Facilitating Evolutionary Algorithm Analysis with Persistent Data Structures

Erik Pitzer[1](✉) and Michael Affenzeller[1,2]

[1] Department Software Engineering, University of Applied Sciences Upper Austria, Softwarepark 11, 4232 Hagenberg, Austria
{erik.pitzer,michael.affenzeller}@fh-hagenberg.at

[2] Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr 68, 4040 Linz, Austria

Abstract. Evolutionary algorithm analysis is often impeded by the large amounts of intermediate data that is usually discarded and has to be painstakingly reconstructed for real-world large-scale applications. In the recent past persistent data structures have been developed which offer extremely compact storage with acceptable runtime penalties. In this work two promising persistent data structures are explored in the context of evolutionary computation with the hope to open the door to simplified analysis of large-scale evolutionary algorithm runs.

1 Introduction

Evolutionary algorithms especially those that employ a whole population of solution candidates such as e.g. evolution strategy [12] or genetic algorithms [7] are very popular methods for solving complex problems. However, typical optimization scenarios often require long evolutionary processes with thousands or even billions of evaluations. In practice, this means that many methods are developed using rather small sample problems, i.e. to test and tune parameters or select a suitable algorithm variant. Later, the developed technique is applied to a much bigger problem, in the hope that these properties are – at least somewhat – preserved and the algorithm still performs well. For the smaller scenarios, the algorithm runs can be closely supervised and the performance can be tracked and recorded as the amount of data is still manageable. However, as problems approach practical sizes, analysis and continuous tracking involves prohibitively large volumes of data. For example, an algorithm with a population of 100 solution candidates solving a problem with 100 dimensions over one million generations would require storage of 10^{10} elements which are at least 10 to 40 GB depending on the data type for a single run. The idea of this work is to leverage persistent data structures and their nature to reuse old parts when modified, to track historical values without using too much additional space. In essence, we get a data structure that technically only saves differences between different versions but practically provides full copies of all versions with very little overhead.

R. Moreno-Díaz et al. (Eds.): EUROCAST 2017, Part I, LNCS 10671, pp. 416–423, 2018.
https://doi.org/10.1007/978-3-319-74718-7_50

2 Persistent Data Structures

The first prototype of persistent data structures, the persistent linked list, can be dated back to 1955 where it was the primary data structure of the Information Processing Language (IPL) [10] one important predecessor of Lisp [8,9]. It can be used to illustrate the idea of persistent data structures. As shown in Fig. 1, in a persistent data structure, parts of "old" structures can be *reused* inside new structures as the old data and the old structure cannot change. While it is obvious that this works for immutable linked lists it is much harder to imagine for arrays or other commonly-used data structures.

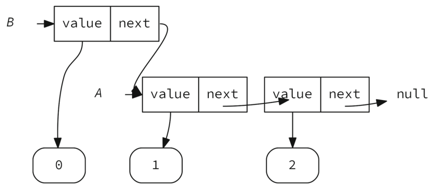


Fig. 1. Example of shared structure in singly-linked list

As computer processors have received fewer improvements in speed but more improvements in multi core execution [13], parallel processing has become not only more prominent but almost a necessity to fully utilize current hardware [1]. This, in turn, has revived interest in functional languages [5] that typically use immutable data structures. In the same vein, many classically imperative data structures have received a functional pendant [11]. To achieve similar run time complexity, however, functional data structures need additional or different tricks that rely mostly on their immutability and reuse of existing parts of the "old" copy of the data structure as immutable data has to be *copied* in case any modification is made. With these tricks, the data does not actually need to be copied but can merely be referenced with great savings in both speed (the original intention) and also space (the intention of this work). In particular, two variants of immutable arrays with structure sharing are explored in this work. These are Array Mapped Tries (AMTs) [2] and Relaxed Radix Balanced Trees (RRB-Trees) [3].

AMTs are an extension of radix trees which in turn are a compact variant of prefix trees [4]. The idea of a radix tree is to identify all elements by index. This index is subsequently split across the levels of a tree depending on the radix or the fixed maximum number of children of each tree as shown in Fig. 2. The benefit of this structure in comparison to e.g. a binary search tree is that no comparisons are necessary to find the correct element, the index of the element is sufficient to completely navigate the tree to any leaf. When this structure is used for hash maps, it can be beneficial to allow any 32 bit integer value; In this case not all slots might be needed. This fact is exploited by Hash Array

Mapped Tries that do not contain all pointers in all nodes, but only those that lead further down to existing leaf nodes [2]. A clever use of the Hamming Weight of the subindex often called population count (`popcount`) operation – available on many processors as a single instruction – makes the lookup of which children actually exist particularly cheap [6].

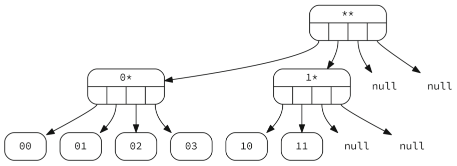


Fig. 2. Example of radix-4 tree

Most importantly however, this structure allows the reuse of parts of the "array" it is representing when making copies or changes. As shown in Fig. 3 changing a single element in this immutable tree does not require copying the whole tree. As all data is also immutable, large parts of the tree can be reused. This is what makes this data structure so interesting for keeping a history. While the modified data structure has the same structure as a "fresh" tree, it can reuse most of the data from its previous version. In essence, we are saving deltas but accessing complete data structures with minimal overhead. As described in [2] updating or accessing an element in this radix tree with typical radix size 32 is in the order of $\log_{32} N$. Moreover as $N \leq 7$ for any index in the range of a 32 bit integer, the overhead can be considered practically constant. This is a great bonus when tracking modifications of large populations over many generations.

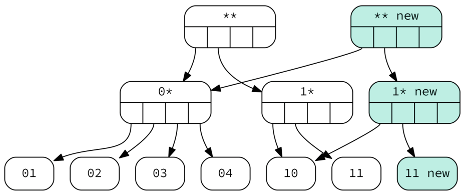


Fig. 3. "Modification" of immutable radix-4 tree

3 Persistent Evolutionary Operations

Every single point modification leads to a short chain of pointers to the changed location, reusing many pointers along the way. Therefore, with constant overhead any single point mutation can be tracked. However, in evolutionary algorithms, frequently a whole slice of an array is updated. An easy fix for this situation would be to discard intermediate trees and only keep the "snapshots" that represent actually-visited solution candidates of the evolutionary search process. However, in addition, many evolutionary algorithms, in particular genetic algorithms, use crossover operations that take information from several individuals and combine them. Using AMTs it would have to be decided which individual is the one being modified with the information of other individuals incorporated as a series of (mostly discarded) single point modifications. This is unfortunate as the overhead of this operation would be linear with the size of the individual arrays. Fortunately, another variation of radix trees, so called, Relaxed Radix Balanced Trees (RRB-Trees) have been proposed in [3]. This variant allows slicing trees and even joining them back together in practically constant time, similar to what AMTs allow for single point modifications.

RRB-Trees enable this modification by slightly relaxing the radix requirement and allowing index shifts on some of the nodes. This enables, for example, the removal of nodes in the beginning without having to rebuild the remaining tree as shown in Fig. 4.

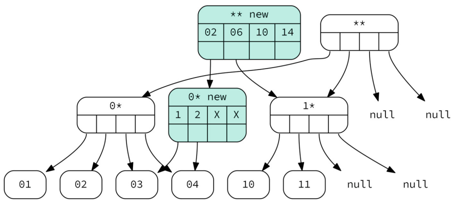


Fig. 4. Removal of the first two elements in an RRB-Tree (Skipping)

The most complex operation is concatenating two trees, but again the overhead is practically constant for a single join operation. For this operation an error bound can be chosen, that allows more or fewer empty slots in the intermediate nodes when two trees are joined. Using practically constant operations for splitting and joining yields practically constant time (and space) for crossover operations as they occur in genetic algorithms [7]. Every *crossover* operation can now be stored as a single "delta" reusing parts from both parents and therefore requiring only constant additional storage for each operation. It has to be noted that the involved computational overhead is significant, however. Array copying is highly optimized and can hardly be outdone by following pointers in a tree even for large trees. One could imagine that copying 1000 values from one array to another must be slower than updating a few pointers on the heap. However, the memory locality and its caching effects give a huge advantage to arrays in terms of speed. The real advantage of trees lies in their space savings. Moreover, in evolutionary algorithms, the internal data handling such as copying and modifying of solution candidates is hardly an issue as it is usually completely dwarfed in comparison to the cost of a solution candidate's *fitness evaluation*.

Figure 5 shows an example of a crossover operation on two RRB-Trees A and B. For the sake of simplicity, this time, only a radix-2 tree is shown. As can be seen, large parts of the original data can be reused. In this – admittedly lucky – example even each leaf node can be reused as-is. In general, only two logarithmic "paths" from the glue points to the root have to be newly generated. All remaining data and data structure parts can always be reused.

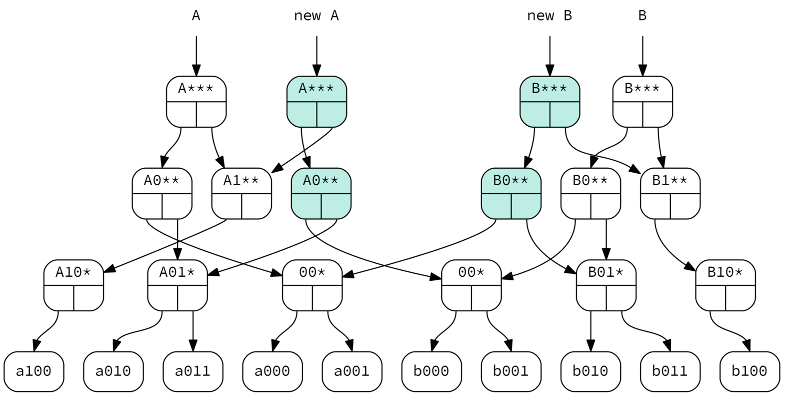


Fig. 5. Crossover of two RRB-Trees

As an extension to this method, other crossover variants can be implemented with reduced memory consumption. Obviously, two-point or multi-point crossover will work the same and other crossover operators that reuse slices of the original solution candidates will also benefit from this scheme, e,g, the edge recombination crossover [15]. Only at the fusion points, new nodes might become necessary to sufficiently satisfy the (relaxed) radix constraints of the tree nodes, depending on the errors parameter E that specifies how many slots may remain empty in any internal node.

Another rather easy extension is to allow (partial) reversal of the represented arrays. This can be achieved by simply replacing a node at a higher level with a special node that marks the reversal of all indexes as shown in Fig. 6, where a continuous array is shown using a reverse node. This node can be used as the intermediate node at the top of a reversed subtree, supporting operations such as partial inversion.

4 Experimental Results

We have implemented both Array Mapped Tries (AMTs) as well as Relaxed Radix Balanced Trees (RRB-Trees) and subjected them to different test scenarios. Our implementation of AMTs has been integrated into the open-source optimization platform HeuristicLab[1] [14]. In HeuristicLab all optimization-relevant

[1] http://dev.heuristiclab.com in the branch `PersistentDataStrcutures`.

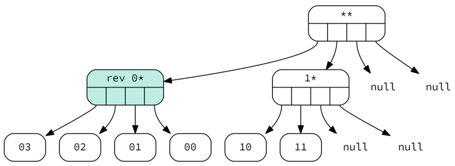


Fig. 6. Inversion-based operations in RRB-Trees

array data structures are derived from the generic class `ValueTypeArray<T>` which – unsurprisingly – uses regular `C#` arrays internally. We have replaced these regular arrays with our implementation `ArrayMappedTrie<T>`, or more precisely with a wrapper called `HistoryArray<T>` that can either automatically make snapshots after a specified number of modifications or with the help of a so-called Analyzer inside HeuristicLab, that is typically called once per generation or iteration in different algorithms. Surprisingly, the run-time of the tested optimization algorithms creating snapshots of all intermediate versions did not increase in comparison to using plain `C#` arrays. It seems that the additional effort required to navigate the tree when setting and getting values is offset by the benefit that the (frequent) cloning that is done in HeuristicLab is a constant operation now.

Table 1a shows the runtime and most importantly the space required to record a series of one million mutations on a vector of size 1000. It has to be noted, that the AMT implementation always uses full depth (7), while the RRB implementation dynamically adapts depth to the actual maximum index. It can be seen, that the trees use significantly more time than the plain array. For small mutations it is beneficial to have little duplication and therefore use a smaller radix (node size). On the other hand a smaller radix has more overhead, i.e. more steps, to reach the leaves. In this scenario, radixes 4, 8 and 16 appear to provide the best compromises between speed and size. In Table 1b the results for crossover are shown. Obviously, using more complicated data structures causes a significant penalty in run time when looking purely at data manipulation operations. In this case, the best tree is slower by a factor of 35, however, using less than a third of the storage. For purely crossover-based workloads, all RRB trees between radix 2 and 16 provide a pareto-optimal compromise between speed and size. Interestingly, in this case the more relaxed trees i.e. error 1 and 2 do not benefit from reduced run-time.

The real benefits of tree-based structures, appear unfortunately only for larger data structures which are less common in practice. In theses cases, as shown in Fig. 7 however, the tries could even outperform plain arrays on mutations and can compete very well on crossovers even in terms of speed.

Finally it has to be noted that in population-based algorithms, where "younger" solution candidates contain much of the genetic material of older ones, obtained by recombination, the ratio of reuse could be even higher as not only two directly related individuals share the same genetic material but many or almost all in later generations.

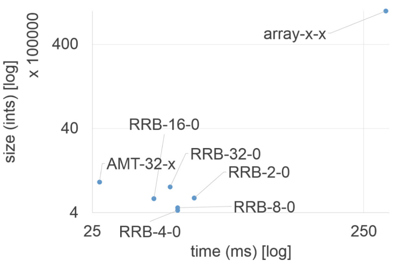
Table 1. One million operations (mutations and crossovers) in a vector of length 1 000: structure parameters are radix and allowed number of skipped pointers, size is the percentage compared to an array and time is the multiple of array processing times.

	no history		with history	
structure	time	size	time	size
AMT/32	38	129%	7.1	54%
RRB/2	72	405%	11.0	28%
RRB/4	42	204%	5.5	**22%**
RRB/8	33	147%	5.3	23%
RRB/16	24	123%	4.3	27%
RRB/32	25	114%	**5.1**	39%

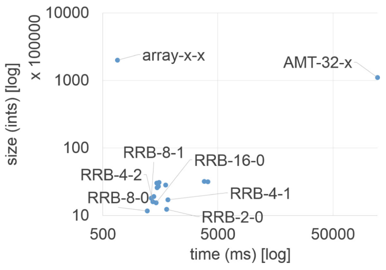
(a) mutations

	no history		with history	
structure	time	size	time	size
AMT/32/-	678	258%	62	92%
RRB/2/0	814	822%	56	**28%**
RRB/4/0	673	418%	46	29%
RRB/4/2	544	740%	37	34%
RRB/8/1	525	340%	36	32%
RRB/16/2	523	268%	**35**	36%
RRB/32/0	572	236%	38	47%

(b) crossovers



(a) mutations



(b) crossovers

Fig. 7. Speed vs. size for 10 000 operations on vectors of length 10 000

5 Conclusions

The recent progress in persistent data structures are a godsend for the analysis of evolutionary algorithms especially the recent addition of split and join operations enable the implementation of space-efficient crossover tracking. Ideally, this kind of information recording can be habitually enabled in the future, as it records changes at a fraction of the cost of previous methods, while incurring acceptable run-time penalties.

We are working to incorporate RRB-Trees into HeuristicLab and to modify its evolutionary operators to be able to take full advantage of the history tracking capabilities of RRB-Trees. This is only an early prototype but we are confident that the simplified tracking will provide great benefits for the postmortem analysis of strangely behaving algorithm runs on large problem instances.

Acknowledgements. The work described in this paper was performed within the COMET Project Heuristic Optimization in Production and Logistics (HOPL), #843532 funded by the Austrian Research Promotion Agency (FFG).

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from Berkeley. Electrical Engineering and Computer Sciences University of California at Berkeley, Technical report UCB/EECS-2006-183, December 2006
2. Bagwell, P.: Ideal hash trees. Technical report, École Polytechnique Fédéerale de Lausanne (2001)
3. Bagwell, P., Rompf, T.: RRB-Trees: efficient immutable vectors. Technical report, École Polytechnique Fédéerale de Lausanne (2011)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
5. Eyler, P.: The rise of functional languages. Linux J. (2007). https://www.linuxjournal.com/node/1000217
6. Warren Jr., H.S.: Hacker's Delight, 2nd edn. Addison-Wesley, Reading (2013)
7. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
8. McCarthy, J.: Lisp prehistory - summer 1956 through summer 1958 (1958). http://www-formal.stanford.edu/jmc/history/lisp/node2.html
9. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Commun. ACM **3**(4), 184–195 (1960)
10. Newell, A., Shaw, J.: Programming the logic theory machine. In: Proceedings of the Western Joint Computer Conference, pp. 230–240 (1957)
11. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1999)
12. Rechenberg, I.: Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Frommann-Holzboog, Stuttgart (1973)
13. Sutter, H.: The free lunch is over: a fundamental turn toward concurrency in software. Dr. Dobbs J. **30**(3), 202–210 (2005)
14. Wagner, S., et al.: Architecture and design of the heuristiclab optimization environment. In: Klempous, R., Nikodem, J., Jacak, W., Chaczko, Z. (eds.) Advanced Methods and Applications in Computational Intelligence. Topics in Intelligent Engineering and Informatics, vol. 6, pp. 197–261. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-319-01436-4_10
15. Whitley, D., Starkweather, T., Fuquay, D.: Scheduling problems and traveling salesman: the genetic edge recombination operator. In: International Conference on Genetic Algorithms, pp. 133–140 (1989)