Competitive Online Search Trees on Trees

SODA 2020

Prosenjit Bose, Jean Cardinal, John Iacono,
Grigorios Koumoutsos, Stefan Langerman

Presenters:
黃光輝 R09922052
劉厚辰 R09922016
吳禹璇 R09944012

Outline

Introduction

Searching Vertices of a Tree

- Searching for an element that is not part of a linearly ordered set, but rather a vertex of a tree G.
- Generalize binary search trees to search trees on trees.

Online and offline search

- Given a search sequence $X = x_1, \ldots, x_m$, where each x_i is nodes of the BST.
- *Offline search*: the sequence X is known in advance and the rotations performed might be based on the knowledge of next request.
- *Online search*: each request x_i is revealed after the previous search x_{i-1} has been performed.

Introduction

Adaptive Binary Search Trees - BST Model of Computation

- The two actions below can be done using unit cost:
  1. A pointer moves from a node to a adjacent node.
  2. Rotation of a node.
- Example of such model: Red-Black tree[1], AVL-tree[2].

Adaptive Search Trees on Trees

- Adaptive by changing the search tree on tree has never been considered.
- **Goal: Generalize from BST to General Search Tree (GST) and consider the design of competitive online search trees on trees in this model.**

[1] LJ Guibas et al. A dichromatic framework for balanced trees. IEEE Computer Society, 1978.

[2] GM Adel'son-Vel'skii et al. An algorithm for the organization of information. 1962

Our Approach - From Binary to General Search Trees

- Inspired by the BST-model Tango trees with the notion of Steiner-closed (specific to the GST model).
- While entropy-based lower bounds fail in the GST model, we are able to adapt one of the lower bounds[3] which can matched by a factor $O(\log\log n)$ to our data structure using a **two-level decomposition**.
  1. Decompose a balanced search tree into preferred paths.
  2. Resorting to link-cut trees for handling the changes in preferred paths.

[3] Robert E. Wilber et al. Lower bounds for accessing binary search trees with rotations. SIAM, 1989

Our Results

- We define the GST model (generalize the BST model) which corresponds to the special case where the underlying tree is a path.
- Lower and upper bounds for GST model match the ones known for the BST model.

Lower Bound

Lower bound on the cost of any algorithm in the GST model is generalized from the *interleave lower bound* of BST[3] to search trees on trees.

[3]Robert E. Wilber et al. Lower bounds for accessing binary search trees with rotations. SIAM, 1989

Upper Bound

- An online algorithm for executing search sequences in search trees on trees that is $O(\log \log n)$-competitive even knowing all search requests in advance.
- **Idea:**
  - Connect the cost of the algorithm to the interleave lower bound.
  - Lower bound increases by 1, the algorithm incurs a cost at most $O(\log \log n)$.

- This is based on the paradigm for Tango trees[4].
- More ideas and techniques are involved:
  - Steiner-closed search trees
  - A subset of k vertices (defined as preferred path later) can be stored easily in a BST data structure that supports split and merge in $O(\log k)$ time.
  - Two-level decomposition involving link-cut trees[5] and show that the resulting data structure is a valid search tree on tree.

[4] Erik D. Demaine et al. The geometry of binary search trees. SODA, 2009

[5] Daniel Dominic Sleator et al. A data structure for dynamic trees. J. Comput. Syst. Sci., 1983

Related Work

Dynamic optimality of binary search trees

- *Dynamic optimality* conjecture for BSTs which posits the existence of $O(1)$-competitive online binary search trees.
- Although both splay trees and the greedy algorithm are conjectured to be $O(1)$-competitive, the best known upper bound on their competitive ratio is $O(\log n)$.
- The best competitive ratio known is $O(\log \log n)$, which is achieved by using *tango trees*.

- **Note:** Tango trees are designed to approximately match the *interleave lower bound*.[6] Thus, we are able to use this data structure to generalize to search tree on trees.

[6] Erik D. Demaine et al. Dynamic optimality - almost. SIAM, 2007

Definition 2.1 (Search Tree On A Tree)

A rooted tree T is a valid search tree on a given unrooted tree $G = (V, E)$ if the root r of T stores a vertex of G and the rooted subtrees of $T \setminus r$ are valid search trees on the connected components of $G \setminus r$.

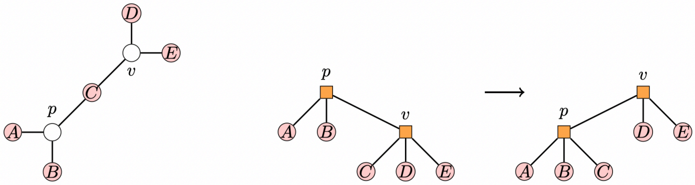
- T and G do not have degree restrictions.
- While T is rooted, there is no order among the children of a node.

- **Note:** In this work, we assume a fixed tree G unless otherwise indicated and n denotes the number of vertices in G.

Definition 2.2 (Rotation)

A rotation on a non-root node v of T is a local change which yields another search tree constructed as follows:

- Let p be the parent of v in T. Swap p and v in T.
- All children of p remain children of p.
- For a child u of v, let S_u be the set of nodes in its subtree. For at most one child u of v, there might be a node of S_u adjacent to p in G; then u becomes a child of p; all other children of v remain children of v.



(a) Two vertices p and v in G. (b) A rotation on v in the search tree on G.

Computation Model

Definition 2.3 (GST model of computation)

In the GST model of computation, we are given a tree G and we maintain a tree T which is valid search tree on G. At each time, there is a single node pointer at T. At unit cost we can perform the following operations:

1. Move the pointer to a child or the parent of the current node.
2. Rotate the current node v.

- A search operation for $v \in V$ is any sequence of unit-cost operations where the pointer starts at the root r of T and points to v at some point during the execution of the operation.

- **Note:** By this definition, we can see that the GST model is a generalization of the BST model of computation.

Computation Model

Definition 2.4 (Optimal)

Let $OPT(G,X)$ be the optimal cost of any GST-model algorithm to execute the sequence of searches X starting from any initial search tree T on G.

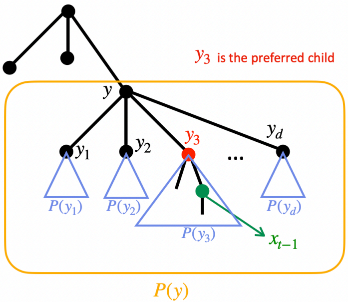
- Sequence $X = x_1, x_2, \dots, x_m$ is a valid search sequence in a tree $G = (V, E)$ if all $x_i \in V$.

Computation Model

Definition 2.5 (Preferred Child)

Let P be a valid search tree on G. Let y be a non-leaf node of P, with children $y_1, \ldots, y_d$. At each time $t \in [1, m]$, we define the preferred child of y to be the child y_i whose subtree $P(y_i)$ contains the most recent searched vertex in $x_1, \ldots, x_t$ that is in a node of $P(y)$ (or is undefined if none of these searches are in $P(y)$). In case last request in $P(y)$ is to y, we set preferred child of y to be y_1.

- Preferred child of a node changes throughout the execution of sequence X.
- **Note:** Preferred child of a node in a search tree is crucial for the lower/upper bound.



In this section, we show how to generalize the interleave lower bound of binary search trees to search trees on trees.

Definition 3.1 (Interleave Bound)

Let P be a valid search tree on G. The interleave bound of a node y of P is the number of times the preferred child of y changes over time $1, 2, \ldots, m$. The interleave bound $I(G, P, X)$ is the sum of the interleave bounds of the nodes.

- **Note:** P is a fixed search tree and does not change throughout the execution of X.

Theorem 3.1

Let P be a valid search tree on G. For any search sequence X in the GST model of computation, we have that $OPT(G,X) \geq I(G,P,X)/2 - n$.

- Let ALG be any GST-model algorithm. At a high-level, the proof consists of two main steps:

Step 1: We show that if for a fixed node y in P the interleave bound value is q, then there are at least $q/2 - 1$ unit-cost operations performed by ALG. We charge those operations to node y.

Step 2: We show that for two different nodes $y \neq z$ of P, the unit-cost operations charged to y and z are disjoint.

- The two steps imply the theorem; by summing overall nodes y of P, we get that ALG has cost at least $I(G,P,X)/2 - n$.

Proof of Theorem 3.1

Definition 3.2 (Dominating Node/ Subtree)

Let l_{i_t} be the node with smallest depth in T_t among $l_1, ..., l_d$, for some $1 \leq i_t \leq d$. Then, l_{t_i} is the lowest common ancestor in T_i of all nodes stores in $P(y)$. We call l_{i_t} the dominating node of $P(y)$ in T_t and $P(y_{i_t})$ the dominating subtree of $P(y)$

- Let T_t be the tree maintained by ALG after the tth search.
- y is a node of P of degree d with children $y_1, ..., y_d$
- $P(y)$ denote the subtree of P rooted at y.
- l_i is the node of subtree $P(y)$ with the smallest depth in tree T_t.

- **Note:** As the tree T_t evolves over time, the dominating subtree of y might change.

Proof of Theorem 3.1

Definition 3.3 (Transition Point)

Let l_{i_t} be the dominating node of $P(y)$ in T_t. For each $i \neq i_t$, we call l_i to be the transition point of y for $P(y_i)$ at time t.

Observation 3.1 (Property of transition points)

A transition point of a node $y \in P$ can not be the root of T_t, since l_{i_t} is its ancestor. Thus whenever ALG has to touch a transition point of y , it incurs a cost of at least 1.

Observation 3.2 (Property of transition points)

Let l_{t_i} be the dominating node of $P(y)$ in T_t. If the request x_{t+1} is to a node of subtree $P(y)$ in T_t. If the request x_{t+1} is to a node of subtree $P(y_i)$ for some $i \neq i_t$, then the transition point l_i has to be touched by ALG.

- **Note:** Given time t, we will have exactly $d - 1$ transition points of y.

Proof of Theorem 3.1

Proof of Step (1):

Assume $IB(y)$ equals q and any two consecutive requests x_{j_k}, $x_{j_{k+1}}$ are from different subtrees $P(y_k)$, $P(y_{k+1})$. We can consider two situation:

- Requests in non-dominating subtree: By Observation 3.2, when a node from a non-dominating subtree $P(y_i)$ is requested, the transition node l_i has to be touched. By Observation 3.1, at least one unit-cost operation has to be performed.
- Requests in dominating subtree: Since $P(y_k)$, $P(y_{k+1})$ are different, the dominating tree changed at least once during (j_k, j_{k+1}), which means there should have been a rotation between the transition point of y and the dominating point of $P(y)$. So, the transition point of y for $P(y_{k+1})$ is touched at least once during (j_k, j_{k+1}) and it will charge 1.

Proof of Step (1) (cont.):
Let $q1$, $q2$ be the number of requests to non-dominating and dominating subtrees of $P(y)$ and $q_1 + q_2 = q$. Consider two case:

- If $q_2 \leq \lceil q/2 \rceil$, we count only the unit-cost operations charged by q_1. We have that $q_1 = q - q_2 \geq q/2 - 1$.
- If $q_2 \geq \lceil q/2 \rceil$, we count the unit-cost operations charged by q_1 and the consecutive requests of q_2 which is the number of q_2 precedes q_1, that is $q_2 - q_1$. We have that $q_1 + (q_2 - q_1) \geq q/2 - 1$.

In conclusion, we charged at least $q/2 - 1$ requests.

Proof of Step (2):

Lemma 3.1.

At any given time t, each node v of T_t can be a transition node of at most one node y of P.

Proof.

- Take two nodes y and z of P, y is the ancestor of z.
- If the dominating subtree for y is the subtree including $P(z)$, transition points of y will not be in $P(z)$.
- Otherwise, transition point l for y and l is the lowest common ancestor of all points of $P(z)$. By Definition 3.2, l can not be a transition point for z. □

Since preferred child changes for node y are charged to touches of transition points for y. Lemma 3.1. implies no unit cost operation is counted twice by summing overall nodes. **We conclude that cost of ALG is at least $I(G, P, X)/2 - n$.** □

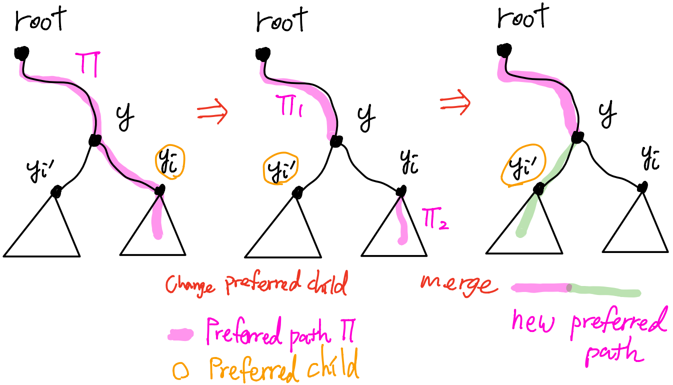
Preferred path

Let P be a fixed valid search tree of a tree G. Start from a node that is not the preferred child of its parent (or start from the root) and perform a walk by following the preferred child of the current node, until reaching a leaf. If the preferred child is undefined, pick one arbitrarily.

Tango Trees on Trees

Each change of preferred child during a search sequence results to changes in the preferred paths of P:

- Let y be a node in a preferred path Π. If y changes preferred child from y_i to $y_{i'}$, then Π splits into two paths Π_1 and Π_2.
- Then, Π_1 is merged with the preferred path previously rooted at $y_{i'}$.



Observation 4.1

During a search sequence X, there are at most $I(G, P, X) + n$ preferred path changes.

- The additive n stems from the fact that when the preferred child of a node v is undefined, we pick one of them arbitrarily in order to form a preferred path.
- Thus when the preferred child of v is defined for first time, a preferred path change might occur. Over all nodes there are at most n such preferred path changes.

Steiner closed sets and trees

Definition 4.1 (Convex Hull)

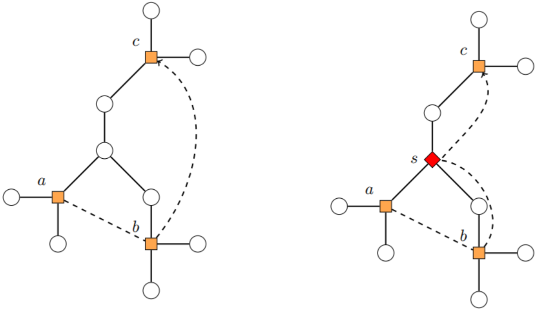
Given a tree $G = (V, E)$, for a set $S \subseteq V$ of vertices, we define the convex hull $CH(S)$ be the subgraph of G induced by the vertices on all paths $P(a, b)$, for all pairs of points a, b in S.

Definition 4.2. (Steiner-closed set)

A set S is a Steiner-closed set of vertices of a tree G provided that every vertex in $CH(S) \setminus S$ has degree exactly two in $CH(S)$.

Definition 4.3. (Steiner-closed tree)

A search tree T of a tree G is a Steiner-closed tree provided that the set of nodes on the path in T from the root to an arbitrary node in T is a Steiner-closed set with respect to G.



(a) A prefix of a path in a search tree on G that is not Steiner-closed.

(b) A Steiner-closed path.

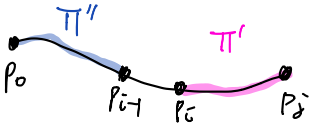
Figure 4: Steiner-closed paths.

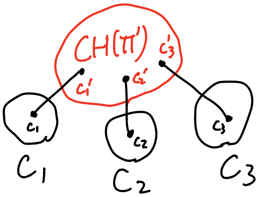
Lemma 4.1.

Let $\; = p_0, ..., p_j$ be a path from the root p_0 to a node $p_j$in a Steiner-closed search tree T of tree G. For any $i \in 1, ..., j$, let $\Pi' = p_i, ...p_j$. Removing $CH(\Pi')$ from $CH(\Pi)$ results in at most two connected components.

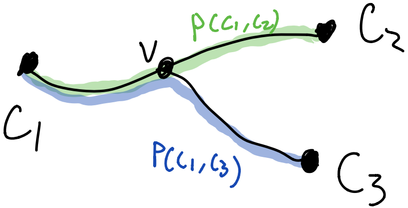
Proof:

- Let $\Pi'' = p_0, ..., p_{i-1}$.
- Suppose that removing $CH(\Pi')$ from $CH(\Pi)$ in G results in at least 3 connected components, denote C_1, C_2 and C_3, respectively.
- Let $c_i c_i'$ with $i \in \{1, 2, 3\}$ be the cut edges that connect C_i to $CH(\Pi')$ with $c_i \in C_i$ and $c_i' \in CH(\Pi')$.





- Let $P(c_1, c_2)$ be the path in $CH(\Pi)$ from c_1 to c_2 and $P(c_1, c_3)$ the path from c_1 to c_3.
- Let v be the first vertex where $P(c_1, c_2)$ and $P(c_1, c_3)$ diverge.
- Note that $v \notin \Pi''$. However, $v \in CH(\Pi'')$ since c_1, c_2 and c_3 are in Π''.
- Moreover, $deg(v) \geq 3$ in $CH(\Pi'') \Rightarrow$ **Contradiction!!** (Violate Definition 4.2) $\square$



Lemma 4.2

Given a valid search tree T on a tree G, we can create another valid search tree T' of G, such that T' is Steiner-closed and $height(T') \leq 2height(T)$.

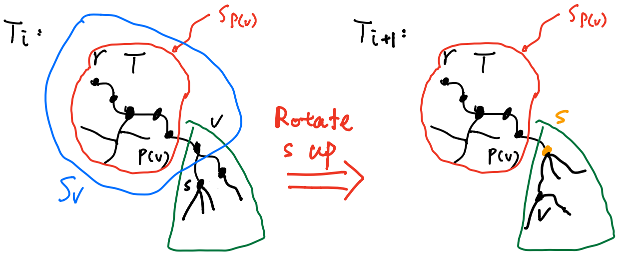
Proof:

- Perform a depth-first search on T and build our Steiner-closed tree incrementally.
- Let r be the root of T. For any non-root node v with parent $p(v)$, let S_v be the set of elements in the path from the root r to v.
- At each step i we transform the tree T_i into the tree T_{i+1} such that $T_0 = T$ and $T_{final} = T'$.
- Consider in ith step of transformation, our DFS visits a node $v \in T$ such that:
  1. The set $S_{p(v)}$ is Steiner-closed in G.
  2. The set S_v is not Steiner-closed.

  This means that $CH(S_v)$ contains a unique vertex $s \in CH(S_v) \setminus S_v$ with degree at least 3.

Steiner closed sets and trees

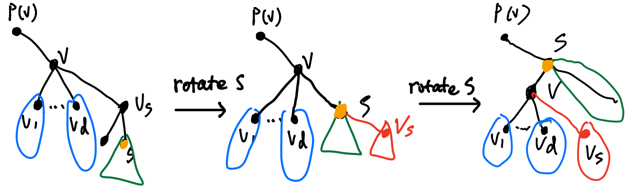
- Observe that the vertices on the path between $p(v)$ and v in G are contained in the subtree rooted at v in T_i. Since s is on this path, it is in the subtree rooted at v in T_i .
- We obtain T_{i+1} by rotating s up the tree until it is between $p(v)$ and v, that is, we make it a parent of v and a child of $p(v)$.



Now, in T_{i+1} the path from v up to root is now Steiner-closed by construction.

Steiner closed sets and trees

- Let v_s be the child of v in T_i such that s is in the subtree rooted at v_s. Let $v_1, \ldots, v_d$ be the other children of v.
- From T_i to T_{i+1}, the depth of all nodes in subtrees rooted at $v_1, \ldots, v_d$ increases by 1 and the depth of all other nodes of T_i does not increase.
- For a node w in a subtree rooted at $v_j, 1 \leq j \leq d$, we have that the depth of w increases by 1, the new root(v_j)-to w path is the same as before augmented by node s and the path from root to v is Steiner-closed.

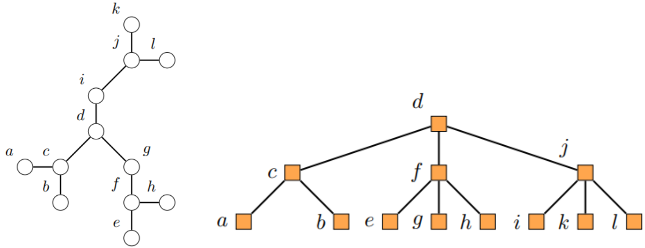


- All possible depth increases of w are caused by nodes in the path between v and w.
- Summing overall changes, for any node of the tree at depth d in T_0, its depth can increase by 1 at most d times, i.e., $height(T') \leq 2d = 2height(T)$. □

Lemma 4.3

Given a tree G, there is a search tree C of G with height at most $\log_2 n + 1$. The tree C is a centroid decomposition tree obtained by recursive application of Jordan' s theorem [Jor69, Har69]: Given a tree G with n vertices, there exists a vertex whose removal partitions the tree into components, each with at most $n/2$ vertices.

Note: A centroid decomposition can be computed in time $O(n \log n)$.



(a) A tree G.

(b) The centroid decomposition of G.

Using Lemma 4.2 and Lemma 4.3 we get the following corollary.

Corollary 4.1

For any tree G on n vertices, there exists a valid search tree P on G which is Steiner-closed and it has height at most $2\log n + 2$.

Proof:

Since centroid decomposition C has height $\log n + 1$, thus the tree P can be obtained by applying Lemma 4.2 for $T = C$. The tree P will be our reference tree in the rest of this paper. □

Building the Reference tree

Observation 4.2

If a search tree T of a tree G is Steiner-closed, then for all nodes v in T, the subtree T_v rooted at v is also Steiner-closed.

Definition 4.4

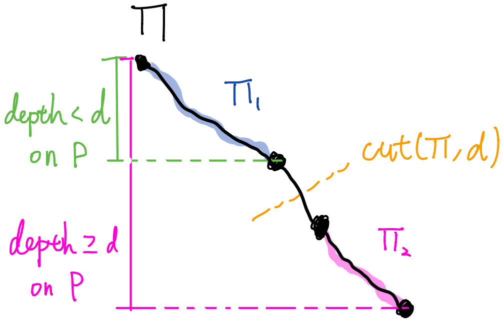
Let $\bar{P}(a,b)$ denote the set of nodes $v \neq \{a,b\}$ of the path from a to b.
For a Steiner-closed set of vertices S of G, let $G(S)$ to be the graph with vertex set S where two vertices $a,b \in S$ are connected by an edge iff. no $c \in S$ is in $\bar{P}(a,b)$.

Lemma 4.4

For any Steiner-closed set S, $G(S)$ is a tree.

Proof: Follows from Definition 4.3. □

- During an execution of a search sequence we need to perform the following operations on preferred paths:
  - (i) Search for a node in a preferred path Π.
  - (ii) Cut a preferred path Π into two paths, one consisting of nodes of depth smaller than d in P and the other of nodes of depth at least d. We denote this operation $Cut(\Pi, d)$.
  - (iii) Merge two preferred paths Π_1 and Π_2, where the bottom node of Π_1 is the parent of the top node of Π_2.



Maintaining preferred paths with link-cut trees

Let Π be a preferred path containing a Steiner-closed set of nodes S.

- Split Π into two paths:
  Split $G(S)$ into two tree $G(S_1)$ and $G(S_2)$ where S_1 and S_2 are the nodes in Π_1 and Π_2. By Observation 4.2, we can know Π_2 is also Steiner-closed, which implies $G(S_2)$ is a tree.
- Merge two preferred paths Π_1 and Π_2 into Π:
  We can construct the tree $G(S)$ where S is the union of the sets of nodes S_1 and S_2 in the paths Π_1 and Π_2.

Note: By Lemma 4.1, we can get $G(S_2)$ by cutting at most two edges of $G(S)$ and $G(S)$ can be obtained by cutting $G(S_1)$ at most two places and linking $G(S_1)$ and $G(S_2)$ by two edges.

Basic operations that need to be supported in logarithmic time

- We need to implement a data structure supporting the above operations on the forest of trees $G(S)$ at $O(\log k)$ cost in the GST model.
- Each of these operations can be split into a constant number of one of these two operations:
  1. *Cut* a tree into two by removing an edge.
  2. *Link* two trees into one by adding an edge.
- Resort link-cut trees data structure from Sleator and Tarjan
  - Heavy-path decomposition on the represented trees. Each heavy path represented by a splay tree.
- Data structure eventually consists of a hierarchy of splay trees, each representing a path in a tree $G(S)$, which corresponds to a path in the reference tree P.

Basic operations that need to be supported in logarithmic time

- Check the whole data structure is a search tree on G and the binary search tree operations are elementary operations in the GST model.
- Considering the preferred path Π in P with nodes S and the heavy path in the decomposition of $G(S)$.
  - Searching in a splay tree amounts to searching along a path of $G(S)$, whose convex hull is a path in G. By Observation 2.2, it is a proper search in the GST model.
  - Similarly, rotations in splay trees are rotations of the search tree on G as defined in the GST model

Proposed Algorithm

- Given the graph G, we construct a balanced Steiner-closed search tree P on G, which we refer to as the reference tree.
- We dynamically maintain a decomposition of P into preferred paths.
- Each such preferred path with nodes S corresponds to an unrooted tree $G(S)$, which is a minor of G.
- As searches are performed, preferred paths are updated, and these updates correspond to linking and cutting trees $G(S)$. For this, we use link-cut trees.
- Those in turn decompose the trees $G(S)$ into paths and reduce the operations to link and cut on paths. These operations can be handled by splay trees.
- Together, they form a search tree on G.

Lemma 4.5

Let l be the number of preferred child changes during a search. Then the cost of this search is $O((l+1)(1+\log\log n))$.

Proof:

During the search, the pointer touches exactly $l+1$ preferred paths. We account separately for the search cost and the update cost.

- **Search cost:** For each preferred path touched, the search cost is $O(\lceil \log\log n \rceil)$. Thus the total search cost is clearly $O((l+1)(1+\log\log n))$.
- **Update cost:** Time for cut and merge preferred paths on k nodes: $O(1+\log k)$. Since each preferred path has at most $O(\log n)$ nodes, we can perform those updates in $O(1+\log\log n)$. There are l preferred path changes, and there are one cut and and one merge operation for each change. So the total time for merging and cutting is $O(l\cdot(1+\log\log n))$. $\square$

Finally, combine Lemma 4.5 with Theorem 3.1, to get the competitive ratio of Tango Search Tree (TST).

Theorem 4.1

For any X of length $m = \Omega(n)$, Tango Search Tree are $O(\log \log n)$-competitive.

Proof:

Note: Account only for the cost during searches, since the cost of transforming the input tree into a valid TST is just a fixed additive term that doesn't depend on X.

- By Obs. 4.1, the total number of preferred path changes is at most $I(G, P, X) + n$.
- For all search requests, we get that the cost of Tango Search Tree

$$\sum_{x_i \in X} \overbrace{(l_i + 1)(1 + \log \log n)}^{\text{Lemma 4.5}} = (I(G, P, X) + n + m)(1 + \log \log n)$$

$$\stackrel{\text{Thm. 3.1}}{=} O(OPT(G, X))(1 + \log \log n) \quad \square$$