How to Approximate Leximax-optimal Solutions

Miguel Cabral[1], Mikoláš Janota[2], and Vasco Manquinho[3]

[1] IST - Universidade de Lisboa, Portugal
miguel.cabral@tecnico.ulisboa.pt
[2] Czech Technical University Prague, Czechia
mikolas.janota@gmail.com
[3] INESC-ID, IST - Universidade de Lisboa, Portugal
vasco.manquinho@tecnico.ulisboa.pt

Abstract

Many real-world problems can be modelled as Multi-Objective Combinatorial Optimisation (MOCO) problems. In the multi-objective case, there is not a single optimum value but a set of optima known as Pareto-optima. However, the number of Pareto-optima can be too large to enumerate. Instead, one can compute a minimum Pareto-optimum according to an order. The leximax order selects a Pareto-optimum such that the objective functions with the worst values are penalised the least. Unlike other orders, such as the lexicographic order or the weighted sum order, the leximax order does not favour any objective function at the expense of others. Also, the leximax-optimum has a guaranteed small trade-off between the objective functions.

In some real-world MOCO problems, the time to find a solution may be limited and computing a leximax-optimal solution may take too long. In such problems, we search for solutions that can be computed in the given admissible amount of time and that are as close to the leximax-optimum as possible. In other words, we approximate the leximax-optimum.

In this paper, we present and evaluate SAT-based algorithms and heuristics for approximating the leximax-optimum of Multi-Objective Boolean Satisfiability problems. The evaluation is performed in the context of the package upgradeability problem, on the set of benchmarks from the Mancoosi International Solver Competition, with combinations of up to five different objective functions.

1 Introduction

In Boolean Satisfiability (SAT) and Pseudo-Boolean Satisfiability (PBS), the goal is to find an assignment to binary variables (assuming one of two possible values) such that a set of constraints is satisfied. In SAT, the constraints are clauses, whereas in PBS the constraints are linear inequalities. Maximum Satisfiability (MaxSAT) and Pseudo-Boolean Optimisation (PBO) are the optimisation versions of the SAT and PBS problems, respectively. In these problems, the goal is to find a satisfying assignment that minimises the value of the objective function. The objective function is a weighted sum of the variables. Combinatorial Optimisation is used to refer to any kind of problem where the goal is to find an assignment to variables that satisfies constraints and minimises the value of an objective function. The type of constraints and the domain of the variables is left unspecified. Thus, both MaxSAT and PBO are Combinatorial Optimisation problems. A satisfying assignment is also called feasible solution or just solution. A satisfying assignment that minimises the objective function is called optimal solution. Multi-Objective Combinatorial Optimisation (MOCO) is the extension of Combinatorial Optimisation, with multiple objective functions to minimise. The problem with having more than one objective function is that the objective functions may be conflicting. Decreasing one objective function may lead to an increase of another objective function. Despite this

trade-off, if, for a solution, there exists another solution that decreases the value of all objective functions, then the second solution should clearly be preferred. This leads to the well-known basic notion of Pareto-optimality. A Pareto-optimal solution is such that there does not exist another solution that decreases all objective functions at the same time.

When solving a MOCO problem, one possibility is to try to enumerate the set of Pareto-optima and leave it to an expert to choose one of those solutions. Another approach is to choose an order to compare objective vectors and then minimise according to that order. For example, there is the lexicographic order and the leximax order. The lexicographic-optimum corresponds to minimising the highest priority objective function, then the second highest priority objective function, and so on. For instance, the vector $(20, 50)$ is lexicographically better than the vector $(40, 30)$, assuming the first coordinate is the priority. On the other hand, the leximax-optimum corresponds to minimising the maximum of the objective functions, then the second maximum of the objective functions, and so on. For instance, the vector $(40, 30)$ is leximax-better than the vector $(20, 50)$, since the maximum 40 is smaller than the maximum 50. The lexicographic-optimum and the leximax-optimum are both a Pareto-optimum.

Because of prioritisation, the lexicographic-optimum will likely have a large trade-off between the objective functions. That is, some objective functions will have very small values and other objective functions will have very large values. On the other hand, the leximax-optimum tends to have a small trade-off between the objective functions, meaning that all the objective values are close.

The interest in the leximax order comes from the notion of *fairness* [8]. In some real-world problems, namely in ones involving human agents, it is necessary to find *fair* solutions. *Fairness* in this context means making "the worst-off as well-off as possible" [8], which means minimising the maximum of the objective functions. Thus, while the lexicographic order is suitable in problems with objective functions that should be prioritised, the leximax order is suitable in problems with objective functions that should be treated equally and *fairly*.

The package upgradeability problem emerges in the context of package management systems - tools that automate the installation, removal, and upgrade of software packages. Some examples of package managers are `apt-get`, used in the Debian Linux distribution, `dnf`, used in the RedHat Linux distribution, `pip`, used in the programming language Python, and `opam`, used in the programming language Ocaml. Every time a user makes a request to a package manager to install, remove or upgrade some software packages, the package manager needs to solve the package upgradeability problem: a set of packages to be installed, removed or upgraded must be found to meet the user request and at the same time satisfy all dependencies and conflicts between the installed packages. It is known that the package upgradeability decision problem is NP-complete [9]. However, real upgradeability instances can be solved extremely fast, by encoding the problem into a Propositional Satisfiability (SAT) formula and calling a SAT solver. Moreover, in some cases it is possible to lexicographically optimise several objective functions in a few seconds. Some examples of interesting objectives are: minimising the number of removed packages and minimising the number of not-up-to-date packages. In the Mancoosi European research project[1], a new generation of package upgradeability solvers was developed with the goal of finding lexicographically optimal solutions to the package upgradeability problem. These solvers relied on encodings to Answer Set Programming [12], Maximum Satisfiability [15], Pseudo Boolean Optimisation [2] and Integer Linear Programming [20, 18].

The leximax order has not been vastly explored in package upgradeability. To our best knowledge, there is only one package upgradeability solver that has implemented leximax opti-

[1] http://www.mancoosi.org

misation, the `mccs`[2] tool [20, 18]. Furthermore, in many benchmarks, these solvers are unable to find a lexicographically optimal solution in an admissible amount of time. The user can not wait for more than a few seconds for a solution. When there are time restrictions for solving a problem, as is the case of package upgradeability, the focus shifts from optimisation to finding a good enough approximation. When approximating an optimisation problem, the goal is to find solutions as close to the optimum as possible in a short amount of time. In MaxSAT, SAT-UNSAT search algorithms find the optimum by repeatedly obtaining a solution and then searching for another solution with a smaller objective value [17]. Hence, intermediate solutions of SAT-UNSAT algorithms can be used to approximate MaxSAT. It is well-known that MaxSAT can also be approximated by finding a Maximal Satisfiable Subset (MSS). Hence, one can enumerate multiple MSSes and output the best solution found in the allowed time interval. The MSS enumeration approach usually obtains better quality solutions than a MaxSAT SAT-UNSAT algorithm, when the computation time is very limited [19, 13].

In this paper, we propose two approaches to approximate the leximax optimum: MSS enumeration and a modified Guided Improvement Algorithm [14]. The Guided Improvement Algorithm is an algorithm for enumerating Pareto-optima. The modification that we propose forces the algorithm to seek leximax-better solutions while still obtaining guaranteed Pareto-optimal intermediate solutions. Moreover, we propose two heuristics in the search of a single MSS that can guide the search towards leximax-better solutions. Although we focus on the leximax optimisation of Boolean Satisfiability problems, the algorithms can be easily adapted to Pseudo-Boolean Satisfiability.

This paper is organised as follows. In Section 2 we briefly review the fundamental concepts and notation, as well as the definition of the package upgradeability problem. In Section 3 related work in multi-objective combinatorial optimisation is highlighted. In Section 4 we explain the proposed algorithms and heuristics for approximating the leximax-optimum. In Section 5 a detailed experimental evaluation is performed on a large set of benchmarks of the multi-objective package upgradeability problem. Finally, Section 6 concludes the paper.

2 Preliminaries

Standard definitions and notation of Boolean variable, literal, clause, conjunctive normal form (CNF) and the SAT problem are used [7]. The MaxSAT problem and its partial and weighted variants are defined in [19], as well as Minimal Correction Subsets (MCSes) and Maximal Satisfiable Subsets (MSSes). Pseudo-Boolean Satisfiability is defined in [22]. Some important notions in Multi-Objective Combinatorial Optimisation are defined next.

Given a set of constraints, Single-Objective Combinatorial Optimisation consists in finding a *feasible* solution (an assignment to the variables that satisfies the constraints) that minimises the value of the objective function. We often refer to a feasible solution as just a solution. A solution should not be confused with an optimal solution, which is a feasible solution that minimises the objective function.

Definition 2.1. Given an assignment α and objective functions $f_1, \ldots, f_n$, the vector

$$\vec{f}(\alpha) = (f_1(\alpha), \ldots, f_n(\alpha))$$

is called the objective vector. $f_i(\alpha)$ is the value of objective function f_i under α, for each $i = 1, \ldots, n$.

[2] https://www.i3s.unice.fr/~cpjm/misc/mccs.html

In Multi-Objective Combinatorial Optimisation we still want to obtain a feasible solution, but the notion of minimum of an objective vector is not immediate.

The notion of Pareto optimality captures the idea of universal optimality - regardless of the nature of the problem or whether some objective functions weigh more than others, a Pareto-optimal solution should always be preferred to a non-Pareto-optimal solution.

Definition 2.2 (Pareto-optimal)**.** Let $\vec{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$ and $\vec{b} = (b_1, \ldots, b_n) \in \mathbb{R}^n$. We write $\vec{a} \prec_{\text{Par}} \vec{b}$, if for all $i \in \{1, \ldots, n\}$, $a_i \leq b_i$ and there exists $j \in \{1, \ldots, n\}$ such that $a_j < b_j$. A feasible solution α is Pareto-optimal if there does not exist another feasible solution α' such that $\vec{f}(\alpha') \prec_{\text{Par}} \vec{f}(\alpha)$.

Example 2.1. Given two objective vectors $(20, 20, 20)$ and $(40, 40, 40)$, we have

$$(20, 20, 20) \prec_{\text{Par}} (40, 40, 40),$$

and we conclude that the second assignment is not Pareto-optimal. However, it is not the case that

$$(20, 20, 20) \prec_{\text{Par}} (10, 40, 40),$$

nor

$$(10, 40, 40) \prec_{\text{Par}} (20, 20, 20).$$

Definition 2.3 (Lexicographically-optimal)**.** Let $\vec{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$ and $\vec{b} = (b_1, \ldots, b_n) \in \mathbb{R}^n$. We define the lexicographic relation, $\prec_{\text{lexico}}$, as follows. We write $\vec{a} \prec_{\text{lexico}} \vec{b}$ whenever there exists $i \in \{1, \ldots, n\}$ such that $a_i < b_i$ and, for all $j \in \{1, \ldots, i-1\}$, $a_j = b_j$. A feasible solution α is lexicographically optimal if there does not exist a feasible solution α' such that $\vec{f}(\alpha') \prec_{\text{lexico}} \vec{f}(\alpha)$.

Example 2.2. We have that

$$(10, 40, 40) \prec_{\text{lexico}} (20, 20, 20),$$

since $10 < 20$.

Proposition 2.1. *Every lexicographically-optimal solution is Pareto-optimal [10].*

Definition 2.4 (Leximax-optimal)**.** Let $\vec{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$ and $\vec{b} = (b_1, \ldots, b_n) \in \mathbb{R}^n$. Let $\vec{a}^{\downarrow}$ denote the n-tuple with the elements of $\vec{a}$ sorted in decreasing order. We call the i-th component of $\vec{a}^{\downarrow}$ the i-th maximum of $\vec{a}$. The tuples $\vec{a}$ and $\vec{b}$ are leximax-indistinguishable if $\vec{a}^{\downarrow} = \vec{b}^{\downarrow}$. A tuple $\vec{a}$ is leximax-better than $\vec{b}$, written $\vec{a} \prec_{\text{leximax}} \vec{b}$, if $\vec{a}^{\downarrow} \prec_{\text{lexico}} \vec{b}^{\downarrow}$. A feasible solution α is leximax-optimal if there does not exist a feasible solution α' such that $\vec{f}(\alpha') \prec_{\text{leximax}} \vec{f}(\alpha)$.

Example 2.3. We have that

$$(20, 20, 20) \prec_{\text{leximax}} (10, 40, 40),$$

since the following holds for their sorted versions:

$$(20, 20, 20) \prec_{\text{lexico}} (40, 40, 10).$$

Proposition 2.2. *Every leximax-optimal solution is Pareto-optimal [10].*

In short, lexicographical minimisation is the sequential minimisation of the most important objective function to the least important objective function. Leximax minimisation is the sequential minimisation of the maxima of the objective vector.

Remark. Recall that, in MaxSAT, the goal is to maximise the sum of the weights of the satisfied soft clauses. On the other hand, objective functions are to be minimised. Hence, whenever we mention an objective function in the context of MaxSAT, we refer to the weighted sum of the *falsified* soft clauses.

Package Upgradeability Problem. This problem consists in: (1) a universe of packages with dependencies and conflicts; (2) a set of initially installed packages; (3) a user request to install, remove or upgrade some packages. Given a package p, a dependency of p is a set of packages and it imposes that if p is installed, then one of the packages must also be installed. A conflict is a binary relation between packages. Package p_1 and p_2 having a conflict means both packages can not be installed at the same time. The problem is easily modelled in SAT by taking a Boolean variable for each package p that is true if and only if p is installed.

3 Related Work

Leximax Optimisation. Some algorithms for obtaining leximax-optimal solutions have been developed in Constraint Programming [8]. One of the algorithms has been adapted to Integer Linear Programming and implemented in the package upgradeability tool `mccs`[3] [20, 18]. In Boolean and Pseudo-Boolean Satisfiability (SAT and PBS), no leximax optimisation algorithm requiring only SAT/PBS solvers is currently known.

Approximation. Recent work done by Ignatiev *et al.* [13] explores the computation of MSSes to approximate the lexicographic optimum. The work involved extending the package upgradeability tool `packup` [15] with MSS enumeration, motivated by the practical time restrictions of package upgradeability. Our work is similar, but with a focus on the leximax order.

The work of Terra-Neves *et al.* [21] is also related to approximation in Multi-Objective Combinatorial Optimisation and MSS enumeration. The goal of their work was to enumerate the set of Pareto-optimal solutions using MSS enumeration.

4 Contribution

In this Section we present (1) an adaptation of the well-known MSS linear search algorithm [3] to the multi-objective paradigm with heuristics to guide the search towards the leximax optimum and (2) a modification of the Guided Improvement Algorithm [14] to force that each time a solution is found, the objective vector is closer to the leximax optimum.

4.1 MSS Heuristics

The MSS linear search algorithm is perhaps the simplest MSS/MCS algorithm. The MSS and the MCS are initially empty, and are constructed as we iterate through all soft clauses. In each iteration we call a SAT solver to test if the soft clause can be added to the MSS. If it can, then the clause is added to the MSS, otherwise, the clause is added to the MCS.

[3] https://www.i3s.unice.fr/~cpjm/misc/mccs.html

Algorithm 1: MSS extended linear search with multiple objective functions

Input: Hard clauses $\mathcal{H}$ and multiple sets of soft clauses $\mathcal{F}_1 \dots \mathcal{F}_n$
Output: Assignment α. The set of satisfied clauses under α is an MSS of $(\mathcal{H}, \bigcup_{i=1}^{n} \mathcal{F}_i)$

```
1  S_1 ← H ... S_n ← H                                   // MSS per objective
2  T_1 ← F_1 ... T_n ← F_n                               // unprocessed clauses
3  while ⋃_{i=1}^{n} T_i ≠ ∅ do
4      i ← ChooseNextObjective(T_1, ..., T_n, S_1, ..., S_n)
5      c ← GetClause(T_i)
6      T_i ← T_i \ {c}
7      (st, α') ← SAT(⋃_{i=1}^{n} S_i ∪ {c})
8      if st then
9          α ← α'
10         S_i ← S_i ∪ {c}
11         UpdateSatisfiedClauses(α, T_1, ..., T_n, S_1, ..., S_n)
12 return α
```

The linear search algorithm's performance can be improved by reducing the number of calls to the SAT solver. Each time a solution is found, we can check if there are more satisfied soft clauses and add them to the MSS. Thus, in the *extended* [19] linear search algorithm, the MSS in construction can grow by more than one element in each iteration.

When there are multiple objective functions, i.e. multiple sets of soft clauses $\mathcal{F}_1, \dots, \mathcal{F}_n$, the subset of clauses that we obtain is an MSS of the single-objective problem with the following set of soft clauses $\bigcup_{i=1}^{n} \mathcal{F}_i$. Algorithm 1 shows the extended linear search algorithm with multiple objective functions. In the pseudocode, the SAT solver takes as input the set of clauses, and returns a pair (st, α'), where st is a Boolean: true if and only if the formula is satisfiable, and α' is the solution, if there is one. The sets T_i, $i \in \{1, \dots, n\}$, contain the clauses that have not been checked ('T' for 'TODO'). The sets S_i, $i \in \{1, \dots, n\}$, contain the clauses that are part of the MSS ('S' for 'Satisfiable'). The `UpdateSatisfiedClauses` routine removes clauses that α satisfies in T_i and puts them in S_i, $i \in \{1, \dots, n\}$. The `ChooseNextObjective` routine outputs the index $i \in \{1, \dots, n\}$ of the sets T_i, from which the next clause will be removed and tested for addition to the MSS. Our heuristics for approximating leximax address the implementation of these two routines. We address two questions:

1. Which objective function do we choose the next clause from? Suppose we start by always choosing the next clause from the same objective function, without adding satisfied clauses from other objective functions. We expect that, when the MSS search is completed, the chosen objective function's value will be quite small, which, in general, will impose a high lower bound on some of the remaining objective functions. Since the goal of leximax optimisation is to minimise the maxima of the objective vector, this scenario should be avoided.

2. Should we add all satisfied clauses to the MSS? The more clauses are added to the MSS, without having to call the SAT solver, the better the performance. However, note that the clauses that are added to the MSS impose an upper bound on the respective objective functions. By blindly adding all satisfied clauses to the MSS, we risk bounding an objective function too much.

We propose:

1. choosing the next clause from an objective function whose upper bound is the largest among all objective functions.

2. if the chosen clause can be satisfied and there are other satisfied clauses, add to the MSS as many satisfied clauses as possible while trying to even out all the upper bounds. For example, suppose we have three objective functions, and in a certain iteration the upper bounds are 1500, 1800 and 1600, and there are 500 satisfied clauses that can be added to the MSS, for each objective function. Instead of adding all clauses, resulting in the upper bounds 1000, 1300 and 1100, we add 200 clauses of the first objective function, all 500 clauses of the second objective function, and 300 clauses of the third objective function. The upper bounds become 1300, 1300 and 1300.

4.2 Guided Improvement Algorithm for Leximax

In a nutshell, the Guided Improvement Algorithm [14] for enumerating the set of Pareto-optima works by repeatedly finding a Pareto-optimal solution and then blocking its objective vector with a constraint. A single Pareto-optimal solution is computed using the following loop:

1. Compute a solution.
2. Get the objective vector $(k_1, \ldots, k_n)$.
3. Add constraints $f_1 \leq k_1, \ldots, f_n \leq k_n$ and $\bigvee_{i=1}^{n} f_i \leq k_i - 1$.
4. Repeat until the formula becomes unsatisfiable (Pareto optimality is proven).

The algorithm can be adapted to approximate the leximax optimum by essentially replacing the constraint $\bigvee_{i=1}^{n} f_i \leq k_i - 1$ by the constraint $f_j \leq k_j - 1$, where $k_j = \max(k_1, \ldots, k_n)$.

Algorithm 2: Guided Improvement Algorithm adapted to leximax

Input: Hard clauses $\mathcal{H}$, objective functions $f_1, \ldots, f_n$, initial satisfying assignment α
Output: Assignment α is successively updated with a leximax-better objective vector

```
1 i ← 1
2 while i ≤ n do
3 |   α ← FindParetoSol(H, f_1, ..., f_n, α, i)
4 |   (i, α) ← BlockOptimum(H, f_1, ..., f_n, α, i)
5 return α
```

Algorithm 2 shows the pseudocode of the Guided Improvement Algorithm adapted to the leximax order. It is composed of a while loop. In each iteration, as in the original Guided Improvement Algorithm, we start by finding a Pareto-optimal solution (function `FindParetoSol`). Then, that Pareto-optimum is blocked (function `BlockOptimum`), so as to find a different Pareto-optimum in the next iteration. Instead of finding random Pareto-optimal solutions, our approach is to find Pareto-optimal solutions such that each new intermediate solution is leximax-better than the previous one. As in the MSS linear search pseudocode, the SAT solver returns a pair (st, α'), where st is true if and only if the formula is satisfiable and α' is a solution, if there is one. Note that, in practice, the algorithm may stop if it reaches a timeout. In that case, the best solution so far is output.

Function `FindParetoSol`, shown in Algorithm 3, consists of a while loop. The loop terminates when Pareto-optimality of α is proven. $\vec{k} = (k_1, \ldots, k_n)$ denotes the objective vector of α. Recall that $\vec{k}^{\downarrow}$ denotes $\vec{k}$ sorted in decreasing order. The index i refers to the i-th maximum of the objective vector, i.e. $k_i^{\downarrow}$. The idea is to (1) fix $i-1$ objective functions with value greater

Algorithm 3: Function `FindParetoSol`

Input: Hard clauses $\mathcal{H}$, objective functions $f_1, \ldots, f_n$, assignment α and index i
Output: Pareto-optimal solution α

$(k_1, \ldots, k_n) \leftarrow (f_1(\alpha), \ldots, f_n(\alpha))$
while $i \leq n$ **do**
  $P \leftarrow \{p \in \{1, \ldots, n\} : k_p > k_i^{\downarrow}\}$
  $M \leftarrow \{m \in \{1, \ldots, n\} : k_m = k_i^{\downarrow}\}$
  while $|P| < i - 1$ **do**
    $m \leftarrow \texttt{SelectFrom}(M)$ `// get any element of` M
    $M \leftarrow M \setminus \{m\}$
    $P \leftarrow P \cup \{m\}$
  $R \leftarrow \{r \in \{1, \ldots, n\} : k_r < k_i^{\downarrow}\}$
  $(st, \alpha') \leftarrow \texttt{SAT}(\mathcal{H} \cup \bigcup_{m \in M}\{f_m \leq k_m - 1\} \cup \bigcup_{p \in P}\{f_p = k_p\}) \cup \bigcup_{r \in R}\{f_r \leq k_r\})$
  if st **then**
    $\alpha \leftarrow \alpha'$
    $(k_1, \ldots, k_n) \leftarrow (f_1(\alpha), \ldots, f_n(\alpha))$
  else
    $i \leftarrow i + 1$
return α

or equal to the i-th maximum (set P), (2) try to decrease the remaining objective functions equal to the i-th maximum (set M) and (3) bound all objective functions smaller than the i-th maximum (set R). Set P is named after 'Previous', as it corresponds to the objective functions equal to previous maxima (the current maximum being the i-th). These objective functions can no longer be decreased without increasing other objective functions. Set M is named after 'Maximum' as it contains the objective functions not in P equal to the i-th maximum. These objective functions may possibly be decreased (without increasing other objective functions). Set R is named after 'Remaining'. These constraints ensure that (1) when the formula is satisfiable, the new solution is leximax-better than the previous solution; and (2) if the formula is unsatisfiable and i is incremented to $n + 1$, the loop ends and α is Pareto-optimal.

In function `BlockOptimum`, shown in Algorithm 4, we simply remove the set R, responsible for bounding objective functions smaller than the current maximum. By removing the set R, we check if the i-th maximum can be decreased without bounding further the smaller objective functions, but still fixing the previous $i - 1$ maxima stored in P. If the formula is satisfiable, we get a leximax-better solution. Otherwise, we increment i. This process is repeated until a solution is found or i exceeds n.

Example 4.1. Suppose we have three objective functions f_1, f_2 and f_3. Suppose we have an initial solution with objective vector $(400, 500, 300)$. We enter function `FindParetoSol` where we will find a Pareto-optimal solution. The SAT solver is called on the hard clauses and the following constraints: $f_1 \leq 400$, $f_2 \leq 499$ and $f_3 \leq 300$. Suppose the SAT solver returns a solution with objective vector $(300, 200, 300)$. The SAT solver is now called with the following additional constraints: $f_1 \leq 299$, $f_2 \leq 200$, $f_3 \leq 299$. Suppose the SAT solver proves unsatisfiability. Then, we will try to decrease the 2nd maximum, with the 1st fixed. The SAT solver is called on the hard clauses and the following constraints: $f_1 = 300$, $f_2 \leq 200$

Algorithm 4: Function `BlockOptimum`

Input: Hard clauses $\mathcal{H}$, objective functions $f_1, \ldots, f_n$, assignment α and index i
Output: Assignment α and index i

$(k_1, \ldots, k_n) \leftarrow (f_1(\alpha), \ldots, f_n(\alpha))$
repeat
  $P \leftarrow \{p \in \{1, \ldots, n\} : k_p > k_i^{\downarrow}\}$
  $M \leftarrow \{m \in \{1, \ldots, n\} : k_m = k_i^{\downarrow}\}$
  while $|P| < i - 1$ **do**
    $m \leftarrow$ `SelectFrom`(M) `// get any element of` M
    $M \leftarrow M \setminus \{m\}$
    $P \leftarrow P \cup \{m\}$
  $(st, \alpha') \leftarrow$ `SAT`$(\mathcal{H} \cup \bigcup_{r \notin P}\{f_r \leq k_i^{\downarrow} - 1\} \cup \bigcup_{p \in P}\{f_p = k_p\})$
  if st **then**
    $\alpha \leftarrow \alpha'$
  else
    $i \leftarrow i + 1$
until st **or** $i > n$
return (i, α)

and $f_3 \leq 299$. Suppose the SAT solver outputs a solution with objective vector $(300, 200, 250)$. The SAT solver is now called on the hard clauses and the following constraints: $f_1 = 300$, $f_2 \leq 200$ and $f_3 \leq 249$. Suppose the formula is unsatisfiable. Then, we move on to decreasing the 3rd maximum. The SAT solver is called on the hard clauses and the following constraints: $f_1 = 300$, $f_2 \leq 199$ and $f_3 = 250$. Suppose the formula is unsatisfiable. We conclude that the last solution with objective vector $(300, 200, 250)$ is Pareto-optimal and function `FindParetoSol` ends. Now, in function `BlockOptimum`, we try to find a leximax-better solution by decreasing the i-th maximum without bounding the objective functions smaller than the i-th maximum. In this case, $i = 1$, and the SAT solver is called on the hard clauses and the following constraints: $f_1 \leq 299$, $f_2 \leq 299$ and $f_3 \leq 299$. If the formula is satisfiable, we execute `FindParetoSol` to find another Pareto-optimal solution. Otherwise, we conclude that 300 is the optimal 1st maximum, and we proceed by fixing $f_1 = 300$ and moving on to the 2nd maximum. In this case, we call the SAT solver on the hard clauses and the following constraints: $f_1 = 300$, $f_2 \leq 249$ and $f_3 \leq 249$. And so on.

The cardinality constraints (of the form $f = k$ and $f \leq k$) can be encoded to CNF in several ways. We implemented a sorting network encoding, using Batcher's odd even merge sorting network construction [5, 16]. This encoding is explained for example in [1].

Guided Improvement Algorithm for Leximax versus MSS Enumeration. In the Guided Improvement Algorithm for Leximax, we construct a sorting network for each objective function. As a result, the number of variables and clauses increases significantly. Consequently, each call to a SAT solver can be quite expensive. Hence, in a short amount of time, the Guided Improvement Algorithm may only be able to find a few solutions with a very bad objective value. However, each time a solution is found, we have a guaranteed leximax-better solution. On the other hand, MSSes are generally much faster to compute, and much more solutions will be obtained with MSS enumeration than with the Guided Improvement Algorithm. The

disadvantage of MSS enumeration is that there is no guarantee that the next MSS found will have a leximax-better objective vector than the previous one.

Incremental SAT Solving. The algorithms we have discussed can be implemented using incremental SAT solving (the IPASIR interface is described in [4]). In both MSS and Guided Improvement Algorithm approaches, we highlight two approaches: (1) using only one SAT solver throughout the entire computation and (2) using a new SAT solver in each single MSS/Pareto-optimal search.

Approach (1) does not allow the addition of the clauses of the MSS and MCS in construction (or cardinality constraints in the Guided Improvement Algorithm) to the hard clauses. We can only add them as assumptions, which means that all consequences of those clauses have to be deduced in each call to the SAT solver. On the other hand, it allows keeping the learned clauses of the formula during the entire computation.

Approach (2) allows the addition of the aforementioned clauses not as assumptions but as hard clauses. However, each time we use a different SAT solver we lose the learned clauses of the previous MSS/Pareto-optimal solution search.

When finding a single MSS/Pareto-optimal solution, approach (2) is obviously faster. The question is the performance of the approaches when obtaining several MSSes/Pareto-optimal solutions. It may happen that as more solutions are found and the number of SAT calls increases, the performance of approach (1) surpasses that of approach (2).

5 Evaluation

The experiments were run on 24 core Intel(R) Xeon(R) E5-2630 v2 CPU 2.60GHz machines with Debian Linux operating system and 4 processes running at the same time. The 142 upgradeability benchmarks[4] used are from the Mancoosi International Solver Competition of 2011. The 5 objective functions (or user criteria) used are: *removed*, *notuptodate*, *changed*, *unsat_recommends* and *new*.[5] The 142 benchmarks were run for all 26 combinations of two, three, four and five objective functions, resulting in 3692 problems. All solvers were run with a single thread.

First of all, the Integer Linear Programming approach [18] of the package upgradeability tool `mccs`[6] [20, 18] was evaluated (version 1.1), with a time limit of 180 seconds (3 minutes). `mccs` was run with the commercial solver CPLEX[7] (version 12.10.0), and the non-commercial solver SCIP [11] (version 7.0.1). Within the 180 seconds time limit, `mccs` with CPLEX was able to solve around 97% of the instances, and `mccs` with SCIP was able to solve around 69% of the instances. Figure 1 shows the *cactus plot* of the solving times of these solvers. In the case of CPLEX, a large percentage of the problems (roughly 80%) can be solved in under 30 seconds, and more than 50% of the problems can be solved in under 10 seconds. Hence, our new SAT-based approaches will likely not be competitive with `mccs` using CPLEX, in these package upgradeability problems.

Next, our new SAT-based approaches and the Integer Linear Programming approach are compared. Four main algorithms are compared:

[4] Benchmarks: http://data.mancoosi.org/misc2011/problems/
[5] Criteria definition: https://www.mancoosi.org/misc-2011/criteria/
[6] https://www.i3s.unice.fr/~cpjm/misc/mccs.html
[7] https://www.ibm.com/analytics/cplex-optimizer

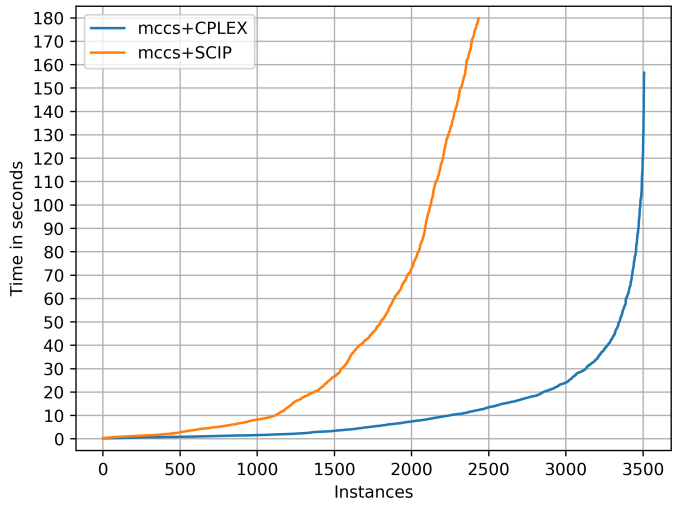

Figure 1: Cactus plot of the run times of solved instances, for `mccs` with CPLEX and SCIP.

1. Linear search MSS enumeration with our two heuristics of choosing the next clause from the maximum and adding satisfiable clauses to even out the upper bounds of the objective functions. Abbreviation: MSS-H (H from Heuristics).

2. Linear search MSS enumeration with sequential choice of the next clause and addition of all satisfiable clauses. Abbreviation: MSS-N (N from Normal).

3. Guided Improvement Algorithm for Leximax. Abbreviation: GIA.

4. Intermediate solutions of the Integer Linear Programming approach, using the package upgradeability tool `mccs` with CPLEX and SCIP. Abbreviations: CPLEX and SCIP.

Our approximation algorithms were implemented in the package upgradeability tool `packup` [15, 13] and the IPASIR interface of `cadical` [6] (version 1.3.1) was used for the SAT calls. Each algorithm was tested with both incremental SAT solving variants: using always the same SAT solver in the enumeration versus using a new SAT solver in the search for a single MSS or Pareto-optimal solution. When using the same SAT solver during the entire computation, the algorithms' abbreviations are appended with the string '-I' (from 'Incremental').

For package upgradeability problems, we considered an admissible time limit of 10 seconds. However, larger time limits are also evaluated, as the algorithms may be used to approximate other Multi-Objective Combinatorial Optimisation problems. The results are summarised as *victory tables*. For each timeout t there is a table. In each table, the algorithms are compared pairwise. In each instance, if algorithm x obtains a leximax-better solution than algorithm y, in the course of t seconds, then x wins against y. If the best solutions of algorithms x and y are leximax-indistinguishable, then it is a draw. In row x, column y of each table we have the difference between the percentage of wins of algorithm x and the percentage of losses of algorithm x against algorithm y. Hence, a positive number (highlighted in **bold**) in row x, column y, means algorithm x won more instances than algorithm y. Thus, rows with mainly

positive numbers are the winners. Also, note that the tables are symmetric, in the sense that if in row x column y there is a number n, then in row y column x there is the number $-n$. The results are shown in Tables 1, 2, 3, 4 and 5, for timeouts of 10, 30, 60, 120 and 180 seconds, respectively.

10 Seconds	MSS-H	MSS-N	GIA	MSS-H-I	MSS-N-I	GIA-I	CPLEX	SCIP
MSS-H	-	**34**	**46**	**9**	**56**	**51**	−66	**20**
MSS-N	−34	-	**34**	−5	**20**	**41**	−65	**20**
GIA	−46	−34	-	−41	−21	**9**	−64	**28**
MSS-H-I	−9	**5**	**41**	-	**40**	**45**	−68	**20**
MSS-N-I	−56	−20	**21**	−40	-	**29**	−68	**20**
GIA-I	−51	−41	−9	−45	−29	-	−67	**27**
CPLEX	**66**	**65**	**64**	**68**	**68**	**67**	-	**61**
SCIP	−20	−20	−28	−20	−20	−27	−61	-

Table 1: Victory Table for a timeout of 10 seconds. Row x, column y, is the difference between the percentage of wins and the percentage of losses of solver x against solver y. A positive number (in **bold**) in row x, column y, means more victories for solver x against solver y.

30 Seconds	MSS-H	MSS-N	GIA	MSS-H-I	MSS-N-I	GIA-I	CPLEX	SCIP
MSS-H	-	**45**	**9**	−24	**48**	**17**	−85	−5
MSS-N	−45	-	−17	−32	−9	−2	−85	−4
GIA	−9	**17**	-	−5	**21**	**14**	−65	**12**
MSS-H-I	**24**	**32**	**5**	-	**37**	**18**	−85	−5
MSS-N-I	−48	**9**	−21	−37	-	−5	−84	−4
GIA-I	−17	**2**	−14	−18	**5**	-	−67	**12**
CPLEX	**85**	**85**	**65**	**85**	**84**	**67**	-	**51**
SCIP	**5**	**4**	−12	**5**	**4**	−12	−51	-

Table 2: Victory Table for a timeout of 30 seconds. Row x, column y, is the difference between the percentage of wins and the percentage of losses of solver x against solver y. A positive number (in **bold**) in row x, column y, means more victories for solver x against solver y.

60 Seconds	MSS-H	MSS-N	GIA	MSS-H-I	MSS-N-I	GIA-I	CPLEX	SCIP
MSS-H	-	**47**	**13**	−39	**46**	−5	−86	−30
MSS-N	−47	-	−44	−44	−17	−31	−85	−28
GIA	−13	**44**	-	**13**	**43**	**9**	−60	−7
MSS-H-I	**39**	**44**	−13	-	**45**	−3	−86	−29
MSS-N-I	−46	**17**	−43	−45	-	−31	−85	−28
GIA-I	**5**	**31**	−9	**3**	**31**	-	−61	−7
CPLEX	**86**	**85**	**60**	**86**	**85**	**61**	-	**40**
SCIP	**30**	**28**	**7**	**29**	**28**	**7**	−40	-

Table 3: Victory Table for a timeout of 60 seconds. Row x, column y, is the difference between the percentage of wins and the percentage of losses of solver x against solver y. A positive number (in **bold**) in row x, column y, means more victories for solver x against solver y.

120 Seconds	MSS-H	MSS-N	GIA	MSS-H-I	MSS-N-I	GIA-I	CPLEX	SCIP
MSS-H	-	**47**	−24	−45	**46**	−18	−86	−54
MSS-N	−47	-	−51	−48	−21	−47	−85	−53
GIA	**24**	**51**	-	**21**	**50**	**4**	−56	−26
MSS-H-I	**45**	**48**	−21	-	−47	−15	−86	−54
MSS-N-I	−46	**21**	−50	**47**	-	−46	−84	−53
GIA-I	**18**	**47**	−4	**15**	**46**	-	−56	−25
CPLEX	**86**	**85**	**56**	**86**	**84**	**56**	-	**32**
SCIP	**54**	**53**	**26**	**54**	**53**	**25**	−32	-

Table 4: Victory Table for a timeout of 120 seconds. Row x, column y, is the difference between the percentage of wins and the percentage of losses of solver x against solver y. A positive number (in **bold**) in row x, column y, means more victories for solver x against solver y.

180 Seconds	MSS-H	MSS-N	GIA	MSS-H-I	MSS-N-I	GIA-I	CPLEX	SCIP
MSS-H	-	**47**	−29	−47	**45**	−23	−86	−66
MSS-N	−47	-	−57	−50	−23	−51	−85	−64
GIA	**29**	**57**	-	**25**	**56**	**6**	−54	−34
MSS-H-I	**47**	**50**	−25	-	**49**	−20	−86	−65
MSS-N-I	−45	**23**	−56	−49	-	−51	−84	−64
GIA-I	**23**	**51**	−6	**20**	**51**	-	−54	−34
CPLEX	**86**	**85**	**54**	**86**	**84**	**54**	-	**26**
SCIP	**66**	**64**	**34**	**65**	**64**	**34**	−26	-

Table 5: Victory Table for a timeout of 180 seconds. Row x, column y, is the difference between the percentage of wins and the percentage of losses of solver x against solver y. A positive number (in **bold**) in row x, column y, means more victories for solver x against solver y.

5.1 Discussion

General Comparison. The Integer Linear Programming approach using the commercial solver CPLEX is clearly the best approach, for all the analysed time limits. However, the SAT-based approaches had a better performance than the Integer Linear Programming approach using the non-commercial solver SCIP, in the 10 second timeout, and are competitive in the 30 second timeout. For larger timeouts, we observe that SCIP is better than all of the SAT-based approaches. And, the greater the timeout, the greater the difference in performance. For small timeouts, such as a 10 second timeout, the MSS enumeration approaches have the best performance among all algorithms excluding CPLEX. For sufficiently large time limits, the Guided Improvement Algorithm is better than MSS enumeration, the difference being accentuated after the 60 second timeout.

Comparison of MSS Heuristics. In general, the two heuristics proposed for the computation of one MSS helped to obtain leximax-better solutions than if it were implemented using the typical approach of choosing the clauses sequentially and adding all satisfiable clauses to the MSS.

Comparison of Incremental Approaches. The tables show similar behaviour between using the same SAT solver during the entire computation and using a new SAT solver in each MSS/Pareto-optimal solution search. However, the results suggest that using only one SAT solver can become slightly better than using many SAT solvers, if the computation time is sufficiently large, in the MSS enumeration case. In the Guided Improvement Algorithm case, using multiple SAT solvers is slightly better for all timeouts. This is expected, as the number of calls to the SAT solver in this case is small, and so adding constraints as hard clauses (not as assumptions) at the expense of losing learned clauses pays off.

6 Conclusions

We proposed two adaptations of well-known algorithms in Multi-Objective Combinatorial Optimisation, with the goal of approximating the leximax-optimum. The leximax-optimum is *fair* [8] Pareto-optimum, having typically a small trade-off between the objective functions.

The adaptation of the Guided Improvement Algorithm [14] and MSS enumeration [21, 13] to leximax optimisation were presented. Two heuristics were proposed for the MSS extended linear search algorithm [19]. These new SAT-based approaches for approximating the leximax-optimum were empirically compared with an earlier Integer Linear Programming approach [18, 8], in a large set of benchmarks of the package upgradeability problem. The results show that the Integer Linear Programming approach with the commercial solver CPLEX[8] performs better than all the other approaches, in timeouts of 10, 30, 60, 120 and 180 seconds. Excluding CPLEX, the SAT-based approaches are better than or competitive with the Integer Linear Programming approach, using the non-commercial solver SCIP [11], for small timeouts (10 and 30 seconds). The MSS enumeration algorithm performed better than all the other approaches, excluding CPLEX, in the 10 second timeout, which corresponds to a reasonable practical time limit in package upgradeability. Moreover, the two heuristics proposed for the MSS extended linear search algorithm enhanced its performance. However, the number of MSSes that the algorithm is able to find, is very small compared to the total number of MSSes. In our implementation of the MSS enumeration algorithms, we did not include any other heuristics not mentioned in this paper. In particular, the order of traversal of clauses in the algorithms was fixed. We believe this contributed to finding very similar MSSes. Hence, as future work, a more rigorous assessment should be made of the MSS enumeration algorithms taking into account other known heuristics, such as diversification techniques [22].

Acknowledgements

This work is supported in part by the Fundação para a Ciência e Tecnologia projects UIDB/50021/2020, PTDC/CCI-COM/31198/2017 and PTDC/CCI-COM/32378/2017. The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN with reference LL1902. This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

[8] https://www.ibm.com/analytics/cplex-optimizer

References

[1] Ignasi Abío and Peter J Stuckey. Encoding linear constraints into sat. *International Conference on Principles and Practice of Constraint Programming*, pages 75–91, 2014.

[2] Josep Argelich, Daniel Le Berre, Inês Lynce, João Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. *Electronic Proceedings in Theoretical Computer Science*, 29:11–22, 07 2010.

[3] James Bailey and Peter Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. *Lecture Notes in Computer Science*, 3350:174–186, 2005.

[4] Tomáš Balyo, Armin Biere, Markus Iser, and Carsten Sinz. Sat race 2015. *Artificial Intelligence*, 241:45–65, 2016.

[5] K.E. Batcher. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computer Conference*, 32:307–314, 1968.

[6] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. B-2020-1:51–53, 2020.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *The Art of Computer Programming: Fundamental algorithms*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[8] Sylvain Bouveret and Michel Lemaître. Computing leximin-optimal solutions in constraint networks. *Artificial Intelligence*, 173(2):343–364, 2009.

[9] Roberto Di Cosmo, Berke Durak, Xavier Leroy, Fabio Mancinelli, and Jérôme Vouillon. Maintaining large software distributions: new challenges from the FOSS era. *Proceedings of the FRCSS 2006 workshop*, 2006. EASST Newsletter.

[10] Matthias Ehrgott. *Multicriteria Optimization*. Springer-Verlag, Berlin, Heidelberg, 2005.

[11] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020.

[12] Martin Gebser, Roland Kaminski, and Torsten Schaub. aspcud: A linux package configuration tool based on answer set programming. *Electronic Proceedings in Theoretical Computer Science*, 65, 09 2011.

[13] Alexey Ignatiev, Mikoláš Janota, and Joao Marques-Silva. Towards efficient optimization in package management systems. 2014.

[14] Daniel Jackson, H.-Christian Estler, and Derek Rayside. The guided improvement algorithm for exact, general-purpose, many-objective combinatorial optimization. 07 2009.

[15] Mikoláš Janota, Inês Lynce, Vasco Manquinho, and Joao Marques-Silva. Packup: Tools for package upgradability solving: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 8, 01 2012.

[16] D.E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[17] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 8, 2012.

[18] Mancoosi deliverable 4.3. http://www.mancoosi.org/reports/d4.3.pdf.

[19] Joao Marques-Silva, Federico Heras, Mikolas Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, page 615–622, 2013.

[20] Claude Michel and Michel Rueher. Handling software upgradeability problems with milp solvers. *Electronic Proceedings in Theoretical Computer Science*, 29:1–10, 07 2010.

[21] Miguel Terra-Neves, Inês Lynce, and Vasco Manquinho. Introducing pareto minimal correction subsets. *Proceedings of the Twentieth International Conference Theory and Applications of Satisfiability Testing*, pages 195–211, 2017.

[22] Miguel Terra-Neves, Inês Lynce, and Vasco Manquinho. Enhancing constraint-based multi-objective combinatorial optimization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.