A Support-Based Algorithm for the Bi-Objective Pareto Constraint

Renaud Hartert[*] and **Pierre Schaus**
UCLouvain, ICTEAM,
Place Sainte Barbe 2,
1348 Louvain-la-Neuve, Belgium
{renaud.hartert, pierre.schaus}@uclouvain.be

Abstract

Bi-Objective Combinatorial Optimization problems are ubiquitous in real-world applications and designing approaches to solve them efficiently is an important research area of Artificial Intelligence. In Constraint Programming, the recently introduced bi-objective `Pareto` constraint allows one to solve bi-objective combinatorial optimization problems exactly. Using this constraint, every non-dominated solution is collected in a single tree-search while pruning sub-trees that cannot lead to a non-dominated solution. This paper introduces a simpler and more efficient filtering algorithm for the bi-objective `Pareto` constraint. The efficiency of this algorithm is experimentally confirmed on classical bi-objective benchmarks.

Bi-Objective Combinatorial Optimization (BOCO) aims at optimizing two objective functions simultaneously. Since these objective functions are often conflicting, there is usually no perfect solution that is optimal for both objectives at the same time. In this context, decision makers are looking for all the "best compromises" between the objectives to choose a posteriori the solution that best fits their needs. Hence, the notion of optimal solution is replaced by the notion of efficiency and we are searching for the set of all the efficient solutions (usually called *efficient set* or *Pareto frontier*) instead of one single solution (Ehrgott 2005).

During the past years, many approaches were developed to tackle BOCO problems exactly. However, many of them were developed in the context of Mathematical Programming (Mavrotas 2007; Ralphs, Saltzman, and Wiecek 2006) and only a few can be applied efficiently in Constraint Programming. Among these approaches, the ϵ-constraint is probably the most widely used (Haimes, Lasdon, and Wismer 1971; Le Pape et al. 1994; Van Wassenhove and Gelders 1980). The idea is to decompose the original problem into a sequence of subproblems to optimize with regard to the first objective function. At each iteration, a new subproblem is generated by constraining the second objective to take a better value than its value in the optimal solution of the previously solved subproblem. Notice that the number of single-objective problems to solve is linear in the efficient set's cardinality (Haimes, Lasdon, and Wismer 1971).

The bi-objective `Pareto` constraint is an alternative (and more efficient (Gavanelli 2002)) approach to compute the efficient set exactly. The idea behind this constraint is to build an approximation of the efficient set incrementally during the search and to use this approximation to detect and to prune sub-trees that can only lead to solutions that are less efficient than the ones already contained in the approximation. Eventually, the approximation becomes the efficient set and its optimality is proven when the search is completed.

The algorithm of the `Pareto` constraint relies on two operations. The first operation is used to update the approximation (by inserting new solutions in it) while the second consists to use this approximation to reduce the search space of the problem. In this work, we show how to use specific BOCO properties to improve the efficiency of the bi-objective `Pareto` constraint. Precisely, we show that both operations (update and filtering) can benefit from each other in an iterative way to build a simpler and more efficient algorithm for the constraint.

This document is structured as follows. First, we briefly introduce constraint programming and its main concepts. Then, we formalize multi-objective combinatorial optimization in the context of constraint programming and present some important definitions. The third section is dedicated to the `Pareto` constraint in its general multi-objective form. Our support-based algorithm is presented in the fourth section. Section five directly follows with our experiments and results on two classical benchmarks i.e. the bi-objective knapsack problem and the bi-objective travelling salesman problem. Finally, the last section offers some conclusions and perspectives.

Constraint Programming Background

Constraint Programming is a powerful paradigm to solve constraint satisfaction problems and combinatorial optimization problems. A constraint programming problem is usually defined by a set of variables with their respective domain (i.e. the set of values that can be assigned to a variable), and a set of constraints on these variables. The objective is to find an assignment of the variables that respects all the constraints of the problem. The constraint programming process interleaves a tree-search exploration (common in ar-

[*]This work was (partially) supported by the ARC grant 13/18-054 from Communauté française de Belgique.

tificial intelligence) with an inference procedure (also called propagation or filtering) to remove values that cannot appear in a solution. The inference part prunes the branches of the search-tree and thus reduces the search space to explore.

In an optimization context, an objective function can also be considered. Usually, this is formulated by the addition of an objective variable having as domain the set of possible values that can be taken by the objective function. Each time a solution is found, a constraint is added to the problem dynamically to force the objective variable to be assigned to a better value.[1] The optimality is proven when the search is exhausted, the last solution being the optimal one.

Multi-Objective Combinatorial Optimization

A *Multi-Objective Combinatorial Optimization* (MOCO) problem is a quadruple $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \mathbf{F} \rangle$ where $\mathbf{X} = \{x_1, \dots, x_n\}$ is a set of variables, $\mathbf{D} = \{D_1, \dots, D_n\}$ is a set of domains, $\mathbf{C}$ is a set of constraints on the variables in $\mathbf{X}$, and $\mathbf{F} = \{f_1, \dots, f_m\}$ is a set of objective functions to minimize simultaneously. Each objective function $f_i(\mathbf{X})$ associates a discrete cost to the assignment of the variables in $\mathbf{X}$. In the following, we assume that m objective variables $obj_1, \dots, obj_m$ have been added to the set of variables $\mathbf{X}$ and constrained to be equal to the value taken by their corresponding objective function i.e. $obj_i = f_i(\mathbf{X})$. In particular, the minimal and maximal values of the domain of an objective variable obj_i are the bounds of this objective with regard to a given partial assignment of the variables in $\mathbf{X}$.

A *solution* of a MOCO problem $\mathcal{P}$ is a complete assignment of the variables in $\mathbf{X}$ that satisfies all the constraints in $\mathbf{C}$. In this work, we represent a solution *sol* by its associated objective vector $(sol_1, \dots, sol_m)$ where sol_i is the value assigned to the corresponding objective variables obj_i.

As it is not likely that a solution is simultaneously optimal for all objectives at the same time, we are interested in a way to define a partial order on the objective vector of the solutions. Among these orderings, the *weak Pareto dominance* is a popular choice (see Figure 1). It is defined as follows.

Definition 1 (Weak Pareto dominance). *Consider* $sol = (sol_1, \dots, sol_m)$ *and* $sol' = (sol'_1, \dots, sol'_m)$*, two solutions of a MOCO problem* $\mathcal{P}$*. We say that sol dominates sol', denoted* $sol \preceq sol'$*, if and only if:*

$$\forall i \in \{1, \dots, m\} : sol_i \leq sol'_i. \quad (1)$$

Let $sols(\mathcal{P})$ denote the set of all the solutions of a MOCO problem $\mathcal{P}$. A solution *sol* of $\mathcal{P}$ is *efficient* if and only if no solution sol' in $sols(\mathcal{P})$ dominates *sol*:

$$\nexists sol' \in sols(\mathcal{P}) : sol' \neq sol \wedge sol' \preceq sol. \quad (2)$$

In other words, a solution is efficient if it is impossible to improve the value of one objective without degrading the value of at least one other objective (i.e. an efficient solution is an optimal compromise between the objectives).

Solving a MOCO problem usually means finding the set of all the efficient solutions that is generally called the *efficient set* or *Pareto front* (the efficient set of an arbitrary solution space is depicted in Figure 2). Unfortunately, the size of the efficient set often grows exponentially with the size of the problem to solve (Ehrgott 2005). Hence, in practice, only an approximation of the efficient set can be found in a reasonable amount of time and of memory. We call such an approximation of the efficient set an *archive*.

[1]Notice that the search does not stop but continues after the addition of the constraint to the problem.

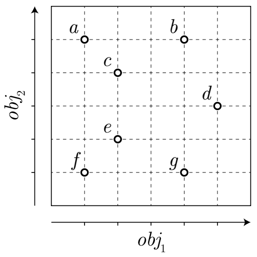


Figure 1: Solution e dominates the solutions b, c, and d while being dominated by solution f. Solutions a and g do not dominate solution e and are not dominated by solution e.

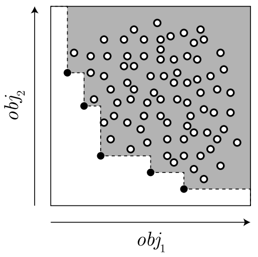


Figure 2: An arbitrary bi-objective solution space. *Efficient solutions* are colored in black. The *efficient set* is the set of all the efficient solutions.

Definition 2 (Archive). *An archive* $\mathcal{A}$ *is a domination-free set of solutions i.e. a set of solutions such that no solution in* $\mathcal{A}$ *dominates another solution in* $\mathcal{A}$*:*

$$\forall sol \in \mathcal{A}, \nexists sol' \in \mathcal{A} : sol' \neq sol \wedge sol' \preceq sol. \quad (3)$$

Clearly, the efficient set is also an archive.

The Pareto Constraint

The `Pareto` constraint (Schaus and Hartert 2013) is a global constraint defined over the m objective variables of a MOCO problem $\mathcal{P}$ and an archive $\mathcal{A}$:

$$\texttt{Pareto}(obj_1, \dots, obj_m, \mathcal{A}). \quad (4)$$

The aim of the `Pareto` constraint is to use the archive $\mathcal{A}$ to prune solutions that are dominated by at least one solution contained in the archive $\mathcal{A}$. In other words, the `Pareto` constraint ensures that a newly discovered solution is not dominated by any solution in the archive. This definition is formalized as follows:

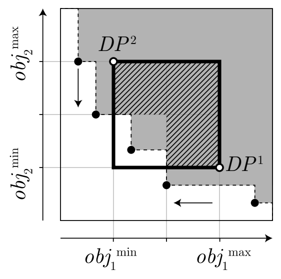
$$\bigwedge_{sol \in \mathcal{A}} \bigvee_{i=1}^{m} obj_i < sol_i \quad (5)$$

Filtering

The filtering rule of the `Pareto` constraint was originally proposed in (Gavanelli 2002). The idea is to use artificial objective vectors, called *dominated points*, to adjust the upper bounds of the objective variables in order to prevent the discovery of dominated solutions (5).

Definition 3 (Dominated point). *Let* $obj_i^{\min}$ *and* $obj_i^{\max}$ *denote the lower and upper bounds of the objective variable* obj_i*, the dominated point* DP^i *is defined as follows:*

$$DP^i = (DP_1^i, \dots, DP_m^i)$$
$$\text{where} \quad DP_j^i = \begin{cases} obj_j^{\max} & \text{if } j = i \\ obj_j^{\min} & \text{otherwise} \end{cases} \quad (6)$$



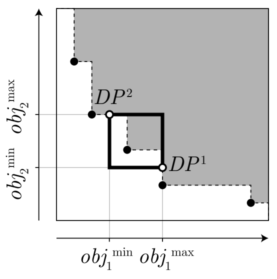


Figure 3: Filtering of the `Pareto` constraint. Black solutions correspond to the solutions contained in the archive $\mathcal{A}$. Grey areas represent the parts of the objective space that are dominated by at least one solution in $\mathcal{A}$. The dominated points DP^1 and DP^2 are represented by the white solutions. The hashed area corresponds to the reduction of the objective variables' domains after applying the filtering rule of Proposition 1.

In other words, the dominated point DP^i can be seen as an artificial solution for which each objective variable is assigned to its best possible value except for obj_i which is assigned to its worst possible value.

Proposition 1. *Let sol be a solution in the archive $\mathcal{A}$ such that sol dominates DP^i. Then, the value of the objective variable obj_i has to be smaller than the value sol_i:*

$$\exists sol \in \mathcal{A}, sol \preceq DP^i \;\Rightarrow\; obj_i < sol_i. \qquad (7)$$

Proof. Consider a solution sol in the archive $\mathcal{A}$ such that $sol \preceq DP^i$. According to (6), we know that $sol_j \leq obj_j^{\min}$ ($\forall j \in \{1, \ldots, m\}, j \neq i$). Hence, each newly discovered solution sol^{new} (i.e. a solution contained in the Cartesian product of the objective variables) such that $sol_i \leq sol_i^{\text{new}}$ is dominated by sol and can thus be safely removed. □

Figure 3 illustrates the domain of the objective variables before (left part) and after (right part) applying the filtering rule of Proposition 1.

If a dominated point DP^i is dominated by several solutions at the same time, the filtering rule of Proposition 1 has to be applied until no solution in the archive dominates DP^i. Clearly, the order in which the dominating solutions are selected affects the number of calls of the filtering rule. This situation is illustrated in Figure 4 where the worst possible selection order is a, b, and c. From this observation, it appears that the `Pareto` constraint can reach its fixed point in one step if it is able to access directly the solution that dominates DP^i with the lowest value in obj_i. Such a solution is called the *tightest solution* of obj_i.

Definition 4 (Tightest solution). *The tightest solution of obj_i is the solution in the archive $\mathcal{A}$ that dominates DP^i and that has the lowest value for obj_i:*

$$\text{argmin}_{sol \in \mathcal{A}} \{sol_i \mid sol \preceq DP^i\}. \qquad (8)$$

If DP^i is not dominated by at least one solution in $\mathcal{A}$, then, obj_i has no tightest solution.

Tightest solutions have been used in (Schaus and Hartert 2013) to propose the following idempotent filtering rule:

$$\exists sol \in \mathcal{A}, sol \preceq DP^i \;\Rightarrow\; obj_i < T_i^i \qquad (9)$$

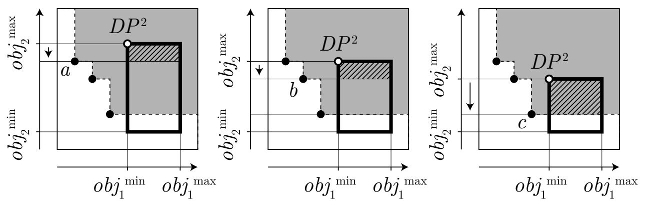


Figure 4: Worst possible selection order. Solution c is the tightest solution of obj_2.

where T^i is the tightest solution of obj_i.

Multi-Objective Branch-and-Bound

The `Pareto` constraint can be used to find the exact efficient set of any MOCO problem. The idea is to improve the quality of the archive $\mathcal{A}$ used by the constraint dynamically during the search. Initially, the archive is the empty set. Then, each time a new solution is discovered, it is inserted into the archive and the search continues. According to (5), every newly discovered solution is guaranteed to be dominated by none of the solutions in the archive. Hence, a newly discovered solution inserted in the archive improves its quality and consequently strengthens the filtering. Two situations are possible when inserting a new solution in the archive:

1. The new solution does not dominate any solution in the archive and nothing has to be done after the insertion (see Figure 5 left);
2. The new solution dominates one or several solutions in the archive. In this case, these dominated solutions have to be removed from the archive to ensure that the archive remains *domination-free* (see Figure 5 right).

In both cases, adding the new solution in the archive strictly increases the size of the subspace of the objective space that is dominated by the archive.[2] Eventually, the archive becomes the efficient set and its optimality is proven when the search is exhausted. The use of the `Pareto` constraint with a dynamically improving archive can be seen as a multi-objective branch-and-bound that generalizes the classical branch-and-bound algorithms used in constraint programming (Gavanelli 2002).[3]

Implementation and data structure

The multi-objective branch-and-bound algorithm based on the `Pareto` constraint relies on two distinct operations:

- `Access` the tightest solution of each objective to adjust the upper bound of the objective variables (9);
- `Insert` a newly discovered solution in the archive and remove potentially dominated solutions from the archive.

[2]The size of the subspace that is dominated by an archive is a common indicator, known as the *Hypervolume*, used to measure the quality of an archive (Zitzler et al. 2003).

[3]In single objective optimization, the archive is either the empty set or a singleton containing the best-so-far solution. The only dominated point DP^1 corresponds to the upper bound of the objective variable to minimize.

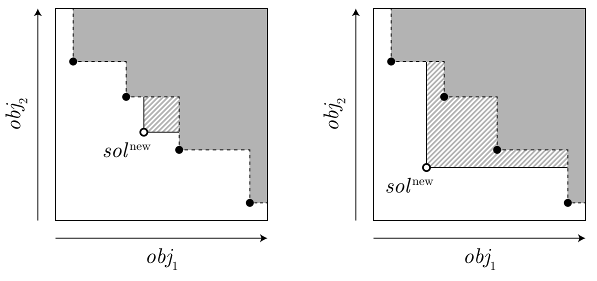


Figure 5: Insertion of a new solution in an archive. Hashed areas correspond to the additional parts of the objective space that are dominated after the insertion of the new solution in the archive.

Clearly, the efficiency of these operations is impacted by the underlying data structure used to implement the inner mechanisms of the constraint and to store its archive.

In (Gavanelli 2002), the author suggests the use of *point quad trees* (simply quad-trees in the sequel) to implement the archive. A quad-tree (Finkel and Bentley 1974; Habenicht 1983; Samet 2006) is a data structure that generalizes binary search trees to store m-dimensional vectors. Any node of a quad-tree divides its subspace into 2^m disjoint subspaces. More precisely, the root of the tree divides the space into 2^m subspaces originating at the root. The 2^m children of the root (exactly one for each subspace of the root) divides their subspaces into 2^m subspaces and so on (see Figure 6).

As for binary search trees, quad-trees are very sensitive to the order in which solutions are inserted. In the best case, when the tree is well-balanced, a quad-tree ensures a logarithmic complexity for the `access` operation. In the worst case however, the quad-tree is structured as a linear list and the `access` operation needs to traverse the entire data structure to find the tightest solutions. The removal operations are also a weakness of this data structure. Indeed, removing a solution in a quad-tree leads to the destruction of the subtree rooted at this solution and often leads to an expensive computational cost to repair it. Unfortunately, the `insert` operation may require several expensive removals to maintain the archive *domination-free* (i.e. to remove the solutions that are dominated by the inserted solution).

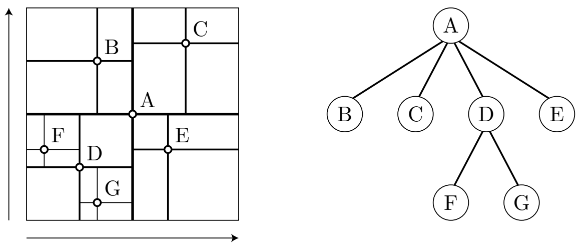


Figure 6: Illustration of the space partition (left) of a bi-dimensional quad-tree (right). The node A is the root of the quad-tree. Nodes B, C, D, and E are the sons of A. Nodes F and G are the sons of D.

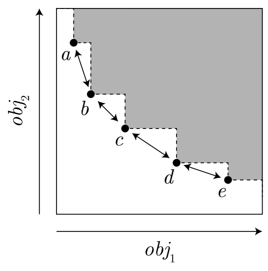


Figure 7: Illustration of an arbitrary bi-objective archive $\mathcal{A}$ stored in a bi-ordered linked-list. Each solution has a pointer to its direct successor in $\mathcal{A}^{>_1}$ and in $\mathcal{A}^{>_2}$.

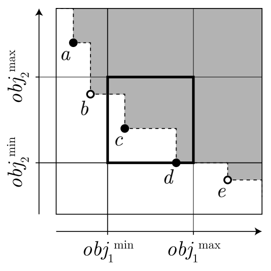


Figure 8: Solutions b and e are respectively the supports of obj_2 and obj_1. Both supports are not included in the Cartesian product of the objective variables.

Support-Based Algorithms

Bi-Objective Combinatorial Optimization (BOCO) problems are MOCO problems with only two objectives ($m = 2$). Usually, BOCO problems are easier to reason about and have properties that cannot be generalized to MOCO problems with more than two objectives. One of these properties is the *bi-objective ordering* property that is defined as follows.

Property 1 (bi-objective ordering). *Let $\mathcal{A}$ be an arbitrary bi-objective archive. We denote $\mathcal{A}^{>_i}$ (resp. $\mathcal{A}^{<_i}$) the archive $\mathcal{A}$ ordered by decreasing (resp. increasing) value of obj_i. Then, sorting the solutions of $\mathcal{A}$ by decreasing order of their value in a first objective (i.e. $\mathcal{A}^{>_1}$) amounts to sorting these solutions by increasing order of their value in the second objective (i.e. $\mathcal{A}^{<_2}$) and vice-versa.*

Proof. Let sol and sol' be two solutions in $\mathcal{A}$. Since $\mathcal{A}$ is *domination-free*, sol does not dominate sol' and sol' does not dominate sol. Hence, if $sol_1 < sol'_1$ we know that $sol_2 > sol'_2$. Symmetrically, if $sol_1 > sol'_1$ then $sol_2 < sol'_2$. □

In the remainder of this section, we introduce an incremental algorithm to implement the bi-objective `Pareto` constraint. The idea behind this algorithm is to use the bi-objective ordering property to store the archive $\mathcal{A}$ in a bi-ordered linked-list i.e. a linked-list such that each solution has a pointer to its direct successor in $\mathcal{A}^{>_1}$ and in $\mathcal{A}^{>_2}$ (see Figure 7). Precisely, the algorithm relies on special solutions, called *supports*, to maintain the tightest solutions of each objective (Definition 4) incrementally during the exploration of the search tree.

Definition 5 (Support). *Let sol be a solution in a bi-objective archive $\mathcal{A}$. We say that sol is the support of obj_1 if and only if:*

$$sol = \text{argmin}_{sol' \in \mathcal{A}} \{sol'_1 \mid sol'_2 < obj_2^{\min}\}. \qquad (10)$$

The support of obj_2 is defined symmetrically.

Figure 8 illustrates the supports of both objective variables obj_1 and obj_2.

Proposition 2. *Supports are never included in the Cartesian product of the domain of the objective variables i.e. supports cannot be dominated by newly discovered solutions.*

Proof. Let sol^{sup} be the support of obj_1 (resp. obj_2). By Definition 5, we know that $sol_2^{\text{sup}} < obj_2^{\min}$ (resp. $sol_1^{\text{sup}} < obj_1^{\min}$). Hence, sol^{sup} cannot be dominated by a newly discovered solution that is at best $(obj_1^{\min}, obj_2^{\min})$. □

Proposition 3. *If it exists, the tightest solution of obj_i is the support of obj_i or its direct successor in $\mathcal{A}^{>_i}$.*

Proof. Let DP_1 be dominated by a solution sol such that sol is the tightest solution of obj_1. If $sol_2 < obj_2^{\min}$, by Definition 5 we know that sol is the support of obj_1. Else, $sol_2 = obj_2^{\min}$ and sol is the direct successor of the support of obj_1. □

The link between supports and tightest solutions is illustrated in Figure 8 where d is the tightest solution of obj_1 and b is the tightest solution of obj_2.

Proposition 4. *Let $sol^{\text{new}} = (obj_1^{\min}, obj_2^{\min})$ be a newly discovered solution. If the archive is ordered according to the value of one objective (i.e. $\mathcal{A}^{>_1}$ or $\mathcal{A}^{>_2}$), then, sol^{new} dominates all the solutions contained between both supports.*

Proof. Let S_i denote the set of all the successors of the support of obj_i in the ordered archive $\mathcal{A}^{>_i}$. By Definition 5, we know that $S_i = \{sol \in \mathcal{A} \mid sol_i \geq obj_i^{\min}\}$. The intersection of S_1 and S_2 is thus the set of solutions that is dominated by $(obj_1^{\min}, obj_2^{\min})$:

$$S_1 \cap S_2 = \{sol \in \mathcal{A} \mid sol_1 \geq obj_1^{\min} \wedge sol_2 \geq obj_2^{\min}\}.$$

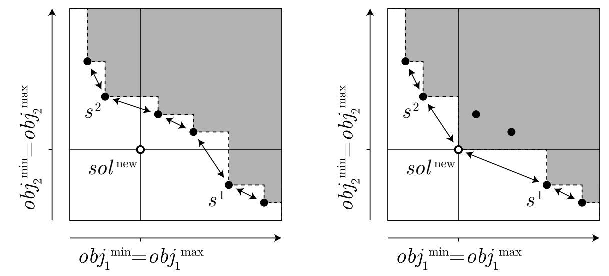
Since $sol^{\text{new}} = (obj_1^{\min}, obj_2^{\min})$, the solutions contained in the intersection of the successors of both supports are dominated by sol^{new}. □

The insertion of a new solution is illustrated in Figure 9. This operation is performed in constant time by updating the pointers of both supports.

We use Propositions 2, 3, and 4 to design an incremental algorithm for the bi-objective `Pareto` constraint. As mentioned above, the algorithm maintains the tightest solutions of each objective by adjusting the supports incrementally during the exploration of the search tree. More precisely, we describe the algorithm to adjust the upper bound of obj_1 as follows:[4]

1. Each time the lower bound of obj_2 is adjusted, we know that we have to reconsider the support of obj_1. To do so, we iterate on the direct successors of the old support in $\mathcal{A}^{>_1}$ until we reach the new support. Let Δ denote this number of iterations. Clearly, the sum of the Δ cannot exceed the size of $\mathcal{A}$ along a branch of the search tree.
2. When the new support is found, we `access` and use the tightest solution of obj_1 to adjust the upper bound of obj_1 (see Proposition 3 and (9)).

[4]The upper bound of obj_2 is adjusted symmetrically.



s^2
sol^{new}
s^1
$obj_2^{\min} = obj_2^{\max}$
$obj_1^{\min} = obj_1^{\max}$
s^2
sol^{new}
s^1
$obj_2^{\min} = obj_2^{\max}$
$obj_1^{\min} = obj_1^{\max}$

Figure 9: Insertion of a new solution in constant time. Solutions s^1 and s^2 are respectively the support of obj_1 and obj_2.

3. When a new solution sol^{new} is discovered, we `insert` it in $\mathcal{A}$ by simply updating the pointers of its supports (see Proposition 4).

Assuming a trail-based CP solver, *reversible pointers* can be used to maintain the supports.[5] Hence, each time a backtrack occurs, the algorithm is able to restore its previous supports in amortized constant time. Observe that, according to Proposition 2, we know that these supports are not dominated by a previously discovered solution.

Experiments and Results

This section presents the experimental evaluation of our support-based algorithm against the quad-tree algorithm commonly used to implement the bi-objective `Pareto` constraint. Our experiments used classical instances of the bi-objective binary knapsack problem (Xavier Gandibleux 2013) and instances of the bi-objective traveling salesman problem (Paquete and Stützle 2009). All algorithms were implemented in the open-source OscaR solver (OscaR Team 2012) that runs on the Java Virtual Machine using a computer running Mac Os X 10.9 on an Intel i7 2.6 Ghz processor.

First, we compare the number of search nodes explored using both algorithms on instances of the bi-objective binary knapsack problem within a time limit of 30 seconds. In this experiment, the bi-objective `Pareto` constraint starts with an empty archive and explores the search-space with a random heuristic. The usage of a random heuristic should favor the quad-tree, highly sensitive to the order in which solutions are inserted. Indeed, using a heuristic dedicated to one of the objectives would lead to an unbalanced quad-tree negatively impacting its efficiency. Table 1 gives the mean and the standard deviation of the number of nodes explored over 10 runs for each algorithm. We observe that our support-based algorithm is always the fastest despite the fair heuristic that we have chosen. It explores on average 20% more nodes.

Our second experiment compares both algorithms on instances of the bi-objective traveling salesman problem. In this experiments, the `Pareto` constraint starts with an initial archive that is a good approximation of the exact efficient set. In fact, the efficient set of these instance is currently

[5]This functionality can easily be adapted for copy-based CP Solvers.

instance	Quad-tree mean	st. dev.	Support-based mean	st. dev.
500A	635461.0	4953.8	**758585.8**	6567.3
500B	548185.3	6864.1	**653769.3**	7307.3
500C	531152.3	7631.5	**642077.7**	8158.2
500D	593904.0	4658.0	**712250.0**	5818.4

Table 1: Number of search nodes explored within a time limit of 30 seconds on instances of the bi-objective binary knapsack problem with 500 items.

instance	Quad-tree mean	st. dev.	Quad-tree (bal.) mean	st. dev.	Support-based mean	st. dev.
KroAB100	505.3	17.6	60034.0	2565.0	**63850.0**	227.3
KroAB150	234.3	9.4	23776.3	950.1	**25138.7**	339.8
KroAB200	361.0	1.6	18574.0	542.0	**21443.3**	143.5
KroAB300	247.3	17.9	9080.3	362.4	**9337.7**	176.4
KroAB400	176.0	0.4	4516.0	174.5	**4798.3**	70.4
KroAB500	116.0	1.6	3893.7	137.7	**4033.0**	107.3

Table 2: Number of search nodes explored within a time limit of 30 seconds on instances of the bi-objective travelling salesman problem with 100, 150, 200, 300, 400, and 500 cities.

unknown and the given set is the union of the approximation of many state-of-the-art algorithm to solve this problem (Lust and Teghem 2010). Since the internal structure of the `Pareto` constraint is initialized in advance, we consider both cases of an unbalanced and well-balanced quad-tree. As for the previous experiments, the mean and standard deviation over 10 runs are presented in the Table 2.

Again, the support-based is the fastest approach. As mentioned above, we observe that the quad-tree algorithm is very sensitive to the order in which solutions are added in its structure. On one hand, a well-balanced quad-tree is competitive with the support-based algorithm (but still always dominated). On the other hand, a poorly balanced quad-tree substantially deteriorates the number of search nodes explored. Interestingly, the balanced quad-tree and the support-based algorithms were able to add 3 solutions in the approximation of the KroAB300 efficient set and 4 solutions in the approximation of the KroAB500 efficient set. This experiment illustrates the flexibility of the `Pareto` constraint and one of its possible uses to improve an existing approximation of the efficient set or to prove its optimality.

Conclusion

This paper introduced a support-based algorithm to implement the bi-objective `Pareto` constraint. This incremental filtering algorithm relies on the bi-objective ordering property of BOCO problems. Experiments demonstrate that the support-based algorithm is more efficient than the classical algorithm used to implement the `Pareto` constraint. Also, this algorithm is simpler to implement than the quad-tree based propagator since it only relies on a linked list and two reversible pointers.

References

Ehrgott, M. 2005. *Multicriteria optimization*, volume 2. Springer Berlin.

Finkel, R. A., and Bentley, J. L. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4(1):1–9.

Gavanelli, M. 2002. An algorithm for multi-criteria optimization in CSPs. *ECAI'02* 136–140.

Habenicht, W. 1983. Quad trees, a datastructure for discrete vector optimization problems. In *Essays and Surveys on Multiple Criteria Decision Making*. Springer. 136–145.

Haimes, Y. Y.; Lasdon, L. S.; and Wismer, D. A. 1971. On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE Transactions on Systems, Man, and Cybernetics* 1(3):296–297.

Le Pape, C.; Couronné, P.; Vergamini, D.; and Gosselin, V. 1994. Time-versus-capacity compromises in project scheduling.

Lust, T., and Teghem, J. 2010. Two-phase Pareto local search for the biobjective traveling salesman problem. *Journal of Heuristics* 16(3):475–510.

Mavrotas, G. 2007. Generation of efficient solutions in multiobjective mathematical programming problems using GAMS. Effective implementation of the ε-constraint method.

OscaR Team. 2012. OscaR: Scala in OR. Available from https://bitbucket.org/oscarlib/oscar.

Paquete, L., and Stützle, T. 2009. Design and analysis of stochastic local search for the multiobjective traveling salesman problem. *Computers & operations research* 36(9):2619–2631.

Ralphs, T. K.; Saltzman, M. J.; and Wiecek, M. M. 2006. An improved algorithm for solving biobjective integer programs. *Annals of Operations Research* 147(1):43–70.

Samet, H. 2006. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.

Schaus, P., and Hartert, R. 2013. Multi-Objective Large Neighborhood Search. In *19th International Conference on Principles and Practice of Constraint Programming*.

Van Wassenhove, L. N., and Gelders, L. F. 1980. Solving a bicriterion scheduling problem. *European Journal of Operations Research* 4:42–48.

Xavier Gandibleux. 2013. A collection of test instances for multiobjective combinatorial optimization problems. Available from http://xgandibleux.free.fr/MOCOlib/.

Zitzler, E.; Thiele, L.; Laumanns, M.; Fonseca, C. M.; and da Fonseca, V. G. 2003. Performance assessment of multiobjective optimizers: An analysis and review. *Evolutionary Computation, IEEE Transactions on* 7(2):117–132.