# Set Variables and Local Search*

Magnus Ågren, Pierre Flener, and Justin Pearson

Department of Information Technology
Uppsala University, Box 337, SE – 751 05 Uppsala, Sweden
{agren,pierref,justin}@it.uu.se

**Abstract.** Many combinatorial (optimisation) problems have natural models based on, or including, set variables and set constraints. This was already known to the constraint programming community, and solvers based on constructive search for set variables have been around for a long time. In this paper, set variables and set constraints are put into a local-search framework, where concepts such as configurations, penalties, and neighbourhood functions are dealt with generically. This scheme is then used to define the penalty functions for five (global) set constraints, and to model and solve two well-known applications.

## 1 Introduction

Many combinatorial (optimisation) problems have natural models based on, or including, set variables and set constraints. Classical examples include set partitioning and set covering, and such problems also occur as sub-problems in many real-life applications, such as airline crew rostering, tournament scheduling, time-tabling, and nurse rostering. This was already known to the constraint programming community, and constructive search (complete) solvers for set variables have been around for a long time now (see for example [11, 15, 19, 2]).

Complementary to constructive search, local search [1] is another common technique for solving combinatorial (optimisation) problems. Although not complete, it usually scales very well to large problem instances and often compares well to, or outperforms, other techniques. Historically, the constraint programming community has been mostly focused on constructive search and has only recently started to apply its ideas to local search. This means that concepts such as high declarativeness, global constraints with underlying incremental algorithms, and high-level modelling languages for local search have been introduced there (see [12, 25, 22, 16, 10, 7, 13, 23, 14, 6] for instance).

In this paper, we introduce set variables and (global) set constraints to constraint-based local search. More specifically, our *contributions* are as follows:

– We put the local-search concepts of penalties, configurations, and neighbourhood functions into a *set-variable framework*. (Section 2)

* This paper significantly extends and revises Technical Report 2004-015 of the Department of Information Technology, Uppsala University, Sweden.

– In order to be able to use (global) set constraints generally in local search, we propose a *generic penalty scheme*. We use it to give the *penalty definitions of five (global) set constraints*. Other than their well-known *modelling* merits, we show that (global) set constraints provide opportunities for a hardwired global reasoning while *solving*, which would otherwise have to be hand-coded each time for lower-level encodings of set variables, such as integer variables for the characteristic functions of their set values. (Section 3)
– In order to obtain efficient solution algorithms, we propose methods for the *incremental penalty maintenance* of the (global) set constraints. (Section 4)
– The (global) set constraints are used to *model and solve two well-known problems*, with promising results that motivate further research. (Section 5)

After this, Section 6 discusses related and future work and concludes the paper.

## 2 Local Search on (Set) Constraint Satisfaction Problems

A *constraint satisfaction problem (CSP)* is a triple $\langle V, D, C\rangle$, where $V$ is a finite set of variables, $D$ is a finite set of finite domains, each $D_v \in D$ containing the set of possible values for the corresponding variable $v \in V$, and $C$ is a finite set of constraints, each $c \in C$ being defined on a subset of the variables in $V$ and specifying their valid combinations of values.

The definition above is very general and may be used with any choice of finite-domain variables. The variables in $V$ may, for example, range over sets of integers (integer variables), strings, or, as in our case, sets of values of some type (set variables, defined formally below). Of course, a CSP may also contain variables with several kinds of domains. As an example, consider a CSP $\langle V, D, C\rangle$ in which some variables $\{i_1, \ldots, i_k\} \subset V$ are integer variables, and some other variables $\{s_1, \ldots, s_k\} \subset V$ are set variables. These could for instance be connected with constraints stating that the cardinality of each $s_j$ must not exceed $i_j$.

In this paper, we assume that *all* the variables are set variables, and that all the constraints are stated on variables of this kind. This is of course a limitation, since many models contain both set variables and integer variables. However, mixing integer variables and set variables makes the constraints harder to define, and we consider this to be future work. Fortunately, interesting applications, such as the two in this paper, are already possible to model.

**Definition 1 (Set Variable and its Universe).** *Let $P = \langle V, D, C\rangle$ be a CSP. A variable $s \in V$ is a* set variable *if its corresponding domain $D_s = 2^{U_s}$, where $U_s$ is a finite set of values of some type, called the* universe *of $s$.*

Note that this definition does not allow the indication of a non-empty set of required values in the universe of a set variable, hence this must be done here by an explicit constraint. This is left as future work, as not necessary for our present purpose.

**Definition 2 (Configuration).** *Let $P = \langle V, D, C\rangle$ be a CSP. A* configuration *for $P$ is a total function $k : V \to \bigcup_{s \in V} D_s$ such that $k(s) \in D_s$ for all $s \in V$.*

**Definition 3 (Delta of Configurations).** *Let $P = \langle V, D, C\rangle$ be a CSP and let $k$ and $k'$ be two configurations for $P$. The* delta of $k$ and $k'$*, denoted $delta(k, k')$, is the set $\{(s, v, v') \mid s \in V \ \ \& \ \ v = k(s) - k'(s) \ \ \& \ \ v' = k'(s) - k(s) \ \ \& \ \ v \neq v'\}$, where $-$ stands for the set difference.*

*Example 1.* Consider a CSP $P = \langle\{s_1, s_2, s_3\}, \{D_{s_1}, D_{s_2}, D_{s_3}\}, C\rangle$ where $D_{s_1} = D_{s_2} = D_{s_3} = 2^{\{d_1,d_2,d_3\}}$ (hence $U_{s_1} = U_{s_2} = U_{s_3} = \{d_1, d_2, d_3\}$). One possible configuration for $P$ is defined as $k(s_1) = \{d_3\}, k(s_2) = \{d_1, d_2\}, k(s_3) = \emptyset$, or equivalently as the set of mappings $\{s_1 \mapsto \{d_3\}, s_2 \mapsto \{d_1, d_2\}, s_3 \mapsto \emptyset\}$. Another configuration for $P$ is defined as $k' = \{s_1 \mapsto \emptyset, s_2 \mapsto \{d_1, d_2, d_3\}, s_3 \mapsto \emptyset\}$. Now, the delta of $k$ and $k'$ is $delta(k, k') = \{(s_1, \{d_3\}, \emptyset), (s_2, \emptyset, \{d_3\})\}$.

**Definition 4 (Neighbourhood Function).** *Let $K$ denote the set of all possible configurations for a CSP $P$ and let $k \in K$. A* neighbourhood function *for $P$ is a function $\mathcal{N} : K \to 2^K$. The* neighbourhood *of $P$ with respect to $k$ and $\mathcal{N}$ is the set of configurations $\mathcal{N}(k)$.*

*Example 2.* Consider $P$ and $k$ from Example 1. A possible neighbourhood of $P$ with respect to $k$ and some neighbourhood function $\mathcal{N}$ for $P$ is the set $\mathcal{N}(k) = \{k_1 = \{s_1 \mapsto \emptyset, s_2 \mapsto \{d_1, d_2, d_3\}, s_3 \mapsto \emptyset\}, k_2 = \{s_1 \mapsto \emptyset, s_2 \mapsto \{d_1, d_2\}, s_3 \mapsto \{d_3\}\}$. This neighbourhood function moves the value $d_3$ in $s_1$ to variable $s_2$ or variable $s_3$, decreasing the cardinality of $s_1$ and increasing the one of $s_2$ or $s_3$.

We will use two *general neighbourhoods* in this paper, which are defined next. For both, let $s \in V$, $S \subseteq V - \{s\}$, and let $k \in K$ be a configuration for a CSP $P = \langle V, D, C\rangle$, where $K$ is the set of all configurations for $P$. The first one, called *move*, is defined by the neighbourhood function with the same name:

$$move(s, S)(k) = \{k' \in K \mid \exists d \in k(s) : s' \in S \ \ \& \ \ d \in U_{s'} - k(s') \ \ \& \ \ delta(k, k') = \{(s, \{d\}, \emptyset), (s', \emptyset, \{d\})\}\}$$

This neighbourhood, given $k$, is the set of all neighbourhoods $k'$ that differ from $k$ in the definition of two distinct set variables $s$ and $s'$, the difference being that there exists exactly one $d \in k(s)$ such that $d \in k(s) \Leftrightarrow d \notin k'(s)$ and $d \notin k(s') \Leftrightarrow d \in k'(s')$. Hence, $d$ was moved from $s$ to $s'$.

The second one, called *swap*, is defined by the neighbourhood function:

$$swap(s, S)(k) = \{k' \in K \mid \exists d \in k(s) : \exists d' \in U_s - k(s) : s' \in S \ \ \& \ \ d' \in k(s') \ \ \& \ \ d \in U_{s'} - k(s') \ \ \& \ \ delta(k, k') = \{(s, \{d\}, \{d'\}), (s', \{d'\}, \{d\})\}\}$$

This neighbourhood, given $k$, is the set of all neighbourhoods $k'$ that differ from $k$ in the definition of two distinct set variables $s$ and $s'$, the difference being that there exists exactly one pair ($d \in k(s)$, $d' \in U_s - k(s)$) such that $d \in k(s) \Leftrightarrow d \notin k'(s)$ and $d \notin k(s') \Leftrightarrow d \in k'(s')$, and the opposite for $d'$. Hence, $d$ and $d'$ were swapped between $s$ and $s'$.

We will now define the notion of penalty of a constraint, which, informally, is an estimate on how much a constraint is violated. Below is a general definition, followed by a generic scheme for balancing the penalties of different constraints, which is then specialised for each constraint in Section 3.

**Definition 5 (Penalty).** *Let $P = \langle V, D, C \rangle$ be a CSP and let $K$ denote the set of all possible configurations for $P$. A* penalty of a constraint $c \in C$ *is a function $penalty(c) : K \rightarrow \mathbb{N}$. The* penalty of $P$ with respect to $k$ *is the sum $\sum_{c \in C} penalty(c)(k)$.*

*Example 3.* Consider once again $P$ from Example 1 and let $c_1$ and $c_2$ be the constraints $s_1 \subseteq s_2$ and $d_3 \in s_3$ respectively. Let the penalty functions of $c_1$ and $c_2$ be defined as: $penalty(c_1)(k) = |k(s_1) - k(s_2)|$, and $penalty(c_2)(k) = 0$, *if* $d_3 \in k(s_3)$, *or* 1, *otherwise* . Now, the penalties of $P$ with respect to the different configurations in the neighbourhood of Example 2 are $penalty(c_1)(k_1) + penalty(c_2)(k_1) = 1$, and $penalty(c_1)(k_2) + penalty(c_2)(k_2) = 0$ respectively.

In order for a constraint-based local-search approach to be effective, different constraints should have balanced penalty definitions [6]: i.e. for a set of constraints $C$, no $c \in C$ should be easier in general to satisfy compared to any other $c' \in C$. This may be application dependent, in which case weights could be added to tune the penalties, see [13] for example. For set constraints, we believe that one such penalty definition is to let (by extension of the integer-variable ideas in [10]) the penalty of a set constraint $c$ be the length of the shortest sequence of atomic set operations (defined below) that must be performed on the variables in $c$ under a configuration $k$ in order to satisfy $c$.

**Definition 6 (Atomic Set Operations).** *Let $P = \langle V, D, C \rangle$ be a CSP, let $k$ be a configuration for $P$, and let $s \in V$. An* atomic set operation on $k(s)$ *is one of the following changes to $k(s)$:*

1. *Add a value $d$ to $k(s)$ from its complement $U_s - k(s)$, denoted $Add(k(s), d)$.*
2. *Remove a value $d$ from $k(s)$, denoted $Remove(k(s), d)$.*

Note that no value-replacement operation is considered here; its inclusion would imply a reduction of some of the penalties in Section 3.

*Example 4.* Performing $\Delta = [Add(k(s), d),\ Remove(k(s), b),\ Add(k(s'), b)]$ on $k(s) = \{a, b, c\}$ and $k(s') = \emptyset$ will yield $\Delta(k(s)) = \{a, c, d\}$ and $\Delta(k(s')) = \{b\}$.

**Definition 7 (Operation-Based Penalty for Set Constraints).** *Let $P = \langle V, D, C \rangle$ be a CSP and let $K$ be the set of all configurations for $P$. Let $c \in C$ be a constraint defined on a set of set variables $S \subseteq V$. The* penalty of $c$, $penalty(c) : K \rightarrow \mathbb{N}$, *is the length of the shortest sequence of atomic set operations that must be performed in order to satisfy $c$ given a specific configuration $k$.*

From this definition it follows that $penalty(c)(k) = 0$ if and only if $c$ is satisfied with respect to $k$. Also, as will be seen, to find a penalty that complies with this definition for a given set constraint is not always obvious.

## 3 (Global) Set Constraints and Their Penalties

We now present five (global) set constraints and define their penalties. Throughout this section, we assume that $k$ is a configuration for a CSP $P = \langle V, D, C \rangle$, and that $c \in C$.

### 3.1 AllDisjoint

The global constraint $\mathit{AllDisjoint}(S)$, where $S = \{s_1, \ldots, s_n\}$ is a set of set variables, expresses that all distinct pairs in $S$ are disjoint, i.e. that $\forall i < j \in 1 \ldots n : s_i \cap s_j = \emptyset$. The penalty of an $\mathit{AllDisjoint}(S)$ constraint under $k$ is equal to the length of the shortest sequence $\Delta$ of atomic set operations of the form $\mathit{Remove}(k(s), d)$ that must be performed in order for $\forall i < j \in 1 \ldots n : \Delta(k(s_i)) \cap \Delta(k(s_j)) = \emptyset$ to hold. We define the penalty as:

$$\mathit{penalty}(\mathit{AllDisjoint}(S))(k) = \left(\sum_{s \in S} |k(s)|\right) - \left|\bigcup_{s \in S} k(s)\right| \tag{1}$$

Indeed, we need to remove all repeated occurrences of any value, and their number equals the difference between the sum of the set sizes and the size of their union. Hence the following proposition:

**Proposition 1.** *The penalty (1) is correct with respect to Definition 7.*

### 3.2 Cardinality

The constraint $\mathit{Cardinality}(s, m)$, where $s$ is a set variable and $m$ a natural-number constant, expresses that the cardinality of $s$ is equal to $m$, i.e. that $|s| - m$. This constraint would of course be more powerful if we allowed $m$ to be an integer variable. However, as was mentioned earlier, the penalty would be more complicated if we did this, and we see this as future work.

The penalty of a $\mathit{Cardinality}(s, m)$ constraint under $k$ is equal to the length of the shortest sequence $\Delta$ of atomic set operations of the form $\mathit{Add}(k(s), d)$ or $\mathit{Remove}(k(s), d)$ that must be performed in order for $|\Delta(k(s))| = m$ to hold. The penalty below expresses this:

$$\mathit{penalty}(\mathit{Cardinality}(s, m))(k) = \mathit{abs}(|k(s)| - m) \tag{2}$$

where $\mathit{abs}(e)$ denotes the absolute value of the expression $e$. Indeed, we need to add (remove) exactly as many values to (from) $k(s)$ in order to increase (decrease) its cardinality to $m$. Hence the following proposition:

**Proposition 2.** *The penalty (2) is correct with respect to Definition 7.*

### 3.3 MaxIntersect

The global constraint $\mathit{MaxIntersect}(S, m)$, where $S = \{s_1, \ldots, s_n\}$ is a set of set variables and $m$ a natural-number constant, expresses that the cardinality of the intersection between any distinct pair in $S$ is at most $m$, i.e. that $\forall i < j \in 1 \ldots n : |s_i \cap s_j| \leq m$. This constraint expresses the same as an $\mathit{AllDisjoint}(S)$ constraint when $m = 0$. However, as will be seen, keeping the *AllDisjoint* constraint is useful for this special case. Again, allowing $m$ to be an integer variable would make the constraint more powerful and is future work.

The penalty of a $\mathit{MaxIntersect}(S, m)$ constraint under $k$ is equal to the length of the shortest sequence $\Delta$ of atomic set operations of the form $\mathit{Remove}(k(s), d)$ that must be performed such that $\forall i < j \in 1 \ldots n : |\Delta(k(s_i)) \cap \Delta(k(s_j))| \leq m$ holds. In fact, finding a closed form for the exact penalty of a *MaxIntersect* constraint with respect to Definition 7 turns out not to be that easy. The following expression gives an upper bound on this penalty, namely the sum of the excesses of the intersection sizes:

$$\mathit{penalty}(\mathit{MaxIntersect}(S, m))(k) \leq \sum_{1 \leq i < j \leq n} \mathit{max}(|k(s_i) \cap k(s_j)| - m, 0) \quad (3)$$

*Example 5.* Assume that $k(s_1) = \{d_1, d_2, d_3\}$, $k(s_2) = \{d_2, d_3, d_4\}$, $k(s_3) = \{d_1, d_3, d_4\}$, and that $c = \mathit{MaxIntersect}(\{s_1, s_2, s_3\}, 1)$. The penalty of $c$ according to (3) is $2 + 2 + 2 = 3$. Indeed, we may satisfy $c$ by performing the sequence of 3 operations $[\mathit{Remove}(k(s_1), d_1), \mathit{Remove}(k(s_2), d_2), \mathit{Remove}(k(s_3), d_3)]$. However, this is not the shortest sequence that achieves this, since after performing $[\mathit{Remove}(k(s_1), d_3), \mathit{Remove}(k(s_2), d_3)]$, the constraint $c$ is also satisfied.

**Proposition 3.** *The bound of (3) is an optimal upper bound w.r.t. Definition 7.*

**Proposition 4.** *The upper bound of (3) is zero iff $\mathit{MaxIntersect}(S, m)$ holds.*

However, the upper bound of (3) is not correct with respect to Definition 7 when $m = 0$. Consider $s_1 = \{d_1, d_2\}$, $s_2 = \{d_2, d_3\}$, and $s_3 = \{d_2, d_3\}$. The penalty under (3) of $\mathit{MaxIntersect}(\{s_1, s_2, s_3\}, 0)$ is $1 + 1 + 2 = 4$ whereas the one of $\mathit{AllDisjoint}(\{s_1, s_2, s_3\})$ correctly is $6 - 3 = 3$ under (1).

We may also obtain a lower bound, by using a lemma due to Corrádi [8].

**Lemma 1 (Corrádi).** *Let $s_1, \ldots, s_n$ be $r$-element sets and $U$ be their union. If $|s_i \cap s_j| \leq m$ for all $i \neq j$, then $|U| \geq \frac{r^2 \cdot n}{r + (n-1) \cdot m}$.*

This lemma can be applied for $n$ ground sets that do not necessarily all have the *same* cardinality $r$, but rather with $r$ being the *maximum* of their cardinalities, as is the case with $\mathit{MaxIntersect}(S, m)$ and $|S| = n$. It suffices to apply the corrective term $\delta = n \cdot r - \sum_{s \in S} |k(s)|$ when using the lower bound for a configuration $k$ where $r = \mathit{max}_{s \in S} |k(s)|$. Note that $\delta$ is the amount of distinct new elements (from a sufficiently large fictitious universe disjoint from $\bigcup_{s \in S} k(s)$) that one must add to the sets in $\{k(s) \mid s \in S\}$ in order to make them all be of size $r$.

We now have the following lower bound on the penalty of a $\mathit{MaxIntersect}(S, m)$ constraint under a configuration $k$ (where $|S| = n$ and $r = \mathit{max}_{s \in S} |k(s)|$):

$$\mathit{penalty}(\mathit{MaxIntersect}(S, m))(k) \geq \left\lceil \frac{r^2 \cdot n}{r + (n-1) \cdot m} \right\rceil - \left( n \cdot r - \sum_{s \in S} |k(s)| \right) - \left| \bigcup_{s \in S} k(s) \right| \quad (4)$$

*Example 6.* Recall Example 5, where $m = 1$ and the $n = 3$ sets are of the same size $r = 3$, hence $\delta = 0$, and have a union of 4 elements. We get $\mathit{penalty}(c)(k) \geq \lceil \frac{27}{5} \rceil - 0 - 4 = 2$, which is correct with respect to Definition 7.

Now, the following proposition follows from Lemma 1:

**Proposition 5.** *The bound of (4) is an optimal lower bound w.r.t. Definition 7.*

The next proposition establishes what happens when $m = 0$, in which case $\mathit{MaxIntersect}(S, m)$ is equivalent to $\mathit{AllDisjoint}(S)$:

**Proposition 6.** *The lower bound of (4) is correct wrt Definition 7 when $m = 0$.*

*Proof.* When $m = 0$, then $\left\lceil \frac{r^2 \cdot n}{r+(n-1)\cdot m} \right\rceil = r \cdot n$ and the lower bound of (4) simplifies into the penalty expression (1). Hence it is correct, by Proposition 1.

Unfortunately, the lower bound is sometimes zero even though the constraint is violated. Consider $n = 10$ sets, all of size $r = 3$ (hence $\delta = 0$), that should have pairwise intersections of at most $m = 1$ element and that have a union of 8 elements. Then (4) gives 0 as lower bound on the penalty, but the constraint is violated as there are no such 10 sets, hence $m$ would have to be at least 2.

However, we may still use (4) for the *MaxIntersect* constraint, but it would have to be in conjunction with (3), with the condition that if the lower bound of (4) is zero, then one uses the upper bound of (3) instead. In our experience, the lower bound of (4) is frequently correct. This also argues for keeping the explicit constraint *AllDisjoint*, since for that constraint (4) gives the correct penalty.

An often tighter upper bound than the one of (3) can be obtained by Algorithm 1. It obtains an estimate of the penalty by returning the length of a sequence of atomic set operations constructed in the following way: (i) Start with the empty sequence. (ii) Until the constraint is satisfied, add an atomic set operation removing a value that belongs to a set variable that takes part in the largest number of violating intersections. The algorithm uses the upper bound of (3) as the exit criterion, as it is zero only upon satisfaction of the constraint, by Proposition 4.

**Algorithm 1** Calculating the penalty of a *MaxIntersect* constraint.

**function** $\mathit{penalty\_max\_intersect}(S, m)(k)$
  $l \leftarrow 0$
  **while** $\mathit{penalty}(\mathit{MaxIntersect}(S, m))(k) > 0$ **do** ▷ According to (3)
    *choose* $d \in \bigcup_{s \in S} k(s)$ s.t. $|\{(i, j) \mid i < j \ \ \& \ \ d \in k(s_i) \cap k(s_j) \ \ \& \ \ |k(s_i) \cap k(s_j)| > m\}|$ is maximised.
    *choose* $s_i \in S$ s.t. $|\{s_j \in S \mid i \neq j \ \ \& \ \ d \in k(s_i) \cap k(s_j) \ \ \& \ \ |k(s_i) \cap k(s_j)| > m\}|$ is maximised.
    $l \leftarrow l + 1$ ▷ i.e. an imaginary $\mathit{Remove}(k(s_i), d)$ operation was added
    Replace the binding for $s_i$ in $k$ by $s_i \mapsto k(s_i) - \{d\}$
  **return** $l$

In the current implementation of the *MaxIntersect* constraint, we use the upper bound given by (3). As we have seen, this is not always a good estimate on the penalty with respect to Definition 7. In the future, we plan to use (4) in conjunction with (3) or (an incremental variant of) Algorithm 1.

### 3.4 MaxWeightedSum

The constraint $\mathit{MaxWeightedSum}(s, w, m)$, where $s$ is a set variable, $w : U_s \to \mathbb{N}$ is a weight function from the universe of $s$ to the natural numbers, and $m$ is a natural-number constant, expresses that $\sum_{d \in s} w(d) \leq m$. Note that we do not allow negative weights nor $m$ to be an integer variable. Allowing these would need a redefinition of the penalty below.

The penalty of a $\mathit{MaxWeightedSum}(s, w, m)$ constraint under $k$ is equal to the length of the shortest sequence $\Delta$ of operations of the form $\mathit{Remove}(k(s), d)$ that must be performed in order for $\sum_{d \in \Delta(k(s))} w(d) \leq m$ to hold. We define the following penalty:

$$\begin{aligned} &\mathit{penalty}(\mathit{MaxWeightedSum}(s, w, m))(k) = \\ &\qquad \mathit{min\_card}\left(\left\{s' \subseteq k(s) \mid \textstyle\sum_{d' \in s'} w(d') \geq \left(\sum_{d \in k(s)} w(d)\right) - m\right\}\right) \end{aligned} \tag{5}$$

where $\mathit{min\_card}(Q)$ denotes the cardinality of a set $q \in Q$ such that for all $q' \in Q$, $|q| \leq |q'|$, or 0 if $Q = \emptyset$. Indeed, we must remove at least the smallest set of values from $k(s)$ such that their weighted sum is at least the difference between the weighted sum of all values in $k(s)$ and $m$. Hence the following proposition:

**Proposition 7.** *The penalty (5) is correct with respect to Definition 7.*

### 3.5 Partition

The global constraint $\mathit{Partition}(S, q)$, where $S = \{s_1, \ldots, s_n\}$ is a set of set variables and $q$ is a ground set of values, expresses that the variables in $S$ are all disjoint, i.e. that $\forall i < j \in 1 \ldots n : s_i \cap s_j = \emptyset$, and that their union is equal to $q$, i.e. that $\bigcup_{s \in S} s = q$. Note that this definition of a partition allows one or more variables in $S$ to be empty, which is useful in some applications, such as the progressive party problem below. The set $q$, called the *reference set*, could be generalised to be a set variable. The applications we currently look at do not expect this but this may change in the future. In that case, the penalty function below would have to be changed to take this into account.

The penalty of a $\mathit{Partition}(S, q)$ constraint under $k$ is equal to the length of the shortest sequence $\Delta$ of atomic set operations that must be performed in order for $\forall i < j \in 1 \ldots n : \Delta(k(s_i)) \cap \Delta(k(s_j)) = \emptyset \;\;\&\;\; \bigcup_{s \in S} \Delta(k(s)) = q$ to hold. The following penalty expresses this:

$$\mathit{penalty}(\mathit{Partition}(S, q))(k) = \left(\sum_{s \in S} |k(s)|\right) - \left|\bigcup_{s \in S} k(s)\right| + \left|q - \bigcup_{s \in S} k(s)\right| \tag{6}$$

Indeed, the first two terms are those in (1) for *AllDisjoint* and the third term expresses that all unused elements of the reference set must be added to some set of the partition for the union to hold. Hence the following proposition:

**Proposition 8.** *The penalty (6) is correct with respect to Definition 7.*

Note that this penalty could be reduced by allowing replacement operations.

## 4 Incrementally Maintaining Penalties

This section presents how the penalties are maintained for two of the presented constraints, *AllDisjoint* and *MaxIntersect*. For the other three, *Partition* is similar to *AllDisjoint*, while *Cardinality* and *MaxWeightedSum* are rather straightforward to maintain. Since in local search one may need to perform many iterations, and since each iteration usually requires searching through a large neighbourhood, it is crucial that the penalty of a neighbouring configuration is computed efficiently. In order to do this, it is important to use *incremental algorithms* that, given a current configuration $k$, do not recompute from scratch the penalty of a neighbouring configuration $k'$, but rather compute the penalty with respect to the penalty of $k$ and the difference between $k$ and $k'$.

This technique is used, for instance, in [12, 22] where invariants are used to get efficient incremental algorithms from high-level, declarative descriptions. In this paper, the incrementality is achieved explicitly for each constraint, and we consider it to be future work to implement this in a more general and elegant way. The aim of this paper is to explore the usefulness of the proposed framework and penalty definitions for set constraints.

### 4.1 Incrementally Maintaining *AllDisjoint*

Recall the penalty (1) for an *AllDisjoint* constraint in Section 3.1. In order to maintain this incrementally, we use a table *count* of integers, indexed by the values in $U = \bigcup_{s \in S} U_s$, such that $count[d]$ is equal to the number of variables that contain $d$. Now, the sum in (1) is equal to $\sum_{d \in U}(count[d]-1)$ as it suffices to remove a value $d \in \bigcup_{s \in S} k(s)$ from all but one of the set variables in $\{s \in S \mid d \in k(s)\}$ in order to satisfy the constraint. This is easy to maintain incrementally given an atomic set operation.

### 4.2 Incrementally Maintaining *MaxIntersect*

Recall the penalty bound of (3) for a *MaxIntersect* constraint. In order to maintain this incrementally, we use the following two data structures: (i) A table *variables* indexed by the values in $U = \bigcup_{s \in S} U_s$, such that $variables[d]$ is the set of variables that $d$ is a member of; (ii) for each variable $s_i$, a table $s_i.intersects$ indexed by the values in $\{i+1, \ldots, n\}$ such that $s_i.intersects[j] = |k(s_i) \cap k(s_j)|$.

The sum in (3) is then equal to $\sum_{1 \leq i < j \leq n} max(s_i.intersects[j] - m, 0)$ and all this may be maintained incrementally in the following way, given an atomic set operation $o$. If $o = Add(k(s_i), d)$ then (i) add $s_i$ to $variables[d]$; (ii) for each variable $s_j$ in $variables[d]$ such that $j > i$: if $s_i.intersects[j] \geq m$ then increase the sum in (3) by 1; and (iii) for each variable $s_j$ in $variables[d]$ such that $j > i$: increase $s_i.intersects[j]$ by 1. If $o = Remove(k(s_i), d)$ then (i) remove $s_i$ from $variables[d]$; (ii) for each variable $s_j$ in $variables[d]$ such that $j > i$: if $s_i.intersects[j] > m$ then decrease the sum in (3) by 1; and (iii) for each variable $s_j$ in $variables[d]$ such that $j > i$: decrease $s_i.intersects[j]$ by 1.

Implementing these ideas with respect to the lower bound of (4) and Algorithm 1 is future work.

## 5 Applications

This section presents two well-known applications for constraint programming: the *Progressive Party Problem* and the *Social Golfers Problem*. They both have natural models based on set variables. They have previously been solved both using constructive and local search. See, for instance, the references [21, 10, 25, 13, 6, 24] and [3, 20, 18, 9], respectively. The constraints in Section 3 as well as the search algorithms were implemented in OCaml and the experiments were run on an Intel 2.4 GHz Linux machine with 512 MB memory.

### 5.1 The Progressive Party Problem (PPP)

The problem is to timetable a party at a yacht club. Certain boats are designated as hosts, while the crews of the remaining boats are designated as guests. The crew of a host boat remains on board throughout the party to act as hosts, while the crew of a guest boat together visits host boats over a number of periods. The crew of a guest boat must party at some host boat each period (constraint $c_1$). The spare capacity of any host boat is not to be exceeded at any period by the sum of the crew sizes of all the guest boats that are scheduled to visit it then (constraint $c_2$). Any guest crew can visit any host boat in at most one period (constraint $c_3$). Any two distinct guest crews can visit the same host boat in at most one period (constraint $c_4$).

**A Set-Based Model.** Let $H$ be the set of host boats and let $G$ be the set of guest boats. Furthermore, let $\mathit{capacity}(h)$ and $\mathit{size}(g)$ denote the spare capacity of host boat $h$ and the crew size of guest boat $g$, respectively. Let *periods* be the number of periods we want to find a schedule for and let $P$ be the set $\{1, \ldots, \mathit{periods}\}$. Now, let $s_{(h,p)}$, where $h \in H$ and $p \in P$, be a set variable containing the set of guest boats whose crews boat $h$ hosts during period $p$. Then the following constraints model the problem:

$$(c_1) : \forall p \in P : \mathit{Partition}(\{s_{(h,p)} \mid h \in H\}, G)$$
$$(c_2) : \forall h \in H : \forall p \in P : \mathit{MaxWeightedSum}(s_{(h,p)}, \mathit{size}, \mathit{capacity}(h))$$
$$(c_3) : \forall h \in H : \mathit{AllDisjoint}(\{s_{(h,p)} \mid p \in P\})$$
$$(c_4) : \mathit{MaxIntersect}(\{s_{(h,p)} \mid h \in H \ \ \& \ \ p \in P\}, 1)$$

**Solving the PPP.** If we are careful when defining an initial configuration and a neighbourhood for the PPP, we may be able to exclude some of its constraints. For instance, it is possible to give the variables $s_{(h,p)}$ an initial configuration and a neighbourhood that respect $c_1$. We can do this (i) by assigning random disjoint subsets of $G$ to each $s_{(h,p)}$, where $h \in H$, for each period $p \in P$, making sure that each $g \in G$ is assigned to some $s_{(h,p)}$ and (ii) by using a neighbourhood specifying that guests from a host boat $h$ are moved to another host boat $h'$ in the same period, and nothing else.

Algorithm 2 is the solving algorithm we used for the PPP. It takes the constant sets $P$, $G$, $H$, and the functions *capacity* and *size* as defined above as parameters, specifying an instance of the PPP, and returns a configuration $k$ for a CSP with respect to that instance. *MaxIter* and *MaxNonImproving* are additional arguments as described below. If $penalty(\langle V, D, C\rangle)(k) = 0$, then a solution was found within *MaxIter* iterations. The algorithm uses the notion of *conflict of a variable* (line 10), which, informally, is an estimate on how much a variable contributes to the total penalty of a set of constraints with respect to a configuration.

**Algorithm 2** Solving the PPP

1: **procedure** *solve_progressive_party*$(P, G, H, capacity, size)$
2: Initialise $\langle V, D, C\rangle$ w.r.t. $P$, $G$, $H$, *capacity*, and *size* to be a CSP $\in$ PPP
3: $iteration \leftarrow 0$, $non\_improving \leftarrow 0$, $best \leftarrow \infty$
4: $k \leftarrow \emptyset$, $tabu \leftarrow \emptyset$, $history \leftarrow \emptyset$
5: **for all** $p \in P$ **do** ▷ Initialise s.t. $c_1$ is respected
6: Add a random mapping $s_{(h,p)} \mapsto G'$, where $G' \subset G$, for each $h \in H$ to $k$
7: s.t. $penalty(Partition(\{s_{(h,p)} \mid h \in H\}, G))(k) = 0$
8: **while** $penalty(\langle V, D, C\rangle)(k) > 0$ & $iteration < MaxIter$ **do**
9: $iteration \leftarrow iteration + 1$, $non\_improving \leftarrow non\_improving + 1$
10: *choose* $s_{(h,p)} \in V$ s.t. $\forall s' \in V : conflict(s_{(h,p)}, C)(k) \geq conflict(s', C)(k)$
11: $N \leftarrow move(s_{(h,p)}, \{s_{(h',p)} \mid h' \in H \ \&\ h' \neq h\})(k)$
12: *choose* $k' \in N$ s.t. $\forall k'' \in N : penalty(\langle V, D, C\rangle)(k') \leq penalty(\langle V, D, C\rangle)(k'')$
13: and $((s_{(h',p)}, d, iteration) \notin tabu$ or $penalty(\langle V, D, C\rangle)(k') < best)$,
14: where $delta(k, k') = \{(s_{(h,p)}, \{d\}, \emptyset), (s_{(h',p)}, \emptyset, \{d\})\}$
15: $k \leftarrow k'$, $tabu \leftarrow tabu \cup \{(s_{(h',p)}, d, iteration + rand\_int(5, 40))\}$
16: **if** $penalty(\langle V, D, C\rangle)(k) < best$ **then**
17: $best \leftarrow penalty(\langle V, D, C\rangle)(k)$, $non\_improving \leftarrow 0$,
18: $history \leftarrow \{k\}$, $tabu \leftarrow \emptyset$
19: **else if** $penalty(\langle V, D, C\rangle)(k) = best$ **then**
20: $history \leftarrow history \cup \{k\}$
21: **else if** $non\_improving = MaxNonImproving$ **then**
22: $k \leftarrow$ a random element in *history*
23: $non\_improving \leftarrow 0$, $history \leftarrow \{k\}$, $tabu \leftarrow \emptyset$
24: **return** $k$

The algorithm starts by initialising a CSP for the PPP, necessary counters, bounds, and sets (lines $2 - 4$), as well as the variables of the problem (lines $5 - 7$). As long as the penalty is positive and a maximum number of iterations has not been reached, lines $8 - 23$ explore the neighbourhood of the problem in the following way. (i) Choose a variable $s_{(h,p)}$ with maximum conflict (line 10). (ii) Determine the neighbourhood of type *move* for $s_{(h,p)}$ with respect to the other variables in the same period (line 11). (iii) Move to a neighbour $k'$ that minimises the penalty (lines $12 - 14$).

In order to escape local minima it also uses a tabu list and a restarting component. The tabu list *tabu* is initially empty. When a move from a configuration $k$ to a configuration $k'$ is performed, meaning that for two variables $s_{(h,p)}$ and $s_{(h',p)}$, a value $d$ in $k(s_{(h,p)})$ is moved to $k(s_{(h',p)})$, the triple $(s_{(h'p)}, d, \mathit{iteration} + t)$ is added to *tabu*. This means that $d$ cannot be moved to $s_{(h',p)}$ again for the next $t$ iterations, where $t$ is a random number between 5 and 40 (empirically chosen). However, if such a move would imply the lowest penalty so far, it is always accepted (lines $13-15$). By abuse of notation, we let $(s, d, t) \notin \mathit{tabu}$ be false iff $(s, d, t') \in \mathit{tabu}\ \ \&\ \ t \leq t'$.

The restarting component (lines $16-23$) works in the following way. Each configuration $k$ such that $\mathit{penalty}(\langle V, D, C\rangle)(k)$ is at most the current lowest penalty is stored in the set *history* (lines $16-20$). If a number *MaxNonImproving* of iterations passes without any improvement to the lowest overall penalty, then the search is restarted from a random element in *history* (lines $21-23$). A similar restarting component was used in [13, 24] (saving one best configuration) and [6] (saving a set of best configurations), both for integer-domain models of the PPP.

### 5.2 The Social Golfers Problem (SGP)

In a golf club, there is a set of golfers, each of whom play golf once a week (constraint $c_1$) and always in $ng$ groups of size $ns$ (constraint $c_2$). The objective is to determine whether there is a schedule of $nw$ weeks of play for these golfers, such that there is at most one week where any two distinct players are scheduled to play in the same group (constraint $c_3$).

**A Set-Based Model.** Let $G$ be the set of golfers and let $s_{(g,w)}$ be a set variable containing the players playing in group $g$ in week $w$. Then the following constraints model the problem:

$(c_1) : \forall w \in 1 \ldots nw : \mathit{Partition}(\{s_{(g,w)} \mid g \in 1 \ldots ng\}, G)$
$(c_2) : \forall g \in 1 \ldots ng : \forall w \in 1 \ldots nw : \mathit{Cardinality}(s_{(g,w)}, ns)$
$(c_3) : \mathit{MaxIntersect}(\{s_{(g,w)} \mid i \in 1 \ldots ng\ \ \&\ \ w \in 1 \ldots nw\}, 1)$

**Solving the SGP.** Similar to the PPP, we need to define an initial configuration and a neighbourhood for the SGP. This, and a slightly changed tabu list, are the only changes in the algorithm compared to the one we used for the PPP, hence the algorithm for the SGP is not shown.

We choose an initial configuration $k$ and a neighbourhood that respects the constraints $c_1$ and $c_2$, i.e. that each golfer plays every week and that each group is of size $ns$. We do this (i) by assigning random disjoint subsets of size $ns$ of $G$ to each $s_{(g,w)}$ where $g \in 1 \ldots ng$ for each week $w \in 1 \ldots nw$ and (ii) by choosing the neighbourhood called *swap*, specifying the swap of two distinct golfers between a given group $g$ and another group $g'$ in the same week. Given such a swap of golfers between two different groups $s_{(g,w)}$ and $s_{(g',w)}$, what is now inserted in the tabu list are both $(s_{(g,w)}, d, t)$ and $(s_{(g',w)}, d, t)$ with $t$ being as for the PPP.

**Table 1.** Run times in seconds for the PPP. Mean run time of successful runs (out of 100) and number of unsuccessful runs (if any) in parentheses.

| *H/periods* (fails) | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| 1-12,16 | | | 1.2 | 2.3 | 21.0 |
| 1-13 | | | 7.0 | 90.5 | |
| 1,3-13,19 | | | 7.2 | 128.4 (4) | |
| 3-13,25,26 | | | 13.9 | 170.0 (17) | |
| 1-11,19,21 | 10.3 | 83.0 (1) | | | |
| 1-9,16-19 | 18.2 | 160.6 (22) | | | |

**Table 2.** Run times in seconds for the SGP. Mean run time of successful runs (out of 100) and number of unsuccessful runs (if any) in parentheses.

| *ng-ns-nw* | time | (fails) | *ng-ns-nw* | time | (fails) | *ng-ns-nw* | time | (fails) |
|---|---|---|---|---|---|---|---|---|
| 6-3-7 | 0.4 | | 6-3-8 | 215.0 | (76) | 7-3-9 | 138.0 | (5) |
| 8-3-10 | 14.4 | | 9-3-11 | 3.5 | | 10-3-13 | 325.0 | (35) |
| 6-4-5 | 0.3 | | 6-4-6 | 237.0 | (62) | 7-4-7 | 333.0 | (76) |
| 8-4-7 | 0.9 | | 8-4-8 | 290.0 | (63) | 9-4-8 | 1.7 | |
| 10-4-9 | 2.5 | | 6-5-5 | 101.0 | (1) | 7-5-5 | 1.3 | |
| 8-5-6 | 8.6 | | 9-5-6 | 0.9 | | 10-5-7 | 1.7 | |
| 6-6-3 | 0.2 | | 7-6-4 | 1.2 | | 8-6-5 | 18.6 | |
| 9-6-5 | 1.0 | | 10-6-6 | 3.7 | | 7-7-3 | 0.3 | |
| 8-7-4 | 4.9 | | 9-7-4 | 0.8 | | 10-7-5 | 3.4 | |
| 8-8-3 | 0.5 | | 9-8-3 | 0.6 | | 10-8-4 | 1.4 | |
| 9-9-3 | 0.7 | | 10-9-3 | 0.8 | | 10-10-3 | 1.1 | |

### 5.3 Results

Tables 1 and 2 show the experimental results for the PPP and SGP, respectively. For both, each entry in the table is the mean value of successful runs out of 100. The numbers in parentheses are the numbers of unsuccessful runs, if any, for that instance. We empirically chose $MaxIter = 500,000$ and $MaxNonImproving = 500$ for both applications. For the PPP, the instances are the same as in [25, 6, 24] and for the SGP, the instances are taken from [9]. For both applications, our results are comparable to, but not quite as fast as, the current best results ([6, 24] and [9] respectively) that we are aware of. We believe that they can be improved by using more sophisticated neighbourhoods and meta-heuristics, as well as by implementing the ideas in Section 3.3 for the *MaxIntersect* constraint.

## 6 Conclusion

We have proposed to use set variables and set constraints in local search. In order to do this, we have introduced a generic penalty scheme for (global) set constraints and used it to give incrementally maintainable penalty definitions for five such constraints. These were then used to model and solve two well-known combinatorial problems.

This research is motivated by the fact that set variables may lead to more intuitive and simpler problem models, providing the user with a richer set of tools, as well as more preserved structure in underlying solving algorithms such as the incremental algorithms for maintaining penalties: (global) set constraints

provide opportunities for hard-wired global reasoning that would otherwise have to be hand-coded each time for lower-level encodings of set variables.

In terms of related work, Localizer [12, 22], by Michel and Van Hentenryck, was the first modelling language to allow the definition of local search algorithms in a high-level, declarative way. It introduces invariants to obtain efficient incremental algorithms. It also stresses the need for globality by making explicit the invariants *distribute* and *dcount*.

In [10], Galinier and Hao use a similar scheme to ours for defining the penalty of a constraint in local search: they define as the penalty of a (global) constraint $c$ the minimum number of variables in $c$ that must change in order for it to be satisfied. Note, however, that this work is for integer variables only. Nareyek uses global constraints in [16] and argues that this is a good compromise between low-level CSP approaches, using only simple (e.g., binary) constraints, and problem-tailored local search approaches that are hard to reuse.

Comet [13], also by Van Hentenryck and Michel, is an object-oriented language tailored for the elegant modelling and solving of combinatorial problems. With Comet, the concept of differentiable object was introduced, which is an abstraction that reconciles incremental computation and global constraints. A differentiable object may for instance be queried to evaluate the effect of local moves. Comet also introduced abstractions for controlling search [23] and modelling using constraint-based combinators such as logical operators and reification [24]. Both Localizer and Comet support set invariants, but these are not used as variables directly in constraints.

Generic penalty definitions for constraints are useful also in the soft-constraints area. Petit *et al.* [17] use a similar penalty definition to the one of Galinier and Hao [10] as well as another definition where the primal graph of a constraint is used to determine its cost. This definition of cost is then refined by Petit and Beldiceanu in [5], where the cost is expressed in terms of graph properties [4]. Bohlin [6] also introduces a scheme built on the graph properties in [4] for defining penalties, which is used in his Composer library for local search. To our knowledge, none of these approaches considers set variables and set constraints.

Open issues exist as well. Other than fine-tuning the performance of our current prototype implementation, further (global) set constraints should be added. What impact will a change to the penalty of *MaxIntersect* with respect to Section 3.3 have? In what way should the penalties of the (global) set constraints in this paper be generalised to allow problems containing variables with several kinds of domains? For instance, it would be useful to be able to replace $m$ with an integer variable in the *Cardinality*, *MaxIntersect*, and *MaxWeightedSum* constraints, to allow negative weights in the latter, and to have a variable reference set in the *Partition* constraint.

Overall, our results are already very promising and motivate such further research.

**Acknowledgements.** This research was partially funded by Project C/1.246/HQ/JC/04 of EuroControl. We thank the referees for their useful comments.

## References


1. E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons Ltd., 1997.
2. F. Azevedo and P. Barahona. Applications of an extended set constraint solver. In *Proc. of the ERCIM / CompulogNet Workshop on Constraints*, 2000.
3. N. Barnier and P. Brisset. Solving the Kirkman's schoolgirl problem in a few seconds. In *Proc. of CP'02*, volume 2470 of *LNCS*, pages 477–491. Springer, 2002.
4. N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Proc. of CP'00*, volume 1894 of *LNCS*, pages 52–66. Springer-Verlag, 2000.
5. N. Beldiceanu and T. Petit. Cost evaluation of soft global constraints. In *Proc. of CPAIOR'04*, volume 3011 of *LNCS*, pages 80–95. Springer-Verlag, 2004.
6. M. Bohlin. Design and Implementation of a Graph-Based Constraint Model for Local Search. PhL thesis, Mälardalen University, Västerås, Sweden, April 2004.
7. P. Codognet and D. Diaz. Yet another local search method for constraint solving. In *Proc. of SAGA'01*, volume 2264 of *LNCS*, pages 73–90. Springer-Verlag, 2001.
8. K. Corrádi. Problem at Schweitzer competition. *Mat. Lapok*, 20:159–162, 1969.
9. I. Dotú and P. Van Hentenryck. Scheduling social golfers locally. In *Proc. of CPAIOR'05*, *LNCS*, Springer-Verlag, 2005.
10. P. Galinier and J.-K. Hao. A general approach for constraint solving by local search. In *Proc. of CP-AI-OR'00*.
11. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
12. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Proc. of CP'97*, volume 1330 of *LNCS*. Springer-Verlag, 1997.
13. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):101–110, 2002. *Proc. of OOPSLA'02*.
14. L. Michel and P. Van Hentenryck. Maintaining longest paths incrementally. In *Proc. of CP'03*, volume 2833 of *LNCS*, pages 540–554. Springer-Verlag, 2003.
15. T. Müller and M. Müller. Finite set constraints in Oz. In *Proc. of 13th Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 1997.
16. A. Nareyek. Using global constraints for local search. In *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 9–28. AMS, 2001.
17. T. Petit, J.-C. Régin, and C. Bessière. Specific filtering algorithms for over constrained problems. In *Proc. of CP'01*, volume 2293 of *LNCS*, Springer, 2001.
18. S. Prestwich. Supersymmetric modeling for local search. In *Proc. of 2nd International Workshop on Symmetry in Constraint Satisfaction Problems, at CP'02*.
19. J.-F. Puget. Finite set intervals. In *Proc. of CP'96 Workshop on Set Constraints*.
20. J.-F. Puget. Symmetry breaking revisited. In *Proc. of CP'02*, volume 2470 of *LNCS*, pages 446–461. Springer-Verlag, 2002.
21. B. M. Smith *et al.* The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.
22. P. Van Hentenryck and L. Michel. Localizer. *Constraints*, 5(1–2):43–84, 2000.
23. P. Van Hentenryck and L. Michel. Control abstractions for local search. In *Proc. of CP'03*, volume 2833 of *LNCS*, pages 65–80. Springer-Verlag, 2003.
24. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-based combinators for local search. In *Proc. of CP'04*, volume 3258 of *LNCS*. Springer-Verlag, 2004.
25. J. P. Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *LNCS*. Springer-Verlag, 1999.