Revisiting Constraint-Directed Search

Magnus Ågren[1], Pierre Flener[2⋆1], and Justin Pearson[1]

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

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

Abstract We revisit 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 constraint, such neighbourhoods can be represented via new attributes or primitives for the corresponding constraint object. We show how to define these neighbourhoods for set constraints, whether built-in or specified in monadic existential second-order logic. We also present an implementation of the corresponding primitives in our local search framework. Using these new primitives, we show how some common local-search algorithms are simplified, compared to using just a variable-directed neighbourhood, while not incurring any run-time overhead.

1 Introduction

Constraint-based local search (CBLS, e.g., [11]) integrates ideas from constraint programming into local search. Of particular interest to this paper 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 seen as an object [6,11], storing attributes, such as its set of variables and its penalty, and providing primitives 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 primitives must be maintained incrementally upon each move. This paper contributes to the CBLS endeavour, enriching the interface of constraint objects with new primitives.

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 to variables and to pick a most conflicting variable. However, the abstraction of constraint objects also offers opportunities for *constraint-directed*

⋆ Work done while a Visiting Faculty Member at Sabancı University.

search (e.g., [5,12,11]), 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 argue that *the knowledge of the semantics of a built-in constraint, or even just of a constraint specification, allows the design of the corresponding constraint object to accommodate constraint-directed neighbourhoods whose moves are known to achieve a penalty decrease (or preservation, or increase), without forcing the iteration over the other moves.* This simplifies the design and maintenance of some local search algorithms.

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

- We show how some constraint-directed neighbourhoods can be represented via new primitives for constraint objects: **(i)** For a built-in constraint, these primitives 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 compositional calculi for inferring the existing constraint attributes and primitives from such specifications [3], 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)
- We present common local search heuristics constructed by constraint-directed neighbourhoods as well as by 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, while not incurring any run-time overhead. (Section 4)

In Section 5, we conclude, discuss related work, and outline future work.

2 Preliminaries

A *constraint satisfaction problem* (*CSP*) P 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 each variable in $\mathcal{X}$, and $\mathcal{C}$ is a finite set of constraints, each being defined on a subset of $\mathcal{X}$ and specifying its valid combinations of values. By abuse of language, we often identify a constraint with the singleton set containing it, and P with $\mathcal{C}$.

In this paper, 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*. 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 superiority.

An initial assignment of values to *all* the variables is maintained:

Definition 1. *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP. 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 set*

$\mathcal{C}' \subseteq \mathcal{C}$ *if and only if (iff) each constraint in* $\mathcal{C}'$ *is satisfied under* k*. The set of all configurations for* P *is denoted* $\mathcal{K}_P$*.*

Example 1. Consider the 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 $S \subset T$ is given by $\{S \mapsto \{a, b\}, T \mapsto \{a, b, c\}\}$.

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 2. *Let* $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}\rangle$ *be a CSP. A* neighbourhood function *for* $\mathcal{C}' \subseteq \mathcal{C}$ *is a function* $n : \mathcal{K}_P \to \mathscr{P}(\mathcal{K}_P)$*, and we call the set* $n(k)$ *the* neighbourhood *of* $\mathcal{C}'$ *under* k*. A* move function *for* P *is a function* $m : \mathcal{K}_P \to \mathcal{K}_P$*, and we call the configuration* $m(k)$ *a* move*.*

Focusing on set-CSPs, we here consider the following move functions, for all set variables S, T and universe elements u, v of the considered CSP: $\mathit{add}(S, v)$ adds v to S; $\mathit{drop}(S, u)$ drops u from S; $\mathit{flip}(S, u, v)$ replaces u in S by v; $\mathit{transfer}(S, u, T)$ transfers u from S to T; and $\mathit{swap}(S, u, v, T)$ swaps u of S with v of T. Given a configuration k, the effects of these moves are only defined if $u \in k(S) \wedge v \notin k(S) \wedge u \notin k(T) \wedge v \in k(T)$. For each such move function, we may define a corresponding neighbourhood function for a constraint set $\mathcal{C}$ over the variable set $\mathcal{X}$. For example, given a configuration k, the neighbourhood function $\mathit{Add}(\mathcal{C})$ returns the set of all moves of the form $\mathit{add}(S, v)(k)$, where $S \in \mathcal{X}$, given a configuration k for $\mathcal{X}$. The neighbourhood functions $\mathit{Drop}(\mathcal{C})$, $\mathit{Flip}(\mathcal{C})$, $\mathit{Transfer}(\mathcal{C})$, and $\mathit{Swap}(\mathcal{C})$ are defined similarly. We let $N(\mathcal{C})$ denote the *universal neighbourhood function*, resulting from the union of all these functions.

Example 2. Consider the constraint $S \subset T$ and the configuration $k = \{S \mapsto \{a\}, T \mapsto \{b\}\}$. Assuming that $\mathcal{U} = \{a, b\}$, we have $\mathit{Add}(S \subset T)(k) = \{\mathit{add}(S, b)(k), \mathit{add}(T, a)(k)\}$; $\mathit{Drop}(S \subset T)(k) = \{\mathit{drop}(S, a)(k), \mathit{drop}(T, b)(k)\}$; $\mathit{Flip}(S \subset T)(k) = \{\mathit{flip}(S, a, b)(k), \mathit{flip}(T, b, a)(k)\}$; $\mathit{Transfer}(S \subset T)(k) = \{\mathit{transfer}(S, a, T)(k), \mathit{transfer}(T, b, S)(k)\}$; and $\mathit{Swap}(S \subset T)(k) = \{\mathit{swap}(S, a, b, T)(k)\}$.

The penalty of a constraint set, which is an estimate on how much it is violated, is used to rank the configurations of a neighbourhood.

Definition 3. *Let* $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}\rangle$ *be a CSP. A* penalty function *of* $\mathcal{C}' \subseteq \mathcal{C}$ *is a function* $\mathit{penalty}(\mathcal{C}') : \mathcal{K}_P \to \mathbb{N}$ *such that (s.t.)* $\mathit{penalty}(\mathcal{C}')(k)$*, called the* penalty *of* $\mathcal{C}'$ *under* k*, is zero iff* k *is a solution to* $\mathcal{C}'$*.*

Example 3. $\mathit{AllDisjoint}(\mathcal{X})$ is satisfied under configuration k iff the intersection between any two distinct set variables in $\mathcal{X}$ is empty. The penalty function

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

computes the total number of *drop* moves needed to nullify the penalty of the constraint, that is to transform the current configuration 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 indeed it suffices to drop the three shared elements b, c, d from any set each to get a solution.

When a necessary constraint is not available in our local search framework, we use monadic existential second-order logic (∃MSO) for specifying that constraint.

Example 4. The constraint $AllDisjoint(\{S, T, V\})$ may be specified in ∃MSO by $\Omega = \exists S \exists T \exists V \forall x((x \notin S \vee x \notin T \wedge x \notin V) \wedge (x \notin T \vee x \notin V))$.

We introduced ∃MSO in local search in [1] (in [10], it is used for generating set constraint propagators), and we will use the inductive penalty function proposed there. For example, the penalty of a literal (a constraint here) under a configuration k is 0 if the literal is satisfied under k and 1, otherwise. The penalty of a conjunction (disjunction) is the sum (minimum) of the penalties of its conjuncts (disjuncts). The penalty of a 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.

Example 5. Recall $k = \{S \mapsto \{a, b, c\}, T \mapsto \{b, c, d\}, V \mapsto \{d, e\}\}$ of Ex. 3 and consider Ω of Ex. 4. Then $penalty(\Omega)(k) = 3$, i.e., the same value as obtained by the handcrafted $penalty(AllDisjoint(\mathcal{X}))$ function of Ex. 3.

Example 6. The *progressive party problem* [9] 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 boats whose crews boat h hosts during period p. The following 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 $Partition(\mathcal{X}, Q)$ is satisfied under configuration k iff the values of the set variables in $\mathcal{X}$ partition the constant set Q, where the value of each $S \in \mathcal{X}$ may be the empty set. The constraint $MaxWeightedSum(S, w, m)$ is satisfied under k iff $\sum_{u \in k(S)} w(u) \leq m$. The global constraint $MaxIntersect(\mathcal{X}, m)$ is satisfied under k iff the cardinality of the intersection of any two distinct set variables in $\mathcal{X}$ is at most the constant m.

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. The advantage is that we can exploit combinatorial substructures of the model, and focus on constructing neighbourhoods with particular properties. By doing this, we extend the idea of constraint-directed search [5,12,11] to accommodate moves known to decrease, preserve, or increase the penalty.

Definition 4. *Let c be a constraint on the set variables $\mathcal{X}$, let k be a configuration for $\mathcal{X}$, and let $penalty(c)$ be a penalty function of c. The* decreasing, preserving, and increasing neighbourhoods *of c w.r.t. k and $penalty(c)$ are:*

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

This definition gives the properties of moves of decreasing, preserving, and increasing neighbourhoods, respectively. Given this target, we may now define such neighbourhoods for particular constraints. To present the idea, we do this for the built-in global *AllDisjoint* constraint as well as for any ∃MSO-specified constraint. We then (in Section 3.3) present a feasible implementation approach.

3.1 Built-in Constraints

Let $|\mathcal{X}|_u^k$ denote the number of sets in $\mathcal{X}$ that contain u under k.

Example 7. Let k be a configuration for $\mathcal{X}$. The *decreasing*, *preserving*, and *increasing* neighbourhoods of $AllDisjoint(\mathcal{X})$ under k and (1) are defined by:

$$\begin{aligned}
&\{AllDisjoint(\mathcal{X})\}_k^{\downarrow} = \{drop(S,u)(k) : |\mathcal{X}|_u^k > 1\} \cup \\
&\quad \{flip(S,u,v)(k) : drop(S,u)(k) \in \{AllDisjoint(\mathcal{X})\}_k^{\downarrow} \wedge add(S,v)(k) \in \{AllDisjoint(\mathcal{X})\}_k^{=}\} \\
&\{AllDisjoint(\mathcal{X})\}_k^{=} = \{drop(S,u)(k) : |\mathcal{X}|_u^k = 1\} \cup \{add(S,v)(k) : |\mathcal{X}|_v^k = 0\} \cup \\
&\quad \{flip(S,u,v)(k) : drop(S,u)(k) \in \{AllDisjoint(\mathcal{X})\}_k^{\downarrow} \wedge add(S,v)(k) \in \{AllDisjoint(\mathcal{X})\}_k^{\uparrow} \vee \\
&\qquad\qquad drop(S,u)(k) \in \{AllDisjoint(\mathcal{X})\}_k^{=} \wedge add(S,v)(k) \in \{AllDisjoint(\mathcal{X})\}_k^{=}\} \cup \\
&\quad \{transfer(S,u,T)(k)\} \cup \{swap(S,u,v,T)(k)\} \\
&\{AllDisjoint(\mathcal{X})\}_k^{\uparrow} = \{add(S,v)(k) : |\mathcal{X}|_v^k > 0\} \cup \\
&\quad \{flip(S,u,v)(k) : drop(S,u)(k) \in \{AllDisjoint(\mathcal{X})\}_k^{=} \wedge add(S,v)(k) \in \{AllDisjoint(\mathcal{X})\}_k^{\uparrow}\}
\end{aligned}$$

The condition $S,T \in \mathcal{X} \wedge u,v \in \mathcal{U} \wedge u \in k(S) \wedge v \notin k(S) \wedge u \notin k(T) \wedge v \in k(T)$ is always implicit. Technically, the preserving neighbourhood must also be expanded with all moves on the set variables of the CSP that are not in $AllDisjoint(\mathcal{X})$.

For instance, assume that $k = \{S \mapsto \{b\}, T \mapsto \{b\}, V \mapsto \emptyset\}$ and $\mathcal{U} = \{a,b\}$:

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

Even though these definitions are mutually recursive (for *flip*), 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 the efficiency in practice.

3.2 ∃MSO Constraints

We now define the same 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. *Let c be a constraint and let k be a configuration for the set variables of c. A* delta *for c under k is a pair (ℓ, δ) s.t. ℓ is a neighbour of k and δ is the penalty difference when moving from k to ℓ: $\delta = penalty(c)(\ell) - penalty(c)(k)$.*

Let $A_{|1}$ be the pairs in A projected on their first element. Let $\ell \triangleright M$ be δ if $(\ell, \delta) \in M$, and 0 otherwise, where ℓ is a configuration and M is a delta set.

Definition 6. *Let Φ be a formula in* ∃MSO *and let k be a configuration for the set variables of Φ. The* delta set *$\Delta(\Phi)(k)$ of Φ under k is inductively defined by:*

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

(b) $\Delta(\forall x \phi)(k) = \{(\ell, \delta) : \ell \in (\cup_{u \in \mathcal{U}} \Delta(\phi)(k \cup \{x \mapsto u\}))_{|1} \wedge \delta = \sum_{u \in \mathcal{U}} (\ell \triangleright \Delta(\phi)(k \cup \{x \mapsto u\}))\}$

(c) $\Delta(\exists x \phi)(k) = \{(\ell, \delta) : \ell \in (\cup_{u \in \mathcal{U}} \Delta(\phi)(k \cup \{x \mapsto u\}))_{|1} \wedge \delta = \min_{u \in \mathcal{U}}(penalty(\phi)(k \cup \{x \mapsto u\}) + (\ell \triangleright \Delta(\phi)(k \cup \{x \mapsto u\}))) - penalty(\exists x \phi)(k)\}$

(d) $\Delta(\phi \wedge \psi)(k) = \{(\ell, \delta) : \ell \in (\Delta(\phi)(k) \cup \Delta(\psi)(k))_{|1} \wedge \delta = \ell \triangleright \Delta(\phi)(k) + \ell \triangleright \Delta(\psi)(k)\}$

(e) $\Delta(\phi \vee \psi)(k) = \{(\ell, \delta) : \ell \in (\Delta(\phi)(k) \cup \Delta(\psi)(k))_{|1} \wedge \delta = \min(penalty(\phi)(k) + (\ell \triangleright \Delta(\phi)(k)), penalty(\psi)(k) + (\ell \triangleright \Delta(\psi)(k))) - penalty(\phi \vee \psi)(k))\}$

(f) $\Delta(x \leq y)(k) = \emptyset$ (∗ *similarly for* $<, =, \neq, \geq, >$, *omitted for lack of space* ∗)

(g) $\Delta(x \in S)(k) =$ (∗ *similarly for* $\notin$, *omitted for lack of space* ∗)

$$\begin{cases} \{(drop(S,k(x))(k),1)\} \cup \{(flip(S,k(x),v)(k),1) : v \in \mathcal{U} \setminus k(S)\} \cup \\ \{(transfer(S,k(x),T)(k),1) : T \in \mathcal{X} \wedge k(x) \in \mathcal{U} \setminus k(T)\} \cup \\ \{(swap(S,k(x),v,T)(k),1) : v \notin k(S) \wedge T \in \mathcal{X} \wedge \\ \qquad k(x) \notin k(T) \wedge v \in k(T)\}, \text{ if } k(x) \in k(S) \\ \{(add(S,k(x))(k),-1)\} \cup \{(flip(S,u,k(x))(k),-1) : u \in k(S)\} \cup \\ \{(transfer(T,k(x),S)(k),-1) : T \in \mathcal{X} \wedge k(x) \in k(T)\} \cup \\ \{(swap(S,u,k(x),T)(k),-1) : u \in k(S) \wedge T \in \mathcal{X} \wedge \\ \qquad k(x) \in k(T) \wedge u \notin k(T)\}, \text{ otherwise} \end{cases}$$

Now, the decreasing, preserving, increasing, *and* delta neighbourhoods *of Φ under k and $penalty(\Phi)$ are respectively:*

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

Example 8. Consider Ω of Ex. 4 as well as k and $\mathcal{U}$ of Ex. 7:

$$\begin{aligned}\Delta(\Omega)(k) = \{&(drop(S,b)(k),-1),(drop(T,b)(k),-1),(add(S,a)(k),0),(add(T,a)(k),0),\\&(add(V,a)(k),0),(add(V,b)(k),1),(flip(S,b,a)(k),-1),\\&(flip(T,b,a)(k),-1),(transfer(S,b,V)(k),0),(transfer(T,b,V)(k),0)\}\end{aligned}$$

The obtained neighbourhoods are the same as the handcrafted ones in Ex. 7.

All and only the possible moves are captured in a delta set:

Lemma 1. *Let Φ be in* $\exists$MSO *and k a configuration for Φ: $\Delta(\Phi)(k)_{|1} = N(\Phi)(k)$.*

Proof. ($\subseteq$) Trivial, as $N(\Phi)(k)$ is the set of all possible moves for the set variables of Φ. ($\supseteq$) Assume now that $\ell \in N(\Phi)(k)$. If ℓ is of the form $add(S,v)(k)$, then there must be a subformula ϕ in Φ of the form $v \in S$ or $v \notin S$. Since $v \notin k(S)$ by definition of $add(S,v)$, we then have that $add(S,v)(k) \in (\Delta(\phi)(k))_{|1}$ 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. □

Def. 6 correctly captures the penalty differences in deltas according to Def. 5:

Lemma 2. *Let Φ be a formula in* $\exists$MSO *and let k be a configuration for Φ. If $\ell \in N(\Phi)(k)$, then $\ell \triangleright \Delta(\Phi)(k) = penalty(\Phi)(\ell) - penalty(\Phi)(k)$.*

Proof. Let $A = N(\Phi)(k)$ be the set of all moves on Φ under k. 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 $\ell \in A$. We have that:

$$\begin{aligned}&penalty(\phi \wedge \psi)(\ell) - penalty(\phi \wedge \psi)(k)\\ &= penalty(\phi)(\ell) - penalty(\phi)(k) + penalty(\psi)(\ell) - penalty(\psi)(k), \text{ by def. of penalty}\\ &= \ell \triangleright \Delta(\phi)(k) + \ell \triangleright \Delta(\psi)(k), \text{ by induction}\\ &= \ell \triangleright \Delta(\phi \wedge \psi)(k), \text{ by Def. 6.}\end{aligned}$$

Case (e): $\phi \vee \psi$. Consider a configuration $\ell \in A$. We have that:

$$\begin{aligned}&penalty(\phi \vee \psi)(\ell) - penalty(\phi \vee \psi)(k)\\ &= \min(penalty(\phi)(\ell), penalty(\psi)(\ell)) - penalty(\phi \vee \psi)(k), \text{ by def. of penalty}\\ &= \begin{matrix}\min(penalty(\phi)(k) + \ell \triangleright \Delta(\phi)(k),\\ \quad penalty(\psi)(k) + \ell \triangleright \Delta(\psi)(k)) - penalty(\phi \vee \psi)(k)\end{matrix}, \text{ by induction}\\ &= \ell \triangleright \Delta(\phi \vee \psi)(k), \text{ by Def. 6.}\end{aligned}$$

□

It follows directly from Lemmas 1 and 2 that Def. 6 correctly captures the considered neighbourhoods according to Def. 4:

Proposition 1. *Let Φ be a formula in* $\exists$MSO*, let k be a configuration for the set variables $\mathcal{X}$ of Φ, and let $\ell \in N(\Phi)(k)$. We then have that:*

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

Algorithm 1 *member* and *iterate* primitives for *AllDisjoint*.

```
1: function member({AllDisjoint(𝒳)}↓k)(ℓ, k)
2:     match ℓ with
3:         drop(S, u)(k) ⟶ |𝒳|^k_u > 1
4:       | flip(S, u, v)(k) ⟶ |𝒳|^k_u > 1 ∧ |𝒳|^k_v = 0
5:       | any_other ⟶ false
6:     end match
7: procedure iterate({AllDisjoint(𝒳)}↓k)(S, k, σ)
8:     for all u ∈ {x ∈ k(S) : |𝒳|^k_x > 1} do
9:         σ(drop(S, u)(k))
10:         for all v ∈ {x ∈ 𝒰 \ k(S) : |𝒳|^k_x = 0} do σ(flip(S, u, v)(k)) end for
```

3.3 Implementation Issues

For built-in constraints, the decreasing, preserving, and increasing neighbourhoods are represented procedurally (sometimes with the support of underlying data structures) by *member* and *iterate* primitives. In Algo. 1, we only show these primitives for $\{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k}$. The $\mathit{member}(\{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k})(\ell, k)$ primitive takes two configurations ℓ and k and returns **true** iff $\ell \in \{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k}$. This is the case only when ℓ is of the form $\mathit{drop}(S, u)(k)$ and u occurs more than once in $\mathcal{X}$, or $\mathit{flip}(S, u, v)(k)$ and u (respectively v) occurs more than once (respectively not at all) in $\mathcal{X}$ (lines 3 to 4). A call $\mathit{member}(\{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k})(\ell, k)$ can be performed in constant time, assuming that $|\mathcal{X}|^{k}_{u}$ and $|\mathcal{X}|^{k}_{v}$ are maintained incrementally. The $\mathit{iterate}(\{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k})(S, k, \sigma)$ primitive takes a set variable S, a configuration k, and a function σ and applies σ to each $\ell \in \{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k}$ involving S. This is the case for each configuration ℓ of the form $\mathit{drop}(S, u)(k)$ or $\mathit{flip}(S, u, v)(k)$ s.t. $\mathit{member}(\{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k})(\ell, k)$ holds (lines 9 to 10). The function σ takes a move and works by side effects. A call $\sigma(\ell)$ could, e.g., evaluate the penalty difference between ℓ and the current configuration, and update some internal data structure keeping track of the best such move. A call $\mathit{iterate}(\{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k})(S, k, \sigma)$ can be performed in time $\mathcal{O}(|\{\mathit{AllDisjoint}(\mathcal{X})\}^{\downarrow}_{k}|)$, assuming that the set comprehensions on lines 8 and 10 are maintained incrementally, and that a call to σ takes constant time.

For ∃MSO-specified constraints, the neighbourhoods are represented partly extensionally and partly intensionally. Given a constraint Φ and a configuration k, the subset $\Delta_{|\{a,d\}}(\Phi)(k)$ of $\Delta(\Phi)(k)$ with elements of the form $(\mathit{add}(S, v)(k), \delta)$ or $(\mathit{drop}(S, u)(k), \delta)$ is represented extensionally at every node in the constraint DAG of Φ, and updated incrementally between moves, similarly to incrementally updating penalties [3].

Example 9. Consider $k = \{S \mapsto \{b\}, T \mapsto \{b\}, V \mapsto \emptyset\}$ and $\Delta(\Omega)(k)$ of Ex. 8. The constraint DAG of Ω under k, shown in Fig. 1, contains penalty information (shaded sets) and the sets $\Delta_{|\{a,d\}}(\omega)(k) \subseteq \Delta(\omega)(k)$, for each subformula ω of Ω.

We present *member* and *iterate* primitives only for the decreasing neighbourhood of ∃MSO-specified constraints in Algo. 2. Since $\Delta_{|\{a,d\}}(\phi)(k)$ is represented extensionally for each subformula, we access it in constant time. Both

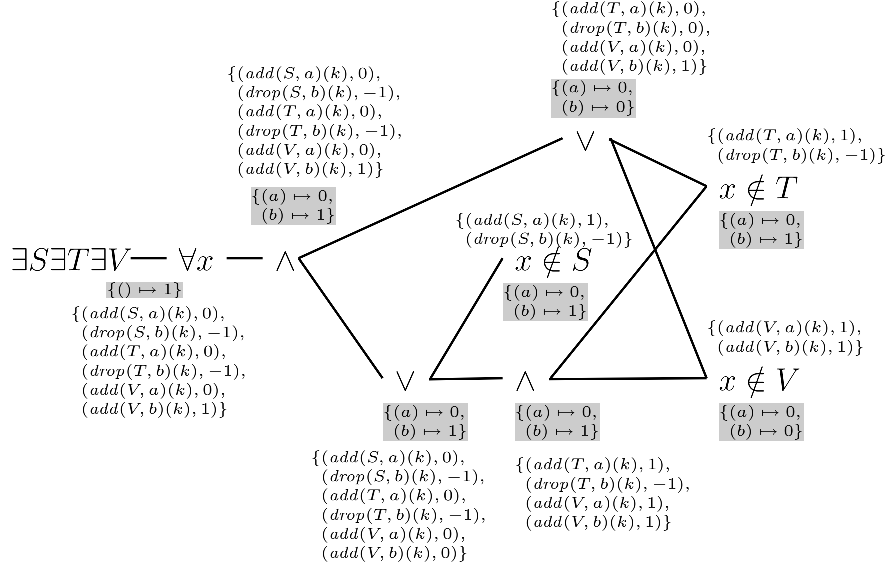

Figure 1. Constraint DAG of $\exists$MSO constraint Ω under configuration k of Ex. 8.

primitives call $\mathit{collect}(\Phi)(S,k,M)$, which takes a set variable S, a configuration k, and a move set M, where S is affected by each move in M and M only contains *flip*, *transfer*, *swap* moves, and returns the delta set for Φ under k, where the configuration ℓ is taken from M for each delta (ℓ, δ). This function is partly described in Algo. 2; all other cases follow similarly from Def. 6. For $\exists S_1 \cdots \exists S_n(\phi)$, the function is called recursively for ϕ (line 11). 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 (lines 12 to 13). For $\phi \land \psi$: **(i)** if S is in both conjuncts, then the value of δ, given a move of the form $\mathit{transfer}(S,u,T)(k)$, is recursively determined as the sum of $\mathit{transfer}(S,u,T)(k) \triangleright \mathit{collect}(\phi)(S,k,M)$ and $\mathit{transfer}(S,u,T)(k) \triangleright \mathit{collect}(\psi)(S,k,M)$ (lines 15 to 17); **(ii)** if S is only in one of the conjuncts, say ϕ, then the value of δ, given a move of the form $\mathit{transfer}(S,u,T)(k)$, is recursively determined as the sum of $\mathit{transfer}(S,u,T)(k) \triangleright \mathit{collect}(\phi)(S,k,M)$ and $\mathit{add}(T,u)(k) \triangleright \Delta_{|\{a,d\}}(\psi)(k)$ (lines 18 to 20). The benefit of representing $\Delta_{|\{a,d\}}(\Phi)(k)$ 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 $\mathit{transfer}(S,u,T)(k)$ move, the value of δ is 1, since u is removed from S (line 23).

Example 10. Consider Ω and $k = \{S \mapsto \{b\}, T \mapsto \{b\}, V \mapsto \emptyset\}$ of Ex. 8. Then $\mathit{collect}(\Omega)(V,k,\{\mathit{flip}(S,b,a)(k)\}) = \{(\mathit{flip}(S,b,a)(k), -1)\}$. Hence, similarly to Ex. 7, $\mathit{flip}(S,b,a)(k) \in \{\Omega\}_k^{\downarrow}$.

By a similar reasoning as in [3, Sect. 5.3], we can see that the time complexity of $\mathit{collect}(\Phi)$ is at worst proportional to the size of Φ. The $\exists$MSO specification we

Algorithm 2 Generic *member* and *iterate* primitives for ∃MSO constraints.

1: **function** $\mathit{member}(\{\Phi\}_k^{\downarrow})(\ell, k)$
2: **match** ℓ **with**
3: $\mathit{drop}(S,u)(k), \mathit{add}(S,v)(k) \longrightarrow \ell \triangleright \Delta_{|\{a,d\}}(\Phi)(k) < 0$
4: $\mid \mathit{flip}(S,u,v)(k), \mathit{transfer}(S,u,T)(k), \mathit{swap}(S,u,v,T)(k) \longrightarrow \ell \triangleright \mathit{collect}(\Phi)(S,k,\{\ell\}) < 0$
5: **end match**
6: **procedure** $\mathit{iterate}(\{\Phi\}_k^{\downarrow})(S,k,\sigma)$
7: **for all** $(\ell,\delta) \in \Delta_{|\{a,d\}}(\Phi)(k)_{|S} \cup \mathit{collect}(\Phi)(S,k,\{\ell : \ell \in N(\Phi)(k)_{|S}\})$ **do**
8: **if** $\delta < 0$ **then** $\sigma(\ell)$ **end if**
9: **function** $\mathit{collect}(\Phi)(S,k,M)$
10: **match** Φ **with**
11: $\exists S_1 \cdots \exists S_n(\phi) \longrightarrow \mathit{collect}(\phi)(S,k,M)$
12: $\mid \forall x(\phi) \longrightarrow \{(\mathit{transfer}(S,u,T)(k),\delta) : \mathit{transfer}(S,u,T)(k) \in M \wedge$
13: $\delta = \mathit{transfer}(S,u,T)(k) \triangleright \mathit{collect}(\phi)(S,k,M)\}$
14: $\mid \phi \wedge \psi \longrightarrow$
15: **if** $S \in \mathit{set_vars}(\phi) \cap \mathit{set_vars}(\psi)$ **then**
16: $\{(\mathit{transfer}(S,u,T)(k),\delta) : \mathit{transfer}(S,u,T)(k) \in M \wedge$
17: $\delta = \mathit{transfer}(S,u,T)(k) \triangleright \mathit{collect}(\phi)(S,k,M) + \mathit{transfer}(S,u,T)(k) \triangleright \mathit{collect}(\psi)(S,k,M)\}$
18: **else if** $S \in \mathit{set_vars}(\phi)$ **then**
19: $\{(\mathit{transfer}(S,u,T)(k),\delta) : \mathit{transfer}(S,u,T)(k) \in M \wedge$
20: $\delta = \mathit{transfer}(S,u,T)(k) \triangleright \mathit{collect}(\phi)(S,k,M) + \mathit{add}(T,u)(k) \triangleright \Delta_{|\{a,d\}}(\psi)(k)\}$
21: **else** (∗ symmetric to the case when $S \in \mathit{set_vars}(\phi)$ ∗)
22: ⋯ (∗ omitted cases ∗) ⋯
23: $\mid x \in S \longrightarrow \{(\mathit{transfer}(S,u,T)(k),1) : \mathit{transfer}(S,u,T)(k) \in M\}$
24: **end match**

have used for *AllDisjoint* is quadratic in the number of variables. In general, an ∃MSO specification will have some overhead in terms of formula length, which is the price to pay for the convenience of using ∃MSO. We come back to this issue in Section 4.3 with experimental evidence that such an overhead can in practice be only linear.

The $\mathit{member}(\{\Phi\}_k^{\downarrow})(\ell,k)$ primitive takes two configurations ℓ and k and returns **true** iff $\ell \in \{\Phi\}_k^{\downarrow}$. If ℓ is an *add* or *drop*, the result is obtained directly from $\Delta_{|\{a,d\}}(\Phi)(k)$ (line 3). Otherwise, the result is obtained from a call $\mathit{collect}(\Phi)(S,k,\{\ell\})$, where S is a variable affected by ℓ (line 4).

The $\mathit{iterate}(\{\Phi\}_k^{\downarrow})(S,k,\sigma)$ primitive takes a set variable S, a configuration k, and 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 $\Delta_{|\{a,d\}}(\Phi)(k)$ and the result of a call $\mathit{collect}(\Phi)(S,k,M)$, 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-specified constraint Φ, the time complexities of *member* and *iterate* are both at worst proportional to the size of Φ, since both call *collect*.

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.

Algorithm 3 Simple heuristic using constraint-directed neighbourhoods.

1: **function** $\textsc{Cds}(\mathcal{C})$
2: $\quad k \leftarrow \textsc{RandomConfiguration}(\mathcal{C})$
3: $\quad$ **while** $penalty(\mathcal{C})(k) > 0$ **do**
4: $\quad\quad$ **choose** $c \in \mathcal{C}$ s.t. $penalty(c)(k) > 0$ **for**
5: $\quad\quad\quad$ **choose** $\ell \in \{c\}_k^{\downarrow}$ **minimising** $penalty(\mathcal{C})(\ell)$ **for** $k \leftarrow \ell$ **end choose**
6: $\quad\quad$ **end choose**
7: $\quad$ **return** k

4.1 Constraint-Directed Heuristics

All heuristics use a **choose** operator to pick a member in a set with some property. For picking a decreasing/preserving/increasing neighbour, this operator can be implemented using the *member* and *iterate* primitives of the constraints.

Simple heuristics. The heuristic Cds in Algo. 3 greedily picks the best neighbour in the set of decreasing neighbours of an unsatisfied constraint. More precisely, Cds takes a set of constraints $\mathcal{C}$ and returns a solution if one is found. It starts by initialising k to a random configuration for all variables in $\mathcal{C}$ (line 2). It then iterates as long as there are any unsatisfied constraints (lines 3 to 6). 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 $\mathcal{C}$ (line 5). A solution is returned if there are no unsatisfied constraints (line 7).

Cds is a variant of `constraintDirectedSearch` [11]. Apart from the additional tabu mechanism of the latter, the main difference is line 5. While in Cds, the decreasing moves are obtained directly from the constraint, meaning that no other moves are evaluated, the decreasing moves of `constraintDirectedSearch` are obtained by evaluating all moves, i.e., also the preserving and increasing ones.

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 line 5 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** $penalty(\mathcal{C})(\ell)$ **for** $k \leftarrow \ell$ **end choose**

While this algorithm is simple to express also in a variable-directed approach (by, e.g., evaluating the penalty differences w.r.t. changing a particular set of variables according to some neighbourhood function, focusing on those giving a lower/constant/higher penalty), the constraint-directed approach allows us to focus directly on the particular kind of moves that we are interested in.

Multi-phase heuristics. One of the advantages with the considered constraint-directed neighbourhoods is the possibilities they open up for the simple design of multi-phase heuristics. This is a well-known method and often crucial to obtain efficient algorithms (see [4], for example). In a multi-phase heuristic, a configuration satisfying a subset $\Pi \subseteq \mathcal{C}$ of the constraints is first obtained. This

Algorithm 4 Multi-phase heuristics using constraint-directed neighbourhoods.

```
1: function CdsPreservingFull(Π, Σ)
2:     k ← Solve(Π)
3:     while penalty(Σ)(k) > 0 do
4:         choose ℓ ∈ Π_k^= minimising penalty(Σ)(k) for k ← ℓ end choose
5:     return k
6: function CdsPreserving(Π, Σ)
7:     k ← Solve(Π)
8:     𝒳 ← the set of all variables of the constraints in Π
9:     while penalty(Σ)(k) > 0 do
10:        choose x ∈ 𝒳 maximising conflict(Σ)(x, k) for
11:            choose ℓ ∈ (Π_|x)_k^= minimising penalty(Σ_|x)(k) for k ← ℓ end choose
12:        end choose
13:    return k
```

configuration is then transformed into a solution satisfying all constraints by only considering the 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 style [7].

In Algo. 4, we show the two multi-phase heuristics CdsPreservingFull and CdsPreserving. Both take two sets of constraints Π and Σ, where $\Pi \cup \Sigma = \mathcal{C}$, and return a solution to $\mathcal{C}$ if one is found. In CdsPreservingFull, a configuration k for all the variables of $\mathcal{C}$, satisfying the constraints in Π, is obtained by the call Solve(Π) (line 2). The function Solve could be 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 to 4). At each iteration, we update k to be any neighbour ℓ that preserves all constraints in Π, minimising the total penalty of Σ (line 4). If there are no more unsatisfied constraints in Σ, then the current configuration (a solution) is returned (line 5).

A problem with CdsPreservingFull is that if Π is large or contains constraints involving many variables, the size of the intersection of the preserving neighbourhoods on line 4 may be too large to obtain an efficient heuristic. We here present one method to overcome this problem, using *conflicting variables*. The conflict of a variable is a measure of how much it contributes to the penalty of the constraints it is involved in. By focusing on moves involving such 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 differs from CdsPreservingFull in the following way: After k is assigned initially, $\mathcal{X}$ is assigned the set of all variables of the constraints in Π (line 8). Then, at each iteration, a most conflicting variable $x \in \mathcal{X}$ is picked (line 10) before the preserving neighbourhoods of the constraints in Π are searched. Next, when the best neighbour is chosen (line 11), the constraints in Π and Σ are projected onto those containing x, drastically reducing the size of the neighbourhood. We use $\mathcal{C}_{|x}$ to denote the constraints in $\mathcal{C}$ 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 a variable-directed and a constraint-directed approach for generating neighbourhoods.

4.2 Avoiding Necessary 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 Ex. 6 is based on a set of set variables $\mathcal{X}$ where each $S_{(h,p)} \in \mathcal{X}$ denotes the set of guest boats whose crews boat h hosts 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 7, the only moves preserving Π are moves that transfer a guest boat from a host boat in a particular period to another host boat in the same period, or moves that swap two guest boats between two host boats in the same period. 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 10. 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.

4.3 Experimental Results

We implemented the ideas presented in this paper for all the constraints used in the model of the progressive party problem as well as for any ∃MSO constraint. The classical instances [9] for the progressive party problem were then run, mimicking the algorithm of [2] but using a variant of CdsPreserving. This meant that the *Partition* constraints were chosen as the preserved constraints Π, that we extended CdsPreserving with the same metaheuristics, maximum number of iterations, and so on, and that the preserving neighbourhood of the *Partition* constraints was restricted to *transfer* moves from the chosen most conflicting set variable to any other set variable.

To show the feasibility of algorithms using the proposed constraint-directed neighbourhoods, the first purpose of experiments is to compare them, within *a given local search framework, with algorithms not using such neighbourhoods, for both built-in and* ∃MSO *constraints. The purpose here is thus not to compare with other models of the progressive party problem in other frameworks.*

We show the experimental comparison with [2] in Table 1. Each entry is the mean time in seconds of 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 memory. We see that, for using the built-in *Partition* constraint, the times are similar, and that there are no overhead problems to mention with constraint-directed neighbourhoods. When using the ∃MSO-specified *Partition* constraint, the run times are 3 to 4 times higher for all these instances. This is

CdsPreserving and built-in $Partition(\mathcal{X}, Q)$

H/periods (fails)	6	7	8	9	10
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)			

CdsPreserving and ∃MSO $Partition(\mathcal{X}, Q)$

H/periods (fails)	6	7	8	9	10
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)			

Algo. of [2] and built-in $Partition(\mathcal{X}, Q)$

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

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

not a surprise since ∃MSO specifications of *Partition* are at least of quadratic length in its number of set variables, leading to an at worst quadratic slowdown for the ∃MSO algorithms compared to the built-in *Partition*. The experiments suggest that the slowdown is actually at worst linear! Compared to the built-in *Partition*, it must also be noted that efforts such as designing penalty and variable-conflict functions with incremental maintenance algorithms as well as implementing *member* and *iterate* primitives were not necessary, since all this is obtained automatically given the ∃MSO specification, as shown in [3] and this paper, respectively.

5 Conclusion

In summary, we 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 the syntax of a specification of a new constraint, neighbourhoods consisting only of configurations with decreased/preserved/increased penalty can be represented via new constraint primitives. We then presented an implementation of the corresponding primitives in our local search framework and, using these new primitives, showed how some local-search algorithms are simplified, compared to using just a variable-directed neighbourhood, while not incurring any run-time overhead.

In terms of related work, the constraint objects of [6,11] have the primitives $getAssignDelta(x, v)$ and $getSwapDelta(x_1, x_2)$ in their interface, returning the penalty changes upon the moves $x := v$ and $x_1 :=: x_2$, respectively. Although it is possible to construct decreasing/preserving/increasing neighbourhoods using these primitives, the signs of their penalty changes are not known in advance. So if one wants to construct, say, a decreasing neighbourhood (as is done in `constraintDirectedSearch` on page 68 in [11], for example), then one may

have to iterate over many moves that turn out to be non-decreasing. This contrasts using the primitives for representing constraint-directed neighbourhoods proposed in this paper, where it is known in advance that exploring the decreasing neighbourhood, say, will only yield moves with a lower penalty. Of course, using the invariants of Comet, it is possible to extend its constraint interface with primitives similar to those proposed in this paper, thus achieving similar results in the (scalar) Comet framework. Conducting payoff experiments (like the one of Section 4.3) *within* the Comet framework is considered future work, while comparisons *between* frameworks are beyond the purpose of this paper.

In [8], 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.

There are many directions for future work. Currently, the preserving neighbourhood $\Pi_k^=$ in line 4 of Algo. 4 is still calculated dynamically as $\bigcap_{c\in\Pi}\{c\}_k^=$, even though the proposed compositional calculus for ∃MSO can handle this. One might even choose a neighbour among $\Pi_k^= \cap \Sigma_k^\downarrow$, 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 parameterised by the neighbourhood function to be used, rather than hardwiring the universal neighbourhood function $N(\mathcal{C})$, and the programmer should be supported in the choice of this parameter.

References

1. M. Ågren, P. Flener, and J. Pearson. Incremental algorithms for local search from existential second-order logic. *Proc. of CP'05*, vol. 3709 of *LNCS*. Springer, 2005.
2. M. Ågren, P. Flener, and J. Pearson. Set variables and local search. In *Proceedings of CP-AI-OR'05*, volume 3524 of *LNCS*, pages 19–33. Springer-Verlag, 2005.
3. M. Ågren, P. Flener, and J. Pearson. Generic incremental algorithms for local search. *Constraints*, 12(3), September 2007.
4. I. Dotú and P. Van Hentenryck. Scheduling social golfers locally. In *Proceedings of CP-AI-OR'05*, volume 3524 of *LNCS*. Springer-Verlag, 2005.
5. M. S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. PhD thesis, Computer Science Department, Carnegie Mellon University, USA, 1983.
6. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):101–110, 2002. *Proceedings of OOPSLA'02*.
7. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1-2):7–43, 1996.
8. A. Nareyek. Using global constraints for local search. In Vol. 57 of *DIMACS: Series in Discrete Maths and Theoretical Computer Science*, pages 9–28. AMS, 2001.
9. B. M. Smith *et al.* The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.
10. G. Tack, C. Schulte, and G. Smolka. Generating propagators for finite set constraints. In *Proceedings of CP'06*, volume 4204 of *LNCS*. Springer-Verlag, 2006.
11. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press 2005.
12. J. P. Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *LNCS*. Springer-Verlag, 1999.