ambos filhos esquerdos ou ambos filhos direitos então (2)
7                  Rot(pai(s))
8                  Rot(s)
9              senão                                                          (3)
10                 Rot(s)
11                 Rot(s)
12             fim
13         fim
14     fim
15 fim
```

Figura 4 – Operações *Splay*

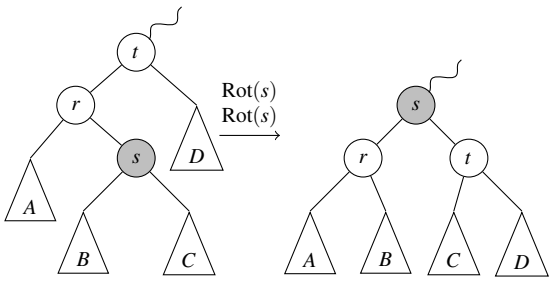
(a) *Zig*



(b) *Zig-Zig*



(c) *Zig-Zag*



Fonte: Elaborada pelo autor

Sleator e Tarjan [9] também propuseram uma variação da Árvore *Splay*, a Árvore *Semi-Splay*, onde a operação *Zig-Zig* é modificada. Os autores mostraram que a Árvore *Semi-Splay* dá um fator constante menor do que a Árvore *Splay* para algumas sequências de buscas.

Subramanian [39] apresentou um algoritmo, sugerido por Daniel Sleator em comunicação pessoal, como um problema em aberto, onde apenas os nós do caminho de busca da chave buscada são visitados e a subárvore correspondente a este caminho é balanceada.

Balasubramanian e Raman [40] mostraram que este algoritmo tem complexidade amortizada $\mathcal{O}\left(\frac{\log n \log\log n}{\log\log\log n}\right)$. Mais recentemente esta complexidade foi melhorada por Dorfman *et al.* [41] para $\mathcal{O}(\log n 2^{\log^* n}(\log^* n)^2)$, onde $\log^*(x)$ denota a função de logaritmo iterado de x (esta função cresce muito lentamente).

Demaine *et al.* [2] propuseram a Árvore *Tango*, que é uma *BST online*. A Árvore Tango é $\mathcal{O}(\log\log n)$-competitiva. Esta foi a primeira *BST* a atingir este fator competitivo. Quando a Árvore Tango foi introduzida, o custo de uma busca no pior caso era $\mathcal{O}(\log n \log\log n)$, mas Demaine *et al.* [42] apresentaram uma modificação que melhora este custo para $\mathcal{O}(\log n)$.

Outras *BSTs online* semelhantes a Árvore Tango e de mesmo fator competitivo foram propostas, a *Multi-Splay Tree* e a *Chain-Splay Tree* propostas por Wang, Derryberry e Sleator [3, 12] e Georgakopoulos [4], respectivamente. Ambas utilizam ideias semelhantes à utilizada na Árvore Tango. Além destas, outras *BSTs online* de mesmo fator competitivo também foram propostas, a *Skip-Splay Tree* e a *Zipper Tree*, propostos por Derryberry e Sleator [43] e Bose *et al.* [44], respectivamente.

3.3 A conjectura da Otimalidade Dinâmica

Um algoritmo *BST online* que é $\mathcal{O}(1)$-competitivo (ou *contante-competitivo*) é dito *dinamicamente ótimo*. Sleator e Tarjan [1] quando propuseram a Árvore *Splay*, conjecturaram que sua árvore é dinamicamente ótima. Lucas [10] e Munro [11] conjecturaram que o *Greedy Future* também é dinamicamente ótimo. Apesar de décadas de pesquisa, ainda não foi provado nem que a Árvore *Splay* nem o *Greedy Future* ou qualquer outro algoritmo é dinamicamente ótimo. A competitividade da Árvore Tango foi um dos principais avanços na tentativa de provar a otimalidade dinâmica, pois até então o melhor fator competitivo conhecido era $\log n$.

Kozma [21] apresentou três perguntas relacionadas à conjectura da otimalidade dinâmica: (1) "Qual a melhor sequência de rearranjos para realizar uma determinada sequência de buscas?". Ainda não foi apresentado um algoritmo eficiente para resolver esse problema. (2) "Existe uma lacuna significativa entre os algoritmos *online* e *offline*?". Podemos imaginar que algoritmos *offline* são mais poderosos, pois pode se preparar com antecedência para toda a sequência de buscas. No entanto, a conjectura da otimalidade dinâmica sugere que esse conhecimento do futuro não dá vantagens para o algoritmo *offline*. (3) "Algum algoritmo *online* se comporta bem para qualquer sequência de buscas?".

4 VISÃO GEOMÉTRICA

Para visualizar uma execução *BST* de forma mais fácil, Demaine *et al.* [5] apresentaram uma representação de uma execução através de um conjunto de pontos no plano. Os autores denominaram esta representação por *Visão Geométrica*. Demaine *et al.* definiram o problema do superconjunto de pontos arboreamente satisfeito (*Arborally Satisfied Superset* - ArbSS, do inglês) e mostram que este problema equivale ao problema dinâmico de busca em *BST*.

Neste capítulo apresentamos a definição da visão geométrica de Demaine *et al.* (Seção 4.1) e mostramos a equivalência entre um conjunto de pontos arboreamente satisfeito e uma execução *BST* (Seção 4.2). Além disto, descrevemos o problema ArbSS (Seção 4.3) e o algoritmo *online* proposto por Demaine *et al.* para este problema. Derryberry, Sleator e Wang [14] propuseram uma representação no plano semelhante a de Demaine *et al.* independentemente.

4.1 Definição

Na visão geométrica definida por Demaine *et al.* [5], um *ponto a* se refere a um ponto em $2D$ com coordenadas inteiras (a_x, a_y) tal que $1 \leq a_x \leq n$ e $1 \leq a_y \leq m$. Denotamos por $\square ab$ o *retângulo* alinhado aos eixos definido pelos pontos a e b não alinhados horizontalmente ou verticalmente, isto é,

$$\square ab = \{(x,y) \mid \min\{a_x, b_x\} \leq x \leq \max\{a_x, b_x\} \text{ e } \min\{a_y, b_y\} \leq y \leq \max\{a_y, b_y\}\}.$$

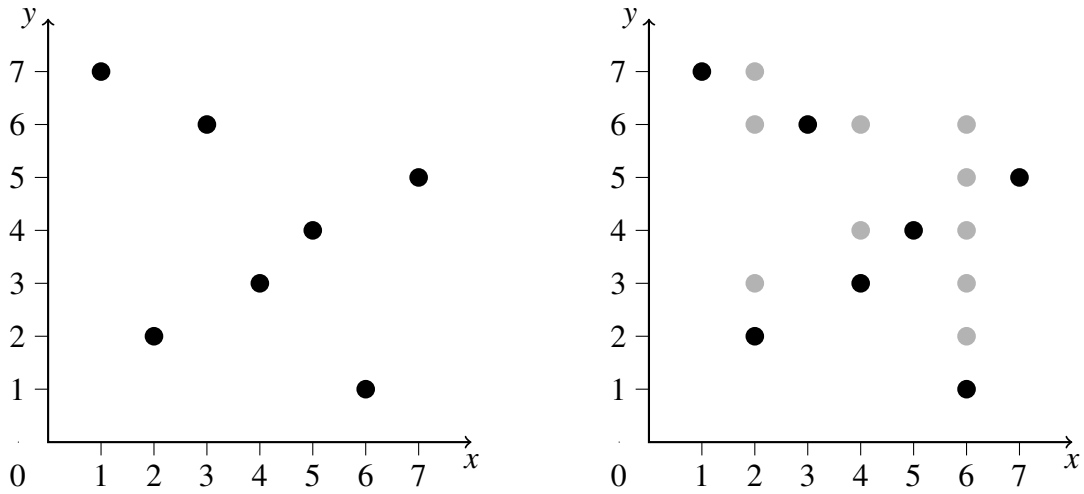
Dizemos que um ponto c está nos *lados incidentes* de a em $\square ab$, se c está no segmento de reta de $(a_x, \min(a_y, b_y))$ até $(a_x, \max(a_y, b_y))$ ou no segmento de reta de $(\min(a_x, b_x), a_y)$ até $(\max(a_x, b_x), a_y)$ e $c \neq a$. Dizemos que dois pontos $a, b \in P$ são *arboreamente satisfeitos* com relação a um conjunto de pontos P se, (1) a e b são ortogonalmente colineares (são alinhados horizontalmente ou verticalmente); ou (2) existe pelo menos um ponto $c \in P \backslash \{a, b\}$ tal que $c \in \square ab$. Um conjunto de pontos P é dito arboreamente satisfeito se todos os pares de pontos em P são arboreamente satisfeitos.

Na visão geométrica nos referimos às coordenadas x e y no plano pelas as chaves contidas na *BST* $(1, \ldots, n)$ e o tempo que a chave é visitada $(1, \ldots, m)$, respectivamente. A visão geométrica de uma sequência de buscas S é o conjunto de pontos $P(S) = \{(s_1, 1), \ldots, (s_m, m)\}$, e a visão geométrica de uma execução *BST* $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$ é o conjunto de pontos $P(E) = \{(x,i) \mid x \in Q_i\}$, ou seja, na visão geométrica de uma execução *BST* mapeamos para a linha i (tempo i) os nós visitados em T_{i-1}.

Na Figura 5a temos um exemplo da visão geométrica da sequência de buscas $S = \{6,2,4,5,7,3,1\}$, e na Figura 5b temos um conjunto de pontos P arboreamente satisfeito tal que $P(S) \subseteq P$.

Figura 5 – Visão geométrica

(a) Visão geométrica de $S = \{6,2,4,5,7,3,1\}$ (b) Conjunto arboreamente satisfeito P



Fonte: Elaborada pelo autor

Lema 4.1.1. *Em um conjunto de pontos P arboreamente satisfeito, para quaisquer dois pontos $a,b \in P$ não ortogonalmente colineares, existe pelo menos um ponto de $P\backslash\{a,b\}$ nos lados de $\square ab$ incidentes a a, e existe pelo menos um ponto nos lados incidentes a b. Estes pontos não são necessariamente distintos.*

Demonstração. Suponha que o resultado é falso e seja $\square ab$ um retângulo que não satisfaz o Lema e de área mínima. Além disto, por simetria, assuma que não há um ponto de $P\backslash\{a,b\}$ nos lados de $\square ab$ incidentes a a. Como P é arboreamente satisfeito, existe um ponto $c \in P\backslash\{a,b\}$ em $\square ab$, e como $\square ab$ é um retângulo de área mínima que não possui pontos nos lados incidentes a a, no retângulo $\square ac$ existe pelo menos um ponto nos lados incidentes a a. Logo, chegamos a uma contradição. □

4.2 Equivalência entre execução *BST* e conjunto arboreamente satisfeito

A equivalência entre uma execução *BST* e um conjunto arboreamente satisfeito foi apresentada por Demaine *et al.* com uma ideia da prova, mas sem os devidos detalhes. Esta equivalência é apresentada nos lemas 4.2.1 e 4.2.2.

Lema 4.2.1. *O conjunto de pontos $P(E)$ para qualquer execução* BST *E é arboreamente satisfeito.*

Demonstração. Seja $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$ e sejam nós $x \in Q_i$ e $y \in Q_j$. Sem perda de generalidade, assumimos que $x < y$ e $i < j$. Se $\mathrm{LCA}_{T_{i-1}}(x,y) \neq x$, então para visitar x é necessário visitar $\mathrm{LCA}_{T_{i-1}}(x,y)$ no tempo i, portanto, $\mathrm{LCA}_{T_{i-1}}(x,y) \in Q_i$. Além disso, como $x < \mathrm{LCA}_{T_{i-1}}(x,y) \leq y$, então o ponto $(\mathrm{LCA}_{T_{i-1}}(x,y), i)$ satisfaz arboreamente $\square(x,i)(y,j)$. Similarmente, se $\mathrm{LCA}_{T_{j-1}}(x,y) \neq y$, então é necessário visitar $\mathrm{LCA}_{T_{j-1}}(x,y)$ para visitar y, assim $\mathrm{LCA}_{T_{j-1}}(x,y) \in Q_j$, e como $x \leq \mathrm{LCA}_{T_{j-1}}(x,y) < y$, o ponto $(\mathrm{LCA}_{T_{j-1}}(x,y), j)$ satisfaz arboreamente $\square(x,i)(y,j)$. Resta agora analisar o caso em que $\mathrm{LCA}_{T_{i-1}}(x,y) = x$ e $\mathrm{LCA}_{T_{j-1}}(x,y) = y$. Se $y \in Q_k$, onde $i \leq k < j$, o ponto (y,k) satisfaz $\square(x,i)(y,j)$. Mostramos que a situação que resta é impossível, pois, como $y \notin Q_{j-1}$, a subárvore enraizada em y de T_{j-2} não é modificada e x não está na subárvore enraizada em y de T_{j-1}. Assim, chegamos a uma contradição, pois $\mathrm{LCA}_{T_{j-1}}(x,y) = y$, mas x não é descendente de y em T_{j-1}. □

Antes de mostrar o outro lado desta equivalência, precisamos de algumas definições para construir a execução *BST* corresponde a visão geométrica. A *propriedade heap* para uma árvore binária é definida da seguinte forma: se $x = \mathrm{pai}(y)$, então $\mathrm{chave}(x) \leq \mathrm{chave}(y)$ em uma *heap* mínima, e analogamente, se $x = \mathrm{pai}(y)$, então $\mathrm{chave}(x) \geq \mathrm{chave}(y)$ em uma *heap* máxima.

Uma *treap* é uma estrutura de dados que armazena pares (x,y) em cada nó de uma árvore binária T de tal forma que T é uma árvore binária de busca com a propriedade de ordenação por x e ordenado pela propriedade *heap* binária por y. Uma *treap de mínimo* é uma *treap*, onde a *heap* binária é de mínimo, e uma *treap de máximo* é uma *treap*, onde a *heap* binária é de máximo.

Lema 4.2.2. *Para qualquer conjunto de pontos arboreamente satisfeito P, existe uma execução* BST *E com $P(E) = P$.*

Demonstração. Seja o *próximo tempo de acesso* de a ($1 \leq a \leq n$) no tempo i, denotado por $\mu(a,i)$, como a menor coordenada y de um ponto qualquer em P na semirreta de (a,i) até (a,∞). Se tal ponto não existe, então $\mu(a,i) = \infty$. Seja T_0 uma *treap* de mínimo, onde o nó a corresponde ao par $(a, \mu(x,1))$. Vamos construir uma execução $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$, onde T_{i-1} é uma *treap* de mínimo e seus nós são definidos pelos pares $(a, \mu(a,i))$ e Q_i contém todos os pontos de P com coordenada $y = i$. Observe que os nós de Q_i induzem uma subárvore de T_{i-1} contendo a raiz, pois i é o valor mínimo para $\mu(a,i)$, com $a \in \{1, \ldots, n\}$, critério de ordenação

da *heap* de mínimo em T_{i-1}. T_i é obtida pelo rearranjo $T_{i-1} \xrightarrow{Q_i} T_i$, onde os nós de Q_i em T_{i-1} são rearranjados baseado no próximo tempo de acesso, ou seja, Q_i é induz uma *treap* de mínimo em T_i sobre os nós $\{(a, \mu(a, i+1)) \mid a \in Q_i\}$.

O rearranjo $T_{i-1} \xrightarrow{Q_i} T_i$ mantem a propriedade de ordenação da *BST* e, portanto, vamos mostrar apenas que a propriedade *heap* é mantida em T_i analisando todos os nós a em T_i. Se a e $\text{pai}(a)$ estão em Q_i a propriedade é mantida por construção, e se a e $\text{pai}(a)$ não estão em Q_i a propriedade também é mantida, pois $\mu(x,i) = \mu(x,i+1)$, para todo x com $\mu(x,i) \neq i$. Agora se $\text{pai}(a) \in Q_i$ e $a \notin Q_i$ e se $\mu(a,i+1) < \mu(\text{pai}(a),i+1)$, o Lema 4.1.1 é violado, pois não existe um ponto nos lados de $\square(\text{pai}(a),i)(a,\mu(a,i))$ incidente a $(\text{pai}(a),i)$. No lado vertical não existe pontos, pois assumimos que $\mu(\text{pai}(a),i+1) > \mu(a,i+1)$, nem no lado horizontal, pois se existe um ponto (c,i), temos que $a < c < \text{pai}(a)$ e, portanto, c é descendente de a, mas como $a \notin Q_i$, então $c \notin Q_i$. □

4.3 O problema do superconjunto arboreamente satisfeito (*ArbSS*)

O *problema do superconjunto arboreamente satisfeito* (ArbSS) é encontrar um superconjunto arboreamente satisfeito de um conjunto de pontos P dado. Denotamos por $minASS(S)$ o tamanho do menor superconjunto arboreamente satisfeito de $P(S)$. Assim, pela equivalência entre uma execução *BST* e um conjunto arboreamente satisfeito mostrada na seção anterior (Seção 4.2), temos que $\text{OPT}(S) = minASS(S)$.

A versão *online* deste problema *(ArbSS online)* é projetar um algoritmo que receba um conjunto de pontos $(s_1,1),(s_2,2),\ldots,(s_m,m)$, nesta ordem, e depois de receber o ponto (s_i,i), o algoritmo deve produzir um conjunto de pontos P_i na linha i (coordenada $y = i$) tal que $\{(s_1,1),(s_2,2),\ldots,(s_i,i)\} \cup P_1 \cup P_2 \cup \ldots \cup P_i$ seja arboreamente satisfeito.

A partir deste problema, Demaine *et al.* mostraram que para qualquer algoritmo ArbSS *online* existe um algoritmo *BST online*. Para construir este algoritmo *BST online* é necessário a definição de *árvore split*. Uma árvore *split* é uma estrutura de dados que implementa duas operações no modelo *BST*: $MakeTree(x_1,x_2,\ldots,x_n)$, que constrói uma Árvore *Split* com os nós $x_1,x_2,\ldots,x_n$, onde estes nós são conexos, através de um rearranjo, e $Split(x)$, que move x para a raiz da Árvore *Split* deixando as suas subárvores esquerda e direita como árvores *split*.

Lema 4.3.1 ([5])**.** *Existe uma estrutura de dados de árvore* split *que suporta* MakeTree *e qualquer sequência de n* Splits *em tempo* $\mathcal{O}(n)$.

As árvores *split* podem mover um nó para a raiz e, em seguida, excluir este nó deixando duas árvores *split* com custo $\mathcal{O}(1)$ amortizado. Usando árvores *split*, no Lema 4.3.2 é mostrada a equivalência entre algoritmos ArbSS *online* e *BST online*. Este resultado é enunciado por Demaine *et al.* [5] com uma ideia da prova, mas sem os devidos detalhes.

Precisamos de algumas definições para o Lema 4.3.2. Uma *general heap* é uma árvore com a propriedade *heap*, onde qualquer empate entre um conjunto de nós conexos resulta em um *supernó*. Uma *general treap* é uma estrutura de dados que armazena pares (x,y) em uma árvore binária de busca de tal forma que é uma árvore binária de busca com a propriedade de ordenação por x e ordenado pela propriedade *general heap* binária por y. Uma *general treap de mínimo* é uma *general treap*, onde a *general heap* para o valor de y é de mínimo. E uma *general treap de máximo* é uma *general treap*, onde a *general heap* para o valor de y é de máximo.

Lema 4.3.2. *Para qualquer algoritmo* ArbSS online $\mathcal{A}$*, existe um algoritmo* BST *online* $\mathcal{A}'$*, tal que em qualquer sequência de buscas, o custo de* $\mathcal{A}'$ *é limitado por um fator constante do custo de* $\mathcal{A}$*.*

Demonstração. Defina o *último tempo de acesso* de a $(1 \leq a \leq n)$ no tempo i, denotado por $\sigma(a,i)$, como a maior coordenada y de um ponto do algoritmo $\mathcal{A}$ na semirreta de (a,i) até $(a,-\infty)$. Se tal ponto não existe, então $\sigma(a,i) = -\infty$. Vamos construir uma execução $E = \langle G_0, Q_1, G_1, Q_2, G_2, \ldots, Q_m, G_m \rangle$, onde G_i é uma *general treap* de máximo definida por todos os pontos $(a, \sigma(a,i))$. Podemos observar que G_0 possui um único supernó com todas as chaves, pois $\sigma(a,0) = -\infty$ para todo a.

Os rearranjos de $G_{i-1} \xrightarrow{Q_i} G_i$, onde Q_i contem os nós correspondentes aos pontos na linha i no algoritmo $\mathcal{A}$, ocorrem com todos os nós de Q_i sendo visitados em G_{i-1} e movidos para um supernó na raiz de G_i, pois i é o maior valor para $\sigma(a,i)$. Quando um nó a é visitado em G_{i-1}, seu predecessor p e seu sucessor s em seu (super)nó pai também devem ser visitados, se existirem. Isto corresponde a satisfazer os retângulos $\square((p,\sigma(p,i)),(a,i))$ e $\square((s,\sigma(s,i))(a,i))$. Pelo Lema 4.1.1 para um retângulo ser satisfeito é necessário ter pelo menos um ponto nos lados incidentes a cada ponto que define este retângulo. Para $\square((p,\sigma(p,i))(a,i))$, suponha que existe um ponto $(b,\sigma(p,i))$ no lado horizontal incidente a $(p,\sigma(p,i))$. Se $\sigma(b,i) = \sigma(p,i)$, temos uma contradição, pois $p < b < a$ e p não é o predecessor de a no supernó pai de a. Agora, se $\sigma(b,i) > \sigma(p,i)$, pela propriedade de ordenação *BST*, também temos uma contradição, pois, como b é ancestral de p e p é ancestral de a, então a está na subárvore esquerda de b, mas $a > b$. Logo, não existe um ponto no lado horizontal incidente a $(p,\sigma(p,i))$ em $\square((p,\sigma(p,i)),(a,i))$. Assim,

resta apenas o ponto (p,i) para satisfazer o Lema 4.1.1. Portanto, sempre que um nó a é visitado, seu predecessor em seu (super)nó pai também é visitado. O retângulo $\square((s,\sigma(s,i)),(a,i))$ é simétrico ao retângulo $\square((p,\sigma(p,i)),(a,i))$. Assim, para cada nó visitado, pelo menos um nó em seu nó pai também é visitado. Logo, todos os nós visitados são conexos e podemos movê-los para a raiz de G_i. Primeiro movemos os nós visitados que estão na raiz G_{i-1} para um novo supernó raiz e depois, enquanto houver algum nó visitado em algum nó filho da raiz, os movemos para o supernó raiz. Desta forma, temos um algoritmo *BST online*, pois a execução é baseada apenas em buscas passadas.

Agora para mostrar que o custo do algoritmo construído é limitado por um fator constante do custo de algum algoritmo ArbSS *online*, em cada supernó de G_i as chaves são armazenadas em uma árvore *split*. Para cada chave a visitada, é aplicado um $Split(a)$ transformando-os em um nó com apenas uma chave. Agora, todos os nós simples são transformados em uma árvore *split* correspondente ao supernó raiz aplicando um *MakeTree*. Pelo Lema 4.3.1 cada *Split* custa tempo $\mathcal{O}(1)$ amortizado e cada *MakeTree* custa $\mathcal{O}(n)$, o custo de simular as buscas no tempo i é uma constante vezes o número de pontos nessa linha. □

Demaine *et al.* propuseram o algoritmo guloso e *online GreedyASS* para o problema ArbSS *online*. O algoritmo varre o conjunto de pontos com uma linha horizontal iniciando na coordenada y menor para a maior (de 1 até m). Em cada linha i o algoritmo adiciona a quantidade mínima de pontos para que o conjunto de pontos, onde a coordenada $y \leq i$, seja arboreamente satisfeito. Na Figura 5b temos o superconjunto resultante do *GreedyASS* para o conjunto de pontos representados na Figura 5a, onde os pontos cinzas representam os pontos adicionados pelo *GreedyASS*.

A decisão gulosa de visitar apenas no caminho da busca do *Greedy Future* é equivalente a adicionar o número mínimo de pontos na linha de varredura no *GreedyASS*. No Lema 4.3.3 mostramos esta equivalência.

Lema 4.3.3. *Um nó $y \in Q_i$ de uma execução $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$ do* Greedy Future *equivale ao ponto (y,i) no superconjunto arboreamente satisfeito gerado pelo* GreedyASS.

Demonstração. Suponha que até o tempo $i-1$ cada nó visitado pelo *Greedy Future* corresponde a um ponto no *GreedyASS*. Para ambos os lados da prova, podemos observar que, se $y = x_i$, é válido, pois no *Greedy Future* a chave buscada x_i deve pertencer a Q_i e o ponto (x_i,i) faz parte da entrada do *GreedyASS*. Portanto, assuma sem perda de generalidade que $y < x_i$.

Para mostrar que um nó y visitado pelo *Greedy Future* no tempo i corresponde ao ponto (y,i) no *GreedyASS*, suponha que o *Greedy Future* visitou o nó y na busca por x_i e o *GreedyASS* não adicionou o ponto (y,i). Seja $j = \sigma(y,i)$. Como o ponto (y,i) não foi adicionado, então existe um ponto (z,k), onde $y < z \leq x_i$ e $j \leq k < i$, que satisfaz o retângulo $\square((y,j)(x_i,i))$. Seja (z,k) o ponto com maior valor possível para z e maior valor possível para k. Podemos observar que z é ancestral de x_i em T_{k-1}, pois se $\mathrm{LCA}_{T_{k-1}}(z,x_i) \neq z$ temos que $\mathrm{LCA}_{T_{k-1}}(z,x_i) > z$ e z não foi escolhido como maior valor possível. Também podemos observar que y é ancestral de x_i em T_j, pois se $\mathrm{LCA}_{T_j}(y,x_i) \neq y$, então, pela forma que o *Greedy Future* rearranja o caminho de busca, $\mathrm{LCA}_{T_{i-1}}(y,x_i) = x_i$ e y não seria visitado no tempo i. Portanto, como y não foi visitado no intervalo de tempo (j,i), seus descendentes não mudam e, assim, pela propriedade de ordenação, temos que y também é ancestral de z. Então y é visitado no tempo k na busca por z e consequentemente temos que $k = j$. Por y ser ancestral de z em T_j, temos que o próximo acesso de y é menor do que o próximo acesso de z e, portanto, o próximo acesso de y é menor do que i. Assim, y está no caminho de busca por x_t, onde $j < t < i$. Isto contradiz que $j = \sigma(y,i)$.

Para o outro lado da prova, suponha que o *GreedyASS* adicionou o ponto (y,i) e o nó y não está no caminho de busca por x_i. Seja (y,j), onde $-\infty \leq j < i$, o ponto com o maior valor para j. Observe que se $j = -\infty$, o ponto (y,i) não seria adicionado pelo *GreedyASS*, pois $\square((y,j),(x_i,i))$ não é um retângulo. Logo, temos que $0 < j < i$. Pelo fato do ponto (y,i) ter sido adicionado, não existe um ponto (z,k), onde $y < z \leq x_i$ e $j \leq k < i$, no retângulo $\square((y,j)(x_i,i))$. Podemos observar que, y é ancestral de x_i em T_j. Observe que, se $\mathrm{LCA}_{T_{j-1}}(y,x_i) \neq y$, o ponto $(\mathrm{LCA}_{T_{j-1}}(y,x_i),j)$ satisfazia o retângulo $\square((y,j),(x_i,i))$ e o ponto (y,i) não seria adicionado. Portanto, $\mathrm{LCA}_{T_{j-1}}(y,x_i) = y$. Como y não foi visitado no intervalo de tempo (j,i), seus descendentes não mudam neste intervalo de tempo e então y é ancestral de x_i em T_{i-1}. Portanto, y está no caminho de busca por x_i no *Greedy Future* contradizendo que y não está no caminho de busca por x_i. □

Este resultado apresentado no Lema 4.3.3, juntamente com o resultado do Lema 4.3.2, é surpreendente, pois Lucas [10] e Munro [11] apresentaram o *Greedy Future*, independentemente, como um algoritmo *BST offline*, mas Demaine *et al.* [5] mostraram que o *Greedy Future* pode ser transformado em um algoritmo *BST online*. Esta transformação é feita quando olhamos para o *Greedy Future* na visão geométrica como um problema ArbSS *online* e aplicamos o Lema 4.3.2.

Esta correspondência entre o *Greedy Future* e o *GreedyASS* no lema anterior é válida quando a *BST* inicial T_0 no *Greedy Future* é uma *treap* de mínimo sobre o conjunto de nós $\{(x, \mu(x,0))\}$, onde $x \in \{1,2,\ldots,n\}$. Para manter a correspondência para uma *BST* inicial T_0 qualquer e uma sequência de buscas S podemos fazer um pré-processamento adicionando o seguinte conjuntos de pontos a $P(S)$: $P_{T_0} = \{(x, -d_{T_0}(x)), (x, -d_{T_0}(x)-1), \ldots, (x, -n+1)\}$ para todo $x \in \{1,2,\ldots,n\}$. O conjunto de pontos P_{T_0} já é arboreamente satisfeito e as coordenadas y de seus pontos possuem valor no máximo 0. Assim, o conjunto de pontos gerados pelo *GreedyASS* para $P(S) \cup P_{T_0}$ com coordenada $y \geq 1$ é a visão geométrica da execução do *Greedy Future* para a instância (T_0, S).

O *GreedyASS* é um algoritmo ArbSS *online*, porque suas decisões dependem apenas do passado (pontos em coordenadas y menores). Assim, pelo Lema 4.3.2, o *GreedyASS* pode ser transformado em um algoritmo *BST online* com o mesmo custo assintótico. Isso sugere que um algoritmo *offline* pode não ser assintoticamente melhor do que um algoritmo *online* no modelo *BST*. Ou seja, a otimalidade dinâmica pode ser atingida.

5 LIMITANTES

Quando há dificuldade para resolver um problema, alguns limitantes são mensurados para delimitar a distância entre o custo de um algoritmo e o valor da solução ótima. Neste capítulo apresentamos limitantes superiores (Seção 5.1) e limitantes inferiores (Seção 5.2) para $\mathrm{OPT}(S)$ propostos na literatura.

5.1 Limitantes superiores

Um limitante superior trivial para $\mathrm{OPT}(S)$ é o custo de qualquer algoritmo *BST* para executar a sequência de buscas S, pois qualquer algoritmo *BST* tem custo maior ou igual ao valor $\mathrm{OPT}(S)$. Analisando um algoritmo *BST* é possível descrever alguns limitantes. Quando Sleator e Tarjan [9] propuseram a Árvore *Splay*, eles mostram alguns limitantes para a Árvore *Splay* com o conjunto de chaves $\{1,\ldots,n\}$: o *Static Optimality Bound*, onde para todas as sequências de buscas suficientemente longas $S = \{s_1,\ldots,s_m\}$, o custo total para buscar por S em uma Árvore *Splay* é no máximo um fator constante do custo para buscar por S em uma *BST* estática ótima para S, ou seja, o custo para buscar por S em uma Árvore *Splay* é $\mathcal{O}(\mathrm{OPT}^{stat}(S))$; o *Static Finger Bound*, onde para qualquer chave s fixada da árvore, o custo para buscar por s é $\mathcal{O}(n\log n + m + \sum_{i=1}^{m}\log(|s_i - s|+1))$; e o *Working Set Bound*, onde para uma chave s_i qualquer na árvore, seja $t(s_i)$ a quantidade de chaves distintas acessadas entre s_i e a última ocorrência de s_i na sequência de buscas (ou o início da sequência, se s_i é a primeira ocorrência da chave na sequência), o custo para buscar por S é $\mathcal{O}(n\log n + m + \sum_{j=1}^{m}\log(t(s_j)+1))$.

Tarjan [45] mostrou que realizar uma sequência de buscas em uma Árvore *Splay*, onde a sequência é ordenada, tem custo $\mathcal{O}(n)$. Tarjan chamou este limite de *Sequential Access*. Sundar [46] e Elmasry [47] também mostraram que este limitante tem custo $\mathcal{O}(n)$, este último com um fator contante menor. Sleator e Tarjan [9] apresentaram a conjectura do *Traversal Access* que, quando a sequência de buscas é a sequência pré-ordem de uma *BST* qualquer com o mesmo conjunto de nós da *BST* inicial, o custo de buscar esta sequência é $\mathcal{O}(n)$. Esta conjectura ainda não foi resolvida. Chaudhuri e Höft [48] provaram um caso especial desta conjectura, que é quando a sequência de buscas é a sequência pré-ordem da árvore inicial. Kozma [21] também provou esta conjectura para quando a *BST* inicial é uma *treap* de mínimo sobre o conjunto de nós $\{(x,\mu(x,0))\}$, onde $x \in \{1,\ldots,n\}$. Nota-se que, o *Sequential Access* é um caso especial do *Traversal Access*. Tarjan também apresentaram o limitante *Balance Bound*, onde o custo para

acessar $\mathcal{O}((m+n)\log(n+m))$.

Sleator e Tarjan [9] apresentaram a conjectura do *Dynamic Finger Bound*, tal que se um algoritmo *BST* possui o *Dynamic Finger Bound*, seu custo é $\mathcal{O}(m+n+\sum_{j=1}^{m-1}\log(|s_{j+1}-s_j|+1))$. 15 anos mais tarde, Cole *et al.*; Cole [49, 50] provaram o *Dynamic Finger Bound* para a Árvore *Splay*. Iacono [51] apresentou um limitante, que é uma combinação dos limitantes *Working Set Bound* e *Dynamic Finger Bound*, chamado *Unified Bound* (UB). No *Unified Bound* o custo amortizado para acessar uma chave s_i é $\mathcal{O}(\min_{s_j}\log(t(s_j)+|s_i-s_j|+2))$. Elmasry, Farzan e Iacono [52] mostraram que o *Working Set Bound* é assintoticamente equivalente ao *Unified Bound*.

Lucas [10] conjeturou que seu algoritmo *Greedy Future* fornece uma aproximação de fator constante para $\mathrm{OPT}(S)$. Munro [11] conjecturou que o *Greedy Future* tem custo $\mathrm{OPT}(S)+\mathcal{O}(m)$. Fox [53] provou que *Greedy Future* tem os limitantes: *Balance Bound*, *Static Optimality Bound*, *Static Finger Bound*, *Working Set Bound* e *Sequential Access*. Independentemente, Goyal e Gupta [54] também mostraram o limitante *Balance Bound*. Goyal e Gupta também mostraram que o algoritmo *GreedyASS* é $\mathcal{O}(\log n)$-competitivo.

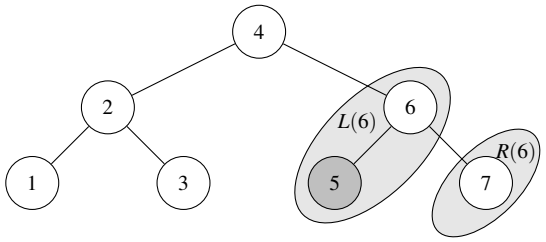
Para a *Multi-Splay Tree* [3] foi provado os limitantes: *Static Finger Bound*, *Static Optimality Bound*, *Working Set Bound* e *Sequential Access*. Derryberry e Sleator [43] provaram que o custo do algoritmo *Skip-Splay* é $\mathcal{O}(m\log\log n+UB(S))$.

5.2 Limitantes inferiores

Um limitante inferior trivial para $\mathrm{OPT}(S)$ é o tamanho da sequência de buscas, pois pelo menos um nó será visitado durante cada busca, mas este limitante pode está muito distante de $\mathrm{OPT}(S)$. Wilber [15] foi um dos primeiros autores a propor limitantes inferiores. Wilber propôs dois limitantes para $\mathrm{OPT}(S)$. No seu primeiro limitante, chamado de *Limite Inferior de Alternações* (*Interleave Lower Bound* - ILB, do inglês) por Wilber, uma *BST* balanceada P, chamada de *árvore de limite inferior*, com as mesmas chaves da *BST* inicial é usada. Para cada nó y de P, é definida a *região esquerda* de y, denotada por $L(y)$, por y e os nós de $\mathrm{T_L}(y)$ e a *região direita* de y, denotada por $R(y)$, pelos nós de $\mathrm{T_R}(y)$. Para cada y em P, rotulamos cada acesso $s_i \in S$ com L, se $s_i \in L(y)$, ou com R, se $s_i \in R(y)$. Para cada nó y de P, contamos a quantidade de alternações entre os rótulos "L" e "R" na sequência de buscas S. O limite inferior de alternações, denotado por $W_1(S)$, é a soma das quantidades de alternações entre os rótulos para cada y.

Na Figura 6 temos em destaque as regiões esquerda e direita do nó 6. Podemos ver também na mesma figura que para acessar o nó 5 há uma alternação de $L(4)$ para $R(4)$. Para a sequência de buscas $S = \{6,2,4,5,7,3,1\}$, temos $W_1(S) = 5$.

Figura 6 – Primeiro limite inferior de Wilber



Fonte: Elaborada pelo autor

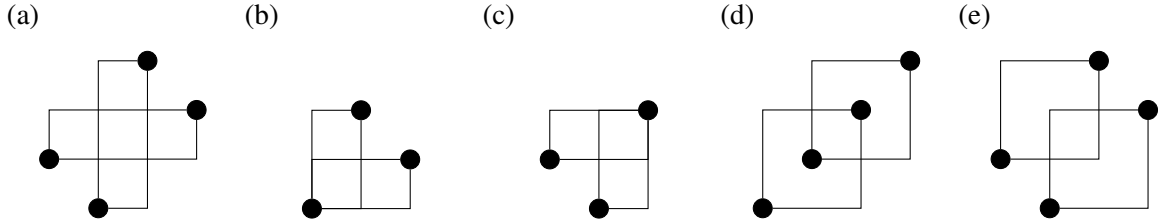
O segundo limite proposto Wilber não depende da árvore de limite inferior nem da *BST* inicial. Para cada acesso $s_j \in S$, definimos a sequência de chaves $S_j = (s_i, s_{i+1}, \ldots, s_{j-1})$, onde $s_i = s_j$ e i é o maior valor possível tal que $1 \leq i < j$, se s_j aparece em $(s_1, \ldots, s_{j-1})$. Caso contrário, $i = 1$. Agora, dizemos que $a_k < s_j < b_k$, onde k decresce de $j-1$ até i e $a_k, b_k \in \{s_k, s_{k+1}, \ldots, s_{j-1}\}$ são os limites mais apertados para s_j. Cada vez que k for decrementado, ou a_k aumenta ou b_k diminui, ou nada acontece. O segundo limite de Wilber, denotado por $W_2(S)$, é a soma do número de alternações entre a_k aumentando e b_k diminuindo para cada s_j. Ao propor limitantes, Wilber conjecturou que o segundo domina o primeiro. Esta conjectura ficou em aberto por 30 anos e foi respondida em 2019 por Lecomte e Weinstein [16].

Outro limitante inferior foi proposto por Demaine *et al.* [5] usando a visão geométrica (apresentado no Capítulo 4), o *Limite de Retângulos Independentes* (*Independent Rectangles Bound* - IRB, do inglês). Dizemos que dois retângulos $\square ab$ e $\square cd$ são independentes, onde $a, b, c, d \in P$, em P se os retângulos não são arboreamente satisfeitos e nenhum canto de qualquer dos retângulos estiver estritamente dentro do outro retângulo. Este limitante é válido quando as coordenadas x dos pontos de P são distintas. Denotamos a quantidade máxima retângulos independentes em um conjunto de pontos P por $maxIRB(P)$. Temos exemplos de retângulos independentes nas figuras 7a, 7b e 7c, e de retângulos dependentes nas figuras 7d e 7e.

Demaine *et al.* apresentaram uma construção de conjuntos de retângulos independentes para cada limite inferior proposto por Wilber. No Teorema 5.2.1 abaixo mostra que um conjunto de retângulos independentes I é limite inferior para um conjunto de pontos P.

Teorema 5.2.1 ([5]). *Se um conjunto de pontos P contém um conjunto de retângulos independentes I, então $minASS(P) \geq \frac{|I|}{2} + |P|$. Em particular, se $P = P(S)$ para uma sequência de buscas S,*

Figura 7 – Retângulos independentes e retângulos dependentes



Fonte: Elaborada pelo autor

então $\text{OPT}(S) \geq \frac{|I|}{2} + |S|$.

Na representação geométrica apresentada por Derryberry, Sleator e Wang [14], os autores também propuseram um limite inferior chamado *Rectangle Cover Lower Bound*. Também mostram que seu limite é uma generalização dos dois limites de Wilber.

Dizemos que um retângulo $\square ab$ é um $\boxslash$-retângulo ($\boxbslash$-retângulo) se a inclinação da reta $\overline{ab}$ é positiva (negativa). Um conjunto de pontos P é $\boxslash$-satisfeito se todo par de pontos $(a,b) \in P$ que formam um $\boxslash$-retângulo é arboreamente satisfeito. Denotamos por $minASS_{\boxslash}(P)$ o tamanho do menor superconjunto $\boxslash$-satisfeito de P.

Para encontrar um superconjunto $\boxslash$-satisfeito de um conjunto de pontos P, Demaine *et al.* [5] propuseram o algoritmo guloso *Signed Greedy* descrito abaixo. O algoritmo funciona semelhante ao *GreedyASS*, mas ignorando os $\boxbslash$-retângulos.

O *Signed Greedy* varre o conjunto de pontos P com uma linha horizontal, incrementalmente na coordenada y. Para cada $\boxslash$-retângulo ($\boxbslash$-retângulo) insatisfeito formado por um ponto na linha de varredura e um ponto abaixo da linha de varredura, um ponto é adicionado no canto superior esquerdo (direito) do retângulo para torná-lo satisfeito.

Assim, podemos executar o *Signed Greedy* em suas duas versões, para satisfazer os $\boxslash$-retângulos e para satisfazer os $\boxbslash$-retângulos de um conjunto de pontos P, e obter o melhor limitante inferior por $\max\{minASS_{\boxslash}(P), minASS_{\boxbslash}(P)\}$. Demaine *et al.* [5] mostra que este limitante obtido com o *Signed Greedy* está a um fator constante de $maxIRB(P)$.

6 RESULTADOS RECENTES

Neste capítulo descrevemos algumas tentativas recentes de atingir a otimalidade dinâmica. Na Seção 6.1 apresentamos uma formulação linear inteira de Demaine *et al.* [13] para encontrar o menor superconjunto arboreamente satisfeito de um conjunto de pontos dado. Na Seção 6.2 apresentamos uma proposta de Levy e Tarjan [55] de provar que a Árvore *Splay* é dinamicamente ótima.

6.1 Aproximação com Programação Linear

Demaine *et al.* [13] propuseram uma abordagem de programação linear para o problema ArbSS. Dois modelos de programação linear inteira para calcular o menor superconjunto arboreamente satisfeito dado um conjunto de pontos P foram propostos.

Os pontos com coordenadas inteiras do *grid* $n \times m$ são considerados. Uma variável binária b_{ij} corresponde ao ponto (i, j) no *grid*, onde recebe o valor 1, se o ponto é incluído na solução, ou 0, caso contrário. A função objetivo dos modelos consiste na soma das variáveis b_{ij} nas quais queremos minimizá-las. O primeiro modelo proposto é quadrático. As equações 6.1 e 6.2 são as restrições do primeiro modelo para garantir que o conjunto de pontos seja arboreamente satisfeito. A Equação 6.1 garante a satisfação dos ◹-retângulos e a Equação 6.2 a satisfação dos ◺-retângulos.

$$2b_{ij} + 2b_{rs} - \left(\sum_{k=i}^{r} \sum_{l=j}^{s} b_{kl} \right) \leq 1 \ \forall \ i < r, j < s \tag{6.1}$$

$$2b_{is} + 2b_{rj} - \left(\sum_{k=i}^{r} \sum_{l=j}^{s} b_{kl} \right) \leq 1 \ \forall \ i < r, j < s \tag{6.2}$$

Pelo Lema 4.1.1, em um conjunto de pontos qualquer arboreamente satisfeito, existe pelo menos um ponto nos lados do retângulo incidentes a cada ponto que definem o retângulo. A partir deste lema, Demaine *et al.* definiram novas restrições em que tenha pelo menos um ponto nos lados do retângulo além dos pontos que definem o retângulo. Este novo modelo é linear, pois são considerados apenas os pontos nos lados do retângulo. As equações 6.3 e 6.4 são as

restrições do segundo modelo.

$$b_{ij} + b_{rs} - \left(\sum_{l=i+1}^{r-1} (b_{lj} + b_{ls}) + \sum_{l=j+1}^{s-1} (b_{il} + b_{rl}) + b_{is} + b_{rj} \right) \leq 1 \ \forall \ i < r, j < s \tag{6.3}$$

$$b_{is} + b_{rj} - \left(\sum_{l=i+1}^{r-1} (b_{lj} + b_{ls}) + \sum_{l=j+1}^{s-1} (b_{il} + b_{rl}) + b_{ij} + b_{rs} \right) \leq 1 \ \forall \ i < r, j < s \tag{6.4}$$

Da mesma forma do modelo anterior, a Equação 6.3 garante a satisfação dos ◹-retângulos e a Equação 6.4 a satisfação dos ◺-retângulos.

6.2 Monotonicidade

Levy e Tarjan [55] tentam provar que a Árvore *Splay* é dinamicamente ótima combinando os conceitos de monotonicidade e simulação de instâncias. Antes de apresentar as definições destes conceitos, dizemos que Y é uma *subsequência* de uma sequência X, se Y é obtido através de uma sequência de eliminações de chaves de X. Por exemplo, $(2,4,6)$ é subsequência de $(1,2,3,4,5,6)$. Dizemos que um algoritmo $\mathcal{A}$ é *aproximadamente monótono* (ou simplesmente *monótono*), se existe alguma constante $b > 0$ tal que $custo_{\mathcal{A}}(Y,T) \leq b\ custo_{\mathcal{A}}(X,T)$ para toda sequência de busca X, subsequência Y de X e árvore inicial T, onde $custo_{\mathcal{A}}(Z,T_0)$ é o custo para o algoritmo $\mathcal{A}$ executar a sequência Z com a árvore inicial T_0. Se $b = 1$, então dizemos que $\mathcal{A}$ é *estritamente monótono*.

Uma *simulation embedding* é definido por Levy e Tarjan [55], onde uma *simulation embedding* $\mathcal{S}_{\mathcal{A}}$ de um algoritmo *BST* $\mathcal{A}$ é um mapeamento de execuções de sequências de buscas, tal que para algum $c > 0$, $custo_{\mathcal{A}}(\mathcal{S}_{\mathcal{A}},T) \leq c|E|$, onde $|E|$ é o custo de uma execução $|E|$; e X é uma subsequência de $\mathcal{S}_{\mathcal{A}}$ para toda sequência de buscas X, árvore inicial T e execução E de X em T. Assim, se um algoritmo é aproximadamente monótono e com *simulation embedding*, então este algoritmo é dinamicamente ótimo. Como apresentado no Teorema 6.2.1.

Teorema 6.2.1 ([55]). *Algoritmos aproximadamente monótono com* simulation embedding *são* $\mathcal{O}(1)$*-competitivo.*

No trabalho de Levy e Tarjan [55], eles assumem que as rotações no modelo *BST* são rotações restritas. *Rotações restritas* são rotações apenas nos nós cujo nó pai ou é a raiz ou é o filho esquerdo da raiz da árvore. Cleary e Taback [56] e Lucas [57] provaram que qualquer *BST*

T_1 de tamanho n pode ser transformada em qualquer outra *BST* T_2 com o mesmo conjunto de nós com no máximo $4n$ rotações restritas. Levy e Tarjan [55] mostram que com custo no máximo $80n$ é possível transformar uma *BST* T_1 em qualquer outra *BST* T_2 com o mesmo conjunto de nós aplicando *Splayings* (busca em uma Árvore *Splay*), como descrito no Teorema 6.2.2.

Teorema 6.2.2 ([55]). *Seja* T_1 *e* T_2 BSTs *de tamanho* $n \geq 4$ *com as mesmas chaves. Existe uma sequência de* Splayings *que transforma* T_1 *em* T_2 *com custo no máximo* $80n$.

Dado duas *BSTs* T_1 e T_2 com o mesmo conjunto de nós, denotamos por $\mathbb{T}(T_1, T_2)$ a sequência de *Splayings* descrita no Teorema 6.2.2 para transformar T_1 em T_2. Se $T_1 = T_2$, então $\mathbb{T}(T_1, T_2) = \{raiz(T_1)\}$. Além disso, denotamos por $X \bigoplus Y$ das sequências $X = (x_1, \ldots, x_k)$ e $Y = (y_1, \ldots, y_l)$ em $(x_1, \ldots, x_k, y_1, \ldots, y_l)$.

A entrada para uma *simulation embedding* é uma execução $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$ de uma sequência de buscas X em uma árvore inicial T. O custo de uma *simulation embedding* de uma execução E com uma *BST* inicial T é denotada por $custo(\mathcal{S}(E), T)$.

Teorema 6.2.3 ([55]). *O mapa* $\mathcal{S}(E) \equiv \mathbb{T}(Q_1, Q_1') \bigoplus \mathbb{T}(Q_2, Q_2') \bigoplus \ldots \bigoplus \mathbb{T}(Q_m, Q_m')$ *de uma sequência de buscas é uma* simulation embedding *para a Árvore* Splay.

Demonstração. Por construção, na execução de cada bloco $\mathbb{T}(Q_i, Q_i')$ $(1 \leq i \leq m)$, Q_i é substituída por Q_i' em T_{i-1}. Observe que a Árvore *Splay* sempre rotaciona a chave buscada para a raiz da árvore. Consequentemente, a última chave de $\mathbb{T}(Q_i, Q_i')$ é sempre x_i, significa que X é subsequência de $\mathcal{S}(E)$. Por fim, $custo(\mathcal{S}(E), T) = \sum_{i=1}^{m} custo(\mathbb{T}(Q_i, Q_i'), T_{i-1}) \leq 80(|Q_1'| + \ldots + |Q_m'|)$. A desigualdade é dada pelo Teorema 6.2.2. O custo da execução inicial é $|Q_1'| + \ldots + |Q_m'|$. Consequentemente, uma constante multiplicativa do custo da execução inicial é um limite superior para a Árvore *Splay* executar a simulação. Assim, concluímos que $\mathcal{S}$ é uma *simulation embedding* para a Árvore *Splay*. □

A partir deste teorema, podemos concluir que:

Teorema 6.2.4 ([55]). *Se a Árvore* Splay *é aproximadamente monótona, então ela é dinamicamente ótima.*

A prova do Teorema 6.2.4 é consequência dos teoremas 6.2.1 e 6.2.3.

Levy e Tarjan mostram que a otimalidade dinâmica requer a monotonicidade. Então eles mostram que OPT é monótono, apresentado no Teorema 6.2.5.

Teorema 6.2.5 ([55])**.** OPT *é estritamente monótono.*

Demonstração. Seja Y uma sequência estrita de $X = \{x_1, x_2, \ldots, x_m\}$ e T_0 a árvore inicial contendo as chaves de X. Seja $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$ uma execução ótimo de X iniciando com a árvore T_0. Vamos construir uma execução E' de Y iniciando com T_0 de custo menor do que $\mathrm{OPT}(X)$ iniciando com T_0.

Observe que Y é formado por eliminações de algumas buscas de X correspondente a um subconjunto de índices de 1 a m. Seja j o menor índice eliminado e seja k o menor índice maior do que j que não foi eliminado de X para formar Y. Se k não existir, (isto é, as buscas $\{x_j, \ldots, x_m\}$ foram eliminadas), então definimos $E' = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_{j-1}, T_{j-1} \rangle$, e a execução de Y está definida. Caso contrário, substituímos $T_{j-1} \overset{Q_j}{\to} T_j, \ldots, T_{k-1} \overset{Q_k}{\to} T_k$ por $T_{j-1} \overset{Q}{\to} T_k$, onde $Q = Q_j \cup \ldots \cup Q_k$. Os rearranjos $T_{i-1} \overset{Q_i}{\to} T_i$ $(j \leq i \leq k)$ são aplicados em ordem em T_{j-1}. Assim, definimos $E' = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_{j-1}, T_{j-1}, Q, T_k, \ldots, Q_m, T_m \rangle$. Veja que $|Q| < |Q_j| + \ldots + |Q_k| - (k - j)$ e como $k - j > 1$, o custo de $T_{j-1} \overset{Q}{\to} T_k$ é estritamente menor do que as operações $T_{j-1} \overset{Q_j}{\to} T_j, \ldots, T_{k-1} \overset{Q_k}{\to} T_k$. Repetindo este processo para cada subsequência contíguas de buscas eliminadas de X para formar Y forma uma execução E' de Y custando menos do que E. Como E é uma execução ótima de X, então $\mathrm{OPT}(Y) < \mathrm{OPT}(X)$ com a mesma árvore inicial. □

A partir do Teorema 6.2.5, podemos observar:

Teorema 6.2.6. *Se a Árvore* Splay *é dinamicamente ótima, então ela é aproximadamente monótona.*

Demonstração. Para toda subsequência Y de X, $custo_{Splay}(Y,T) \leq c\,\mathrm{OPT}(Y,T) \leq c\,\mathrm{OPT}(X,T) \leq c\ custo_{Splay}(X,T)$. □

Uma outra proposta de provar a otimalidade dinâmica através da monotonicidade foi apresentada há mais de uma década por Harmon [58]. Harmon em seu trabalho tenta mostrar que o algoritmo *GreedyASS* (apresentado na Seção 4.3) é dinamicamente ótimo se ele é aproximadamente monótono. Harmon constrói sua simulação da seguinte forma: seja $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$ uma execução da sequência de buscas S e árvore inicial T_0. Definimos para uma *BST* T $\mathbb{G}(T) = (\bigoplus_{i=1}^{|T|}(q_i, i)) \bigoplus q_1$, onde $q_1, q_2, \ldots, q_{|T_0|}$ são os nós de T em ordem crescente. A *simulation embedding* do *GreedyASS* é o conjunto de pontos $P(E) = \mathbb{G}(Q_1) \bigoplus \ldots \bigoplus \mathbb{G}(Q_m)$. Harmon prova que o *GreedyASS* adiciona no máximo $\mathcal{O}(|Q_1| + \ldots + |Q_m|)$ pontos para satisfazer $P(E)$.

7 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Neste capítulo, apresentamos as considerações finais sobre o estudo realizado (Seção 7.1).

7.1 Considerações finais

Neste *survey* sobre a otimalidade dinâmica do problema de busca em *BST*, realizamos uma coleta de artigos, dissertações e teses sobre a otimalidade dinâmica e reunimos neste trabalho os principais resultados e melhorias propostas na literatura no decorrer destes 35 anos desde que a conjectura foi apresentada.

Árvores Binárias de Busca são estruturas de dados fundamentais para a Ciência da Computação, onde utilizadas na prática em *kernels* de sistemas operacionais e em memória *cache* em redes de computadores [7]. Durante muito tempo o melhor fator competitivo de um algoritmo *BST* para o problema de busca em *BST* dinâmico foi $\mathcal{O}(\log n)$, onde n é a quantidade de chaves na árvore. Até que em 2004, Demaine *et al.* [2] melhoraram este fator competitivo para $\mathcal{O}(\log\log n)$, quando propuseram a Árvore Tango. Posteriormente, este fator também foi alcançado por Wang, Derryberry e Sleator [3] e Georgakopoulos [4] nas árvores *Multi-Splay Tree* e *Chain-Splay Tree*, respectivamente. O fator competitivo $\mathcal{O}(\log\log n)$ é o melhor conhecido. Para o problema de busca em *BST* estático existem dois algoritmos de programação dinâmica que constroem uma *BST* estática ótima com custo $\mathcal{O}(n^2)$ propostos por Knuth [8] e Yao [25].

A visão geométrica proposta por Demaine *et al.* [5], e independentemente por Derryberry, Sleator e Wang [14], é um resultado surpreendente, pois é uma maneira mais fácil de visualizar uma execução *BST* e, além disto, o problema do superconjunto de pontos arboreamente satisfeito, da visão geométrica, é equivalente ao problema dinâmico de busca em *BST*. Contribuímos com uma prova desta equivalência e outras provas relacionadas à visão geométrica, nas quais Demaine *et al.* apresentaram apenas uma ideia das provas sem os devidos detalhes.

Por fim, espera-se que este trabalho possa contribuir através desta sumarização dos principais resultados relacionados a otimalidade dinâmica propostos na literatura.

REFERÊNCIAS

1 SLEATOR, D. D.; TARJAN, R. E. Self-adjusting binary trees. In: **Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA**. [S.l.: s.n.], 1983. p. 235–245.

2 DEMAINE, E. D. *et al.* Dynamic optimality - almost. In: **45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings**. [S.l.: s.n.], 2004. p. 484–490.

3 WANG, C. C.; DERRYBERRY, J. C.; SLEATOR, D. D. $O(\log\log n)$-competitive dynamic binary search trees. In: **Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm**. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006. (SODA '06), p. 374–383. ISBN 0-89871-605-5.

4 GEORGAKOPOULOS, G. F. Chain-splay trees, or, how to achieve and prove $\log\log n$-competitiveness by splaying. **Information Processing Letters**, v. 106, n. 1, p. 37–43, 2008.

5 DEMAINE, E. D. *et al.* The geometry of binary search trees. In: **Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009**. [S.l.: s.n.], 2009. p. 496–505.

6 HIBBARD, T. N. Some combinatorial properties of certain trees with applications to searching and sorting. **J. ACM**, v. 9, n. 1, p. 13–28, 1962.

7 PFAFF, B. Performance analysis of bsts in system software. In: **Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA**. [S.l.: s.n.], 2004. p. 410–411.

8 KNUTH, D. E. Optimum binary search trees. **Acta Informatica**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 1, n. 1, p. 14–25, 1971. ISSN 0001-5903.

9 SLEATOR, D. D.; TARJAN, R. E. Self-adjusting binary search trees. **J. ACM**, ACM, New York, NY, USA, v. 32, n. 3, p. 652–686, 1985. ISSN 0004-5411.

10 LUCAS, J. M. **Canonical Forms for Competitive Binary Search Tree Algorithms**. [S.l.]: Rutgers University, Department of Computer Science, Laboratory for Computer Science Research, 1988. (DCS-TR-).

11 MUNRO, J. I. On the competitiveness of linear search. In: **Proceedings of the 8th Annual European Symposium on Algorithms**. London, UK, UK: Springer-Verlag, 2000. (ESA '00), p. 338–345. ISBN 3-540-41004-X.

12 WANG, C. C. **Multi-Splay Trees**. Tese (Doutorado), Carnegie Mellon University, 2006.

13 DEMAINE, E. D. *et al.* Arboral satisfaction: Recognition and LP approximation. **Information Processing Letters**, v. 127, p. 1–5, 2017.

14 DERRYBERRY, J. C.; SLEATOR, D. D.; WANG, C. C. **A lower bound framework for binary search trees with rotations**. [S.l.], 2005.

15 WILBER, R. E. Lower bounds for accessing binary search trees with rotations. **SIAM Journal on Computing**, v. 18, n. 1, p. 56–67, 1989.

16 LECOMTE, V.; WEINSTEIN, O. Settling the relationship between wilber's bounds for dynamic optimality. **CoRR**, abs/1912.02858, 2019.

17 SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. [S.l.]: Ed. LTC, 1994. ISBN 9788521610144.

18 CORMEN, T. H. *et al.* **Introduction to Algorithms, Third Edition**. 3rd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262033844, 9780262033848.

19 ADELSON-VELSKIĬ, G. M.; LANDIS, E. M. An algorithm for the organization of information. **Soviet Mathematics Doklady**, v. 3, p. 1259–1263, 1962.

20 GUIBAS, L. J.; SEDGEWICK, R. A dichromatic framework for balanced trees. In: **19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978**. [S.l.: s.n.], 1978. p. 8–21.

21 KOZMA, L. **Binary search trees, rectangles and patterns**. Tese (Doutorado) — Saarland University, Saarbrücken, Germany, 2016.

22 CULIK, K.; WOOD, D. A note on some tree similarity measures. **Information Processing Letters**, v. 15, n. 1, p. 39–42, 1982.

23 SLEATOR, D. D.; TARJAN, R. E.; THURSTON, W. P. Rotation distance, triangulations, and hyperbolic geometry. In: **Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing**. [S.l.: s.n.], 1986. (STOC '86), p. 122–135.

24 GILBERT, E. N.; MOORE, E. F. Variable-length binary encodings. **The Bell System Technical Journal**, v. 38, n. 4, p. 933–967, 1959.

25 YAO, F. F. Speed-up in dynamic programming. **SIAM Journal on Algebraic and Discrete Methods**, v. 3, n. 4, p. 532–540, 1982.

26 MEHLHORN, K. Nearly optimal binary search trees. **Acta Informatica**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 5, n. 4, p. 287–295, 1975. ISSN 0001-5903.

27 LEVCOPOULOS, C.; LINGAS, A.; SACK, J. Heuristics for optimum binary search trees and minimum weight triangulation problems. **Theoretical Computer Science**, v. 66, n. 2, p. 181–203, 1989.

28 HU, T. C.; TUCKER, A. C. Optimal computer search trees and variable-length alphabetical codes. **SIAM Journal on Applied Mathematics**, Society for Industrial and Applied Mathematics, v. 21, n. 4, p. 514–532, 1971.

29 GARSIA, A. M.; WACHS, M. L. A new algorithm for minimum cost binary trees. **SIAM Journal on Computing**, v. 6, n. 4, p. 622–642, 1977.

30 GAREY, M. R. Optimal binary search trees with restricted maximal depth. **SIAM Journal on Computing**, v. 3, n. 2, p. 101–110, 1974.

31 WESSNER, R. L. Optimal alphabetic search trees with restricted maximal height. **Information Processing Letters**, v. 4, n. 4, p. 90–94, 1976.

32 ITAI, A. Optimal alphabetic trees. **SIAM Journal on Computing**, v. 5, n. 1, p. 9–18, 1976.

33 BECKER, P. Optimal binary search trees with near minimal height. **CoRR**, abs/1011.1337, 2010.

34 KARLIN, A. R. *et al.* Competitive snoopy caching. In: **27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986**. [S.l.: s.n.], 1986. p. 244–254.

35 SLEATOR, D. D.; TARJAN, R. E. Amortized efficiency of list update and paging rules. **Communications of the ACM**, v. 28, n. 2, p. 202–208, 1985.

36 ALLEN, B.; MUNRO, J. I. Self-organizing binary search trees. **J. ACM**, v. 25, n. 4, p. 526–535, 1978.

37 RIVEST, R. L. On self-organizing sequential search heuristics. In: **15th Annual Symposium on Switching and Automata Theory, New Orleans, Louisiana, USA, October 14-16, 1974**. [S.l.: s.n.], 1974. p. 122–126.

38 MCCABE, J. On serial files with relocatable records. **Operations Research**, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 13, n. 4, p. 609–618, 1965.

39 SUBRAMANIAN, A. An explanation of splaying. In: **Foundations of Software Technology and Theoretical Computer Science, 14th Conference, Madras, India, December 15-17, 1994, Proceedings**. [S.l.: s.n.], 1994. p. 354–365.

40 BALASUBRAMANIAN, R.; RAMAN, V. Path balance heuristic for self-adjusting binary search trees. In: **Foundations of Software Technology and Theoretical Computer Science, 15th Conference, Bangalore, India, December 18-20, 1995, Proceedings**. [S.l.: s.n.], 1995. p. 338–348.

41 DORFMAN, D. *et al.* Improved bounds for multipass pairing heaps and path-balanced binary search trees. In: **26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland**. [S.l.: s.n.], 2018. p. 24:1–24:13.

42 DEMAINE, E. D. *et al.* Dynamic optimality - almost. **SIAM Journal on Computing**, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 37, n. 1, p. 240–251, 2007. ISSN 0097-5397.

43 DERRYBERRY, J. C.; SLEATOR, D. D. Skip-splay: Toward achieving the unified bound in the BST model. In: **Algorithms and Data Structures, 11th International Symposium, WADS 2009, Banff, Canada, August 21-23, 2009. Proceedings**. [S.l.: s.n.], 2009. p. 194–205.

44 BOSE, P. *et al.* An $O(\log\log n)$-competitive binary search tree with optimal worst-case access times. **CoRR**, abs/1003.0139, 2010.

45 TARJAN, R. E. Sequential access in splay trees takes linear time. **Combinatorica**, v. 5, n. 4, p. 367–378, 1985.

46 SUNDAR, R. On the deque conjecture for the splay algorithm. **Combinatorica**, v. 12, n. 1, p. 95–124.

47 ELMASRY, A. On the sequential access theorem and deque conjecture for splay trees. **Theoretical Computer Science**, v. 314, n. 3, p. 459–466, 2004.

48 CHAUDHURI, R.; HÖFT, H. Splaying a search tree in preorder takes linear time. **SIGACT News**, v. 24, n. 2, p. 88–93, 1993.

49 COLE, R. *et al.* On the dynamic finger conjecture for splay trees. part I: Splay sorting $\log n$-block sequences. **SIAM Journal on Computing**, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 30, n. 1, p. 1–43, 2000.

50 COLE, R. On the dynamic finger conjecture for splay trees. part II: The proof. **SIAM Journal on Computing**, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 30, n. 1, p. 44–85, 2000.

51 IACONO, J. Alternatives to splay trees with $O(\log n)$ worst-case access times. In: **Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.** [S.l.: s.n.], 2001. p. 516–522.

52 ELMASRY, A.; FARZAN, A.; IACONO, J. On the hierarchy of distribution-sensitive properties for data structures. **Acta Informatica**, v. 50, n. 4, p. 289–295, 2013.

53 FOX, K. Upper bounds for maximally greedy binary search trees. In: **Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings**. [S.l.: s.n.], 2011. p. 411–422.

54 GOYAL, N.; GUPTA, M. On dynamic optimality for binary search trees. **CoRR**, abs/1102.4523, 2011.

55 LEVY, C. C.; TARJAN, R. E. **New Paths from Splay to Dynamic Optimality**. Tese (Doutorado) — Princeton University, 2019.

56 CLEARY, S.; TABACK, J. Bounding restricted rotation distance. **Information Processing Letters**, v. 88, n. 5, p. 251–256, 2003.

57 LUCAS, J. M. A direct algorithm for restricted rotation distance. **Information Processing Letters**, v. 90, n. 3, p. 129–134, 2004.

58 HARMON, D. **New Bounds on Optimal Binary Search Trees**. Tese (Doutorado), Cambridge, MA, USA, 2006.