Utilizing Quad–Trees for Efficient Design Space Exploration with Partial Assignment Evaluation

Kai Neubauer, Christian Haubelt
University of Rostock, Germany
{kai.neubauer, christian.haubelt}@uni-rostock.de

Philipp Wanko, Torsten Schaub
University of Potsdam, Germany
{wanko, torsten}@cs.uni-potsdam.de

Abstract — Recently, it has been shown that constraint-based symbolic solving techniques offer an efficient way for deciding binding and routing options in order to obtain a feasible system level implementation. In combination with various background theories, a feasibility analysis of the resulting system may already be performed on partial solutions. That is, infeasible subsets of mapping and routing options can be pruned early in the decision process, which fastens the solving accordingly. However, allowing a proper design space exploration including multi-objective optimization also requires an efficient structure for storing and managing non-dominated solutions. In this work, we propose and study the usage of the *Quad–Tree* data structure in the context of partial assignment evaluation during system synthesis. Out experiments show that unnecessary dominance checks can be avoided, which indicates a preference of *Quad–Trees* over a commonly used list-based implementation for large combinatorial optimization problems.

I. Introduction

As the complexity of applications and hardware architectures is drastically increasing, system level descriptions of embedded many core systems have gained a lot of attention in the last decade. That is, the abstraction level is raised allowing a combined design of hardware and software components in order to diminish the complexity from lower abstraction such as transistor or block level. However, the increasing complexity of applications and the introduction of many core systems with a number of heterogeneous processing and communication elements leads to vast decision spaces when searching for feasible implementations. In recent studies, symbolic search techniques have been shown to be an efficient way for finding feasible system implementations even in stringently constrained environments [1, 2], i.e., mapping tasks onto processing resources and routing messages over the communication infrastructure. A huge advantage of symbolic solving is that not only complete but rather partial solutions where only a subset of decisions has been made can be checked for feasibility (e.g., [3]). That is, if a partial solution is already invalid with respect to some constraints, the search space can be pruned early in the decision process.

Nonetheless, there often exists no single optimal solution when multiple objectives (e.g., performance, predictability, energy efficiency, monetary costs, etc.) are considered in the evaluation of such systems. Rather, a set of Pareto-optimal solutions exist that represent compromises between variant and often conflicting objectives. In state-of-the-art frameworks, multi-objective optimization problems (MOOPs) are solved using meta heuristics like multi-objective evolutionary algorithms (MOEAs) and multi-objective particle swarm optimizations (MOPSOs). Those techniques are inspired by biological processes and work on sets of solutions (populations) concurrently. Each individual (i.e., potential solution) is evaluated by a fitness function before it is either discarded or considered as the basis for following populations. This process is repeated until a predefined abort criterion has been reached.

One of the main problems with these techniques is that the search is generally not executed systematically but based on combining previously found solutions. As a consequence, after an arbitrary number of iterations, these algorithms tend to run into saturation and stop finding novel feasible solutions.

Therefore, we propose using a complete symbolic approach for solving *multi-objective combinatorial optimization problems (MOCOPs)* as a subset of MOOPs with a Boolean parameter space (cf. [4]). It is based on *conflict-driven clause learning* (CDCL) which originates from the area of Boolean satisfiability (SAT) solving. CDCL-based techniques are able to directly leverage the concept of partial solutions for dominance checks of whole regions of the search space whenever the considered problem is assignment monotonous. That means, making an additional decision during solving must not lead to a better evaluation for any objective function. As this property holds true for most objective functions considered in system synthesis, we assume this property to also hold true for the problems considered in this paper.

Compared to MOEAs where only complete solutions are evaluated, working on partial assignments also implies that dominance checks have to be executed more regularly. This leads to a massive overhead in execution time especially for a high number of objectives and Pareto-optimal solutions. Consequently, an efficient way of performing *dominance checking* is imperative. To this end, we propose *Quad–Trees* as a data structure to store found non-dominated solutions in archives for an improved dominance check for partial solutions. The basic idea of Quad–Trees is that m-dimensional solutions are organized in a tree where each node represents a non-dominated solution and is itself the root of at most $2^m - 2$ children. This way, only a reduced number of vectors has to be compared in order to check for dominance of novel solutions. In the paper at hand, we show for the first time that Quad–Trees are especially well suited for dominance checking of partial assignments.

II. Related Work

In the following, we present a number of methods that have been developed to store and manage archives of non-dominated solutions.

The most simple technique is the utilization of linear lists. Consequently, in the worst case, each novel solution is compared to all other solutions before it can be added to or rejected from the archive. The most significant advantage of linear lists is that solutions of the current archive that are dominated by a novel solution can be removed from it very efficiently in $\mathcal{O}(1)$ without the need for adjustment of other solutions (e.g., reordering). The drawback, on the other hand, is that the complexity w.r.t. the number of necessary dominance checks is always $\mathcal{O}(N)$ where N is the length of the list.

To overcome this shortcoming, Habenicht [5] first proposed to use *Quad–Trees*, a k-ary tree with k depending on the number of objective functions. In this data structure, solutions are stored according to an ordered policy that allows faster dominance checks without the need to compare novel solutions against each existing element. However, deleting solutions from the tree whenever they are dominated by a new element is challenging. Therefore, Mostaghim and Teich [6] later introduced and compared three techniques to improve the performance. As Quad–Trees are described thoroughly in the following section, we omit further explanation here.

More recently, the authors of [7] proposed another approach for managing non-dominated solutions in MOEAs. The *M–Front* combines a list-based data structure with a k-dimensional binary search tree (k-d tree). Here, the solutions are stored into m sorted linked lists (M–List) and the k-d tree simultaneously where m is the number of objectives. The basic idea is to select one reference point for each novel solution and calculate a narrow area that contains a set of solutions for which the dominance checks have to be performed. To determine this area, they utilize geometric properties of the dominance relation to convert the problem into interval queries which can be answered using the M–List. An additional key requirement to minimize the number of solution that must be checked and thus the complexity of the algorithm is to find an appropriate reference point. In order to do so, the authors use an nearest neighbor search with the help of the k-d tree.

Jaszkiewicz and Lust [8] propose *ND–Tree*, a technique to online update Pareto archives. In the tree-based structure, each node represents a subset of solutions that are contained in a hypercube defined by its local nadir and optimal points (pointwise maximum/minimum of all objective functions). The actual solutions are stored in lists in the leaf nodes whereas internal nodes represent a union of all of their children storing the accumulated optimal and nadir points. That is, traversing the tree from the root to a leaf isolates potentially dominated candidates without the need to check each solution individually. Whenever a new solution is, for example, dominated by the combined nadir point, it can be discarded. On the other hand, if it dominates the ideal point, the whole sub tree is automatically dominated and can hence be deleted.

In this work, we address the problem of efficiently managing the Pareto archives while solving multi-objective combinatorial optimization problems (MOCOPs) (cf. [4] for an overview) based on CDCL and partial assignment checking. All previously mentioned methods have in common that they were developed in order to manage the archive for population based meta-heuristics and do not work directly for CDCL-based solving techniques with partial assignment checking. M–Front has the drawback that for every novel solution an appropriate reference point has to be calculated. That is, this calculation has to be executed for each partial assignments which would ultimately result in a high complexity.

In recent years, there have been few works that deal with deterministic methods for solving MOCOPs. The authors of [9] propose a method to solve such problems. However, they do not consider the ability of partial assignments. Though mentioning the possibility to utilize Quad–Trees for managing the archive as an outlook for future work, they only use linear list to store and update found non-dominated solutions as their focus lies on the solving process itself.

ASPRIN (*ASP* for *pr*eference handl*in*g) [10] is a framework for defining and computing preferred (optimal) solutions among stable models of logic programs. It offers a wide spectrum of different predefined preference types such as cardinality minimization as well as composite (i.e., multi-objective) preference types such as lexicographic and Pareto optimization. With *ASPRIN*, MOCOPs can be solved using the stable model semantics. However, no archive is used to store non-dominated solutions. Rather, constraints are added that exclude found and dominated solutions from the search space.

Note that the details of the solving process itself are out of scope of this paper and we refer to [11] and [12] for further information. Instead, we focus on the management of the Pareto archive that can be included into other frameworks directly.

III. Prerequisites

In this section, we will lay out basic definitions of multi-objective optimization as well as the fundamental idea and the basic operations of the Quad–Tree data structure.

A. Multi-objective Optimization

Given an electronic system specification containing various applications that have to be executed on complex heterogeneous processing architectures, the space of possible design decisions is enormous. While performing the design space exploration (DSE), the intertwined synthesis sub-problems allocation, binding, and scheduling are being solved. The resulting systems are characterized by quality indicators (i.e., objective functions) such as response time, energy consumption, and chip area. However, depending on the decisions that have been made in the individual synthesis steps, some objectives may be evaluated better than other objectives. The overall aim of the DSE is to find solutions that are optimal w.r.t. all objectives simultaneously. Generally, such multi-objective optimization problems (MOOPs) do not consist of a single optimal solution but rather a set of Pareto-optimal solutions (*Pareto front*).

Each found solution is evaluated by m objective functions corresponding to the criteria of the MOOP which results in an m-dimensional *fitness vector*. Comparing two solution vectors

x and y, there are three possible relations between them: x *is dominated by* y, x *dominates* y or x *is incomparable to* y. Formally, the dominance relation is defined as follows:

Def. 1. *Given two solution vectors* $x = \langle x_1, \ldots, x_m \rangle$ *and* $y = \langle y_1, \ldots, y_m \rangle$, x *dominates* y *if and only if* $\forall i = 1, \ldots, m : x_i \succeq y_i$ *and* $\exists i : x_i \succ y_i$ *with* m *representing the number of objectives and* $\succeq$ *as well as* $\succ$ *indicate "*better or equal*" and "*better*" relations, respectively.*

Given above definition, two solutions x and y are incomparable to each other if neither x dominates y nor y dominates x. Finally, Pareto-optimality is defined as follows:

Def. 2. *A solution* x *is said to be Pareto-optimal if it is not dominated by any other solution.*

Hence, all solutions located on the Pareto front are mutually non-dominated and all solutions not located on the front are dominated by at least one other solution.

Without loss of generality, we assume in the following minimization problems only.

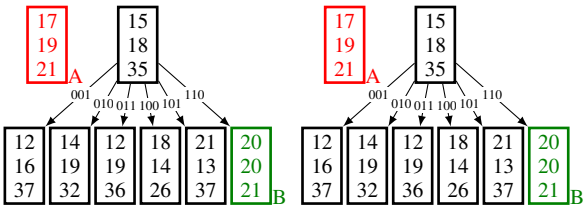
B. Quad–Tree data structure

When solving an MOOP, the true Pareto front is generally unknown. Therefore, newly found solutions are often inserted into a dominance free *archive*. This property is achieved by consequently deleting dominated solutions from the archive whenever a better solution is found during the search. To this end, the *Quad–Tree* data structure offers an efficient implementation as unnecessary comparisons can be avoided.

The basic structure (Fig. 1) of a Quad–Tree for an m-dimensional multi-objective optimization problem consists of a single *root* node that holds up to $2^m - 2$ child nodes. An m-bit index number, the *k-successor*, is associated with each child and is calculated as follows [5, 6]:

$$\forall i \in [0, m[: k_i = \begin{cases} 1, & \textit{if} \quad n[i] \geq r[i] \\ 0, & \textit{else.} \end{cases} \tag{1}$$

Here, n and r are m-dimensional vectors containing the fitness values for each objective and represent the child and the root, respectively and k_i represents the i-th bit of k. The *k-successor* expresses which objectives of a solution are better ($k_i = 0$) or worse ($k_i = 1$) w.r.t a reference vector and determines on which position a new solution must be inserted. For instance, node B in Fig. 1a is a 110_b*-successor* of the root $\langle 15, 18, 35 \rangle$ as its first two objectives evaluate worse ($k_0 = k_1 = 1$) and the third objective better ($k_2 = 0$). Note that no children with a *k-successor* equal to 0 and $2^m - 1$ exist as they would dominate or be dominated by the reference point, respectively. Informally, an m-dimensional solution vector x represents the origin of 2^m adjoining hyperboxes numbered from 0 to $2^m - 1$ that are occupied by different solution vectors. Note that all solutions located in hyperbox 0 ($2^m - 1$) are better (worse) than x for all objective values. Thus, solutions of these regions must not be saved as the archive would not be dominance free anymore. This results in a maximum of $2^m - 2$ children per solution. For example, assuming a two-dimensional solution vector x, hyperbox 0 located left below of x and hyperbox 3 located right above of x can not contain incomparable solutions.



(a) Original Quad–Tree (b) Updated Quad–Tree

Fig. 1. Example Quad–Tree with three objectives. The novel solution A (red) dominates and thus replaces solution B (green).

Using Quad–Trees, a new solution can be efficiently checked for non-dominance. Given above definitions, solution vectors in the Quad–Tree which *may* dominate a novel solution n are located in the sub trees of a root r whose indices contain zeros in the same location as n. To this end, for each k, the *k-sets* $S_0(k)$ and $S_1(k)$ specify all positions of zeros and ones, respectively and are defined as follows:

$$S_0(k) = \{i \mid k_i = 0, i = 1, \ldots, m\} \tag{2}$$

$$S_1(k) = \{i \mid k_i = 1, i = 1, \ldots, m\} \tag{3}$$

Formally, only *l-successors* of r with $l < k$ and $S_0(k) \subset S_0(l)$ have to be traversed. Consider for example the novel solution vector A in Fig. 1a. First, the *k-successor* is determined according to Eq. (1). The new solution may be dominated by children whose index l is smaller or equal to k and contains zeros at the same position as k. Hence, with $k = 110_b$, the novel solution may be dominated by the children with indices 010_b, 100_b, and 110_b.

Analogously, in order to check which solutions of the Quad–Tree are dominated by a new solution, only *l-successors* of r with $l > k$ and $S_1(k) \subset S_1(l)$ have to be traversed. For solution A in Fig. 1a with $k = 110_b$, only the sub tree with index $l = 110_b$ has to be checked.

In the example, A is not dominated by any previously inserted solution but it dominates vector $B = \langle 20, 20, 21 \rangle$. Note that the calculation of the *k-successor* as described by [6] is insufficient to test if a new vector dominates a solution in the tree. That is, A would be wrongly declared as 001_b*-successor* of B and hence be inserted in the sub tree of B at index 001_b although A clearly dominates B. Using this method, the archive would not be dominance free anymore. As a consequence, we propose to use an additional calculation, called *k*-successor*, to test if a new vector dominates an already found solution. The differentiation between k and k^* will be explained and discussed in the following section.

After identifying a dominated solution in the archive, it has to be deleted (Fig. 1b). However, deleting a solution from the Quad–Tree is not trivial in the general case. When deleting a node that itself contains child sub trees, those children have to be reinserted into the tree. The authors of [6] addressed this problem by proposing different strategies for updating the Quad–Tree and deleting dominated solutions in the context of multi-objective evolutionary algorithms (MOEAs). As their experiments show that processing the reinsertion in lower levels of the Quad–Tree leads to best results, we take that concept as a starting point for our approach detailed in the next section.

Algorithm 1: ISDOMINATED

Input: Partial solution fitness $\langle n_m \rangle$, Archive root $\langle r_m \rangle$
Output: Dominated by archive

1 $k = \texttt{KSUCC}(n, r)$
2 **if** $k = 0$ **then return** *False* // n dominates r
3 **elif** $k = 2^m - 1$ **then return** *True* // r dominates n
4 **else**
5 $\quad L = \langle l \rangle \mid l = [1, k] \wedge l \rightarrow k = 2^m - 1$
6 $\quad \forall l \in L$: **if** $\texttt{ISDOMINATED}(n, r.getChild(l))$ **then**
7 $\quad\quad$ **return** *True* // child dominates n
8 $\quad$ **return** *False*

IV. Quad–Trees for partial assignment evaluation

Compared to MOEAs, CDCL-based approaches can leverage partial assignment evaluation. That is, an incomplete solution (where not all decisions have been made) can be discarded if it is already dominated by a solution in the archive. In turn, this excludes all solutions containing the incomplete solution. That way, a large area of the search space can be pruned earlier during the solving process. Note that this approach is only feasible for assignment monotonous problems, i.e., an additional decision must not improve the evaluation of a partial solution. Thus, in the following, we will always assume this property.

As each partial solution has to be checked if it is dominated, the ratio of dominance checking to inserting (and hence deleting) grows with the number of decisions necessary to complete a solution. Consequently, the need for an efficient dominance check becomes apparent. However, as partial solutions are subject to deterioration throughout the solving, checking whether a novel solution dominates solutions from the archive *must* be delayed until the assignment is complete. As a consequence, the management strategy described in [6] cannot be used as it performs dominance checks for both direction (i.e., *dominates* and *is dominated*) per step. We therefore split the algorithm into two separate steps, namely the dominance check (Alg. 1) and the update (Alg. 2) steps described below.

A. Dominance Check

Algorithm 1 outlines the necessary steps to check if a partial assignment n is dominated by a vector in the archive. First, the *k-successor* for the current fitness w.r.t. the root node r is determined (line 1) which corresponds to Eq. (1). If k equals to 0, the partial assignment still dominates the root and the algorithm returns $False$ (line 2). This signals the solver that the solution is still feasible. Otherwise, if k equals to $2^m - 1$ (all bits are set to 1), the root of the tree already dominates the novel solution. Thus, the algorithm will return $True$ and the solver can exclude the partial solution and prune the search space accordingly. For every other value of k (lines 4 to 8), the children of r have to be tested. Therefore, first, the *l-successors* (line 5) are calculated which complies to the constraints described in the previous section and "$\rightarrow$" represents the bitwise imply-operator. Finally, the corresponding children l of r are tested recursively if they dominate n (line 6). This process is repeated until all decisions have been made and the solution is complete.

B. Update

At this point, it is clear that the new solution n is not dominated by any vector of the archive. Hence, we are able to prune such tests from the final update algorithm (Alg. 2) such that we only have to consider the cases where the novel solution n either dominates (lines 2 to 18) or is incomparable (lines 19 to 33) to the root r. Besides n and r, two Boolean values *insert* and *parentAlive* complete the input parameters of the algorithm. While *insert* determines whether the new solution n is supposed to be inserted into the sub tree of root r or not, *parentAlive* provides the information if one or more predecessor nodes of r were already dominated by n.

Algorithm 2: UPDATE

Input: New $\langle n_m \rangle$, Root $\langle r_m \rangle$, insert, parentAlive
Output: List of Solutions to be reinserted

1 $k^* = \texttt{K*SUCC}(n, r)$
2 **if** $k^* = 0$ **then** // r is dominated by n
3 $\quad L = \langle l \rangle \mid l \in [1, 2^m - 2]$
4 $\quad$ **foreach** $l \in L$ **do**
5 $\quad\quad reinsert \mathrel{+}= \texttt{UPDATE}$(*n, r.getChild(l), False, False*)
6 $\quad\quad r.removeChild(l)$ // Remove Child
7 $\quad$ **if** *parentAlive* $\wedge$ *insert* **then**
8 $\quad\quad r = n$
9 $\quad\quad \forall node \in reinsert$: $\texttt{INSERTSUB}(r, node)$
10 $\quad\quad$ **return** $\langle\rangle$ // Nothing to reinsert
11 $\quad$ **elif** *parentAlive* $\wedge \neg$*insert* **then**
12 $\quad\quad$ **if** $|reinsert| \geq 1$ **then**
13 $\quad\quad\quad r = reinsert.pop\,()$
14 $\quad\quad\quad \forall node \in reinsert$: $\texttt{INSERTSUB}(r, node)$
15 $\quad\quad$ **else** $\texttt{DELETE}(r)$
16 $\quad\quad$ **return** $\langle\rangle$ // Nothing to reinsert
17 $\quad$ **elif** $\neg$*parentAlive* $\wedge \neg$*insert* **then**
18 $\quad\quad$ **return** $reinsert$
19 **else** // r is incomparable to n
20 $\quad$ **if** *parentAlive* **then**
21 $\quad\quad$ **if** $r.hasChild(k^*)$ **then**
22 $\quad\quad\quad \texttt{UPDATE}$(*n, r.getChild(k*), insert, True*)
23 $\quad\quad$ **elif** *insert* **then**
24 $\quad\quad\quad r.addChild(n, k^*)$
25 $\quad\quad L = \langle l \rangle \mid l \in]k^*, 2^m - 2] \wedge k^* \rightarrow l = 2^m - 1$
26 $\quad\quad \forall l \in L$: $\texttt{UPDATE}$(*n, r.getChild(l), False, True*)
27 $\quad$ **else** // Ancestor is dominated
28 $\quad\quad L = \langle l \rangle \mid l \in [1, 2^m - 2]$
29 $\quad\quad$ **foreach** $l \in L$ **do** // Check all children
30 $\quad\quad\quad reinsert \mathrel{+}= \texttt{UPDATE}$(*n, r.getChild(l), False, False*)
31 $\quad\quad\quad r.removeChild(l)$ // Remove Child
32 $\quad\quad reinsert \mathrel{+}= r$
33 $\quad$ **return** $reinsert$

As indicated in the example for Fig. 1a, the original *k-successor* is unable to detect if a new solution dominates a vector from the archive in every case. Hence, for the update algorithm, we use the *k*-successor* (line 1) defined as follows:

$$\forall i \in [0, m[: k_i^* = \begin{cases} 1, & \textit{if} \quad n[i] > r[i] \\ 0, & \textit{else.} \end{cases} \tag{4}$$

Though the only difference between k and k^* are the "$\geq$" and "$>$" operators, respectively, the *k*-successor* correctly determines whether a vector A dominates another vector B if one or more objectives of A are smaller than the corresponding objectives of B and at least one objective is indifferent (cf. A and B in Fig. 1a). Formally, the *k*-successor* is necessary if there exist two complementary proper subsets I, J of all indices in the interval from 0 to $m - 1$ such that A is better in all $i \in I$

as well as A and B are indifferent in all $j \in J$ objectives:

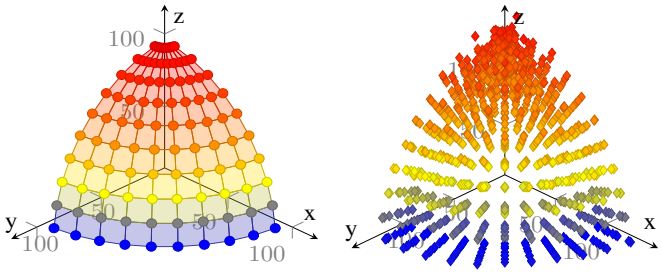
$$\exists I, J \subsetneq [0, m[\mid J = [0, m[\setminus I :$$
$$\forall i \in I : A[i] < B[i] \wedge \forall j \in J : A[j] = B[j].$$

If k^* evaluates to 0, i.e., r is dominated by n, all children of r have to be checked recursively whether they are also dominated by n and otherwise have to be marked for reinsertion into the Quad–Tree (lines 3 to 6). As n will replace r ($insert = True$) or will be inserted into another sub tree ($insert = False$), i.e., it will not be inserted into one of the children, and r is dominated, the recursive call of the update routine is parameterized with both $insert$ and $parentAlive$ equal to $False$ (line 5). Afterwards, $insert$ and $parentAlive$ are analyzed. If both parameters are true (lines 7 to 10), n replaces r and marked nodes are being reinserted into the sub tree with the new root $r = n$. Otherwise, if $parentAlive = True$ and $insert = False$, the first vector to be reinserted becomes the new root (line 13) and the remaining nodes will be reinsert there (line 14). However, if there are no nodes to be reinserted, r is simply deleted (line 15). Finally, if both $parentAlive$ and $insert$ are $False$, the list of vectors to be reinserted is returned (line 18). Note that the combination $parentAlive = False, insert = true$ cannot occur.

In case n is incomparable to r, i.e., $k^* \neq 0$, n has to be inserted in the sub tree k^* of r if $parentAlive = True$ (lines 21 to 26) or r and its sub trees have to marked for reinsertion if $parentAlive = False$ (lines 28 to 32). In the former case, `UPDATE` is called recursively with the parameters $parentAlive = True$ and $insert$ derived from the current context if there is already a solution at position k^* (lines 21-22). Otherwise, n is simply added at position k^* if $insert = True$. Subsequently, all sub trees of r whose position indices are greater than k^* and contain ones at the same position i as k^* are checked if they are dominated by n (lines 25-26). In the latter case ($ancesterAlive = False$), all children of r have to be checked if they are dominated by n and marked for reinsertion (including r itself). Finally, the nodes marked for reinsertion are returned to the parent of r.

C. Discussion on k and k*

Note that the *k*-successor* in Alg. 2 cannot detect whether A is dominated by B if all objectives are indifferent but one objective is worse in A as it is in B. However, as it is already known that the new solution is not dominated by a vector in the archive, it is unnecessary to detect this anyway. In general, the *k*-successor* evaluates a solution better (regarding the number of zeros) than the *k-successor*. As the insertion of new solutions is based on k^*, this leads to a mismatch between check (Alg. 1) and insertion (Alg. 2) if one or more objectives are indifferent to each other. Exemplary, consider the root node $r = \langle 5, 5, 5 \rangle$ and a new solution $n = \langle 6, 5, 4 \rangle$. With $k^* = 100_b$, both vectors are indifferent to each other and thus, n will be inserted as the child 100_b of r. Later, another solution $m = \langle 7, 5, 4 \rangle$ is found that is dominated by n. Even though the check is based on the *k-successor* which evaluates m w.r.t. r to $k = 110_b$, it detects the dominance by also searching *l-successors* that include 100_b. That is, the difference between k and k^* does not create problems in the detection of dominated solutions but is necessary for their identification as shown in the example in section III.B.



(a) Pareto front (b) All solution vectors

(c) Intermediate steps

Fig. 2. Experimental setup for three objectives showing (a) the Pareto front, (b) all solutions and (c) the intermediate steps for one solution

V. Experimental Evaluation

In this section, we evaluate the proposed Quad–Tree implementation for partial assignment checking against a list-based approach. Therefore, we compare the scalability by generating solutions that are split into a fixed number of intermediate steps in order to reflect the ability for partial assignment checking of CDCL-based solving. All tests were carried out on an Intel Core i7-4770 with 32 GiB RAM and prototypically implemented in *Python 2.7*.

A. Scalability

For testing the scalability of our approach, we simulate a design space exploration for various complex problems by creating a set of different solutions that are evaluated by arbitrary objective functions. More precisely, we first created an m-dimensional spherical Pareto front that consists of a varying number of mutually non-dominated points (Fig. 2a). Second, along the trajectory from the coordinate origin to each of the points, we randomly calculated a specific number of dominated solutions x_i (Fig. 2b) that consist of a fixed length list (`hops`) of intermediate valuations (i.e $x_i = \langle x_{i1}, \ldots, x_{ihops} \rangle$) to simulate partial assignments (Fig. 2c). Finally, each solution x_i is inserted into the archive. That is, the partial solutions are checked if they are non-dominated w.r.t. all solutions in the archive. If x_{ij} is already dominated, x_i must not be inserted.

In our experiments, we generated several instances with two to five objectives resulting in a varying number of non-dominated solutions and inserted them into a Quad–Tree and a list-based archive, respectively. Furthermore, we varied the number of intermediate solutions from 50 to 200 to achieve a wide range of granularity for the partial assignment evaluation. For each test case, 20 independent runs were performed.

Fig. 3 shows two important results. First, the bar chart depicts the overall number of comparisons carried out that are needed to filter the Pareto-optimal solutions. Here, one comparison corresponds to one calculation of k (k^*) in the Quad–Tree and one dominance check for the list-based implementation. The results indicate a significant advantage of the Quad–Tree data structure (blue) over the list (orange) with one order of magnitude difference in most of the test cases. Second, the average time in seconds needed to filter the Pareto

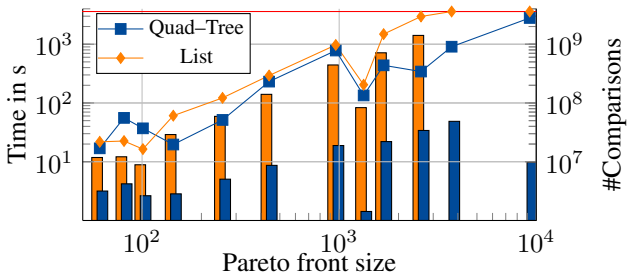


Fig. 3. Run times and number of comparisons for our experiments of various numbers of non-dominated solutions

front from the complete set of solutions is represented by the line chart. For any test with more than two objectives (all but the first three), the Quad–Tree implementation outperforms the list. Note that the list even timed out for the last two test cases (timeout set to 3600 s). However, the Quad–Trees are not able to keep the same advantage as indicated by the number of comparisons. The main reason for this is, that the update and deletion procedures are much more complex than for linear list where deleting a particular solution always takes constant time. Nevertheless, especially test cases with numerous mutually non-dominated solutions benefit from Quad–Trees as the dominance checks outnumber the deletion procedures.

B. Ordering

In the previous set of experiments, the order in which the solutions are inserted into the archive were random. Now, we first consider an ordered (solutions near the Pareto front are inserted first) and second an inversely ordered insertion strategy. That is, solutions that are already in the archive are less or more often dominated by new solutions, respectively.

An ordered inserting of the solutions results in a significant decrease of delete operations as newly found solutions are already dominated by the archive. Thus, the Quad–Tree data structure loses its disadvantage over the list. While the relation w.r.t. number of comparisons is nearly identical to the random test cases, the run time for Pareto filtering decreases for the Quad–Trees especially for large archive sizes. Considering a archive size of approx. 2500 solutions, the Quad–Tree implementation requires only 400 s while the list-based method runs 3250 s. For the largest considered test case (8412 solutions), the list times out while the Quad–Tree finishes in under 550 s.

For the other extreme, the inversely ordered insertion strategy, the execution times of both approaches are generally higher and also closer together as solutions from the archive must be deleted more regularly. However, even here, the Quad–Tree outperforms the list in most test cases. Only small two-dimensional optimization problems are filtered by the list (19 s) slightly faster than by the Quad–Tree (31 s). Similar to random test cases, the Quad–Tree is two times faster than the list for large test cases with three and more objectives.

VI. Conclusion

In this work, we proposed and studied Quad–Trees as the data structure to store and manage the Pareto archive for CDCL-based multi-objective optimizations with partial assignment evaluation. In contrast to multi-objective evolutionary algorithms (MOEAs), dominance checks must be additionally performed for incomplete solutions. We have shown that Quad–Trees therefore offer a fast dominance identification for newly found solutions as unnecessary comparisons can be avoided. Though deleting dominated solutions from the archive is arduous in Quad–Trees, the experiments showed that the state-of-the-art list-based implementation performs worse than the Quad–Tree due to a significantly higher number of comparisons for each test case with more than two objectives.

Acknowledgments

This work was funded by the German Science Foundation (DFG) under grants HA 4463/4-1 and SCHA 550/11-1.

References

[1] B. Andres, M. Gebser, T. Schaub, C. Haubelt, F. Reimann, and M. Glaß. Symbolic system synthesis using answer set programming. In *Proc. of LPNMR*, pages 79–91, 2013.

[2] A. Biewer, J. Gladigau, and C. Haubelt. A novel model for system-level decision making with combined ASP and SMT solving. *Proc. of DATE 2014*, pages 1–4, 2014.

[3] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt. Enhancing symbolic system synthesis through ASPmT and partial assignment evaluation. In *Proc. of DATE*, pages 306–309, 2017.

[4] C. A. Coello Coello, C. Dhaenens, and L. Jourdan. Multi-objective combinatorial optimization: Problematic and context. In *Advances in multi-objective nature inspired computing*, pages 1–21. Springer, 2010.

[5] W. Habenicht. *Essays and Surveys on Multiple Criteria Decision Making*, chapter Quad Trees, a Datastructure for Discrete Vector Optimization Problems, pages 136–145. Springer Berlin Heidelberg, 1983.

[6] S. Mostaghim and J. Teich. *Evolutionary Multiobjective Optimization*, chapter Quad-trees: A Data Structure for Storing Pareto Sets in Multiobjective Evolutionary Algorithms with Elitism, pages 81–104. Springer London, 2005.

[7] M. Drozdík, Y. Akimoto, H. Aguirre, and K. Tanaka. Computational cost reduction of nondominated sorting using the m-front. *IEEE Transactions on Evolutionary Computation*, 19(5):659–678, Oct 2015.

[8] A. Jaszkiewicz and T. Lust. Nd-tree: a fast online algorithm for updating a pareto archive and its application in many-objective pareto local search. *arXiv preprint arXiv:1603.04798*, 2016.

[9] R. Marinescu. Exploiting problem decomposition in multi-objective constraint optimization. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming*, pages 592–607, Springer Berlin Heidelberg, 2009.

[10] G. Brewka, J. Delgrande, J. Romero, and T. Schaub. asprin: Customizing answer set preferences without a headache. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[11] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52 – 89, 2012.

[12] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. Theory solving made easy with clingo 5. In *Technical Communications of ICLP*, 2016.