Constraint-based Problem Decomposition for a Key Configuration Problem

Ulrich Junker

ILOG
9, rue de Verdun, BP 85
F-94253 Gentilly Cedex
junker@ilog.fr

Abstract. Finding good problem decompositions is crucial for solving large-scale key/lock configuration problems. We present a novel approach to problem decomposition where the detection of a subproblem hierarchy is formulated as a constraint satisfaction problem (CSP) on set variables. Primitive constraints on set variables such as an inverse and a union-over-set-constraint are used to formalize properties of trees and to ensure that solutions of these CSPs represent a tree. An objective for optimizing properties of the tree is presented as well. Experimental results from an industrial prototype are reported.

Keywords: constraints on trees, problem decomposition, key configuration, set variables.

1 Introduction

Domain filtering techniques as elaborated in the field of CSPs have successfully been applied to solve combinatorial problems such as job-shop scheduling and car sequencing. An example is the famous edge-finder constraint that allows solving difficult job-shop examples to optimality. The following techniques are crucial for reducing an initially large search space to a smaller space that still contains all solutions or all optimal solutions:

1. Improved filtering by global constraints or good constraint models.
2. Heuristics defining the order in which choices are made.
3. Non-chronological search (e.g. best-first or limited discrepancy search).

The search space is reduced by cutting off inconsistent subtrees having no solution (1), by making inconsistent subtrees small by using first-fail heuristics (2), and by cutting subtrees if lower bounds on the solutions in the subtrees are strictly greater than the costs of the best solution found so far. Finding better solutions first (2,3) thus allows cutting the search space further.

A principal problem, however, is the scalability of these techniques, especially when we face real-world problems such as production scheduling, personnel planning, resource allocation, and others that are large in size. For example, if 1000

M. Maher and J.-F. Puget (Eds.): CP'98, LNCS 1520, pp. 265-279, 1998.

tasks have to be allocated to 1000 resources then a complete search to find an optimal solution is no longer practicable. In order to find acceptably good solutions, problem decomposition methods are usually applied in AI and OR. We distinguish three methods:

1. **Solution Synthesis:** a large problem is decomposed into subproblems such that each variable occurs in exactly one of the subproblems. If a constraint belongs to a subproblem then all of its variables belong to this subproblem. For each subproblem, several possible solutions are determined. The master problem then consists of (optimally) selecting one solution of each subproblem s.t. all remaining constraints are satisfied. These are these constraints that do not belong to any subproblem. This method is used, for example, to solve crew scheduling problems [1].
2. **Subproblem Improvement:** a current solution for the complete problem is improved in several steps. In each step, a subproblem is chosen and changes to the current solution are restricted to the variables in this subproblem.
3. **Subproblem Separation:** a large problem is decomposed into subproblems and a common problem, such that each variable appears in exactly one of these problems. There can be constraints between the variables of a subproblem and those of the common problem, but not between variables of two different subproblems. As a consequence, the subproblems can be solved separately after the common problem has been solved. Such an approach has been used, for example, in [2] for resource allocation problems.

In this paper, we study the last method for a particular example, namely the configuration of locking systems for large complexes such as airports, plants, and office buildings. A given locking matrix describes which key should open which lock (of which door). The task is to choose the physical properties of keys and locks (e.g. the tooth lengths) such that this locking matrix is respected. In order to treat large cases, the human planners decompose the problem and build up a hierarchy of subproblems.

In this paper, we address the central question of how to find a good decomposition into subproblems. It turned out that this task can be formulated as a CSP. Different properties of the tree of subproblems can be formalized in terms of constraints on set variables [7]. An objective on the form of the tree and the kind of subproblems can be introduced in the same manner.

The paper is organized as follows: The key configuration problem is presented in section 2. Section 3 gives a short survey on set variables. Analyzing properties of solutions of key configuration problems allows the introduction of a problem decomposition scheme (section 4). The CSP for the configuration of a tree is introduced in section 5 and extended to the subproblem hierarchy in 6. Section 7 describes an algorithm for finding such a hierarchy and section 8 reports results from an industrial prototype.

	l_1	l_2	l_3
k_1	X		X
k_2		X	X

k_1	k_2	l_1	l_2	l_3
$\{t_{1,5}, t_{2,6}\}$	$\{t_{1,7}, t_{2,4}\}$	$\{t_{1,5}, t_{2,6}\}$	$\{t_{1,7}, t_{2,4}\}$	$\{t_{1,5}, t_{1,7}, t_{2,6}, t_{2,4}\}$

Fig. 1. A locking matrix (left side) and a possible solution (right side).

2 The Key Configuration Problem

We briefly present the problem of the configuration of locking systems. Large complexes of buildings can have thousands of different doors that have their respective locks and keys. Obviously, the door of an office should not be opened by the keys of the other offices. Moreover, the entrances of the floors, buildings, and special-purpose rooms are opened by several keys, but not necessarily all. Access to several rooms is provided by master keys. Master keys can open several doors, but not necessarily all. A so-called locking matrix specifies which key opens which lock. An example is given in figure 1. Each row corresponds to a key and each column to a lock. A key opens a lock if there is a cross in the corresponding row and column. Mathematically, a locking matrix can be described by a binary relation between the set of keys and the set of locks.

A locking matrix is technically realized by choosing various physical properties for each key and each lock (e.g. the teeth and the holes of a key). For the sake of simplicity, we summarize all these properties by the term *pin*. A key opens a lock if the pins of the key are a subset of the pins of the lock. For example, a key has several teeth $t_{i,d}$ each of which has a depth d and a unique position i. At each these of positions, a lock has a pin that can be subdivided several times. If a key opens a lock then the depths of the teeth must correspond to the depths of the subdivisions. Key k_1 in figure 1 has a tooth $t_{1,5}$ of depth 5 at position 1 and a tooth $t_{2,6}$ of depth 6 at position 2. Key k_2 has the teeth $t_{1,7}$ and $t_{2,4}$. Since k_1 opens lock l_1 and k_2 opens l_2 those locks have the same pin sets. Since l_3 is opened by both keys its pin set is the union of the pin sets of k_1 and k_2. We now define a key configuration problem as follows:

Definition 1. *A* key configuration problem *is defined by a finite set $\mathcal{K}$ of keys, a finite set $\mathcal{L}$ of locks, a locking relation $\mathcal{O} \subseteq \mathcal{K} \times \mathcal{L}$, and a finite set $\mathcal{P}$ of pins. A* solution *to this problem consists of a pin set $P(k) \subseteq \mathcal{P}$ for each key $k \in \mathcal{K}$ and of a pin set $Q(l) \subseteq \mathcal{P}$ for each lock $l \in \mathcal{L}$ such that*

1. *$P(k) \subseteq Q(l)$ iff $(k, l) \in \mathcal{O}$ for each $k \in \mathcal{K}$ and $l \in \mathcal{L}$,*
2. *$P(k)$ is legal for each $k \in \mathcal{K}$,*
3. *$Q(l)$ is legal for each $l \in \mathcal{L}$.*

The pin sets $P(k)$ and $Q(l)$ are legal if they satisfy mechanical and security constraints that are manufacturer-specific. Some examples are:

1. *Security constraints:* the difference between the shortest and the longest tooth of a key is at least 3.

2. *Mechanical constraints:* a tooth of depth 0 cannot be between two teeth of depth 8.

A key configuration problem would not be difficult to solve if the number of pins were large. The difficulties stem from the fact that cases with 10000 keys and locks must be realized with less than 100 pins. Even if the number of keys and locks is small the number of used pins should be minimized. A pin is *used* iff it is contained in the pin set $P(k)$ of at least one key $k \in \mathcal{K}$ and if it is not contained in the pin set $Q(l)$ of at least one lock $l \in \mathcal{L}$. In order to find a solution having a minimal number of used pins, we decompose the problem as follows. In the first step, we ignore legality constraints and use anonymous pins.

1. Find an optimal solution that satisfies constraint (1) in def. 1.
2. Assign the anonymous pins to concrete pins by obeying legality constraints.

Whereas the first step leads to a general mathematical model, which is difficult to solve, the second problem requires manufacturer-specific strategies and knowledge, which are beyond the scope of this paper. We therefore restrict further discussion to the first problem.

3 CSPs on Finite Set Variables

The CSPs presented in this paper will be based on constrained set variables as introduced by [7, 3]. Given a finite (element) domain $\mathcal{D}$, the value of a *set variable* is a subset of the element domain $\mathcal{D}$. The domain of a set variable therefore is the power-set $2^{\mathcal{D}}$. Filtering techniques for set variables either add elements of $\mathcal{D}$ to all possible values of the set variable or remove elements from all possible values. The current domain of a set variable X is therefore represented by a set of *required elements* $req(X)$ (those that have been added) and a set of *possible elements* $pos(X)$ (those that have not been removed). A variable X has a single possible value (and is called *instantiated*) iff $req(X)$ is equal to $pos(X)$.

An n-ary *constraint* C on set variables is associated with an n-ary relation $R_C \subseteq 2^{\mathcal{D}^n}$. An n-ary *constraint literal* is an n-ary constraint C applied to an n-ary tuple of variables $(X_1, \ldots, X_n)$ and is written in the form $C(X_1, \ldots, X_n)$.

Definition 2. *A CSP on finite set variables $(\mathcal{D}, \mathcal{X}, \mathcal{C})$ is defined by a finite element domain $\mathcal{D}$, a set of variables $\mathcal{X}$, and a set of constraint literals $\mathcal{C}$. A domain assignment of the CSP is a pair (req, pos) of two mappings $req : \mathcal{X} \to 2^{\mathcal{D}}$ and $pos : \mathcal{X} \to 2^{\mathcal{D}}$. A solution for the CSP under a given domain assignment (req, pos) is a value assignment $val : \mathcal{X} \to 2^{\mathcal{D}}$ satisfying*

$$\begin{array}{l} 1.\ req(X) \subseteq val(X) \subseteq pos(X) \quad \text{for all } x \in \mathcal{X}, \\ 2.\ (val(X_1), \ldots, val(X_n)) \in R_C \text{ for all } C(X_1, \ldots, X_n) \in \mathcal{C}. \end{array} \tag{1}$$

As with standard CSPs, filtering techniques can be applied that reduce the domains of set variables. For each constraint C, a specific filtering operation can be defined that exploits the semantics of the relation R_C [4]. We give an example

for a simplified union-over-set constraint that is defined for $3n$ set variables with $\mathcal{D} := \{1, \ldots, n\}$. The constraint $union(X_1, Y_1, Z_1, \ldots, X_n, Y_n, Z_n)$ is satisfied by val iff

$$\bigcup_{j \in val(X_i)} val(Y_j) \subseteq val(Z_i)$$

for all $i = 1, \ldots, n$. The following filtering operation defines new sets of required and possible elements:

$$\begin{array}{l} req'(Z_i) := req(Z_i) \cup \{k \in \mathcal{D} \mid \exists j : j \in req(X_i), k \in req(Y_j)\} \\ pos'(Y_j) := pos(Y_j) - \{k \in \mathcal{D} \mid \exists i : j \in req(X_i), k \notin pos(Z_i)\} \\ pos'(X_i) := pos(X_i) - \{j \in \mathcal{D} \mid \exists k : k \notin pos(Z_i), k \in req(Y_j)\} \end{array} \quad (2)$$

We say that a domain assignment is *locally consistent* iff the filtering operation of each constraint maps the domain assignment to itself.

Throughout this paper, we use constraints with the following semantics. Let X, Y, X_i, Y_i, Z_i be constrained set variables, d be an element of $\mathcal{D}$, and $C(\ldots, X_i, \ldots)$ be a constraint literal having X_i among its variables:

1. unary constraints: $\mid X \mid, d \in X, d \notin X$.
2. binary constraints: $X \subseteq Y, X \not\subseteq Y, X = Y$.
3. ternary constraints: $X \cap Y = Z, X \cup Y = Z$.
4. set-of: $\{i \in \mathcal{D} \mid C(\ldots, X_i, \ldots)\}$.
5. inverse: $j \in X_i$ iff $i \in Y_j$.
6. aggregations: $Z_i = \bigcup_{j \in X_i} Y_j$ and $Z_i = max\{Y_j \mid j \in X_i\}$.

These constraints and the corresponding filtering are provided by the constraint library ILOG SOLVER [5] or have been added as extension [6].

As an example, we introduce a CSP for the key configuration problem. We choose the set $\mathcal{P}$ of pins as the element domain of the CSP. We introduce a set variable X_k for each key k and a set variable Y_l for each lock l denoting the respective pin sets. For each $(k, l) \in \mathcal{O}$, we introduce a constraint literal subset$(\mathrm{X}_k, \mathrm{Y}_l)$. Furthermore, we introduce a constraint literal not-subset$(\mathrm{X}_k, \mathrm{Y}_l)$ for each $(k, l) \in \overline{\mathcal{O}}$ where $\overline{\mathcal{O}} := (\mathcal{K} \times \mathcal{L}) - \mathcal{O}$. A value assignment val is a solution of this CSP iff

$$\begin{array}{l} 1.\ val(\mathrm{X}_k) \subseteq val(\mathrm{Y}_l) \text{ for } (k, l) \in \mathcal{O} \text{ (opening)} \\ 2.\ val(\mathrm{X}_k) \not\subseteq val(\mathrm{Y}_l) \text{ for } (k, l) \in \overline{\mathcal{O}} \text{ (locking)} \end{array} \quad (3)$$

Since the current domain assignment (req, pos) is locally consistent, the value assignment $val := req$ satisfies the first constraint. It satisfies also the second constraint if for each pair (k, l) in $\overline{\mathcal{O}}$ there exists a pin p that is in the required set $req(\mathrm{X}_k)$, but not in the possible set of $pos(\mathrm{Y}_l)$. A possible search strategy is to pick a pair in $\overline{\mathcal{O}}$ and a pin p and then satisfy the constraint by adding p to $req(\mathrm{X}_k)$ and removing it from $pos(\mathrm{Y}_l)$. We say that the *conflict* $(k, l) \in \overline{\mathcal{O}}$ is solved by the pin p in this case.

In general, the number of pairs in $\overline{\mathcal{O}}$ is large compared to the number of available pins. As a consequence, several of the conflicts are solved by the same pin. The crucial question is which conflicts can be solved by the same pins.

	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8
k_1	X		X				X	X
k_2		X	X				X	X
k_3	X	X	X				X	
k_4				X		X	X	X
k_5					X	X	X	
k_6				X	X	X	X	X
k_7	X	X	X	X	X	X	X	X
k_8		X	X	X		X	X	

	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8
k_1		4		2	2 4	2		
k_2	3			2 3	2	2		
k_3				2	2	2		6
k_4	1	1 4	1		4			
k_5	1 3	1	1	3				6
k_6	1	1	1					
k_7								
k_8	5				5			5

Fig. 2. A nearly hierarchical example (left) and the chosen conflict areas (right).

4 Problem Decomposition

In this section, we determine which conflicts in $\overline{\mathcal{O}}$ can be solved by the same pins. The result is used to elaborate a problem decomposition scheme that allows a good exploitation of the available pins.

Consider a solution to a key configuration problem (defined by P and Q). For each pin p, we define the set of keys and locks to which the pin has been assigned in the given solution:

$$\begin{aligned} K(p) &:= \{k \in \mathcal{K} \mid p \in P(k)\} \\ L(p) &:= \{l \in \mathcal{L} \mid p \in Q(l)\} \end{aligned} \tag{4}$$

If a key k is in $K(p)$ and a lock l is not in $L(p)$ then $P(k)$ is not a subset of $Q(l)$. As a consequence, k does not open l (i.e. $(k,l) \in \overline{\mathcal{O}}$). Thus, we get the following property of a solution:

$$K(p) \times (\mathcal{L} - L(p)) \subseteq \overline{\mathcal{O}} \tag{5}$$

Suppose that two conflicts (k_1, l_1), (k_2, l_2) are solved by the same pin p. Then the keys k_1, k_2 are elements of $K(p)$ and the locks l_1, l_2 are not elements of $L(p)$. As a consequence, (k_1, l_2) and (k_2, l_1) are conflicts as well.

We conclude that a pin p can be assigned to a set K' of keys and suppressed for a set L' of locks if $K' \times L'$ is a subset of $\overline{\mathcal{O}}$, i.e. none of the keys in K' opens a lock in L'. We call such a pair (K', L') a *conflict area*. The key configuration problem can be formulated as the problem of finding a minimal number of conflict areas that cover all conflicts in $\overline{\mathcal{O}}$.

We discuss some examples for finding large conflict areas. The first example consists of 8 keys and 8 locks. The locking matrix is shown in the left part of figure 2. In total, there are 27 conflicts. All conflicts that are in the same row (or in the same column) can be solved by the same pin. Since 1 row does not contain a conflict, 7 areas would be sufficient. An even better solution is obtained if we explore the (nearly) hierarchical structure of the problem. In fact, we can decompose the complete problem into two subproblems $k_1, k_2, k_3, l_1, l_2, l_3$ and $k_4, k_5, k_6, l_4, l_5, l_6$ and a common problem k_7, l_7, k_8, l_8. There is a conflict area between the keys of the second subproblem and the locks of the first one. The

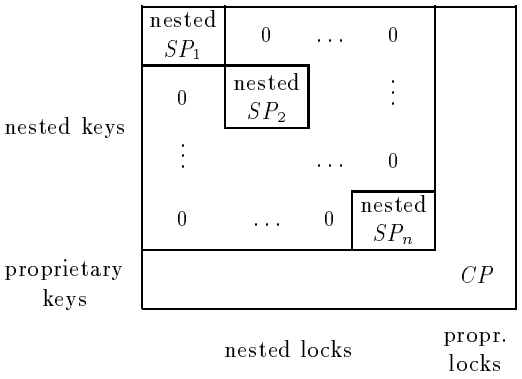

Fig. 3. Structure of (sub)problem

second conflict area is between the keys of the first subproblem and the locks of the second one. Hence, two pins are sufficient to separate both subproblems. Each subproblem consists of two conflicts and two pins are necessary to solve them. However, the pins needed for the first subproblem can be reused for the second one. Since the keys of a subproblem are in conflict with the locks of the other subproblem this does not provide problems. Keys and locks of the common problem that are involved in conflicts are called *flexible keys* and *flexible locks*. In the example, k_8 and l_8 are flexible and one additional pin per flexible object is needed. As a result, we need only 6 pins. Although the gain of 1 is not very impressive, the possibilities of reusing the same pins in different subproblems pays off if they are large.

The second example consists of 2^n keys $k_1, \ldots, k_{2^n}$ and 2^n locks $l_1, \ldots, l_{2^n}$. The key k_i opens the lock l_i, but no other lock. The simplest way is to choose a conflict area per row ending up with 2^n pins. A second solution is to recursively divide the initial problem into two subproblems of same size. Then $2n$ pins are sufficient. For $2^n = 1024$ keys (and locks) we thus need 20 pins. An even better solution is obtained as follows: Consider a set of m pins. We want to choose a maximal number of subsets such that none of the sets is a subset of any other set. Then it is sufficient to assign to each key/lock pair k_i,l_i one of these sets. The subsets of cardinality $\frac{m}{2}$ satisfy the required property. There are $\binom{m}{m/2}$ such subsets. Hence, we determine the smallest m s.t. $\binom{m}{m/2}$ is greater than or equal to 2^n. For the 1024 keys, we thus need only 13 pins. The 20 pins needed for the first approach can be used for 184756 keys, i.e. 180 times more.

We generalize both examples to the following problem decomposition scheme. A given problem can be decomposed into several subproblems and a common part. As illustrated in figure 3, the subproblems are arranged in a diagonal. This decomposition can recursively be applied to the subproblems. Thus, we obtain a hierarchy of subproblems. The root of the hierarchy is the initial problem. We require that all subproblems in the hierarchy satisfy the following properties:

1. Each key of a subproblem opens at least one lock of the subproblem and each lock of a subproblem is opened by at least one key of the subproblem.
2. A subproblem has either none or more than one nested subproblem.
3. A key (or lock) of a subproblem either belongs to a nested subproblem or it is called proprietary key (or proprietary lock) of the subproblem. If a key (or lock) is a proprietary key (or proprietary lock) of a subproblem then this subproblem is called the proprietary problem of the key (or the lock).
4. If a key opens a lock then one of them belongs to the proprietary problem of the other.
5. A proprietary key of a subproblem is a flexible key iff there exists a lock of the subproblem that it does not open. A proprietary lock of a subproblem is a flexible lock iff there exists a key of the subproblem that does not open it.

Such a problem decomposition exists if and only if each key of the key configuration problem opens at least one lock and each of the locks is opened by at least one key.

We determine the number of pins needed for a subproblem. Suppose a subproblem s has n nested subproblems $s_1, \ldots, s_n$ and s_i needs n_i pins. Furthermore, let m be the smallest number s.t. $\binom{m}{m/2}$ is greater than or equal to n. Then s needs m pins for separating the subproblems. Since pins can be reused in the nested subproblems, s needs only $max\{n_i \mid i = 1, \ldots, n\}$ pins inside the subproblems. Furthermore, s needs 1 pin for each flexible key and flexible lock. The sum of these numbers gives the number of pins needed for s. The number of pins needed for the complete problem is obtained at the root of the subproblem hierarchy.

5 Tree Configuration

The crucial problem now is to find a subproblem hierarchy for a given key configuration problem that minimizes the number of pins. This problem again is a combinatorial problem and it is therefore convenient to formulate it as a CSP. This CSP has a general part namely that of the configuration of a tree which will be presented in this section. In the next section, we add further constraints for ensuring that the resulting tree has the properties of a subproblem hierarchy as introduced in the last section.

The tree configuration problem (TCP) consists in arranging a given set of objects $\mathcal{V} := \{v_1, \ldots, v_n\}$ in form of a tree. Several of these objects can belong to the same node in the tree. In this case, we say that they are *equivalent*. We suppose that there is one distinguished object r in $\mathcal{V}$ that belongs to the root node. An object is an ancestor (descendant) of another object if the node of the first object is an ancestor (descendant) of the node of the second object. An object is related to another object if there is a path between the nodes of both objects in the tree. If there is no such path the objects exclude each other and can be separated. In order to determine these relations, we introduce the

following constrained set variables for each $v_i \in \mathcal{V}$:

$$\begin{array}{ll} \mathrm{A}_i & \text{the set of ancestors of } v_i. \\ \mathrm{D}_i & \text{the set of descendants of } v_i. \\ \mathrm{E}_i & \text{the set of objects equivalent to } v_i. \\ \mathrm{S}_i & \text{the set of descendants or equivalent objects of } v_i. \\ \mathrm{R}_i & \text{the set of objects related to } v_i. \\ \mathrm{X}_i & \text{the set of excluded objects of } v_i. \end{array} \tag{6}$$

There are two reasons for choosing a representation in terms of ancestors instead of parents. First, initial constraints on ancestors and related nodes can be expressed directly. Second, this allows less strict decisions and is less prone to failure. After deciding that v is an ancestor of w, it is possible to insert additional objects between v and w.

We now introduce the constraint (literals) of the CSP. For the sake of readability, we only present the semantics of these constraints and write $A(v_i)$, $D(v_i)$, $E(v_i)$, $S(v_i)$, $R(v_i)$, $X(v_i)$ for the values $val(\mathrm{A}_i)$, $val(\mathrm{D}_i)$, $val(\mathrm{E}_i)$, $val(\mathrm{S}_i)$, $val(\mathrm{R}_i)$, $val(\mathrm{X}_i)$ of the variables $\mathrm{A}_i, \mathrm{D}_i, \mathrm{E}_i, \mathrm{S}_i, \mathrm{R}_i, \mathrm{X}_i$.

Union: $S(v)$ has been defined as the union of $D(v)$ and $E(v)$. The set of related objects is the union of $D(v)$, $E(v)$, and $A(v)$. The excluded objects are the complement of the related nodes.

$$S(v) = D(v) \cup E(v) \quad R(v) = S(v) \cup A(v) \quad \mathcal{V} = R(v) \cup X(v) \tag{7}$$

Disjointness: The set of ancestors, the set of equivalent objects, the set of descendant objects, and the set of excluded objects of an object v are mutually disjoint. Three constraints are sufficient to express this property:

$$D(v) \cap E(v) = \emptyset \quad S(v) \cap A(v) = \emptyset \quad R(v) \cap X(v) = \emptyset \tag{8}$$

Reflexivity: Each object is equivalent to itself:

$$v \in E(v) \tag{9}$$

Inverse: If an object v is an ancestor of an object w then the w is a descendant of v and vice versa. Furthermore, if v is equivalent to w then w is equivalent to v. The same property holds for related nodes.

$$\begin{array}{l} v \in A(w) \text{ iff } w \in D(v) \\ v \in E(w) \text{ iff } w \in E(v) \\ v \in R(w) \text{ iff } w \in R(v) \end{array} \tag{10}$$

Transitivity: If v is an ancestor of w then all ancestors of v are ancestors of w. Furthermore, if v is equivalent to w then all objects that are equivalent to v are also equivalent to w. We formulate both properties by a union-over-set-constraint:

$$\bigcup_{w \in A(v)} A(w) \subseteq A(v) \qquad \bigcup_{w \in E(v)} E(w) \subseteq E(v) \tag{11}$$

Tree properties: The root r has no ancestors and no excluded objects:

$$A(r) = \emptyset \quad X(r) = \emptyset \tag{12}$$

If an object v has two ancestors w_1, w_2 then these ancestors are related. This property guarantees the existence of a unique parent of v. We express it by a union-over-set-constraint in the following way:

$$\bigcup_{w \in D(v)} A(w) \subseteq R(v) \tag{13}$$

Definition of Equivalence: If a node in the tree has only a single son then there is no reason to put these two objects in different nodes. We avoid this case by the following property: Two objects v and w have the same sets of related objects iff they are equivalent. We formulate this by the setof-constraints:

$$E(v) := \{w \in \mathcal{V} \mid R(v) = R(w)\} \tag{14}$$

Properties of Equivalence: If two nodes are equivalent then they have the same set of related nodes, of ancestors, and so on. It is sufficient to state:

$$\bigcup_{w \in E(v)} R(w) \subseteq R(v) \qquad \bigcup_{w \in E(v)} S(w) \subseteq S(v) \qquad \bigcup_{w \in E(v)} A(w) \subseteq A(v) \tag{15}$$

Ancestors: The following constraint is implied by the others, but allows more propagation: An object w' is an ancestor of an object v iff they are related and there exists an excluded object w of v such that w' is also related to w. We formulate this by a union-over-set-constraint:

$$A(v) = R(v) \cap \bigcup_{w \in X(v)} R(w) \tag{16}$$

We obtain a further implied constraint: If w is a descendant of v then all objects related to w are also related to v:

$$\bigcup_{w \in D(v)} R(w) \subseteq R(v) \tag{17}$$

The TCP-CSP for $\mathcal{V}$ and r consists of the variables (6) and the constraints (7 - 17). The solutions of the TCP-CSP are trees that are labelled with sets of objects and don't have nodes with only one son.

Definition 3. *A labelled tree for $\mathcal{V}$ and r is defined by a set of nodes $\mathcal{N}$, a root $r' \in \mathcal{N}$, a mapping $\pi : \mathcal{N} - \{r'\} \rightarrow \mathcal{N}$ defining the parent of a node, and a mapping $\nu : \mathcal{V} \rightarrow \mathcal{N}$ defining the node of an object*[1] *such that*

1. *The graph $(\mathcal{N}, \{(\pi(n), n) \mid n \in \mathcal{N})$ is a directed tree,*
2. *$\nu(r) = r'$,*
3. *ν is surjective,*
4. *if $n_1 \in \mathcal{N} - \{r'\}$ then there exists $n_2 \in \mathcal{N} - \{r', n_1\}$ s.t. $\pi(n_1) = \pi(n_2)$.*

(18)

[1] In fact, each node $n \in \mathcal{N}$ is labelled with the set of objects v satisfying $\nu(v) = n$.

Each solution of the CSP corresponds to a unique labelled tree:

Proposition 1. *There exists a bijective mapping from the set of solutions of the TCP-CSP for $\mathcal{V}$ and r to the set of labelled trees for $\mathcal{V}$ and r.*

A tree can be extracted as follows from a solution of the TCP-CSP. Each equivalence class $E(v)$ is a node of the tree. A node $E(v)$ is the parent of node $E(w)$ iff $\mid A(w) \mid = \mid A(v) \mid + \mid E(v) \mid$.

6 Tree of Subproblems

In this section, we apply the CSP for tree configuration to the problem of finding a subproblem hierarchy for the key configuration problem. We apply the TCP-CSP to the given key and lock objects (plus an additional root object $r \notin \mathcal{K} \cup \mathcal{L}$) and add further constraints that are based on the locking relation $\mathcal{O}$. Hence, we consider the following set of objects:

$$\mathcal{V} := \mathcal{K} \cup \mathcal{L} \cup \{r\} \tag{19}$$

Each object v then defines a subproblem containing the objects $S(v)$. In section 4, we called it the proprietary subproblem of v. The proprietary keys and locks of this subproblem are in $E(v)$. The keys and locks of the nested subproblems are in $D(v)$.

Locking relation: Given any object v in $\mathcal{V}$, we are interested in the objects that are either opened by v or that v opens.

$$O(v) := \begin{cases} \{w \in \mathcal{K} \mid (w,v) \in \mathcal{O}\} & \text{if } v \in \mathcal{L} \\ \{w \in \mathcal{L} \mid (v,w) \in \mathcal{O}\} & \text{if } v \in \mathcal{K} \\ \mathcal{V} & \text{if } v = r \end{cases} \tag{20}$$

If an object w is in $O(v)$ then one of these objects must be in the proprietary subproblem of the other. This means that both objects must be related (by a path in the resulting tree):

$$O(v) \subseteq R(v) \tag{21}$$

Non-empty subproblems: An object w can only be in the proprietary subproblem $S(v)$ of another object v if it opens or is opened by another object of the subproblem (i.e. $O(w) \cap S(v) \neq \emptyset$ for all $w \in S(v)$). We formulate this condition in terms of *setof*-constraints:

$$S(v) \subseteq \{w \in \mathcal{V} \mid O(w) \cap S(v) \neq \emptyset\} \tag{22}$$

Subsumption: (implied constraint) If the set of opened/opening objects of an object v is a subset of the set of opened/opening objects of an object w then these both objects are related:

$$\text{if } O(v) \subseteq O(w) \text{ then } v \in R(w) \tag{23}$$

Flexible Keys/Locks: Given any object v in $\mathcal{V}$, we are interested in the forbidden objects of v, i.e. the objects that are not opened by v or that v does not open.

$$\overline{O}(v) := \begin{cases} \{w \in \mathcal{K} \mid (w,v) \in \overline{\mathcal{O}}\} & \text{if } v \in \mathcal{L} \\ \{w \in \mathcal{L} \mid (v,w) \in \overline{\mathcal{O}}\} & \text{if } v \in \mathcal{K} \\ \emptyset & \text{if } v = r \end{cases} \tag{24}$$

A proprietary object w of a subproblem is flexible iff this subproblem contains a forbidden object of w. Again, we use *setof*-constraints to formulate this condition:

$$F(v) := \{w \in E(v) \mid \overline{O}(w) \cap S(v) \neq \emptyset\} \tag{25}$$

Objective: The set of flexible objects allows us to define the objective. We use a simplified objective that counts only the number of flexible objects and neglects the number of pins needed for separating the subproblems.

$$c(v) = \mid F(v) \mid + max\{c(w) \mid w \in D(v)\} \tag{26}$$

The CSP for the key configuration decomposition problem (KDP-CSP) is a CSP that consists of the variables(6), the constraints (7 - 17) and (20 - 26), and the objective of minimizing $c(r)$. Each solution of this CSP corresponds to a subproblem hierarchy for our key configuration problem.

7 Algorithm

We briefly describe a non-deterministic algorithm for finding the solutions of the KDP-CSP. It is sufficient to consider the decision variables $X(v)$ for all objects v. All other sets are uniquely determined by a value assignment to the $X(v)$s according to the constraints (7), (8), (14), (16), and (10).

A trivial solution of the KDP-CSP consists in setting all the $req(X(v))$s to the empty set. In this case, none of the objects exclude each other and there is a single subproblem that contains all objects. This solution has the highest number of flexible keys and locks. In order to reduce the number of flexible objects, we try to separate as many objects as possible by maximizing the sets $X(v)$. Our algorithm consists of two loops. In the outer loop, the algorithm chooses a non-instantiated $X(v)$. It instantiates it $X(v)$ by considering all possible elements w that are not required. A non-deterministic choice is made: Either w is added to the required set of $X(v)$ (the preferred choice) or removed from the possible set of $X(v)$. The algorithm finds a solution iff all variables $X(v)$s can be instantiated with success:

1. **while** there are non-instantiated variables $X(v)$ **do**
2. select a $v \in \mathcal{V}$ s.t. $req(X(v)) \neq pos(X(v))$.
3. **while** $req(X(v)) \neq pos(X(v))$ **do**
4. select a $w \in pos(X(v)) - req(X(v))$.
5. **choose:** add w to $req(X(v))$ or remove w from $req(X(v))$.
6. apply filtering algorithm to update current domain assignment.
7. **if** this domain assignment is inconsistent **then** return with failure.
8. return with success.

Such a non-deterministic algorithm is easy to implement by the search programming facilities of tools such as ILOG SOLVER.

A big concern, however, is the efficiency of this approach. If there are n objects in $\mathcal{V}$ then maximal n^2 decisions are taken by one non-deterministic execution of the algorithm. For each of these decisions, two alternatives are considered. This means that we can obtain maximal 2^{n^2} different executions. Despite these worst-case considerations, the CSP has several good properties that reduce effort:

1. **Unary constraints:** The constraints 21, 23 involve a single variable and allow reducing the domain of this variable initially (i.e. before any non-deterministic choice has been made),
2. **Variable ordering:** In order to find larger subproblems first, we select a v with largest $req(R(v))$ (i.e. smallest $pos(X(v))$) first (in line 2).
3. **Value ordering:** The same principle can be applied when selecting a value w. Thus, we determine relations between larger subproblems first.
4. **Separation of subproblems:** Once $X(v)$ is instantiated, the set of ancestors of v is determined as well. There remain two subproblems, which are independent and can be solved separately: First, determine the relations among the objects in $S(v)$ and second, the relations among the objects in $X(v) \cup A(v)$.

The first three principles have been used in an industrial prototype for key configuration and helped to find good solutions quickly. More work is needed to explore the last principle. This could allow treating examples of large size efficiently, supposing that these examples have a nearly hierarchical structure.

Ex.	#initial objects	#groups	#flexible objects	height	#leaves	# choice points	#fails	CPU time
1	51	15	2	4	6	105	176	4sec.
2	46	33	4	6	12	803	1828	33sec.
3	1672	318	4	7	92	262	563	2222sec.
4	4444	252	36	12	94	3706	8158	837sec.

Fig. 4. Results of Industrial Prototype

8 Preliminary Experimental Results

In this section, we report preliminary experimental results that were obtained by an industrial prototype, which was developed for a manufacturer of locking systems. The need for problem decomposition methods was discovered quite early in the development phase. Even for smaller examples, initial experiments

	1	3	4	5	6	8	9	10	14	15	17	23	26	28	29	30	31	33	34
350	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
501	X	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	.	.	.
502	.	.	.	.	.	.	.	.	X	.	X	.	X	.	.	.	.	.	.
503	.	.	.	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.
504	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	.	.	X	.
505	.	.	.	.	.	.	.	.	X	X	.	.	.	.	.	.	.	.	.
506	X	.	.	X	.	.	X	.	X	.	.	.	X	.	.	.	.	.	.
507	.	.	.	.	.	X	X	.	.	.	.	.	.	.	.	.	.	.	.
508	.	.	.	.	X	.	.	.	.	.	.	.	.	X	X	X	X	X	.
509	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	.	.	.	.
510	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	X	.	.	.
511	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	X	.
512	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X
513	.	.	.	.	X	.	.	X	.	.	.	.	.	.	.	.	.	.	.

Fig. 5. Locking matrix for example 2.

showed that the assignment of pins to keys and locks should be guided by hierarchical relations between these objects. First attempts were made to find subsumption relations between keys/locks in a procedural way and to directly build a tree. This approach lacked flexibility (e.g. no backtracking possible) and a clear mathematical specification and worked only for smaller examples. Based on this experience, we developed a first CSP for finding a subproblem hierarchy.

Compared to the CSP presented in this paper, the prototype has some drawbacks. First, equivalence classes $E(v)$ and some of the constraints were missing. As a consequence, inner nodes could have a single son, which led to weird trees. Second, the objective was to minimize the total number of flexible keys/locks, which was simpler to express and which is an upper bound of the objective $c(r)$. Third, keys to doors of offices were not allowed to be flexible due to manufacturer-specific conventions.

In order to find good solutions quickly, search occurs in two phases. In a first phase, the search tree is artificially cut. Heuristics are used to estimate whether only one of the alternatives is promising or whether all have to be considered. In the second, all alternatives are considered, but the best solution of the first phase puts an upper bound on the objective.

Figure 4 shows the results for some selected examples. In order to treat larger examples efficiently, keys/locks of offices that belong to the same floor are grouped together. The second columns contains the number of objects before preprocessing and the third column contains the number after this operation. We discuss two of the examples briefly. The second example is small, but lacks a clear hierarchical structure. Figure 5 shows the locking matrix of this example, whereas figure 6 gives the resulting tree[2]. Example 3 is large, but has a clear hierarchical structure and thus a small number of flexible keys/locks.

[2] The equivalence relation is depicted by horizontal links.

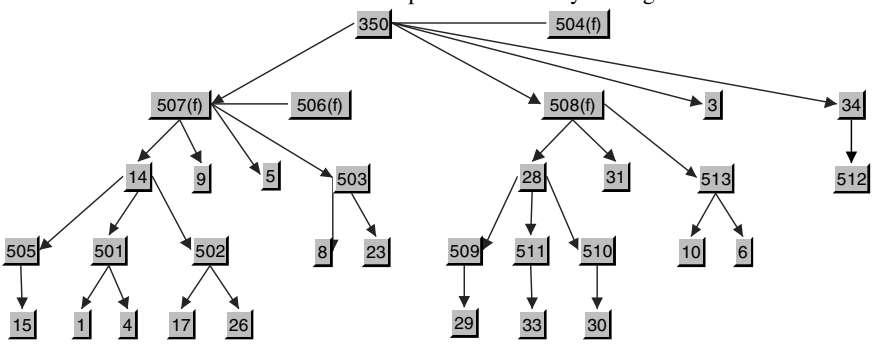

Fig. 6. Subproblem hierarchy for example 2.

9 Conclusion

We presented a novel approach to problem decomposition for key configuration problems. An initially large problem is recursively divided into subproblems and a common problem such that the subproblems can be treated independently after the common problem has been solved. Most key configuration problems in practice have this hierarchical structure and can thus be treated efficiently.

The novelty of our approach is the use of constraint satisfaction techniques for finding a subproblem hierarchy. This CSP-approach has several advantages over a procedural or purely heuristic decomposition method. Mathematical properties of the subproblem hierarchy can be stated clearly and added easily. Search algorithms allow exploring several solutions and finding a good or a best subproblem hierarchy. The CSP makes a sophisticated usage of constrained set variables (cf. [7,3]) and clearly shows the usefulness of constraints such as set-of, inverse, max-over-set, and union-of-sets-over-set [6].

References

1. E. Andersson, E. Housos, N. Kohl, and D. Wedelin. Crew pairing optimization. In Gang Yu, editor, *OR in Airline Industry*, Kluwer Academic Publishers, Boston/London/Dordrecht, 1998.
2. B.Y. Choueiry, B. Faltings, and R. Weigel. Abstraction by interchangeability in resource allocation. In *Proc. IJCAI'95*, Montréal, 1995.
3. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints. An International Journal*, 1, 1997.
4. P. van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57, 1992.
5. ILOG. Ilog Solver. V4.0. Reference manual and user manual, ILOG, 1997.
6. D. Mailharo. A classification and constraint based framework for configuration. *AI-EDAM: Special Issue on Configuration*, 12(4), 1998.
7. J.F. Puget. Programmation par contraintes orientée objet. In *Intl. Conference on Expert Systems*, Avignon, 1992.