Dynamic Optimality and Tango Trees

Yash Maniyar, Nikhil Sardana, Vinjai Vale

June 2019

1 Alice in Theoryland

Imagine you're the average, everyday balanced binary search tree. You're walking along, minding your leaves, when all of a sudden, a wild, unshaven computer scientist jumps out in front of you and starts peppering you with queries.

"Quick, find x_1," the computer scientist shouts. You sigh, descend down your nodes, and retrieve the element for him from one of your leaves.

"Great. Now, find x_1 again," the computer scientist demands. "Really, dude? You just asked me for that," you protest, but there's no convincing him. You once more perform logarithmic work, and give the computer scientist back x_1.

Over and over, the computer scientist asks for x_1. Over and over, you perform logarithmic work. Finally, after n queries, you gasp, exhausted: "You know, if you just told me ahead of time you were only looking for x_1, I would have just maintained a pointer to it. I could have done a factor of $\log(n)$ less work."

The computer scientist relents, admitting: "You're right. That wasn't very fair. How about I ask you a different set of queries?"

"Thanks," you reply. You're a pretty good binary search tree. You're balanced. You've only got $O(\log(n))$ layers. For worst-case sequences of queries, you're provably optimal. What could a computer scientist possibly throw at you?

"Find me x_1," the computer scientist asks. You perform logarithmic work, and give it to him. "Find me x_2", he queries again, and you once more do logarithmic work and give it to him. The computer scientist proceeds to ask for x_3, x_4, $x_5, \ldots$, asking once for each of your nodes, in order. At the end of the day, you've performed $n \log(n)$ work—you're tired, hungry, and bruised, and you remark: "You know, if you had just told me up front you were searching for the keys in order, I would have just done an inorder traversal. It would have saved me a factor of $\log(n)$ time."

The computer scientist replies: "Ah, but if you were *dynamically optimal*, you wouldn't need to know my queries ahead of time. No matter the sequence of queries, a dynamically optimal tree can match the best binary search tree."

2 Dynamic Optimality

An *online* algorithm is one that receives and processes its input serially—like a computer scientist bombarding an unsuspecting tree with queries—rather than all at once. Inherently, an online algorithm is harder to optimize than an *offline* algorithm, where we can glean information about operation patterns to organize our data.

Dynamic optimality refers to an online binary search tree that matches the best binary search tree for any sequence of queries. The "best" binary search tree for any given access sequence X (denoted $\text{OPT}(X)$) refers to the fastest any *offline* binary search tree can operate on the sequence.

How do we find this "best" offline binary search tree? Given exponential time, $\text{OPT}(X)$ can be computed exactly for any access sequence—however, we're unlikely to ever find a practical method for computing $\text{OPT}(X)$ for large sequences, since it's believed that computing $\text{OPT}(X)$ is NP-complete [1].

Nevertheless, our focus centers not around offline optimality but dynamic optimality—does there exist a dynamically optimal binary search tree? The Tango Tree [2] is the first real progress towards finding a dynamically optimal binary search tree since the Dynamic Optimality Conjecture was posed in 1985. But before we get into Tango Trees and their construction, we'll need to introduce the notion of *competitiveness* and find a new way of looking at binary search trees.

2.1 Competitiveness

An algorithm T has a competitive ratio a if for any sequence of operations π

$$\text{COST}_T(\pi) \leq a \cdot \text{COST}_{OPT}(\pi)$$

where OPT is a theoretically optimal algorithm for the sequence π. In other words, if T is a-competitive, the cost of any set of operations for T is within a multiple of a of the optimal algorithm.

It is important to emphasize that an algorithm is a-competitive if and only if it differs from the theoretically optimal time by at most a factor of a *for all sequences of operations*. For example, consider the numbers $1, 2, 3, \ldots, n$. Suppose these n values are the only ones in the tree, and I wish to search for all n, in order. There exists an offline binary search tree which can perform these n searches in amortized $O(n)$ time (think about a BST which

rotates the queried element to the top of the tree; or alternatively, one that maintains the sequential access property). Suppose, on the other hand, that I receive my searches one at a time—in an online manner—I have no notion of what my next query will be. In this case, with no knowledge of my future operations, I might build standard red-black tree, which will be $\log(n)$ competitive—it can perform each search in $\log(n)$ time (and will therefore take $O(n \log(n))$ time to process the sequence $1, 2, 3, \ldots, n$, slower than optimal by a factor of $\log(n)$).

Sure, in the worst case, I can find a sequence of inputs that will take all BSTs $\log(n)$ search time per element, but *dynamic optimality* considers more than just the worst case—dynamic optimality is concerned with all cases, all sequences, all inputs, and our performance relative to the theoretically optimal BST.

2.2 Binary Search Tree Properties

As we covered in lecture, there are four main properties that a BST can satisfy:

- Balance: All elements are equally important.
- Entropy: Minimize Shannon entropy by storing more frequently queried elements higher.
- Dynamic finger: Access time scales by the log of the distance from last accessed key.
- Working set: Access time for x_i time scales by the log of the number of keys accessed since last time x_i accessed.

However, there are also a few more properties worth keeping in mind.

- Sequential Access property: Accessing $1, 2, \ldots, n$ in sequential order takes $O(1)$ amortized time per operation (trivially, an inorder traversal of a BST satisfies this).
- Unified property: Access time between x_i and x_j scales with the log of the distance between x_i and x_j plus the number of keys accessed between x_i and x_j. Trivially, this implies both working set and dynamic finger, because in general

$$\log(\text{Distance} + \text{Time}) < \log(\text{Distance} \cdot \text{Time}) = \log(\text{Distance}) + \log(\text{Time})$$

  and $\log(\text{Distance}) = \text{Dynamic Finger}$ and $\log(\text{Time}) = \text{Working Set}$.

The splay tree satisfies the first four properties above (along with, of course, sequential access), but it remains a conjecture that the splay tree (and any BST for that matter) satisfies the unified property, which is stronger than both working-set and dynamic finger.

Any binary search tree that satisfies the balanced property is $O(\log(n))$ competitive, since every operation takes at most $O(\log(n))$ time, and amortizing these offline over a sequence

of operations reduces each operation to minimum $O(1)$ time. The question becomes: *does there exist an $O(1)$-competitive binary search tree?*

The *Dynamic Optimality Conjecture*, posed by Sleator and Tarjan in 1985 [3], theorizes that a splay tree is $O(1)$-competitive. But, to this day, the conjecture remains unproven. The first real progress towards the Dynamic Optimality Conjecture came in 2007 with Tango trees and their $O(\log \log n)$ competitive ratio.

3 Binary Search Tree Geometry

At this point, I'm sure you're thinking: *OK, there's this notion of competitiveness, which is defined as a ratio of the optimal binary search tree. But computing the optimal binary search tree is hard, so how can we know how competitive a Tango Tree is (or any BST, for that matter) if we don't know how to calculate OPT(X)?*

Well, it turns out we can find a lower bound on the number of operations an optimal BST must take by completely rethinking how we look at binary search trees.

3.1 Geometric Interpretation

There is a very clever bijection between valid binary search trees and geometric point sets called *arborally satisfied sets*.

We define an *arborally satisfied set* (ASS) to be a subset of the lattice points in the first quadrant such that any rectangle spanned by two points in the set with positive area must contain another point in the set (in its interior or boundary).

We can construct a point set for each BST execution as follows. Suppose we have a BST storing the elements $1, 2, \ldots, n$, and we query the elements $x_1, x_2, \ldots, x_k$ in that order. For each query, we add a row to the point set starting from $y = 1$ and working upwards. We mark a dot at each node that we visit in the query, whether we are simply traversing through or performing a tree rotation. It turns out that this point set is precisely an arborally satisfied set. For example, if $n = 4$ and our BST was as depicted in Figure 1, we could construct the corresponding arborally satisfied set for the search queries $3, 1, 4, 2$.

Theorem 3.1. *There is a bijection from the point sets of BST executions to arborally satisfied sets.*

Proof. This proof is long, tedious, and technical, so we won't reproduce it here. The outline of the BST $\rightarrow$ ASS direction is to consider a rectangle, and then take a lowest common ancestor and prove that if they do not lie inside the rectangle then various tree rotations

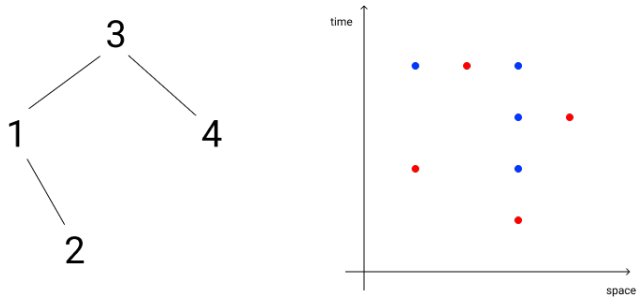

Figure 1: An example of an Arborally Satisfied Set. Red dots are the queried nodes and blue dots are the nodes in the BST that we visit (traverse/rotate) while executing.

must have happened, leading to other points being forced to lie in our rectangle. The proof of the other direction involves an algorithm to construct a treap that satisfies a given ASS. □

In addition, Demaine et al. have shown that an online algorithm to construct an arborally satisfied set given an access sequence can be mapped to the execution of an online BST, with an $O(1)$ slowdown. The proof is similar to that of the bijection above, using a treap-based construction.

Now that we've established this bijection, we can bound the total number of operations required by the optimal BST by the minimal number of points in an arborally satisfied set that covers a set of queries.

3.2 Greedy Algorithm

To get an upper bound on the number of operations required by OPT, we need to be able to construct an ASS from a series of accesses. We use the following Greedy algorithm to do so:

1. Start with a blank plot with time as the y-axis and space as the x-axis and an empty point set P.
2. Iterate over each time step from least to greatest (starting at the time of the first access and increasing until the time of the last access). For each $y = t$:
   (a) If an access is made at time t, plot the point (x_t, t) and add it to P. Else continue to the next time step.
   (b) If the point set so far P is not arborally satisfied, add as few points as possible on the line $y = t$ to make the set arborally satisfied by doing the following.

   If some point (x_t, t) on $y = t$ exists such that it forms a rectangle R with some other point $(a, b) \in P$ where $b < t$ and where no other point in P is inside R, then simply add the point (a, t) to P.

The Greedy algorithm is an online one, because it works on inputs given one at a time instead of getting all the inputs at once. As we learned in the previous section, the arborally satisfied point set generated by the greedy algorithm is in fact a binary search tree. This BST can, with each access, rearrange itself for future searches. This means that the greedy algorithm is actually an online BST algorithm. Also, Greedy and provides an upper bound on the optimal BST, as proven by Demaine [4].

3.3 Preliminary Lower Bounds

We will now further investigate the geometry of arborally satisfied sets. This will allow us to bound the performance of the greedy algorithm, and also develop a lower bound on the optimal number of points needed to build an arborally satisfied set for an input access sequence.

Define two rectangles to be *independent* if they are formed by pairs of points in the input set (red points, corresponding to the BST accesses), and no corner of either rectangle is *strictly* inside the other. (See Figure 2.)

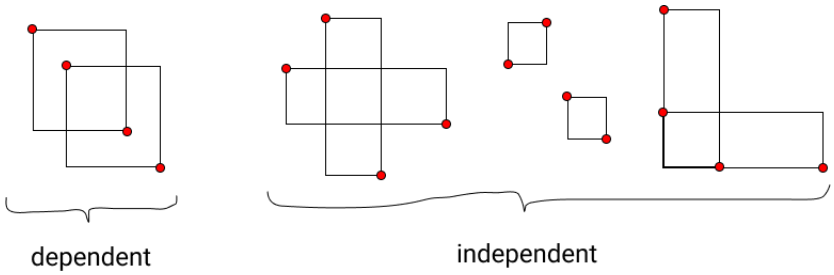

Figure 2: Examples of dependent and independent rectangles.

We now introduce the concept of *signed rectangles*: define ◪-*rectangles* as rectangles formed by two points at the upper right and lower left (i.e. *slash-rectangles*), and similarly define ◹-*rectangles* as rectangles formed by two points at the lower right and upper left (i.e. *backslash-rectangles*).

We call a superset of the original access points ◪-satisfied if all ◪-rectangles have another point inside them, and ◹-satisfied if all ◹-rectangles have another point inside them. An arborally satisfied set is both ◪-satisfied and ◹-satisfied.

Lemma 3.1. The minimal number of points in a ◪-satisfied superset, $OPT_{◪}$, is bounded below as

$$OPT_{◪} \geq k + (\max \#\text{ of independent } ◪\text{-rectangles}) .$$

(Recall that k is the number of red points, i.e. the number of accesses in the BST execution.)

Proof. This proof is fairly long and not critically important for a decent understanding of the landscape, so we will not prove it here. Intuitively, the smallest possible arborally satisfied point set (with OPT points) is a superset of the input (or access) point set (which has k points). The number of independent ◪-rectangles in this access point set will each require at least one additional point to be added somewhere in them in order for the overall point set to become arborally satisfied. For now, we are not counting the points that would need to be added within ◪-rectangles to reach the optimal ASS, so it is clear the size of the optimal ASS must be at least as large as k + (max # of independent ◪-rectangles). □

Because all independent rectangles are either ◪-rectangles or ◧-rectangles, this Lemma implies the following theorem:

Theorem 3.2. *(Independent Rectangle Bound.)*

$$OPT \geq k + \frac{1}{2}\left(\text{max \# of independent rectangles}\right).$$

3.4 Signed Greedy Algorithm

We now introduce the ◪-Greedy or Signed-Greedy algorithm:

1. Sweep the access point set in the same way that the Greedy algorithm does.
2. For each access at time t_i, add points on line $y = t_i$ in the same manner as in Greedy, but only to satisfy ◪-rectangles.

For every added point, we introduce a new independent ◪-rectangle. So the point set $S_{◪}$ produced by ◪-Greedy will have at least $|S_{◪}|$ independent ◪-rectangles.

◧-Greedy is defined analogously. Let the union of the supersets produced by ◪-Greedy and ◧-Greedy on an access sequence be called $S_{⊠}$. Unlike Greedy, $OPT_{⊠}$ is not a valid BST (for example, in satisfying a ◧-rectangle, a ◪-rectangle may have become unsatisfied, and therefore the overall point set is not arborally satisfied). However, $S_{⊠}$ does provide us with a lower bound on OPT: additional points may have to be added to $S_{⊠}$ to make it arborally satisfied and, equivalently, correspond to valid binary search tree.

It turns out that looking at $S_{⊠}$ provides good lower bound for OPT, by the following theorem.

Theorem 3.3. *Let $S_{◪}$ be the set of rectangles formed by ◪-Greedy, and similarly define $S_{◧}$. Then*

$$\max(|S_{◪}, S_{◧}|) = \Theta(\textit{biggest independent-rectangle lower bound}).$$

Proof. Let $OPT_{\boxtimes}$ be the smallest union of a $\boxslash$-satisfying superset and a $\boxbslash$-satisfying superset of the original access points. Then by Lemma 3.1, we have

$$OPT_{\boxtimes} \geq k + \frac{1}{2}(\text{max \# of independent rectangles}),$$

where k is the size of the input access sequence. In turn, we have

$$\frac{1}{2}(\text{max \# of independent rectangles}) \geq \frac{1}{2}\max(|S_{\boxslash}|, |S_{\boxbslash}|).$$

But each of $|S_{\boxslash}|$ and $|S_{\boxbslash}|$, as explicit constructions, are greater than $OPT_{\boxslash}$ and $OPT_{\boxbslash}$, respectively. Since the maximum of two quantities is greater than their average, we can continue the inequality chain with

$$\frac{1}{2}\max(|S_{\boxslash}|, |S_{\boxbslash}|) \geq \frac{1}{4}(OPT_{\boxslash} + OPT_{\boxbslash}) \geq \frac{1}{4}OPT_{\boxtimes}.$$

Hence we've created a constant factor sandwich on either side of the inequality, so all these expressions must be within a constant factor of each other. In particular, the maximum number of independent rectangles must be within a constant factor of $\max(|S_{\boxslash}|, |S_{\boxbslash}|)$, proving the theorem. □

This proves that Signed Greedy gives us a lower bound for OPT. We now have both a lower and upper bound for OPT, both based on variants of a greedy algorithm – perhaps we can compare them.

3.5 Exploration of Greedy vs. Signed Greedy

The big question is whether Greedy (an upper bound on OPT) is within a constant factor of $OPT_{\boxtimes}$, generated from Signed Greedy (a lower bound on OPT). The two algorithms are very similar; the only difference is that Greedy satisfies both $\boxslash$-rectangles and $\boxbslash$-rectangles at the same time, while we satisfy the $\boxslash$and $\boxbslash$rectangles separately in two calls to Signed Greedy, and then take the union of the sets to compute $OPT_{\boxtimes}$. It is conjectured, but unproven, that these two differ by just a constant factor.

Conjecture 3.1 (G = SG Conjecture). *The sizes of the point sets generated by the Greedy and Signed Greedy algorithms differ by a constant factor across all access sequences.*

Erik Demaine, one of the fathers of dynamic optimality and the inventor of tango trees, has proposed this formulation of the dynamic optimality problem, and has said that a solution would be a breakthrough in the field.

To investigate, we implemented the Greedy and Signed Greedy algorithms for arbitrary access sequences. Then, for each $n = 1, \ldots, 13$, we generated a number of random access

sequence of length 2^n and fed it to the Greedy, ◪-Greedy, and ◹-Greedy algorithms. $OPT_{\boxtimes}$ was generated by taking the union of the point sets returned by the later two algorithms. This process was repeated several times for each n. Then we plotted the average ratio between the size of the point set generated by Greedy and $OPT_{\boxtimes}$.

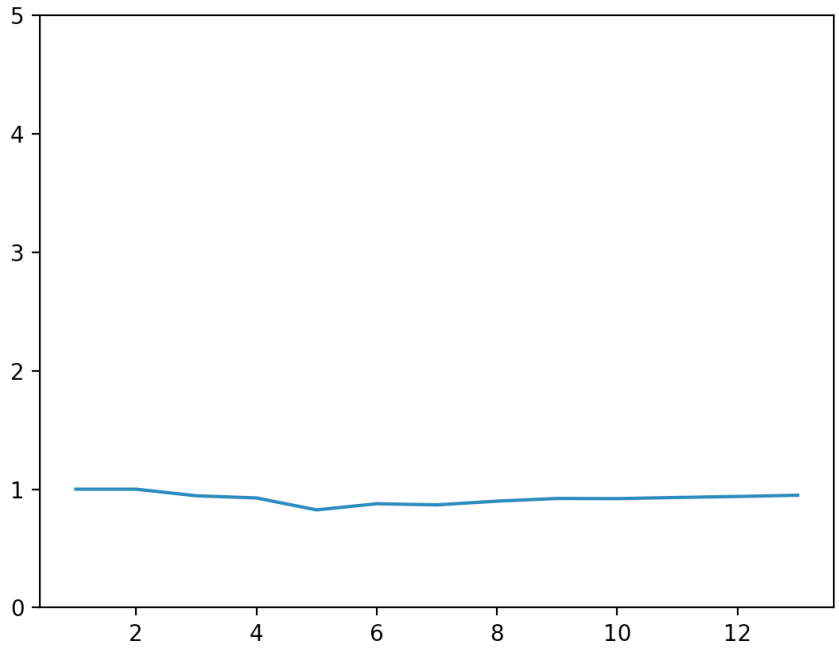

Figure 3: Length of access sequence (lg-scale) on the y-axis, versus the ratio $|Greedy| / \mathrm{OPT}_{\boxtimes}$ on the x-axis.

From Figure 3, we see that as the size of the access sequence grows, the ratio between Greedy and $OPT_{\boxtimes}$ seems to remain constant. With this empirical evidence that Greedy and Signed Greedy differ by only a constant factor for a wide variety of access sequences, we can say with more confidence that Greedy is a tight bound on OPT.

This is important, because it supports the idea that Greedy is in fact a dynamically optimal online BST, rather than just an upper bound for one. If true, this would be the answer to the biggest open question in the field of Dynamic Optimality: does a dynamically optimal structure exist? For now, we contribute compelling evidence that it probably does.

4 Interleave Bound

The interleave bound is another lower bound on the number of operations by an optimal BST, like the independent rectangle bounds. While it is unknown whether greedily generated

arborally satisfied sets are $O(1)$-competitive, we can use the interleave bound to build a data structure whose competitiveness we can quantify. Tango trees are an elegant translation of the structure of the interleave bound into a viable, working data structure.

4.1 Intuition

The interleave bound $\text{IB}(X)$ is a valid lower bound for the work done by the optimal BST P given an access sequence X. Intuitively, this is because every time a node y's label is changed (an alternation that is counted by the interleave bound) during the access sequence, y has to have been touched—the only way its label could be changed is, by definition, if the key that is being accessed is a descendent of y. So, the interleave bound gives us a count of some of the times a number of nodes has been touched during an access sequence. This set of touches is a subset of the total number of touches (a node y can be touched during an access x_i, but its label is left unchanged if x_i exists in the same subtree that y's label specifies). Because this total number of touches is, in fact, proportional to the total amount of work the BST P does during an access sequence X, the size of our subset of the total touches is a (loose) lower bound on the total amount of work.

4.2 Formal Discussion

We now present a formal discussion of the interleave bound. For a static binary tree P with n nodes given an access sequence X, for each access x_i, we note whether x_i is in the left or right subtree of each node. Within each node, we keep track of the number of alternations between accesses to the left and right subtrees, called *interleaves*. The total number of interleaves across all nodes in P is denoted $\text{IB}(x)$.

It turns out that the total number of interleaves provides a lower bound for the work done by any binary search tree on a particular access sequence.

Suppose we maintain a perfect binary search tree P on the keys $\{1, 2, \ldots, n\}$. As we carry out our sequence of queries $X = (x_1, x_2, \ldots, x_n)$, the structure of P will remain fixed. For a node y in P, we define the *left region* of y as the set of y along with all the nodes in its left subtree. Then define the *right region* of y as the set of the nodes in its right subtree (not including y), so that the subtree of P rooted at y is perfectly partitioned into y's left and right regions. Now for each node y, we label each access $x_i \in X$ with either an L or an R for whether x_i lies in the left or right region of y, respectively. (If x_i does not lie in the subtree of P rooted at y, we simply discard it.)

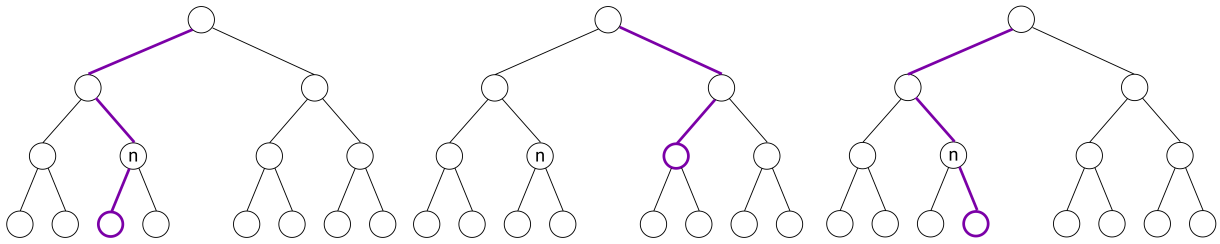

Figure 4: Example of three accesses. In each image, the purple node is the accessed node, and purple edges are the edges touched by the access. The access on the left is an L-access, the access in the middle is neither L or R, and the access on the right is an R-access.

In this list of L's and R's, we now count the number of alternations from L to R and vice-versa; these are called *interleaves* through y. For example, the list $LLLRRLRL$ has four interleaves, while $RRRRR$ has none.

Now suppose that we have a fixed but arbitrary search algorithm on our BST T. Let the state of T after executing access x_i be T_i. Define the *transition point* for y at time i to be the node in T_i satisfying these two properties:

1. The path from z to the root of T_i contains a node from both the left region of y a node from the right region of y, and
2. z is the minimum-depth node in T_i such that this holds.

We will eventually prove that interleaves are linked to touching transition points, and by proving some initial results about transition points we can ultimately obtain the bound we desire with interleaves. Essentially, the notion of transition points allows us to bridge the gap between interleaves (swapping between the left and right regions of a node) and the minimum number of points in the BST that we touch while accessing (which we bound by the number of transition points that we touch). With this high-level road map in mind, let us develop a better understanding of these transition points.

Lemma 4.1. *The transition point not only always exists, but is always unique.*

Proof. Note that the lowest common ancestor of two nodes in a BST must have a key value in between those two nodes. Hence the lowest common ancestor of two nodes in T_i that are in the left region of y must lie in between the keys of those two nodes, and thus will be another node in the left region of y. So if we let ℓ be the the lowest common ancestor of all the nodes in T_i that are in the left region of y, then ℓ will also be in the left region of y. Hence ℓ is the unique node of minimum depth in T_i of all nodes in the left region of y; if it weren't unique, then there would be another node at the same depth, and we could take the lowest common ancestor of ℓ with this node to contradict the minimality of ℓ.

Similarly, define r to be the lowest common ancestor of all nodes in T_i in the right region

of y; then r is the unique node in the right region of y with minimum depth in T_i. Also, the lowest common ancestor in T_i of all nodes in the left and right regions of y must lie in either the left or right region, since they are consecutive in keyspace. Hence it must be either ℓ or r, which are each shallowest nodes in their respective regions. Suppose without loss of generality that it is ℓ, and thus ℓ is an ancestor of r. It follows that r is a transition point for y in T_i. Indeed, the path from r to the root contains both a node in the right region of y (r) and a node in the left region (ℓ). Furthermore, r is the unique transition point because any path in T_i from the root to r must pass through ℓ before any other node in the left region, and then r before any other node in the right region. Hence r is the unique transition point for y in T_i. □

Now that we have established that the transition point is well-defined, we will show that it is stable, i.e. it does not change until it is accessed:

Lemma 4.2. *Let z be the transition point of a node y at time j, and suppose that the BST access algorithm does not touch z for all i in the time interval $[j, k]$. Then z remains the transition point for node y during the entire time interval $[j, k]$.*

Proof. Define ℓ and r as in the proof of the last lemma, and again without loss of generality suppose that ℓ is an ancestor of r. Because the BST access algorithm doesn't touch r, it doesn't touch any of the nodes in the right region of y. So r will indeed remain the lowest common ancestor of those nodes.

The nodes in the left region aren't so straightforward. The algorithm could still touch nodes in the left region of y without touching r; even ℓ could change. Let ℓ_i denote the new lowest common ancestor of all the nodes in the left region of y. We claim that even if $\ell_i \neq \ell$, ℓ_i is still an ancestor of r. To see why, note that nodes in the left region of y can't newly enter r's subtree in T_i, because that entire subtree remains unchanged. So some node ℓ_i' in the left region of y must remain outside the subtree of r in T_i. So the lowest common ancestor of ℓ_i' and r must also be outside r, which means that it is in the left region of y and that ℓ_i is its ancestor. Thus by transitivity ℓ_i is also r's ancestor, and so we are done. □

Our last building block is to prove that these transition points are different over all nodes in P.

Lemma 4.3. *At any time i, no node in T_i is the transition point for multiple nodes in P.*

Proof. Consider two distinct nodes y_1 and y_2 in P, and define the corresponding ℓ_1, ℓ_2, r_1, and r_2 correspondingly. The transition point for y_1 is either ℓ_1 or r_1, and the same for y_2. If y_1 and y_2 are not ancestrally related in P, then their left and right regions are disjoint from each other, so ℓ_1 and r_1 are distinct from ℓ_2 and r_2. So in this case the transition points are indeed distinct. On the other hand, suppose that one of y_1, y_2 is an ancestor of the other;

without loss of generality, say y_1 is an ancestor of y_2 (in P). Then if the transition point for y_1 is not in the same side of y_1 in P as y_2, it must be different from ℓ_2 and r_2 and thus the transition point for y_2. The only remaining case is if the transition point for y_1 is the lowest common ancestor of all nodes of y_2's subtree in P. Hence it is either ℓ_2 or r_2, which ever is *less deep*. But the transition point for y_2 is ℓ_2 or r_2, whichever is *more deep*. It follows that the transition points of y_1 and y_2 differ in all cases. □

Finally, we are ready to tackle the main theorem.

Theorem 4.1 (Interleave Bound Theorem.). Suppose we have a BST P with n nodes, which remain fixed during querying. We will show that the time it takes to perform a series of accesses $X = (x_1, x_2, \ldots, x_n)$ on P is at least $\mathrm{OPT}(X) \geq \frac{\mathrm{IB}(X)}{2} - n$. This number is called the *interleave bound*.

Proof. We will lower bound the cost of the optimal offline BST by counting the number of transition points it touches. By Lemma 4.3, rather than count access by access aggregating over nodes, we can instead count node by node and aggregate over accesses. Let us count the total number of times the transition point of a node y is touched. Define ℓ and r as usual, so the transition point of y is either ℓ or r, whichever is deeper. Suppose we pick an ordered subsequence $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$ of maximal length of accesses to nodes that alternate between being in the left and right regions of y. Then p is the amount of interleaving through y. Suppose without loss of generality that the odd accesses $x_{i_{2j-1}}$ are nodes in the left region of y, and the even accesses are nodes in the right region of y. Any access to a node in the left region of y must touch ℓ, and any access to a node in the right region of y must touch r. So for each j with $1 \leq j \leq \lfloor p/2 \rfloor$, in order for both accesses $x_{i_{2j-1}}$ and $x_{i_{2j}}$ to avoid touching the transition point for y, the transition point must change from r to ℓ in between. But the only way this can happen is if we actually touch the transition point for y. So the BST access algorithm must touch the transition point at least once during the time interval $[i_{2j-1}, i_{2j}]$. There are $\lfloor p/2 \rfloor$ such intervals, so the BST access algorithm must touch the transition point for y at least $\lfloor p/2 \rfloor \geq p/2 - 1$ times. Summing over all y gives the interleave bound $\frac{\mathrm{IB}(X)}{2} - n$, as desired. □

5 Tango Trees

Despite all the complexity of independent rectangles and space-time representations of BSTs, the structure of a tango tree is refreshingly simple. We define a *preferred child* of a node x to be the left child if the left subtree of x has been accessed (touched) more recently, or the right child if the right subtree of x has been touched more recently. As a base case, the preferred child is "none" if no access has occurred in either subtree yet.

Preferred paths form chains throughout the tree, and we can group nodes into these chains, and store each chain as a red-black tree itself, with pointers between them falling

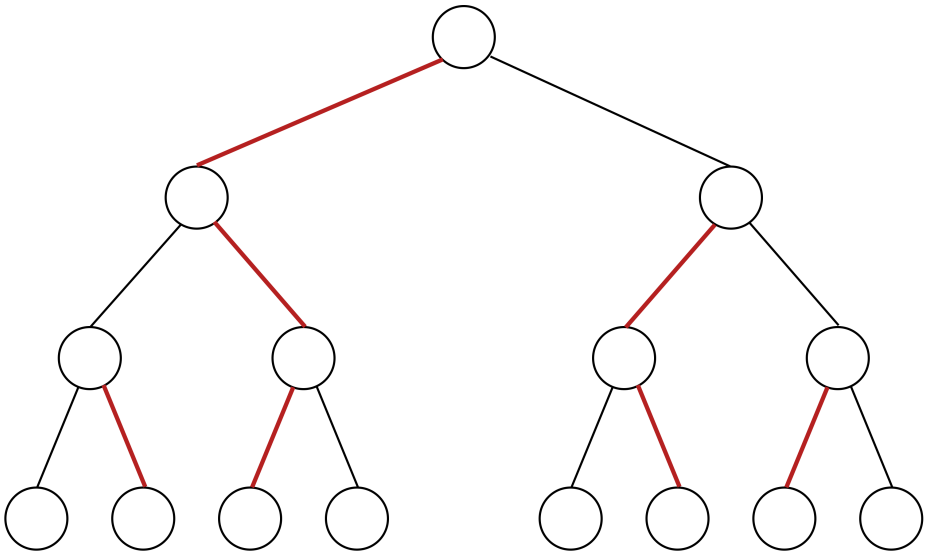

Figure 5: A Tango tree. Red edges mark "preferred paths".

from tree to tree (see Figure 6 below). Since each chain has a height of at most $\log(n)$, each auxiliary red-black tree has a height of $O(\log \log(n))$.

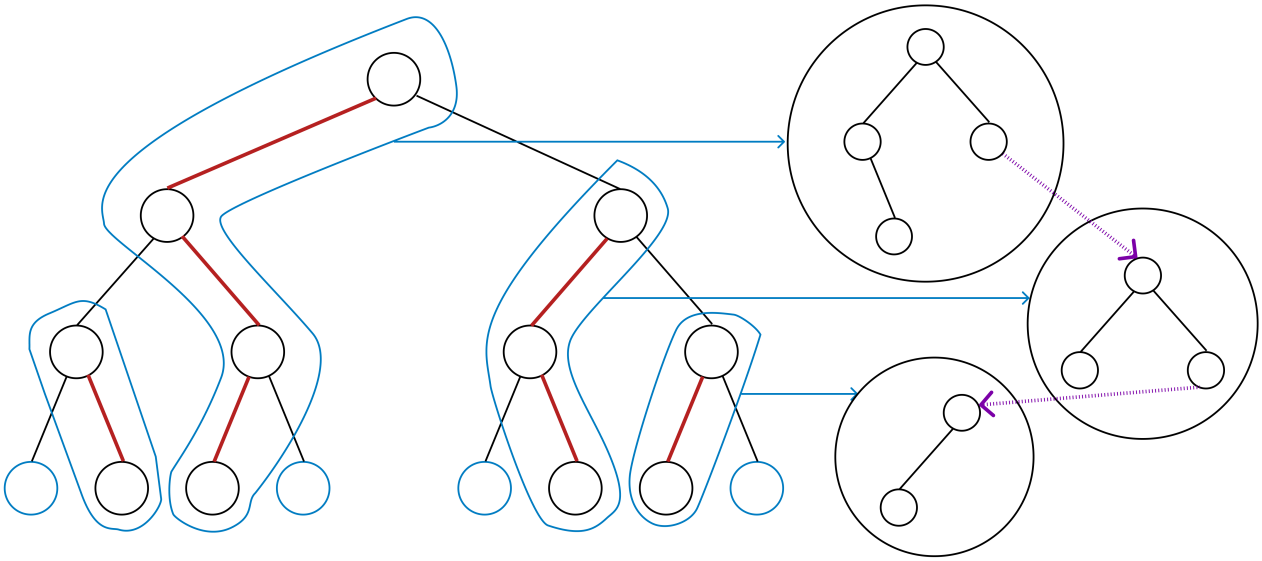

Figure 6: The Red-black tree representation of preferred chains. Each chain is a red-black tree, with pointers between red-black trees (purple arrows) representing non-preferred edges.

To search for a node, we start at the top-most auxiliary red-black tree (the one containing

the root) and descend down. We make k jumps across auxiliary trees (non-preferred edges), resulting in $kO(\log \log n)$ operations for a search.

Since preferred edges represent the nodes touched most recently, after every search, we need to update preferred edges. Changing a preferred child is akin to cutting one path and joining two others. This is akin to a constant number of split and join operations on two red-black trees (since each preferred path is a red-black tree). Since each red-black tree has $O(\log(n))$ nodes, and the split and join operations take log(Height) for a red-black tree, changing a preferred child takes time $O(\log(\log(n)))$.

Accounting for updating preferred edges, searching for an element using a Tango Tree takes time $(k+1)O(\log \log n)$.

5.1 $O(\log \log n)$-Competitive

Nothing about the structure of the Tango tree indicates it's anything special—so then what makes the Tango tree $O(\log \log n)$ competitive?

Note that a search on a Tango tree takes time $(k+1)O(\log \log(n))$. So, if a Tango tree is $\log \log(n)$ competitive, the optimal binary search tree must take at least $O(k)$. It turns out, this follows pretty naturally from the interleave lower bound.

On a search for a node in a tango tree, every time we take a non-preferred edge, we hop from one red-black tree to another red-black tree. This means that whenever we hop between red-black trees, we are changing which subtree of a node was accessed last. Thus, the number of hops between red-black trees, k, is precisely the number of subtree changes. The number of changes between subtrees is exactly the change in the interleave bound between two accesses.

So, over an entire access sequence X, the number of preferred edges changed is the total number of subtree changes, which is exactly the interleave bound and at most $\mathrm{OPT}(X)$. Therefore, the total cost of the access sequence is at worst $\mathrm{OPT}(X)(O(\log \log n))$ (plus some constant factors). Thus, the tango tree is $O(\log \log n)$-competitive.

We disregard lower factors and small details regarding differences between the interleave bound and the tango tree, but the general idea should be clear—the tango tree is a direct translation of the interleave bound into a data structure. Changing between two preferred chains is akin to increasing the interleave bound by one.

5.2 Exploration beyond Tango Trees

How tight is the bound for Tango trees? Could we make a Tango tree better than $\Theta(\log \log n)$-competitive?

The fundamental idea behind a tango tree is to start with a balanced binary search tree, note the "preferred children", and then turn each of the "preferred children paths" into a red-black tree. This is where the $\log \log n$ factor arises—a binary search across a red-black tree which has at most $O(\log n)$ elements, since the length of a "preferred path" is at most the height of the original tree.

The Multi-Splay Tree [5], replaced these "preferred path" red-black trees with splay trees for better bounds on the auxiliary tree operations. In particular, the authors of the Multi-Splay Tree proved that the search time was improved from $O(\log n \log \log n)$ to amortized $O(\log n)$. Building off this idea, we propose a modification of the tango tree which replaces these "preferred path" red-black trees with tango trees.

However, our *multi-tango* tree, or *mango* tree for short, has no better bounds than a traditional tango tree.

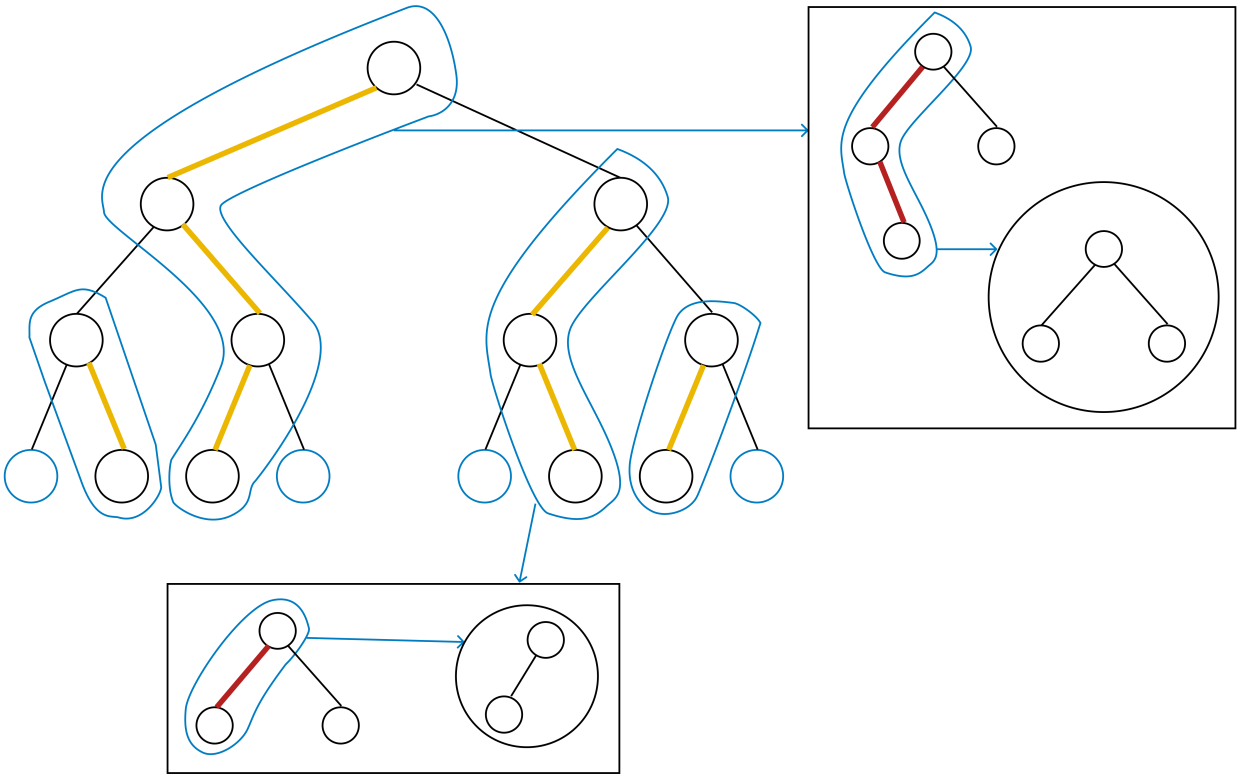

Figure 7: The Mango Tree. Preferred chains in the mango tree, which are implemented as auxiliary tango trees, are marked in yellow. The preferred chains of the auxiliary tango trees (marked in red) are implemented as red-black trees, as before.

Consider one of the auxiliary tango trees of the mango tree. Since a tango tree is $O(\log \log n)$-competitive, there exists a sequence for which a tango tree takes $O(\log \log n)$

times worse than a standard balanced BST (think about a search that maximizes the number of hops between auxiliary trees). In worst case sequences, Bose et al. [6] claim, the tango tree can actually take $\Theta(\log n \log\log n)$ time.

However, since there are $O(\log n)$ nodes in each of the auxiliary tango trees of a mango tree, these auxiliary tango trees can take worst-case time of $O(\log\log n \log\log\log n)$. Since we can take a maximum of $O(\log n)$ jumps between auxiliary trees, a mango tree search that maximizes the hops between auxiliary tango trees should take $O(\log n \log\log n \log\log\log n)$ time. Since a balanced BST has worst-case $O(\log n)$ search time, the mango tree should be $O(\log\log n \log\log\log n)$-competitive. Certainly, because the mango tree is bounded by the tango tree's competitiveness, the mango tree is no better than $O(\log\log n)$-competitive.

5.3 Tightness

So the mango tree can't possibly beat the tango tree's $O(\log\log n)$-competitiveness. A natural question to ask is then: Does there exist *any* way to modify the tango tree to improve it's competitiveness?

Sadly, the answer is no. Because the chain from the root node is of length $\log(n+1)$, no matter how we create the Binary Search Tree for these nodes, we must have $\Omega(\log\log(n))$ access time. In addition, every time we access a node along this chain, we change at most one preferred child from left to right. See Figure 8 for an example.

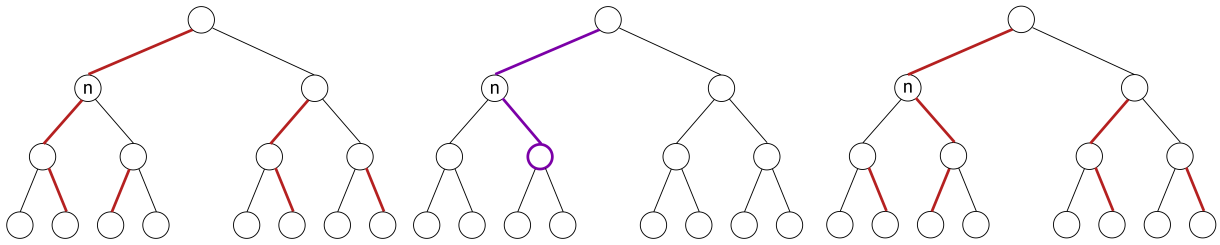

Figure 8: An example of the preferred chains of a Tango tree before (left image) and after (right image) an access (center image, shown in purple). At most one preferred edge along the chain from the root node is changed after an access to a node along the chain.

Every time an edge along this chain changes, the interleave bound increases by one. This means that every time we do $\Omega(\log\log n)$ work to binary search across the chain, we increase our lower bound for $\mathrm{OPT}(X)$ by at most one. So, as our tango tree operations grow by a factor of $\Omega(\log\log n)$, our lower bound for the fastest binary search tree increases by $O(1)$. Thus, using the interleave bound, the Tango tree is at best $\Theta(\log\log n)$-competitive. Fundamentally, *the Tango tree cannot be improved without better lower bound*—a tighter lower bound—and without one, we can't beat $O(\log\log n)$ competitiveness. The interleave bound simply won't cut it.

The question remains: Do there exist better bounds than the interleave bound? And how "tight" is the interleave lower bound, exactly? In fact, the interleave bound isn't tight at all—there exists sequences for which the interleave bound can be as far as $\Omega(\log \log n)$ smaller than the $\mathrm{OPT}(X)$. None of the bounds found so far are provably tight; finding and proving a tight bound remains a critical open question for solving the dynamic optimality conjecture.

6 Experimental Analysis

We know a Tango tree is $O(\log \log n)$-competitive—it's no worse than $\log \log n$ times the best binary search tree. It's also conjectured that the Splay tree is $O(1)$-competitive. If the Splay tree truly is $O(1)$-competitive, experimental results should tell us that if we select a set of $O(n)$ searches on a Tango Tree and a Splay Tree with n nodes, as $n \to \infty$, the lower bound on the number of Tango tree operations should grow at most a factor of $O(\log \log n)$ compared to the Splay tree. We couldn't find a set of data online comparing the two, so we did exactly that.

Using standard implementations of the trees, we initialized the trees with nodes from $\{1, \ldots, n\}$. Then, we performed $O(n)$ operations on each tree, and counted the number of operations. We chose not to use runtime as our metric because of hardware effects—caching, memory optimization, etc.—and instead simply added counters inside the `find` operations of the trees to get a lower bound on the number of rotations, edge traversals, and pointer changes. Since all algorithms have hidden constant factors not present in Big-O analysis, we ran a baseline for each algorithm of $n = 2^5$, and then for $n = \{2^6, 2^7, \ldots, 2^{22}\}$, we calculated the ratio, r, of the number of operations to the number of operations of $n = 2^5$. Lastly, for each n, we plotted $\frac{r_{\text{tango}}}{r_{\text{splay}}}$, so we could determine how fast one algorithm's operation count was growing relative to the other.

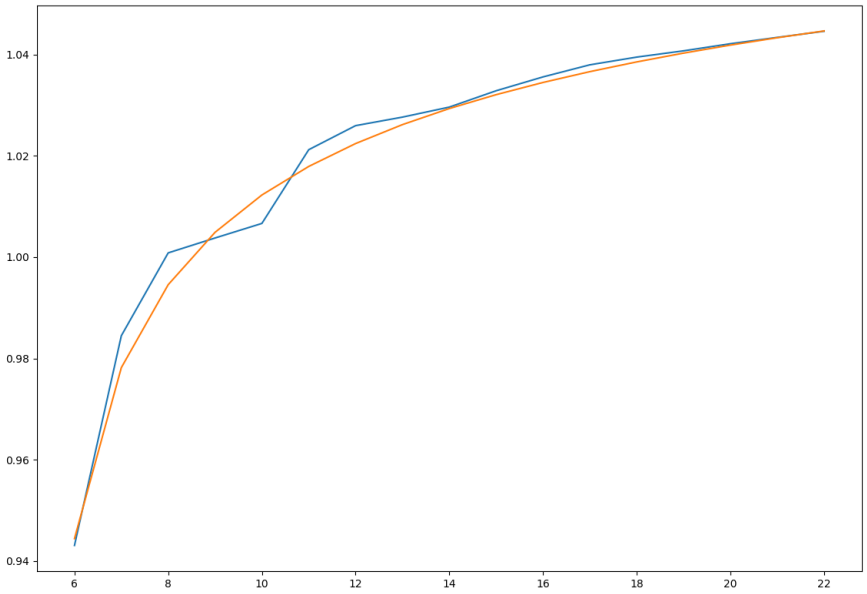

Figure 9: Length of access sequence (log-scale) vs. $\frac{r_{\text{tango}}}{r_{\text{splay}}}$.

The orange line is an $O(\log\log(n))$ curve fit to the set of data points. Since the plot is in log space, this is simply a shifted $O(\log(n))$ curve. This suggests that the Splay tree truly is dynamically optimal, since it seems to perform a factor of $O(\log\log n)$ better than the Tango tree.

7 Future Work

There are plenty of unsolved problems in the field of dynamic optimality. It seems the field is still a ways away from proving or disproving Sleator and Tarjan's *Dynamic Optimality Conjecture*, since we still have to wrestle with a few more basic conjectures regarding bounds:

- For all access sequences, does there exist a tree P such that $OPT = \Theta(\text{Wilber 1})$?
- Is $OPT = \Theta(\text{Wilber 2})$? (Wilber has a second bound which we do not present in this write-up, but which is based on alterations left and right across a vertical line passing through each point p [7].)
- Does there exist a tight lower bound? Can we define and prove one, or disprove the existence of one entirely?

The *Dynamic Optimality Conjecture*, although by no means a life-saving or time-critical problem, is still an important theoretical question nonetheless. If proven true, it begs the question: What is so special about splaying that makes it provably optimal? How are these sequences of rotations able to best every other BST algorithm? If the conjecture is disproven, we're stuck wondering if there exists $O(1)$-competitive BST at all, and if so, what type of rotation and movement might allow us to reach dynamic optimality.

8 Conclusion

In this paper, we explored dynamic optimality and many prominent lower bounds on optimal binary search trees. We presented a rigorous proof of the interleave lower bound before diving into Tango trees and proving their $O(\log \log n)$-competitive ratio using the bound.

In addition, we introduced the *mango tree*, which by all accounts performs worse than a traditional Tango tree, but is still an interesting thought experiment into the nature of the Tango tree. The recursive tango, it turns out, is slow precisely because Tango Trees themselves are slow in the worst-case. Perhaps, if Tango trees had a better worst-case runtime (like the Multi-Splay tree), a recursive strategy would be more viable).

Furthermore, we presented two experiments that support important conjectures in the field. First, experimental evidence that signed greedy is within a constant factor of greedy indicates there exists a tight bound for $\mathrm{OPT}(X)$, since signed greedy is a lower bound, and greedy is an upper bound. Although we first attempted to prove this theoretically, we found it intractable given our time constraints. Still, this experiment is fairly strong support towards the existence of a tight bound.

Second, we present experimental evidence that Splay Trees are indeed $O(1)$-competitive. By counting the number of Splay Tree operations and comparing the ratios appropriately, we were able to find that the Tango tree grows by a factor of $O(\log \log n)$ relative to the Splay Tree. However, given our limited computational resources, more trials should be run before drawing any further conclusions.

Ultimately, none of the experimental evidence we provide brings us any closer to a rigorous proof of the dynamic optimality conjecture or a tight bound. Even the interleave bound, for which there exists a sequences such that the bound is too low by a factor of $\log \log n$, is relatively tight for most sequences.

Dynamic Optimality is a relatively young and accessible field of theoretical computer science. Binary Search Trees, despite their ubiquity, still have an air of mystery around their true bounds. When we take a look at BSTs from an entirely different perspective—like access sequence geometry—an entire world of combinatorial geometry and algorithmic analysis collide to produce awe-inspiring results and beautiful proofs.

9 Acknowledgments

We would like to thank the CS166 teaching staff (Keith Schwarz, Anton de Leon, Ryan Smith, and Michael Zhu) for this wonderful class and the opportunity to pursue this topic for our final project. We are also thankful to Professor Erik Demaine and the MIT Open Courseware. We learned most of what we know about dynamic optimality and tango trees from Demaine's course 6.851, Advanced Data Structures [4]. Many of the technical proofs presented in this paper arose from our understanding of the material from a combination of Professor Demaine's lectures and the original papers.

References

[1] J. Iacono, "In pursuit of the dynamic optimality conjecture," *CoRR*, vol. abs/1306.0207, 2013.

[2] E. D. Demaine, D. Harmon, J. Iacono, and M. P a traşcu, "Dynamic optimality—almost," *SIAM Journal on Computing*, vol. 37, no. 1, pp. 240–251, 2007.

[3] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

[4] E. Demaine, "Advanced data structures," Spring 2012.

[5] C. C. Wang, J. Derryberry, and D. D. Sleator, "O (log log n)-competitive dynamic binary search trees," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pp. 374–383, Society for Industrial and Applied Mathematics, 2006.

[6] P. Bose, K. Douïeb, V. Dujmović, and R. Fagerberg, "An o (log log n)-competitive binary search tree with optimal worst-case access times," in *Scandinavian Workshop on Algorithm Theory*, pp. 38–49, Springer, 2010.

[7] R. Wilber, "Lower bounds for accessing binary search trees with rotations," *SIAM Journal on Computing*, vol. 18, no. 1, pp. 56–67, 1989.