Data Structures for Non-Dominated Sets: Implementations and Empirical Assessment of Two Decades of Advances

Jonathan. E. Fieldsend
J.E.Fieldsend@exeter.ac.uk
University of Exeter
Exeter, EX4 4QF, UK

ABSTRACT

Many data structures have been developed over the last two decades for the storage and efficient update of unconstrained sets of mutually non-dominating solutions. Typically, analysis has been provided in the original works for these data structures in terms of worst/average case complexity performance. Often, however, other aspects such as rebalancing costs of underlying data structures, cache sizes, etc., can also significantly affect behaviour. Empirical performance comparison has often (but not always) been limited to run-time comparison with a basic linear list. No comprehensive comparison between the different specialised data structures proposed in the last two decades has thus far been undertaken. We take significant strides in addressing this here. Eight data structures from the literature are implemented within the same overarching open source Java framework. We additionally highlight and rectify some errors in published work — and offer additional efficiency gains. Run-time performances are compared and contrasted, using data sequences embodying a number of different characteristics. We show that in different scenarios different data structures are preferable, and that those with the lowest big O complexity are not always the best performing. We also find that performance profiles can vary drastically with computational architecture, in a non-linear fashion.

CCS CONCEPTS

• **Theory of computation → Computational geometry**; • **Information systems → Data structures**; • **Applied computing → Multi-criterion optimization and decision-making**; • **General and reference → Estimation**; **Performance**; • **Mathematics of computing → Evolutionary algorithms**.

KEYWORDS

Data structures, real-time statistics, real-time analysis, computational efficiency

ACM Reference Format:
Jonathan. E. Fieldsend. 2020. Data Structures for Non-Dominated Sets: Implementations and Empirical Assessment of Two Decades of Advances. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377930.3390150

GECCO '20, July 8–12, 2020, Cancún, Mexico

1 INTRODUCTION

Building on initial work by researchers from the Multi-Criterion Decision Making (MCDM) community, numerous data structures have been developed by Evolutionary Multi-criterion Optimisation (EMO) researchers over the last two decades specifically for the storage and efficient update of sets of mutually non-dominating solutions. These sets of solutions, which approximate the Pareto set of designs for a problem, are used to maintain designs during during an optimisation run, and can be queried for e.g. online quality statistics. Data structures proposed over the last two decades include:

(1) Linear Lists (effectively since EMO commenced).
(2) Dominated and Non-Dominated Trees (2002-2003) [9, 11].
(3) QuadTrees 1-3 (2002-2003) [19, 20].
(4) Dominance Decision Trees (2003) [21].
(5) Bi-objective Trees (Mak_Trees) (2006) [2][1].
(6) M-front (2015) [7].
(7) BSPTree (2017) [13].
(8) ND-Tree (2018) [16].

In 2012 there was an entire review paper dedicated to describing work on data structures in this domain [1]. Work on related problems of non-dominated sorting have also mentioned data structures for archive maintenance, for instance [17] discusses the computational complexity of a solution based on Dynamic Range Trees (but does not implement them). It is worth emphasising at this point we are only concerned here with those data structures specifically developed for non-dominated sets, and do not consider those developed in the EMO community to store sets with *general* dominance relationships between solutions (e.g. [4, 10, 22]). We also do not consider data structures from the literature which can store non-dominated sets, but which don't have methods described to determine using the structure if a new solution is non-dominated with members and/or automatically remove dominated members (e.g. [25]).

Although there is a substantial body of work in this area now spanning two decades, nearly all published comparisons *between* these data structures that we could find have been limited to big O complexity analysis. Empirical assessment of each newly proposed data structure is typically limited to the linear list baseline (bar [16], which is more extensive, if not exhaustive). This is outlined in Table 1, which shows which data structures have been empirically compared to which others in the corresponding works from the literature.

Often realised performance of a data structure is significantly affected by rebalancing costs, and how regularly these occur (which

[1] Limited to two-objectives.

Table 1: Empirical comparisons in previous work, chronologically ordered. A cross indicates that the algorithm in the row was not compared empirically to the algorithm in the column in the paper(s) referenced in the row. A tick indicates that the algorithm in the row was compared empirically to the algorithm in the column in the published work. A hyphen indicates the algorithm in the row was published after (or in parallel to) the algorithm in the column.

	LL	D&NDT	QT1	QT2	QT3	DDT	BBT	M-Front	BSPT	NDT
Dominated and non-dominated trees (D&NDT), 2002 & 2003 [9, 11]	✓	-	-	-	-	-	-	-	-	-
Quad-Tree 1 (QT1), 2002 & 2003 [19, 20]	✓	-	-	✓	✓	-	-	-	-	-
Quad-Tree 2 (QT2), 2002 & 2003 [19, 20]	✓	-	✓	-	✓	-	-	-	-	-
Quad-Tree 3 (QT3), 2002 & 2003 [19, 20]	✓	-	✓	✓	-	-	-	-	-	-
Dominance Decision Trees (DDT), 2003 [21]	✓	×	×	×	×	-	-	-	-	-
($m = 2$) Balanced Binary Trees (BBT), 2006 [2]	✓	×	×	×	×	×	-	-	-	-
M-Front (MFront), 2015 [7]	✓	×	×	×	×	×	×	-	-	-
BSPTree (BSPT), 2017 [13]	✓	×	×	×	×	×	×	×	-	-
ND-Tree (NDT), 2018 [16]	✓	×	×	✓	×	×	✓	✓	×	-
This work, 2020	✓	×	✓	✓	✓	✓	✓	×	✓	✓

are amortised in big O analysis), along with machine-specific limitations (cache sizes, memory speeds, etc.). As such, a rigorous empirical comparison of these competing data structures covering some of these aspects would undoubtedly be useful. Furthermore, an easily usable framework in which all are implemented, where a data structure may be selected that is best suited to the combination of problem and the particular computational architecture at hand, and which may readily extended with implementations of new approaches, is also lacking. We confront these points in this work.

Although the BSPTree of [13] is integrated into the black box optimization competition (BBComp) framework[2], none of the advanced data structures appear to be integrated into the widely used general purpose packages for multi-objective optimisation, e.g. jMETAL [8][3], MOEA [14][4] and DEAP [12][5], which instead all rely on potentially costly sequential list processing for non-dominated set maintenance. Recent work on EMO comparison has identified that contrasting optimiser algorithm performance in terms the non-dominated set of *all* solutions visited during a search can lead to different conclusions than comparing the final populations [23], and is a better reflection of an optimiser's performance. Such comparison necessitates the storing of a potentially large passive archive, which these advanced data structures are much better suited to (especially if an anytime performance analysis is required). Furthermore, as such data structures are independent of the search algorithm — their inclusion in these optimisation frameworks has the potential to reduce the computational cost across optimiser types. The main contributions of this work are:

- Alongside high-level descriptions of published data structures, we highlight errata in these work. These errata have been reproduced in later papers — e.g. [1], and tend not to be obvious until implementation.
- We highlight efficiency improvements for some data structures.
- We provide an empirical comparison between data structures on a range of scenarios. This is the broadest such comparison conducted thus far in the literature, with nearly twice as many data structures compared than in any other published work.
- We provide reference implementations for eight data structures in a single extendable framework — providing a base for the community to work in this area, and which is open to integration into optimisation platforms and researchers' optimisation pipelines.

Our Java package also provides a platform for researchers in the future to develop their own data structures for this task, and which will be easily comparable to the implementations we provide.

The rest of the paper proceeds as follows. In Section 2 the Pareto archive updating problem is formally described, and high-level descriptions of a range of data structures developed in the EMO community for this task is presented. Due to space limitations these are necessarily short and omit a considerable number of technical details — however we crucially highlight where there are errata we have identified in these works, which should aid other researchers wishing to implement them in alternative languages. For more technical details, please refer to the original cited works. In Section 3 we detail an empirical comparison of these implementations on a range of problems. The paper concludes in Section 4 with a discussion and highlights of future work directions in this area.

2 DATA STRUCTURES FOR PARETO ARCHIVE UPDATING

Given a feasible search space $\mathcal{X}$, a design $\mathbf{x}$ from this space is said to *dominate* another design $\mathbf{x}'$, denoted $\mathbf{x} \prec \mathbf{x}'$, if it is no worse on all assessment criteria, $f_i(\mathbf{x})$, and better on at least one. $\mathbf{x}$ is said to *weakly dominate* another design $\mathbf{x}'$, denoted $\mathbf{x} \preceq \mathbf{x}'$, if it is no worse on all assessment criteria. The set of Pareto optimal solutions (the *Pareto set*) is defined as $\mathcal{P} = \{\mathbf{x} \in \mathcal{X} \,|\, \nexists\, \mathbf{x}' \prec \mathbf{x}, \mathbf{x}' \in \mathcal{X}\}$. As

[2] https://bbcomp.ini.rub.de

[3] Version 5.6, https://github.com/jMetal/jMetal uses `ArrayLists` in the algorithm archive implementations.

[4] The `NondominatedPopulation` class in release 2.13, http://moeaframework.org, uses an `ArrayList`.

[5] `deap.tools.ParetoFront` in version 1.2.2 uses a linear list https://github.com/DEAP/deap/blob/master/deap/tools/support.py

Algorithm 1 Updating non-dominated archive A with $\mathbf{x}$

Require: A ▷ Current non-dominated set of solutions
Require: $\mathbf{x}$ ▷ New solution to check against
1: **if** $\nexists \mathbf{x}' \in A \mid \mathbf{x}' \preceq \mathbf{x}$ **then** ▷ If $\mathbf{x}$ not dominated
2: $A := A \setminus \{\mathbf{x}' \in A \mid \mathbf{x} \preceq \mathbf{x}'\}$ ▷ Remove dominated members
3: $A := A \cup \{\mathbf{x}\}$ ▷ Add $\mathbf{x}$ to non-dominated set
4: **return** A

a multi-objective optimiser explores $\mathcal{X}$ it commonly maintains an approximation to $\mathcal{P}$, called its Pareto archive. As described in [21] as the dynamic non-dominance problem, the task of maintaining and updating an unconstrained Pareto archive is summarised in Algorithm 1.

Early work in the EMO field identified the various issues caused by truncating an approximation archive [11, 15], and more recent research has highlighted the change in relative performance of algorithms that is observed when unconstrained approximation sets are tracked [23]. It is best practice to at least keep a passive unconstrained archive [24] — as from a practical point of view even if a restricted set will be considered for presentation to the problem owner, these solutions should still be selected from the best approximation discovered to the Pareto set.

We now give a high-level overview of the various data structures employed and developed in the last two decades for the task described in Algorithm 1. Although we eschew deep algorithmic details here, we highlight any errata in the original works that may cause issues when undertaking an implementation, which unavoidably includes some technical details. We also highlight any efficiency gains that we have noted may be achieved beyond those described in the original works.

2.1 Linear list

The linear list (LL) is a basic data structure, and the most widely used in the EMO community due to its ease of implementation.

The idea: Non-dominated solutions are stored in a linear list, arbitrarily ordered, and new solutions are compared sequentially to the list members. A new solution $\mathbf{x}$ is compared until a current list member is found which weakly dominates it (i.e. the 'if' check in Algorithm 1 returns false) and A is returned unchanged. If no member dominates $\mathbf{x}$ the new solution must be non-dominated. Prior to its insertion the list is again processed sequentially for any dominated solutions to be checked and marked for removal, with the new solution added at the end. Insertion and deletion are $O(n)$ domination comparisons — where n is the length of the list — i.e. the number of elements of the current approximation to the Pareto front. This is commonly stored in an array.

2.2 Dominated and Non-dominated Trees

The idea: In the Dominated and Non-dominated Trees (D&NDT) dual data structures of Everson et al. [9] and Fieldsend et al. [11] each node in a tree is a *composite point* defined by m solutions, each solution providing one of the m objectives represented by the composite point. The main two trees are effectively a chain of such composite points, ordered by the weak dominance relation, with the minimal chain length therefore being $\lceil n/m \rceil$. The two trees are constructed and maintained such that, when a new solution $\mathbf{x}$ is sequentially compared to the non-dominated tree from its head, when a node is reached which does not weakly dominate $\mathbf{x}$, then no solutions contributing to the composite points of all subsequent nodes can possibly dominated $\mathbf{x}$. Similarly, when processing from the tail of the dominated tree, when a node is reached that is not weakly dominated by $\mathbf{x}$, then no solutions contributing to nodes further toward the head can possibly be dominated by $\mathbf{x}$. The process of insertion/removal means the trees may grow significantly beyond the $\lceil n/m \rceil$ minimum length, and therefore are periodically reconstructed. This is enabled by also maintaining m sorted-lists of the n solutions — each list ordered by a different objective.

Efficiency improvements: In the work described in [11] solutions may appear in more than one composite point when the trees grow through insertion and copying a composite point's contents lower down. This can mean the same solution is queried multiple times when these trees are traversed at later time points. A more efficient implementation would be still to employ the objective value required at that dimension in a composite point, but have the reference solution `null` in all but the lowest composite point, and therefore skipped over for any subsequent domination comparisons.

2.3 Quad-trees

Mostaghim et al. [20] and Mostaghim and Teich [19] introduce three quad-tree based approaches, named Quad-Tree1, Quad-Tree2 and Quad-Tree3 (QT1–3). A quad-tree [5] is a tree-based structure each of whose nodes store an item whose quality is defined by vectors of elements.

The idea: Each node in a quad tree has up to 2^m children. These children are labelled by the unique bit-strings of length m, where a 0 at a particular index indicates the child holds a *better* value at that index than the parent. When storing mutually non-dominated solutions there can be at most $2^m - 2$ children, as the child with bit-string of all zeroes will dominate its parent, and the child with bit-string of all ones will by dominated by its parent. There are some necessary definitions to understand the properties of a quad-tree.

k-successor: A node $\mathbf{u}$ is a k-successor of a root node $\mathbf{v}$ where

$$k = \sum_{i=1}^{m} I(u_i \geq v_i) 2^{m-i} \tag{1}$$

where $I()$ is the Kronecker delta or identity function. k therefore handily serves as an index into the children of a node.

k-child: $\mathbf{u}$ is k-child of $\mathbf{v}$ if it is a k-successor of $\mathbf{v}$ and stored in a direct child node.

k-set: A k-set is a set of all possible indices derived from (1), who match a mask derived from the binary successor vector used to generate k. The $S_0(k)$ set holds all those indices who originating successor vectors have zeros at least in all the same locations as k. The $S_1(k)$ set holds all those indices who originating successor vectors have ones at least in all the same locations as k.

Errata: There are a few errors in the algorithms presented in [19], which we list below.

- `Quad-Tree1` algorithm, error in Step 2: "If $k = 2^m$ or if $x_i = y_i, \forall i \in S_0(k)$ STOP, $\mathbf{x}$ is dominated by $\mathbf{y}$" — in the first condition an index of 2^m is not possible, as the algorithms are defined with 0 as the first index rather than 1, so $2^m - 1$ is actually meant (this error is also repeated in the text below (5) in the work). In the second condition $\mathbf{x}$ is not dominated by $\mathbf{y}$, the reverse is true (as $\mathbf{x}$ is equal on all criteria that contribute to the k value, and better on the others), and needs to be compared on $i \in S_1(k)$.
- In `TEST2` of `Quad-Tree1`, before attempting to reinsert subtree of dominating node, all its elements need to be compared against the newly proposed point for domination, otherwise dominated elements may be reinserted (as the point which dominated the subtree root is not inserted until algorithm step 5).
- In `Quad-Tree2`, the else if Flag = 1 check in Step 2 – here flag should always be 1 as root of tree is dominated in Step 2 if k = 0 – a non-dominated z must always be marked to be reinserted as its relationship (successor index) may not match that of the original (now -dominated) root, which is being replaced.
- In Step 2 of `Quad-Tree2`, if a node has been identified as dominated, *all* successors need to be checked subsequently, not just the l-children.
- In Step 2 of `Quad-Tree2` not all flagged nodes should be inserted from x, as some may have been flagged from higher nodes in the tree being checked/of on different paths (as step 4 can loop back to step 2) and these ones should be inserted from the root of the tree.
- The statement regarding `DELETE` in `Quad-Tree3` and the `REINSERT` guarantee of non-dominated is false.

2.4 Dominance Decision Tree

The idea: In the Dominance Decision Tree (DDT) of Schütze [21] each node contains a design, and has up to m children. The ith child of a parent is always better or equal on the $1, \ldots, i-1$th objectives, and worse on the $i, \ldots, m$th objectives. New solutions query down paths where they are better on the corresponding objective to ascertain if they are dominated. Similarly the structure property is used to guide search down branches where dominated solutions may be stored.

Errata: in [21].

- The call to `TreeInsert` in `DeleteDominated` is given one argument, however the `TreeInsert` routine is defined for two arguments.

2.5 Bi-objective case: balanced binary tree/Mak_Tree

The idea: In the case where there are only two objectives the observation that sorting the set in terms of one criterion is equivalent to reverse sorting by the other my be exploited to obtain performance levels not achievable with $m > 2$. By storing in a balanced binary tree (BBT), e.g. a red-black tree, the check for insertion is $O(1 + \log n)$ objective comparisons, and each deletion is $O(1 + \log n)$. This is a special case, but given the wide range real-world bi-objective problems encountered in practice, it should not be ignored. It was highlighted by Berry and Vamplew [2] in developing their Mak_Tree data structure.

2.6 M-front

The idea: Like the data structure in [11], the M-fronts of Drozdik et al. [7] utilise m sorted lists of solutions on each objective. These are employed in the M-fronts structure to answer interval queries, which correspond to range queries converted based on a nearest neighbour search. To conduct approximate nearest neighbour search a K-d tree is also employed (see, e.g. [5]). Reference points enable the identification of sub-ranges of each sorted list that need to be enumerated.

Errata: There is a small error in [7]:

- The `Insert` algorithm (on page 666 of [7]) uses dominance rather than weak-dominance checks, meaning duplicates can be inserted (weak-dominance should be used).

2.7 Binary Space Partitioning Tree

The idea: In the Binary Space Partitioning Tree (BSPT) approach developed by Galsmachers [13], each leaf holds a list of solutions (the maximum list size being a user-defined argument). The interior nodes hold a threshold (θ) and an objective index i. Solutions that reach a node are compared to this threshold value on objective i, and pass left if better and right otherwise. This process continues until a leaf is reached for comparison, or until it can be identified as dominating or dominated before reaching a leaf (by tracking the performance against thresholds for each objective). As an approximation set's front moves forwards in objective space over time the tree can get quite imbalanced. Interior nodes keep count of the total number of solutions covered under their left and right branches. When a user-defined factor is exceeded the more densely populated subtree replaces the node, and the less densely populated branch contents are reinserted into the tree.

Efficiency improvements: [13] contains relatively few details on the rebalancing process, apart from at a high level. When reinserting the solutions in a subtree to be rebalanced, this may be most efficiently achieved if it is started from the node that was moved up to replace its parent, rather than inserting from the root (as the reinserted subtree members will always pass through this node on insertion from the root).

2.8 Non Dominance Tree

Each node in the Non Dominance Tree (NDT) of Jaszkiewcz and Lust [16] represents a subset of the non-dominated front which lies within a hyperrectangle defined by the subset's approximate nadir and ideal point. These nadir and ideal locations are used to identify whether new solutions need be compared to any of the designs covered by the node. A new solution $\mathbf{x}$ dominated by the nadir point will be dominated by all members represented by a node. Conversely, if $\mathbf{x}$ dominates the ideal point it will dominate all members represented by the node. If $\mathbf{x}$ is mutually non-dominating with respect to both the ideal and the nadir points, it is also mutually non-dominating with respect to all solutions covered by the corresponding node. Traversing down the tree the volumes covered by the internal nodes decrease, and the corresponding ideal and nadir locations shift, until a leaf is reach containing a list of solutions residing in the objective space volume defined by the corresponding hyperrectangle.

Table 2: Parameterised data structures.

	Parameter list
D&NDT	(i) threshold-factor for rebalancing.
M-front	(i) α threshold for K-d imbalance and rebuilding.
BSPT	(i) max list size at leaf. (ii) threshold-factor for rebalancing.
NDT	(i) max list size per node.

2.9 Implementations

We have implemented eight of the ten extant data structures so far in our Java package: LL, QT1, QT2, QT3, DDT, BBT, BSPT and NDT. The open source package, along with unit tests and code to reproduce the empirical work below is available at https://github.com/fieldsend/multiobjective_data_structures.

In the empirical section below we employ the default data structure parameters defined in the original works (the parameters required for each data structure are given in Table 2).

3 EMPIRICAL RESULTS

We conduct our empirical work on two machines: these are a high-performance Linux server, and a laptop running O/S X. This affords us a preliminary insight into how the performance of the data structures vary with architecture.[6] The machine specifications are:

Server 16-core 2.10GHz CPU. L1d cache: 32KB, L1i cache: 32KB, L2 cache: 256KB, L3 cache: 20MB. 128GB RAM @ 2400MHz.

Laptop Dual-core 2.66 GHz CPU. L1d cache: 32KB, L1i cache: 32KB, L2 cache: 3MB, 8GB Ram @ 1067MHz.

3.1 Simulation runs

In our first set of experiments we use just the server machine, and employ the protocol used in [13] for generation of objective vectors from controlled analytical distributions, which removes the stochastic element of the optimiser from the results. Archives are constructed from a sequence of N normally distributed objective vectors. N_d of these are dominated, and N_{nd} are non-dominated ($N = N_d + N_{nd}$). The tth objective vector $\mathbf{y}^t$ is drawn from:

$$\mathbf{y}^t \sim \mathcal{N}\left(\frac{d^t N}{t}\mathbf{1}, \mathbb{I} - \frac{1}{m}\mathbf{1}\mathbf{1}^\top\right) \tag{2}$$

where $\mathbb{I} \in \mathbb{R}^{m\times m}$ is the identity matrix and $\mathbf{1} = (1, 1, \ldots, 1)^\top \in \mathbb{R}^m$ is the vector of all ones. The variable d^t controls the systematic improvement of points. It is assigned a value of 0 with a probability of $cN_d'^t/(N-k)$, where $N_d'^t$ is the number of dominated points still to draw in the sequence, otherwise d^t is assigned a value > 0. For $c > 1$ there is a preference for seeing more dominated points earlier in the sequence, and for $c < 1$ there is a preference for seeing more dominated points later in the sequence.

We investigate $c = \{0.9, 1.1\}$ here. [13] used $c = \{1, 1.1\}$, but we are also interested in the situation where an archive rapidly moves forward, then slows down (lots of non-dominated points earlier in a sequence, then proportionally more dominated points later). The non-zero value employed for d is omitted from [13]. We use $1/N$ here. We measure the CPU time dedicated to the execution thread when interacting with the data structure, but exclude all other time costs (e.g., the cost of sampling from the analytical distribution). The panels in Figure 1 show the run-time characteristics of the different data structures for $c = 0.9$. When $m = 2$, and when the number of dominated solutions is relatively small (i.e. 2^{10} and 2^{14}) the three QT data structures perform the best, even better than the theoretical optimum structure, the BBT. However, when the number of dominated points in the sequence is large, the QT variants are the worse performing (orders of magnitude so). For $m = 3$ and $m = 5$ NDT is the best performing by the end of the sequence, though BSPT is better for the two larger N_d sizes earlier in the sequence. For $m = 10$, NDT, DT variants and BSPT are all performing similarly by the end.

The panels in Figure 2 show the run-time characteristics of the different data structures for $c = 1.1$. These are largely similar to those for $c = 0.9$, though for $m = 2$ and N_d^{18} the run-time costs of the worse performing data structures are even poorer than their comparable performance at $c = 0.9$.

3.2 Optimiser runs

To illustrate the performance of the data structures in an optimisation environment, we employ a simple (1+1)–Evolution Strategy (ES). These results are therefore for an algorithm which proposes and evaluates only a single new design at each iteration. These results are however generalisable to population-based approaches — which would likewise be processed in a sequence on single-thread data-structure. The optimiser is based on the PAES algorithm of [18], but rather than using a gridded constrained archive, an unconstrained archive is used, which is stored in one of the data structures. The parent has a single design variable mutated with Gaussian noise, with width 0.1. Rejection sampling is used on mutations which lead to boundary violations. If the child is not weakly dominated by A, the child replaces the parent at the next generation. In these experiments we run the ES with each data structure on a problem 10 times, plotting the average.

We use the test problems DTLZ1 and DTLZ2 from [6] here, as they exemplify two distinct profiles of convergence. In DTIZ1, the objective values of random design vectors are many orders of magnitude worse than those of the Pareto set. Additionally, as there are many deceptive fronts in this problem, the approximation set tends to converge, expand, and then rapidly contract once a better local front is identified — repeatedly. In DTLZ2 random solutions are only a couple of times worse than Pareto optimal ones on the quality criteria. Additionally, there is a single multi-objective basin of attraction in the problem, so the approximation set tends to steadily grow over time, rather than violently changing size. In both cases we set the number of design parameters as $m - 1 + 9$.

Figure 3 shows how the data structures compare for the various number of objectives on both server and laptop machine on DTLZ1 — with significant outperformance highlighted on the abscissa. For 2–5 objectives the average time cost is seen to fall for all approaches initially, before rising again later in the optimisation run (except for NDT and BBT in $m = 2$, where it is still decreasing at generation 200 000). For $m = 10$ there is a much shorter period of decrease

[6] The Java run-time environment was initialised with the -Xss4M flag to ensure the stack was large enough for the recursion depth experienced by the quad tree data structures, as the default -Xss512k resulted in stack overflow for some problem instances.

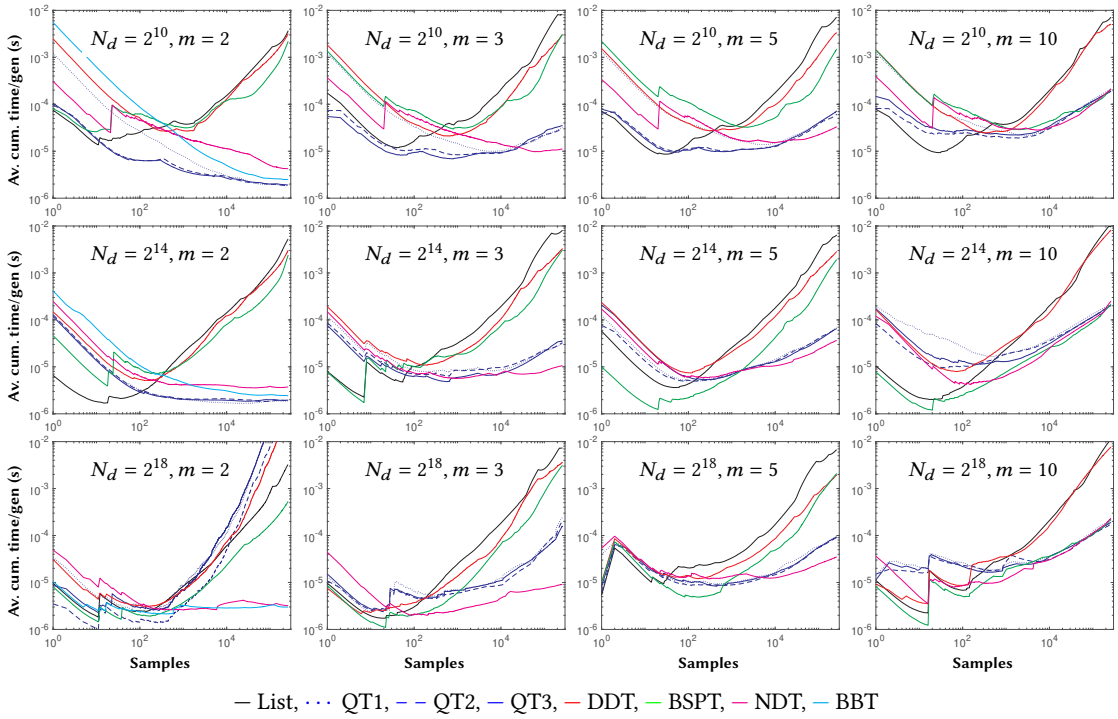

Figure 1: Mean time taken to update data structure per sample. Results on analytical function, $c = 0.9$.

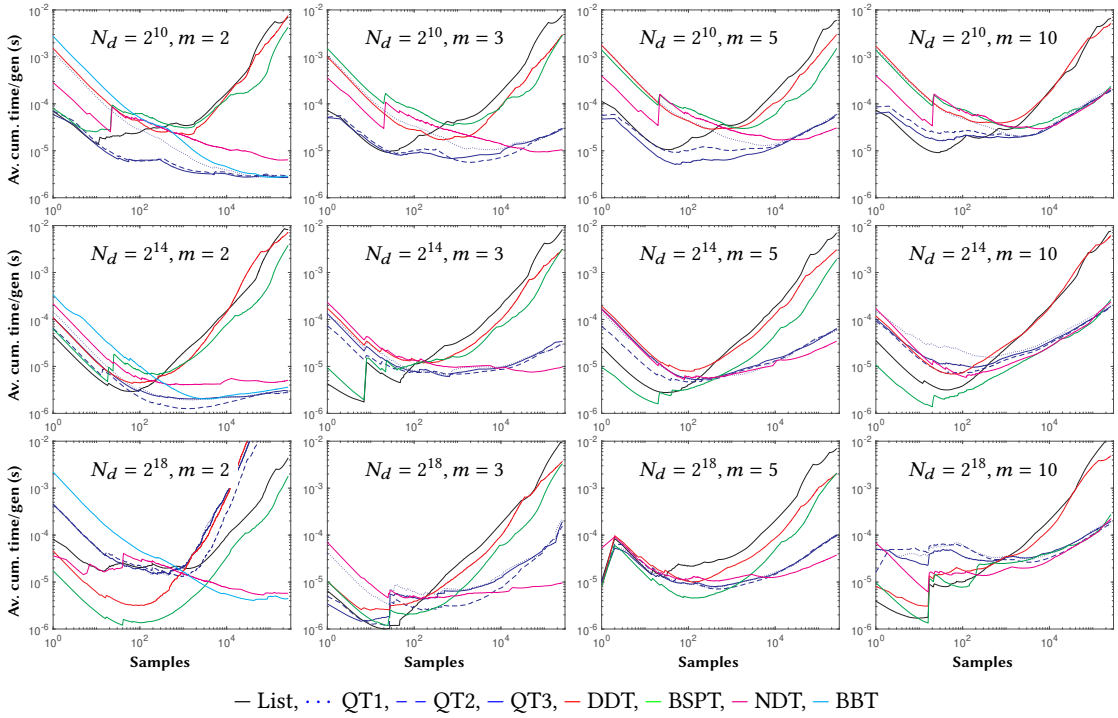

Figure 2: Mean time taken to update data structure per sample. Results on analytical function, $c = 1.1$.

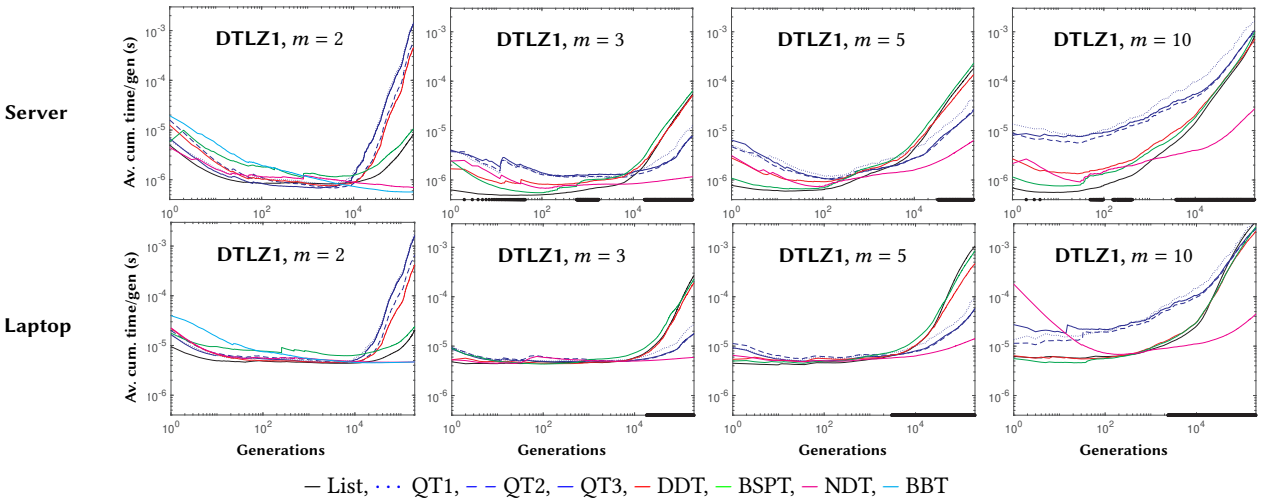

Figure 3: Mean time taken to update data structure up to marked (1+1)–ES generation. Averaged over 10 repetitions (paired). DTLZ1. We highlight on the abscissa when the data structure with the lowest average is significantly better than all others (all paired runs having lower average at the corresponding generation than all other competing data structures) *Top:* experiments run on server. *Bottom:* identical experiments run on laptop.

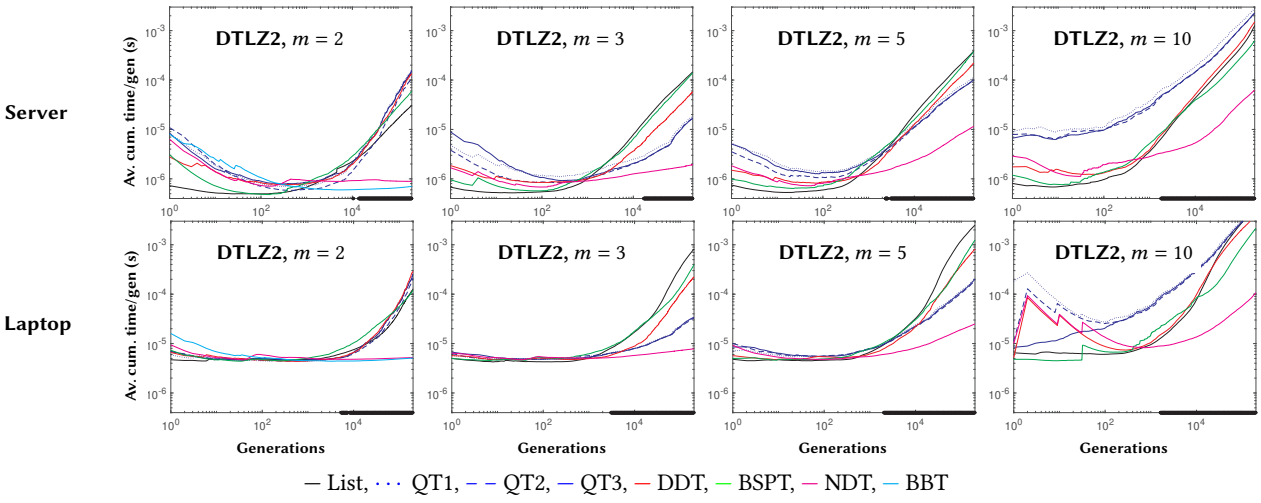

Figure 4: DTLZ2 data structure timing results. Details as in caption of Figure 3.

in average time cost before the uptick for the majority of data structures. This is likely to be due to the initial building cost at the start as the data structures gain depth, before a period of churn as the fronts move forward, the large uptick at the end corresponding to the approximation set expansion at it converges in the region of the Pareto front. As initially expected, BBT outperforms the other approaches by the end of the $m = 2$ runs, however interestingly on both architectures it is performing worse than the other approaches until around 10 000 generations. Excluding the special case of BBT, NDT is better than all other approaches from 10 000 generations onwards – though in earlier generations it can be relatively more costly. The ranking of other data structures varies with number of objectives *and* with platform architecture.

Figure 4 shows how the data structures compare for the various number of objective selected on DTLZ2. As with DTLZ1, the average time cost is seen to fall and then rise as the number of generations increases. The trend of BBT ($m = 2$) is again similar on a gross level to DTLZ1. Again, excluding the special case of BBT, NDT is better than all other approaches from 10 000 generations onwards, and on some occasions as early as 1 000 generations. As before though, in earlier generations it is relatively more costly than e.g. BSPT and LL. Again, the ranking of other data structures varies with number of objectives *and* with platform architecture.

Figure 5 underlines the effect of system architecture on relative data structure performance, and illustrates why this may lead to contradictory empirical results being published in the literature

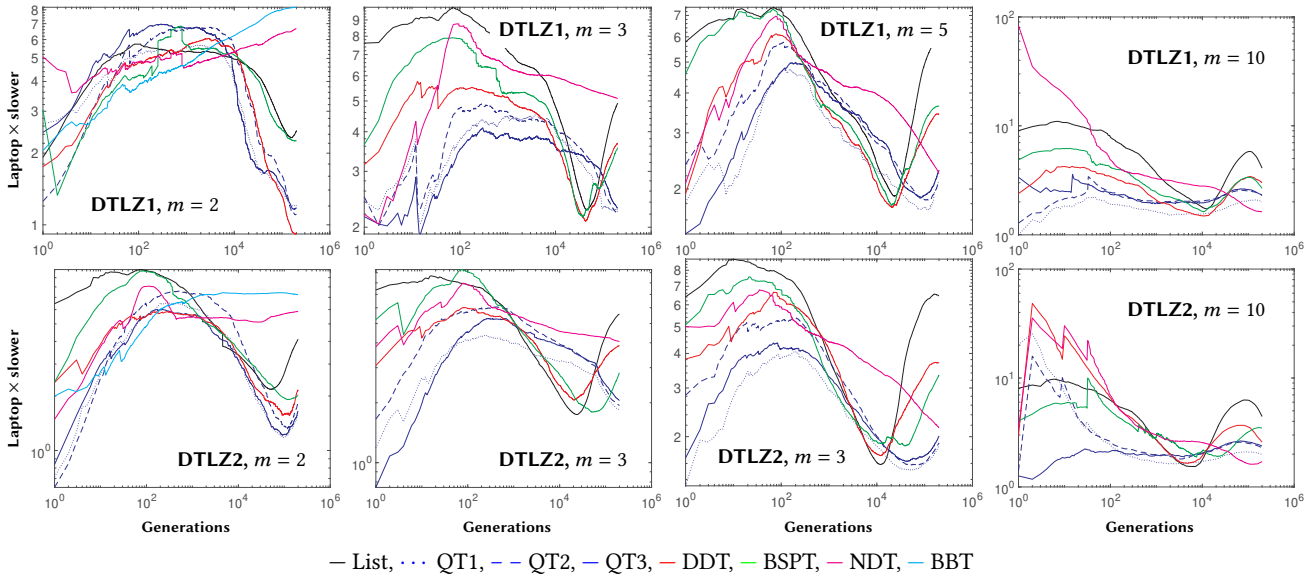

Figure 5: Mean time taken to update data structure up to marked (1+1)–ES generation on laptop divided by the corresponding time to generate the server results (i.e. the multiple the laptop is slower than server for this cumulative task).

when comparing data structures. Each panel shows the average CPU time expended by a data structure on the laptop machine, divided by the corresponding time on the server. As the draws are paired with the same random seed across the two platforms, the same 10 sequences of solutions are being summarised in the averages in Figures 3 and 4, so we might expect the plots in Figure 5 to consist of constant horizontal lines corresponding to a constant performance factor the server provides. However, it is clear that the complex interaction of processor speed, memory speed, relative cache sizes, and run-time environment on the particular operating system, along with the state of particular data structures at different generations, means that there can be a large variation in relative timings between the machines on different problems, and *between* different data structures on the same problem. For instance, on DTLZ2, $m = 3$, QT1 is actually *quicker* on the laptop to begin with (driven by the slightly faster processor on the laptop), but ends up being twice as slow by generation 200 000 (memory speed, cache sizes, etc., having a larger effect). On the other hand, on DTLZ1, $m = 2$, NDT starts twice as slow on the laptop, and by generation 200 000 is eight times slower. In many other instances there is even more complex behaviour, with striking changes in direction of the ratio curve and magnitudes varying back and forth over time.

4 DISCUSSION AND FUTURE WORK

We find that the most recently proposed data structure, NDT, generally performs best by the end of a sequence — but this is not a universal finding. Furthermore we find that data structure performance can vary considerably between architectures, and in a non-constant fashion. We have not tuned the data structures which are parameterised — using the author-recommended values. All these factors mean that there is a potential to *optimise*, via both selection and tuning, the data structure to the task and computational architecture. This is excessive for short runs and small problems – however for tasks with relatively cheap to evaluate objective functions and long optimisation run (like, for instance, many practical network optimisation tasks), or for batches of algorithm comparisons on test functions, significant computational cost reductions are possible.

Beyond this work, there are two remaining data structures to be implemented in the package: those described in [11] and [7] (which are composites of a number of different data structures). We look forward to implementing these also.

Very little has been discussed in the literature regarding parallelisation capabilities of these various data structures. [7] provides a small section on this, and the authors note that they are not sure if a significant speedup is possible for M-Fronts, but that other data structures may be more amenable to parallelisation. Certainly with the prevalence of multi-core computational resources in modern computing infrastructure, the performance of parallelised versions of these structures is an area we are keen to explore.

In this study we recognise a number of different factors interplay in the relative performance of the data structures on the two example architectures. It would be useful to be able to isolate and identify which particular component(s) are having different effects — this will require access to easily re-configurable hardware, and close interaction with researchers in the high-performance computing and profiling community, where work on performance counter measurements looks to more precisely characterise performance dependencies. Preliminary work in this area may be found in [3].

5 ACKNOWLEDGEMENTS

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/N017846/1] and Innovate UK [grant number 104400].

REFERENCES

[1] Najwa Altwaijry and Mohamed E. B. Menai. 2012. Data Structures in Multi-Objective Evolutionary Algorithms. *Journal of Computer Science and Technology* 27, 6 (2012), 1197–1210.

[2] Adam Berry and Peter Vamplew. 2006. An Efficient Approach to Unbounded Bi-objective Archives -: Introducing the Mak_Tree Algorithm. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*. Association for Computing Machinery, New York, NY, USA, 619–626. https://doi.org/10.1145/1143997.1144110

[3] Silvia I. Candia. 2020. *Performance Counter Measurements of Data Structures: Implementations for Multi-Objective Optimisation.* Master's thesis. Department of Computer Science, University of Exeter, Exeter, UK.

[4] Xianming Chen. 2002. *Pareto Tree Searching Genetic Algorithm: Approaching Pareto Optimal Front by Searching Pareto Optimal Tree.* Technical Report NK-CS-2001-002. Department of Computer Science, Nankai University.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 2000. *Computational Geometry: Algorithms and Applications* (second ed.). Springer-Verlag. 367 pages. http://www.cs.uu.nl/geobook/

[6] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. 2001. *Scalable Test Problems for Evolutionary Multi-Objective Optimization.* Technical Report 112. Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland.

[7] Martin Drozdík, Youhei Akimoto, Hernán Aguirre, and Kiyoshi Tanaka. 2015. Computational Cost Reduction of Nondominated Sorting Using the M-Front. *IEEE Transactions on Evolutionary Computation* 19, 5 (2015), 659–678.

[8] Juan J Durillo and Antonio J Nebro. 2011. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software* 42, 10 (2011), 760–771.

[9] Richard M Everson, Jonathan E Fieldsend, and Sameer Singh. 2002. Full elite sets for multi-objective optimisation. In *Adaptive Computing in Design and Manufacture V*. Springer, 343–354.

[10] Jonathan E. Fieldsend and Richard M. Everson. 2014. Efficiently identifying Pareto solutions when objective values change. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO '14)*. Association for Computing Machinery, 605–612. https://doi.org/10.1145/2576768.2598279

[11] Jonathan E. Fieldsend, Richard M. Everson, and Sameer Singh. 2003. Using unconstrained elite archives for multiobjective optimization. *IEEE Transactions on Evolutionary Computation* 7, 3 (2003), 205–232.

[12] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.

[13] Tobias Glasmachers. 2017. A Fast Incremental BSP Tree Archive for Non-dominated Points. In *International Conference on Evolutionary Multi-Criterion Optimization, EMO 2017 (LNCS)*, H. Trautmann et al. (Ed.), Vol. 10173. Springer, 252–266.

[14] David Hadka. 2012. MOEA Framework-A Free and Open Source Java Framework for Multiobjective Optimization.

[15] Thomas Hanne. 1999. On the convergence of multiobjective evolutionary algorithms. *European Journal of Operational Research* 117, 3 (1999), 553–564.

[16] Andrzej Jaszkiewicz and Thibaut Lust. 2018. ND-Tree-Based Update: A Fast Algorithm for the Dynamic Nondominance Problem. *IEEE Transactions on Evolutionary Computation* 22, 5 (2018), 778–791.

[17] Mikkel T. Jensen. 2003. Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. *IEEE Transactions on Evolutionary Computation* 7, 5 (2003), 503–515.

[18] Joshua D. Knowles and David W. Corne. 2000. Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation* 8, 2 (2000), 149–172.

[19] Sanaz Mostaghim and Jürgen Teich. 2003. Quad-trees: A Data Structure for Storing Pareto-sets in Multi-objective Evolutionary Algorithms with Elitism. In *Evolutionary Multiobjective Optimization*, Ajith Abraham, Lakhmi Jain, and Robert Goldberg (Eds.). Springer, 81–104.

[20] Sanaz Mostaghim, Jürgen Teich, and Ambrish Tyagi. 2002. Comparison of data structures for storing Pareto-sets in MOEAs. In *IEEE Congress on Evolutionary Computation, 2002. CEC '02*. IEEE.

[21] Oliver Schütze. 2003. A New Data Structure for the Nondominance Problem in Multi-objective Optimization. In *International Conference on Evolutionary Multi-Criterion Optimization, EMO 2003 (LNCS)*, Vol. 2632. Springer, 509–518.

[22] Chuan Shi, Zhenyu Yan, Kevin Lü, Zhongzhi Shi, and Bai Wang. 2009. A dominance tree and its application in evolutionary multi-objective optimization. *Information Sciences* 179 (2009), 3540–3560.

[23] Ryoji Tanabe and Akira Oyama. 2017. Benchmarking MOEAs for Multi- and Many-objective Optimization Using an Unbounded External Archive. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. Association for Computing Machinery, 633–640.

[24] David A. Van Veldhuizen and Gary B. Lamont. 2000. Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art. *Evolutionary Computation* 8, 2 (2000), 125–147.

[25] J. Yuen, S. Gao, M. Wagner, and F. Neumann. 2012. An adaptive data structure for evolutionary multi-objective algorithms with unbounded archives. In *IEEE Congress on Evolutionary Computation, 2012, CEC '12*. IEEE, 1–8.