Generic Incremental Algorithms for 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. When a new (global) constraint is introduced in local search, measures for the penalty and variable conflicts of that constraint must be defined, and incremental algorithms for maintaining these measures must be implemented. These are complicated and time-consuming tasks, which clearly reduces the productivity of the local-search practitioner. We introduce a generic scheme that, from a description of a constraint in monadic existential second-order logic extended with counting, automatically gives penalty and variable-conflict measures for such a constraint, as well as incremental algorithms for maintaining these measures. We prove that our variable-conflict measure for a variable x is lower-bounded by the maximum penalty decrease that may be achieved by only changing the value of x, as well as upper bounded by the penalty measure. Without these properties, the local search performance may degrade. We also demonstrate the usefulness of the approach by replacing a built-in global constraint by a modelled version, while still obtaining competitive results in terms of runtime and robustness. This is especially attractive when a particular (global) constraint is not built in.

1 Introduction

Local search is a powerful and well-established method for solving hard combinatorial problems [1]. Yet, until recently, it has provided very little user support, leading to time-consuming and error-prone implementation tasks. The recent emergence of languages and systems for local search, sometimes based on novel abstractions, has alleviated the user of much of this burden [11, 20, 13, 12].

However, if a problem cannot readily be modelled using the primitive (global) constraints of such a local search system, then the *user* has to perform some of those time-consuming and error-prone tasks. These include coming up with measures and implementing efficient incremental maintenance algorithms for the penalties and variable conflicts of (global) constraints. Coming up with good measures is crucial in order to drive the local search to promising regions of the search space. Implementing efficient maintenance algorithms is crucial since these are called very often in the innermost loop of the search: incrementality is of great importance. The need to perform these tasks, assuming that the user

[⋆] Part of this work was done while this author was a Visiting Faculty Member at Sabancı University in İstanbul, Turkey.

has the necessary skills, clearly reduces her productivity, since much time must usually be devoted to them.

In this article (which collects the results of [3, 2, 4]), we propose the usage of monadic existential second-order logic extended with counting (here denoted $\exists$MSO$^+$) as a *modelling language* for (user defined) constraints in local search. Towards this, we propose inductive definitions for measuring the penalty and variable conflicts of a formula in $\exists$MSO$^+$ with respect to a configuration (assignment of values to all variables), as well as incremental algorithms for maintaining these measures efficiently between different configurations. The proposed penalty and variable-conflict measures are based on the idea of combinators [19] and extended also to encompass quantifiers and set variables.

We show that our conflict measure of a variable x is lower-bounded by the intuitive target value, namely the maximum penalty decrease of the formula that may be achieved by only changing the value of x, as well as upper bounded by the penalty of the formula. Without these properties, the local search performance may degrade.

We demonstrate the usefulness of the approach by replacing a built-in global constraint of our local-search framework by a modelled $\exists$MSO$^+$ version, while still obtaining competitive results in terms of run time and robustness. Since no effort at all is then spent on defining and incrementally implementing penalty and variable-conflict functions for the modelled global constraint, as our generic algorithms take care of that, we bring constraint-based local search a step closer to the level of expressiveness of constraint programming.

In the following, we introduce necessary background information concerning constraint-based local search (for set-CSPs) and monadic existential second-order logic extended with counting in Section 2. We present our penalty measure for $\exists$MSO$^+$ formulas in Section 3. After that, we introduce our variable-conflict measure for $\exists$MSO$^+$ formulas, as well as prove upper and lower bounds for this measure, in Section 4. Next, we show how the proposed measures can be maintained incrementally between two different configurations and discuss complexity issues in Section 5. After that, Section 6 contains experimental results showing the usefulness of the approach by replacing a global constraint in a problem model by a modelled $\exists$MSO$^+$ version thereof, while still obtaining competitive results. Finally, we conclude the article in Section 7. Appendix A contains all longer proofs of the article.

2 Preliminaries

A *constraint satisfaction problem* (*CSP*) is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a finite set of variables, $\mathcal{D}$ is a finite domain containing the possible values for the variables in $\mathcal{X}$, and $\mathcal{C}$ is a finite set of constraints, each constraint being defined on a subset of $\mathcal{X}$ and specifying which assignments of values from $\mathcal{D}$ to those variables make the constraint hold. Note that a common domain $\mathcal{D}$ does not imply any loss of generality since membership in a smaller domain can be required by a constraint. By abuse of language, we often identify a constraint with the singleton set containing it, and a CSP with its constraint set.

In this article we focus on set-CSPs, that is CSPs where the domain $\mathcal{D}$ is the power set $\mathscr{P}(\mathcal{U})$ of some set $\mathcal{U}$, called the *universe*. Using such *set variables* is already a common practice in constraint programming (e.g., [14, 8, 5]), but not yet in (constraint-based) local search. Note that scalar variables can be mimicked by set variables constrained to be singletons. Even though we only consider set-CSPs, we make no claims about their superiourity, and many of our results transpose to other variants of CSPs, such as the traditional scalar CSPs.

2.1 Local Search

In local search (e.g., [1]), an initial, possibly arbitrary, assignment of values to *all* the variables is maintained:

Definition 1 (Configuration and Solution). *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP and let $c \in \mathcal{C}$ be defined on the set of variables $\{x_1, \ldots, x_n\} \subseteq \mathcal{X}$.*

- *A* configuration *for P (or $\mathcal{X}$) is a total function $k : \mathcal{X} \to \mathcal{D}$.*
- *A configuration k is a* solution to a constraint *$c \in \mathcal{C}$ (written $k \models c$) if and only if c holds when each x_i is replaced by $k(x_i)$.*
- *A configuration k is a* solution to *P if and only if $k \models c$, for all $c \in \mathcal{C}$.*

We will use $\mathcal{K}_P$ to denote the *set of all configurations* for a given CSP P.

Example 1. Consider a CSP $P = \langle \{S, T\}, \mathscr{P}\{a, b, c\}, \{S \subset T\} \rangle$. A configuration for P is given by $k(S) = \{a, b\}$ and $k(T) = \emptyset$, or equivalently by $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$. A solution to P is given by $s = \{S \mapsto \{a, b\}, T \mapsto \{a, b, c\}\}$. Indeed, $s \models S \subset T$ since $s(S) = \{a, b\}$ is a strict subset of $s(T) = \{a, b, c\}$.

Local search iteratively makes a small change to the current configuration, upon examining the merits of many such changes, until a solution is found or allocated resources have been exhausted. The configurations thus examined constitute the neighbourhood of the current configuration:

Definition 2 (Neighbourhood and Variable-Oriented Neighbourhood). *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP and let $k \in \mathcal{K}_P$.*

- *A* neighbourhood function *for P is a function $n : \mathcal{K}_P \to \mathscr{P}(\mathcal{K}_P)$.*
- *The* neighbourhood *of P with respect to k and n is the set $n(k)$.*
- *The* variable-oriented neighbourhood *for $x \in \mathcal{X}$ with respect to k is the subset of $\mathcal{K}_P$ reachable from k by keeping the bindings of all variables other than x unchanged: $n_x(k) = \{\ell \in \mathcal{K}_P \mid \forall y \in \mathcal{X} : y \neq x \to k(y) = \ell(y)\}$.*

Note that the size of $n_x(k)$ is determined by the size of the domain $\mathcal{D}$. For set-CSPs, this is exponential in the size of $\mathcal{U}$. However, we use variable-oriented neighbourhoods to define concepts only, and do not enumerate them in practice.

Example 2. The neighbourhood of $S \subset T$ with respect to $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$ and the neighbourhood function that transfers an element from S to T is the set $\{k_a = \{S \mapsto \{b\}, T \mapsto \{a\}\}, k_b = \{S \mapsto \{a\}, T \mapsto \{b\}\}$. The variable-oriented neighbourhood for S with respect to k is the set:

$$\begin{aligned} n_S(k) = \{&k, \\ &k_1 = \{S \mapsto \emptyset, T \mapsto \emptyset\}, \\ &k_2 = \{S \mapsto \{a\}, T \mapsto \emptyset\}, \\ &k_3 = \{S \mapsto \{b\}, T \mapsto \emptyset\}, \\ &k_4 = \{S \mapsto \{c\}, T \mapsto \emptyset\}, \\ &k_5 = \{S \mapsto \{a, c\}, T \mapsto \emptyset\}, \\ &k_6 = \{S \mapsto \{b, c\}, T \mapsto \emptyset\}, \\ &k_7 = \{S \mapsto \{a, b, c\}, T \mapsto \emptyset\}\} \end{aligned}$$

Local search uses the penalty of a constraint, which is an estimate on how much it is violated with respect to a given configuration, in order to rank the different configurations of a neighbourhood:

Definition 3 (Penalty). *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP, let $c \in \mathcal{C}$, and let $k \in \mathcal{K}_P$.*

- *A* penalty function *of c is a function $penalty(c) : \mathcal{K}_P \to \mathbb{N}$ such that $penalty(c)(k) = 0$ if and only if $k \models c$.*
- *The* penalty *of c with respect to k is $penalty(c)(k)$.*
- *The* penalty *of P with respect to k is the sum $\sum_{c \in \mathcal{C}} penalty(c)(k)$.*

In order for a constraint-based local-search approach to be effective, different constraints should have balanced penalties [6], i.e., for a set of constraints $\mathcal{C}$, no $c \in \mathcal{C}$ should be easier in general to make hold compared to any other $d \in \mathcal{C}$. This may be application dependent, in which case weights could be added to tune the penalties, see [12] for example. For set constraints, we believe that one such approach is to let (by extension of the integer-variable ideas in [7]) the penalty of a set constraint c with respect to a configuration k be the length of the shortest sequence of elementary set operations (defined below) that must be performed on the variables of c in order for c to hold.

Definition 4 (Elementary Set Operations). *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP, let $k \in \mathcal{K}_P$, and let $S \in \mathcal{X}$. An* elementary set operation *on $k(S)$ is one of the following changes to $k(S)$:*

1. *Add a value u to $k(S)$ from its complement $\mathcal{U} \setminus k(S)$, denoted $add(S, u)$.*
2. *Drop a value u from $k(S)$, denoted $drop(S, u)$.*

Example 3. Consider $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$. Performing the sequence $\Delta = [drop(S, b), add(T, a), add(T, b)]$ of elementary set operations on k gives $\Delta(k) = \{S \mapsto \{a\}, T \mapsto \{a, b\}\}$.

One can of course also consider the *transfer* of an element from one set to another set, the *swap* of single elements between two sets, and the *flip* of an element from one set into an element from its complement set as elementary set operations, even though they are expressible as sequences of *add* and *drop* operations. Evaluating the impact of such alternative choices is beyond the scope of this paper.

It is often crucial to limit the size of the neighbourhood for efficiency reasons. One way of doing this is to focus on *conflicting variables.* A variable x has a positive conflict with respect to a constraint c and a configuration k if we may decrease the penalty of c by only changing the value of x in k. This may also be more refined such that different variables can be ranked based on their conflicts in order to concentrate on those with maximum conflict.

Definition 5 (Variable Conflict). *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP, let $penalty(c)$ be a penalty function of $c \in \mathcal{C}$, let $x \in \mathcal{X}$, and let $k \in \mathcal{K}_P$.*

- *A* variable-conflict function *of c is a function $conflict(c) : \mathcal{X} \times \mathcal{K}_P \to \mathbb{N}$ such that if $conflict(c)(x, k) = 0$ then $\forall \ell \in n_x(k) : penalty(c)(k) \leq penalty(c)(\ell)$.*
- *The* variable conflict *of x with respect to c and k is $conflict(c)(x, k)$.*
- *The* variable conflict *of x with respect to P and k is the sum $\sum_{c \in \mathcal{C}} conflict(c)(x, k)$.*

When we define the variable-conflict function of a constraint, we want to measure the *maximum* amount of penalty decrease possible by only changing a given variable. In order to state theoretical properties of our variable-conflict function for $\exists\text{MSO}^+$ formulas in Section 4, we now formalise this targeted value and state some of its properties.

Definition 6 (Abstract Variable Conflict). *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP, let $penalty(c)$ be a penalty function of $c \in \mathcal{C}$, let $x \in \mathcal{X}$, and let $k \in \mathcal{K}_P$.*

- *The* abstract variable-conflict function *of c with respect to $penalty(c)$ is the function $abstractConflict(c) : \mathcal{X} \times \mathcal{K}_P \to \mathbb{N}$ such that:*

$$abstractConflict(c)(x, k) = \max\{penalty(c)(k) - penalty(c)(\ell) \mid \ell \in n_x(k)\}$$

- *The* abstract variable conflict *of x with respect to c, $penalty(c)$, and k is $abstractConflict(c)(x, k)$.*

Note that, given a penalty function of a constraint c, there is only one abstract-variable-conflict function of c. However, variable-conflict functions as in Definition 5 are not unique.

Example 4. Let the penalty and variable-conflict functions of $S \subset T$ be defined by

$$penalty(S \subset T)(k) = |k(S) \setminus k(T)| + \begin{cases} 1, & \text{if } k(T) \subseteq k(S) \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

and

$$\mathit{conflict}(S \subset T)(Q,k) =$$
$$|k(S) \setminus k(T)| + \begin{cases} 1, \text{ if } Q = T \text{ and } k(T) \subseteq k(S) \\ 1, \text{ if } Q = S \text{ and } \emptyset \neq k(T) \subseteq k(S) \\ 0, \text{ otherwise} \end{cases} \tag{2}$$

respectively. Now, the penalties of $S \subset T$ with respect to $k = \{S \mapsto \{a,b\}, T \mapsto \emptyset\}$ and $k_a = \{S \mapsto \{b\}, T \mapsto \{a\}\}$ are respectively $\mathit{penalty}(S \subset T)(k) = 3$ and $\mathit{penalty}(S \subset T)(k_a) = 1$. Indeed, we may satisfy $S \subset T$ with respect to k by, e.g., performing the sequence $\Delta = [\mathit{drop}(S,b),\ \mathit{add}(T,a), \mathit{add}(T,b)]$ of elementary set operations, and with respect to k_a by, e.g., performing the sequence $[\mathit{drop}(S,b)]$ of elementary set operations.

The variable conflicts of S and T with respect to $S \subset T$ and k are $\mathit{conflict}(S \subset T)(S,k) = 2$ and $\mathit{conflict}(S \subset T)(T,k) = 3$, respectively. Indeed, by changing the value of S, we may decrease the penalty of $S \subset T$ by two (by performing the sequence $[\mathit{drop}(S,a), \mathit{drop}(S,b)]$ of elementary set operations). Similarly, by changing the value of T, we may decrease the penalty of $S \subset T$ by three (by performing the sequence $[\mathit{add}(T,a), \mathit{add}(T,b), \mathit{add}(T,c)]$ of elementary set operations). This suggests (if we rank S and T based on their variable conflicts) that we should try to satisfy $S \subset T$ by changing the value of T.

The variable-conflict function (2) gives the abstract conflicts of variables with respect to the penalty function (1). Consider, for example, $k = \{S \mapsto \{a,b\}, T \mapsto \emptyset\}$ and recall the variable-oriented neighbourhood $n_S(k) = \{k, k_1, \ldots, k_7\}$ for S of Example 2. A configuration $\ell \in n_S(k)$ that maximises

$$\mathit{penalty}(S \subset T)(k) - \mathit{penalty}(S \subset T)(\ell) \tag{3}$$

is $\ell = k_1 = \{S \mapsto \emptyset, T \mapsto \emptyset\}$. Indeed, the maximum value of (3) is $2 = 3 - 1 = \mathit{penalty}(S \subset T)(\{S \mapsto \{a,b\}, T \mapsto \emptyset\}) - \mathit{penalty}(S \subset T)(\{S \mapsto \emptyset, T \mapsto \emptyset\})$, the same value as given by $\mathit{conflict}(S \subset T)(S,k)$.

If a variable x is not in the set of variables of a constraint c, then the abstract variable conflict of x with respect to c is always zero:

Proposition 1. *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP, let $c \in \mathcal{C}$, and let $x \in \mathcal{X}$. If x is not in the set of variables that c is defined on, then $\mathit{abstractConflict}(c)(x,k) = 0$, for all $k \in \mathcal{K}_P$.*

Proof. For any $k \in \mathcal{K}_P$ and for each $y \neq x$ it holds that $\forall \ell \in n_x(k) : \ell(y) = k(y)$, which implies that $\forall \ell \in n_x(k) : \mathit{penalty}(c)(k) = \mathit{penalty}(c)(\ell)$, and hence that $\max\{\mathit{penalty}(c)(k) - \mathit{penalty}(c)(\ell) \mid \ell \in n_x(k)\} = 0$. □

The abstract-variable-conflict function is a variable-conflict function:

Proposition 2. *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP and let $\mathit{penalty}(c)$ be a penalty function of $c \in \mathcal{C}$. Then the function $\mathit{abstractConflict}(c)$ of Definition 6 is a variable-conflict function according to Definition 5.*

Proof. Given a variable $x \in \mathcal{X}$ and a configuration $k \in \mathcal{K}_P$, we must show that:

1. $\mathit{abstractConflict}(c)(x,k) \geq 0$.
2. $\mathit{abstractConflict}(c)(x,k) = 0 \Rightarrow \forall \ell \in n_x(k) : \mathit{penalty}(c)(k) \leq \mathit{penalty}(c)(\ell)$.

Consider first 1. We know that there exists at least one $\ell \in n_x(k)$ (namely $\ell = k$) such that $\mathit{penalty}(c)(k) - \mathit{penalty}(c)(\ell) = 0$. Since the abstract variable conflict is defined as the maximum difference between $\mathit{penalty}(c)(k)$ and $\mathit{penalty}(c)(\ell)$, for any $\ell \in n_x(k)$, this maximum difference is at least 0.

Consider now 2. Assume that $\mathit{abstractConflict}(c)(x,k) = 0$. Then we have that

$$\max\{\mathit{penalty}(c)(k) - \mathit{penalty}(c)(\ell) \mid \ell \in n_x(k)\} = 0$$

and hence that $\forall \ell \in n_x(k) : \mathit{penalty}(c)(k) \leq \mathit{penalty}(c)(\ell)$. □

In Example 4 above, we presented penalty and variable-conflict functions for the constraint $S \subset T$. Even though this constraint is simple, it still takes some time and effort to define and implement incremental versions of those functions. Imagine then the time and effort needed to define and implement incremental penalty and variable-conflict functions for a more complicated (possibly global) constraint. This is exactly what we aim at by allowing the user to model constraints in a simple language and obtaining such incremental penalty and variable-conflict functions for free.

2.2 Monadic Existential Second-Order Logic

We use monadic existential second-order logic extended with counting for modelling user-defined set constraints [2] or entire problems, which are just conjunctions of constraints. This language, referred to by $\exists\text{MSO}^+$ and shown in BNF in Figure 1, is very expressive as it captures at least the complexity class NP [10]. A constraint in $\exists\text{MSO}^+$ is of the form $\exists S_1 \cdots \exists S_n \phi$, i.e., a sequence of existentially quantified set variables, ranging over the power set of an implicit common finite universe $\mathcal{U}$, and constrained by a logical formula ϕ. Note that we only consider finite models and that the restriction to a monadic logic implies no nested set values. Also note that our intended usage of $\exists\text{MSO}^+$ here is to model set constraints that can be expressed by reasoning with the elements of the universe (constraints such as $S \subset T$, $S_1 \cap S_2 = T$, and $S \neq T$, for example). Although it is possible to model, e.g., arithmetic constraints in $\exists\text{MSO}^+$ this would be very tedious. The use of an extended language that simplifies this is considered future work.

In Sections 3 and 4 we will define penalty and variable-conflict functions of formulas in $\exists\text{MSO}^+$. Before we do this, we define a core subset of this language that will be used in those definitions. This is only due to the way we define the penalty and variable-conflict functions and does not pose any limitations on the expressiveness of the language: Any formula in $\exists\text{MSO}^+$ may be transformed into an equivalent formula in that core subset, in a way shown next.

The transformations are standard and are only described briefly. Given is a formula $\exists S_1 \cdots \exists S_n \phi$ in $\exists\text{MSO}^+$. First, we can obtain an equivalent formula

$$
\begin{array}{lll}
\langle \mathit{Constraint} \rangle & ::= & (\underline{\exists}\ \langle S \rangle)^{+}\ \langle \mathit{Formula} \rangle \\
 & & \\
\langle \mathit{Formula} \rangle & ::= & \underline{(}\langle \mathit{Formula} \rangle\underline{)} \\
 & \mid & (\underline{\forall} \mid \underline{\exists})\langle x \rangle\ \langle \mathit{Formula} \rangle \\
 & \mid & \langle \mathit{Formula} \rangle\ (\underline{\land} \mid \underline{\lor} \mid \underline{\rightarrow} \mid \underline{\leftrightarrow} \mid \underline{\leftarrow})\ \langle \mathit{Formula} \rangle \\
 & \mid & \underline{\neg}\langle \mathit{Formula} \rangle \\
 & \mid & \langle \mathit{Literal} \rangle \\
 & & \\
\langle \mathit{Literal} \rangle & ::= & \langle x \rangle\ (\underline{\in} \mid \underline{\notin})\ \langle S \rangle \\
 & \mid & \langle x \rangle\ (\underline{<} \mid \underline{\leq} \mid \underline{=} \mid \underline{\neq} \mid \underline{\geq} \mid \underline{>})\ \langle y \rangle \\
 & \mid & \underline{|}\langle S \rangle\underline{|}\ (\underline{<} \mid \underline{\leq} \mid \underline{=} \mid \underline{\neq} \mid \underline{\geq} \mid \underline{>})\ \langle a \rangle
\end{array}
$$

Fig. 1. The BNF grammar for the language $\exists\mathrm{MSO}^+$ where terminal symbols are underlined. The non terminal symbol $\langle S \rangle$ denotes an identifier for a bound set variable S such that $S \subseteq \mathcal{U}$, where $\mathcal{U}$ is the common universe, while $\langle x \rangle$ and $\langle y \rangle$ denote identifiers for bound first-order variables x and y such that $x, y \in \mathcal{U}$, and $\langle a \rangle$ denotes a natural number constant. The core subset of $\exists\mathrm{MSO}^+$ corresponds to the language given by the non-highlighted production rules.

$\exists S_1 \cdots \exists S_n \psi$, where ψ does not contain equivalences or implications by replacing equivalences by conjunctions of implications and implications by disjunctions. Assuming that ϕ is the formula $\psi_1 \leftrightarrow \psi_2$, it is replaced by $(\neg\psi_1 \lor \psi_2) \land (\neg\psi_2 \lor \psi_1)$. Second, we can obtain an equivalent formula $\exists S_1 \cdots \exists S_n \psi$, where ψ does not contain the negation symbol $\neg$, by pushing those symbols downward, all the way to the literals, which are replaced by their negated counterparts. Assuming that ϕ is the formula $\forall x(\neg(x \in S \land x \notin T))$, it is transformed into $\forall x(x \notin S \lor x \in T)$. This is possible because the set of relational operators in $\exists\mathrm{MSO}^+$ is closed under negation.

By performing these transformations for ϕ (and recursively for the subformulas of ϕ) in any formula $\exists S_1 \cdots \exists S_n \phi$, we end up with a formula in the non-highlighted subset of the language in Figure 1, for which we will define penalty and variable-conflict functions. In the rest of this article, we only give formulas in this core subset.

3 Penalty of an $\exists\mathrm{MSO}^+$ Formula

In order to use a formula in $\exists\mathrm{MSO}^+$ as a constraint or an entire model in our local search framework, we must define a penalty function of such a formula according to Definition 3, which is done inductively below. It is important to stress that *its calculation will be totally generic and automatable, as it will be based only on the syntax of the formula and the semantics of the quantifiers, connectives, and relational operators of the $\exists\mathrm{MSO}^+$ language, but not on the intended semantics of the formula. A human might well give a different penalty function to that formula, and a way of calculating it that better exploits globality, but the scheme below requires* no *such user participation.*

We need to express the penalty with respect to the values of any bound first-order variables. We will therefore recursively extend the configuration at hand also with this information for the relevant cases.

Definition 7 (Penalty of a Formula). *Let Φ be a formula in* $\exists$MSO$^+$ *and let k be a configuration for the set variables in Φ. The* penalty *of Φ with respect to k is defined by:*

$$\begin{array}{lcl}
(a)\ \mathit{penalty}(\exists S_1 \cdots \exists S_n \phi)(k) & = & \mathit{penalty}(\phi)(k) \\
(b)\ \mathit{penalty}(\forall x \phi)(k) & = & \sum\limits_{u \in \mathcal{U}} \mathit{penalty}(\phi)(k \cup \{x \mapsto u\}) \\
(c)\ \mathit{penalty}(\exists x \phi)(k) & = & \min\{\mathit{penalty}(\phi)(k \cup \{x \mapsto u\} \mid u \in \mathcal{U}\}) \\
(d)\ \mathit{penalty}(\phi \wedge \psi)(k) & = & \mathit{penalty}(\phi)(k) + \mathit{penalty}(\psi)(k) \\
(e)\ \mathit{penalty}(\phi \vee \psi)(k) & = & \min\{\mathit{penalty}(\phi)(k), \mathit{penalty}(\psi)(k)\} \\
(f)\ \mathit{penalty}(|S| \leq a)(k) & = & \begin{cases} 0, \text{ if } |k(S)| \leq a \\ |k(S)| - a, \text{ otherwise} \end{cases} \\
(g)\ \mathit{penalty}(x \in S)(k) & = & \begin{cases} 0, \text{ if } k(x) \in k(S) \\ 1, \text{ otherwise} \end{cases} \\
(h)\ \mathit{penalty}(x \leq y)(k) & = & \begin{cases} 0, \text{ if } k(x) \leq k(y) \\ k(x) - k(y), \text{ otherwise} \end{cases}
\end{array}$$

In the definition above, for subformulas of the form $x \lozenge y$, $|S| \lozenge c$, and $x \bigtriangleup S$, only the cases where $\lozenge \in \{\leq\}$ and $\bigtriangleup \in \{\in\}$ are shown. The other cases are defined similarly. Note that cases (d) and (e) were originally proposed by [19]. We extend these ideas here to the logical quantifiers ($\forall$ and $\exists$). This is not just a matter of simply generalising the arities and penalty calculations of the $\wedge$ and $\vee$ connectives, respectively, but made necessary by our handling of set variables over which one would like to iterate, unlike the scalar variables of [19].

Recall from Section 2 that, given a constraint c and a configuration k for the variables of c, our aim when defining *penalty*(c) is that *penalty*$(c)(k)$ is the length of the shortest sequence of elementary set operations that must be performed on the variables in c in order for c to hold. This is not true in general for the penalty function induced by Definition 7. For example, given a configuration k for the variables of a formula Φ, the penalty of the formula $\Phi \wedge \Phi$ with respect to k is always twice the penalty of Φ with respect to k. But a sequence of elementary set operations that is performed to make Φ hold will make $\Phi \wedge \Phi$ hold as well. However, it is true for the base cases (f) and (g) containing set variables. Indeed, consider the subformulas $\phi = |S| \leq a$ and $\psi = x \in S$ and a configuration k for $\{S\}$ such that $k \not\models \phi$ and $k \not\models \psi$. A shortest sequence of elementary set operations that makes ϕ hold is a sequence of the form $[\mathit{drop}(S, u_1), \ldots, \mathit{drop}(S, u_n)]$, where $n = |k(S)| - a$ is the number of elements in $k(S)$ to drop such that the cardinality of S is not greater than a. The shortest sequence of elementary set operations that makes ψ hold is the sequence $[\mathit{add}(S, k(x))]$, which is indeed of length one.

We now show that the definition above of the penalty of an $\exists$MSO$^+$ formula induces a penalty function. In order not to interrupt the reading flow of the article, all longer proofs (including the proof of this proposition) can be found in Appendix A.

Proposition 3. *The function induced by Definition 7 is a penalty function according to Definition 3.*

Proof. See Appendix A.1. □

In other words, the penalty of a (formula describing a) constraint meets the basic requirement of a penalty function: it is zero when the constraint is satisfied, and positive otherwise.

In our experience, the calculated penalties of violated constraints are often meaningful, as seen in the following example.

Example 5. Let $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$, let $\mathcal{U} = \{a, b, c\}$, and let

$$\Phi = \exists S \exists T((\forall x(x \notin S \vee x \in T)) \wedge (\exists x(x \in T \wedge x \notin S))) \quad (4)$$

According to Definition 7, $penalty(\Phi)(k) = 3$, which is meaningful since we may satisfy Φ by, e.g., adding the three values a, b, and c to $k(T)$, and there is no shorter sequence of elementary set operations achieving this. The value 3 is obtained in the following way. The initial call $penalty(\exists S \exists T \phi)(k)$ matches case (a), which gives the recursive call $penalty(\phi)(k)$. Since ϕ is of the form $\psi_1 \wedge \psi_2$, this call matches case (d), which gives a summation of the recursive calls $penalty(\psi_1)(k)$ and $penalty(\psi_2)(k)$. The call $penalty(\psi_1)(k)$ matches case (b) since $\psi_1 = \forall x(x \notin S \vee x \in T)$. This implies a summation of three recursive calls where x is replaced by each value in the universe $\{a, b, c\}$. For a, the recursive call is $penalty(a \notin S \vee a \in T)(k)$. This call matches case (e), which is the minimum of the recursive calls $penalty(a \notin S)(k)$ and $penalty(a \in T)(k)$. These calls match case (g). The first one is 1, since a is in $k(S)$. The second one is also 1, since a is not in $k(T)$. Hence the minimum is 1. Similarly, for b and c in the universe, the values are respectively 1 and 0. This implies that $penalty(\psi_1)(k) = 2$. The call $penalty(\psi_2)(k)$ matches case (c) since $\psi_2 = \exists x(x \notin S \wedge x \in T)$. This implies the minimum of three recursive calls where x is again replaced by each value in the universe. For the value a, the recursive call is $penalty(a \notin S \wedge a \in T)(k)$. This call matches case (d) which is the summation of the recursive calls $penalty(a \notin S)(k)$ and $penalty(a \in T)(k)$. As above, these calls both give 1 so the sum is 2. Similarly, for b and c in the universe, the values are respectively 2 and 1. This implies that $penalty(\psi_2)(k) = 1$, and we have that $penalty(\exists S \exists T \phi)(k) = 2 + 1 = 3$. Note that this is the same value as obtained by the penalty function 1 of $S \subset T$ in Example 4.

4 Variable Conflict of an $\exists$MSO$^+$ Formula

In order to use formulas in $\exists$MSO$^+$ as constraints or entire models in our local search framework, we must define the variable-conflict function of such a formula according to Definition 5, which is done inductively below. Similarly to the penalty function of Definition 7, it is important to stress that its calculation will be totally generic and automatable.

The aim is to design variable-conflict functions that are exact with respect to the corresponding abstract variable-conflict functions. However, this is not possible in general for formulas in $\exists\text{MSO}^+$. Instead, as shown below, the following definition of the conflict of a variable overapproximates the abstract variable conflict (see Definition 6):

Definition 8 (Variable Conflict of a Formula). *Let Φ be a formula in $\exists\text{MSO}^+$, let k be a configuration for the set variables in Φ, and let S be one of those variables. The* variable conflict *of S with respect to Φ and k is defined by:*

(a) $\mathit{conflict}(\exists S_1 \cdots \exists S_n \phi)(S,k) = \mathit{conflict}(\phi)(S,k)$

(b) $\mathit{conflict}(\forall x \phi)(S,k) = \sum_{u \in \mathcal{U}} \mathit{conflict}(\phi)(S, k \cup \{x \mapsto u\})$

(c) $\mathit{conflict}(\exists x \phi)(S,k) =$
$$\mathit{penalty}(\exists x \phi)(k) - \min\{\mathit{penalty}(\phi)(k \cup \{x \mapsto u\}) - \mathit{conflict}(\phi)(S, k \cup \{x \mapsto u\}) \mid u \in \mathcal{U}\}$$

(d) $\mathit{conflict}(\phi \wedge \psi)(S,k) = \mathit{conflict}(\phi)(S,k) + \mathit{conflict}(\psi)(S,k)$

(e) $\mathit{conflict}(\phi \vee \psi)(S,k) =$
$$\mathit{penalty}(\phi \vee \psi)(k) - \min\{\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k), \mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k)\}$$

(f) $\mathit{conflict}(|T| \leq a)(S,k) = \begin{cases} 0, \text{ if } S \neq T \\ \mathit{penalty}(|S| \leq a)(k), \text{ otherwise} \end{cases}$

(g) $\mathit{conflict}(x \in T)(S,k) = \begin{cases} 0, \text{ if } S \neq T \\ \mathit{penalty}(x \in S)(k), \text{ otherwise} \end{cases}$

As in Definition 7, we only show the cases for subformulas of the form $|S| \Diamond c$ and $x \bigtriangleup S$ where $\Diamond \in \{\leq\}$ and $\bigtriangleup \in \{\in\}$.

This definition is specific to set variables and set constraints (modelled in $\exists\text{MSO}^+$), but its principle also applies to scalar variables and constraints. Note that the conflict of a set variable S with respect to a formula Φ is never negative (as shown in Proposition 6 below).

Example 6. Let $k = \{S \mapsto \{a,b\}, T \mapsto \emptyset\}$ and let Φ be (4). According to Definition 8, $\mathit{conflict}(\Phi)(S,k) = 2$ and $\mathit{conflict}(\Phi)(T,k) = 3$, which is meaningful since we may satisfy Φ by, e.g., adding the *three* values a, b, and c to $k(T)$, or by removing the *two* values a and b from $k(S)$ and adding any value to $k(T)$, and there are no shorter sequences of elementary set operations achieving this. Note that these are the same values as obtained by $\mathit{conflict}(S \subset T)$ of Example 4.

The key to understanding Definition 8 lies in the rules (c) and (e) for disjunctive formulas. The following example clarifies these in terms of rule (e).

Example 7. Consider the formula $\Phi = (|S| = 5 \vee (|T| = 3 \wedge |S| = 6))$ and let k_1 be a configuration for $\{S, T\}$ such that $|k_1(S)| = 6$ and $|k_1(T)| = 4$.

Then $\mathit{penalty}(\Phi)(k_1) = 1$ and, according to Definition 8, we have $\mathit{conflict}(|S| = 5)(S, k_1) = 1$ and $\mathit{conflict}(|T| = 3 \wedge |S| = 6)(S, k_1) = 0$. Rule (*e*) then applies for calculating $\mathit{conflict}(\Phi)(S, k_1)$, which is the penalty of Φ subtracted by the minimum penalty one may obtain for each disjunct by changing $k_1(S)$. This is 0 (decrease by 1) for the first disjunct since we may decrease $\mathit{penalty}(|S| = 5)(k_1)$ by one by changing $k_1(S)$ as witnessed by $\mathit{penalty}(|S| = 5)(k_1) - \mathit{conflict}(|S| = 5)(S, k_1) = 1 - 1 = 0$. It is 1 (unchanged) for the second disjunct since we cannot decrease $\mathit{penalty}(|T| = 3 \wedge |S| = 6)(k_1)$ by changing $k_1(S)$ as witnessed by $\mathit{penalty}(|T| = 3 \wedge |S| = 6)(k_1) - \mathit{conflict}(|T| = 3 \wedge |S| = 6)(S, k_1) = 1 - 0 = 1$. The minimum value of these is 0 and hence $\mathit{conflict}(\Phi)(S, k_1) = 1 - \min\{0, 1\} = 1 - 0 = 1$.

Consider now the configuration k_2 for $\{S, T\}$ such that $|k_2(S)| = 4$ and $|k_2(T)| = 4$. Then $\mathit{penalty}(\Phi)(k_2) = 1$ and we have $\mathit{conflict}(|S| = 5)(T, k_2) = 0$ and $\mathit{conflict}(|T| = 3 \wedge |S| = 6)(T, k_2) = 1$. The minimum penalty one may obtain by changing $k_2(T)$ in the first disjunct is 1 (unchanged) as witnessed by $\mathit{penalty}(|S| = 5)(k_2) - \mathit{conflict}(|S| = 5)(T, k_2) = 1 - 0 = 1$. For the second disjunct, this value is 2 (decrease by 1) as witnessed by $\mathit{penalty}(|T| = 3 \wedge |S| = 6)(k_2) - \mathit{conflict}(|T| = 3 \wedge |S| = 6)(T, k_2) = 3 - 1 = 2$. The minimum value of these is 1 and hence $\mathit{conflict}(\Phi)(S, k_2) = 1 - \min\{2, 1\} = 1 - 1 = 0$. Indeed, we cannot decrease $\mathit{penalty}(\Phi)(k)$ by changing $k_2(T)$ since even if we satisfy $|k_2(T)| = 3$, the other conjunct $|k_2(S)| = 6$ still implies a penalty larger than 1 and 1 is the minimum penalty of the two disjuncts of Φ.

If a variable-conflict function underestimates the abstract conflict of a variable, we may lose the reachability of some solution from any configuration. The following proposition states that the conflict of a variable x with respect to a formula Φ in $\exists\mathrm{MSO}^+$ according to Definition 8 is lower-bounded by the abstract variable-conflict of x with respect to Φ.

Proposition 4. *Let Φ be a formula in $\exists\mathrm{MSO}^+$, let k be a configuration for the set variables in Φ, and let S be one of those variables. Then $\mathit{conflict}(\Phi)(S, k) \geq \mathit{abstractConflict}(\Phi)(S, k)$.*

Proof. See Appendix A.2. □

The maximum possible penalty decrease of a constraint equals its penalty, hence the abstract conflict of any variable x with respect to a formula Φ in $\exists\mathrm{MSO}^+$ is upper bounded by the penalty of Φ. The next proposition states that the conflict of x with respect to Φ according to Definition 8 is also upper bounded by the penalty of Φ.

Proposition 5. *Let Φ be a formula in $\exists\mathrm{MSO}^+$, let k be a configuration for the set variables in Φ, and let S be one of those variables. Then $\mathit{conflict}(\Phi)(S, k) \leq \mathit{penalty}(\Phi)(k)$.*

Proof. See Appendix A.3. □

The definition of the variable conflict of an $\exists\mathrm{MSO}^+$ formula induces a variable-conflict function:

Proposition 6. *The function induced by Definition 8 is a variable-conflict function according to Definition 5.*

Proof. Let Φ be a formula in $\exists$MSO$^+$, let k be a configuration for the set variables in Φ, and let S be one of those variables. We must show that:

1. $\mathit{conflict}(\Phi)(S,k) \geq 0$.
2. $\mathit{conflict}(\Phi)(S,k) = 0 \Rightarrow \forall \ell \in n_S(k) : \mathit{penalty}(\Phi)(k) \leq \mathit{penalty}(\Phi)(\ell)$.

Consider first 1. Since $\mathit{abstractConflict}(\Phi)(S,k) \geq 0$ by Proposition 2, the result follows directly by Proposition 4.

Consider now 2. Assume that $\mathit{conflict}(\Phi)(S,k) = 0$. Then $\mathit{abstractConflict}(\Phi)(S,k) = 0$ by Proposition 4 and hence $\forall \ell \in n_S(k) : \mathit{penalty}(\Phi)(k) \leq \mathit{penalty}(\Phi)(\ell)$ by Proposition 2. □

5 Incremental Algorithms for Penalty and Variable-Conflict Functions of $\exists$MSO$^+$ Formulas

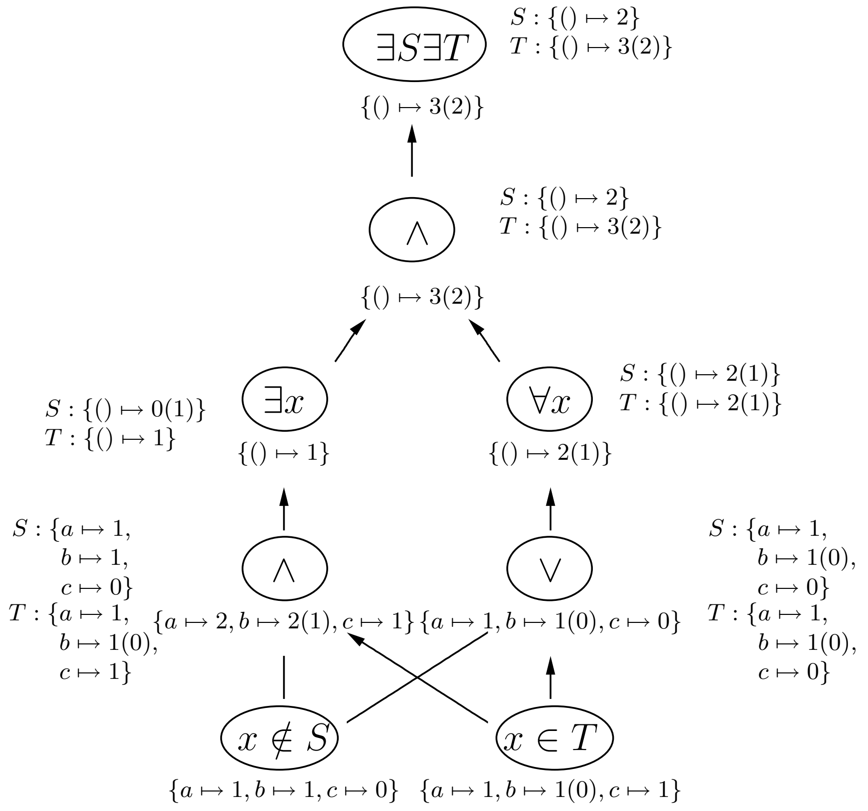
In our local search framework, given a formula Φ in $\exists$MSO$^+$, we could use Definitions 7 and 8 to calculate the penalty and variable conflicts of Φ with respect to a configuration k, and then similarly for each configuration ℓ in a neighbourhood $n(k)$ to be evaluated. However, such complete recalculations are impractical, since $n(k)$ is usually a very large set.

In local search it is therefore crucial to use *incremental algorithms* when evaluating the penalty and variable conflicts of a constraint with respect to a neighbour of a current configuration. We will now present a scheme for incremental maintenance of the penalty and variable conflicts of a formula in $\exists$MSO$^+$ with respect to Definitions 7 and 8. This scheme is based on viewing a formula in $\exists$MSO$^+$ as a syntax *DAG (directed acyclic graph)* and observing that, given the penalty and variable conflicts with respect to k, only the paths from the leaves that contain variables that are changed in ℓ (compared to k) to the root need to be updated to obtain the penalty and variable-conflicts with respect to ℓ.

In Section 5.1 we introduce the syntax DAG of an $\exists$MSO$^+$ formula and show how to extend it into a penalty and variable-conflict DAG. Then we show algorithms for initialising and incrementally updating such a DAG in Section 5.2. In Section 5.3 we provide an analysis of their worst-case space and time complexities.

5.1 The Penalty and Variable-Conflict DAG of a Formula

First, a *syntax DAG* D of a formula Φ in $\exists$MSO$^+$ of the form $\exists S_1 \cdots \exists S_n \phi$ is constructed in the usual way. Literals in Φ of the form $x \in S$, $x \notin S$, $x \lozenge y$, and $|S| \lozenge k$ (where $\lozenge \in \{<, \leq, =, \neq, \geq, >\}$) are leaves in D. Subformulas in Φ of the form $\psi_1 \square \psi_2$ (where $\square \in \{\wedge, \vee\}$) are sub-DAGs in D with $\square$ as parent node and the DAGs of ψ_1 and ψ_2 as children. When possible, formulas of the form $\psi_1 \square \cdots \square \psi_m$ give rise to one parent node with m children. Subformulas in Φ



$\exists S \exists T$
$S : \{() \mapsto 2\}$
$T : \{() \mapsto 3(2)\}$
$\{() \mapsto 3(2)\}$
$\wedge$
$S : \{() \mapsto 2\}$
$T : \{() \mapsto 3(2)\}$
$\{() \mapsto 3(2)\}$
$\exists x$
$S : \{() \mapsto 0(1)\}$
$T : \{() \mapsto 1\}$
$\{() \mapsto 1\}$
$\forall x$
$S : \{() \mapsto 2(1)\}$
$T : \{() \mapsto 2(1)\}$
$\{() \mapsto 2(1)\}$
$\wedge$
$S : \{a \mapsto 1, b \mapsto 1, c \mapsto 0\}$
$T : \{a \mapsto 1, b \mapsto 1(0), c \mapsto 1\}$
$\{a \mapsto 2, b \mapsto 2(1), c \mapsto 1\}$
$\vee$
$S : \{a \mapsto 1, b \mapsto 1(0), c \mapsto 0\}$
$T : \{a \mapsto 1, b \mapsto 1(0), c \mapsto 0\}$
$\{a \mapsto 1, b \mapsto 1(0), c \mapsto 0\}$
$x \notin S$
$\{a \mapsto 1, b \mapsto 1, c \mapsto 0\}$
$x \in T$
$\{a \mapsto 1, b \mapsto 1(0), c \mapsto 1\}$

Fig. 2. Penalty and variable-conflict DAG of (4).

of the form $\forall x \psi$ (resp. $\exists x \psi$) are sub-DAGs in D with $\forall x$ (resp. $\exists x$) as parent node and the DAG of ψ as only child. Finally, $\exists S_1 \cdots \exists S_n$ is the root of D with the DAG of ϕ as only child. We show the syntax DAG of (4) in Figure 2. Note that it contains additional information, to be explained below.

It is crucial to limit the size of the syntax DAG D of a formula Φ as much as possible. We do this by the sharing of sub-DAGs. For example, in D, if two subformulas ϕ and ψ of Φ both contain another subformula γ, there will be only one occurrence of the syntax DAG of γ in D. This is illustrated in Figure 2 by, e.g., the subformula $x \notin S$ only occurring once although it occurs twice in (4).

Assume that D is the syntax DAG of a formula Φ. We extend D into a *penalty and variable-conflict DAG* in the following way. Let k be a configuration for the set variables $\mathcal{X}$ in Φ. At each node d representing a subformula ψ of Φ, the penalty and variable conflicts for all $S \in \mathcal{X}$ with respect to k are stored. This implies that the penalty and variable-conflicts stored in the root of D are equal to *penalty*$(\Phi)(k)$ and *conflict*$(\Phi)(S, k)$, for all S in $\mathcal{X}$. This is illustrated in Figure 2, where the penalties and variable conflicts with respect to $k = \{S \mapsto$

$\{a, b\}, T \mapsto \emptyset\}$ are displayed below and beside the nodes respectively. The reader should disregard the numbers in parentheses and the directed edges for now, this will be explained below.

As shown in Figure 2, for the descendants of nodes representing subformulas that introduce bound first-order variables, we must store the penalty and variable conflicts with respect to *every* possible mapping of those variables. For example, the child node d of a node for a subformula of the form $\forall x \phi$ will have a penalty and variable conflicts stored for each $u \in \mathcal{U}$. Generally, the penalty and variable conflicts stored at a node d are mappings, denoted $p(d)$, from the possible tuples of values of the bound first-order variables at d to $\mathbb{N}$. Assume, for example, that at d there are two bound first-order variables x and y (introduced in that order) and that $\mathcal{U} = \{a, b\}$. Then the penalty and variable conflicts stored at d after initialisation are mappings of the form $\{(a, a) \mapsto q_1, (a, b) \mapsto q_2, (b, a) \mapsto q_3, (b, b) \mapsto q_4\}$ where $\{q_1, q_2, q_3, q_4\} \subset \mathbb{N}$. The first element of each tuple corresponds to x and the second one to y. If there are no bound first-order variables at a particular node, then the penalty and variable conflicts are mappings of the form $\{() \mapsto q\}$, i.e., the empty tuple mapped to some $q \in \mathbb{N}$.

5.2 Initialising and Updating the Penalty and Variable Conflicts

An incremental algorithm consists of two parts, the initialisation part and the updating part. The initialisation part is used when the value is calculated from scratch, without any prior information. The updating part is used when there is already some information available. We will now present the initialisation and updating parts of the incremental algorithm for maintaining a penalty and variable-conflict DAG. We only show algorithms for initialising and updating the penalty, and not the variable conflicts. The algorithms for initialising and updating variable conflicts are similar.

We start by showing the function *initialise*$(D, \mathcal{U}, k)$ in Algorithm 1. This recursive function initialises a penalty and variable-conflict DAG D of a formula with penalty mappings with respect to a universe $\mathcal{U}$ and a configuration k. By abuse of notation, we let formulas in $\exists\text{MSO}^+$ denote their corresponding penalty and variable-conflict DAGs. For example, $\forall x \phi$ denotes the DAG with $\forall x$ as root and the DAG representing ϕ as only child, $\phi_1 \wedge \cdots \wedge \phi_m$ denotes the DAG with $\wedge$ as root and the sub-DAGs of all the ϕ_i as children, etc. Note that we use an auxiliary function *tuple* that, given a configuration k, returns the tuple of first-order values with respect to k in the order they are introduced. We also assume that before *initialise* is called for a DAG D, all penalty mappings of D equal the empty set.

Example 8. Let $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$ and $\mathcal{U} = \{a, b, c\}$. The penalty and variable-conflict DAG D of (4) in Figure 2 is the result of calling *initialise*$(D, \mathcal{U}, k)$.

Now, given a penalty and variable-conflict DAG D of a formula Φ with respect to a configuration k for the set variables in Φ, it is important to realise the following: When a configuration ℓ in the neighbourhood of k is to be evaluated, the only paths in D that may have changed are those leading from leaves

Algorithm 1 Initialising a penalty and variable-conflict DAG.

```
function $initialise(D, \mathcal{U}, k)$
    if $D$ has not been initialised then
        match $D$ with
            $\exists S_1 \cdots \exists S_n \phi \longrightarrow p(D) \leftarrow \{() \mapsto initialise(\phi, \mathcal{U}, k)\}$
            $\mid \forall x \phi \longrightarrow p(D) \leftarrow p(D) \cup \{tuple(k) \mapsto \sum_{u \in \mathcal{U}} initialise(\phi, \mathcal{U}, k \cup \{x \mapsto u\})\}$
            $\mid \exists x \phi \longrightarrow$
                $p(D) \leftarrow p(D) \cup \{tuple(k) \mapsto \min\{initialise(\phi, \mathcal{U}, k \cup \{x \mapsto u\}) \mid u \in \mathcal{U}\}\}$
            $\mid \phi_1 \wedge \cdots \wedge \phi_m \longrightarrow p(D) \leftarrow p(D) \cup \{tuple(k) \mapsto \sum_{1 \leq i \leq m} initialise(\phi_i, \mathcal{U}, k)\}$
            $\mid \phi_1 \vee \cdots \vee \phi_m \longrightarrow$
                $p(D) \leftarrow p(D) \cup \{tuple(k) \mapsto \min\{initialise(\phi, \mathcal{U}, k) \mid \phi \in \{\phi_1, \ldots, \phi_m\}\}\}$
            $\mid x \leq y \longrightarrow p(D) \leftarrow p(D) \cup \left\{ tuple(k) \mapsto \begin{cases} 0, \text{ if } k(x) \leq k(y) \\ k(x) - k(y), \text{ otherwise} \end{cases} \right\}$
            $\mid |S| \leq m \longrightarrow p(D) \leftarrow p(D) \cup \left\{ tuple(k) \mapsto \begin{cases} 0, \text{ if } |k(S)| \leq m \\ |k(S)| - m, \text{ otherwise} \end{cases} \right\}$
            $\mid x \in S \longrightarrow p(D) \leftarrow p(D) \cup \left\{ tuple(k) \mapsto \begin{cases} 0, \text{ if } k(x) \in k(S) \\ 1, \text{ otherwise} \end{cases} \right\}$
        end match
    else
        ()
    return $p(D)(tuple(k))$
function $tuple(k)$
        $\triangleright$ $\{x_1, \ldots, x_n\}$ is the first-order variables in $domain(k)$, introduced in that order.
    return $(k(x_1), \ldots, k(x_n))$
```

containing any of the set variables that are affected by the change of k to ℓ, to the root. By updating the penalties with respect to the change on those paths, we incrementally calculate $penalty(\Phi)(\ell)$ given $penalty(\Phi)(k)$ and the difference between k and ℓ.

In Algorithm 2, we show the function $submit(d_t, d_f, \mathcal{T})$ that updates the penalty mappings of a penalty and variable-conflict DAG incrementally with respect to a current configuration k and a configuration ℓ in the neighbourhood of k. It is a recursive function where information from the node d_f (*void* when d_t is a leaf) is propagated to the node d_t. The additional argument $\mathcal{T}$ is a set of tuples of values that are affected by changing k to ℓ. It uses the auxiliary function $update(d_t, d_f, \mathcal{T})$ that performs the actual update of the penalty mappings of d_t with respect to (the change of the penalty mappings of) d_f.

Some of the notation used in Algorithm 2 needs explanation. The set of affected tuples $\mathcal{T}$ depends on the maximum number of bound first-order variables in the penalty and variable-conflict DAG, the universe $\mathcal{U}$, and the configurations

Algorithm 2 Updating a penalty and variable-conflict DAG.

```
function $submit(d_t, d_f, \mathcal{T})$
    $update(d_t, d_f, \mathcal{T})$                    ▷ First update $d_t$ with respect to $d_f$.
    if All children affected by the change of $k$ to $\ell$ are done then
        if $d_t$ is not the root and $p(d_t)$ changed then
            for all $d \in parents(d_t)$ do
                $submit(d, d_t, \mathcal{T} \cup changed(d_t))$
            $changed(d_t) \leftarrow \emptyset$
        else ()        ▷ We are at the root or $p(d_t)$ did not change. Done!
    else $changed(d_t) \leftarrow changed(d_t) \cup \mathcal{T}$        ▷ Not all children done. Save tuples.
function $update(d_t, d_f, \mathcal{T})$
    $p'(d_t) \leftarrow p(d_t)$          ▷ Save the old penalty mapping.
    for all $t \in \mathcal{T}_{|bounds(d_t)}$ do
        match $d_t$ with
            $\exists S_1 \cdots \exists S_n \phi \longrightarrow p(d_t) \leftarrow p(d_t) \oplus \{() \mapsto p(d_f)(())\}$
          | $\forall x \phi \longrightarrow$
                for all $t' \in \mathcal{T}_{|bounds(d_f)}$ such that $t'_{|bounds(d_t)} = t$ do
                    $p(d_t) \leftarrow p(d_t) \oplus \{t \mapsto p(d_t)(t) + p(d_f)(t') - p'(d_f)(t')\}$
          | $\exists x \phi \longrightarrow$
                for all $t' \in \mathcal{T}_{|bounds(d_f)}$ such that $t'_{|bounds(d_t)} = t$ do
                    Replace the value for $t'$ in $RBTree(d_t, t)$ with $p(d_f)(t')$
                    $p(d_t) \leftarrow p(d_t) \oplus \{t \mapsto \min(RBTree(d_t, t))\}$
          | $\phi_1 \wedge \cdots \wedge \phi_m \longrightarrow p(d_t) \leftarrow p(d_t) \oplus \{t \mapsto p(d_t)(t) + p(d_f)(t) - p'(d_f)(t)\}$
          | $\phi_1 \vee \cdots \vee \phi_m \longrightarrow$ Replace the value for $d_f$ in $RBTree(d_t, t)$ with $p(d_f)(t)$
                                $p(d_t) \leftarrow p(d_t) \oplus \{t \mapsto \min(RBTree(d_t, t))\}$
          | $|S| \leq m \longrightarrow p(d_t) \leftarrow p(d_t) \oplus \left\{ t \mapsto \begin{cases} 0, \text{ if } |\ell(S)| \leq m \\ |\ell(S)| - m, \text{ otherwise} \end{cases} \right\}$
          | $x \in S \longrightarrow p(d_t) \leftarrow p(d_t) \oplus \left\{ t \mapsto \begin{cases} 0, \text{ if } t(x) \in \ell(S) \\ 1, \text{ otherwise} \end{cases} \right\}$
        end match
```

k and ℓ. Recall $\mathcal{U} = \{a, b, c\}$ and $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}\}$ of Example 1 and assume that $\ell = \{S \mapsto \{a, b\}, T \mapsto \{b\}\}$ (b was added to $k(T)$). In this case $\mathcal{T}$ would be the singleton set $\{(b)\}$, since this is the only tuple affected by the change of k to ℓ. However, if the maximum number of bound variables was two (instead of one as in Example 1), then $\mathcal{T}$ would be the set $\{(b, a), (b, b), (b, c), (a, b), (c, b)\}$ since all of these tuples might be affected by the change.

Given a set $\mathcal{T}$ of tuples, each of arity n, we use $\mathcal{T}_{|m}$ to denote the set of tuples in $\mathcal{T}$ projected on their first $m \leq n$ positions. For example, if $\mathcal{T} = \{(a, a), (a, b), (a, c), (b, a), (c, a)\}$, then $\mathcal{T}_{|1} = \{(a), (b), (c)\}$ and $\mathcal{T}_{|0} = \{()\}$. We use a similar notation for projecting a particular tuple: if $t = (a, b)$ then $t_{|1} = (a)$ and $t_{|0} = ()$. We also use $t(x)$ to denote the value of the position of x in t. For example, if x was the first introduced bound first-order variable, then $t(x) = a$ for $t = (a, b)$.

We let *changed*(d) denote the set of tuples that has affected node d. We let *bounds*(d) denote the number of bound first-order variables at node d (which is equal to the number of nodes of the form $\forall x$ or $\exists x$ on the path from d to the root). We use the operator $\oplus$ for replacing the current bindings of a mapping with new ones. For example, the result of $\{x \mapsto a, y \mapsto a, z \mapsto b\} \oplus \{x \mapsto b, y \mapsto b\}$ is $\{x \mapsto b, y \mapsto b, z \mapsto b\}$. Finally, we assume that nodes of the form $\exists x$ and $\vee$ have a binary search tree *RBTree* for maintaining the minimum value of each of its penalty mappings.

Now, given a change to a current configuration k, resulting in ℓ, assume that $\mathcal{X}$ is the set of affected set variables in a formula Φ with an initialised penalty DAG D. The call $\mathit{submit}(d_S, \mathit{void}, \mathcal{T}_S)$ must be made for each leaf d_S of D that represents a subformula stated on any S in $\mathcal{X}$, where $\mathcal{T}_S$ is the set of affected tuples with respect to the change of $k(S)$ to $\ell(S)$.

Example 9. Recall $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$ of Example 1, and keep the initialised DAG D of Figure 2 in mind. Let $\ell = \{S \mapsto \{a, b\}, T \mapsto \{b\}\}$. We will now explain how D is updated with respect to ℓ. The only paths that must be updated are the ones leading from leaves that contain T to the root node (marked by directed arcs in Figure 2).

There is only one leaf in D that contains T, namely the leaf representing the subformula $x \in T$. Starting at this leaf, *submit* is called with $\mathit{submit}(x \in T, \mathit{void}, \{(b)\})$. This gives the call $\mathit{update}(x \in T, \mathit{void}, \{(b)\})$ which replaces the binding of (b) in $p(x \in T)$ with $(b) \mapsto 0$ (since b is now in T). Since a leaf node has no children and $x \in T$ is not the root, *submit* is called for each parent of $x \in T$. This results in the calls $\mathit{submit}(d_\wedge, x \in T, \{(b)\})$ and $\mathit{submit}(d_\vee, x \in T, \{(b)\})$, where $d_\wedge$ and $d_\vee$ are respectively the conjunctive and disjunctive parents of $x \in T$.

Consider first $\mathit{submit}(d_\wedge, x \in T, \{(b)\})$. The call $\mathit{update}(d_\wedge, x \in T, \{(b)\})$ is made, which replaces the binding of (b) in $p(d_\wedge)$ with $(b) \mapsto 1$ (since the sum of the penalties of the children of $d_\wedge$ with respect to (b) is now 1). Since all affected children of $d_\wedge$ are done, *submit* is called for its only parent, resulting in the call $\mathit{submit}(d_\exists, d_\wedge, \{(b)\})$. The subsequent call $\mathit{update}(d_\exists, d_\wedge, \{(b)\})$ does not change the penalty mapping of $p(d_\exists)$ (since the minimum of the penalties of $d_\wedge$ with respect to each value in the universe is still 1). Hence there is no change to propagate to the parent of $d_\exists$.

Consider now $\mathit{submit}(d_\vee, x \in T, \{(b)\})$. The call $\mathit{update}(d_\vee, x \in T, \{(b)\})$ is made, which replaces the binding of (b) in $p(d_\vee)$ with $(b) \mapsto 0$ (since the minimum of the penalties of the children of $d_\vee$ with respect to (b) is now 0). Since all affected children of $d_\vee$ are done, *submit* is called for its only parent, resulting in the call $\mathit{submit}(d_\forall, d_\wedge, \{(b)\})$. The subsequent call $\mathit{update}(d_\forall, d_\wedge, \{(b)\})$ replaces the penalty mapping of $p(d_\forall)$ with $() \mapsto 1$ (since the sum of the penalties of $d_\vee$ with respect to each value in the universe is now 1). The call to $\mathit{submit}(d'_\wedge, d_\forall, \{b\})$, where $d'_\wedge$ is the parent of $d_\forall$, results in replacing the penalty mapping of $p(d'_\wedge)$ with $() \mapsto 2$ and, finally, also in replacing the penalty mapping of the root with $() \mapsto 2$.

Adding b to T resulted in a penalty decrease of 1 and, hence, the penalty of (4) with respect to ℓ is 2. The changed penalty mappings with respect to ℓ are shown in Figure 2 in parentheses along the oriented edges in the DAG. The changed variable conflicts with respect to ℓ are also shown.

5.3 Complexity

We will now present the worst-case space complexity of a penalty and variable-conflict DAG of a formula, as well as the worst-case time complexities of the algorithms of the previous sub-section.

Given is a penalty and variable-conflict DAG D of a formula Φ. In the following,

- let b be the maximum number of nested first-order quantifiers in Φ;
- let m be the number of disjuncts of the longest disjunctive subformula in Φ;
- let n be the number of set variables in Φ;
- let q be the number of nodes in D;
- let t be the number of tuples in $\mathcal{T}$ (recall from Section 5.2 that $\mathcal{T}$ is the set of tuples that are affected by changing a configuration k to another configuration ℓ);
- let u be the size of $\mathcal{U}$;
- let v be the maximum number of paths from leaf nodes to the root of D;
- let w be the maximum number of nodes on paths from leaf nodes to the root of D.

In each node d of D, the number of penalty mappings that must be stored at d is upper bounded by u^b. The number of variable-conflict mappings that must be stored at d is then upper bounded by $u^b \cdot n$.

The worst-case space complexity of D is $\mathcal{O}(q \cdot u^b \cdot n)$. Note that many interesting constraints can be modelled with $b = 1$. See, for example, (4) and (7) as well as those in [16].

The worst-case time complexity of a call of the form *initialise*$(D,\mathcal{U},k)$ is $\mathcal{O}(q \cdot u^b)$, since the function visits each node d in D exactly once, updating the penalty mappings of d.

Let d_t be the node representing a subformula ϕ, let d_f be a successor of d_t, and let $\mathcal{T}$ be the affected tuples with respect to some change from a configuration k to another configuration ℓ. The worst-case time complexity of a call of the form *update*$(d_t, d_f, \mathcal{T})$ is $\mathcal{O}(t \cdot t \cdot \gamma)$ where

$$\gamma = \begin{cases} 1, \text{ if no formulas of the form } \exists\psi \text{ or } \psi_1 \vee \cdots \vee \psi_r \text{ are subformulas in } \Phi, \text{ or} \\ \log m, \text{ if no formulas of the form } \exists\psi \text{ are subformulas in } \Phi, \text{ or} \\ \log u, \text{ if no formulas of the form } \psi_1 \vee \cdots \vee \psi_r \text{ are subformulas in } \Phi, \text{ or} \\ \log(\max\{u, m\}), \text{ otherwise.} \end{cases}$$

The penalty mappings with respect to each element in $\mathcal{T}_{|bounds(d_t)}$ are updated (first factor t). Each such update may (in the cases where ϕ is of the form

$\forall\psi$ or $\exists\psi$) have to be calculated with respect to all elements in $\mathcal{T}_{|bounds(d_f)}$ (second factor t). Note that $|\mathcal{T}_{|bounds(d_t)}|$ and $|\mathcal{T}_{|bounds(d_f)}|$ are upper bounded by t. Finally, each actual update may (in the cases where ϕ is of the form $\exists\psi$ or $\psi_1 \vee \cdots \vee \psi_r$) imply updating a balanced binary search tree with u (when ϕ is of the form $\exists\psi$) or m (when ϕ is of the form $\psi_1 \vee \cdots \vee \psi_r$) nodes (factor γ).

The worst-case time complexity of a call of the form *submit*$(d, void, \mathcal{T})$ is $\mathcal{O}(v \cdot w \cdot t^2 \cdot \gamma)$. Each node on any path from the leaf node d to the root node is visited (factor $v \cdot w$). For each such node a call to *update* is made (factor $t^2 \cdot \gamma$).

The most interesting part above is the complexity of a call to the *submit* function, since calls to that function will be made each time a change from a configuration k to another configuration ℓ is made. As seen above, this complexity is $\mathcal{O}(v \cdot w \cdot t^2 \cdot \gamma)$ which may seem high at first. However, for many formulas, including (4) above and (7) below, most of the factors are constant. For example, if the maximum number of nested first-order quantifiers in Φ is $b = 1$, then the factors t are constant, bringing down the complexity to $\mathcal{O}(v \cdot w \cdot \gamma)$. Furthermore, the maximum number of nodes on paths from leaf nodes to the root of D is often constant, bringing down the complexity further to $\mathcal{O}(v \cdot \gamma)$, where v is often linear in n.

6 Application: The Progressive Party Problem

Let us now present some experimental results. Our objective is to show that one may use (global) constraints modelled in $\exists$MSO$^+$ in a local search framework, while still obtaining competitive results compared to built-in (global) constraints. Since no effort at all is then spent on defining and incrementally implementing penalty and variable-conflict functions for the modelled (global) constraints, as our generic algorithms take care of that, we bring constraint based local search a step closer to the level of expressiveness of constraint programming. We simulate this by assuming that a built-in global constraint is not available and modelling it by an $\exists$MSO$^+$ version thereof. By doing this, we may compare a modelled $\exists$MSO$^+$ version of a constraint with a built-in version, and hence evaluate the feasibility of our approach.

The *progressive party problem* (*PPP*) [15] is about timetabling a party at a yacht club, where the crews of certain boats (the guest boats) party at other boats (the host boats) over a number of periods. The crew of a guest boat must party at some host boat in each period (constraint c_1). The spare capacity of a host boat is never to be exceeded (constraint c_2). The crew of a guest boat may visit a particular host boat at most once (constraint c_3). The crews of two distinct guest boats may meet at most once (constraint c_4).

6.1 A Set Based Model

Let H and G be the sets of host boats and guest boats, respectively. Let *capacity*(h) and *size*(g) denote the spare capacity of host boat h and the crew size of guest boat g, respectively. Let P be the set of periods. Let $S_{(h,p)}$ be a set variable denoting the set of guest boats whose crews boat h hosts during period p. The following constraints then model the problem:

$$\begin{array}{l}(c_1) : \forall p \in P\ (\mathit{Partition}(\{S_{(h,p)} \mid h \in H\}, G)) \\ (c_2) : \forall h \in H\ (\forall p \in P\ (\mathit{MaxWeightedSum}(S_{(h,p)}, \mathit{size}, \mathit{capacity}(h)))) \\ (c_3) : \forall h \in H\ (\mathit{AllDisjoint}(\{S_{(h,p)} \mid p \in P\})) \\ (c_4) : \mathit{MaxIntersect}(\{S_{(h,p)} \mid h \in H \wedge\ p \in P\}, 1)\end{array} \tag{5}$$

The global constraint $\mathit{Partition}(\mathcal{X}, Q)(k)$ holds if and only if the values, with respect to k, of the set variables in $\mathcal{X}$ partition the set Q, where the value of a set variable in $\mathcal{X}$ may be the empty set. The constraint $\mathit{MaxWeightedSum}(S, w, m)(k)$ holds if and only if $\sum_{u \in k(S)} w(u) \leq m$. The global constraint $\mathit{MaxIntersect}(\mathcal{X}, m)(k)$ holds if and only if the cardinality of the intersection with respect to k between any two distinct set variables in $\mathcal{X}$ is at most m. The global constraint $\mathit{AllDisjoint}(\mathcal{X})(k)$ holds if and only if the intersection with respect to k between any two distinct set variables in $\mathcal{X}$ is empty.

6.2 Modelling *AllDisjoint* in $\exists$MSO$^+$

In our local search system, the four different constraints above are all built in, having specialised penalty and variable-conflict measures as well as incremental algorithms for maintaining these measures. We take a closer look at these elements for the *AllDisjoint* constraint in the following example.

Example 10. The following penalty function:

$$\mathit{penalty}(\mathit{AllDisjoint}(\mathcal{X}))(k) = \left(\sum_{S \in \mathcal{X}} |k(S)|\right) - \left|\bigcup_{S \in \mathcal{X}} k(S)\right| \tag{6}$$

computes the length of the shortest sequence of elementary set operations needed to decrease $\mathit{penalty}(\mathit{AllDisjoint}(\mathcal{X}))(k)$ to zero. For instance, the penalty of $\mathit{AllDisjoint}(\{S, T, V\})$ under configuration $k = \{S \mapsto \{a, b, c\}, T \mapsto \{b, c, d\}, V \mapsto \{d, e\}\}$ is $8 - 5 = 3$, and indeed it suffices to drop the three shared elements b, c, d from any set each to get a solution. The following variable conflict function:

$$\mathit{conflict}(\mathit{AllDisjoint}(\mathcal{X}))(S, k) = |\{u \in k(S) \mid \exists T \in \mathcal{X} \setminus \{S\} : u \in k(T)\}|$$

computes the length of the shortest sequence of elementary set operations needed to decrease $\mathit{conflict}(\mathit{AllDisjoint}(\mathcal{X}))(S, k)$ to zero. For instance, the conflict of variable S with respect to the penalty and configuration above is 2, and indeed it suffices to drop the two elements b, c it shares with other sets to get a zero conflict of S (but not a zero penalty).

In order to maintain (6) incrementally, we use a table *count* of integers, indexed by the values in $\mathcal{U}$, such that $\mathit{count}[u]$ is equal to the number of variables in $\mathcal{X}$ that contain u. Now, the sum (6) is equal to $\sum_{u \in \mathcal{U}} \max(\mathit{count}[u] - 1, 0)$ as it suffices to drop each value $u \in \bigcup_{S \in \mathcal{X}} k(S)$ from all but one of the set variables in $\mathcal{X}$ in order to satisfy the constraint. This is easy to maintain in $\mathcal{O}(1)$ time given an elementary set operation of the form $\mathit{add}(S, u)$ or $\mathit{drop}(S, u)$.

Assume now that the global *AllDisjoint* constraint is not available in our local search system. We may then model it in $\exists$MSO$^+$ and use that version in (5) instead of (c_3). Hence, we may experiment with the $\exists$MSO$^+$ version without having to come up with penalty and variable-conflict measures and implement incremental algorithms for a new built-in global constraint. We use the following $\exists$MSO$^+$ formula for modelling the global $\mathit{AllDisjoint}(\{S_1, \ldots, S_n\})$ constraint:

$$\begin{aligned}\exists S_1 \cdots \exists S_n \forall x\ (\ & (x \notin S_1 \lor (x \notin S_2 \land \cdots \land x \notin S_n)) \land \\ & (x \notin S_2 \lor (x \notin S_3 \land \cdots \land x \notin S_n)) \land \cdots \land \\ & (x \notin S_{n-1} \lor x \notin S_n))\end{aligned} \tag{7}$$

For every value u in the universe we state that: if u is in a set S_i, then u cannot be in any set S_j where $j > i$. This encoding is quadratic in the number of variables n, which is inevitable since there must be a subformula for every pair of different variables in $\{S_1, \ldots, S_n\}$. The chosen experiment is thus quite challenging since we replace a constant size encoding having a constant time incremental penalty maintenance by a quadratic sized encoding. However, as we show in Section 6.4, the penalty of the quadratic encoding is incrementally maintained in linear time.

6.3 Solving the Model

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 3 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 $\mathit{penalty}(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(k) = 0$, then a solution was found within *MaxIter* iterations.

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 *MaxIter* (we empirically chose $\mathit{MaxIter} = 500,000$ for the experiments in this article) 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 consisting of transferring an element from $S_{(h,p)}$ to another variable in the same period. (line 11). (iii) Move to a neighbour ℓ that minimises the penalty (lines $12-14$).

In order to escape local minima it also uses a tabu list [9] and a restarting component. The tabu list *tabu* is initially empty. When a move from a

Algorithm 3 Solving the PPP

```
1: procedure $solve\_progressive\_party(P, G, H, capacity, size)$
2:     Initialise $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ 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 such that $c_1$ is respected
6:         Add a random mapping $S_{(h,p)} \mapsto G'$, where $G' \subset G$, for each $h \in H$ to $k$
7:         such that $penalty(Partition(\{S_{(h,p)} \mid h \in H\}, G))(k) = 0$
8:     while $penalty(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(k) > 0$ & $iteration < MaxIter$ do
9:         $iteration \leftarrow iteration + 1$, $non\_improving \leftarrow non\_improving + 1$
10:        $choose$ $S_{(h,p)} \in \mathcal{X}$ such that $\forall T \in \mathcal{X} : conflict(\mathcal{C})(S_{(h,p)}, k) \geq$
               $conflict(\mathcal{C})(T, k)$
11:        $N \leftarrow transfer(S_{(h,p)}, \{S_{(h',p)} \mid h' \in H$ & $h' \neq h\})(k)$
12:        $choose$ $\ell \in N$ such that $\forall \ell' \in N : penalty(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(\ell) \leq$
               $penalty(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(\ell')$
13:        and $((S_{(h',p)}, d, iteration) \notin tabu$ or $penalty(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(\ell) < best)$,
14:        where $delta(k, \ell) = \{(S_{(h,p)}, \{d\}, \emptyset), (S_{(h',p)}, \emptyset, \{d\})\}$
15:        $k \leftarrow \ell$, $tabu \leftarrow tabu \cup \{(S_{(h',p)}, d, iteration + rand\_int(5, 40))\}$
16:        if $penalty(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(k) < best$ then
17:            $best \leftarrow penalty(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(k)$, $non\_improving \leftarrow 0$,
18:            $history \leftarrow \{k\}$, $tabu \leftarrow \emptyset$
19:        else if $penalty(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(k) = best$ 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 $k$
```

configuration k to a configuration ℓ 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 $(S_{(h',p)}, d, iteration + t)$ 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 interval). However, if such a move would give the lowest penalty so far, it is always accepted (lines $13-15$). By abuse of notation, we let $(s, d, t) \notin tabu$ be false if and only if $(s, d, t') \in tabu$ & $t \leq t'$.

The restarting component (lines $16-23$) works in the following way. Each configuration k such that $penalty(\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle)(k)$ is at most the current lowest penalty is stored in the set $history$ (lines $16-20$). If a number $MaxNonImproving$ (we empirically chose $MaxNonImproving = 500$ for the experiments in this article) 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 [12, 19] (saving one best configuration) and [6] (saving a set of best configurations), both for scalar models of the PPP.

Table 1. Run times in seconds for the PPP with the $\exists$MSO$^+$ and built-in *AllDisjoint* constraint respectively. Mean run time of successful runs (out of 100) and number of unsuccessful runs (if any) in parentheses.

Using the modelled *AllDisjoint* constraint.

H/periods (fails)	6	7	8	9	10
(a) 1-12,16			1.3	3.5	42.0
(b) 1-13			16.5	239.3	
(c) 1,3-13,19			18.9	273.2 (3)	
(d) 3-13,25,26			36.5	405.5 (16)	
(e) 1-11,19,21	19.8	186.7			
(f) 1-9,16-19	32.2	320.0 (12)			

Using the built-in *AllDisjoint* constraint (values from [3]).

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

6.4 Results

We have run the same classical instances as in [20, 12, 6, 3] on an Intel 2.4 GHz Linux machine with 512 MB memory. In Table 1 we show the results when using the $\exists$MSO$^+$ and built-in versions of the *AllDisjoint* constraint. The run times for the $\exists$MSO$^+$ version are only 2 to 3 times higher, for all the instances, though it must be noted that efforts such as designing penalty and variable-conflict functions as well as incremental penalty and variable-conflict maintenance algorithms for *AllDisjoint* were not necessary. The slowdown comes from the difference in complexity of updating the constraints given an elementary set operation, as shown next.

Given the penalty and variable-conflict DAG $D_{(7)}$ of the formula (7) for n set variables and a universe of size u, the following properties can be proved by construction:

- the maximum number of nested first-order quantifiers in (7) is $b = 1$;
- the number of disjuncts of the longest disjunctive subformula in (7) is $m = 2$;
- the number of nodes in $D_{(7)}$ is $q = 3 \cdot n$;
- the maximum number of paths from leaf nodes to the root of $D_{(7)}$ is $v = n-1$;
- the maximum number of nodes on paths from leaf nodes to the root of $D_{(7)}$ is $w = 5$.

Let t be the number of tuples in $\mathcal{T}$, which is a constant value for (7) since the maximum number of nested first-order quantifiers in (7) is $b = 1$. Here we assume that $t = 1$, which is the case if we consider updating $D_{(7)}$ with respect to one elementary set operation of the form *add*(S, u) or *drop*(S, u). Recall now the worst-case space and time complexities of Section 5.3 with respect to a penalty and variable-conflict DAG D of a formula Φ. We obtain the following worst-case space and time complexities for (7) and $D_{(7)}$:

– The worst-case space complexity of $D_{(7)}$ is:

$$\mathcal{O}(q \cdot u^b \cdot n) = \mathcal{O}(3 \cdot n \cdot u^1 \cdot n) = \mathcal{O}(n^2 \cdot u)$$

– The worst-case time complexity of a call of the form *initialise*$(D_{(7)}, \mathcal{U}, k)$ is:

$$\mathcal{O}(q \cdot u^b) = \mathcal{O}(3 \cdot n \cdot u^1) = \mathcal{O}(n \cdot u)$$

– The worst-case time complexity of a call of the form *update*$(d_t, d_f, \mathcal{T})$ is:

$$\mathcal{O}(t \cdot t \cdot \gamma) = \mathcal{O}(1 \cdot 1 \cdot \log 2) = \mathcal{O}(1)$$

where $\gamma = \log 2$ since there are disjunctive formulas but no formulas of the form $\exists \phi$ are subformulas in (7).

– The worst-case time complexity of a call of the form *submit*$(d, \mathit{void}, \mathcal{T})$ is

$$\mathcal{O}(v \cdot w \cdot t^2 \cdot \gamma) = \mathcal{O}((n-1) \cdot 5 \cdot 1^2 \cdot \log 2) = \mathcal{O}(n)$$

where, again, $\gamma = \log 2$ as explained above.

Hence, incrementally updating the penalty and variable-conflict DAG $D_{(7)}$ of (7) with respect to an elementary set operation is linear in the number of variables as this can be done by one call to the function *submit*. This compares to incrementally updating (6) for the built-in *AllDisjoint* constraint, which can be done in constant time. Of course, the built-in version may take advantage of global properties of the constraint that make it possible to define a faster incremental algorithm. However, as the results show, the overall performance for using the modelled *AllDisjoint* is still acceptable, making a modelled constraint in $\exists$MSO$^+$ a good alternative when a particular constraint is not built in. Note also that the robustness of the local search algorithm does not degrade when using the $\exists$MSO$^+$ version of the *AllDisjoint* constraint, as witnessed by the number of solved instances when using the $\exists$MSO$^+$ version being at least as large as the number of solved instances when using the built-in version.

7 Conclusion

There is a need for more user support in local search. This is witnessed by, e.g, the complexity of coming up with new (global) constraints. To do this, the user must define penalty and variable-conflict measures, as well as implement efficient incremental algorithms for maintaining these measures. The need to perform these tasks, assuming that the user has the necessary skills, clearly reduces her productivity, since much time must usually be devoted to them.

In this article, we introduced tools to help overcome this situation. Towards this, we proposed the usage of monadic existential second-order logic extended with counting ($\exists$MSO$^+$) as a modelling language for (user defined) constraints in local search. Furthermore, we introduced inductive definitions for measuring the penalty and variable conflicts of an $\exists$MSO$^+$ formula with respect to a given configuration. We also showed that the proposed measure of the conflict of a variable x is lower-bounded by the intuitive target value, i.e., the maximum penalty decrease that may be achieved by only changing the value of x, as well as upper bounded by the penalty of the formula. Without these important properties, the local search performance may degrade.

On the practical side, we came up with and implemented incremental algorithms for maintaining penalty and variable conflicts of $\exists$MSO$^+$ formulas according to the proposed measures. Using these algorithms, we replaced a built-in global *AllDisjoint* constraint by an $\exists$MSO$^+$ version thereof in a set based model of the progressive party problem. The results show the usefulness of the approach by the $\exists$MSO$^+$ version not incurring too high losses in run time, and by being at least as robust in terms of the number of solved instances, compared to the built-in version. Had that *AllDisjoint* constraint not been built-in, the user would have obtained those rather efficient incremental algorithms at the low cost of just modelling that constraint.

The adaptation of the traditional combinators of constraint programming for local search was pioneered for the *Comet* system [19, 17]. The combinators there include logical connectives (such as $\wedge$ and $\vee$), cardinality operators (such as *exactly* and *atmost*), reification, and expressions over variables. We have extended these ideas here to the logical quantifiers, namely $\forall$ and $\exists$. This is not just a matter of simply generalising the arities and the existing definitions [19, 17] of the penalties and variable conflicts for the $\wedge$ and $\vee$ connectives, respectively, but was made necessary by our handling of set variables over which one would like to iterate, unlike the scalar variables of *Comet*.

In parallel and independently of our work on a generic variable-conflict function for $\exists$MSO$^+$ formulas [4], the work of [11, 12, 19, 17] was generalised in order to introduce differentiable invariants [18]. The latter are a novel, unifying abstraction for local search that lifts arbitrary expressions and formulas into differentiable objects [12, 17], which incrementally maintain a value, its maximum possible increase and decrease, and can be queried for the effects of local moves. Generic definitions and incremental maintenance algorithms for penalty and variable-conflict functions, such as ours, are thus inherited for free as particular cases of the differentiable-invariant calculus [18]. While our penalty and variable-conflict functions (when adapted to scalar variables) seem to give the same results as theirs and enjoy the same desirable properties (which in their case follow directly by structural induction on the expression), it should be noted that they also support negation and reification. Our work, which took place independently and in parallel, is not completely subsumed by their development of the state of the art, as we support second-order set variables, unlike their scalar variables, and as we also address first-order bounded quantification.

Acknowledgements

This research was partially funded by EuroControl Project C/1.246/HQ/JC/04 and its Amendments 1/04 and 2/05. We thank Christian Schulte and the anonymous referees of [3, 2, 4] and this special issue for their useful comments. The second author thanks Sabancı University, for the research time included in his Visiting Faculty Member position in 2006/07.

References

1. Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.
2. Magnus Ågren, Pierre Flener, and Justin Pearson. Incremental algorithms for local search from existential second-order logic. In Peter van Beek, editor, *Proceedings of CP'05*, volume 3709 of *LNCS*, pages 47–61. Springer-Verlag, 2005.
3. Magnus Ågren, Pierre Flener, and Justin Pearson. Set variables and local search. In Roman Barták and Michela Milano, editors, *Proceedings of CP-AI-OR'05*, volume 3524 of *LNCS*, pages 19–33. Springer-Verlag, 2005.
4. Magnus Ågren, Pierre Flener, and Justin Pearson. Inferring variable conflicts for local search. In Frédéric Benhamou, editor, *Proceedings of CP'06*, volume 4204 of *LNCS*, pages 665–669. Springer-Verlag, 2006.
5. Francisco Azevedo and Pedro Barahona. Applications of an extended set constraint solver. In *Proceedings of the ERCIM / CompulogNet Workshop on Constraints*, 2000.
6. Markus Bohlin. *Design and Implementation of a Graph-Based Constraint Model for Local Search*, 2004. PhL thesis, Mälardalen University, Västerås, Sweden.
7. Philippe Galinier and Jin-Kao Hao. A general approach for constraint solving by local search. In *Proceedings of CP-AI-OR'00*, 2000.
8. Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
9. Fred Glover and Manuel Laguna. Tabu search. In *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–150. John Wiley & Sons, 1993.
10. Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
11. 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*, pages 237–251. Springer-Verlag, 1997.
12. 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*.
13. Alexander Nareyek. Using global constraints for local search. In E.C. Freuder and R.J. Wallace, editors, *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 9–28. American Mathematical Society, 2001.
14. Jean-François Puget. Finite set intervals. In *Proceedings of CP'96 Workshop on Set Constraints*, 1996.
15. Barbara M. Smith, Sally C. Brailsford, Peter M. Hubbard, and H. Paul Williams. The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.

16. Guido Tack, Christian Schulte, and Gert Smolka. Generating propagators for finite set constraints. In Frédéric Benhamou, editor, *Proceedings of CP'06*, volume 4204 of *LNCS*, pages 575–589. Springer-Verlag, 2006.
17. Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.
18. Pascal Van Hentenryck and Laurent Michel. Differentiable invariants. In Frédéric Benhamou, editor, *Proceedings of CP'06*, volume 4204 of *LNCS*, pages 604–619. Springer-Verlag, 2006.
19. Pascal Van Hentenryck, Laurent Michel, and Liyuan Liu. Constraint-based combinators for local search. In Mark Wallace, editor, *Proceedings of CP'04*, volume 3258 of *LNCS*, pages 47–61. Springer-Verlag, 2004.
20. Joachim Paul Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *LNCS*. Springer-Verlag, 1999.

A Proofs

In order to simplify the proofs, we assume that a formula of the form $\forall x\phi$ is replaced by the equivalent formula $(\phi_1 \wedge (\phi_2 \wedge \cdots \wedge (\phi_{n-1} \wedge \phi_n)\cdots))$, where ϕ_i denotes the formula ϕ in which any occurrence of x is replaced by u_i and where $\mathcal{U} = \{u_1, \dots, u_n\}$ with $n \geq 2$. Similarly, a formula of the form $\exists x\phi$ is replaced by the equivalent formula $(\phi_1 \vee (\phi_2 \vee \cdots \vee (\phi_{n-1} \vee \phi_n)\cdots))$.

For example, assuming that $\mathcal{U} = \{a, b\}$, the formula stating that S and T have at most one common element:

$$\exists S \exists T(\forall x(\forall y(x \geq y \vee (x \notin S \vee y \notin S \vee x \notin T \vee y \notin T))))$$

is replaced by:

$$\begin{aligned}\exists S \exists T\ &(a \geq a \vee (a \notin S \vee a \notin S \vee a \notin T \vee a \notin T)) \wedge\\ &(a \geq b \vee (a \notin S \vee b \notin S \vee a \notin T \vee b \notin T)) \wedge\\ &(b \geq a \vee (b \notin S \vee a \notin S \vee b \notin T \vee a \notin T)) \wedge\\ &(b \geq b \vee (b \notin S \vee b \notin S \vee b \notin T \vee b \notin T)).\end{aligned}$$

A.1 Proof of Proposition 3

Proposition 3. The function induced by Definition 7 is a penalty function according to Definition 3.

Proof. Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP, let $\Phi \in \mathcal{C}$ be a formula in $\exists\text{MSO}^+$, and let $k \in \mathcal{K}_P$. We must show that:

1. $\mathit{penalty}(\Phi)(k) \geq 0$.
2. $\mathit{penalty}(\Phi)(k) = 0 \Leftrightarrow k \models \Phi$.

Consider first 1. The proof is by structural induction on Φ. The result holds for the base cases (f), (g), and (h). For case (a), the result follows by induction from the definition.

Case $\Phi = \phi \wedge \psi$. We have that $penalty(\phi)(k) \geq 0$ and that $penalty(\psi)(k) \geq 0$ by induction. Hence we must have that

$$penalty(\phi \wedge \psi)(k) = penalty(\phi)(k) + penalty(\psi)(k) \geq 0.$$

Case $\Phi = \phi \vee \psi$. We have that $penalty(\phi)(k) \geq 0$ and that $penalty(\psi)(k) \geq 0$ by induction. Hence we must have that

$$penalty(\phi \vee \psi)(k) = \min\{penalty(\phi)(k), penalty(\psi)(k)\} \geq 0.$$

Consider now 2 and ($\Rightarrow$). Assume that $penalty(\Phi)(k) = 0$. The proof is by structural induction on Φ. The result holds for the base cases (f), (g), and (h). For case (a), the result follows by induction from the definition.

Case $\Phi = \phi \wedge \psi$. We have that

$$penalty(\phi \wedge \psi)(k) = penalty(\phi)(k) + penalty(\psi)(k) = 0$$

by assumption and we must show that $k \models \phi \wedge \psi$, i.e., that $k \models \phi$ and that $k \models \psi$. We know that $penalty(\phi)(k) = 0$ and that $penalty(\psi)(k) = 0$ (since $penalty(\phi)(k) \geq 0$ and $penalty(\psi)(k) \geq 0$ by 1 above) and, hence, the result follows by induction.

Case $\Phi = \phi \vee \psi$. We have that

$$penalty(\phi \vee \psi)(k) = \min\{penalty(\phi)(k), penalty(\psi)(k)\} = 0$$

by assumption and we must show that $k \models \phi \vee \psi$, i.e., that $k \models \phi$ or that $k \models \psi$. Since either $penalty(\phi)(k) = 0$ or $penalty(\psi)(k) = 0$, the result follows by induction.

Consider now 2 and ($\Leftarrow$). Assume that $k \models \Phi$. The proof is by structural induction on Φ. The result holds for the base cases (f), (g), and (h). For case (a), the result follows by induction from the definition.

Case $\Phi = \phi \wedge \psi$. We have that $k \models \phi \wedge \psi$ by assumption and we must show that $penalty(\phi \wedge \psi)(k) = penalty(\phi)(k) + penalty(\psi)(k) = 0$, i.e., that $penalty(\phi)(k) = 0$ and that $penalty(\psi)(k) = 0$. Since $k \models \phi \wedge \psi$ we have that $k \models \phi$ and that $k \models \psi$ and, hence, the result follows by induction.

Case $\Phi = \phi \vee \psi$. We have that $k \models \phi \vee \psi$ by assumption and we must show that $penalty(\phi \vee \psi)(k) = \min\{penalty(\phi)(k), penalty(\psi)(k)\} = 0$, i.e., that $penalty(\phi)(k) = 0$ or that $penalty(\psi)(k) = 0$. Since $k \models \phi \vee \psi$ we have that $k \models \phi$ or that $k \models \psi$ and, hence, the result follows by induction. $\square$

A.2 Proof of Proposition 4

Proposition 4. Let Φ be a formula in $\exists\text{MSO}^+$, let k be a configuration for the set variables in Φ, and let S be one of those variables. Then $conflict(\Phi)(S, k) \geq abstractConflict(\Phi)(S, k)$.

Proof. The proof is by structural induction on Φ. The result holds for the base cases (f) and (g). For case (a), the result follows by induction from the definition.

Case $\Phi = \phi \wedge \psi$. We have that

$$\begin{aligned}&\mathit{abstractConflict}(\phi \wedge \psi)(S,k) = \\ &\qquad \max\{(\mathit{penalty}(\phi)(k) + \mathit{penalty}(\psi)(k)) - \\ &\qquad\quad (\mathit{penalty}(\phi)(\ell) + \mathit{penalty}(\psi)(\ell)) \mid \ell \in n_S(k)\} = \\ &\qquad \max\{\mathit{penalty}(\phi)(k) - \mathit{penalty}(\phi)(\ell) + \\ &\qquad\quad \mathit{penalty}(\psi)(k) - \mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\} = e.\end{aligned}$$

Now, to see that

$$\begin{aligned}e \leq\ &\max\{\mathit{penalty}(\phi)(k) - \mathit{penalty}(\phi)(\ell') \mid \ell' \in n_S(k)\} + \\ &\qquad \max\{\mathit{penalty}(\psi)(k) - \mathit{penalty}(\psi)(\ell'') \mid \ell'' \in n_S(k)\} = f\end{aligned}$$

we pick an $\ell \in n_S(k)$ that maximises

$$\mathit{penalty}(\phi)(k) - \mathit{penalty}(\phi)(\ell) + \mathit{penalty}(\psi)(k) - \mathit{penalty}(\psi)(\ell).$$

For that ℓ we have that either it maximises both $\mathit{penalty}(\phi)(k) - \mathit{penalty}(\phi)(\ell)$ and $\mathit{penalty}(\psi)(k) - \mathit{penalty}(\psi)(\ell)$, or there exist $\ell', \ell'' \in n_S(k)$ that make the sum of those expressions larger than e. Now

$$f = \mathit{abstractConflict}(\phi)(S,k) + \mathit{abstractConflict}(\psi)(S,k)$$

by definition. By induction it then follows that

$$f \leq \mathit{conflict}(\phi)(S,k) + \mathit{conflict}(\psi)(S,k) = \mathit{conflict}(\phi \wedge \psi)(S,k).$$

Case $\Phi = \phi \vee \psi$. We have that

$$\begin{aligned}&\mathit{abstractConflict}(\phi \vee \psi)(S,k) = \\ &\qquad \max\{\underbrace{\min\{\overbrace{\mathit{penalty}(\phi)(k)}^{\alpha}, \overbrace{\mathit{penalty}(\psi)(k)}^{\beta}\}}_{A} - \\ &\qquad\quad \underbrace{\min\{\overbrace{\mathit{penalty}(\phi)(\ell)}^{\gamma}, \overbrace{\mathit{penalty}(\psi)(\ell)}^{\delta}\}}_{B} \mid \ell \in n_S(k)\} = e.\end{aligned}$$

Consider first the case where $A = \alpha$ (i.e., $\mathit{penalty}(\phi)(k) \leq \mathit{penalty}(\psi)(k)$) and $B = \gamma$ above for an $\ell \in n_S(k)$ that maximises $A - B$. Then we have that

$$e = \max\{\mathit{penalty}(\phi)(k) - \mathit{penalty}(\phi)(\ell) \mid \ell \in n_S(k)\} = \mathit{abstractConflict}(\phi)(S,k).$$

By definition we have that

$$\begin{aligned}&\mathit{conflict}(\phi \vee \psi)(S,k) = \mathit{penalty}(\phi \vee \psi)(k) - \\ &\min\{\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k), \mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k)\} = f.\end{aligned}$$

Assume first that

$$f = \mathit{penalty}(\phi \vee \psi)(k) - (\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k)).$$

This simplifies into $f = \mathit{conflict}(\phi)(S,k)$ and the result follows by induction.

Assume now that

$$f = \mathit{penalty}(\phi \vee \psi)(k) - (\mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k))$$

which can be simplified and rewritten as

$$\begin{aligned}f = \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k) + \mathit{conflict}(\psi)(S,k) = \\ \mathit{conflict}(\psi)(S,k) + (\mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k)).\end{aligned}$$

We have that

$$\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k) \geq \mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k)$$

and consequently that

$$\mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k) \geq \mathit{conflict}(\phi)(S,k) - \mathit{conflict}(\psi)(S,k).$$

Hence

$$\begin{aligned}f = \mathit{conflict}(\psi)(S,k) + (\mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k)) \geq \\ \mathit{conflict}(\psi)(S,k) + (\mathit{conflict}(\phi)(S,k) - \mathit{conflict}(\psi)(S,k)) = \mathit{conflict}(\phi)(S,k)\end{aligned}$$

and the result follows by induction.

The case where $A = \beta$ and $B = \delta$ above for an $\ell \in n_S(k)$ that maximises $A - B$ is symmetric to the previous.

Consider now the case where $A = \alpha$ (i.e., $\mathit{penalty}(\phi)(k) \leq \mathit{penalty}(\psi)(k)$) and $B = \delta$ above for an $\ell \in n_S(k)$ that maximises $A - B$. Then we have that

$$e = \max\{\mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\}.$$

By definition we have that

$$\begin{aligned}\mathit{conflict}(\phi \vee \psi)(S,k) = \mathit{penalty}(\phi \vee \psi)(k) - \\ \min\{\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k), \mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k)\} = f.\end{aligned}$$

Assume first that

$$f = \mathit{penalty}(\phi \vee \psi)(k) - (\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k))$$

which simplifies into $f = \mathit{conflict}(\phi)(S,k)$.

We have that

$$\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k) \leq \mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k)$$

and consequently by induction that

$$\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k) \leq \mathit{penalty}(\psi)(k) - \mathit{abstractConflict}(\psi)(S,k).$$

By definition of abstract conflict we get

$$\begin{aligned}&\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k) \leq \\ &\qquad \mathit{penalty}(\psi)(k) - \max\{\mathit{penalty}(\psi)(k) - \mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\}\end{aligned}$$

which is equivalent to

$$\begin{aligned}&\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k) \leq \\ &\qquad \mathit{penalty}(\psi)(k) - (\mathit{penalty}(\psi)(k) - \min\{\mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\})\end{aligned}$$

which is simplified to

$$\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k) \leq \min\{\mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\}.$$

Hence we have that

$$\mathit{penalty}(\phi)(k) - \min\{\mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\} \leq \mathit{conflict}(\phi)(S,k)$$

which is equivalent to

$$e = \max\{\mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\} \leq \mathit{conflict}(\phi)(S,k) = f.$$

Assume now that

$$f = \mathit{penalty}(\phi \vee \psi)(k) - (\mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k))$$

which can be simplified and rewritten as

$$\begin{aligned}&f = \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k) + \mathit{conflict}(\psi)(S,k) = \\ &\qquad \mathit{conflict}(\psi)(S,k) + \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k).\end{aligned}$$

We have that

$$\begin{aligned}&f = \mathit{conflict}(\psi)(S,k) + \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k) \geq \\ &\qquad \mathit{abstractConflict}(\psi)(S,k) + \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k)\end{aligned}$$

by induction and that

$$\begin{aligned}&f = \mathit{conflict}(\psi)(S,k) + \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k) \geq \\ &\qquad \max\{\mathit{penalty}(\psi)(k) - \mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\} + \\ &\qquad\qquad \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k)\end{aligned}$$

by definition of abstract conflict. Rewriting the right hand side gives us

$$\begin{aligned}&f = \mathit{conflict}(\psi)(S,k) + \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k) \geq \\ &\qquad \mathit{penalty}(\psi)(k) - \min\{\mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\} + \\ &\qquad\qquad \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k)\end{aligned}$$

and consequently by cancelling terms

$$f = \mathit{conflict}(\psi)(S,k) + \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k) \geq \\ \mathit{penalty}(\phi)(k) - \min\{\mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\}$$

which is equivalent to

$$f = \mathit{conflict}(\psi)(S,k) + \mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(k) \geq \\ \max\{\mathit{penalty}(\phi)(k) - \mathit{penalty}(\psi)(\ell) \mid \ell \in n_S(k)\} = e.$$

The case where $A = \beta$ and $B = \gamma$ above for an $\ell \in n_S(k)$ that maximises $A - B$ is symmetric to the previous one. □

A.3 Proof of Proposition 5

Proposition 5. Let Φ be a formula in $\exists$MSO$^+$, let k be a configuration for the set variables in Φ, and let S be one of those variables. Then $\mathit{conflict}(\Phi)(S,k) \leq \mathit{penalty}(\Phi)(k)$.

Proof. The proof is by structural induction on Φ. The result holds for the base cases (f) and (g). For case (a), the result follows by induction from the definition.

Case $\Phi = \phi \wedge \psi$. We have that

$$\mathit{conflict}(\phi \wedge \psi)(S,k) = \mathit{conflict}(\phi)(S,k) + \mathit{conflict}(\psi)(S,k)$$

and that

$$\mathit{penalty}(\phi \wedge \psi)(k) = \mathit{penalty}(\phi)(k) + \mathit{penalty}(\psi)(k)$$

by definition. Since $\mathit{conflict}(\phi)(S,k) \leq \mathit{penalty}(\phi)(k)$ and $\mathit{conflict}(\psi)(S,k) \leq \mathit{penalty}(\psi)(k)$ by induction, we must have that

$$\mathit{conflict}(\phi)(S,k) + \mathit{conflict}(\psi)(S,k) \leq \mathit{penalty}(\phi)(k) + \mathit{penalty}(\psi)(k)$$

and hence $\mathit{conflict}(\phi \wedge \psi)(S,k) \leq \mathit{penalty}(\phi \wedge \psi)(k)$.

Case $\Phi = \phi \vee \psi$. By definition we have that

$$\mathit{conflict}(\phi \vee \psi)(S,k) = \mathit{penalty}(\phi \vee \psi)(k) - \\ \min\{\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k), \mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k)\} = e.$$

Assume first that

$$e = \mathit{penalty}(\phi \vee \psi)(k) - (\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k)).$$

The result follows directly since $\mathit{penalty}(\phi)(k) - \mathit{conflict}(\phi)(S,k) \geq 0$ by induction.

The case when

$$e = \mathit{penalty}(\phi \vee \psi)(k) - (\mathit{penalty}(\psi)(k) - \mathit{conflict}(\psi)(S,k))$$

is symmetric to the previous one. □