Versioning Tree Structures by Path-Merging

Khaireel A. Mohamed, Tobias Langner, and Thomas Ottmann

Albert-Ludwigs-Universität Freiburg, D-79110 Freiburg, Germany
{khaireel,langneto,ottmann}@informatik.uni-freiburg.de

Abstract. We propose path-merging as a refinement of techniques used to make linked data structures partially persistent. Path-merging supports bursts of operations between any two adjacent versions in contrast to only one operation in the original variant. The superiority of the method is shown both theoretically and experimentally. Details of the technique are explained for the case of binary search trees. Path-merging is particularly useful for the implementation of scan-line algorithms where many update operations on the sweep status structure have to be performed at the same event points. Examples are algorithms for planar point location, for answering intersection queries for sets of horizontal line segments, and for detecting conflicts in sets of 1-dim IP packet filters.

Subject Classifications: E.1 [**Data**]: Data Structures – *trees*; E.2 [**Data**]: Data Storage Representations – *linked representations*; F.2.2 [**Analysis of Algorithms and Problem Complexity**] Nonnumerical Algorithms and Problems – *Geometrical problems and computations.*

Keywords: Partial persistence, path-merging, path-copying, node-copying.

1 Introduction

A data structure supporting access to multiple versions is called a *persistent* data structure, and to date, there are various problems in computer science where such structures are often sought after. This is mainly due to their elegance of maintaining a historical list of the ever-changing primary structure through efficient *update operations*, and then providing convenient ways to get to the archived data by means of well-designed *access operations*.

There are mainly two different degrees of persistence; partial and full. A *partially persistent* structure allows only read access to previous versions, while a *fully persistent* structure allows write access to earlier versions, on top of the read access. In this article, we shall concentrate on the former degree of persistence and discuss the two well-known, classical methods of making linked data structures persistent, namely the 'path-copying' method and the 'node-copying' method. Our perusal of the two methods shall be applied primarily onto binary search tree (BST) structures.

As their names suggest, the path-copying method reproduces an entire path in the BST to effect a single update operation, while the node-copying method

F.P. Preparata, X. Wu, and J. Yin (Eds.): FAW 2008, LNCS 5059, pp. 101–112, 2008.

copies only single nodes (one node in amortized average) per update operation. An update operation creates a new persistent version in the BST. But ever so often, in many domain specific applications, we find that we do not actually require every single one of these versions. Suffice to keep only those versions that we find essential and remove all others as they play no significant part in the broader view of the application it ministers. However, we cannot simply delete those non-essential intermediate versions, as in a partially persistent BST, nodes in one version may share subtrees belonging to other versions.

Hence, we introduce our *path-merging* technique to show how we can comprehensively collate and properly link the non-essential intermediate versions to keep only those versions we think are essential. The correctness of our technique leads to the cost savings in both time and space when compared to the original methods it overlays.

2 Implications of Access Operations in Partially Persistent Structures

Following a series of successful *update* operations on a partially persistent linked data structure τ, we can acquire and assemble a previously persisted version with an *access* operation. For the purpose of our deliberations, we shall exploit τ as a partially persistent binary search tree (BST), where an access operation refers to a search for an item or items in τ, at some past version v_i given a query object q. The accessed set forms a path in τ that starts at the root at v_i, and is extended one node at a time, ensuing an *access heuristic* until the desired items matching the query q are found. Particularly, we shall discuss the access heuristic of Sarnak and Tarjan's *path-copying* method [1] and Driscoll *et al.*'s *node-copying* method [2], which will lead to the discourse of our *path-merging* method later on in this article.

Let us first observe the importance of access operations in persistent data structures when they are used to support surrogate applications.

2.1 Planar Point Location

The key to efficiently solve the planar point location problem is to build a systematically organized data structure to represent a planar subdivision S of n edges. In order to report the face f of S that contains a given query point q, de Berg *et al.* [3] showed how to decompose S into a trapezoidal map $T(S)$, which uses $O(n)$ space and answers the query in $O(\log n)$ time.

An alternative method introduced by Sarnak and Tarjan [1] is to use a partially persistent RB-BST as an improvement over Cole's [4] persistent representation of sorted sets. This also answers the same query in $O(\log n)$ time. The space consumption, however, depends on the method of persistence; the path-copying method takes up $O(n \log n)$ space, while the node-copying method requires $O(n)$ storage.

Vertical lines are drawn through each vertex in S which split the plane into $O(n)$ slabs as shown in Fig. 1(a). Each slab contains the edges of S, which are

associated to the faces just above them, ordered from bottom to top. An RB-BST τ is first built for the left-most slab to hold the sorted edges. Afterwhich, a left to right sweep is carried out on all the slabs, stopping at each vertical line and persisting τ by creating one new version for every update operation. The x-coordinate value of the vertical line is also augmented to the top-most version pointer during the update operation, and these are indexed in another balanced BST on a higher level. This vertical partitioning of the subdivision gives us exactly $2n$ update operations; where at every vertical line, one edge is deleted at version v_i and one other edge is inserted at version v_{i+1}.

Thus, to locate the face f in which q lies, we first perform an $O(\log n)$ time search on the upper level BST to locate the correct version of τ corresponding to the x-coordinate of q, and then access τ at version v_{q_x} to perform a second $O(\log n)$ time search using the y-coordinate of q to determine the correct f.

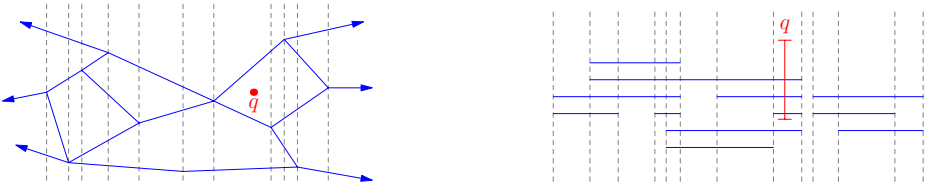

Fig. 1. (a) Planar point location problem. (b) Stabbing range query q on a set of n horizontal line-segments.

2.2 Range Query of Horizontal Line-Segments

We can apply a similar technique as we did before in handling the point location problem with a persistent data structure when we are presented with a set S of n horizontal line-segments. Given a vertical query range q, we are to report all the line-segments in S that intersect q. Again, we draw vertical lines at both endpoints of each line-segment, splitting the plane into at most $2n - 1$ slabs as depicted in Fig. 1(b). Each slab contains the line-segments in S sorted in ascending y order. We then perform an identical sweepline approach to build a partially persistent RB-BST τ on the slabs, such that at every encounter of the vertical line l_i, all line-segments whose left-endpoints match l_i are inserted into τ and all other line-segments whose right-endpoints match l_i are deleted from τ. Every single one of these update operations creates a new version of τ, and unlike the slabs in the point location problem, we tend to face an arbitrarily many insertions and deletions per vertical line as we transit between adjacent slabs during the sweep.

Using the resultant partially persistent structure, we can report the solution in time $O(\log n + k)$, where k is the number of line-segments in S intersecting the vertical range q. Like before, we first execute a binary search to access the correct version v_{q_x} of τ where q lies, and then perform a range query on τ at v_{q_x} to return the active line-segments that intersect q.

2.3 Essential and Non-essential Versions

Clearly, as evidenced by the two examples above, it is not necessary to persist every single version, every time we perform an update operation. It is sufficient to store only those versions that are the collective results of multiple update operations after completely handling an event point. In other words, the versions that we persist must be substantially *essential*, such that all other versions leading to an essential version will have no effect on the overall correctness of the application that τ serves. Thus, we should be able to omit the *non-essential* versions in the partially persistent τ without breaking the temporal flow of the essential versions within.

3 Merging Non-essential Versions in Partially Persistent BSTs

Our problem involving the partially persistent data structures laid out in the previous section is what Driscoll *et al.* and Sarnak and Tarjan [2,1] termed the "persistent sorted set" problem. Here, we maintain a set of elements that changes over time, where each element has a distinct key that is comparable to all other keys in the elements in the same set, such that these keys can be totally ordered. The BST τ is a structure that represents such a set, where it contains one element per node arranged in a symmetric ordering.

We begin by characterizing the different types of nodes that can exist during the intercession of two adjacent essential versions of τ. As we collate the series of non-essential versions effected on τ by the corresponding series of intermediate update operations, we need to distinguish the set of *ephemeral* nodes from the set of *persistent* nodes. They tend to appear simultaneously in τ amidst this transition period, but always in a some formal ordering.

An 'ephemeral node' is a node created during an intermediate update operation and is ephemerally modifiable until it becomes a persistent node, or until it is deleted. On the other hand, a 'persistent node' is a *versioned* node belonging to an existing persistent version v_i of τ, and that any modification on it is strictly not allowed.

Let v_{m-1} be the latest essential version of a partially persistent BST τ under our path-merging technique. Let v_m be the next essential version of τ to be spawned. Let all intermediate versions contributing to the non-essential versions of τ between v_{m-1} and v_m be v_m^*. Then τ at version v_m^* is a 'semi-ephemeral' BST containing both ephemeral nodes and persistent nodes. Note that v_m^* may change over time.

3.1 Path-Merging Via Path-Copying

Sarnak and Tarjan [1] compiled from several sources and presented the idea of the path-copying method to make a linked data structure persistent. During an update operation on τ, we copy only those nodes that are effected by the said operation and percolate the copying procedure to any node with a direct pointer

to the copied nodes. Consequently, if τ is a BST, then entire paths from the effected nodes to the roots are copied, creating a set of search trees for all the partially persistent versions of τ, having different roots per version but sharing common subtrees.

In our *path-merging* technique, only the first update operation that immediately follows the successful persistence of the latest essential version v_{m-1}, adheres to the original path-copying method. This effectively gives us a new semi-ephemeral BST rooted at v_m^*. The newly copied path forms a set of linked ephemeral nodes in v_m^*. All other nodes linked from the subtrees of the ephemeral nodes in v_m^* belong to other partially persistent versions of τ, and they make up the set of persistent nodes in v_m^*. All subsequent update operations contributing to the non-essential versions of τ shall begin with a search at the root at v_m^*. Note that each update will change v_m^*.

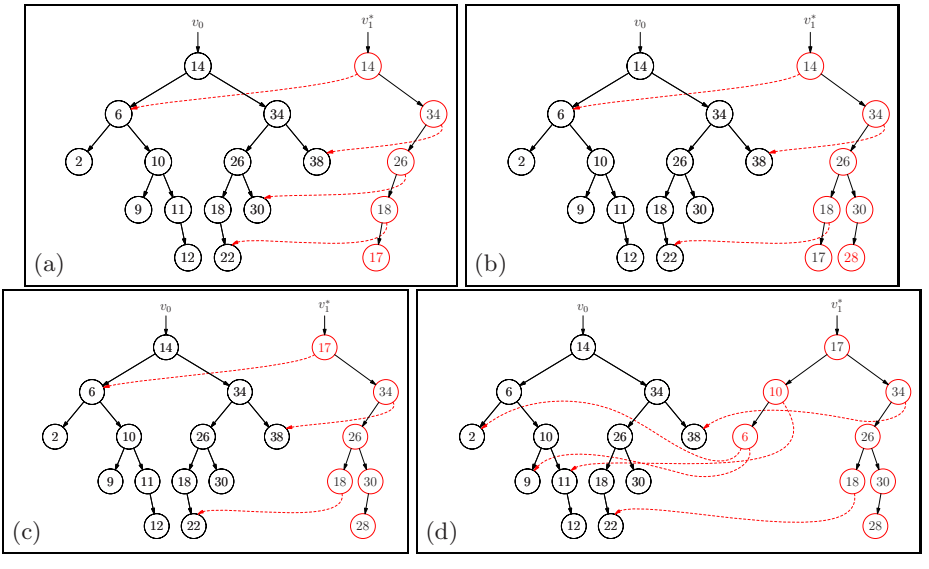

Fig. 2. Path-merging via path-copying. Version v_0: Insert {14, 6, 34, 38, 26, 10, 2, 18, 30, 22, 11, 12, 9}. Version v_1: {Insert {17}, Insert{28}, Delete {14}, Rotate-Left {6}}, showed in sequence from (a) to (d), respectively. The BST rooted at v_1^* during the intercession is a semi-ephemeral structure. Red nodes are ephemeral nodes, and black nodes are persistent nodes.

Let x be a node in v_m^* in which an update operation will be effected upon, and let $c(x)$ denote an ephemeral copy of the node x. Then we only need to copy the persistent nodes in v_m^* if our search for x breaks away from the traversal of the path of ephemeral nodes. We note here that a newly created copy of a persistent node is an ephemeral node.

The next two rules complete the handling of the path-merging technique once we have identified the node x:

1. If x is an ephemeral node, then we treat the update operation on x as a normal ephemeral instruction, overriding the effects of a previous operation made on x.
2. If x is a persistent node, then we perform the update operation on $c(x)$.

Fig. 2 shows an example of path-merging four successive intermediate update operations between v_0 and v_1, with each sub-figure showing an intermediate update operation.

After a series of i intermediate update operations, the "net" set of ephemeral nodes in v_m^* is the result of merging i non-essential versions of the original path-copying method of persistence. What is left to be executed in persisting this set of merged paths for the next essential version of τ is to set the status of all the ephemeral nodes in version v_m to 'persistent'. Since the set of ephemeral nodes in v_m is a connected subtree at the root, we can carry out this change of status in time proportional to the number of ephemeral nodes in v_m using a simple depth-first traversal.

3.2 Path-Merging Via Node-Copying

The node-copying method was conceived to eliminate the shortcomings of the naïve fat node method. Where there can be arbitrarily many outgoing pointers in a fat node in a persistent structure, a node following the node-copying method is allowed to have only a fixed number of such pointers.

In fact, Driscoll *et al.* [2] showed that the improved node for the persistent BST τ needs to store, apart from its key element, only three obligatory pointers: one left pointer, one right pointer, and one other *modification* pointer, each with a version stamp. When such a node becomes full, we create a new copy of the node containing only the newest value of each pointer field. Also, as was with the case of the path-copying method, every time we copy an existing node, its predecessor must be informed of the change. In this case, the parent of the copied node must update its modification pointer to point to the newly copied node and accorded the newest version stamp. But if the parent is also full, then the parent too, must be copied. This copy-percolation process may end up with the root itself being copied.

Accessing Persisted Versions. Unlike the path-copying method where every update operation always produces a new root, the node copying method tends to be more conservative in its expansion of the main tree structure. Every update operation leaves a distinct version stamp in the pointers to the nodes *inside* τ that are effected by the operation. Thus, traversing a persistent BST τ made by the node-copying method must abide by the following access heuristic (which is similar to the access heuristic of the fat node method):

1. Find the correct root for a given version v_i.
2. Traverse the nodes by choosing only pointers with the maximum version stamp that is less than or equal to v_i.

Intermediate Update Operations. As before, let x be a node in v_m^* in which an update operation will be effected upon, and let $c(x)$ denote an ephemeral copy of the node x.

We impose a slight variant on the original node-copying procedures in our path-merging technique when an update operation contributes to a non-essential version. Navigating and manipulating this conservative structure of τ at version v_m^*, influenced by the node-copying method, requires a different kind of attention to be paid when managing the internal nodes. This is not as forthright as compared to the more discernible arrangement of ephemeral and persistent nodes in τ created by the path-copying method. That is, whenever we access τ at v_m^* to search for the node x, we may end up retrieving a path from the root to x that contains both ephemeral and persistent nodes, in random sequence! Furthermore, the nodes in τ have an additional modification pointer field that exhibits special properties when we apply our path-merging technique via the node-copying method.

Hence, here, we extend the notions of our earlier terminologies so as to apply them to the context of the node-copying method, in order to handle the path-merging procedure efficiently.

At one end of the access-spectrum, we have the persistent nodes. A 'persistent' node is a versioned and *full* node in τ belonging to an existing essential version, and is strictly unmodifiable. By *full*, we mean that this persistent node has its key and all three of its pointer fields, particularly the modification pointer, assigned to objects (even to a null object). On the opposite end of the spectrum, lies the ephemeral nodes. An 'ephemeral' node in τ is always created during an intermediate update operation at version v_m^*, either as a new entity or as an ephemeral copy of an existing persistent node. All the contents in this ephemeral node are ephemerally modifiable, and remain so until the node is deleted, or until the node is versioned at v_m. What is now left to be considered are those nodes that are in the middle of the spectrum, and they fit neither of the two descriptions above. We shall call them *semi-persistent* nodes.

An ephemeral node becomes a 'semi-persistent' node, if and only if its modification pointer is empty at the time of spawning a new essential version of τ. In other words, only its key and its left and right pointers can be versioned. The modification pointer, left untouched, is ephemerally modifiable by any future update operation, and remains so as long as the node is in transition. Furthermore, the semi-persistent node becomes a persistent node if at the next essential version v_m the modification pointer is no longer empty.

Given an intermediate update operation, we first invoke the access heuristic on τ at version v_m^* (or at version v_{m-1} if v_m^* does not yet exist) and traverse τ until we arrive at the node x in which to effect the operation. We then execute the node-copying method on x implicitly, while explicitly adhering to an additional set of rules when administering any nodal changes.

Rule 1. If x is an ephemeral node, then we treat the update operation on x as a normal ephemeral instruction, overriding the effects of a previous operation made on x.

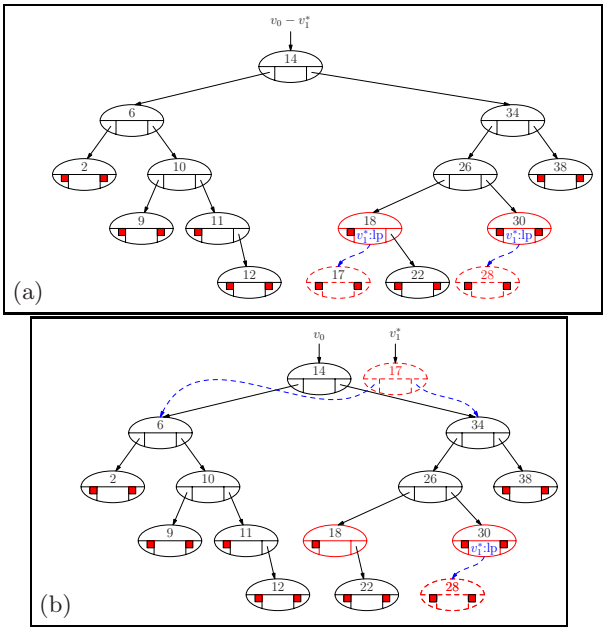

Fig. 3. For example, deleting {14} from (a) results in (b), where the dotted red nodes are ephemeral nodes and all others are semi-persistent nodes. Note that {17} was deleted in (a) without consequence following Rule 1 below, and a copy of the root was made in (b) following Rule 3(a).

Rule 2. If x is a persistent node, then we perform the update operation on $c(x)$.
Rule 3. If x is a semi-persistent node, then we react according to the update operation as follows:

(a) If the update operation changes the key in x, then we perform the change of key in $c(x)$.
(b) If the update operation changes the modification pointer of x *without* causing a *node-contention*, then we simply execute the change. We give a further explanation of what a node-contention is in the next sub-section.
(c) If the update operation changes the modification pointer of x *and* causes a node-contention, then we make a copy of x and resolve the conflict between x and the update operation, in the new $c(x)$.

In addition to the rules above, the pointers in every intermediate update operation effecting or effected by x shall carry the version stamp v_m^*. Fig. 4 depicts an example of path-merging the same four successive intermediate update operations as performed in the previous section.

After a series of i intermediate update operations, the "net" set of ephemeral nodes is again the result of merging i non-essential versions of the original node-copying method. The partially persistent BST τ in Fig. 4 underwent exactly the same sequence of intermediate update operations as was in the case of the tree

produced in Fig. 2. The stark difference between both these trees is that the ephemeral nodes in Fig. 4 are sporadically dispersed, rather than being ordered as a proper subtree as we saw in Fig. 2.

Hence, it may require an $O(n)$ effort to locate the ephemeral nodes in τ at v_m^* in order to change their access statuses when spawning the next essential version v_m. One way to counter this problem is not to find them at all in the first place. That is, instead of stamping v_m^* to pointers effecting x when executing intermediate update operations, we simply stamp the identity of the next essential version v_m. Since each node knows which essential version it belongs to, past or future, stamping the version v_m during the merging of non-essential versions holds for our path-merging technique via the node-copying method.

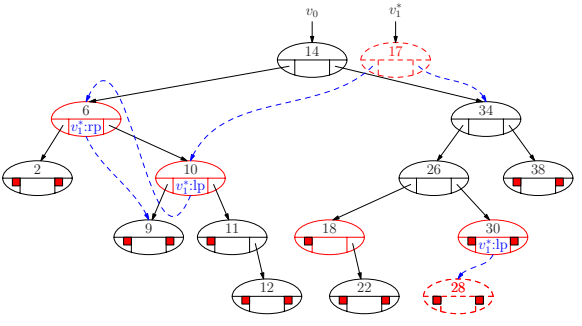

Fig. 4. Path-merging via node-copying. Version v_0: Insert {14, 6, 34, 38, 26, 10, 2, 18, 30, 22, 11, 12, 9}. Version v_1: {Insert {17}, Insert{28}, Delete {14}, Rotate-Left {6}}. Dotted red nodes are ephemeral nodes. Red nodes are 'marked' semi-persistent nodes.

Node-Contentions in Semi-persistent Nodes. A 'node-contention' can occur only in a semi-persistent node during its transition between two essential versions. The contention is caused between a current update operation and the non-empty modification pointer in the node. More specifically, it happens when the modification pointer is already pointing to an object meant to override the node's right (left) pointer, while the current update operation contains an instruction to override the node's left (right) pointer.

When such a case happens, and if we are to replace the modification pointer in favour of the instruction, we then end up with an incorrect routing path in τ. And since we cannot modify the node's original left and right pointers, we resolve this contention by making an ephemeral copy of this semi-persistent node, and directly assign its new left and right pointers from the instruction and from the reference from the original modification pointer. Afterwhich, we delete the modification pointer in the original node to complete the reassignment.

For example, suppose we need to handle one more intermediate update operation in v_1^* in Fig. 4 – to Delete {2}. A search for the node x in v_1^* to effect the delete operation returns the parent of the node {2}, so that x = node {6} and where x is a semi-persistent node. Now, in order to delete {2}, we need the *left* pointer of x to be null. Since x is a semi-persistent node, we can only change

its modification pointer. However, its modification pointer is already assigned to point *right* to node {9}, and that overriding this pointer will be erroneous to v_1^*. Thus, we make a copy of x and resolve the contention by assigning the latest left and right pointers to the new ephemeral $c(x) = \text{copy}(\text{node } \{6\})$, and then remove the modification pointer in the original x. The result is shown in Fig. 5.

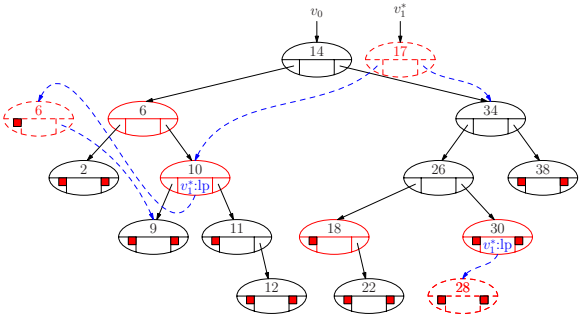

Fig. 5. Resolving a node-contention: Deleting {2} from v_1^* in Fig. 4, where the parent node $\{6\} = x$ was a semi-persistent node, which would have triggered a node-contention if 2 was deleted without checking

4 Analysis of the Path-Merging Technique

The time required for a single update operation in the path-merging technique is the same as the time taken to execute a single update operation by the original underlying methods of path-copying and node-copying [2,1]. However, we note the stern reduction in the overall time by a large factor in the path-merging technique, since no ephemeral nodes are copied more than once.

In terms of space consumption, the path-merging techniques surpass both its predecessors', as it supports bursts of operations between any two essential versions. That is, only the "net" set of ephemeral nodes, which comes from the resultant set of newly created nodes after merging the non-essential versions, contributes to the net increase in space after i intermediate operations. This net increase can even be zero, particularly for the path-merging via node-copying technique, in the case that exactly the same set of keys is inserted into and then deleted from τ several times during the transition period between essential versions. This, compared to the original node-copying method which will end up spawning entire paths after $O(h)$ insertions and deletions of a single key, resulting in $h + (h-1) + \ldots + 1 = O(h^2)$ additional space, where h is the height of the BST τ.

Using the examples in Section 2, we can expect to see a significant reduction in storage space when using the path-merging technique; particularly for the problem of the 'Range Query of Horizontal Line-Segments', where we can anticipate handling arbitrarily many insertions and deletions at an event point. Furthermore, our benchmarked results in Fig. 6 proves the space efficiency between the original path-copying method versus our path-merging technique. For

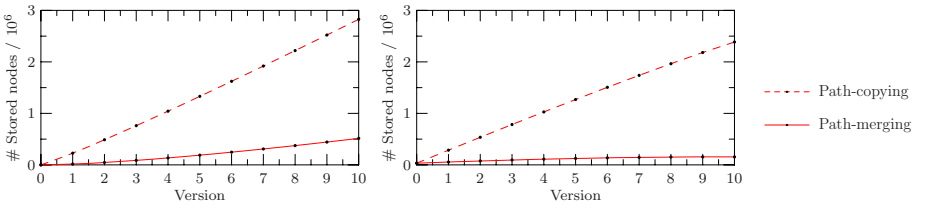

Fig. 6. Space complexity comparison between 'path-merging via path-copying' and the original 'path-copying' methods. Graphs show the execution of 2^{14} operations between v_i and v_{i+1}.

the complete details of the benchmarking process, the reader is invited to check out the work by Langner [5].

5 Conclusion

The pertinence of the path-merging technique reignites the relevance of the applicative components of the classical path-copying and node-copying methods in partially persistent data structures. The technique's strengths lie in their subtle, yet effective ways of merging non-essential versions of the original underlying methods of persistence to derive efficient time and space bounds, that are primed for handling applications where it makes sense to store only the substantially essential versions.

We conclude with a real-world example to prove the usefulness of the path-merging technique. We invite the reader to review our completed works of detecting conflicts in internet router tables [6,7,8], where the 1-dim IP packet filters resemble that of the horizontal line-segments on the plane, similar to the problem discussed in Section 2.2. In the summarized context of the IP-Lookup problem, q is taken to be an incoming packet filter and becomes a stabbing query for the set S of n filters. We need to return the most-specific filter that q stabs. The advantage in this case is two-fold: We were able to solve the conflict detection problem in optimal time of $O(n \log n)$ – *while* building the partially persistent structure; and then utilize the benefits of path-merging's space saving output to store the entire conflict-free set S, which is immediately ready for packet classification.

Now, to appreciate the solution to this problem better, usually, we would require two separate structures to handle the two independent problems of conflict detection and packet classification. But by executing path-merging as described above, we are able to unite them and take advantage of path-merging's adeptness to kill two birds with one stone.

Acknowledgement

This research is funded by the Deutschen Forschungsgemeinschaft (DFG) as part of the research project ,,Algorithmen und Datenstrukturen für ausgewählte diskrete Probleme (DFG-Projekt Ot64/8-3)“.

References

1. Sarnak, N., Tarjan, R.E.: Planar point location using persistent search trees. Communications of the ACM 29(7), 669–679 (1986)
2. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. In: STOC 1986: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing, pp. 109–121. ACM Press, New York (1986)
3. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications, 2nd edn. Springer, Heidelberg (2000)
4. Cole, R.: Searching and storing similar lists. Journal of Algorithms 7(2), 202–220 (1986)
5. Langner, T.: Using partial persistence to support bursts of operations in IP-lookup. Bachelor Thesis, Albert-Ludwigs-Universität Freiburg (March 2007)
6. Maindorfer, C., Mohamed, K.A., Ottmann, T., Datta, A.: A new output-sensitive algorithm to detect and resolve conflicts in internet router tables. In: INFOCOM 2007: Proceedings of the 26th IEEE International Conference on Computer Communications, May 2007, pp. 2431–2435. IEEE Press, Los Alamitos (2007)
7. Mohamed, K.A., Kupich, C.: An $O(n \log n)$ output-sensitive algorithm to detect and resolve conflicts for 1D range filters in router tables. Technical Report 226, Institut für Informatik, Albert-Ludwigs-Universität Freiburg (August 2006)
8. Kupich, C., Mohamed, K.A.: Conflict detection in internet router tables. Technical Report 225, Institut für Informatik, Albert-Ludwigs-Universität Freiburg (August 2006)