Experimental Comparison of PC-Trees and PQ-Trees

Simon D. Fink
Faculty of Informatics and Mathematics, Universität Passau, Germany

Matthias Pfretzschner
Faculty of Informatics and Mathematics, Universität Passau, Germany

Ignaz Rutter
Faculty of Informatics and Mathematics, Universität Passau, Germany

Abstract

PQ-trees and PC-trees are data structures that represent sets of linear and circular orders, respectively, subject to constraints that specific subsets of elements have to be consecutive. While equivalent to each other, PC-trees are conceptually much simpler than PQ-trees; updating a PC-tree so that a set of elements becomes consecutive requires only a single operation, whereas PQ-trees use an update procedure that is described in terms of nine transformation templates that have to be recursively matched and applied.

Despite these theoretical advantages, to date no practical PC-tree implementation is available. This might be due to the original description by Hsu and McConnell [14] in some places only sketching the details of the implementation. In this paper, we describe two alternative implementations of PC-trees. For the first one, we follow the approach by Hsu and McConnell, filling in the necessary details and also proposing improvements on the original algorithm. For the second one, we use a different technique for efficiently representing the tree using a Union-Find data structure. In an extensive experimental evaluation we compare our implementations to a variety of other implementations of PQ-trees that are available on the web as part of academic and other software libraries. Our results show that both PC-tree implementations beat their closest fully correct competitor, the PQ-tree implementation from the OGDF library [6, 15], by a factor of 2 to 4, showing that PC-trees are not only conceptually simpler but also fast in practice. Moreover, we find the Union-Find-based implementation, while having a slightly worse asymptotic runtime, to be twice as fast as the one based on the description by Hsu and McConnell.

2012 ACM Subject Classification Mathematics of computing → Permutations and combinations; Mathematics of computing → Trees; Mathematics of computing → Graph algorithms

Keywords and phrases PQ-Tree, PC-Tree, circular consecutive ones, implementation, experimental evaluation

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.43

Related Version *Full Version*: https://arxiv.org/abs/2106.14805

Supplementary Material *Software (Source Code)*: https://github.com/N-Coder/pc-tree/
archived at swh:1:dir:00616576c1a16938cc5c376685086c7c2c368f7d

Funding Work partially supported by DFG-grant Ru-1903/3-1.
Simon D. Fink: DFG-grant Ru-1903/3-1.
Matthias Pfretzschner: DFG-grant Ru-1903/3-1.
Ignaz Rutter: DFG-grant Ru-1903/3-1.

1 Introduction

PQ-trees represent linear orders of a ground set subject to constraints that require specific subsets of elements to be consecutive. Similarly, PC-trees do the same for circular orders subject to consecutivity constraints. PQ-trees were developed by Booth and Lueker [3] to

29th Annual European Symposium on Algorithms (ESA 2021).
Editors: Petra Mutzel, Rasmus Pagh, and Grzegorz Herman; Article No. 43; pp. 43:1–43:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

solve the consecutive ones problem, which asks whether the columns of a Boolean matrix can be permuted such that the 1s in each row are consecutive. PC-trees are a more recent generalization introduced by Shih and Hsu [16] to solve the circular consecutive ones problem, where the 1s in each row only have to be circularly consecutive.

Though PQ-trees represent linear orders and PC-trees represent circular orders, Haeupler and Tarjan [10] show that in fact PC-trees and PQ-trees are equivalent, i.e., one can use one of them to implement the other without affecting the asymptotic running time. The main difference between PQ-trees and PC-trees lies in the update procedure. The update procedure takes as input a PQ-tree (a PC-tree) T and a subset U of its leaves and produces a new PQ-tree (PC-tree) T' that represents exactly the linear orders (circular orders) represented by T where the leaves in U appear consecutively. The update procedure for PC-trees consists only of a single operation that is applied independently of the structure of the tree. In contrast, the update of the PQ-tree is described in terms of a set of nine template transformations that have to be recursively matched and applied.

PQ-trees have numerous applications, e.g., in planarity testing [3, 16], recognition of interval graphs [3] and genome sequencing [1]. Nevertheless, PC-trees have been adopted more widely, e.g., for constrained planarity testing problems [2, 5] due to their simpler update procedure. Despite their wide applications and frequent use in theoretical algorithms, few PQ-tree implementations and even fewer PC-tree implementations are available. Table 1 shows an overview of all PC/PQ-tree implementations that we are aware of, though not all of them are working.

In this paper we describe the first correct and generic implementations of PC-trees. Section 2 contains an overview of the update procedure for applying a new restriction to a PC-tree. In Section 3, we describe the main challenge when implementing PC-trees and how our two implementations take different approaches at solving it. In Section 4, we present an extensive experimental evaluation, where we compare the performance of our implementations with the implementations of PC-trees and PQ-trees from Table 1. Our experiments show that, concerning running time, PC-trees following Hsu and McConnell's original approach beat their closest competitor, the PQ-tree implementation from the OGDF library [6] by roughly a factor 2. Our second implementation using Union-Find is another 50% faster than this first one, thus beating the OGDF implementation by a factor of up to 4.

2 The PC-tree

A *PC-tree* T is a tree without degree-2 vertices whose inner nodes are partitioned into *P-nodes* and *C-nodes*. Edges incident to C-nodes have a circular order that is fixed up to reversal, whereas edges incident to P-nodes can be reordered arbitrarily. Traversing the tree according to fixed orders around the inner nodes determines a circular ordering of the leaves L of the tree. Any circular permutation of L that can be obtained from T after arbitrarily reordering the edges around P-nodes and reversing orders around C-nodes is a *valid permutation* of L. In this way a PC-tree represents a set of circular permutations of L.

When applying a *restriction* $R \subseteq L$ to T, we seek a new tree that represents exactly the valid permutations of L where the leaves in R appear consecutively. We call a restriction *impossible* if there is no valid permutation of L where the leaves in R are consecutive. Thus, restriction R is possible if and only if the edges incident to P-nodes can be rearranged and orders of edges incident to C-nodes can be reversed in such a way that all leaves in R are consecutive. Updating a PC-tree to enforce the new restriction can thus be done by identifying and adapting the nodes that decide about the consecutivity of the elements of R and then changing the tree to ensure that this consecutivity can no longer be broken.

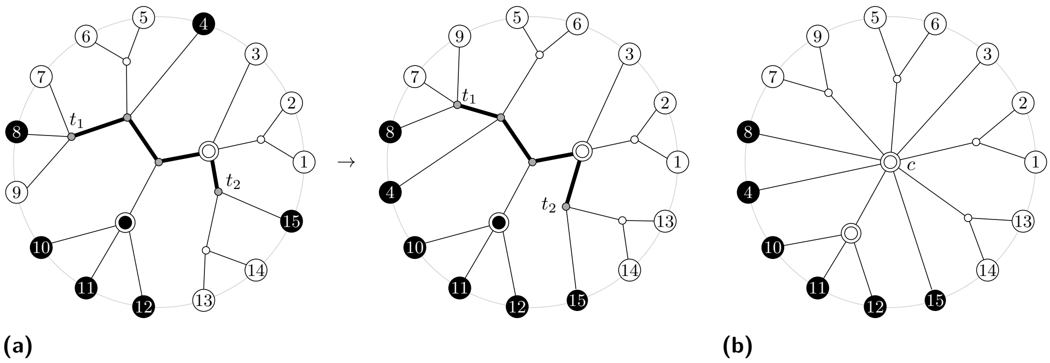


Figure 1 (a) Two equivalent PC-Trees with their nodes colored according to the restriction $\{4, 8, 10, 11, 12, 15\}$. C-nodes are represented by big double circles and the P-nodes are represented by small circles. The white nodes represent empty nodes, the black nodes represent full nodes and the gray nodes represent partial nodes. The thick edges represent the terminal path with terminal nodes t_1 and t_2. As the restriction is possible, all full leaves of the tree on the left can be made consecutive, as shown on the right. Furthermore all nodes that must be modified lie on a path.
(b) Updated PC-tree with new central C-node c.

Let a leaf $x \in L$ be *full* if $x \in R$ and *empty* otherwise. We call an edge *terminal* if the two subtrees separated by the edge both contain at least one empty and at least one full leaf. Exactly the endpoints of all terminal edges need to be "synchronized" to ensure that all full leaves are consecutive. Hsu and McConnell [14, 13] show that R is possible if and only if the terminal edges form a path and all nodes of this path can be flipped so that all full leaves are on one side and all empty leaves are on the other. This path is called the *terminal path*, the two nodes at the ends of the terminal path are the *terminal nodes*. Observe that each node in T that is adjacent to two subtrees of which one only contains full leaves and the other contains only empty leaves is contained in the terminal path. Figure 1a illustrates the terminal path.

When updating T in order to apply the restriction, every node on the terminal path is split into two nodes, one of which holds all edges to neighbors of the original node whose subtree has only full leaves, the other holds all edges to empty neighbors, while terminal edges are deleted. A new central C-node c is created that is adjacent to all the split nodes in such a way that it preserves the order of the neighbors around the terminal path. Contracting all edges to the split C-nodes incident to c and contracting all nodes with degree two results in the updated tree that represents the new restriction [14, 13]. Figure 1 shows an example of this update, while Figure 2 details changes made to the terminal path.

It remains to efficiently find the terminal edges, and thus the subtrees with mixed full and empty leaves. To do so, Hsu and McConnell first choose an arbitrary node of the tree as root. They also assign labels to the inner nodes of the tree, marking an inner node (and conceptually the subtree below it) *partial* if at least one of its neighbors (i.e. children or parent) is full, *full* if all its neighbors except one (which usually is the parent) are full, and *empty* otherwise. Then, an edge is terminal if and only if it lies on a path between two partial nodes [14, 13]. Assigning the labels and subsequently finding the terminal edges can be done by two bottom-up traversals of the tree. We summarize these steps in the following, more fine-granular description of Hsu and McConnell's algorithm for updating the PC-tree [13, Algorithm 32.2]:

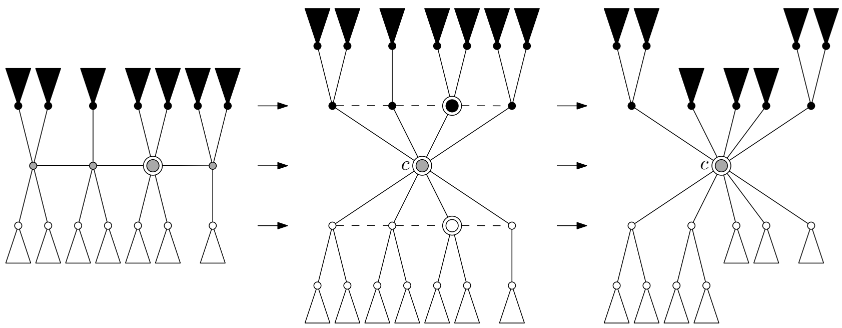


Figure 2 Left: The terminal path with all full subtrees shown in black on top and all empty subtrees shown in white on the bottom. Middle: The updated PC-tree, where all terminal edges were deleted, all nodes on the terminal were split in a full and empty half and all new nodes were connected to a new C-node c. Right: The PC-Tree after contracting all new C-nodes and all degree-2 P-nodes into c.

Algorithm for Applying Restrictions. To add a new restriction R to a PC-tree T:

1. Label all partial and full nodes by searching the tree bottom-up from all full leaves.
2. Find the terminal path by walking the tree upwards from all partial nodes in parallel.
3. Perform flips of C-nodes and modify the cyclic order of edges incident to P-nodes so that all full leaves lie on one side of the path.
4. Split each node on the path into two nodes, one incident to all edges to full leaves and one incident to all edges to empty leaves.
5. Delete the edges of the path and replace them with a new C-node c, adjacent to all split nodes, whose cyclic order preserves the order of the nodes on this path.
6. Contract all edges from c to adjacent C-nodes, and contract any node that has only two neighbors.

3 Our Implementations

The main challenge posed to the data structure for representing the PC-tree is that, in step 6, it needs to be able to merge arbitrarily large C-nodes in constant time for the overall algorithm to run in linear time. This means that, whenever C-nodes are merged, updating the pointer to a persistent C-node object on every incident edge would be too expensive. Hsu and McConnell (see [13, Definition 32.1]) solve this problem by using C-nodes that, instead of having a permanent node object, are only represented by the doubly-linked list of their incident half-edges, which we call *arcs*. This complicates various details of the implementation, like finding the parent pointer of a C-node, which are only superficially covered in the initial work of Hsu and McConnell [14]. These issues are in part remedied by the so called *block-spanning pointers* introduced in the later published book chapter [13], which are related to the pointer borrowing strategy introduced by Booth and Lueker [3]. These block-spanning pointers link the first and last arc of a consecutive block of full arcs (i.e. the arcs to full neighbors) around a C-node and can be accompanied by temporary C-node objects. Whenever a neighbor of a C-node becomes full, either a new block is created for the corresponding arc of the C-node, an adjacent block grows by one arc, or the two blocks that now became adjacent are merged.

Using this data structure, Hsu and McConnell show that the addition of a single new restriction R takes $O(p + |R|)$ time, where p is the length of the terminal path, and that applying restrictions $R_1, \dots, R_k$ takes $\Theta(|L| + \sum_{i=1}^{k} |R_i|)$ time [14, 13]. Especially for steps 1 and 2, they only sketch the details of the implementation, making it hard to directly put it into practice. In the full version, we fill in the necessary details for these steps and also refine their runtime analysis, showing that step 1 can be done in $O(|R|)$ time and step 2 can be done in $O(p)$ time. Using the original procedures by Hsu and McConnell, steps 3 and 4 can be done in $O(|R|)$ time and steps 5 and 6 can be done in $O(p)$ time.

For our first implementation, which we call HsuPC, we directly implemented these steps in C++, using the data structure without permanent C-node objects as described by Hsu and McConnell. During the evaluation, we realized that traversals of the tree are expensive. This is plausible, as they involve a lot of pointer-dereferencing to memory segments that are not necessarily close-by, leading to cache misses. To avoid additional traversals for clean-up purposes, we store information that is valid only during the update procedure with a timestamp. Furthermore, we found that keeping separate objects for arcs and nodes and the steps needed to work around the missing C-node objects pose a non-negligible overhead.

To remove this overhead, we created a second version of our implementation, which we call UFPC, using a Union-Find tree for representing C-node objects: Every C-node is represented by an entry in the Union-Find tree and every incident child edge stores a reference to this entry. Whenever two C-nodes are merged, we apply `union` to both entries and only keep the object of the entry that survives. This leads to every lookup of a parent C-node object taking amortized $O(\alpha(|L|))$ time, where α is the inverse Ackermann function. Although this makes the overall runtime super-linear, the experimental evaluation following in the next section shows that this actually improves the performance in practice. As a second change, the UFPC no longer requires separate arc and node objects, allowing us to use a doubly-linked tree consisting entirely of nodes that store pointers to their parent node, left and right sibling node, and first and last child node. Edges are represented implicitly by the child node whose parent is the other end of the edge. Note that of the five stored pointers, a lookup in the Union-Find data structure is only needed for resolving the parent of a node.

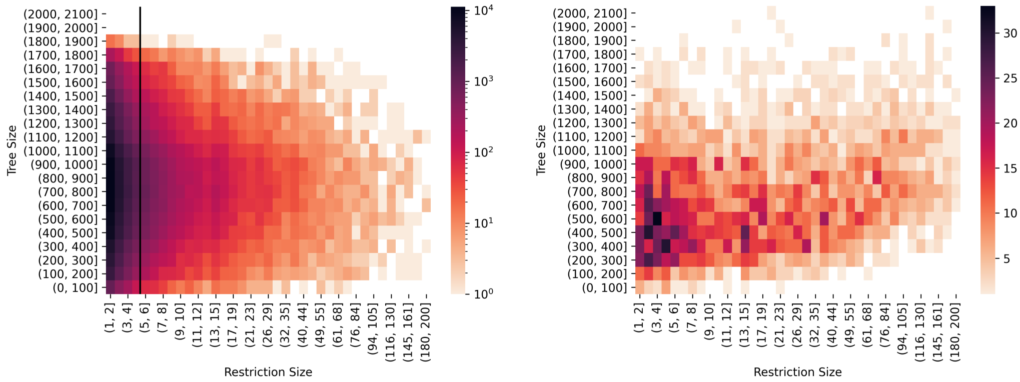
Our algorithmic improvements and differences of both implementations are described in more detail in the full version. We use the Union-Find data structure from the OGDF [6] and plan to merge our UFPC implementation into the OGDF. Furthermore, both implementations should also be usable stand-alone with a custom Union-Find implementation. The source code for both implementations, our evaluation harness and all test data are available on GitHub (see Table 1).

4 Evaluation

In this section, we experimentally evaluate our PC-tree implementations by comparing the running time for applying a restriction with that of various PQ- and PC-tree implementations that are publicly available. In the following we describe our method for generating test cases, our experimental setup and report our results.

4.1 Test Data Generation

To generate PQ-trees and restrictions on them, we make use of the planarity test by Booth and Lueker [3], one of the initial applications of PQ-trees. This test incrementally processes vertices one by one according to an *st*-ordering. Running the planarity test on a graph with n vertices applies $n-1$ restrictions to PQ-trees of various sizes. Since not all implementations provide the additional modification operations necessary to implement the planarity test, we rather export, for each step of the planarity test, the current PQ-tree and the restriction that is applied to it as one instance of our test set. We note that the use of *st*-orderings ensures that the instances do not require the ability of the PC-tree to represent circular permutations, making them good test cases for comparing PC-trees and PQ-trees.



(a) **(b)**

Figure 3 Distribution of tree and restriction size for the data sets **(a)** `SER-POS` and **(b)** `SER-IMP`. Please note the different color scales. The `SER-POS` instances that are left of the black line are too small and filtered out.

In this way, we create one test set `SER-POS` consisting of only PQ-trees with possible restrictions by exporting the instances from running the planarity test on a randomly generated biconnected planar graph for each vertex count n from 1000 to $20,000$ in steps of 1000 and each edge count $m \in \{2n, 3n-6\}$. To avoid clutter, to remedy the tendency of the planarity test to produce restrictions with very few leaves, and to avoid bias from trivial optimizations such as filtering trivial restrictions with $|R| \in \{1, |L|-1, |L|\}$, which is present in some of the implementations, we filter test instances where $|R|$ lies outside the interval $[5, |L|-2]$. Altogether, this test set contains $199,831$ instances, whose distribution with regards to tree and restriction size is shown in Figure 3a.

To guard against overly permissive implementations, we also create a small test set `SER-IMP` of impossible restrictions. It is generated in the same way, by adding randomly chosen edges to the graphs from above until they become non-planar. In this case the planarity test fails with an impossible restriction at some point; we include these $3,800$ impossible restrictions in the set, see Figure 3b.

As most of the available implementations have no simple means to store and load a PQ-/PC-tree, we serialize each test instance as a set of restrictions that create the tree, together with the additional new restriction. When running a test case, we then first apply all the restrictions to reobtain the tree, and then measure the time to apply the new restriction from the test case. The prefix `SER-` in the name of both sets emphasizes this serialization.

To be able to conduct a more detailed comparison of the most promising implementations, we also generate a third test set with much larger instances. As deserializing a PC- or PQ-tree is very time-consuming, we directly use the respective implementations in the planarity test by Booth and Lueker [3], thus calling the set `DIR-PLAN`. We generated 10 random planar graphs with n vertices and m edges for each n ranging from $100,000$ to $1,000,000$ in steps of $100,000$ and each $m \in \{2n, 3n-6\}$, yielding 200 graphs in total. The planarity test then yields one possible restriction per node. As we only want to test big restrictions, we filter out restrictions with less than 25 full leaves, resulting in `DIR-PLAN` containing $564,300$ instances.

Table 1 Implementations considered for the evaluation. Implementations that are entirely unusable as they are incomplete or crash/produce incorrect results on almost all inputs (marked with –) and those where no stand-alone PC-/PQ-tree implementation could be extracted (marked with n.a.) could not be evaluated. Correct implementations are marked with ✓ and implementations that are functional, but do not always produce correct results are marked with ✗. These two categories are included in our experimental evaluation. The last column shows how many of 203,630 restrictions in the sets SER-POS and SER-IMP failed.

Name	Type	Context	Language	Correct	Errors	URL
HsuPC	PC-Tree	our impl., based on [13]	C++	✓	0	https://github.com/N-Coder/pc-tree/tree/HsuPCSubmodule
UFPC	PC-Tree	our impl. using Union-Find	C++	✓	0	https://github.com/N-Coder/pc-tree
Luk&Zhou	PC-Tree	student course project	C++	–	–	https://github.com/kwmichaelluk/pc-tree
Hsu [12]	PC-Tree	planarity test prototype	C++	n.a.	–	http://qa.iis.sinica.edu.tw/graphtheory
Noma [4]	PC-Tree	planarity test evaluation	C++	n.a.	–	https://www.ime.usp.br/~noma/sh
OGDF [15]	PQ-Tree	planarity testing	C++	✓	0	https://ogdf.github.io
Gregable	PQ-Tree	biclustering	C++	✓	0	https://gregable.com/2008/11/pq-tree-algorithm.html
BiVoC [9]	PQ-Tree	automatic layout of biclusters	C++	✗	71	https://bioinformatics.cs.vt.edu/~murali/papers/BiVoC
Reisle	PQ-Tree	student project	C++	✗	236	https://github.com/creisle/pq-trees
GraphSet [8]	PQ-Tree	visual graph editor	C++	✗	580	http://graphset.cs.arizona.edu
Zanetti [17]	PQR-Tree[1]	extension of PQ-Trees	Java	✗	454	https://github.com/jppzanetti/PQRTree
CppZanetti	PQR-Tree[1]	our C++ conversion of Zanetti	C++	✗	454	https://github.com/N-Coder/pc-tree#installation
JGraphEd [11]	PQ-Tree	visual graph editor	Java	✗	11	https://www3.cs.stonybrook.edu/~algorith/implement/jgraphed/implement.shtml
GTea [7]	PQ-Tree	visual graph theory tool	Java	–	–	https://github.com/rostam/GTea
TryAlgo	PQ-Tree	consecutive-ones testing	Python	–	–	https://tryalgo.org/en/datastructures/2017/12/15/pq-trees
SageMath	PQ-Tree	interval graph detection	Python	✓	0	https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/pq_trees.html

[1] PQR-Trees are a variant of PQ-Trees that can also represent impossible restrictions, replacing any node that would make a restriction impossible by an R-node (again allowing arbitrary permutation). To make the implementations comparable, we abort early whenever an impossible restriction is detected and an R-node would be generated.

4.2 Experimental Setup

Table 1 gives an overview of all implementations we are aware of, although not all implementations could be considered for the evaluation.

The three existing implementations of PC-trees we found are incomplete and unusable (Luk&Zhou) or tightly intertwined with a planarity test in such a way that we were not able to extract a generic implementation of PC-trees (Hsu, Noma). We further exclude two PQ-tree implementations as they either crash or produce incorrect results on almost all inputs (GTea) or have an excessively poor running time (TryAlgo). Among the remaining PQ-tree implementations only three correctly handle all our test cases (OGDF, Gregable, SageMath). Several other implementations have smaller correctness issues: After applying a fix to prevent segmentation faults in a large number of cases for BiVoC, the remaining implementations crash (BiVoC, GraphSet, Zanetti) and/or produce incorrect results (Reisle, JGraphEd, Zanetti) on a small fraction of our tests; compare the last column of Table 1. We nevertheless include them in our evaluation.

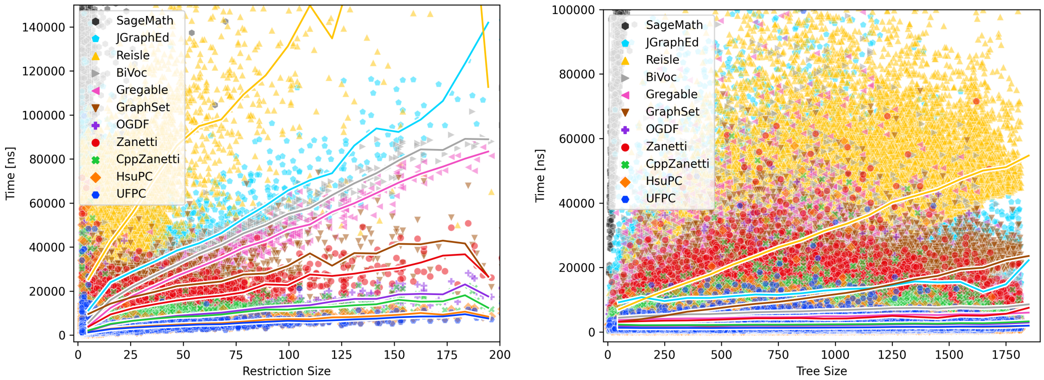
We changed the data structure responsible for mapping the input to the leaves of the tree for BiVoC and Gregable from `std::map` to `std::vector` to make them competitive. Moreover, BiVoC, Gregable and GraphSet use a rather expensive cleanup step that has to be executed after each update operation. As this could probably largely be avoided by the use of timestamps, we do not include the cleanup time in their reported running times. For SageMath the initial implementation turned out to be quadratic, which we improved to linear by removing unnecessary recursion. As Zanetti turned out to be a close competitor to our implementation in terms of running time, we converted the original Java implementation to C++ to allow a fair comparison. This decreased the runtime by one third while still producing the exact same results. All other non-C++ implementations were much slower or had other issues, making a direct comparison of their running times within the same language environment as our implementations unnecessary. Further details on the implementations can be found in the full version.

Each experiment was run on a single core of a Intel Xeon E5-2690v2 CPU (3.00 GHz, 10 Cores) with 64 GiB of RAM, running Linux Kernel version 4.19. Implementations in C++ were compiled with GCC 8.3.0 and optimization `-O3 -march=native -mtune=native`. Java implementations were executed on OpenJDK 64-Bit Server VM 11.0.9.1 and Python implementations were run with CPython 3.7.3. For the Java implementations we ran each experiment several times, only measuring the last one to remove startup-effects and to facilitate optimization by the JIT compiler. We used OGDF version 2020.02 (Catalpa) to generate the test graphs.

4.3 Results

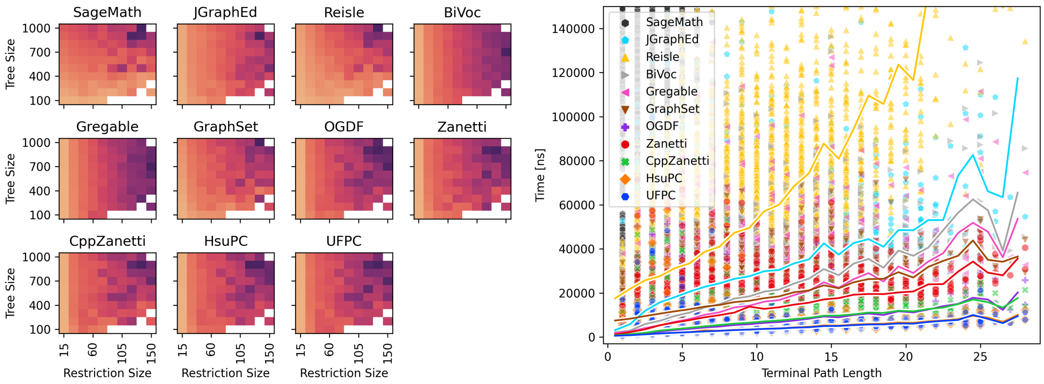
Our experiments turn out that SageMath, even with the improvements mentioned above, is on average 30 to 100 times slower than all other implementations.[2] For the sake of readability, we scale our plots to focus on the other implementations. As the main application of PC-/PQ-trees is applying possible restrictions, we first evaluate on the dataset `SER-POS`. Figure 4 shows the runtime for individual restrictions based on the size of the restriction (i.e. the number of full leaves) and the overall size of the tree. Figure 4a clearly shows that for all implementations the runtime is linear in the size of the restriction. Figure 4b suggests that

[2] Part of this might be due to the overhead of running the code with CPython. As the following analysis shows, SageMath also has other issues, allowing us to safely exclude it.



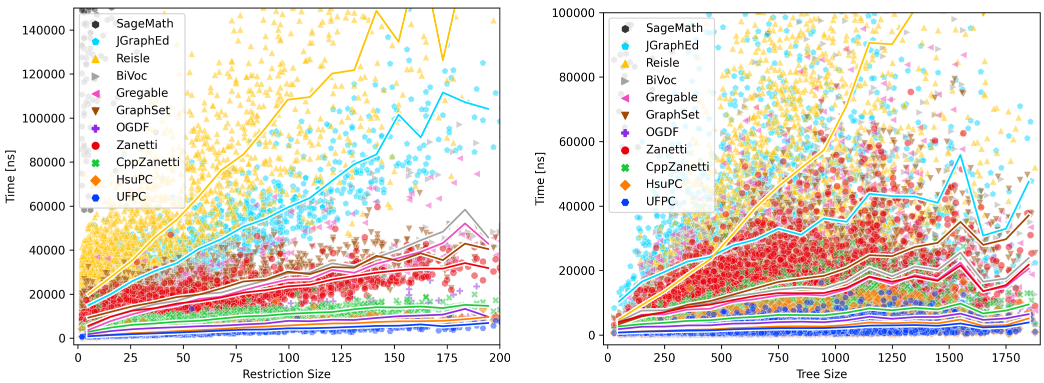
(a) (b)

Figure 4 Runtime for SER-POS restrictions depending on **(a)** restriction size and **(b)** tree size.



(a) (b)

Figure 5 **(a)** A heatmap showing the average runtime of SER-POS restrictions, depending on both the size of the restriction and the size of the tree. The color scale is based on the maximum runtime of each respective implementation. **(b)** Runtime for SER-POS restrictions depending on the terminal path length.



(a) (b)

Figure 6 Runtime for SER-IMP restrictions depending on **(a)** restriction size and **(b)** tree size for all implementations.

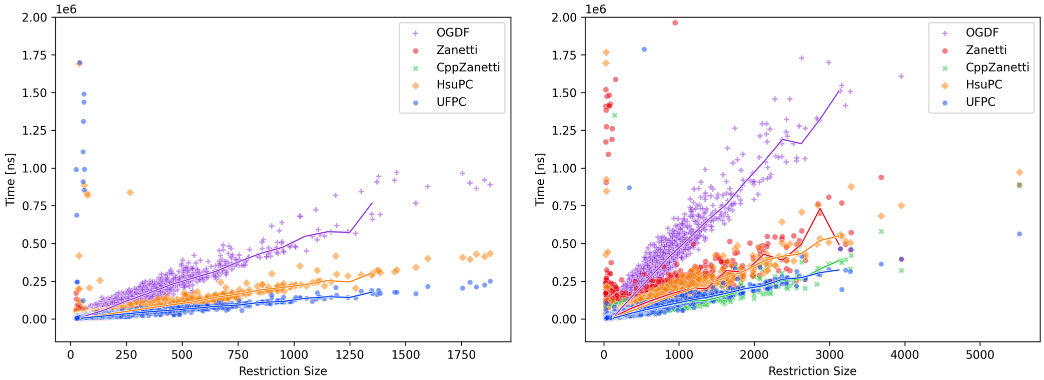
the runtime of Reisle and GraphSet does not solely depend on the restriction size, but also on the size of the tree. To verify this, we created for each implementation a heatmap that indicates the average runtime depending on both the tree size and the restriction size, shown in Figure 5a. The diagonal pattern shown by SageMath, Reisle, and GraphSet confirms the dependency on the tree size. All other implementations exhibit vertical stripes, which shows that their runtime does not depend on the tree size. Finally, Figure 5b shows the runtime compared to the terminal path length. As expected, all implementations show a linear dependency on the terminal path length, with comparable results to Figure 4a.

Figure 6 shows the performance on the dataset `SER-IMP`. The performance is comparable with that on `SER-POS`. Noteworthy is that Zanetti performs quite a bit worse, which is due to its implementation not being able to detect failure during a labeling step. It always performs updates until a so-called R-node would be generated. Altogether, the data from `SER-POS` and `SER-IMP` shows that the implementations GraphSet, OGDF, Zanetti, HsuPC and UFPC are clearly superior to the others. In the following, we conduct a more detailed comparison of these implementations by integrating them into a planarity test and running them on much larger instances, i.e., the data set `DIR-PLAN`. In addition to an update method, this requires a method for replacing the now-consecutive leaves by a P-node with a given number of child leaves. Adding the necessary functionality would be a major effort for most of the implementations, which is why we only adapted the most efficient implementations to run this set. We also exclude GraphSet from this experiment; the fact that it scales linearly with the tree size causes the planarity test to run in quadratic time. Figure 7 again shows the runtime of individual restrictions depending on the restriction size. Curiously, Zanetti produces incorrect results for nearly all graphs with $m = 2n$ in Figure 7a. As the initial tests already showed, the implementation has multiple flaws; one major issue is already described in an issue on GitHub and another independent error is described in the full version. Both plots show that HsuPC is more than twice as fast as OGDF and that UFPC is again close to two times faster than HsuPC. Zanetti's runtime is roughly the same as that of HsuPC, while converting its Java code to C++ brings the runtime down close to that of UFPC.

As OGDF is the slowest, we use it as baseline to calculate the speedup of the other implementations. Figure 8a shows that the runtime improvement for all three implementations is the smallest for small restrictions, quickly increasing to the final values of roughly 0.4 times the runtime of OGDF for HsuPC and 0.25 for both CppZanetti and UFPC. Figure 8b shows the speedup depending on the length of the terminal path. For very short terminal paths (which are common in our datasets), both implementations are again close; but already for slightly longer terminal paths UFPC quickly speeds up to being roughly 20% faster than CppZanetti. This might be because creating the central node in step 5 is more complicated for UFPC, as the data structure without edge objects does not allow arbitrarily adding and removing edges (which is easier for HsuPC) and allowing circular restrictions forces UFPC to also pay attention to various special cases (which are not necessary for PQ-trees).

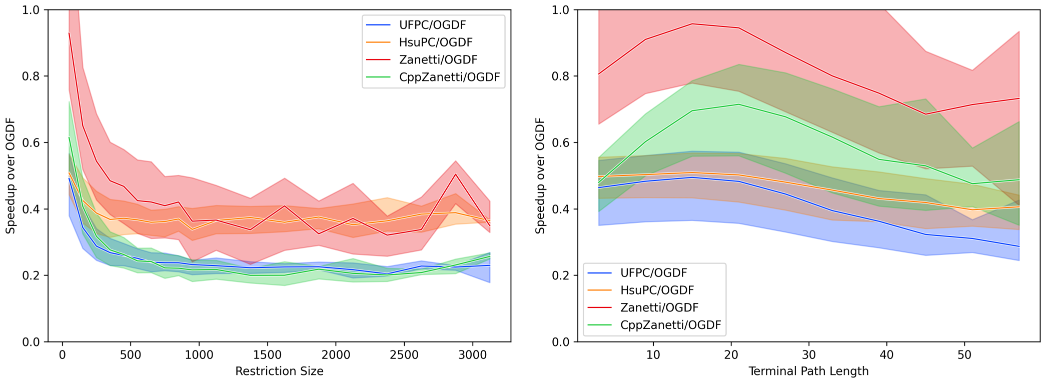
5 Conclusion

In this paper we have presented the first fully generic and correct implementations of PC-trees. One implementation follows the original description of Hsu and McConnell [14, 13], which contains several subtle mistakes in the description of the labeling and the computation of the terminal path. This may be the reason why no fully generic implementation has been available so far. A corrected version that also includes several small simplifications is described in the full version of this paper.



(a) **(b)**

Figure 7 Runtime of individual restrictions of `DIR-PLAN` with OGDF, Zanetti and our implementations for graphs of size **(a)** $m = 2n$ and **(b)** $m = 3n - 6$.



(a) **(b)**

Figure 8 Median performance increase depending on **(a)** the size of the restriction and **(b)** the terminal path length, with OGDF as baseline. The shaded areas show the interquartile range.

Furthermore, we provided a second, alternative implementation, using Union-Find to replace many of the complications of Hsu and McConnell's original approach. Technically, this increases the runtime to $O((|R| + p) \cdot \alpha(|L|))$, where α is the inverse Ackerman function. In contrast, our evaluations show that the Union-Find-based approach is even faster in practice, despite the worse asymptotic runtime.

Our experimental evaluation with a variety of other implementations reveals that surprisingly few of them seem to be fully correct. Only three other implementation have correctly handled all our test cases. The fastest of them is the PQ-tree implementation of OGDF, which our Union-Find-based PC-tree implementation beats by roughly a factor of 4. Interestingly, the Java implementation of PQR-trees by Zanetti achieves a similar speedup once ported to C++. However, Zanetti's Java implementation is far from correct and it is hard to say whether it is possible to fix it without compromising its performance.

Altogether, our results show that PC-trees are not only conceptually simpler than PQ-trees but also perform well in practice, especially when combined with Union-Find. To put the speedup of factor 4 into context, we compared the OGDF implementations of the planarity test by Booth and Lueker and the one by Boyer and Myrvold on our graph instances. The Boyer and Myrvold implementation was roughly 40% faster than the one based on Booth and Lueker's algorithm. Replacing the PQ-trees, which are the core part of the latter, by an implementation that is 4 time faster, might make this planarity test run faster than the one by Boyer and Myrvold. We leave a detailed evaluation, also taking into account the embedding generation, which our PC-tree based planarity test not yet provides, for future work.

References

1 S. Benzer. On the topology of the genetic fine structure. *Proceedings of the National Academy of Sciences*, 45(11):1607–1620, November 1959. doi:10.1073/pnas.45.11.1607.

2 Thomas Bläsius and Ignaz Rutter. Simultaneous PQ-ordering with applications to constrained embedding problems. *ACM Trans. Algorithms*, 12(2):16:1–16:46, 2016. doi:10.1145/2738054.

3 Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976. doi:10.1016/S0022-0000(76)80045-1.

4 John M. Boyer, Cristina G. Fernandes, Alexandre Noma, and José C. de Pina. Lempel, Even, and Cederbaum planarity method. In *Experimental and Efficient Algorithms*, pages 129–144. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-24838-5_10.

5 Guido Brückner and Ignaz Rutter. Partial and constrained level planarity. In Philip N. Klein, editor, *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'17)*, pages 2000–2011. SIAM, 2017. doi:10.1137/1.9781611974782.130.

6 Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 17. CRC Press, 2014.

7 Alex William Cregten and Hannes Kristján Hannesson. Implementation of a planarity testing method using PQ-trees. Technical report, Reykjavík University, 2017. URL: https://skemman.is/bitstream/1946/29618/1/Planarity_testing_with_PQTrees.pdf.

8 Alejandro Estrella-Balderrama, J. Joseph Fowler, and Stephen G. Kobourov. Graph simultaneous embedding tool, GraphSET. In *Graph Drawing*, pages 169–180. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-00219-9_17.

9 Gregory A. Grothaus, Adeel Mufti, and T. M. Murali. Automatic layout and visualization of biclusters. *Algorithms for Molecular Biology*, 1(1):15, 2006. doi:10.1186/1748-7188-1-15.

10 Bernhard Haeupler and Robert E. Tarjan. Planarity algorithms via PQ-trees (extended abstract). *Electronic Notes in Discrete Mathematics*, 31:143–149, August 2008. doi:10.1016/j.endm.2008.06.029.

11 Jon Harris. JGraphEd – a java graph editor and graph drawing framework. Technical report, Carleton University, School of Computer Science, Comp 5901 Directed Studies, 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.188.5066&rank=1.

12 Wen-Lian Hsu. An efficient implementation of the PC-tree algorithm of Shih & Hsu's planarity test. Technical report, Institute of Information Science, Academia Sinica, 2003. URL: http://iasl.iis.sinica.edu.tw/webpdf/paper-2003-PLANAR_implementation.pdf.

13 Wen-Lian Hsu and Ross McConnell. PQ trees, PC trees, and planar graphs. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 32. Chapman and Hall/CRC, January 2004. doi:10.1201/9781420035179.

14 Wen-Lian Hsu and Ross M. McConnell. PC trees and circular-ones arrangements. *Theoretical Computer Science*, 296(1):99–116, March 2003. doi:10.1016/s0304-3975(02)00435-8.

15 Sebastian Leipert. PQ-trees, an implementation as template class in C++. Technical report, University of Cologne, 1997. URL: http://e-archive.informatik.uni-koeln.de/id/eprint/259.

16 Wei-Kuan Shih and Wen-Lian Hsu. A new planarity test. *Theoretical Computer Science*, 223(1-2):179–191, July 1999. doi:10.1016/s0304-3975(98)00120-0.

17 João Paulo Pereira Zanetti. Complexidade de construção de árvores PQR. Master's thesis, Universidade Estadual de Campinas, Instituto de Computação, 2012. URL: http://bdtd.ibict.br/vufind/Record/CAMP_0b551865d78ef032289f17f95e3ccee7.