On Constraint-Oriented Neighbours for Local Search

Magnus Ågren[1], Pierre Flener[1,2]*, 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. In the context of local search, we investigate the exploration of constraint-oriented neighbourhoods, where a set of constraints is picked before considering the neighbouring configurations where those constraints may have a different penalty. Given the semantics of a constraint, neighbourhoods consisting only of configurations with decreased (or preserved, or increased) penalty can be represented intensionally as a new attribute for constraint objects. We present a framework for combining neighbourhoods that allows different local search heuristics to be expressed, including multi-phase heuristics where an automatically identifiable suitable subset of the constraints is satisfied upon a first phase and then preserved in a second phase. This simplifies the design of local search algorithms compared to using just a variable-oriented neighbourhood, while not incurring any runtime overhead.

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 tend to guide the search to a local optimum. Metaheuristics such as tabu search or simulated annealing are thus usually 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 (e.g., [10]) integrates ideas from constraint programming (CP) and software engineering to the traditional artificial-intelligence (AI) flavour of research on local search. Of particular interest to this paper is

* Currently an ERASMUS Exchange Teacher visiting from Uppsala University.

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 substructure that is common in combinatorial problems. For instance, the $\mathit{AllDifferent}(x_1, \ldots, x_n)$ constraint requires its arguments to be pairwise different. A constraint can be seen as an object [6, 10], 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 primitive results must be maintained incrementally upon each move.

This paper contributes to the integration of CP ideas into the AI field of local search, enriching the interface of constraint objects with a new attribute and a new primitive. The objective is to continue to simplify the design and maintenance of local search algorithms, along two orthogonal but complementary directions.

First, many neighbourhoods are variable-oriented, in the sense that a 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 offers opportunities for *constraint-oriented neighbourhoods*, in the sense that a set of constraints is picked before considering the neighbouring configurations where those constraints may have a different penalty. The knowledge of the semantics of a constraint allows the designer of the corresponding constraint object to perform a lot of static precomputations for such constraint-oriented neighbourhoods, and these precomputations simplify the design and maintenance of local search algorithms and may even save search time. By doing this, we extend the idea of constraint-directed search (e.g., [5, 12, 10]) to accommodate moves having specific properties, without having to actually evaluate their effect on the penalty.

Second, it is common practice to satisfy in a first phase a subset of the model constraints in the initial configuration, and then to focus during a second phase on neighbourhoods whose moves preserve the truth of these constraints while trying to satisfy the remaining constraints (see [4], for example). However, the desirable separation of concerns propounded by the slogan "*Constraint-Based Local Search = Model + Search*" [10] is not quite achieved yet. If some constraints are no longer preserved by the local search procedure, then they must be made explicit in the model and any newly preserved constraints could be commented out there. A cleaner separation of concerns would be achieved if *all* the problem constraints were always made explicit in the model. Other than easier maintenance and enabled reuse of the model under other solving technologies than local search, the benefit would be the possibility of tool support in the choice of the preserved constraints. Otherwise, their determination is entirely left to the human intuition (and willingness to experiment) of the programmer.

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 problems 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-oriented neighbourhoods can be represented intensionally as an attribute for a constraint object. (Section 3)
- We present a framework for combining neighbourhoods that allows common local search heuristics to be expressed, including multi-phase heuristics where a subset of the constraints is satisfied upon a first phase. We successfully experiment with one of these heuristics, showing how it simplifies the design of the local search algorithm compared to using just a variable-oriented neighbourhood, while not incurring any runtime overhead. (Section 4)
- We provide some experimental evidence for an automatable empirical approach to identifying the subset of constraints that could be satisfied in a first phase and then preserved in a second phase while trying to satisfy the remaining constraints. This approach is partially based on an optional new primitive for a constraint object, namely a search-free construction primitive for computing a configuration that satisfies the constraint. (Section 5)

In Section 6, we conclude, discuss related work, and outline future work.

2 Preliminaries

A *constraint satisfaction problem* (*CSP*) is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a finite set of variables, $\mathcal{D}$ is a finite domain containing the possible values for each variable $x \in \mathcal{X}$, 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 superiourity, and our results transpose to other variants of CSPs, such as the traditional scalar CSPs.

Definition 1 (Configuration and Solution). *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 each constraint $c \in \mathcal{C}'$ is satisfied under k.*

Let $\mathcal{K}_P$ denote the set of all configurations for 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 $s = \{S \mapsto \{a, b\}, T \mapsto \{a, b, c\}\}$. Indeed, $s(S) = \{a, b\}$ is a strict subset of $s(T) = \{a, b, c\}$.

Definition 2 (Neighbourhood and Move). *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 $n : \mathcal{K}_P \to \mathscr{P}(\mathcal{K}_P)$.*
- *The* neighbourhood *of $\mathcal{C}' \subseteq \mathcal{C}$ with respect to a configuration $k \in \mathcal{K}_P$ and a neighbourhood function n is the configuration set $n(k)$.*
- *A* move function *for P is a function $m : \mathcal{K}_P \to \mathcal{K}_P$.*
- *The* variable neighbourhood *for $x \in \mathcal{X}$ with respect to k is the subset of $\mathcal{K}_P$ reachable from k by keeping the bindings of all variables other than x: $n_x(k) = \{\ell \in \mathcal{K}_P : \forall y \in \mathcal{X} : y \neq x \to k(y) = \ell(y)\}$.*

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 $u \in k(S) \wedge v \notin k(S) \wedge u \notin k(T) \wedge v \in k(T)$. This allows us to introduce the following neighbourhood functions for a constraint set $\mathcal{C}$ over the variable set $\mathcal{X}$:

- $\mathit{Add}(\mathcal{C})$ such that $\mathit{Add}(\mathcal{C})(k) = \{\mathit{add}(S, v)(k) : S \in \mathcal{X}\}$.
- $\mathit{Drop}(\mathcal{C})$ such that $\mathit{Drop}(\mathcal{C})(k) = \{\mathit{drop}(S, u)(k) : S \in \mathcal{X}\}$.
- $\mathit{Flip}(\mathcal{C})$ such that $\mathit{Flip}(\mathcal{C})(k) = \{\mathit{flip}(S, u, v)(k) : S \in \mathcal{X}\}$.
- $\mathit{Transfer}(\mathcal{C})$ such that $\mathit{Transfer}(\mathcal{C})(k) = \{\mathit{transfer}(S, u, T)(k) : S, T \in \mathcal{X}\}$.
- $\mathit{Swap}(\mathcal{C})$ such that $\mathit{Swap}(\mathcal{C})(k) = \{\mathit{swap}(S, u, v, T)(k) : S, T \in \mathcal{X}\}$.

Let $N(\mathcal{C})$ denote the neighbourhood function resulting from all these functions:

$$N(\mathcal{C})(k) = \mathit{Add}(\mathcal{C})(k) \cup \mathit{Drop}(\mathcal{C})(k) \cup \mathit{Flip}(\mathcal{C})(k) \cup \mathit{Transfer}(\mathcal{C})(k) \cup \mathit{Swap}(\mathcal{C})(k)$$

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 that:

$$\begin{aligned}
\mathit{Add}(S \subset T)(k) &= \{\mathit{add}(S, b)(k), \mathit{add}(T, a)(k)\} \\
\mathit{Drop}(S \subset T)(k) &= \{\mathit{drop}(S, a)(k), \mathit{drop}(T, b)(k)\} \\
\mathit{Flip}(S \subset T)(k) &= \{\mathit{flip}(S, a, b)(k), \mathit{flip}(T, b, a)(k)\} \\
\mathit{Transfer}(S \subset T)(k) &= \{\mathit{transfer}(S, a, T)(k), \mathit{transfer}(T, b, S)(k)\} \\
\mathit{Swap}(S \subset T)(k) &= \{\mathit{swap}(S, a, b, T)(k)\}
\end{aligned}$$

Definition 3 (Penalty and Variable Conflict). *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 $\mathit{penalty}(\mathcal{C}')(k) = 0$ if and only if k is a solution to $\mathcal{C}'$.*
- *The* penalty *of $\mathcal{C}' \subseteq \mathcal{C}$ under a configuration k is $\mathit{penalty}(\mathcal{C}')(k)$.*
- *A* variable-conflict function *of $\mathcal{C}' \subseteq \mathcal{C}$ is a function $\mathit{conflict}(\mathcal{C}') : \mathcal{X} \times \mathcal{K}_P \to \mathbb{N}$ such that if $\mathit{conflict}(\mathcal{C}')(x, k) = 0$ then $\forall \ell \in n_x(k) : \mathit{penalty}(\mathcal{C}')(k) \leq \mathit{penalty}(\mathcal{C}')(\ell)$. (See [3] for more details).*
- *The* variable conflict *of a variable $x \in \mathcal{X}$ with respect to $\mathcal{C}' \subseteq \mathcal{C}$ under a configuration k is $\mathit{conflict}(\mathcal{C}')(x, k)$.*

Example 3. The global constraint $\mathit{AllDisjoint}(\mathcal{X})$ is satisfied under configuration k if and only if the intersection between any two distinct set variables in $\mathcal{X}$ is empty. The following penalty function:

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

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, and has been experimentally used in [2]. For instance, the penalty of $\mathit{AllDisjoint}(\{S, T, V\})$ under configuration $k = \{S \mapsto \{a, b, c\}, T \mapsto \{b, c, d\}, V \mapsto \{d, e\}\}$ is $8 - 5 = 3$, and indeed it suffices to drop the three shared elements b, c, d from any set each to get a solution. The following variable conflict function:

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

computes the total number of *drop* moves needed on S to nullify the conflict of S. For instance, the conflict of variable S with respect to the penalty and configuration above is 2, and indeed it suffices to drop the two elements b, c it shares with other sets to get a zero conflict of S (but not a zero penalty).

To finish these preliminaries, we now present two set-based models of classical benchmark problems in local search, on which we shall conduct our experiments.

Example 4. 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).

A set-based model of this problem can be designed as follows in our local search framework. Let H and G be the sets of host boats and guest boats, respectively. Let $\mathit{capacity}(h)$ and $\mathit{size}(g)$ denote the spare capacity of host boat h and the crew size of guest boat g, respectively. Let 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) $\forall p \in P : \mathit{Partition}(\{S_{h,p} : h \in H\}, G)$
(c_2) $\forall h \in H : \forall p \in P : \mathit{MaxWeightedSum}(S_{h,p}, \mathit{size}, \mathit{capacity}(h))$
(c_3) $\forall h \in H : \mathit{AllDisjoint}(\{S_{h,p} : p \in P\})$
(c_4) $\mathit{MaxIntersect}(\{S_{h,p} : h \in H \land p \in P\}, 1)$

The global constraint $\mathit{Partition}(\mathcal{X}, Q)$ is satisfied under configuration k if and only if 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 $\mathit{MaxWeightedSum}(S, w, m)$ is satisfied under k if and only if $\sum_{u \in k(S)} w(u) \leq m$. The global constraint $\mathit{MaxIntersect}(\mathcal{X}, m)$ is satisfied under k if and only if the cardinality of the intersection of any two distinct set variables in $\mathcal{X}$ is at most the constant m.

Example 5. 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).

A set-based model of this problem can be designed as follows. 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 constraints then model the problem:

$$\begin{aligned}
&(c_5)\ \forall w \in 1 \ldots nw : \mathit{Partition}(\{S_{g,w} : g \in 1 \ldots ng\}, G)\\
&(c_6)\ \forall g \in 1 \ldots ng : \forall w \in 1 \ldots nw : \mathit{Cardinality}(S_{g,w}, ns)\\
&(c_7)\ \mathit{MaxIntersect}(\{S_{g,w} : g \in 1 \ldots ng \wedge w \in 1 \ldots nw\}, 1)
\end{aligned}$$

The constraint $\mathit{Cardinality}(S, n)$ is satisfied under k if and only if $|k(S)| = n$.

3 Constraint-Oriented Neighbourhoods

We here view the neighbourhoods from a constraint perspective, as opposed to a variable perspective as is often the case. When we construct a neighbourhood from a variable perspective, we usually start from a set of variables and apply changes to one or more of those variables, while evaluating the effect that these changes have on the constraint penalties. 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 then exploit combinatorial substructures of the constraints, and for example focus on constructing neighbourhoods with particular properties. By doing this, we extend the idea of constraint-directed search [5, 12, 10] to accommodate moves with specific properties. In this way, without having to actually evaluate them, moves that decrease, preserve, and increase the penalty are directly obtained from the constraints.

Definition 4. *Let c be a constraint defined on the set of variables $\mathcal{X}$, let k be a configuration for $\mathcal{X}$, and let $\mathit{penalty}(c)$ be a penalty function of c:*

- *The* decreasing neighbourhood *of c with respect to k and $\mathit{penalty}(c)$ is the set $\{c\}_k^{\downarrow} = \{\ell \in N(c)(k) : \mathit{penalty}(c)(k) > \mathit{penalty}(c)(\ell)\}$.*
- *The* preserving neighbourhood *of c with respect to k and $\mathit{penalty}(c)$ is the set $\{c\}_k^{=} = \{\ell \in N(c)(k) : \mathit{penalty}(c)(k) = \mathit{penalty}(c)(\ell)\}$.*
- *The* increasing neighbourhood *of c with respect to k and $\mathit{penalty}(c)$ is the set $\{c\}_k^{\uparrow} = \{\ell \in N(c)(k) : \mathit{penalty}(c)(k) < \mathit{penalty}(c)(\ell)\}$.*

Example 6. Consider the constraints $\mathit{AllDisjoint}(\mathcal{X})$ and $\mathit{Partition}(\mathcal{X}, Q)$ and let k be a configuration for $\mathcal{X}$. The decreasing, preserving, and increasing neighbourhoods for these constraints are listed in Figure 1. In order to keep the notation lean, we use $|\mathcal{X}|_u^k$ as a shorthand for the number of sets in $\mathcal{X}$ that contain u with respect to k, i.e., $|\mathcal{X}|_u^k = |\{T \in \mathcal{X} : u \in k(T)\}|$. Also, 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 neighbourhoods must also be expanded with

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

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

Fig. 1. Decreasing, preserving, and increasing neighbourhoods of *AllDisjoint* and *Partition*. 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, and so are all moves on all variables not in $\mathcal{X}$ for $\{\mathit{Partition}(\mathcal{X},Q)\}^{=}_{k}$ and $\{\mathit{AllDisjoint}(\mathcal{X})\}^{=}_{k}$.

the effects of all moves on the set variables of the CSP that are not part of the considered constraint. The underlying penalty function for *AllDisjoint* is formula (1) of Example 3, and the one for *Partition* is defined by:

$$
\begin{aligned}
&\mathit{penalty}(\mathit{Partition}(\mathcal{X},Q))(k) = \\
&\qquad \left(\sum_{S\in\mathcal{X}} |k(S)|\right) - \left|\bigcup_{S\in\mathcal{X}} k(S)\right| + \left|Q \setminus \bigcup_{S\in\mathcal{X}} k(S)\right| + \left|\bigcup_{S\in\mathcal{X}} k(S) \setminus Q\right| \qquad (2)
\end{aligned}
$$

We only explain the preserving cases for both constraints; the other cases are explained similarly. Regarding $\{\mathit{AllDisjoint}(\mathcal{X})\}^{=}_{k}$: we may keep the value of (1) constant by **(i)** removing an element u from S if S is the only variable containing u, or by **(ii)** adding an element v to S if there is no other variable containing v, or by **(iii)** flipping an element in S if dropping the old element decreases the value of (1) by the same amount (which we know to be 1 here) as adding the new element increases it, or if dropping the old element and adding the new element both keep the value of (1) constant, or by **(iv)** transferring an element from S to T, or by **(v)** swapping an element in S with an element in T.

Regarding $\{Partition(\mathcal{X}, Q)\}^{=}_{k}$: we may keep the value of (2) constant by **(i)** flipping an element in S if dropping the old element increases the value of (2) by the same amount (which we know to be 1 here) as adding the new element decreases it, or the opposite, or by **(ii)** transferring an element from S to T, or by **(iii)** swapping an element in S with an element in T.

It should be noted that, in practice, given a constraint c, the sets $\{c\}^{\downarrow}_{k}$, $\{c\}^{=}_{k}$, and $\{c\}^{\uparrow}_{k}$ are represented intensionally and procedurally by *iterate* and *member* primitives. Hence, they need not be maintained incrementally between configurations. For example, the *iterate* and *member* functions for the *AllDisjoint* constraint can be found in Appendix A. As a result of this, even though the definitions of the sets $\{c\}^{\downarrow}_{k}$, $\{c\}^{=}_{k}$, and $\{c\}^{\uparrow}_{k}$ are mutually recursive in the sense that membership tests are done, those membership tests can be done in constant time, without having to actually recursively construct the sets.

4 Constraint-Oriented Heuristics

We will now show three common heuristics constructed by using the ideas presented in the previous section. All heuristics have a greedy approach and would usually be extended with metaheuristics (e.g., tabu search, simulated annealing, and restarting mechanisms) in real applications.

All heuristics in this section use a choose operator that picks a member in a set having a particular property. In the case of picking a member of the decreasing/preserving/increasing sets of neighbours, this choose operator is implemented by using the *iterate* and *member* functions of the constraints, as mentioned in Example 6 above.

Simple heuristics. We start by showing a simple heuristic BestNeighbour in Algorithm 1. It is a greedy algorithm picking the best neighbour in the set of decreasing neighbours of an unsatisfied constraint. BestNeighbour takes a set of constraints $\mathcal{C}$ as argument and returns a solution to all the constraints if one is found. In the algorithm, we start by initialising k to be a random configuration for all variables in $\mathcal{C}$ (line 2). We then iterate as long as there are any unsatisfied constraints (lines 3 to 8). At each iteration, we pick a constraint c with maximum penalty (line 4), and update k to be any configuration in the decreasing neighbourhood of c minimising the total penalty of $\mathcal{C}$ (line 5). If there are no more unsatisfied constraints, then the current configuration (a solution) is returned (line 9).

BestNeighbour is a variant of `constraintDirectedSearch` of [10]. Apart from the tabu mechanism of `constraintDirectedSearch`, the main difference is the way line 5 is implemented. While in BestNeighbour, 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, given a current configuration, that there always exists at least one decreasing neighbour, BestNeighbour is easily trapped in local minima.

Algorithm 1 Choosing neighbours with maximum penalty decrease.

```
1: function BestNeighbour($\mathcal{C}$)
2:     $k \leftarrow$ RandomConfiguration($\mathcal{C}$)
3:     while $\exists\ c \in \mathcal{C}\ (penalty(c)(k) > 0)$ do
4:         choose $c \in \mathcal{C}$ maximising $penalty(c)(k)$ for
5:             choose $\ell \in \{c\}_k^{\downarrow}$ minimising $\sum_{c \in \mathcal{C}} penalty(c)(\ell)$ for $k \leftarrow \ell$
6:             end choose
7:         end choose
8:     end while
9:     return $k$
10: end function
```

We may improve the algorithm by also allowing preserving and increasing moves of the constraint picked in line 4. This can be done by simply replacing line 5 with the following:

```
if $\{c\}_k^{\downarrow} \neq \emptyset$ then
    choose $\ell \in \{c\}_k^{\downarrow}$ minimising $\sum_{c \in \mathcal{C}} penalty(c)(\ell)$ for $k \leftarrow \ell$ end choose
else if $\{c\}_k^{=} \neq \emptyset$ then
    choose $\ell \in \{c\}_k^{=}$ minimising $\sum_{c \in \mathcal{C}} penalty(c)(\ell)$ for $k \leftarrow \ell$ end choose
else choose $\ell \in \{c\}_k^{\uparrow}$ minimising $\sum_{c \in \mathcal{C}} penalty(c)(\ell)$ for $k \leftarrow \ell$ end choose
end if
```

It should be noted here that, while this algorithm is simple to express also in a variable-oriented 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 lower/constant/higher penalty), the constraint-oriented approach allows us to focus directly on the particular kind of moves (decreasing/preserving/increasing) that we are interested in.

Multi-phase heuristics. One of the advantages with the approach presented in this paper is the possibilities it opens 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 configuration is then transformed into a solution satisfying all constraints by only searching the preserving neighbourhoods of the constraints in Π.

In Algorithm 2, we show a multi-phase heuristic BestPreserving. The algorithm takes two sets of constraints Π and Σ as arguments, where $\Pi \cup \Sigma = \mathcal{C}$, and returns a solution to all constraints in $\mathcal{C}$ if one is found. In the algorithm, a configuration k for all the variables of the constraints in $\mathcal{C} = \Pi \cup \Sigma$, satisfying the constraints in Π, is obtained by the call Solve(Π, Σ) (line 2). The function Solve could be a heuristic method or some other suitable solution method. (Some sets of constraints may even be trivial to satisfy by an initial configuration, without search.) As in Algorithm 1, we then iterate as long as there are any unsatisfied constraints in Σ (lines 3 to 6). At each iteration, we update k to be any neighbour ℓ that preserves all constraints in Π, minimising the total penalty

Algorithm 2 Choosing preserving neighbours with maximum penalty decrease.

```
1: function BestPreserving($\Pi, \Sigma$)
2:     $k \leftarrow$ Solve($\Pi, \Sigma$)
3:     while $\exists\ c \in \Sigma\ (penalty(c)(k) > 0)$ do
4:         choose $\ell \in \bigcap_{c \in \Pi} \{c\}^{=}_{k}$ minimising $\sum_{c \in \Sigma} penalty(c)(k)$ for $k \leftarrow \ell$
5:         end choose
6:     end while
7:     return $k$
8: end function
```

of Σ (lines 4 and 5). If there are no more unsatisfied constraints in Σ, then the current configuration (a solution) is returned (line 7).

One problem with Algorithm 2 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 algorithm. There are many methods to overcome this problem and we will here present one such method using conflicting variables. The conflict of a variable is a measure of how much a particular variable contributes to the penalty of the constraints it is involved in (see Definition 3). 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 BestPreservingProjected shown in Algorithm 3 differs from BestPreserving in the following way: After k is assigned initially, $\mathcal{X}$ is assigned the set of all variables of the constraints in Π (line 3). Then, at each iteration, a most conflicting variable $x \in \mathcal{X}$ is picked (line 5) before the preserving neighbourhoods of the constraints in Π are searched. Next, when the best neighbour is to be picked (line 6), the constraints in Π and Σ are projected onto those containing x, drastically reducing the number of neighbours to consider. We use $\mathcal{C}_{|x}$ to denote the constraints in $\mathcal{C}$ containing x, e.g., $\{x < a, x \neq b, y > a\}_{|x} = \{x < a, x \neq b\}$.

Note that projecting neighbourhoods onto those containing a particular set of variables, such as conflicting variables, is a very useful *variable-oriented approach* for speeding up heuristic methods. In this way, Algorithm 3 is a fruitful cross-fertilisation between a variable-oriented and a constraint-oriented approach for generating neighbourhoods.

Necessary data-structures for free. Another advantage with the approach presented in this paper is that necessary data structures for generating neighbourhoods that traditionally had to be explicitly created come for free here.

The model of the progressive party problem of Example 4 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 model using Algorithm 3 where $\Pi = \{Partition(\{S_{h,p} : h \in H\}, G) : p \in P\}$. Now,

Algorithm 3 Choosing preserving neighbours with maximum penalty decrease using conflicting variables.

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

having obtained a partial solution that satisfies Π in line 2 of the algorithm, 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. The reason for why moves that flip a guest boat of a host boat in a particular period is not possible, even though *flip* moves are in the set $\{Partition(\mathcal{X}, Q)\}_k^{=}$ defined in Figure 1, is that $Q = G = \mathcal{U}$ in the progressive party problem. Whenever this is the case, *flip* moves are not in $\{Partition(\mathcal{X}, Q)\}_k^{=}$, which can be seen in Figure 1. Now, to generate these preserving moves from a variable-oriented 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 6 of the algorithm. By instead viewing this problem from a constraint-oriented perspective, we can 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 5, which is based on a set of set variables $\mathcal{X}$ where each $S_{g,w} \in \mathcal{X}$ denotes the set of golfers in group g of week w. Assuming that

$$\begin{aligned}\Pi = \{&Partition(\{S_{g,w} : g \in 1 \ldots ng\}, G) : w \in 1 \ldots nw\} \cup \\ &\{Cardinality(S_{g,w}, ns) : g \in 1 \ldots ng \wedge w \in 1 \ldots nw\},\end{aligned}$$

the only moves preserving Π are moves that swap two golfers in different groups in the same week. Again, by looking at this from a constraint-oriented perspective, the preserving moves are obtained directly from the constraints in Π and no additional data structures for accessing the different weeks are needed.

Experimental results. We implemented the ideas presented so far in this paper for all the constraints used in the models of the progressive party problem and the social golfer problem. The same experiments as those in [2] were then run, mimicking the algorithms there but using the approach of this paper. For

Progressive Party (Algorithm 3)

H/periods (fails)	6	7	8	9	10
1-12,16			0.7	1.9	18.0
1-13			9.0	89.4	
1,3-13,19			9.1	132.8 (4)	
3-13,25,26			19.6	206.9 (16)	
1-11,19,21	11.4	90.7			
1-9,16-19	17.7	176.6 (16)			

Progressive Party (Algorithm of [2])

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)			

Social Golfer (Algorithm 3)

ng-ns-nw	time	(fails)	*ng-ns-nw*	time	(fails)
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	

Social Golfer (Algorithm of [2])

ng-ns-nw	time	(fails)	*ng-ns-nw*	time	(fails)
6-3-7	0.4		6-3-8	215.0	(76)
7-3-9	138.0	(5)	8-3-10	14.4	
9-3-11	3.5		10-3-13	325.0	(35)
6-4-5	0.3		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	

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

both problems, this meant that all constraints were present in the models and those discussed in the previous section were the ones chosen as the sets of preserved constraints Π. It also meant that we extended Algorithm 3 with the same metaheuristics, maximum number of iterations, and so on as in [2]. For the progressive party problem, this additionally meant that we restricted the preserving neighbourhoods of the partition constraints used in line 6 to only *transfer* moves from the picked most conflicting variable, as these are the only kind of moves considered in [2].

We show the experimental results (the ones obtained by using the approach of this paper as well as those of [2]) in Table 1. Each entry in the tables is the mean value 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 implemented in Objective Caml (`http://www.ocaml.org`) and run on an Intel 2.4 Ghz Linux machine with 512 MB memory. Comparing the different tables, we see that the results are similar. For some instances, the results of [2] are slightly better, and for some other instances, the results presented here are slightly better. Hence there are no overhead problems with the approach taken in this paper.

5 Identifying the Constraints To Be Preserved

The multi-phase heuristics of the previous section hinge entirely on the identification of a suitable subset Π of the problem constraints $\mathcal{C}$ that ought to be satisfied upon a first phase by an initial configuration and then preserved during the second phase while trying to satisfy the remaining constraints $\mathcal{C} \setminus \Pi$.

While especially global constraints might carry enough information to enable an offline static analysis of the model leading to a suggestion of Π, we here take an empirical approach to this open research question.

A conjunction of constraints bundled under a possibly nested universal quantification is here called a *quantified constraint*. We shall reason at the level of quantified constraints rather than at the level of constraints, as we see no reason to lower the grain of the analysis that much, due to the otherwise ensuing combinatorial explosion (of at least our approach).

Suppose the constraint set $\mathcal{C}$ of the model has n quantified constraints. The idea is very simple, and amenable to full automation. For each of the 2^n subsets $\pi \subseteq \mathcal{C}$ of quantified constraints, we add up the following two amounts of time:

1. The time t^s_π to find a configuration k_π satisfying π.
2. The time t^p_π to find a configuration $k_\mathcal{C}$ satisfying $\mathcal{C} \setminus \pi$, proceeding by local search from k_π under moves preserving π. Note that $k_\mathcal{C}$ even satisfies $\mathcal{C}$ and is thus a solution to the considered problem.

The subset $\Pi \subseteq \mathcal{C}$ of quantified constraints that are to be preserved is then empirically determined to be the set π such that the total solution time $T_\pi = t^s_\pi + t^p_\pi$ is minimal for all $\pi \subseteq \mathcal{C}$.

Note that $t^s_\emptyset = \epsilon \approx 0 = t^p_\mathcal{C}$, if we consider negligible the time to find a configuration satisfying the empty set of constraints, that is the time ϵ to construct a random configuration. Also note that $t^p_\emptyset + \epsilon = t^s_\mathcal{C}$, as a random initial configuration has to be constructed first when trying to satisfy the full set of constraints, which itself takes the same time as preserving the empty set of constraints. Hence $T_\emptyset = T_\mathcal{C}$. We are in practice only interested in the non-empty strict subsets π of $\mathcal{C}$ where $T_\pi \leq T_\mathcal{C} = T_\emptyset$, so upon determining $T_\emptyset$ (or $T_\mathcal{C}$) first, we can set that value as a time-out for the other runs. A useful generalisation is to add up, at every iteration, the times for several training instances, rather than just one.

Although the number of subsets is exponential in n, this is not a problem as there are often not that many quantified constraints in a model. For instance, the progressive party model in Example 4 has only $n = 4$ quantified constraints.

In the absence of a search-free construction primitive for finding k_π, we proceed only by local search from a random initial configuration. While such construction primitives may be available for individual constraints, and thus for quantified constraints (if the scopes of the quantifications do not overlap), it is an open question how to compose them for a set of (quantified) constraints.

If many instances of the problem have to be solved, then such a possibly lengthy pre-processing of the model may pay off. We assume that a representative set of training instances of the problem is available, and that any conclusions drawn from them carry over to the actual instances.

Example 7. Let us check this automatable approach on our model of the progressive party problem in Example 4, even though this is not the kind of problem

Preserved set	3 periods	4 periods	5 periods	6 periods	7 periods	8 periods
$\{c_1\}$, by search	0.97	1.39	1.98	2.76	4.30	8.29
$\{c_1\}$, by construction	0.28	0.41	0.72	1.21	2.38	5.93
$\{c_2\}$	18.98	52.35	92.40	–	–	–
$\{c_3\}$, by search	15.56	41.16	125.49	–	–	–
$\{c_3\}$, by construction	15.10	40.47	124.46	–	–	–
$\{c_4\}$	2.25	3.27	5.74	12.87	31.72	435.56
$\{c_1, c_2\}$	6.59	6.55	10.30	27.38	27.69	33.84
$\{c_1, c_3\}$	0.92	1.31	1.81	2.47	3.57	7.89
$\{c_1, c_4\}$	2.04	2.88	4.59	7.21	10.85	22.28
$\{c_2, c_3\}$	22.70	53.84	78.54	149.97	–	–
$\{c_2, c_4\}$	8.50	16.44	28.22	76.26	415.34	–
$\{c_3, c_4\}$	2.38	3.83	6.82	13.60	48.38	635.96
$\{c_1, c_2, c_3\}$	30.81	78.67	–	–	–	–
$\{c_1, c_2, c_4\}$	17.50	51.03	114.70	–	–	–
$\{c_1, c_3, c_4\}$	1.83	3.02	4.61	6.85	13.29	30.24
$\{c_2, c_3, c_4\}$	7.71	14.39	29.88	148.66	–	–
$\{c_1, c_2, c_3, c_4\}$	97.48	–	–	–	–	–

Table 2. Times T_π to solve the $1-13$ instance of the progressive party problem over 3 to 8 periods, while preserving various non-empty subsets π of the constraints

where many instances have to be run in practice before recovering the preprocessing cost. There are $n = 4$ quantified constraints in the constraint set $\mathcal{C}$, hence only $2^4 = 16$ experiments over subsets of $\mathcal{C}$ are necessary, or 15 actually, as $T_\emptyset = T_\mathcal{C}$. Table 2 reports, for each of the non-empty subsets $\pi \subseteq \mathcal{C}$, the average times T_π it takes to solve over 3 to 8 periods the classical instance of [9] where the first 13 of the 42 boats are the host boats. A dash (–) indicates that less than 50% of the 10 runs on that instance succeeded in finding solutions. The times t^s_π were obtained by local search from a random initial configuration, as well as, for some of the singleton sets π, by available search-free construction primitives.

The table reveals that, in all instances, the subset $\pi_{13} = \{c_1, c_3\}$ leads to the shortest overall runtime when no construction primitives are used, suggesting that the quantified constraints over *Partition* and *AllDisjoint* ought to be satisfied first and then preserved. The rankings are relatively consistent across the instances. The subset $\pi_1 = \{c_1\}$ is always a close runner-up, but is actually overall the best one when we use a search-free construction primitive for the *Partition* constraint, and thus for the quantified constraint c_1 over *Partition*. This mechanisable empirical determination of $\Pi = \pi_1$ confirms our human intuition in our experiments of [2], but also revealed with π_{13} an interesting other candidate for Π that we had not thought of.

6 Conclusion

In summary, we have first argued for the exploration of constraint-oriented neighbourhoods in local search, where a set of constraints is picked before considering the neighbouring configurations where those constraints may have a different penalty. Given the semantics of a constraint, neighbourhoods consisting only of configurations with decreased (or preserved, or increased) penalty can be represented intensionally as a new attribute for a constraint object. We have then presented a framework using constraint-oriented neighbourhoods that allows different local search heuristics to be expressed, including multi-phase heuristics where an automatically identifiable suitable subset of the constraints is satisfied upon a first phase and then preserved in a second phase. This simplifies the design of local search algorithms compared to using just a variable-oriented neighbourhood, while not incurring any runtime overhead.

In terms of related work, the constraint objects of [6, 10] have the methods $\mathit{getAssignDelta}(x, v)$ and $\mathit{getSwapDelta}(x_1, x_2)$ 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 methods, the signs of their results are not known in advance, so if one, e.g., wants to construct the decreasing neighbourhood, then one may have to iterate over many moves that turn out to be non-decreasing. This is in contrast to our proposed attribute storing constraint-oriented neighbourhoods, as it is known in advance that exploring the decreasing neighbourhood, say, will only yield neighbourhoods with a lower penalty with respect to that constraint.

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.

Our approach to identifying a subset of constraints that are satisfied in a first phase and then preserved in a second phase was inspired by MultiTAC [7], which automatically synthesises an instance-distribution-specific solver, given a high-level model of some CSP and a set of training instances (or a generator thereof). MultiTAC uses an offline brute-force approach to generate candidate problem-specific heuristics from a set of heuristics described by a grammar, just like our offline exhaustive enumeration of all candidate subsets.

In this paper we have started to explore new directions in automatic neighbourhood generation for local search and there are many further areas for future work. First, considering that *flip*, *transfer*, and *swap* moves essentially are transactions over *add* and *drop* moves, it might be possible to assist the designer of a constraint object by inferring the constraint-oriented neighbourhoods for the former from the latter. Also, in our first approach, we just precompute the *sign* of the penalty change in our constraint-oriented neighbourhoods, but it might be possible to precompute the actual *value* of that change. This would open opportunities for precomputing the penalty change of a *conjunction* of constraints, rather than just a single constraint. For instance, noting that $\mathit{Partition}(\mathcal{X}, Q) \equiv \mathit{AllDisjoint}(\mathcal{X}) \wedge \mathit{Union}(\mathcal{X}, Q)$, it would be useful to precompute compositionally, in the spirit of the compositional calculi in [3, 11], the neighbourhoods of *Partition* in Figure 1 from those of *AllDisjoint* in the same figure and those of *Union* (not listed here). Similarly for precomputing $\bigcap_{c \in \Pi} \{c\}_k^=$ in line 4 of Algorithm 2 as $\Pi_k^=$. Currently, that intersection must be calculated dynamically, by iterating over the extensions under the current configuration of these intensional sets. One could even improve that line 4 into choosing ℓ among $\Pi_k^= \cap \Sigma_k^{\downarrow}$, by statically precomputing 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 hard-wiring the universal neighbourhood function $N(\mathcal{C})$, and the programmer should be supported in the choice of this parameter.

Acknowledgements. 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.

References

1. E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization.* John Wiley & Sons, 1997.
2. M. Ågren, P. Flener, and J. Pearson. Set variables and local search. In R. Barták and M. Milano, editors, *Proceedings of CP-AI-OR'05*, volume 3524 of *LNCS*, pages 19–33. Springer-Verlag, 2005.
3. M. Ågren, P. Flener, and J. Pearson. Inferring variable conflicts for local search. In F. Benhamou, editor, *Proceedings of CP'06*, volume 4204 of *LNCS*, pages 665–669. Springer-Verlag, 2006.
4. I. Dotú and P. Van Hentenryck. Scheduling social golfers locally. In R. Barták and M. Milano, editors, *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, December 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 E. Freuder and R. Wallace, editors, *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 9–28. American Mathematical Society, 2001.
9. B. M. Smith, S. C. Brailsford, P. M. Hubbard, and H. P. Williams. The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.

10. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* The MIT Press, 2005.
11. P. Van Hentenryck and L. Michel. Differentiable invariants. In F. Benhamou, editor, *Proceedings of CP'06*, volume 4204 of *LNCS*, pages 604–619. Springer-Verlag, 2006.
12. J. P. Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *LNCS*. Springer-Verlag, 1999.

A Constraint Primitives

A.1 Member Predicates

– $member(\{AllDisjoint(\mathcal{X})\}_k^{\downarrow})(m,k) = \mathbf{T}$ if and only if $m \in N(AllDisjoint(\mathcal{X}))(k)$ and $penalty(AllDisjoint(\mathcal{X}))(m) < penalty(AllDisjoint(\mathcal{X}))(k)$.

```
function $member(\{AllDisjoint(\mathcal{X})\}_k^{\downarrow})(m,k)$
    match $m$ with
        $drop(S,u)(k) \longrightarrow |\{T \in \mathcal{X} : u \in k(T)\}| > 1$
      | $flip(S,u,v)(k) \longrightarrow$
            $member(\{AllDisjoint(\mathcal{X})\}_k^{\downarrow})(drop(S,u)(k),k) \wedge$
            $member(\{AllDisjoint(\mathcal{X})\}_k^{=})(add(S,v)(k),k)$
      | $any\_other \longrightarrow \mathbf{F}$
    end match
end function
```

– $member(\{AllDisjoint(\mathcal{X})\}_k^{=})(m,k) = \mathbf{T}$ if and only if $m \in N(AllDisjoint(\mathcal{X}))(k)$ and $penalty(AllDisjoint(\mathcal{X}))(m) = penalty(AllDisjoint(\mathcal{X}))(k)$.

```
function $member(\{AllDisjoint(\mathcal{X})\}_k^{=})(m,k)$
    match $m$ with
        $drop(S,u)(k) \longrightarrow |\{T \in \mathcal{X} : u \in k(T)\}| = 1$
      | $add(S,v)(k) \longrightarrow |\{T \in \mathcal{X} : v \in k(T)\}| = 0$
      | $flip(S,u,v)(k) \longrightarrow$
            $member(\{AllDisjoint(\mathcal{X})\}_k^{\downarrow})(drop(S,u)(k),k) \wedge$
            $member(\{AllDisjoint(\mathcal{X})\}_k^{\uparrow})(add(S,v)(k),k)$
            $\vee$
            $member(\{AllDisjoint(\mathcal{X})\}_k^{=})(drop(S,u)(k),k) \wedge$
            $member(\{AllDisjoint(\mathcal{X})\}_k^{=})(add(S,v)(k),k)$
      | $transfer(S,u,T)(k) \longrightarrow \mathbf{T}$
      | $swap(S,u,v,T)(k) \longrightarrow \mathbf{T}$
    end match
end function
```

– $member(\{AllDisjoint(\mathcal{X})\}_k^{\uparrow})(m,k) = \mathbf{T}$ if and only if $m \in N(AllDisjoint(\mathcal{X}))(k)$ and $penalty(AllDisjoint(\mathcal{X}))(m) > penalty(AllDisjoint(\mathcal{X}))(k)$.

```
function $member(\{AllDisjoint(\mathcal{X})\}_k^{\uparrow})(m,k)$
    match $m$ with
        $add(S,v)(k) \longrightarrow |\{T \in \mathcal{X} : v \in k(T)\}| > 0$
      | $flip(S,u,v)(k) \longrightarrow$
            $member(\{AllDisjoint(\mathcal{X})\}_k^{=})(drop(S,u)(k),k) \wedge$
            $member(\{AllDisjoint(\mathcal{X})\}_k^{\uparrow})(add(S,v)(k),k)$
        | $any\_other \longrightarrow$ **F**
    **end match**
**end function**
```

A.2 Iterate Primitives

– $iterate(\{AllDisjoint(\mathcal{X})\}_k^{\downarrow})(k, f)$ applies f to each $m \in N(AllDisjoint(\mathcal{X}))(k)$ such that $penalty(AllDisjoint(\mathcal{X}))(m) < penalty(AllDisjoint(\mathcal{X}))(k)$.

```
function $iterate(\{AllDisjoint(\mathcal{X})\}_k^{\downarrow})(k, f)$
    for all $S \in \mathcal{X}$ do
        for all $u \in k(S)$ $s.t.$ $|\{T \in \mathcal{X} : u \in k(T)\}| > 1$ do
            $f(drop(S, u)(k))$
            for all $v \in \mathcal{U} \setminus k(S)$ $s.t.$ $|\{T \in \mathcal{X} : v \in k(T)\}| = 0$ do
                $f(flip(S, u, v)(k))$
            end for
        end for
    end for
end function
```

– $iterate(\{AllDisjoint(\mathcal{X})\}_k^{\downarrow})(k, f)$ applies f to each $m \in N(AllDisjoint(\mathcal{X}))(k)$ such that $penalty(AllDisjoint(\mathcal{X}))(m) = penalty(AllDisjoint(\mathcal{X}))(k)$.

```
function $iterate(\{AllDisjoint(\mathcal{X})\}_k^{=})(k, f)$
    for all $S \in \mathcal{X}$ do
        for all $u \in \mathcal{U}$ do
            if $u \in k(S)$ then
                if $|\{T \in \mathcal{X} : u \in k(T)\}| = 1$ then
                    $f(drop(S, u)(k))$
                    for all $v \in \mathcal{U} \setminus k(S)$ $s.t.$ $|\{T \in \mathcal{X} : v \in k(T)\}| = 0$ do
                        $f(flip(S, u, v)(k))$
                    end for
                else if $|\{T \in \mathcal{X} : u \in k(T)\}| > 1$ then
                    for all $v \in \mathcal{U} \setminus k(S)$ $s.t.$ $|\{T \in \mathcal{X} : v \in k(T)\}| > 0$ do
                        $f(flip(S, u, v)(k))$
                    end for
                end if
                for all $T \in \mathcal{X} \setminus \{S\}$ do
                    if $u \notin k(T)$ then
                        $f(transfer(S, u, T)(k))$
                        for all $v \in k(T)$ $s.t.$ $v \notin k(S)$ do
                            $f(swap(S, u, v, T)(k))$
                        end for
                    end if
                end for
                else
                    if $|\{T \in \mathcal{X} : u \in k(T)\}| = 0$ then
                        $f(add(S, u)(k))$
                    end if
                end if
            end for
        end for
    end function
```

– $iterate(\{AllDisjoint(\mathcal{X})\}_k^{\downarrow})(k, f)$ applies f to each $m \in N(AllDisjoint(\mathcal{X}))(k)$ such that $penalty(AllDisjoint(\mathcal{X}))(m) > penalty(AllDisjoint(\mathcal{X}))(k)$.

```
    function $iterate(\{AllDisjoint(\mathcal{X})\}_k^{\uparrow})(k, f)$
        for all $S \in \mathcal{X}$ do
            for all $v \in \mathcal{U} \setminus k(S)$ $s.t.$ $|\{T \in \mathcal{X} : v \in k(T)\}| > 0$ do
                $f(add(S, v)(k))$
                for all $u \in k(S)$ $s.t.$ $|\{T \in \mathcal{X} : u \in k(T)\}| = 1$ do
                    $f(flip(S, u, v)(k))$
                end for
            end for
        end for
    end function
```