UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GEORGE EDSON ALBUQUERQUE PINTO

OTIMALIDADE DINÂMICA: UM *SURVEY*

FORTALEZA

2021

GEORGE EDSON ALBUQUERQUE PINTO

OTIMALIDADE DINÂMICA: UM *SURVEY*

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Victor Almeida Campos

FORTALEZA

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P728o Pinto, George Edson Albuquerque.
Otimalidade dinâmica: Um survey / George Edson Albuquerque Pinto. – 2021.
55 f. : il.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2021.
Orientação: Prof. Dr. Victor Almeida Campos.

1. árvore binária de busca. 2. otimalidade dinâmica. 3. estrutura de dados. 4. limitantes. I. Título.
CDD 005

GEORGE EDSON ALBUQUERQUE PINTO

OTIMALIDADE DINÂMICA: UM *SURVEY*

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

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 Cariri (UFCA)

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

AGRADECIMENTOS

Agradeço primeiramente a Deus, que permitiu que tudo isso acontecesse ao longo de minha vida.

A minha família, pelo amor, incentivo e apoio incondicional desde o princípio dos meus estudos, que nos momentos de minha ausência dedicados aos estudos, sempre fizeram entender que o futuro é feito a partir da constante dedicação no presente.

A Universidade Federal do Ceará, seu corpo docente, direção e administração que oportunizaram a janela que hoje vislumbro um horizonte superior.

Ao Prof. Dr. Victor Almeida Campos, pela orientação, confiança, pela oportunidade e apoio na elaboração desta dissertação.

Agradeço a todos os professores por me proporcionarem o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação no processo de formação profissional, tamanha dedicação a mim, não somente por terem me ensinado, mas por terem me feito aprender. Sempre terão os meus eternos agradecimentos.

Meus agradecimentos aos meus amigos que fizeram parte da minha formação e que vão continuar presentes em minha vida, com certeza.

A todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

E à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo financiamento da pesquisa de mestrado, via bolsa de estudos.

RESUMO

Árvores Binárias de Busca (*BSTs*) pertencem às estruturas de dados clássicas dentro da Ciência da Computação e, apesar de sua simplicidade, possuem muitas questões em aberto após décadas de pesquisa. A principal questão em aberto sobre *BSTs* é a seguinte: "Qual é a melhor Árvore Binária de Busca sobre uma dada sequência de buscas qualquer de elementos da árvore?". Em 1983, a Árvore *Splay* foi proposta e conjecturada que seu 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 mostrado que a Árvore *Splay*, ou qualquer outra *BST*, é dinamicamente ótima. O melhor custo conhecido é $\mathcal{O}(\log\log n \cdot \mathrm{OPT}(S))$, onde n é a quantidade de nós na árvore, da Árvores Tango, *Multi-Splay Tree* e *Chain-Splay Tree*. Em 2009, foi proposto um modelo de pontos no plano Cartesiano que equivale ao problema de busca em uma *BST*. Este resultado é surpreendente, pois representa uma maneira visual da execução de um algoritmo *BST* sobre uma sequência de buscas. Finalmente, esta dissertação trata-se de um *survey* da literatura sobre a otimalidade dinâmica, onde reunimos os principais resultados relacionados ao problema.

Palavras-chave: árvore binária de busca; otimalidade dinâmica; estrutura de dados; limitantes.

ABSTRACT

Binary Search Trees (*BSTs*) belong to the classic data structures within Computer Science and, despite their simplicity, have many open questions after decades of research. The main open question about *BSTs* is the following: “What is the best Binary Search Tree over any given search sequence of elements in the tree?”. In 1983, the *Splay* Tree was proposed and conjectured that its cost to perform any search sequence S on 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 dynamically 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 another BST, is dynamically optimal. The best known cost is $\mathcal{O}(\log\log n \cdot \mathrm{OPT}(S))$, where n is the number of nodes in the tree, from the Tango Tree *Multi-Splay Tree* and *Chain-Splay Tree*. In 2009, a point model in the Cartesian plane was proposed, which is equivalent to the search problem in a *BST*. This result is surprising, as it represents a visual way of executing a *BST* algorithm on a search sequence. Finally, this dissertation is a survey of the literature on dynamic optimality, where we gather the main results related to the problem.

Keywords: binary search tree; dynamic optimality; data structure; bounds.

LISTA DE FIGURAS

Figura 1 – Árvore Binária de Busca (*BST*). 17
Figura 2 – Rotação em *BST*. 18
Figura 3 – Exemplo de busca em uma *BST*. 19
Figura 4 – Exemplo da aplicação do Algoritmo LIMPAESQUERDA. 20
Figura 5 – Exemplo de rearranjo. 25
Figura 6 – Exemplo do Greedy Future. 27
Figura 7 – Operações *Splay*. 29
Figura 8 – Visão geométrica. 32
Figura 9 – Exemplos de *Treap*. 34
Figura 10 – Exemplos de *General treap*. 36
Figura 11 – Exemplo do primeiro limitante inferior de Wilber. 43
Figura 12 – Exemplo do segundo limitante inferior de Wilber. 43
Figura 13 – Retângulos independentes e retângulos dependentes. 44
Figura 14 – Exemplo da relaxação linear dos modelos de programação linear. 48

LISTA DE ALGORITMOS

Algoritmo 1 – LimpaEsquerda 19
Algoritmo 2 – Transformar 20
Algoritmo 3 – Splay 29

LISTA DE SÍMBOLOS

$\square ab$	Retângulo alinhado aos 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 de uma *BST* T
$\text{Rot}(x)$	Rotação do nó x em uma *BST*
$T_1 \overset{Q}{\to} T_2$	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** 11
2 **ÁRVORES BINÁRIA DE BUSCA E SEUS MODELOS** 14
2.1 **Árvore Binária de Busca** 14
2.2 **Modelos *BST*** 17
3 **ESTUDO DO PROBLEMA DE BUSCA EM BST** 23
3.1 **Problema estático** 23
3.2 **Problema dinâmico** 24
3.2.1 ***Problema* offline** 25
3.2.2 ***Problema* online** 27
3.3 **A conjectura da Otimalidade Dinâmica** 30
4 **VISÃO GEOMÉTRICA** 31
4.1 **Definição** 31
4.2 **Equivalência entre execução *BST* e conjunto arboreamente satisfeito** . . 33
4.3 **O problema do superconjunto arboreamente satisfeito (*ArbSS*)** 35
5 **LIMITANTES** 41
5.1 **Limitantes superiores** 41
5.2 **Limitantes inferiores** 42
6 **RESULTADOS RECENTES** 46
6.1 **Aproximação com Programação Linear** 46
6.2 **Monotonicidade** 48
7 **CONSIDERAÇÕES FINAIS** 52
REFERÊNCIAS 53

1 INTRODUÇÃO

O problema de busca é um dos problemas clássicos de Ciência da Computação. Supondo que precisamos armazenar um conjunto R de n elementos de algum universo ordenado $\mathcal{U}$ e queremos realizar consultas do tipo: um certo elemento $x \in \mathcal{U}$ está em R? E em caso de resposta "SIM", recuperamos dados adicionais associados a x. Para verificar se uma chave x está em R, podemos comparar x com cada elemento de R. Esta estratégia nos leva à resposta corretamente, mas pode despender tempo da ordem do número de elementos de R, que pode ser muito grande. Uma outra estratégia é fazer comparações, dos tipos $<$, $>$ e $=$, entre x e os elementos de R, pois como $\mathcal{U}$ é um universo ordenado, quando comparamos x com $y \in R$ e $x < y$, não precisamos mais comparar x com elementos de R maiores ou iguais a y. De forma simétrica, esta observação também é válida quando $x > y$. Nesse sentido, para verificar se $x \in R$ com esta nova estratégia, na maioria dos casos, ser feita com menos comparações do que a primeira e, consequentemente, pode demandar 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 disso, podemos realizar buscas por tais chaves de modo eficiente. Árvores Binárias de Busca são estruturas de dados clássicas, e foram apresentadas, inicialmente, por Hibbard [1] em 1962. Uma *BST* é uma estrutura de dados simples, mas que apesar de décadas de pesquisa, ainda possui muitas questões em aberto. Uma das principais questões sobre *BSTs*, apresentada por Sleator e Tarjan [2] em 1983, ainda permanece sem solução: "Qual é a melhor Árvore Binária de Busca sobre uma dada sequência de buscas qualquer de elementos da árvore?". As Árvores Binárias de Busca são utilizadas em *kernels* de sistemas operacionais e em memória *cache* em redes de computadores [3].

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. Esses 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 [4] em 1971, para obter uma *BST* que pode atingir o menor custo possível para uma sequência de buscas específica S.

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, caso pertença

à arvore. 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. Ademais, uma busca em uma *BST* utiliza a segunda estratégia descrita anteriormente, na tentativa de reduzir o custo da busca. O menor custo possível para realizar uma sequência de buscas S, dentre todos os algoritmo *BST*, 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 [5] 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 [6] também propôs um algoritmo *BST* guloso e conjecturou que seu algoritmo fornece uma aproximação de fator constante para o custo ótimo. Munro [7] 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 uma *BST* era $\mathcal{O}(\log n \, \mathrm{OPT}(S))$. Este custo pode ser obtido buscando por S em uma *BST* balanceada (*BST* de altura $\mathcal{O}(\log n)$). Recentemente, Demaine *et al.* [8] apresentaram a Árvore Tango, que tem custo $\mathcal{O}(\log\log n \cdot \mathrm{OPT}(S))$ para buscar por S. Este é o melhor resultado conhecido atualmente. Posteriormente, Wang *et al.* [9, 10] propuseram a *Multi-Splay Tree* e Georgakopoulos [11] propôs a *Chain-Splay Tree*, que também possuem custo $\mathcal{O}(\log\log n \cdot \mathrm{OPT}(S))$.

Recentemente, Demaine *et al.* [12] apresentaram uma representação de uma sequência de buscas em *BST* através de um conjunto de pontos no plano Cartesiano, chamada de Visão Geométrica, para visualizar as buscas em uma *BST* de forma mais simples. Os autores definiram um problema através dessa representação equivalente ao problema dinâmico de busca em *BST*. Além disso, na tentativa de provar a otimalidade dinâmica através desta representação, Demaine *et al.* [13] propuseram dois modelos de programação linear inteira, mas não obtiveram bons resultados. Derryberry *et al.* [14] também apresentaram uma representação no plano Cartesiano semelhante, independentemente.

Por 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 resulta em um bom limitante superior. O melhor limitante superior conhecido é $\mathcal{O}(\log\log n \cdot \mathrm{OPT}(S))$, obtido nas Árvores Tango [8], *Multi-Splay Tree* [9] e *Chain-Splay Tree* [11]. 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 a visão geométrica. Demaine *et al.* [12] e Derryberry *et al.* [14] propuseram, independentemente, limitantes inferiores baseados na visão geométrica. Outra forma de tentar atingir otimalidade dinâmica é caracterizar algumas sequências S de buscas, nas quais um determinado algoritmo *BST* tem custo $\mathcal{O}(\mathrm{OPT}(S))$.

Esta dissertação trata-se de um *survey* a respeito do problema de busca em Árvores Binárias 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 como segue. No Capítulo 2 formalizamos os conceitos e definições de Árvore Binária de Busca necessários 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.* [12] 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 apresentamos as considerações finais.

2 ÁRVORES BINÁRIA DE BUSCA E SEUS MODELOS

Neste capítulo, formalizamos os conceitos e definições de Árvore Binária de Busca necessários 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

Supondo que precisamos armazenar um conjunto R de n elementos de algum universo ordenado $\mathcal{U}$ e queremos realizar consultas do tipo: um certo elemento $x \in \mathcal{U}$ está em R? 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, denotado por esq(x) (dir(x)), enquanto x é o nó *pai* de esq(x) (dir(x)). O nó pai de x é denotado por $\text{pai}_T(x)$, ou simplesmente pai(x) quando está claro qual 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', $\text{pai}_{T'}(x) = \text{pai}_T(x)$. A subárvore contendo exatamente 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. Um *caminho* em uma árvore binária é uma sequência de nós $P = (x_1, x_2, \ldots, x_p)$ sem repetições, tal que ou $\text{pai}(x_i) = x_{i+1}$ ou $\text{pai}(x_{i+1}) = x_i$. Dizemos que um conjunto de nós Q *induz* uma subárvore 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.

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 $\mathrm{d_T}(x)$. A *altura de T* é a maior profundidade dentre todos os nós de T. Já 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 $\mathrm{d_T}(x) = d$. Além disso, os níveis são ordenados da menor profundidade para a maior. Portanto, a raiz pertence ao primeiro nível e o nó mais profundo da árvore pertence ao último nível.

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. Desse modo, 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, portanto, o menor ancestral comum em uma árvore T 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, para todo nó x que possui alguma de suas subárvores vazia, 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 vazias estão 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 aquelas que possuem altura mínima. Esses 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 de mostrar que $h = \lfloor \log_2 n \rfloor$, precisamos fazer algumas observações. Primeiro, podemos observar que em cada nível k de uma árvore binária existe 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.

A seguir, mostraremos que $h = \lfloor \log_2 n \rfloor$. Seja T' a árvore binária obtida de T pela remoção dos r nós do último nível. Como T é completa, T' é uma árvore binária cheia com altura

$h-1$ e, portanto, possui $n' = 2^{h-1+1} - 1 = 2^h - 1$ nós. Pela observação feita anteriormente, $1 \leq r \leq 2^h$. Assim, temos que:

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

□

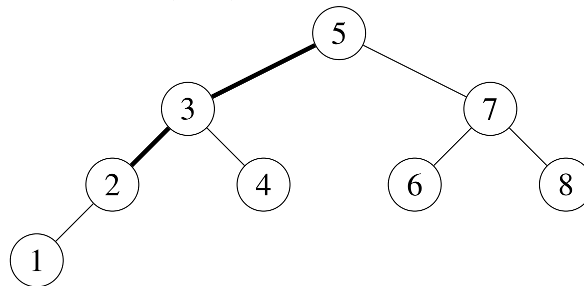
Lema 2.1.2 ([17]). *Se T é uma árvore binária completa, então T possui altura mínima.*

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, efetuamos a seguinte operação em T_1: retiramos uma folha w de seu último nível e tornamos w filho de algum nó v, localizado em algum nível menor do que o 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 maior. Dessa forma, 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 $\text{chave}(y_1) \leq \text{chave}(x)$ e $\text{chave}(y_2) \geq \text{chave}(x)$. 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 $\text{chave}(x) \leq \text{chave}(y)$, por exemplo. Observamos que em uma *BST* T, $\min\{x,y\} \leq \mathrm{LCA}_T(x,y) \leq \max\{x,y\}$. Dizemos que duas *BSTs* T' e T'' são *idênticas* se ambas são vazias ou se ambas possuem a mesma raiz r, $T'_E(r)$ é idêntica a $T''_E(r)$ e $T'_D(r)$ é idêntica a $T''_D(r)$.

Na Figura 1 temos uma representação visual de uma *BST* com o conjunto de chaves $\{1,\dots,8\}$. 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 possuem altura $\mathcal{O}(\log n)$. As *AVL Trees* e *Red-Black Trees*, 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.

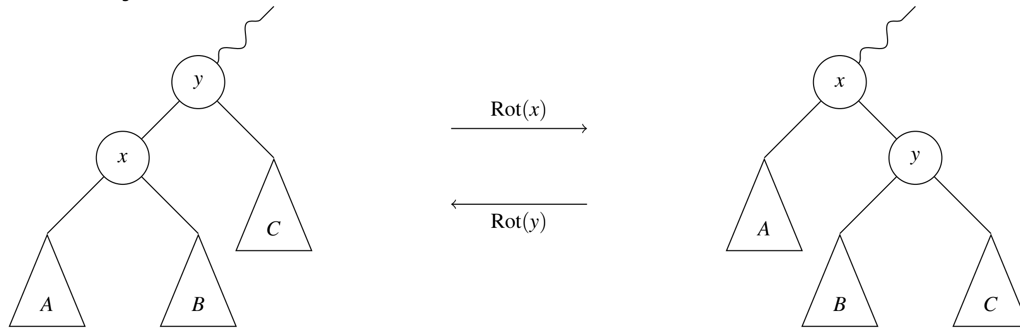
Definição 2.1.1 (Rotação)**.** *Se T_1 e T_2 são* BSTs*, tal que T_2 é resultante da rotação do nó x e seu pai y em T_1, então* $\text{pai}_{T_2}(y) = x$ *e* $\text{pai}_{T_2}(x) = \text{pai}_{T_1}(y)$*, se* $\text{pai}_{T_1}(y)$ *existir. Além disso, se x é filho esquerdo (direito) de y e z é o filho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

Uma rotação do nó x, denotada por $\text{Rot}(x)$, modifica a estrutura da *BST*, mas 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*. A seguir, na Figura 2 temos exemplos dos dois tipos de rotações, direita e esquerda.

2.2 Modelos *BST*

Para consultar se um certo elemento $x \in \mathcal{U}$ está em um conjunto R, podemos utilizar uma *BST* cujas chaves correspondem aos elementos de R. Esta consulta é chamada de *busca*. Para realizar uma busca em uma *BST* é utilizado um *ponteiro* z apontando para um nó da árvore,

Figura 2 – Rotação em *BST*.



Fonte: Elaborada pelo autor

de tal maneira podemos realizar as seguintes operações *BST* com z: movê-lo 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.

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. Uma busca por uma chave x em uma *BST* T consiste em fazer z apontar para a raiz de T, onde podemos realizar uma sequência de operações *BST* em qualquer ordem com z; no entanto, z deve visitar o nó com chave x em algum momento da busca.

Alguns modelos *BST* foram propostos. Wilber [15] definiu um modelo em que cada operação *BST* tem custo 1. Desta forma, o custo para buscar uma chave é a quantidade de movimentos do ponteiro mais a quantidade de rotações. Outro modelo *BST* é definido por Demaine *et al.* [12], onde o custo de uma busca é a quantidade de nós visitados pelo ponteiro durante a busca.

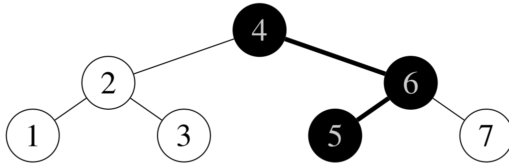
Na Figura 3 temos um exemplo de uma busca por 5, onde os nós visitados estão em destaque, e o nó 6 é rotacionado para a raiz da árvore. Na Figura 3a temos a *BST* em que a busca é realizada, e na Figura 3b, a *BST* resultante da busca. O menor custo possível para esta busca é 4, no modelo de Wilber, com a sequência de operações *BST*: $z \leftarrow \text{dir}(z)$, $z \leftarrow \text{esq}(z)$, $z \leftarrow \text{pai}(z)$ e $\text{Rot}(z)$, e custo 3 no modelo de Demaine *et al.*, por definição.

Os modelos *BST* de Demaine *et al.* e de Wilber são assintoticamente equivalentes. Mostramos esta equivalência no Lema 2.2.3, mas antes disso apresentamos o Algoritmo TRANSFORMAR, que transforma uma *BST* em uma árvore enviesada à 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. Esse algoritmo e esse lema são ferramentas para o Lema 2.2.3.

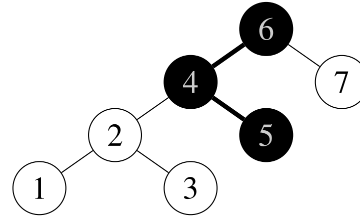
O Algoritmo TRANSFORMAR usa o Algoritmo LIMPAESQUERDA, que realiza ope-

Figura 3 – Exemplo de busca em uma *BST*.

(a) Busca pela chave 5 em T_1



(b) *BST* resultante T_2



Fonte: Elaborada pelo autor

rações *BST* para que a subárvore esquerda do nó apontado seja vazia, para isso, movemos o ponteiro para o filho esquerdo, se existir, e o rotacionamos. O algoritmo recebe uma pilha P para empilhar operações *BST*, onde armazena em P as operações *BST* reversas às realizadas. Além disso, tal algoritmo tem acesso ao ponteiro z do TRANSFORMAR. Para cada $z \leftarrow \text{esq}(z)$ e $\text{Rot}(z)$, são empilhadas as operações $z \leftarrow \text{dir}(z)$ e $\text{Rot}(z)$ pelo comando $\text{Empilha}(P, op_1, \ldots, op_n)$, onde a sequência de operações $(op_1, \ldots, op_n)$ é empilhada 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

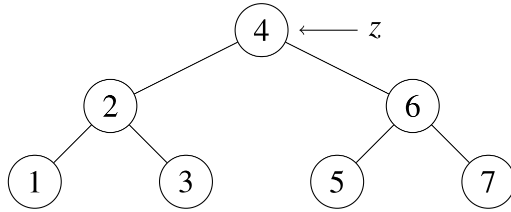
```
  Entrada: Pilha de operações BST P
1 início
2   enquanto T_L(z) ≠ ∅ faça
3     z ← esq(z)
4     Rot(z)
5     Empilha(P, z ← dir(z), Rot(z))
6   fim
7 fim
```

Na Figura 4, temos um exemplo da aplicação do Algoritmo LIMPAESQUERDA, em que, na Figura 4a, temos a *BST* T do Algoritmo TRANSFORMAR, e na Figura 4b, a *BST* resultante. A pilha P possui a seguinte sequência de operações *BST*: $z \leftarrow \text{dir}(z), \text{Rot}(z), z \leftarrow \text{dir}(z), \text{Rot}(z)$.

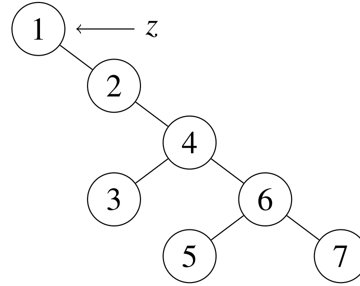
O Algoritmo TRANSFORMAR tem como entrada uma *BST* T e transforma T em uma *BST* enviesada à direita, o qual se inicia com um ponteiro z apontado para a raiz de T. O Algoritmo LIMPAESQUERDA é executado enquanto a subárvore esquerda de z não é vazia, e então z move-se para seu filho direito, se existir.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

Figura 4 – Exemplo da aplicação do Algoritmo LIMPAESQUERDA.

(a) *BST T* do Algoritmo TRANSFORMAR



(b) *BST* resultante



Fonte: Elaborada pelo autor

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 da* BST *resultante 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 no máximo* $6n-6$ *operações* BST*. Além disso, essa 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. Ademais, 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 é movido 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*.

Ademais 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, observamos 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. □

Esse resultado de Culik e Wood foi melhorado por Sleator *et al.* [23] para *BSTs* de tamanho pelo menos 11 ($n \geq 11$), para $2n-6$ rotações como descrito no Lema 2.2.2. A demonstração deste lema não é apresentada por ser complexa e não estar no escopo do trabalho.

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 assintótica do custo dos modelos *BST* propostos por Wilber [15] e por Demaine *et al.* [12] (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 após a 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 muito mais que isso. Como mostramos no Lema 2.2.3, um algoritmo *BST* pode realizar uma busca com $\mathcal{O}(|Q|)$ 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, onde Q é o conjunto dos nós visitados durante a busca por s.*

Demonstração. Como uma busca inicia na raiz da árvore e as operações *BST* são realizadas entre nós conectados (pai e filho), 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 disso, $s \in Q$. Portanto 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. Desta maneira, é possível fazer estas operações com custo $\mathcal{O}(|Q|)$. $\square$

Pela Observação 2.2.2 e pelo Lema 2.2.3 podemos concluir que o algoritmo $\mathcal{A}$ pode realizar uma busca por uma chave em uma *BST* com custo $\Theta(|Q|)$. Assim, dado o resultado anterior, no restante do texto, sempre que mencionado o modelo *BST*, nos referiremos ao modelo *BST* proposto por Demaine *et al.*, cujo o custo de uma busca é a quantidade de nós visitados.

3 ESTUDO DO PROBLEMA DE BUSCA EM BST

O *problema de busca em BST* é definido da seguinte forma: dada uma *BST* 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 [5] 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, esse 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 $\sum_{i=1}^{m}(d_T(s_i)+1) = m+\sum_{i=1}^{m} d_T(s_i)$. Denotamos por $\mathrm{cost}_T(S)$ o custo para realizar uma sequência de buscas S em uma *BST* T. Além disso, 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 $\mathrm{OPT}^{stat}(S)$, ou seja, $\mathrm{OPT}^{stat}(S)=\min_T \mathrm{cost}_T(S)$. Desse modo, dizemos que uma *BST* estática T é *ótima* para uma sequência de buscas S, se $\mathrm{cost}_T(S) = \mathrm{OPT}^{stat}(S)$. Notamos, portanto, que $\mathrm{OPT}^{stat}(S)$ 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 [4] também apresentou um algoritmo de programação dinâmica capaz de encontrar uma *BST* estática ótima, mas com custo $\mathcal{O}(n^2)$. Esse 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 $\mathrm{cost}_T(S) = \mathcal{O}(\mathrm{OPT}^{stat}(S))$. Levcopoulos *et al.* [27] também apresentaram um algoritmo de custo linear que 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 $\varepsilon > 0$. Essas árvores de custo $\mathcal{O}(\mathrm{OPT}^{stat}(S))$ 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 esse 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 o tempo para $\mathcal{O}(n\log n)$. Outra versão é quando a altura da árvore é limitada por uma constante L. Podemos observar que, $L \geq \lfloor \log_2 n \rfloor$, pelos Lemas 2.1.1 e 2.1.2. Para esta versão, Garey [30] propôs um algoritmo de custo $\mathcal{O}(Ln^2)$, onde as chaves são armazenadas nos nós folhas. 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 $\mathcal{O}(n^2)$. Portanto, este algoritmo de Becker melhora os resultados de Wessner e Itai, pois como $L \geq \lfloor \log_2 n \rfloor$. Já 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 esse problema é uma sequência de buscas S e uma *BST* T. Tal problema pode ser *offline* ou *online*, que é quando a sequência de buscas é conhecida antes de iniciar e quando são conhecidas uma a uma, respectivamente.

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

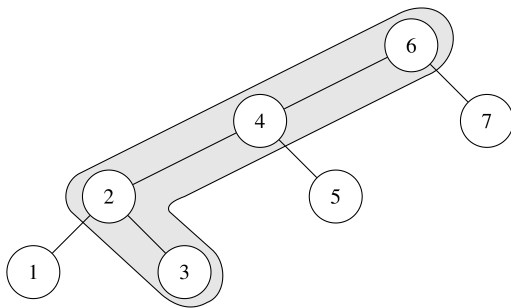
Definição 3.2.1 (Rearranjo)**.** *Sejam T_1 e T_2* BSTs *com o mesmo conjunto de nós e Q um subconjunto conexo dos nós de T_1 contendo* $\mathrm{raiz}(T_1)$. *Dizemos que T_1 pode ser rearranjada para T_2, se*

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 $T_1 \overset{Q}{\to} T_2$. Se Q contém todos os nós de T_1, denotamos simplesmente por $T_1 \to T_2$. O custo desta operação é $|Q|$. Na Figura 5a temos um exemplo de uma *BST* com o conjunto de nós $\{1,\dots,7\}$ e na Figura 5b uma *BST* resultante do rearranjo $T_1 \overset{Q}{\to} T_2$, onde $Q = \{2,3,4,6\}$ (os nós 4 e 2 são rotacionados, nesta ordem, em T_1 para obter T_2).

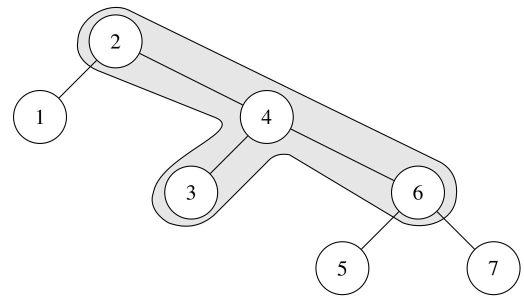
Figura 5 – Exemplo de rearranjo.

(a) *BST* T_1



$\longrightarrow$

(b) *BST* T_2



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 $S = (s_1,s_2,\dots,s_m)$ na *BST* T_0. Dizemos que $\mathcal{A}$ executa a instância (T_0,S) por uma *execução* $E = \langle T_0,Q_1,T_1,Q_2,T_2,\dots,Q_m,T_m\rangle$, se todos os rearranjos $T_{i-1} \overset{Q_i}{\to} T_i$ transformam T_{i-1} em T_i e $s_i \in Q_i$ para todo $i \in \{1,\dots,m\}$. 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 iniciá-las. Assim, por conhecer as próximas buscas, após cada uma delas, 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 $\min_{\mathcal{A}} \text{cost}_{\mathcal{A}}(T_1,S) \le \min_{\mathcal{A}} \text{cost}_{\mathcal{A}}(T_2,S) + \mathcal{O}(n)$, pois

antes de iniciar a execução, T_1 pode ser reestruturada com custo linear para ser idêntica a T_2 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* $\text{OPT}(S) \leq \text{OPT}^{stat}(S)$.

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 ($1 \leq i \leq m$) contém apenas os nós do caminho de busca de s_i e cada T_i é idêntica a T_0. □

Para este problema, Lucas [6] propôs um algoritmo *BST offline* guloso (Demaine *et al.* [12] 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 esses rearranjos para as buscas seguintes.

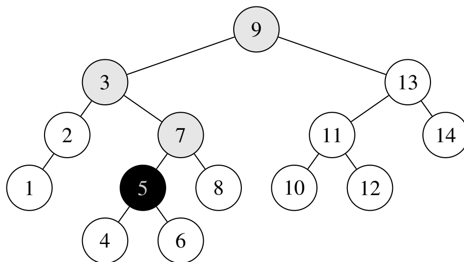
O *Greedy Future* executa uma instância (T_0, S), onde $S = (s_1, s_2, \ldots, s_m)$, por uma execução $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$, onde cada Q_i contém apenas os nós do caminho de busca por s_i em T_{i-1}. Seja S' a subsequência $(s_{i+1}, \ldots, s_m)$ e seja τ_i a subárvore induzida por Q_i em T_{i-1}, definimos uma *BST* τ'_i sobre os nós de Q_i, como a descreveremos a seguir, e construiremos T_i ao fazer o rearranjo $\tau_i \to \tau'_i$. Definimos τ'_i, recursivamente, a partir de Q_i e S', como segue: se $Q_i = \emptyset$, o algoritmo não executa nenhuma operação. Se $S' = \emptyset$, então escolhemos τ'_i como uma *BST* qualquer contendo 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 contém 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 $\text{raiz}(\tau'_i)$ é o nó de menor valor em R, e se R possui um segundo nó, então $\text{dir}(\text{raiz}(\tau'_i))$ é este nó. Em seguida, as subárvores esquerda do nó mínimo de R e direita do nó máximo de R, se existirem, em τ'_i são definidas repetindo este processo recursivamente para o subconjunto de nós $Q'_- = \{x \mid x \in Q_i \text{ e } x < \min(R)\}$, com a subsequência S'_- e para o subconjunto de nós $Q'_+ = \{x \mid x \in Q_i \text{ e } x > \max(R)\}$, com a subsequência S'_+, respectivamente. Notamos que, no

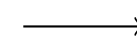
caso em que R possui dois nós, o filho esquerdo do nó máximo de R não está em Q_i, pois, caso contrário, esq(max(R)) $<$ max(R) e, portanto, max(R) não seria o sucessor de s em Q_i.

Na Figura 6 temos um exemplo de uma execução do *Greedy Future* em que é feita a busca pela chave $s_i = 5$ em T_{i-1} (Figura 6a) da sequência de buscas $S = (\ldots, 5, 8, 6, 13)$, onde os nós em destaque são os nós visitados. Desta forma, temos $Q_{i+1} = \{3, 5, 7, 9\}$, $S' = (8, 6, 13)$ e $R = \{7, 9\}$. Assim, 7 é a raiz de τ_i', pois é o menor valor em R, e 9 seu filho direito. Além disso, há, também, $Q_+' = \emptyset$, $Q_-' = \{3, 5\}$, $S_+' = (13)$ e $S_-' = (6)$. Em seguida, a subárvore esquerda de 7 é definida recursivamente pelo mesmo procedimento. Já para a subárvore direita de 9, o algoritmo não realiza nenhum procedimento, pois $Q_+' = \emptyset$. Assim, após as execuções recursivas, a *BST* T_i (Figura 6b) é definida.

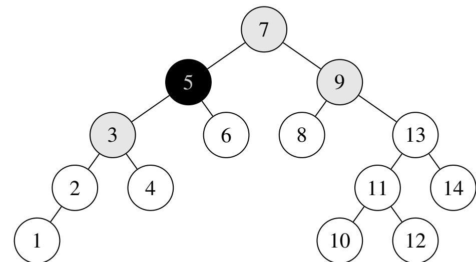
Figura 6 – Exemplo do Greedy Future.

(a) *BST* T_{i-1}





(b) *BST* T_i



Fonte: Elaborada pelo autor

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. Ademais, um algoritmo semelhante ao de Lucas foi proposto por Munro [7], 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. Dessa maneira, devido ao 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 comparado 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 [5, 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, $\text{cost}_{\mathcal{A}}(T,S) \leq f(n)\,\text{OPT}(S) + O(n)$. O fator aditivo $O(n)$ dá-se pelo motivo de que $\text{OPT}(S)$ é obtido de uma *BST* específica T^*. Dessa forma, como $\mathcal{A}$ recebe uma *BST* inicial T, $\mathcal{A}$ pode transformar T em T^* antes de iniciar sua execução com custo linear, como apresentado no Lema 2.2.1. 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 $\text{pai}(s)$ existir. O *Simple Exchange* é baseado na heurística *Transposition* proposta por Rivest [37] para o problema de busca em listas lineares. Allen e Munro mostraram que o custo deste algoritmo é $\Theta(\sqrt{n})$, se as chaves forem buscadas com probabilidades iguais e independentes.

Allen e Munro também apresentaram o algoritmo *Move to Root*, no qual a chave buscada é rotacionada até a raiz da árvore. Tal chave é rotacionada enquanto possuir nó pai, ou seja, enquanto não é a raiz da árvore. Esse algoritmo também é baseado em uma heurística para o problema de busca em listas lineares, a *Move to Front* [37, 38]. O custo do *Move to Root* é $\Omega(\sqrt{n})$, quando os acessos possuem distribuição de frequências independentes.

Sleator e Tarjan [2] 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 $\text{pai}(s)$, se ou s ou $\text{pai}(s)$ 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 7a, 7b e 7c, respectivamente, nas quais a chave buscada sempre é s. Sleator e Tarjan denominaram estas operações por *Zig*, *Zig-Zig* e *Zig-Zag*, respectivamente.

Sleator e Tarjan [5] 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* resulta em um fator constante menor em relação ao da Árvore *Splay* para algumas sequências de buscas.

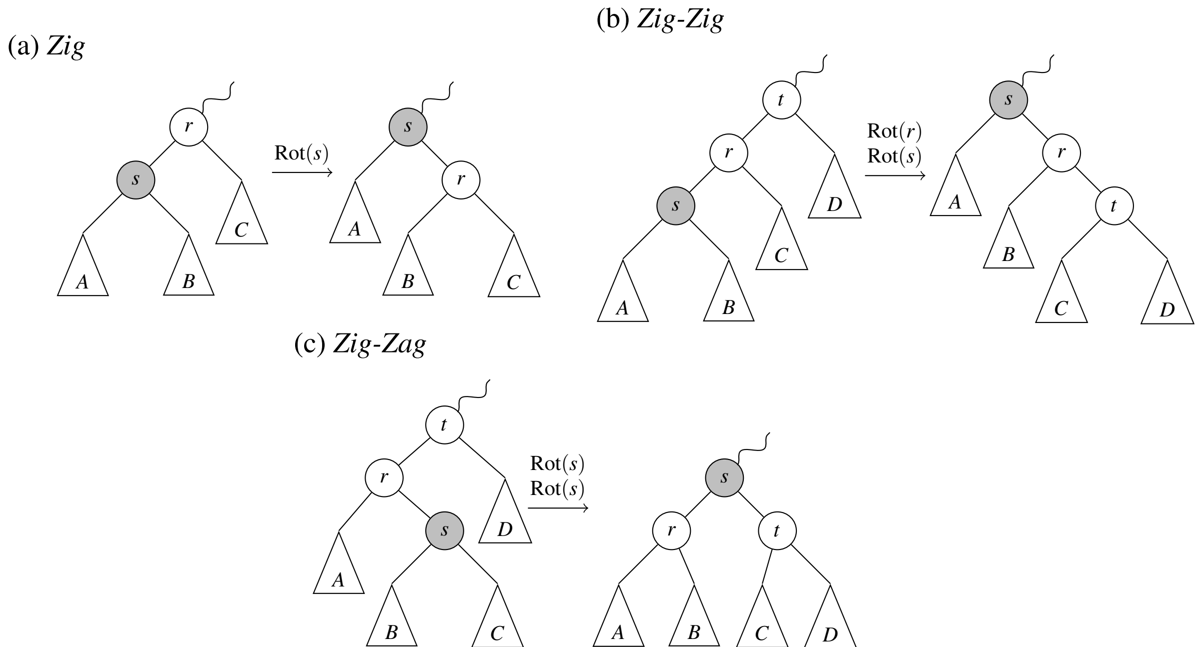
Subramanian [39] apresentou um algoritmo sugerido por Daniel Sleator, em comunicação pessoal, como um problema em aberto, em que apenas os nós do caminho de busca da chave buscada são visitados e a subárvore correspondente a este caminho é balanceada.

Algoritmo 3: SPLAY

Entrada: *BST* T e chave buscada s de T

1 **início**
2   **enquanto** $s \neq \text{raiz}(T)$ **faça**
3   **se** $\text{pai}(s) = \text{raiz}(T)$ **então** (1)
4             $\text{Rot}(s)$
5   **senão**
6   **se** $\text{pai}(s)$ *e s são ou ambos filhos esquerdos ou ambos filhos direitos* **então** (2)
7                 $\text{Rot}(\text{pai}(s))$
8                 $\text{Rot}(s)$
9   **senão** (3)
10                 $\text{Rot}(s)$
11                 $\text{Rot}(s)$
12   **fim**
13   **fim**
14   **fim**
15 **fim**

Figura 7 – Operações *Splay*.



Fonte: Elaborada pelo autor

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, essa 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.* [8] 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 esse custo para $\mathcal{O}(\log n)$.

Outras *BSTs online* semelhantes à Árvore Tango e de mesmo fator competitivo foram apresentadas, a *Multi-Splay Tree* e a *Chain-Splay Tree*, propostas por Wang *et al.* [9, 10] e Georgakopoulos [11], respectivamente. Ambas possuem ideias semelhantes à utilizada na Árvore Tango. Além destas, a *Skip-Splay Tree* [43] e a *Zipper Tree* [44] também são *BSTs online* de mesmo fator competitivo.

3.3 A conjectura da Otimalidade Dinâmica

Um algoritmo *BST online* que é $\mathcal{O}(1)$-competitivo (ou *constante-competitivo*) é dito *dinamicamente ótimo*. Sleator e Tarjan [2], quando propuseram a Árvore *Splay*, conjecturaram que sua árvore é dinamicamente ótima. Já Lucas [6] e Munro [7] 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 $\mathcal{O}(\log\log n)$ 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 podem 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 os algoritmos *offline*. (3) “Algum algoritmo *online* se comporta bem para qualquer sequência de buscas?”.

4 VISÃO GEOMÉTRICA

Para facilitar a visualização de uma execução *BST*, Demaine *et al.* [12] apresentaram uma representação de uma execução através de um conjunto de pontos no plano Cartesiano. Esta representação é chamada de *visão geométrica*. Demaine *et al.* definiram o problema do superconjunto de pontos arboreamente satisfeito (*Arborally Satisfied Superset* - ArbSS, do inglês) e mostraram 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 disso, descrevemos o problema ArbSS (Seção 4.3) e o algoritmo *online* proposto por Demaine *et al.*. Derryberry *et al.* [14] propuseram uma representação no plano semelhante à de Demaine *et al.*, independentemente.

4.1 Definição

Na visão geométrica definida por Demaine *et al.* [12], nos referimos às coordenadas x e y no plano pelas chaves contidas na *BST* $(1,\ldots,n)$ e o tempo que a chave é visitada $(1,\ldots,m)$, respectivamente. Um *ponto* a se refere a um ponto em $\mathbb{Z}^2$, 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,

$$\square ab = \{(x,y) \in \mathbb{Z}^2 \mid \min\{a_x,b_x\} \leq x \leq \max\{a_x,b_x\} \text{ e } \min\{a_y,b_y\} \leq y \leq \max\{a_y,b_y\}\}.$$

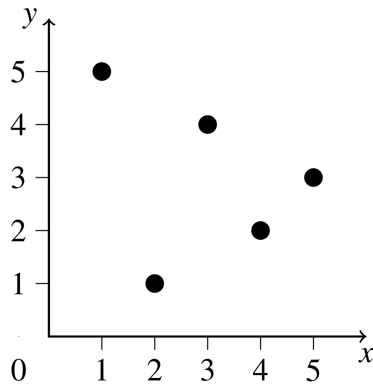
Dizemos que um ponto c está nos *lados incidentes* de a em $\square ab$ se c está no segmento de reta de $(a_x, \min(a_y,b_y))$ até $(a_x, \max(a_y,b_y))$, ou no segmento de reta de $(\min(a_x,b_x),a_y)$ até $(\max(a_x,b_x),a_y)$. Acrescentamos, ainda, que dois pontos $a,b \in P$ são *arboreamente satisfeitos* 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 $c \in \square ab$. Desse modo, um conjunto de pontos P é dito arboreamente satisfeito se todos os pares de pontos em P são arboreamente satisfeitos.

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}.

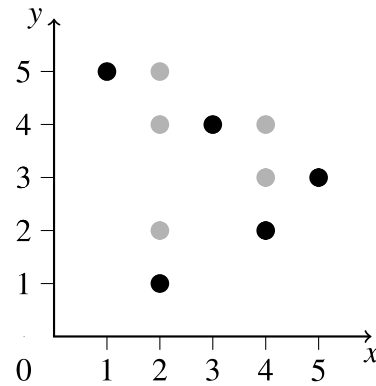
Abaixo temos um exemplo da visão geométrica de uma sequência de buscas e da visão geométrica de uma execução *BST*. Na Figura 8a temos a visão geométrica de $S = \{2,4,5,3,1\}$ e na Figura 8b temos a visão geométrica de uma execução *BST* $E = \langle T_0, Q_1, T_1, Q_2, T_2, Q_3, T_3, Q_4, T_4, Q_5, T_5 \rangle$ da instância (T_0, S). Além disso, temos a representação das *BSTs* T_0, T_1, T_2, T_3, T_4 e T_5 de E nas figuras 8c, 8d, 8e, 8f, 8g e 8h, respectivamente.

Figura 8 – Visão geométrica.

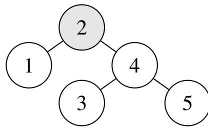
(a) Conjunto de pontos $P(S)$



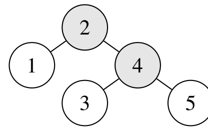
(b) Conjunto de pontos $P(E)$



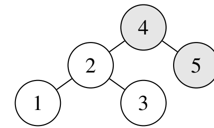
(c) T_0



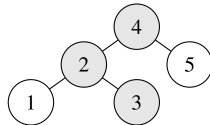
(d) T_1



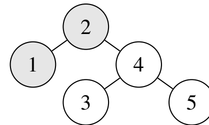
(e) T_2



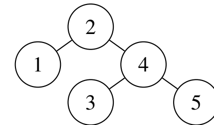
(f) T_3



(g) T_4



(h) T_5



Fonte: Elaborada pelo autor

Abaixo, apresentamos uma demonstração para o Lema 4.1.1 apresentado por Demaine *et al.* [12].

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. Esses pontos não são necessariamente distintos.*

Demonstração. Supomos que o resultado seja falso e $\square ab$ um retângulo que não satisfaz o lema e de área mínima e, além disso, 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 $\square ac$, que tem área menor do que $\square ab$, existe, pelo menos, um ponto nos lados incidentes a a. Logo, chegamos a uma contradição. $\square$

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.* [12]. Apresentamos esta equivalência de forma mais detalhada 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.*

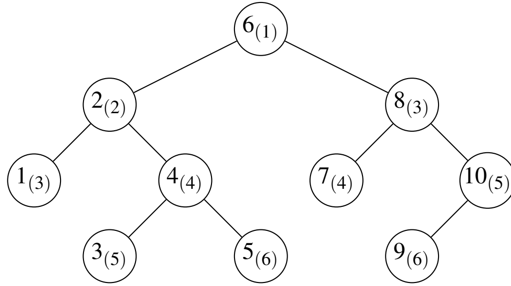
Demonstração. Seja $E = \langle T_0, Q_1, T_1, Q_2, T_2, \ldots, Q_m, T_m \rangle$ e sejam dois nós $x \in Q_i$ e $y \in Q_j$. Sem perda de generalidade, assumimos que $x < y$ e $i < j$. Se $\mathrm{LCA}_{T_{i-1}}(x,y) \neq x$, então para visitar x é necessário visitar $\mathrm{LCA}_{T_{i-1}}(x,y)$ no 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 $\mathrm{LCA}_{T_{j-1}}(x,y) = y$. Analisando os lados incidentes a (x,i) em $\square(x,i)(y,j)$, se existe um ponto (z,i) no lado horizontal, onde $x < z < y$, então $\square(x,i)(y,j)$ está satisfeito, e no lado vertical, além do ponto (x,i), existe um ponto (x,k), onde $i < k \leq j$, pois, na primeira vez que algum nó da subárvore enraizada em x é visitado, x também é visitado. Assim, $\square(x,i)(y,j)$ é satisfeito. $\square$

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

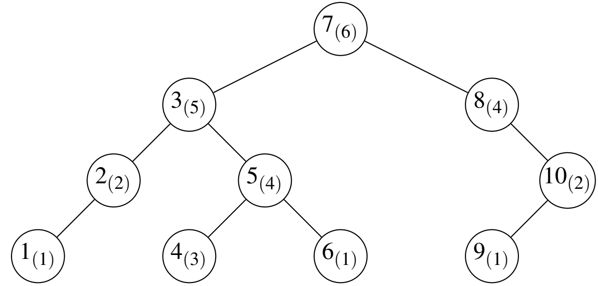
Uma *treap* é uma estrutura de dados que armazena pares (x,y) em cada nó de uma árvore binária T, de tal forma que T é uma árvore binária de busca com a propriedade de ordenação por x e ordenado pela propriedade *heap* binária por y. Uma *treap de mínimo* é uma *treap*, onde a *heap* binária é de mínimo,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

Figura 9 – Exemplos de *Treap*.

(a) *Treap* de mínimo.



(b) *Treap* de máximo.



Fonte: Elaborada pelo autor

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 $(a,\mu(a,1))$. Construiremos 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 exatamente 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 $T_{i-1} \overset{Q_i}{\to} T_i$, onde os nós de Q_i em T_{i-1} são rearranjados com base no próximo tempo de acesso, ou seja, Q_i induz uma *treap* de mínimo em T_i pelos pares $\{(a,\mu(a,i+1)) \mid a \in Q_i\}$.

O rearranjo $T_{i-1} \overset{Q_i}{\to} T_i$ mantém a propriedade de ordenação *BST* e, portanto, vamos mostrar apenas que a propriedade *heap* é mantida em T_i analisando todos os nós a em T_i. Nesse sentido, se a e pai(a) estão em Q_i, a propriedade é mantida por construção, e se a e 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 resta apenas o caso em que pai$(a) \in Q_i$ e $a \notin Q_i$. Dessa forma, se $\mu(a,i+1) \geq \mu(\text{pai}(a),i+1)$, a propriedade *heap* é mantida, 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 existem pontos, visto que assumimos que $\mu(a,i+1) < \mu(\text{pai}(a),i+1)$, nem no lado horizontal, uma vez se existe um ponto (c,i) no lado horizontal, temos que c pertence a subárvore enraizada em a e, portanto, a deveria ser visitado para visitar c, mas como $a \notin Q_i$, então $c \notin Q_i$. Assim, chegamos a uma contradição e, por fim, podemos notar que o caso em que pai$(a) \notin Q_i$ e $a \in Q_i$ não acontece, pois Q_i é uma subárvore induzida de T_{i-1}. $\square$

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

O *problema do superconjunto arboreamente satisfeito* (ArbSS) é encontrar um superconjunto arboreamente satisfeito P' de um conjunto de pontos P dado. Chamamos de algoritmo ArbSS um algoritmo projetado para encontrar um superconjunto arboreamente satisfeito P'. O custo de um algoritmo ArbSS $\mathcal{A}$ para encontrar P', denotado por $\text{costArbSS}_A(P)$, corresponde ao tamanho de P'. Denotamos por $minASS(S)$ o tamanho do menor superconjunto arboreamente satisfeito de $P(S)$ e 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* desse 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 $\{(s_1,1),(s_2,2),\ldots,(s_i,i)\} \cup P_1 \cup P_2 \cup \cdots \cup P_i$ seja arboreamente satisfeito.

A partir desse problema, Demaine *et al.* [12] mostraram que para qualquer algoritmo ArbSS *online* existe um algoritmo *BST online*. Para construir este algoritmo *BST online* é necessária a definição de *árvore split*, que 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*, remove x, deixando as suas subárvores esquerda e direita como árvores *split*.

Lema 4.3.1 ([12]). *Existe uma estrutura de dados de árvore* split *que suporta* MakeTree *e qualquer sequência de n* Splits *em tempo $\mathcal{O}(n)$, onde n é o tamanho da árvore.*

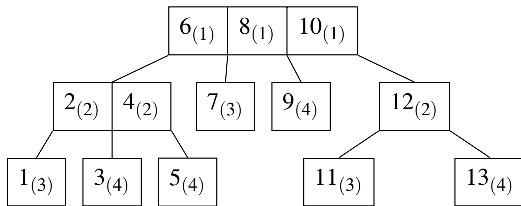
As árvores *split* podem mover um nó para a raiz e, em seguida, excluir este nó deixando duas árvores *split* com custo $\mathcal{O}(1)$ amortizado. Usando-as, no Lema 4.3.2 é mostrada a equivalência entre algoritmos ArbSS *online* e *BST online*. Esse resultado é enunciado por Demaine *et al.* [12].

Precisamos de algumas definições para o Lema 4.3.2. Uma *árvore de busca* é uma árvore usada para buscar chaves específicas dentre as chaves contidas na árvore. Em uma árvore de busca, um nó pode conter $k \geq 1$ chaves $ch_1,\ldots,ch_k$, onde $ch_1 \leq \ldots \leq ch_k$, permitindo que cada nó possua $k+1$ filhos, identificados por $f_0, f_1, \ldots, f_k$. Além disso, se y está na subárvore enraizada em f_0, então $y \leq ch_1$, se y está na subárvore enraizada em f_k, então $ch_k \leq y$, e se y está na subárvore enraizada em f_i ($0 < i < k$), então $ch_i \leq y \leq ch_{i+1}$. Uma *general treap* é uma

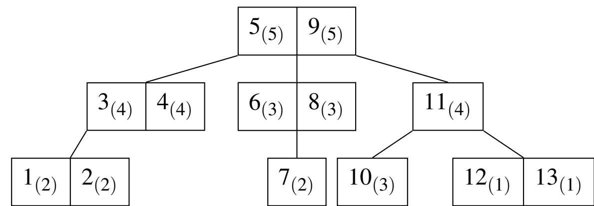
estrutura de dados que armazena pares (x,y) em uma árvore de busca de tal forma que as chaves da árvore são os valores de x e a árvore é ordenada com a propriedade *heap* pelos valores de y. Ademais, as chaves de um mesmo nó tem o mesmo valor de y, que chamamos do valor de y deste nó. Nesse sentido, uma general treap *de mínimo* é uma *general treap*, tal que se um nó é pai do outro, então o valor de y do nó pai é menor do que o valor de y do nó filho. Já uma general treap *de máximo* é uma *general treap*, tal que se um nó é pai do outro, então o valor de y do nó pai é maior que o valor de y do nó filho. Na Figura 10 temos o exemplo de uma *general treap* de mínimo (Figura 10a) e uma *general treap* de máximo (Figura 10b), onde em cada nó o número representa a coordenada x e o índice entre parênteses representa a coordenada y do par armazenado.

Figura 10 – Exemplos de *General treap*.

(a) *General treap* de mínimo.



(b) *General treap* de máximo.



Fonte: Elaborada pelo autor

Sejam G_1 e G_2 *general treaps* com o mesmo conjunto de chaves e Q um subconjunto conexo das chaves de G_1 contendo uma chave do nó raiz de G_1. Dizemos que G_1 pode ser rearranjada para G_2, se, para toda chave $x \notin Q$, suas subárvores, esquerda e direita, são idênticas as suas subárvores de x em G_2. Além disso, dizemos que um algoritmo $\mathcal{A}$ executa uma instância (G_0, S), onde G_0 é uma *general treap* e $S = (s_1, \ldots, s_m)$ uma sequência de buscas, por uma *execução* $E = \langle G_0, Q_1, G_1, Q_2, G_2, \ldots, Q_m, G_m \rangle$, se todos os rearranjos $G_{i-1} \overset{Q_i}{\to} G_i$ transformam G_{i-1} em G_i e $s_i \in Q_i$ para todo $i \in \{1, \ldots, m\}$.

Na visão geométrica, definimos 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 $\sigma(a,i) = -\infty$.

Abaixo, apresentamos uma demonstração de forma mais detalhada para o Lema 4.3.2.

Lema 4.3.2 ([12])**.** *Para qualquer algoritmo* ArbSS online $\mathcal{A}$*, existe um algoritmo* BST *online* $\mathcal{A}'$*, tal que em qualquer sequência de buscas* S *e* BST T*,* $\text{cost}_{\mathcal{A}'}(T,S) = O(\text{costArbSS}_A(P(S)) + n)$.

Demonstração. Vamos construir uma execução $E = \langle G_0, Q_1, G_1, \ldots, Q_m, G_m \rangle$, onde G_i é uma *general treap* de máximo definida por todos os pontos $(a, \sigma(a,i))$. Podemos observar, portanto, que G_0 possui um único nó com todas as chaves, pois $\sigma(a,0) = -\infty$ para todo a.

Os rearranjos de $G_{i-1} \overset{Q_i}{\to} G_i$, onde Q_i contém 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 nó 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 sucessor s em seu nó pai também devem ser visitados, se existirem. Isso 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 $\square((p, \sigma(p,i))(a,i))$, suponhamos 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 nó pai de a. Agora, se $\sigma(b,i) > \sigma(p,i)$, pela propriedade de ordenação *BST*, também temos uma contradição, pois como $\text{LCA}_{T_{i-1}}(p,b) \neq p$ e p é ancestral de a, então a está na subárvore esquerda de $\text{LCA}_{T_{i-1}}(p,b)$, mas $a > \text{LCA}_{T_{i-1}}(p,b)$. Dessa forma, não existe um ponto (b,k), onde $\sigma(p) < k < i$. Logo, p é o predecessor de a no seu nó pai e 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 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 nó raiz e, depois, enquanto houver algum nó visitado em algum nó filho da raiz, os movemos para o nó raiz. Dessa forma, temos um algoritmo *online*, pois a execução é baseada apenas em buscas passadas.

Ademais, iremos mostrar que o custo do algoritmo construído é limitado por um fator constante do custo de algum algoritmo ArbSS *online*. A estrutura é organizada de forma que para cada nó de G_i as chaves sejam armazenadas em uma árvore *split*, e os nós raiz de cada árvore *split* serão filhos de um nó folha da árvore *split* correspondente ao seu nó pai em G_i, obedecendo a propriedade de ordenação. Em cada busca, o algoritmo percorre G_i iniciando pela raiz e, para cada chave x visitada, é aplicado um $Split(x)$. Agora, aplicamos um *MakeTree* com todos os nós excluídos durante a busca para formar uma nova árvore *split* T' e colocamos T' na raiz de G_{i+1}. Conectamos as árvores *split* esquerda e direita restantes das operações de $Split(x)$

a T', de modo que as propriedades da árvore de ordenação de G_{i+1} seja mantida. Pelo Lema 4.3.1, cada *Split* custa tempo $\mathcal{O}(1)$ amortizado e cada *MakeTree* custa $\mathcal{O}(n)$, logo, o custo de uma busca no tempo i é uma constante vezes o número de pontos nessa linha. □

Demaine *et al.* [12] 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 e terminando na 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 8b temos o superconjunto resultante do *GreedyASS* para o conjunto de pontos representados na Figura 8a, nos quais os pontos cinzas representam os pontos adicionados pelo *GreedyASS*.

A decisão gulosa de visitar apenas o caminho da busca do *Greedy Future* é equivalente a adicionar o número mínimo de pontos na linha de varredura no *GreedyASS*, quando a *BST* inicial T_0 no *Greedy Future* é uma *treap* de mínimo sobre o conjunto de nós $\{(x, \mu(x, 0))\}$, onde $x \in \{1, 2, \dots, n\}$. No Lema 4.3.3 mostramos esta equivalência.

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

Demonstração. Suponhamos 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, assumimos 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*, suponhamos que o *Greedy Future* visitou o nó y na busca por x_i e o *GreedyASS* não adicionou o ponto (y, i). Seja $j = \sigma(y, i)$. Suponhamos que $j = -\infty$, ou seja, y não foi visitado antes do tempo i. Então, na construção de T_0, a prioridade de x_i é menor que a prioridade de y, uma vez que x_i é acessado antes de y. Assim, y não pode ser ancestral de x_i, pois, como y não foi visitado antes do tempo i, não pode ganhar novos descendentes, portanto, não é um ancestral de x_i no tempo i. Dessa forma, assumimos que $0 < j < i$. 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) tal 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 $\text{LCA}_{T_j}(y,x_i) \neq y$, então y não seria visitado no tempo i, visto que a subárvore enraizada em y não é modificada entre os tempos j e $i-1$, x_i não seria ancestral de y e, portanto, y não estaria no caminho de busca por x_i. Logo, como y não foi visitado no intervalo de tempo $(j+1,i)$, seus descendentes não mudam e, assim, pela propriedade de ordenação de *BSTs*, temos que y também é ancestral de z. Desse modo 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, que é no máximo i, e, assim, o próximo acesso de y é menor do que i. Dessa maneira, y está no caminho de busca de x_t, onde $j < t < i$. Isto contradiz que $j = \sigma(y,i)$, pois $j < t$.

Para o outro lado da prova, suponhamos 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. Observamos 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 $\square((y,j)(x_i,i))$. Agora em T_{j-1}, observamos que, se $\text{LCA}_{T_{j-1}}(y,x_i) \neq y$, então o nó $\text{LCA}_{T_{j-1}}(y,x_i)$ tem que ser visitado para visitar y e, como $y < \text{LCA}_{T_{j-1}}(y,x_i) \leq x_i$, o ponto $(\text{LCA}_{T_{j-1}}(y,x_i), j)$ satisfaria o retângulo $\square((y,j),(x_i,i))$, mas já mostramos que este ponto não existe. Assim, temos que $\text{LCA}_{T_{j-1}}(y,x_i) = y$. Como y não é visitado no intervalo de tempo $(j+1,i)$, seus descendentes não mudam neste intervalo, então y é ancestral de x_i em T_{i-1}. Logo, $\text{LCA}_{T_{i-1}}(y,x_i) = y$ e, portanto, y está no caminho de busca de x_i. Dessa forma, temos uma contradição. $\square$

Este resultado apresentado no Lema 4.3.3, juntamente com o resultado do Lema 4.3.2, é surpreendente, pois Lucas [6] e Munro [7] apresentaram o *Greedy Future*, independentemente, como um algoritmo *BST offline*, mas Demaine *et al.* [12] 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.

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 conjunto 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.

5 LIMITANTES

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

5.1 Limitantes superiores

Um limitante superior trivial para 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 OPT(S) e, analisando, é possível descrever alguns limitantes. Juntamente com a Árvore *Splay*, Sleator e Tarjan [5] mostraram alguns limitantes para árvores com o conjunto de chaves $\{1,\ldots,n\}$: o *Static Optimality Bound*, o qual 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 essa busca 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, ainda, que realizar uma sequência de buscas em uma Árvore *Splay*, onde a sequência é ordenada, tem custo $\mathcal{O}(n)$, e chamou esse limitante de *Sequential Access*. Sundar [46] e Elmasry [47] também mostraram tal resultado, o último com um fator constante menor. Já Sleator e Tarjan [5] 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 permanece aberta. Mais adiante, 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 um caso especial desta conjectura, que é 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 apresentou o limitante *Balance*

Bound, onde o custo total para realizar uma sequência de buscas é $\mathcal{O}((m+n)\log(n+m))$.

Sleator e Tarjan [5] apresentaram a conjectura do *Dynamic Finger Bound*, onde o custo de um algoritmo que satisfaz esta conjectura é $\mathcal{O}(m+n+\sum_{j=1}^{m-1}\log(|s_{j+1}-s_j|+1))$. Após 15 anos, Cole *et al.* [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 *et al.* [52] mostraram que o *Working Set Bound* é assintoticamente equivalente ao *Unified Bound*.

Lucas [6] conjeturou que seu algoritmo *Greedy Future* fornece uma aproximação de fator constante para $\text{OPT}(S)$. Munro [7] conjecturou que o *Greedy Future* tem custo $\text{OPT}(S)+\mathcal{O}(m)$. 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* e que o algoritmo *GreedyASS* é $\mathcal{O}(\log n)$-competitivo.

Para a *Multi-Splay Tree* [9] 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 $\mathcal{O}(m\log\log n+UB(S))$.

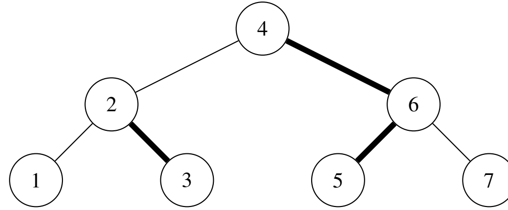
5.2 Limitantes inferiores

Um limitante inferior trivial para $\text{OPT}(S)$ é o tamanho da sequência de buscas, pois pelo menos um nó será visitado durante cada busca, mas esse limitante pode estar muito distante de $\text{OPT}(S)$. Wilber [15] foi um dos primeiros autores a estudar limitantes inferiores e propôs dois limitantes para $\text{OPT}(S)$. 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 o mesmo conjunto de chaves da *BST* inicial é usada. Para cada nó *y* em *P*, chamamos por *filho preferido*, o nó filho de *y* que foi visitado mais recentemente em uma busca. Se *y* não possui nós filhos ou nenhum de seus filhos foi visitado ainda, então *y* não possui filho preferido. Executando a sequência de buscas *S* em *P*, definimos como uma *alternação* cada vez que o filho preferido de um nó é alterado. O limitante inferior de alternações, denotado por $W_1(S)$, é a quantidade de alternações mais o tamanho de $|S|$.

Na Figura 11, temos um exemplo da *BST P* para a sequência $S=\{6,3,4,5,7,2,1\}$, onde os filhos preferidos definidos após a busca pela chave 5 estão em destaque. Para a sequência

de buscas S, temos $W_1(S) = 12$.

Figura 11 – Exemplo do primeiro limitante inferior de Wilber.

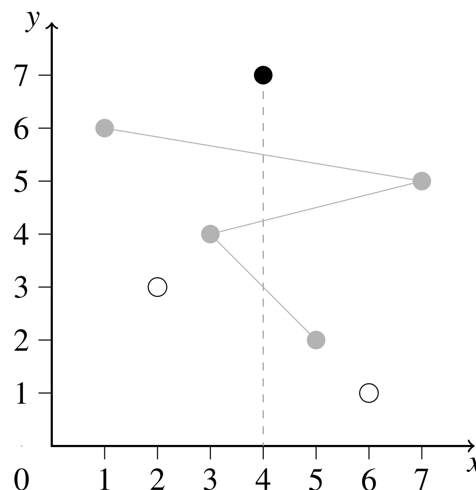


Fonte: Elaborada pelo autor

O segundo limitante proposto por Wilber não depende da árvore de limite inferior nem da *BST* inicial. Iacono [55] apresenta uma definição através da visão geométrica. Dado um ponto $(s_i, i) \in P(S)$, para uma sequência de buscas S, seja $F(s_i)$ o conjunto de pontos, com coordenada y menor do que i, que formam retângulos insatisfeitos com (s_i, i). Para cada $F(s_i)$, observamos os pontos, ordenados pela coordenada y, e contamos o número de alternâncias entre os lados esquerdo e direito de (s_i, i) que ocorrem. A soma das alternâncias para todo s_i em S fornece o limitante inferior. Este limitante é denotado por $W_2(S)$.

Na Figura 12 temos um exemplo do segundo limitante de Wilber para a sequência de buscas $S = (6, 5, 2, 3, 7, 1, 4)$, no qual temos a representação dos pontos de $P(S)$. Os pontos em cinza, são os pontos de $F(4) = \{(5,2), (3,4), (7,5), (1,6)\}$ e as linhas em cinza representam as alternâncias entre pontos do lado esquerdo e direito de $(4, 7)$, quando os pontos de $F(4)$ estão ordenados pela coordenada y.

Figura 12 – Exemplo do segundo limitante inferior de Wilber.



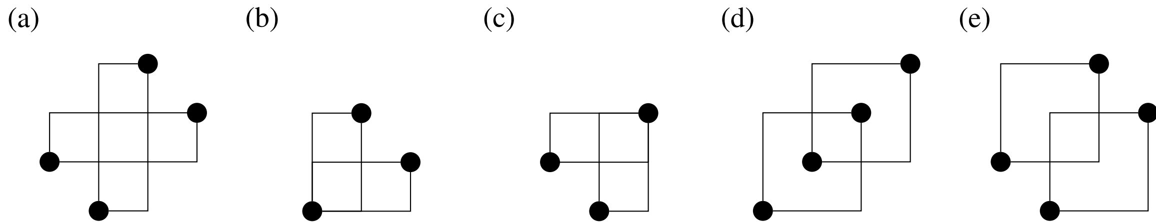
Fonte: Elaborada pelo autor

Wilber conjecturou que este segundo limitante domina o primeiro. Tal conjectura

ficou em aberto por 30 anos e foi provada em 2020 por Lecomte e Weinstein [16] que é verdadeira.

Outro limitante inferior foi proposto por Demaine *et al.* [12] usando a visão geométrica (apresentada no Capítulo 4), o *Limite de Retângulos Independentes* (*Independent Rectangles Bound* - IRB, do inglês), que é válido quando as coordenadas x dos pontos de P são distintas. 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. Denotamos, então, a quantidade máxima de retângulos independentes dois a dois em um conjunto de pontos P por $maxIRB(P)$. Temos exemplos de retângulos independentes nas Figuras 13a, 13b e 13c, e de retângulos dependentes nas Figuras 13d e 13e.

Figura 13 – Retângulos independentes e retângulos dependentes.



Fonte: Elaborada pelo autor

Demaine *et al.* apresentaram uma construção de conjuntos de retângulos independentes para cada limitante inferior proposto por Wilber. O Teorema 5.2.1, abaixo, mostra que a cardinalidade de um conjunto de retângulos independentes I é utilizada em um limitante inferior para $minASS(P)$, dado um conjunto de pontos P.

Teorema 5.2.1 ([12]). *Se um conjunto de pontos P, onde as coordenadas x dos pontos de P são distintas, origina 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, então* $\mathrm{OPT}(S) \geq \frac{|I|}{2} + |S|$.

Na representação geométrica apresentada por Derryberry *et al.* [14], é apresentado um limitante inferior chamado *Rectangle Cover Lower Bound*. Os autores também mostram que seu limitante é uma generalização dos dois limitantes de Wilber.

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

Para encontrar um superconjunto $\boxslash$-satisfeito de um conjunto de pontos P, Demaine *et al.* [12] propuseram o algoritmo guloso *Signed Greedy*, descrito abaixo. O algoritmo funciona de forma 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. Além disso, 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.* [12] mostram 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 [56] 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} a qual queremos minimizar.

As Inequações 6.1 e 6.2 são as restrições do primeiro modelo para garantir que o conjunto de pontos seja arboreamente satisfeito. A Inequação 6.1 garante a satisfação dos ⧄-retângulos e a Inequação 6.2 a satisfação dos ⧅-retângulos.

$$\min \quad \sum_{i=1}^{n}\sum_{j=1}^{m} b_{ij}$$

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

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

$$b_{uv} = 1 \qquad \forall\ (u, v) \in P \quad (6.3)$$

$$b_{ij} \in \{0, 1\}\ \forall\ 1 \leq i \leq n, 1 \leq j \leq m \quad (6.4)$$

Na restrição 6.1, se $b_{ij} = 1$ e $b_{rs} = 1$, então o retângulo ⧄$((i, j), (r, s))$ existe e este retângulo deve ser satisfeito por um ponto (k, l), diferente de (i, j) e (r, s), onde $i \leq k \leq r$, $j \leq l \leq s$. No termo $\sum_{k=i}^{r}\sum_{l=j}^{s} b_{kl}$, da restrição, deve existir algum $b_{kl} = 1$, senão $\sum_{k=i}^{r}\sum_{l=j}^{s} b_{kl} = 2$ e o lado esquerdo da desigualdade será maior do que 1. Ou seja, não existe nenhum ponto além dos

pontos (i,j) e (r,s) no ⧄$((i,j),(r,s))$ e, portanto, o retângulo não é satisfeito. A restrição 6.2, simetricamente a 6.1, garante a satisfação dos ⧅-retângulos.

Ademais, 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.* [13] definiram um novo modelo, onde as restrições garantem que exista pelo menos um ponto nos lados incidentes aos pontos que definem um retângulo. As Inequações 6.5 e 6.6 são as restrições do segundo modelo.

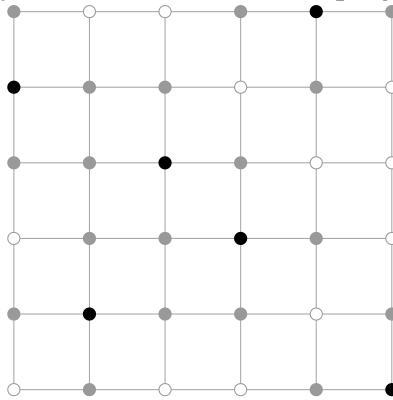
$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} b_{ij} & & \\
s.a. \quad & b_{ij}+b_{rs}-\left(\sum_{l=i+1}^{r-1}(b_{lj}+b_{ls})+\sum_{l=j+1}^{s-1}(b_{il}+b_{rl})+b_{is}+b_{rj}\right)\leq 1 & \forall\, i<r, j<s \quad & (6.5)\\
& b_{is}+b_{rj}-\left(\sum_{l=i+1}^{r-1}(b_{lj}+b_{ls})+\sum_{l=j+1}^{s-1}(b_{il}+b_{rl})+b_{ij}+b_{rs}\right)\leq 1 & \forall\, i<r, j<s \quad & (6.6)\\
& b_{uv}=1 & \forall\,(u,v)\in P \quad & (6.7)\\
& b_{ij}\in\{0,1\}\ \forall\ 1\leq i\leq n, 1\leq j\leq m & & (6.8)
\end{aligned}
$$

A Inequação 6.5 é a restrição que garante a satisfação dos ⧄-retângulos. Se $b_{ij}=1$ e $b_{rs}=1$, então o retângulo ⧄$((i,j),(r,s))$ existe e esse retângulo deve ser satisfeito por um ponto (k,l), diferente de (i,j) e (r,s), nos lados incidentes a (i,j) ou (r,s) no ⧄-retângulo $((i,j),(r,s))$. Na restrição, o termo $\left(\sum_{l=i+1}^{r-1}(b_{lj}+b_{ls})+\sum_{l=j+1}^{s-1}(b_{il}+b_{rl})+b_{is}+b_{rj}\right)$ soma as variáveis b_{kl} dos lados incidentes a (i,j) e (r,s) no ⧄-retângulo $((i,j),(r,s))$. Assim, se $b_{ij}=1$, $b_{rs}=1$, então o termo $\left(\sum_{l=i+1}^{r-1}(b_{lj}+b_{ls})+\sum_{l=j+1}^{s-1}(b_{il}+b_{rl})+b_{is}+b_{rj}\right)$ deve ser maior do que 0, ou então o lado esquerdo da desigualdade será maior do que 1. O termo $\left(\sum_{l=i+1}^{r-1}(b_{lj}+b_{ls})+\sum_{l=j+1}^{s-1}(b_{il}+b_{rl})+b_{is}+b_{rj}\right)>0$ significa que o ⧄-retângulo $((i,j),(r,s))$ é satisfeito, pois algum b_{kl} é 1 e, portanto, o ponto (k,l) existe. A restrição 6.6, simetricamente, garante a satisfação dos ⧅-retângulos.

Demaine *et al.* [13] mostraram que ambos os modelos, em sua relaxação linear, podem ser satisfeitos definindo todas as variáveis correspondentes aos pontos vizinhos (pontos cuja diferença ou da coordenada x ou da coordenada y é 1 no *grid*) dos pontos em P em $\frac{1}{2}$. Na Figura 14, temos um exemplo para o conjunto de pontos $P=\{(6,1),(2,2),(4,3),(3,4),(1,5),(5,6)\}$, onde os pontos pretos são as variáveis com valor 1, pontos cinza são as variáveis com valor $\frac{1}{2}$ e os pontos brancos são as variáveis com valor 0.

Esta solução será sempre viável. Observamos que cada restrição correspondente a retângulos formado por dois pontos de P será satisfeita, uma vez que eles contêm pelo menos outros 2 pontos de valor $\frac{1}{2}$. As restrições correspondentes a retângulos em que um canto é definido como 1 e o outro canto é definido como $\frac{1}{2}$ serão satisfeitas, pois existe pelo menos uma outra variável de valor $\frac{1}{2}$ (adjacente ao canto definido como 1), que será subtraída. Por fim, cada restrição correspondente a um retângulo com cantos definidos como $\frac{1}{2}$ será satisfeita diretamente. Com a relaxação linear dos modelos, dificilmente será encontrada uma aproximação, pois o intervalo para a integralidade é grande.

Figura 14 – Exemplo da relaxação linear dos modelos de programação linear.



Fonte: Elaborada pelo autor

6.2 Monotonicidade

Levy e Tarjan [56] tentaram provar que a Árvore *Splay* é dinamicamente ótima combinando os conceitos de monotonicidade e simulação de instâncias. Antes de introduzir estes conceitos, dizemos que R é uma *subsequência* de uma sequência S, se R é obtido através de uma sequência de eliminações de chaves de S. Por exemplo, $(2,4,6)$ é subsequência de $(\not{1},2,\not{3},4,\not{5},6)$. Dizemos que um algoritmo $\mathcal{A}$ é *aproximadamente monótono* (ou simplesmente *monótono*), se existe alguma constante $b > 0$ tal que $\text{cost}_{\mathcal{A}}(T,R) \leq b\ \text{cost}_{\mathcal{A}}(T,S)$ para toda sequência de busca S, subsequência R de S e árvore inicial T, onde $\text{cost}_{\mathcal{A}}(T_0,Z)$ 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*.

Levy e Tarjan [56] definem uma *simulation embedding* $\mathcal{S}_{\mathcal{A}}$ de um algoritmo *BST* $\mathcal{A}$ como um mapeamento de execuções de sequências de buscas em uma *BST* inicial T_0, tal que, para algum $c > 0$, $\text{cost}_{\mathcal{A}}(T_0,\mathcal{S}_{\mathcal{A}}(E)) \leq c \cdot \text{cost}_{A}(T_0,S)$, para toda sequência de buscas S, onde S é

uma subsequência de $\mathcal{S}_{\mathcal{A}}(E)$ e E é uma execução da instância (T_0, S).

A *simulation embedding* pode ser usada para simular qualquer execução E de uma determinada instância (T, S), inclusive uma execução cujo custo é $\text{OPT}(S)$. Assim, o custo para um algoritmo $\mathcal{A}$ executar uma simulação é, por definição, no máximo uma constante de vezes de $\text{OPT}(S)$. Além disso, pela definição de *simulation embedding*, a simulação de E contém S como uma subsequência. Se $\mathcal{A}$ também for aproximadamente monótono, o custo de E é no máximo um múltiplo constante do custo da simulação e, consequentemente, de $\text{OPT}(S)$. 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 ([56])**.** *Algoritmos aproximadamente monótonos com* simulation embedding *são* $\mathcal{O}(1)$*-competitivos.*

Para continuar, precisamos de algumas definições. *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 [57] e Lucas [58] 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. Na técnica apresentada por Levy e Tarjan [56] para tentar provar que a Árvore *Splay* é dinamicamente ótima, os autores assumem que as rotações no modelo *BST* são rotações restritas. Levy e Tarjan [56] 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 operações de *Splay*, como descrito no Teorema 6.2.2.

Teorema 6.2.2 ([56])**.** *Sejam T_1 e T_2* BSTs *de tamanho $n \geq 4$ com o mesmo conjunto de nós. Existe uma sequência de operações de* Splay *que transforma T_1 em T_2 com custo de no máximo $80n$.*

Dadas duas *BSTs* T_1 e T_2 com o mesmo conjunto de nós, denotamos por $\mathbb{T}(T_1, T_2)$ a sequência dos nós em que são aplicadas operações de *Splay* descritas no Teorema 6.2.2 para transformar T_1 em T_2. Se $T_1 = T_2$, então $\mathbb{T}(T_1, T_2) = (\text{raiz}(T_1))$. Além disso, denotamos por $S \bigoplus R$ a concatenação das sequências $S = (s_1, \ldots, s_k)$ e $R = (r_1, \ldots, r_l)$ em $(s_1, \ldots, s_k, r_1, \ldots, r_l)$.

A entrada de uma *simulation embedding* é uma execução $E = \langle T_0, Q_1, T_1, \ldots, Q_m, T_m \rangle$ de uma sequência de buscas S em uma *BST* inicial T_0. Seja τ_i a subárvore induzida por Q_i em T_{i-1} e τ_i' a subárvore induzida por Q_i em T_i.

Teorema 6.2.3 ([56]). *O mapa* $\mathcal{S}(E) \equiv \mathbb{T}(\tau_1, \tau_1') \bigoplus \ldots \bigoplus \mathbb{T}(\tau_m, \tau_m')$ *de uma sequência de buscas* $S = (s_1, \ldots, s_m)$ *é uma* simulation embedding *para a Árvore* Splay.

Demonstração. Por construção, na execução de cada bloco $\mathbb{T}(\tau_i, \tau_i')$ $(1 \leq i \leq m)$, τ_i é substituída por τ_i' em T_{i-1}. Como a Árvore *Splay* sempre rotaciona a chave buscada para a raiz da árvore, a última chave de $\mathbb{T}(\tau_i, \tau_i')$ é sempre s_i, o que significa que S é subsequência de $\mathcal{S}(E)$. Por fim, $\text{cost}(T, \mathcal{S}(E)) = \sum_{i=1}^{m} \text{cost}(T_{i-1}, \mathbb{T}(\tau_i, \tau_i')) \leq 80(|\tau_1'| + \ldots + |\tau_m'|)$. A desigualdade é dada pelo Teorema 6.2.2. O custo da execução inicial é $|\tau_1'| + \ldots + |\tau_m'|$. Portanto, 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, conclui-se que:

Teorema 6.2.4 ([56]). *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 os autores mostram que o valor de $\text{OPT}(S)$, para uma sequência de buscas S, é monótono. Este resultado é apresentado no Teorema 6.2.5.

Teorema 6.2.5 ([56]). $\text{OPT}(S)$ *é estritamente monótono.*

Demonstração. Seja R uma sequência estrita de $S = \{s_1, \ldots, s_m\}$ e T_0 a árvore inicial contendo as chaves de S. Seja $E = \langle T_0, Q_1, T_1, \ldots, Q_m, T_m \rangle$ uma execução ótima de S iniciando com a árvore T_0, construiremos uma execução E' de R, iniciando com T_0, de custo menor do que $\text{OPT}(S)$.

Observamos que R é formado por eliminações de algumas buscas de S correspondentes a um subconjunto de índices de 1 a m. Seja j o menor índice eliminado e seja $k > j$ o menor índice que não foi eliminado de S para formar R. Se k não existir, (isto é, as buscas $\{s_j, \ldots, s_m\}$ foram eliminadas), então definimos $E' = \langle T_0, Q_1, T_1, \ldots, Q_{j-1}, T_{j-1} \rangle$, e a execução de R está definida. Caso contrário, substituímos $T_{j-1} \stackrel{Q_j}{\to} T_j, \ldots, T_{k-1} \stackrel{Q_k}{\to} T_k$ por $T_{j-1} \stackrel{Q}{\to} T_k$, onde $Q = Q_j \cup \ldots \cup Q_k$. Os rearranjos $T_{i-1} \stackrel{Q_i}{\to} T_i$ $(j \leq i \leq k)$ são aplicados em ordem em T_{j-1}. Assim, definimos $E' = \langle T_0, Q_1, T_1, \ldots, Q_{j-1}, T_{j-1}, Q, T_k, \ldots, Q_m, T_m \rangle$. Veja que $|Q| < |Q_j| + \ldots + |Q_k| - (k - j)$ e como $k - j > 1$, o custo de $T_{j-1} \stackrel{Q}{\to} T_k$ é estritamente menor do que as operações $T_{j-1} \stackrel{Q_j}{\to} T_j, \ldots, T_{k-1} \stackrel{Q_k}{\to} T_k$. Repetimos esse processo para cada subsequência

contígua de buscas eliminadas de S. Dessa forma, é definida uma execução E' de R com custo menor do que E. Como E é uma execução ótima de S, então $\mathrm{OPT}(R) < \mathrm{OPT}(S)$ com a mesma árvore inicial. □

A partir do Teorema 6.2.5, conclui-se que:

Teorema 6.2.6 ([56]). *Se a Árvore* Splay *é dinamicamente ótima, então ela é aproximadamente monótona.*

Demonstração. Para toda subsequência R de S:

$$\mathrm{cost}_{Splay}(T,R) \leq c \cdot \mathrm{OPT}(R) \leq c \cdot \mathrm{OPT}(S) \leq c \cdot \mathrm{cost}_{Splay}(T,S).$$

□

Uma outra proposta para tentar provar a otimalidade dinâmica através da monotonicidade foi apresentada há mais de uma década por Harmon [59]. Em seu trabalho, o autor 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 $E = \langle T_0, Q_1, T_1, \dots, Q_m, T_m \rangle$ 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 $q_1, \dots, q_{|T_0|}$ são os nós de T em ordem crescente. A *simulation embedding* do *GreedyASS* é o conjunto de pontos $P(E) = \mathbb{G}(Q_1) \bigoplus \dots \bigoplus \mathbb{G}(Q_m)$. Desse modo, seu estudo prova que o *GreedyASS* adiciona no máximo $\mathcal{O}(|Q_1| + \dots + |Q_m|)$ pontos para satisfazer $P(E)$.

7 CONSIDERAÇÕES FINAIS

No presente capítulo, apresentamos as considerações finais acerca do estudo realizado sobre a otimalidade dinâmica do problema de busca em *BST*.

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 de 35 anos, desde que a conjectura foi apresentada.

Árvores Binárias de Busca são estruturas de dados fundamentais para a Ciência da Computação, que são utilizadas na prática em *kernels* de sistemas operacionais e em memória *cache* em redes de computadores [3]. 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.* [8] melhoraram esse fator competitivo para $\mathcal{O}(\log\log n)$, quando propuseram a Árvore Tango. Posteriormente, tal fator também foi alcançado por Wang *et al.* [9] e Georgakopoulos [11] 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 [4] e Yao [25].

A visão geométrica proposta por Demaine *et al.* [12] e, independentemente, por Derryberry *et al.* [14], é um resultado surpreendente, pois é uma maneira mais simples de visualizar uma execução *BST* e, além disso, o problema do superconjunto de pontos arboreamente satisfeito, da visão geométrica, é equivalente ao problema dinâmico de busca em *BST*. Contribuímos com uma prova dessa equivalência e outras provas relacionadas à visão geométrica, de forma mais detalhada do que as provas apresentadas por Demaine *et al.*

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

REFERÊNCIAS

1 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.

2 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.

3 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.

4 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.

5 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.

6 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-).

7 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.

8 DEMAINE, E. D.; HARMON, D.; IACONO, J.; PĂTRAŞCU, M. 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.

9 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.

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

11 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.

12 DEMAINE, E. D.; HARMON, D.; IACONO, J.; KANE, D. M.; PĂTRAŞCU, M. 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.

13 DEMAINE, E. D.; GANESAN, V.; KONTSEVOI, V.; LIU, Q.; LIU, Q. C.; MA, F.; NACHUM, O.; SIDFORD, A.; WAINGARTEN, E.; ZIEGLER, D. 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. In: GRANDONI, F.; HERMAN, G.; SANDERS, P. (Ed.). **28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)**. [*S. l.*]: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. (LIPIcs, v. 173), p. 68:1–68:21.

17 SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3rd. ed. [*S. l.*]: Ed. LTC, 2010. ISBN 9788521617501.

18 CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **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.; MANASSE, M. S.; RUDOLPH, L.; SLEATOR, D. D. 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.; KAPLAN, H.; KOZMA, L.; PETTIE, S.; ZWICK, U. 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.; HARMON, D.; IACONO, J.; KANE, D. M.; PĂTRAŞCU, M. 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.; DOUÏEB, K.; DUJMOVIC, V.; FAGERBERG, R. An $O(\log\log n)$-competitive binary search tree with optimal worst-case access times. In: KAPLAN, H. (Ed.). **Algorithm Theory - SWAT 2010**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 38–49. ISBN 978-3-642-13731-0.

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.; MISHRA, B.; SCHMIDT, J.; SIEGEL, A. 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 IACONO, J. In pursuit of the dynamic optimality conjecture. In: BRODNIK, A.; LÓPEZ-ORTIZ, A.; RAMAN, V.; VIOLA, A. (Ed.). **Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday**. [*S. l.*]: Springer, 2013. (Lecture Notes in Computer Science, v. 8066), p. 236–250.

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

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

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

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