Incremental Algorithms for Local Search from Existential Second-Order Logic

Magnus Ågren, Pierre Flener, and Justin Pearson

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

Abstract. Local search is a powerful and well-established method for solving hard combinatorial problems. Yet, until recently, it has provided very little user support, leading to time-consuming and error-prone implementation tasks. We introduce a scheme that, from a high-level description of a constraint in existential second-order logic with counting, automatically synthesises incremental penalty calculation algorithms. The performance of the scheme is demonstrated by solving real-life instances of a financial portfolio design problem that seem unsolvable in reasonable time by complete search.

1 Introduction

Local search is a powerful and well-established method for solving hard combinatorial problems [1]. Yet, until recently, it has provided very little user support, leading to time-consuming and error-prone implementation tasks. The recent emergence of languages and systems for local search, sometimes based on novel abstractions, has alleviated the user of much of this burden [10, 16, 12, 11].

However, if a problem cannot readily be modelled using the primitive constraints of such a local search system, then the *user* has to perform some of those time-consuming and error-prone tasks. These include the design of algorithms for the calculation of penalties of user-defined constraints. These algorithms are called very often in the innermost loop of local search and thus need to be implemented particularly efficiently: incrementality is crucial. Would it thus not be nice if also this task could be performed fully automatically and satisfactorily by a local search system? In this paper, we design a scheme for doing just that, based on an extension of the idea of combinators [15] to quantifiers. Our *key contributions* are as follows:

- We propose the usage of existential second-order logic with counting as a *high-level modelling language* for (user-defined) constraints. It accommodates set variables and captures at least the complexity class NP.
- We design a scheme for the *automated synthesis of incremental penalty calculation algorithms* from a description of a (user-defined) constraint in that language. We have developed an *implementation* of this scheme.

– We propose a *new benchmark problem* for local search, with applications in finance. Using our local search framework, we *exactly solve real-life instances* that seem unsolvable in reasonable time by complete search; the performance is competitive with a fast approximation method based on complete search.

The rest of this paper is organised as follows. In Section 2, we define the background for this work, namely constraint satisfaction problems over scalar and set variables as well as local search concepts. The core of this paper are Sections 3 to 6, where we introduce the used modelling language and show how incremental algorithms for calculating penalties can be automatically synthesised from a model therein. In Section 7, we demonstrate the performance of this approach by solving real-life instances of a financial portfolio design problem. Finally, we summarise our results, discuss related work, and outline future work in Section 8.

2 Preliminaries

As usual, a *constraint satisfaction problem* (*CSP*) is a triple $\langle V, D, C\rangle$, where V is a finite set of variables, D is a finite set of domains, each $D_v \in D$ containing the set of possible values for the corresponding variable $v \in V$, and C is a finite set of constraints, each $c \in C$ being defined on a subset of the variables in V and specifying their valid combinations of values.

Definition 1 (Set Variable and its Universe). *Let $P = \langle V, D, C\rangle$ be a CSP. A variable $S \in V$ is a* set variable *if its corresponding domain $D_S = 2^{\mathcal{U}_S}$, where $\mathcal{U}_S$ is a finite set of values of some type, called the* universe *of S.*

Without loss of generality, we assume that all the set variables have a common universe, denoted $\mathcal{U}$. We also assume that *all* the variables are set variables, and denote such a set-CSP by $\langle V, \mathcal{U}, C\rangle$. This is of course a limitation, since many models contain both set variables and scalar variables. Fortunately, interesting applications, such as the ones in this paper and in [2], can be modelled using only set variables.

A constraint program assigns values to the variables one by one, but local search maintains an (initially arbitrary) assignment of values to *all* the variables:

Definition 2 (Configuration). *Let $P = \langle V, \mathcal{U}, C\rangle$ be a set-CSP. A* configuration *for P (or V) is a total function $k : V \to 2^{\mathcal{U}}$.*

As usual, the notation $k \models \phi$ expresses that the open formula ϕ is satisfied under the configuration k.

Example 1. Consider a set-CSP $P = \langle\{S_1, S_2, S_3\}, \{d_1, d_2, d_3\}, \{c_1, c_2\}\rangle$. A configuration for P is given by $k(S_1) = \{d_3\}, k(S_2) = \{d_1, d_2\}, k(S_3) = \emptyset$, or equivalently as the set of mappings $\{S_1 \mapsto \{d_3\}, S_2 \mapsto \{d_1, d_2\}, S_3 \mapsto \emptyset\}$. Another configuration for P is given by $k' = \{S_1 \mapsto \emptyset, S_2 \mapsto \{d_1, d_2, d_3\}, S_3 \mapsto \emptyset\}$.

Local search iteratively makes a small change to the current configuration, upon examining the merits of many such changes. The configurations thus examined constitute the neighbourhood of the current configuration:

Definition 3 (Neighbourhood). *Let K be the set of all configurations for a (set-)CSP P and let $k \in K$. A* neighbourhood function *for P is a function $\mathcal{N} : K \to 2^K$. The* neighbourhood *of P with respect to k and $\mathcal{N}$ is the set $\mathcal{N}(k)$.*

Example 2. Reconsider P and k from Example 1. A neighbourhood of P with respect to k and some neighbourhood function for P is the set $\{k_1 = \{S_1 \mapsto \emptyset, S_2 \mapsto \{d_1, d_2, d_3\}, S_3 \mapsto \emptyset\}, k_2 = \{S_1 \mapsto \emptyset, S_2 \mapsto \{d_1, d_2\}, S_3 \mapsto \{d_3\}\}$. This neighbourhood function moves the value d_3 in S_1 to S_2 or S_3.

The penalty of a CSP is an estimate on how much its constraints are violated:

Definition 4 (Penalty). *Let $P = \langle V, D, C \rangle$ be a (set-)CSP and let K be the set of all configurations for P. A* penalty function *of a constraint $c \in C$ is a function $penalty(c) : K \to \mathbb{N}$ such that $penalty(c)(k) = 0$ if and only if c is satisfied under configuration k. The* penalty *of a constraint $c \in C$ with respect to a configuration $k \in K$ is $penalty(c)(k)$. The* penalty *of P with respect to a configuration $k \in K$ is the sum $\sum_{c \in C} penalty(c)(k)$.*

Example 3. Consider once again P from Example 1 and let c_1 and c_2 be the constraints $S_1 \subseteq S_2$ and $d_3 \in S_3$ respectively. Let the penalty functions of c_1 and c_2 be defined by $penalty(c_1)(k) = |k(S_1) \setminus k(S_2)|$ and $penalty(c_2)(k) = 0$ if $d_3 \in k(S_3)$ and 1 otherwise. Now, the penalties of P with respect to the configurations k_1 and k_2 from Example 2 are $penalty(c_1)(k_1) + penalty(c_2)(k_1) = 1$ and $penalty(c_1)(k_2) + penalty(c_2)(k_2) = 0$, respectively.

3 Second-Order Logic

We use *existential second-order logic* ($\exists$SOL) [8], extended with counting, for modelling the constraints of a set-CSP. $\exists$SOL is very expressive: it captures the complexity class NP [5]. Figure 1 shows the BNF grammar for the used language, which we will refer to as $\exists\text{SOL}^+$. Some of the production rules are highlighted and the reason for this is explained below. The language uses common mathematical and logical notations. Note that its set of relational operators is closed under negation. A formula in $\exists\text{SOL}^+$ is of the form $\exists S_1 \cdots \exists S_n \phi$, i.e., a sequence of existentially quantified set variables, ranging over the power set of an implicit common universe $\mathcal{U}$, and constrained by a logical formula ϕ. The usual precedence rules apply when parentheses are omitted, i.e., $\neg$ has highest precedence, $\wedge$ has higher precedence than $\vee$, etc.

Example 4. The constraint $S \subset T$ on the set variables S and T may be expressed in $\exists\text{SOL}^+$ by the formula:

$$\exists S \exists T((\forall x(x \notin S \vee x \in T)) \wedge (\exists x(x \in T \wedge x \notin S))) \tag{1}$$

The constraint $|S \cap T| \leq m$ on the set variables S and T and the natural-number constant m may be expressed in $\exists\text{SOL}^+$ by the formula:

$$\exists S \exists T \exists I((\forall x(x \in I \leftrightarrow x \in S \wedge x \in T)) \wedge |I| \leq m) \tag{2}$$

Note that we used an additional set variable I to represent the intersection $S \cap T$.

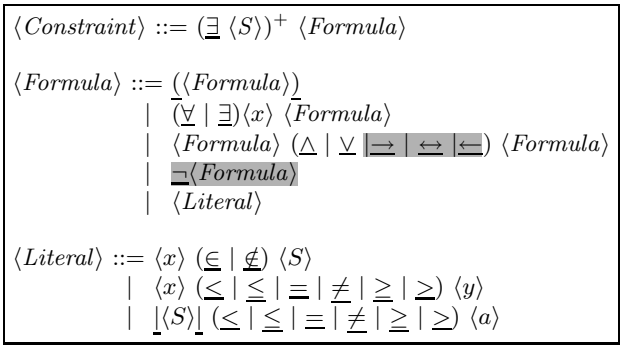

Fig. 1. The BNF grammar for the language $\exists\text{SOL}^+$ where terminal symbols are underlined. The non-terminal symbol $\langle S\rangle$ denotes an identifier for a bound set variable S such that $S \subseteq \mathcal{U}$, while $\langle x\rangle$ and $\langle y\rangle$ denote identifiers for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

In Section 4 we will define the penalty of formulas in $\exists\text{SOL}^+$. Before we do this, we define a core subset of this language that will be used in that definition. This is only due to the way we define the penalty and does not pose any limitations on the expressiveness of the language: Any formula in $\exists\text{SOL}^+$ may be transformed into a formula in that core subset, in a way shown next.

The transformations are standard and are only described briefly. First, given a formula $\exists S_1 \cdots \exists S_n \phi$ in $\exists\text{SOL}^+$, we remove its negations by pushing them downward, all the way to the literals of ϕ, which are replaced by their negated counterparts. Assuming that ϕ is the formula $\forall x(\neg(x \in S \wedge x \notin S'))$, it is transformed into $\forall x(x \notin S \vee x \in S')$. This is possible because the set of relational operators in $\exists\text{SOL}^+$ is closed under negation. Second, equivalences are transformed into conjunctions of implications, which are in turn transformed into disjunctions. Assuming that ϕ is the formula $\forall x(x \in S_1 \leftrightarrow x \in S_2)$, it is transformed into $\forall x((x \notin S_1 \vee x \in S_2) \wedge (x \in S_1 \vee x \notin S_2))$.

By performing these transformations for ϕ (and recursively for the subformulas of ϕ) in any formula $\exists S_1 \cdots \exists S_n \phi$, we end up with the non-highlighted subset of the language in Figure 1, for which we will define the penalty.

Example 5. (1) is in the core subset of $\exists\text{SOL}^+$. The core equivalent of (2) is:

$$\exists S \exists T \exists I((\forall x((x \notin I \vee x \in S \wedge x \in T) \wedge (x \in I \vee x \notin S \vee x \notin T))) \wedge |I| \leq m) \quad (3)$$

From now on we assume that any formula said in $\exists\text{SOL}^+$ is already in the core subset of $\exists\text{SOL}^+$. The full language just offers convenient shorthand notations.

4 The Penalty of an $\exists\text{SOL}^+$ Formula

In order to use (closed) formulas in $\exists\text{SOL}^+$ as constraints in our local search framework, we must define the penalty function of such a formula according

to Definition 4, which is done inductively below. It is important to stress that *this calculation is totally generic and automatable, as it is based only on the syntax of the formula and the semantics of the quantifiers, connectives, and relational operators of the* $\exists\mathrm{SOL}^+$ *language, but not on the intended semantics of the formula. A human might well give a different penalty function to that formula, and a way of calculating it that better exploits globality, but the scheme below requires* no *such user participation.*

We need to express the penalty with respect to the values of any bound first-order variables. We will therefore pass around an (initially empty) environment Γ in the definition below, where Γ is a total function from the currently bound first-order variables into the common universe of values.

Definition 5 (Penalty of an $\exists\mathrm{SOL}^+$ Formula). *Let $\mathcal{F}$ be a formula in $\exists\mathrm{SOL}^+$ of the form $\exists S_1 \cdots \exists S_n \phi$, let k be a configuration for $\{S_1, \ldots, S_n\}$, and let Γ be an environment. The* penalty *of $\mathcal{F}$ with respect to k and Γ is given by a function penalty$'$ defined by:*

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

Now, the penalty function *of $\mathcal{F}$ is the function $\mathit{penalty}(\mathcal{F}) = \mathit{penalty}'(\emptyset)(\mathcal{F})$.*

In the definition above, for (sub)formulas of the form $x \Diamond y$ and $|S| \Diamond c$, where $\Diamond \in \{<, \leq, =, \neq, \geq, >\}$, we only show the cases where $\Diamond \in \{\leq\}$; the other cases are defined similarly. (The same applies to the algorithms in Section 5.) The following proposition is a direct consequence of the definition above:

Proposition 1. *The penalty of a formula $\mathcal{F}$ with respect to a configuration k is 0 if and only if $\mathcal{F}$ is satisfied under k: $\mathit{penalty}(\exists S_1 \cdots \exists S_n \phi)(k) = 0 \Leftrightarrow k \models \phi$.*

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

Example 6. Let $\mathcal{U} = \{a, b\}$ and let k be the configuration for $\{S, T\}$ such that $k(S) = k(T) = \{a\}$. Let us calculate $\mathit{penalty}(\exists S \exists T \phi)(k)$, where $\exists S \exists T \phi$ is

the formula (1) The initial call matches case (a) which gives the recursive call $\mathit{penalty}'(\emptyset)(\phi)(k)$. Since ϕ is of the form $\psi \wedge \psi'$ this call matches case (d), which is defined as the sum of the recursive calls on ψ and ψ'. For the first recursive call, ψ is the formula $\forall x(x \notin S \vee x \in T)$. Hence it will match case (b), which is defined as the sum of the recursive calls $\mathit{penalty}'(\{x \mapsto a\})(x \notin S \vee x \in T)(k)$ and $\mathit{penalty}'(\{x \mapsto b\})(x \notin S \vee x \in T)(k)$ (one for each of the values a and b in $\mathcal{U}$). Both of these match case (e) which, for the first one, gives the minimum of the recursive calls $\mathit{penalty}'(\{x \mapsto a\})(x \notin S)(k)$ and $\mathit{penalty}'(\{x \mapsto a\})(x \in T)(k)$. This value is $\min\{1, 0\} = 0$ since $a \in T$. A similar reasoning for the second one gives the value $\min\{0, 1\} = 0$ as well since $b \notin S$. Hence the recursive call on ψ gives $0 + 0 = 0$. This means that ψ is satisfied and should indeed contribute nothing to the overall penalty. A similar reasoning for the recursive call on ψ', which is $\exists x(x \in T \wedge x \notin S)$, gives $\min\{1, 1\} = 1$. This means that ψ' is violated: the calculated contribution of 1 to the overall penalty means that no value of $\mathcal{U}$ belongs to T but not to S. Hence the returned overall penalty is $0 + 1 = 1$.

5 Incremental Penalty Maintenance using Penalty Trees

In our local search framework, given a formula $\mathcal{F}$ in $\exists\mathrm{SOL}^+$, we could use Definition 5 to calculate the penalty of $\mathcal{F}$ with respect to a configuration k, and then similarly for each configuration k' in a neighbourhood $\mathcal{N}(k)$ to be evaluated. However, a complete recalculation of the penalty with respect to Definition 5 is impractical, since $\mathcal{N}(k)$ is usually a very large set.

In local search it is crucial to use *incremental algorithms* when evaluating the penalty of a constraint with respect to a neighbour k' to a current configuration k. We will now present a scheme for incremental maintenance of the penalty of a formula in $\exists\mathrm{SOL}^+$ with respect to Definition 5. This scheme is based on viewing a formula $\mathcal{F}$ in $\exists\mathrm{SOL}^+$ as a syntax tree and observing that, given the penalty with respect to k, only the paths from the leaves that contain variables that are changed in k' compared to k to the root node need to be updated to obtain the penalty with respect to k'.

5.1 The Penalty Tree of a Formula

First, a syntax tree $\mathbf{T}$ of a formula $\mathcal{F}$ in $\exists\mathrm{SOL}^+$ of the form $\exists S_1 \cdots \exists S_n \phi$ is constructed in the usual way. Literals in $\mathcal{F}$ of the form $x \in S$, $x \notin S$, $x \Diamond y$, and $|S| \Diamond k$ (where $\Diamond \in \{<, \leq, =, \neq, \geq, >\}$) are leaves in $\mathbf{T}$. Subformulas in $\mathcal{F}$ of the form $\psi \Box \psi'$ (where $\Box \in \{\wedge, \vee\}$) are subtrees in $\mathbf{T}$ with $\Box$ as parent node and the trees of ψ and ψ' as children. When possible, formulas of the form $\psi_1 \Box \cdots \Box \psi_m$ give rise to one parent node with m children. Subformulas in $\mathcal{F}$ of the form $\forall x\psi$ (resp. $\exists x\psi$) are subtrees in $\mathbf{T}$ with $\forall x$ (resp. $\exists x$) as parent node and the tree of ψ as only child. Finally, $\exists S_1 \cdots \exists S_n$ is the root node of $\mathbf{T}$ with the tree of ϕ as child. As an example of this, Figure 2 shows the syntax tree of formula (3). Note that it contains additional information, to be explained in Section 5.2.

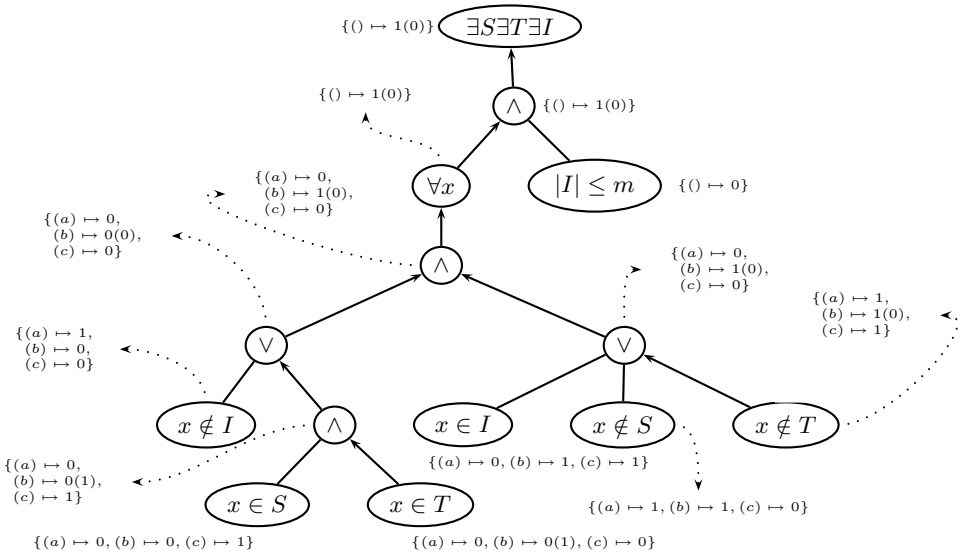

Fig. 2. Penalty tree of formula (3).

Assume that $\mathbf{T}$ is the syntax tree of a formula $\mathcal{F} = \exists S_1 \cdots \exists S_n \phi$. We will now extend $\mathbf{T}$ into a penalty tree in order to obtain incremental penalty maintenance of $\mathcal{F}$. Given an initial configuration k for $\{S_1, \ldots, S_n\}$, the penalty with respect to k of the subformula that the tree rooted at node $\mathbf{n}$ represents is stored in each node $\mathbf{n}$ of $\mathbf{T}$. This implies that the penalty stored in the root node of $\mathbf{T}$ is equal to $\mathit{penalty}(\mathcal{F})(k)$. When a configuration k' in the neighbourhood of k is to be evaluated, the only paths in $\mathbf{T}$ that may have changed are those leading from leaves containing any of the set variables S_i that are affected by the change of k to k'. By starting at each of these leaves $l(S_i)$ and updating the penalty with respect to the change of S_i of each node on the path from l to the root node of $\mathbf{T}$, we can incrementally calculate $\mathit{penalty}(\mathcal{F})(k')$ given k.

5.2 Initialising the Nodes with Penalties

For the descendants of nodes representing subformulas that introduce bound variables, we must store the penalty with respect to *every* possible mapping of those variables. For example, the child node $\mathbf{n}$ of a node for a subformula of the form $\forall x \phi$ will have a penalty stored for each $u \in \mathcal{U}$. Generally, the penalty stored at a node $\mathbf{n}$ is a mapping, denoted $p(\mathbf{n})$, from the possible tuples of values of the bound variables at $\mathbf{n}$ to $\mathbb{N}$. Assume, for example, that at $\mathbf{n}$ there are two bound variables x and y (introduced in that order) and that $\mathcal{U} = \{a, b\}$. Then the penalty stored at $\mathbf{n}$ after initialisation will be the mapping $\{(a, a) \mapsto p_1, (a, b) \mapsto p_2, (b, a) \mapsto p_3, (b, b) \mapsto p_4\}$ where $\{p_1, p_2, p_3, p_4\} \subset \mathbb{N}$. The first element of each tuple corresponds to x and the second one to y. If there are no bound variables at a particular node, then the penalty is a mapping $\{() \mapsto q\}$, i.e., the empty tuple mapped to some $q \in \mathbb{N}$.

Algorithm 1 Initialises the penalty mappings of a penalty tree.

function $\mathit{initialise}(\mathbf{T}, \Gamma, \mathcal{U}, k)$
  match T with
    $\exists S_1 \cdots \exists S_n \phi \longrightarrow p(\mathbf{T}) \leftarrow \{\mathit{tuple}(\Gamma) \mapsto \mathit{initialise}(\phi, \Gamma, \mathcal{U}, k)\}$
  $\mid \forall x \phi \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \{\mathit{tuple}(\Gamma) \mapsto \sum_{u \in \mathcal{U}} \mathit{initialise}(\phi, \Gamma \cup \{x \mapsto u\}, \mathcal{U}, k)\}$
  $\mid \exists x \phi \longrightarrow$
      $p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \{\mathit{tuple}(\Gamma) \mapsto \min\{\mathit{initialise}(\phi, \Gamma \cup \{x \mapsto u\}, \mathcal{U}, k) \mid u \in \mathcal{U}\}\}$
  $\mid \phi_1 \wedge \cdots \wedge \phi_m \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \{\mathit{tuple}(\Gamma) \mapsto \sum_{1 \leq i \leq m} \mathit{initialise}(\phi_i, \Gamma, \mathcal{U}, k)\}$
  $\mid \phi_1 \vee \cdots \vee \phi_m \longrightarrow$
      $p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \{\mathit{tuple}(\Gamma) \mapsto \min\{\mathit{initialise}(\phi, \Gamma, \mathcal{U}, k) \mid \phi \in \{\phi_1, \ldots, \phi_m\}\}\}$

  $\mid x \leq y \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \left\{ \mathit{tuple}(\Gamma) \mapsto \begin{cases} 0, & \text{if } \Gamma(x) \leq \Gamma(y) \\ 1, & \text{otherwise} \end{cases} \right\}$

  $\mid |S| \leq m \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \left\{ \mathit{tuple}(\Gamma) \mapsto \begin{cases} 0, & \text{if } |k(S)| \leq m \\ |k(S)| - m, & \text{otherwise} \end{cases} \right\}$

  $\mid x \in S \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \left\{ \mathit{tuple}(\Gamma) \mapsto \begin{cases} 0, & \text{if } \Gamma(x) \in k(S) \\ 1, & \text{otherwise} \end{cases} \right\}$

  $\mid x \notin S \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \left\{ \mathit{tuple}(\Gamma) \mapsto \begin{cases} 0, & \text{if } \Gamma(x) \notin k(S) \\ 1, & \text{otherwise} \end{cases} \right\}$
  end match
  return $p(\mathbf{T})(\mathit{tuple}(\Gamma))$
function $\mathit{tuple}(\Gamma)$
  return $(\Gamma(x_1), \ldots, \Gamma(x_n))$ ▷ $\{x_1, \ldots, x_n\} = \mathit{domain}(\Gamma)$, introduced into Γ in that order.

Algorithm 1 shows the function $\mathit{initialise}(\mathbf{T}, \Gamma, \mathcal{U}, k)$ that initialises a penalty tree $\mathbf{T}$ of a formula with penalty mappings with respect to an (initially empty) environment Γ, a universe $\mathcal{U}$, and a configuration k. By abuse of notation, we let formulas in $\exists\mathrm{SOL}^+$ denote their corresponding penalty trees, e.g., $\forall x \phi$ denotes the penalty tree with $\forall x$ as root node and the tree representing ϕ as only child, $\phi_1 \wedge \cdots \wedge \phi_m$ denotes the penalty tree with $\wedge$ as root node and the subtrees of all the ϕ_i as children, etc. Note that we use an auxiliary function *tuple* that, given an environment Γ, returns the tuple of values with respect to Γ. We also assume that before *initialise* is called for a penalty tree $\mathbf{T}$, the penalty mapping of each node in $\mathbf{T}$ is the empty set.

Example 7. Let $k = \{S \mapsto \{a, b\}, T \mapsto \{a, b, c\}, I \mapsto \{a\}\}$, let $\mathcal{U} = \{a, b, c\}$, and let $m = 1$. Figure 2 shows the penalty tree $\mathbf{T}$ with penalty mappings (dotted arrows connect nodes to their mappings) after $\mathit{initialise}(\mathbf{T}, \emptyset, \mathcal{U}, k)$ has been called for formula (3). As can be seen at the root node, the initial penalty is 1. Indeed, there is *one* value, namely b, that is in S and T but not in I.

5.3 Maintaining the Penalties

We will now present a way of incrementally updating the penalty mappings of a penalty tree. This is based on the observation that, given an initialised penalty tree $\mathbf{T}$, a current configuration k, and a configuration to evaluate k', only the paths leading from any leaf in $\mathbf{T}$ affected by changing k to k' to the root node of $\mathbf{T}$ need to be updated.

Algorithm 2 shows the function $\mathit{submit}(\mathbf{n}, \mathbf{n}', \mathcal{A}, k, k')$ that updates the penalty mappings of a penalty tree incrementally. It is a recursive function where infor-

Algorithm 2 Updates the penalty mappings of a penalty tree.

function $submit(\mathbf{n}, \mathbf{n}', \mathcal{A}, k, k')$
  $update(\mathbf{n}, \mathbf{n}', \mathcal{A})$ ▷ First update $\mathbf{n}$ with respect to $\mathbf{n}'$.
  if All children affected by the change of k to k' are done **then**
    if $\mathbf{n}$ is not the root node **then**
      $submit(parent(\mathbf{n}), \mathbf{n}, \mathcal{A} \cup changed(\mathbf{n}), k, k')$
      $changed(\mathbf{n}) \leftarrow \emptyset$
    else () ▷ We are at the root. Done!
  else $changed(\mathbf{n}) \leftarrow changed(\mathbf{n}) \cup \mathcal{A}$ ▷ Not all children done. Save tuples and wait.
function $update(\mathbf{n}, \mathbf{n}', \mathcal{A})$
  $p'(\mathbf{n}) \leftarrow p(\mathbf{n})$ ▷ Save the old penalty mapping.
  for all $t \in \mathcal{A}|_{bounds(\mathbf{n})}$ **do**
    match n with
      $\exists S_1 \cdots \exists S_n \phi \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{() \mapsto p(\mathbf{n}')(())\}$
    | $\forall x \phi \longrightarrow$
        for all $t' \in \mathcal{A}|_{bounds(\mathbf{n}')}$ s.t. $t'|_{bounds(\mathbf{n})} = t$ **do**
          $p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{t \mapsto p(\mathbf{n})(t) + p(\mathbf{n}')(t') - p'(\mathbf{n}')(t')\}$
    | $\exists x \phi \longrightarrow$
        for all $t' \in \mathcal{A}|_{bounds(\mathbf{n}')}$ s.t. $t'|_{bounds(\mathbf{n})} = t$ **do**
          Replace the value for t' in $min\_heap(\mathbf{n}, t)$ with $p(\mathbf{n}')(t')$
          $p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{t \mapsto \min(min\_heap(\mathbf{n}, t))\}$
    | $\phi_1 \wedge \cdots \wedge \phi_m \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{t \mapsto p(\mathbf{n})(t) + p(\mathbf{n}')(t) - p'(\mathbf{n}')(t)\}$
    | $\phi_1 \vee \cdots \vee \phi_m \longrightarrow$ Replace the value for $\mathbf{n}'$ in $min\_heap(\mathbf{n}, t)$ with $p(\mathbf{n}')(t)$
        $p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{t \mapsto \min(min\_heap(\mathbf{n}, t))\}$
    | $x \leq y \longrightarrow error$ ▷ Only leaves representing formulas on set variables apply!

    $$| \; |S| \leq m \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \left\{ t \mapsto \begin{cases} 0, & \text{if } |k'(S)| \leq m \\ |k'(S)| - m, & \text{otherwise} \end{cases} \right\}$$

    $$| \; x \in S \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \left\{ t \mapsto \begin{cases} 0, & \text{if } t(x) \in k'(S) \\ 1, & \text{otherwise} \end{cases} \right\}$$

    $$| \; x \notin S \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \left\{ t \mapsto \begin{cases} 0, & \text{if } t(x) \notin k'(S) \\ 1, & \text{otherwise} \end{cases} \right\}$$

    end match

mation from the node $\mathbf{n}'$ (*void* when $\mathbf{n}$ is a leaf) is propagated to the node $\mathbf{n}$. The additional arguments are $\mathcal{A}$ (a set of tuples of values that are affected by changing k to k' at $\mathbf{n}$), k (the current configuration), and k' (the configuration to evaluate). It uses the auxiliary function $update(\mathbf{n}, \mathbf{n}', \mathcal{A})$ that performs the actual update of the penalty mappings of $\mathbf{n}$ with respect to (the change of the penalty mappings of) $\mathbf{n}'$.

The set $\mathcal{A}$ depends on the maximum number of bound variables in the penalty tree, the universe $\mathcal{U}$, and the configurations k and k'. Recall $\mathcal{U}$ and k of Example 7 and assume that $k' = \{S \mapsto \{a, b\}, T \mapsto \{a, c\}, I \mapsto \{a\}\}$ (b was removed from $k(T)$). In this case $\mathcal{A}$ would be the singleton set $\{(b)\}$ since this is the only tuple affected by the change of k to k'. However, if the maximum number of bound variables was two (instead of one as in Example 7), $\mathcal{A}$ would be the set $\{(b, a), (b, b), (b, c), (a, b), (c, b)\}$ since all of these tuples might be affected.

Some of the notation used in Algorithm 2 needs explanation: Given a set $\mathcal{A}$ of tuples, each of arity n, we use $\mathcal{A}|_m$ to denote the set of tuples in $\mathcal{A}$ projected on their first $m \leq n$ positions. For example, if $\mathcal{A} = \{(a, a), (a, b), (a, c), (b, a), (c, a)\}$, then $\mathcal{A}|_1 = \{(a), (b), (c)\}$. We use a similar notation for projecting a particular tuple: if $t = (a, b, c)$ then $t|_2$ denotes the tuple (a, b). We also use $t(x)$ to denote the value of the position of x in t. For example, if x was the second introduced

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

Now, given a change to a current configuration k, resulting in k', assume that $\{S_i\}$ is the set of affected set variables in a formula $\mathcal{F}$ with an initialised penalty tree $\mathbf{T}$. The call *submit*($\mathbf{n}$, *void*, $\mathcal{A}, k, k'$) must now be made for each leaf $\mathbf{n}$ of $\mathbf{T}$ that represents a subformula stated on S_i, where $\mathcal{A}$ is the set of affected tuples.

Example 8. Recall $k = \{S \mapsto \{a, b\}, T \mapsto \{a, b, c\}, I \mapsto \{a\}\}$ and $m = 1$ of Example 7, and keep the initialised tree $\mathbf{T}$ in Figure 2 in mind. Let $k' = \{S \mapsto \{a, b\}, T \mapsto \{a, c\}, I \mapsto \{a\}\}$, i.e., b was removed from $k(T)$. The function *submit* will now be called twice, once for each leaf in $\mathbf{T}$ containing T.

Starting with the leaf $\mathbf{n}_{11}$ representing the formula $x \in T$, *submit* is called with *submit*($\mathbf{n}_{11}$, *void*, $\{(b)\}, k, k'$). This gives the call *update*($\mathbf{n}_{11}$, *void*, $\{(b)\}$) which replaces the binding of (b) in $p(\mathbf{n}_{11})$ with $(b) \mapsto 1$ (since b is no longer in T). Since a leaf node has no children and $\mathbf{n}_{11}$ is not the root node, *submit*($\mathbf{n}_{12}$, $\mathbf{n}_{11}$, $\{(b)\}$, k, k') is called where $\mathbf{n}_{12} = \mathit{parent}(\mathbf{n}_{11})$. Since $\mathbf{n}_{12}$ is an $\wedge$-node, *update*($\mathbf{n}_{12}, \mathbf{n}_{11}, \{b\}$) implies that the binding of (b) in $p(\mathbf{n}_{12})$ is updated with the difference $p(\mathbf{n}_{11}) - p'(\mathbf{n}_{11})$ (which is 1 in this case). Hence, the new value of $p(\mathbf{n}_{12})(b)$ is 1. Since there are no other affected children of $\mathbf{n}_{12}$ and $\mathbf{n}_{12}$ is not the root node, *submit*($\mathbf{n}_{13}, \mathbf{n}_{12}, \{(b)\}, k, k'$) is called where $\mathbf{n}_{13} = \mathit{parent}(\mathbf{n}_{12})$. Since $\mathbf{n}_{13}$ is an $\vee$-node, *update*($\mathbf{n}_{13}, \mathbf{n}_{12}, \{b\}$) gives that the binding of (b) in $p(\mathbf{n}_{13})$ is updated with the minimum of $p(\mathbf{n}_{12})(b)$ and the values of $p(\mathbf{n})(b)$ for any other child $\mathbf{n}$ of $\mathbf{n}_{13}$. Since the only other child of $\mathbf{n}_{13}$ gives a 0 for this value, $p(\mathbf{n}_{13})(b)$ remains 0. Now, call *submit*($\mathbf{n}_3, \mathbf{n}_{13}, \{(b)\}, k, k'$) where $\mathbf{n}_3 = \mathit{parent}(\mathbf{n}_{13})$. The call *update*($\mathbf{n}_3, \mathbf{n}_{13}, \{b\}$) gives that $p(\mathbf{n}_3)(b)$ is unchanged (since $p(\mathbf{n}_{13})(b)$ was unchanged). Now, not all possibly affected children of $\mathbf{n}_3$ are done since the leaf $\mathbf{n}_{21}$ representing the formula $x \notin T$ has not yet been propagated. By following a similar reasoning for the nodes $\mathbf{n}_{21}$ and $\mathbf{n}_{22} = \mathit{parent}(\mathbf{n}_{21})$ we will see that the value of $p(\mathbf{n}_{22})(b)$ changes from 1 to 0 (since b is now in T). When this is propagated to $\mathbf{n}_3$ by *submit*($\mathbf{n}_3, \mathbf{n}_{22}, \{(b)\}, k, k'$), the value of $p(\mathbf{n}_3)(b)$ will also change from 1 to 0. A similar reasoning for $\mathit{parent}(\mathbf{n}_3)$, $\mathit{parent}(\mathit{parent}(\mathbf{n}_3))$ and the root node gives the same changes to their penalty mappings consisting of only $() \mapsto 1$. This will lead to an overall penalty decrease of 1 and hence, the penalty of formula (3) with respect to k' is 0, meaning that (3) is satisfied under k'. The values of the changed penalty mappings with respect to k' of $\mathbf{T}$ are shown in parentheses in Figure 2.

6 Neighbourhood Selection

When solving a problem with local search, it is often crucial to restrict the initial configuration and the neighbourhood function used so that not all the constraints need to be stated explicitly. It is sometimes hard by local search alone to satisfy a constraint that can easily be guaranteed by using a restricted initial configuration and neighbourhood function. For example, if a set must have a fixed cardinality, then, by defining an initial configuration that respects this and by using a neighbourhood function that keeps the cardinality constant (for example by swapping values in the set with values in its complement), an explicit cardinality constraint need not be stated. Neighbourhoods are often designed in such an ad-hoc fashion. With the framework of $\exists\text{SOL}^+$, it becomes possible to reason about neighbourhoods and invariants:

Definition 6. *Let formula ϕ model a CSP P, let K be the set of all configurations for P, and let formula ψ be such that $k \models \phi$ implies $k \models \psi$ for all configurations $k \in K$. A neighbourhood function $\mathcal{N} : K \to 2^K$ is* invariant *for ψ if $k \models \psi$ implies $k' \models \psi$ for all $k' \in \mathcal{N}(k)$.*

Intuitively, the formula ψ is implied by ϕ and all possible moves take a configuration satisfying ψ to another configuration satisfying ψ. The challenge then is to find a suitable neighbourhood function for a formula ϕ.

Sometimes (as we will see in Section 7), given formulas ϕ and ψ satisfying Definition 6, it is possible to find a formula δ such that ϕ is logically equivalent to $\delta \wedge \psi$. If the formula δ is smaller than ϕ, then the speed of the local search algorithm can be greatly increased since the incremental penalty maintenance is faster on smaller penalty trees.

7 Application: A Financial Portfolio Problem

After formulating a financial portfolio optimisation problem, we show how to exactly solve real-life instances thereof in our local search framework. This is impossible with the best-known complete search algorithm and competitive with a fast approximation method based on complete search.

7.1 Formulation

The synthetic-CDO-Squared portfolio optimisation problem in financial mathematics has practical applications in the credit derivatives market [7]. Abstracting the finance away and assuming (not unrealistically) interchangeability of all the involved credits, it can be formulated as follows.[1] Let $V = \{1, \ldots, v\}$ and let $B = \{1, \ldots, b\}$ be a set of credits. An *optimal portfolio* is a set of v subsets $B_i \subseteq B$, called *baskets*, each of size r (with $0 \leq r \leq b$), such that the maximum intersection size of any two distinct baskets is minimised.

[1] We use the notation of the related balanced incomplete block design problem.

	c r e d i t s							
basket 1	1	1	1	0	0	0	0	0
basket 2	1	1	0	1	0	0	0	0
basket 3	1	1	0	0	1	0	0	0
basket 4	1	1	0	0	0	1	0	0
basket 5	0	0	1	1	1	0	0	0
basket 6	0	0	1	1	0	1	0	0
basket 7	0	0	1	1	0	0	1	0
basket 8	0	0	0	0	1	1	0	1
basket 9	0	0	0	0	1	0	1	1
basket 10	0	0	0	0	0	1	1	1

Table 1. An optimal solution to $\langle 10, 8, 3, \lambda \rangle$, with $\lambda = 2$.

There is a universe of about $250 \leq b \leq 500$ credits. A typical portfolio contains about $4 \leq v \leq 25$ baskets, each of size $r \approx 100$. Such real-life instances of the portfolio *optimisation* problem are hard, so we transform it into a CSP by also providing a targeted value, denoted λ (with $\lambda < r$), for the maximum of the pairwise basket intersection sizes in a portfolio. Hence the following formulation of the problem:

$$\forall\ i \in V : |B_i| = r \tag{4}$$

$$\forall\ i_1 \neq i_2 \in V : |B_{i_1} \cap B_{i_2}| \leq \lambda \tag{5}$$

We parameterise the portfolio CSP by a 4-tuple $\langle v, b, r, \lambda \rangle$ of independent parameters. The following formula gives an optimal lower bound on λ [13]:[2]

$$\lambda \geq \frac{\lceil \frac{rv}{b} \rceil^2 (rv \bmod b) + \lfloor \frac{rv}{b} \rfloor^2 (b - rv \bmod b) - rv}{v(v-1)} \tag{6}$$

7.2 Using Complete Search

One way of modelling a portfolio is in terms of its *incidence matrix*, which is a $v \times b$ matrix, such that the entry at the intersection of row i and column j is 1 if $j \in B_i$ and 0 otherwise. The constraints (4) and (5) are then modelled by requiring, respectively, that there are exactly r ones (that is a sum of r) for each row and a scalar product of at most λ for any pair of distinct rows. An optimal solution, under this model, to $\langle 10, 8, 3, \lambda \rangle$ is given in Table 1, with $\lambda = 2$.

The baskets are indistinguishable, and, as stated above, we assume that all the credits are indistinguishable. Hence any two rows or columns of the incidence matrix can be freely permuted. Breaking all the resulting $v! \cdot b!$ symmetries can in theory be performed, for instance by $v! \cdot b! - 1$ (anti-)lexicographical ordering constraints [4]. In practice, strictly anti-lexicographically ordering the rows (since baskets cannot be repeated in portfolios) as well as anti-lexicographically

[2] It often improves the bound reported in [7] and negatively settles the open question therein whether the $\langle 10, 350, 100, 21 \rangle$ portfolio exists or not.

ordering the columns (since credits can appear in the same baskets) works quite fine for values of b up to about 36, due to the constraint (5), especially when labelling in a row-wise fashion and trying the value 1 before the value 0. However, this is one order of magnitude below the typical value for b in a portfolio. In [7], we presented an approximate and often extremely fast method of solving real-life instances of this problem by complete search, even for values of λ quite close, if not identical, to the lower bound in (6). It is based on embedding (multiple copies of) independent sub-instances into the original instance. Their determination is itself a CSP, based on (6).

7.3 Using Local Search

It is easy to model the portfolio problem in $\exists\text{SOL}^+$ using additional set variables. The problem can be modelled by the following formula:

$$\exists B_1, \ldots, \exists B_v \exists_{i<j} I_{(i,j)} \ \phi_1 \wedge \phi_2 \wedge \phi_3 \tag{7}$$

where $\exists_{i<j} I_{(i,j)}$ is a shorthand for the sequence of quantifications $\exists I_{(1,2)}, \ldots, I_{(i,j)}$, ... for all $i < j$.[3] The formula $\phi_1 = |B_1| = r \wedge \cdots \wedge |B_v| = r$ states that each set B_i is of size r. Using similar conventions, the formula $\phi_2 = \forall i < j \ \forall x (x \in I_{(i,j)} \leftrightarrow (x \in B_i \wedge x \in B_j))$ states that each set $I_{(i,j)}$ is the intersection of B_i and B_j. Finally, the formula $\phi_3 = \forall i < j | I_{(i,j)}| \leq \lambda$ states that the intersection size of any B_i and B_j should be less than or equal to λ.

The local search algorithm can be made more efficient by using the ideas in Section 6. First, we define a neighbourhood function that is invariant for the formula ϕ_1. Assuming that the initial configuration for (7) respects ϕ_1, the neighbourhood function that swaps any value in any B_i to any value in its complement is invariant for ϕ_1. We denote this neighbourhood function by *exchange*. We may even extend *exchange* such that it is invariant also for ϕ_2. In order to do this, we assume that the initial configuration for (7) respects $\phi_1 \wedge \phi_2$. Now, we extend *exchange* in the following way. Given a configuration k and a configuration k' in *exchange*(k) where B_i is the only variable affected by the change of k to k', the variables $I_{(i,j)}$ such that there exists a subformula $x \in I_{(i,j)} \leftrightarrow (x \in B_i \wedge x \in B_j)$ or $x \in I_{(j,i)} \leftrightarrow (x \in B_j \wedge x \in B_i)$ are all updated (by adding or removing a value to $I_{(i,j)}$) so that those formulas still hold.

We use a similar algorithm to the one in [2] for solving the portfolio problem with local search, i.e., a Tabu-search algorithm with a restarting criterion if no overall improvement was reported after a certain number of iterations.

7.4 Results

The experiments were run on an Intel 2.4 GHz Linux machine with 512 MB memory. The local search framework was implemented in OCaml and the complete search algorithm was coded in SICStus Prolog.

[3] This shorthand is a purely conservative extension of $\exists\text{SOL}^+$ and does not increase the expressiveness.

The local search algorithm performs well on this problem. For example, the easy instance $\langle 10, 35, 11, 3\rangle$ is solved in 0.2 seconds, the slightly harder instance $\langle 10, 70, 22, 6\rangle$ in 0.6 seconds, and the real-life instance $\langle 15, 350, 100, 24\rangle$ in 133.9 seconds. Bear in mind that these results were achieved (by our current prototype implementation) under the assumption that no built-in constraints existed, and thus that the incremental penalty maintenance algorithms were automatically generated as described in this paper.

For comparison, the complete search approach without embeddings needs 0.6 seconds for finding a first solution of $\langle 10, 35, 11, 3\rangle$, 929.8 seconds for $\langle 10, 70, 22, 6\rangle$, and does not terminate within several hours of CPU time for $\langle 15, 350, 100, 24\rangle$.

Using the extended implementation [13] of the embedding method of [7] for the real-life instance $\langle 15, 350, 100, 24\rangle$, two embeddings were constructed but both timed out after 100 seconds. Hence, local search approaches can outperform even this approximation method.

8 Conclusion

Summary. In the context of local search, we have introduced a scheme that, from a high-level problem model in existential second-order logic with counting ($\exists$SOL$^+$), automatically synthesises incremental penalty calculation algorithms. This bears significant benefits when ad hoc constraints are necessary for a particular problem, as no adaptation by the user of the modelling part of the local search system is then required. The performance of the scheme has been demonstrated by solving real-life instances of a financial portfolio design problem that seem unsolvable in reasonable time by complete search.

Related Work. The usage of existential second-order logic ($\exists$SOL) as a modelling language has also been advocated in [9]. The motivation there was rather that any automated reasoning about constraint models must *necessarily* first be studied on this simple core language before moving on to extensions thereof. Modern, declarative constraint modelling languages, such as NP-SPEC [3], OPL [14], and ESRA [6], are extensions of $\exists$SOL. In contrast, our motivation for $\exists$SOL is that it is a *sufficient* language for our purpose, especially if extended (only) with counting.

The adaptation of the traditional combinators of constraint programming for local search was pioneered in [15]. The combinators there include logical connectives (such as $\wedge$ and $\vee$), cardinality operators (such as *exactly* and *atmost*), reification, and expressions over variables. We extend these ideas here to the logical quantifiers ($\forall$ and $\exists$). This is not just a matter of simply generalising the arities and penalty calculations of the $\wedge$ and $\vee$ connectives, respectively, but made necessary by our handling of set variables over which one would like to iterate, unlike the scalar variables of [11, 15].

Future Work. We have made several simplifying assumptions in order to restrict this paper to its fundamental ideas. For instance, the handling of both scalar variables and set variables requires special care in the calculation of penalties, and has been left as future work. Also, many more shorthand notations than

the ones used in this paper could be added for the user's convenience, such as quantification bounded over a set rather than the entire universe. Furthermore, it would be useful if appropriate neighbourhood functions that are invariant for some of the constraints could automatically be generated from an $\exists\text{SOL}^+$ model.
Conclusion. Our first computational results are encouraging and warrant further research into the automatic synthesis of local search algorithms.
Acknowledgements. This research was partially funded by Project C/1.246/ HQ/JC/04 of EuroControl. We thank Olof Sivertsson for his contributions to the experiments on the financial portfolio problem, as well as the referees for their useful comments.

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. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NPSPEC: An executable specification language for solving all problems in NP. In G. Gupta, editor, *Proceedings of PADL'99*, volume 1551 of *LNCS*, pages 16–30. Springer-Verlag, 1999.
4. J. M. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Proceedings of KR'96*, pages 148–159. Morgan Kaufmann, 1996.
5. R. Fagin. *Contributions to the Model Theory of Finite Structures*. PhD thesis, UC Berkeley, California, USA, 1973.
6. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In M. Bruynooghe, editor, *LOPSTR'03: Revised Selected Papers*, volume 3018 of *LNCS*, pages 214–232. Springer-Verlag, 2004.
7. P. Flener, J. Pearson, and L. G. Reyna. Financial portfolio optimisation. In M. Wallace, editor, *Proceedings of CP'04*, volume 3258 of *LNCS*, pages 227–241. Springer-Verlag, 2004.
8. N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
9. T. Mancini. *Declarative Constraint Modelling and Specification-Level Reasoning*. PhD thesis, Università degli Studi di Roma "La Sapienza", Italy, 2004.
10. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In G. Smolka, editor, *Proceedings of CP'97*, volume 1330 of *LNCS*, pages 237–251. Springer-Verlag, 1997.
11. 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*.
12. 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.
13. O. Sivertsson. Construction of synthetic CDO squared. Master's thesis, Computing Science, Department of Information Technology, Uppsala University, Sweden, December 2005. Available as Technical Report 2005-042 at `http://www.it.uu.se/research/reports/2005-042/`.

14. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
15. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-based combinators for local search. In M. Wallace, editor, *Proceedings of CP'04*, volume 3258 of *LNCS*, pages 47–61. Springer-Verlag, 2004.
16. J. P. Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *LNCS*. Springer-Verlag, 1999.