New Bounds on Optimal Binary Search Trees

by

Dion Harmon

B.A. Mathematics
Cornell University, 2000

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philsophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author . . . .
Department of Mathematics
May 22, 2006

Certified by . . . .
Erik Demaine
Associate Professor of Electrical Engineering and Computer Science
Thesis Advisor

Accepted by . . . .
Rodolfo Ruben Rosales
Chairman, Applied Mathematics Committee

Accepted by . . . .
Pavel I. Etingof
Chairman, Department Committee on Graduate Students

New Bounds on Optimal Binary Search Trees

by

Dion Harmon

Submitted to the Department of Mathematics
on May 22, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philsophy

Abstract

Binary search trees (BSTs) are a class of simple data structures used to store and access keys from an ordered set. They have been around for about half a century. Despite their ubiquitous use in practical programs, surprisingly little is known about their optimal performance. No polynomial time algorithm is known to compute the best BST for a given sequence of key accesses, and before our work, no $o(\log n)$-competitive online BST data structures were known to exist.

In this thesis, we describe tango trees, a novel $O(\log \log n)$-competitive BST algorithm. We also describe a new geometric problem equivalent to computing optimal offline BSTs that gives a number of interesting results. A greedy algorithm for the geometric problem is shown to be equivalent to an offline BST algorithm posed by Munro in 2000. We give evidence that suggests Munro's algorithm is dynamically optimal, and strongly suggests it can be made online. The geometric model also lets us prove that a linear access algorithm described by Munro in 2000 is optimal within a constant factor. Finally, we use the geometric model to describe a new class of lower bounds that includes both of the major earlier lower bounds for the performance of offline BSTs, and construct an optimal bound in this new class.

Thesis Supervisor: Erik Demaine
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

Finishing a thesis can be mysterious time of life. Excitement and anticipation for the future mix with a melancholy contemplation of the difficult past. I could not have gotten here without a substantial network of social support.

Family. My parents, Bruce and Bonnie, and my sister, Alyssa, have given me strong, unwavering support for many years. They encouraged me to follow my interests despite setbacks and uncertainty. Particularly during the past few years, they helped me keep a steady and healthy perspective on life.

Colleagues. My professional colleagues deserve many thanks. The research for this thesis was undertaken in close collaboration with Erik Demaine, John Iacono, and Mihai Pătraşcu. I thank them for many fruitful discussions and correspondence, and for their continuing enthusiasm.

Before I started work with Erik, Martin Bazant and Ruben Rosales strongly encourged to continue work despite many setbacks. I also thank Linda Okun for helping me through administrative hurdles.

Collocateurs. I feel that in this sphere, my time here at MIT has been extraordinarily fortituitous. Most of my collocateurs[1] have been exceedingly amiable and interesting people with whom I continue to enjoy lasting friendships.

- My first roommates at Ashdown: John Bible then Charles Dumont.
- My last roommate at Ashdown: Alexandre Parisot. He is an entertaining and often exasperating Parisian. Our shared love of wine and good food and his continuing adventures have kept me in good spirits for years.
- My official (and unofficial) housemates at 26 River Street deserve many thanks. Caroline Boudoux and Israel Veilleux, the inimitable couple from Quebec. I am

[1] *Collocateur* is French word. It means, "one you live with," including both roommates and housemates.

thankful for Michelle Guerette and her intense and contagious enthusiasm in everything she does. Dr. Reejis Stephen was always ready for a searing curry and a good argument, particularly when I was feeling out of it. Kimberly Insel was a latecomer to our little home; she brought a playful and positive view to our house at a time when the rest of us were feeling a bit little tired.

With all I have enjoyed sharing many good meals, discussions, and adventures.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents new bounds on the behavior of binary search trees. This work was initially undertaken in an attempt to prove or disprove the dynamic optimality conjecture posed by Sleator and Tarjan [30] in 1985. Though we neither proved nor disproved the conjecture, the tools and methods of attack we discovered may prove useful for latter attempts at the problem.

There are two major results presented: the box model with related lower bounds, and the tango algorithm. The box model provides a new geometric method of analyzing the behavior of binary search trees. The box model provides interesting results bearing on Munro's paper on linear search and optimal BST search from 2000 [27]. In particular, we provide evidence that suggests that the offline BST algorithm Munro proposed is dynamically optimal and may be made online. We also use the box model to describe a new class of lower bounds called *cut bounds* or *independent set bounds*, and show that most earlier lower bounds are in this class. Moreover, we show how to compute the largest lower bound of this class, within a constant factor. This implies that it is the best known lower bound for the behavior of offline binary search trees, although we cannot show that it is strictly better than the second bound of Wilber [33].

The tango algorithm is an online $O(\log\log n)$-competitive binary search tree algorithm. It is the first online $o(\log n)$-competitive BST algorithm.

Collaboration. The work in this thesis is the result of a close collaboration with following people: Erik D. Demaine, John Iacono, and Mihai Pătraşcu. In particular,the results from Chapter 4 appeared in FOCS05 [10].

1.1 Outline of Thesis

Chapter 1 introduces some notation and conventions for BSTs and reviews the literature. We then introduce the box model in Chapter 2, and results related to Munro's 2000 paper [27]. The box model requires a bit more notation than provided in the introduction. (See Appendix A.) Chapter 3 presents the lower bounds associated with the box model, including forms equivalent to the first and second bounds of Wilber [33]. A slightly modified version of the tango algorithm [10] is presented in Chapter 4. This presentation uses a modified cut bound to provide the required lower bound, showing that tango is $O(\log \log n)$-competitive. Chapter 5 provides some open questions, and outlines a method that may be useful for using geometric algorithms to find a polynomial time dynamically optimal BST algorithm.

1.2 Basic Concepts

In this section we cover some basic concepts, notation, and conventions for the thesis, and point out some alternate conventions used in the literature. The basic BST is described in Sections 1.2.1 and 1.2.2 and the BST access problem is covered in Section 1.2.3. Section 1.2 finishes with a brief note on different types of BST algorithms.

1.2.1 Binary search trees

1.2.1.1 Basic structure

A *binary search tree* (BST) is a binary tree in which each node has at least four associated pieces of information:

- A key value. The key value is an element from an ordered set. In general the key need not be a number, but in this thesis we assume that it is an integer from a fixed universe of keys: the integers 1 through n. The key value of a node p is denoted by $key(p)$. Each key is unique in the BST. We usually do not distinguish between a node and its key.

- A *parent node*. This is either the special value *null*, or another node in the BST. One and only one node in a BST will have a null parent at any time, the *root* node of the BST.

- A *left child* and *right child*. Either or both of these may also be null.

In some cases, we store additional bits of information in each node of the tree, related to the structure of the tree.

1.2.1.2 Ancestors, descendants, and subtrees

A node p is the *ancestor* of a node q if and only if there is a sequence of nodes $(s_1, \ldots, s_k)$ such that $s_1 = p$, $s_k = q$, and s_i is the parent of s_{i+1}. In this case, the node q is a *descendent* of p. The *subtree* of a node p is the empty set if p is null, or the tree including p and all p's descendants otherwise. The *left subtree* of a node is the subtree of its left child and the *right subtree* is the subtree of its right child.

1.2.1.3 Depth and lowest common ancestor

The *depth* of a node is the number of edges on the unique path from the node to the root. The *lowest common ancestor* or *lca* of a set S of two or more nodes is the node p with greatest depth in the BST such that p contains all the nodes in S in its subtree.

1.2.1.4 Key order invariant of BSTs

This invariant makes a tree a binary search tree. All nodes in the left subtree of a node p in a BST must have key values less than p, and all nodes in p's right subtree must have key values greater than p. For this reason, we say that nodes with values less than p are *left* of p, and similarly nodes with key values greater than p are *right* of p.

1.2.1.5 Example

A BST on the integers 1 through 8 is shown in Figure 1-1.

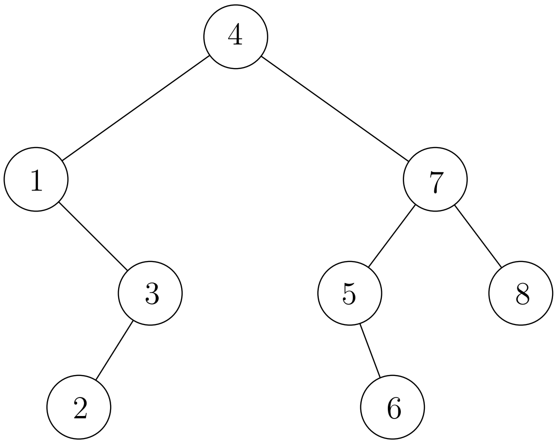

Figure 1-1: A BST on integers 1 through 8.

1.2.2 Searching and structural changes in BSTs

Although BSTs are used for many reasons, their fundamental operation involves repeatedly looking for keys in the BST, and often modifying the structure of the BST. In some applications, keys are added or removed from the BST. However, in this thesis, we assume that the keys in a tree do not change: the BSTs we consider contain a fixed and known set of keys, and we never look for keys not in this set. Most of the results presented in this thesis can be extended to the case where we can add and remove keys, and search for keys not present in the tree. See papers by Sleator and others [12], [5] for some ideas on how to perform these extensions for the tango algorithm, for example.

1.2.2.1 Searching for nodes

The order invariant allows us to search for keys in the tree. In the model we assume in this thesis, all searches begin at the root of the BST. We can find a node in the tree in the following manner: if the search key is in the root node, the search terminates. If the search key is less than the root, we search the left subtree, otherwise we search the right subtree. The order invariant for BSTs guarantees such a search will terminate by finding a node with the desired key if the BST contains such a node. Moreover, the only nodes we encounter in such a search are on the path from the root to the node with the desired key.

Other models for searching exist. In a model described by Wilber [33], for example, one search begins at the node where the last search left off. In such a case, we might sometimes take parent pointers to lower depth before we again descend to the node with the desired key. However, most common search models have costs within a constant factor of the model assumed in this thesis. (For information on cost, see Section 1.2.3.)

1.2.2.2 Changing structure in BSTs

There are many BSTs for a fixed set of nodes: the number of distinct BSTs on k nodes is the kth Catalan number, or $\binom{2k}{k}/(k+1)$. For 4 nodes, for example, there are 14 trees, as displayed in Figure 1-2.

When using BSTs, we often wish to change between the available trees in order to reduce the depth of keys we access at a later time. Rotations are a common method of changing the structure of BSTs. We rotate edges. If we rotate the edge between q and its parent p, the node q becomes the parent of p, and the other edges involving p and q are rearranged to maintain the order invariant. (See Figure 1-3.)

The node p in the left tree, or q in the right tree in Figure 1-3, may either be the root of the BST, or may have a parent in the BST. When applying a sequence of rotations to a tree, we apply them in order. The sequence is a *valid* rotation sequence when applied to the tree if the ith rotation in the sequence rotates an edge that exists

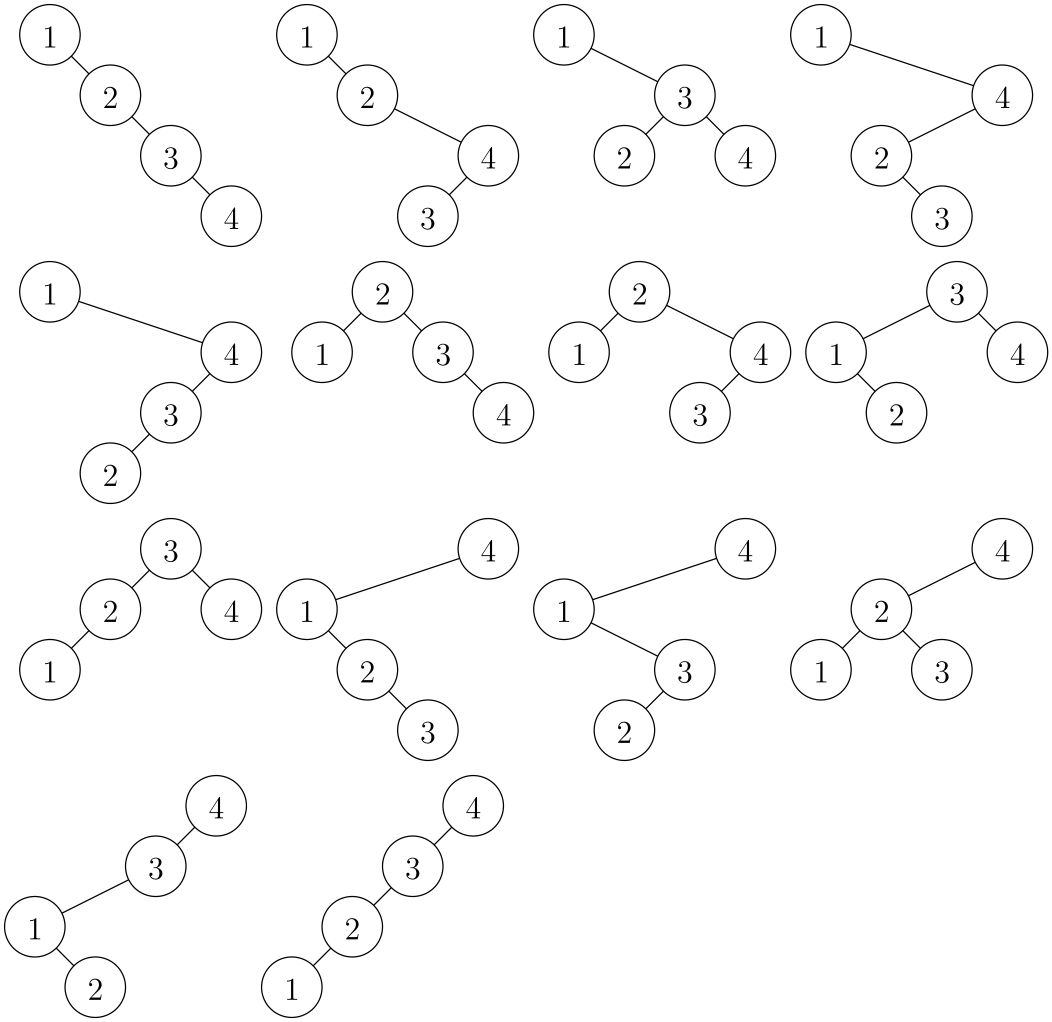

Figure 1-2: All 14 BSTs on four nodes.

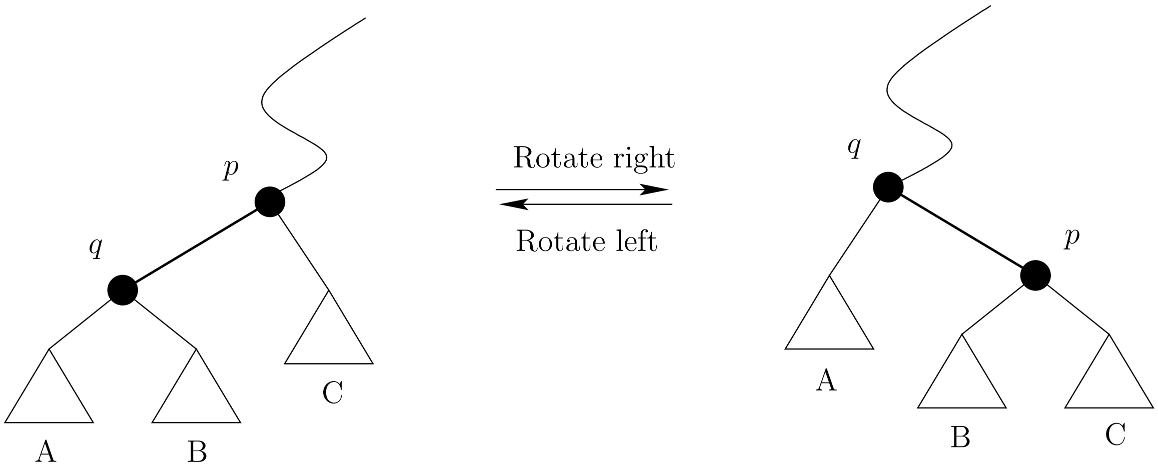

Figure 1-3: Rotating an edge.

in the tree after the first $i-1$ rotations are applied.

In Chapter 2 we introduce a different method of describing changes to the BST, called reorganization trees. Appendix A describes reorganization trees in detail.

1.2.3 The BST access problem

Let S be a sequence of integers in the range 1 through n, with each integer occurring at least once. An *access service sequence* for S is a pair (T, Λ) of a BST T on the integers and a sequence Λ of rotations and keys. T is the *initial tree*, and Λ is the *service sequence*, or *sequence*. The subsequence of keys in Λ must be S, and the subsequence of rotations in must be valid when applied to T. A key in Λ is an *access* of the access service sequence. The index of an access in the subsequence of accesses is the *time* of the access.

1.2.3.1 Cost of an access service sequence

Let (T, Λ) be an access service sequence for S. Let Λ_s be the sth item of Λ. Let T_s be the structure of T when all rotations at or before Λ_s have been applied to T. The *access path* for an access Λ_s is the root-to-node path of the node Λ_s in T_s, including Λ_s. The *cost* of an access Λ_s is the number of nodes in its node-to-root path. The *search cost* of an access service sequence is the total cost of all accesses in its access sequence. The *cost* of an access service sequence is the number of rotations in its access sequence plus the search cost of the sequence. The *BST access problem* posed by a sequence of keys, S, is to find an access service sequence with smallest cost.

1.2.4 BST algorithms

A *binary search tree algorithm*, or *BST algorithm*, is an algorithm that computes an access service sequence given a sequence of keys. There are many different kinds of BST algorithms.

1.2.4.1 Online BST algorithms

An *online* BST algorithm is given the keys of a sequence one at a time. It must compute the initial tree before it gets the first key of the sequence, and must access each node in the current tree before it gets the next key. The algorithm has no knowledge of the sequence before it begins, or of any future keys before it finds the current key in the tree.

1.2.4.2 Strict online BST algorithms

A *strict online* BST algorithm has restrictions on what it can do, with the result that it runs in time proportional to the cost of the access service sequence it produces (at least in many memory models). This model is assumed implicitly in many places, particularly in the initial discussion of splay trees [30]. (It is is explicitly described in [10].) Strict online BST algorithms store information between accesses using the BST. The information stored in the BST includes the BST structure and $O(\log n)$ or "constant extra" bits in each node.[1] Strict online BST algorithms have one pointer

[1] Although $O(\log n)$ is not constant, it is often called "constant extra space" as information related to keys in the tree takes $O(\log n)$ bits to store, and we assume that we can manipulate keys in $O(1)$ time. However, this means we can also store

into the tree. The pointer is initialized to the root of the BST before searching for a key and then the algorithm can perform the following operations[2]:

- Move the pointer from the current node to the left, right, or parent of the current node, read and change the data in the node to which it is moving.

- Perform a rotation of the edge between the current node and its parent.

Once the algorithm has found the key it is searching for, it can continue to perform these operations before it gets the next key of the sequence and begins searching for it. The algorithm should take take $O(1)$ time to perform any of the operations above.

The *cost* of the strict online BST algorithm is the number of rotations, plus the number of edges it traverses by moving its pointer from node to node. This is a common definition of the cost of a BST algorithm, but provided the algorithm does not wander about the tree too much without making structural changes, this cost is within a constant factor of the cost as defined in Section 1.2.3. An argument from Lucas [24] implies that we do not have to wander about too much, so any BST algorithm that does is inefficient, and may be replaced by an algorithm that achieves equivalent structural changes with lower cost. This argument is discussed briefly in Appendix A.

1.2.4.3 Offline BST algorithms

An *offline* BST algorithm gets the whole sequence at once, and can use arbitrary additional memory and computation to produce a BST access sequence. The cost of an offline BST algorithm on a given sequence of keys is the cost of the access service sequence it computes for the keys, not the time the algorithm takes to run.

information about depth or number of nodes in a subtree, for example, as these also take $O(\log n)$ bits.

[2] As noted, other models for searching exist: Wilber [33] describes a model where the pointer is not initialized to the root at the beginning of each search.

1.2.4.4 Static BST algorithms

A *static* BST algorithm never includes rotations in any access service sequence it produces, so the only freedom it has in picking the initial tree for the sequence. This is in contrast to *dynamic* BST algorithms which can include rotations.

1.3 Previous Work

The review of previous work is similar to that found in the introduction of the initial tango paper [10], though it is not quoted verbatim. Research on BSTs has been going on about half a century[3]. We divide previous progress into three categories:

- *Upper bounds.* These are upper bounds on the behavior of specific BST algorithms. Many, perhaps most, are related to splay trees [30].
- *Lower bounds.* These are primarily due to Wilber [33], but are proved elsewhere as well [17], [28]). See Sections 3.3 and 3.4 for detailed descriptions and a generalization of these bounds. Note that this bound generalization was discovered independently [12], [11]), though without the box model.
- *Assorted optimality results.* These give optimal algorithms for restricted classes of BST algorithms, or for slightly different cost models.

1.3.1 Upper bounds

The upper bound results come in a variety of flavors. Some BST algorithms have upper bounds on search and rearrangement cost *per access*, such as RB trees, AVL trees, or even static balanced BSTs. (See [9] Chapter 14, [1], and [21] respectively.) These BSTs are all online and usually guarantee search cost $O(\log n)$ for each access and also for other modifications such as adding and splitting the trees. However these

[3] BSTs were discovered by a number of independent authors in the 1950's ([9] in reference to [22])

trees cannot achieve better costs per access even for serial access of the keys (which has cost at most $O(n)$).

Other online BST algorithms give upper bounds for the total cost over the whole sequence, using amortization: a way of averaging the access costs. One of the best-known of the latter is the splay tree algorithm [30]. Many upper bounds have been proved on the behavior of splay trees, and many of these may be extended to more general (online) rearrangement heuristics [31], [15].

Let $S = (s_1, s_2, \ldots, s_m)$ be a sequence of keys of length m with n distinct keys. The run time of a splay tree on S has certain upper bounds. Many of these were proved in the classic paper by Sleator and Tarjan [30]:

- *Static optimality.* If every key in the tree is accessed at least once, the total cost for the splay tree is within a constant factor of the optimal static BST algorithm. Any BST with this property is *statically optimal*.

- *Static finger.* Let f be a key in the tree. This is the fixed "finger." For key i, let $d(i, S)$ be number of keys tree between i and f (including the end points). The static finger property says that the splay algorithm has cost $O(n \log n + m + \sum \log d(i, S))$. This holds for any fixed f in the tree.

- *The working set.* Let $d(S, i)$ be the number of distinct keys in S between s_i and the access of key s_i just before s_i in S, or between s_i and the beginning of S if i is the first index where the key s_i occurs. The *working set cost* is $O(\sum \log(d(i, S) + 1))$; let $W(S)$ be the working set cost on S. The splay tree algorithm on S has cost $O(n \log n + m + W(S))$, where $\sum \log(d(i, S) + 1)$. Once can think of the working set for an access s_i as the number of distinct elements we accessed since last searching for the key s_i.

Others properties of splay trees were proved a little later. The dynamic finger property, in particular, was only recently shown.

- *Dynamic finger.* This property is similar to the static finger property, but the finger moves to the last key we accessed. In particular, let $d(i, S)$ be the number of distinct keys in the tree between keys s_i and s_{i-1} (including the end points). The dynamic finger property implies that the cost of the splay tree algorithm on S is $O(m + n + \log d(i, S))$. This is described and proved by Richard Cole et al. [8], [7]) in a highly non-trivial fashion. (It is tough.)

- *Sequential access.* If we access each key in the tree in order, the splay algorithm has cost $O(n)$. This was proved in the classic paper by Sleator and Tarjan [32].

- *Preorder access.* If we start with an arbitrary initial tree T and use splay to access the nodes in preorder *for* T, the splay algorithm runs with cost $O(|T|)$ [4]. This is a slight generalization of sequential access. Note that neither of these two arguments is a potential argument; they use detailed arguments about the operation of splay trees to achieve their bounds.

1.3.2 Lower bounds

Before 2004, there were two main lower bounds, both described by Wilber [33]. These are described in detail in Section 3, where we describe a class of lower bounds on BST performance that includes both Wilber bounds. A group working with Sleator independently discovered this class of bounds [12], [11]. Both Wilber bounds are lower bounds on the cost of an optimal BST algorithm. They involve moderate computation, and are not easily expressible in terms of simple properties of the access sequence.

The first Wilber bound. The first bound of Wilber is based on a bound tree. The keys in the sequence are the leaves of the bound tree. We search for keys in the sequence in the bound tree, and mark which direction we take when leaving each node; this is the *preferred direction*. When we encounter a node and take the opposite direction, we *flip* the preferred direction. The number of times we flip the preferred direction is a lower bound on the cost of an optimal BST algorithm. This bound has been described a number of times in the literature. Wilber described it in his classic bound paper [33]; the bound follows from the partial sums lower bound [17], [28]; and a bound similar to the first bound of Wilber was proved in the paper introducing tango trees [10]. The first Wilber bound bound is not tight for a *fixed tree* and *arbitrary sequence*. However, it is not known if the bound is tight when we choose the tree based on sequence.

The second Wilber bound. This bound is similar to the first Wilber bound, but the bound tree is dynamic and equivalent to the dual-Cartesian tree (see definition on page 37) on the keys when they sorted by last access time (the most recent access is ordered first). This was only been stated in Wilber's paper [33], though it has been summarized elsewhere elsewhere [20]. Although the second Wilber bound is $\Omega(n \log n)$ on some sequences[4], it is not known if this bound is optimal, or even as large as the cut set lower bounds described in Chapter 3.

1.3.3 Assorted optimality results

There are very few optimality results, but some BST algorithms have been proved to be optimal within a restricted class of BST algorithms, a restricted class of access sequence, or with a different cost model. Let $\mathrm{OPT}(S)$ be the optimal BST cost for the sequence S, and let $\mathrm{OPT}_s(S)$ be the optimal static BST cost for S.

1.3.3.1 Optimal static BSTs

The first optimal result was the the *statically optimal* BST. This is an optimality result on either a restricted set of algorithms (static BST algorithms), or on the expected cost of sequences of iid samples of the keys, with a fixed probability for each key. Knuth [21] showed how to compute the optimal static BST. The computation is essentially dynamic programming: one computes the optimal subtree on every key interval, using the results from smaller intervals to get optimal results for the larger intervals. Knuth's algorithm needs to know the relative frequencies with which each

[4] The bit-reversal sequence, for example [33].

key is accessed, but for sequences where each key in the tree is accessed at least once, splay trees have cost $O(\mathrm{OPT}_s(S))$, without knowing the frequencies of the keys.

This result is very closely related to the entropy of the sequence. If f_i is the number of times the ith key is accessed in the sequence (with m total accesses), the *entropy* of the sequence is $\sum f_i \log(m/f_i)$. Call this $H(S)$. Knuth showed that $\mathrm{OPT}_s(S)$ is $O(H(S))$. It is known that optimal BSTs perform with cost at most $O(H(S))$, but often $\mathrm{OPT}(S) = o(H(S))$. However, if S is a sequence of length m of iid variables where we pick key i with frequency f_i/m, then $E(\mathrm{OPT}(S)) = \Theta(H(S))$.

1.3.3.2 Key-independent optimality

> **Definitions:** Let $b(\cdot)$ be a (uniformly) random map from a set Q (with $|Q| = n$) to the integers 1 through n. For a sequence $S = (s_1, \ldots)$ of elements of Q, let $b(S) = (b(s_1), b(s_2), \ldots)$. For a sequence of keys S let $W(S)$ be the working set cost on S[5]. A BST algorithm is *key-independently optimal* if it executes with cost $O(E(\mathrm{OPT}(b(S))))$. The *key-independent cost* of a BST algorithm on S is the expected cost of the algorithm on a random map of the keys of S to the integers 1 through $|S|$.

As the definitions suggest, key-independent optimality is a statement that an algorithm performs as well the expected cost of any BST algorithm when an order is imposed on keys from a set in a random fashion. It is likely that such algorithms are useful for keeping track of elements efficiently. Iacono [20] has shown that splay trees have this property[6]

We exploit the relation between key-independent optimality and the working set cost in Section 2.5.4. We show that a lower bound on the key-independent cost of a BST algorithm on a sequence is also a lower bound on the cost of any linear access service sequence under the model described by Munro [27]. Munro also proposed an

[5] See Section 1.3.1 for a definition of *working set cost*.

[6] He goes further, and shows that any BST algorithm with the working set property is key-independently optimal.

offline linear access algorithm that achieves the working set cost, so his algorithm is optimal (within a constant factor).

1.3.3.3 Search optimality

Upon cursory examination of the BST problem, it is not clear that online BSTs can get even their *search* costs (see Section 1.2.3) to be $O(\mathrm{OPT}(S))$. Blum, Chowla, and Kalai [2] used a learning algorithm to show that online BSTs can (in principle) achieve this. Their result is an optimality result under a different cost model.

Blum, Chowla, and Kalai describe an online BST algorithm that has total *search cost* within a constant factor of the optimal total cost of an offline BST algorithm. Any algorithm that achieves this cost bound is *dynamically search-optimal*. This algorithm is not practical: it does not store information only in the BST between accesses, and it runs in time exponential in the number of keys in the sequence. However, their algorithm does prove a necessary condition for the existence of online dynamically optimal BSTs: the search costs for an online algorithm do not dominate the total optimal offline cost.[7]

It should be clear that something slightly fishy is going on as it is possible for an offline algorithm to be clever about its starting tree: it can achieve cost 1 for the first access regardless of the node, but any online algorithm can be made to have cost much higher. The reason for this discrepancy is that the authors restrict the offline algorithm to begin with the same tree that their online algorithm begins with. This restriction can be eliminated at the cost of only achieving search cost $O(n+\mathrm{OPT}(S))$ instead of just $O(\mathrm{OPT}(S))$.

[7] Blum, Chowla, and Kalai [2] also prove an interesting result related to the information required to describe BST algorithms: The number of sequences having optimal offline cost k is at most 2^{12k} for all $k \geq 0$ independent of the length of the sequence or the number of distinct keys in the sequence. This is Theorem 4.1 in a paper by Knuth [2].

1.3.4 Related structures and results

The upper bounds on the performance of (predominantly) splay trees are not tight; there are some sequences on which splay trees perform in linear time for the sequence while the upper bounds give cost logarithmic per access [18]. Iacono hypothesized a tighter upper bound. Let $S = (s_1, \ldots, s_m)$ be a sequence of keys, let $t_i(y)$ be the distinct number of times accessed after the last access to y and at or before s_i. Let $d_i(y)$ be the number of distinct keys (in the universe from which we take S) between y and s_i, including endpoints. Iacono hypothesizes the amortized time to access s_i using splay trees is

$$O(\log \min_{y \in S}(t_i(y) + d_i(y) + 1))$$

Iacono [18] and then Demaine and Bădoiu [3] used a data structure loosely based on BSTs, to achieve this upper bound with $O(\log n)$ worst-case run time for any access.

1.3.5 Conjectured performance and dynamic optimality

If the cost of a BST algorithm is $O(n + \mathrm{OPT}(S))$ for any S on the keys 1 through n, the algorithm is said to be *dynamically optimal*. This is a difficult property to show. It is unknown if any online BST algorithm is dynamically optimal, and there is not a known polynomial time algorithm to compute $\mathrm{OPT}(S)$, even within a constant factor[8]. It is hypothesized [30] that splay trees are dynamically optimal. We provide evidence that suggests an algorithm proposed by Munro [27] is dynamically optimal and may be made online.

‘

[8] If one allows exponential time, it is trivial to compute $\mathrm{OPT}(S)$. Upper bounds show that $\mathrm{OPT}(S) = O(|S| \log n)$ where n is the number of distinct keys in S. We can simply run through all access service sequences shorter than this, and we will find the optimal sequence.

Chapter 2

The Box Model

In this chapter, we introduce two novel geometric problems: the box stabbing problem (box stabbing) and a variant, the edge stabbing problem (edge stabbing). These problems are closely related to the BST access problem and a linear access problem from Munro [27], respectively. The box stabbing problem is used extensively in Chapter 3 to derive new lower bounds.

We begin this chapter by proving a relationship between the box stabbing problem and the BST access problem: the offline versions of each are essentially equivalent. Section 2.2 then discusses geometric problems superficially similar to the box stabbing problem.

We then turn to a simple greedy algorithm for the reorganization tree BST problem from Munro [27], and show that it has an extremely simple form in the box model. We provide evidence suggesting this algorithm is dynamically optimal and may be made online. Proofs related to this algorithm are used later in Chapter 3 to construct the lower bound up to a constant factor in our novel class of lower bounds.

Munro's 2000 paper [27] also introduced a new model for linear access. The edge stabbing problem and Munro's linear access problem [27] are related in a fashion similar to the BST and box stabbing problem. We prove this relation and exploit the results of Iacono's key-independent optimality [20] to show that the offline algorithm proposed by Munro [27] is within a constant factor of optimal for his linear access problem. The chapter concludes with some open questions and hypotheses related to the box model.

2.1 Box Stabbing

2.1.1 Notation and definitions

Box stabbing problems are posed by giving a set of points in the (standard Euclidean) plane. In this thesis, we only consider finite sets of points.

> **Definitions:** Pick orthogonal axes on the plane, and call them the x and y axes. The x coordinate of a point p is $x(p)$, and the y coordinate is $y(p)$. The closed axis-aligned rectangle between two points with distinct x and y coordinates is their *axis box*, and the points are the *generators* or *generating points* of the axis box. $B(p, q)$ is the box generated by points p and q. An axis box b is *stabbed* by a point p if p is in b but is not a generator of b. An axis box b is *stabbed by a point set P*, or *satisfied in or by P* if there is a point in P stabbing b. We will often simply call a box stabbed if the set of points is clear from context. A point set P is *satisfied* if all axis boxes between pairs of points in P are stabbed by P. Given a point set P the *box stabbing problem* posed by P is to find a satisfied superset of P of minimum cardinality. If X_P is the set of x coordinates of points in P and Y_P is the set of y coordinates, the *lattice* of P is the set of all points with x coordinate in X_P and y coordinate in Y_P. The *cost* of a satisfied set is the size of the set. We follow the notation for BST access service sequences: For a point set P, $\mathrm{OPT}(P)$ is the size of the smallest satisfied superset of P.

2.1.2 Simplifying assumptions

In this section, we show that for a finite box stabbing problem there is a satisfied subset of the lattice of the box stabbing problem solving the problem optimally. While showing this, we introduce a different formulation of satisfaction in terms of monotone Manhattan paths.

2.1.2.1 Scaling and shifting

First we show that we can always assume that a given set of points is on an integer lattice.

Lemma 2.1. *If a set is satisfied, the set under arbitrary non-degenerate rescaling and shifting of either the x or y coordinates is also satisfied.*

Proof. Without loss of generality we are scaling and shifting the x coordinates. Under such rescaling and shifting, the ordering of all coordinates is preserved, and distinct points map to distinct points. Suppose that the point r stabs $B(p, q)$ in the initial set. This implies that the x coordinate of r falls between the x coordinates of p and q (though not strictly). Rescaling and shifting does not change these orderings, so the image of r stabs the axis box between the image of p and the image of q. □

2.1.2.2 Finite sets

We only need to consider finite solutions to a finite box stabbing problem.

Theorem 2.2. *The lattice of a finite point set is a satisfied set.*

Proof. Suppose that there is an unsatisfied box $B(p, q)$ between two points in the lattice. As points p and q are in the lattice, all four corners of $B(p, q)$ are in the lattice, and $B(p, q)$ is stabbed by the lattice. □

This implies the following:

Corollary 2.3. *The smallest satisfied superset of any finite set of points is finite.*

In this thesis, therefore, we only consider finite point sets.

2.1.2.3 Monotone Manhattan paths

A *monotone Manhattan path* is a sequence of points such that adjacent elements of the sequence share either an x or y coordinate (or both), and both the x and y coordinates the sequence are monotone (but not strictly monotone). Monotone Manhattan paths are closely related to the property of satisfaction, as the Theorem 2.4 shows:

Theorem 2.4. *A set is satisfied if and only if there is a monotone Manhattan path in the set between all pairs of points.*

We prove each direction of this theorem as a separate lemma:

Lemma 2.5. *A set is satisfied if there is a monotone Manhattan path between between each pair of points in the set.*

Proof. Suppose that there is a monotone Manhattan path between each pair of points in the set Q. Consider the axis box $B(p, q)$ in Q. As p and q do not share x or y coordinates, they cannot be adjacent in any monotone Manhattan path between them. There must be a third point r on such a path between p and q. As the path coordinates of a monotone Manhattan path are monotone in both x and y, r must stab $B(p, q)$. □

Lemma 2.6. *There is a monotone Manhattan path between each pair of points in a satisfied set.*

Proof. For the duration of this proof, we extend the definition of axis boxes to include degenerate axis boxes: the two generating points may share either their x or y coordinates. However, degenerate boxes need not be stabbed by a third point.

We induct on the size of boxes. Suppose that all axis boxes in Q smaller in either x or y extent than $B(p, q)$ have a monotone Manhattan path between their generating points. $B(p, q)$ is stabbed by a third point r. The induction hypothesis guarantees there is a monotone Manhattan path between p and r and r and q. Concatenating the paths yields a monotone Manhattan path between p and q.

Suppose that there are no boxes smaller in x or y extent than $B(p,q)$ in Q. If $B(p,q)$ is non-degenerate then it is stabbed by a third point r and both $B(p,r)$ and $B(r,q)$ are smaller in x or y extent than $B(p,q)$. This implies that $B(p,q)$ is degenerate, so (p,q) is a monotone Manhattan path between the generating points of the box. □

2.1.2.4 The lattice is enough

We now have enough machinery to show that the lattice of a box stabbing problem contains an optimal solution to the problem.

Theorem 2.7. *For a box stabbing problem P, the lattice of P contains a superset of P of minimum cardinality.*

Proof. Let Q be a superset of P of minimum cardinality. Lemma 2.1 lets us assume that the lattice of P is the lattice on the integers 1 through n in x and 1 through m in y. Let Q' be Q with all coordinates rounded to the nearest integers. Lemma 2.1 also tells us that without loss of generality the points of Q' are on the lattice of P.

Real numbers have a useful property under rounding. We will call it the *rounding property*: Rounding a set of real numbers does not change their relative order, although degeneracies may be introduced.

The rounding property tells us two things: $|Q'| \leq |Q|$, and the image of a monotone Manhattan path under rounding is a monotone Manhattan path, though there may be duplicate nodes in the rounded path. Theorem 2.4 implies that there are monotone Manhattan paths between all pairs of points in Q and the rounding property implies that there are monotone Manhattan paths between all pairs of points in Q'. Theorem 2.4 then tells us that Q' is satisfied. □

2.1.3 Box stabbing and the BST access problem

In this section we show that the box stabbing problem is closely related to the reorganization tree BST access problem described in Appendix A. The reorganization tree

BST model is an alternate to access service sequences for specifying reorganizations.

> **Definitions:** A *reorganization tree* for a BST T is a BST on a set of keys in a connected subtree of T containing the root of T. When we *apply* a reorganization tree R for a tree T to T, we rearrange the nodes in T containing the keys of R into the structure of R, and move the other subtrees of T appropriately to maintain the BST invariant. An *reorganization tree access service sequence* for a sequence of keys from 1 through n is a sequence of reorganization trees such that the ith tree is a reorganization tree after we apply all earlier reorganization trees in the sequence in order. The 0th tree in the reorganization tree access service sequence is a tree on all n nodes, while the ith reorganization tree contains the ith element of the sequence of keys. The *cost* of access i for a reorganization tree access service sequence is the number of keys in the ith reorganization tree. The *total cost* of a reorganization tree access service sequence is the cost of all access in the sequence. The *reorganization tree BST access problem* posed by a sequence of keys S is to find the reorganization tree access service sequence for S of minimum cost.

See Appendix A for a proof that this model has an optimal cost within a constant factor of the optimal cost in the standard model. The main goal of this section is the following:

Theorem 2.8. *For each reorganization tree BST access problem S there is a corresponding box stabbing problem M such that the following hold:*

- *There is an efficiently computable, cost-preserving map from reorganization tree access service sequences for S to satisfied sets containing M.*
- *There is an efficiently computable, cost-preserving map from satisfied sets containing M (on the lattice of M) to reorganization tree access service sequences for S.*

The proof of Theorem 2.8 has two major parts. After giving all maps mentioned in the theorem, we prove the maps are valid. (They are very easy to compute.) First we develop a little vocabulary.

> **Definitions:** Let K be a set of keys and L be an ordering of the keys. The *Cartesian* tree on the keys is a binary tree, but not a binary search tree. The construction is discussed in detail by Gabow Bentley and Tarjan [14]. A Cartesian tree is constructed by placing the lowest key r as the root, then constructing the left and right subtrees of the root recursively using the keys less than r in L and greater than r in L respectively.
>
> We define a similar construction, but exchange the roles of key ordering and the ordering imposed by L. The *dual-Cartesian tree on a set of keys K with ordering L* is constructed in the following manner. Place the first key r in L at the root of the tree. Build the left and right subtrees of r using the keys greater less and greater than r in the key ordering respectively.
>
> If Q is a set of points on the integer lattice with x coordinate range from 1 through n and y coordinate range from 1 through m, let $r_i^-(Q, j)$ be the largest y coordinate less than or equal to i of points in Q with x value j, or 0 if no such points exist. Let $r_i^+(Q, j)$ be the smallest y coordinate greater than i among points in Q with x value j, or m if no such points exist. The *section tree for Q at i* is the dual-Cartesian tree on the nodes if we order the nodes lexicographically first by $i - r_i^-(Q, \cdot)$ then $r_i^+(Q, \cdot)$ with ties broken arbitrarily. The *section of Q at i* is the set of all points in Q with y coordinate $\leq i$. If the point set is clear from context, we use $r_i^{\pm}(\cdot)$ instead.

2.1.3.1 Maps

We now turn to defining the required maps.

Map from reorganization tree BST access problems to box stabbing problems. The reorganization tree BST access problem $(x_1, \ldots, x_n)$ maps to the set of points $\{(x_1, 1), (x_2, 2), \ldots, (x_n, n)\}$. We designate this map $M_{\text{BST}\to\text{BSP}}$. This map is efficiently computable as we only need to attach an index to each element of S.

Map from reorganization tree access service sequences to satisfied sets. We call this map $M_{\text{RT}\to\text{STAB}}$. Let T be an reorganization tree access service for some set and let T_i be the ith reorganization tree of T. The point (k, i) is in $M_{\text{RT}\to\text{STAB}}(T)$ if and only if the node with key value k is in T_i[1]. Each node in each reorganization tree of T corresponds to exactly one point in $M_{\text{RT}\to\text{STAB}}(T)$ so this map is cost preserving. We only need to check if a node is present in a tree for each reorganization tree tree in the sequence, so the map is efficiently computable.

Map from satisfied supersets (on the lattice of the problem) to reorganization tree access service sequences. We call this map $M_{\text{STAB}\to\text{RT}}$. Let S be a sequence of keys and let $P = M_{\text{BST}\to\text{BSP}}(S)$. Let Q be a satisfied subset of the lattice of P containing P. Let $M_{\text{STAB}\to\text{RT}}(Q) = (T_0, T_1, \ldots, T_{|S|})$. T_0 contains all the keys, and T_i for $i \geq 1$ includes key k if and only if (k, i) is in Q. The tree T_i is the dual-Cartesian tree on its nodes when they are sorted by $r_i^+(Q, \cdot)$ with ties broken arbitrarily. This is efficiently computable:

- We can find the sets of keys in T_i efficiently for all i as these are given by the sets of points for a given y coordinate.
- Computing $r^+(Q, \cdot)$ takes time $O(mn)$ for each of $O(mn)$ points.
- Using $r^+(Q, \cdot)$ to compute T_i from the set of keys in T_i takes time at most $O(n)$ plus $O(1)$ times the cost of the reorganization tree access service sequence (i.e. it's size).

[1] We do not use the initial tree of the reorganization tree access service sequence; this is intentional.

2.1.3.2 Map from reorganization tree access service sequences to satisfied sets is valid

We can show this directly:

Theorem 2.9. *If T is an access service sequence, then $M_{\text{RT}\to\text{STAB}}(T)$ is a satisfied set.*

Proof. Let A be a valid reorganization tree access service sequence for the BST access service problem S. (See Section A.2 for a definition.) Let RT_i be the ith reorganization tree of A. For this proof, we do not need to know the structure of the reorganization tree after organization, only that A is a valid reorganization tree access service sequence.

For sequences of length 1, the theorem holds: a set of points all with the same y coordinates is satisfied. Now suppose that the theorem holds for all sequences of length less than i. This implies that the section of Q at i can only have unsatisfied boxes between points with y coordinate i and points with lower y coordinates. Let $B(p, q)$ be such an unstabbed box with p at y coordinate i, and q at y coordinate j.

By definition of the map, A touched q at time j, but not p or any nodes between p and q: $q = \text{lca}(p, q)$ at time j. Similarly at time i, A touched p but not q or any nodes between p or q: $p = \text{lca}(p, q)$ at time i. The lca of two nodes cannot be changed without a single reorganization tree touching at least two nodes in the closed interval between the nodes by Theorem A.7, a contradiction. □

2.1.3.3 The map from box stabbing problems to reorganization tree BST access problems is valid

In this section we prove that $M_{\text{STAB}\to\text{RT}}$ is a valid map. This proof is more involved than the proof for $M_{\text{RT}\to\text{STAB}}$ in Theorem 2.9.

Lemma 2.10. *If the following hold,*

- *S is a reorganization tree BST access problem,*

- *Q is a satisfied subset of the lattice of $M_{\text{BST}\to\text{BSP}}(S)$,*

- *Q contains $M_{\text{BST}\to\text{BSP}}(S)$, and*

- *A is the image of Q under $M_{\text{STAB}\to\text{RT}}$,*

then A is a valid reorganization tree sequence.

Proof. Let Q_i be the section tree of Q at i. We show that the sequence $(Q_0, Q_1, \ldots)$ is the expanded sequence of A. As the first trees of both sequences match, we only need to show that the nodes containing the keys in the ith tree of A for all $i > 1$ form a connected subtree containing the root of the section tree of Q at time $i - 1$.

Let RT_i be the ith reorganization tree of A. We want to show that for each key q in RT_i, either q is the root of Q_{i-1} or q's parent in Q_{i-1} is also in RT_i. If q is the root of Q_{i-1}, we are done. Suppose that q has a parent p in Q_{i-1}. Without loss of generality $p < q$. For this proof we assume the set Q for the functions $r_i^{\pm}(Q, \cdot)$ to get $r^{\pm}(\cdot)$.

We first show that the rectangle $(p, q) \times (r_{i-1}^{-}(q), i - 1]$ cannot contain any points of Q. Note that $r_{i-1}^{-}(p) \geq r_{i-1}^{-}(q)$ by definition of section tree and the parent relation. Call this the *parent restriction*. Consider two cases:

- Suppose that there is an $s \in (p, q)$ such that $r_{i-1}^{-}(s) > r_{i-1}^{-}(p)$. By definition of section tree, some key satisfying these conditions will be $\text{lca}(p, q)$ in Q_{i-1}, a contradiction in assumption.

- If there is an $s \in (p, q)$ such that $r_{i-1}^{-}(p) \geq r_{i-1}^{-}(s) > r_{i-1}^{-}(q)$, one such s will be in the node-to-root path of q in Q_{i-1}. However, if s is between p and q and s is in the node-to-root path of q, p cannot be q's parent, a contradiction.

This implies that the rectangle $(p, q) \times (\min(r_{i-1}^{-}(p), r_{i-1}^{-}(q)), i - 1]$ cannot contain any points of Q. As $r_{i-1}^{-}(p) \geq r_{i-1}^{-}(q)$ by the parent restriction, we can instead say $(p, q) \times (r_{i-1}^{-}(q), i - 1]$ is empty.

Now consider the possibilities allowed by the parent restriction.

- Suppose that $r^-_{i-1}(q) = r^-_{i-1}(p)$. In this situation, because p is q's parent, the definition of section tree implies that $r^+_{i-1}(p) \leq r^+_{i-1}(q)$. We know $r^+_{i-1}(q) = i$. This and the property $r^+_{i-1}(\cdot) \geq i$ implies that $r^+_{i-1}(p) = i$ as we need.

- Now suppose that $r^-_{i-1}(q) < r^-_{i-1}(p)$. Let s be the smallest key in the range $(p, q]$ such that the point (s, i) is in Q. The parent restriction implies that there are no points in $(p, q) \times (r^-_{i-1}(q), i - 1]$, so the box $B((p, r^-_{i-1}(p)), (s, i))$ can only be stabbed by a point at (p, i); the theorem is satisfied.

□

Theorem 2.11. *If Q is a satisfied subset of the lattice of $M_{\text{BST}\to\text{BSP}}(S)$ and contains $M_{\text{BST}\to\text{BSP}}(S)$, then $M_{\text{STAB}\to\text{RT}}(Q)$ is a valid reorganization tree access service sequence for S.*

Proof. By construction, (k, i) is in the ith reorganization tree of $M_{\text{STAB}\to\text{RT}}(Q)$, and by Lemma 2.10 $M_{\text{STAB}\to\text{RT}}(Q)$ is a valid reorganization tree sequence. This is the definition of a reorganization tree access service sequence. □

2.2 Box Stabbing: Related Problems

In this section we point out a few computational problems superficially related to the box stabbing problem, and note why these problems do not help us solve the box stabbing problem.

2.2.1 Box piercing

Box stabbing problems bear superficial resemblance to rectilinear p-piercing problems. If we are given a set of rectilinear rectangles in the plane, the *rectilinear p-piercing problem* is to find p points such that each region intersects at least one of the points, or determine that this is not possible. See the paper by Sharir and Welzl [29] for a more detailed description and references to the literature.

The complexity results for rectilinear p-piercing problems are perhaps not surprising: they are NP-complete if we are required to find the smallest p such that the regions can be pierced by p points [25], but for any fixed p there is an algorithm solving the p-piercing problem that is polynomial in the number of rectangles ([29],[13],[23]).

Interesting though these results may be, it is not immediately clear how to apply them to an analysis of box stabbing problems. Taking advantage of the superficial resemblance does not provide immediate results, as the rest of this section shows.

The *corresponding piercing problem for a box stabbing problem* is the rectilinear p-piercing problem consisting of all unstabbed rectangles from the box stabbing problem. The smallest p for which a piercing problem may be solved is the *optimal* p for the problem. The following theorem destroys most hope of using bounds on the optimal p for a corresponding piercing problem.

Theorem 2.12. *Consider a box stabbing problem P with its points on an integer lattice. The set of all unsatisfied rectangles in P can be pierced with $4|P|$ points, even if the unstabbed rectangles cannot be pierced on their generating corners.*

Proof. For each point $p \in P$, the piercing set contains points at the corners of an axis-aligned rectangle of side length less than 2 centered at the point. Every unstabbed rectangle in P has generating points (by definition). P is on the integer lattice, so both the x and y extent of all unstabbed boxes is at least 1 and no point in the piercing set is the generating point of any unstabbed box. This implies that the corners added to the square around either of the generating points will be inside any unstabbed box: the piercing set pierces all unstabbed rectangles in P. □

Theorem 2.12 implies that any bounds on the the optimal p for a p-piercing are not likely to be of interest, as any box stabbing problem problem must have cost at least linear in the size of the box stabbing problem.

2.2.2 Minimum Manhattan networks

A *Manhattan network* on a set of points in the plane is a graph. The vertex set of the Manhattan network on S contains S and possibly additional points, called *Steiner points*, also represented by points in the plane. The edges of the Manhattan network can only be between pairs of points with the same x or y coordinates such that no points in the vertex set are on the line segment between the two vertices. Subject to this constraint, a Manhattan network on a set must contain a monotone Manhattan path between each pair of points in the set, though not necessarily between every pair of points in the vertex set.

With each edge of a Manhattan network, we associate a *length*: the distance between its points. The length of a Manhattan network is the sum of the lengths of all its edges. The usual definition of a *minimum Manhattan network* on a set of points is a Manhattan network with minimum length. Another definition is one with a minimum number of Steiner points. See the paper by Gudmundsson, Levcopoulos, and Narasimhan [16] for a more extensive discussion and pointers to the literature. This paper provides a polynomial time algorithm for computing an $O(1)$-competitive Manhattan network of minimum length. The complexity of this problem is still unknown [6], although approximation algorithms for the minimum length Manhattan network are improving.

These results do not provide us with much useful information for the analysis of box stabbing problems, unfortunately. The metric usually considered is the *length* of the edges of the graph, not the number of points in the vertex set. Bounds on this problem would not be immediately useful for analysis of the box stabbing problem, as the box stabbing problem only cares about the number of Steiner points added. Moreover, even if we consider the metric of added Steiner points, a solution to the problem would not be immediately useful as the box stabbing problem requires Manhattan paths between all pairs of points, including the added Steiner points. The one result that might be of interest is an NP-completeness proof for computing the Manhattan network with the fewest added Steiner points. Such a result might imply the box stabbing problem is NP-complete.

2.3 The IAN and MUNRO Algorithms

Consider the box model. Given a set of points P, there are many ways one might imagine making a satisfied superset of P. We describe here a simple greedy algorithm for producing a satisfied superset, the *IAN algorithm*. Given a set of points P, IAN produces a satisfied superset IAN(P) of P (on the lattice of P). IAN(P) is the *IAN set of* P or the *IAN points of* P. We consider IAN interesting predominantly because we think it will provide a method of constructing a strict online dynamically optimal BST algorithm.

> **Definitions:** Let P be a set of points in the plane, with y coordinates in the set 1 through m. To compute the *IAN set* or *IAN points* for P, begin with P in the IAN set. For each y coordinate i in standard order, add to the IAN points the minimum set of points at $y = i$ such that the section of the IAN points at $y = i$ is satisfied. We denote the complete set of IAN points on P by IAN(P).

We named this algorithm IAN after Ian Munro. A few years ago, Ian Munro proposed a simple offline BST algorithm [27]. It is most easily phrased in terms of its reorganization tree equivalent. If P is the box stabbing problem corresponding to a BST access problem S, the IAN points of $M_{\text{BST}\to\text{BSP}}(S)$ are the image under $M_{\text{RT}\to\text{STAB}}$ of Munro's algorithm on S. We call Ian Munro's BST algorithm *MUNRO*. We first show that IAN is well defined and produces a satisfied point set, then prove the correspondence between IAN and MUNRO.

2.3.1 IAN points are well defined

In this section we prove that the description of IAN points given above is well defined and unique and produces a satisfied set. The proof here is used in a slightly modified form in Section 3.5.2 to construct a strong lower bound on the cost of a box stabbing problem. The parallels between IAN points and NISIAN points (see Section 3.5.2) may eventually help to show that the IAN algorithm is dynamically optimal.

Theorem 2.13. *Let P be a point set.* IAN(P) *is well defined: when we reach $y = i$ during construction, there is a unique minimum set of points we can add to eliminate all unstabbed boxes with generating points at or below $y = i$.*

Proof. Let IAN$(P)_{i,j}$ be the section at i of the set of IAN points after processing $y = j$. Let M_i be the set of empty upper corners of unstabbed boxes between pairs of points in IAN$(P)_{i,i-1}$. We prove that M_i is the set of points we add to the IAN points at $y = i$ by induction.

At $i = 1$, the property holds: there are no unstabbed boxes in IAN$(P)_{1,1}$. By the induction hypothesis, when we reach $y = i$, there are no unstabbed boxes in IAN$(P)_{i-1,i-1}$: all points in M_i are at $y = i$. Under this assumption, we now show that M_i is the unique minimum set of points we need to add at $y = i$ to eliminate all unstabbed boxes in IAN$(P)_{i,i-1}$.

Suppose that we do not need to add all points in M_i to the IAN points while processing $y = i$: the upper empty corner of an unstabbed box, $B(p, q)$, is need not be added when we process $y = i$. Without loss of generality we can assume that $y(p) = i$ and $y(q) < i$. Let r be the point added to the IAN set at $y = i$ with x coordinate closest to $x(q)$ such that $x(r) > x(q)$. See Figure 2-1. As $B(p, q)$ is only stabbed by points at $y = i$, $B(q, r)$ can only be stabbed by points at $y = i$. However, we assume that there can be no points in $[x(q), x(r)]$: $B(q, r)$ is unstabbed. This implies that we must add at least the points in M_i. Reflecting about the y axis gives the appropriate argument when $y(p) = i$ and $y(q) > i$.

Now we show that we only need to add the points in M_i at $y = i$. Suppose that there is a box $B(p, q)$ unstabbed after M_i is added to the IAN points, with $y(p) = i$ and $y(q) < i$. If p is in P, the upper unsatisfied corner of $B(p, q)$ is in M_i, so p cannot be in P. By definition of M_i, p is at the upper empty corner of a box $B(r, s)$, with $y(r) = i$, and $B(r, s)$ is only stabbed by points with $y = i$. If $x(r) < x(q)$, then

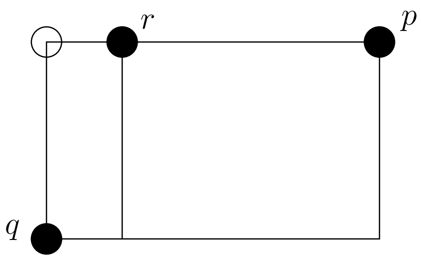

Figure 2-1: The empty circle is in M_i.

either q would stab $B(r, s)$ or s would stab $B(p, q)$ as illustrated in Figure 2-2. This implies that $x(r) > x(q)$, giving us the situation in Figure 2-3. $B(p, q)$ is unstabbed so $y(s) \leq y(q)$. The box $B(q, r)$ can also only be stabbed by points at $y = i$, but this implies that the upper left corner of $B(q, r)$ is in M_i. The boxes $B(q, r)$ and $B(p, q)$ share upper left corners, so $B(p, q)$ must be stabbed, a contradiction. Reflecting about the y axis gives the appropriate argument for when $y(p) = i$ and $y(q) < i$. This implies that adding only M_i eliminates all unstabbed boxes in $\text{IAN}(P)_{i,i-1}$. □

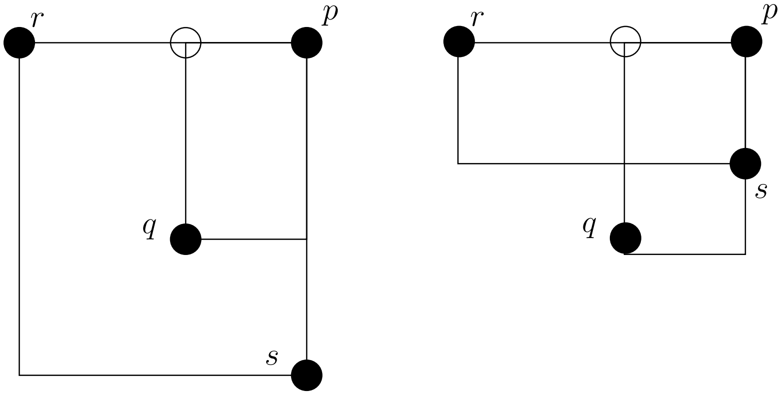

Figure 2-2: Two impossible situations.

The implicit corollary in the statement of Theorem 2.13 is the following:

Corollary 2.14. *Let P be a point set. There are no unstabbed boxes in the IAN set of P.*

Proof. In the proof of Theorem 2.13, we showed that we can add a unique minimum

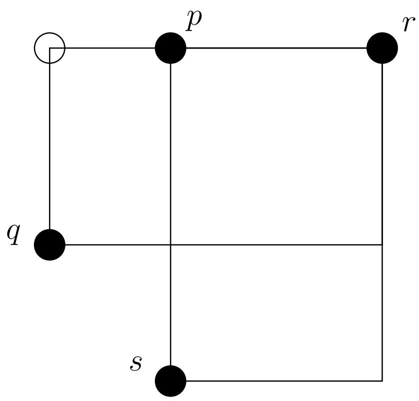

Figure 2-3: The empty circle is in M_i.

set at $y = i$ to eliminate all unstabbed boxes between points with $y \leq i$. This includes the largest y coordinate for any points in P so the corollary follows. □

2.3.2 The MUNRO algorithm

Before we show that IAN and MUNRO are the same, we describe the MUNRO algorithm from Munro's paper [27] in terms of the reorganization tree BST access sequence it produces.

> **Definitions:** Let K be subset of keys of a BST and let $S = (s_1, \ldots, s_m)$ be a sequence of these keys. For a key $a \in K$, the *associated key interval for a in K* is the open interval of keys in the BST (on keys of S) between the key to the left of a in K and the key to the right of a in K. In the *MUNRO ordering of K (in S) at i*, key a is ordered before key b if some key in the associated key interval of a is accessed in $(s_{i+1}, \ldots, s_m)$ before any keys in the associated key interval of b, with ties broken arbitrarily[2].
>
> Let MUNRO$(S) = (T_0, R_1, \ldots, R_m)$ be the reorganization tree BST sequence MUNRO produces for S and let $(T_0, \ldots, T_m)$ be its expanded sequence. The initial tree of MUNRO(S) is the dual-Cartesian tree on all

[2] Ties can occur because the associated key intervals of adjacent keys in the subtree intersect.

> the keys when they are sorted by the MUNRO ordering at 0. R_i contains the keys on the node-to-root path of s_i in T_{i-1}. R_i is the dual-Cartesian tree on its nodes when they are sorted by the MUNRO ordering in S at i, with ties broken arbitrarily.

Although MUNRO is offline, an online linearly splittable BST can make it online. We discuss this statement and in more detail after we show that IAN and MUNRO are the same.

2.3.3 IAN is MUNRO

The algorithms are not quite the same as MUNRO runs on an reorganization tree BST access problem and IAN runs on a box stabbing problem. However, the following theorem shows that they are "morally" equivalent:

Theorem 2.15. *Let $S = (s_1, \ldots, s_m)$ be a sequence of keys. The following equality holds:*

$$M_{\text{RT}\to\text{STAB}}(\text{MUNRO}(S)) = \text{IAN}(M_{\text{BST}\to\text{BSP}}(S))$$

This theorem does require a little machinery to prove, so we postpone the proof until page 50. The following lemmas deal with some simple properties of the MUNRO algorithm.

> **Definitions:** Let $S = (s_1, \ldots, s_m)$ be a sequence of keys, and let $A = (T_0, R_1, \ldots, R_m)$ be a reorganization tree BST service sequence on S. An access s_j *newly touches* a node p in R_i if R_j is the first reorganization tree containing p after R_i in A. We extend this to T_0, by letting $R_0 = T_0$ in the above definition.

Lemma 2.16. *Let $S = (s_1, \ldots, s_m)$, $\text{MUNRO}(S) = (T_0, R_1, \ldots, R_m)$, and let $R_0 = T_0$. If a node $p \in R_i$ has not been newly touched by any s_k for $k \in (i, j)$ and s_j is in the associated key interval of p in R_i, then p is newly touched by s_j.*

Proof. After applying R_i in in the reorganization tree BST sequence, node p is on the node-to-root path of s_j. Nodes cannot be removed from a node-to-root path unless they are touched during a reorganization (Corollary A.8). As p is not in any reorganization trees after R_i until R_j, p must be on s_j's node-to-root path when accessing s_j: p is in R_j. □

Lemma 2.17. *Let* $S = (s_1, \ldots, s_m)$, $\text{MUNRO}(S) = (T_0, R_1, \ldots, R_m)$, *and* $R_0 = T_0$. *A node* $p \in R_i$ *cannot be newly touched by* s_j *if* s_j *is not in the associated key interval of* p *in* R_i.

Proof. This is a simple consequence of Lemma 2.16 and Corollary A.9. We proceed by contradiction, so suppose that there is a p in R_i that is newly touched by s_j such that s_j is not in the associated interval of p in R_i. Let q be the node in R_i closest to s_j and such that s_j is not between p and q. (We can have $q = s_j$.)

Lemma 2.16 implies that p must be ordered after q in the MUNRO ordering of R_i at i. If p were ordered before q, the lemma tells us p would have been newly touched before j. The MUNRO algorithm thus ensures that $\text{lca}(p, q) \neq p$ at i, and that we will not touch p while accessing q unless $\text{lca}(p, q) = p$. By Corollary A.9, p cannot get onto q's node-to-root path without being touched. Thus p must be in some R_k for $k \in (i, j)$, but this contradicts the definition of newly touched. □

Theorem 2.18. *Let* $S = (s_1, \ldots, s_m)$, *let* $\text{MUNRO}(S) = (T_0, R_1, \ldots, R_m)$, *and let* $R_0 = T_0$. *A node* $p \in R_i$ *is newly touched by* s_j *if and only if* s_j *is the first key in* S *after* i *in the associated key interval of* p *in* R_i.

Proof. This follows directly from Lemmas 2.16 and 2.17. □

Lemma 2.19. *Let* $S = (s_1, \ldots, s_m)$, *let* $(T_0, R_1, \ldots, R_m)$ *be any valid reorganization tree access service sequence for* S, *and let* $R_0 = T_0$. *If* s_i *touches a node* p *then* s_i *newly touches* p *in some* R_k *with* $k < i$.

Proof. Let j be the latest index before i such that R_j contains p. At least one such j exists because R_0 contains all nodes in the full BST. The definition of *newly touching* implies that s_i newly touches p in R_j. □

The above proofs have shown us that when we rearrange a tree during the MUNRO algorithm, we newly touch at most one or two nodes in a reorganization tree at a time, and only if the nodes in the reorganization tree are the closest (untouched) nodes to the node we are trying to access. Using a little argument about the structure of BSTs, we now give the proof of Theorem 2.15 from page 48.

Proof of Theorem 2.15, page 48. Let $\mathrm{MUNRO}(S) = (T_0, R_1, \ldots, R_m)$ and T be the expanded sequence of $\mathrm{MUNRO}(S)$. Let $Q = M_{\mathrm{RT}\to\mathrm{STAB}}(\mathrm{MUNRO}(S))$ and $P = \mathrm{IAN}(M_{\mathrm{BST}\to\mathrm{BSP}}(S))$. P_i and Q_i are the sections at i of P and Q respectively. We prove that $P_i = Q_i$ for all $i \in [1, m]$ by induction on i.

The hypothesis holds for $i = 2$ as the section at 1 of both P and Q contains only one point at $x = s_1$. Suppose that the induction hypothesis holds at index i. Let p be a key in R_{i+1}. Lemma 2.19 implies that p is newly touched in some R_j for $j < i + 1$. Theorem 2.18 implies that this happens if and only if s_{i+1} is the first key after s_j in the associated key interval of p in R_j. We now show that IAN adds a point at $(p, i + 1)$ during the construction of P in the same situation.

- First suppose that s_{i+1} newly touches p in R_j. Suppose that the box $b = B((p, j), (s_{i+1}, i + 1))$ is stabbed in $V = P_{i-1} \cup \{(s_{i+1}, i + 1)\}$. The box b cannot be stabbed at $y = j$ by definition of associated key interval and the the induction hypothesis. Let $(q, k) \in V$ be the point farthest down, then farthest left that stabs b. The box $B((p, j), (q, k))$ will be unstabbed unless $q = p$, but this is a contradiction in the assumption that s_{i+1} newly touches p (and the induction hypothesis). Thus IAN adds p at $y = i + 1$ as required.

- Now suppose that s_{i+1} does not newly touch p for any R_j with $j < i + 1$. Let k be the index of the latest index before $i + 1$ when p was touched. By Theorem 2.18, s_{i+1} is not in the associated key interval for p in R_k, so there is a node q in R_k between p and s_{i+1}. The point (q, k) stabs $B((p, j), (s_{i+1}, i + 1))$: IAN will not add p at $y = i + 1$.

□

2.3.4 Monotonicity of MUNRO implies it is dynamically optimal

The IAN and MUNRO algorithms have an interesting property. If their costs are (somewhat) monotonic, then they are within a constant factor of OPT on point sets or key sequences respectively. The proof for IAN is easy:

Theorem 2.20. *If* $|\text{IAN}(P)| = O(|\text{IAN}(Q)|)$ *for every point set* P *and* Q *such that* Q *is an on-lattice superset of* P*, then* $|\text{IAN}(P)| = O(\text{OPT}(P))$ *for every point set* P*.*

Proof. Let Q be an (on lattice) satisfied superset of P such that $|Q| = \text{OPT}(P)$. As Q is satisfied, $\text{IAN}(Q) = Q$, since IAN only adds points to eliminate unstabbed boxes. □

Although IAN and MUNRO are similar algorithms, Theorem 2.20 cannot be used for MUNRO because the image of an access sequence under $M_{\text{BST}\to\text{BSP}}$ produces a point set with at most one point at each y coordinate. However, MUNRO is sufficiently well behaved that we can get a similar result by adding keys to the sequence instead of the corresponding point set.

Theorem 2.21. *Let* S *be a sequence of keys. If* $|\text{MUNRO}(S)| = O(|\text{MUNRO}(S')|)$ *for any supersequence* S' *of* S *then* $|\text{MUNRO}(S)| = O(\text{OPT}(S))$

The rest of this section gives the proof. It is in some sense a constructive proof, but it is not short. For notational convenience, instead of using $M_{rttosat}(\text{MUNRO}(S))$, we use $\text{IAN}(M_{\text{BST}\to\text{BSP}}(S))$. This is justified by Theorem 2.15. We assume that R is an optimal reorganization tree access service sequence for S and construct a supersequence S' of S such that $|\text{MUNRO}(S')| = O(\text{OPT}(S))$. This is enough.

Let $P = M_{\text{BST}\to\text{BSP}}(S)$, R be an optimal reorganization tree sequence for S, and $r_{(i,j)}$ be the jth key (from lowest to highest) in R_i. Create S' from S by replacing s_i in S with

$$\left(r_{(i,1)}, r_{(i,1)}, r_{(i,2)}, r_{(i,2)}, \ldots, r_{(i,|R_i|)}, r_{(i,|R_i|)}, r_{(i,1)}\right).$$

Let $Q = M_{\text{BST}\to\text{BSP}}(S')$. Clearly Q may also be acquired by replacing (s_i, i) in P with the set of points

$$\Big\{(r_{(i,1)}, y_{(i,1)}), (r_{(i,1)}, y_{(i,2)}), (r_{(i,2)}, y_{(i,3)}), (r_{(i,2)}, y_{(i,4)}), \ldots,$$
$$(r_{(i,|R_i|)}, y_{(i,2|R_i|-1)}), (r_{(i,|R_i|)}, y_{(i,2|R_i|)}), (r_{(i,1)}, y_{(i,2|R_i|+1)})\Big\}$$

where $y_{(i,j)}$ are integers such that $y_{(i,j)} < y_{(r,s)}$ if $i < j$ or if $i = j$ and $j < s$.

Let $[\text{IAN}(Q)](a, b)$ denote the points in $\text{IAN}(Q)$ with y coordinates in the y range $[a, b]$, and define $Q(a, b)$ and $[\text{MUNRO}(Q)](a, b)$ similarly. We now prove that $\text{IAN}(Q)$ has the following properties. These will be called the *necessary MUNRO properties* in the rest of the proof.

1. Every point in $[\text{IAN}(Q)](y_{(i,1)}, y_{(i,2|R_i|+1)})$ has x coordinate given by a key in R_i.

2. The points at $y_{(i,2|R_i|+1)}$ are those with x coordinates in R_i.

3. There are at most $6|R_i|$ points in $[\text{IAN}(Q)](y_{(i,1)}, y_{(i,2|R_i|+1)})$.

This situation is depicted in Figure 2-4 Figure 2-4 has the following properties which shown later.

- Black points are points are in Q while grey points are points in $\text{IAN}(Q)$ but not Q. This is by definition.

- Region A consists of all points on the lattice below and right of the black points in region$B$. A may contain many grey points, but no black points.

- No points are above points in region B save those in region C.

- Region A in Figure 2-4 contains no more than $2|R_i|$ points, region B contains $3|R_i|$ points, and region C contains $|R_i|$ points.

We will refer to these regions in the rest of the proof. The key idea is to show that the points in Figure 2-4 are functionally equivalent to just the points in region C.

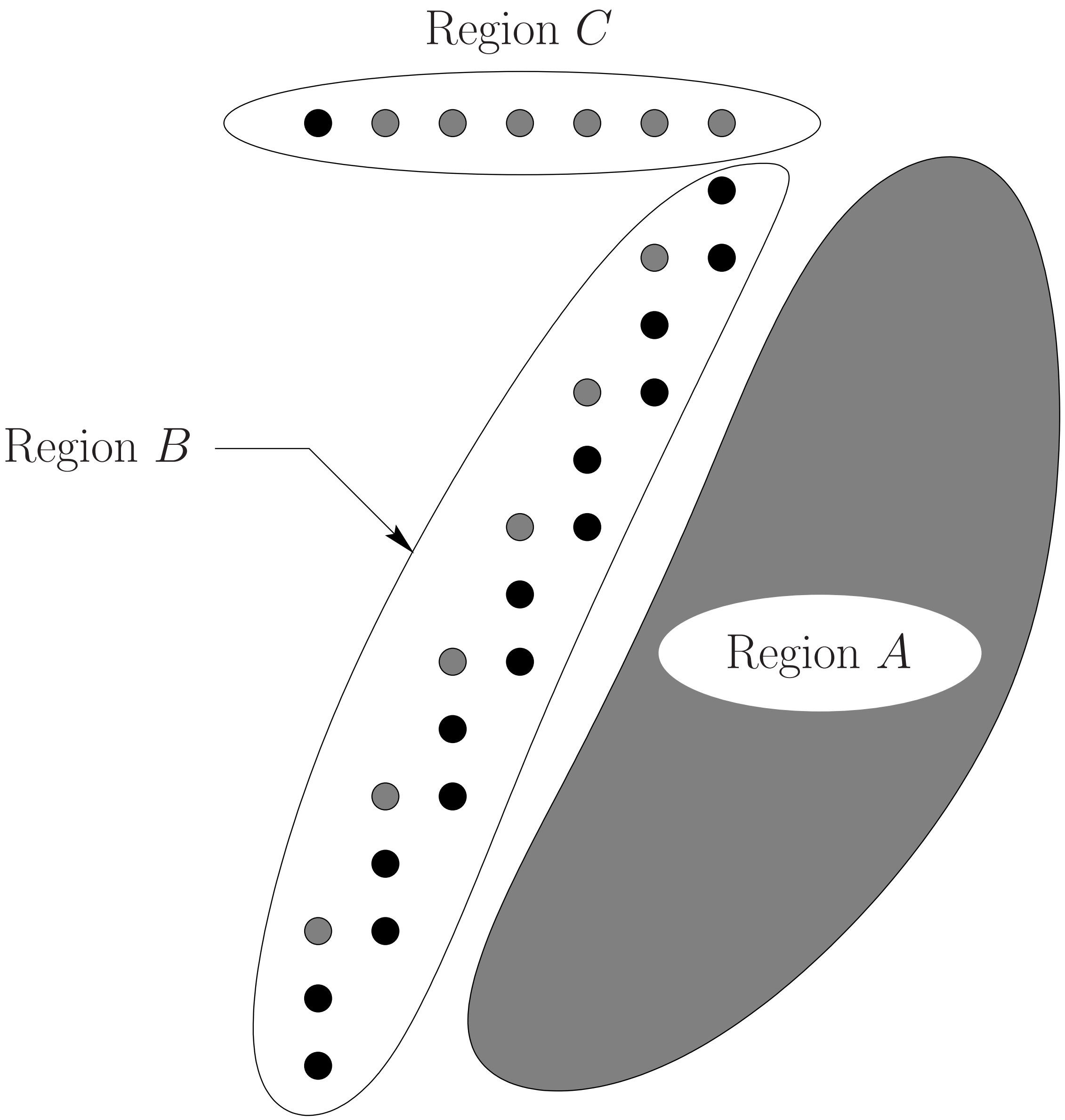

Figure 2-4: Image of $[\text{MUNRO}(S')](y_{(i,1)}, y_{(i,2|R_i|+1)})$: a replacement for R_i.

Lemma 2.22. *Let $R = (R_0, R_1, \ldots, R_m)$ be a valid reorganization tree service sequence. The tree R_i in R is a dual-Cartesian tree on its keys if they are ordered by the index when they are newly touched in R, with ties broken arbitrarily.*

Proof. If we newly touch a node $p \in R_i$ at j, we must have newly touched p's parent (if it exists) in R_i at $k \leq j$. □

Lemma 2.23. *If the necessary MUNRO properties hold then there is a MUNRO ordering so that the structure of the reorganization tree of* MUNRO(S') *at index $y_{(i,2|R_i|+1)}$ is the same as R_i.*[3]

Proof. This is a simple consequence of Lemma 2.22. If a node is in R_k, then we will access it in the index range $[y_{(k,1)}, y_{(k,2|R_k|+1)}]$. Thus both R_i and the $y_{(i,2|R_i|+1)}$th reorganization tree in MUNRO(S') are the dual-Cartesian trees when keys are ordered by the index of their next occurrence *in* $(R_{i+1}, \ldots, R_m)$, with ties broken arbitrarily. If we break ties in the same way, then the trees will be identical. □

Corollary 2.24. *In the statement of Lemma 2.23, instead of changing the MUNRO ordering of the given tree, we may instead pick a different optimal reorganization tree sequence R.*

Proof. This follows directly from the proof of Lemma 2.23. □

Base case. We now prove the necessary MUNRO properties hold by induction. The base case is $i = 1$. As there are no points below $y_{(1,1)}$, the only unstabbed boxes in $[\text{IAN}(Q)](y_{(1,1)}, y_{(1,2|R_1|+1)})$ below $y_{(1,2|R_1|+1)}$ are between two points with x coordinates adjacent in R_1. There are then unstabbed boxes between the top point in Q_i and $(r_{(1,j)}, y_{(1,2|R_1|+1)})$ for all $j \in [1, |R_1|]$, so the x coordinates of points in IAN(Q) at $y_{(i,2|R_1|+1)}$ is the set of keys from R_1. Region A for this case is empty as there are no points below $y_{(1,1)}$. Note that there are exactly $4|R_1|$ points in regions A and C, and no other points in $[\text{IAN}(Q)](y_{(1,1)}, y_{(1,2|R_1|+1)})$.

[3] There may be other valid MUNRO orderings; the definition lets us "break ties arbitrarily." See page 47.

Inductive case. Now assume that the inductive hypothesis holds for $j \leq i$. The second necessary MUNRO property and Lemma 2.23 ensure that the section tree of $\mathrm{IAN}(Q)$ at $y_{(i,2|R_i|+1)}$ is the same as the ith BST in the expanded sequence of R. The subsequence $[\mathrm{MUNRO}(S')](y_{(i+1,1)}, y_{(i+1,2|R_i|+1)})$ only accesses nodes in R_{i+1} so no other nodes will be touched. This also implies that Figure 2-4 (page 53) is accurate in that there are no nodes above region A save those in region C.

We now only have to show that the third necessary MUNRO property holds. $[\mathrm{MUNRO}(S')](y_{(i+1,1)}, y_{(i+1,2|R_i|+1)})$ accesses each node twice in R_{i+1} in key order. This brings each node to the root of the tree.

Lemma 2.25. *During the accesses in* $[\mathrm{MUNRO}(S')](y_{(i+1,1)}, y_{(i+1,2|R_i|+1)})$*, a node only appears on an access path at depth greater than two at most once.*

Proof. As we access nodes in ascending order, all nodes on the access path will be rearranged into a single path with the smaller keys higher in the tree. If p is at depth greater than two during one access, it will not be touched again until it is the child of the root of the subtree containing the keys in R_{i+1}. It is then at depth two. Although it may subsequently be pushed down, it will never again be touched when it is at depth greater than two. □

The points in region A can never correspond to nodes at the root in $\mathrm{MUNRO}(S')$ under $M_{\mathrm{RT}\to\mathrm{STAB}}$; the root is *always* in region B or C by definition of MUNRO ordering. This implies there are at most $|R_i|$ points in region A from nodes that MUNRO first touches at depth greater than two in $[\mathrm{MUNRO}(S')](y_{(i+1,1)}, y_{(i+1,2|R_i|+1)})$, and at most $|R_i|$ from when a node at depth two is touched. Regions B and C contain $4|R_i|$ points between them, so there are $6|R_i|$ points in $[\mathrm{IAN}(Q)](y_{(i,1)}, y_{(i,2|R_i|+1)})$. □

2.3.5 BST splitting algorithms

In this section, we define a linearly splittable online BST algorithm. In Section 2.3.6 we show that an online version of such an algorithm will let us make an online version of MUNRO.

A *forest* of BSTs is a set of BSTs. Although forests in general may be fairly arbitrary, the key intervals of trees in our forests are disjoint. Moreover each tree in a forest is either *alive* or *dead*. An *associated key interval for node p in forest F* is the open interval of key space between the successor and predecessor of p's key in F.

To *split* a BST at a node p in a tree, bring p to the root of the BST, then divide the BST into tree distinct BSTs: the left subtree of p, p as a single-node tree, and the right subtree of p. If a forest F has a tree T with node p, *split F at p* by splitting T at p and replacing T in F with the three subtrees resulting from the split. *To split a forest F at key k* not in F, split at all nodes in F whose associated key interval contains k. By the way the key intervals are constructed, at most two such nodes exist.

To describe how we bring nodes to the top of their tree in the forest, we can use either rotations as from Chapter 1, or reorganization trees as from this chapter. In either case, for a single split, structural rearrangements are applied to at most one tree at a time in the forest, bringing the appropriate node(s) to the top of the tree. Once there, a *split* operator is then applied to divide the tree into two, three, or four parts. (We can split at one or two nodes, depending on how many associated key intervals the key at which we are splitting hits.)

When splitting at a node, we mark it as *dead*. Trees that are not dead are *alive* or *live*. Nodes in live trees are *live* and nodes in dead trees are *dead*. The *cost* of structural rearrangements is the same as the corresponding rearrangement models, while the cost for splitting is 0 if splitting at a dead node, and 1 otherwise. The *cost* of a splitting service sequence on keys K and sequence S is $|K|$ plus the cost of all splits and structural rearrangements.

A *BST splitting algorithm* is an algorithm that given a set of keys K and a sequence S of keys produces a splitting service sequence on K and S. An *online BST splitting algorithm* does the same, but constructs the initial tree on K before knowing any elements of S, and splits at each key in S before getting the next key. An *online BST splitting algorithm* has one more property: To perform a split at a key k, it is given only k and the single live tree containing all nodes with k in their associated key intervals. It cannot use any information not in the tree. Theorem 2.26 implies this definition makes sense.

Theorem 2.26. *When performing splits at a key k in a sequence, all live nodes containing k in their associated key intervals are in the same tree in the forest.*

Proof. There are at most two live nodes containing k in their associated key intervals. If k is in two associated key intervals, then there is no node in the forest with a key between the two nodes. However, for these two nodes to be in different trees in the forest, the sequence must have split at one of them, or at some node between them. This is a contradiction as both are live nodes. □

Search costs. The "time" spent serching for a key can be absorbed into the cost for structural rearrangements. Suppose that we are searching for a key k. We only need to pay for searches when k is in the associated key interval of any nodes in a live tree T. In this case the splitting algorithm brings the successor and predecessor of k in T to the root of T. The cost for this rearrangement is proportional to the search cost because the set of nodes touched in the rearrangement is a superset of the nodes touched in the search.

A *linear BST splitting algorithm* is a BST splitting algorithm that for any set K of keys runs with cost $O(|K|)$ on *any* sequence of keys S, of arbitrary length.

Linear offline splitting algorithm. An offline linearly splittable BST algorithm is easy to construct: let the initial tree on K be the dual-Cartesian tree when the nodes are sorted by the index of S when the are first split (with ties broken so we only split at the root of any tree). Since we split at all ancestors of a node p before splitting at p, we will only split at roots of live trees: no rearrangements are necessary, so we only pay for splits. The total cost is $2|K|$.

2.3.6 Online BST splitting algorithms to online Munro

Theorem 2.18 gives the following corollary, which is suggestive in conjunction with the splitting algorithms described above in Section 2.3.5.

Corollary 2.27. *Let* $S = (s_1, \ldots, s_m)$*,* $\text{MUNRO}(S) = (T_0, R_1, \ldots, R_m)$*, and* $R_0 = T_0$*. If* p *and* q *are in* R_i *and* p *is ordered before* q *in the MUNRO ordering of* R_i *at* i *then* p *is newly touched at or before the index when* q *is newly touched.*

Proof. Theorem 2.18 tells us p is first touched by the first key s_j such that $j > i$ and s_j is in the associated key interval of p in R_i, with a similar situation holding for q and a key s_k, with $k \geq j$. If $j = k$ then p and q are newly touched at the same index, while if $k > j$ q is newly touched after p is newly touched. □

This implies the following.

> **Definitions:** Let $\text{MUNRO}(S) = (T_0, R_1, \ldots)$ and $T = (T_0, T_1, \ldots)$ be the expanded sequence of $\text{MUNRO}(S)$. A *maximal untouched node set of* R_i *at index* j is a subset K_i of nodes of R_i such that no other nodes in R_i have keys in the key range of K_i, no nodes in K_i are newly touched at or before index j and K_i is a maximal subset of nodes from R_i satisfying these properties.

Corollary 2.28. *Let* $\text{MUNRO}(S) = (T_0, R_1, \ldots)$ *and* $T = (T_0, T_1, \ldots)$ *be the expanded sequence of* $\text{MUNRO}(S)$*. Let* K_i *be a maximal untouched set of nodes of* R_i *at* j*. The nodes with keys in* K_i *are a subtree of* T_k *for all* $k \leq j$*.*

Proof. Corollary 2.27 implies that any nodes in R_i touched before nodes in K_i will be placed above all nodes in K_i: the nodes of K_i must form a subtree of R_i no other nodes of R_i are in the key range of K_i. As no node in K_i is newly touched by any access s_k for $k \leq j$, the subtree is maintained until at least s_k. □

This leads us to define a forest for a Munro reorganization tree:

> **Definitions:** Let $\mathrm{MUNRO}(S) = (T_0, R_1, \ldots)$ and $T = (T_0, T_1, \ldots)$ be the expanded sequence of $\mathrm{MUNRO}(S)$. A *maximal untouched subtree of R_i at index j* is the subtree formed by the maximal untouched subset of nodes of R_i. The *forest of R_i at index j* is the union of all maximal untouched subtrees of R_i at j and the other nodes of R_i as single-node (dead) trees. The single-node trees comprising the nodes of R_i that have been touched at j are marked *dead*. The other trees in the forest are *alive*.

These forests have two useful properties, described by the following theorems.

Theorem 2.29. *Let* $\mathrm{MUNRO}(S) = (T_0, R_1, \ldots)$ *and* $T = (T_0, T_1, \ldots)$ *be the expanded sequence of* $\mathrm{MUNRO}(S)$*. The keys in the nodes of a live tree τ in the forest of R_i at j are disjoint from the keys in any live tree in the forest of any R_k for $i < k \leq j$.*

Proof. At index i, this property holds as all nodes in R_i are touched, so none of them can be in any live trees of earlier reorganization trees. This implies that the property holds for R_k at k for any $k > i$. Later accesses will newly touch nodes in the forest of R_k and R_i, removing nodes from the keys in the live trees in the two forests, never adding. This implies that the property holds for any $j > k$ and all $k > i$. □

Theorem 2.30. *Let* $S = (s_1, \ldots)$*,* $\mathrm{MUNRO}(S) = (T_0, R_1, \ldots)$ *and* $T = (T_0, T_1, \ldots)$ *be the expanded sequence of* $\mathrm{MUNRO}(S)$*. For each tree τ in the forest of R_i at j, there is a tree σ with exactly the same keys as τ in the forest from a BST splitting algorithm run on the keys in R_i and the sequence $(s_{i+1}, \ldots, s_j)$. The tree σ is dead if and only if τ is dead.*

Proof. This follows directly from the definitions of a BST splitting algorithm and the way MUNRO touches nodes. In particular, nodes are removed from the live portions of either forest if and only if a node is accessed in their associated key interval, and the only way to split a tree in to is to split at a key in the key interval of the tree. (See Corollary 2.27.) □

We now propose a simple construction: use a BST splitting algorithm to maintain the forests of each reorganization tree in the MUNRO reorganization tree BST sequence. Whenever we encounter a tree in a forest, use the splitting algorithm to split the tree of the forest at the two nodes. MUNRO places just these two nodes into the reorganization tree for the current access. It should be clear that we only hit the appropriate trees on the search path:

Corollary 2.31. *Let $S = (s_1, \ldots)$, $\mathrm{MUNRO}(S) = (T_0, R_1, \ldots)$ and $T = (T_0, T_1, \ldots)$ be the expanded sequence of $\mathrm{MUNRO}(S)$. Suppose that a live tree τ in the forest for R_i is on the search path for s_j for some $j > i$. The node s_j is in the associated key interval for some node p in τ in the forest of R_i at $j - 1$.*

Proof. This follows from Theorem 2.18. □

Now, the construction:

Theorem 2.32. *If there is an online linear BST splitting algorithm, there is a strict online BST algorithm that runs with cost proportional to the cost of MUNRO.*

Proof. Let $S = (s_1, \ldots, s_m)$ and let M_i be the ith rearrangement tree in $\mathrm{MUNRO}(S)$. We now describe a BST algorithm that uses a splittable BST algorithm to maintain subtrees which we call *live subtrees*. These are the subtrees consisting of live trees in the forests of each rearrangement tree in $\mathrm{MUNRO}(S)$. To mark bboundaries between the live subtrees, we mark the root of each subtree with a special *forest tree root* mark.

Use a linear online forest splitting algorithm to maintain the forests of each M_i. Instead of *splitting* at root r of a live tree, mark r's children with the *forest tree root* mark; they are the new roots of trees in the forest after splitting at r.

When searching for a node s_i in S, split each distinct tree found in the search path at s_i. Theorem 2.30 implies that the splitting algorithm will place all nodes in M_{i+1} onto the node-to-root path of s_i. We then use the online linear splitting algorithm to turn this path into a tree than may subsequently be split.

Now we show that the total cost for this BST algorithm is proportional to the cost from the linear splitting algorithm used to maintain the live subtrees. To do this, we use an accounting scheme. We do not charge the full cost of accessing s_i and performing the necessary rearrangements before accessing s_{i+1}. Instead, we charge the cost for performing a splitting of a tree τ in the forest of R_j to s_j and the cost for rearranging the node-to-root path of s_i after the splits to s_i. The cost charged to s_i for the initial rearrangement of its node-to-root path is $|M_i|$, while the costs for subsequent forest splits is $O(|M_i|)$ as we hypothesize a *linear* online BST splitting algorithm. □

2.3.7 Toward online linearly splittable BST algorithms

We strongly suspect an online linearly splittable BST algorithm may be constructed using red-black trees, and it is likely that splay trees are linearly splittable.

Mehlhorn shows a related result ([26]) in Exercise 33 on page 311. A BST joining algorithm is similar to a BST splitting algorithm, but the joining algorithm begins with a forest of isolated nodes, and puts them into a single BST through a sequence of joins of adjacent trees in the forest. Mehlhorn shows that if we use red-black trees to perform the joins, the total reorganization cost is proportional to the number of nodes. This strongly suggests that red-black trees provide an online linearly splittable BST algorithm as the joining operation is essentially the reverse of our splitting operation.

Splay trees may also provide an online linearly splittable BST. Sleator and Tarjan [30] conjecture that accessing the nodes of a splay tree in the preorder of any BST (on the same nodes) takes time $O(n)$. This conjecture has been proved only for the special cases of linear access [32] and preorder access of the initial tree [4]. This conjecture is suggestive as an online splitting algorithm effectively performs splits in the preorder access of a dual-Cartesian tree when we sort keys by the order of splitting.

Theorem 2.33. *Let S be a permutation of the integers 1 through n. The cost of an online splitting algorithm on the keys 1 through n and sequence S is the same as when* *we split at the keys in the preorder sequence of the dual-Cartesian trees on* 1 *through* n *with order given by* S.

Proof. Once we have split a tree into two subtrees, the order is S of splits at two keys, one in each subtree, does not matter. An online BST splitting algorithm only takes a tree and a node at which to split; the subsequent development of each new tree in the forest resulting from a split is independent of the other. □

We conjecture that if splay trees can perform preorder access in time linear in the size of the tree, then they can do it in linear time when we split trees after each access. Even if splay trees *cannot* perform preorder access in linear time, they may be able to perform splitting in linear time. Both questions are open.

2.4 Edge Stabbing

In this section, we discuss edge stabbing, a model similar to box stabbing but for the linear access problem instead of the binary search tree access problem. We begin by defining edge stabbing, then show how edge stabbing is related to linear search in a manner similar to the way box stabbing is related to binary search. Finally, we use key independent optimality by Iacono [20] to show that the offline algorithm for a modified linear search problem by Munro [27] is within a constant factor of optimal.

2.4.1 Definition

Edge stabbing is similar to box stabbing, but with a twist. The necessity of the asymmetry between x and y is discussed in Section 2.4.2.

> **Definitions:** An axis box b is *edge stabbed* by a point p if p is on one of the vertical edges of b, but is not a generator of b. An axis box b is *edge stabbed by a point set* P, or *edge satisfied in or by* P, if there is a point in P edge stabbing b. A point set P is *self edge stabbing* or *edge satisfied* if all axis boxes for all pairs of points in P are edge stabbed by P. Given

> a point set P, the *edge stabbing problem* posed by P is to find an edge satisfied set of minimum cardinality containing P.

2.4.2 A less useful definition

In Section 2.4.1, we defined edge stabbing. This definition may seem restrictive: we require a stabbing point be on the vertical edges of a box. In this section, we show that the less restrictive problem where stabbing the horizontal edges of a box is also allowed is equivalent to the box stabbing problem.

> **Definitions:** An axis box b is *uniformly edge stabbed* by a point p if p is on any edge of b, but not a generator of b. An axis box, b, is uniformly edge stabbed by a point set P or *uniformly edge satisfied in or by* P if there is a point in P uniformly edge stabbing b. A point set P is *uniformly self edge stabbing* or *uniformly edge satisfied* if all axis boxes for all pairs of points in P are uniformly edge stabbed by P. Given a point set, P, the *uniform edge stabbing problem* posed by P is to find a uniformly edge satisfied set of minimum cardinality containing P.

Lemma 2.34. *If a set is uniformly edge satisfied, it is satisfied.*

Proof. If a point uniformly edge stabs a box, it stabs the box. Thus, if every axis box in a set of points is edge uniformly edge stabbed, every box is stabbed. □

Lemma 2.35. *If a finite set of points is satisfied, it is uniformly edge satisfied.*

Proof. Let Q be a finite, satisfied set of points. For any pair of points in Q, the axis box that they generate is stabbed by some point in the box. Let p and q_0 be two points in Q that generate a non-degenerate axis box.

We will show that $B(p, q_0)$ is edge stabbed. Let q_1 be a point stabbing $B(p, q_0)$. If q_1 is on an edge of $B(p, q_0)$, we are done. If not, then box $B(p, q_1)$ is non-degenerate, so there is a point q_2 stabbing $B(p, q_2)$. We can construct a sequence of points of Q in this manner: if q_i is not on the exterior of $B(p, q_0)$ then $B(p, q_i)$ is non-degenerate, so there must be a point q_{i+1} stabbing $B(p, q_i)$. As q_i stabs box $B(p, q_{i-1})$, both the x and y coordinates of the sequence $(q_0, q_1, \ldots)$ must monotonically approach the x and y coordinates of p. This implies that no point is repeated. As Q is finite, the sequence must terminate at some q_k on the edge of $B(p, q_0)$: $B(p, q_0)$ is uniformly edge stabbed. □

Theorem 2.36. *A set is uniformly edge satisfied if and only if it is satisfied.*

Proof. This follows directly from lemmas 2.34 and 2.35. □

2.4.3 Linear search

Before we discuss edge stabbing, we review the model for linear search from Munro [27] that we use in Section 2.4.4. The basic problem is to repeatedly find keys in a list. To find a key in the list, we must touch the desired key and all keys in front of it in in the list. We may continue past the accessed key during a search and touch as many more keys in the list as desired. We then rearrange all touched keys arbitrarily before the next access. The total cost is the number of keys touched during all accesses. The initial list is assumed to be ordered by the time the keys are first touched while serving the sequence. We formalize this model below.

> **Definitions:** We again assume a fixed set of keys. For linear accesses, the keys can be unordered. However, in order to make the correspondence between linear access and edge stabbing, we assume that the keys are consecutive integers beginning with one. Let S be a sequence of keys. A *reorganization list sequence* is a sequence of lists of keys. These lists do not in general include all the keys. A reorganization list sequence represents a sequence of lists containing all the keys, called the *expanded sequence* for the reorganization list sequence. Let $L = (L_0, L_2, \ldots)$ be a reorganization list sequence, and $M = (M_0, M_1, \ldots)$ be its expanded sequence. The first

elements are the same: $M_0 = L_0$. For $i > 0$, we compute M_i from M_{i-1} and L_i. For L to be a valid RL sequence, the keys in L_i must be the first $|L_i|$ keys in M_{i-1}. M_i is M_{i-1} with the first $|L_i|$ keys removed and replaced with L_i. The first element of the reorganization list sequence is the *initial list*, and the other lists are called the *reorganization lists*. A *linear access service sequence* for a sequence of keys is a reorganization list sequence such that the ith reorganization list is the ith element of the sequence of keys. The *cost* of a linear access service sequences is the number of keys in all reorganization trees in the list. The linear access problem posed by a sequence keys is to find a linear access service sequence for the sequence of keys of minimum cost.

2.4.4 Edge stabbing and linear search

Edge stabbing is related to linear search in much the same way box stabbing is related to binary search. In this section, we describe this relationship and provide related proofs. The structure of this section is very similar to Section 2.1.3, to the extent that many theorems are duplicated with only a few changes of words.

Theorem 2.37. *For each linear access problem L there is a corresponding edge stabbing problem M such that the following hold:*

- *There is an efficiently computable, cost preserving map from access sequences for L to edge satisfied sets containing M.*
- *There is an efficiently computable, cost preserving map from edge satisfied sets containing M (on the lattice of M) to linear access service sequences for L.*

The proof of Theorem 2.37 has two major sections. After giving all maps mentioned in the theorem, we prove the maps between linear access service sequences and edge satisfied sets are valid and cost preserving. First we develop a little notation.

The notation is similar to that used in Section 2.1.3, but we use section lists rather than section trees.

> **Definitions:** For a set of points Q let $r_i^-(Q,j)$ be the largest y coordinate less than or equal to i among points with x value j. Let $r_i^+(Q,j)$ be the smallest y coordinate greater than i among points with x value j. The *section list for* Q *at* i is the list of keys if we order the nodes lexicographically first by $i - r_i^-(Q,\cdot)$ then $r_i^+(Q,\cdot)$ with ties broken arbitrarily. The *section of* Q *at* i is the set of all points in Q with y coordinate $\leq i$. If the point set is clear from context, the initial subscript may be dropped.

2.4.4.1 Maps

Map from linear access problems to ESPs. We call this map $M_{\text{LIN}\to\text{ESP}}$.

$$M_{\text{LIN}\to\text{ESP}}(x_1,\ldots,x_n) = \{(x_1,1),(x_2,2),\ldots,(x_n,n)\}.$$

Map from linear access service sequences to edge satisfied sets. This map is $M_{\text{SEQ}\to\text{ESAT}}$. Let S be a linear access problem, and L be a linear access service sequence for S. Let L_i be the ith reorganization tree of L. The point (k,i) is in $M_{\text{SEQ}\to\text{ESAT}}(L)$ if and only if the key k is in L_i. Each key in each reorganization list in L corresponds to exactly one point in the image, so $M_{\text{SEQ}\to\text{ESAT}}$ is cost preserving.

Map from edge satisfied supersets (on the lattice of the problem) to access service sequences. This map is $M_{\text{ESAT}\to\text{SEQ}}$. Let S be a linear search problem. Let Q be an edge satisfied set containing $M_{\text{LIN}\to\text{ESP}}(S)$. The key k is in the ith reorganization list of $M_{\text{ESAT}\to\text{SEQ}}(Q)$ if and only if the point (k,i) is in Q. Let $M_{\text{ESAT}\to\text{SEQ}}(Q) = (L_0, L_1, \ldots)$. The keys in L_i are sorted by $r_i^+(Q,\cdot)$.

2.4.4.2 Map from linear access service sequences to edge satisfied sets is valid

Theorem 2.38. *The map $M_{\text{SEQ}\to\text{ESAT}}$ is a valid map.*

Proof. Let A be a linear access service sequence for the linear access problem S. Let A_i be the ith reorganization list in A. For this proof, we do not need to know the order of the reorganization list after reordering, only that A is a valid reorganization list sequence. Let Q be $M_{\text{SEQ}\to\text{ESAT}}(A)$.

For sequences of length 1, the theorem holds: a set of points all with the same y coordinates is edge satisfied. Now suppose that the theorem holds for all sequences of length less than i. This implies that the section of Q at i can only have non-edge-stabbed boxes between points with y coordinate i and points with lower y coordinates. Let $B(p, q)$ be such an non-edge-stabbed box with p at y coordinate i, and q at y coordinate $j < i$.

By definition of $M_{\text{SEQ}\to\text{ESAT}}$, $x(q)$ is in A_j, but $x(p)$ is not: $x(q)$ is ahead of $x(p)$ in the main list at time j. Similarly $x(p)$ is in A_i but $x(q)$ is not: $x(p)$ is ahead of $x(q)$ at time j. In order to reverse the relative order of two keys in the main list, both must be touched at some time $k \in (j, i)$. Since A_k contains $x(p)$ and $x(q)$, the points $(x(p), k)$ and $(x(q), k)$ are present in Q, and so edge stab $B(p, q)$. □

2.4.4.3 Map from edge satisfied sets to linear access service sequences is valid

Lemma 2.39. *If the following hold,*

- *S is an linear access problem,*
- *Q is an edge satisfied subset of the lattice of $M_{\text{LIN}\to\text{ESP}}(S)$,*
- *Q contains $M_{\text{LIN}\to\text{ESP}}(S)$, and*
- *A is the image of Q under $M_{\text{ESAT}\to\text{SEQ}}$,*

then A is a valid reorganization list sequence.

Proof. Let Q_i be the section list of Q at i. We show that the sequence $(Q_0, Q_1, \ldots)$ is the expanded sequence of A. The first lists of both sequences match, as they both consist of all the keys sorted by $r_0^+(\cdot)$. Now we need to show that the keys in the ith list of A for all $i \geq 1$ are at the front of the section list of Q at time $i-1$.

Let L_i be the ith reorganization list of A. We want to show that for each key q in L_i, either q is at the front of Q_{i-1} or the key directly in front of q in Q_{i-1} is also in L_i. If q is at the front of Q_{i-1}, we are done. Suppose that there is a key in front of q in Q_{i-1}: p. For this proof we drop the (pre)scripts on $r^-(Q,\cdot)$ and $r^+(Q,\cdot)$ to get r^- and r^+ respectively.

By definition of section list, $r_{i-1}^-(p) \geq r_{i-1}^-(q)$. First we consider when equality holds. In this case, we must have $r_{i-1}^+(p) \leq r_{i-1}^+(q)$. By definition, $r_{i-1}^+(\cdot) \geq i$. Since $r_{i-1}^+(q) = i$, we must have $r_{i-1}^+(p)$: p is in L_i.

Now suppose $r_{i-1}^-(p) > r_{i-1}^-(q)$. The inequality implies that $B((p, r_{i-1}^-(p)), (q, i))$ can only be edge stabbed at i, so (p, i) is in Q. □

Theorem 2.40. *If S is a linear access problem, and and Q is an on lattice edge satisfied set containing $M_{\text{LIN}\to\text{ESP}}(S)$, then $M_{\text{ESAT}\to\text{SEQ}}(Q)$ is a valid linear access service sequence for S.*

Proof. By construction, (k, i) is in the ith reorganization list of $M_{\text{ESAT}\to\text{SEQ}}(Q)$, and by Lemma 2.39, $M_{\text{STAB}\to\text{RT}}(Q)$ is a valid reorganization list sequence. This is the definition of a valid linear access service sequence. □

2.5 Optimality of Linear Search

In this section, we use the results from Sections 2.4.4 and 2.1, and by Iacono [20], to show that the offline linear search algorithm proposed by Munro [27] is within a constant factor of optimal.

2.5.1 Satisfaction and edge satisfaction

In this section, we show that a lower bound for a box stabbing problem on a permutation of a sequence of keys is a lower bound for the ESP on the sequence.

Lemma 2.41. *An edge satisfied set is satisfied.*

Proof. If a point edge stabs a box, it stabs the box. □

This is an interesting property as it implies that any lower bound on a box stabbing problem is also a lower bound for the same set of points posed as an edge stabbing problem. This property becomes even more interesting if we note that for the linear access problem, the key ordering is irrelevant.

Lemma 2.42. *If a set Q is edge satisfied and we arbitrarily permute the x coordinates of Q to get Q', then Q' is edge satisfied.*

Proof. Consider the axis box $B(p, q)$ with $y(p) < y(q)$. If the point r edge stabs the box $B(p, q)$ before the permutation, r shares the x coordinate of either p or q and $y(r) \in [y(p), y(q)]$. Let p', q', r' be the images of p, q, r respectively. After permutation, $r' \notin \{p', q'\}$, but r' does share x coordinates with either p' or q' and $r' \in [y(p'), y(q')]$, so r' stabs $B(p', q')$. □

These lemmas give us the desired result of the section.

Lemma 2.43. *If P be a set of points, P' is P with x coordinates arbitrarily permuted, and $L_{\text{box}}(P')$ is a lower bound on the number of points we must add to P' to get a satisfied set, the bound $L_{\text{box}}(P')$ is also a lower bound on the number of points we must add to P to get an edge satisfied set.*

Proof. Suppose that we can add fewer than $L_{\text{box}}(P')$ points to P to get an edge satisfied set Q. Lemma 2.42 tells us if we permute the x coordinates of Q using the P to P' permutation, we get an edge satisfied set Q' containing P'. Since Lemma 2.41 implies that Q' is satisfied, $L_{\text{box}}(P')$ is not a lower bound. □

2.5.2 BST and linear access service sequences

We can reformulate the results of Section 2.5.1 in terms of BST and linear access service sequences.

Theorem 2.44. *Let U be a sequence of keys, and let $b(U)$ be an arbitrary permutation of the sequence. If L is a lower bound on the cost of a reorganization tree BST access service sequence for $b(U)$, then L is also a lower bound on the cost of a linear service sequence for U.*

Proof. The results of Sections 2.1 and 2.4 imply the bounds on BST access service sequences and linear access service sequences are the same as bounds on the box stabbing problem and ESP respectively. Lemma 2.43 implies that a lower bound on the box stabbing problem for an an arbitrary permutation of the key values is equivalent to a lower bound on the the edge stabbing problem on the same set of points. □

2.5.3 Lower bounds on linear access service sequences

We can now use Iacono's key-independent optimality results [20] to provide lower bounds on linear access service sequences.

> **Definitions:** Let $U = (u_1, \ldots, u_|U|)$ be a sequence of integers and $b(U) = (b(u_1), b(u_2), \ldots)$ be an arbitrary permutation of the integers. Let ℓ_i be the index in U of the last access to u_i before the ith, or 1 if i is the first access to u_i in U. For each index $i \in [1, |U|]$, the *working set* of element u_i is the set of distinct integers in U in the index range $[\ell_i + 1, i]$. Let $w_s(U, i)$ be the size of the working set of element i in U.

The result from Iacono [20] of interest to us is the following. We state it without proof:

Theorem 2.45. *If U is a sequence of integers, and $b(U)$ is a (uniformly) random permutation of the integers in U, then the expected cost of a reorganization tree BST access service sequence for $b(U)$ is at least*

$$\Omega\left(\sum_{i=1}^{|U|} w_s(U,i)\right).$$

Although we use a different cost model than Iacono uses [20], the cost of his model is within a constant factor of the cost model used in this thesis, so the theorem still applies. The following corollaries give us the desired bounds.

Corollary 2.46. *If U is a sequence of integers, there is a permutation of U, $b(U)$, such that the cost of a reorganization tree BST access service sequence on $b(U)$ is at least*

$$\Omega\left(\sum_{i=1}^{|U|} w_s(U,i)\right).$$

Proof. The expected value of the lower bound for a BST access service sequence cannot be greater than the largest lower bound for such a sequence. □

Corollary 2.47. *If U is a sequence of integers, the cost of a linear access service sequence on U is at least*

$$\Omega\left(\sum_{i=1}^{|U|} w_s(U,i)\right).$$

Proof. This follows from Theorem 2.44 and Corollary 2.46. □

2.5.4 An optimal linear access service sequence

We now have all the results collected. The paper from Munro [27] describes an algorithm that on a sequence of integers U achieves a linear access service sequence on U with cost

$$O\left(\sum_{i=1}^{|U|} w_s(U,i)\right).$$

This gives the desired result of 2.5:

Theorem 2.48. *The algorithm for constructing linear service sequences described by Munro [27] is within a constant factor of an optimal algorithm for constructing linear access service sequences.*

Proof. The asymptotic cost of the linear access service sequences from Munro [27] matches the lower bound on these sequences from Corollary 2.47. □

2.6 Conclusion and Open Questions

Major result. In this chapter, we introduced the *box model* of binary search trees, and the related question, the *box stabbing problem*. The major result using this new model is Theorem 2.48: the linear access service sequence proposed by Munro [27] is $O(1)$-competitive.

Dynamic optimality. Unfortunately, the box model did not move us far along in providing a dynamically optimal BST. In fact, the major question posed by the box model is equivalent to computing a dynamically optimal BST: Is there a polynomial algorithm that can compute an $O(1)$-competitive satisfied set? Although we have not been successful in finding such an algorithm, we describe some methods we have looked at in Chapter 5. Moreover, we hypothesize that MUNRO is dynamically optimal, and also has a strict online version.

How complex is the box model? There are, however, some interesting auxiliary questions related to the box model, such as: is the box stabbing problem NP-complete? It is clearly in NP, as we can guess some satisfied subset of the lattice of the problem of appropriate size if it exists, but we have not yet been able to prove it is NP-hard. Some reduction approaches using planar 3SAT look promising, but are not yet conclusive [19].

Other questions. With the introduction of a new model, one might ask if there are other models. Is there another (substantially different) model of the BST access problem that makes computing an $O(1)$-competitive solution more practical? It is not clear this will be useful even if we find more models.

Chapter 3

Lower Bounds on BST Access Problems

In this chapter, we unify and extend the known lower bounds for BST access problems. First we introduce two novel bounds, the cut and independent set lower bounds, and show that they are equivalent. These bounds are powerful: all earlier types of lower bounds may be phrased as cut lower bounds. We then describe an $O(1)$-competitive independent set lower bound. It is as large as any independent set lower bound within a constant factor. The chapter concludes with hypotheses and open questions related to the new bounds.

Independent discovery. Although we did not publish the cut and independent set lower bounds, we discovered them in the summer of 2004. A bound similar to the cut bound was discovered independently by a group working with Sleator ([12],[11]) in 2005.

3.1 Cut Bounds

This section describes cut bounds and shows that they provide a lower bound on the cost of an optimal solution to a box stabbing problem. First we introduce vocabulary for cuts and cut sets, and flesh out some of their properties. Section 3.1.2 then describes cut bounds and proves they are lower bounds on the cost of an optimal box stabbing problem.

3.1.1 Cuts

In this section, we define cuts, cut sets, and crossing numbers for cuts. These concepts form the basis of the cut bound.

> **Definitions:** Let P be a box stabbing problem problem and Q a satisfied set containing P. A *primary monotone Manhattan path* is a monotone Manhattan path in Q between two points in P. Consider an unsatisfied box in P. Though we might not know where monotone Manhattan paths from the generators will pass, any primary monotone Manhattan path between them must cross from the left edge to the right edge of the box. This simple observation leads us to propose cuts.
>
> A *cut* is a segment of a curve in the plane. Though cuts can be more varied than simple line segments, we only consider vertical line segments. A cut *splits* an unstabbed axis box if the cut's x coordinate is in the interior of the box's x coordinate range and the y range of the cut contains (not necessarily strictly) the y range of the box. A cut *nicks* an unstabbed box if it intersects the interior of the box. A *cut set* is an ordered set of cuts. A cut in a cut set *cuts an unstabbed axis box* if the cut splits the box, and no cuts earlier in the set nick the box. The set of unstabbed axis boxes that a cut cuts is the *cutting set* of the cut. The size of a maximal set of boxes with disjoint interiors in a cutting set is the *crossing number* of the cut. We describe below how to compute a such maximum subset, the *crossing set*.
>
> We will need to link the point set from which we draw the unsatisfied boxes to the cut set we use to get a lower bound. If the unsatisfied boxes

> come from a point set P, then a relevant quantity or set related to the cut set C is said to be *of C over P*. For example, we might refer to the crossing set of c in C over P, or the sum of crossing numbers of C over P.

Computing the cutting set of a cut in a cut set is straight forward, but it may not be clear there is an efficient means of computing the crossing number. The following procedure does the trick, as we show below (Theorem 3.1).

> **Definitions:** To compute the *crossing set* of a cut, initialize the crossing set to the cutting set. Discard from the crossing set any box with a y range that contains the y range of another box in the cutting set.

Theorem 3.1. *The size of the crossing set of a cut is the crossing number for the cut.*

Before we can prove Theorem 3.1, we need a few lemmas regarding the properties of cuts.

> **Definitions:** A $+$ type ($-$ type) box is an axis box with generators on a line with positive (negative) slope.

Lemma 3.2. *Let P be a point set. If a and b are $+$ and $-$ type unstabbed boxes in P respectively, neither can contain a corner of the other in its interior.*

Proof. Let a have upper generating corner a_u and lower generating corner a_l, and define b_u and b_l similarly. Suppose that a contains one empty corner of b. Since a is not stabbed, a must contain either the lower left or upper right corner of b. If it is the lower left, and we require that a be unstabbed, a cannot contain another corner of b. This implies that $y(b_u) > y(a_u)$ and $x(b_l) > x(a_u)$. However these inequalities imply b contains a_u, a contradiction. This is illustrated in Figure 3-1.

Similarly if a contains the upper right corner of b, $y(b_l) < y(a_l)$ and $x(b_u) < x(a_l)$ as a is unstabbed. These inequalities imply b contains a_l. This is illustrated in Figure 3-2. □

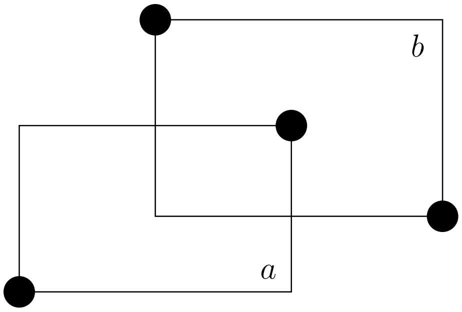

Figure 3-1: a contains the lower left corner of b.

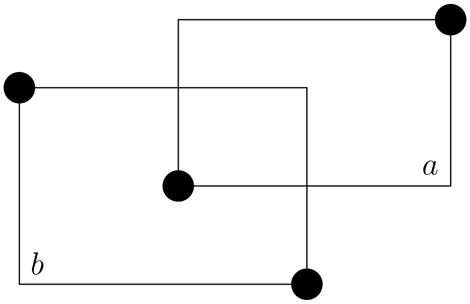

Figure 3-2: a contains the upper right corner of b.

Lemma 3.3. *If a and b are $+$ and $-$ type unstabbed boxes respectively with intersecting interiors, the y range of one contains the y range of the other.*

Proof. The boxes' interiors overlap, so their x and y ranges intersect. Suppose that b's y range contains the upper endpoint of a's y range. Applying Lemma 3.2 twice shows that the x range of b cannot extend to the endpoints of a's x range, and then that the y range of b must extend below the y range of a. The argument when a contains the upper endpoint of b's y range is similar. This is illustrated in Figure 3-3. □

Lemma 3.4. *The boxes in the crossing set of a cut have disjoint interiors.*

Proof. Suppose that boxes a and b in a crossing set have intersecting interiors. The discard algorithm and Lemma 3.3 tells us that without loss of generality a and b are both $+$ type unsatisfied boxes. Again let a_u, a_l, b_u, and b_l be the upper and lower generating corners of their respective rectangles. Assume $y(a_u) > y(b_u)$. We now

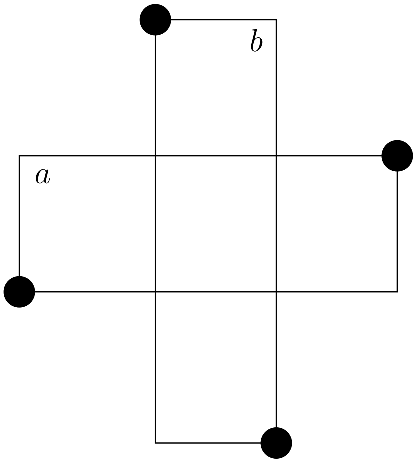

Figure 3-3: b's y range contains a's y range.

argue that the boxes are in the configuration shown in Figure 3-4. Since the y range

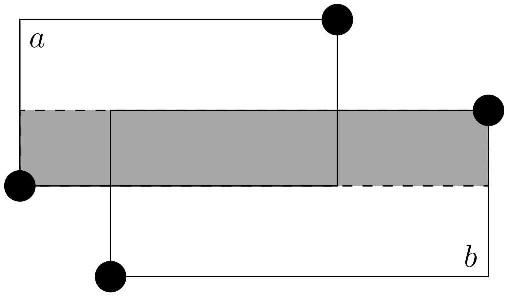

Figure 3-4: Intersecting boxes in a crossing set.

of one cannot contain the y range of the other, we must have $y(b_l) < y(a_l) < y(b_u) < y(a_u)$. As the boxes are unstabbed, and their interiors intersect, the x coordinates must be related in the following manner: $x(a_l) < x(b_l) < x(a_u) < x(b_u)$.

Note that the shaded box in Figure 3-4, $B(a_l, b_u)$, is also unsatisfied. As $B(a_l, b_u)$ has an x range containing a and b and y range contained by both a and b, the following properties hold:

- Any cut splitting both a and b also splits $B(a_l, b_u)$.

- Any cut nicking $B(a_l, b_u)$ also nicks either a or b.

Together these imply $B(a_l, b_u)$ is in a cutting set if both a and b are in the set, a contradiction. A similar argument holds when $y(a_u) < y(b_u)$. □

Lemma 3.5. *Two boxes in a cutting set will have intersecting interiors if and only if their y ranges have intersecting interiors.*

Proof. This is fairly obvious, but we state it explicitly. The interior of all boxes in the cutting set is split by a (single) vertical line, so all of their x ranges have intersecting interiors. Boxes have intersecting interiors if and only if their x and y ranges have intersecting interiors. □

We can now finish with the proof of Theorem 3.1 (stated on page 77).

Proof. Let X be a maximum subset of boxes of the cutting set such that the boxes in X have disjoint interiors. Suppose that a is in X. If b is in the cutting set and the y range of a contains the y range of b, we may replace a with b in X. By Lemma 3.5, the interior of b cannot intersect the interior of any box that the interior of a does not. We can therefore replace a with b in X to get another maximum interior-disjoint subset of the cutting set.

This argument implies that there is a maximum interior disjoint subset of the cutting set with boxes strictly from the crossing set. The boxes of the crossing set are interior disjoint by Lemma 3.4. □

3.1.2 The cut bound

In this section, we define and prove the cut lower bound.

> **Definitions:** Let P be a point set and Q be a satisfied superset of P on the lattice of P. The *cut lower bound C over P* is the sum of cutting numbers for all the cuts in C over P. The *adjacency graph* of a satisfied point set has vertices for each point in the set, and edges between vertices if the corresponding points share either x or y coordinates and there are no other points in the set between them. For purposes of computation, the

x coordinate range of an edge in the adjacency graph is an open interval, not a closed one.

We will label certain points and horizontal edges in the adjacency graph of Q, based on C and unsatisfied boxes in P. The labels are descriptors of unsatisfied boxes in P. We provide each edge and each point in the adjacency graph with two slots for labels, and each slot can describe or point to exactly one box. The slots are called the *top slot* and *bottom slot.* Once a slot has a label, the slot is *filled.* We cannot add another label to a filled slot.

For a cut c and box b in c's cutting set over P, a *critical object* for b is the point or edge of Q's adjacency graph where a monotone Manhattan path between b's generators crosses c. The top slot of a critical object is *critical* if the object is at the top edge of b, the bottom slot is critical if the object is at the bottom edge of b, and both slots are critical otherwise.

Lemma 3.6. *The x coordinate range of any critical object for box b is strictly contained in the x range of b's x coordinate range. The y coordinate of any critical object for box b is in the y range of b.*

Proof. By definition of cutting, the x coordinate of c is strictly in the x range of b. If the critical object is a point, we are done. If the critical object is an edge, the x range of the edge must be contained in the x range of b as the edge is on a monotone Manhattan path between the generators of b. The x range of the edge is open and the x range of the box is closed.

As the critical object is on a monotone Manhattan path between the generators of b, its y coordinate is in the y range of b. □

We now introduce the marking procedure.

Labelling For each cut, compute a set of interior disjoint boxes. For each cut in order, and for each box in the set of interior disjoint boxes for the cut, place a label corresponding to the box on a critical slot of the box.

Lemma 3.7. *The marking procedure never tries to add a label to a filled slot.*

Proof. Suppose that the labeling procedure tries to add the label describing box b' to a filled slot. Label b' is added while processing the cut c'. The label for box b in the cut set for cut c is already in the slot.

This cannot happen if c and c' are the same cut. If they were, the interiors of b and b' would be disjoint. Thus they cannot share critical slots: if b and b' intersect at all, the top of one is the bottom of the other.

By definition of critical slots and objects, both c and c' intersect the critical object, hence are both in the interior of the x coordinate ranges of b and b'. There are now three cases to consider:

- If the object for b' is in the interior of b''s y range, then c nicks b', a contradiction.
- If the critical object for b' is at the top of b', then the critical object is either at the top of b or somewhere in the middle. (The critical slot is a top slot). In either case, the interior of the y ranges of b and b' must overlap. As c cuts b, its y range extends the entire length of b. This implies that c also intersects the interior of the y range of b': c nicks b'.
- If the critical object for b' is at the bottom of b' a similar argument holds.

□

Theorem 3.8. *If P is a point set and Q is a satisfied point set containing P, any cut lower bound over P is at most $4|Q|$.*

Proof. The number of labels added is equal to the cut set by Lemma 3.7. There are at most $|Q|$ points and $|Q|$ horizontal edges in the adjacency graph. As there are at most 2 labels for each point and edge, there are at most $4|Q|$ labels. □

3.1.3 Weaker cut bounds

In some cases (in Chapter 4, for example) it is not necessary to compute the cutting number. We can, instead, compute a lower bound for the cutting number.

Corollary 3.9. *Let S be a sequence of keys, and let C be a cut set. Let $\alpha_c(S, C)$ for all $c \in C$ be a set of interior disjoint boxes in the cut set of c in C over $M_{\mathrm{BST} \to \mathrm{BSP}}(S)$ (recall this map from page 38). $\Sigma_{c \in C} |\alpha_c(S, C)|$ is a lower bound (within a factor of 4) for the reorganization tree BST problem posed by S.*

Proof. This follows from Theorem 3.8 and the fact that $|\alpha_c(S, C)|$ is a lower bound for the cutting number of c in C over $M_{\mathrm{BST} \to \mathrm{BSP}}(S)$. □

3.2 Independent Set Bound

In this section, we describe the independent set bound and show that it is equivalent to the cut bound. We do not show a direct relation. We show how to compute a cut bound from each independent set bound at least as large, and visa versa. For lower bounds this is tantamount to equivalence.

3.2.1 The interaction graph and the independent set lower bound

Here we give more definitions. The bound is simple to describe compared to the cut bound, but proving it is a lower bound is difficult without the cut bound.

> **Definitions:** Two unsatisfied boxes *interact* if one contains an (empty) corner of the other in its interior. The *interaction graph* of a point set P is a graph with one vertex for each unsatisfied box in P and edges between vertices if the corresponding boxes interact. The size of an independent set in the interaction graph of P *independent set bound on* P.

3.2.2 Cut lower bound implies independent set lower bound

In this section, we prove the following theorem:

Theorem 3.10. *For every cut lower bound over a point set, there is an independent set bound on the set of at least the same size.*

Lemma 3.11. *If C is a cut set and P a point set, the union of all crossing sets for C over P is an independent set in the interaction graph of P.*

Proof. The proof of this lemma uses an argument similar to that of Lemma 3.4. Lemma 3.3 tells us $+$ and $-$ type unsatisfied boxes cannot interact with each other. Suppose that boxes a and b are two $+$ type boxes in the union of crossing sets for C over P and that they interact with each other.

Let a_u, a_l, b_u, and b_l be the upper and lower generating corners of their respective rectangles. Without loss of generality $y(a_u) > y(b_u)$. The assumption of interaction and that a and b are unsatisfied imply $y(b_l) < y(a_l) < (y(b_u) < y(a_u)$ and $x(a_l) < x(b_l) < x(a_u) < x(b_u)$, as shown in Figure 3-5. A more extended argument justifying this ordering may be found in Lemma 3.4.

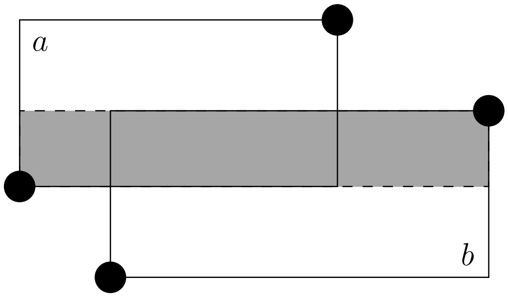

Figure 3-5: a and b interact.

Note that the shaded box in Figure 3-5, $B(a_l, b_u)$, is also unsatisfied. As $B(a_l, b_u)$ has an x range containing a and b and y range contained by both a and b, the following properties hold:

- Any cut splitting both a and b also splits $B(a_l, b_u)$.

- Any cut nicking $B(a_l, b_u)$ also nicks either a or b.

Together these imply $B(a_l, b_u)$ is in the cutting set for the earliest cutting set that either a or b is in. This implies that either a or b is not in a cutting set, a contradiction.

□

Now we can prove Theorem 3.10 (stated on page 84):

Proof. Lemma 3.11 implies that the union of crossing sets for the cut bound of a cut set C over the point set P is independent. □

3.2.3 Independent set lower bound implies cut lower bound

Now we prove the other direction:

Theorem 3.12. *For every independent set bound, there is a cut lower bound of at least the same size.*

3.2.3.1 An ordering

To prove this, we construct an appropriate cut lower bound. First we impose some structure on the independent set. Let P be a point set, and let S_I be an independent set of boxes in the interaction graph of P. If the interiors of two boxes in S_I overlap, the definition of interaction implies that one is wider than the other. We call the narrower box a *descendent* of the wider box. Impose a partial order on S_I by requiring that all boxes are ordered before all their descendants. Extend the partial order to a full order in an arbitrary fashion; call this order the *independent set order*.

3.2.3.2 A cut set

For each box $b \in S_I$, we define a function $x_c(b)$ with the following properties:

- The value of $x_c(b)$ is always strictly in the interior of the x range of b.
- If there is a descendent of b in S_I, $x_c(b)$ is the leftmost x coordinate of an edge of the first descendent of b subject to the first condition.

- If there is not a descendent of $b$in S_I, $x_c(b)$ may be chosen arbitrarily subject to the first condition.

Now we construct the cut set C. Let c_i be the ith element of C and let b_i be the ith box of S_I under the independent set order. The cut c_i has x coordinate $x_c(b_i)$ and shares the y range of b_i.

3.2.3.3 The cut set is large enough

Now we show that the cut set for each c_i is at least one. The following lemma does most of the work.

Lemma 3.13. *Cut c_i does not nick any descendents of b_i in S_I.*

Proof. The lemma holds if b_i has no descendents, so assume that there is a descendent. Let b_j be the first descendent of b_i in S_I. As c_i is on an edge of b_j, it cannot nick b_j. Suppose c_i nicks another descendent b_k of b_i. By definition of nicking, there is an open ball around some point of c_i completely contained in b_k. Since c_i is on the edge of b_j, this ball also intersects the interior of b_j, so b_j and b_k have intersecting interiors. As they do not interact, b_j is ordered before b_k, and their interiors intersect, b_j's x range must contain b_k's x range. As c_i on the edge of b_j's x range, c_i cannot nick b_k. □

Lemma 3.14. *Cut c_i does not nick b_j for any $j > i$*

Proof. Suppose c_i nicks b_j for $j > i$. Lemma 3.13 implies that b_j is not a descendant of b_i so b_i and b_j have disjoint interiors. Any open ball about any point on c_i intersects the interior of b_i. If c_i nicks b_j, then an open ball about some point of c_i is completely contained in the interior of b_j: the interiors of b_i and b_j intersect, a contradiction. □

Lemma 3.15. *The cutting number of c_i in C over P is at least one.*

Proof. By construction, c_i splits b_i. Lemma 3.14 implies that box b_i is not nicked by any cut c_j for $j < i$, so b_i will be in c_i's cut set. As the cut set of c_i is non-empty, its crossing set will be non-empty. □

This implies that the cut bound of C on P is at least $|S_I|$, proving Theorem 3.12.

3.2.3.4 Conclusion

Corollary 3.16. *The largest cut and independent set lower bounds on a point set are equal.*

Proof. This follows from theorems 3.12 and 3.10. □

3.3 Wilber I

In this section, we show that the first bound described by Wilber [33] is closely related to a cut bound. We first review the original bound, then describe the related cut bound and prove the relationship between the two.

3.3.1 Bounds and structures

The two bounds are bounds on different structures: the original Wilber I bound is a bound on cost of an optimal access service sequence, but the equivalent cut bound is a bound on cost of an optimal satisfied set. These bounds are related through the map $M_{\text{BST}\to\text{BSP}}$ from Section 2.1.3 (page 38): if the original Wilber I bound provides a lower bound on the RT BST access problem, S, then the equivalent cut bound provides the same lower bound on $M_{\text{BST}\to\text{BSP}}(S)$.

3.3.2 Original bound: Wilber I

The original Wilber I bound is based on constructing a bound tree for the given set of nodes. Without loss of generality the nodes are 1 through n in the BST problem. A *Wilber I bound tree* is a BST with integers 1 through n at the leaves and the half-integers $1 + 1/2$ through $n - 1/2$ in the internal nodes.

> **Definitions:** For each key in the BST access problem in order, search for the key in the Wilber I bound tree. When we encounter an internal

> node during a search, mark the direction we leave the node. This is called the *preferred direction*. We *cross* an internal node, p, when we take a direction opposite from the preferred direction of the node. When we cross a node p there are two associated indices. The index of the key we are searching for when we take the direction opposite from the preferred direction is the *second crossing index*. The index of the key where we last last encountered p during the search is the *first crossing index* of the crossing.

The original Wilber I bound is the number of crossings for all internal nodes plus the number of accesses in the sequence. The cut bound described below does not include the number of accesses in the sequence. We call the number of crossings for all internal nodes the *partial Wilber I bound*.

3.3.3 Equivalent cut bound I (ECBI)

The ECBI cut set has cuts at the half integers corresponding to the internal nodes of the Wilber I bound tree. Let L be an ordering of the keys in the internal nodes of the bound tree so that the dual-Cartesian tree on the nodes is the Wilber I bound tree. The cuts in the ECBI cut set are ordered by L if we identify a cut with the key corresponding to its x coordinate. The rest of Section 3.3.3 proves that the cut bound provided by the ECBI cut set is equivalent to the partial Wilber I bound.

We show that the bounds are equivalent by showing that there is a one to one map between crossings of nodes in the bound tree and boxes in the crossing sets.

Throughout this section, when we refer to a cut set or cuts, the set is the ECBI cut set and the cuts are cuts from the set.

3.3.4 Maps

Let $S = (s_1, s_2, \ldots)$ be a sequence of integers. Each crossing of S over an internal node may be mapped to a pair of indices: the first and second crossing indices of the crossing. Call this map $M_{\text{XING}_1\to\text{PAIR}}$. It is injective by Theorem 3.18. Because $M_{\text{XING}_1\to\text{PAIR}}$ is injective, it may be inverted on its image.

Each point in $M_{\text{BST}\to\text{BSP}}(S)$ (page 38) has a unique y coordinate, so each unstabbed box in $M_{\text{BST}\to\text{BSP}}(S)$ may also be mapped to a pair of indices of S: the y values of its generating points. We will call this map $M_{\text{XBOX}_1\to\text{PAIR}}$. This map is injective because no two unstabbed boxes can share two generators.

This gives the following result:

Theorem 3.17. *Let S be a sequence of keys. There is a bijective map from crossings of S over nodes in the Wilber I bound tree and the union of crossing sets of the ECBI cut set over $M_{\text{BST}\to\text{BSP}}(S)$.*

Proof. Theorems 3.19 and 3.24 show that the image of crossings under $M_{\text{XING}_1\to\text{PAIR}}$ is the same as the image of the union of all crossing sets over $M_{\text{BST}\to\text{BSP}}(S)$ under $M_{\text{XBOX}_1\to\text{PAIR}}$. As the image of both maps is the same, we can invert one map and compose with the other: $M^{-1}_{\text{XING}_1\to\text{PAIR}}(M_{\text{XBOX}_1\to\text{PAIR}}(\cdot))$. This is a one-to-one map from crossings to boxes. □

3.3.5 An injective map

Theorem 3.18. *The map $M_{\text{XING}_1\to\text{PAIR}}$ is injective.*

Proof. Let $S = (s_1, s_2, \ldots)$ be a sequence of integers. Suppose two distinct crossings of S in the bound tree have first and second crossing indices t and u respectively. As the search for s_u changes the preferred direction set by the search for s_t, there is only one node where the crossing can take place: the last node that the root-to-node paths of s_t and s_u share in the bound tree. Any before the last node will not have their direction set by s_t and changed by s_u. A node can only have its direction switched once per access, so the pair of indices can only refer to one crossing. □

3.3.6 The image of crossings

Let $\mathrm{lca}_{\mathrm{WI}}(p, q)$ be the lowest common ancestor of nodes with keys p and q in the Wilber I bound tree.

Theorem 3.19. *Let $S = (s_1, s_2, \ldots)$. A pair of indices (t, u) is in the image of the crossings of S over the Wilber I bound tree under $M_{\mathrm{XING}_1 \to \mathrm{PAIR}}$ if and only if there is no index $r \in (t, u)$ such that s_r is in the subtree of $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$ in the bound tree.*

Proof. Suppose there is an index r such that s_r is in $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$'s subtree and $r \in (t, u)$. Without loss of generality s_t is left of $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$ and s_u is right of $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$. The key s_r is either left of right of $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$. If left, then t cannot be the the first index of a crossing with u the second index. If right, then u cannot be the second index for a crossing with t the first. In either case, the indexes t and u cannot be the first and second indexes for the same crossing.

Now suppose there is not an index $r \in (t, u)$ such that s_r is in the subtree of $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$. The definition of $\mathrm{lca}_{\mathrm{WI}}$ implies that searches for s_t and s_u must take different directions from $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$. As no other searches reach $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$'s subtree between the two indices, u must flip the direction that t set: t and u are the first and second indices of a crossing of $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$. □

3.3.7 The image of boxes

We now turn to showing that the images of $M_{\mathrm{XING}_1 \to \mathrm{PAIR}}$ and $M_{\mathrm{XBOX}_1 \to \mathrm{PAIR}}$ are the same. We begin by linking unstabbed boxes to the locations of their generators in the bound tree.

> **Definitions:** A cut at x coordinate j corresponds to the node with key value j in the Wilber I bound tree. When we refer a cut in the bound tree, we are referring to the node with the key equal to the x coordinate of the cut. Similarly, a point in the plane is *in a subtree* of the bound tree if the x coordinate of the point is a key in the subtree.

Lemma 3.20. *A cut c cuts all and only those unstabbed boxes with one generating point in the subtree of each child of c in the Wilber I bound tree.*

Proof. Let P be a point set. Any cut will split any unstabbed box if the cut is in the interior of the x range of the box, as cuts extend beyond the y range of all boxes in both directions. This also implies that no boxes are ever nicked.

Let $B(p, q)$ be an unsatisfied box in P. There are few cases to consider:

- The cut c is not in the interior of the x range of $B(p, q)$. This implies that p and q cannot be in subtrees of different children of c and that c cannot split $B(p, q)$.

- The cut c is in the interior of the x range of $B(p, q)$, but at least one of the nodes p is not in either of the subtrees of c's children in the bound tree. This implies that the lca of p and q in the bound tree is ordered earlier in the cut set than c and splits $B(p, q)$: c cannot cut $B(p, q)$.

- Again c is in the interior of the x range of $B(p, q)$, but p and q are in the subtrees of different children of c. Since the lca of p and q is c, there are no nodes ranked before c in the dual-Cartesian tree construction in the x range of $B(p, q)$. In other words, c is the earliest cut in the cut set that splits $B(p, q)$: c cuts $B(p, q)$.

□

Lemma 3.21. *Let $S = (s_1, s_2, \dots)$ be a BST access problem. If t and u are indices such that there is no $r \in (t, u)$ with s_r in the subtree of $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$, then (t, s_t) and (u, s_u) generate an unstabbed box in $M_{\mathrm{BST}\to\mathrm{BSP}}(S)$.*

Proof. The point set $M_{\mathrm{BST}\to\mathrm{BSP}}(S)$ is (i, s_i) for $i \in [1, n]$ by definition, so (t, s_t) and (u, s_u) are in $M_{\mathrm{BST}\to\mathrm{BSP}}(S)$. Suppose a point (r, s_r) stabs $B((t, s_t), (u, s_u))$. By definition of stabbing, s_r is between s_t and s_u (and hence is in $\mathrm{lca}_{\mathrm{WI}}(s_t, s_u)$'s subtree) and $r \in (t, u)$. Lemma 3.20 implies that no such r exists. □

Lemma 3.22. *Let $S = (s_1, s_2, \ldots)$ be a BST access problem. If t and u are indices such that there is no $r \in (t, u)$ with s_r in the subtree of $\text{lca}_{\text{WI}}(s_t, s_u)$ then the box $B((t, s_t), (u, s_u))$ is in the crossing set of $\text{lca}_{\text{WI}}(s_t, s_u)$ over $M_{\text{BST}\to\text{BSP}}(S)$.*

Proof. Box $B((t, s_t), (u, s_u))$ is unstabbed $M_{\text{BST}\to\text{BSP}}(S)$ by Lemma 3.21. Lemma 3.20 implies that $\text{lca}_{\text{WI}}(s_t, s_u)$ cuts $B((t, s_t), (u, s_u))$. Suppose $\text{lca}_{\text{WI}}(s_t, s_u)$ also cuts some another box $B((v, s_v), (w, s_w))$ with $(v, w) \subset (t, u)$. By Lemma 3.20, s_v and s_w are in c's subtree with both v and w in (t, u), and $\{s, t\} \neq \{v, w\}$. This contradicts the assumption of the lemma. □

Lemma 3.23. *Let $S = (s_1, s_2, \ldots)$ be a BST access problem. If t and u are indices such that there is an $r \in (t, u)$ with s_r in the subtree of $\text{lca}_{\text{WI}}(s_t, s_u)$, then the box $B((t, s_t), (u, s_u))$ is not in the crossing set of $\text{lca}_{\text{WI}}(s_t, s_u)$ over $M_{\text{BST}\to\text{BSP}}(S)$.*

Proof. As s_t and s_u are in the subtrees of different children of $\text{lca}_{\text{WI}}(s_t, s_u)$ and s_r is in the subtree of $\text{lca}_{\text{WI}}(s_t, s_u)$, either $\text{lca}_{\text{WI}}(s_t, s_r) = \text{lca}_{\text{WI}}(s_t, s_u)$ or $\text{lca}_{\text{WI}}(s_u, s_r) = \text{lca}_{\text{WI}}(s_t, s_u)$. This implies that there are two indices $v, w \in [t, u]$ with $[v, w]$ strictly contained in $[t, u]$ such that we can apply Lemma 3.22 to see that $\text{lca}_{\text{WI}}(s_t, s_u)$ cuts $B((v, s_v), (w, s_w))$. In other words, the y range of $B((t, s_t), (u, s_u))$ contains the y range of another box in $\text{lca}_{\text{WI}}(s_t, s_u)$'s cut set: $B((t, s_t), (u, s_u))$ cannot be in $\text{lca}_{\text{WI}}(s_t, s_u)$'s crossing set. □

Theorem 3.24. *Let $S = (s_1, s_2, \ldots)$. A pair of indices (t, u) is in the image of the crossing sets of $M_{\text{BST}\to\text{BSP}}(S)$ over the ECBI cut set if and only if there is no index $r \in (t, u)$ such that s_r is in the subtree of $\text{lca}_{\text{WI}}(s_t, s_u)$.*

Proof. This follows directly from lemmas 3.22 and 3.23. □

3.4 Wilber II

In this section, we show that the second bound described by Wilber [33] is a cut bound. The construction and proof are more complicated than those found in 3.3, though similar in approach: we give a bijection between the boxes in the union of crossing sets for some cut bound and Wilber II crossings.

3.4.1 Original bound: Wilber II

> **Definitions:** Let $S = (s_1, \ldots)$ be a BST problem. For each index in S, we construct the *past strictly verging sequence*. The past strictly verging sequence of i is a subsequence of $(1, \ldots, i-1)$. If $s_j < s_i$ ($s_j > s_i$), then j is in the past strictly verging sequence for i if and only if there is no $k \in (j, i)$ such that $s_k \in [s_j, s_i]$ ($s_k \in [s_i, s_j]$). The *past verging sequence* is defined similarly, but for $s_j < s_i$ ($s_j > s_i$), s_j is in the past verging sequence for i if and only if there is no $k \in (j, i)$ such that $s_k \in (s_j, s_i]$ ($s_k \in [s_i, s_j)$). A sequence of indices $U = (u_1, \ldots, u_{|U|})$, *crosses integer k at index u_j in S* if $s_{u_j} < k$ and $s_{u_{j+1}} > k$ or $s_{u_j} > k$ and $s_{u_{j+1}} < k$, and the indices u_j and u_{j+1} are the *first and second indices* for the crossing. Each crossing of the past strictly verging sequence of i over s_i in S is a *Wilber II crossing of i*. The *crossing number* for index i is the number of times the past strictly verging sequence of i crosses s_i in S.

The sum of crossing numbers for all indices of a sequence is the *partial Wilber II bound*. The full Wilber II bound is the partial Wilber II bound plus the length of the sequence.

3.4.2 Equivalent cut bound II (ECBII)

In this section, we describe the ECBII bound, a cut bound that is at least as large as the partial Wilber II bound.

> **Definitions:** Let $S = (s_1, \ldots)$ be a BST access problem. The ECBII cut set depends on S, unlike the Wilber I cut set.

- Each cut c in the ECBII cut set corresponds to a unique $(i, s_i) \in M_{\text{BST}\to\text{BSP}}(S)$. The *index* of a cut c corresponding to (i, s_i) is i. By definition of $M_{\text{BST}\to\text{BSP}}$, there is only one cut with a given index. We will refer to the cut with index k as c_k.
- The cut c_i is a vertical line segment extending down from (i, s_i) to the next point in $M_{\text{BST}\to\text{BSP}}(S)$ with x coordinate s_i, or to $y = 0$ if there is no such point.
- The cuts in the ECBII are ordered by their index.

3.4.3 Unproved results

Before we describe the maps, we present some interesting but not vital results without proof. These may be proved with fairly minor modifications of the proofs in the rest of the chapter.

The partial Wilber II and ECBII cut bounds are equal for sequences where each key is accessed only once. If we use the past verging sequence instead of the past strictly verging sequence to compute Wilber II crossings of indices, the partial Wilber II bound is equal to the ECBII cut bound on all sequences.

3.4.4 Maps

Let $S = (s_1, s_2, \ldots)$ be a sequence of integers. Each Wilber II crossing of an index of S may be mapped to a pair of indices: the first and second crossing indices of the crossing. Call this map $M_{\text{XING}_2\to\text{PAIR}}$. It is injective by Theorem 3.26.

Each point in $M_{\text{BST}\to\text{BSP}}(S)$ has a unique y coordinate, so each unstabbed box in $M_{\text{BST}\to\text{BSP}}(S)$ may also be mapped to a pair of indices of S: the y values of its generating points. We will call this map $M_{\text{XBOX}_2\to\text{PAIR}}$. $M_{\text{XBOX}_2\to\text{PAIR}}$ maps the lower index to the first element of the pair. This map is injective because no two unstabbed boxes can share two generators, so it may be inverted on its image, or any subset of its image.

This gives us the following result:

Theorem 3.25. *Let S be a sequence of keys. There is an injective map from Wilber II crossings of indices of S to the union of crossing sets of the ECBII cut set over $M_{\mathrm{BST}\to\mathrm{BSP}}(S)$.*

Proof. Lemma 3.30 shows that there is an injective map from the image of crossings under $M_{\mathrm{XING}_2\to\mathrm{PAIR}}$ to the image of crossing sets under $M_{\mathrm{XBOX}_2\to\mathrm{PAIR}}$. This completes the proof as $M_{\mathrm{XBOX}_2\to\mathrm{PAIR}}$ may be inverted on its image. □

3.4.5 An injective map

Theorem 3.26. *The map $M_{\mathrm{XING}_2\to\mathrm{PAIR}}$ is injective.*

Proof. Let $S = (s_1, \ldots)$ be a sequence of keys. Suppose t and u are the first and second crossing indices of Wilber II crossings for i and j. If $i = j$ the crossings are the same, so assume that $i \neq j$. Without loss of generality $i < j$ and $s_t < s_u$. By definition of crossing indices $u < i$ and $s_i \in (s_t, s_u)$. If $s_i < s_j$ then t cannot be the in the past strictly verging sequence of j as $i \in (t, j)$ and $s_i \in (s_t, s_j)$. Similarly if $s_i > s_j$ then u cannot be in the past strictly verging sequence of j as $i \in (u, i)$ and $s_i \in (s_j, s_u)$. This is a contradiction as both crossing indices of a Wilber II crossing for an index must be in the past strictly verging sequence of the index. □

3.4.6 The image of boxes

Lemma 3.27. *Let $S = (s_1, \ldots)$ be a sequence of keys. If j is in the past strictly verging sequence of i in S then $B((s_i, i), (s_j, j))$ is unstabbed in $M_{\mathrm{BST}\to\mathrm{BSP}}(S)$.*

Proof. Suppose there is an r such that (s_r, r) stabs $B((s_i, i), (s_j, j))$ in $M_{\mathrm{BST}\to\mathrm{BSP}}(S)$. By definition of stabbing, this implies that $r \in (j, i)$ and s_r is in the closed interval between s_i and s_j. No such r exists by definition of past strictly verging sequence. □

Lemma 3.28. *Let $S = (s_1, \dots)$ be a sequence of keys. If the box $B((s_i, i), (s_j, j))$ (with $s_i < s_j$) is unstabbed in $M_{\text{BST}\to\text{BSP}}(S)$, and (s_k, k) is the point with smallest y coordinate greater than $\max(i, j)$ with x coordinate in the range (s_i, s_j) then $B((s_i, i), (s_j, j))$ is in the cutting set of cut c_k in the ECBII cut set.*

Proof. The c_k splits $B((s_i, i), (s_j, j))$ as $s_k \in (s_i, s_j)$, and the cut extends to the next point with x coordinate s_k. This must be below $y = \min(i, j)$, as $B((s_i, i), (s_j, j))$ is unstabbed and (s_k, k) was chosen to be the lowest point above the box in the box's x coordinate range.

As (s_k, k) was the lowest point above $B((s_i, i), (s_j, j))$ in its x coordinate range, no cuts with index $\leq k$ can nick the box. This implies that $B((s_i, i), (s_j, j))$ is in the cutting set of c_k. □

Lemma 3.29. *Let $S = (s_1, \dots)$ be a sequence of keys. If t and u are the first and second crossing indices respectively of a Wilber II crossing of i in S, then there are indices t' and u' in the interval (t, u) such that $B((s_{t'}, t'), (s_u, u'))$*[1] *is in the crossing set of c_i over $M_{\text{BST}\to\text{BSP}}(S)$.*

Proof. Suppose there is an i such that t and u are the first and second crossing indices of a Wilber II crossing of i in S. By Lemma 3.27, the boxes $B((s_i, i), (s_t, t))$ and $B((s_i, i), (s_u, u))$ are both unstabbed. This give us the situation in Figure 3-6, or a mirror image of it if $s_t < s_u$.

If there is a point in $M_{\text{BST}\to\text{BSP}}(S)$ in the interior of the shaded rectangle in Figure 3-6, one such point will generate an unstabbed box with (s_i, i). Let (s_k, k) be such a point. By Lemma 3.27, k is in the past strictly verging sequence of i, and by the location of (s_k, k), $k \in (t, u)$. Since t and u are the first and second indices of a Wilber II crossing of i, there can be no $k \in (t, u)$ in the past strictly verging sequence of i: this is a contradiction.

As images of all sequences under $M_{\text{BST}\to\text{BSP}}$ have unique y coordinates for each point, the shaded box in 3-6 can only have points on the left edge above the bottom

[1] The s_u in the box specification is not a typographic error.

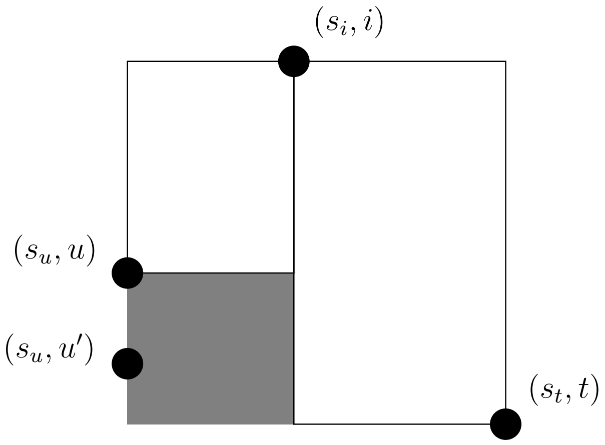

Figure 3-6: A crossing of i in $M_{\text{BST}\to\text{BSP}}(S)$.

corner. Let (s_u, u') be the lowest such point. Since there are no lower points in the grey box and $B((s_i, i), (s_t, t))$ is unstabbed, $B((s_u, u'), (s_t, t))$ is also unstabbed.

Cuts in the ECBII cut set are ordered by their top endpoint. The point (s_i, i) is the point in $M_{\text{BST}\to\text{BSP}}(S)$ with lowest y coordinate above u' and x coordinate in (s_u, s_t). This and Lemma 3.28 imply $B((s_u, u'), (s_t, t))$ is in c_k's cutting set. Now we show that it is in c_k's crossing set.

Suppose there is another unstabbed box $B((s_v, v), (s_w, w))$ (with $v < w$) in the cutting set of c_k such that the interval (t, u) strictly contains the interval (v, w). Such a box must satisfy certain properties:

- We must have $\min(s_v, s_w) = s_u$. If $\min(s_v, s_w) < s_u$, cut $c_{u'}$ will split box $B((s_v, v), (s_w, w))$ as s_u is then in the interior of the open interval between s_v and s_w, with y extent that contains (t, u). If $\min(s_v, s_w) > s_u$ c_i cannot cut $B((s_v, v), (s_w, w))$. Since $s_u = \min(s_v, s_w)$, $(v, w) \subset (t, u)$, and $B((s_u, u'), (s_t, t))$ is unstabbed, (s_u, u') must be the upper generator of $B((s_v, v), (s_w, w))$. In other words, $u' = w$. Let $t' = v$ to get the situation depicted in Figure 3-7.

- Rectangle A in Figure 3-7 must have no points in the x range $[s_t, s_{t'})$. If there is a point in A in the interior of A's x range, the cut from at least one such point will split $B((s_{t'}, t'), (s_u, u'))$, contradicting our assumption that the box is

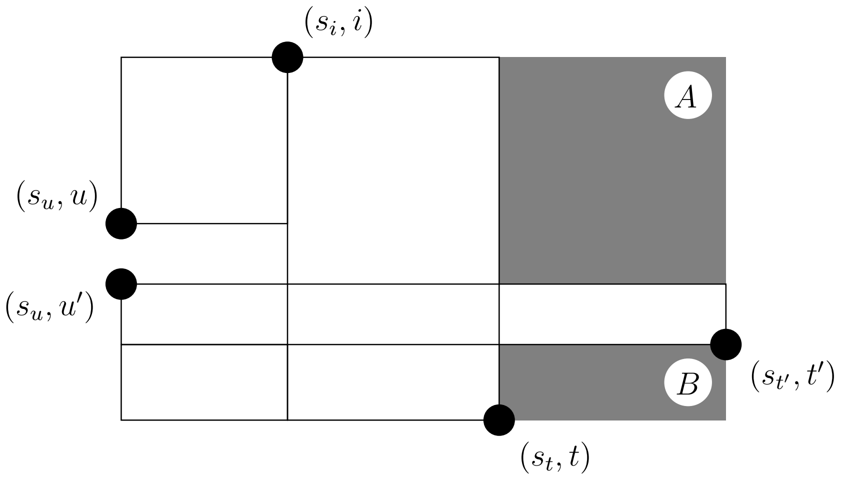

Figure 3-7: A more detailed crossing of i in $M_{\mathrm{BST}\to\mathrm{BSP}}(S)$.

in c_i's cutting set.

- Rectangle B in Figure 3-7 can have no points in its interior, or top or left edges. Suppose it does. One such point must generate an unstabbed box with (s_i, i) (either the highest or farthest left). Let (s_k, k) be such a point. By Lemma 3.27, k is in the past strictly verging sequence of i. As k must between t and u, t and u cannot be adjacent in the past strictly verging sequence of i, a contradiction.

There are now two possibilities: there is a point on the right edge of A above $y = u$ or there is not.

- Suppose there is a point on the right edge of A above $y = u$. Let z be the y coordinate of such a point. This gives us the situation in Figure 3-8. There cannot be yet another box in the c_i's cutting set over $M_{\mathrm{BST}\to\mathrm{BSP}}(S)$ with y range in (t', u'), as either $c_{u'}$ or $c_{z'}$ for $z' \in (t', z)$ would split the box. This implies that $B((s_{t'}, t'), (s_u, u'))$ is in c_i's crossing set.

- Now suppose there is not a point on the right edge of A above $y = u$. If $t' \neq t$, either $(s_{t'}, t')$ or some point above it, but below $y = u$ will form an unstabbed box with (s_i, i). This implies that there is an index in the past strictly verging

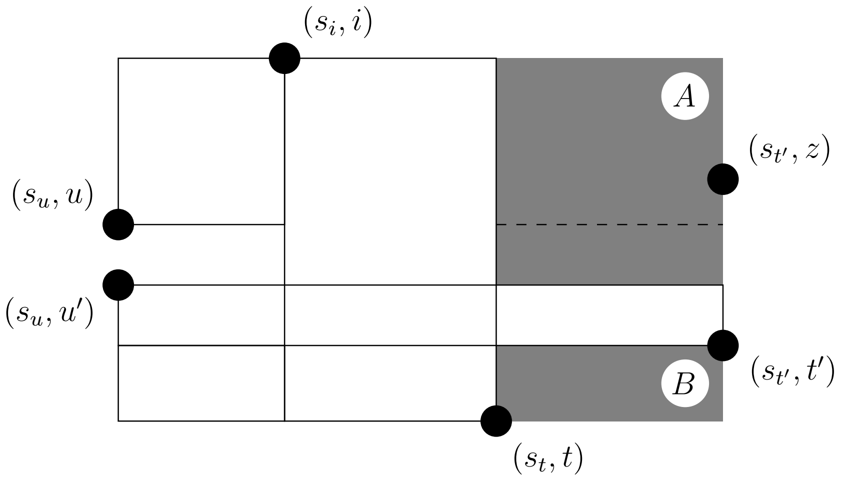

Figure 3-8: More details of a crossing of i in $M_{\text{BST}\to\text{BSP}}(S)$.

sequence of i over S between t and u, a contradiction. We must have $t' = t$, so $B((s_u, u'), (s_t, t))$ is in the crossing set of c_i. This completes the proof.

□

Lemma 3.30. *Let $S = (s_1, \ldots)$ be a sequence of keys. There is an injective map from the image of the Wilber II crossings of S under $M_{\text{XING}_2\to\text{PAIR}}$ to the image of the union of crossing sets of the ECBII cut bound over $M_{\text{BST}\to\text{BSP}}(S)$.*

Proof. This follows simply from Lemma 3.29. Consider a Wilber II crossing of index i with first and second crossing indices t and u. The image of this crossing is (t, u) under $M_{\text{XING}_2\to\text{PAIR}}$. Lemma 3.29 implies that there is a pair (t', u') such that the interval $[t', u'] \subseteq [t, u]$ and the pair (t', u') is in the image of the crossing set of c_i over $M_{\text{BST}\to\text{BSP}}(S)$ under $M_{\text{XBOX}_2\to\text{PAIR}}$. Let M map the pair (t, u) to some (t', u') satisfying this property. This gives us two properties:

- The pair (t', u') cannot be in the image of the crossing set of another cut as crossing sets are disjoint.
- Moreover, $[t', u']$ cannot be between the crossing indices for another Wilber II crossing of i as the indices in a past strictly verging sequence are in order, and crossing indices are adjacent in the past strictly verging sequence.

These imply no pair of crossing indices for another Wilber II crossing of S can map to (t', u') under M: M is injective. As M takes the image of Wilber II crossings of S under $M_{\mathrm{XING}_2 \to \mathrm{PAIR}}$ to pairs in the image of $M_{\mathrm{XBOX}_2 \to \mathrm{PAIR}}$, we are done □

3.5 NISIAN and a Nearly Optimal Independent Set

In this section, we consider the limits of the cut or independent set lower bounds. We describe NISIAN, an algorithm that computes an upper bound on the size of an independent set lower bound, while simultaneously computing a lower bound of that nearly that size. NISIAN does not get the upper and lower bounds to match but it gets them within a constant factor of each other. We begin by describing the property that lets us construct an upper bound on our lower bounds, then describe NISIAN and some of its properties.

3.5.1 NISIAN and Manhattan satisfaction

The cut and independent set lower bound provided lower bounds on the number of points that must be added to a point set to have monotone Manhattan paths between the points in the initial set. Although we cannot efficiently compute a fully satisfied set of the appropriate size, we can efficiently compute a set that has monotone Manhattan paths between the points of the initial set. We give this property a name:

> **Definitions:** Let Q be a set of points and P a distinguished subset of Q. The set Q is *Manhattan satisfied over P* if there is a monotone Manhattan path in Q between each pair of points in P. Q is $+$ *Manhattan satisfied over P* ($-$ *Manhattan satisfied over P*) if there is a monotone Manhattan path in Q between each pair of points in P that are generators of a $+$ type ($-$ type) box unsatisfied in P.

Manhattan satisfied sets are interesting because they provide upper bounds on the size of independent and cut bounds. The following theorem makes this rigorous.

Theorem 3.31. *If Q is a Manhattan satisfied set over P, no cut or independent set bound is larger than* $4(|Q| - |P|)$.

Proof. The definition of an adjacency graph and critical object, and the labelling procedure in Section 3.1.2 relied only on the existence of monotone Manhattan paths between every pair of points in P. This implies that any cut bounds is at most 4 times the number of points we need to add to P to get a Manhattan satisfied set over P. Theorem 3.16 extends the result to independent set bounds. □

3.5.2 NIAN, PIAN, and NISIAN points

In this section, we describe algorithms for computing different types of Manhattan satisfied supersets. We assume that point sets are on an integer lattice. In this section, we deal with three types of points, described below.

> **Definitions:** Let P be a set of points in the plane, with y coordinates in the set 1 through m. To compute the set of *positive Ian* (PIAN) points for P, begin with P in the set. For each y coordinate i on P's lattice in standard order, add to the positive Ian points the minimum set of points at $y = i$ such that there are no $+$ type unstabbed boxes in the current PIAN set with generating points at or below $y = i$. The *negative Ian points* (NIAN) for P are computed similarly, but we consider $-$ type unstabbed boxes rather than $+$ type unsatisfied boxes. The union of the NIAN and PIAN points are the *non-interacting signed IAN* (NISIAN) points for P[2].

The following theorem shows that PIAN, NIAN, and NISIAN sets are well defined and unique.

[2] The names come from running the IAN algorithm but ignoring some unsatisfied boxes of particular types. If we pay attention to both boxes, but ignore the points added to the set of the other sign, the signs could be said to be *non-interacting*.

Theorem 3.32. *Let P be a point set. The PIAN (NIAN) points of P are well defined: when we reach $y = i$ during construction, there is a unique minimum set of points we can add to eliminate all $+$ type ($-$ type) unstabbed boxes with generating points at or below $y = i$.*

Proof. The proof from Theorem 2.13 on page 45 in Section 2.3 may be used if we substitute "(+)-type unstabbed box" for "unstabbed box."

□

We also get a similar corollary:

Corollary 3.33. *There are no $+$ type ($-$ type) unsatisfied boxes in a PIAN (NIAN) set.*

Proof. In the (modified) proof of Theorem 2.13, we see that we can add a unique minimum set at $y = i$ to eliminate all $+$ type unsatisfied boxes between points with $y \leq i$. This includes the largest y coordinate for any points in P so the corollary follows. □

3.5.3 NISIAN and Manhattan satisfaction

In this section, we show that the set of NISIAN points for any point set, P, is Manhattan satisfied over P. We use an argument similar to that for Lemma 2.6, but with a limited point set:

Lemma 3.34. *Let P be a point set. The PIAN (NIAN) set of P is $+$ Manhattan satisfied ($-$ Manhattan satisfied) over P.*

Proof. For the duration of this proof, we extend the definition of axis boxes to include degenerate axis boxes: the two generating points of an axis box may share either their x or y coordinates. However, degenerate boxes need not be stabbed by a third point to be satisfied. Degenerate axis boxes are considered $+$ type for this proof.

We now show that there is a monotone Manhattan path between every pair of points in the PIAN set that are the generating corners of a $+$ type axis box. We induct on the size of boxes. Suppose all + type axis boxes in the PIAN set smaller in either x or y extent than $B(p, q)$ have a monotone Manhattan path between their generating points. Corollary 3.33 implies that $B(p, q)$ is stabbed by a third point r. The induction hypothesis guarantees there is a monotone Manhattan path between p and r and r and q as $B(p, r)$ and $B(r, q)$ are both + type axis boxes. Concatenating the two paths yields a monotone Manhattan path between p and q.

Now suppose there are no boxes smaller in x or y extent than $B(p, q)$ in Q. If $B(p, q)$ is non-degenerate, Corollary 3.33 implies that it is stabbed by a third point r and both $B(p, r)$ and $B(r, q)$ are + type and smaller in x or y extent than $B(p, q)$. This implies that $B(p, q)$ is degenerate. The path (p, q) is a monotone Manhattan path between the generating points of $B(p, q)$, so the lemma holds.

This argument works for the NIAN set if we reflect about the y axis. □

Theorem 3.35. *The NISIAN set of P is Manhattan satisfied.*

Proof. If $B(p, q)$ is an axis box between points in P, Lemma 3.34 implies that either the PIAN or NIAN set will have a monotone Manhattan path between p and q. As the NISIAN set is the union of these sets, the theorem follows. □

3.5.4 NISIAN and an independent set bound

In this section, we prove the following theorem:

Theorem 3.36. *Let P be a point set with NISIAN point set* $\mathrm{NISIAN}(P)$*. There is a map from an independent set in the interaction graph of P onto the points of* $\mathrm{NISIAN}(P)\backslash\{P\}$ *with at most 2 boxes mapping to any point.*

> **Definitions:** The upper left (right) corner of a + type ($-$ type) axis box is its *upper non-generating corner.*

Lemma 3.37. *Let P be a point set. If, while constructing the PIAN set of P, a point p is the upper non-generating corner of an axis box unsatisfied in the current PIAN set* *of P at any time during the construction, then p is the upper non-generating corner of an axis box with generators in P and unsatisfied in P.*

Proof. Without loss of generality P is on the integer lattice and its x and y coordinates are contiguous integers beginning with 1. Let m be the largest y coordinate of any point in P. Let $i_{ng}(p)$ be the smallest y value such that after we add points to PIAN at $y = i_{ng}(p)$, p is an upper non-generating corner of a $+$ type unsatisfied box during the construction of the PIAN set of P. If p is initially the upper non-generating corner of a $+$ type unsatisfied box, let $i_{ng}(p)$ be 0, and if p is never the upper non-generating corner of a $+$ type unsatisfied box let $i_{ng}(p)$ be $m+1$. We now show that $i_{ng}(p)$ is either 0 or $m+1$.

Suppose $i_{ng}(p) > 0$, and that p is the upper non-generating point of unstabbed $+$ type axis box $B(q,r)$ (with $y(q) < y(r)$) immediately after we add the points at $y = i_{ng}(p)$. The point p is not an upper non-generating corner of any unsatisfied box before $i_{ng}(p)$. The box $B(q,r)$ must have the following properties:

- Either $y(q)$ or $y(r)$ is equal to $i_{ng}(p)$. This follows as $B(q,r)$ cannot exist before $i_{ng}(p)$. If it did, p would be the upper non-generating corner of a $+$ type box after we processed some y coordinate less than $i_{ng}(p)$, a contradiction in the definition of $i_{ng}(p)$.

- We cannot have $y(r) \leq i_{ng}(p)$. As we do not again add points at or before $y = i_{ng}(p)$, the box $B(q,r)$ would not be satisfied in the final PIAN set, a contradiction of Corollary 3.33.

- Together the above properties imply $y(q) = i_{ng}(p)$.

As q was added to the PIAN set at time $i_{ng}(p)$, it was the upper non-generating corner of a $+$ type axis box $B(s,t)$ (with $y(s) < y(t)$). The box $B(q,r)$ is unstabbed immediately after we process $y = i_{ng}(p)$, and $B(s,t)$ is unstabbed immediately before. This gives us the situation depicted in Figure 3-9.

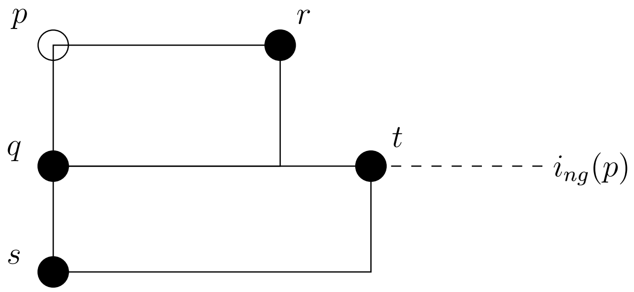

Figure 3-9: Creating a new upper empty corner.

The box $B(s, r)$ is in the union of $B(s, t)$ and $B(q, r)$, so it was unstabbed just before we processed $y = i_{ng}(p)$. Point p is the upper left corner of $B(s, r)$ and $B(s, r)$ is a $+$ type axis box: $i_{ng}(p)$ is not the smallest y coordinate after which p was the upper non-generating point of an unsatisfied $+$ box. This is a contradiction.

This implies that if p is ever the upper non-generating corner of a $+$ type unstabbed box, it was before we started processing y coordinates to construct the PIAN set. In other words, $i_{ng}(p)$ is either 0 or $m + 1$.

If we reflect about the y axis, the argument works for the NIAN set. □

Lemma 3.38. *Let P be a point set and let P_i^+ (P_i^-) be the result of running the PIAN (NIAN) construction for P up to and including the y coordinate i. The PIAN (NIAN) sets of P_i^+ (P_i^-) and P are the same.*

Proof. Corollary 3.33 applied to the set of points in P with $y \leq i$ implies that running PIAN on P_i^+ will add no points with $y \leq i$. After $y = i$, the PIAN construction on P and P_i^+ is identical, as the sets coincide just after processing $y = i$. □

Lemma 3.39. *Let P be a point set. If a point p is not initially the upper non-generating corner of a $+$ type ($-$ type) box of P before we begin construction, p will never become the non-generating corner of an axis box in the PIAN (NIAN) set, and will never be added to the PIAN set.*

Proof. This follows from Lemma 3.37. □

Lemma 3.40. *Let P be a point set. A point may be the upper non-generating corner of at most one $+$ type unsatisfied box in P.*

Proof. If p is the upper non-generating corner of a $+$ type unsatisfied box in P, the generating points of the box are the first points in P directly to the right and directly below p. A box is uniquely specified by its generating points. □

Lemma 3.41. *Let P be a point set. If a point p is the upper non-generating corner of an unstabbed $+$ type ($-$ type) axis box b, then if we ever add a point in the interior of b during a PIAN (NIAN) construction, we will never add p during the construction.*

Proof. Let P_i be the set of points in the PIAN set after processing $y = i$. Let $b = B(r, q)$ with $x(r) < x(q)$. Before we process $y(q)$ in the PIAN construction, q will be the first point in P right of p. Thus if p is the upper non-generating corner of an unstabbed $+$ type box before we process $y(q)$, q must be a generator of the box.

Let s be one of the first points added in the interior of $B(r, q)$ during a PIAN construction. (We might add many simultaneously if they all have the same y coordinate.) By assumption, $x(r) < x(s) < x(q)$. The method of PIAN construction implies that if p is an upper non-generating corner of a $+$ type box in $P_{y(s)}$, the lower generating corner of the box must be at or below $y = y(s)$. The point s stabs any such box, so p cannot be the upper non-generating corner of any $+$ type axis box in $P_{y(s)}$. Lemmas 3.38 and 3.39 now imply p will not be added to PIAN.

Reflecting about the y axis gives us the same result for the NIAN set. □

Lemma 3.42. *Let P be a point set and let b and b' be two (unsatisfied) interacting boxes in P. If the upper non-generating corner of b is in the PIAN (NIAN) set of P, then the PIAN (NIAN) set will not contain the upper non-generating corner of b'.*

Proof. If b and b' interact, either the upper non-generating corner of b' is in the interior of b or visa versa:

- If the upper non-generating corner of b is in b', Lemma 3.41 tells us the upper non-generating corner of b' will not be added to the PIAN set.

- If the upper non-generating corner of b' is in b, Lemma 3.41 tells us the upper non-generating corner of b will not be added to the PIAN set.

In both cases, the upper non-generating corner of only one of a pair of interacting boxes may be added to the PIAN set. Reflecting about the y axis gives us the argument for the NIAN set. □

Theorem 3.36 now follows:

Proof. Let B_+ be the set of $+$ type unsatisfied boxes in P whose upper non-generating corners are in the PIAN set of P, and define B_- similarly but with the NIAN set. Lemmas 3.42 and 3.3 imply $B_+ \cup B_-$ cannot contain two boxes that interact with each other. In other words, $B_+ \cup B_-$ is an independent set in the interaction graph of P. If we map each $b \in B_+ \cup B_-$ to its upper non-generating corner, at most one box from B_+ and one from B_- will map to any one point in $\text{NISIAN}(P)\backslash P$. Every point in $\text{NISIAN}(P)\backslash P$ will be the upper non-generating corner of some unsatisfied box in P by Lemma 3.37, so the map is onto. □

Theorem 3.36 has a significant implication:

Corollary 3.43. *If r is the ratio of the largest independent set lower bound on P to $|\text{NISIAN}(P)\backslash P|$, then $r \in [1, 2]$.*

Proof. Theorem 3.36 implies that some independent set, and hence the largest such set on the interaction graph of P is at least as large as $|\text{NISIAN}(P)\backslash P|$. (The map in the theorem is onto.) Theorem 3.31 implies that $4|\text{NISIAN}(P)\backslash P|$ is greater than the largest independent set as the NISIAN set of P is Manhattan satisfied by 3.35. □

In other words, $|\text{NISIAN}(P)\backslash P|$ gives us the best independent set bound within a factor of 4.

3.6 Open Questions and Speculation

In this chapter, we outlined a new class of bounds that unifies both Wilber bounds. Among this class, we described a way to compute nearly the best bound among this class for a given BST access problem. There is some room for progress in this direction. The major open question related to these bounds is: Is this class of lower bounds tight? Despite much effort to use these bounds to construct geometric solutions to the box stabbing problem problems (see Chapter 5 for a small example), we have not made any further progress in this direction.

A related, but interesting question is whether the second bound of Wilber is within a constant factor of the NISIAN bound. We showed that NISIAN is at least as large as Wilber II (within a constant factor), but we have not yet been able to show that NISIAN is strictly larger.

We also cannot show that NISIAN is a tight lower bound for a box stabbing problem cost. It is reasonable to expect NISIAN is not tight. The nodes we add to get a Manhattan satisfied set over the initial point set probably interact, forcing us to add more points to satisfy the created boxes. The additional points may again interact with each other. Some way to bound the interactions either above or below would be interesting. If we can bound above, then we might be able to show that NISIAN is tight. If we can bound from below, we might be able to show that it is not.

Chapter 4

Tango Trees: a New Upper Bound for Online BST-access

This chapter describes tango trees, the first online BST algorithm that is provably better than $O(\log n)$-competitive. The bound and algorithm as described here are slightly different than that originally published [10], resulting in some changes in the constants in certain bounds. These changes were made so we could use the machinery of the box stabbing problem bounds from Chapter 3. The description of the machinery for the algorithm is very similar to [10].

The goal in this chapter is to describe an online BST algorithm with the restrictions that it can only store constant extra space in each node, and can only store information in the BST between serving each access. As such, we are concerned with some details of how the tango algorithm uses rotations on the BST tree during a search and rearrangement. To do this, we use a well-known online BST data structure, red-black trees. (See [9], Chapter 14.)

We begin the chapter by establishing some notation and giving an overview of the basic structures related to tango and their relationship to each other in Section 4.1. Many properties of the structures and algorithms are claimed, but not proved in the overview. We will prove these properties later in the chapter.

The major result, Theorem 4.24, is that tango is $O(\log \log n)$-competitive compared to an optimal offline BST algorithm. This is found in the final Section 4.5, on page 132.

4.1 Overview

As in earlier chapters, we assume a fixed universe of keys. We assume that there are n keys in the universe and m accesses in a sequence. Moreover, we generally assume that each key is accessed at least once, so $m \geq n$. We discuss two BST trees that store the entire universe of keys in this algorithm: the *bound tree* and the *structure tree*. The tango algorithm uses and maintains the structure tree. However, we use another BST to analyze the behavior of the tango algorithm. To this end, we make use of an algorithm called the super tango algorithm, described in Section 4.1.1.

4.1.1 Super tango

The super tango algorithm maintains both bound and structure trees. It uses the tango to maintain the structure tree, and maintains the bound tree using information from the tango algorithm. However, *tango can be run as a stand-alone algorithm without reference to super tango.*

4.1.2 Nodes and tree membership

Throughout this chapter, we need to refer to both the bound and structure trees. Through a slight abuse of notation, we speak of nodes as if they are present in both the bound and structure trees. This is not strictly accurate as the nodes in one tree are not in the other. However, when we refer to a node p in the structure tree as *in the bound tree*, we are actually referring to the node in the bound tree with the same key value as p. The same convention holds for sets of nodes and even subtrees. For subtrees, we do not assume that a subtree of the bound tree is necessarily a subtree in the structure tree, though we prove this for certain types of subtrees.

4.1.3 Bound tree

As noted above, the tango algorithm does not deal specifically with the bound tree. The bound tree is a balanced BST on the set of nodes with fixed structure. We store some additional information in each node. Each node has a *preferred child*: either left, right, or neither. An edge in the bound tree between a node and its preferred child (if any) is a *preferred edge*. Other edges are *non-preferred edges*. A group of nodes connected by preferred edges is a *preferred subtree* of the bound tree. Preferred and non-preferred edges are also defined (in Section 4.1.4) for the structure tree. The non-preferred edges in the bound tree may be identified with non-preferred edges in the structure tree in a logical manner. The correspondence is discussed in Section 4.1.4.

The super tango algorithm changes preferred children while processing an access sequence. While the tango portion of super tango searches for a node in the structure tree, super tango searches for the same node in the bound tree. When super tango traverses a non-preferred edge in the bound tree during a search, it changes the non-preferred edge to a preferred edge in two steps. It first changes the preferred child of the parent of the non-preferred edge to neither, then to the direction required to make the traversed edge a preferred edge. The timing of these changes is closely synchronized with similar behavior in the tango algorithm, and is discussed in detail in Section 4.3.

The number of non-preferred edges traversed while searching for a node is the *bound cost* for the search. We show in Section 4.4 that the number of non-preferred edges traversed in the bound tree during all searches is a lower bound for the performance of an optimal offline BST algorithm.

4.1.4 Structure tree

The structure tree is a BST. Tango maintains two invariants related to the structure tree:

- This is the *first structure tree invariant*. Each preferred subtree of the bound tree is also a subtree in the structure tree, though the structures of the subtrees are usually different. This lets us extend the notion of preferred subtrees and edges to the structure tree. Preferred subtrees from the bound tree when found in the in the structure tree are called preferred subtrees. Edges between nodes of the same preferred subtree of the structure tree are *preferred edges*, and edges between nodes from different preferred subtrees in the structure tree are *non-preferred edges*.

- This invariant is the *second structure tree invariant*. If we shrink all preferred edges in the bound and structure trees, and identify each shrunk node with the set of keys in the preferred tree that the node represents, both bound and structure trees are the same. The tree with each preferred subtree shrunk to a single node is the *non-preferred tree* (of the bound or structure tree). The non-preferred tree is a tree with a set of keys associated with each node. Each non-preferred edge may be identified by the preferred subtrees it connects. A non-preferred edge in the bound tree *corresponds* to a non-preferred edge in the structure tree if the edges connect nodes with the same sets of keys in the non-preferred trees of the bound and structure trees.

These invariants are maintained without reference to the current state of the bound tree, though they do require knowledge about the behavior of super tango. Tango does, however, use the depth of each node in the bound tree. This depth is the *bound depth* and can be computed once before any access searches, at a cost of $O(n)$. As the length of the sequence is assumed to be at least $O(n)$, this is acceptable.

The number of non-preferred edges on a node-to-root path for a node, p, in either the bound or structure tree is p's *non-preferred depth*. By the second structure tree invariant, this depth is the same for a node in the bound and structure trees. This equivalence lets us compare the performance of the tango algorithm to the bound provided from the bound tree.

Tango uses a BST data structure called an *auxiliary tree*, described below. Each preferred subtree of the structure tree is maintained as an auxiliary tree. The tango algorithm must be able to determine if edges between nodes in the structure tree are edges between nodes within a preferred tree or between different preferred subtrees. To do this, it places a special mark on the lower node of each preferred edge. This is an acceptable method of separating the auxiliary trees, as the following (obvious) theorem shows.

Theorem 4.1. *An edge is non-preferred if and only if the lower node of the edge is the root of a preferred subtree.*

Proof. Let r be the root of a preferred subtree. By definition of root, r's parent in the structure tree cannot be in r's preferred subtree: the edge between r and its parent is non-preferred.

Now suppose that r is not the root of its preferred subtree. If the root from r to it's parent s in the structure tree is non-preferred, r cannot have a parent in its preferred subtree as s is the only candidate. □

4.1.5 Auxiliary trees

Auxiliary trees may be implemented in many ways, but they must support the following operations, all in cost logarithmic in the number of nodes in the auxiliary tree:

- Search for a key value, possibly located between two nodes in the auxiliary tree.

- Split the auxiliary tree into nodes that are below a certain depth in the bound tree, and those above.

- Join two auxiliary trees if the union of their nodes forms a contiguous section of a node-to-root path in the bound tree.

The following theorem motivates the last property.

Theorem 4.2. *Every preferred subtree in the bound tree is a contiguous section of a node-to-root path (in the bound tree).*

Proof. Each node in the bound tree can have at most one preferred child. If a preferred subtree is not a contiguous section of a node-to-root path, it must have at least one node that has two children in the subtree. Such a node would have two preferred children, a contradiction. □

4.1.6 Top level subtrees

In the previous sections, we discussed preferred subtrees in both the bound and structure trees. These subtrees partition the two trees. The *top level* preferred subtree is preferred subtree that contains the root node of the bound or structure tree in which it is located.

Lemma 4.3. *The top level preferred subtree in the bound tree is also the top level preferred subtree in structure tree.*

Proof. Let T be the top level preferred subtree of the bound tree. If we shrink all preferred edges, the node corresponding to the subtree with keys in T will be the root in both. □

4.2 Auxiliary Tree Structure and Behavior

In this section, we show that the necessary operations for auxiliary trees described in Section 4.1.5 can be achieved using slightly modified red-black trees (red-black trees). Modified splay trees may also be used to achieve similar results in expected performance [12]. In this section, we assume that the size of the largest subtree involved in any operation is k. Since an auxiliary tree contains nodes corresponding to a node-to-root path in a balanced BST with n nodes, k is $O(\log n)$ for an access sequence with n distinct keys.

For the split and join operations for auxiliary trees, we assume that the structure tree invariants hold before the operation. Theorem 4.4 shows that the rotations during the operations will not invalidate the invariants, so we can assume that the invariants hold up until we perform the root markings for splitting and joining. We will show later in Section 4.3 that the invariants hold through the markings during the split and join operations.

Theorem 4.4. *If the structure tree invariants hold, then rotating an edge in the structure tree between two nodes in the same preferred subtree will maintain the invariants.*

Proof. The two nodes are connected to each other before and after the rotation, and the tree we acquire from shrinking the edge between them is the same before and after the rotation. This is sufficient to imply both invariants hold after if they both hold before. □

4.2.1 Basic operations with red-black trees

The operations on the red-black trees we use are the following:

- *Split* the red-black tree at a node p. This brings the node p to the root of the red-black tree, with red-black trees as its left and right subtrees.

- *Join* two red-black trees whose roots are the children of a node, p. In other words, if p is between two red-black-trees in key-space, we join p and the nodes of the two red-black trees into a single red-black-tree.

- *Search* for a node with a particular key value. The key value may or may not be found in the red-black tree. We must also find the successor and predecessor nodes for a key value. The predecessor of a key value v not in the red-black tree is the largest key value in the tree less than v. The successor of a key value is defined similarly.

The second of these operations is outlined in the reference [9] in exercise 14-2, and the first is a reverse of the second. Both may be implemented in time and cost logarithmic in the size of the trees involved in the operations. The third operation is logarithmic in the size of the tree as red-black trees are balanced.

4.2.2 Searching

Searching for a node in the auxiliary tree is simple: perform the normal BST search for a given key until we encounter a non-preferred edge or find the node with the given key value. This takes $O(\log k)$ nodes as red-black trees are balanced.

4.2.3 Splitting at a depth

In order to perform a split, each node of our red-black trees maintains both its bound depth the maximum bound depth any node in the same auxiliary tree (within the same preferred subtree). We can maintain this information with at most a constant-factor increase in time. (See [9], Chapter 15.) As noted in the beginning of this section, we assume that both structure tree invariants hold before we begin the split.

Before we begin the description, we first make note of a property of key intervals for preferred subtrees:

Theorem 4.5. *If the first structure tree invariant holds, then the keys in the nodes of a preferred subtree of the bound tree with bound depth greater than a given bound depth d are in a sub-interval of the interval of key space spanned by nodes in the preferred subtree, and this sub-interval contains no nodes whose bound depth is less than d.*

Proof. This follows from Theorem 4.2. Cutting a contiguous section of a node-to-root path at a given depth in any BST produces two halves. The key span of the half with greater bound depth is a sub-interval of the key span of the entire section and contains no keys from the half with lesser depth. As this is a statement about key values, the same property will hold whether the nodes are in the bound or structure tree. □

The maximum bound depth in a subtree lets us find the node with smallest key value that has bound depth greater than d: we can repeatedly descend to the leftmost subtree with bound depth greater than d. Let l be the node with smallest key value that has bound depth greater than d. Similarly we can find the node with largest key value with depth greater than d. Call this node r. The predecessor of l and the successor of r are l' and r'. Call the initial auxiliary tree T. Figure 4-1 (in the clockwise direction) shows how we split the auxiliary tree into the desired pieces.

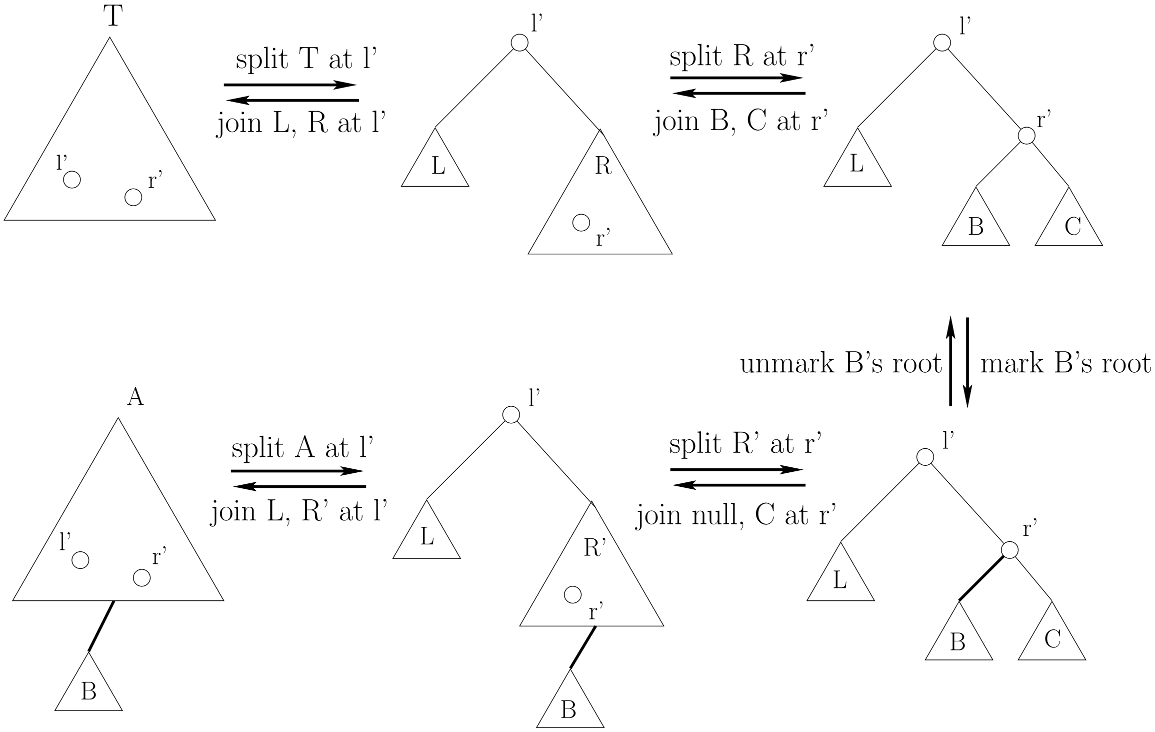

Figure 4-1: Splitting and joining auxiliary trees.

1. First we split at l', dividing the tree into two parts: L and R.

2. We then split R at r', into trees B and C. Theorem 4.5 implies that all nodes with bound depth greater than d are in B.

3. In order to split B from the current auxiliary tree, but maintain the BST structure of the structure tree, we change the edge between r' and B to a non-preferred edge. If we are running super tango, it makes appropriate changes in the bound tree to reflect this change. This cuts B from r'.

4. We can now join the "empty" left child of r' and C at r' to make R', and then join L and R' at l' to make A.

This procedure may be modified slightly (but obviously) if some of the trees A, B, or C are empty.

Theorem 4.6. *Let T be an auxiliary (and also preferred) subtree of the structure tree. If we perform a split of T at a bound depth d, then following the split there will be no nodes of T with bound depth $> d$ on the node-to-root path of any node of T with depth $\leq d$ in the structure tree.*

Proof. Assume that the same notation as that of the splitting procedure found above. The theorem states that after splitting, no node of B will be on the node-to-root path of any node of T not in B. After splitting at r', this holds. After we mark the edge connecting B to r' as non-preferred the desired condition cannot change as we never perform any rotations of an edge touching a node of B. □

4.2.4 Joining

Joining is the reverse of splitting. This is depicted in Figure 4-1, moving in the counter-clockwise direction. We want to join two auxiliary trees such that the union of the nodes of the two auxiliary trees is a sub-path of a node-to-root path in the bound tree. We assume that both structure tree invariants hold before we begin this operation, so that the set of nodes of each of the auxiliary trees is a sub-path of a node-to-root path in the bound tree as are the corresponding nodes of each path separately. Label the auxiliary tree with nodes with smaller bound depth A, and the one with greater bound depth B. This assumption and the second structure tree invariant imply no node of B can be on the node-to-root path of A, so B hangs off of a node of A through a non-preferred edge. The node l' is the node with largest key value in A less than any key value in B, and r' is the node with smallest key value in A greater than any key value in B. These may be found in time $O(\log k)$ by searching for the root of B (or any node in B or one of the subtrees of B) in A until we reach a non-preferred edge, and keeping track of the largest and smallest keys in the appropriate ranges.

- Split A at l' into L and R', then split R' at r' to an empty tree and C.
- By construction, there is no node in A between l' and r' so the left subtree of r' is empty, and we can mark the non-preferred edge from r' to B as preferred. If super tango is running, then it will make an appropriate adjustment to the bound tree at this stage.
- Now join B and C at r' to make R,
- then join L and R at l'.

The resulting tree contains all nodes from A and B as desired.

4.3 Consistency and Structure Tree Invariants

In Section 4.2, we described split and join operations on auxiliary trees. In this section, we discuss tango and super tango behavior, show that they are valid, and show that super tango maintains the structure tree invariants.

We begin the section by giving a more explicit description of tango and super tango behavior. We then show that the structure tree invariants hold through splits and joins in sections 4.3.3 and 4.3.4. In these sections we assume as in Section 4.2 that the structure invariants hold until just before the split or join in question. We finish by using these results to form a simple proof by induction that the structural tree invariants always hold, and that the descriptions of tango (and super tango) are consistent.

4.3.1 Tango operation

The initial structure of the structure tree is the same as the bound tree, and each node is in an auxiliary tree by itself (all of the preferred directions are neither). When we perform a search in the structure tree for a node p, we first search for p in the auxiliary tree A containing the root of the structure tree. If p is in A, we split A at the bound depth of p. If p is not in A, we will eventually encounter a non-preferred edge during the search, leading to the root of another auxiliary tree B. When this happens, we find the bound depth of the successor and predecessor nodes in A of p. If only one of the nodes exists, split at the bound depth of the single node. Let A' be the portion of A with lower depth. After splitting A, we join A' and B to get B', then begin the process again. It is useful to note that all splits and joins involve the top level preferred subtree.

4.3.2 Super tango operation

Super tango performs exactly the operations described above on the bound tree as tango performs on the structure tree. However, splitting and joining in the bound tree do not entail structural rearrangement, only manipulating the preferred children of the bound tree nodes.

4.3.3 Structure tree invariants: splitting preferred subtrees

When tango performs a split, it creates a new non-preferred edge. In order to maintain the structure tree invariants, super tango must change one of the preferred edges in the bound tree to a non-preferred edge. Super tango can do this by changing the preferred child of one node to *neither*, as Theorem 4.7 shows.

Theorem 4.7. *Suppose that the structure tree invariants are maintained until just before the tango algorithm marks a node as a new root of an auxiliary tree during a split of an auxiliary tree. When tango performs this marking, super tango can change* *the preferred child of some node in the bound tree to* neither *at the same time to maintain both structure tree invariants.*

Proof. Let A be the preferred subtree of the structure tree that tango is splitting, and let B and C be the two preferred subtrees A is being split into, such that B has nodes with greater bound depth than C. Let A', B', and C' be the trees A, B, and C in the bound tree. The first structure tree invariant implies that A' is a (contiguous) section of a node-to-root path in the bound tree. The auxiliary tree split procedure includes in B all nodes in A with bound depth above a certain threshold. This implies that B' consists of the deeper portion of A', so there is one edge from p, the deepest node of C', to the shallowest node of B'. Changing the preferred child of p to neither will split A' into two preferred subtrees, B' and C'. This shows that the first structure tree invariant is maintained.

The second structure tree invariant is also maintained. The nodes that A and A' represent split into B and C or B' and C' respectively. The node for B is below C in the non-preferred structure tree as described in Section 4.2.4, and the node for B' is below C' in the non-preferred bound tree by definition of bound depth. □

Super tango performs this flip. Tango splits the top level preferred subtree at the deepest node that is on the path to the current search node, as does super tango (see Section 4.1.3). This gives the following result:

Corollary 4.8. *If the structure tree invariants hold before a split, they hold after a split.*

4.3.4 Structural tree invariants: joining preferred subtrees

When tango performs a join, it changes a non-preferred edge to a preferred edge. Super tango can do the same thing to maintain both structural invariants:

Theorem 4.9. *Suppose that the structure tree invariants are maintained until just before the tango algorithm removes the root mark of an auxiliary tree during a join* *of two auxiliary trees. Assume also that the union of nodes of the two trees being joined are a contiguous section of a node-to-root path in the bound tree. When tango unmarks the root, super tango can at the same time change the preferred child of some node without a preferred child in the bound tree to either* left *or* right *to maintain both structure tree invariants.*

Proof. Let A and B be the two preferred subtrees of the structure tree that tango is joining, and let B be the tree with nodes of greater bound depth. Let A' and B' be A and B in the bound tree.

When tango unmarks the root of B as the root of a preferred subtree, it joins A and B into a single preferred subtree, and coalesces their nodes in the non-preferred structure tree into a single node.

Super tango only needs to join A' and B' into a single preferred subtree to maintain the structure tree invariants. As A' and B' are not in the same preferred subtree, the edge from p, the deepest node in A', to q, the node of least depth in B', is non-preferred. As the nodes of A' and B' form a section of a node-to-root path, p cannot have a preferred child other than q, so p has *neither* as its preferred child. Changing p's preferred child to q (either *left* or *right* as appropriate) joins A' and B' to a single preferred subtree as necessary. □

Super tango performs this change. Tango joins the top level preferred subtree to the subtree between the successor and predecessor of the search node, as does super tango (see Section 4.1.3). This gives the following result:

Corollary 4.10. *If the structure tree invariants hold before a join, they hold after a join.*

4.3.5 Structure tree invariants and tango validity

We have shown some useful properties in this section. Theorem 4.4 shows that the structure tree invariants do not change during rotations performed during tango. Tango only performs rotations between two nodes in the same preferred subtree, so we only need worry about the structure tree invariants and validity of tango during the marking and unmarking steps of the split and join operations. Corollaries 4.8 and 4.10 imply that the invariants are not invalidated during a valid split or join. The bound and structure trees are initially the same tree, and are also the same as their non-preferred trees, so the invariants hold initially. There is only one thing left to show:

Theorem 4.11. *The tango algorithm will only try to join preferred subtrees A and B if the union of their keys forms a section of a node-to-root path in the bound tree.*

Proof. We will prove this by induction on the number of splits and joins, using the structure tree invariants. The structure tree invariants hold initially, and also through splits and rotations.

Suppose that tango is going to perform a join of trees A and B with the root of A on the node-to-root path of B in the structure tree. Let A' and B' be these subtrees in the bound tree. (This makes sense as the first structure tree invariant holds by the induction hypothesis.) Before tango attempts this join it searched for the predecessor and successor of B in the top level preferred subtree and split at the largest bound depth of these two nodes. Splitting at that depth implies that there are no nodes of A' that are not on the node-to-root path of the root of B'. As B is connected to A by a single non-preferred edge, the second structure tree invariant implies that there are no nodes in the bound tree on the node-to-root path of any node of B' and not in the node-to-root path of the nodes of A', save the nodes of B'. In other words, the union of nodes of A' and B' form a section of a node-to-root path in the bound tree as needed. □

This implies the following:

Theorem 4.12. *The structure tree invariants are always maintained.*

4.4 The Tango Lower Bound

We now show that the number of non-preferred edges encountered in all searches in the bound tree can be used to construct a cut lower bound on the performance of any BST on a given sequence. By the second structural invariant, the number of non-preferred edges encountered on searches in the bound tree is equal to the number of splits and an upper bound on the number of joins the tango algorithm performs. We use this equivalence to show that tango is $O(\log \log n)$-competitive. This section is modeled largely after Section 3.3, as the tango lower bound is just a basic modification of the first Wilber bound.

4.4.1 Bound tree form

We describe the bound in two ways, first in terms of preferred and non-preferred children, then in terms of cuts, much as in the descriptions of the Wilber I bound in Chapter 3.

> **Definitions:** Consider the searches performed for a BST access problem with key sequence S. When we encounter a non-preferred edge when leaving a node p in the bound tree, and then change the preferred child of p from *left* to *right* or *right* to *left*, the sequence S *crosses* p. The index of the key in S for which we are searching for when we make the flip is the *second crossing index* of the crossing. The index in S of the key where we last encountered p during a search is the *first crossing index* of the crossing.
>
> So far, the terminology is similar to the Wilber I terminology. For the super tango bound tree, there are other situations when we encounter p during an access. When we encounter a non-preferred edge when leaving p during a search, then change the preferred child of p to *left* or *right* from *neither*, the sequence *decides* p (it has dictated to p which child to prefer). When we find p during a search and change its preferred child from *left*

or *right* to *neither* (because we are searching for p), the sequence S *hits* p.

The *partial tango bound* on S is the number of crossings of S over all nodes in the bound tree. The *tango bound* for S is the partial tango bound plus the number of times we decide nodes in the bound tree.

Theorem 4.13. *The tango bound is the number of times we leave a node, encounter a non-preferred edge, and change the edge to a preferred edge in the bound tree during all searches.*

Proof. When we encounter a non-preferred edge when leaving a node p and change the edge to a preferred edge, we either change the preferred child of p from left to right, right to left, or from neither to left or right. The partial tango bound is the number of times we toggle a preferred child between right and left, while the number of decisions is the number of times we change a preferred child from neither to left or right. □

4.4.2 Cut lower bound

We now use a cut lower bound to show that the partial tango tree bound is a lower bound (within a constant factor) of the optimal BST access service sequence. The bound tree for this construction is very similar to the bound tree of the first Wilber bound, described in Section 3.3. In this case, however, vertical cuts pass through each key value rather than between key values. We place one vertical cut at the x coordinate corresponding to each key. Each cut extends below the first point and above the last point. Let L be the ordering of the keys so the dual-Cartesian tree on them is the bound tree for the super tango algorithm. The cut at x coordinate k is then ordered before the cut at x coordinate k' if and only if k is ordered before k' in L. Let this cut set be the tango cut set.

We exploit some theorems of Section 3.3. The major lemmas we need are stated below, with references to the lemmas in Section 3.3, and required modifications noted.

For ease of reading, the lemma bodies are replicated here. We do not make a distinction between a cut c and the node to which it corresponds in the bound tree. The bound tree mentioned in the following lemmas is the bound tree for the super tango algorithm, not the bound tree for the Wilber I bound.

We show that the partial tango tree bound is a lower bound by by showing that there is an injective map between crossings of nodes in the bound tree to boxes in the union of slightly modified crossing sets of the tango cut set. We do not use crossing sets. Instead we compute interior-disjoint subsets of the cut sets, called tango crossing sets, described below. By Corollary 3.9, this is sufficient to show that the tango tree bound is a lower bound (within a factor of 4).

4.4.3 Tango crossing sets

> **Definitions:** We call the modified crossing sets tango crossing sets. A tango crossing set is computed in the same way as a crossing set, but it begins with a subset of the cut set, called a tango cut set. Call an unstabbed box such that the x coordinate of one of the generators is an ancestor of the x coordinate of the other in the tango bound tree an *ancestor* box. The *tango cut set* is the cut set with all ancestor boxes discarded. The *tango crossing set* is constructed in the same way as the crossing set but begins with the tango cut set. Tango crossing sets are interior disjoint by Theorem 4.14. For keys (or nodes) p and q in the tango bound tree, let $\mathrm{lca}_b(p, q)$ be their lowest common ancestor.

Theorem 4.14. *If boxes a and b are in the same tango crossing set, they have disjoint interiors.*

Proof. Suppose that boxes a and b in the tango crossing set have intersecting interiors. We can use the same argument as in Lemma 3.4. The discard algorithm and Lemma 3.3 tell us that without loss of generality a and b are both + type unsatisfied boxes. Let a_u, a_l, b_u, and b_l be the upper and lower generating corners of their respective rectangles. Assume that $y(a_u) > y(b_u)$. We now argue that the boxes are in the configuration shown in Figure 4-2. Since the y range of one cannot contain the y range of the other, we must have $y(b_l) < y(a_l) < y(b_u) < y(a_u)$. As the boxes are unstabbed and their interiors intersect, the x coordinates must be related in the following manner: $x(a_l) < x(b_l) < x(a_u) < x(b_u)$.

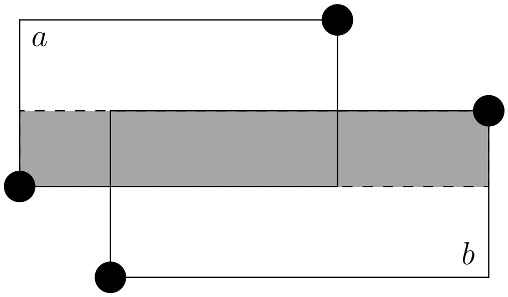

Figure 4-2: Intersecting boxes in a tango crossing set.

Note that $B(a_l, b_u)$, the shaded box in Figure 4-2, is also unsatisfied. Since a and b are in the same tango cut set, we know that $\text{lca}_b(x(a_l), x(a_u)) = \text{lca}_b(x(b_l), x(b_u))$, $x(a_l) \neq \text{lca}_b(x(a_l), x(a_u))$, and $x(b_u) \neq \text{lca}_b(x(b_l), x(b_u))$. As $x(a_l) < x(b_l)$ and $x(b_u) > x(a_u)$, $\text{lca}_b(x(a_l), x(b_u)) = \text{lca}_b(x(a_l), x(a_u))$. This and the earlier inequalities imply that $B(x(a_l), x(b_u))$ is in the tango cut set with a and b, a contradiction. A similar argument holds when $y(a_u) < y(b_u)$. □

This gives the following corollary, implying we can use Theorem 3.9.

Corollary 4.15. *The tango crossing set is a lower bound on the size of the crossing set for a cut.*

Proof. By Theorem 4.14, a tango crossing set consists of interior disjoint boxes. An interior disjoint subset of a cut set cannot be larger than a maximal interior disjoint subset of the cut set. □

4.4.4 Maps

Let $S = (s_1, s_2, \ldots)$ be a sequence of integers. Each crossing of S over an internal node may be mapped to a pair of indices: the first and second crossing indices of the crossing. Call this map $M_{\text{XING}_t \to \text{PAIR}}$. It is injective by Theorem 4.17. Because $M_{\text{XING}_t \to \text{PAIR}}$ is injective, it may be inverted on its image.

Each point in $M_{\text{BST} \to \text{BSP}}(S)$ has a unique y coordinate, so each unstabbed box in $M_{\text{BST} \to \text{BSP}}(S)$ may also be mapped to a pair of indices of S: the y values of its generating points. We will call this map $M_{\text{XBOX}_t \to \text{PAIR}}$. This map is injective because no no two unstabbed boxes cannot share both generators.

This gives the following result:

Theorem 4.16. *Let S be a sequence of keys. There is an injective map from crossings of S over nodes in the tango bound tree to the union of tango crossing sets of the tango cut set over $M_{\text{BST} \to \text{BSP}}(S)$.*

Proof. Theorems 4.18 and 4.23 show that the image under $M_{\text{XING}_t \to \text{PAIR}}$ of the crossings is the same as the image of the crossing sets. Invert one map and compose with the other to get an injective map from crossings to boxes: $M^{-1}_{\text{XING}_t \to \text{PAIR}}(M_{\text{XBOX}_t \to \text{PAIR}}(\cdot))$. □

4.4.5 An injective map

Theorem 4.17. *The map $M_{\text{XING}_t \to \text{PAIR}}$ is injective.*

Proof. Let $S = (s_1, s_2, \ldots)$ be a sequence of integers. Suppose that two distinct crossings of S in the bound tree have first and second crossing indices t and u. As the search for s_u changes the preferred direction that the search for s_t had set, there is only one node where the crossing can take place: the last node that the root-to-node paths of s_t and s_u share in the bound tree. Nodes closer to the root will not have their direction set by s_t and changed by s_u. A node can only have its direction switched once per access, so the pair of indices can only refer to one crossing. □

4.4.6 The image of crossings

Theorem 4.18. *Let $S = (s_1, s_2, \ldots)$ be a sequence of keys. A pair of indices (t, u) is in the image of the crossings of S of the bound tree under $M_{\text{XING}_t \to \text{PAIR}}$ if and only if $s_u \neq \text{lca}_b(s_t, s_u) \neq s_t$ and there is no index $r \in (t, u)$ such that s_r is in the subtree of $\text{lca}_b(s_t, s_u)$.*

Proof. See page 124 for definitions of crossing nodes, deciding nodes, and hitting nodes.

Suppose that there is an index r such that s_r is in $\text{lca}_b(s_t, s_u)$'s subtree and $r \in (t, u)$. Without loss of generality $s_t \leq \text{lca}_b(s_t, s_u) \leq s_u$. If $s_r \leq \text{lca}_b(s_t, s_u)$, then t cannot be the the first index of a crossing with u the second index. If $s_r \geq \text{lca}_b(s_t, s_u)$, u cannot be the second index of a crossing with t the first.

Now suppose that there is not an index $r \in (t, u)$ such that s_r is in the subtree of $\text{lca}_b(s_t, s_u)$. If $s_u = \text{lca}_b(s_t, s_u)$ then u will hit s_u and if $s_t = \text{lca}_b(s_t, s_u)$ then s_u will decide s_u. In either case, s_u cannot *cross* s_u. Suppose therefore, without loss of generality, that $s_t < \text{lca}_b(s_t, s_u) < s_u$). By definition of lca, searches for s_t and s_u must take different directions from $\text{lca}_b(s_t, s_u)$. As no other searches reach $\text{lca}_b(s_t, s_u)$'s subtree between the two indices, u must flip the direction that t set: t and u are the first and second indices of a crossing of $\text{lca}_b(s_t, s_u)$. □

4.4.7 The image of boxes

Now we turn to showing that the images of $M_{\text{XING}_t \to \text{PAIR}}$ and $M_{\text{XBOX}_t \to \text{PAIR}}$ are the same. We begin by linking unstabbed boxes to the locations of their generators in the bound tree.

> **Definitions:** A cut at x coordinate j corresponds to the node with key value j in the tango bound tree. When we refer a cut in the bound tree, we are referring to the node with the key equal to the x coordinate of the cut. Similarly, a point in the plane is *in a subtree* of the bound tree if the x coordinate of the point is a key in the subtree.

Lemma 4.19. *The tango cutting set of c is exactly the set of unstabbed boxes with one generating point in each subtree of c in the tango bound tree.*

Proof. Let P be a point set. Cuts extend beyond the y range of all boxes in both directions so any cut will split an unstabbed box if the cut is in the interior of the x range of the box. Also, no boxes are ever nicked.

Let $B(p, q)$ be an unstabbed box in P. There are few cases to consider.

- The cut c is not in the interior of the x range of $B(p, q)$. This implies that p and q cannot be in subtrees of different children of c and that c cannot split $B(p, q)$. This covers the case when $c \in \{p, q\}$.

- The cut c is in the interior of the x range of $B(p, q)$, but at least one of the nodes p is not in either c's subtrees. Two situations are now possible. If $\text{lca}_b(p, q)$ is ordered earlier in the cut set than c and splits $B(p, q)$ then c cannot cut $B(p, q)$. Otherwise $\text{lca}_b(p, q)$ is ordered earlier in the cut set that c, but does not split $B(p, q)$. In this case, the lowest of the $x(p)$ and $x(q)$ in the bound tree will be the ancestor of the other. In this case $B(p, q)$ is an ancestor box, so it will not be included in the tango cut set for c.

- Finally, suppose c is in the interior of the x range of $B(p, q)$, but p and q are in the subtrees of different children of c. Since $c = \text{lca}_b(p, q)$, there are no nodes ranked before c in the dual-Cartesian tree construction in the x range of $B(p, q)$. In other words, c is the earliest cut in the cut set that splits $B(p, q)$, so c cuts $B(p, q)$.

□

Lemma 4.20. *Let $S = (s_1, s_2, \ldots)$ be a BST access problem. If t and u are indices such that $s_t \neq \text{lca}_b(s_t, s_u) \neq s_u$ and there is no $r \in (t, u)$ with s_r in the subtree of $\text{lca}_b(s_t, s_u)$, then (t, s_t) and (u, s_u) generate an unstabbed non-ancestor box in $M_{\text{BST}\to\text{BSP}}(S)$.*

Proof. The point set $M_{\text{BST}\to\text{BSP}}(S)$ is (i, s_i) for $i \in [1, n]$ by definition, so (t, s_t) and (u, s_u) are in $M_{\text{BST}\to\text{BSP}}(S)$. Suppose that a point (r, s_r) stabs $B((t, s_t), (u, s_u))$. By definition of stabbing, s_r is between s_t and s_u (hence is in $\text{lca}_b(s_t, s_u)$'s subtree) and $r \in (t, u)$. Lemma 4.19 implies that no such r exists. □

Lemma 4.21. *Let $S = (s_1, s_2, \ldots)$ be a BST access problem. If t and u are indices such that $s_t \neq \text{lca}_b(s_t, s_u) \neq s_u$ and there is no $r \in (t, u)$ with s_r in the subtree of $\text{lca}_b(s_t, s_u)$, then the box $B((t, s_t), (u, s_u))$ is in the tango crossing set of $\text{lca}_b(s_t, s_u)$ over $M_{\text{BST}\to\text{BSP}}(S)$.*

Proof. Box $B((t, s_t), (u, s_u))$ is unstabbed in $M_{\text{BST}\to\text{BSP}}(S)$ by Lemma 4.20. Lemma 4.19 implies that $B((t, s_t), (u, s_u))$ is in the tango cutting set of $\text{lca}_b(s_t, s_u)$. Suppose that the tango cutting set of $\text{lca}_b(s_t, s_u)$ also contains another box $B((v, s_v), (w, s_w))$ with $(v, w) \subset (t, u)$. By Lemma 4.19, s_v and s_w are in $\text{lca}_b(s_t, s_u)$'s subtree with both v and w in (t, u). If $B((v, s_v), (w, s_w))$ is distinct from $B((t, s_t), (u, s_u))$ then $\{v, w\} \neq \{s, t\}$. However, this contradicts the assumption of the lemma. □

Lemma 4.22. *Let $S = (s_1, s_2, \ldots)$ be a BST access problem. If t and u are indices such that $s_t \neq \text{lca}_b(s_t, s_u) \neq s_u$ and there is an $r \in (t, u)$ with s_r in the subtree of $\text{lca}_b(s_t, s_u)$, then the box $B((t, s_t), (u, s_u))$ is not in the tango crossing set of $\text{lca}_b(s_t, s_u)$ over $M_{\text{BST}\to\text{BSP}}(S)$.*

Proof. As s_t and s_u are in the subtrees of different children of $\text{lca}_b(s_t, s_u)$ and s_r is in the subtree of $\text{lca}_b(s_t, s_u)$, either $\text{lca}_b(s_t, s_r) = \text{lca}_b(s_t, s_u)$ or $\text{lca}_b(s_u, s_r) = \text{lca}_b(s_t, s_u)$. This implies that there are two indices $v, w \in [t, u]$ with $[v, w]$ strictly contained in $[t, u]$ such that we can apply Lemma 4.21 to see that $\text{lca}_b(s_t, s_u)$ cuts $B((v, s_v), (w, s_w))$. In other words, the $B((t, s_t), (u, s_u))$ contains the y range of another box in $\text{lca}_b(s_t, s_u)$'s cut set: $B((t, s_t), (u, s_u))$ cannot be in $\text{lca}_b(s_t, s_u)$'s tango crossing set. □

Theorem 4.23. *Let $S = (s_1, s_2, \ldots)$ be a sequence of keys. A pair of indices, (t, u), with $s_t \neq \text{lca}_b(s_t, s_u) \neq s_u$ is in the image of the tango crossing sets of $M_{\text{BST}\to\text{BSP}}(S)$*

over the tango cut set if and only if there is no index $r \in (t, u)$ such that s_r is in the subtree of $\mathrm{lca}_b(s_t, s_u)$ in the bound tree.

Proof. This follows directly from lemmas 3.22 and 3.23. □

4.5 Tango Cost

Finally we have sufficient results to show that tango is an $O(\log\log n)$ competitive algorithm compared to an optimal offline BST algorithm. This is stated below:

Theorem 4.24. *Let S be a BST access problem of length m with n keys, with $m = \Omega(n)$. Let $\mathrm{OPT}(S)$ be the minimum search and rearrangement cost for any BST access service sequence for S. The tango algorithm has total search and rearrangement cost at most $O(\mathrm{OPT}(S)\log\log n)$.*

Proof. Let A be the number of non-preferred edges encountered and changed to preferred edges in the bound tree during a run of super tango. The structure tree invariants imply that $A+m$ is an upper bound on the number of splits, joins, and searches with auxiliary trees: when we encounter a non-preferred edge, we perform one search, one split, and possibly a join. To then find the node we are searching for, we perform at most one search and one split. By Theorem 4.13, A is equal to the tango lower bound for S. As the size of any auxiliary tree is at most $\log n$, and the split, join, and search operations are logarithmic in the size of the largest tree involved in the operation, the total search and reorganization cost for tango is $(A + m)\log\log n$.

By Theorem 4.16, the partial tango bound is $O(\mathrm{OPT}(S))$. A node has preferred child *neither* only if it has not yet been encountered during a search, or if it has not been encountered since it was last hit. Only one node is hit per search, and there are at most n nodes with no preferred children initially, so we can decide at most $O(m + n)$ nodes during all searches. This implies that the tango lower bound is $O(\mathrm{OPT}(S) + m + n)$. As $m \geq n$ and $\mathrm{OPT}(S) \geq m$, this is $O(\mathrm{OPT}(S))$.

This implies that the total tango cost is

$$O((\mathrm{OPT}(S) + m) \log \log n) = O(\mathrm{OPT}(S) \log \log n).$$

□

4.6 Tango Limitations: bound and algorithm limitations

Although tango is $o(\log n)$-competitive, it is $\Omega(\log \log n)$-competitive as an adversary may always pick a leaf node of the bound tree at cost $\Omega(\log \log n)$, even if the adversary always picks the same node.

Note that any algorithm using the tango bound cannot to better. Consider accessing just the nodes in the left backbone of the tango bound tree, but in bit reversed sequence. The cost for an optimal sequence is $\Omega(k \log k)$ for a k key bit-reversal sequence [33], so the cost for accessing the backbone nodes is $\Omega((\log n)(\log \log n))$. However, the lower bound cost is $O(\log n)$: we never flip a node's preferred direction from left to right or right to left.

4.7 Conclusion

The major result of this chapter is tango, an $o(\log n)$-competitive online algorithm. This result has been extended by Sleator et al. [5] to use splay trees instead of red-black trees for the auxiliary trees.

The main questions in this line of research are what they usually are: can we improve the upper or lower bounds. Specifically:

- Can we exhibit another online algorithm and prove it is strictly more efficient than tango on worst-case sequences?

- Failing that, can we prove no online algorithm can do better than $O(\log\log n)$-competitive?

Another interesting question posed during the defense of this thesis was: What is the largest constant c for which we can prove there is no c-competitive online dynamically optimal algorithm?

Chapter 5

Conclusion, Conjectures, and Open Questions

5.1 Summary

The major driving question in thesis was: Can we produce a dynamically optimal online algorithm. We have made progress in this direction on two fronts:

- We exhibited the first $o(\log n)$-competitive online algorithm.
- Using a novel geometric model, we described a new (unifying) class of lower bounds on the cost of optimal offline BST algorithms, and provided the largest such bound in this class.

The next questions fall into roughly two categories: can we make a better BST and can we make better lower bounds? The ultimate goal is to make the the asymptotic cost of a BST match the asymptotic value of a lower bound on the behavior of an optimal BST.

5.1.1 Dynamically optimal BSTs: better upper bounds

Can we construct any polynomial time computable BST that is dynamically optimal? In other words, can we compute OPT within a constant factor in polynomial time?

This is clearly a necessary condition for the existence of a strict, dynamically optimal online BST algorithm. We think that MUNRO is the right algorithm to consider: we hypothesize it is dynamically optimal, and that it has a (strict) online version.

However even a polynomial time offline BST would be interesting and would probably let us analyze current BST behavior in new cases. In Section 5.2 below, we discuss some one way to the box model and a geometric algorithm to consider this problem.

5.1.2 Better lower bounds

Although we constructed new lower bounds in this thesis, it is not clear these bounds are tight. We may need to construct a better lower bound that accounts for interactions between the points we add to satisfy the initial unstabbed boxes. Instead of looking for better lower bounds on offline BST algorithms, we may be able to find a lower bound on *online* BST algorithms. It is still unknown if the best online can do as well as the best offline algorithms.

5.2 Upper Corner Lattice

After formulating the box model equivalents for BSTs, we tried many geometric methods to find an $O(1)$-competitive polynomial time algorithm for the box stabbing problem. We describe the basis for some of these attacks in this section: the upper corner lattice. The upper corner lattice of a set of points is a subset of the lattice of the points. We show that the upper corner lattice contains a satisfied subset. There is no guarantee it contains a subset within a constant factor of the size of an optimal subset, but we intended to use other means for justifying each point added. Although we assume that all points have unique x and y coordinates, a modified version of these proofs should work for degenerate coordinates.

> **Definitions:** Let P be a point set. An *upper corner* of P is the upper empty corner of an unsatisfied box in P. The *upper corner lattice* of P is

P and all upper corners of P. The *sign* of an upper corner lattice point is zero if the point is in P, + if the point is the upper empty corner of a + type unsatisfied box, and − otherwise.

Theorem 5.1. *Let S be an arbitrary subset of the upper corner lattice of P containing P. Every unsatisfied box between points in S is stabbed by at least one point in the upper corner lattice of P.*

Proof. Let B be an unsatisfied box between two points in S. The upper generating point of B may be a + type upper corner, a − type upper corner, or an initial point, as may the bottom generating point. B may be a + type or a − type unsatisfied box. This leads to $2 \cdot 3^2 = 18$ distinct cases to analyze. However, reflecting about the y axis maps + to −, halving the number of cases we must analyze. We refer to cases by triples. The first two elements of the triple refer to the signs of the upper and lower generating corners of the box respectively; the last refers to the sign of the box. The coordinates for the corners may be 0, +, or −. The sign 0 is a point in P, and + and − refer to upper corners of + or − type unsatisfied boxes. The box type may be either + or −.

When discussing the different cases, p will always be the upper generating point and q the lower. If p is not an initial point, p_U and p_L are the upper and lower generating points of the box for which p is a corner point. The points q_U and q_L are defined similarly. The axis-box between two points s and t is $B(s,t)$. The upper corner lattice points present on the box $B(p,q)$ are designated a or b. We will have need of referring to one other point in P for some of the cases, and will call this point c.

- $(-,-,-)$: We must have $x(q_U) < x(p_U)$ as $B(p,q)$ is unsatisfied and $B(p_U, p_L,)$ contains no points in P. Similarly $y(p_L) < y(q_L)$. Let a be the point $(x(q), y(p))$. These positions imply the box $B(p_U, q_L)$ contains no points of P so a is a point of the upper corner lattice.

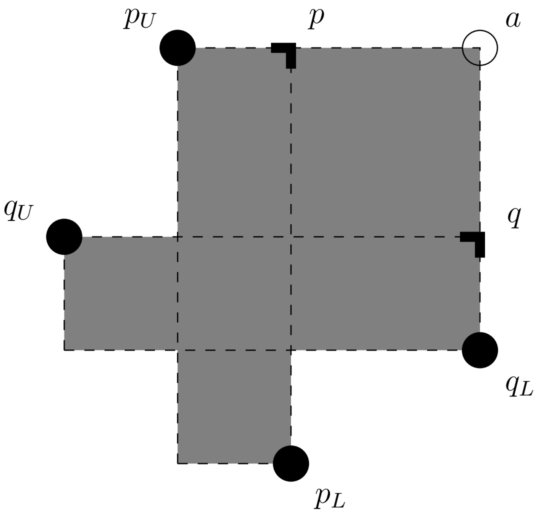

Figure 5-1: $(-,-,-)$

- $(-,-,+)$: Let a be the point $(x(q), y(p))$, and let b be the point $(x(p), y(q))$. We must have $x(p_U) < x(q)$ and $y(p_L) < y(q)$ because the box $B(p,q)$ is unsatisfied. The points q_U and q_L must be outside the boundaries of the box $B(p_U, p_L)$ as $B(p_U, p_L)$ contains no points in P. These restrictions imply $B(p_U, q_L)$ and $B(q_U, p_L)$ are unsatisfied in P: the points a and b are in the upper corner lattice.

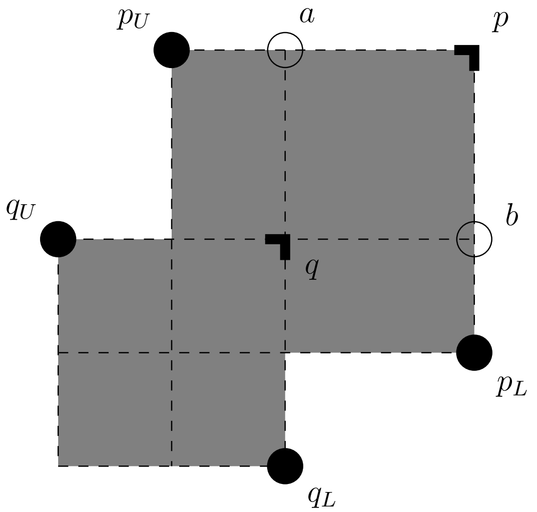

Figure 5-2: $(-,-,+)$

- $(-, 0, -)$: Let a be the point $(x(q), y(p))$. We must have $y(p_L) < y(q)$ as $B(p, q)$ is empty. This implies that $B(p_U, q)$ contains no points of P so a is on the upper corner lattice.

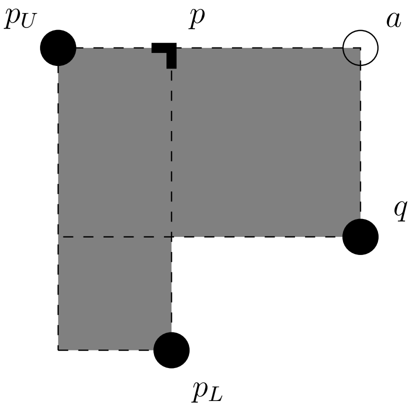

Figure 5-3: $(-, 0, -)$

- $(-, 0, +)$: Because $B(p, q)$ is empty, $x(p_U) < x(q)$ and $y(p_L) < y(q)$. This implies that q is in $B(p_U, p_L)$, a contradiction.

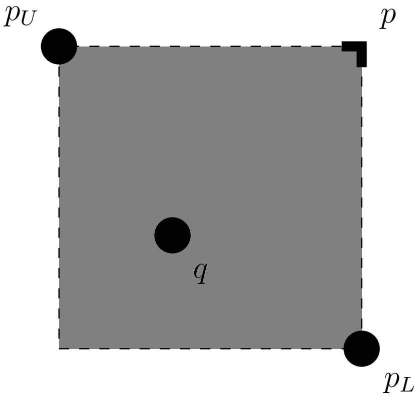

Figure 5-4: $(-, 0, +)$

- $(-, +, -)$: Let R be the rectangle with x coordinate range $[x(p), x(q)]$ and y coordinate range $[y(q), y(q_L)]$. Let c be the highest initial point in R. The point c could be p_L, q_L, or another point. Let a be the point $(x(c), y(p))$ and let b be the point $(x(c), y(q))$. Because $B(p, q)$ is unsatisfied, $y(p_L) < y(q)$, and thus

$y(p_L) \leq y(c)$. As all points of P have unique y coordinates, the boxes $B(p_U, c)$ and $B(q_U, c)$ can contain no points of P: the points a and b are in the upper corner lattice, and at least one is distinct from p and q.

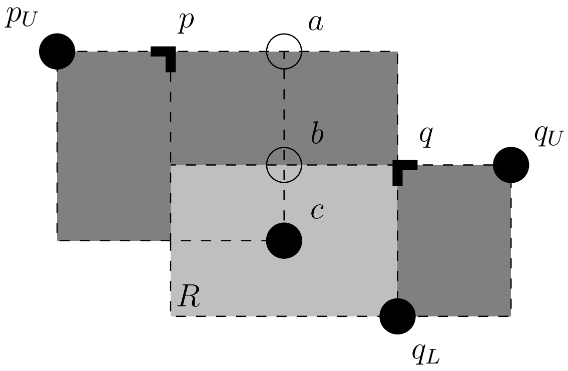

Figure 5-5: $(-, +, -)$

- $(-, +, +)$: We have $x(p_U) < x(q)$ and $y(p_L) < y(q)$ as $B(p, q)$ is unsatisfied. Since $B(p_U, p_L)$ contains no points of P, $y(q_L) < y(p_L)$ and $x(p) < x(q_U)$. This implies that p_L is in $B(q_U, q_L)$, a contradiction.

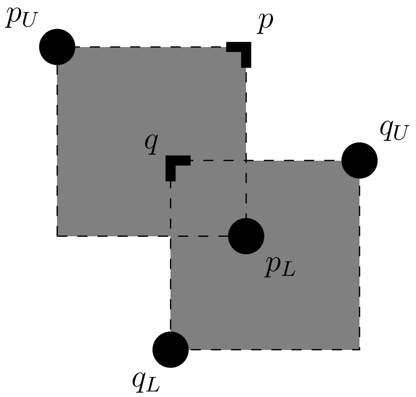

Figure 5-6: $(-, +, +)$

- $(0, -, -)$: Let a be the point $(x(q), y(p))$. We must have $x(q_U) < x(p)$ as $B(p, q)$ is unsatisfied. As $B(q_U, q_L)$ contains no points of P, $B(p, q_L)$ cannot either: a is on the upper corner lattice.

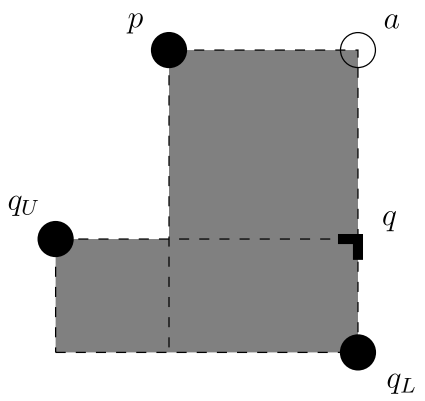

Figure 5-7: $(0,-,-)$

- $(0,-,+)$: Let R be the rectangle with x coordinate range $[x(q),x(p)]$ and y coordinate range $[y(q_L),y(q)]$. Let c be the highest point in P in R: c could be q_L or another point. Let a be the point $(x(c),y(p))$ and let b be the point $(x(c),y(q))$. As points in P have unique y coordinates, $B(p,c)$ and $B(q_U,c)$ must be empty: both a and b are on the upper corner lattice, and at least a is distinct from q.

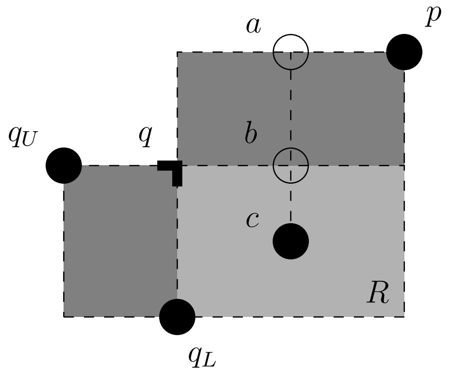

Figure 5-8: $(0,-,+)$

- $(0,0,-)$: The upper unoccupied corner of the box is on the upper corner lattice by definition.

- If we reflect the point set about the y axis, all $+$ corners become $-$ corners: all remaining cases are reflections of the earlier cases.

□

5.3 Decoupling the upper plus lattice and the upper minus lattice

We can decouple the upper corner lattice into two distinct sets of points, as we now show.

> **Definitions:** Let P be a point set. The *upper plus lattice* of P is the set of all points of the upper corner lattice of P with positive sign. The *upper minus lattice* is defined similarly. An upper corner p of P is a *Wilber II point* of P if the closest point of the upper corner lattice directly below, above, to the left, or to the right of p is of different sign.

Lemma 5.2. *Let S be a subset of the upper corner lattice of P that contains all Wilber II points. Any unsatisfied box between points in S is between upper corners of the same, non-zero sign.*

Proof. Again we proceed by case analysis of the cases remaining from Lemma 5.1. Points and regions are labeled as for Lemma 5.1

- $(-, 0, -)$: Let R be the rectangle with x coordinate range $[x(p), x(q))$ and y coordinate range $[y(p_L), y(q))$. Let c be the highest point in R. Let b be the point $(x(c), y(q))$ and let a be the point $(x(c), y(p))$. Since $B(p_U, q)$ is empty, a and b are on the upper minus lattice and upper plus lattice respectively. As they have the same x coordinate but different sign, at least two Wilber II points exist in S $\{(\xi, \gamma) | \xi = x(c), y(q) \leq \gamma \leq y(p)\}$.

- $(-, +, -)$: See Figure 5-5 on page 140. Points a and b are of opposite sign with the same x coordinate so there must be at least two Wilber II points in the set $\{(\xi, \gamma) | \xi = x(c), y(q) \leq \gamma \leq y(p)\}$.

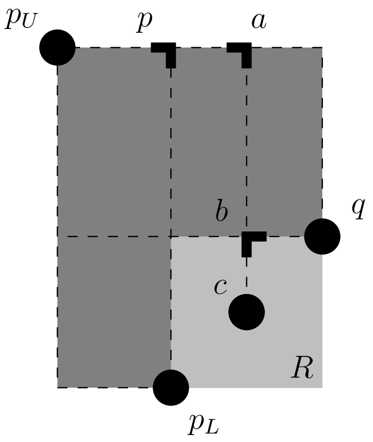

Figure 5-9: $(-, 0, -)$

- $(0, -, -)$: As a is on the minus corner lattice, the point on the minus corner lattice closest to p with y coordinate $y(p)$ is in the x coordinate range $(x(p), x(a)]$: there is a Wilber II point in $B(p, q)$.

- $(0, -, +)$: See Figure 5-8 on page 141. The same argument as for case $(-, +, -)$ applies.

- $(0, 0, -)$: See Figure 5-7 on page 141. The same argument as for case $(0, -, -)$ applies.

□

Appendix A

The Reorganization Tree Cost Model

In this appendix, we describe the reorganization tree model and show that the BST access problem under the reorganization tree model is closely related to the standard BST access problem from Section 1.2.3. In particular, the optimal cost under the reorganization tree model is within a constant factor of the optimal cost under the standard model.

A.1 Motivation

Any access service sequence specifies a method of finding each accessed node and rearranging the BST in some way before the next access. The conventional way of rearranging the tree is to use *rotations*, as described in Section 1.2.3. We call this the *standard model*.

Lucas [24] shows that we may assume that the edges touched during an access and all edges involved in rotations following an access form a connected subtree of the BST containing the root and the accessed node. The use of edges in her discussion is slightly confusing, but is useful for Lemma A.3. A rotation changes the edges surrounding it as shown in Figure A-1. These changes result from rotating edge 3, between nodes p and q. Edge 5 is present if neither p nor q is the root of the BST.

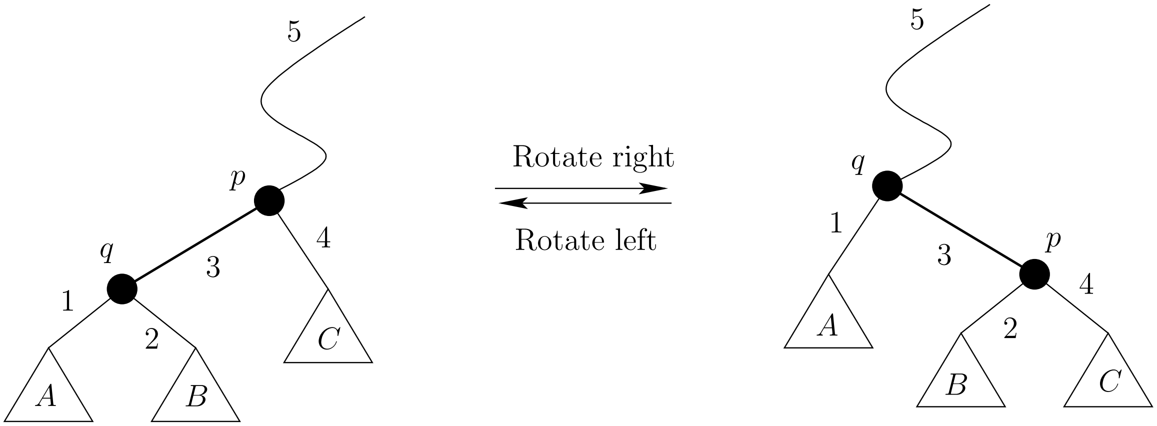

Figure A-1: Edge changes from a rotation.

The informal argument for Lucas' theorem is that any rotations involving edges not connected to such a tree may be delayed until a later access. We state the theorem without proof.

> **Definitions:** An edge *touched* during an access is an edge that is on the node-to-root path of the accessed node at the time of access, and an edge *touched* during a rotation is simply the edge that the rotation rotates.

Theorem A.1. *The set of all edges touched during an access a and involved in rotations before the next access b forms a connected subtree of the BST containing the root and the accessed node, both just before a and just before b.*

Proof. See page 3 of Lucas' paper [24]. □

This leads us to propose reorganization trees. Instead of worrying about specific rotations necessary for a particular rearrangement, we concentrate instead on the set of nodes touched in finding a node, and in all rotations before the next access. For each access in an access service sequence, this set of nodes is the *reorganization tree* for the access, and is a subtree of the BST containing the root by Theorem A.1. The structure of the reorganization tree when the corresponding key is accessed is the *initial structure* of the reorganization tree, while the structure immediately before the next node is accessed is the *final structure* of the reorganization tree.

A.2 A New Model for the BST Access Problem

This section defines reorganization tree sequences, then uses them to define a variant of the BST access problem.

> **Definitions:** A *reorganization tree sequence* is a sequence of BSTs. A reorganization tree sequence represents another sequence of BSTs called the *expanded sequence* of the reorganization tree sequence. Each BST in the represented sequence of an reorganization tree contains a fixed set of keys. However, the trees in the reorganization tree sequence will not, in general, contain all of these keys. The first tree in a reorganization tree sequence is called the *initial tree* of the sequence. The other trees are called reorganization trees, and only contain keys found in the initial tree. Each tree in the expanded sequence of a reorganization tree contains the same keys as the initial tree of the reorganization tree.
>
> Let T be an reorganization tree sequence, and let B be T's expanded sequence. The trees T_i and B_i are the ith tree in T and B respectively. B_1 and T_1 are the same tree. To compute B_i, first compute B_{i-1}. The structure of B_i is found by rearranging the subtree of B_{i-1} containing the nodes of T_i to match the structure of T_i. *This can only be accomplished if the nodes in B_{i-1} corresponding to the keys in T_i form a subtree of B_{i-1} containing the root. This condition is the defining characteristic of a valid reorganization tree sequence.*
>
> Given a sequence of keys S, a *reorganization tree access service sequence* is a valid reorganization tree sequence T such that the ith reorganization

tree of T contains the ith element of S. The initial tree of T may be picked arbitrarily, but must contain all keys in S. The *cost* of an reorganization tree access service sequence is the number of nodes in all reorganization trees of the sequence. The *reorganization tree BST access problem* for a sequence of keys S is to find the reorganization tree access service sequence for S of minimum cost.

A.3 Equivalence of BST Access Problems

The rest of this appendix is dedicated to proving the following theorem:

Theorem A.2. *Let S be a sequence. The following statements hold:*

- *If A is an access service sequence for S, there is a reorganization tree access service sequence B such that the cost of B is at most the cost of A.*
- *If B is a reorganization tree access service sequence for S, then there is an access service sequence A such that the cost of A is at most three times that of B.*

This shows that the optimal access service sequence and optimal reorganization tree access service sequence have costs within a constant factor of each other, so solving the reorganization tree BST access problem is equivalent to solving the BST access problem if we do not care about constant factors. We do not in this thesis.

A.3.1 From the Standard Model to the Reorganization Tree Model

To prove the first statement of A.2, we provide a simple map.

Lemma A.3. *If S is a sequence of keys and A is an access service sequence for S, then there is a reorganization tree access service sequence B such that the cost of B is no more than the cost of A.*

Proof. From Theorem A.1, without loss of generality the set of edges touched in an access a and involved in all rotations before the access immediately following a in A form a connected subtree of the BST containing the root.

For some access a of A, the *reorganization tree of* a consists of the nodes in the tree of edges touched during a's access and in all rotations between a and the next access in A. The structure of the reorganization tree of a is the structure of the subtree of the BST containing these nodes just before the access following a in A. B is the sequence of trees beginning with the initial tree of A followed by followed by the reorganization tree of each access in A in order.

Each node but the first in a's access path adds at most one edge to the reorganization tree of a, as does each rotation between a and the next access in A. The number of nodes in a tree is one more than the number of edges in the tree, so the size (in nodes) of the reorganization tree of a is at most the cost of a in A plus the number of rotations between a and the next access in A. □

A.3.2 From the Reorganization Tree Model to the Standard Model

The following result is well known, but we refer once again to Lucas for the proof:

Lemma A.4. *If we know the initial shape and final shape of the reorganization tree with* k *nodes, there is a sequence of at most* $2k$ *rotations that changes the initial structure to the final structure.*

Proof. See page 3 of Lucas' paper [24] for a slightly more detailed proof. A *right path* is the tree where each node has a right child, but no left child. It takes at most k rotations to change from any tree on k nodes to a right path, and as rotations are reversible at most k rotations to change from a right path to any tree on k nodes. If we go from the initial structure to a right path and then to the final structure, it takes at most $2k$ rotations. □

This lets us prove the second part of Theorem A.2:

Lemma A.5. *If S is a sequence and B is a reorganization tree access service sequence for S, then there is an access service sequence A such that the cost of A is at most three times the cost of B.*

Proof. A's initial tree is the first element of B. Let B' be B without its first tree. Let b_i be the ith tree in B'. Replace b_i with the following sequence: an access to the ith key in S followed by at most $2|b_i|$ rotations that transform the initial structure of the subtree containing the nodes of b_i to b_i. This is possible by Lemma A.4. B' with these replacements is the service sequence for A. □

A.4 A Property of Reorganization Tree Sequences

In this section, we prove a simple property of reorganization tree sequences used elsewhere in the thesis.

Lemma A.6. *After applying a single reorganization tree, no node not in the reorganization tree is on the node-to-root path of any node in the reorganization tree.*

Proof. The nodes of the reorganization tree form a subtree containing the root both before and after the reorganization tree is applied. □

Theorem A.7. *Let R be a reorganization tree sequence and T be R's expanded sequence. If $\mathrm{lca}(p, q)$ is r in T_i but not in T_j for some $j > i$, then r and and another node in $[p, q]$ are in a tree R_k for $k \in (i, j]$. Moreover, if k is the smallest index after i such that $i < k$ and $\mathrm{lca}(p, q) \neq r$ in T_k then r and $\mathrm{lca}(p, q)$ are in R_k.*

Proof. Let $s = \mathrm{lca}(p, q)$ in T_k. As $\mathrm{lca}(p, q) = r$ in T_{k-1}, and s is between p and q, s is on the node-to-root path of r in T_k, and r is on the node-to-root path of s in T_{k-1}. After applying a single reorganization tree, no node of the reorganization tree is below a node that is not in the reorganization tree by Lemma A.6. This implies s is in R_{k-1}. However, r is on s's node-to-root path in T_{k-1} so r is also in R_{k-1}. □

Corollary A.8. *Let R be a reorganization tree sequence and T be R's expanded sequence. If p is on q's node-to-root path in T_i but not in T_j for $j > i$ then there is a $k \in (i, j]$ such that $p \in R_k$.*

Proof. Let k be the lowest index after i such that p is not in q's node-to-root path in T_k, and let $r = \text{lca}(p, q)$ in T_k. By construction, $\text{lca}(p, q) = p$ in T_{k-1}. Theorem A.7 now implies p is in R_k. □

Corollary A.9. *Let R be a reorganization tree sequence and T be R's expanded sequence. If p is not on q's node-to-root path in T_i but is in q's node-to-root path in T_j for $j > i$ then there is a $k \in (i, j]$ such that $p \in R_k$.*

Proof. Let k be the lowest index after i such that $p = \text{lca}(p, q)$ in T_k. Theorem A.7 implies p is in R_k. □

Bibliography

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] Avrim Blum, Suchi Chawla, and Adam Kalai. Static optimality and dynamic search-optimality in lists and trees. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–8, 2002.

[3] Mihai Bădoiu and Erik D. Demaine. A simplified and dynamic unified structure. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics*, volume 2976 of *Lecture Notes in Computer Science*, pages 466–473, Buenos Aires, Argentina, April 2004. Springer-Verlag.

[4] R. Chaudhuri and H. Höft. Splaying a search tree in preorder takes linear time. *ACM SIGACT News*, 24(2):88–93, 1993.

[5] Chris Wang Chengwen, Jonathan Derryberry, and Daniel Dominic Sleator. $o(\log \log n)$-competitive dynamic binary search trees. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2006.

[6] Victor Chepoi, Karim Nouioua, and Yan Vaxés. A rounding algorithm for approximating minimum Manhattan networks. In Chandra Chekuri, Klaus Jansen, Jos D. P. Rolim, and Luca Trevisan, editors, *Proceedings of the 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems.*, volume 3624 of *Lecture Notes in Computer Science*, pages 40–51. Springer-Verlag, 2005.

[7] Richard Cole. On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof. *SIAM Journal of Computing*, 30(1):44–85, 2000.

[8] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting (log n)-Block Sequences. *SIAM Journal of Computing*, 30(1):1–43, 2000.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1999.

[10] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic Optimality—Almost. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pages 484–490, October 17-=19 2004.

[11] J. Derryberry, A. Gupta, D. Sleator, and C. Wang. LP Cover Lower Bound. Technical report, Carnegie Mellon University, Feb 2006.

[12] Jonathan Derryberry, Daniel Dominic Sleator, and Chengwen Chris Wang. A Lower Bound Framework for Binary Search Trees with Rotations. Technical report, Carnegie Mellon University, 2005.

[13] Z. Drezner. On the rectangular p-center problem: Heuristics and optimal algorithms. *Journal of the Operations Research Society*, 35:741–748, 1984.

[14] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, New York, NY, 1984. ACM Press.

[15] G. F. Georgakopoulos and Dj. J. McClurkin. General splay: a basic theory and calculus. In *Proceedings of 10th ISAAC*, volume 1741 of *Lecture Notes in Computer Science*, pages 4–17. Springer-Verlag, 1999.

[16] Joachim Gudmundsson, Christos Levcopoulos, and Giri Narasimhan. Approximating Minimum Manhattan Networks. *Nordic Journal of Computing*, 8:216–229, 2001.

[17] Haripriyan Hampapuram and Michael L. Fredman. Optimal biweighted binary trees and the complexity of maintaining partial sums. *SIAM Journal on Computing*, 28(1):1–9, 1998.

[18] John Iacono. Alternatives to splay trees with $o(\log n)$ worst-case access times. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 516–522, Washington D.C., January 2001.

[19] John Iacono. Unpublished correspondence. Email, Sept. 2004. An attempt at showing the box model is NP-complete using reduction to planar 3NAE (and more or less to planar 3SAT). We cannot come up with a convincing splitter.

[20] John Iacono. Key Independent Optimality. *Algorithmica*, 2(1):3–10, 2005.

[21] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[22] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[23] M. T. Ko, R. C. T. Lee, and J. S. Chang. Rectilinear m-center problem. In *Proceedings of the National Computer Symposium*, pages 325–329, Taipai, Taiwan, 1987.

[24] Joan M. Lucas. Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Rutgers University, 1988.

[25] N. Megiddo and K. Supowit. On the complexity of some common geometric locaiton problems. *SIAM Journal of Computing*, 13:182–196, 1984.

[26] Kurt Mehlhorn. *Data Structures and Algorithms I: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.

[27] J. Ian Munro. On the Competitiveness of Linear Search. In *Proceedings of the 8th Annual European Symposium on Algorithms*, pages 338–345, September 2000.

[28] Mihai Pătraşcu and Erik D. Demaine. Tight bounds for the partial-sums problem. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 20–29, New Orleans, Louisiana, January 2004.

[29] Micha Sharir and Emo Welzl. Rectilinear and Polygonal p-Piercing and p-Center Problems. In *Proceedings of the twelfth annual symposium on Computatoinal geometry*, Annual Symposium on Computational Geometry, pages 122–132, 1996.

[30] Danieal Dominic Sleator and Robert Endre Tarjan. Self-Adjusting Binary Search Trees. *Journal of the Association for Computing Machinery*, 32(3):652–686, July 1985.

[31] Ashok Subramanian. An Explanation of Splaying. *Journal of Algorithms*, 20:512–525, 1996.

[32] R. E. Tarjan. Sequential Access in Splay Trees Takes Linear Time. *Combinatorica*, 5(4):367–378, 1985.

[33] Robert Wilber. Lower Bounds for Accessing Binary Search Trees with rotations. *SIAM Journal on Computing*, 18:56–67, 1989.