Bifurcation: How to Explore a Tree

Sariel Har-Peled*

October 3, 2025

Abstract

Parametric search is a powerful but hard-to-use technique in computational geometry. Avraham et al. [AFK+15] presented a new approach, called *bifurcation*, that performs faster under certain circumstances. Intuitively, when the underlying decider execution can be rolled back cheaply and the decider has a near-linear running time. For some problems, this leads to fast algorithms that beat the seemingly natural lower bound arising from distance selection.

Bifurcation boils down to a tree exploration problem. You are given a binary (unfortunately implicit) tree of height n and k internal nodes with two children (all other internal nodes have a single child), and assume each node has an associated parameter value. These values are sorted in the inorder traversal of the tree. Assume there is (say) a node (not necessarily a leaf) that is the target node that the exploration needs to discover.

The player starts from the root. At each step, the player can move to adjacent nodes to the current location (i.e., one of the children or the parent). Alternatively, the player can call an oracle on the current node, which returns either that it is the target (thus, mission accomplished![1]) or whether the target value is strictly smaller or larger than the current one.

A naive algorithm explores the whole tree, in $O(nk)$ time, then performs $O(\log kn)$ calls to the oracle to find the desired leaf. Avraham et al. [AFK+15] showed that this can be improved to $O(n\sqrt{k})$ time, and $O(\sqrt{k}\log n)$ oracle calls.

Here, we improve this to $O(n\sqrt{k})$ time, with only $O(\sqrt{k} + \log n)$ oracle calls. We also show matching lower bounds, under certain assumptions. The new algorithm implies a minor improvement (i.e., roughly a $\log n$ factor) in the running times for bifurcation algorithms. Unfortunately, these algorithms are infested with large polylog terms, and this improvement is underwhelming. Nevertheless, we believe our interpretation of bifurcation as a tree exploration problem, and the associated algorithm, are of independent interest.

1. Introduction

Bifurcation. The bifurcation technique was first presented by Avraham et al. [AFK+15] (a somewhat more readable presentation is provided by Kaplan et al. [KKSS23]). A nifty recent paper using this technique is by Chan and Huang [CH25]. See the introduction of [CH25] for a recent literature review.

*Department of Computer Science; University of Illinois; 201 N. Goodwin Avenue; Urbana, IL, 61801, USA; sariel@illinois.edu; http://sarielhp.org/. Work on this paper was partially supported by NSF AF award CCF-2317241.

[1]The player can withdraw,

The input. An (n,k)-***tree*** $\mathcal{T}$ is a rooted tree with exactly k internal nodes having two children, and all its other interval nodes having only one child. Furthermore, all the leaves of $\mathcal{T}$ are at a distance of at most n from the root $\mathbf{r}$ of $\mathcal{T}$. An internal node with two children is a ***fork***. A fork has a ***left*** and ***right*** child, which readily induces an ordering on the nodes of the tree – by the DFS traversal of $\mathcal{T}$ that recursively visits the left child before the node itself, and then the right child. A special (unknown) node of $\mathcal{T}$ is the ***target***. The task at hand is to discover the target.

The tree is given implicitly – initially, only the root $\mathbf{r}$ is given. The algorithm can traverse the tree by walking on it locally. Namely, it maintains a current node $\mathbf{c}$ in the tree. In a single step, the algorithm can move $\mathbf{c}$ to any neighboring nodes, either upward (to its parent) or downward to one of its children (if it is a fork, or to its only child otherwise). When the algorithm enters a previously unexplored vertex, it is also given its status: a fork, a regular vertex, or a leaf.

Definition 1.1. For any node $x \in \mathcal{T}$, let $\pi(x)$ denote the unique *node-to-root* path from x to the root $\mathbf{r}$. For two nodes $u, v \in V(\mathcal{T})$, u is ***smaller*** than v, denoted by $u \prec_{\mathcal{T}} v$, if u appears before v in the inorder DFS traversal of $\mathcal{T}$.

The algorithm may consult (for free[2]) with an oracle.

Definition 1.2 (Target oracle). Given a query node $\mathbf{c} \in V(\mathcal{T})$, the ***target oracle*** returns one of the following:

(I) $\mathbf{c}$ is the target (the algorithm is done),

(II) the target is smaller than $\mathbf{c}$, or

(III) the target is larger than $\mathbf{c}$.

The challenge is to find the target using a minimal number of steps and oracle calls. We describe a minor variant of this algorithm here (with a minor performance improvement).

The "obvious" algorithm explores the whole tree first. As an (n,k)-tree have $O(nk)$ nodes. Thus, a straightforward DFS would take $O(nk)$ steps. One can then find the target using $O\big(\log(nk)\big)$ oracle calls (using binary search). This algorithm pays upfront for the exploration – to do better, one needs to mix the exploration with the binary search.

A natural way to do so is to set an upper bound on the depth of exploration – that is, the DFS does not continue the search below nodes of a certain depth, thus avoiding overpaying for exploration, an idea we use below. Specifically, in the ith round, one explores the tree up to depth $in/\sqrt{k}$, then uses binary search using the oracle to discover the leaf leading to the target, and continues the search into this leaf in the next round. This algorithm is (essentially) from Avraham et al. [AFK+15].

Our results. We describe a different approach that performs only partial decimation of the nodes in each round, leading to a faster algorithm. We also show matching lower bounds, first under the assumption that an oracle call costs $\Omega(n)$ time, and the second lower bound is under the assumption that the target is at a leaf, and oracle calls are allowed only on leaves. The latter lower bound leads to an interesting, and somewhat counter-intuitive, lower bound of $\Omega(\log^2 n)$, on searching in a complete binary search tree, where all the values are stored in the leaves, and one has to pay for each edge traversed during the search. See Lemma 3.3 for details.

[2]Well, if you were to bribe the oracle with a gift, it would not refuse you. But, really, cash is best in such cases, judging by Greek mythology.

2. The algorithm

In the following, all logs are in base 2. For integers $x < y$, let $[\![x : y]\!] = \{x, x+1, \ldots, y\}$ and $[\![y]\!] = [\![1 : y]\!]$.

2.1. Description of the algorithm

The algorithm maintains a subtree $\mathcal{C} \subseteq \mathcal{T}$, of the part of the tree $\mathcal{T}$ that was explored so far.

2.1.1. Basic operations

A node u of $\mathcal{C}$ marked as a *stub* can not have the target in its subtree $\mathcal{T}_u$. In particular, when a node is marked as a stub, all its children are removed.

Trimming. For a specified vertex $u \in V(\mathcal{C})$, the algorithm consults with the oracle to decide if u is the target. If so, the algorithm is done. Otherwise, consider the path $\pi(u)$ from $\boldsymbol{r}$ to u. If, according to the oracle, the target is bigger (resp., smaller) than u, then the algorithm marks all the direct left (resp., right) children of nodes of $\pi(u)$ in $\mathcal{C}$ as being stubs.

Halving. If the current tree $\mathcal{C}$ becomes too large (say, it has more than αn nodes, where α is a sufficiently large constant), it is useful to shrink it to roughly half its current size. For a leaf $u \in \mathcal{C}$, let $R(u)$ (resp., $L(u)$) be the set of all the vertices of $\mathcal{C}$ that are smaller (resp., larger) than u. Using DFS, the algorithm computes, in linear time in the size of $\mathcal{C}$, a node u, such that $\big||L(u)| - |R(u)|\big| \leq 1$ (i.e., the "median" in the inorder of the nodes of $\mathcal{C}$). Next, the algorithm applies the trimming operation to u, resulting in an updated $\mathcal{C}$ having roughly half of its original size – more precisely, if $\mathcal{C}$ had $m \geq \alpha n$ nodes, the new tree has at most $1 + (\alpha/2 + 1)n$ nodes (as the nodes on $\pi(u)$ are preserved). This algorithm requires $O(|\mathcal{C}|)$ time/steps, and one oracle call.

A straightforward modification of this procedure leads to an alternative version that halves the number of leaves in $\mathcal{C}$ (as a reminder, stubs are not considered to be leaves).

2.1.2. The algorithm in detail

Let ψ be a prespecified parameter – roughly, the number of oracle calls the algorithm would perform. For simplicity of exposition, assume the quantities

$$L = \frac{k}{\psi}, \qquad \Delta = \frac{2n}{\psi}, \qquad \text{and} \qquad \mathcal{W} = L\Delta = \frac{2kn}{\psi^2}.$$

are integers. Initially, the explored tree $\mathcal{C}_0$ contains only the root $\boldsymbol{r}$ of $\mathcal{T}$.

The ith round. In the beginning of the ith round, for $i = 1, \ldots, \psi/2$, the algorithm starts at the root of $\mathcal{C}_{i-1}$, and performs DFS up to depth $D_i = i\Delta$. The DFS does not continue through stubs but continues into the unexplored parts of $\mathcal{T}$ when reaching leaves of $\mathcal{C}_{i-1}$ at depth smaller than D_i. Let $\mathcal{T}_i$ be the resulting tree. All the leaves of $\mathcal{T}_i$ are at depth D_i. The algorithm now repeatedly performs halving on $\mathcal{T}_i$ till it has at most $\mathcal{W}$ nodes and at most L leaves. Let $\mathcal{C}_i$ be the resulting tree.

After the final round. The above results in a tree $\mathcal{C}_\psi$, that might have up to $\mathcal{W}$ nodes (and $k+1$ leaves). The target is found via a binary search on these nodes, performing $O(\log \mathcal{W})$ oracle calls.

2.2. Analysis

Lemma 2.1. *In the ith round, the total number of oracle calls is $o_i = O(1+f_i/L)$, and the total number of steps is $O\big(\mathcal{W} + f_i\Delta\big)$, where f_i is the number of new forks discovered in this round.*

Proof: During the computation of $\mathcal{T}_i$, each edge of $\mathcal{C}_{i-1}$ (which forms the upper part of $\mathcal{T}_i$) is traversed at most twice by the DFS process. Thus, the work involved in the DFS over $\mathcal{C}_{i-1}$ is bounded by $O(\mathcal{W})$, as the algorithm maintains the invariant that the tree computed in each round has size $O(\mathcal{W})$ and at most L leaves.

Each new fork discovered during the ith round can contribute an additional path of length Δ to $\mathcal{T}_i$ till the DFS either reaches a new fork, or the bottom level D_i. Thus, the total size of $\mathcal{T}_i$, and thus the total amount of work spent on exploration in this round, is bounded by

$$n_i = O(\mathcal{W} + \#_{\text{leafs}}(\mathcal{C}_{i-1})\Delta + f_i\Delta) = O\big(\mathcal{W} + (L+f_i)\Delta\big) = O\big(\mathcal{W} + f_i\Delta\big),$$

since $\mathcal{W} = \Delta L$. The halving, done at the end of the round, requires at most

$$o_i = O\left(1 + \frac{n_i}{\mathcal{W}} + \frac{f_i}{L}\right) = O\left(1 + \frac{(L+f_i)\Delta}{\Delta L} + \frac{f_i}{L}\right) = O\left(1 + \frac{f_i}{L}\right)$$

oracle calls. ∎

Theorem 2.2. *Given an implicit (n,k)-tree containing an unknown target, and a parameter $\psi \in [\![k]\!]$, the target can be found in $O(nk/\psi)$ time, using $O(\psi + \log n)$ calls to an oracle.*

Proof: The number of oracle calls done before the final cleanup stage is

$$\sum_{i=1}^{\psi} o_i = \sum_{i=1}^{\psi} O\big(1 + f_i/L\big) = O(\psi + k/L) = O(\psi)$$

since $\sum_i f_i = k$ and $L = k/\psi$. The final cleanup requires an additional $O(\log \mathcal{W}) = O(\log n)$ oracle calls. The total number of steps/work is

$$\sum_{i=1}^{\psi} n_i = \sum_{i=1}^{\psi} O\big(\mathcal{W} + f_i\Delta\big) = O(\psi\mathcal{W} + k\Delta) = O\left(\frac{kn}{\psi}\right),$$

as $\mathcal{W} = kn/\psi^2$ and $\Delta = n/\psi$. ∎

Corollary 2.3. *Given an implicit (n,k)-tree containing an unknown target node, the target can be found in $O(n\sqrt{k})$ time, using $O(\sqrt{k} + \log n)$ calls to the oracle.*

Remark 2.4. A natural variant of the above is when all the leaves are at depth n, the target must be a leaf, and the oracle queries are performed only on the leaves. In such a case, one can replace the $O(\log n)$ by $O(\log k)$ in the number of oracle queries, but that does not seem to be significant[3].

[3]Not clear anything in this writeup is significant.

3. Lower bounds

3.1. When the decider takes linear time

Lemma 3.1. *Assume that a call to the decider takes linear time (i.e., $O(n)$). Then, in the worst case, an algorithm exploring an (n, k) tree must take $O(n\sqrt{k})$ time.*

Proof: Consider the "complete" binary tree of height $h = \sqrt{k}$, where we replace each edge by a path of length $\Delta = n/\sqrt{k}$ (i.e., it is made out of Δ edges). Let $\mathcal{T}$ denote this tree.

The adversary strategy is now the following – whenever the oracle is called, it returns the answer that maximizes the number of vertices of the tree $\mathcal{T}$ that might contain the target. If the player had exposed k forks, then the adversary amends all the undiscovered forks in $\mathcal{T}$ into regular nodes by deleting one of their children (thus, after this "trimming" $\mathcal{T}$ has size at most $O(nk)$).

Now, if the player forced the adversary to trim $\mathcal{T}$, then it already spent $\Omega(k\Delta) = \Omega(n\sqrt{k})$ time on exploration as it had to traverse at least $k-1$ edges, each one of length Δ. Otherwise, the standard lower bound on binary search over $L = 2^{\sqrt{k}}$ elements, implies the algorithm must have issues $\Omega(\log L) = \Omega(\sqrt{k})$ oracle calls, which requires $\omega(\sqrt{k}n)$ time, as claimed. ∎

3.2. When the oracle can be consulted only in leaves

An interesting variant is that an oracle call still costs only $O(1)$ time, but one can perform such calls on leaves of depth n (here we assume the target is also a leaf of depth n).

3.2.1. Searching for a needle in a complete binary tree

Consider a complete binary tree $\mathcal{T}$ of height h. There are $\nu = 2^h$ leafs in $\mathcal{T}$, and let $i \in \mathcal{I} = [\![0 : \nu - 1]\!]$ be the label associated with the ith leaf in the inorder traversal of $\mathcal{T}$. We label the ith leaf with the binary string $B = \{0,1\}^h$ whose binary value is i (i.e., the binary string encodes the path from the root to this leaf). The target t is one of these leaves, and an oracle query must be one of the leaves of $\mathcal{T}$.

The player asks a sequence of queries, where a query is a label in $\mathcal{I}$. For a query q, the adversary either answers that $q \prec t$, $q \succ b$, or $q = t$ (i.e., the target t was found). After the ith iteration, there is an *active* range $R_i = [\![x_i : y_i]\!]$ where t might be, where $R_1 = \mathcal{I}$.

The pricing model. The ***rank*** of a node $x \in V(\mathcal{T})$ is the height of its subtree. Thus, a leaf of $\mathcal{T}$ has rank zero, while the root $\boldsymbol{r}$ has rank h. Given two strings $p, q \in B$, their *LCA distance* is the rank of their $\mathrm{lca}(p, q)$, and it is denoted by $d_{\mathrm{lca}}(q, q')$.

Given a sequence of queries $Q = \{q_1, \ldots, q_k\}$, let $Q_i = \{q_1, \ldots, q_i\}$ denote the ith prefix. The set of edges in the tree that must be traversed before one can use the oracle is the set

$$E(Q) = \cup_{q \in Q} \pi(q).$$

In particular, the ***price*** of the ith query is

$$|E(Q_i)| - |E(Q_{i-1})| = d_{\mathrm{lca}}(q_i, Q_{i-1}) = \min_{p \in Q_{i-1}} d_{\mathrm{lca}}(q, p).$$

And the overall price of Q is $|E(Q)| = \sum_{i=1}^{k} d_{\mathrm{lca}}(q_i, Q_{i-1})$.

Here, we show that the optimal strategy has a price $\Omega(h^2)$ in the worst case.

The adversary strategy. In the ith round, if the query $\beta_i \in \mathcal{I}$ is outside the current active range $R_i = [\![x_i : \beta_i]\!]$, the adversary returns the relation between β_i and (say) x_i (and sets $R_{i+1} \leftarrow R_i$). Otherwise, the query $\beta_i \in R_i$, and it breaks R_i into two ranges

$$A_i = [\![x_i : \beta_i - 1]\!] \qquad \text{and} \qquad B_i = [\![\beta_i + 1 : y_i]\!] .$$

The adversary answers "$t \prec \beta_i$" if $|A_i| > |B_i|$ (i.e., $R_i \leftarrow A_i$), and answers "$t \succ \beta_i$" (i.e., $R_{i+1} \leftarrow B_i$) otherwise. The game ends when the target is isolated, that is, $|R_{i+1}| = 1$.

For the simplicity of exposition, we assume the player is aware of the adversary strategy, so it is not going to waste time on useless queries – such as a query outside the active range, or at the start/end values in the current active range if it has more than two values (both assumptions can be removed with some tedium).

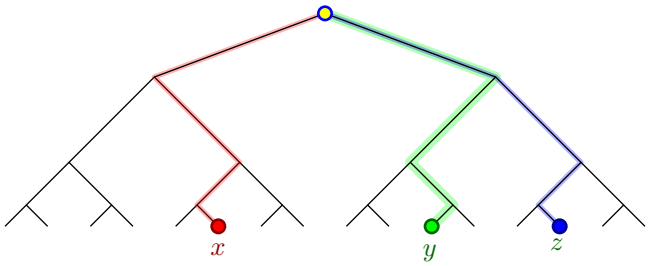

Figure 3.1:

Analysis.

Lemma 3.2. *Consider three strings $x, y, z \in B$, with $x < y < z$, where first the $\pi(x) \cup \pi(y)$ are queries were already performed (i.e., $Q = \{x, z, y\}$). Then, the price of y is $\rho = d_{\mathrm{lca}}(y, \{x, z\})$, where $\rho \geq \min\Big(\log(y - x), \log(z - y)\Big)$. and also $\rho \geq 1$.*

Proof: Assume the situation is as depicted in Figure 3.1, and $\rho = d_{\mathrm{lca}}(y, z) < d_{\mathrm{lca}}(x, y)$, as the other case follows by symmetry. Let $\alpha = z' - y'$. For a node of rank ρ, the distance between its two extreme leaves is $2^\rho - 1$. Thus,

$$2^\rho - 1 \geq \alpha \implies \rho \geq \lceil \log(\alpha + 1) \rceil = 1 + \lfloor \log \alpha \rfloor \geq \log \alpha. \qquad \blacksquare$$

Lemma 3.3. *Consider a complete binary search tree $\mathcal{T}$ of height h. Against an adaptive adversary, when queries can be issued only at the leaves, any game that ends with the target found has a price of at least $\Omega(h^2)$.*

Proof: By induction on the length of the active range. Let $P(\Delta)$ be the minimum price of any game carried out once the active range has Δ elements. The claim is that $P(\Delta) \geq c \log^2 \Delta$, where $c > 0$ is some constant, and $\Delta > 1$. The claim clearly holds for $\Delta \leq 10$, so assume $\Delta > 10$.

So, the game is at the beginning of the ith round, and let R_i be the active range with $|R_i| = \Delta$. Let j be the minimum index such that $|R_j| \leq |R_i|/2$ (observe that $|R_j| > \Delta/4$). Let $q_i, \ldots, q_{j-1}$ be the queries used in these rounds – we assume for simplicity of exposition that q_k is in the interior of R_k. Let $J_i, \ldots, J_{j-1}$ be the ranges thrown away. Specifically, $R_{k+1} = R_k \setminus J_k$, and q_k is an endpoint of J_k, for

all k. By the adversarial behavior, we have that $|J_k| \leq |R_{k+1}| + 1$ (the $+1$ is for the case where $|R_k|$ is odd, and q_k is the median).

For simplicity of exposition, assume that 0 and $2^h - 1$ are not in R_u, for all $u \in [\![i : j]\!]$. The range $R_k = [\![x_k : y_k]\!]$ was created because the player asked the queries $x_k - 1$ and $y_k + 1$ in some earlier rounds. Thus, the price of q_k is $\xi_k = d_{\mathrm{lca}}(q_k, \{x_k - 1, y_k + 1\})$, as the root path $\pi(q_k)$ is "trapped" between $\pi(x_k - 1)$ and $\pi(y_k + 1)$. Thus, by Lemma 3.2, $\xi_k \geq \log |J_{k+1}| \geq 1$, since $|J_{k+1}| \geq 2$. We have

$$P(\Delta) \geq \sum_{k=i}^{j-1} \xi_k + P(|R_j|) \geq \log \prod_{k=i+1}^{j} |J_k| + P(\Delta/4).$$

The latter is minimized by taking the refuse intervals J_k to be as large as possible (and minimizing their overall number). In particular, setting $|J_{i+1}|$ to its maximum possible value $\lceil \Delta/2 \rceil$, we have

$$P(\Delta) \geq \log \frac{\Delta}{2} + P(\Delta/4),$$

which readily implies that $P(\Delta) = \Omega(\log^2 \Delta)$. ∎

Lower bound for the (n, k)-tree case We assume $n > k$. The idea is to create an (n, k)-tree, by taking the complete binary tree of height $h = \sqrt{k}$, and replacing an edge by a path of length $\nabla = n/\sqrt{k}$. Clearly, the resulting tree is an (n, k)-tree, but it has way too many forks (i.e., $2^{\sqrt{k}}$ instead of k). The key insight is that this does not matter – as soon as the player visits k forks, the adversary has already won, and it can pretend that the rest of the unexplored tree never existed.

We change the charging scheme in the above for exploring (n, k)-tree – the player has to pay for all the steps made along a path, only when it arrives at a fork, and then it has to pay for all the vertices visited on this path. The user's strategy here can be interpreted as one applied to a complete binary tree. Lemma 3.3 implies that this strategy must visit $\Omega(h^2) = \Omega(k)$ forks. Still, the total price of the edges arriving at these forks is $\Omega(h^2 \nabla) = \Omega(nh) = \Omega(\sqrt{k}n)$.

Theorem 3.4. *Any algorithm that finds an unknown target in an (n, k)-tree, and issues queries only at the leaves, must make $\Omega(\sqrt{k}n)$ steps, in the worst case.*

Acknowledgments. The author thanks Timothy Chan and Micha Sharir for insightful discussions on this problem.

References

[AFK+15] R. B. Avraham, O. Filtser, H. Kaplan, M. J. Katz, and M. Sharir. The discrete and semi-continuous fréchet distance with shortcuts via approximate distance counting and selection. *ACM Trans. Algo.*, 11(4): 29:1–29:29, 2015.

[CH25] T. M. Chan and Z. Huang. Faster algorithms for reverse shortest path in unit-disk graphs and related geometric optimization problems: improving the shrink-and-bifurcate technique. *Proc. 41st Int. Annu. Sympos. Comput. Geom.* (SoCG), vol. 332. 32:1–32:14, 2025.

[KKSS23] H. Kaplan, M. J. Katz, R. Saban, and M. Sharir. The unweighted and weighted reverse shortest path problem for disk graphs. *Proc. 32nd Annu. Euro. Sympos. Alg.* (ESA), vol. 274. 67:1–67:14, 2023.