Set Variables and Local Search

Magnus Ågren

Department of Information Technology
Uppsala University, Box 337, S – 751 05 Uppsala, Sweden
`agren@it.uu.se`

Abstract. Many combinatorial (optimisation) problems have natural models based on, or including, set variables and set constraints. This modelling device has been around for quite some time in the constraint programming area, and proved its usefulness in many applications. This paper introduces set variables and set constraints also in the local search area. It presents a way of representing set variables in the local search context, where we deal with concepts like transition functions, neighbourhoods, and penalty costs. Furthermore, some common set constraints and their penalty costs are defined. These constraints are later used to model three problems and some initial experimental results are reported.

1 Introduction

For some time now, set variables and set constraints have been around in the *Constraint Programming (CP)* area. Most of the popular CP systems of today have features for modelling problems using set variables, i.e., variables taking values that are subsets of some universe. Consider the work by Gervet [6, 7], Müller and Müller [13], and Puget [16].

Indeed, many problems have natural models using set variables. Classical examples include the set partitioning problem and the set covering problem. However, such problems also appear frequently as sub-problems in many real-life applications. Examples include airline crew rostering, tournament scheduling, time-tabling, and nurse rostering.

To be able to reason about these problems at the higher level that set variables mean may reduce the development time and increase the understanding of an application considerably. Since this is already known from the CP area, we believe that it should be put also in a local search framework.

Local search is becoming more and more popular as an alternative or complement to CP when it comes to tackling hard combinatorial problems such as the *Constraint Satisfaction Problem (CSP).* It has proven to be very efficient and often outperforms other techniques [4, 1, 15].

In recent years, there has been much research on the integration of CP techniques and local search, investigating such concepts as high declarativeness, incrementality, and global constraints (see [10, 14, 5, 3, 11, 17, 12, 2] for instance).

In the following, Section 2 presents the Finite Domain CSP in a local search setting. We define the concepts of configurations, neighbourhoods, and penalty costs. Section 3 generalises these concepts for Set CSPs. Then, in Section 4, we introduce some constraints on set variables. We present their semantics and their penalty costs. Section 5 briefly discusses the search component in local search, and some commonly used heuristics. Section 6 presents set-based models for three problems and Section 7 reports on computational results for one of these. Finally, Section 8 contains a discussion about the methods and the results, and Section 9 concludes the paper.

2 Local Search for Finite Domain CSPs

A local search algorithm for solving a combinatorial (optimisation) problem P starts from some initial *configuration* k of P and, using the *penalty costs* of P with respect to k and the *neighbours* of k, moves to neighbours having less penalty cost. The algorithm stops and returns the current configuration when an optimal penalty cost is obtained, or when some maximum number of iterations is reached. In this section, we define a class of problems, FDCSPs, that local search applies to. We also define the concepts above such as configurations, neighbours, and penalty costs for these problems.

Definition 1. *A* Finite Domain CSP (FDCSP) *is a triple* $\langle V_{fd}, D_{fd}, C\rangle$ *where*

- $V_{fd} = \{v_1, \ldots, v_n\}$ *is a finite set of variables.*
- $D_{fd} = \{D_1, \ldots, D_n\}$ *is a finite set of finite domains, each* D_i *containing the set of possible values for the corresponding variable* v_i*.*
- $C = \{c_1, \ldots, c_m\}$ *is a finite set of constraints, each* c_i *being defined on a subset of the variables in* V_{fd} *specifying the valid combinations of values for those.*

Definition 2. *Let* $P = \langle V_{fd}, D_{fd}, C\rangle$ *be an FDCSP. A* configuration *for* P *is a function* $k : V_{fd} \to D_1 \cup \cdots \cup D_n$ *with the condition that* $k(v_i) \in D_i$ *for all* $v_i \in V_{fd}$*.*

Example 1. Let $P = \langle\{v_1, v_2, v_3\}, \{D_1, D_2, D_3\}, \{c_1, c_2\}\rangle$ be an FDCSP where $D_1 = D_2 = D_3 = \{1, 2\}$. One possible configuration k for P is defined as $k(v_1) = k(v_2) = 1, k(v_3) = 2$ or equivalently as the set $k = \{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 2\}$.

Definition 3. *Let* K *denote the set of all possible configurations for an FDCSP* P*. A* transition function *for* P *is a function* $trans : K \to 2^K$*.*

Definition 4. *Let* P *be an FDCSP, let* k *be a configuration for* P*, and let* $trans$ *be a transition function for* P*. The* neighbourhood *of* P *with respect to* k *and* $trans$ *is the set of configurations* $trans(k)$*.*

Example 2. Consider P and k from Example 1. A possible neighbourhood $trans(k)$ of P with respect to some transition function $trans$ is the set of configurations $\{\{v_1 \mapsto 2, v_2 \mapsto 1, v_3 \mapsto 2\}, \{v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 2\}, \{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 1\}\}$.

Example 3. While the neighbourhood of Example 2 was specified given a particular configuration, we define more general neighbourhoods as shown below. These define a set of configurations given any configuration k of some FDCSP $\langle V_{fd}, D_{fd}, C\rangle$.

$$\begin{aligned}\mathit{Flip}(k) = \{k' \mid \exists v_i \in V_{fd} : \exists d \neq d' \in D_i : k - k' = \{v \mapsto d\} \ \& \\ k' - k = \{v \mapsto d'\}\}\end{aligned}$$

$$\begin{aligned}\mathit{Swap}(k) = \{k' \mid \exists v_i \neq v_j \in V_{fd} : \exists d \neq d' \in D_i : \exists d \neq d' \in D_j : \\ k - k' = \{v_i \mapsto d, v_j \mapsto d'\} \ \& \ k' - k = \{v_i \mapsto d', v_j \mapsto d\}\}\end{aligned}$$

A member of the set $\mathit{Flip}(k)$ differs from k in the value of one variable. A member of the set $\mathit{Swap}(k)$ differs from k in the values of two variables, where their values have been swapped. The neighbourhood given in Example 2 is an example of a *Flip* neighbourhood.

Definition 5. *Let $P = \langle V_{fd}, D_{fd}, C\rangle$ be an FDCSP and remember that K denotes the set of all possible configurations for P. The* penalty cost function *of a constraint $c \in C$ is a function $cost(c) : K \to \mathbb{N}$.*

Definition 6. *Let $P = \langle V_{fd}, D_{fd}, C\rangle$ be an FDCSP and let k be a configuration for P. The* penalty cost *of P with respect to k is the sum:*

$$\sum_{c \in C} cost(c)(k)$$

Example 4. Consider once again P from Example 1 and let c_1 and c_2 be the binary constraints $v_1 = v_2$ and $v_2 \neq v_3$ respectively. Let the penalty cost functions of c_1 and c_2 be defined as:

$$cost(c_1) = \begin{cases} 0, & \mathit{if}\ v_1 = v_2 \\ 1, & \mathit{otherwise} \end{cases} \quad \text{and } cost(c_2) = \begin{cases} 0, & \mathit{if}\ v_2 \neq v_3 \\ 1, & \mathit{otherwise} \end{cases}$$

Now, the penalty costs of P with respect to the different configurations

$k_1 = \{v_1 \mapsto 2, v_2 \mapsto 1, v_3 \mapsto 2\}$, $k_2 = \{v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 2\}$, and
$k_3 = \{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 1\}$

are $cost(c_1)(k_1)+cost(c_2)(k_1) = 1+0 = 1$, $cost(c_1)(k_2)+cost(c_2)(k_2) = 1+1 = 2$, and $cost(c_1)(k_3) + cost(c_2)(k_3) = 0 + 1 = 1$ respectively.

3 Local Search for Set CSPs

Let us now generalise the concepts introduced in Section 2 for another, extended, class of problems.

Definition 7. *A* Set CSP (SCSP) *is a triple $\langle\langle V_{fd}, V_{set}\rangle, \langle D_{fd}, D_{set}\rangle, C\rangle$ where*

– *$V_{fd} = \{v_1, \ldots, v_m\}$ and $V_{set} = \{s_1, \ldots, s_n\}$ are finite sets of variables. A member $v \in V_{fd}$ denotes a* finite domain variable*, and a member $s \in V_{set}$ denotes a* finite set variable*.*
– *$D_{fd} = \{D_1, \ldots, D_m\}$ is a finite set of finite domains, each D_i containing the set of possible values for the corresponding variable v_i.*
– *$D_{set} = \{U_1, \ldots, U_n\}$ is a finite set of finite sets, each U_i being a* universe*, defined below, for the corresponding variable s_i.*
– *$C = \{c_1, \ldots, c_m\}$ is a finite set of constraints, each $c_i \in C$ being defined on a subset of the variables in $V_{fd} \cup V_{set}$, specifying the valid combinations of values for those.*

Definition 8. *Let s be a finite set variable. The* universe *of s is a finite set U such that $s \subseteq U$.*

Definition 9. *Let s be a finite set variable and let U be the universe of s. The* characteristic function *of s is the function $\chi_s : U \to \{0, 1\}$ defined as:*

$$\chi_s(v) = \begin{cases} 1, & \textit{if } v \in s \\ 0, & \textit{otherwise} \end{cases}$$

Definition 10. *Let $P = \langle\langle V_{fd}, V_{set}\rangle, \langle D_{fd}, D_{set}\rangle, C\rangle$ be an SCSP. A* configuration *for P is a pair of functions*

$$\langle k_{fd} : V_{fd} \to D_1 \cup \cdots \cup D_m, k_{set} : V_{set} \to \chi\rangle$$

where χ is a set of characteristic functions for the variables in V_{set}. A condition for k_{fd} is that $k_{fd}(v_i) \in D_i$ for all $v_i \in V_{fd}$. Similarly, a condition for k_{set} is that $\{u \mid k_{set}(s_i)(u) = 1\} \subseteq U_i$ for all $s_i \in V_{set}$.

From the above, we may conclude that the *value* of a finite set variable s with respect to its universe U and a current configuration $\langle k_{fd}, k_{set}\rangle$ is the set $\{u \in U \mid k_{set}(s)(u) = 1\}$. Let us now look at an example of a configuration for a specific SCSP.

Example 5. Consider an SCSP $P = \langle\langle\{v_1, v_2\}, \{s_1, s_2\}\rangle, \langle\{D_1, D_2\}, \{U_1, U_2\}\rangle, C\rangle$ where $D_1 = D_2 = \{1, 2\}$ and $U_1 = U_2 = \{a, b, c\}$. Let χ_1 and χ_2 be the sets of mappings $\{a \mapsto 0, b \mapsto 0, c \mapsto 1\}$ and $\{a \mapsto 1, b \mapsto 1, c \mapsto 0\}$ respectively.

One possible configuration $\langle k_{fd}, k_{set}\rangle$ for P is defined as $k = k_{fd}(v_1) = 1, k_{fd}(v_2) = 1$, $k_{set}(s_1) = \chi_1$, $k_{set}(s_2) = \chi_2$ or equivalently as the pair of sets $\langle\{v_1 \mapsto 1, v_2 \mapsto 1\}, \{s_1 \mapsto \chi_1, s_2 \mapsto \chi_2\}\rangle$. Under this configuration, the values of the variables in P are $v_1 = 1$, $v_2 = 1$, $s_1 = \{c\}$, and $s_2 = \{a, b\}$.

Example 6. Consider P and k from Example 5 and let *trans* be a transition function for P. A possible neighbourhood for P with respect to k and *trans* is the set of configurations $\mathit{trans}(k) = \{\langle\{v_1 \mapsto 1, v_2 \mapsto 1\}, \{s_1 \mapsto \chi_1, s_2 \mapsto \chi_2\}\rangle, \langle\{v_1 \mapsto 1, v_2 \mapsto 1\}, \{s_1 \mapsto \chi_3, s_2 \mapsto \chi_4\}\rangle\}$ where

$\chi_1 = \{a \mapsto 1, b \mapsto 0, c \mapsto 0\}$, $\chi_2 = \{a \mapsto 0, b \mapsto 1, c \mapsto 1\}$,
$\chi_3 = \{a \mapsto 0, b \mapsto 1, c \mapsto 0\}$, and $\chi_4 = \{a \mapsto 1, b \mapsto 0, c \mapsto 1\}$.

This neighbourhood does not affect the finite domain variables of the problem but flips any two values in the finite set variables s_1 and s_2.

Example 7. As we did for FDCSPs, we will define general neighbourhoods also for SCSPs. Let us do this for an SCSP $\langle\langle V_{fd}, V_{set}\rangle, \langle D_{fd}, D_{set}\rangle, C\rangle$ and a configuration $\langle k_{fd}, k_{set}\rangle$.

$$\begin{aligned} \mathit{trans}(\langle k_{fd}, k_{set}\rangle) = \{\langle k_{fd}, k\rangle \mid\ & k_{set} - k = \{s \mapsto \chi_s\} \ \ \& \\ & k_{set}(s) - k(s) = \{u \mapsto 1, u' \mapsto 0\} \ \ \& \\ & k(s) - k_{set}(s) = \{u \mapsto 0, u' \mapsto 1\}\} \end{aligned}$$

The above neighbourhood leaves k_{fd} untouched. k_{set} and k differ in exactly one mapping $s \mapsto \chi_s$. For this variable, the characteristic function $k_{set}(s)$ differs from $k(s)$ in exactly two values, say u and u'. For k_{set}, $u \in s$ and $u' \notin s$, while the opposite is true for k. We have thus defined a neighbourhood that flips one value in one finite set variable.

Let us generalise the neighbourhood presented in the above example and also present additional useful neighbourhoods for SCSPs. We focus here on neighbourhoods that change the finite set variables of the problem and do nothing to the finite domain variables. One could of course also consider neighbourhoods that change both sets of variables.

Example 8. Roughly, there are five different things one can do with a finite set variable: 1. Add p values to it taken from its complement, increasing its cardinality. 2. Remove p values from it, decreasing its cardinality. 3. Flip p values in it with values from its complement. 4. Swap p values in it to values in another finite set variable. 5. Any combination of the previous. The first four of these, with respect to the SCSP and configuration of Example 7, are shown below.

$$\begin{aligned} \mathit{AddP}(\langle k_{fd}, k_{set}\rangle) = \{\ \langle k_{fd}, k\rangle \mid\ & k_{set} - k = \{s \mapsto \chi_s\} \ \ \& \\ & k_{set}(s) - k(s) = \{v_1 \mapsto 0, \dots, v_p \mapsto 0\} \ \ \& \\ & k(s) - k_{set}(s) = \{v_1 \mapsto 1, \dots, v_p \mapsto 1\}\} \end{aligned}$$

$$\begin{aligned} \mathit{RemoveP}(\langle k_{fd}, k_{set}\rangle) = \{\ \langle k_{fd}, k\rangle \mid\ & k_{set} - k = \{s \mapsto \chi_s\} \ \ \& \\ & k_{set}(s) - k(s) = \{v_1 \mapsto 1, \dots, v_p \mapsto 1\} \ \ \& \\ & k(s) - k_{set}(s) = \{v_1 \mapsto 0, \dots, v_p \mapsto 0\}\} \end{aligned}$$

$$\begin{aligned} \mathit{FlipP}(\langle k_{fd},\ k_{set}\rangle) = \{\langle k_{fd}, k\rangle \mid\ & k_{set} - k = \{s \mapsto \chi_s\} \ \ \& \\ & k_{set}(s) - k(s) = \{v_1 \mapsto 1, \dots, v_p \mapsto 1, v'_1 \mapsto 0, \dots, v'_p \mapsto 0\} \ \ \& \\ & k(s) - k_{set}(s) = \{v_1 \mapsto 0, \dots, v_p \mapsto 0, v'_1 \mapsto 1, \dots, v'_p \mapsto 1\}\} \end{aligned}$$

$$\begin{aligned} \mathit{SwapP}(\langle k_{fd},\ k_{set}\rangle) = \{\langle k_{fd}, k\rangle \mid\ & k_{set} - k = \{s \mapsto \chi_s, s' \mapsto \chi_{s'}\} \ \ \& \\ & k_{set}(s) - k(s) = \{v_1 \mapsto 1, \dots, v_p \mapsto 1, v'_1 \mapsto 0, \dots, v'_p \mapsto 0\} \ \ \& \\ & k_{set}(s') - k(s') = \{v_1 \mapsto 0, \dots, v_p \mapsto 0, v'_1 \mapsto 1, \dots, v'_p \mapsto 1\} \\ & k(s) - k_{set}(s) = \{v_1 \mapsto 0, \dots, v_p \mapsto 0, v'_1 \mapsto 1, \dots, v'_p \mapsto 1\} \\ & k(s') - k_{set}(s') = \{v_1 \mapsto 1, \dots, v_p \mapsto 1, v'_1 \mapsto 0, \dots, v'_p \mapsto 0\}\} \end{aligned}$$

Note that the neighbourhood *FlipP* is a generalisation of the one presented in Example 7. Also, while these are generic neighbourhoods, applicable to any model of an SCSP, one could also think of more problem specific neighbourhoods that depend on the actual model. This is discussed further in Section 6.4.

4 Constraints and Penalty Costs

In this section, we present five constraints on finite set variables: the *AllDisjoint* constraint, the *Cardinality* constraint, the *Intersection* constraint, the *MaxSum* constraint, and the *Partition* constraint. We discuss their semantics and penalty costs at a high and abstract level.

Concerning penalty costs, we have followed the approach taken for instance in [5]; the penalty cost for a constraint c is, whenever possible, the least number of values that must change in order to satisfy c. This means that, when there is a choice between adding values to, removing values from, or swapping values between the variables in c, we will assume the operation resulting in the minimal cost.

4.1 The AllDisjoint Constraint.

The constraint $\mathit{allDisjoint}(\{s_1, \ldots, s_n\})$, where $\{s_1, \ldots, s_n\}$ is a set of finite set variables, expresses that all these variables are disjoint, i.e., $\forall i \neq j \in 1 \ldots n : s_i \cap s_j = \emptyset$.

The penalty cost for an $\mathit{allDisjoint}(\{s_1, \ldots, s_n\})$ constraint is equal to the number of values that must be removed from any of the variables in $\{s_1, \ldots, s_n\}$ in order to make them all disjoint. This is equal to the sum

$$\sum_{v \in s_1 \cup \cdots \cup s_n} (v_{occ} - 1) \tag{1}$$

where v_{occ} is the number of variables in $\{s_1, \ldots, s_n\}$ that contain v. Clearly, we need to remove a value from $n-1$ finite set variables if it is contained in n finite set variables in order to satisfy the constraint.

Next, let us show that a satisfied *allDisjoint* constraint has zero penalty cost. A satisfied instance $\mathit{allDisjoint}(\{s_1, \ldots, s_n\})$ of this constraint has the property that $\forall i \neq j \in 1 \ldots n : s_i \cap s_j = \emptyset$. This implies that for each $v \in s_1 \cup \cdots \cup s_n$ we have that $v_{occ} = 1$. Hence, the equation (1) above must be equal to 0.

The following example illustrates the above. Assume that $s_1 = \{1, 2, 3\}$, $s_2 = \{1, 2, 4\}$, and $s_3 = \{1, 4\}$. The values 1, 2, 3, and 4, that comprise the union of s_1, s_2, and s_3, appear in 3, 2, 1, and 2 variables respectively. Thus, the above sum and the penalty cost for $\mathit{allDisjoint}(\{s_1, s_2, s_3\})$ is $(3-1)+(2-1)+(1-1)+(2-1) = 4$.

4.2 The Cardinality Constraint.

The constraint $\mathit{card}(s, v_{min}, v_{max})$, where s is a finite set variable and v_{min} and v_{max} are finite integer domain variables, expresses that the cardinality of s is in the range $v_{min} \ldots v_{max}$.

The penalty cost for a $\mathit{card}(s, v_{min}, v_{max})$ constraint is equal to the number of values that must be added to or removed from s so that its cardinality is in the range $v_{min} \ldots v_{max}$. This is calculated as follows. If $|s|$ is less than v_{min}, the penalty cost of the constraint is $v_{min} - |s|$. If $|s|$ is larger than v_{max}, the penalty cost of the constraint is $|s| - v_{max}$. Otherwise, the penalty cost of the constraint is 0. Hence

$$CardCost = \begin{cases} v_{min} - |s|, \ \mathit{if} \ |s| < v_{min} \\ |s| - v_{max}, \ \mathit{if} \ |s| > v_{max} \\ 0, \ \mathit{otherwise} \end{cases} \tag{2}$$

denotes the penalty cost of this constraint. Now, to see that a satisfied instance of this constraint has zero penalty cost is trivial.

Now, assume that the current value of s is $\{1, 2, 3, 4, 5\}$, that $v_{min} = 1$, and that $v_{max} = 3$. Since the cardinality of s is 5 and $v_{max} = 3$, we need to remove at least $5 - 3 = 2$ values from s in order to make the constraint satisfied.

4.3 The Intersection Constraint.

The constraint $\mathit{intersect}(s_1, s_2, s)$, where s_1, s_2, and s are finite set variables, expresses that the intersection of s_1 and s_2 is equal to s, i.e., that $s_1 \cap s_2 = s$.

The penalty cost for an $\mathit{intersect}(s_1, s_2, s)$ constraint is equal to the least number of values that must be removed from, added to, or swapped between any of the variables s_1, s_2, and s so that $s_1 \cap s_2 = s$. This is calculated in four steps as follows. First, the set of values that are in s_1 and s_2 but not in s is identified. This set should be removed from both s_1 and s_2, or added to s. Second, the set of values that are in s but not in any of s_1 and s_2 is identified. This set should be removed from s, or added to both s_1 and s_2. Third, the set of values that are in s and s_1 but not in s_2 is identified. This set should be added to s_2, or removed from s. Fourth, the set of values that are in s and s_2 but not in s_1 is identified. This set should be added to s_1, or removed from s. From this, we obtain the following sum expressing the penalty cost for the constraint:

$$|(s_1 \cap s_2) - s| + |s - (s_1 \cup s_2)| + |(s \cap s_1) - s_2| + |(s \cap s_2) - s_1| \tag{3}$$

In fact, we can improve on the above by observing the following. For the identified set of values, say s_a and s_b, in the first and second step, we have decided to respectively add these to and remove these from s when calculating the penalty cost. Since s_b is to be removed from s and s_a is to be added to s, we may instead swap values from s_b to values in s_a. This is true for the first and third step and the first and fourth step as well. By doing this whenever possible, i.e., for a maximum of $|s_a| = |(s_1 \cap s_2) - s|$ values, we obtain the following new expression for the penalty cost:

$$\begin{aligned} &|(s_1 \cap s_2) - s| + |s - (s_1 \cup s_2)| + |(s \cap s_1) - s_2| + |(s \cap s_2) - s_1| - \\ &\mathit{min}(|(s_1 \cap s_2) - s|, |(s - (s_1 \cup s_2)) \cup ((s \cap s_1) - s_2) \cup ((s \cap s_2) - s_1)|) \end{aligned} \tag{4}$$

Now, let us show that a satisfied instance $\mathit{intersect}(s_1, s_2, s)$ of this constraint has zero penalty cost. For such an instance, we have that $s_1 \cap s_2 = s$. In the expression (4) above, the first term is 0 since all elements in s appear also in the intersection of s_1 and s_2. The second term is 0 since all elements in s appear also in s_1 or s_2. The third term is 0 since the intersection of s and s_1 cannot contain an element that is not in s_2. A similar reasoning holds for the fourth term. The fifth term is also 0 by the above and hence we obtain a total penalty cost of 0.

As an example, assume that $s_1 = \{1,2,3,4,5\}$, $s_2 = \{4,5,6,7,8\}$, and $s = \{1,2,3,6,7\}$. The values 4 and 5 in the intersection of s_1 and s_2 are not in s. Also, the values 1, 2, and 3 are in the intersection of s and s_1 but are not in s_2. Similarly, the values 6 and 7 are in the intersection of s and s_2 but are not in s_1. Finally, the fifth term in expression (4) evaluates to 2. This results in a penalty cost of $2 + 0 + 3 + 2 - 2 = 5$.

4.4 The MaxSum Constraint.

Let s be a finite set variable, $\mathit{weight} : U \to \mathbb{N}$ a weight function from the universe U of s to the integers, and m a finite integer domain variable. The constraint $\mathit{maxSum}(s, \mathit{weight}, m)$ expresses that the sum $\sum_{u \in s} \mathit{weight}(u)$ is less than or equal to m.

The penalty cost for a $\mathit{maxSum}(s, \mathit{weight}, m)$ constraint, where we let Sum denote the sum $\sum_{u \in s} \mathit{weight}(u)$, is equal to the least number of values that must be removed from s in order for Sum to be less than or equal to m. This penalty cost is calculated by finding the minimal subset s' of values of s such that the sum $\sum_{u' \in s'} \mathit{weight}(u')$ is larger than or equal to the difference between Sum and m. By removing the set s' from s we will obtain a new sum Sum' that is less than or equal to m. Hence, the expression:

$$\left| \mathit{min}(\{s' \subseteq s | \sum_{u' \in s'} \mathit{weight}(u') \geq \left(\sum_{u \in s} \mathit{weight}(u) \right) - m\}) \right| \tag{5}$$

denotes the penalty cost for this constraint, where $\mathit{min}(\{s_1, \ldots, s_n\})$ denotes a set s_i with minimal cardinality among $s_1, \ldots, s_n$.

Now, let us show that a satisfied instance $\mathit{maxSum}(s, \mathit{weight}, m)$ of this constraint has zero penalty cost. For such an instance, we have that the sum $\sum_{u \in s} \mathit{weight}(u)$ is less than or equal to m. This means that the difference d between these in expression (5) is less than or equal to 0. Then we need to find the minimal subset s' of s such that $\sum_{u' \in s'} \mathit{weight}(u') \geq d$. Clearly, the least such subset of s is $\emptyset$ and $|\emptyset| = 0$.

As an example, let $s = \{1,2,3\}$, $\mathit{weight}(1) = 2$, $\mathit{weight}(2) = \mathit{weight}(3) = 1$, and $m = 2$. Then the sum $\sum_{u \in s} \mathit{weight}(u) = 4$. In order to satisfy this constraint, the least subset of values to remove from s is $\{1\}$ since $\mathit{weight}(1) = 2 \geq 4 - 2$ and there is no smaller subset $s' \subseteq s$ such that $\sum_{u \in s'} \mathit{weight}(u) \geq 2$. Hence the penalty cost of this constraint is 1.

4.5 The Partition Constraint.

The constraint $\mathit{partition}(\{s_1, \ldots, s_n\}, s)$, where $\{s_1, \ldots, s_n\}$ is a set of finite set variables and s is a finite set variable, expresses that the variables in $\{s_1, \ldots, s_n\}$

are all disjoint, i.e., that $\forall i \neq j \in 1 \ldots n : s_i \cap s_j = \emptyset$, and that their union is equal to s, i.e., that $s_1 \cup \cdots \cup s_n = s$. Note that this definition of a partition allows one or more variables in $\{s_1, \ldots, s_n\}$ to be empty.

The penalty cost for a *partition*$(\{s_1, \ldots, s_n\}, s)$ constraint is equal to the number of values that must be removed from any variable in $\{s_1, \ldots, s_n\}$ in order to make them all disjoint plus the number of values that must be removed from, added to, or swapped between any of the variables in $\{s, s_1, \ldots, s_n\}$ so that the union $s_1 \cup \cdots \cup s_n$ is equal to s.

We start by calculating the penalty cost for the overlapping values among the variables $s_1, \ldots, s_n$. This is done in the same way as for the *allDisjoint* constraint, i.e., the expression (1) denotes this penalty cost. Next, we calculate the penalty cost for the values in $s_1 \cup \cdots \cup s_n$ that are not in s. These values must be removed from $s_1 \cup \cdots \cup s_n$, or added to s. Finally, we calculate the penalty cost for the values in s that are not in $s_1 \cup \cdots \cup s_n$. These values must be removed from s, or added to $s_1 \cup \cdots \cup s_n$. Now, by following the same reasoning as for the *intersect* constraint, we see that a number of values in s and $s_1 \cup \cdots \cup s_n$ can be swapped directly, giving us a subtrahend in our penalty cost expression. Hence, the expression:

$$\begin{gathered}\left(\sum_{u \in s_1 \cup \cdots \cup s_n} (v_{occ} - 1)\right) + |(s_1 \cup \cdots \cup s_n) - s| + |s - (s_1 \cup \cdots \cup s_n)| - \\ min(|(s_1 \cup \cdots \cup s_n) - s|, |s - (s_1 \cup \cdots \cup s_n)|)\end{gathered} \tag{6}$$

denotes the penalty cost for the constraint.

Now we show that a satisfied instance *partition*$(\{s_1, \ldots, s_n\}, s)$ of this constraint has zero penalty cost. We know from the *allDisjoint* constraint that, for a satisfied instance, the first term in expression (6) above is 0. For the second and third term, note that for a satisfied instance of this constraint we have that $s_1 \cup \cdots \cup s_n = s$. The difference between two equal sets is the empty set $\emptyset$. Hence, the last two terms are 0, giving us a total penalty cost of 0.

As an example, assume that $s_1 = \{1, 2, 3\}, s_2 = \{2, 3, 4\}, s_3 = \{2, 5\}$, and that $s = \{4, 5, 6, 7\}$. First, in order for s_1, s_2, and s_3 to be disjoint, we need to remove the value 2 from two of the variables in $\{s_1, s_2, s_3\}$ and the value 3 from one of the variables in $\{s_1, s_2\}$. Second, in order for $s_1 \cup s_2 \cup s_3$ to be a subset of s, we need to remove 1, 2, and 3 from that union, or add these values to s. Third, in order for s to be a subset of $s_1 \cup s_2 \cup s_3$, we need to remove 6 and 7 from s, or add these values to the union. Since two of the values in the second and third steps may be swapped directly, we obtain a subtrahend of 2. This gives us a total penalty cost of $3 + 3 + 2 - 2 = 6$.

5 Search and Heuristics

One crucial point when it comes to local search is the ability to avoid getting stuck in local optima. If nothing is done to avoid this, the algorithm will probably perform very badly. Much research has gone into this area and some approaches are different randomisation algorithms, simulated annealing, and tabu search. *Randomisation algorithms* have, in addition to the penalty cost of the current configuration and its neighbours, some random-based way of choosing the next configuration. This means that the algorithm does not always have to move to a configuration having less penalty cost compared to the current one. *Simulated annealing* [9] is actually a variant of such a randomisation algorithm, in which the probability of choosing a degrading configuration changes over time. This method was derived from the process of growing a crystal in thermodynamics and has proven to be very efficient.

In this paper we have used *tabu search* [8] in our experiments. Such an algorithm may also move to worse configurations with respect to their penalty costs. When the next move is considered, a configuration with minimal penalty cost among the neighbours is chosen, even though this penalty cost may be larger than the penalty cost of the current configuration. In order to avoid a cyclic behaviour the method uses a memory called *tabu list*. This list stores configurations from a number, the *tabu tenure*, of previous iterations and any configuration contained in it is not allowed.

There are many ways to represent the tabu list. We have chosen the following simple approach for the finite set variables. For each value $u \in U$, where U is the universe of a finite set variable s, we keep a counter that denotes the number of future iterations where u is tabu. When s is instantiated to a particular value, say given by the characteristic function χ_s, the counters for all elements in the set $\{u \in U \mid \chi_S(u) = 1\}$ are set to the tabu tenure value. At each iteration, the counters for all values in U greater than 0 are decreased by 1.

6 Example Problems

This section presents possible models using finite set variables and the constraints presented in Section 4 for three different problems: the progressive party problem, the social golfer problem, and the minimal intersection subset problem. It demonstrates the usefulness of having finite set variables as a modelling device.

6.1 The Progressive Party Problem

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 (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 (c_2). Any guest crew can visit any host boat in at most one period (c_3). Any two distinct guest crews can visit the same host boat in at most one period (c_4).

Model. Let $H = \{h_1, \dots, h_m\}$ be the set of host boats and let $G = \{g_1, \dots, g_n\}$ be the set of guest boats. Furthermore, let *capacity*(h) and *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 $P = 1 \dots \mathit{periods}$ be the range of periods. Now, let $p_{(h,j)}$ be a finite set variable containing the set of guest boats whose crews boat h hosts at time period j. Then the following constraints model the problem:

$(c_1) : \forall j \in P : \mathit{partition}(\{p_{(h_1,j)}, \dots, p_{(h_m,j)}\}, G)$
$(c_2) : \forall h \in H : \forall j \in P : \mathit{maxSum}(p_{(h,j)}, \mathit{size}, \mathit{capacity}(h))$
$(c_3) : \forall h \in H : \forall j \neq j' \in P : \mathit{intersect}(p_{(h,j)}, p_{(h,j')}, \emptyset)$
$(c_4) : \forall h \neq h' \in H : \forall j \neq j' \in P : \mathit{intersect}(p_{(h,j)}, p_{(h',j')}, s_{(h,j,h',j')})$ & $\mathit{card}(s_{(h,j,h',j')}, 0, 1)$

where the introduced finite set variables $s_{(h,j,h',j')}$ in (c_4) have G as universe.

6.2 The Social Golfer Problem

In a golf club, there are n players, each of whom play golf once a week (c_1) and always in g groups of size s (c_2), hence $n = gs$. The objective is to determine whether there is a schedule of w 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 (c_3).

Model. Let $P = \{p_1, \dots, p_n\}$ be the set of golfers. Let $g_{(i,j)}$ be a finite set variable containing the players playing in group i in week j. Then the following constraints model the problem:

$(c_1) : \forall j \in 1 \dots w : \mathit{partition}(\{g_{(1,j)}, \dots, g_{(g,j)}\}, P)$
$(c_2) : \forall i \in 1 \dots g : \forall j \in 1 \dots w : \mathit{card}(g_{(i,j)}, s, s)$
$(c_3) : \forall i, i' \in 1 \dots g : \forall j \neq j' \in 1 \dots w : \mathit{intersect}(g_{(i,j)}, g_{(i',j')}, s_{(i,i',j,j')})$ & $\mathit{card}(s_{(i,i',j,j')}, 0, 1)$

where the introduced finite set variables $s_{(i,i',j,j')}$ in (c_3) have P as universe.

6.3 The Minimal Intersection Subset Problem

Assume a set R of r elements. The objective is to find $k \geq 2$ subsets of R of size p (c_1) such that the size of the largest intersection between any two different subsets is minimised (c_2, c_3).

Model. Let $R = \{1, \dots, r\}$ and let r_i be a finite set variable denoting the ith subset of R. Then the following constraints model the problem and the objective is to minimise δ:

$(c_1) : \forall i \in 1 \dots k : \mathit{card}(r_i, p, p)$

$(c_2) : \forall i \neq j \in 1 \dots k : \mathit{intersect}(r_i, r_j, s_{(i,j)})$ & $\mathit{card}(s_{(i,j)}, \delta_{(i,j)}, \delta_{(i,j)})$
$(c_3) : \mathit{max}(\{\delta_{(i,j)} | i \neq j \in 1 \dots k\}, \delta)$

In the above, the introduced finite set variables $S_{(i,j)}$ have R as their universe, the introduced finite integer domain variables in $\{\delta\} \cup \{\delta_{i,j} | i \neq j \in 1 \dots k\}$ can take values in the domain $0 \dots r$, and $\mathit{max}(\mathit{Set}, m)$ is satisfied iff m takes the same value as the maximum element in Set.

6.4 Problem-Specific Neighbourhoods

The models presented above respect any choice of neighbourhood for the finite set variables of the problems. By this, we mean that one may add, remove, flip, or swap values in the variables when defining a neighbourhood and still be sure that a solution to the model is a solution to the modelled problem.

One could also think of more neighbourhood-specific models, i.e., models that do not allow all possible neighbourhoods. This means that the problems can be modelled using fewer constraints, allowing for more efficient solving.

For example, the model for the social golfer problem, presented in Section 6.2, could also be modelled by dropping the constraints in (c_2). These constraints force each finite set variable $g_{(i,j)}$ to have cardinality s, i.e., they force the group size constraint. If these constraints are not added to the model, care must be taken when defining neighbourhoods. In this case any neighbourhood that does not change the size of the finite set variables, and an initial assignment respecting (c_2), would be valid. In addition to this, one could also replace the *partition* constraint in (c_1) by an *allDisjoint* constraint. If this is done, we need to make sure that any chosen neighbourhood keeps the property that the union of all the finite set variables in each week is equal to the total set of golfers. Of course, the initial assignment would also have to respect this property. Actually, we could even drop the *partition* constraint completely if we use a neighbourhood that respects it. For instance, a *SwapP* neighbourhood that only swaps values between variables in the same week would be a valid one.

7 Results

Due to lack of space we only present experimental results for the social golfer problem. We compared our set based model for this problem with a model based on finite domain variables. Both models were implemented in Objective Caml (`http://www.ocaml.org`) and the results are displayed in Table 1.

The finite domain model is based on each group in each week being modelled as a list of size s of finite domain variables. Each such variable has the domain $1 \dots n$, where n is the total number of golfers, and relevant finite domain constraints for modelling the problem are stated. For this model, we used the neighbourhood *Flip* as was given in Example 3 of Section 2, i.e., the neighbourhood where the value of any variable is flipped to another value in its domain.

For the finite set variables in the set based model we used the neighbourhood *SwapP* as was given in Example 8 of Section 3, i.e., the neighbourhood where

Table 1. Experimental results on the social golfer problem. Numbers displayed in **bold** correspond to the set based approach.

g-*s*-*w*	Iterations		Solved		Improvement	*g*-*s*-*w*	Iterations		Solved		Improvement
3-3-2	7	**1**	10	**10**	7.0	5-3-5	98	**10**	10	**10**	9.8
3-3-3	12	**2**	10	**10**	6.0	5-3-6	216	**68**	1	**4**	3.2
3-3-4	32	**4**	10	**10**	8.0	5-3-7	?	**136**	0	**4**	?
4-3-2	9	**1**	10	**10**	9.0	5-4-2	16	**3**	10	**10**	5.3
4-3-3	18	**3**	10	**10**	6.0	5-4-3	41	**6**	10	**10**	6.8
4-3-4	91	**6**	10	**10**	15.2	5-4-4	271	**19**	10	**10**	14.3
4-4-2	12	**2**	10	**10**	6.0	5-4-5	476	**91**	1	**4**	5.2
4-4-3	36	**6**	10	**10**	6.0	8-4-2	23	**2**	10	**10**	11.5
4-4-4	226	**16**	10	**10**	14.1	8-4-3	37	**5**	10	**10**	7.4
4-4-5	255	**25**	7	**10**	10.2	8-4-4	58	**10**	10	**10**	5.8
5-3-2	11	**1**	10	**10**	11.0	8-4-5	97	**15**	10	**10**	6.5
5-3-3	18	**3**	10	**10**	6.0	8-4-6	208	**33**	10	**10**	6.3
5-3-4	32	**4**	10	**10**	8.0	8-4-7	698	**111**	9	**10**	6.3

any two values for any two variables are swapped. This neighbourhood needs an initial assignment respecting the constraints (c_1) and (c_2).

In Table 1, **bold** numbers correspond to the set variable approach. For both approaches, each instance was run 10 times. For the finite domain based approach, a maximum of 1000 iterations was allowed in each run. When 1000 iterations were reached the instance was considered not solved. For the set based approach, this number was set to 200. The first column displays the instance solved, where g is the number of groups, s is the size of each group, and w is the number of weeks to find a schedule for. The second column displays the average number of iterations needed to find a solution among the runs where a solution was actually found. A non-solved instance was thus not contributing to this value. The third column displays the number of runs, out of 10, where a solution was found. Finally, the fourth column displays the improvement ratio, i.e., the average number of iterations in the finite domain based model divided by the average number of iterations in the set based model.

8 Discussion

As can be seen in Table 1, the number of iterations needed in the model based on set variables is less than the number of iterations needed in the model based on finite domain variables in all instances tried. The set based model also solves instances where the finite domain based model fails in all runs.

One reason for this is probably the different choices of neighbourhoods. The *SwapP* neighbourhood used in the set based model is more powerful than the simple *Flip* neighbourhood used in the finite domain based model. However, the neighbourhoods picked are natural for their respective models. Defining a neighbourhood similar to the *SwapP* one for the finite domain model would be tedious. Having finite set variables at ones disposal simplifies this process considerably.

Another measurement that could be used is processing time. However, the current implementation of the set based constraints uses naïve algorithms for updating the penalty costs when exploring neighbourhoods. No incrementality whatsoever is used which, of course, implies poor performance. This is also seen by looking at the solved instances of the social golfer problem; none of them are particularly hard. Now, the aim of this paper was not (yet) to break any new records, but to present a new way of solving combinatorial (optimisation) problems using local search. Efficiency will come later and there are many options available to achieve this. One could for instance use the approach taken in Comet [11] by introducing *invariants* in the system, or the approach taken in [2] by reasoning about the constraints from a *graph-based* point of view.

9 Conclusion

We introduced finite set variables into local search. We defined the concepts of configurations, transition functions, neighbourhoods, and penalty costs for finite set variables and constraints on these. We also introduced a number of set constraints and defined their penalty costs, and used these to model three combinatorial problems.

We believe that local search techniques are ready to be taken to a higher and more abstract level, and that one step in this direction is to introduce finite set variables. This will allow a user to utilise a modelling device that has been around for quite some time in the neighbouring constraint programming area. This will also mean that the designers and implementors of local search systems may reason about the components in their systems at a higher level. This should be beneficial for the necessary incremental algorithms and data structures, leading to more efficient systems.

Acknowledgements

I thank Pierre Flener and Justin Pearson for fruitful discussions and comments on earlier drafts of this paper.

References

1. Adam Ameur and Jakub Orzechowski Westholm. Local search methods in gene expression analysis. Technical Report T2002-12, Swedish Institute of Computer Science, 2002. Available from `http://www.sics.se/libindex.html`.
2. Markus Bohlin. Design and Implementation of a Graph-Based Constraint Model for Local Search, April 2004. PhL thesis, Department of Computer Science and Engineering, Mälardalen University, Västerås, Sweden.

3. P. Codognet and D. Diaz. Yet another local search method for constraint solving. In K Steinhöfel, editor, *Proceedings of SAGA 2001, First International Symposium on Stochastic Algorithms : Foundations and Applications*, volume 2264 of *LNCS*, pages 73–90. Springer-Verlag, 2001.
4. Philippe Galinier and Jin-Kao Hao. Solving the progressive party problem by local search. In S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors, *Metaheuristics: Advances and Trends in Local Search Paradigms for Optimization*, chapter 29, pages 418–432. Kluwer Academic Publishers, 1998.
5. Philippe Galinier and Jin-Kao Hao. A general approach for constraint solving by local search. In *Proceedings of CP-AI-OR'00*, 2000.
6. Carmen Gervet. *Set Intervals in Constraint Logic Programming: Definition and Implementation of a Language*. PhD thesis, Université de Franche-Comté, France, September 1995. European thesis, in English.
7. Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
8. Fred Glover and Manuel Laguna. Tabu search. In *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–150. John Wiley & Sons, 1993.
9. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, may 1983.
10. Laurent Michel and Pascal Van Hentenryck. Localizer: A modeling language for local search. In Gert Smolka, editor, *Proceedings of CP'97*, volume 1330 of *LNCS*. Springer-Verlag, 1997.
11. Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):101–110, 2002. Proceedings of OOPSLA'02.
12. Laurent Michel and Pascal Van Hentenryck. Maintaining longest paths incrementally. In Francesca Rossi, editor, *Proceedings of CP'03*, volume 2833 of *LNCS*, pages 540–554. Springer-Verlag, 2003.
13. Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 17–19 September 1997.
14. Alexander Nareyek. Using global constraints for local search. In *Proceedings of DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization*, pages 1–18, 1998.
15. Bertrand Neveu and Gilles Trombettoni. When local search goes with the winners. In Michel Gendreau, Gilles Pesant, and Louis-Martin Rousseau, editors, *Proceedings of CP-AI-OR'03*, pages 180–194, 2003.
16. Jean-François Puget. Finite set intervals. In *Proceedings of Workshop on Set Constraints, held at CP'96*, 1996.
17. Pascal Van Hentenryck and Laurent Michel. Control abstractions for local search. In Francesca Rossi, editor, *Proceedings of CP'03*, volume 2833 of *LNCS*, pages 65–80. Springer-Verlag, 2003.