Dynamic Trees with Alternative Search Trees

Xue Zhenliang
Department of Computer Science, Fudan University

Abstract. We present an analysis for weighted treaps in link-cut trees (LCT) with amortized logarithmic time bounds and conduct an empirical study that investigates the discrepancies between different underlying data structures in LCT. We implemented three variants of LCT with biased search trees, self-adjusting search trees and weighted treaps and ran several experiments on them. Splay trees are the most efficient in LCT with amortized time bounds in our experiments while there is a 1.2 to 2 times slowdown in treaps and biased trees.

Keywords: algorithms, data structures, dynamic trees, randomized search trees, self-adjusting search trees, biased search trees, experimental evaluation.

1 Introduction

Dynamic tree problem focuses on the maintenance of a collection of trees under several operations, including adding an edge, removing an existing edge, designating tree roots (for rooted forest) and querying on the unique path between two nodes. Data can be stored on either nodes or edges. For simplicity, a new node can represent an edge in the original trees in order to put all data only on nodes. Dynamic trees have numerous applications especially in graph theory, such as network flows[13][21][23] and dynamic graphs[8][10][11][14][18][25].

Dynamic tree problem has been well studied in literature, and there are many data structures that solve this problem, such as link-cut trees (LCT)[21][22], topology trees[10][11][12], Euler-tour trees (ETT)[14][23] and top trees[2][3][26]. They all use search trees as underlying data structures but with different techniques: LCT uses path decomposition; ETT uses linearization; both topology trees and top trees use tree contraction.[25] LCT seems to be the first complete solution to the original dynamic tree problem and has been applied to many combinatorial algorithms. Although all the data structures guarantee logarithmic time bounds for most dynamic tree operations, their efficiency differs in many aspects, due to the overheads of non-sequential accesses, heavy memory allocations and huge constant factors hidden in the asymptotic complexities. Meanwhile, the choice of underlying data structures may have significant impacts on the performance of dynamic trees due to different methods of implementations. There have been some reports on the performance issues between various dynamic trees[25], but there seem to be few investigations in underlying data structures. In our research, we attempted to conduct empirical studies on the discrepancies of underlying search trees in LCT. The reason for our choice of LCT is that LCT is one of the lightest data structures among all variants of dynamic trees, since the overheads are primarily from the JOIN and SPLIT of search trees rather than LCT itself. In previous studies, researchers usually use biased search trees[21] and splay trees[22] to maintain solid paths. The maintenance of solid paths relies heavily on biased searches, which implies alternatives such as weighted treaps[20][7]. In this paper, we provide an analysis for weighted treaps as implementation of underlying search trees in LCT, which should have amortized logarithmic time bounds.

The remainder of this paper is organized as follows. Section 2 introduces dynamic tree problem and outlines the technique of path decomposition and LCT. Section 3 reviews biased search trees, splay trees and randomized search trees, i.e. treaps, in the setting of dynamic trees and we will sketch the proof for treaps in this section. Section 4 describes the platform setup used for comparisons and presents the experiments with discussions of results. The last section contains the overall conclusions and implications for further researches. Appendix A contains graph terminologies used in this paper.

2 Dynamic Trees

2.1 Dynamic Tree Problem. The original dynamic tree problem[21] is aimed to maintain the structures of a collection of vertex-disjoint rooted trees under three kinds of operations: LINK, CUT and EVERT. Initially there are n single-vertex trees and no edge in the forest.

- LINK$(x,\ y)$: Put a new edge $(x,\ y)$ and thus combine two trees into one. For rooted forest, we designate the root of the original tree that contains node x as the root of the new tree.
- CUT(x): Assuming that node x is not the tree root, remove the edge $(x,\ p(x))$, and thus divide the tree into two parts. One part contains $p(x)$ and the other part contains x and x is the tree root.
- EVERT(x): Let node x be the new tree root. If x is already the tree root, it will be ignored. This operation actually does not alter the structure of the tree but changes the ancestor-descendant relations, making all other nodes the descendants of x.

In addition to operations that may alter tree structures, data can be associated with both nodes and edges, which enables queries on the chain between any two nodes. For simplicity, we can create new nodes to supersede the edges in original trees. Therefore, we can stored the data associated with edges on the corresponding nodes. Such simplification only leads to at most n new nodes, which has no effect on the asymptotic complexities. Thereby we assume only nodes may contain additional data.

2.2 Path Decomposition. In this paper, we focus on *link-cut trees*, or LCT as the abbreviation, which was introduced by Sleator and Tarjan[21] and uses a technique called *path decomposition*. In path decomposition, each edge is marked as either solid or dashed and we guarantee that no more than one solid edge entering any node (solid edge invariant). Consecutive solid edges form a solid path. It is trivial to see that any solid path is within an access path. Intuitively, solid paths do not bend in the tree that the depths of nodes along the path are monotone. In this way, a tree is decomposed into a set of solid paths connected by dashed edges. In the remainder of this paper, we use the notion $c(x)$ to represent the chain containing x.

In order to implement basic dynamic tree operations, we need two fundamental operations:

- SPLICE(x): Provided that x is the topmost node of $c(x)$ and not the tree root, if there is a solid edge from $p(x)$, we make this edge dashed. Thereby we are able to mark edge $(x,\ p(x))$ as solid to connect $c(x)$ and $c(p(x))$ without violating the solid edge invariant.

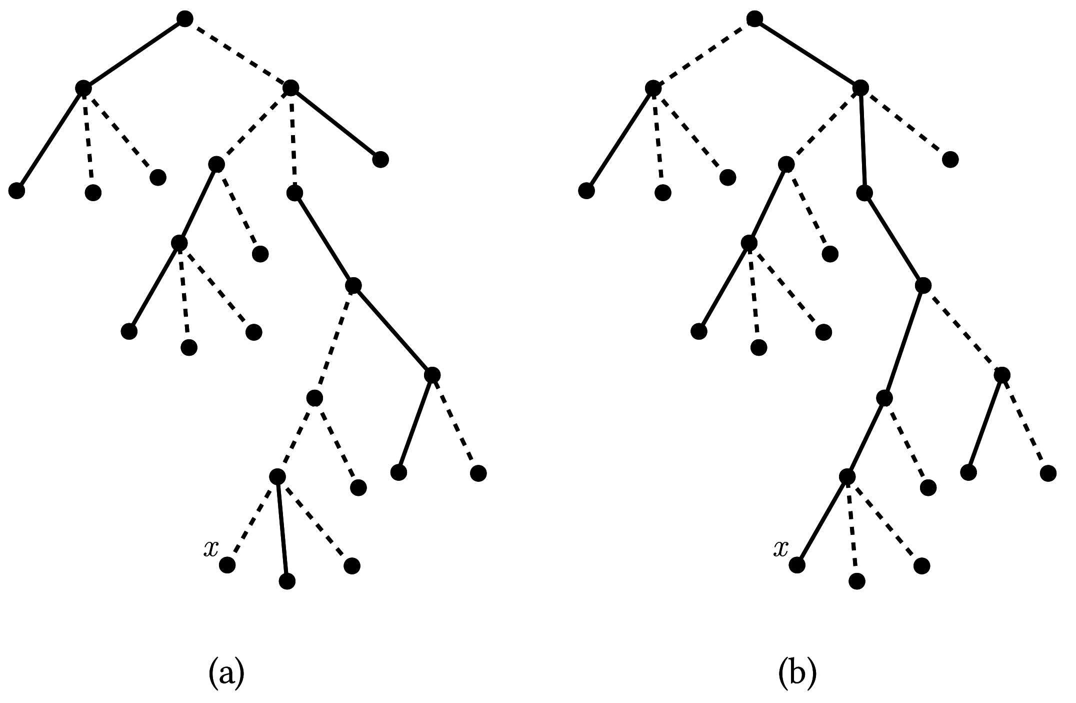

Fig.1. (a) An example of LCT. (b) LCT after $\text{Expose}(x)$.

- $\text{Expose}(x)$: Make the access path of x a solid path and all edges incident to this path dashed. This operation typically utilizes Splice.

With Splice and Expose, other dynamic tree operations can be viewed as combinations of two operations.[21]

Since the depth of node in a solid path ascends from top to bottom, nodes can be kept in search trees such as biased search trees[21] and self-adjusting search trees[22]. For Evert, a reversal bit is supposed to be attached to each internal node in search trees. Because Splice requires splitting and concatenating of search trees, we describe Split and Join formally as follows:

- $\text{Split}(r,\ x)$: r is a search tree and x is the target node to be divided. We divide the tree into 1) at most three parts $(L,\ I,\ R)$ where L denotes all nodes prior to x and R denotes all nodes after x and $I = \{x\}$ if x exists in the search tree and otherwise $I = \varnothing$ (3-way Split); 2) two parts $(L,\ R)$ where L denotes all nodes prior to x and x itself if existed and R denotes all nodes after x (2-way Split). In both cases, L and R are still search trees.
- Join: Given two search trees x and y where all elements in x precede those in y (2-way Join $(x,\ y)$), or two search trees x and y along with a new node k placed between the last node of x and the first node of y (3-way $\text{Join}(x,\ y,\ k)$), they are merged into one search tree with correct connections. A 3-way join is usually interpreted into two 2-way joins.

Remark. Biased trees only support 2-way splits and 2-way joins while self-adjusting trees allow both variants of splits and joins.

One of the most prominent properties of LCT is illustrated by the following theorem.

Theorem 1. If the overall number of Link, Cut and Expose is m, then there are at most $m(3\lfloor\log n\rfloor + 1)$ Splice.[21]

Corollary 1. If the time complexity of Splice is $\mathrm{O}(f(n))$, then Link, Cut, Evert and Expose take

$O(f(n)\log n)$ amortized time. Particularly, LCT has $O(\log^2 n)$ amortized time bounds with arbitrary acceptable search trees with logarithmic time bounds.

Proof. Directly derived from Theorem 1. ∎

3 Reviews and Analyses

Among three schemes of LCT, the concept of *ranks* is used in analyses. The rank of x, or $r(x)$, usually equals to $\Theta(\log s(x))$. In amortized analysis, ranks can be viewed as credits assigned to nodes and therefore ranks are supposed to be non-negative values. One credit can be used to perform a $\Theta(1)$ procedure so that the overall running time is measured by the number of credits allocated in the algorithm. In order to evaluate ranks, we shall associate each node with a weight $w(x)$ explicitly or implicitly, where

$$w(x) = \begin{cases} s(x) - s(u) & \text{(if solid edge from } x \text{ enters } u\text{)} \\ s(x) & \text{(otherwise)} \end{cases}$$

and then $s(x)$ can be calculated by aggregating over the subtree x in search tree. We may update weights in SPLICE and other operations that may alter the tree structure. LCT keeps the invariant that total weight of an underlying search tree equals to the size of corresponding subtree.

3.1 Biased Search Trees. Bent et al.[4] proposed biased 2-b trees and Feigenbaum et al.[9] refined their work and examined biased a-b trees. Biased trees are intended to solve biased dictionary problem[9], where items are assigned with access frequencies, i.e. weights, and the search tree is restructured based on the weights in order to achieve minimal total access time. In biased trees, an access to x takes $O(\log W/w(x))$ time, where W is the overall weight in search tree. Such accesses are called ideal accesses[4][9]. Bent et al.[4] showed that JOIN(x, y) takes only $|r(x) - r(y)| + 1$ credits and the amortized time to SPLIT at node x is proportional to the access time to x. These time bounds are excellent and with a careful analysis on credit changes in EXPOSE, Sleator et al.[21] proved that the amortized running time over a sequence of m dynamic tree operations is $O(m \log n)$. Although biased trees are the first data structures applied to dynamic trees, the complicated case analysis[4] leads to the inefficiency of biased trees. Thus biased trees are not preferred in dynamic trees.

3.2 Splay Trees. One of the most well-known self-adjusting search trees is the splay tree, introduced by Sleator and Tarjan[22]. Compared to conventional search trees such as AVL tree and red-black tree, both of which rely on extra information maintained in search tree nodes to preserve low tree height, splay tree adjusts itself by tree rotations whenever accessed and manages to gather all frequently accessed nodes in the vicinity of search tree root, optimizing for future accesses. Splay trees utilize a variation of move-to-root heuristic[5], i.e. the so-called SPLAY, that unexpectedly improves the performance. The original move-to-root heuristic simply moves the node to root by rotating once at each step, while SPLAY may rotate at most twice at each step. Provided that all splay tree operations are based on SPLAY, the only non-trivial work is to devise the upper bound of rotations in SPLAY, which is depicted by the Access Lemma[22].

Lemma 1. (Access Lemma) The amortized time to splay a tree with root t at a node x is at most $3(r(t) - r(x)) + 1 = O(\log s(t)/s(x))$.[22]

The magic SPLAY actually predicts the weight of each node based on previous accesses instead of keeping them explicitly. The wonderful property of implicit weights enables us to obtain heterogeneous time bounds under different circumstances without the great cost of “re-weighting”. Splay trees have been integrated into LCT to allow simple and efficient implementations and Sleator et al.[22] showed that such LCT achieved amortized logarithmic time bounds. However, splay trees show a significant increase in changes to tree structures, which contributes to the huge overheads of node updating and slows down splay trees when accesses are the primary operations, since other search trees do not alter the tree structure while accessing.

3.3 Randomized Search Trees. Seidel and Aragon[20] thoroughly explored randomizations in balanced search trees and coined the term “treap” for search trees with random priorities on nodes. In addition to obeying symmetric order of search trees, treaps use tree rotations to stipulate that nodes with larger priorities are not descendants of those with smaller priorities. Seidel[20] claimed that an access to x can be completed in expected $\mathrm{O}(\log W/w(x))$ time if we take the maximum value of $w(x)$ randomly generated numbers as the priority for each x, or store $(\log r)/w(x)$ instead for priority comparisons where r is randomly generated. Weighted treaps are our main concerns, since they support ideal accesses. The first method mentioned above does not support fast re-weighting while the second method involves floating point numbers that are not as efficient as fixed-size integers on modern computers, so we prefer *implicit priorities* in this paper, which are simulated according to subtree sizes. In $\textsc{Join}(x,\ y)$, we toss a coin with bias $p = s(x)/(s(x) + s(y))$ so that the probability that the priority of x is larger than that of y is p. Assuming that priorities have already computed in $\textsc{Join}$, implicit priorities are equivalent to ordinary priorities in analysis.

Applying weighted treaps to LCT is feasible, as we will show in the remainder of this section. In our analysis, any node can represent the treap containing it, and $s(x)$ is the overall weight in the treap x, even if x is not the tree root. Let $r(x) = \log s(x)$ and m be the number of dynamic tree operations. We distinguish *special splices* from *normal splices* that a special splice at x marks a new solid edge $(x,\ p(x))$ when $p(x)$ is the tail of $c(p(x))$, rather than switching solid edges in normal splices.

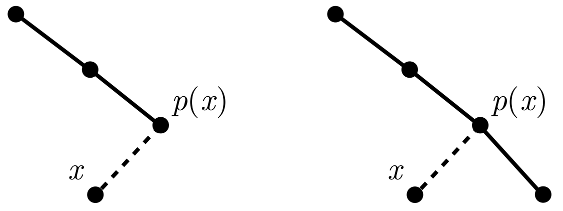

Fig.2. Special splice (left) and normal splice (right). Both splice at x and the edge $(x,\ p(x))$ will be solid.

Since every special splice induces a new solid edge, it is suggested that there seem to be few special splices. Sleator et al.[21] showed that there are at most m special splices, i.e. one $\textsc{Splice}$ per $\textsc{Expose}$ on average. It indicates that special splices are not the major overheads in $\textsc{Expose}$. To carry out the amortized analysis, we store $\Theta(r(x) - r(y))$ *credits* on each node y where x is the parent of y in treap. Meanwhile, the root of a treap may also have credits. We say treap u is *cast to* $c \geqslant r(u)$ iff. $\Theta(c - r(u))$ credits have been allocated on u. We often refer to these two properties as *credit invariants*. We will devise two key properties pertaining to $\textsc{Join}$ and $\textsc{Split}$. Proofs for the next two lemmas are deferred to Appendix B and Appendix C.

Lemma 2. (Join Lemma) If treap z is obtained by $\textsc{Join}(x,\ y,\ k)$ and both treap x and treap y are cast to $c \geqslant r(z)$, then treap z is also cast to c. $\Theta(c - r(k))$ new credits are required to perform the $\textsc{Join}$.

Lemma 3. (Split Lemma) Treap u and treap v are obtained via $\textsc{Split}(x,\ y)$. If treap x is cast to $c \geqslant r(x)$ and we allocate $\Theta(c - \log w(y))$ credits, both treap u and treap v are cast to c.

Finally we investigate credit changes in EXPOSE.

Lemma 4. (Expose Lemma) EXPOSE takes $O(\log n)$ amortized time.

Proof. We interpret $\text{EXPOSE}(u_0)$ into four phases.

(1) Setup: Turn the solid edge from u_0 to dashed by a 2-way SPLIT. It takes $O(\log n)$ time.

(2) Split Phase: Traverse from u_0 to the tree root and label the head nodes of visited chains with u_1, u_2, ..., u_p (u_p is the tree root). Parents of them are v_1, v_2, ..., v_{p-1} and $\varnothing$. In step i, we SPLIT at v_i and thus $c(v_i)$ is divided into at most three parts represented by u_{i+1} (possibly empty), v_i and q_i (for normal splices). Let k be the number of special splices in $\text{EXPOSE}(u_0)$ and $r'(u_{i+1})$, $r'(v_i)$, $r'(q_i)$ be the ranks of u_{i+1}, v_i, q_i respectively after SPLIT. Note that $k = O(1)$ in amortized sense and $r'(v_i) \geqslant r(u_i)$. Treap u_{i+1} prior to SPLIT is obviously cast to $r(u_{i+1})$. Split Lemma implies that it takes only $O(r(u_{i+1}) - r'(v_i)) = O(r(u_{i+1}) - r(u_i))$ amortized time for both special and normal splices. If the splice is normal, it leaves $\Theta(r(u_{i+1}) - r'(q_i))$ credits on q_i.

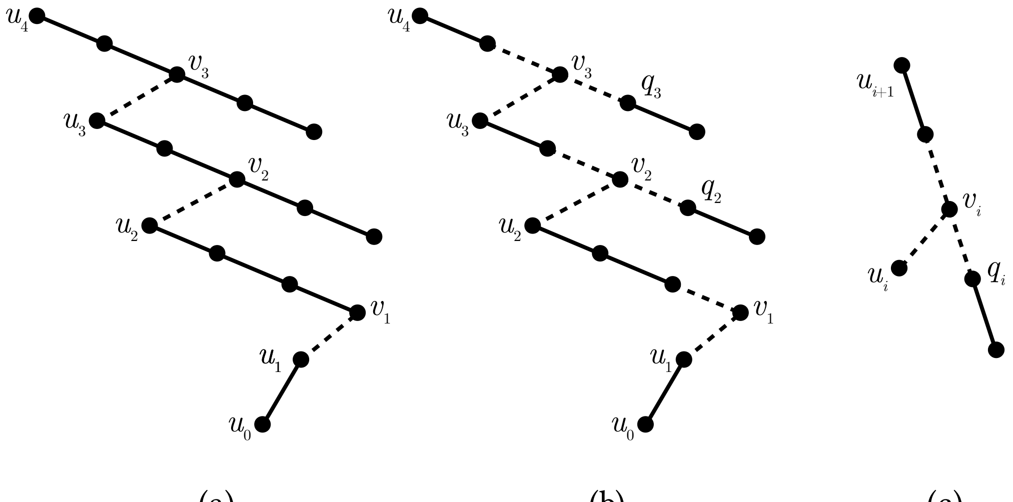

Fig.3. (a) Prior to Split Phase. (b) After Split Phase. (c) An example of normal splice during step i.

(3) Re-weight: $w(v_i)$ is changed to $s(v_i) - s(u_i)$. Let $r''(v_i)$ be the new rank of v_i. It takes constant time.

(4) Join Phase: Perform 3-way $\text{JOIN}(u_{i+1}, u_i, v_i)$ in step i. We have to place $\Theta(r(u_{i+1}) - r(u_i))$ credits on u_i to make treap u_i cast to $r(u_{i+1})$ and thereby JOIN only takes $O(r(u_{i+1}) - r''(v_i))$ amortized time by Join Lemma. In normal splices, saved credits from q_i can eliminate the time consumed by JOIN since $r''(v_i) \geqslant r'(q_i)$. Therefore only special splice takes additional $O(r(u_{i+1}) - r''(v_i)) = O(\log n)$ time.

The sum of costs of all steps telescopes and gives a bound of $O(p + r(u_p) - r(u_1) + (k+1)\log n)$. By Theorem 1 and the fact $k = O(1)$ and the definition of rank, the bound turns out to be $O(\log n)$. ■

Our result is stated by the following theorem, which is derived immediately from Expose Lemma.

Theorem 2. If weighted treaps are used as underlying data structures in LCT, then all dynamic tree operations have amortized $O(\log n)$ upper bounds with high probability.

Finally we emphasize that the bounds are not averaged on worst-case operation sequences due to the randomness of treaps.

4 Experiments

4.1 Experimental Setup. In this section we present an empirical study of impacts of different underlying data structures on LCT. Three kinds of search trees are used: biased 2-3 trees (biased), self-adjusting search trees (splay) and randomized search trees (treap). We tested all algorithms for minimum spanning forest problem (MSF) along with Kruskal's $O(m \log m)$ algorithm[15] (kruskal) as the standard program to ensure the correctness of our algorithms. Here n is the number of vertices in graph and m denotes the number of edges. All algorithms were implemented in C++ by the authors and are available at http://github.com/riteme/toys/tree/master/dynamic-trees. Source code was compiled by Clang 6.0.0 with -O3 -Ofast -DNDEBUG options for full optimizations. Experiments were carried out on Intel™ i5 8300H running Ubuntu 18.04 (64bit operating system, Linux Kernel 4.20.13, Xorg not started) at 4.0GHz in a single core, with 128KB L1 data cache, 1MB L2 cache, 8MB L3 cache and 16GB RAM. CPU time was measured with high_resolution_clock from Standard Template Library (STL) in C++ with 1ns precision. Each individual execution was repeated until the number of processed edges exceeded 5×10^6 and we took the average on all edges. An untimed execution prior to timed runs for each algorithm was aimed to warm up the cache and prevent inconsistent result in the first run. Test data were generated by pseudo-random generator (rand in GNU C library) and we randomly chose seeds obtained from random_device in STL to generate five different inputs for each set of parameters and report the average results. Time for initialization, finalization and reading files was not accounted.

In LCT-based solutions, m edges are added into an initially empty graph one by one and we maintain the MSF as dynamic trees. When processing an edge $(u,\ v,\ w)$ (w is the length of the edge), if u and v are not in the same components, this edge will be immediately added. Otherwise there is a chain between u and v in dynamic trees. In this case, we pick up the longest edge $(u,\ v,\ w_0)$ in the chain from u to v. If $w_0 \geqslant w$, the longest edge will be discarded by CUT and the new edge will be incorporated into dynamic trees via LINK. The new edge will be ignored if $w_0 \leqslant w$ and we continue to process the next edge. This procedure can be implemented in $O(\log n)$ amortized time by LCT.

Three solutions based on LCT used the same interface exported to the main program to guarantee uniformity. Treaps utilized Xorshift algorithm[16] for extremely fast pseudo-random generations. Biased trees required extra memory for internal nodes and we implemented a stack-based memory pool for biased trees. All solutions were implemented in non-recursive fashion.

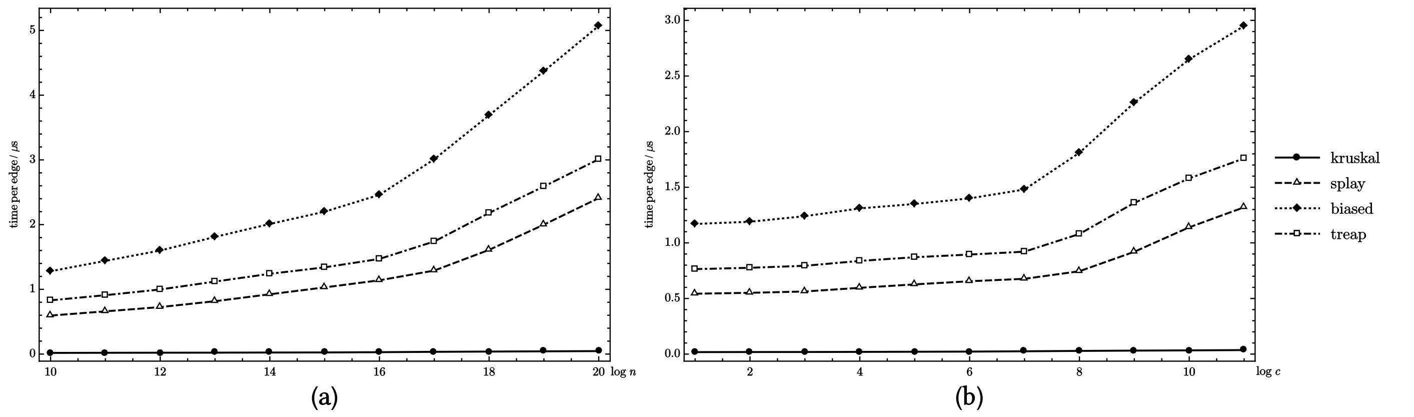

Fig.4. Experimental results. (a) Figure for randomized test data. (b) Figure for cache tests. Horizontal axes are scaled by binary logarithm and time is measured in microseconds averaged on all edges.

4.2 Random Data. We first tested on randomly generated graphs. Random graphs were obtained by generating m edges $(u_i,\ v_i,\ w_i)$ where both u_i and v_i are distinct and randomly chosen from $[1..n]$ and w_i is the length of edge i and within $[1..10^9]$. In this experiment, n varied from 2^{10} to 2^{20} and m was set to $8n$. Results are shown in Fig.4. (a). It suggests that biased trees are the slowest among all solutions, taking approximately twice the time of splay trees, while treaps show competitively performance compared to splay trees. Kruskal's algorithm is the fastest owing to its simplicity and the efficient implementation of quick sort algorithm in STL. Nearly identical figures between treaps and splay trees are consonant to Theorem 2. Furthermore, our results are also consistent with Pfaff's conclusion[17] that splay trees outperform conventional binary search trees such as AVL trees and red-black trees in system softwares. We assume that splay trees are advantageous in the setting of amortized LCT since the data structure is ephemeral and changes even during accesses. The results offer a different aspect of prominent performance of splay-based LCT as suggested by Tarjan et al.[25] in experiments among various types of dynamic trees.

4.3 Cache Tests. Since the time complexity is logarithmic in theory, figures are expected to be linear. However, we noticed that figures approximately comprise two linear functions. Such peculiar phenomena was observed by Tarjan et al.[25] as well and they deduced that cache-missing contributes to the increased slopes in right parts of figures. We attempted to examine the impacts of cache by a new data generator. We fixed $B = 2^9 = 512$ as the block size and $n = cB$ and $m = 8n$, where c is the number of blocks. The only difference from the previous random generator is that we first choose a block for each edge rather than putting it arbitrarily. The inputs can be viewed as a sequence of small graphs with size B, and now algorithms cannot concentrate on a single block but switching frequently between blocks. Accesses to RAM are time-consuming and much slower than operating on cache. When c increases, data may not completely fit in the cache and thus force CPU to access RAM, i.e. cache-missing. Fig.4. (b) reports that when $c \leqslant 2^7$ $(n \leqslant 2^{16})$, the relative increments of time are less than 27% for three trees, in contrast to nearly 90% in Fig.4. (a) when n varies from 2^{10} to 2^{16}. Significant increases after $c = 2^7$ $(c = 2^8$ for splay trees) demonstrate that the dominant performance-dropping factor seems to be the limited cache size. For example, the size of node in LCT with treaps is 36 bytes in our implementation and therefore the total memory allocated is $2 \times 2^8 \times 512 \times 36 = 9,437,184$ bytes when $c = 2^8$, which cannot fit in an 8MB L3 cache anymore. Note that edges are replaced with new nodes in dynamic trees so the actual number of nodes is $2n$. For splay trees, this amount decreases to only $7,340,032$ bytes so that splay-based LCT can be preloaded into cache. Biased trees are slowed down by the huge extra memory consumption from internal nodes for similar reasons. We conclude that the advantage of no extra information in splay tree nodes makes splay tree more cache-friendly and contributes to its amazing performance. With smaller cache size, Tarjan et al.[25] provided a more detailed figure when c gets larger and we speculate that the growth rate will eventually converges to a constant due to the tendency to direct manipulations of data in RAM. Meanwhile, our platform has an extra 1MB L2 cache compared to Tarjan's experiments[25] which may accelerate algorithms when $n \leqslant 2^{12}$ and enlarge the discrepancies for $n \leqslant 2^{16}$.

5 Conclusions and Discussions

In this paper, we reviewed and briefly analyzed three schemes to implement link-cut trees in amortized logarithmic time. They share an identical interface to users and only differ in the underlying data structures. It is mainly an interest of theoretical research to devise alternatives in amortized LCT because empirical study has demonstrated that splay-based version is the most efficient and other ordinary search trees take no advantage in an ever-changing data structure and splay trees are more memory-friendly and cache-friendly in the comparisons. Werneck[26] mentioned that Alstrup et al. attempted to implement LCT with weighted treaps and compared its performance to splay-based LCT, but their experimental part is unpublished. However, if we need to keep historical versions of dynamic trees, LCT must guarantee worst-case time complexities instead of amortized ones. Sleator et al.[21] presented a modified version of biased trees, i.e. globally-biased 2-b trees[9], that ensures worst-case performance of LCT. We leave it as an open problem that whether there is an alternative to globally-biased trees in LCT with worst-case time bounds.

6 References

[1] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo, "Dynamizing Static Algorithms, with Applications to Dynamic Trees and History Independence," in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, PA, USA, 2004, pp. 531–540.

[2] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup, "Minimizing Diameters of Dynamic Trees," in *Automata, Languages and Programming*, 1997, pp. 270–280.

[3] S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup, "Maintaining Information in Fully Dynamic Trees with Top Trees," *ACM Trans. Algorithms*, vol. 1, no. 2, pp. 243–264, Oct. 2005.

[4] S. Bent, D. Sleator, and R. Tarjan, "Biased Search Trees," *SIAM J. Comput.*, vol. 14, no. 3, pp. 545–568, Aug. 1985.

[5] J. Bitner, "Heuristics That Dynamically Organize Data Structures," *SIAM J. Comput.*, vol. 8, no. 1, pp. 82–110, Feb. 1979.

[6] G. E. Blelloch, D. Ferizovic, and Y. Sun, "Just Join for Parallel Ordered Sets," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2016, pp. 253–264.

[7] G. E. Blelloch and M. Reid-Miller, "Fast Set Operations Using Treaps," in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, NY, USA, 1998, pp. 16–26.

[8] G. Cattaneo, P. Faruolo, U. F. Petrillo, and G. F. Italiano, "Maintaining Dynamic Minimum Spanning Trees: An Experimental Study," in *Algorithm Engineering and Experiments*, 2002, pp. 111–125.

[9] J. Feigenbaum and R. E. Tarjan, "Two New Kinds of Biased Search Trees," *The Bell System Technical Journal*, vol. 62, no. 10, pp. 3139–3158, Dec. 1983.

[10] G. Frederickson, "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications," *SIAM J. Comput.*, vol. 14, no. 4, pp. 781–798, Nov. 1985.

[11] G. Frederickson, "Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and k Smallest Spanning Trees," *SIAM J. Comput.*, vol. 26, no. 2, pp. 484–538, Mar. 1997.

[12] G. N. Frederickson, "A Data Structure for Dynamically Maintaining Rooted Trees," *Journal of Algorithms*, vol. 24, no. 1, pp. 37–65, Jul. 1997.

[13] A. V. Goldberg, M. D. Grigoriadis, and R. E. Tarjan, “Use of Dynamic Trees in a Network Simplex Algorithm for the Maximum Flow Problem,” *Mathematical Programming*, vol. 50, no. 1, pp. 277–290, Mar. 1991.

[14] M. R. Henzinger, V. King, and V. King, “Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time Per Operation,” *J. ACM*, vol. 46, no. 4, pp. 502–516, Jul. 1999.

[15] J. B. Kruskal, “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem,” *Proc. Amer. Math. Soc.*, vol. 7, no. 1, pp. 48–50, 1956.

[16] G. Marsaglia, “Xorshift RNGs,” *Journal of Statistical Software*, vol. 8, no. 1, pp. 1–6, Jul. 2003.

[17] B. Pfaff, “Performance Analysis of BSTs in System Software,” in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2004, pp. 410–411.

[18] T. Radzik, “Implementation of Dynamic Trees with In-subtree Operations,” *J. Exp. Algorithmics*, vol. 3, Sep. 1998.

[19] G. Ramalingam and T. Reps, “On the Computational Complexity of Dynamic Graph Problems,” *Theoretical Computer Science*, vol. 158, no. 1, pp. 233–277, May 1996.

[20] R. Seidel and C. R. Aragon, “Randomized Search Trees,” *Algorithmica*, vol. 16, no. 4, pp. 464–497, Oct. 1996.

[21] D. D. Sleator and R. E. Tarjan, “A Data Structure for Dynamic Trees,” in *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 1981, pp. 114–122.

[22] D. D. Sleator and R. E. Tarjan, “Self-adjusting Binary Search Trees,” *J. ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985.

[23] R. E. Tarjan, “Dynamic Trees as Search Trees via Euler Tours, Applied to the Network Simplex Algorithm,” *Mathematical Programming*, vol. 78, no. 2, pp. 169–177, Aug. 1997.

[24] R. E. Tarjan and R. F. Werneck, “Self-Adjusting Top Trees,” in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, PA, USA, 2005, pp. 813–822.

[25] R. E. Tarjan and R. F. Werneck, “Dynamic Trees in Practice,” *J. Exp. Algorithmics*, vol. 14, pp. 5:4.5–5:4.23, Jan. 2010.

[26] R. Werneck, “Design and Analysis of Data Structures for Dynamic Trees,” Princeton University, 2006.

7 Appendixes

A. Graph Terminology. In graph theory, a *graph* $G = (V,\ E)$ consists of a set of vertices or nodes (represented by V) and a set of edges connecting them (represented by E). We may assume $n = |V|$ in analysis. A graph is *connected* iff. any two nodes are connected by a *path*. A *tree* is a connected graph with exactly $n - 1$ edges and a *forest* consists of some tree-like components. It is obvious that the number of edges in a forest is strictly less than n. One important characteristic of trees is that there is only one simple path connecting any two nodes, and we often refer to those unique paths as *chains*. In some applications, we choose a node as *tree root* for each tree, i.e. *rooted trees*, in contrast to *free trees*. In rooted trees, *access path* of node x is the chain from x to tree root, and all the nodes in this path are called *ancestors* of x. Note that x is also an ancestor of itself. Meanwhile, y is a *descendant* of x iff. x is an ancestor of y. All descendants of x as well as the edges between them form a *subtree* rooted at x (including x itself). For convenience, we often use the subtree root to refer to the entire subtree. Specially, for edge $(x,\ y)$, if x is an ancestor of y, then x is also called the *parent* of y or $x = p(y)$ and y is a child of x. For tree root r, $p(r)$ is defined to be $\varnothing$. The *depth* of x is the number of nodes in the access path of x, and $s(x)$ stands for the number of vertices in the subtree x.

B. Proof for Join Lemma. Consider the *right spine* of treap x (*left spine* of treap y) during JOIN. JOIN will end up node k reaching its appropriate position with two subtrees x_0 and y_0 (possibly empty). Let $r'(k)$ be the rank of k after JOIN. The time to travel on both spines is $\mathrm{O}(\log s(z)/w(k)) = \mathrm{O}(r(z) - r(k)) = \mathrm{O}(c - r(k))$ by ideal access. To maintain credit invariants, we observe that the number of required credits summing over z to k is $\Theta(c - r'(k))$ by telescoping, which is not greater than the newly allocated credits since $r'(k) \geqslant r(k)$, and credits on x_0 can be compensated by the $\Theta(c - r(x_0))$ credits aggregated over x to x_0 in original treap x since $c \geqslant r(z) \geqslant r'(k)$. Arguments for y_0 are similar. ■

C. Proof for Split Lemma. SPLIT is the inverse of JOIN (figure omitted). Let $r'(y) = \log w(y)$. The time to access y is $\mathrm{O}(\log s(x)/w(y)) = \mathrm{O}(r(x) - r'(y)) = \mathrm{O}(c - r'(y))$. Suppose y has two children u_0 and v_0 (possibly empty). $\Theta(c - r(u_0))$ credits aggregated over x to u_0 are distributed on all nodes from u to u_0 to guarantee that treap u is cast to c. $\Theta(c - r(y)) = \mathrm{O}(c - r'(y))$ new credits are supplied to treap v. Along with the original $\Theta(r(y) - r(v_0))$ credits on v_0, the final $\Theta(c - r(v_0))$ credits are sufficient to ensure treap v is also cast to c. ■

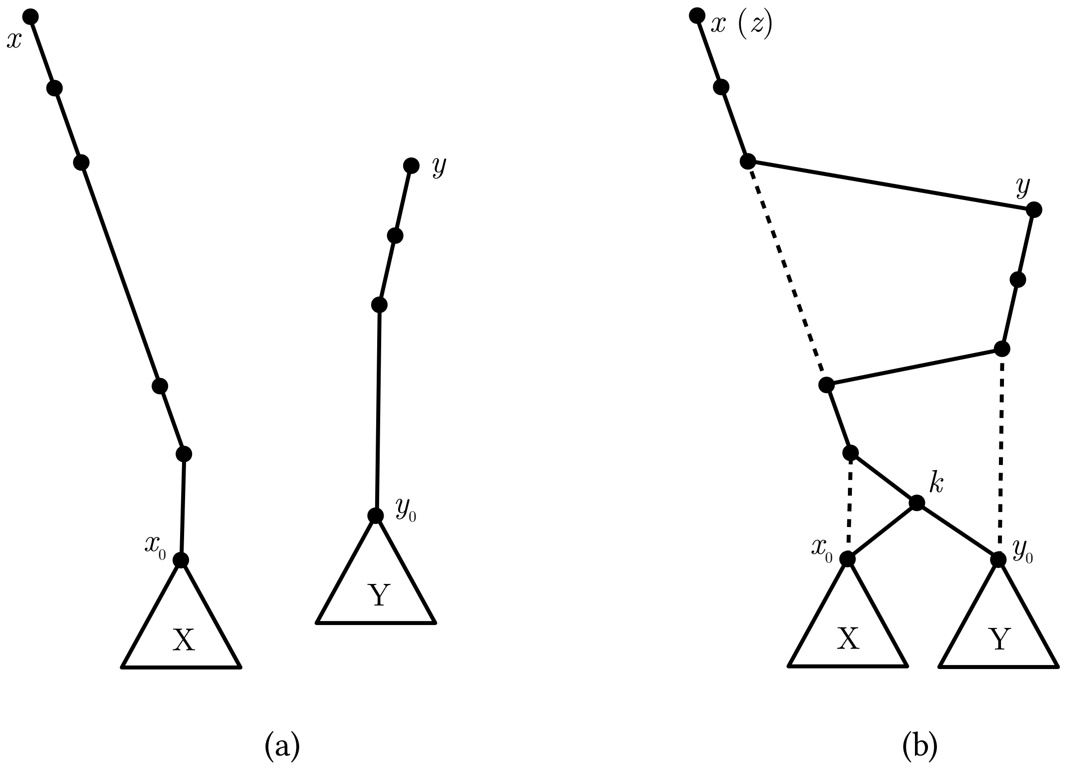

Fig.5. An example of JOIN. (a) Left spine and right spine. (b) Treap z after JOIN. Dashed edges are removed during JOIN.