Experimental Comparison of PC-Trees and PQ-Trees

SIMON D. FINK, MATTHIAS PFRETZSCHNER, and IGNAZ RUTTER, University of Passau, Germany

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.

CCS Concepts: • Mathematics of computing → Permutations and combinations; Trees; Graph algorithms; • General and reference → Evaluation; Performance;

Additional Key Words and Phrases: PQ-tree, PC-tree, consecutive ones, experimental evaluation

ACM Reference format:
Simon D. Fink, Matthias Pfretzschner, and Ignaz Rutter. 2023. Experimental Comparison of PC-Trees and PQ-Trees. ACM J. Exp. Algor. 28, 1, Article 1.10 (October 2023), 24 pages.
https://doi.org/10.1145/3611653

A preliminary version of this paper has appeared as Simon D. Fink, Matthias Pfretzschner, Ignaz Rutter: Experimental Comparison of PC-Trees and PQ-Trees in Proceedings of the 29th Annual European Symposium on Algorithms (ESA 2021), 43:1-43:13. Our source code is available at github.com/N-Coder/pc-tree and also archived at softwareheritage.org as swh:1:snp:47991f983dd72e5d7148e7496f5e02493c3807.

Funded by the Deutsche Forschungsgemeinschaft (German Research Foundation, DFG) under grant RU-1903/3-1.

Authors’ address: S. D. Fink, M. Pfretzschner, and I. Rutter, University of Passau, Faculty of Computer Science and Mathematics, Innstraße 33, 94032, Passau, Deutschland, Germany; emails: {finksd, pfretzschner, rutter}@fim.uni-passau.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1084-6654/2023/10-ART1.10 $15.00

https://doi.org/10.1145/3611653

ACM Journal of Experimental Algorithmics, Vol. 28, No. 1, Article 1.10. Publication date: October 2023.
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 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. By 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]. 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 in Section 4 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 \( L \). While PC-trees have no designated root nodes and are thus conceptually unrooted, in practice they are usually still rooted at an arbitrary inner node or leaf to simplify implementation.
Fig. 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 thick edges represent the terminal path with terminal nodes \(t_1\) and \(t_2\). The white nodes represent empty nodes, the black nodes represent full nodes and the gray nodes represent partial nodes. 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 \(c\).

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.\(^1\) 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.

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”, that is have their incident edges ordered in a compatible way, to ensure that all full leaves are consecutive. Hsu and McConnell [13, 14] 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. Figure 1(a) illustrates the terminal path.

When updating \(T\) in order to apply the restriction, every P-node (C-node) on the terminal path is split into two P-nodes (two C-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 subtrees with only empty leaves, 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 [13, 14]. Figure 1 shows an example of the results of this update, while Figure 2 details the changes made to the terminal path during the update.

It remains to efficiently find the terminal edges, and thus the subtrees with mixed full and empty leaves. To do so, Hsu and McConnell extend the categorization of the leaves as either full or empty to the inner nodes of the tree as follows; see also Figure 1(a). An inner node is full, if all but one of

\(^1\)Note that if \(|R| \in \{0, 1, |L| - 1, |L|\}\), the restriction is already trivially satisfied by \(T\). Thus, such restrictions require no changes to the PC-tree and, as they can be recognized easily, we ignore them for the remainder of this work.
its adjacent subtrees, that is the separate trees created by removing the node, have only full leaves. An inner node is partial, if it has at least one full neighbor and two or more non-full neighbors. Otherwise, that is without a full neighbor, an inner node is empty. Then, an edge is terminal if and only if it lies on a path between two partial nodes [13, 14]. Note that the terminal path may contain empty nodes, but cannot contain full nodes, because no full node can be on a path between partial nodes.

Hsu and McConnell incrementally compute this labeling as follows. All inner nodes are initially considered empty, while the leaves are labelled full or empty according to their containment in \( R \). An inner node turns partial once it has at least one full neighbor. A partial node turns full once all but one of its neighbors are full.\(^{2}\) Assigning the labels and subsequently finding the terminal edges can be done by two bottom-up traversals of the tree, first choosing an arbitrary node of the tree as root. In the following section, we discuss in greater detail how these steps can be implemented. 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].

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 nodes lie on one side of the path.
4. Split each node on the path into two nodes, one incident to all edges to full nodes and one incident to all edges to empty nodes.
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

\(^{2}\)Note that all neighbors can only be full if the restriction makes all leaves consecutive, a case which we trivially excluded.
Fig. 3. (a) A newly created block $x$ of size one growing by one as the neighboring arc $v$ becomes full and is appended. (b) Two blocks that share a common neighbor $x$ being merged once $x$ becomes full. The block-spanning pointers are shown as blue, dashed half-arcs.

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, see the blue dashed arcs in Figures 3, 7(c), and 7(d) for an example. Whenever a neighbor of a C-node becomes full, either a new block is created for the corresponding arc of the C-node (Figure 3(a) left), an adjacent block grows by one arc (Figure 3(a) right), or the two blocks that now became adjacent are merged (Figure 3(b)).

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 $R_1, \ldots, R_k$ takes $\Theta(|R| + \sum_{i=1}^k |R_i|)$ time [13, 14]. 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 following subsections, 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; see Figure 4(a). 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, and therefore lead 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 $\alpha$ is the inverse Ackermann function. Although this makes the overall runtime super-linear, the experimental evaluation in Section 4 shows that this actually improves the performance in
In the following, we describe further details in which our implementations differ from the description given by Hsu and McConnell and explain the corrections needed for a working implementation. Section 3.1 describes how the labeling procedure (step 1) can be properly implemented. Note that the technical complications involving arcs and block-spanning pointers only concern the missing C-node objects of HsuPC. UFPC uses direct references to adjacent C-nodes instead of arcs and does not need to maintain block-spanning pointers. Thus, those parts can be greatly simplified for our second implementation, but we still give the full details in our description using the perspective of the HsuPC datastructure. The same holds for Section 3.2, where we give a corrected algorithm for enumerating the terminal path (step 2). Section 3.3 then describes the generic steps needed to detect impossible restrictions. Lastly, Section 3.4 explains the differing update procedure of UFPC (steps 5 and 6), while the update procedure of HsuPC follows Hsu and McConnell’s original description.
ALGORITHM 1: Label full and partial nodes given a set of consecutive leaves $R$.

**LABELNODES($R$):**

1. $Q \leftarrow \{\}$; // A queue of unprocessed arcs.
2. for $l \in R$ do
3.   label leaf $l$ as full;
4.   add the single incident arc of $l$ to queue $Q$;
5. while $Q$ is not empty do
6.   remove next arc $a$ from $Q$;
7.   mark the twin arc of $a$ as leading to a full node;
8.   if $a$ has a reference to its target node $u$ then // $u$ must be a P-node
9.     add $a$ to $u$'s list of full neighbors $F_u$ and mark $u$ as partial;
10.    if $|F_u| = \deg(u) - 1$ then // P-node became full
11.       mark $u$ as full;
12.       if $u$ has a parent arc that is not full then
13.         add $u$'s parent arc to $Q$; // optimization for the common case
14.         else
15.          search through all arcs incident to $u$ to find the single non-full arc $f$;
16.          add $f$ to $Q$;
17.     else // manage blocks at C-node
18.       create/append/merge the full blocks adjacent to $a$, yielding a full block $b$;
19.       if both ends of $b$ are adjacent to a single non-full arc $f$ then
20.         mark $b$ (as proxy for the missing C-node object) as full;
21.         add $f$ to $Q$; // C-node became full
22.       else
23.         mark $b$ (as proxy for the missing C-node object) as partial;

3.1 Efficiently Labeling and Finding Partial Nodes

In our description of the labeling step, we follow the general procedure of Hsu and McConnell [13], using a bottom-up traversal of the tree, starting at the full leaves. Algorithm 1 gives the pseudo-code for this procedure. Initially, all inner nodes are considered empty. Recall that arcs only have a reference to their target node if it is a P-node. We thus need to be careful when following edges as there might not exist an object for the (C-)node at the other end of the edge. To implement the traversal, we keep a queue of unprocessed arcs pointing from full nodes to non-full neighbors. Furthermore, each P-node object stores a list of incoming arcs from full neighbors. At the start of the traversal, the queue is initialized with the incident arcs of all full leaves. If the arc at the front of the queue has a reference to its target node object (via its twin arc; see Figure 4(a)), this has to be a P-node (as only these are represented by actual objects) and we can simply append the incoming arc to the P-node’s list of full children. Recall that each full node has exactly one non-full neighbor. Thus, if this list reaches a size one smaller than the P-node’s degree, we mark it as full and enqueue the pointer to the single non-full neighbor of the P-node. Usually, this non-full neighbor is the parent of the P-node, so we directly enqueue the parent arc if it is not null and the parent is not yet full. The other case, when the now-full P-node is the root or when a parent has become full before its child P-node, is missing in the description by Hsu and McConnell. Here, we need to search all incident arcs for the single arc pointing to a non-full node, and queue this arc instead. As the number of searched arcs is bounded by the number of full leaves, this does not affect the overall runtime.
If the arc at the front of the queue does not have a reference to a (P-)node, we need to maintain (i.e., create/append/merge) the block-spanning pointers around the respective C-node $x$. We will also use the full blocks managing these block-spanning pointers as proxy objects for partial C-nodes. The merging of full blocks is illustrated in Figure 3, see the book chapter by Hsu and McConnell [13] for more details. If the new endpoints of the C-node’s full block are now adjacent to the same arc pointing to a non-full neighbor, the C-node is full and we queue the arc to this neighbor $z$. Note that similar to the case of P-nodes, this node is most often but not necessarily (e.g., if the root became full) the parent arc. Still, there is no explicit search required, as we already know the arc to the only non-full neighbor. Once the queue runs empty, the labeling is complete. The partial nodes are now represented by the non-full P-nodes that have full neighbors (but at least two non-full neighbors) and the full blocks around non-full C-nodes (i.e., where both endpoints are not adjacent to the same arc).

Finally, let us make a further minor correction regarding the original description. In their definition of the data structure Hsu and McConnell note that “no two C nodes are adjacent, so each of these edges [incident to a C-node] has one end that identifies a neighbor of the C node, and another end that indicates that the end is incident to a C node, without identifying the C node” [13, page 32-10]. See Figure 5 for a simple counterexample where two C-nodes are indeed adjacent. Hsu and McConnell use this property within their planarity test and after they test whether the endpoints of a full block of a C-node $x$ are adjacent to the same arc leading to a neighbor $z$: “if $x$ passes this test, it is full, and the full-neighbor counter of $z$ is incremented” [13, page 32-11]. According to their argumentation, $z$ has to be a P-node as no two C nodes are adjacent, which is incorrect. Still, the important information is not the type of the nodes, but that the neighbor $x$ of the non-full node $z$ became full and we thus need to queue the arc from $x$ to $z$. Thus, our queue-based approach also correctly handles the case where a chain of multiple adjacent C-nodes becomes full in a cascading fashion.

**3.2 Efficiently Finding the Terminal Path**

Hsu and McConnell show that an edge is terminal if and only if it lies on a path in the tree between two partial nodes. This allows them to conduct parallel searches, starting at every partial node and extending ascending paths through their ancestors at the same rate [13]. Whenever an already processed node is encountered, expansion of the current path is stopped and the path is instead merged into the path of the already processed node. Once all paths have met, the search can be terminated. There are two possible structures for the finished terminal path, assuming the restriction is possible. Let the apex be the highest node on the terminal path, i.e., the lowest node that is an ancestor of all other nodes on the terminal path. The two cases can now be differentiated based on the position of the apex, which in turn depends on the position of the root node:
Fig. 6. Two examples of a PC-tree with root $r$. (a) An I-shaped terminal path where $t_2$ is both apex and terminal node. (b) An A-shaped terminal path where two ascending paths join in the apex $a$.

**I-Shaped:** If the apex lies on one of the ends of the terminal path and is therefore a terminal node at the same time, the terminal path extends from the other endpoint of the path upwards to the apex, as shown in Figure 6(a). In this case, every node on the terminal path has exactly one child on the terminal path, except for the lower terminal node $t_1$. This also covers the special case where there is only a single terminal node $t_1 = t_2$.

**A-Shaped:** If the apex does not lie on one of the ends of the terminal path, two ascending paths join in the apex, as shown in Figure 6(b). In this case, the apex $a$ has two children on the terminal path, the terminal nodes $t_1$ and $t_2$ have none, and all other nodes on the terminal path have exactly one child that is also on the terminal path.

The apex can be found using a bottom-up traversal, this time starting from all partial nodes found during the labeling step. As before, traversing a P-node can be done easily, but again as C-nodes have no object registered with their incident arcs, finding their parent arc can be difficult. To do so, Hsu and McConnell distinguish two cases depending on whether they arrived at the C-node via an arc from a partial or empty node, or via an arc from a full node. We note that the parallel searches can never actually ascend to another node coming from a full node, as a full node cannot be part of the terminal path. Thus, we focus exclusively on the case where we arrived at a C-node from a partial or empty neighbor. Here, Hsu and McConnell “look at the two neighbors of the child edge in the cyclic order, and one of them must be the parent edge” [13, page 32-12]. This is only correct in the first case of the four cases shown in Figure 7, as the parent arc might lie behind a full block adjacent to the incoming arc or the current node might be the apex of the terminal path.

Algorithm 2 shows our corrected procedure for finding the parent arc for any given arc $a$. It either returns said arc, detects that we can stop ascending or aborts the algorithm as the restriction is impossible. If the parent arc of an empty C-node is part of an I-shaped terminal path, it has to lie next to the incoming terminal path arc $a$, otherwise the empty neighbors would be non-consecutive and the restriction thus impossible; see Figure 7(a) and lines 11 to 14 of Algorithm 2. If the parent arc of a partial C-node is part of an I-shaped terminal path, it has to lie on the opposite side of the full block adjacent to the incoming terminal path arc $a$ for the restriction to be possible; see Figure 7(c) and lines 17 and 21. If this arc is not the parent arc, it may still be part of the terminal path if the current node is the apex of an A-shaped terminal path as shown in Figures 7(b) and 7(d), see also line 15 as well as lines 18 and 22. Note that in this case, the second incoming terminal...
Fig. 7. Different cases of the terminal path crossing a C-node. Empty, partial, and full nodes are drawn in white, gray, and black, respectively. The thicker, green nodes and edges are part of the terminal path, the blue dashed half-arcs depict the block-spanning pointers. The edges are oriented towards the root node. In case (c), the node can be a final node of the terminal path, i.e., a terminal node. Thus, there might be zero, one or two incident terminal edges. In cases (b), (c), and (d), the node can also be the root, i.e., lacking a parent arc.

path arc will only be found at a later iteration, as we cannot identify the shape or position of the apex beforehand. If we neither find a parent arc nor a second incoming arc, ascending through the current node would make the current restriction impossible as full and empty leaves could then not be separated on different sides of the terminal path. We can thus simply stop ascending at any C-node for which we did not find a parent (line 26). Similarly, we stop ascending if we arrive at a full node (which cannot be part of the terminal path) or the root node (lines 2, 5 and 23).

To now correctly enumerate the terminal path, we again use a queue of unprocessed arcs; see Algorithm 3. We initialize the queue with the parent arcs of all partial nodes found in the labeling step, which is easy to do for partial P-nodes; see lines 15 to 23 of Algorithm 3. For partial C-nodes, we can check for a parent arc that is adjacent to the respective full block, see Figure 7(c) and line 6. If we are unable to find a parent for the current partial node, we store it both as apex and as highest point of a stopped search path (lines 12 and 14).

We process the queue arc by arc, using FindParentArc from Algorithm 2 to find the parent arc of each dequeued arc (lines 24, 25, and 31). Recall that we never process an arc twice by merging the respective search paths once we encounter an already processed arc (lines 26 and 27). If we are unable to find a parent for the current arc, we store it as highest arc of a stopped search path (line 43). We report an impossible restriction if multiple search paths stopped early, as the terminal path would then be disconnected. This is handled by the SetApex method in lines 12, 21, and 45; see Section 3.3 for more details on impossible restrictions. We also stop processing if there is only one arc left in the queue and we have not yet stopped a search path at a highest point (for which we could not find a parent arc); see lines 28 to 30. In this case, all parts of the parallel search have already converged into a single ascending path and we are extending the terminal path above the actual apex.

The apex will be the node that has two incident terminal edges in case of an A-shaped terminal path, or the highest partial node in case of an I-shaped terminal path. The first case can be identified by checking whether a node has two predecessors on the terminal path; see lines 33 and 40. The backtracking needed in the second case can be done in constant time (lines 46 and 47) by storing the highest partial predecessor for each processed arc (lines 9, 13, 18, 22, and 37).

Observe that the number of arcs on the terminal path is proportional to the length $p$ of the terminal path. Furthermore, we only check a constant number of neighbors of each arc and any
**Algorithm 2**: Process an arc $a$ on the terminal path to either return the parent of its target node, detect that we can stop ascending and return null, or abort the algorithm as the restriction is impossible.

**FindParentArc($a$):**

1. If $a$ has a reference to its target node $u$ then // $u$ must be a P-node
   2. If $u$ is full then
      3. // we already ascended too far
      4. Else if $u$ has a parent arc then
         5. Return parent arc of $u$; // $u$ is a P-node
      6. Else
         7. // we reached the root and cannot ascend further

2. Else // C-node
   3. $a_1, a_2 \leftarrow$ the two arcs adjacent to $a$;
   4. $b_1, b_2 \leftarrow$ the full blocks ending at $a_1, a_2$ or null;
   5. If $b_1 = \text{null}$ then
      6. If $b_2 = \text{null}$ then // no blocks adjacent
         7. // we can ascend if parent arc is adjacent to $a$; see Figure 7(a)
         8. If $a_1$ leads to parent then
            9. Return $a_1$;
         10. Else if $a_2$ leads to parent then
             11. Return $a_2$;
         12. Else
            13. // no parent nearby, we cannot ascend further; see Figure 7(b)
      14. Else if parent arc is at other side of $b_2$ then
         15. Return parent arc adjacent to $b_2$ similar to above; // see Figure 7(c)
      16. Else
         17. // no parent nearby, we cannot ascend further; see Figure 7(d)
   18. Else if $b_2 = \text{null}$ then
      19. If parent arc is at other side of $b_1$ then
         20. Return parent arc adjacent to $b_1$; // see Figure 7(c)
      21. Else
         22. // no parent nearby, we cannot ascend further; see Figure 7(d)
   23. Else if $b_1 = b_2$ then
      24. // the C-node is full and we ascended too far
   25. Else // two different full blocks adjacent
      26. Raise impossible restriction;

27. Return null; // we cannot ascend further

The highest arc requiring backtracking is at most $p$ nodes above the actual apex. Thus, the overall runtime of our search for the terminal path is in $O(p)$ if the restriction is possible. Note that this slightly refines the analysis of Hsu and McConnell [13], who sometimes scan the full children of a node and thus have a runtime in $O(p + |R|)$.

### 3.3 Efficiently Detecting Impossible Restrictions

Recall that a restriction is possible if and only if (1) all terminal edges form a path and (2) all nodes on the terminal path can be flipped or rearranged such that their empty and full children are consecutive while separated by the terminal edges. The only way to violate the first property
Algorithm 3: Enumerate the terminal path as a list of predecessors of the apex.

EnumerateTP():

1. apex ← null; // A partial P-node or block of a partial C-node in case of an I-shaped apex, a pair of terminal path arcs pointing to the same node in case of an A-shaped apex.

   // The chain of predecessors of the apex forms the terminal path. Property partial-predecessor indicates the highest partial node on the terminal path below, and that an arc is part of the terminal path.

2. foreach arc a do  a.predecessor ← null; a.partial-predecessor ← null;

3. highest ← null; // The node, block or arc at which a search path stopped. Knows the highest partial node on said path as its partial-predecessor.

4. Q ← {}; // A queue of unprocessed arcs.

   // Initialize Q with partial nodes

5. foreach block b at a partial C-node do

6.   if the parent arc p of the C-node lies directly before or after b then

7.     if p.partial-predecessor ≠ null then

8.        raise impossible restriction; // see Figure ??(a) and Section 3.3

9.       p.partial-predecessor ← b;

10.      add p to queue Q;

11.     else

12.      SetApex(b); // I- or A-shaped

13.      b.partial-predecessor ← b;

14.      highest ← b;

15. endforeach partial P-node u do

16.   p ← parent of u;

17.   if p ≠ null then

18.      p.partial-predecessor ← u;

19.      add p to queue Q;

20.   else

21.      SetApex(u); // I- or A-shaped

22.      u.partial-predecessor ← u;

23.      highest ← u;

   // continued on next page...

is when a node has more than two incident terminal edges. P-nodes can detect directly when this case occurs, while a C-node with more than two incident terminal edges leads to multiple stopped search paths, as the parent edge of a C-node is only found when it is next to the incoming terminal path edge. This is detected by our algorithm when calling SetApex multiple times with different highest arcs.

As P-nodes allow arbitrary arrangements of their children, only C-nodes can violate the second property by either having multiple distinct full blocks, by having a full block that is not adjacent to all incident terminal edges, or by having no full block and two or more non-adjacent terminal edges. All these cases lead to a disconnected terminal path and thus multiple stopped search paths. In our pseudo-code there is one exception to this: two full blocks or terminal path edges before and
Experimental Comparison of PC-Trees and PQ-Trees

// Main Routine: Process Queue
while Q is not empty do
    remove next arc a from Q;
    if a was already visited then
        continue;
    if Q reached length 0 and highest = null then
        // search paths already converged, we are extending above apex
        highest ← a;
        break;
    p ← FindParentArc(a);
    if p ≠ null then
        if p.predecessor = null then
            if p.partial-predecessor ≠ null then
                raise impossible restriction; // see Figure ??(b) and Section 3.3
                p.predecessor ← a;
                p.partial-predecessor ← a.partial-predecessor;
                add p to Q;
            else
                SETAPEX((a, p.predecessor)); // A-shaped
        else
            if highest = null then
                highest ← a;
            else
                SETAPEX((a, highest)); // A-shaped
    if apex = null then
        // I-shaped, the apex is the highest partial node
        SETAPEX(highest.partial-predecessor);

after the parent edge of a C-node could both independently use the parent edge for ascending; see Figure ?? We detect this situation with two full blocks, one full block and one terminal path edge, and two terminal paths in lines 8 and 35 of Algorithm 3 and line 9 of Algorithm 4, respectively. Here, we use the node property partial-predecessor, which is also used to efficiently find the apex when we extended an I-shaped terminal path too far, to detect that an edge has already been used for another path.

Finally, note that our pseudo-code might identify the same node as apex twice if it is a partial node for which we could not find a parent arc, and it is apex of an A-shaped terminal path, i.e., has two predecessors on the terminal path; see lines 11 and 16 of Algorithm 4. Otherwise, we report an impossible restriction once we encounter a second apex (or highest arc, i.e., stopped search path) as shown in Algorithm 4.

3.4 Deletion and Contraction
Deleting and inserting new edges is simple when using the arc-based tree representation described by Hsu and McConnell. When using a doubly-linked tree structure similar like the one used by UFPC, no explicit edge objects exists and they are instead encoded by the child-parent relationship
ALGORITHM 4: Set a P-node, full block around a C-node, or a pair of arcs \( o \) to be the apex or report an invalid restriction.

**SetApex**(*)

```latex
1  if \( o \) is a pair of arcs \((a, b)\) then // validate an A-shaped apex
2    if \( a \) or \( b \) point to a P-node then
3      if \( a \) and \( b \) do not point to the same P-node then
4        raise impossible restriction;
5      else // C-node
6        if \( a \) and \( b \) do not lie next to each other // see Figure 7(b)
7          or \( a \) and \( b \) do not lie next to the same full block // see Figure 7(d)
8        then
9          raise impossible restriction; // see Figure ??(c)
10       if apex is a P-node then
11         if \( o \) is the pair of arcs pointing to apex then
12           apex ← \( o \);
13         else
14           raise impossible restriction;
15       else if apex is a block at a C-node then
16         if \( o \) is the pair of arcs that lie before and after apex then
17           apex ← \( o \);
18         else
19           raise impossible restriction;
20       else if apex is a pair of arcs then
21         raise impossible restriction;
22     else // apex = null
23     apex ← \( o \);
```

of the nodes. This means that for the deletions and contractions in steps 5 and 6 of the overall update procedure, the child-parent relationship needs to be set immediately and correctly for every change and cannot easily be updated later, as done by Hsu and McConnell. Thus, UFPC uses a different approach for these two steps. First, when creating the central node, we need to make sure that we directly assign it its neighbors. This is trivial if the apex is a C-node and thus can simply be reused as is. Otherwise, we create a new C-node and add up to four neighbors: the apex’ first child on the terminal path, a newly created P-node that was reassigned as parent of all full children of the apex, the second child on the terminal path, and finally the apex with all its empty children remaining. Note that not all four neighbors might exist or be required, e.g., when the apex has no full or empty children or the terminal path is I-shaped. Furthermore, the root of the tree is either among the full or the empty children and thus the node that is still connected to the root needs to be installed as parent of the new central node. Second, we iteratively contract a child of the central node that is part of the terminal path into the central node. C-nodes can again be simply merged, while a P-node \( x \) needs to be split into a full and an empty node. P-node \( x \) is then replaced by the full node and the empty node with the other terminal path neighbor of \( x \) in between, if the latter exists.
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 methods 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 \cite{booth1976testing}, one of the initial applications of PQ-trees. This test incrementally processes vertices one by one according to an \textit{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 \textit{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.

In this way, we create one test set \textsc{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 1,000 to 20,000 in steps of 1,000 and each edge count \( m \in \{2n, 3n - 6\} \). Altogether, this test set contains 199,831 instances, whose distribution with regards to tree and restriction size and terminal path length is shown in Figures 8(a) and 9(a).

To guard against overly permissive implementations, we also create a small test set \textsc{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 8(b).

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 \textsc{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 \cite{booth1976testing}, thus calling the set \textsc{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 large restrictions, we filter out restrictions with less than 25 full leaves, resulting in \textsc{DIR-PLAN} containing 564,300 instances.

In order to evaluate additional test cases that stem from a different application, we also create a fourth test set \textsc{MAT-POS} containing instances of the consecutive ones problem. To this end, we generate 1,000 matrices in \( \{0, 1\}^{m \times n} \) with \( m, n \) chosen uniformly at random from the interval \([10, 500]\). We pick a random range of cells as the consecutive block of ones in each row and then shuffle

\footnote{Note that simple graphs with more than \( 3n - 6 \) edges are always non-planar, while connected graphs with only \( n \) edges are always planar. Furthermore maximal planar graphs, i.e., those with exactly \( 3n - 6 \) edges, have an up to mirroring unique planar embedding. Thus, we chose \( 2n \) edges as a natural middle ground between being trivially planar and having no embedding choices.}
Fig. 8. Distribution of tree and restriction size for the data sets (a) SER-POS and (b) SER-IMP. Please note the different color scales.

Fig. 9. Distribution of tree and restriction size for the data sets (a) SER-POS and (b) SER-IMP.

the columns of the matrix, which yields a yes-instance of the consecutive ones problem. We also add a zero column to each matrix to ensure that the PC- and PQ-trees represent the same valid permutations. In total, the set MAT-POS contains 249,080 restrictions whose distribution is shown in Figures 9(b) and 12(b).

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 two correctly handle all our test cases (OGDF, 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, Gregable) 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
## Table 1. Implementations Considered for the Evaluation

| 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                             |
| Ooma [4]      | PC-Tree    | planarity test prototype         | C++      | −       | −      | http://www.immuni.com/~tonial.sh                                    |
| Gregable      | PQ-Tree    | automatic layout of biclusters   | C++      | ×       | ✓      | https://bioinformatics.cs.vt.edu/~murphy/papers/BiVoC               |
| BiVoC [9]     | PQ-Tree    | student project                  | C++      | ×       | ✓      | 252                                                                  |
| OGDPh [5]     | PQ-Tree    | planarity testing                | C++      | −       | n.a.   | 1551                                                                 |
| Reisle        | PQ-Tree    | visual graph editor              | C++      | ×       | ×      | 73                                                                   |
| GraphSet [8]  | PQ-Tree    | visual graph editor              | C++      | ×       | ×      | 252                                                                  |
| Zanetti [17]  | PQ-Tree    | our C++ conversion of Zanetti    | Java     | ×       | ×      | 728                                                                   |
| CppZanetti    | PQR-Tree   | extension of PQR-Trees           | C++      | ×       | ×      | 728                                                                   |
| JGraphEd [11] | PQ-Tree    | visual graph theory tool         | Java     | ×       | ×      | 11                                                                    |
| GraphTea      | PQ-Tree    | interval graph detection        | Python    | ×       | ×      | https://github.com/rostam/GTea/                                      |
| GTea [7]      | PQ-Tree    | consecutive-ones testing        | Python    | ×       | ×      | 11                                                                    |
| SageMath      | PQ-Tree    | interval graph detection        | Python    | ✓       | 0      | https://github.com/jppzanetti/PQTree                                 |
| CppZanetti    | PQR-Tree   | our C++ conversion of Zanetti    | C++      | ×       | ×      | 728                                                                   |
| JGraphEd      | PQ-Tree    | visual graph tool                | Java      | ×       | ×      | 11                                                                    |
| GraphTea      | PQ-Tree    | interval graph detection        | Python    | ×       | ×      | 11                                                                    |
| TryAlgo       | PQ-Tree    | consecutive-ones testing        | Python    | ×       | ×      | 11                                                                    |

Implementations that are entirely unusable as they are incomplete or crash/produce incorrect results on almost all inputs (marked with −) and those where no generic executable is available are not included in the comparison. 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 the number of errors for the 203,630 restrictions in the sets SER-POS and SER-IMP and the 1000 matrices in the set MAT-POS. 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.
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 an 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 (CppZanetti). 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 are given in Appendix A.

Each experiment was run on a single core of an Intel Xeon E5-2690v2 CPU (3.00 GHz, 10 Cores) with 64 GiB of RAM, running Linux Kernel version 5.10. Implementations in C++ were compiled with GCC 10.2.1 and optimization -O3 -march=native -mtune=native. Java implementations were executed on OpenJDK 64-Bit Server VM 11.0.14 and Python implementations were run with CPython3.9.2. 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 graphs from which we derive our test data. We did not analyze the memory consumption of the implementations, as in theory the linear runtime also bounds the memory. Furthermore, the size of the used datastructures only differs by a small constant factor. In practice, the use of various different libraries also makes it hard to compare the actual amount of memory used.

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.\footnote{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.} 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 10 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 10(a) clearly shows that for all implementations the runtime is linear in the size of the restriction. Figure 10(b) suggests that 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 11(a). 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 11(b) 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 10(a).

Figure 12(a) shows the performance on the dataset MAT-POS depending on the restriction size. The ranking of the implementations is similar to the results on SER-POS; only OGDF performs noticeably worse on MAT-POS. Note that Figure 12(b) shows that this dataset contains more larger restrictions, which effectively result in rather short terminal paths (see Figures 9(a) and 9(b)). As the rows get darker towards their end in Figure 12(b), the restrictions originating from the consecutive ones matrices exhibit a correlation between restriction size and tree size. Thus, the runtime of every implementation would exhibit a linear dependence on the tree size on this dataset. We
Fig. 10. Runtime for SER-POS restrictions depending on (a) restriction size and (b) tree size. The solid lines show the arithmetic mean for the respective implementation.

Fig. 11. (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.

Fig. 12. (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. (b) Runtime for SER-POS restrictions depending on the terminal path length.
therefore focus on the test cases generated using the planarity test for a more detailed analysis of the performance.

Figure 13 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 MAT–POS 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 (see also Appendix A). Figure 14 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 14(a). As the initial tests already showed, the implementation has multiple flaws; one major issue is already described in an issue on GitHub, while we give a small example of another independent error in Figure 17. 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 15(a) 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 15(b) 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).
Experimental Comparison of PC-Trees and 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 [13, 14], 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. We give a corrected version that also includes several small simplifications.

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 $\alpha$ 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 are fully correct. Only two 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 times 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.

APPENDIX

A DETAILS ABOUT EVALUATED IMPLEMENTATIONS

**BiVoC, Gregable.** In the implementations of BiVoC and Gregable, we improved the mapping from the input to the tree’s leaves by replacing std::map with std::vector, as suggested in the code’s comments. As a result, this mapping now takes constant time. The Bubble method of BiVoC caused segmentation faults due to undefined behavior, because a set iterator is dereferenced and incremented after its corresponding element has been removed. We resolved this issue for our evaluation. Still, the method qNextChild of BiVoC sometimes caused program hangs due to undefined behavior, when the past-the-end iterator of an empty set is incremented. In the Gregable repository, the author notes that the code “is known to be buggy on some rare inputs. A believed to be correct, but harder to use version of this code can be found as a library within BiVoC”. In our tests, Gregable produced a segmentation fault in the reduce-step on one input, while BiVoC failed for 73 instances. Figure 16 shows an example where Gregable’s implementation produces an invalid PQ-tree.

**GraphSet.** In the implementation of GraphSet, we removed the entanglement with Microsoft Foundation Classes by replacing its data structures with their corresponding variants from the standard library. We were unable to get GraphSet’s Bubble method to work for our tests. Instead, we used the approach from their quadratic-time variant of Booth and Lueker’s planarity test, where they traverse the entire tree before each reduction in order to find and prepare the pertinent subtree. Still, GraphSet produced segmentation faults due to null pointer dereferencing in Template Q3 and several invalid writes when accessing already freed memory.

**TryAlgo.** In June 2020, the authors of the TryAlgo implementation noted on their website that they “have problems implementing this data structure, and cannot provide at this point a correct implementation in tryalgo”. Furthermore, they note that “the current implementation has a complexity in the order of \( n \cdot m \), however an implementation in \( O(n + m + s) \) is possible”. As we thus assumed their implementation to be neither correct nor linear-time, we excluded it from our evaluation.

**SageMath.** The main routine set_contiguous of the PQ-tree of SageMath recursively traverses the tree starting from its root as follows: it first calls set_contiguous recursively on all children of the current node, then calls flatten, calls set_contiguous recursively on all children again and then proceeds to sort the children depending on whether they are full, partial, or empty. The flatten function for removing degree-2 nodes is implemented to recurse itself on all children in the subtree, making the runtime of set_contiguous quadratic in the tree size. We modified the implementation to only flatten the current level and
Experimental Comparison of PC-Trees and PQ-Trees

Fig. 16. Left: A matrix with the consecutive ones property. Middle: The state of Gregable’s PQ-tree datastructure before applying the last restriction of the matrix. Q-nodes are depicted as rectangles, P-nodes as small circles. Right: The state of the datastructure after applying the last restriction \(\{4, 5\}\) to the former tree. The new Q-node created in Template P6 is erroneously chosen as the new root node of the tree. Therefore, the tree loses all other leaves.

Fig. 17. Left: The PQR-tree \([0 \ 1 \ 2 \ (3 \ (4 \ 5)) \ 6]\) not containing any R-nodes with its root Q-node depicted as rectangle and the two P-nodes depicted as small circles. Right: The result of Zanetti’s implementation applying the restriction \(\{3, 4\}\) to the former tree, the tree \([[3 \ 4 \ 5] \ 0 \ 1 \ 2 \ 6]\) which clearly represents a different set of restrictions.

...dropped the second recursive call to `set_contiguous`, improving the runtime to linear in the tree size without generating incorrect results.

**Zanetti.** We found that Zanetti’s data structures became inconsistent after some restrictions, which was also already independently reported on GitHub.\(^5\) This happened mostly after restrictions having a terminal path length of greater than 1. As the restrictions generated when serializing a PC-tree only have very short terminal paths and the inconsistency is usually only found when modifying the same area of the tree again, only few of these cases surfaced in our tests on SER-POS. Only when applying multiple bigger restrictions consecutively, these issues surfaced more often, i.e., at some point during the planarity test for close to all graphs with \(m = 2n\) and also some of the graphs with \(m = 3n - 6\). We also found a second, independent issue, where Zanetti’s implementation generates C-nodes with their children in the wrong order. An example where this happens is shown in Figure 17.

As Zanetti’s Java implementation still has a very good runtime in practice, we decided to port its Java code to C++ to be able to perform a direct comparison with the other C++ implementations. As the implementation uses almost no Java-specific features, the conversion mostly involved replacing Java Object variables with C pointers and Java utility classes with their C++ `stdlib` equivalents. The only non-trivial change was that, because Zanetti stores

\(^5\)https://github.com/jppzanetti/PQRTree/issues/2
the Union-Find information directly in the nodes and not in an external array, we had to implement reference counting for Zanetti’s tree nodes to ensure that the lifetime of nodes which are no longer part of the tree, but still referenced in the Union-Find data structure, is handled properly. We made sure that both the Java and the C++ version not only produced equivalent output, but actually keep the same PQR-tree state in memory. Where both implementations differ is that Java immediately reports inconsistencies of the data structure, e.g., by throwing a NullPointerException, whereas the SIGSEGVFAULT of C++ might not be immediately triggered. This generates a few more data points with an invalid result, where the Java implementation already crashed.

REFERENCES

[1] S. Benzer. 1959. On the Topology of the Genetic Fine Structure. Proceedings of the National Academy of Sciences 45, 11 (1959), 1607–1620. https://doi.org/10.1073/pnas.45.11.1607
[2] Thomas Bläsius and Ignaz Rutter. 2016. Simultaneous PQ-ordering with applications to constrained embedding problems. ACM Trans. Algorithms 12, 2 (2016), 16:1–16:46. https://doi.org/10.1145/2738054
[3] Kellogg S. Booth and George S. Lueker. 1976. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. J. Comput. Syst. Sci. 13, 3 (1976), 335–379. https://doi.org/10.1016/S0022-0000(76)80045-1
[4] John M. Boyer, Cristina G. Fernandes, Alexandre Noma, and José C. de Pina. 2004. Lempel, Even, and Cederbaum planarity method. In Experimental and Efficient Algorithms. Springer Berlin, 129–144. https://doi.org/10.1007/978-3-540-24838-5_10
[5] Guido Brückner and Ignaz Rutter. 2017. Partial and constrained level planarity. In Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA’17), Philip N. (Ed.). SIAM, 2000–2011. https://doi.org/10.1137/1.9781611974782.130
[6] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. 2014. The open graph drawing framework (OGDF). In Handbook of Graph Drawing and Visualization, R. Tamassia (Ed.). CRC Press, Chapter 17.
[7] Alex William Cregten and Hannes Kristján Hannesson. 2017. Implementation of a Planarity Testing Method using PQ-Trees. Technical Report. Reykjavik University. https://skemman.is/bitstream/1946/29618/1/Planarity_testing_with_PQTrees.pdf
[8] Alejandro Estrella-Balderrama, J. Joseph Fowler, and Stephen G. Kobourov. 2009. Graph simultaneous embedding tool, GraphSET. In Graph Drawing. Springer Berlin, 169–180. https://doi.org/10.1007/978-3-642-00219-9_17
[9] Gregory A. Grothaus, Adeeel Mufffi, and T. M. Murali. 2006. Automatic layout and visualization of biclusters. Algorithms for Molecular Biology 1, 1 (2006), 15. https://doi.org/10.1186/1748-7188-1-15
[10] Bernhard Haeupler and Robert E. Tarjan. 2008. Planarity algorithms via PQ-trees (extended abstract). Electronic Notes in Discrete Mathematics 31 (2008), 143–149. https://doi.org/10.1016/j.endm.2008.06.029
[11] Jon Harris. 2004. jGraphEditor – A Java Graph Editor and Graph Drawing Framework. Technical Report. Carleton University, School of Computer Science, Comp 5901 Directed Studies. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.188.5066&rank=1
[12] Wen-Lian Hsu. 2003. An Efficient Implementation of the PC-Tree Algorithm of Shih & Hsu’s Planarity Test. Technical Report. Institute of Information Science, Academia Sinica. http://iasl.iis.sinica.edu.tw/webpdf/paper-2003-PLANAR_implementation.pdf
[13] Wen-Lian Hsu and Ross McConnell. 2004. PQ trees, PC trees, and planar graphs. In Handbook of Data Structures and Applications, Dinesh P. Mehta and Sartaj Sahni (Eds.). Chapman and Hall/CRC, Chapter 32. https://doi.org/10.1201/9781420035179
[14] Wen-Lian Hsu and Ross M. McConnell. 2003. PC trees and circular-ones arrangements. Theoretical Computer Science 296, 1 (2003), 99–116. https://doi.org/10.1016/s0304-3975(02)00435-8
[15] Sebastian Leipert. 1997. PQ-Trees, An Implementation as Template Class in C++. Technical Report. University of Cologne. http://e-archive.informatik.uni-koeln.de/id/eprint/259
[16] Wei-Kuan Shih and Wen-Lian Hsu. 1999. A new planarity test. Theoretical Computer Science 223, 1-2 (1999), 179–191. https://doi.org/10.1016/S0304-3975(98)00120-0
[17] João Paulo Pereira Zanetti. 2012. Complexidade de construção de árvores PQR. Master’s Thesis. Universidade Estadual de Campinas, Instituto de Computação. http://bdtd.ibict.br/vufind/Record/CAMP_0b551865d78ef032289f17f95e3ceec7

Received 4 August 2021; revised 26 January 2023; accepted 19 July 2023