# Comparison of Data Structures for Storing Pareto-sets in MOEAs

Sanaz Mostaghim, Jürgen Teich and Ambrish Tyagi
Electrical Engineering Department
Paderborn University, Paderborn, Germany
{*mostaghim, teich, tyagi*}@*date.upb.de*

**Abstract - In MOEAs with elitism, the data structures and algorithms for storing and updating archives may have a great impact on the CPU time, especially when optimizing continuous problems with larger population sizes. In this paper, we introduce Quad-trees as an efficient data structure for storing Pareto-points. Apart from conventional linear lists, we have implemented three kinds of Quad-trees for the archives. These data structures were examined for different examples. The results presented show that linear lists perform better in terms of CPU time for small population sizes whereas tree structures perform better for large population sizes.**

## I. Introduction

Multi-objective optimization (MO) has been investigated a lot during the last years [1], and it is proven that stochastic search methods such as evolutionary algorithms (EA), simulated annealing (SA) and Tabu search (TS) often provide the best solutions for complex optimization problems. Up to now there are a few multi-objective evolutionary algorithms (MOEA), which can be divided into two groups. The first group contains the MOEAs that always keep the best solutions (Pareto-points) of each generation in an *archive*, and they are called MOEAs with elitism. In the second group, there is no archive for keeping best solutions and MOEA may loose them during generations. MOEAs with elitism are studied in a few methods like Rudolph's Elitist MOEA, Elitist NSGA-II, SPEA, PAES (see [2] for all) and SPEA2 [7].

In this paper, we address the problem of storing the Pareto-points in the archive in such a way, which helps to attain the desired results in the least possible time. Hence, the need for an efficient data structure to hold these solutions. The motivation of our work comes from the *Quad-tree* data structure proposed by Finkel and Bentley in 1974. Later Habenicht [4] adapted Quad-trees to the problem of identifying non-dominated criterion vectors. Finally, Sun and Steuer [6] improved [4] to make the storage more efficient. However, no one has so far investigated Quad-tree data structures in the context of evolutionary algorithms, i.e. MOEAs. Up to now, linear lists were used as the archives for storing Pareto-points in MOEAs, see e.g. [8]. We apply the Quad-tree structure from [6] to MOEAs (especially SPEA) with improvements for shortcoming that it had in dealing with some special cases. In this paper, three kinds of Quad-trees are examined, which are called Quad-tree1, Quad-tree2 and Quad-tree3. The Quad-tree1 data structure, derived from [4], has two disadvantages: The CPU time depends on the order in which the vectors are added and deleting a vector means deleting all the subtree and reinserting all vectors again from the root, which takes a long time. Quad-tree2, which is a new improved data structure proposed by us, uses some flags for deleting the dominated vectors in the Quad-tree1. The third data structure is the implementation of Sun and Steuer's Quad-tree [6]. We have extended this data structure to make it more efficient for MOEA archives, calling it Quad-tree3 in the following. In this paper, a comparative study of the traditional linear lists with the new Quad-trees is given, i.e. the CPU times are compared for each data structure. We have used different kinds of test problems, which are applied for testing corresponding MOEAs. These tests, taken from [8] and [3], contain 2- and 3-objective test problems. Our results show that for larger population sizes Quad-trees are more efficient in CPU time than linear lists.

In Section 1, we have given some brief definitions and explain the MOEA, which is used to test different data structures for the archive. Section 2 introduces the linear lists and Quad-tree data structures as well as the algorithms for maintaining the archive for the three variants called Quad-tree1, Quad-tree2 and Quad-tree3. Section 3 introduces the test functions and experimental results that show that Quad-trees perform better for problems with larger population size. We finally give conclusions and an outlook of further work.

### A. Multi-objective Optimization

A multi-objective optimization problem is of the form

minimize $\{f_1(\vec{x}), f_2(\vec{x}), \cdots, f_m(\vec{x})\}$
subject to $\vec{x} \in S$

involving $m(\geq 2)$ conflicting objective functions $f_i : \Re^n \rightarrow \Re$ that we want to minimize simultaneously. The *decision vectors* $\vec{x} = (x_1, x_2, \cdots, x_n)^T$ belong to the feasible region $S \subset \Re^n$. The feasible region is formed by constraint functions.
We denote the image of the feasible region by $Z \subset \Re^m$ and call it a feasible objective region. The elements of $Z$ are called objective vectors and they consist of objective (function) values $\vec{f}(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_m(\vec{x}))$.
A decision vector $\vec{x}^* \in S$ is Pareto-optimal if there does not exist another $\vec{x} \in S$ such that $f_i(\vec{x}) \leq f_i(\vec{x}^*)$ *for all* $i = 1, \cdots, m$ *and* $f_j(\vec{x}) < f_j(\vec{x}^*)$ *for at least one index* $j$. An objective vector is Pareto-optimal if the corresponding decision vector is Pareto-optimal.

### B. The Strength Pareto Approach

The MOEA we used here to test the different data structures is the Strength Pareto Approach (SPEA) as introduced by Zitzler and Thiele [8]. This algorithm introduces elitism by explicitly maintaining an external population (archive). The archive is used to evaluate the individuals in the population and stores the non-dominated solutions. At every generation, newly found non-dominated solutions are compared with the existing archive and the resulting non-dominated solutions are preserved. Figure 1 shows the structure of this evolutionary algorithm.

## II. DATA STRUCTURES AND ALGORITHMS

### A. Linear List Approach

A linear list is the most simple way to implement an archive. But if we want to maintain the archive of Pareto-points as a linear list, we have to compare every decision vector in the population with every decision vector in archive in worst case. In SPEA, Pareto-points (objective vectors) are stored in an array. In this archive, if a candidate is not dominated and if it also doesn't dominate any vector in the archive, it is added to the end of the archive. On the other hand, if the new vector $\vec{x}$ is dominated by another vector $\vec{y}$ in archive, then $\vec{x}$ is rejected. If $\vec{x}$ dominates $\vec{y}$, then all such $\vec{y}$s are discarded. Hence, deletion is easy whereas insertion is costly.

```
SPEA Algorithm
BEGIN
  Step 1: Initialization: Generate an
    initial population and initial empty external
    set (archive).
  Step 2: Update of archive: Copy all
    individuals whose decision vectors are
    non-dominated to archive (These individuals
    are from population and archive, if it is
    not empty). Remove individuals from archive
    which are dominated by the new individuals.
  Step 3: Fitness assignment: Calculate the
    strength of each individual in archive. Calculate
    the fitness of each individual in archive
    and population.
  Step 4: Selection: Select two individuals
    from population and archive at random.
    The individual with better fitness value is selected.
  Step 5: Recombination : ...
  Step 6: Mutation : ...
  Step 7: Termination : ...
END
```

Fig. 1. SPEA algorithm.

### B. Quad-tree Approach

A Quad-tree [4] is a tree based data structure for storing objective vectors. Each node is a vector with $m$ elements and can have a maximum of $2^m$ sons, which are defined by a successorship. The Quad-tree is a domination-free tree. This means that the nodes in this tree can not dominate each other. For inserting a new vector, there are always two questions to be answered:
1- Is the new vector dominated by a node in the Quad-tree?
2- Can any node in the Quad-tree be dominated by the new vector?
In order to explain the algorithm for insertion and replacement, some definitions are required:
**$k$-Successor:** A node $\vec{x}$ is called $k$-successor of node $\vec{y}$ where

$$k = \sum_{i=1}^{m} k_i 2^{m-i} \tag{1}$$

and the successorship is expressed by a binary string:

$$k_i = \begin{cases} 1 & \text{if } x_i \geq y_i \\ 0 & \text{if } x_i < y_i \end{cases} \tag{2}$$

**$k$-Son:** Node $\vec{x}$ is $k$-son of node $\vec{y}$, if $\vec{x}$ is a $k$-successor of $\vec{y}$ and also the direct son of $\vec{y}$.
**$k$-Set:** $S_i(k)$ is a set of $i$ in $k$-successors($i$ is 0 or 1):

$$S_0(k) = \{i | k_i = 0, k = (k_1, k_2, ..., k_m)_2\} \tag{3}$$
$$S_1(k) = \{i | k_i = 1, k = (k_1, k_2, ..., k_m)_2\} \tag{4}$$

Figure 2 shows an example of a simple Quad-tree for $m = 3$. The successorship of the son of the root is written by a binary string. For example, the vector $\vec{x}_1 = (5\ \ 5\ \ 23)$ has the successorship 001 to the root $\vec{y} = (10\ \ 10\ \ 10)$. It is hence called a 1-successor of $\vec{y}$. Vector $\vec{x}_2 = (15\ \ 11\ \ 5)$ is a 6-successor of $\vec{y}$.

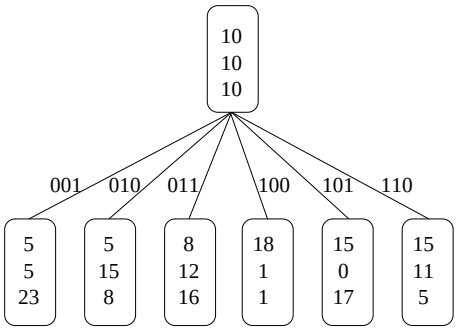


Fig. 2. A Quad-tree

Assuming without loss of generality, that all objectives in our MO optimization problems are to be minimized. In a domination-free Quad-tree, there exist no branches with the successorship 0 and $2^m$, because nodes with $k$ equal to 0 will by definition dominate the root and the nodes with $k$ equal to $2^m$ will always be dominated by the root.

When processing a vector for the possible inclusion in a domination-free Quad-tree, the vector is either discarded as being dominated by a vector already in the Quad-tree or it is inserted into the Quad-tree. However, when a vector is inserted, it may dominate other vectors in the Quad-tree and now these vectors must be deleted. By deleting a vector, we destroy the structure of subtree rooted at the deleted vector. This means that all the successors of the deleted vector must be again considered for inclusion in the Quad-tree. The corresponding algorithm is shown in Figure 3. Assume $\vec{x}$ is going to be inserted to the tree with root $\vec{y}$.

The main difference in the following variants of Quad-trees is the way in which vectors are processed for possible inclusion in the Quad-tree in order to maintain it domination-free. A criterion vector is admitted to a domination-free Quad-tree, if and only if, it is not dominated by any of the vectors already in the Quad-tree. Moreover, when admitted to the Quad-tree, all criterion vectors in the Quad-tree dominated by the new entry are identified and deleted. According to the above algorithm Quad-trees have two disadvantages: The first is that they are dependent on the order of inserting vectors into it. It means that if we want to make a Quad-tree from a list, we can have two different trees with the same list for different orders. The second disadvantage is that deleting a node means deleting all its subtrees and reinserting them again from the root. That will take a lot of time compared to a linear list. We therefore propose the following variant, Quad-tree2 in which the problem of deletion is improved.

```
Quad-tree1 Algorithm
BEGIN
  Step 1: let y⃗ be the root of tree.
  Step 2: calculate k such that x⃗ is
    k-successor of y⃗.
    If k = 2^m or if x_i = y_i, ∀i ∈ S_0(k),
      x⃗ is dominated by y⃗, STOP.
    If k = 0, delete y⃗, y⃗ is dominated by x⃗.
  Step 3: for all z⃗, such that z⃗ is a l-son of y⃗,
    l < k and S_0(k) ⊂ S_0(l):
    TEST1:
    check if x⃗ is dominated by z⃗ or the
    sons of z⃗.
    if x⃗ is dominated, STOP.
    else do TEST1 for the sons of z⃗.
  Step 4: for all z⃗, such that z⃗ is a l-son of y⃗,
    k < l and S_1(k) ⊂ S_1(l):
    TEST2 :
    check if x⃗ dominates z⃗ or the sons of z⃗.
    if z⃗ or one of its sons are dominated,
      delete them.
    else do TEST2 for the sons of z⃗.
  Step 5: if a k-son of y⃗ already exists,
      replace y⃗ by the k-son and goto step2.
    else x⃗ is the k-son of y⃗.
END
```

Fig. 3. Quad-tree1 Method.

In **Quad-tree2**, the dominated node is not deleted, but it is marked as deleted by a flag and its subtree is traversed, setting the flag for dominated nodes. So, the tree is not temporally domination-free, and must be cleaned after each insertion step. This algorithm makes the deletion process faster, because for each insertion we don't need to reinsert all the subtree and then repeat it for dominated vectors in the subtree. The algorithm of Quad-tree2 is shown in Figure 4.

In our third variant called **Quad-tree3** [6], the deletion problem is solved in another way, which is described in the DELETE routine. This algorithm is shown in Figure 5 and the routines are as below:

**DELETE($\vec{z}$):** This routine deletes the node $\vec{z}$. If the subtree is not empty, the lowest numbered son of $\vec{z}$ is assigned to be the new root of the subtree. So, the nodes in subtrees that are not in the lowest number subtree must be reinserted again. It is obvious that the lowest number son of a node can always be the root for other sons, it is at least better in first elements (objectives). For example, in Figure 2, the 1-son of the root, $(5\ \ 5\ \ 23)$, can also be the root for the other sons, because it is at least better in the first two elements than the other sons.

```
Quad-tree2 Algorithm
BEGIN
  Step 1: let y⃗ be the root of tree.
  Step 2: calculate k such that x⃗ is k-successor of y⃗.
    If k = 2^m or if x_i = y_i, ∀i ∈ S_0(k),
      x⃗ is dominated by y⃗, STOP.
    If k = 0, y⃗ must be deleted, y⃗ is dominated by x⃗:
      Mark y⃗ as deleted
      for all sons of y⃗, calculate k such that x⃗ is
       k-successor of y⃗ →son:
      If k = 0, Mark them as deleted
        clean all vectors that are marked deleted
         goto step1
  Step 3: for all z⃗, such that z⃗ is a l-son of y⃗,
       l < k and S_0(k) ⊂ S_0(l):
    TEST1: check if x⃗ is dominated by z⃗ or sons of z⃗.
      If x⃗ is dominated, STOP.
      else do TEST1 for the sons of z⃗.
  Step 4: for all z⃗, such that z⃗ is a l-son of y⃗,
      k < l and S_1(k) ⊂ S_1(l):
    TEST2: check if x⃗ dominates z⃗ or the sons of z⃗.
      If z⃗ or one of its sons are dominated,
        they must be deleted :
        Mark them as deleted.
        Do the same for all sons of z⃗
        If x⃗ dominates them, mark them as deleted
     clean all vectors that are marked deleted
  Step 5: if a k-son of y⃗ already exists,
      replace y⃗ by the k-son and goto step2.
      else x⃗ is the k-son of y⃗.
END
```

Fig. 4. Quad-tree2 Method.

**REPLACE($\vec{c}, \vec{s}$):** In this routine we replace $\vec{c}$ with $\vec{s}$, because $\vec{c}$ is dominated by $\vec{s}$. So the successorship in the subtree is not any more valid and they must be reconsidered again.

**REINSERT($\vec{c}, \vec{s}$):** This routine finds the right position in the subtree rooted at $\vec{c}$ at which to insert $\vec{s}$ and its successors. When reinserting vectors, those in the lowest level of the subtree rooted at $\vec{s}$ are processed first.

**RECONSIDER($\vec{c}, \vec{s}$):** The only difference between this routine and REINSERT is that in this routine the nodes in the subtree of $\vec{c}$, which must be reinserted from $\vec{s}$, may be dominated by $\vec{s}$.

So, first of all, $k$ must be calculated such that $\vec{s}$ is a $k$-successor of $\vec{c}$, then the vectors in $\vec{c}$ that are dominated by $\vec{s}$ must be discarded and the vectors in $\vec{s}$ that are dominated must be deleted. Then if $\vec{c}$ has a $k$-son, REINSERT($\vec{c}$ →son, $\vec{s}$) else move $\vec{s}$ to the position of $k$-son of $\vec{c}$ in the Quad-tree. The main idea of this algorithm comes from [6]. We have added this part to this routine in order to make it efficient for use within MOEA.

More details and Examples of these Quad-trees are explained in [5].

```
Quad-tree3 Algorithm
BEGIN
    Step 1: let y⃗ be the root of tree.
    Step 2: calculate k such that x⃗ is k-successor of y⃗.
      If k = 2^m or if x_i = y_i, ∀i ∈ S_0(k),
          x⃗ is dominated by y⃗, STOP.
      If k = 0, delete y⃗, y⃗ is dominated by x⃗.
         REPLACE(y⃗, x⃗)
         RECONSIDER(x⃗, y⃗ →son)
         reinsert all of the sons of y⃗ again from x⃗.
    Step 3: for all z⃗, such that z⃗ is a l-son of y⃗,
          l < k and S_0(k) ⊂ S_0(l):
      TEST1: check if x⃗ is dominated by z⃗ or sons of z⃗.
      if x⃗ is dominated, STOP.
      else do TEST1 for the sons of z⃗.
    Step 4: for all z⃗, such that z⃗ is a l-son of y⃗,
          k < l and S_1(k) ⊂ S_1(l) :
      TEST2 :check if x⃗ dominates z⃗ or the sons of z⃗.
      if z⃗ or one of its sons are dominated,
         if z⃗ is dominated by x⃗, DELETE(z⃗)
         check also all sons of z⃗ and if necessary
      else do TEST2 for the sons of z⃗.
    Step 5: if a k-son of y⃗ already exists,
       replace y⃗ by the k-son and goto step2.
      else x⃗ is the k-son of y⃗.
END
```

Fig. 5. Quad-tree3 Method.

## III. EXPERIMENTS

First, we explain how Quad-trees can be implemented within EAs, i.e. the SPEA algorithm [8]. Next we introduce the test functions used in the experiments and present CPU time comparisons for these data structures.

### A. Quad-trees In SPEA

We have integrated all the Quad-tree variants into the SPEA algorithm. Quad-trees are used as the data structure for the archive, not for the population. In each generation, each individual of the actual population is inserted into the tree. It is obvious that the only non-dominated vectors do remain in the Quad-tree, because the Quad-tree is a domination-free tree.

### B. Test Functions

There exist many test functions that are used for testing MOEAs, e.g. those introduced in [3] and [8]. They are built with consideration to difficulties in MO like converging to the Pareto-optimal front and maintaining diversity within population that may prevent MOEAs from finding Pareto-optimal solutions. We have also used these functions, called $\mathrm{TF}_i$, which are 2- and 3-objective problems. These functions are shown in Table I. In this table, $h$ and $g$ are defined as below:

minimize $f(\vec{x}) = (f_1(x_1), f_2(x))$

subject to
$f_2(\vec{x}) = g(x_2, \cdots, x_n) \cdot h(f_1(x_1), g(x_2, \cdots, x_n))$
where $\vec{x} = (x_1, \cdots, x_n)$.

TABLE I
TEST FUNCTIONS

| $TF_i$ | Function | $x_i$ |
|---|---|---|
| TF1 | $f_1(x_1) = x_1$<br>$g(x_2, \cdots, x_n) = 1 + 9(\sum_{i=2}^{n} x_i)/(n-1)$<br>$h(f_1, g) = 1 - \sqrt{f_1/g}$ | $n = 30$,<br>$x_i \in [0, 1]$ |
| TF2 | $f_1(x_1) = x_1$<br>$g(x_2, \cdots, x_n) = 1 + 9(\sum_{i=2}^{n} x_i)/(n-1)$<br>$h(f_1, g) = 1 - (f_1/g)^2$ | $n = 30$,<br>$x_i \in [0, 1]$ |
| TF3 | $f_1(x_1) = x_1$<br>$g(x_2, \cdots, x_n) = 1 + 9(\sum_{i=2}^{n} x_i)/(n-1)$<br>$h(f_1, g) = 1 - \sqrt{f_1/g} - (f_1/g)\sin(10\pi f_1)$ | $n = 30$,<br>$x_i \in [0, 1]$ |
| TF4 | $f_1(x_1) = x_1$<br>$g(x_2, \cdots, x_n) = 1 + 9(\sum_{i=2}^{n} x_i)/(n-1)$<br>$h(f_1, g) = 1 - (f_1/g)^2 - (f_1/g)\sin(10\pi f_1)$ | $n = 30$,<br>$x_i \in [0, 1]$ |
| TF5 | $f_1(x_1) = x_1$<br>$g(x_2, \cdots, x_n) =$<br>$1 + 10(n-1)\sum_{i=2}^{n}(x_i^2 - 10\cos(4\pi x_i))$<br>$h(f_1, g) = 1 - \sqrt{f_1/g} - (f_1/g)\sin(10\pi f_1)$ | $n = 10$,<br>$x_1 \in [0, 1]$,<br>$x_2, \cdots, x_n \in$<br>$[-5, 5]$ |
| TF6 | $f_1(x_1) = 1 + u(x_1)$<br>$g(x_2, \cdots, x_n) = \sum_{i=2}^{n} v(u(x_i))$<br>$h(f_1, g) = 1/f_1$ | $n = 11$,<br>$x_1 \in \{0, 1\}^{30}$,<br>$x_2, \cdots, x_n \in$<br>$\{0, 1\}^5$ |
| TF7 | $f_1(\vec{x}) = (x_1 - 1)^4 + (x_2 - 1)^2 + (x_3 - 1)^2$<br>$f_2(\vec{x}) = (x_1 + 1)^2 + (x_2 + 1)^4 + (x_3 + 1)^2$<br>$f_3(\vec{x}) = (x_1 - 1)^2 + (x_2 + 1)^2 + (x_3 - 1)^4$ | $n = 3$,<br>$x_i \in [-1, 1]$ |
| TF8 | $f_1(\vec{x}) = (1 + x_3)\cos x_1\pi/2 \cos x_2\pi/2$<br>$f_2(\vec{x}) = (1 + x_3)\cos x_1\pi/2 \sin x_2\pi/2$<br>$f_3(\vec{x}) = 3(1 + x_3)\sin x_1\pi/2$ | $n = 3$,<br>$x_i \in [0, 1]$ |
| TF9 | $f_1(\vec{x}) = (1 + x_3)(x_1^3 x_2^2 - 10x_1 - 4x_2)$<br>$f_2(\vec{x}) = (1 + x_3)(x_1^3 x_2^2 - 10x_1 + 4x_2)$<br>$f_3(\vec{x}) = 3(1 + x_3)x_1^2$ | $1 \le x_1 \le 3.5$,<br>$-2 \le x_2 \le 2$,<br>$0 \le x_3 \le 1$ |

## C. Parameter Setting

The linear list and Quad-tree implementations are used for studying six 2-objective and three 3-objective continuous functions (TF1, ..., TF9). Each simulation run was carried out using the following parameters:
Number of generations : 400
Population size : 100, 500, 1000, 5000, 10000
Number of generations : 30 (for 3-objective functions)
Population size : 100, 1000, 5000, 10000, 15000 (for 3-objective functions)
Cross over probability : 0.8
Mutation probability : 0.01

For more details see [5].

## D. Experimental Results

Figures 6-7 show the average CPU times of two different test functions in different runs and seeds for different populations sizes [1] for the linear list and the Quad-tree algorithms. In these Figures, Array is the symbol of the linear list implementation, Tree1, Tree2 and Tree3 are for SPEA with Quad-tree1, Quad-tree2, and Quad-tree3.

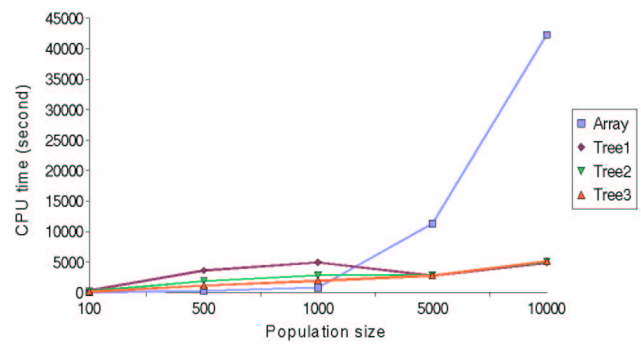


Fig. 6. Average CPU time of test function TF1 for different population sizes

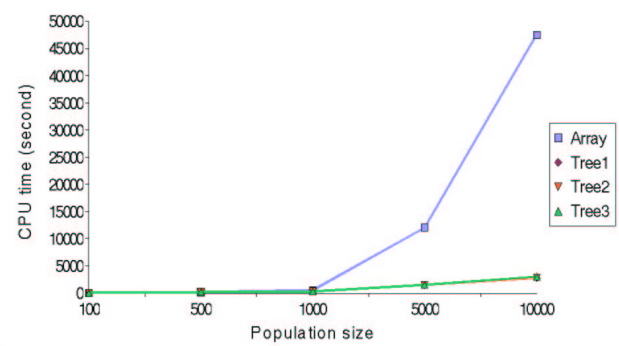


Fig. 7. Average CPU time of test function TF6 for different population sizes

The results of Average CPU time and the ratio of Quad-trees' CPU times to linear lists' CPU times ($T_i$/A), for each 2-objective test function and averaged over 4 different seeds are shown in Table II. According to the tables and graphs, depending on the problems and test functions, Quad-trees use less CPU times for population sizes of more than 2500 individuals for the convex test functions, and of more than 1000 individuals for the other test functions. Among the Quad-trees, Quad-tree1 requires longer CPU time for an equal number of iterations with

[1] We evaluate CPU times depending on the population size (not the archive size), because in each generation, each individual is (tentatively) inserted into the archive.

population sizes up to 5000 individuals. But for population sizes larger than 5000, the classical data structure Quad-tree1, performed the best. When comparing the linear list with the Quad-tree implementation, we see that for the 2-objective tests, Quad-tree3 has the best behavior, being almost 10 times better than linear lists for larger population sizes, but up to factor of 3 worse than linear lists for small population sizes where the linear list data structure with $O(N^2)$ complexity, where N is equal the population size, is the best choice.

TABLE II

AVERAGE CPU TIMES IN SECONDS FOR DIFFERENT POPULATION SIZES OF THE 6 2-OBJECTIVE FUNCTIONS ($T_i$/A IS THE RATIO OF TREE$_i$'S CPU TIME TO ARRAY'S CPU TIME)

| N | Array | Tree1 | Tree2 | Tree3 | T1/A | T2/A | T3/A |
|---|---|---|---|---|---|---|---|
| 100 | 30.68 | 332.9 | 232.83 | 118.99 | 10.85 | 7.59 | 3.88 |
| 500 | 294.71 | 3629.58 | 1859.02 | 1133.81 | 12.32 | 6.31 | 3.85 |
| 1000 | 818.54 | 4952.93 | 2840.45 | 1937.81 | 6.05 | 3.47 | 2.37 |
| 5000 | 11299.92 | 2774.41 | 2826.68 | 2794.71 | 0.25 | 0.25 | 0.25 |
| 10000 | 42274.91 | 4898.97 | 5100.49 | 5160.54 | 0.12 | 0.12 | 0.12 |
| 100 | 24.21 | 120.6 | 91.05 | 74.68 | 4.98 | 3.76 | 3.08 |
| 500 | 186.61 | 244.68 | 253.87 | 234.2 | 1.31 | 1.36 | 1.26 |
| 1000 | 537.58 | 418.46 | 446.45 | 456.18 | 0.78 | 0.83 | 0.85 |
| 5000 | 10328.27 | 2066.73 | 2218.92 | 2326.68 | 0.2 | 0.21 | 0.23 |
| 10000 | 40091.71 | 4142.21 | 4385.24 | 4517.08 | 0.1 | 0.11 | 0.11 |
| 100 | 31.82 | 370.47 | 234.36 | 126.56 | 11.64 | 7.37 | 3.98 |
| 500 | 318.48 | 4389.25 | 2515.66 | 1387.84 | 13.78 | 7.9 | 4.36 |
| 1000 | 916.46 | 9097.29 | 5283.9 | 3178.6 | 9.93 | 5.77 | 3.47 |
| 5000 | 12234.25 | 5383.5 | 4484.65 | 4300 | 0.44 | 0.37 | 0.35 |
| 10000 | 44055.57 | 6163.18 | 6212.26 | 6195.52 | 0.14 | 0.14 | 0.14 |
| 100 | 23.35 | 105.75 | 99.5 | 75.45 | 4.53 | 4.26 | 3.23 |
| 500 | 216.72 | 586.77 | 363.53 | 387.83 | 2.71 | 1.68 | 1.79 |
| 1000 | 626.88 | 745.08 | 640.92 | 580.44 | 1.19 | 1.02 | 0.93 |
| 5000 | 10524.19 | 2176.9 | 2336.12 | 2393.23 | 0.21 | 0.22 | 0.23 |
| 10000 | 40317.49 | 4355.43 | 4707.76 | 4655.86 | 0.11 | 0.12 | 0.12 |
| 100 | 20.2 | 41.66 | 44.25 | 45.8 | 2.06 | 2.19 | 2.27 |
| 500 | 176.19 | 207.11 | 216.82 | 228.72 | 1.18 | 1.23 | 1.3 |
| 1000 | 546.19 | 417 | 437.26 | 459.84 | 0.76 | 0.8 | 0.84 |
| 5000 | 10415.29 | 2104.2 | 2223.27 | 2375.99 | 0.2 | 0.21 | 0.23 |
| 10000 | 40410.43 | 4244.57 | 4454.04 | 4632.82 | 0.11 | 0.11 | 0.11 |
| 100 | 14.5 | 76.92 | 69.04 | 36.82 | 5.3 | 4.76 | 2.54 |
| 500 | 161.5 | 209.84 | 201.86 | 165.53 | 1.3 | 1.25 | 1.02 |
| 1000 | 555.29 | 345.7 | 342.95 | 324.13 | 0.62 | 0.62 | 0.58 |
| 5000 | 12090.32 | 1430.13 | 1451.23 | 1543.43 | 0.12 | 0.12 | 0.13 |
| 10000 | 47489.3 | 2728.1 | 2819.97 | 3038.54 | 0.06 | 0.06 | 0.06 |

Table III, shows the average CPU time and Ratio of CPU times of Quad-tree3 and linear list implementations on 3-objective test function TF8. For small population sizes, Quad-tree3 behaves more than 10 times slower than the linear list implementation. They would be faster for population sizes more than 12000 (in the average case 2 times faster).

## IV. CONCLUSIONS AND FUTURE WORK

We have studied and compared Quad-trees as a data structure for the archive in MOEAs with linear lists, which are used in SPEA. It has been shown that Quad-trees take less CPU time in comparison to linear lists for large population sizes. Hence, as a rule of thumb, we recommend the use of Quad-trees for optimal performance with higher-dimensional Pareto-Sets. This is often the case when optimizing continuous problems. For problems with discrete Pareto-fronts, the conventional linear list implementation might be the superior choice.

Also, we would like to investigate and compare data structures different from Quad-trees and linear lists, i.e. partial order graph. Finally, for each of the combined data structures, other types of operations such as fitness computation (e.g. Pareto-ranking) are currently investigated. Obviously, also this operation might influence the choice of the used data structure.

TABLE III

AVERAGE CPU TIMES IN SECONDS FOR DIFFERENT POPULATION SIZES OF THE 3-OBJECTIVE FUNCTION (TF8)(T/A IS THE RATIO OF TREE'S CPU TIME TO ARRAY'S CPU TIME)

| N | Array | Tree | T/A |
|---|---|---|---|
| 100 | 4.31 | 68.06 | 15.77 |
| 1000 | 340.44 | 3312.97 | 9.74 |
| 5000 | 8615.94 | 84137.38 | 9.76 |
| 10000 | 32095.86 | 307151.1 | 9.57 |
| 15000 | 2037222.4 | 1120085.8 | 0.549 |

## References


[1] D. Corne, M. Dorigo, and F. Glover. New Ideas in Optimization. Mc Graw Hill, 1999.

[2] K. Deb. Multi-Objective Optimization using Evolutionary Algorithms. *John Wiley & Sons*, 2001.

[3] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. Scalable test problems for evolutionary Multi-objective optimization. *Kangal report number 2001001, Kanpur Genetic Algorithms Laboratory(KanGal), Indian Institite of Technology Kanpur, India*, August 7, 2001.

[4] W. Habenicht. Quad Trees, a data structure for discrete vector optimization problems. *Lecture Notes in Economic and Mathematical Systems*, 209, Springer-Verlag, pages 136–145, Berlin, 1983.

[5] S. Mostaghim, J. Teich, and A. Tyagi. Comparison of Different Data Structures for Storing Pareto-points *Date report number 02, Electrical Engineering Laboratory (Datentechnik), Paderborn University, paderborn, Germany*, 2001.

[6] M. Sun and R.E. Steuer. Quad Trees and linear list for identifying nondominated criterion vectors. In *INFORM Journal on Computing*, Vol. 8, No. 4, pages 367–375, 1996.

[7] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. *TIK-report 103*, may, 2001.

[8] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and and the strength pareto approach. *IEEE Trans. on Evolutionary Computation*, 3(4):257–271, November 1999.