An Efficient Approach to Unbounded Bi-Objective Archives – Introducing the Mak_Tree Algorithm

Adam Berry
University of Tasmania
School of Computing
Private Bag 100, Hobart, Tasmania, Australia
Adam.Berry@utas.edu.au

Peter Vamplew
University of Ballarat
School of IT and Mathematical Sciences
PO Box 663, Ballarat, Victoria, Australia
p.vamplew@ballarat.edu.au

ABSTRACT
Given the prominence of elite archiving in contemporary multiobjective optimisation research and the limitations inherent in bounded population sizes, it is unusual that the vast majority of popular techniques aggressively truncate the capacity of archives and are based upon inefficient list representations. By forming better data structures and algorithms for the storage of archival members, the need for truncation is reduced and unbounded elite sets become viable. While work does exist in this vein, it is always of a general nature and significant improvements can be made in the bi-objective case. As such, this paper elucidates the unique properties of two-dimensional non-dominated sets and capitalises on these notions to develop the highly efficient and specialised bi-objective Mak_Tree algorithm. Theoretical results indicate that the specialised approach is preferable to pre-existing general techniques, while empirical analysis illustrates improved performance over both unbounded and bounded list techniques.

Categories and Subject Descriptors
E.1 [**Data Structures**]: Trees; I.2.m [**Artificial Intelligence**]: Miscellaneous.

General Terms
Algorithms and Performance.

Keywords
Multiobjective, multi-criteria, archives, data structures, red-black trees, balanced trees, bi-objective, dual-objective.

1. INTRODUCTION
Contemporary multiobjective optimisation research holds that the use of an elite archive of impressive solutions can fundamentally increase the performance of a given algorithm (see, for example, [12]). It is for this reason that the second generation of evolutionary optimisation techniques – that is, those that have followed the pioneering works of Schaffer [10], Srinivas and Deb [11], and others – are typically reliant on an active store of apparently good solutions. Most notable amongst this burgeoning array of methodologies are PAES [7], SPEA [14], SPEA2 [13], NSGA2 [3] and PESA [8]: all of which are reliant on archive-based elitism to drive solutions towards ever-better approximations of the Pareto optimal front.

Since these approaches implicitly (and often explicitly: see [7]) use a naïve list representation for members of the archive [6], the cost of searching and maintaining large or unbounded stores is prohibitive and must therefore be avoided. Indeed, the worst case complexity while using a list data structure is O(km) per archival insertion, where k is the number of objectives and m is the size of the archive. Given such inefficiency, efforts to reduce the size of m are hardly surprising.

Unfortunately, the artificial truncation of elite stores can lead to a variety of problems that result in performance degradation (as illustrated in Section 3). Chief amongst these issues is the potential for fronts to oscillate or recede due to the removal of members from the archive. The truncation operation also complicates crowding estimation and may lead to poor frontal exploration.

Given the problems which stem from the simple list representation of the archive and the mechanisms required to curb the complexity burden it carries, it is unusual that more work has not been directed at improving or finding better suited data structures or algorithms. Only Fieldsend *et al*. [4], Mostaghin *et al*. [9] and Jensen [6] have offered suitable alternatives via augmented tree structures. While their approaches each yield efficiency gains, significant improvements can be made in the bi-objective case.

As such, this paper proposes a new specialised bi-objective approach – the Mak_Tree algorithm – that capitalises on the unique properties of two-dimensional non-dominated sets to produce an unbounded archiving approach that is both highly efficient and low in storage space.

2. MULTIOBJECTIVE DEFINITIONS

2.1 Dominance
Solution S_a *strongly dominates*, and is thus better than, solution S_b (and conversely, S_b is *strictly dominated*, or bettered, by S_a) for a minimising multiobjective problem f with k objectives iff:

$$f_{1..k}(S_a) < f_{1..k}(S_b) \quad (1)$$

where *weak dominance/domination* is achieved by loosening the condition to allow equality between solutions[1].

2.2 Incomparability
Two solutions are *incomparable* if neither of the solutions weakly dominates the other:

$$\exists i, j \in \{1,..k\} : f_i(S_a) < f_i(S_b) \text{ and } f_j(S_a) > f_j(S_b) \quad (2)$$

[1] Note that all Pareto-based relationships described herein refer to performance – rather than genotypic – characteristics of solutions.

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

2.3 Equality

A solution is considered to be *equal* to another solution, in this context at least, if they produce the same results for all objectives in the given problem. This is typically known as *objective-space equality* and does not necessarily infer that the solutions are composed of identical decision variables.

2.4 Pareto Fronts

Note that unlike single-objective optimisation, where a strict ordering of fitness can be observed, multiobjective problems feature only a partial order and thus no single ideal or optimal solution typically exists. Instead, the explicit goal of all multiobjective optimisers is to find a good approximation of the *Pareto optimal front* – that is, the set of solutions that weakly dominate any other possible proposal. If a solution is non-dominated with respect to only the proposals produced thus far, it is considered *locally optimal* and thus forms part of the *local optimal front*.

While the quality of a produced-front is difficult to quantify, it is generally accepted that the practical ideal is a set of solutions that produce result vectors that are evenly distributed, well-spread and close to the true optimal front in objective-space.

2.5 Special Properties of Bi-Objective Fronts

Non-dominated fronts take on a series of special properties when the number of objectives is strictly two. Since these properties only exist in bi-objective domains, these problems should be considered as a special subset of the general multiobjective case.

2.5.1 Property One: Ordering

If solutions are ordered according to their ascending value (decreasing performance) on the first objective, then it is always the case that the same order represents descending value (increasing performance) on the second objective[2]. This can be verified by considering any two solutions in a non-dominated bi-objective set: if solution S_a outperforms solution S_b on objective one, it follows that for S_b to be non-dominated it must outperform S_a on objective two.

Interestingly, this property means that the extent of any bi-objective non-dominated front can be found by simply retrieving the head and tail of the ordered list. Similarly, the nearest neighbour of a solution in objective-space will always be either its predecessor or successor in the list.

2.5.2 Property Two: Objective Result Variance

Since any two non-dominated solutions may only be incomparable or equal with each other, an interesting property emerges in the two-dimensional case. Assuming that equal solutions may be stored as a single entity in the front, all such entities will contain completely unique values across a given objective. This is not true of the generic multiobjective problem, as solutions sharing a single objective result may still be incomparable.

2.5.3 Property Three: Dominated Sets

Consider a list of solutions ordered on the performance of an arbitrarily chosen objective: if solutions at index i and j (where $j \geq i$) are both dominated by some incoming proposal, then all solutions with an index between i and j (inclusive) are also dominated (referred to here as a *dominated set*). If i is the lowest dominated index and j is the highest, then the dominated set represents every dominated solution in the list. As an example, consider Figure 1. If a solution with results equaling (5,-2) is entered and it dominates solutions at index i and j, then the shaded region must be dominated. Since i and j are also the lowest and highest indexes of dominated solutions, the shaded region represents the complete dominated set.

(0,20)	(10,10)	(20,5)	(25,2)	(31,0)	(40,-5)
	i			*j*	

Figure 1 – An Ordered Bi-Objective List
The shaded region is dominated by incoming vector (5,-2).

Verifying this property is straight-forward. Let S_d be the dominating solution such that:

$$f(S_i) = \left(r_{i_1}, r_{i_2}\right);\ f(S_j) = \left(r_{j_1}, r_{j_2}\right);\ \text{and}\ f(S_d) = \left(r_{d_1}, r_{d_2}\right) \quad (3)$$

then, if property two holds, it is always the case that:

$$r_{d_1} \leq r_{i_1} < r_{j_1} \ \text{and}\ \ r_{d_2} \leq r_{j_2} < r_{i_2} \quad (4)$$

Capitalising on properties one and two, all solutions with index $w > i$ will be worse on objective one than S_d:

$$r_{w_1} > r_{i_1} \geq r_{d_1}\ \ \forall w \in \{i+1,..,m\} \quad (5)$$

and, by property one, solutions with index $w < j$ must be worse on objective two:

$$r_{w_2} > r_{j_2} \geq r_{d_2}\ \ \forall w \in \{1,..,j-1\} \quad (6)$$

Thus, solutions with indexes in the range $[i,j]$ are dominated by S_d.

2.5.4 Property Four: Non-Dominance

If any solution S_a is not dominated by its in-order predecessor S_p, then S_a is non-dominated with respect to the entire ordered list. If S_a has no predecessor, then the solution is the head of the ordered list and represents an extreme point in objective space – it too, will be non-dominated with respect to the set.

Again, the notion is easily verified. By the ordering property, it is always the case that S_a cannot be dominated by any successor S_s, since S_s will always be outperformed on the first objective. If S_p does not dominate S_a then, by property one, the following holds:

$$S_{a_2} < S_{p_2} < S_{p-1_2} ... < S_{0_2} \quad (7)$$

and S_a cannot be dominated by any preceding element in the list. Thus, S_a must be non-dominated with respect to the entire list.

2.5.5 Property Five: Dominance

A simple extension of property four is that if solution S_a is dominated by any member of the list, it must also be dominated by its predecessor. This property is proven by considering some solution S_d that dominates S_a. Since the predecessor S_p is preferable to S_a on objective one and since S_p is guaranteed to be better on objective two than S_d (recalling that S_d may not be a successor of S_a, and by applying property one), S_p must also dominate S_a.

3. ELITIST ARCHIVING

The truncated elitist archive typical of contemporary algorithms is simply a regularly updated approximation of the prevailing local optimal front. The optimisation algorithm capitalises on these locally optimal solutions to bias exploration around previously

[2] Without loss of generality, this type of ordering is assumed for all ordered bi-objective lists discussed herein.

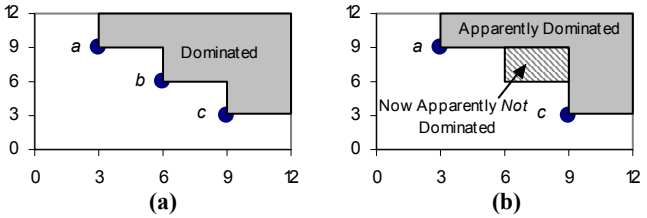


Figure 2 – Frontal Degradation in Objective Space
The loss of solution *b* from the archive means that the highlighted region is incorrectly labelled as non-dominated.

beneficial areas of the search space.

While the results achieved by applying such online storage to multiobjective algorithms are significant, there exist enough limitations and drawbacks in the truncated approach to indicate that unbounded storage is preferable. Specifically, in trying to form a good approximation of the locally optimal front, truncated archives may suffer from frontal degradation, inaccurate crowding measures and the loss of expensive regions of objective space.

3.1 Frontal Degradation

Given that a local optimal front will be composed of solutions that are strictly non-dominated by any other previously generated proposal, it is reasonable to expect all members of an elite archive to meet the same requirements (even if they form only a subset of the true front). However, the truncation operation means that the quality of the archive may degrade over time – invalidating the optimality requirement by allowing weak solutions into the archive.

The potential for decreasing archive quality is best exemplified via a diagram. Consider the extremely simple three-member elite archive illustrated in Figure 2a: if solution *b* is removed due to a truncation operation, the map of dominated space becomes inaccurate – the highlighted region in Figure 2b should be included, but is not. The effect is that a poor solution which was dominated by *b*, but is otherwise incomparable with the remaining members of the archive, will be incorrectly accepted as an elite member.

The effect of degrading fronts can be extremely detrimental to the performance of any given algorithm. In the extreme case, frequent truncation of archives may lead to a retreating front – where the archive becomes a progressively worse approximation of the Pareto optimal set. The more likely case however, and one which Fieldsend *et al.* have verified empirically [4], is frontal oscillation – where good solutions are truncated from the archive and then rediscovered as part of the algorithm's search procedure. Obviously, the time spent rediscovering good ideas is better spent investigating less populated areas of the front and is costly to both the efficiency and efficacy of the search process.

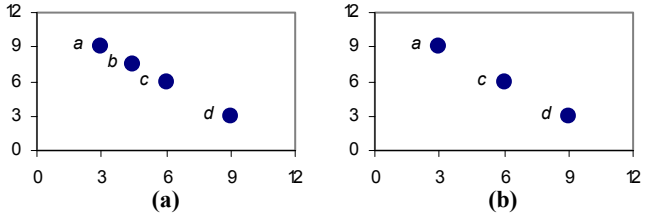


Figure 3 – Poor Crowding Estimation in Objective-Space
The loss of solution *b* from the archive means that the resultant archive appears evenly distributed.

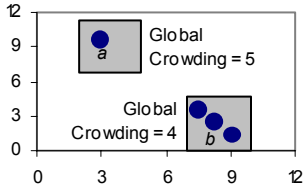


Figure 4 – Inaccurate Global:Local Crowd Mappings
Solution *a* is in a globally crowded region of objective-space, but a locally uncrowded zone. The loss of *a* would lead to excessive search pressure around *b*.

3.2 Inaccurate Crowd Estimation

Given that one of the explicit aims in developing a good approximation of the Pareto optimal front is achieving an evenly distributed set of solutions, crowding estimation is very important in directing the search towards areas of low result-density. Unfortunately, truncation of archives can lead to misleading density approximations, as crowding measures will be based on an incomplete set of the true local optimal front. In Figure 3a it is obvious that solutions *a*, *b* and *c* occupy the most densely explored region of objective space, but the truncation of point *b* from the archive leads to a uniform spacing of points that de-emphasises the isolation of *d* (as illustrated in Figure 3b). The danger that exists here is that solutions may be added and subsequently truncated from the crowded region – worsening the true distribution of points with little effect on the apparent archive-based crowding of the same area.

It seems reasonable then, to apply crowding measures that consider all solutions encountered throughout the run, rather than just those stored locally in the archive. However, such an approach is also open to problems – as illustrated in Figure 4. If truncation is to be applied according to some global crowding measure, the archive will lose solution *a* and become overly concentrated around solution *b*. Such degeneration into locally crowded arrangements not only diminishes the likelihood of finding a richly distributed front efficiently (since the search will become focused around a single area of objective space), but it also permits increased frontal oscillation.

3.3 Loss of Expensive Regions

Under the presence of discontinuous fronts, isolated regions, constraints, deception and bias, certain solutions can reasonably be thought of as being more valuable to the search process than others. The loss of a highly valuable solution that resides isolated in a disconnected and sparsely populated portion of objective space, for instance, would severely inhibit the capacity of the algorithm to search the region in which it resides. Indeed, the effect is even more pronounced than oscillation, as simply rediscovering the lost solution may be extremely difficult.

The principle here is an important one – the more complex the objective space becomes, the more important the size of the archive. Specifically, archival size acts as an artificially enforced threshold on the number of distinct regions that can be explored simultaneously by a search algorithm. Should the number of distinct regions in objective space surpass the size of the archive, the algorithm will either be confined to a subset of the objective space or, more likely, suffer significant performance degradation as the archive consistently cycles between available zones – repetitively discarding solutions that may have been invaluable to a balanced search.

4. ELITE ARCHIVE DATA STRUCTURES

4.1 Quad-Trees

The first series of algorithms described specifically for unbounded archiving are the three variants proposed by Mostaghim, Teich and Tyagi [9]. All three approaches result in the storage of non-dominated solutions in quad-trees and differ primarily in the way deletion occurs. Still, the same general problems beset each technique: deletion may require re-insertion, despite practical improvements made in the *Quad-Tree2* and *Quad-Tree3* algorithms; and empirical evidence suggests that the approach is *more* costly than a simple linear list for a large number of evaluations. Indeed, on the relatively simple *ZDT1* function developed by Zitzler, Deb and Thiele [12], the variants are slower on average than an unbounded list even after 400,000 evaluations.

Thus, the proposed quad-tree-based algorithms are ultimately of limited practical worth (and will therefore not be discussed further in this paper), but provided the impetus for similarly motivated ideas to be explored.

4.2 Dominated Trees

Improving significantly on the performance of the Quad-Tree algorithms, Fieldsend, Everson and Singh [4] introduce both a new selection mechanism for unbounded archives – namely, *Partitioned Quasi-Random Selection* (PQRS) – and efficient data structures for the storage and maintenance of archival solution sets – specifically, *Dominated* and *Non-Dominated Trees*.

Maintenance of the archive occurs in a single data structure – be it either the Dominated or Non-Dominated Tree. Both alternatives are similarly composed of approximately $\lceil m/k \rceil$ unique *composite points*: combinations of the objective-results from multiple independent solutions (*constituents*). Importantly, no composite point is ever incomparable with another, and so the weakly-dominates relation can impose a complete order on any set of points – thus facilitating the use of ordered data structures, such as balanced binary trees.

Empirically, Fieldsend *et al.* demonstrate the efficiency, and efficacy, of using the proposed mechanisms, with particularly impressive timing results when compared to the performance of an unbounded linear list. However, the approach is not without its limitations. In particular, the selected Dominated Tree structure may sporadically require complete re-building in order to maintain a suitable approximation of the optimal number of composite points ($\lceil m/k \rceil$). Moreover, to determine non-dominance of a solution with respect to the archive (which must occur on *every* attempted archival insertion), the constituents of composite points that are incomparable with the applicant solution must each be checked for dominance in-turn. The same constituent verifications must also be made for both the weakest composite point to dominate the solution (C_w) and all composites that share an axis with C_w. The effect is that the logarithmic nature of tree-insertion is offset by the need for periodic linear searches of solution sub-sets. While Fieldsend *et al.* correctly note that the algorithm is unlikely to degenerate into a truly linear search of the entire population of solutions, the over-head it induces is certainly significant enough to degrade performance.

4.3 Orthogonal Range Searching

While much of Jensen's [6] thorough paper regarding the application of data structures to multiobjective optimisation focuses on improving the run-time performance of non-dominated sorting, it also briefly addresses archive maintenance. In particular, the paper suggests the use of fractional cascading and Dynamic Range Trees to enable orthogonal (rectangular) range queries for the identification of dominated or dominating solutions. Providing efficient run time complexities, the approach is of significant merit, though the use of Dynamic Range Trees will incur a non-linear storage complexity. Moreover, since it was not the focus of his work, Jensen does not implement or test the data structure, meaning that empirical performance analysis is presently unavailable. Thus, expansion upon this original proposal, particularly on problems of low dimensionality, lies as an important area of future work.

5. INTRODUCING THE MAK_TREE

All algorithms proposed thus far are, quite reasonably, designed for generic k-objective non-dominated sets. However, as introduced in Section 2.5, the bi-objective case carries a number of unique properties that may be manipulated to form more efficient specialised data structures and storage algorithms. The Mak_Tree algorithm represents such an approach, delivering a highly efficient technique that is specifically tailored to the needs of bi-objective optimisation.

5.1 The Mak_Tree Data Structure

The Mak_Tree is a generic label for any binary tree structure that is dynamic, self-balancing and ordered (arbitrarily) by performance on the first objective. A node in any Mak_Tree represents a collection of solutions with identical objective scores (to enable property two) and the tree, as a whole, is strictly non-dominated. As such, the tree itself is not particularly interesting and can take on a wide variety of forms, from AVL structures to the Red-Black trees [1,5] used herein (see Figure 6 for a Red-Black representation of the data in Figure 1). It is the Mak_Tree algorithms, which build upon the unique properties of bi-objective sets and simple binary search trees, that are interesting and will form the focus of this work.

Still, before moving on, it is worthwhile noting that the structure facilitates the efficient discovery of a number of interesting frontal properties. Since the extent of a front is merely the head and tail of the sorted list (see Section 2.5.1), the Mak_Tree can locate such solutions in $O(\log m)$. Additional information, such as the least/most occupied node, or the oldest/youngest stored solution, can also be maintained via branch annotations; while the structure can be easily extended to support PAES-like cells.

5.2 Updating the Mak_Trees

It is obvious that for a Mak_Tree to remain non-dominated, it must only accept non-dominated solutions into the tree and prune any solution that becomes dominated due to an insertion. The basic algorithm to achieve such behaviour is outlined in Algorithm 1, where solution S is inserted into archive A. To further illustrate the behaviour of the update operations introduced in the algorithm, it is beneficial to consider two general concepts: verifying non-dominance and locating dominated nodes.

Figure 6 – An Example Mak_Tree (Via a Red-Black Structure)

5.2.1.1 Verifying Non-Dominance

In order for a proposal to be inserted into the archive, it must not be dominated by any other stored solution. Considering the procedure adopted in Algorithm 1, it is not necessarily intuitive how this is achieved. However, recall that a unique property of bi-objective non-dominated fronts is that if a solution is non-dominated with respect to its predecessor, then it is also non-dominated with respect to the remainder of the front (property four, Section 2.5.4). Since insertion mirrors simple binary-tree insertion, and this means that the proposal will always be compared with its predecessor (unless it is dominated before this point or no such predecessor exists), any successfully added solution is guaranteed to be non-dominated.

5.2.1.2 Locating Dominated Nodes

Central to the algorithm for locating dominated nodes is the concept of dominated sets introduced in property three (Section 2.5.3) – that is, if a solution dominates nodes at index i and index j, it must also dominate the set of nodes with indices between i and j. Since this essentially represents a range-query, it is useful to build on the range-related properties inherent in simple binary search trees. Specifically, it is always the case that if a node y is the right-child of x, then all left-descendents of y will have in-order indices between the indexes of x and y. This property also holds if y is a left-child of x and all right-descendents of y are considered. The effect is that once an initial dominated node has been identified, any dominated node to its right is guaranteed to have a dominated left sub-tree, while any dominated node to its left must have a dominated right sub-tree. As an example, consider Figure 6; if both (25,2) and (10,10) are dominated, then the (20,5) sub-tree must also be completely dominated.

With this in mind, locating dominated nodes becomes relatively straight-forward and particularly efficient in the Mak_Tree algorithm. After the discovery of the first dominated node B at index b, the location of all dominated nodes with indices between b+1 and j requires at most $O(\log m)$ comparisons. Specifically, the search proceeds as follows (starting at the right-child of B): if the current node is dominated label it and the left sub-tree as dominated and search the right sub-tree; otherwise, everything to the right of the current node is non-dominated, so move left. The discovery of all nodes with indices between i and b-1 requires little modification to the standard insertion procedure – on dominance, the search progresses left as usual, with the node and right sub-tree marked as being completely dominated.

Once discovered, the dominated nodes must be removed from the tree (lines 20–22 in Algorithm 1). It is important to note that dominated sub-trees and individual dominated nodes are handled independently during this deletion procedure. As evidenced in Section 6.1.2.2, the deletion of large sub-trees, in particular, can afford the Mak_Tree algorithm an impressive performance gain.

Algorithm 1 – Insertion into the Mak_Tree

Inputs:

$S = (s_1, s_2)$ — The inserted solution, where s_1 and s_2 denote objective scores.

```
1:  if (A = ∅)                                          If the tree is empty
2:    A := {S}                                          add the solution to the tree.
3:  else
4:    rejected := false; del_nodes := {}
5:    del_subs := {}; B := null; node := root_A         Start the search at the root.
6:    while ((node ≠ leaf) and (rejected ≠ true))
7:      if (node ≺ S)                                   If the current node dominates S
8:        rejected = true                               then the algorithm ends.
9:      else if (S ≺ node)                              If the solution dominates the
10:       handle_dominance (S,node,B,                   current node then call the
          del_nodes,del_subs)                           handle_dominance helper.
12:     node := left_or_insert(node,S)                  Move left, or insert if at leaf.
13:     else if (S = node)                              If the solution and node share
14:       node := node + s                              objective scores, add s to node
15:       rejected = true                               and end the algorithm.
16:     else if (s_1 < node_1)                          If the objective-one score of S
17:       node := left_or_insert(node,S)                is less than that of node, move
18:     else                                            left or insert if node is a leaf.
19:       node := right_or_insert(node,S)               Otherwise, move/insert right.
20:   if ((B ≠ null) and (rejected = false))            If the solution was dominating
21:     delete_all_sub_trees(del_subs)                  then remove all dominated
22:     delete_all_nodes(del_nodes)                     nodes and sub-trees.
```

Algorithm 2 – Handle_Dominance Helper

Inputs:

S	The inserted solution.
$node$	The dominated node being examined.
B	The first found dominated node.
del_nodes	The set of individual nodes that are dominated by S.
del_subs	The set of sub-trees that are completely dominated by S.

```
1:  del_nodes := del_nodes + node
2:  if (B = null)                                       If B has not yet been found.
3:    B := node
4:    current := right(B)
5:    while(current ≠ null)
6:      if (S_2 ≤ current_2)                            If the solution dominates the
7:        del_nodes := del_nodes + current              current node, both the node and
8:        if (left(current) ≠ null)                     its left sub-tree (if it exists)
9:          del_subs := del_subs + left(current)        should be deleted.
10:       current := right(current)                     Move right.
11:     else                                            The solution can only dominate
12:       current := left(current)                      nodes to the left, so move left.
13: else if (right(node) ≠ null)                        If B has been found, all nodes
14:   del_subs := del_subs + right(node)                to the right must be dominated.
```

6. RESULTS

6.1 Complexity Analysis

When considering the performance of a given data structure and algorithm, both time and space complexity are significant. While emphasis is typically placed on efficiency, high space complexity can induce tighter limits on the feasible capacity of a given structure. Since the very nature of unbounded archives is to store particularly large solution sets, it is important to reduce such limits.

6.1.1 Space Complexity

The optimal space complexity for any unbounded archive is $O(m)$, as all solutions in the archival set must be accessible. Since the Mak_Tree contains at most m nodes and the simple nature of the tree requires no repetition of nodes or solutions, the spatial complexity of the Mak_Tree is optimal and equal to $O(m)$.

The Dominated Tree structures achieve similarly optimal space complexity, though it requires cleaning at appropriate thresholds to ensure that such optimality holds.

Finally, the Dynamic Range Tree suggested by Jensen requires $O(m \log m)$ space (see [2]) due to its two-level tree structure and thus represents the most expensive storage option presented in the literature thus far.

6.1.2 Run-Time Complexity

With respect to performance complexity, it is useful to consider the insertion of two distinct types of solution: strictly non-dominating – those that are dominated by or equal to some component of the archive, or otherwise completely incomparable – and dominating.

6.1.2.1 Insertion of Non-Dominating Solutions

As alluded to in Section 5.2.1.1, the Mak_Tree algorithm is particularly efficient when inserting non-dominating solutions, as the algorithm requires at most $O(\log m)$ dominance comparisons during the simple binary navigation. Since insertion in Red-Black Trees will only ever require at most $O(\log m)$ node re-colourings and one rotation, the worst-case time cost for the insertion of any non-dominating solution in the Mak_Tree is $O(\log m)$. Such performance is optimal for any structure based on self-balancing binary search trees.

The Dominated Tree structures achieve similar performance, though the need for linear checking of composite constituents may be costly. If the solution under consideration is dominated, then performance is optimal and requires only $O(\log m)$ dominance comparisons. However, the algorithm is sub-optimal under the insertion of strictly incomparable solutions, with the burden of checking c constituents for dominance resulting in a search cost of $O(\log m + c)$. While not discussed explicitly in the source paper, it also seems likely that verification of solution equality would require the checking of constituents belonging to the composite point sharing an axis with the solution – thus leading to sub-optimal performance if the inserted solution already has an equivalent stored. Additionally, note that these time complexities only hold under suitable maintenance of the corresponding Dominated Tree and assume that the need for cleaning is infrequent.

A query in the two-dimensional fractional cascading Dynamic Range Tree will cost $O(\log m \log(\log m) + \alpha)$, where α is the set of solutions satisfying the range query (see [2] for succinct summaries of Dynamic Range Tree behaviours and costs). For any insertion, the algorithm will require at most two orthogonal range queries – the first identifies those solutions that dominate the proposal, while the second highlights archival members dominated by the proposal. For a non-dominating solution, the second query will always return the empty set, while the first query need only return the first dominating node (since any $\alpha>0$ is enough to disqualify the incoming solution from inclusion). Thus, the total query cost for identifying a non-dominating node is $O(\log m \log(\log m))$. Since the update cost of the structure is also $O(\log m \log(\log m))$, the total time cost for the insertion of a non-dominating solution into the Dynamic Range Tree is also $O(\log m \log(\log m))$.

6.1.2.2 Insertion of Dominating Solutions

Handling dominating solutions is an inherently more expensive proposition as it will also require both the identification of dominated solutions and their subsequent removal. As discussed in Section 5.2.1.2, all η dominated nodes can be efficiently discovered with $O(\log m)$ dominance comparisons using the Mak_Tree. If the naïve approach is taken and each of the η dominated nodes are deleted in-turn, the final cost of identifying and removing dominated nodes with the Mak_Tree is $O(\eta \log m)$. However, given that the query may return dominated sub-trees, it is useful to capitalise on sub-tree deletion in Red-Black Trees. Specifically, for any sub-tree requiring deletion, the cost of removing the sub-tree is $O(\log \tau \log m)$ rather than $O(\tau \log m)$, where τ is the size of the sub-tree. For small η the difference is not particularly significant, but as η increases, so must the size or number of sub-trees identified for deletion. The effect is particularly evident in analysing the worst-case insertion, where the number of dominated nodes tends towards the size of the tree[3] ($\eta \approx m$). In this case there will be approximately $(2 \log m)$ sub-trees and $(2 \log m)$ separate individual nodes that require deletion. The individual nodes will cost a total of $O((\log m)^2)$ to remove (though the amortised cost of this operation can be reduced if individual deletion occurs after sub-tree deletion is completed). The cost of deleting all sub-trees will be $((\log m) \times 2 \times (1 + 2 + 3 + ... + \log (m-1))) \Rightarrow O(\log m (\log m)^2) = O((\log m)^3)$. Thus, the total worst-case bound of identifying and removing dominated nodes from a Mak_Tree is $O((\log m)^3)$.

The Dynamic Range Tree can identify all η dominated solutions using an orthogonal query that costs $O(\log m \log (\log m) + \eta)$. Once discovered, the dominated solutions must then be removed from the tree at a cost of $O(\eta \log m \log (\log m))$. Thus, when worst-case insertion occurs, the Dynamic Range Tree carries a cost of $O(m \log m \log (\log m))$.

To locate all η nodes for deletion, the Dominated Tree must perform two binary searches and subsequently check every constituent of dominated composite points. Thus, the cost of locating all dominated nodes in the archive will be at least $O(\log m + \eta)$ and, more generally, $O(\log m + \eta + \delta)$, where δ is the number of constituents belonging to dominated composite points. To minimise this cost, Fieldsend *et al.* propose the use of the alternative (though conceptually very similar) Non-Dominated Tree structure which ensures that any dominated composite point will feature only dominated constituents. While offering some performance gain, constituents belonging to incomparable composite points will still need to be verified. Thus, the cost of locating dominated nodes using the Non-Dominated Tree variant will be $O(\log m + \mu)$, where μ is the number of incomparable composite constituents that must be examined.

Upon the deletion of any solution, the corresponding Dominated Tree structure must update all composite points for which the solution was a constituent. If the solution represented the only constituent of a composite point, the composite point will be removed from the archive in $O(\log m)$ time (assuming use of a self-balancing tree). If the solution was used in the most dominating composite point then, in the two-dimensional case, the remaining constituent of the composite represents the new coordinates of the dominating point and the update can be completed in constant time. Otherwise, the affected composite

[3] Though it is actually trivial to handle complete domination: if the left-most and right-most nodes in the tree are dominated, simply have the inserted solution become the root.

Table 1 – Big-Oh Space and Performance Complexity for Examined Data Structures

† Assumes constituents do not contribute to a large number of composites. ‡ Can be reduced to $O(\log m)$ for insertion of dominated solutions. ♦Assumes infrequent composite cleaning. ♣ Can be improved as η increases. ♥ Dominated Tree specific. ◊ Non-Dominated Tree specific.

Approach	Spatial Complexity	Cost of Inserting Non-Dominating Solutions	Cost of Searching for Dominated Solutions	Cost of Deleting Dominated Solutions
Mak_Tree	$O(m)$	$O(\log m)$	$O(\log m)$	$O(\eta \log m)^{\clubsuit}$
Dominated Tree	$O(m)^{\dagger}$	$O(\log m + c)^{\ddagger\blacklozenge}$	$O(\log m + \eta + \delta)^{\blacklozenge\heartsuit}$ || $O(\log m + \mu)^{\blacklozenge\diamond}$	$O((V+\eta) \log m)^{\blacklozenge\clubsuit}$
Dynamic Range Tree	$O(m \log m)$	$O(\log m \log(\log m))$	$O(\log m \log (\log m) + \eta)$	$O(\eta \log m \log (\log m))$

point is updated through the re-use of a constituent of the succeeding (dominated) composite – requiring an $O(\log m)$ successor search. Thus, the cost of deleting a single solution from any Dominated Tree is $O(v \log m)$, where v is the number of composite points that the solution is a member of. Deletion of η solutions will cost $O((V+\eta) \log m)$, where V is the total number of extra composite points to which the η dominated solutions belong.

Note that some practical improvements can be made over this performance if it is known when all constituents of a composite point are dominated (as is the case for certain composite points in Non-Dominated Trees). Given this knowledge, the completely dominated composite point can be deleted from the tree without the need for intermediary coordinate re-labeling. However, given that the constituents of the dominated composites may also form part of non-dominated composites elsewhere in the tree, the overriding complexity costs remain much the same unless frequent cleaning occurs. Better improvements are seen when applying binary sub-tree deletion to remove sets of completely dominated nodes, though the need to check and update constituents in non-dominated composite points curb the advantages afforded to such an approach when η is small. Still, such techniques may hold merit and are certainly worth further consideration in future work.

6.1.2.3 Summary of Run-Time Complexity

As evidenced in Table 1, the Mak_Tree provides superior time complexities to those previously discussed in the literature. The Dynamic Range Tree is generally more expensive due to its two level structure – though the optimisation of the approach or the application of one-dimensional range trees rests as an interesting area of future work. The Dominated Tree structures proposed by Fieldsend *et al.* provide similar outcomes to the Mak_Tree, but only under the assumption that constituents contribute to a very small number of points and that composite cleaning is infrequent. Since composite cleaning requires the successive deletion of all constituents from all composite points they contribute to (excluding the least dominating node) or the complete rebuilding of the tree, the cost of such a procedure is prohibitive in all but sparse usage.

6.2 Empirical Analysis

While theoretical analysis of algorithms provides an important grounding for the understanding of performance – particularly with respect to worst-case bounds – empirical examinations can elucidate behaviour under more realistic conditions. As such, the performance of the Mak_Tree was evaluated over five distinct test functions that each exhibit different problem features. Specifically, the tests (*T1–T5*) are equivalent to the *ZDT1–4* and *ZDT6* functions proposed by Zitzler *et al.* [12], though with a linear shift of 0.35 applied to all decision variables involved in the formation of the g function (for reasons to be explored in future work). To place the performance of the Mak_Tree in context, the behaviour of both the common linear-list approach and variously sized truncated archives are also considered. In this work, the truncated archives maintain a constantly ordered list of non-dominated solutions, such that a complete set of nearest neighbour calculations (which take place whenever the archive threshold is breached) can occur in only $O(m)$ time. Since solutions are inserted into the archive individually in these tests, truncations only ever require the removal of the single most crowded solution.

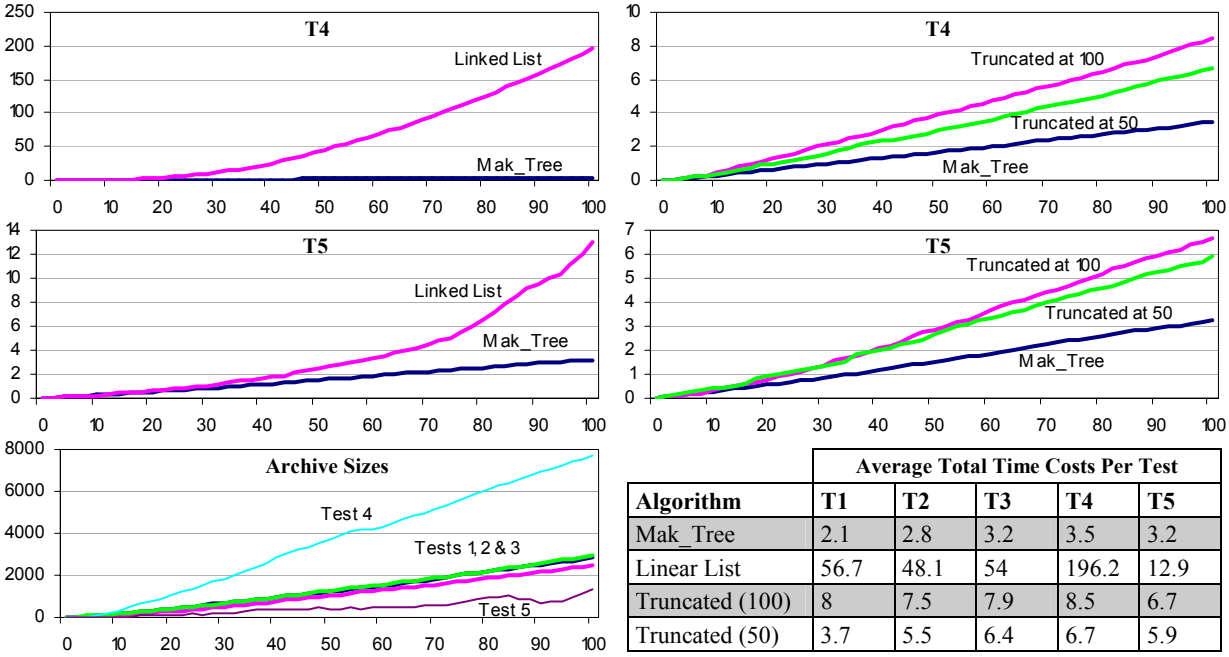
The results illustrated in Figures 7–12, indicate the average cumulative time costs across twenty distinct solution sets on each of the five problems. Each input set was produced by a single standard, real-valued, NSGA-II run (with population size of 100, crossover rate of 0.9 and mutation probability of $1/v$ – where v is the number of decision variables). Note that while only problems *T4* and *T5* are displayed graphically, the results illustrated are indicative of those seen across the complete test suite.

The Mak_Tree algorithm outperforms the naïve list approach on all examined problems. Note also that while the time-cost of the list approach grows exponentially, the Mak_Tree has approximately linear growth. This is particularly significant given that results reported on the performance of the Dominated Tree also indicate exponential growth.

Perhaps more significantly, the Mak_Tree also betters the performance of the truncation techniques. While both approaches yield approximately linear performance, the Mak_Tree is faster on every examined problem. The result is an important one, particularly considering the disadvantages inherent in truncation (see Section 3) – if the Mak_Tree can outperform limited archives, then the approach is applicable not just for particularly complex problems, but for generic use in all bi-objective problem domains.

7. CONCLUSIONS AND FUTURE WORK

By capitalising on the unique properties of non-dominated bi-objective sets, the Mak_Tree algorithm offers an efficient approach for the storage, querying and updating of unbounded two-dimensional elite archives. Big-oh results confirm that this specialist approach achieves optimal run-time complexity with respect to the insertion of strictly non-dominating solutions into a tree structure. In the more complex case, where a dominating solution is added to the archive, the Mak_Tree algorithm is preferable to pre-existing techniques. Unlike more complex structures, the Mak_Tree also achieves optimal space complexity and should be relatively straight-forward to implement. With respect to empirical results, the Mak_Tree is shown to outperform both unbounded and tightly bound archives using the naïve, though common, linear-list technique. Such promising results suggest that unbounded archives, and the Mak_Tree in particular, can be used in bi-objective problem domains with very little, if any, additional cost. Since problems are known to exist with pre-



	Average Total Time Costs Per Test				
Algorithm	**T1**	**T2**	**T3**	**T4**	**T5**
Mak_Tree	2.1	2.8	3.2	3.5	3.2
Linear List	56.7	48.1	54	196.2	12.9
Truncated (100)	8	7.5	7.9	8.5	6.7
Truncated (50)	3.7	5.5	6.4	6.7	5.9

Figures 7–12 – Empirical Results

For graphs, the x-axis represents the number of solutions processed (in thousands); the y-axis on the T4 and T5 graphs is the cumulative processing time in seconds; the y-axis on the archive sizes graph is the total number of unique (non-equal) members stored in the archive for a given test function.

existing truncation approaches, such low-cost inclusion of unbounded archives is a significant step towards improving the performance of bi-objective optimisation techniques.

While the results are promising, the empirical investigations address only a small range of test functions and a more expansive investigation is necessary. Additional avenues of future work include the specialisation of pre-existing archiving techniques to the two-dimensional case; the formation of optimisation algorithms to capitalise on the properties of the Mak_Tree; and the generalisation of the Mak_Tree to higher-dimensional problems. Also note that while the Mak_Tree is designed for unbounded elite archives, it also provides an appropriate structure for complete population storage and truncated elite archives.

8. ACKNOWLEDGMENTS

The authors would like to thank Pauline Mak, Trixie Berry, Michael Berry and the School of Computing at UTas.

9. REFERENCES

[1] Bayer, R., *Symmetric Binary B-Trees: Data Structures and Maintenance Algorithms*. Acta Informatica, 1972.

[2] Chiang, Y.-J. and Tamassia, R., *Dynamic Algorithms in Computational Geometry (Revised Version)*. Proceedings of IEEE Special Issue on Computational Geometry, 1992.

[3] Deb, K., Agrawal, S., Pratab, A., and Meyarivan, T. *A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II*. in *Proceedings of PPSN VI*. 2000: Springer. LNCS No. 1917.

[4] Fieldsend, J.E., Everson, R.M., and Singh, S., *Using Unconstrained Elite Archives for Multiobjective Optimization*. IEEE Transactions on Evolutionary Computation, 2003. 7(3): p. 305-323.

[5] Guibas, L.J. and Sedgewick, R., *A Dichromatic Framework for Balanced Trees*. Proceedings of the 19th Annual Symposium on Foundations of Computer Science, 1978.

[6] Jensen, M.T., *Reducing the Run-Time Complexity of Multiobjective EAs: The NSGA-II and Other Algorithms*. IEEE Transactions on Evolutionary Computation, 2003. 7(5): p. 503-515.

[7] Knowles, J.D. and Corne, D.W., *Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy*. Evolutionary Computation, 2000. 8(2): p. 149-172.

[8] Knowles, J.D., Corne, D.W., and Oates, M.J. *The Pareto-Envelope based Selection Algorithm for Multiobjective Optimization*. in *Proceedings of PPSN VI*. 2000: Springer.

[9] Mostaghim, S., Teich, J., and Tyagi, A. *Comparison of Data Structures for Storing Pareto-sets in MOEAs*. in *Congress on Evolutionary Computation*. 2002: IEEE Service Center.

[10] Schaffer, J.D. *Multiple Objective Optimization with Vector Evaluated Genetic Algorithms*. in *Genetic Algorithms and their Applications: Proceedings of the First International Conference on Genetic Algorithms*. 1985.

[11] Srinivas, N. and Deb, K., *Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms*. Evolutionary Computation, 1994. 2(3): p. 221-248.

[12] Zitzler, E., Deb, K., and Thiele, L., *Comparison of Multiobjective Evolutionary Algorithms: Empirical Results*. Evolutionary Computation, 2000. 8(2): p. 173-195.

[13] Zitzler, E., Laumanns, M., and Thiele, L. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*. in *EUROGEN 2001*. p. 95-100.

[14] Zitzler, E. and Thiele, L., *An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach*. 1998, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH).