# Dynamic Optimality, Geometry, and the Subsequence Property

Dylan Hendrickson, Andy Tockman

December 20, 2022

The question of *dynamic optimality* for binary search trees has been a long-standing one: informally, is there an online binary search tree algorithm that performs only a constant factor worse than the optimal algorithm which knows all queries in advance? A relatively recent approach (on the timeline of dynamic optimality) described by Demaine et al. [DHI+09] is geometric in nature, based on an equivalence of binary search tree algorithm executions and sets of points with a certain geometric property. In the geometric setting, there is a very natural algorithm called Greedy, whose corresponding binary search tree algorithm is conjectured to be dynamically optimal.

In this paper, we describe the transformation from the geometric setting to the binary search tree setting concretely – the current literature is not very clear on some important and non-obvious details. Then, we lay the groundwork for applying a new approach to dynamic optimality, called the "subsequence property" by Levy and Tarjan [LT19]. This approach was initially designed to apply to splay trees, which are the "traditional" path to dynamic optimality, However, as Levy and Tarjan say, "Many of the techniques developed in this paper should also be applicable to Greedy, but we do not focus on Greedy, and applying our results in the geometric model would likely constitute another investigation in its own right."

In Section 1, we describe the concepts and terminology needed for our results. Then in Section 2, we describe how to translate from the point sets to binary search trees in more detail than the current literature. Finally in Section 3, we describe Levy and Tarjan [LT19]'s result for Splay and prove a similar original result for Greedy.

# 1 Background

## 1.1 Binary search trees

Our **BST (binary search tree) model** of choice is as follows: A **search** starts with a pointer to the root, which can follow child and parent pointers and perform rotations arbitrarily. The only constraint is that the pointer must visit the target of the search. The *total* number of nodes visited, counted with multiplicity, is the **cost** of the search, which we seek to minimize.

We make this choice of model for simplicity – this model is constant-factor equivalent to any reasonable BST model. For example, one might desire to minimize the number of rotations performed, but it is possible to restructure any BST into any other BST on the same nodes in linearly many rotations [STT86]. So it is easiest to not worry about counting rotations and just track how many nodes are touched, trusting that the implementation will optimize its sequence of rotations appropriately.

## 1.2 Dynamic optimality

To define dynamic optimality formally, we need a bit of terminology.

A BST algorithm $\mathcal{A}$ takes as input an initial tree[1] and a **request sequence** $X$, which is simply an ordered list of keys that the algorithm must visit in order. The output of a BST algorithm consists of

[1]For most reasonable algorithms, the initial tree is irrelevant up to a linear additive factor. This is true of Greedy, so we are free to disregard it.

pointer-following and rotation operations, but as mentioned, we only care about the nodes touched. So we define the **access sequence** of $\mathcal{A}$ on $X$, denoted $\mathcal{A}(X)$, to be the list containing the set of nodes visited while handling each request in the algorithm's execution. Then the **cost** of $\mathcal{A}$ on $X$, denoted $\text{cost}_{\mathcal{A}}(X)$, is the sum of the sizes of these sets. Note that this is equal to the sum of the costs of each search, as defined above.

Given this setup, we now have a metric to compare BST algorithms with one another. For any request sequence $X$, let $\text{opt}(X)$ denote the minimum possible number of accesses to visit the nodes in $X$ in order. An online algorithm, which receives the request sequence one element at a time and must visit each element before seeing the next, is **dynamically optimal** if there is some constant $c$ such that $\text{cost}_{\mathcal{A}}(X) \leq c\,\text{opt}(X)$ for all access sequences $X$.

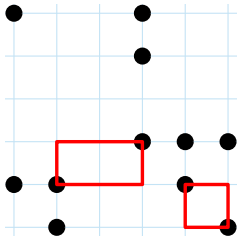
It is widely believed that Splay, the BST algorithm corresponding to the splay tree data structure, is dynamically optimal, and much of the work in dynamic optimality has been focused on this conjecture. We will instead focus on Greedy, a different algorithm which has been less thoroughly explored.

## 1.3 Arboral satisfaction

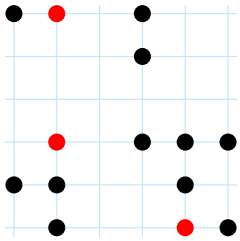
We now move on to describing Demaine et al. [DHI+09]'s geometric framework for dynamic optimality. We first define a problem seemingly unrelated to binary search trees, which we will later show turns out to be equivalent.

The fundamental geometric object we will consider is a **point set**, which is simply a set of pairs $(x, y)$ representing points in the plane. Each pair of points $p$ and $q$ with distinct $x$ and $y$ coordinates defines a **rectangle** $p \diamond q$, which has one corner at $p$ and one corner at $q$. (We also say that $p \diamond q$ is **spanned by** $p$ and $q$.) A rectangle is **arborally satisfied** if it contains, in its interior or on its border, any point in the point set other than $p$ and $q$. An **ASS (arborally satisfied set)** is a point set in which all rectangles $p \diamond q$ are arborally satisfied.

Here is a simple example of a point set. It is not arborally satisfied; the unsatisfied rectangles are drawn in red.



It is always possible to arborally satisfy a point set by adding some number of additional points to the set. This is the job of an **ASS algorithm**, which takes as input an arbitrary point set and outputs a superset which is arborally satisfied. Here is one possible way to satisfy the above point set, with the added points drawn in red.



Note that adding a point to satisfy a rectangle might create new unsatisfied rectangles: the topmost red point is needed to satisfy a rectangle that didn't exist in the original point set. Indeed, computing the minimal arborally satisfied superset of an arbitrary point set is NP-hard [DHI+09].

## 1.4 Greedy as an ASS algorithm

One particularly natural ASS algorithm is Greedy [DHI+09]. Greedy sweeps a line starting from the lowest $y$ coordinate and moving upwards, maintaining the invariant that there are no unsatisfied rectangles between two points which are both below the line.

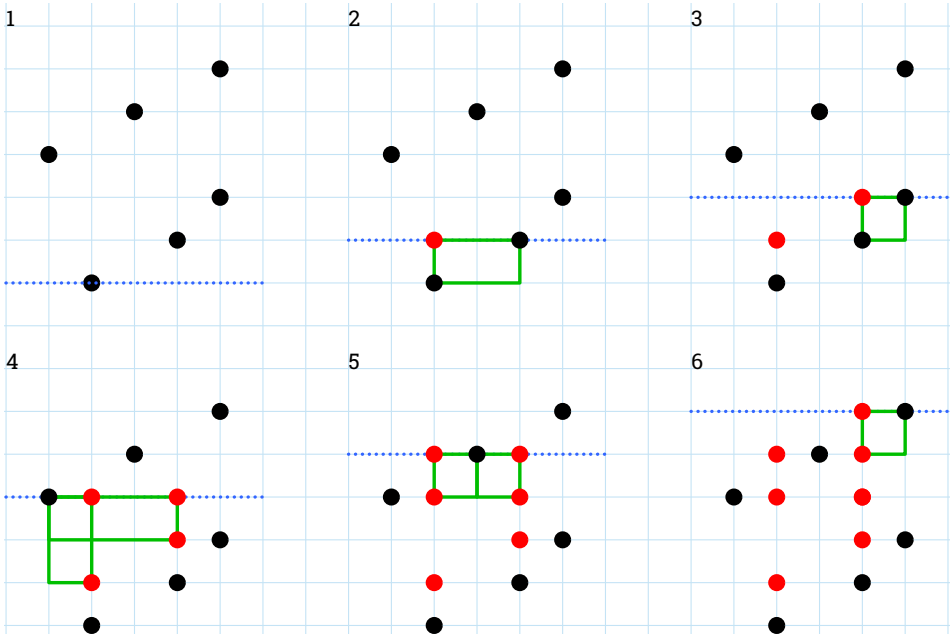



Figure 1: An example execution of the ASS algorithm Greedy. The original point set is drawn in black. The added points are drawn in red. The corresponding unsatisfied rectangles are drawn in green. The sweep line is drawn in blue.

The algorithm accomplishes this by only ever adding points on the sweep line. Specifically, every time the sweep line hits a new point $p$, Greedy simply iterates over all unsatisfied rectangles $p \diamond q$ where $q$ is below the sweep line, and adds a new point $p'$ on the corner of the rectangle which lies on the sweep line and is not occupied by $p$. The entire process is depicted on an example point set in Figure 1. Adding new points this way never increases the number of unsatisfied rectangles below the sweep line; this isn't too important for our discussion, but Figure 2 provides a short proof.

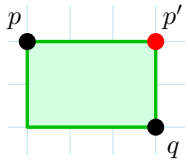



Figure 2: Point $p'$ was added to satisfy the rectangle $p \diamond q$. Suppose this made a new unsatisfied rectangle $p' \diamond q'$ below the sweep line. Where can $q'$ be? It can't be inside the green box, because the green box is empty by hypothesis. If it's left of the green box, then $p' \diamond q'$ is satisfied by $p$. If it's below the green box, then $p' \diamond q'$ is satisfied by $q$. If it's above the green box, then $p' \diamond q'$ is above the sweep line. If it's right of the green box, then $p \diamond q'$ was originally *also* unsatisfied, so this didn't increase the number of unsatisfied rectangles (i.e. Greedy was going to place a point in that corner anyway).

Note that Greedy can compute an arboral satisfaction in quadratic time, by keeping track of the height

of the "terrain" at each $x$ coordinate and making a single horizontal pass for each point the sweep line hits. Given a point set $X$ as input to Greedy, we denote the arborally satisfied superset it computes as Greedy$(X)$.

But what does any of this have to do with binary search trees? We now need to show that BST algorithms and ASS algorithms are equivalent up to constant factors.

## 1.5 Correspondence between request sequences and point sets

A BST algorithm takes as input a request sequence, and an ASS algorithm takes as input a point set. So if we want to claim they're equivalent, we had better be able to convert between one and the other.

To do this, we actually have to impose a minor constraint on the point sets: we consider only **y-generic** point sets, in which no two points have the same $y$ coordinate, as possible inputs.[2] Then, the mapping is remarkably simple: each point in a point set corresponds to a single request in a request sequence. The $x$ coordinate corresponds to the value of the key being requested, and the $y$ coordinate corresponds to the time it is requested. This is most easily explained by example, as in Figure 3.

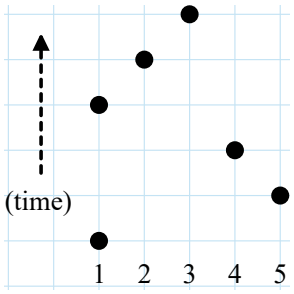



Figure 3: A depiction of the request sequence $\{1, 5, 4, 1, 2, 3\}$ in the geometric setting. By convention, time flows upwards.

A point set can be converted into a request sequence by reading off the $x$ coordinates in order, and a request sequence can be converted into a point set by placing a point at $(k, i)$ if the $i$th request is for key $k$. It is now clear why the $y$-generic condition is needed: you can't request two keys at the same time.

Given this mapping between *inputs*, we also need a mapping between *outputs*. To extract an arborally satisfied superset from the execution of a BST algorithm $\mathcal{A}$, we take the corresponding point set $X$ and add the point $(x', t)$ for every key $x'$ touched during the search at time $t$. Call the resulting set $\mathcal{A}[X]$.

**Lemma 1.1.** *For all BST algorithms $\mathcal{A}$ and point sets $X$, the point set $\mathcal{A}[X]$ is arborally satisfied.*

*Proof.* Consider an arbitrary rectangle $R = (i, t_1) \diamond (j, t_2)$, where $t_1 < t_2$. This means key $i$ was touched at time $t_1$, and key $j$ was touched later at time $t_2$. Let key $k$ be the lowest common ancestor of keys $i$ and $j$ at time $t_1$. Since $k$ is the lowest common ancestor, it lies in the closed interval between $i$ and $j$.

If $k$ is not equal to $i$, then it must have been touched at time $t_1$ in order to get to $i$, and the point $(k, t_1)$ satisfies the rectangle $R$.

Otherwise $k$ is equal to $i$, so at time $t_1$, $i$ was an ancestor of $j$. If $i$ is touched at time $t_1 < t' < t_2$, then the point $(j, t')$ satisfies $R$. Otherwise, $i$ remains an ancestor of $j$ at time $t_2$, so $i$ must be touched at time $t_2$ in order to get to $j$, and the point $(i, t_2)$ satisfies $R$.

In either case, $R$ is satisfied, completing the proof. □

Lemma 1.1 demonstrates that BST algorithms and AST algorithms are in fact closely related. However, to complete the equivalence, we also need a way of going from arborally satisfied sets to BST executions. This is much harder, and is the topic of the next section.

[2] One could define "$x$-generic" similarly, but we only care about $y$-generic point sets for the purposes of this paper.

## 2 ASS algorithms → BST algorithms

Demaine et al. [DHI+09] show that algorithms for arborally satisfied supersets can be converted into binary search tree algorithms. However, their proof is terse and glosses over many details. In this section, we unpack Demaine et al.'s proof and describe this transformation more thoroughly.

The transformation from the set of points touched by an ASS to the specific BST operations that make the algorithm work is complicated and involves lots of details. To provide some intuition, we first quickly summarize a method of conversion that requires seeing the whole point set at once. Of course, this does not give an online algorithm, but the online version (which we describe in the following subsection) essentially just simulates the offline version, using magic data structures that preserve the bounds.

### 2.1 Offline conversion

This section is intentionally very brief and informal; the results are not necessary for the actual online conversion, but they provide good intuition for what the online algorithm is trying to do.

We store the nodes as a treap: a binary search tree that is also heap-ordered by next-touch-time, with ties broken arbitrarily. It is easy to set up the initial tree this way, by starting with an empty tree and inserting each node in heap order (without any rebalancing). To do this, we need to know the next-touch-time of each node, which is only possible in the offline setting.

The reason this works is that at every timestep, we are now able to touch exactly the nodes that the point set claims are touched. Since the treap is heap-ordered by next-touch-time, the set $S$ of nodes that will be touched next form a connected subtree including the root, and we can reheapify them by new next-touch-time for free. (Recall that arbitrary restructuring can be done in linearly many BST operations.)

One might worry that the reheapify operation requires touching points outside $S$. Fortunately, this is impossible by arboral satisfaction. For the sake of contradiction, suppose some node $i$ was touched and a descendent $j$ was not, and $i$'s new next touch time is later than $j$'s. Since everything outside $S$ is unchanged and therefore still heap-ordered, if $j$ is not a direct child of $i$, we can either move up from $j$ or down from $i$ until it is. Now consider the yellow rectangle in Figure 4. There's no point on the left edge by assumption, and there's no point on the bottom edge because its key value would be between $i$ and $j$; therefore, reaching it would have required touching $j$ (since it can only be in $j$'s left subtree if $j$ is a direct child of $i$ and the tree is a BST). So by induction, $(i, \text{now})$ is the corner of some unsatisfied rectangle: any point satisfying $(i, \text{now}) \diamond (j, \text{next-touch}(j))$ would create a new unsatisfied rectangle.

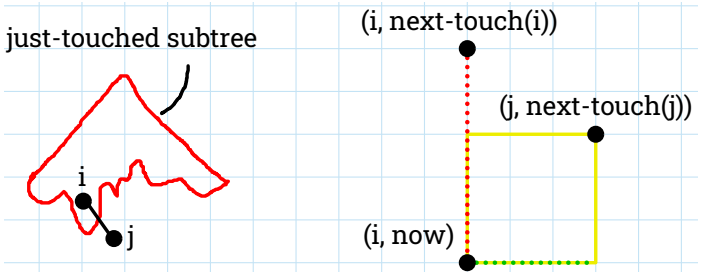



Figure 4: A heap-ordering violation across the boundary of $S$ would be a contradiction. If the dotted lines are empty, the yellow rectangle cannot be satisfied, since any point inside of it creates a new unsatisfied rectangle with $(i, now)$.

### 2.2 Split trees

In the online setting, we don't know the next-touch-time of any nodes. The tool that lets us simulate knowledge of next-touch-time anyway is a data structure known as a *split tree*.

A **split tree** is an abstract data structure with two operations:

- Make-Tree($S$), which constructs a split tree on the nodes given in $S$.
- Split($T, k$), which takes a split tree $T$ and a key $k$ and returns a pair $(A, B)$, where $A$ is a split tree of all keys less than $k$ and $B$ is a split tree of all keys greater than $k$.

After a Split, neither tree contains the key being split on, so the total number of keys decreases by 1.

One way to implement split trees with good bounds is just to use a red-black tree. Since rebalancing costs very little amortized, the amount it costs to split at a key is actually just the number of accesses it takes to find that key. Naively, this is $O(\log n)$. However, if we had pointers to the minimum and maximum nodes in the tree, we could start from the closer one, splitting into trees of size $n_1$ and $n_2$ in $O(\lg \min(n_1, n_2))$ accesses. With the potential function $\sum_T |T| - \lg |T|$, where the sum is over all split trees, potential decreases by

$$\begin{aligned}(n_1 + n_2 + 1 - \lg(n_1 + n_2 + 1)) - (n_1 - \lg(n_1)) - (n_2 - \lg(n_2)) &= 1 + \lg \frac{n_1 n_2}{n_1 + n_2 + 1} \\ &> 1 + \lg \frac{n_1 n_2}{n_1 + n_2} \\ &\geq 1 + \lg \frac{\min(n_1, n_2)^2}{2 \min(n_1, n_2)} \\ &= 1 - \lg 2 + \lg \min(n_1, n_2) \\ &= \lg \min(n_1, n_2),\end{aligned}$$

so split takes $O(1)$ accesses amortized.

The problem is that in the BST model, we can't store pointers to the minimum and maximum nodes. However, we can simulate them with a constant-factor overhead, by instead storing what is effectively a decision tree for the procedure that interleaves the steps of the "search from min" and "search from max" operations. Figure 5 depicts this transformation, which at most doubles the number of steps needed to find the key.

### 2.3 Online conversion

With split trees as a primitive, making the offline algorithm online is actually very straightforward. The idea is to do "lazy evaluation" – after every query, instead of reheapifying by next-touch-time (which is unknown), we throw everything we just touched into a split tree. That way, the next time we need to pull a key out of it, we can just split at that key at cost $O(1)$, and pretend that that key was at the root all along.

More explicitly, the BST is now a tree of "supernodes", which are either ordinary nodes or entire split trees. This can be implemented in practice by storing extra metadata on the nodes (like with red-black trees) indicating the virtual split tree structure. To perform a search, follow pointers as normal, but when you would be about to descend into a split tree, instead perform a split operation on the search key. This brings the search key directly to the root, and leaves all other nodes previously in the split tree as its children (in up to two smaller split trees). Then, after every access, instead of reheapifying, perform a Make-Tree on every node that was touched.

The Split operation is $O(1)$ amortized, and the Make-Tree operation doesn't touch anything that wasn't already touched, so the bounds are preserved. The next-touch-time heap property is effectively preserved: the next node to be touched in a given split tree will be brought to the root next by construction. So this works by the same argument as in the offline version.

## 3 Dynamic optimality and the subsequence property

In this section we summarize Levy and Tarjan [LT19]'s progress towards proving dynamic optimality for Splay, and apply the same ideas to Greedy. They prove that Splay are dynamically optimal if and only if they satisfy a more concrete 'subsequence property'. We summarize the proof of this, and prove the analogous result for Greedy.

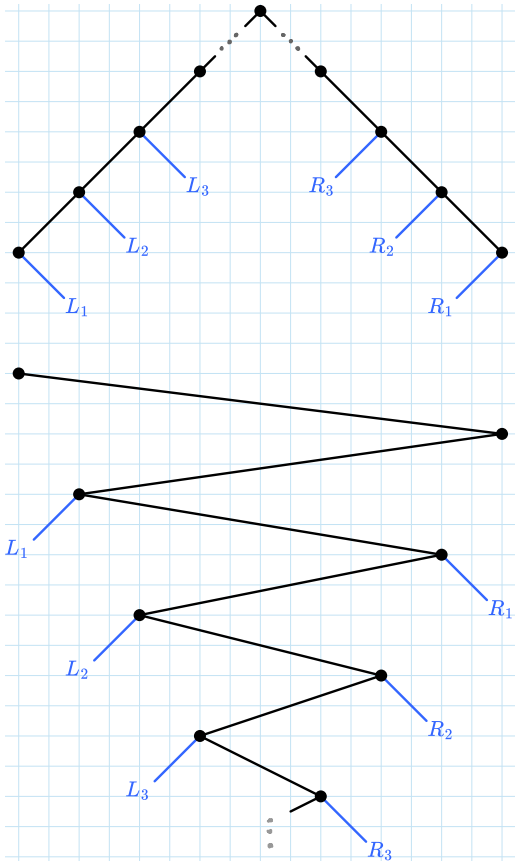



Figure 5: Virtual structure (top) and actual structure (bottom) of a simulated split tree. Finding a node starting from the actual structure's root takes at most twice as long as finding it from either of the virtual structure's extrema. Travel time along the "spine" (black) is doubled, and then travel time in the "interior" (blue) is the same.

**Definition 3.1** ([LT19])**.** A BST algorithm $\mathcal{A}$ satisfies the **subsequence property** if there is a constant $a$ such that for all request sequences $Y$ and $X$ where $Y$ is a subsequence of $X$,

$$\operatorname{cost}_{\mathcal{A}}(Y) \leq a \operatorname{cost}_{\mathcal{A}}(X).$$

Intuitively, the subsequence property says that you can't make $\mathcal{A}$ faster (by more than a constant factor) at handling the sequence $Y$ by giving it 'hints' in the form of sequence $X$. This is clearly a desirable property of a candidate for dynamic optimality, and indeed is necessary for dynamic optimality:

**Lemma 3.2.** *If a BST algorithm is dynamically optimal, then it satisfies the subsequence property.*

*Proof.* Suppose $\mathcal{A}$ is dynamically optimal, so $\operatorname{cost}_{\mathcal{A}}(Y) \leq c \operatorname{opt}(Y)$ for all request sequences $Y$. If $Y$ is a subsequence of $X$, then any access sequence for $X$ also covers the requests of $Y$, and thus is an access sequence for $Y$. In particular, the optimal access sequence for $X$ is an access sequence for $Y$, so $\operatorname{opt}(Y) \leq \operatorname{opt}(X)$. Thus we have $\operatorname{cost}_{\mathcal{A}}(Y) \leq c \operatorname{opt}(Y) \leq c \operatorname{opt}(X) \leq c \operatorname{cost}_{\mathcal{A}}(X)$, and thus $\mathcal{A}$ satisfies the subsequence property with $a = c$. □

## 3.1 Splay

We sketch the proof of the converse of Lemma 3.2 for Splay [LT19].

**Theorem 3.3** ([LT19])**.** *If Splay satisfies the subsequence property, then it is dynamically optimal.*

*Proof Summary.* Levy and Tarjan first show how to reconfigure a splay tree with $k$ nodes into a desired shape via a sequence of splays with total cost at most $80k$.[3] Now suppose Splay satisfies the subsequence property with overhead $a$, and consider a request sequence $Y$. We will construct a supersequence $X$ which guides Splay through the optimal access sequence for $Y$.

To construct $X$, consider the sequence of trees $T_0, \ldots, T_n$ that the optimal access sequence produces: after $i$ requests, the tree is $T_i$. Suppose the optimal access sequence touches $k_i$ nodes during the $i$th request. Then these $i$ nodes form connected subtrees $Q_i$ and $Q_i'$ of $T_{i-1}$ and $T_i$ with the same root. The rest of $T_{i-1}$ and $T_i$ must be the same, though components may be attached to these subtrees in a different way. In particular, if we reconfigure $Q_i$ into $Q_i'$ while only touching these $k_i$ nodes, then $T_{i-1}$ becomes $T_i$. We do this using a sequence of splays which costs at most $80k_i$. Concatenate all of these sequences to obtain the request sequence $X$. Since the $i$th requested node in $Y$ is touched while the optimal access sequence converts $T_{i-1}$ to $T_i$, $X$ is a supersequence of $Y$.

The total cost of $X$ is at most $80\sum k_i$. The cost of the optimal access sequence for $Y$ is $\sum k_i$. Thus we have

$$\text{cost}_{\text{Splay}}(Y) \leq a \,\text{cost}_{\text{Splay}}(X) \leq 80a \sum k_i = 80a \,\text{opt}(Y).$$

□

## 3.2 Greedy

We now pivot to Greedy, and prove the following novel result analogous to Theorem 3.3. Note that the definition of the subsequence property and Lemma 3.2 can be interpreted purely in the geometric model; we do not need to think about trees at all.

**Theorem 3.4.** *If Greedy satisfies the subsequence property, then it is dynamically optimal.*

Our proof will involve modifying a point set to make it $y$-generic:

**Definition 3.5.** Let $X$ and $G$ be point sets, with $G$ $y$-generic. $G$ is a **genericization** of $X$ if the rows of $G$ can be partitioned into intervals called **strips** such that the $i$th strip has points in precisely the same columns as the $i$th row of $Y$. In other words, we can obtain $G$ from $X$ by replacing each row with a strip containing a $y$-generic set of points which becomes the original row when 'flattened' by projecting to the $x$-axis.

An example of a genericization is shown in Figure 6. Note that a point set typically has many genericizations, since we can choose, for each row, how many copies of each point to include, and in what order.

The core of the proof of Theorem 3.4 lies in the following lemma, which is analogous to Levy and Tarjan [LT19]'s algorithm to reshape a splay tree with cost at most $80n$: we are giving Greedy 'hints' $G$ which allow it to efficiently simulate any arborally satisfied set. After proving this lemma, the proof is structurally the same as the proof of Theorem 3.3.

**Lemma 3.6.** *If $X$ is an arborally satisfied point set, then $X$ has a genericization $G$ such that*

$$\text{cost}_{Greedy}(G) \leq 5|X|.$$

*Proof.* We genericize the rows of $X$ one at a time, from bottom to top. A single row is genericized by replacing it with a particular $y$-generic strip (to be described later) with the same occupied columns.

We will describe how to genericize a single row of an arborally satisfied set $Z$, by replacing just that row with a $y$-generic strip which has the same occupied columns, to obtain a point set $Z'$ such that

1. Every point Greedy on $Z'$ places inside the modified strip must be on an occupied column of the genericized row.

[3] This is essentially problem 2c from pset 2, done slightly more efficiently.

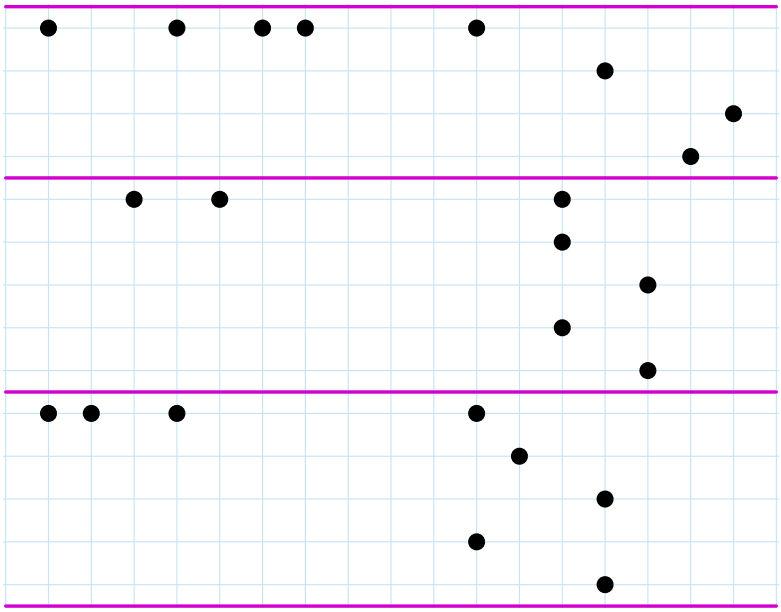


Figure 6: A non-$y$-generic point set (left), and a genericization of it (right). Pink lines indicate the division of the genericization into strips.

2. Once Greedy passes the strip, the point set is arborally satisfied. That is, Greedy does not place any points above the strip.

3. If there are $k$ points in the genericized row of $Z$, the total number of points of Greedy($Z'$) inside the strip is at most $5k$.

Since $Z$ is arborally satisfied, Greedy doesn't place any points before reaching the genericized strip. So $Z$ and Greedy($Z'$) differ only inside the strip that is replaced to build $Z'$, and inside the strip Greedy($Z'$) has at most 5 times as many points as $Z$.

Suppose we have such an operation, and imagine running Greedy on $X$, genericizing each row immediately before Greedy gets to it. Formally, apply the operation to Greedy($X_i$), where $X_i$ is $X$ after genericizing its first $i$ rows. By induction using the second property of our genericization, the point set is arborally satisfied after Greedy passes the $i$th genericized strip but before we genericize the $i+1$th row. In particular, the row of Greedy($X_i$) that is replaced is precisely the $i+1$st row of $X$, so after replacing it and running greedy we obtain Greedy($X_{i+1}$). Ultimately we obtain a $y$-generic point set $G$ after genericizing all rows of $X$. By the third property, the $i$th strip of Greedy($G$) has at most 5 times as many points as $X$, so $\text{cost}_{\text{Greedy}}(G) \leq 5|X|$, as desired.

Figure 7 depicts an operation with the desired properties. Each point in the original row becomes two points in its column in the $y$-generic strip. The leftmost column's points are first $p$ and last $q$, and each other column's points $a_i$ and $b_i$ are consecutive, ordered left to right. For convenience, we also use $a_0$ and $b_0$ to describe $p$.

As before, we call the point sets before and after this replacement $Z$ and $Z'$. We assume $Z$ is arborally satisfied. We must show that this genericization satisfies the three properties mentioned above.

**Greedy points must be on occupied columns.** Assume for contradiction that Greedy on $Z'$ places a point in a column not occupied by a point of $Z'$ in the strip. This point would be to satisfy an unsatisfied rectangle $a_i \diamond c$ for some point $c$ below the strip. If there are multiple, we choose the largest $i$ for which this is the case. This situation is shown in Figure 8.

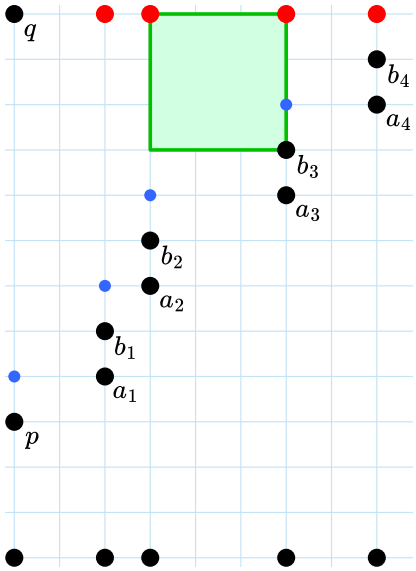



Figure 7: How to genericize a single row. We convert the row of black points at the bottom into the $y$-generic set of black points above it. Red points are produced by Greedy; smaller blue points may be produced by Greedy depending on what happens below.

By assumption, there are no points in $a_i \diamond c$. Then the corresponding rectangle in $Z$ doesn't contain any points below the replaced row, and also doesn't contain any points in the row because we chose the largest possible $i$. This contradicts the assumption that $Z$ is arborally satisfied.

**The point set is arborally satisfied once Greedy finishes the strip.** Consider running Greedy up to and including the replaced strip, and then stopping. Since $Z$ is arborally satisfied, Greedy doesn't place any points below the strip. Any rectangle spanned by two points outside the strip is satisfied in the same way as when running Greedy on $Z$; we can choose any point in the appropriate column of the strip for a rectangle satisfied by a point in the replaced row. Because Greedy leaves behind an arborally satisfied set, any rectangle spanned by points inside or below the strip is satisfied. Thus we only need to concern ourselves with rectangles spanned by one point in the strip and one point above it.

To show that such a rectangle is satisfied, we will argue that the top row of the strip contains points in exactly the same columns as the corresponding row of $Z$. Then, since every point in the strip is on one of these columns, any rectangle spanned by a point $v$ in the strip below the top row and a point above the strip is satisfied by the point in the top row of the strip in the same column as $v$. Any rectangle spanned by a point in the top row and a point above the strip is satisfied in the same way as the corresponding rectangle of $Z$ – the set of points in the top row of the strip and above looks exactly like the set of points in $Z$ at and above the replaced row. This covers all possible rectangles, so the point set is arborally satisfied when Greedy is finished with the strip.

Thus we just need to argue that the red points in Figure 7 must exist. Let $c_i$ be the red point above $b_i$, with $c_0 = q$. We prove that Greedy places $c_i$ by induction on $i$. Consider $c_i \diamond b_{i+1}$, drawn in green in Figure 7 for $c = 2$. It contains no points of $Z'$, and we will determine where Greedy might add points inside it.

Every rectangle spanned by $b_{i+1}$ and a lower point contains $a_{i+1}$, so Greedy doesn't add any points in the row of $b_{i+1}$. In a similar way, $a_{i+2}$ shields $b_{i+1}$ from all other points before $q$: the rectangle $b_{i+1} \diamond b_j$ with $j \geq i+2$ or $b_{i+1} \diamond a_j$ with $j \geq i+3$ contains $a_{i+2}$. In addition, $b_{i+1}$ shields all earlier points below and right of it from all of the points between $b_{i+1}$ and $q$. Hence the only way Greedy can place a point in $c_i \diamond b_{i+1}$ before the top row is to satisfy $b_{i+1} \diamond a_{i+2}$, in which case the blue point $d_{i+1}$ immediately above $b_{i+1}$ exists. Note that $d_{i+1}$ may or may not exist depending on whether Greedy previously added a point at the

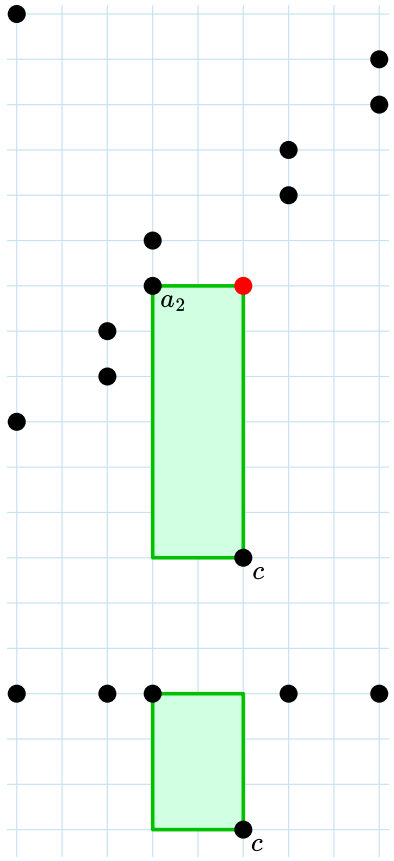



Figure 8: Top: how Greedy might add a point (red) in an unoccupied column. Bottom: the corresponding situation for the pre-replacement point set. The green rectangle violates arborality.

opposite corner of $b_{i+1} \diamond a_{i+2}$.

One of $c_i \diamond b_{i+1}$ or $c_i \diamond d_{i+1}$ (depending on whether $d_{i+1}$ exists) is not satisfied, and thus Greedy places $c_{i+1}$ to satisfy it.

**Greedy puts at most $5k$ total points in the strip.** Recall that $k$ is the number of points in $Z$ in the relevant row, or the number of columns occupied by points in the strip in $Z'$. There are $2k$ points of $Z'$ in the strip. We will separately count the points added below and above the 'staircase' (i.e. below and above $a_i$). Since points are only added in occupied columns, every point is directly above or below some $a_i$ (recall that $a_0 = p$).

In proving the previous fact, we showed that the only places points can be added above $a_i$ are at $d_i$ and $c_i$ (the blue and red points, respectively, in Figure 7). Thus at most $2k$ points are added above the staircase. In fact, since $c_0 = q$ already exists in $Z'$, we can slightly improve this bound to $2k - 1$.

Bounding the number of points added below the staircase is harder. We know that Greedy doesn't add points outside of the columns containing points of $Z'$. Figure 9 shows a typical output of Greedy below the staircase; assume that the region below the strip is arborally satisfied by additional points below the point drawn in each column. In this example, there are eight occupied columns and seven Greedy points below the staircase; we will soon see that this isn't a coincidence.

To bound the number of these points, we first prove a fact which limits where they can exist. Suppose Greedy places a point $t$ in the strip below the staircase, and let $a_i$ be directly above $t$. We claim that there are no points inside or on the top edge of the rectangle extending right from the segment between $t$ and

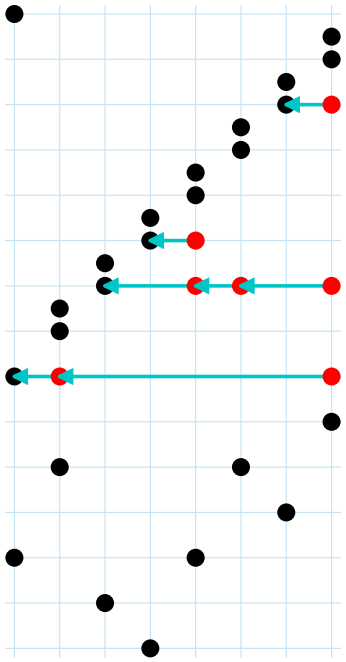


Figure 9: Points added by Greedy below the staircase. Blue arrows show how we assign points to columns.

$a_i$. This situation is shown in Figure 10. To prove this claim, assume for contradiction that Greedy does place a point in this region, and let $u$ be the first such point placed. Then $u$ is placed to satisfy a rectangle spanned by a point in the row of $u$ and a point directly below $u$. Because the green rectangle is empty before placing $u$, the point in its row must be to the left of the rectangle, and the point below $u$ must be below the rectangle; either may be on the edge of the rectangle. But then $t$ satisfies the rectangle that $u$ is placed to satisfy, so there is no need to place $u$.

Now we are ready to bound the number of points Greedy placed below the staircase. We will assign each such point to a column occupied in the strip. Specifically, move each such point $v$ left until it hits another point, which may be another Greedy point or $a_i$, and assign $v$ to the column of this other point. The blue arrows in Figure 9 demonstrate this process.

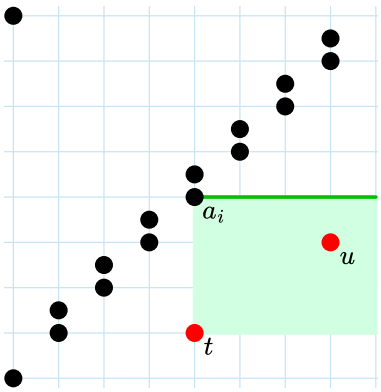



Figure 10: When Greedy places a point $t$, the green region must be empty.

We claim that two points can't be assigned to the same column. For such a thing to happen, we would need a situation like the one shown in Figure 11, where $s$ and $t$ are in the same column with $s$ above $t$, and

$u$ and $v$ move left to $s$ and $t$. Let $a_i$ be directly above $t$. The top left point $s$ might be $a_i$ or a Greedy point; the others must be added by Greedy.

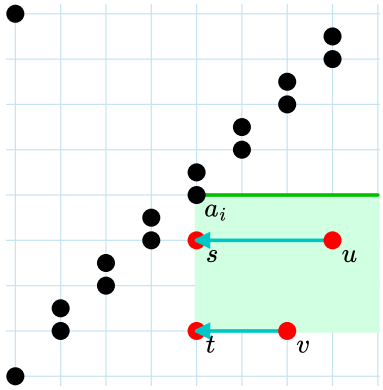



Figure 11: A situation in which we would assign two points $s$ and $t$ to the same column, which turns out to be impossible.

Since Greedy places $t$, we just showed that the green rectangle in Figure 11 must be empty, including its top edge. But $u$ is in this region: $s$ is directly above $t$ at or below the row of $a_i$ and $u$ is directly right of $s$, so $u$ is also at or below the row of $a_i$, above the row of $t$, and right of the column of $t$. Thus Greedy doesn't place $u$, so we never have two points assigned to the same column.

Since there are $k$ columns, and each Greedy point in the strip below the staircase is assigned to a different column, there are at most $k$ such Greedy points. In fact, no point can be assigned to the rightmost column, so we have a slightly tighter bound of $k-1$.

To summarize: there are $2k$ points of $Z'$ in the strip, and Greedy adds at most $2k-1$ points above and $k-1$ points below the staircase. Hence the total number of points in the strip is at most $5k-2<5k$. □

With Lemma 3.6 in hand, the remainder of the proof of Theorem 3.4 straightforwardly mirrors the structure of Theorem 3.3.

*Proof of Theorem 3.4.* Suppose Greedy satisfies the subsequence property with overhead $a$. Consider a request sequence $X$, and let $X'$ be the smallest arborally satisfied superset of $X$. Since $X'$ is arborally satisfied, by Lemma 3.6 it has a genericization $G$ with $\text{cost}_{\text{Greedy}}(G) \le 5|X'| = 5\,\text{opt}(X)$. Then $X$ is a subsequence of $G$, so by assumption we have $\text{cost}_{\text{Greedy}}(X) \le a\,\text{cost}_{\text{Greedy}}(G) \le 5a\,\text{opt}(X)$. So Greedy is dynamically optimal. □

Thanks to Theorem 3.4, all that is needed to prove that Greedy is dynamically optimal is showing that Greedy satisfies the subsequence property. Unfortunately, we have not yet been able to do this.

It is reasonable to wonder whether Greedy satisfies the subsequence property with $a=1$; it other words, does removing a point never cause Greedy to become strictly more expensive? This seems to be true for the vast majority of small $y$-generic point sets, and if true would be a promising approach for proving dynamic optimality. Unfortunately, Figure 12 shows a counterexample. If we extend the pattern in this example, we find that removing a point from the point set increases the number of points from $13n+6$ to $15n$ (in Figure 12, $n=6$). This means that if Greedy satisfies the subsequence property, it has $a \ge \frac{15}{13}$.

# References

[DHI+09] Erik D Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Patraşcu. The geometry of binary search trees. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 496–505. SIAM, 2009.

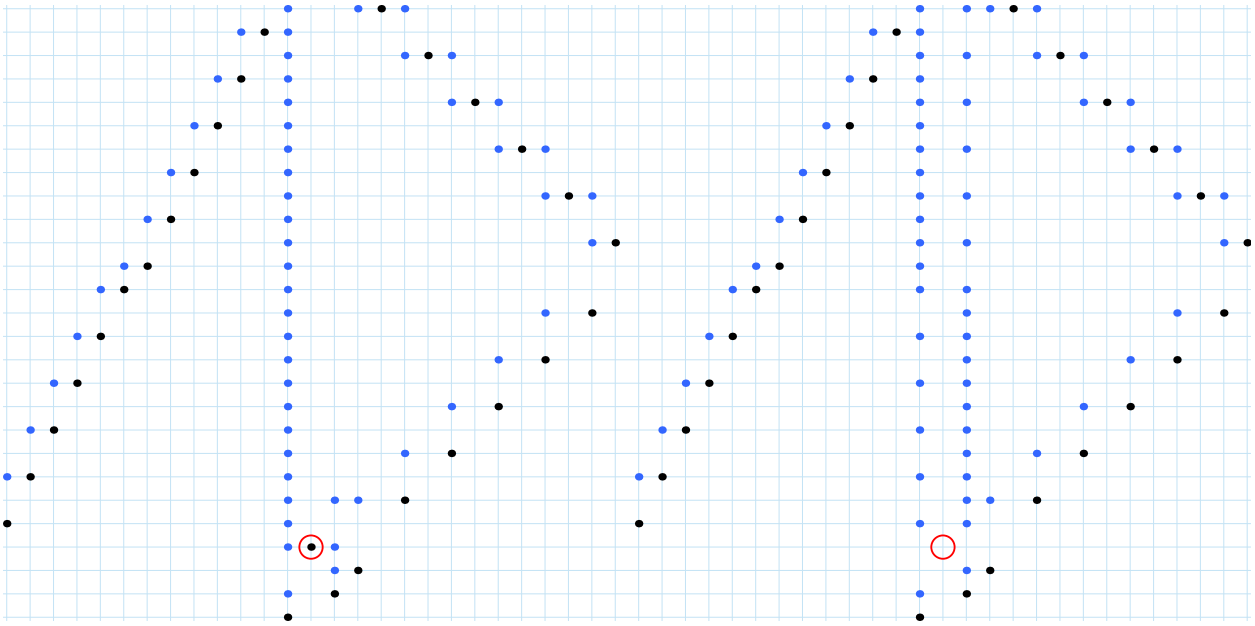


Figure 12: Removing the circled point from the left point set to obtain the right point set increases the number of points of the Greedy superset from 84 to 90. Black points are in the request sequence; blue points are added by Greedy.


[LT19] Caleb Levy and Robert Tarjan. A new path from splay to dynamic optimality. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1311–1330. SIAM, 2019.

[STT86] Daniel Sleator, Robert Tarjan, and William Thurston. Rotation distance, triangulations, and hyperbolic geometry. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 122–135. ACM, 1986.