Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

Revisiting constraint-directed search

Magnus Ågren [a,*], Pierre Flener [c,b,1], Justin Pearson [b]

[a] SICS, Box 1263, SE-164 29 Kista, Sweden

[b] Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden

[c] Faculty of Engineering and Natural Sciences, Sabancı University, Orhanlı, Tuzla, TR-34956 İstanbul, Turkey

ARTICLE INFO

Article history:
Received 3 October 2008
Revised 3 October 2008
Available online 16 December 2008

Keywords:
Constraint-based local search
Constraint-directed search
Monadic existential second-order logic

ABSTRACT

In constraint-based local search the solutions are described declaratively by a conjunction of (often high-level) constraints. In this article we show that this opens up new ideas for constraint-directed search. For a constraint we introduce three neighbourhoods, where the penalty for that constraint alone is decreasing, increasing, or unchanged. We give specialised algorithms for common constraints that efficiently implement these neighbourhoods. Further, we give a general algorithm that implements these neighbourhoods from specifications of constraints in monadic existential second-order logic. Finally, we show how common constraint-directed local search algorithms are often easier to express using these neighbourhoods.

1. Introduction

Local search (e.g. [1]) starts from a possibly random initial configuration (assignment of values to all the variables) of a combinatorial problem. Each configuration has a penalty, which is zero if it is a solution to the problem. Local search iteratively makes small changes to the current configuration in an attempt to reduce its penalty, until either a solution is found or allocated computational resources have been consumed. The configurations examined for each such move constitute the neighbourhood of the current configuration. Heuristics are used to choose a neighbouring configuration, using only local information such as the current configuration and its neighbourhood, but occasionally guide the search to a local optimum. Metaheuristics such as tabu search [2] or simulated annealing [3] are thus needed to escape local optima and guide the search to a global optimum, using information collected or learned during the execution so far.

Constraint-based local search (CBLS, e.g. [4]) integrates ideas from constraint programming into local search. Of particular interest to this article is that rich modelling and search languages are offered towards a clean separation of the model and search components of a local search algorithm, via abstractions that facilitate its design and maintenance. One such abstraction is the concept of *constraint*, which captures some common combinatorial substructure. For instance, the *AllDifferent*$(x_1, \ldots, x_n)$ constraint requires its arguments to be pairwise different. A constraint can be represented as an object [5,4], storing attributes, such as its set of variables and its penalty, and providing methods such as the determination of the penalty change incurred if some of its variables were assigned different values. For efficiency, the attributes and results of the methods must be maintained incrementally upon each move.

Many neighbourhoods are variable-directed, in the sense that a (small) set of variables is picked before considering the neighbouring configurations where those variables take different values. One approach is to attach some level of conflict

* Corresponding author.
E-mail addresses: magnus.agren@sics.se (M. Ågren), pierre.flener@it.uu.se (P. Flener), justin.pearson@it.uu.se (J. Pearson).
[1] Work done while a Visiting Faculty Member at Sabancı University in 2006/07.

0890-5401/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2008.12.001

to variables and to pick a most conflicting variable. However, the abstraction of constraint objects also offers opportunities for *constraint-directed search* (e.g. [6,7,4]), in the sense that a (small) set of constraints is picked before considering the neighbouring configurations where those constraints have, say, a decreased penalty. Now, we show that *the knowledge of the semantics of a built-in constraint, or even just of a constraint specification, allows the exploration of constraint-directed neighbourhoods whose moves are known to achieve a penalty decrease (or preservation, or increase), without forcing the iteration over the other moves*. We claim that this simplifies the design and maintenance of local search algorithms.

The remainder of this article is organised as follows. First, we define the basic concepts of local search more precisely and present the problems on which we shall conduct our experiments (Section 2). The *contributions and importance* of this work can then be stated as follows:

> We abstract some constraint-directed neighbourhoods and show how they can be implemented via new methods for constraint objects: (i) For a built-in constraint, these methods are created using the knowledge of the semantics of the constraint. (ii) For a non-built-in constraint specified in monadic existential second-order logic, we propose a generic algorithm that works compositionally on that specification. Using existing compositional calculi for inferring the existing constraint attributes and methods from such specifications [8], an upper bound on the performance of a local search algorithm can thus be obtained for a missing constraint, before deciding whether it is worth building it in (Section 3).

Then, to show the usefulness of the approach, we present common local search heuristics using constraint-directed neighbourhoods as well as a combination of constraint-directed and variable-directed neighbourhoods. We successfully experiment with one of these heuristics, showing how it simplifies the design of the local search algorithm by not needing a data structure that is necessary when using just a variable-directed neighbourhood (Section 4). Finally we discuss implementation issues (Section 5), conclude, discuss related work, and outline future work (Section 6).

2. Preliminaries

After recalling the concept of constraint satisfaction problems, we precisely define the notions underlying local search. We also recall monadic existential second-order logic and show its convenience for specifying set constraints that are not built in. Finally, we give models based on set constraints for two common benchmark problems, on which we will conduct our experiments.

2.1. Constraint satisfaction problems

We use constraint satisfaction problems to model combinatorial problems formally:

Definition 1 (*CSP*)**.** A *constraint satisfaction problem* (*CSP*) is a three-tuple $\langle \mathbf{V}, \mathbf{D}, \mathbf{C} \rangle$ where

- $\mathbf{V}$ is a finite set of (decision) variables.
- $\mathbf{D}$ is a domain containing the possible values for the variables in $\mathbf{V}$.
- $\mathbf{C}$ is a set of constraints, each constraint in $\mathbf{C}$ being defined on a sequence of decision variables taken from $\mathbf{V}$ and specifying the allowed combinations of values for that sequence.

Let $vars(c)$ denote the set of decision variables of a constraint $c \in \mathbf{C}$.

Without loss of generality, all variables share the same domain: we can always achieve smaller domains for particular variables by additional membership constraints.

In this article, we focus on *set-CSPs*, that is CSPs where the domain $\mathbf{D}$ is the power-set $\mathcal{P}(\mathcal{U})$ of a set $\mathcal{U}$, called the *universe*. Even though we only consider set-CSPs, we make no claims about their superiority. However, the principles underlying the results of this article are *not* specific to set-CSPs: we just illustrate them on set-CSPs, since this is the main theme of our research. Whenever a definition applies to any kind of decision variables, we refrain from giving it specifically for set variables.

2.2. Local search

For each concept of local search, we give both informal (inlined) and formal (numbered) definitions, the latter being *necessary* for the inductive definitions and algorithms of the next two sections.

In local search, an initial assignment of values to *all* the variables is maintained:

Definition 2 (*Configuration and Solution*)**.** Let $P = \langle \mathbf{V}, \mathbf{D}, \mathbf{C} \rangle$ be a CSP:

- A *configuration* is a function $k : \mathbf{V} \rightarrow \mathbf{D}$.
- The *set of all configurations* for P is denoted by $\mathcal{K}_P$.

- A configuration k *is a solution to* $c \in \mathbf{C}$ (or k *satisfies* c, or c is satisfied under k) if and only if $\langle x_1, \dots, x_m \rangle$ is the variable sequence of c and $\langle k(x_1), \dots, k(x_m) \rangle$ is one of the allowed combinations of values for that sequence, as required by c.
- A configuration k *is a solution to* P if and only if k is a solution to all the constraints in $\mathbf{C}$.

For simplicity of notation, we often consider the arbitrary CSP $P = \langle \mathbf{V}, \mathbf{D}, \mathbf{C} \rangle$ to be implicit in the current context. As a result, we often write $\mathcal{K}$ instead of $\mathcal{K}_P$ and when we reason about a variable x, a set of variables X, a value v, a constraint c, or a configuration k, it is always implicit that $x \in \mathbf{V}$, $X \subseteq \mathbf{V}$, $v \in \mathcal{U}$, $c \in \mathbf{C}$, and $k \in \mathcal{K}$.

Example 1 (*Set-CSP, Configuration, and Solution*)**.** Consider the set-CSP $P = \langle \{S, T\}, \mathcal{P}(\{a, b, c\}), \{S \subset T\} \rangle$. A configuration for P is $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$. A solution to $S \subset T$ is $\{S \mapsto \{a, b\}, T \mapsto \{a, b, c\}\}$, whereas the configuration k is not a solution.

Let $\oplus$ be the *acquisition operator*. Given two functions $f : A \to B$ and $g : A' \to B$ such that $A' \subseteq A$:

- $\forall a \in A \setminus A' : (f \oplus g)(a) = f(a)$
- $\forall a \in A' : (f \oplus g)(a) = g(a)$

For example, if $k = \{S \mapsto \{a, b\}, T \mapsto \{b\}\}$ and $\ell = \{T \mapsto \{a\}\}$ then $k \oplus \ell = \{S \mapsto \{a, b\}, T \mapsto \{a\}\}$.

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

Definition 3 (*Move and Neighbourhood*)**.** Let $\langle \mathbf{V}, \mathbf{D}, \mathbf{C} \rangle$ be a CSP:

- A *move function* is a function $m : \mathcal{K} \to \mathcal{K}$. We call the configuration $m(k)$ a *move* from k, or a *neighbour* of k.
- A *neighbourhood function* is a function $n : \mathcal{K} \to \mathcal{P}(\mathcal{K})$. We call the set of configurations $n(k)$ a *neighbourhood* of k, and each element thereof a *neighbour* of k.

Note that the noun 'move' here refers to the *result* (a configuration) of applying a move function to a configuration, rather than to the *act* of changing that given configuration.

Example 2 (*Moves and Neighbourhoods for Set-CSPs*)**.** Given two set variables S, T and a configuration k, we define the following move functions for set-CSPs and we will use them throughout this article:

- $add(S, v)$ adds v to S:

$$add(S, v)(k) \stackrel{\text{def}}{=} k \oplus \{S \mapsto k(S) \cup \{v\}\}$$

- $drop(S, u)$ drops u from S:

$$drop(S, u)(k) \stackrel{\text{def}}{=} k \oplus \{S \mapsto k(S) \setminus \{u\}\}$$

- $flip(S, u, v)$ replaces u in S by v:

$$flip(S, u, v)(k) \stackrel{\text{def}}{=} k \oplus \{S \mapsto (k(S) \setminus \{u\}) \cup \{v\}\}$$

- $transfer(S, u, T)$ transfers u from S to T:

$$transfer(S, u, T)(k) \stackrel{\text{def}}{=} k \oplus \{S \mapsto k(S) \setminus \{u\}, T \mapsto k(T) \cup \{u\}\}$$

- $swap(S, u, v, T)$ swaps u of S with v of T:

$$swap(S, u, v, T)(k) \stackrel{\text{def}}{=} k \oplus \begin{Bmatrix} S \mapsto (k(S) \setminus \{u\}) \cup \{v\}, \\ T \mapsto (k(T) \setminus \{v\}) \cup \{u\} \end{Bmatrix}$$

Note that the move functions $flip(S, u, v)$, $transfer(S, u, T)$, and $swap(S, u, v, T)$ are just transactions over *add* and *drop* moves. As we will see, these move functions are necessary nevertheless since these transactions must be considered as unit operations to construct some of our constraint-directed neighbourhoods.

For each of these move functions, given a set X of set variables and a configuration k, we define the following neighbourhood functions for set-CSPs:

- $Add(X)$ returns the set of all *add* moves with respect to X:

$$Add(X)(k) \stackrel{\text{def}}{=} \{add(S, v)(k) | S \in X \wedge v \in \mathcal{U} \setminus k(S)\}$$

- *Drop*(X) returns the set of all *drop* moves with respect to X:

$$\mathit{Drop}(X)(k) \overset{\text{def}}{=} \{\mathit{drop}(S,u)(k) \mid S \in X \land u \in k(S)\}$$

- *Flip*(X) returns the set of all *flip* moves with respect to X:

$$\mathit{Flip}(X)(k) \overset{\text{def}}{=} \{\mathit{flip}(S,u,v)(k) \mid S \in X \land u \in k(S) \land v \in \mathcal{U} \setminus k(S)\}$$

- *Transfer*(X) returns the set of all *transfer* moves with respect to X:

$$\mathit{Transfer}(X)(k) \overset{\text{def}}{=} \left\{ \mathit{transfer}(S,u,T)(k) \;\middle|\; \begin{array}{l} S \neq T \in X \land \\ u \in k(S) \land u \in \mathcal{U} \setminus k(T) \end{array} \right\}$$

- *Swap*(X) returns the set of all *swap* moves with respect to X:

$$\mathit{Swap}(X)(k) \overset{\text{def}}{=} \left\{ \mathit{swap}(S,u,v,T)(k) \;\middle|\; \begin{array}{l} S \neq T \in X \;\land u \in k(S) \land \\ v \in \mathcal{U} \setminus k(S) \land v \in k(T) \land u \in \mathcal{U} \setminus k(T) \end{array} \right\}$$

For instance, consider the set variables S, T and the universe $\mathcal{U} = \{a,b\}$. Given a configuration $k = \{S \mapsto \{a\}, T \mapsto \emptyset\}$, we have:

$$\begin{array}{rcl}
\mathit{Add}(\{S,T\})(k) & = & \{\mathit{add}(S,b)(k), \mathit{add}(T,a)(k), \mathit{add}(T,b)(k)\} \\
 & = & \left\{ \begin{array}{l} \{S \mapsto \{a,b\}, T \mapsto \emptyset\}, \\ \{S \mapsto \{a\}, T \mapsto \{a\}\}, \\ \{S \mapsto \{a\}, T \mapsto \{b\}\} \end{array} \right\} \\
\mathit{Drop}(\{S,T\})(k) & = & \{\mathit{drop}(S,a)(k)\} \\
 & = & \{\{S \mapsto \emptyset, T \mapsto \emptyset\}\} \\
\mathit{Flip}(\{S,T\})(k) & = & \{\mathit{flip}(S,a,b)(k)\} \\
 & = & \{\{S \mapsto \{b\}, T \mapsto \emptyset\}\} \\
\mathit{Transfer}(\{S,T\})(k) & = & \{\mathit{transfer}(S,a,T)(k)\} \\
 & = & \{\{S \mapsto \emptyset, T \mapsto \{a\}\}\} \\
\mathit{Swap}(\{S,T\})(k) & = & \emptyset
\end{array}$$

Let $N(X)$ denote the *universal neighbourhood function*, resulting from the union of all these functions.

The penalty of a constraint set C is an estimate on how much C is violated. The penalty is used to rank the configurations of a neighbourhood. Furthermore, it is often crucial for efficiency reasons to limit the size of the neighbourhood. One way of doing this is to focus on conflicting variables. The conflict of a variable is an estimate on how much it contributes to the penalty. The variable conflict is used to rank the variables and, say, focus on the variable neighbourhood for the most conflicting variable(s). To be useful these estimates must satisfy (at least) some basic properties:

- A *penalty function penalty*(C) of $C \subseteq \mathbf{C}$ is a function with signature

  $$\mathit{penalty}(C) : \mathcal{K} \to \mathbb{N}$$

  such that $\mathit{penalty}(C)(k)$, called the *penalty* of C under k, is zero if and only if k is a solution to all constraints in C.
- A *variable-conflict function conflict*(C) of $C \subseteq \mathbf{C}$ is a function with signature

  $$\mathit{conflict}(C) : \mathbf{V} \times \mathcal{K} \to \mathbb{N}$$

  such that if $\mathit{conflict}(C)(x,k)$, called the *variable conflict* of x with respect to C under k, is zero then no configuration in the neighbourhood of k where only the value of x is changed has a smaller penalty.

The given requirements on penalty and variable-conflict functions are rather weak. The merits of actual such functions can only be discussed in relationship to the semantics of the given constraint set. Also, by abuse of notation we usually write $\mathit{penalty}(c)$ to denote the penalty function of a single constraint $c \in \mathbf{C}$, instead of the correct $\mathit{penalty}(\{c\})$. We illustrate all this in the following example where we present penalty and variable-conflict functions of the *AllDisjoint*(X) constraint.

Example 3 (*Penalty and Variable Conflict of AllDisjoint*(X))**.** The constraint *AllDisjoint*(X) is satisfied under configuration k if and only if the intersection between any two distinct set variables in X is empty.

The penalty function

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

$$
\begin{array}{lcl}
\langle \exists\text{MSO} \rangle & ::= & (\underline{\exists}\ \langle S \rangle)^{+}\ \langle FORMULA \rangle \\
\langle FORMULA \rangle & ::= & \underline{(}\langle FORMULA \rangle\underline{)} \\
 & \mid & (\underline{\forall} \mid \underline{\exists})\langle x \rangle\ \langle FORMULA \rangle \\
 & \mid & \langle FORMULA \rangle\ (\underline{\wedge} \mid \underline{\vee})\ \langle FORMULA \rangle \\
 & \mid & \langle x \rangle\ (\underline{<} \mid \underline{\leq} \mid \underline{=} \mid \underline{\neq} \mid \underline{\geq} \mid \underline{>})\ \langle y \rangle \\
 & \mid & \langle x \rangle\ (\underline{\in} \mid \underline{\notin})\ \langle S \rangle
\end{array}
$$

Fig. 1. BNF grammar for monadic existential second-order logic (∃MSO).

computes the minimum number of moves needed to nullify the penalty of the constraint, that is to transform the current configuration k into a solution. For instance, the penalty of *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 it suffices to, e.g., drop the three shared elements b, c, and d from, respectively, S, T, and V to get a solution.

The variable-conflict function

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

computes the minimum number of moves on the set variable S that nullify its conflict, under the penalty function (1). For instance, the conflict of set variable S under configuration k above is 2, and 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) under the resulting configuration.

2.3. Constraint specification in monadic existential second-order logic

When a useful constraint is not built-in to our local search framework, we let the modeller use monadic existential second-order logic (∃MSO) for specifying that constraint, and we call such a specification an ∃MSO *constraint*. In the BNF grammar of that logic in Fig. 1, the non-terminal start symbol ⟨∃MSO⟩ denotes a second-order formula and the non-terminal symbol ⟨*FORMULA*⟩ denotes a formula with first-order quantifications. Furthermore, the non-terminal symbol ⟨*S*⟩ denotes an identifier for a bound set variable S such that $S \subseteq \mathcal{U}$, where $\mathcal{U}$ is the common universe for all the set variables. The non-terminal symbols ⟨*x*⟩ and ⟨*y*⟩ denote identifiers for bound first-order variables x and y such that $x, y \in \mathcal{U}$. The terminal symbols have their standard meaning from logic and are underlined. The base cases of the BNF grammar correspond to the *primitive predicates* of ∃MSO (of which $\in$ and $\notin$ are *primitive constraints* of ∃MSO). Note that, at present, the other (built-in) constraints of our local search framework are *not* primitive constraints of ∃MSO. At no gain in expressiveness negation and implication can be added to the first-order fragment of the logic. A formula containing negations can be rewritten with all negation pushed into the primitive predicates, because the relational symbols are closed under negation, while implications can be rewritten using disjunction. We use this form of the logic to simplify the extraction of computational information from formulas.

By overloading, let $vars(\Phi)$ denote the set of decision variables of an ∃MSO formula Φ, i.e., the set of (existentially quantified) second-order (set) variables of Φ, but not any (existentially or universally quantified) first-order (scalar) variables thereof.

Example 4 (∃MSO *Specification of AllDisjoint*$(\{S,T,V\})$)**.** The constraint *AllDisjoint*$(\{S,T,V\})$ may be specified in ∃MSO by

$$\Omega \overset{\text{def}}{=} \forall x((x \notin S \vee (x \notin T \wedge x \notin V)) \wedge (x \notin T \vee x \notin V))$$

Note that Ω in the example above should be considered a constraint of a given set-CSP and, as such, the decision variables of Ω are existentially quantified by the given set-CSP and not by Ω. So S,T,V are free variables of Ω and, hence, the models of Ω denote the semantics of the specified *AllDisjoint* constraint. In the following though, to be able to reason with closed ∃MSO formulas, we will usually add such free variables as existentially quantified second-order variables and will then rather write $\exists S \exists T \exists V \Omega$. Note also that $x \notin vars(\exists S \exists T \exists V \Omega) = \{S,T,V\}$ since x is bound by the first-order universal quantifier in Ω. Furthermore, note that we have specified a special case of the *AllDisjoint* constraint, namely for $n=3$ set variables. Finally, it is also important to note that any ∃MSO specification of *AllDisjoint* over n set variables has a length (measured in number of primitive constraints) that is quadratic in n in this encoding. In consequence, there may be a price to pay for the convenience of using ∃MSO constraints. We will come back to this issue in Sections 4.3 and 5.

We introduced ∃MSO to local search in [9,8] and will use the inductively defined penalty function we proposed there.[2] For example, the penalty of a primitive predicate under a configuration k is 0 if the primitive predicate is satisfied under k, and 1

[2] In [10] ∃MSO is used for generating propagators for set constraints.

otherwise. The penalty of a conjunction (disjunction) is the sum (minimum) of the penalties of its conjuncts (disjuncts). The penalty of a first-order universal (existential) quantification is the sum (minimum) of the penalties of the quantified formula where the occurrences of the bound variable are replaced by each value in the universe. We will also use the inductively defined variable-conflict function for ∃MSO constraints we gave in [11,8]. Since variable conflicts play only a minor role in this article (namely in Algorithm 3), we need not give the intuition of that inductive definition here.

Example 5 (*Penalty and Variable Conflict of an* ∃MSO *Constraint*)**.** Recall the configuration $k = \{S \mapsto \{a, b, c\}, T \mapsto \{b, c, d\}, V \mapsto \{d, e\}\}$ of Example 3 and consider the ∃MSO specification $\exists S \exists T \exists V \Omega$ of the $AllDisjoint(\{S, T, V\})$ constraint in Example 4. Then $penalty(\exists S \exists T \exists V \Omega)(k) = 3$ and $conflict(\exists S \exists T \exists V \Omega)(S, k) = 2$, i.e., the same values as obtained by the handcrafted *penalty* $(AllDisjoint(X))$ and $conflict(AllDisjoint(X))$ functions of Example 3.

2.4. Sample set-CSPs

To finish these preliminaries, we present set-CSPs for two classical benchmark problems (in local search), on which we will conduct our experiments.

Example 6 (*Progressive Party Problem*)**.** The *progressive party problem* [12] is about timetabling a party at a yacht club, where the crews of some guest boats party at host boats over a number of periods. The crew of a guest boat must party at some host boat in each period (c_1). The spare capacity of a host boat is never to be exceeded (c_2). The crew of a guest boat may visit a particular host boat at most once (c_3). The crews of two distinct guest boats may meet at most once (c_4).

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 crews that are hosted by host boat h during period p. The following set constraints then model the problem:

(c_1) $\forall p \in P : Partition(\{S_{h,p} | h \in H\}, G)$
(c_2) $\forall h \in H : \forall p \in P : MaxWeightedSum(S_{h,p}, size, capacity(h))$
(c_3) $\forall h \in H : AllDisjoint(\{S_{h,p} | p \in P\})$
(c_4) $MaxIntersect(\{S_{h,p} | h \in H \wedge p \in P\}, 1)$

The global constraint $Partition(X, Q)$ is satisfied under configuration k if and only if the values of the set variables in X partition the constant set Q, where the value of each $S \in X$ may be the empty set. The constraint $MaxWeightedSum(S, w, m)$ is satisfied under k if and only if the weighted sum of the elements of S under the weight function w (that is $\sum_{u \in k(S)} w(u)$) does not exceed the constant m. The global constraint $MaxIntersect(X, m)$ is satisfied under k if and only if the cardinality of the intersection of any two distinct set variables in X is at most the constant m.

Example 7 (*Social Golfer Problem*)**.** In the *social golfer problem*, there is a set of golfers, each of whom plays golf once a week (c_5) and always in ng groups of ns players (c_6). The objective is to determine whether there is a schedule of nw weeks of play for these golfers, such that there is at most one week where any two distinct players are scheduled to play in the same group (c_7).

Let G be the set of $ng \cdot ns$ golfers. Let $S_{g,w}$ be a set variable denoting the golfers playing in group g in week w. The following set constraints then model the problem:

(c_5) $\forall w \in 1 \dots nw : Partition(\{S_{g,w} | g \in 1 \dots ng\}, G)$
(c_6) $\forall g \in 1 \dots ng : \forall w \in 1 \dots nw : Cardinality(S_{g,w}, ns)$
(c_7) $MaxIntersect(\{S_{g,w} | g \in 1 \dots ng \wedge w \in 1 \dots nw\}, 1)$

The constraint $Cardinality(S, n)$ is satisfied under configuration k if and only if the cardinality of S under k is the constant n.

3. Constraint-directed neighbourhoods

When constructing a neighbourhood from a variable perspective, we start from a set of variables and change some of them, while evaluating (incrementally) the effect that the changes have on the penalty. From a constraint perspective, we start from a set of constraints and obtain the neighbours directly from those constraints. For instance, configurations in such a neighbourhood may have a different penalty of those constraints. The advantage is that we can exploit combinatorial substructures of the CSP, and focus on constructing neighbourhoods with particular properties. For instance, we can extend the idea of constraint-directed search [6,7,4] to accommodate moves guaranteed to decrease, preserve, or increase the penalty.

Definition 4 (*Constraint-Directed Neighbourhoods of Constraints*)**.** Let c be a constraint, let k be a configuration, and let $penalty(c)$ be a penalty function of c. The *decreasing*, *preserving*, and *increasing neighbourhoods* of c under k and $penalty(c)$, respectively, are:

$$\begin{aligned}
\{c\}_k^{\downarrow} &= \{\ell \in N(vars(c))(k) \mid penalty(c)(k) > penalty(c)(\ell)\}\\
\{c\}_k^{=} &= \{\ell \in N(vars(c))(k) \mid penalty(c)(k) = penalty(c)(\ell)\}\\
\{c\}_k^{\uparrow} &= \{\ell \in N(vars(c))(k) \mid penalty(c)(k) < penalty(c)(\ell)\}
\end{aligned}$$

This definition gives the properties of moves of decreasing, preserving, and increasing neighbourhoods, respectively.[3] Given this target concept, we may define such neighbourhoods for particular constraints. We now show how to do this, first for any ∃MSO constraint and then for built-in constraints, just giving the example of the built-in global $AllDisjoint(X)$ constraint.

3.1. Constraint-directed neighbourhoods of ∃MSO *constraints*

We first define decreasing, preserving, and increasing neighbourhoods for any ∃MSO constraint. To do this, we must know the actual impact of a move in terms of the penalty difference.

Definition 5 (*Delta*)**.** Let c be a constraint and let k be a configuration for the variables of c. A *delta* for c under k is a pair (ℓ, δ) such that ℓ is a neighbour of k and δ is the penalty increase when moving from k to ℓ: $\delta = penalty(c)(\ell) - penalty(c)(k)$.

Now, using the set of *all* deltas for a constraint c under k, it is possible to obtain the decreasing, preserving, and increasing neighbourhoods of c under k. Towards this we need some notation. Given a configuration ℓ and a delta set D, let $D_{|1}$ denote the deltas of D projected onto their first components, that is the set of their configurations. Furthermore, let

$$\ell \rhd D \overset{\text{def}}{=} \begin{cases} \delta, & \text{if } (\ell,\delta) \in D \\ 0, & \text{otherwise} \end{cases}$$

which is to be read 'ℓ query D', denote the penalty increase recorded in D for ℓ. Considering, for example, the delta set

$$D = \{(add(S,a)(k), 0), (drop(S,b)(k), -1), (flip(S,b,a)(k), -1)\}$$

we have:

$$\begin{aligned}
&drop(S,b)(k) \rhd D = -1\\
&drop(T,b)(k) \rhd D = 0
\end{aligned}$$

Note that $\rhd$ is a total function since there is at most one delta in D for a given configuration ℓ, and since $\ell \rhd D = 0$ when there is no delta in D for ℓ.

In the inductive definition below we use $\phi[u/x]$ to denote the formula ϕ where all occurrences of variable x are replaced by the (ground) value u.

Definition 6 (*Constraint-Directed Neighbourhoods of* ∃MSO *Constraints*)**.** Let Φ be an ∃MSO constraint and let k be a configuration for $vars(\Phi)$. Let the set $\Delta(\Phi)(k)$ be defined inductively on the structure of Φ by:

$$\Delta(\exists S_1 \cdots \exists S_n \phi)(k) = \Delta(\phi)(k) \tag{a}$$

$$\Delta(\forall x \phi)(k) = \left\{ (\ell,\delta) \;\middle|\; \begin{array}{l} \ell \in \left(\bigcup_{u \in \mathcal{U}} \Delta(\phi[u/x])(k) \right)_{|1} \wedge \\ \delta = \sum_{u \in \mathcal{U}} (\ell \rhd \Delta(\phi[u/x])(k)) \end{array} \right\} \tag{b}$$

$$\Delta(\exists x \phi)(k) = \left\{ (\ell,\delta) \;\middle|\; \begin{array}{l} \ell \in \left(\bigcup_{u \in \mathcal{U}} \Delta(\phi[u/x])(k) \right)_{|1} \wedge \\ \delta = \min_{u \in \mathcal{U}} \begin{pmatrix} penalty(\phi[u/x])(k) + \\ (\ell \rhd \Delta(\phi[u/x])(k)) \end{pmatrix} \\ \qquad\qquad - penalty(\exists x \phi)(k) \end{array} \right\} \tag{c}$$

[3] Note the difference between our $\{c\}_k^{\downarrow}$ decreasing neighbourhood and the $\overset{x}{\underset{k}{\downarrow}} \mathbb{V}[c]$ notation of [13], which gives (in our terminology) the conflict of variable x with respect to constraint c under configuration k, measured as the maximum penalty decrease obtainable by only changing the value of variable x.

$$\Delta(\phi \wedge \psi)(k) = \left\{ (\ell, \delta) \;\middle|\; \begin{array}{l} \ell \in (\Delta(\phi)(k) \cup \Delta(\psi)(k))_{|1} \wedge \\ \delta = \ell \rhd \Delta(\phi)(k) + \ell \rhd \Delta(\psi)(k) \end{array} \right\} \tag{d}$$

$$\Delta(\phi \vee \psi)(k) = \left\{ (\ell, \delta) \;\middle|\; \begin{array}{l} \ell \in (\Delta(\phi)(k) \cup \Delta(\psi)(k))_{|1} \wedge \\ \delta = \min \begin{pmatrix} \mathit{penalty}(\phi)(k) + (\ell \rhd \Delta(\phi)(k)), \\ \mathit{penalty}(\psi)(k) + (\ell \rhd \Delta(\psi)(k)) \end{pmatrix} \\ \qquad\qquad\qquad - \mathit{penalty}(\phi \vee \psi)(k) \end{array} \right\} \tag{e}$$

$$\Delta(u \le v)(k) = \emptyset \; (*\text{similarly for } <, =, \neq, \geq, > *) \tag{f}$$

$$\Delta(u \in S)(k) = \; (*\text{similarly for } \notin *)$$
$$\begin{cases} \quad \{(\mathit{drop}(S,u)(k), 1)\} \\ \cup \{(\mathit{flip}(S,u,v)(k), 1) | v \in \mathcal{U} \setminus k(S)\} \\ \cup \{(\mathit{transfer}(S,u,T)(k), 1) | T \in X \wedge u \in \mathcal{U} \setminus k(T)\} \\ \cup \left\{ (\mathit{swap}(S,u,v,T)(k), 1) \;\middle|\; \begin{array}{l} v \notin k(S) \wedge T \in X \wedge \\ u \notin k(T) \wedge v \in k(T) \end{array} \right\}, & \text{if } u \in k(S) \\ \\ \quad \{(\mathit{add}(S,u)(k), -1)\} \\ \cup \{(\mathit{flip}(S,v,u)(k), -1) | v \in k(S)\} \\ \cup \{(\mathit{transfer}(T,u,S)(k), -1) | T \in X \wedge u \in k(T)\} \\ \cup \left\{ (\mathit{swap}(S,v,u,T)(k), -1) \;\middle|\; \begin{array}{l} v \in k(S) \wedge T \in X \wedge \\ u \in k(T) \wedge v \notin k(T) \end{array} \right\}, & \text{if } u \notin k(S) \end{cases} \tag{g}$$

The *decreasing*, *preserving*, *increasing*, and *delta neighbourhoods* of Φ under k and *penalty*(Φ) (as defined inductively on the structure of Φ in [9,8]) are then, respectively, defined by:[4]

$$\begin{aligned} \{\Phi\}_k^{\downarrow} &= \{\ell | (\ell, \gamma) \in \Delta(\Phi)(k) \wedge \gamma < 0\} \\ \{\Phi\}_k^{=} &= \{\ell | (\ell, \gamma) \in \Delta(\Phi)(k) \wedge \gamma = 0\} \\ \{\Phi\}_k^{\uparrow} &= \{\ell | (\ell, \gamma) \in \Delta(\Phi)(k) \wedge \gamma > 0\} \\ \{\Phi\}_k^{\delta} &= \{\ell | (\ell, \gamma) \in \Delta(\Phi)(k) \wedge \gamma = \delta\} \end{aligned}$$

Given an ∃MSO constraint Φ and a configuration k, the calculation of $\Delta(\Phi)(k)$ in the definition above needs some further explanation. Consider first the result of the base case (g) and assume that $u \in k(S)$. Any move that removes u from S will increase the penalty (of $u \in S$) by one. This includes the move that drops u from S, any move that flips u in S into another value, any move that transfers u from S to another set variable, as well as any move that swaps u of S with a value of another set variable. The case when $u \notin k(S)$ is similar although the considered moves are those that *add* u to S resulting in a penalty decrease (of $u \in S$) by one.

The result of the base case (f) is the empty set since there are no (set) decision variables of Φ in the ground test $u \le v$.

The result of the conjunctive case (d) is the union of the results of the recursive calls on the two conjuncts: the penalty increase of each delta is the sum of the penalty increases calculated for the two conjuncts. This corresponds to the penalty of a conjunction being the sum of the penalties of the two conjuncts.

The result of the disjunctive case (e) is the union of the results of the recursive calls on the two disjuncts: the penalty increase of each delta is the difference between the minimum penalty under the move of the delta with respect to each disjunct, and the penalty of the disjunction. This corresponds to the penalty of a disjunction being the minimum of the penalties of the two disjuncts.

The result of the case for first-order universal quantification (b) is a generalisation of case (d). (Recall that $\phi[u/x]$ denotes the formula ϕ where all occurrences of variable x are replaced by the (ground) value u.) Similarly, the result of the case for first-order existential quantification (c) is a generalisation of case (e).

The result of the case for second-order existential quantification (a) is just the result of the recursive call on the quantified formula.

[4] Note that we do not discuss delta neighbourhoods any further in this article except in the paragraph on future work in Section 6.

Example 8 (*Constraint-Directed Neighbourhoods of AllDisjoint*$(\{S,T,V\})$). Recall the ∃MSO specification $\exists S\exists T\exists V\Omega$ of *AllDisjoint*$(\{S,T,V\})$ in Example 4, the configuration $k=\{S\mapsto\{b\},T\mapsto\{b\},V\mapsto\emptyset\}$, and the universe $\mathcal{U}=\{a,b\}$:

$$\Delta(\exists S\exists T\exists V\Omega)(k)=\left\{\begin{array}{l}(drop(S,b)(k),-1),(drop(T,b)(k),-1),\\(add(S,a)(k),0),(add(T,a)(k),0),\\(add(V,a)(k),0),(add(V,b)(k),1),\\(flip(S,b,a)(k),-1),(flip(T,b,a)(k),-1),\\(transfer(S,b,V)(k),0),(transfer(T,b,V)(k),0)\end{array}\right\}$$

The obtained constraint-directed neighbourhoods are as follows:

$$\begin{aligned}\{\exists S\exists T\exists V\Omega\}_k^{\downarrow}&=\left\{\begin{array}{l}drop(S,b)(k),drop(T,b)(k),\\flip(S,b,a)(k),flip(T,b,a)(k)\end{array}\right\}\\\{\exists S\exists T\exists V\Omega\}_k^{=}&=\left\{\begin{array}{l}add(S,a)(k),add(T,a)(k),add(V,a)(k),\\transfer(S,b,V)(k),transfer(T,b,V)(k)\end{array}\right\}\\\{\exists S\exists T\exists V\Omega\}_k^{\uparrow}&=\{add(V,b)(k)\}\end{aligned}$$

In Example 9, we will show another definition of these constraint-directed neighbourhoods of the *AllDisjoint*(X) constraint (and this for any amount n of set variables, rather than the $n=3$ set variables of $\exists S\exists T\exists V\Omega$), handcrafted directly from the semantics of the constraint, rather than from the syntax of an ∃MSO specification thereof.

We now prove that the sets in Definition 6 are equal to the corresponding sets in Definition 4. First, all and only the possible moves are captured in the inductively computed delta set:

Lemma 1 (Correctness and Completeness of Moves). *Let Φ be an* ∃MSO *constraint and let k be a configuration for Φ. Then* $\Delta(\Phi)(k)_{|1}=N(vars(\Phi))(k)$.

Proof. ($\subseteq$) Trivial, as $N(vars(\Phi))(k)$ is the set of all possible moves for the set variables of Φ. ($\supseteq$) First note that, for a subformula ϕ of a formula Φ in ∃MSO, we have that $\ell\in(\Delta(\phi)(k))_{|1}$ implies $\ell\in(\Delta(\Phi)(k))_{|1}$, since the step cases of Definition 6 are the union of the results of some recursive calls. Assume now that $\ell\in N(vars(\Phi))(k)$ and that ℓ is of the form $add(S,v)(k)$. According to the definitions of $Add(X)$ and $N(X)$ in Example 2 it must be the case that $add(S,v)(k)\in Add(vars(\Phi))(k)\subseteq N(vars(\Phi))(k)$. Furthermore, there must be a subformula ϕ in Φ of the form $v\in S$ or $v\notin S$, since these are the only kinds of primitive constraints of ∃MSO on set variables. Since $v\notin k(S)$ by the definition of $Add(vars(\Phi))$ in Example 2, we have that $add(S,v)(k)\in(\Delta(\phi)(k))_{|1}$ by Definition 6 and hence $add(S,v)(k)\in(\Delta(\Phi)(k))_{|1}$. Similarly for *drop*, as well as for *flip*, *swap*, and *transfer*, which are just transactions over *add* and *drop* moves. □

Second, the inductive definition of $\Delta(\Phi)(k)$ in Definition 6 computes a set of deltas, as defined in Definition 5:

Lemma 2 (Correctness of Deltas). *Let Φ be an* ∃MSO *constraint and let k be a configuration for Φ. For every $\ell\in N(vars(\Phi))(k)$, we have that $\ell\rhd\Delta(\Phi)(k)=penalty(\Phi)(\ell)-penalty(\Phi)(k)$.*

Proof. The proof is by structural induction on Φ. The lemma holds for the base cases (f) and (g), and follows for case (a) by induction from the definition. The quantifier cases (b) and (c) are just generalisations of the following two cases:

Case (d): $\phi\wedge\psi$. Consider a configuration $\ell\in N(vars(\Phi))(k)$. We have that:

$$\begin{aligned}&penalty(\phi\wedge\psi)(\ell)-penalty(\phi\wedge\psi)(k)\\&=penalty(\phi)(\ell)-penalty(\phi)(k)+penalty(\psi)(\ell)-penalty(\psi)(k),\\&\qquad\text{by the inductive definition of } penalty \text{ in [9,8]}\\&=\ell\rhd\Delta(\phi)(k)+\ell\rhd\Delta(\psi)(k),\text{ by induction}\\&=\ell\rhd\Delta(\phi\wedge\psi)(k),\text{ by Definition 6.}\end{aligned}$$

Case (e): $\phi\vee\psi$. Consider a configuration $\ell\in N(vars(\Phi))(k)$. We have that:

$$\begin{aligned}&penalty(\phi\vee\psi)(\ell)-penalty(\phi\vee\psi)(k)\\&=\min(penalty(\phi)(\ell),penalty(\psi)(\ell))-penalty(\phi\vee\psi)(k),\\&\qquad\text{by the inductive definition of } penalty \text{ in [9,8]}\\&=\min\begin{pmatrix}penalty(\phi)(k)+\ell\rhd\Delta(\phi)(k),\\penalty(\psi)(k)+\ell\rhd\Delta(\psi)(k)\end{pmatrix}-penalty(\phi\vee\psi)(k),\text{ by induction}\\&=\ell\rhd\Delta(\phi\vee\psi)(k),\text{ by Definition 6.}\quad\square\end{aligned}$$

In conclusion, Definition 6 correctly captures the considered constraint-directed neighbourhoods according to Definition 4:

Proposition 1 (Soundness of Definition 6)**.** *Let* Φ *be an* $\exists$MSO *constraint, let* k *be a configuration for* Φ*, and let* $\ell \in N(vars(\Phi))(k)$*. We have that*:

$$\begin{aligned}
\ell \in \{\Phi\}_k^{\downarrow} &\Leftrightarrow penalty(\Phi)(\ell) < penalty(\Phi)(k)\\
\ell \in \{\Phi\}_k^{=} &\Leftrightarrow penalty(\Phi)(\ell) = penalty(\Phi)(k)\\
\ell \in \{\Phi\}_k^{\uparrow} &\Leftrightarrow penalty(\Phi)(\ell) > penalty(\Phi)(k)
\end{aligned}$$

Proof. Directly follows from Lemmas 1 and 2. □

3.2. Constraint-directed neighbourhoods for built-in constraints

We here just give constraint-directed neighbourhoods for one built-in constraint, namely the global *AllDisjoint*(X) constraint on set variables. Neighbourhoods for other built-in constraints are handcrafted similarly.

Example 9 (*Constraint-Directed Neighbourhoods of AllDisjoint*(X))**.** We can define the *decreasing*, *preserving*, and *increasing* neighbourhoods of *AllDisjoint*(X) under a configuration k and the penalty function (1) of Example 3 as follows:

$$\begin{aligned}
\{AllDisjoint(X)\}_k^{\downarrow} &= \{drop(S,u)(k) \mid S \in X \wedge u \in k(S) \wedge |X|_u^k > 1\}\\
&\cup \left\{flip(S,u,v)(k) \;\middle|\; \begin{array}{l} drop(S,u)(k) \in \{AllDisjoint(X)\}_k^{\downarrow} \wedge\\ add(S,v)(k) \in \{AllDisjoint(X)\}_k^{=}\end{array}\right\}\\
\{AllDisjoint(X)\}_k^{=} &= \{drop(S,u)(k) \mid S \in X \wedge u \in k(S) \wedge |X|_u^k = 1\}\\
&\cup \{add(S,v)(k) \mid S \in X \wedge |X|_v^k = 0\}\\
&\cup \left\{flip(S,u,v)(k) \;\middle|\; \begin{array}{c} drop(S,u)(k) \in \{AllDisjoint(X)\}_k^{\downarrow} \wedge\\ add(S,v)(k) \in \{AllDisjoint(X)\}_k^{\uparrow}\\ \vee\\ drop(S,u)(k) \in \{AllDisjoint(X)\}_k^{=} \wedge\\ add(S,v)(k) \in \{AllDisjoint(X)\}_k^{=}\end{array}\right\}\\
&\cup \{transfer(S,u,T)(k) \mid S \neq T \in X \wedge u \in k(S) \wedge u \notin k(T)\}\\
&\cup \left\{swap(S,u,v,T)(k) \;\middle|\; \begin{array}{c} S \neq T \in X \wedge u \in k(S) \wedge u \notin k(T) \wedge\\ v \in k(T) \wedge v \notin k(S)\end{array}\right\}\\
&\cup \{\ell \in \mathcal{K} \mid S \in X \wedge \ell(S) = k(S)\}\\
\{AllDisjoint(X)\}_k^{\uparrow} &= \{add(S,v)(k) \mid S \in X \wedge v \notin k(S) \wedge |X|_v^k > 0\}\\
&\cup \left\{flip(S,u,v)(k) \;\middle|\; \begin{array}{l} drop(S,u)(k) \in \{AllDisjoint(X)\}_k^{=} \wedge\\ add(S,v)(k) \in \{AllDisjoint(X)\}_k^{\uparrow}\end{array}\right\}
\end{aligned}$$

where $|X|_u^k$ denotes the number of set variables in X that contain element u under configuration k. Note that the preserving neighbourhood was expanded with all moves on the set variables of the CSP that are not involved in the *AllDisjoint*(X) constraint.

Even though these definitions are mutually recursive (for *flip* moves), this is just a matter of presentation, as they can be finitely unfolded (since a *flip* is just a *drop* and an *add*), and has no impact on runtime efficiency in practice.

For instance, as in Example 8, for the configuration $k = \{S \mapsto \{b\}, T \mapsto \{b\}, V \mapsto \emptyset\}$ and the universe $\mathcal{U} = \{a, b\}$, we get the following neighbourhoods:

$$\begin{aligned}
\{AllDisjoint(\{S,T,V\})\}_k^{\downarrow} &= \left\{\begin{array}{l} drop(S,b)(k), drop(T,b)(k),\\ flip(S,b,a)(k), flip(T,b,a)(k)\end{array}\right\}\\
\{AllDisjoint(\{S,T,V\})\}_k^{=} &= \left\{\begin{array}{l} add(S,a)(k), add(T,a)(k), add(V,a)(k),\\ transfer(S,b,V)(k), transfer(T,b,V)(k)\}\end{array}\right\}\\
\{AllDisjoint(\{S,T,V\})\}_k^{\uparrow} &= \{add(V,b)(k)\}
\end{aligned}$$

Note that these neighbourhoods are the same as those obtained for the $\exists$MSO-specified *AllDisjoint*$(\{S,T,V\})$ in Example 8.

Algorithm 1 Simple heuristic using constraint-directed neighbourhoods

```
 1: function CDS($\mathbf{C}$)
 2:     $k \leftarrow$ RANDOMCONFIGURATION($\mathbf{C}$)
 3:     while $penalty(\mathbf{C})(k) > 0$ do
 4:         choose $c \in \mathbf{C}$ such that $penalty(c)(k) > 0$ for
 5:             choose $\ell \in \{c\}_k^{\downarrow}$ minimising $penalty(\mathbf{C})(\ell)$ for
 6:                 $k \leftarrow \ell$
 7:             end choose
 8:         end choose
 9:     end while
10:     return $k$
11: end function
```

4. Using constraint-directed neighbourhoods

We first revisit three common heuristics using our constraint-directed neighbourhoods. All heuristics are greedy and would be extended with metaheuristics (e.g., tabu search and restarting mechanisms) in real applications. Then we show that our constraint-directed neighbourhoods even avoid certain (usually necessary) data structures. Finally, we present some experimental results.

4.1. Constraint-directed heuristics

All heuristics below use a non-deterministic **choose** operator to pick a member in a set; if that set is empty then the **choose** becomes a **skip**. We start with a simple constraint-directed heuristic and then consider some more sophisticated ones.

4.1.1. Simple heuristics

The heuristic CDS in Algorithm 1 greedily picks the best neighbour in the set of decreasing neighbours of an unsatisfied constraint. More precisely, CDS takes a set of constraints $\mathbf{C}$ and returns a solution if one is found. It starts by initialising k to a random configuration for all variables in $\mathbf{C}$ (line 2). It then iterates as long as there are any unsatisfied constraints (lines 3–9). At each iteration, it picks a violated constraint c (line 4), and updates k to any configuration in the decreasing neighbourhood of c minimising the total penalty of $\mathbf{C}$ (lines 5–7). A solution is returned if there are no unsatisfied constraints (line 10).

CDS is a variant of the heuristic `constraintDirectedSearch` in [4]. Apart from the additional tabu mechanism of the latter (omitted here for readability reasons, as such metaheuristics are orthogonal to heuristics), the only difference is line 5. In CDS, the decreasing moves are obtained directly from the chosen constraint c, meaning that *no* other moves are evaluated if the decreasing neighbourhood of c can be constructed in this way. Note that, for example, the decreasing neighbourhood of $AllDisjoint(X)$ *can* be constructed by not evaluating any other moves, which will be seen in Section 5.1 below. However, it may not be possible to construct the decreasing neighbourhood of an arbitrary constraint by not evaluating any other moves. For example, the decreasing neighbourhood of an ∃MSO constraint may need to evaluate other moves, which will be seen in Section 5.2 below. In contrast, the decreasing moves of `constraintDirectedSearch` are obtained by always evaluating *all* possible moves on the variables of c, i.e., also the moves that turn out to be preserving or increasing.

As it requires that there always exists at least one decreasing neighbour, CDS is easily trapped in local minima. We may improve it by also allowing preserving and increasing moves, if need be. This can be done by replacing lines 5–7 with the following, assuming the set union is evaluated in a lazy fashion:

```
choose $\ell \in \{c\}_k^{\downarrow} \cup \{c\}_k^{=} \cup \{c\}_k^{\uparrow}$ minimising $penalty(\mathbf{C})(\ell)$ for
    $k \leftarrow \ell$
end choose
```

This is still different from `constraintDirectedSearch`, as, say, the preserving moves on the variables of c are only evaluated if there is no decreasing move on the variables of c.

While these heuristics are simple to express also in a variable-directed approach (by, e.g., evaluating the penalty differences with respect to changing a particular set of variables according to some neighbourhood function, focusing on those giving a decreased, preserved, or increased penalty), the constraint-directed approach allows us to focus directly on the particular kind of moves that we are interested in.

4.1.2. Multi-phase heuristics

One of the advantages with the considered constraint-directed neighbourhoods is the possibilities that they open up for the simple design of multi-phase heuristics. This is a well-known method and often crucial to obtain efficient local search algorithms (see [14,15], for example). In a multi-phase heuristic, a configuration satisfying a subset $\Pi \subseteq \mathbf{C}$ of the constraints is first obtained. This configuration is then transformed into a solution satisfying all the constraints by only considering the

Algorithm 2 Multi-phase heuristic using constraint-directed neighbourhoods

```
1: function CdsPreservingFull($\Pi, \Sigma$)
2:     $k \leftarrow$ Solve($\Pi$)
3:     while $penalty(\Sigma)(k) > 0$ do
4:         choose $\ell \in \Pi_k^=$ minimising $penalty(\Sigma)(k)$ for
5:             $k \leftarrow \ell$
6:         end choose
7:     end while
8:     return $k$
9: end function
```

Algorithm 3 Multi-phase heuristic using constraint-directed neighbourhoods

```
1: function CdsPreserving($\Pi, \Sigma$)
2:     $k \leftarrow$ Solve($\Pi$)
3:     $X \leftarrow$ the set of all variables of the constraints in $\Pi$
4:     while $penalty(\Sigma)(k) > 0$ do
5:         choose $x \in X$ maximising $conflict(\Sigma)(x, k)$ for
6:             choose $\ell \in (\Pi_{|x})_k^=$ minimising $penalty(\Sigma_{|x})(k)$ for
7:                 $k \leftarrow \ell$
8:             end choose
9:         end choose
10:    end while
11:    return $k$
12: end function
```

preserving neighbourhoods of the constraints in Π. The difficulty of choosing a good subset Π varies. In order to guide the user in this task, a candidate set Π can be automatically identified in MultiTAC [16] style, as we have shown in [17]. Further, as shown in [15], it is important that the set of move functions be rich enough so that all solutions to **C** are reachable from the initial solution to Π.

In Algorithms 2 and 3, we show the two multi-phase heuristics CdsPreservingFull and CdsPreserving. Both take two sets of constraints Π and Σ, where $\Pi \cup \Sigma = \mathbf{C}$, and return a solution to **C** if one is found. In CdsPreservingFull, a configuration k for all the variables of **C**, satisfying the constraints in Π, is obtained by the call Solve(Π) (line 2). The function Solve could use a heuristic method or some other suitable solution method, possibly without search. We then iterate as long as there are any unsatisfied constraints in Σ (lines 3–7). At each iteration, we update k to be any neighbour ℓ that preserves all constraints in Π and minimises the total penalty of Σ (lines 4–6).

A problem with CdsPreservingFull is that if Π is large or has constraints involving many variables, then the size of the preserving neighbourhood on line 4 may be too large to obtain an efficient heuristic. We here present one way to overcome this problem, using variable conflicts. Recall that the conflict of a variable is an estimate on how much it contributes to the penalty. By focusing on moves involving conflicting variables or perhaps even the most conflicting variables, we may drastically shrink the size of the neighbourhood, obtaining a more efficient algorithm, while still preserving its robustness.

The heuristic CdsPreserving in Algorithm 3 differs from CdsPreservingFull in the following way: After k is initialised, X is assigned the set of all variables of the constraints in Π (line 3). Then, at each iteration, a most conflicting variable $x \in X$ is picked (line 5) before the preserving neighbourhoods of the constraints in Π are searched. When the best neighbour is chosen (lines 6–8), the constraints in Π and Σ are projected onto those containing x, thereby often drastically reducing the size of the neighbourhood; we use $\Gamma_{|x}$ to denote the constraints in constraint set Γ containing x.

Note that projecting neighbourhoods onto those containing a particular set of variables, such as conflicting variables, is a very useful *variable-directed approach* for speeding up heuristic methods. In this way, CdsPreserving is a fruitful cross-fertilisation between the variable-directed and constraint-directed approaches for generating neighbourhoods.

4.2. Avoiding data-structures

Another advantage with the considered constraint-directed neighbourhoods is that data structures for generating neighbourhoods that traditionally have to be explicitly created are not needed here. For example, the model of the progressive party problem of Example 6 is based on set variables $S_{h,p}$ denoting the set of guest boats whose crews are hosted by the crew of boat h during period p. Assume now that we want to solve this problem using CdsPreserving where Π is the set of *Partition* constraints. Having obtained a partial solution that satisfies Π in line 2, the only moves preserving Π are *transfer* moves of a guest boat from a host boat in some period to another host boat in the same period, and *swap* moves of two guest boats between host boats in the same period.[5] To generate these preserving moves from a variable-directed perspective, we would have to create data structures for obtaining the set of variables in the same period as a given variable chosen in line 5. By instead viewing this problem from a constraint-directed perspective, we obtain the preserving moves directly from the constraints in Π and no additional data structures are needed.

A similar reasoning can be done for the model of the social golfer problem of Example 7, which is based on set variables $S_{g,w}$ denoting the set of golfers in group g of week w. Assuming that Π is the set of *Partition* and *Cardinality* constraints, the only moves preserving Π are *swap* moves of two golfers between groups in the same week. Again, by looking at this from a constraint-directed perspective, the preserving moves are obtained directly from the constraints in Π and no additional data structures are needed for accessing the different weeks.

4.3. Experimental results

The first claim of this article is that algorithms exploiting the proposed constraint-directed neighbourhoods are easier to write (in our local-search framework), because at a higher level of abstraction, and this without having to pay for it by a loss of runtime efficiency. The second claim is that such a convenience can even be made available, at reasonable loss of runtime efficiency, when the framework lacks a built-in constraint that would be useful for modelling the problem at hand.

To show this, the purpose of experiments is to compare such algorithms, *within* a given local search framework, with algorithms not using such neighbourhoods, for both built-in and ∃MSO constraints. The purpose here need thus *not* be to compare algorithms with constraint-directed neighbourhoods in our local search framework with algorithms in other local search frameworks, whether they have such neighbourhoods or not. Nor is the purpose a comparison of our problem models (under our framework) with other models (under other frameworks), as our objective is *not* (yet) to beat runtime records (as that requires a very careful implementation).

We implemented a prototype of the ideas presented in this article for all the constraints used in the given models of the progressive party and social golfer problems, as well as for any ∃MSO constraint, using the implementation ideas discussed in Section 5 below. Classical instances for both problems were then run, mimicking the algorithm we used in [18] but using a variant of CdsPreserving. This meant that the preserved constraint sets Π were chosen as indicated in the previous sub-section and that we extended CdsPreserving with the same metaheuristics, maximum number of iterations, and so on, as in [18]. This also meant that the preserving neighbourhood for the progressive party problem had to be restricted to *transfer* moves, because *swap* moves were not considered in [18].

We show the experimental comparison with the algorithm of [18] in Tables 1 and 2. Each entry is the mean runtime in CPU seconds of the successful runs out of 100 for a particular instance, and the numbers in parentheses are the numbers of unsuccessful runs, if any, for that instance. All experiments were run on an Intel 2.4 GHz Linux machine with 512 MB of RAM.

When using built-in constraints, the runtimes in Tables 1(a) and (b) and 2(a) and (b) are quite similar between the designed variant of CdsPreserving and the algorithm in [18], hence (considering that this is just a prototype) there seem to be no runtime overhead problems with our proposed constraint-directed neighbourhoods. However, the programming time was much reduced for CdsPreserving, because reasoning at a higher level of abstraction and thus not needing to initialise and maintain some data structures (as discussed in the previous sub-section). Note that different random seeds were used in CdsPreserving and the algorithm in [18], which explains the differences in the numbers of unsuccessful runs in the two tables.

When pretending that *Partition* is not built in and using an ∃MSO-specified *Partition* instead, the runtimes in Tables 1(a) and (c) are (only) three to four times apart for all the instances. This is not a surprise since the chosen ∃MSO specification of *Partition* is of quadratic length in its number of set variables, leading to an at worst quadratic slowdown for the ∃MSO-based computations compared to the built-in *Partition*. However, on these instances, the slowdown is observed to be linear. Furthermore, compared to using the built-in *Partition*, it must be noted that efforts such as designing penalty and variable-conflict functions with incremental maintenance algorithms, as well as implementing *member* and *iterate* methods were *not* necessary, since all this is obtained automatically given the ∃MSO constraint, as shown in [9,11,8] and this article, respectively. In general, testing the chosen combination of heuristics and meta-heuristics using ∃MSO constraints can help to decide if it is worth producing a faster handcrafted implementation. Again, different random seeds were used, which explains why the numbers of unsuccessful runs differ.

5. Implementation issues

After discussing implementation issues for built-in constraints, we do the same for ∃MSO constraints. In both cases, we give the runtime complexity of the proposed algorithms.

[5] The reason why *flip* moves of a guest boat for a host boat in a period are impossible, even though *flip* moves are in the neighbourhood $\{Partition(X, Q)\}^{=}_{k}$, is that $Q = G = \mathcal{U}$ here and that k satisfies the considered constraint. Whenever this is the case, there are no *flip* moves in $\{Partition(X, Q)\}^{=}_{k}$ because there are no values outside Q that could be flipped for.

Table 1 Runtimes in CPU seconds for classical instances [12] of the progressive party problem. Mean runtime of successful runs (out of 100) and number of unsuccessful runs (if any) in parentheses.

Host boats H	Number of periods				
	6	7	8	9	10
(a) CdsPreserving *with built-in* $Partition(X, Q)$					
$\{1-12, 16\}$			0.7	1.8	19.1
$\{1-13\}$			8.8	105.2	
$\{1, 3-13, 19\}$			10.2	143.9 (1)	
$\{3-13, 25, 26\}$			21.0	220.5 (14)	
$\{1-11, 19, 21\}$	11.8	96.0 (1)			
$\{1-9, 16-19\}$	17.7	184.7 (11)			
(b) Algorithm of [18] with built-in $Partition(X, Q)$					
$\{1-12, 16\}$			1.2	2.3	21.0
$\{1-13\}$			7.0	90.5	
$\{1, 3-13, 19\}$			7.2	128.4 (4)	
$\{3-13, 25, 26\}$			13.9	170.0 (17)	
$\{1-11, 19, 21\}$	10.3	83.0 (1)			
$\{1-9, 16-19\}$	18.2	160.6 (22)			
(c) CdsPreserving *with* ∃MSO*-specified* $Partition(X, Q)$					
$\{1-12, 16\}$			2.4	6.2	72.6
$\{1-13\}$			31.2	411.8	
$\{1, 3-13, 19\}$			37.9	582.4 (3)	
$\{3-13, 25, 26\}$			81.0	903.4 (12)	
$\{1-11, 19, 21\}$	43.6	367.2			
$\{1-9, 16-19\}$	66.5	750.8 (8)			

5.1. Implementation issues for built-in constraints

For built-in constraints, the decreasing, preserving, and increasing neighbourhoods may be represented *procedurally*, with the support of underlying data structures, by two proposed new methods for constraint objects, called *member* and *iterate*. In Algorithm 4, we only show these methods for $\{AllDisjoint(X)\}_k^{\downarrow}$.

The $member(\{AllDisjoint(X)\}_k^{\downarrow})(\ell, k)$ method takes two configurations ℓ and k and returns **true** if and only if $\ell \in \{AllDisjoint(X)\}_k^{\downarrow}$. As observable from the definition of $\{AllDisjoint(X)\}_k^{\downarrow}$ in Example 9, this is the case only when ℓ is of the form $drop(S, u)(k)$ and u occurs more than once in X, or $flip(S, u, v)(k)$ and u (respectively, v) occurs more than once (respectively, not at all) in X (lines 3 and 4). A call $member(\{AllDisjoint(X)\}_k^{\downarrow})(\ell, k)$ can be performed in constant time, assuming that $|X|_u^k$ and $|X|_v^k$ are maintained incrementally.

The $iterate(\{AllDisjoint(X)\}_k^{\downarrow})(S, k, \sigma)$ method takes a set variable S, a configuration k, as well as a function σ and applies σ to each configuration $\ell \in \{AllDisjoint(X)\}_k^{\downarrow}$ involving S. This is the case for each configuration ℓ of the form $drop(S, u)(k)$ or $flip(S, u, v)(k)$ such that $member(\{AllDisjoint(X)\}_k^{\downarrow})(\ell, k)$ holds (lines 10–13).[6] The argument function σ must take a configuration and work by side effects. For example, a call $\sigma(\ell)$ could evaluate the penalty increase between the current configuration and ℓ, and update some internal data structure keeping track of the best such move. A call $iterate(\{AllDisjoint(X)\}_k^{\downarrow})(S, k, \sigma)$ can be performed in $\mathcal{O}(|\{AllDisjoint(X)\}_k^{\downarrow}|)$ time, assuming that the set comprehensions on lines 9 and 11 are maintained incrementally, and that a call to σ takes constant time.

The following example shows how to use these methods in practice.

Example 10. Consider again the heuristic CdsPreserving in Algorithm 3 and assume that the set Π of preserved constraints in that heuristic contains exactly two constraints π_1 and π_2. Given the *member* and *iterate* methods for those constraints, we could implement the **choose** block on lines 6–8 as follows:

```
$\ell s \leftarrow [\,]$
$minPenalty \leftarrow maxInt$
$iterate(\{\pi_1\}_k^{=})(x, k, updateBest(k, \ell s, minPenalty, \pi_2))$
$k \leftarrow$ random element in $\ell s$
```

[6] Note that an explicit call to *member* is not desirable since this would require iterating over *all* moves.

Table 2 Runtimes in CPU seconds for classical instances of the social golfer problem. Mean run time of successful runs (out of 100) and number of unsuccessful runs (if any) in parentheses.

ng-ns-nw	Time	(Fails)	*ng-ns-nw*	Time	(Fails)
(a) CdsPreserving *with built-in constraints*					
6-3-7	0.2		6-3-8	253.4	(79)
7-3-9	127.4	(1)	8-3-10	6.0	
9-3-11	1.1		10-3-13	331.4	(3)
6-4-5	0.1		7-4-7	446.4	(57)
8-4-7	0.3		9-4-8	0.5	
10-4-9	0.7		7-5-5	0.6	
8-5-6	3.8		9-5-6	0.3	
10-5-7	0.6		6-6-3	0.1	
7-6-4	0.6		8-6-5	9.5	
9-6-5	0.4		10-6-6	1.1	
7-7-3	0.1		8-7-4	2.7	
9-7-4	0.3		10-7-5	1.1	
8-8-3	0.2		9-8-3	0.2	
10-8-4	0.6		9-9-3	0.3	
10-9-3	0.3		10-10-3	0.5	
(b) Algorithm of [18] with built-in constraints					
6-3-7	0.4		6-3-8	215.0	(76)
7-3-9	138.0	(5)	8-3-10	14.4	
9-3-11	3.5		10-3-13	325.0	(35)
6-4-5	0.3		7-4-7	333.0	(76)
8-4-7	0.9		9-4-8	1.7	
10-4-9	2.5		7-5-5	1.3	
8-5-6	8.6		9-5-6	0.9	
10-5-7	1.7		6-6-3	0.2	
7-6-4	1.2		8-6-5	18.6	
9-6-5	1.0		10-6-6	3.7	
7-7-3	0.3		8-7-4	4.9	
9-7-4	0.8		10-7-5	3.4	
8-8-3	0.5		9-8-3	0.6	
10-8-4	1.4		9-9-3	0.7	
10-9-3	0.8		10-10-3	1.1	

Hence, the preserving neighbourhood of π_1 is iterated over, applying *updateBest* to each move in that neighbourhood. When this iteration finishes, the buffer ℓs contains the best moves of the neighbourhood, and k is set to a random element of this buffer. The procedure *updateBest* works by side effects as follows:

```
**procedure** $updateBest(k, \ell s, minPenalty, \pi_2)(m)$
    **if** $member(\{\pi_2\}_k^{=})(m, k)$ **then**
        **if** $penalty(\Sigma_{|x})(m) < minPenalty$ **then**
            $minPenalty \leftarrow penalty(\Sigma_{|x})(m)$
            $\ell s \leftarrow [m]$
        **else if** $penalty(\Sigma_{|x})(m) = minPenalty$ **then**
            $\ell s \leftarrow m :: \ell s$
        **end if**
    **end if**
**end procedure**
```

Hence, if the argument move m is also in the preserving neighbourhood of π_2, then it may be added to the buffer ℓs of best moves. This buffer is reset whenever a better move is found. Note that *updateBest* is similar to the neighbour abstraction and neighbour selector constructions of [4, p. 165].

5.2. Implementation issues for ∃MSO *constraints*

For ∃MSO constraints, the decreasing, preserving, and increasing neighbourhoods may be represented partly extensionally, namely for the *add* and *drop* moves, and partly procedurally, since the *flip*, *transfer*, and *swap* moves can be generated from the former, and since representing the latter extensionally would be too costly in terms of both space and time.

Algorithm 4 The *member* and *iterate* methods for $AllDisjoint(X)$

```
1: function member({AllDisjoint(X)}_k^↓)(ℓ, k) : boolean
2:     case ℓ of
3:         drop(S, u)(k) : return $|X|_u^k > 1$
4:       | flip(S, u, v)(k) : return $|X|_u^k > 1 \wedge |X|_v^k = 0$
5:       | any_other : return false
6:     end case
7: end function
8: procedure iterate({AllDisjoint(X)}_k^↓)(S, k, σ)
9:     for all $u \in \{x \in k(S) \mid |X|_x^k > 1\}$ do
10:        σ(drop(S, u)(k))
11:        for all $v \in \{x \in \mathcal{U} \setminus k(S) \mid |X|_x^k = 0\}$ do
12:            σ(flip(S, u, v)(k))
13:        end for
14:    end for
15: end procedure
```

Given an ∃MSO constraint Φ and a configuration k, the subset $\Delta_{|\{add,drop\}}(\Phi)(k)$ of the delta set $\Delta(\Phi)(k)$ with only elements of the form $(add(S,v)(k),\delta)$ or $(drop(S,u)(k),\delta)$ may be represented *extensionally* at every node in the *extended constraint dag* (directed acyclic graph) of Φ, and updated incrementally between moves, similarly to incrementally updating penalties [8]. A *constraint dag* has as nodes the quantifications, connectives, and primitive predicates of the ∃MSO constraint, with the arcs pointing from subformulas to formulas. It originally only contained node annotations about the penalty and variable conflicts under a configuration [9,11].

Example 11 (*Extended constraint dag of* $\exists S\exists T\exists V\Omega$)**.** Recall the ∃MSO specification $\exists S\exists T\exists V\Omega$ of $AllDisjoint(\{S,T,V\})$, the configuration $k=\{S\mapsto\{b\},T\mapsto\{b\},V\mapsto\emptyset\}$, and the delta set $\Delta(\exists S\exists T\exists V\Omega)(k)$ of Example 8. The extended constraint dag of $\exists S\exists T\exists V\Omega$ under k, shown in Fig. 2, contains penalty information (shaded sets, and not further explained here: see [9,8]) as well as the sets $\Delta_{|\{add,drop\}}(\phi)(k)\subseteq\Delta(\phi)(k)$, for each subformula ϕ of $\exists S\exists T\exists V\Omega$.

In Algorithm 6, we present (public) generic *member* and *iterate* methods only for the decreasing neighbourhood of ∃MSO constraints. Both methods call the private $collect(\Phi)$ method of Algorithm 5, which takes a set variable S, a configuration k, and a move set M as arguments such that:

- Each move in M affects S.
- M contains only *flip*, *transfer*, and *swap* moves (since *add* and *drop* moves are already extensional in the dag of Φ).

A call $collect(\Phi)(S,k,M)$ returns the delta set for Φ under k, where the configuration ℓ of any element (ℓ,δ) of this delta set is a member of M. This function is only partly described in Algorithm 5; all other cases follow similarly from Definition 6, and the sets of *flip* and *swap* moves are computed similarly. For $\exists S_1\cdots\exists S_n(\phi)$, the function is called recursively for ϕ (line 3). For $\forall x(\phi)$, it is called recursively for ϕ, and the value of δ, given a *transfer* move, is obtained from the result of that call (line 5). For $\phi\wedge\psi$: (i) if S is in both conjuncts, then the value of δ, given a move of the form $transfer(S,u,T)(k)$, is recursively determined as the sum of $transfer(S,u,T)(k)\triangleright collect(\phi)(S,k,M)$ and $transfer(S,u,T)(k)\triangleright collect(\psi)(S,k,M)$ (line 8); (ii) if S is only in one of the conjuncts, say ϕ, then the value of δ, given a move of the form $transfer(S,u,T)(k)$, is recursively determined as the sum of $transfer(S,u,T)(k)\triangleright collect(\phi)(S,k,M)$ and $add(T,u)(k)\triangleright\Delta_{|\{add,drop\}}(\psi)(k)$ (line 10). *The benefit of representing* $\Delta_{|\{add,drop\}}(\Phi)(k)$ *extensionally can be seen in case (ii), where a recursive call is needed only for the subformula where S appears.* For $x\in S$, given a $transfer(S,u,T)(k)$ move, the value of δ is 1, since u is removed from S (line 15).

Example 12 (*The collect Function*)**.** Consider again $\exists S\exists T\exists V\Omega$ and configuration $k=\{S\mapsto\{b\},T\mapsto\{b\},V\mapsto\emptyset\}$ of Example 8. By stepping through the call $collect(\exists S\exists T\exists V\Omega)(V,k,\{transfer(S,b,V)(k)\})$ while keeping the dag in Fig. 2 in mind, we see that

$$collect(\exists S\exists T\exists V\Omega)(V,k,\{transfer(S,b,V)(k)\})=\{(transfer(S,b,V)(k),0)\}$$

Hence, similarly to the end of Example 8, we have that $transfer(S,b,V)(k)$ is in the preserving neighbourhood of Φ under k.

By a similar reasoning as in [8, Section 5.3], we can argue that the time complexity of $collect(\Phi)$ is at worst proportional to the length of Φ. The ∃MSO specification we have used for $AllDisjoint(\{S,T,V\})$ is of a length (measured in number of primitive constraints) that is quadratic in the number of variables. In general, an ∃MSO specification may have some overhead in terms of the formula length, which is the price to pay for the convenience of using ∃MSO. As seen in Section 4.3, experiments show that a worst-case quadratic overhead can in practice be linear.

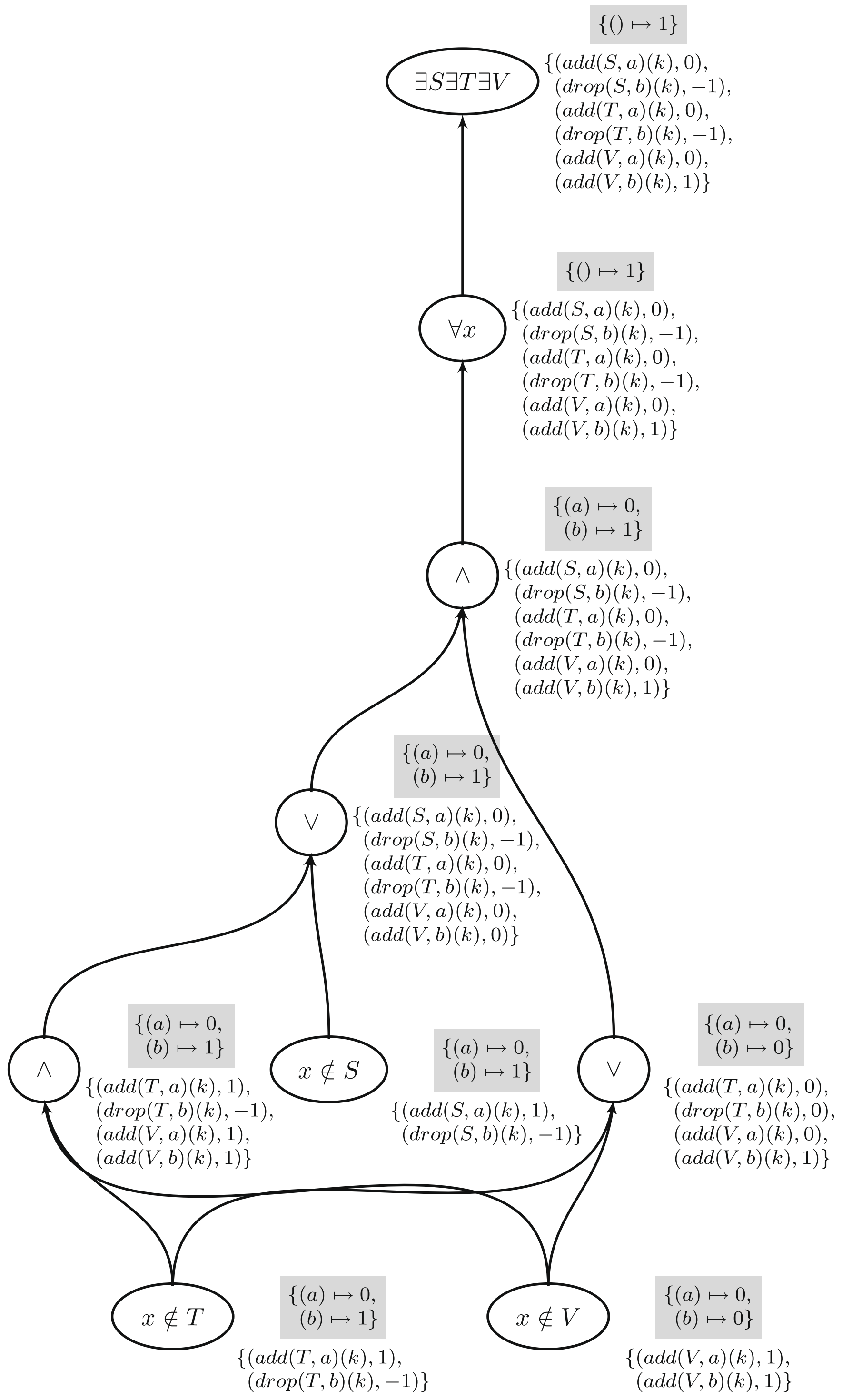
$\{() \mapsto 1\}$
$\exists S \exists T \exists V$
$\{(add(S,a)(k),0), (drop(S,b)(k),-1), (add(T,a)(k),0), (drop(T,b)(k),-1), (add(V,a)(k),0), (add(V,b)(k),1)\}$
$\{() \mapsto 1\}$
$\forall x$
$\{(add(S,a)(k),0), (drop(S,b)(k),-1), (add(T,a)(k),0), (drop(T,b)(k),-1), (add(V,a)(k),0), (add(V,b)(k),1)\}$
$\{(a) \mapsto 0, (b) \mapsto 1\}$
$\wedge$
$\{(add(S,a)(k),0), (drop(S,b)(k),-1), (add(T,a)(k),0), (drop(T,b)(k),-1), (add(V,a)(k),0), (add(V,b)(k),1)\}$
$\{(a) \mapsto 0, (b) \mapsto 1\}$
$\vee$
$\{(add(S,a)(k),0), (drop(S,b)(k),-1), (add(T,a)(k),0), (drop(T,b)(k),-1), (add(V,a)(k),0), (add(V,b)(k),0)\}$
$\{(a) \mapsto 0, (b) \mapsto 1\}$
$\wedge$
$\{(add(T,a)(k),1), (drop(T,b)(k),-1), (add(V,a)(k),1), (add(V,b)(k),1)\}$
$x \notin S$
$\{(a) \mapsto 0, (b) \mapsto 1\}$
$\{(add(S,a)(k),1), (drop(S,b)(k),-1)\}$
$\{(a) \mapsto 0, (b) \mapsto 0\}$
$\vee$
$\{(add(T,a)(k),0), (drop(T,b)(k),0), (add(V,a)(k),0), (add(V,b)(k),1)\}$
$x \notin T$
$\{(a) \mapsto 0, (b) \mapsto 1\}$
$\{(add(T,a)(k),1), (drop(T,b)(k),-1)\}$
$x \notin V$
$\{(a) \mapsto 0, (b) \mapsto 0\}$
$\{(add(V,a)(k),1), (add(V,b)(k),1)\}$

Fig. 2. Extended constraint dag of $\exists S \exists T \exists V \Omega$ under the configuration k of Example 8. The dag contains penalty information (shaded sets) as well as delta sets with *add* and *drop* moves.

Algorithm 5 Private *collect* method for ∃MSO constraints

```
 1: function $collect(\Phi)(S,k,M) : \mathcal{K} \times \mathbb{Z}$
 2:     case $\Phi$ of
 3:         $\exists S_1 \cdots \exists S_n(\phi)$ : return $collect(\phi)(S,k,M)$
 4:       | $\forall x(\phi)$ :
            return $\{(flip(S,u,v)(k),\delta) \mid \cdots$ (* condition omitted *) $\cdots\}$
 5:             $\cup \left\{\begin{array}{l}(transfer(S,u,T)(k),\delta) \mid transfer(S,u,T)(k) \in M \wedge \\ \delta = transfer(S,u,T)(k) \rhd collect(\phi)(S,k,M)\end{array}\right\}$
                $\cup \{(swap(S,u,v,T)(k),\delta) \mid \cdots$ (* condition omitted *) $\cdots\}$
 6:       | $\phi \wedge \psi$ :
 7:         if $S \in vars(\phi) \cap vars(\psi)$ then
                return $\{(flip(S,u,v)(k),\delta) \mid \cdots$ (* condition omitted *) $\cdots\}$
 8:                 $\cup \left\{\begin{array}{l}(transfer(S,u,T)(k),\delta) \mid transfer(S,u,T)(k) \in M \wedge \\ \delta = transfer(S,u,T)(k) \rhd collect(\phi)(S,k,M) + \\ transfer(S,u,T)(k) \rhd collect(\psi)(S,k,M)\end{array}\right\}$
                    $\cup \{(swap(S,u,v,T)(k),\delta) \mid \cdots$ (* condition omitted *) $\cdots\}$
 9:         else if $S \in vars(\phi)$ then
                return $\{(flip(S,u,v)(k),\delta) \mid \cdots$ (* condition omitted *) $\cdots\}$
10:                 $\cup \left\{\begin{array}{l}(transfer(S,u,T)(k),\delta) \mid transfer(S,u,T)(k) \in M \wedge \\ \delta = transfer(S,u,T)(k) \rhd collect(\phi)(S,k,M) + \\ add(T,u)(k) \rhd \Delta_{|\{add,drop\}}(\psi)(k)\end{array}\right\}$
                    $\cup \{(swap(S,u,v,T)(k),\delta) \mid \cdots$ (* condition omitted *) $\cdots\}$
11:         else (* symmetric to the case when $S \in vars(\phi)$ *)
12:         end if
13:         $\cdots$ (* omitted cases *) $\cdots$
14:       | $x \in S$ :
            return $\{(flip(S,u,v)(k),\delta) \mid \cdots$ (* condition omitted *) $\cdots\}$
15:             $\cup \{(transfer(S,u,T)(k),1) \mid transfer(S,u,T)(k) \in M\}$
                $\cup \{(swap(S,u,v,T)(k),\delta) \mid \cdots$ (* condition omitted *) $\cdots\}$
16:     end case
17: end function
```

The generic $member(\{\Phi\}_k^{\downarrow})(\ell,k)$ method takes two configurations ℓ and k and returns **true** if and only if $\ell \in \{\Phi\}_k^{\downarrow}$. If ℓ is an *add* or *drop* move, then the result is obtained directly from $\Delta_{|\{add,drop\}}(\Phi)(k)$ (lines 3 and 4). Otherwise, the result is obtained from a call $collect(\Phi)(S,k,\{\ell\})$, where S is the variable affected by the move from k to ℓ (lines 5–7). Since $\Delta_{|\{add,drop\}}(\phi)(k)$ is represented extensionally for each subformula, we access it in constant time.

The generic $iterate(\{\Phi\}_k^{\downarrow})(S,k,\sigma)$ method takes a set variable S, a configuration k, as well as a function σ, and applies σ to each move in $\{\Phi\}_k^{\downarrow}$ involving S. This set is obtained from a union of the extensionally represented $\Delta_{|\{add,drop\}}(\Phi)(k)$ and the result of a call $collect(\Phi)(S,k,M)$, where M is the set of all moves involving S. We use $M_{|S}$ to denote the deltas in M involving S.

Given an ∃MSO constraint Φ, the time complexities of *member* and *iterate* are both at worst proportional to the length of Φ, since both call *collect*.

6. Conclusion

In summary, we have first revisited the exploration of constraint-directed neighbourhoods, where a (small) set of constraints is picked before considering the neighbouring configurations where those constraints have a decreased (or preserved, or increased) penalty. Given the semantics of a built-in constraint, or just a formal specification of a new constraint, neighbourhoods consisting only of configurations with decreased, preserved, or increased penalty can be represented via new methods for constraint objects. We have then presented a prototype implementation of the corresponding methods in our local search framework and, using these new methods, have shown how some local search algorithms are simplified, compared to using just a variable-directed neighbourhood.

In terms of related work, the constraint objects of [5,4] have the methods $getAssignDelta(x,v)$ and $getSwapDelta(x_1,x_2)$ in their interface, returning the penalty increases upon the (scalar) moves $x := v$ and $x_1 :=: x_2$, respectively. Although it is possible to construct decreasing, preserving, increasing neighbourhoods using these methods, the signs of their penalty increases are not known in advance. So if one wants to construct, say, a decreasing neighbourhood (as done in the procedure `constraintDirectedSearch` in [4, p. 68], for example), then one may have to iterate over many moves that turn out to be non-decreasing. This contrasts using the methods for representing constraint-directed neighbourhoods proposed in this article, where it is known in advance that exploring the decreasing neighbourhood, say, will only yield moves with a lower

Algorithm 6 Generic *member* and *iterate* methods for ∃MSO constraints

```
1: function $member(\{\Phi\}^{\downarrow}_{k})(\ell, k)$ : boolean
2:     case $\ell$ of
3:         $add(S, v)(k)$ : return $\ell \rhd \Delta_{|\{add,drop\}}(\Phi)(k) < 0$
4:       | $drop(S, u)(k)$ : return $\ell \rhd \Delta_{|\{add,drop\}}(\Phi)(k) < 0$
5:       | $flip(S, u, v)(k)$ : return $\ell \rhd collect(\Phi)(S, k, \{\ell\}) < 0$
6:       | $transfer(S, u, T)(k)$ : return $\ell \rhd collect(\Phi)(S, k, \{\ell\}) < 0$
7:       | $swap(S, u, v, T)(k)$ : return $\ell \rhd collect(\Phi)(S, k, \{\ell\}) < 0$
8:     end case
9: end function
10: procedure $iterate(\{\Phi\}^{\downarrow}_{k})(S, k, \sigma)$
11:     $D \leftarrow \Delta_{|\{add,drop\}}(\Phi)(k)_{|S} \cup collect(\Phi)(S, k, \{\ell \mid \ell \in N(vars(\Phi))(k)_{|S}\})$
12:     for all $(\ell, \delta) \in D$ do
13:         if $\delta < 0$ then $\sigma(\ell)$ end if
14:     end for
15: end procedure
```

penalty. Of course, using the invariants of Comet, it is possible to extend its constraint interface with methods similar to those proposed in this article, thus achieving similar results in the (scalar) Comet framework. Conducting payoff experiments (like the ones of Section 4.3) *within* the Comet framework is considered future work, while comparisons *between* the frameworks are beyond the purpose of this article.

In [19], it is shown that the semantics of the constraints can be used to derive suitable neighbourhoods for some models, but that work is orthogonal to ours, which is concerned with a general framework for the implementation and analysis of constraint-directed neighbourhoods.

In [20], it is also suggested that global constraints can be used in local search to generate heuristics to guide search; however, that work differs in that the provided heuristics are defined in an ad-hoc manner for each constraint.

In this article we have started to explore new directions in automatic neighbourhood generation for local search, and there are still many directions for future work.

First, considering that *flip*, *transfer*, and *swap* moves essentially are transactions over *add* and *drop* moves, it should be possible to assist the designer of a constraint object by inferring the constraint-directed neighbourhoods for the former compound moves from the latter atomic moves.

Also, in this article, we just precompute the *sign* of the penalty change for built-in constraints in our constraint-directed neighbourhoods, but it should be possible to precompute the actual *value* of that change, as we have already done for the primitive predicates of ∃MSO in Definition 6. Then, upon adding the built-in constraints as further base cases both to the BNF grammar of ∃MSO in Fig. 1 and to the inductive definition of $\Delta(\Phi)(k)$ in Definition 6, the step cases of Definition 6 enable the precomputation of the penalty change of an arbitrary ∃MSO formula over constraints. For instance, noting that $Partition(X, Q) \overset{\text{def}}{=} AllDisjoint(X) \wedge Union(X, Q)$, we could then precompute the constraint-directed neighbourhoods of *Partition* from those of *AllDisjoint* (in Example 9) and *Union* (not listed here). Also, the preserving neighbourhood $\Pi^{=}_{k}$ in line 4 of Algorithm 2 then does not need to be calculated dynamically as $\bigcap_{c \in \Pi} \{c\}^{=}_{k}$ but could be statically precomputed.

Further, in line 4 of Algorithm 2, instead of choosing a neighbour in the preserving neighbourhood $\Pi^{=}_{k}$ minimising $penalty(\Sigma)(k)$, one might choose a neighbour in $\Pi^{=}_{k} \cap \Sigma^{\downarrow}_{k}$, by representing the intersection of the moves preserving the penalty of Π and the moves decreasing the penalty of Σ, if that intersection is non-empty, thereby saving at each iteration the consideration of the non-decreasing moves on Σ.

Finally, the neighbourhoods of Definition 4 should be parametrised by the neighbourhood function to be used, rather than hardwiring the universal neighbourhood function $N(X)$, and the programmer should be supported in the choice of this parameter.

Acknowledgments

The second author thanks Sabancı University, for the research time included in his Visiting Faculty Member position. Part of this work was made when he was visited there by the first author, on a grant by Ericsson's Research Foundation, or by both other authors, on Erasmus Teacher Exchange stipends: these sponsors are also gratefully acknowledged. Many thanks also to the anonymous referees for their useful comments on this article.

This article contains most, but not all the material of [17]. A preliminary version of this article appears in the informally published *Proceedings of LSCS'07, the 4th International Workshop on Local Search Techniques in Constraint Satisfaction*, held at CP'07.

References

[1] E. Aarts, J.K. Lenstra (Eds.), Local Search in Combinatorial Optimization, John Wiley & Sons, 1997.
[2] F. Glover, M. Laguna, Tabu search, in: Modern Heuristic Techniques for Combinatorial Problems, John Wiley & Sons, 1993, pp. 70–150.
[3] S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[4] P. Van Hentenryck, L. Michel, Constraint-Based Local Search, The MIT Press, 2005.
[5] L. Michel, P. Van Hentenryck, A constraint-based architecture for local search, ACM SIGPLAN Notices 37 (11) (2002) 101–110., (Proceedings of OOPSLA'02).
[6] M.S. Fox, Constraint-directed search: a case study of job-shop scheduling, Ph.D. thesis, Computer Science Department, Carnegie Mellon University, USA, December 1983.
[7] J.P. Walser, Integer Optimization by Local Search: A Domain-Independent Approach, LNCS, vol. 1637, Springer-Verlag, 1999.
[8] M. Ågren, P. Flener, J. Pearson, Generic incremental algorithms for local search, Constraints 12 (3) (2007) 293–324. (Collects the results of papers at *CP-AI-OR'05*, *CP'05*, and *CP'06*, published in LNCS 3524, 3709, and 4204).
[9] M. Ågren, P. Flener, J. Pearson, Incremental algorithms for local search from existential second-order logic, in: P. van Beek (Ed.), Proceedings of CP'05, LNCS, vol. 3709, Springer-Verlag, 2005, pp. 47–61.
[10] G. Tack, C. Schulte, G. Smolka, Generating propagators for finite set constraints, in: F. Benhamou (Ed.), Proceedings of CP'06, LNCS, vol. 4204, Springer-Verlag, 2006, pp. 575–589.
[11] M. Ågren, P. Flener, J. Pearson, Inferring variable conflicts for local search, in: F. Benhamou (Ed.), Proceedings of CP'06, LNCS, vol. 4204, Springer-Verlag, 2006, pp. 665–669.
[12] B.M. Smith, S.C. Brailsford, P.M. Hubbard, H.P. Williams, The progressive party problem: Integer linear programming and constraint programming compared, Constraints 1 (1996) 119–138.
[13] P. Van Hentenryck, L. Michel, Differentiable invariants, in: F. Benhamou (Ed.), Proceedings of CP'06, LNCS, vol. 4204, Springer-Verlag, 2006, pp. 604–619.
[14] I. Dotú, P. Van Hentenryck, Scheduling social golfers locally, in: R. Barták, M. Milano (Eds.), Proceedings of CP-AI-OR'05, LNCS, vol. 3524, Springer-Verlag, 2005.
[15] H. Fang, Y. Kilani, J. Lee, P. Stuckey, The island confinement method for reducing search space in local search methods, Journal of Heuristics 13 (6) (2007) 557–585.
[16] S. Minton, Automatically configuring constraint satisfaction programs: a case study, Constraints 1 (1–2) (1996) 7–43.
[17] M. Ågren, P. Flener, J. Pearson, On constraint-oriented neighbours for local search, Tech. Rep. 2007-009, Department of Information Technology, Uppsala University, Sweden, March 2007. Available from: <http://www.it.uu.se/research/reports/2007-009>.
[18] M. Ågren, P. Flener, J. Pearson, Set variables and local search, in: R. Barták, M. Milano (Eds.), Proceedings of CP-AI-OR'05, LNCS, vol. 3524, Springer-Verlag, 2005, pp. 19–33.
[19] P. Van Hentenryck, L. Michel, Synthesis of constraint-based local search algorithms from high-level models, in: Proceedings of AAAI'07, AAAI Press, 2007, pp. 273–278.
[20] A. Nareyek, Using global constraints for local search, in: E. Freuder, R. Wallace (Eds.), Constraint Programming and Large Scale Discrete Optimization, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 57, American Mathematical Society, 2001, pp. 9–28.