Dynamic Optimality, a Short Overview

Hussein Houdrouge

November 2021

1 Introduction

A binary search tree, BST for short, is one of the most fundamental data structures. Although BST is a very simple data structure, the most fundamental question about it remains unsolved, *what is the best BST?*

In the remaining of the introduction, we will recall the properties of BST, then introduce the problem of dynamic optimally. However, The reader who is familiar with the subject can start from Section 3 where we discuss the model of computation of a BST. After, in Section 4, we discuss a lower bound on the run-time of any BST. We proceed to present *Tango Tree* in Section 5. Finally, we present a geometric equivalence for BST.

1.1 Binary Search Tree

We will borrow the recursive definition of a tree from [Knu97]. A tree is a finite set T of one or more nodes such that

1. There is one special node that is called the root, denoted by $root(T)$.
2. The set $T \backslash root(T)$ are portioned into $m \geq 0$ disjoint sets $T_1, T_2, ..., T_m$ where every T_i is a tree. They are called the sub-trees of the root.

A binary tree is a tree where the root has at most two sub-trees. One sub-tree is called the left sub-tree of the root, the other one is called the right sub-tree. If a sub-tree is not existing, we will denoted by Nil. A leaf is a sub-tree that consists of only one node. The height of the tree is defined as the length of the longest path from the root to a leaf. In addition, each node stores a key value. These keys are chosen from a totally ordered set, and they satisfy the following property.

The binary-search-tree property [Cor09]: Let x be a node of a binary search tree. If y is a node in the left sub-tree of x then the key of y is smaller than or equal to the key of

x. If y is a node in a node in right sub-tree of x, then the key of y is greater than or equal to the key of x.

The main purpose of this BST is to perform efficiently one of the following operations: insertion, deletion, and searching. These operations are performed by comparing the element in the query by root of a tree, the by the above property decide to move to the left or right sub-tree. Then, apply the same procedure recursively till hitting the desired target. More details on the standard implementation can be found in [Cor09]. However, these operation may alter the shape of the tree which will lead to increase in the height of the tree. Consequently, the time taken to perform these operations will increase too. Thus, the main challenge is to perform these operation optimally while keeping the shape of the tree "optimal". For these reasons, several implementation has been developed, some are briefly mentioned in Section 2. One of these implementations is Red-Black tree that performs these operations in $O(\log n)$ [Cor09]. This result is followed from the following lemma.

Lemma 1.1 ([Cor09]). *A red black tree with n nodes has height at most $2\log(n+1)$.*

1.2 Dynamic Optimality

The main objective is to solve the following problem as efficiently as possible.

Problem 1: Given a *BST* data structure over a set of static keys from 1 to n. The BST will only performs successful search operations i.e. search for elements stored in the *BST*.
INPUT: A sequence of of keys $S = \langle k_1, ..., k_m \rangle$ where each $k_i \in \{1, ..., n\}$. We call S an access sequence.
OUTPUT: a sequence of pointers $P = \langle p_1, ..., p_m \rangle$ where p_i points to the node where k_i is stored.

More precisely, we want to solve **Problem 1** in an online setting i.e using an online *BST*. An online BST does not have an access for the entire sequence at the start of the search. It can only make use of the data that already stored in the *BST*. But it has the freedom to augment each node with additional data that should be as small as possible [DHIP07].

Given an access sequence S, $OPT(S)$ will denote the number of unit-cost operations performed by the fastest *BST* that executes S according to a fixed *BST* model described Section 3. This *BST* can be offline, i.e. it has access to the entire sequence before doing the searches. One may assume that this *BST* starts from the best possible *BST*, as converting one BST to another through rotation operation can be performed in linear time. This additional cost will add only a constant factor to $OPT(S)$ assuming $m \geq cn$ for large n. [DHIP07] [STT86].

An online BST is c-competitive if it executes any sequence S in $c \times OPT(S)$. We say an online BST is *dynamically optimal* if c is a constant. By Lemma 1.1, the Red-Black tree can solve **Problem 1** in $O(|S| \log n)$ since $OPT(S) \geq |S|$ for all S. Thus, Red-Black tree is $\log n$-competitive. The *Tango Tree*, described in Section 5 is $O(\log \log n)$-competitive. However, the question whether there is a dynamically optimal BST is still open. It is conjectured that splay tree of [STT86] is dynamically optimal but no proof has been given yet.

2 BST Variations

There are many variations of BST. They mainly differ in the way of restructuring the tree after adding, deleting, and searching for an element. The main tool used in this restructuring is called node rotations as Figure 1 shows.

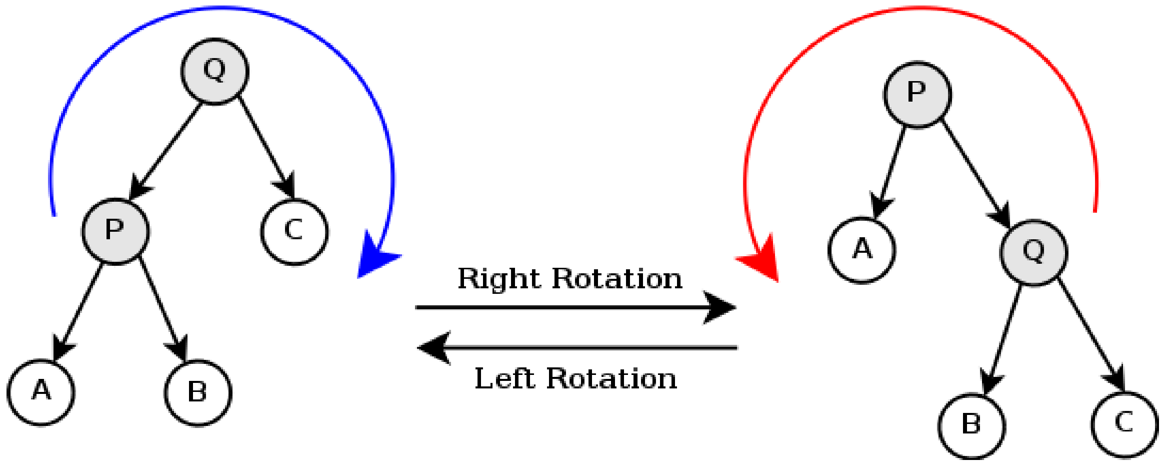


Figure 1: Left and right rotation.

In this section, we will briefly discuss three variations: Treap, Red-Black Tree, and Splay Tree.

2.1 Treap

Given a set of keys, that we want to store in a BST. Each key will be associated with a priority p. The priority could be drawn randomly or associated in a deterministic way. The Treap is a binary search tree of the keys i.e. the keys satisfies the binary-search-property, and a binary heap over the priorities (it could a min heap or max heap). A min binary heap is a binary tree where the priority of the parent is at most the priority of its children. For additional details, the reader can consult Chapter 7 of [Mor].

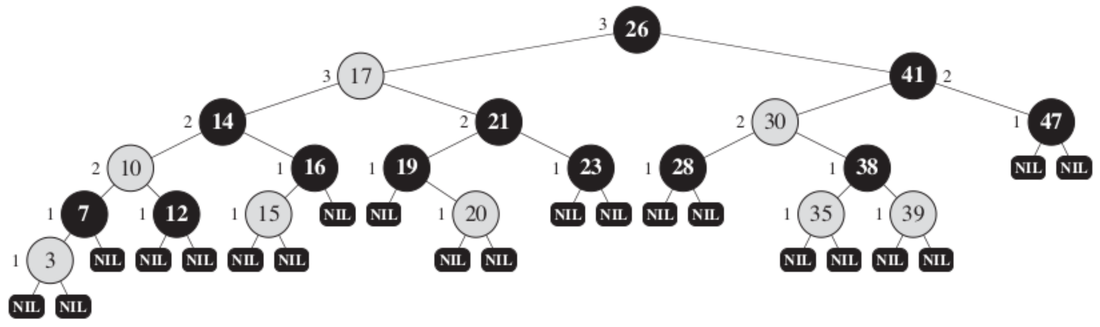


Figure 2: An example of red-black tree, the grey nodes are the red one. [Cor09].

2.2 Red-Black Tree

The Red-Black Tree is another binary search tree that assigns to each node a color (red or black). Beside, the binary-search-tree property it has to satisfy the following ones, known as the red-black tree properties.

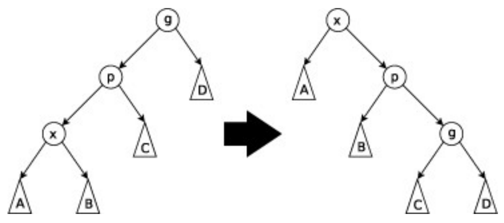
1. Each node is either red or black.
2. The root is colored black.
3. The leaves are denoted by Nil and black colored.
4. The children of a red node are black.
5. For every node, the paths to its leaves contain same number of black nodes.

The details and the implementation of the tree can be found in [Cor09] Chapter 13.

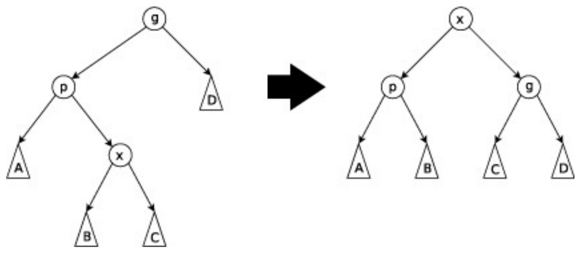
2.3 Splay Tree

Splay tree is introduced in [ST83]. It is a binary search tree that equipped with several operations to do some adjustments. The main operation is splay that bring a node to the root. The splay is composed of several operations such as zig, zig-zig, zig-zag, zag-zag ... Figure 3 illustrates some of these operations that are combination/renaming of left and right rotation.

The *Splay Tree* was conjectured to be dynamically optimal but no proof has given so far.



(a)



(b)

Figure 3: (a) Zig-Zig Operation (Source: Wikipedia), (b) Zig-Zag Operation (Source: Wikipedia).

3 BST's Model of Computation

In this section, we will define the model that abstract and limit the behaviour of a binary search tree.

Definition 3.1. (Tree Reconfiguration, [DHI$^+$09]) Let T_1 be a BST, and τ be a sub-tree of T_1 containing its root. Further, let τ' be a BST on the nodes of τ. We will define the operation $\tau \longrightarrow \tau'$ as changing T_1 to T_2 where T_2 is identical to T_1 except that τ is replaced by τ'. The cost of this reconfiguration is $|\tau| = |\tau'|$.

Definition 3.2 (BST Model, [DHI$^+$09])**.** Let $S = \langle s_1, s_2, ..., s_m \rangle$ be a search sequence. Given any BST algorithm A, an execution of A is a sequence of reconfiguration applied to an initial tree T_0 denoted by $E = \langle T_0, \tau_1 \longrightarrow \tau'_1, \tau_2 \longrightarrow \tau'_2, ..., \tau_m \longrightarrow \tau'_m \rangle$ such that $s_i \in \tau_i$ for any i and each reconfiguration is valid. We will define T_i to be the resulted tree from applying the reconfiguration $\tau_i \longrightarrow \tau'_i$ on T_{i-1}. The cost of E is $\sum_{i=1}^{m} |\tau_i|$.

Indeed, other BST models exist which are constant factor-equivalent. For instance, we can define a BST data structure by an algorithm for queering a key k_i [DHIP07]. The BST access algorithm will have one pointer to a single node in the tree. At the start of the query for k_i, the pointer is initialised to the root. The algorithm can perform one or several operations from the followings.

1. Move pointer to the left child.
2. Move pointer to right child.
3. Move the pointers to its parent.
4. Perform a single rotation on the pointer and its parent.

Whenever a pointer moves from a node to another or initialised to a new one, we say that the node is touched. The execution time of a BST on an access sequence S, is the number of operation performed. Clearly, the number of touched node is a lower bound which is also bounded by the number of elements in S.

4 The Interleave Lower Bound

In this section, we will discuss a lower bounds on $OPT(S)$ where S is an access sequence. More precisely, we will state a variation of Wilber first lower bound [Wil89]. This variation is presented in [DHIP07]. It is called the interleave lower bound. Suppose the keys belong to the universe $\{1, ..., n\}$, fix a perfect binary search tree P (It can be complete if n is not a power of 2). P is called the lower bound tree. For each node y in P, define the left region of y as the nodes of left sub-tree of y plus y, and the right region of y as nodes of the right sub-tree of y. Now, given an access sequence $S =< s_1, ..., s_m >$, for each node y in P, each access s_i in S is either labeled in the left region or the right region of y. We discard any access outside the sub-tree rooted at y. Now, for each y in P, we have a sequence of labels of left and right region, we define **the interleaving through y**, denoted $iv_S(y)$, as the number of alternation between left and right regions in the sequence. Consequently, we will define the interleave lower bound of S, denoted by $IB(S)$, as following

$$IB(S) = \sum_{y \in P} iv_S(y).$$

Using $IB(S)$, we can have a lower bound on $OPT(S)$.

Theorem 4.1 ([DHIP07]). *Let n be the number of elements in the universe of keys, and S be an access sequence, we have the following lower bound on $OPT(S)$,*

$$OPT(S) \geq IB(S)/2 - n.$$

Other lower bounds also exists, one of them knows as Wilber second bound, introduced in [Wil89]. We will introduce this bound in addition the interleave bound in Section 6.4 as wilder family of lower bounds.

5 $\mathbf{O}(\log \log \mathbf{n})$-competitive BST

In this section, we will present the main work done in [DHIP07] which is a new BST algorithm, named *Tango Tree.* This new BST achieve a $O(\log \log n)$-competitive ration using the first Wilber lower bound, the interleave lower bound discussed in Section 4. This tree is based in an augmented lower bound BST P as it defined in Section 4. More precisely, it is a perfect binary tree on the set of keys $\{1, ..., n\}$, assume n to be a power of 2 for simplicity. Each internal node of P stores an additional information *preferred child*. For a node y, if the last access was in the left region of y, that includes y, then the preferred child is the left child. It is set to the right child, if the last access was in the right region. However, if the access does not touch y, then y has no preferred child. Now, given the tree P where each node may have a preferred child, we start from the root following the preferred child pointer, we get a path that is called a preferred path. We store the preferred path in a new auxiliary BST denoted by R. Then, we remove the preferred path from P, which disintegrates P into a forest. We recurse on each tree in the forest. Then, the resulting BST from each piece will be attached to R. In the next section, will describe how this auxiliary tree works.

5.1 Auxiliary Tree

The auxiliary tree is defined as an augmented BST. The purpose of this tree is to store a preferred path. The nodes of the tree are ordered by the key value. Each node of the tree is augmented with additional value corresponds to its depth in the lower bound tree P. The auxiliary tree performs the following operation in $O(\log k)$ where k is the total number of nodes in the auxiliary tree(s) performed in the operation.

- Search for a key in the auxiliary tree.
- Cut the tree into two auxiliary tree based on a given depth at most d. The result will be two auxiliary trees one stores path of nodes at depth greater than d and one less than d.
- Join two auxiliary trees that store two disjoint paths where the deepest node of one path is the parent of the shallowest node of the other.

The auxiliary tree is implement as Red-Black tree [DHIP07], described in Section 2. Each node stores, the key value, the color (red or black), its depth in P, the min and max depth in P for the node in its sub-trees. In the implementation of Red-Black tree the nodes that lack a child points to a special value [Cor09]. However, in this case the pointer to a special value may points to another auxiliary tree. In other words, this pointer points to an auxiliary tree that corresponds to different preferred paths. The auxiliary structure that we have is a tree if of auxiliary trees rather than a forest. To distinguish an auxiliary tree we mark its root. According to [Cor09], the Red-Black tree performs search, split, and concatenate operations in $O(\log k)$, where k is the number of nodes involved in the operation. We will make use of these operation to perform the above mentioned ones. The split and concatenate operations can be described in the following way.

- Given a node x, a split on x returns a tree where x is the root, the left sub-tree of x is a Red-black tree containing all the element small than x, the right sub-tree is also a Red-Black tree containing the remaining element.
- Given a node x, concatenate operation will take the left and the right sub-tree of x and arrange them into one Red-Black tree.

To perform the cut operation on an auxiliary tree A at depth d. The nodes at depth greater than d form an interval over the keys in A. This observation is due that all the nodes with depth greater than d are less than the node with depth at most $d-1$ and consequently its right child (if they are in a left sub-tree, similar reasoning if they are on a right sub-tree). Therefore, no node after this depth is greater than a node in the right of the tree. Now, we want to find the node ℓ of minimum key value with depth greater than d. This operation can be done by constantly moving to the leftmost child whose sub-tree has maximum depth greater than d. Then, find the node r with maximum key value whose depth greater than d. Then, find the predecessor of ℓ, denoted by ℓ', and the successor of r, denoted by r'. All the target nodes are in the closed interval $[\ell, r]$, equivalently the open interval (ℓ', r'). Now, use the split of the red-black tree to split on ℓ' and r'. The first split will create two sub-trees, call B the left one, and C the right one. Since $r' \in C$, the second split will split C into two trees. D is a left sub-tree of r' and E is the right one. Notice that D contains all the nodes in (ℓ', r'). After, the splits, we mark the root of D, as it is a new auxiliary tree by itself. Now, we want make B, E, ℓ', and r' into one red-black tree. For this purpose, we concatenate at r' first, and ℓ' second.

To perform the join operation on two trees A and B, find the tree that stores the node with largest depth by comparing the max depth value stored in the root of each tree. Assume A is the one with node of largest depth, then the tree B has set of nodes that fall into two adjacent nodes of A, with key value ℓ' and r' respectively. Assume ℓ' less than r'. To get these value, search for the root of B in A. Then, split A at ℓ' and r' in that order (ℓ' and r' form an edge in A). We can attach the root of B to the left child of r'. It is clear that r' has no left child since it is strictly larger than ℓ' and no node in A is between these two values. Then unmark the root, and concatenate on r' then ℓ'.

5.2 Tango Algorithm

Given an access x_i, the tango algorithm change its state from T_{i-1} to T_i i.e. it modifies the tree of auxiliary trees that pictures the preferred paths in the lower bound tree P. The access algorithm follows the normal BST search toward the desired element x_i. Along the way to x_i, the preferred children changes to make a new preferred path. The change of preferred children indicates when the algorithm crossed from one augmented BST to the another. In the tree of auxiliary trees that corresponds to when the algorithm touch a marked node y. At this moment, we have to perform cut and join operation to construct the new preferred path. Therefor, we cut the auxiliary tree that contains the parent of y at a depth one less the minimum depth of nodes in the auxiliary tree rooted at y i.e. all the sibling of the minimum depth node in tree rooted at y in P. Then, join the resulted top tree with tree rooted at y. When we reach x_i, we cut at depth of x_i in the auxiliary tree containing x_i, and join the top path with preceding auxiliary tree.

The cost of this algorithm can be divided into two main parts. The first one is the cost of searching of x_i, and the second one is the cost of rearranging the tree which is proportional to the time when a node changes its preferred child. Suppose that k is the number of nodes whose preferred child changed. Then, at most we moved along $k+1$ path and in each path the search cost is $O(\log\log n)$ since its auxiliary tree contains at most $O(\log n)$ node. Thus, the total cost of search is $O((k+1)\log\log n)$. Similarly for rearrangement of the trees, we perform at most one cut and one join per auxiliary tree, i.e. we updated path at most $k+1$ time. The cost of these operation is $O(\log\log n)$ which yields a total of $O((k+1)\log\log n)$.

Now, we want to relate the behavior of the search algorithm to the interleave lower-bound in order to analyse the competitiveness of the algorithm. Define the interleave bound $IB_i(X)$ of access X_i to be the interleave bound on $x_1, ..., x_i$ minus the interleave bound on $x_1, ..., x_{i-1}$. Then, we can have the following lemma.

Lemma 5.1 ([DHIP07])**.** *The additional interleave bound introduced by x_i, $IB_i(X)$, is equal to the number of nodes who changed their preferred children from left to right and right to left during the access x_i.*

The proof of this lemma follows from the definition of the interleave lower bound. From the previous Lemmas the following theorem follows.

Theorem 5.2 ([DHIP07])**.** *Given a sequence X of m access on the universe $\{1, ..., n\}$, the running time of the Tango BST on X is $O((OPT(X)+n)(1+\log\log n))$, where $OPT(X)$ is the cost of the offline optimal BST computing X.*

When $m = \Omega(n)$, the running time of Tango BST will be $O((OPT(X)\log\log n)$.

6 Geometric Representation

In this section, we will introduce the Arborally Satisfied Set (ASS) and its relation to BST. In addition, we will present algorithms to compute the ASS of a set of points, and explain its relation to dynamic optimality. Finally, we will discuss geometric lower bound resulted from this geometric view of BST.

For the rest of this document, we will denote by n the number of elements stored in the BST and m be the total number of search operations performed.

6.1 Arborally Satisfied Set

Consider the $n \times m$ grid subset of $\mathbb{Z} \times \mathbb{Z}$. A point $p = (p.x, p.y)$ is a point in this grid where $1 \leq p.x \leq n$ and $1 \leq p.y \leq m$. We denote by $\square ab$ the axis-aligned rectangular region, including the boundary , with corner a and b. We proceed to define Arborally Satisfied Set of points.

Definition 6.1. Given a set of point P and a pair of points $(a, b) \in P$, we say they are arborally satisfied with respect to P if they satisfy one of these two conditions.

1. They are horizontally or vertically collinear.
2. There exists one point $q \in P \backslash \{ab\}$ in $\square ab$.

Definition 6.2. A set of point P in $n \times m$ grid is arborally satisfied if for any $a, b \in P$, The pair (a, b) is arborally satisfied.

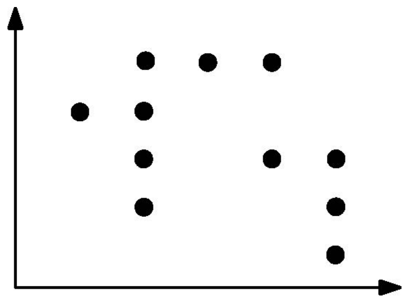


Figure 4: Example of an arborally satisfies set of points where the x-axis represents the key space, and the y-axis the time. Source: Wikipedia

From the above definitions it is easy to see that if a set of points P is arborally satisfied, then for any $a, b \in P$ that are not on the same horizontal or vertical line there is at least one point different from a and b on the sides of $\square ab$ incident to a and at least one on the sides incident to b.

Now, given the BST model in the Definition 3.2, we can define a geometric view of a BST execution E as follow.

Definition 6.3 ([DHI$^+$09])**.** Let E be a BST execution, the geometric view of E is the set of points

$$P(E) = \{(x, y) | x \in \tau_y, \text{ for all } y\}.$$

where τ_y is set of touched node at time y.

In other words, at time i which is on y-axis, we plot all the nodes touched during the execution. The point set resulted from the execution is arborally satisfied as the following lemma states [DHI$^+$09].

Lemma 6.1 ([DHI$^+$09])**.** *Given a BST execution E, the geometric view of E, denoted by $P(E)$ is arborally satisfied.*

Till this moment, we had only a transformation from the execution sequence to a geometric representation. But it is not clear how this geometric model relates to the BST. Surprisingly, given a set of points that are arborally satisfied, we can find a corresponding BST execution as the following lemma states.

Lemma 6.2 ([DHI$^+$09])**.** *Given a set of points X that is arborally satisfied, then there exists a BST execution E such that $P(E) = X$. E is called the arboral view of X and denoted by $E = P^{-1}(X)$.*

Proof. The proof is constructive, we will describe an algorithm that created the execution sequence.

First define $N(x, i)$ to be the next access time of x at i to be the minimum y coordinate for any point in X on the ray from (x, i) to (x, ∞). In other words, it is the most recent time we accessed x from time i forward, it could include i as well. If no such point exists then $N(x, i) = \infty$.

Let T_i be the treap defined on the points $\{(x, N(x, i))\}$. It is BST on x and min heap on $N(x, i)$.

Let $\tau_i = \{(x, i) \in P\}$. This set of nodes in T_i form a connected component that includes the root, since the root has the minimum access time $N(root, i) = i$, because i is the minimum possible access time. Now, to construct T_{i+1}, we just arrange the tree based on the new access time $N(x, i+1)$. It is clear that τ_i only needs to be arranged but. Thus, we have to show only that T_{i+1} is a treap on $(x, N(x, i + 1))$. The BST property is trivially hold by construction, rearrangement happen by right and left rotation that preserve the BST property. However, the heap property is not obvious. We want to show that the heap property holds between every parent/child pair (p, r) in T_{i+1}. If both, parent and child, where in τ_i then the property hold by construction. Similarly if they both where outside, as their next access time does not change by the move from time i to $i + 1$. The only case we need to examine is one the parent $q \in \tau_i$ and the child $r \notin \tau_i$. Suppose that the heap property is violated in T_{i+1}. Then, the rectangle formed by (q, i) and $(r, N(r, i))$ violated the arborally satisfied property. The vertical side at $x = q$ is empty because $N(q, i + 1) > N(r, i)$ by the heap property being violated, and the horizontal line $y = i$ is empty because if it is not the case q would not be the parent of r. Thus, we are contradicting the assumption that X is arborally satisfied point set. □

The interesting idea behind this geometric representation is that we can start seeing a reformulation of the dynamic optimally problem in a geometric stand point which might add an new insight to tackle the problem. From the previous lemmas, we have shown that given a sequence $S = s_1, ..., s_m$ represented geometrically by a set of points $P(S) = \{(s_1, 1), (s_2, 2), ..., (s_m, m)\}$, its execution E induces a super-set of $P(S)$, i.e. $P(S) \subset P(E)$. This super-set is also translate to a binary search tree. So, if we define $minASS(P(S))$ to be the smallest set of points that is arborally satisfied and contains $P(S)$ then we have $OPT(S) = minASS(P(S))$. In other words, $minASS(P(S))$ is the minimum execution we can compute. Now, our problem is reduced to compute $minASS(P(S))$.

6.2 Online Arborally Satisfied Super-set

In the previous section, we saw an equivalence between finding an arborally satisfied set and building an optimal BST. However, we want relate the problem of finding an arborally satisfied set to an online BST i.e we want to be able to produce the transformation $\tau_i \longrightarrow \tau_i'$ after each s_i is revealed without having any information about the next access. Thus, we can define the online arborally satisfied super-set problem as follow.

Definition 6.4 (Online arborally satisfied super-set, online ASS [DHI+09])**.** Given a set of points $\{(s_1, 1), (s_2, 2), ..., (s_m, m)\}$, design an algorithm that receives each point at a time, After receiving a point i, the algorithm must output a set of point P_i on the i^{th} row, i.e $y = i$ such The set of point $\{(s_1, 1), ..., (s_i, i) \cup P_1 \cup P_2 \cup P_i$ is arborally satisfied. The cost of the algorithm is $\sum_{i=1}^{n} P_i$.

Similarly to the previous section, an online BST gives an online ASS. However, the converse is not clear as in the proof of Lemma 6.2, we used a knowledge of the future by using next access time.

6.3 Greedy Algorithms

In this section, we present two greedy algorithms proposed to solve the $minASS$ problem and consequently computing $OPT(S)$ for an access sequence S. The first algorithm is called Greedy-Future and it is due to Lucas and Munro [DHI+09].

The algorithm is as follow.

1. Touch only the nodes in the search path.

2. re-arrange the search path such that the next item in the access sequence is placed as high as possible, and recurse on the remaining item.

Thus, suppose we are given an access $s_i \in T_i$, and τ_i is the search path of s_i. If $s_{i+1} \in \tau_i$ make s_{i+1} the root and recurse on the right and left sub-tree. Otherwise, $s_{i+1} \notin \tau_i$, the algorithm rearranges the predecessor and successor from τ_i as the root and right child of the root. Then recurse on the remaining part of the tree.

This algorithm on the trees side has a geometric counter part. The geometric algorithm goes as follow.

1. Use a sweep line from bottom to up i.e increase the y coordinate.

2. Arborally satisfied the points with $y \leq i$ by adding minimal number of points at $y = i$. More precisely, for any unsatisfied rectangle formed with (s_i, i) in one of its corner, add the other corner at $y = i$.

The greedy aspect of the geometric view is pretty obvious as we want to a minimum number of points. However, for the tree view, the greedy aspect manifests by making the decision of only touching search path which is a cheaper than going outside the path.

Observe that the above mentioned algorithm is only dependent on the past, points with smaller y-coordinate, which make it a suitable choice for online algorithm.

6.4 Geometric Lower Bounds

One important consequence of the geometric view is the reinterpretation of lower bounds in geometric terms. In addition, it comes up with reach lower bounds the previous known bounds by Wilber [Wil89] are special cases [DHI$^+$09]. To introduce this bound, we start by the following definition.

Definition 6.5 (Independent rectangles)**.** Given two rectangles $\square ab$ and $\square cd$, and a set of points X in $n \times m$ integer grids. We say the two rectangles are independent if they are not arborally satisfied and no corner of one rectangle is inside the other one.

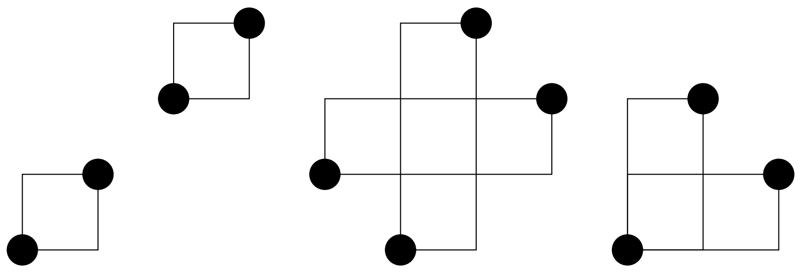


Figure 5: Example of independent rectangles.

The set of independent rectangles form a lower bound on $OPT(X)$.

Theorem 6.3 ([DHI+09])**.** *Given a set of points X, and suppose that I is an independent set of rectangles in X, then*

$$minASS \geq \frac{|I|}{2} + |X|$$

For the special case of $X = P(S)$ where S is an access sequence, then,

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

Notice that maximising $|I|$ will give a better lower bound. We will denote the maximum independent set of rectangles $maxIRB(X)$. We will mention in the remaining of this section an algorithm that computes a constant factor approximation. But before, we will present Wilber 1 and 2 geometrically.
We can reconstruct Wilber first bound using the following algorithm from [DHI+09]. The goal of the algorithm is to build a sequence of independent rectangles.

1. Let $L = (\ell_1, \ell_2, ...)$ be any sequence of vertical lines.
2. Consider ℓ_1.
    (a) Examine the points of X sorted by y-coordinates.
    (b) For each pair of consecutive points where a point p_i on the left of ℓ_1, and another point p_{i+1} on the right.
    (c) Output $\square p_i p_{i+1}$.
3. recurse on all the points of X and the lines to the left and the right of ℓ_1

For Wilber second bound, we use the following algorithm from [DHI+09].

1. Consider X sorted in increasing order over the y-coordinates.
2. For each point p in order,

(a) Consider the Orthogonal envelope of all points in the quadrant

$$\{(x, y)|x < p.x, y < p.y\}$$

(b) Consider also the Orthogonal envelop of all points in the quadrant

$$\{(x, y)|x > p.x, y < p.y\}$$

(c) Merge the two envelops.

(d) Sort the merged points according to y-coordinates.

(e) Draw a rectangle between any two consecutive points alternating between the two quadrants.

The resulted rectangles are indeed independent that yield a lower bound on $OPT(X)$.

In order to approximate $maxIRB(X)$ for any X, we will introduce the following definitions and algorithm.

Definition 6.6 (Signed Rectangles)**.** Given a $\square ab$, we say it is a (+)-rectangle ((−)-rectangle) if the slop of the line ab is positive (negative).

Definition 6.7 (Signed Satisfaction)**.** Given a set of point X, X is (+)-satisfied if all (+)-rectangle are arborally satisfied. Similarly for (−)-satisfied set.

Now, we can define two kinds of $minASS$ one for each type of rectangles. Thus, we will denote by $minASS_{(+)}(X)$ the size of the minimum (+)-satisfied super-set of X. Similarly, we can define $minASS_{(-)}(X)$. To compute $minASS_{(+)}(X)$, the authors of [DHI+09] propose the following algorithm. (The algorithm can be modified for the (−) case).

1. Consider a horizontal line sweeping by increasing y-coordinates.

2. Consider the points on the sweep line

3. For each point p,

   - For each unsatisfied (+)-rectangle formed with a point q bellow p, add the north-west corner of the rectangle that is on the sweep line to satisfy it.

4. Return $add_{(+)}(X)$ as the set of added points disregarding X.

The output of this algorithm is strongly related to the independent (+)-rectangles. More precisely, they are equal by the following lemma [DHI+09].

Lemma 6.4. *Given any point set X, there exists an independent set of $(+)$-rectangles, call it $IRB_{(+)}(X)$ such that*

$$|IRB_{(+)}(X)| = |add_{(+)(X)}|.$$

Combining this lemma with the following one.

Lemma 6.5. *Let I be an independent set of $(+)$-rectangles in a point set X, then for any $(+)$-satisfied super-set Y of X, we have*

$$|Y| \geq |I| + |X|$$

Thus, $|add_{(+)}(X)| + |X| \leq minASS(X) = OPT(S)$ for an access sequence whose $X = P(S)$. Similarly, we can apply the same reasoning on $|add_{(-)}(X)|$.

Consequently, we can define the algorithm signed-greedy that computes both $|add_{(+)}(X)|$ and $|add_{(-)}(X)|$, and it returns the maximum between them plus $|X|$. The output will serve as a lower bound over $minASS(X)$ and consequently $OPT(S)$.

A natural question is how much this algorithm, signed greedy, is close to maximum independent rectangles $maxIRB(X)$. It turns out that signed-greedy is within a constant factor from $maxIRB(X)$. More precisely,

$$signed - greedy(X) \geq \frac{1}{4}maxIRB(X) + \frac{1}{2}|X|.$$

7 Conclusion

To sum up, in this short overview on dynamic optimality. We presented an $\log\log n$-competitive BST, named *Tango Tree.* Furthermore, we discussed the geometric view of BST and its consequences. It brings a new possible algorithm that is different from *Splay Tree* that could be dynamically optimal.

References

[Cor09] T.H. Cormen. *Introduction to Algorithms, 3rd Edition:*. MIT Press, 2009.

[DHI$^+$09] Erik D. Demaine, Dion Harmon, John Iacono, Daniel M. Kane, and Mihai Patrascu. The geometry of binary search trees. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 496–505. SIAM, 2009.

[DHIP07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality - almost. *SIAM J. Comput.*, 37(1):240–251, 2007.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., third edition, 1997.

[Mor] Pat Morin. *Open Data Structures (in pseudocode), Edition: 0.1Gβ*.

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 235–245. ACM, 1983.

[STT86] Daniel Dominic Sleator, Robert Endre Tarjan, and William P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 122–135. ACM, 1986.

[Wil89] Robert E. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM J. Comput.*, 18(1):56–67, 1989.