MASARYK UNIVERSITY BRNO
FACULTY OF INFORMATICS

Algorithms for Constraint Satisfaction Problems

Master Thesis

Brno, April 2003 Kamil Veřmiřovský

Acknowledgments

I would like to thank to my supervisor, Hana Rudová, for help and encouragement throughout the work. I am very grateful for her advice and valuable discussions.

This work was partially supported by the Grant Agency of the Czech Republic under the contract 201/01/0942 and by Purdue University. I would also like to thank to the Supercomputer Center Brno where the experiments with random problems were accomplished.

Declaration

I declare that this thesis was composed by myself, and all presented results are my own, unless otherwise stated. All sources and literature that I have used during the elaboration of this thesis are cited with complete reference to the corresponding source.

Kamil Veřmiřovský

Abstract

The first part of this thesis contains a classification of the algorithms for constraint satisfaction problems. A formal description of some algorithms of each approach is given, including uniform pseudo-codes. An iterative repair search algorithm is introduced. It is a revised and formalized version of the algorithm first introduced for the needs of a real-life large-scale university timetabling problem. It iteratively tries to find a partial solution with the maximal number of assigned variables. It is devoted to tightly constrained problems, where constraint propagation is not strong enough, variable and value ordering heuristics are not good enough, or the problem is over-constrained. Its importance lies in its natural implementation in a constraint logic programming language. One step is a special incomplete version of chronological backtracking with constraint propagation. Subsequent searches try to improve the last computed partial assignment. This is done by developing variable and value heuristics based on the results of previous iterations. Different parameters of the algorithm are compared. Some possible extensions are proposed and their contribution is assessed. The comparison with other algorithms is given. Last but not least, the results for the real-life problem are presented, both original and after the revision, and compared.

Keywords: constraint satisfaction, search algorithm, limited assignment number search, course timetabling

Contents

List of Figures

Chapter 1

Introduction

The constraint paradigm is a useful and well-studied framework expressing many problems of interest in Artificial Intelligence and other areas of Computer Science. A significant progress has been made in the last decade. Many real-life problems can be expressed as a special case of the constraint satisfaction problem. Some examples are scheduling, configuration, hardware verification, graph problems, molecular biology, etc. The search space is often exponential because the problem is NP-complete. Therefore a number of different approaches to the problem have been proposed to reduce the search space and find a feasible solution in a reasonable time.

There are a few textbooks on Constraint Satisfaction. Some of them are [36, 26, 14, 28]. However, they are either obsolete [36] or they do not concentrate on a description and classification of algorithms for constraint satisfaction problems. An interesting survey can be found in a well known *On-line Guide to Constraint Programming* [2]. But a more recent update is here also needed. In this thesis, such an overview is presented. A formal description of representative algorithms of each approach is presented. The classification is given separately for satisfaction and optimization problems. A special section is devoted to the algorithms which try to find a maximal partial solution. Such an algorithm is a main part of this thesis.

The main objective of the work was to revise, formalize, experimentally evaluate and possibly improve the limited assignment search algorithm outlined in [31]. A detailed description of the revised algorithm is given, including proposed improvements. The implementation in both imperative and constraint logic programming language is presented.

A number of experiments have been done to compare particular modifications and extensions on suitably proposed random problems. The algorithm was also compared with some other algorithms presented in first chapters. Results are given and discussed.

Finally, the real-life application of the algorithm is introduced and some extensions, based mainly on the experiments on random problems, are proposed. New results are compared with the original results mentioned in the above cited paper.

This thesis is divided into the following chapters: a brief introduction to Constraint Satisfaction is given in the next chapter. Chapters 3 and 4 present the above mentioned classification of algorithms for satisfaction and optimization problems, resp. The limited assignment search algorithm is described in Chapter 5. The subsequent two chapters cover the empirical results. Chapter 6 deals with the experiments on random problems. Chapter 7 introduces the real-life application of the algorithm. In the last chapter, the contributions of this thesis are discussed and future work mentioned.

Chapter 2

Constraint Satisfaction

In this chapter, a brief introduction to constraint satisfaction is presented. The chapter is divided into three sections. In the first section, the basic definitions are given, including constraint satisfaction problem. The next section presents some extensions of the constraint satisfaction problem that allow to find an optimal solution. Finally, a constraint logic programming paradigm is introduced as one of the most common environments for solving constraint satisfaction problems.

2.1 Constraint Satisfaction Problem

A CSP [36, 34] is defined by a finite set of problem variables along with their associated finite domains of possible values and a set of constraints on acceptable combinations of the values. A constraint can be given either explicitly, by enumerating the allowed combinations, or implicitly, e.g., by an algebraic expression.

Definition 2.1 A *constraint satisfaction problem (CSP)* is a triple $P = (X, D, C)$, where

- $X = \{x_1, \ldots, x_n\}$ is the set of variables called *domain variables*;
- $D = \{D_1, \ldots, D_n\}$ is the set of *domains*. Each domain is a finite set containing the possible values for the corresponding variable;
- $C = \{C_1, \ldots, C_c\}$ is the set of constraints. A *constraint* c_i is a relation defined on a subset $\{x_{i_1}, \ldots, x_{i_k}\}$ of all variables, i.e., $c_i \subseteq \{D_{i_1} \times \ldots \times D_{i_k}\}$. The set of variables $\{x_{i_1}, \ldots, x_{i_k}\}$ is referred to as a *scope* of the constraint c_i.

Example 2.1 The *n-queens problem* is to place n queens on a generalized chessboard, which is a square divided into $n \times n$ smaller squares, in such a way that queens do not attacks each other. A queen *attacks* another if it is on the same row, column or diagonal. This problem can be modeled with the help of n variables, each with domain $D_i = \{1, \ldots, n\}$. A variable x_i corresponds to a queen on row i (there must be one queen per row). The assignment $x_i = j$, where $j \in D_i$, denotes that the queen on row i is placed in column j. The constraints are $x_i \neq x_j$ and $|x_i - x_j| \neq |i - j|$ for $1 \leq i < j \leq n$. An example solution of a 4-queens problem may be seen in Figure 2.1.

An *arity* of a constraint is the number of variables in its scope: a *unary* constraint applies to a single variable; a *binary* constraint has arity two. A CSP is called *binary* if all its constraints are unary or binary. Any non-binary CSP can be transformed into an equivalent binary CSP by the so called *constraint binarization* [1]

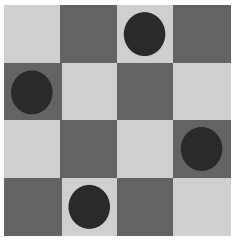

Figure 2.1: The 4-queens problem.

Definition 2.2 An *assignment* is a mapping θ from the set of domain variables $Y \subseteq X$ to their corresponding domains, i.e., $\theta(x_i) \in D_i$ for $x_i \in Y$. Set of all assignments on Y will be represented by Θ_Y.

Let the variables in Y have any fixed order $(x_1, \ldots, x_m)$. Assignment θ corresponding to $\{(x_1, d_{x_1}), \ldots, (x_m, d_{x_m})\}$ will be denoted by $\theta = (d_{x_1}, \ldots, d_{x_m})$. A variable is called *instantiated* when it has been assigned a value from its domain by the given assignment; otherwise it is called *unassigned*. An assignment defined on the set of all domain variables X is *complete* or *total*, otherwise it is called *partial*. The set of all complete assignments $\Theta_X \equiv D_1 \times \ldots \times D_n$ is called *solution space*, in the sense that the solution should be searched within this space.

Definition 2.3 A constraint c is *satisfied* in assignment θ (noted $\theta \models c$) if all the variables in its scope are instantiated and the tuple of values assigned by θ to the variables of c belongs to the relation c. Otherwise it is called *unsatisfied* or *violated*.

A partial assignment θ is *consistent* if all the constraints referring only to the variables assigned by θ are satisfied. A consistent partial assignment is also referred to as a *partial solution*.

Definition 2.4 A *solution of CSP* (X, D, C) is a consistent complete assignment, i.e., all the constraints in the set C are satisfied.

If a CSP does not have any solution, the problem is called *over-constrained* or *inconsistent* and a CSP with more than one solution is called *under-constrained*.

A partial solution is called *locally inconsistent* if it can not be extended by an additional variable and so it can not be part of a solution. A CSP with many local inconsistencies is referred to as a *tightly constrained* problem [35]. Constraint is *relaxed* if there are further element(s) added to its relation. If elements are removed from the relation, the constraint is *tightened*.

Example 2.1 (continuation) Figure 2.1 shows a solution $\theta = (3, 1, 4, 2)$. There is also other possible solution $\delta = (2, 4, 1, 3)$, i.e., the 4-queens problem is under-constrained. For $n < 4$, the n-queens problem is over-constrained. It is not possible to place the queens in such a way that they do not attack each other.

2.2 Optimization Problems

For many practical problems, not one or all solutions, but the best solution according to a criterion is sought. Sometimes certain requirements (constraints) may even be contradictory, i.e., no solution may exist. Still it may be useful to find an assignment of a subset of variables satisfying a subset of constraints of such an over-constrained CSP. Both problems may be expressed as extensions of the classical CSP. They are generally called the *optimization problems*.

2.2.1 Constraint Satisfaction Optimization Problem

Constraint satisfaction optimization problem is a CSP extended by a so called *objective (or cost) function* evaluating each solution (usually to natural numbers).

Definition 2.5 An *objective function* F is a function from variables V to an ordered set W with a maximum element noted $\top$. Smaller values of W wrt. the ordering $\leq_W$ are expected to be more preferred. A *constraint satisfaction optimization problem (CSOP)* is a CSP with an objective function F.

An assignment θ is *preferred* to the assignment δ, if the value of the objective function applied to θ is less than the value of the objective function applied to ω, i.e., $F(\theta) <_W F(\delta)$. An *optimal solution* to the CSOP is such a solution θ that none preferred solution δ to θ exists. The optimal solution is usually sought iteratively, starting with $\top$ and gradually improving the cost.

Example 2.2 The *job-shop scheduling problem (JSSP)* consists of j jobs and m machines. Each job is a chain of operations of a given processing time. Operations must be done one after another in a given order, and each on a given machine. A solution to the JSSP is such a schedule of job sequences on each machines that no machine is processing two different operations at once. The objective function that should be minimized is the total elapsed time between the beginning of the first task and the completion of the last task (the *makespan*).

Maximization problem is easily transformed into an equivalent minimization problem by simply negating the value of the objective function F. It is often not easy to find an optimal solution. Then at least a good enough sub-optimal solution should be found. It will be referred to as a *feasible* solution.

2.2.2 Valued Constraint Satisfaction Problem

The problem that seeks an optimal assignment of a subset of variables satisfying a subset of constraints may be expressed with the help of a so called *valued CSP* (see [29] for a survey). It allows to assign preferences to constraints and possibly also to each value of a variable (defining a unary constraint requiring the value and assigning a preference to it).

Definition 2.6 A *valuation structure* S is a triple $(E, \circledast, \prec)$, where

- $(E, \prec)$ is a totally ordered set with a maximum element noted $\top$ and a minimum element noted $\bot$;
- $\circledast$ is a commutative and associative closed binary operation on E that satisfies:
  - $\forall a \in E, a \circledast \bot = a$ (*identity*);
  - $\forall a, b, c \in E, (a \preceq b) \Rightarrow (a \circledast c) \preceq (b \circledast c)$ (*monotonicity*);
  - $\forall a \in E, (a \circledast \top) = \top$ (*absorbing element*).

Definition 2.7 A *valued CSP (VCSP)* (X, D, C, S, ϕ) is defined by a classical CSP (X, D, C), a valuation structure $S = (E, \circledast, \prec)$, and a function ϕ from C to E, called *valuation* or *cost*. The valuation of an assignment to the variables X is defined as a combination of the valuations of all the violated constraints using $\circledast$.

If a constraint is assigned a valuation $\top$, it must be satisfied. It is referred to as a *hard constraint*. Otherwise it is called a *soft constraint*.

Example 2.3 In school or university timetabling, there are so many demands that it is impossible to satisfy all of them. Therefore the most important requirements must be declared hard, like the fact that two classes can not be taught in the same classroom simultaneously. And the others are assigned preferences, like various demands of the teachers/instructors on the time and classroom where they want to teach.

Again, the total assignment θ is *preferred* to the total assignment δ, if the valuation of θ is less than the valuation δ wrt. the ordering $\prec$. And an *optimal solution* to the VCSP is such a total assignment θ that none preferred total assignment δ to θ exists.

The preferences may be expressed in any algebraic structure satisfying the definition. The most common algebraic structure is the monoid $(\mathcal{N} \cup \{+\infty\}, +, <)$ of natural numbers with the usual ordering and operation of addition. Such a VCSP is called a *weighted CSP*. The preferences express weights of the constraints. Therefore the resulting valuation is the weighted sum of violated constraints.

Example 2.4 For example, the following three constraints may be defined, with the weights in parenthesis:

$$\begin{array}{lll} c_1: & x_1 \in \{1,2,3\} & (+\infty); \\ c_2: & x_2 = 3 & (5); \\ c_3: & x_1 < x_2 & (1). \end{array}$$

The assignment $\theta_1 = (1,3)$ will have valuation $\phi_1(\emptyset) = 0$, because it satisfies all constraints. The assignment $\theta_1 = (2,1)$, will have valuation $\phi_2(\{c_2, c_3\}) = 5+1 = 6$ (c_2 and c_3 are violated), and the assignment $\theta_3 = (4,3)$, $\phi_3(\{c_1, c_3\}) = +\infty + 1 = +\infty$ (c_1 and c_3 are violated).

Many other instances can be considered, like *fuzzy CSP*, *probabilistic CSP* or *possibilistic CSP*. There are also other general frameworks for CSPs with preferences. The most important is named *semiring-based CSP (SCSP)*. It assigns a preference value to each value tuple and allows a partial ordered set to be used as a valuation structure. More details can be found in [29].

Variable Valued Constraint Satisfaction Problem

Sometimes not an assignment satisfying a subset of constraints, but a partial solution maximizing the number of instantiated variables is needed.

Example 2.3 (continuation) For example in a timetabling problem, there may be so many hard constraints that it is impossible to find a solution in a reasonable time. Therefore first a partial timetable is sought. If some classes are left unscheduled, the operator evaluates the partial timetable and tries to place them manually. If this is not possible, some requirements has to be relaxed and the process is repeated.

Such a problem may be defined with the help of so called *variable valued CSP (VVCSP)* [25]. The VVCSP assigns to each variable a valuation expressing its importance. The objective is to produce a partial assignment to the problem variables which satisfies all constraints and minimizes the combination of valuations of the unassigned variables.

The VVCSP is a special case of VCSP, which can be obtained using the following transformation [25]: each domain of a problem variable is extended by a new *rejection value*, meaning that the variable is unassigned. A soft unary constraint with the corresponding cost is then added for each variable, requiring the rejection value.

2.3 Constraint Logic Programming

The constraint satisfaction problems are often (but not always) solved in a so called *Constraint Logic Programming (CLP) paradigm* [18]. It is a combination of logic programming and the idea of constraints. In CLP, constraint satisfaction is embedded into a logic programming language, such as Prolog [5]. This combination is very natural, because the idea of declaring *what* has to be solved instead of *how* to solve it is common to both paradigms.

CLP systems differ in the domains and types of constraints they process. Families of CLP techniques appear under names of the form CLP(X) where X stands for the domain. For example, in CLP($\mathbb{R}$) the domains of variables are real numbers, and constraints are arithmetic equalities and inequalities over real numbers. CLP(X) systems over other domains include: CLP($\mathbb{Z}$) (integers), CLP($\mathbb{Q}$) (rational numbers), CLP($\mathbb{B}$) (Boolean domains), and CLP(FD) (user defined finite domains). The CSPs can be solved employing CLP(FD) libraries of standard Prolog environments. See, for example, SICStus Prolog [17] or ECL^{i}PS^e [15].

The basic structure of constraint logic programs is always the same and can be seen in Figure 2.2. First a domain variables are created, then the problem constraints are posted, and finally

```
solve( Variables ) :–
        declare_variables( Variables ),
        post_constraints( Variables ),
        labeling( Variables ).
```

Figure 2.2: The basic structure of a constraint logic program.

the search for the solution is initiated. Possible implementations of the predicate labeling/1 which performs the search are presented in the next chapters.

CLP libraries are part of many Prolog systems because they make tree search more efficient applying consistency techniques (see Section 3.1). The depth-first search which is a natural part of logic programming is a part of many algorithms for constraint satisfaction problems. Therefore their implementation in CLP is often faster than it would be in an imperative paradigm. The separation of the problem definition and the solver also allows to easily modify or extend the solved problem.

Besides CLP, there are constraint libraries available for imperative programming languages. For example, the ILOG Solver library [16] for C++ allows to define a problem using the object-oriented technology.

Chapter 3

Algorithms for Satisfaction Problems

In this chapter, the main approaches to satisfaction problems are presented. These algorithms seek any solution or all solutions of a CSP. Or they try to prove that no solution exists.

The chapter begins with a brief introduction to consistency techniques that try to remove values that can not be part of a solution. Afterwards, the systematic search methods are introduced. These methods explore systematically the whole search space. The combination of tree search algorithms and consistency techniques is discussed in the next section. The consistency techniques prune some parts of the search tree and therefore improve the efficiency of the tree search algorithm. The chapter ends with a brief survey of incomplete search methods.

3.1 Consistency Techniques

The *consistency* (or *constraint propagation*) *techniques* [3, 34] try to eliminate values that are inconsistent with some constraints. There are many consistency techniques ensuring different levels of consistency.

Definition 3.1 A CSP is 1-*consistent* if the values in the domain of each variable satisfy the problem's unary constraints. A problem is k-*consistent*, $k \geq 2$ if given any consistent partial assignment of any $k-1$ distinct variables, there exists a consistent assignment of any single additional variable.

It is important to note that k-consistent problem need not be $(k-1)$-consistent. In particular, if there exists consistent partial assignment of $k-1$ variables, the problem is trivially k-consistent. A problem that is k-consistent for all $k \leq l$ is called *strongly* l-*consistent*. The term *node-*, *arc-*, and *path-consistency* correspond to strong 1-, 2-, and 3-consistency, respectively.

Most of the theoretical and practical work has been devoted to binary CSPs. For example, well known *AC-*k algorithms [23] ensuring so called *arc consistency*. This is justified by the fact that any CSP can be transformed into an equivalent binary CSP [1]. However, this transformation can make the problem much more complicated by introducing too many additional variables with large domains. Therefore also some direct solution techniques for non-binary CSPs have been proposed, some of them generalizing consistency techniques for binary CSPs [36].

Current constraint programming tools implement mostly direct consistency algorithms for non-binary CSPs and the constraint binarization is used only in special cases. In particular, there are many symbolically defined constraints that are common to many CSPs. These constraints are named *global constraints* and there are special techniques to solve them. A classical example of a global constraint is the *all_different*$(x_1, \ldots, x_n)$ constraint which imposes that variables in the set

$x_1, \ldots, x_n$ must have different values. Modeling of CSPs via suitable global constraints belongs to critical decisions of constraint programming, greatly influencing computational efficiency.

Clearly, a binary CSP with n variables can be easily solved making it strongly n-consistent. However, enforcing k-consistency requires in general exponential time and exponential space in k. Therefore in practice consistency techniques are rarely used alone. Consistency algorithms are usually employed together with other approaches (see Section 3.3).

3.2 Systematic Search

The *systematic search* methods explore systematically (and exhaustively) the whole search space. Most common are so called *tree search methods*. They view the search space as a search tree. Each node represents mutually exclusive choices which partition the remaining search space into disjoint sub-spaces. This structure enables to remember with acceptable memory requirements, which parts of the search space have already been visited. Usually, a node corresponds to an assignment of a particular value to a particular variable.

3.2.1 Generate-and-Test Algorithm

The basic and very inefficient systematic search algorithm is called *generate-and-test* (GT). Its idea is simple: each possible assignment to all variables is systematically generated, and tested to see if it satisfies all the constraints. The first assignment that satisfies all the constraints is the solution. The number of combinations considered by this method is the size of the Cartesian product of all variable domains.

3.2.2 Chronological Backtracking

A more efficient method is referred to as *(chronological) backtracking* [36]. It explores the search tree in a depth-first manner. The search starts with all the variables unassigned. In each step, it extends a partial solution towards a complete one by assigning a value to an unassigned variable (called *labeling*). If a new partial solution violates any of the constraints of the problem (called a *fail* or a *dead-end*), the assignment is retracted to the most recently instantiated variable which still has untried values in its domain. This is called a *backtrack*. Then the search continues with an assignment of another value to this variable. That is, each node of a search tree corresponds to a choice between all values of a particular variable. In other words, it corresponds to an assignment of a particular value to a particular variable.

Clearly, backtracking requires exponential time in the worse case. Therefore many improvements have been developed. Backtracking has three major drawbacks:

- thrashing, i.e., repeated failure due to the same reason;
- redundancy, i.e., conflicting values of variables are not remembered;
- late detection of the conflict, i.e., conflict is not detected before it really occurs.

Some methods that try to avoid these drawbacks are described in Section 3.3.

The efficiency of the search can be strongly influenced also by the order in which the variables and values are tried. It is clear that the most promising values should be tried first. Another possible motivation is that when it is hard to find a suitable value to a certain variable, it is even harder later, when there are even less values in the domains of variables. Therefore it seems reasonable to label these variables preferentially (the principle of instantiating first variables that are most hard to assign is called a *first-fail* principle). The methods that select in each step the variable and its value

that should be tried next are called *variable and value ordering heuristics*. They guide the search process towards the regions of the search space that are likely to contain a solution. The ordering need not be computed only before the algorithm starts (*static ordering*). It may be determined dynamically by the current state of computation (*dynamic ordering*). Dynamic variable ordering heuristics allow to select a variable even after backtrack, instead of continuing with the variable to which the backtrack retracted. That is, a node of a search tree corresponds to a choice between selecting a particular value or removing it from the domain of a particular variable. Many heuristics have been developed. Experimental analysis of some of them can be found in [9].

Function BACKTRACKING
Input: A CSP (X, D, C)
Output: Either a solution, or a decision that the CSP has no solution

```
d(i) ← D_i for 1 ≤ i ≤ n                                   (copy all domains)
i ← 1                                                      (initialize variable counter)
while 1 ≤ i ≤ n do
    reorder variables x_i, ..., x_n according to variable ordering heuristic   (select variable)
    x_i ← SELECT-VALUE                                     (select value)
    if x_i is null then                                    (no value left in domain)
        if i > 1 then
            reset d(i) to its state before x_{i-1} was last instantiated
        i ← i − 1                                          (backtrack)
    else
        i ← i + 1                                          (step forward)
end while
if i = 0 then
    return "no solution"
else
    return instantiated values of {x_1, ..., x_n}
```

Figure 3.1: The backtracking algorithm.

Figure 3.1 shows a possible implementation of the backtracking algorithm with dynamic ordering heuristics. The algorithm maintains an auxiliary value domain $d(i)$ for each $1 \leq i \leq n$. Each $d(i)$ holds the subset of D_i that has not yet been examined. The variable i holds the index of the currently selected variable. In each step (even after backtrack), the not yet assigned variables are reordered according to the variable ordering heuristic. Thereby the variable that should be labeled next is placed to the i-th position. The function SELECT-VALUE selects using the value ordering heuristic a value from the auxiliary domain of x_i, i.e., from $d(i)$, and removes it. If such a variable does not exist, the algorithm backtracks to the variable x_{i-1}. Otherwise it continues with the next variable.

Figure 3.2 shows an implementation of backtracking in a CLP environment. The predicate select_variable/3 selects a variable and select_value/2 assigns it a value from its domain. The operator #=/2 assigns a value to the given variable, and the operator ##/2 removes a value from the domain of the variable. The implementation is very easy because it is usually built-in. The only predicates that have to be implemented are the predicates that select variables and values according to the ordering heuristics.

A node of the search tree may correspond not only to a choice between selection or removal of a particular variable as in the presented pseudo-code, or to a choice between all values. For

```
labeling( [] ).
labeling( Variables ) :–
        select_variable( Variables, Var, Rest ),
        select_value( Var, Value ),
        (
            Var #= Value, Rest1 = Rest
        ;
            Var ## Value, Rest1 = Variables
        ),
        labeling( Rest1 ).
```

Figure 3.2: CLP implementation of backtracking with constraint propagation.

example, it is possible to make disjoint choices by domain splitting, i.e., whether the suitable value will be in the lower or upper half of the domain.

3.2.3 Limited Discrepancy Search Algorithm

For many problems of practical interest, carefully tuned value and variable ordering heuristics successfully help to find a solution. But as shown both theoretically and experimentally in [13], chronological backtracking is not very effective utilization of accurate value ordering heuristics. A good heuristic may make mistakes very rarely. But if it ever makes a mistake high in the search tree, traditional backtracking can get too far from the solution and may not have enough time to find it. In other words, chronological backtracking puts tremendous importance on the heuristics early in the search and relatively light importance deep in the search.

The heuristics would have led to a solution if only they had not made a small number of "wrong turns" that got the search off the right track. Therefore they should be followed at all but small number of decision points. Firstly, the search should try all paths that differ in exactly one decision point. If that fails, it should try paths that differ in two decision points, and so on. If the number of wrong turns is small, the solution is fairly quickly found.

The decision points in which the algorithm does not obey the heuristics are called *discrepancies*. *Limited discrepancy search (LDS)* [13] iteratively searches the space with an increasing limit of allowed discrepancies on the path, starting with zero. The heuristics are the least reliable in the beginning of the search, because the current partial solution is still too small to evaluate properly. Therefore the algorithm starts with discrepancies from the top.

The above given description applies without addition to the CSPs with the domains of size two. Here it is simple to define what a discrepancy means. If the heuristic prefers one value, the discrepancy is the selection of the other value. The situation is different for the CSPs with larger domains. A decision, what exactly the discrepancy is, needs to be made. A discrepancy may be the selection of the second most preferred value, or it may be the selection of any other value than the most preferred one. In the former case, the remaining values may be skipped from the search. Or a selection of the third most preferred value may be counted as two discrepancies, the selection of the fourth most preferred value, three discrepancies, and so on.

LDS is usually already implemented in standard CLP environments. For example, in SICStus Prolog [17], only the maximal number of discrepancies needs to be specified. A possible implementation of one iteration of LDS with the last mentioned type of discrepancies in imperative paradigm is shown in Figure 3.3. The implementation is similar to backtracking in Figure 3.1 with

Function LIMITED-DISCREPANCY-SEARCH
Input: A CSP (X, D, C), a discrepancy limit l
Output: Either a solution, or a notifi cation that no solution was found

```
d(i) ← D_i for 1 ≤ i ≤ n                                  (copy all domains)
i ← 1                                                     (initialize variable counter)
l(i) ← 0 for 1 ≤ i ≤ n                                    (initialize discrepancy counters)
while 1 ≤ i ≤ n do
    reorder variables x_i, ..., x_n according to variable ordering heuristic
    if l(i) = 0 then
        l(i) ← l                                          (make maximal discrepancy)
        l ← 0                                             (no further discrepancies may be made)
    else
        l(i) ← l(i) − 1                                   (make one less discrepancies)
        l ← l + 1                                         (one more allowed discrepancy for successors)
    x_i ← SELECT-VALUE(l(i))                              (select l(i)-th value)
    if x_i is null then                                   (no value selected)
        i ← maximal i such that l(i) > 0 if exists, else 0  (backtrack)
        if i > 0 then
            reset l and each d(k) and l(k), k > i, to its state before x_i was last instantiated
    else
        i ← i + 1                                         (step forward)
end while
if i = 0 then
    return "no solution"
else
    return instantiated values of {x_1, ..., x_n}
```

Figure 3.3: The limited discrepancy search algorithm.

the following modifications. For each variable, a discrepancy counter is maintained which holds the number of discrepancies made by the last assignment to this variable. The actual value of l determines the maximal number of discrepancies that may still be made. When a variable is chosen, the l-th most preferred value is selected. Therefore the maximal possible number of discrepancies is made (search should try first the paths with discrepancies in the beginning). When no such a value exists (either the domain is empty or there are not enough consistent values), a backtrack occurs. But the algorithm does not backtrack to the previous variable. It backtracks to the variable which has still untried values in its domain and they may be tried within the discrepancy limit. This variable may be easily recognized — it is the last variable x_i for which $l(i) > 0$ holds. For such a variable the value corresponding to the last tried number of discrepancies minus one is selected. The search continues with the following variables and the value of l increased by 1.

LDS was shown to be effective on some hard problems, like job shop scheduling or random 3-SAT problems [12]. In general, the number of nodes explored by one iteration of LDS with discrepancy limit of l is $\binom{d}{k}$, where d is the depth of the search tree.

Some improvements of LDS were also developed, like *improved LDS (ILDS)* [22], which tries to remove redundancy of the original LDS, and *depth-bounded discrepancy search (DDS)* [40], which in the i-th iteration explores those branches on which discrepancies occur at a depth of $(i-1)$ or less.

3.3 Tree Search with Consistency Techniques

The efficiency of tree search methods may be significantly improved with the help of consistency techniques. There are two basic ways of employing consistency techniques together with tree search methods. Look back schemes are invoked when the algorithm is preparing to backtrack after encountering a dead-end. Look ahead schemes can be employed whenever the algorithm is preparing to assign a value to the next variable.

3.3.1 Look Back Schemes

The *look-back schemes* employ consistency checks between already instantiated variables. Two classes of algorithms can be distinguished:

- *Intelligent backtracking algorithms*, like backjumping [7] utilize consistency techniques to avoid thrashing. When fail occurs and no value is left in the domain of the current variable, they detect conflicting variables and backtrack directly to the source of the failure, i.e., to the most recent conflicting variable. That is, they decide how far to backtrack.
- *Conflict recording algorithms*, like *backchecking* and *backmarking* [11] try to avoid redundant work by recording the reasons for the dead-ends, so that the same conflicts will not arise again later in the search.

There is no straightforward way how these schemes could be implemented in CLP environments. An implementation of these techniques needs an ability to detect and record conflicting constraints. This is often not available to the users of standard CLP(FD) libraries [17, 15].

3.3.2 Look Ahead Schemes

To detect a future conflict in advance, the *look-ahead schemes* were introduced. They look ahead whether the current branch of the search leads to a dead-end or not. After each assignment of a variable, its value is propagated into the domains of unassigned variables (*future variables*). All values that are not consistent (wrt. the given amount of constraint propagation) with the partial assignment need not be tried and are (temporarily) eliminated. When all values of an unassigned variable are removed, the current assignment can not be a part of a solution. The algorithm must backtrack. Consequently, dead-ends occur earlier in the search, and the search space may be significantly reduced. In general, the stronger the level of constraint propagation is used, the smaller the search space is explored, but the higher is the computational overhead. The consistency techniques can be (and usually are) applied also before the search process is initiated.

The implementation is an extension of the algorithm of chronological backtracking (see Figure 3.1). There are only two differences. The function SELECT-VALUE here selects (and removes) values that are consistent not only with the current partial solution, but also with the future variables. That is the given degree of propagation must not remove all values of any of these variables. Inconsistent values are removed and skipped. Within the function, the selected value is propagated into auxiliary value domains of future variables $x_{i+1}, \ldots, x_n$. If no consistent value exists, backtracking occurs. Upon backtracking, the algorithm resets each auxiliary value domain $d(k), k \geq i$, in order to rescind the elimination of values that were not consistent with no longer valid current partial assignment. That is, the other difference is that not only $d(i)$ is reset to its state before x_{i-1} was last instantiated, but all $d(k), k \geq i$. Implementations usually use n copies of each $d(i)$, one for each level in the search tree, in order to permit the reset action to be performed efficiently.

The implementation in a CLP language is identical to the implementation of standard backtracking (see Figure 3.2). Constraint propagation is covered by the posted constraints and is invoked whenever a domain of a variable in its scope is changed.

Forward Checking

A simple example of look ahead strategy is *forward checking* [11]. It performs constraint propagation of the current instantiation (*past variables*) into each single future variable. That is, the function SELECT-VALUE checks all constraints that refer to the current variable, some past variables, and only one future variable.

The implementation is shown in Figure 3.4. The function successively selects (and removes)

Function SELECT-VALUE
while $d(i)$ is not empty
  select according to value ordering heuristic element $a \in d(i)$ and remove it
  $empty\text{-}domain \leftarrow$ false
  foreach constraint c_m that refers to x_i and only one future variable $x_k (i < k \leq n)$
    foreach value $b \in d(k)$ (check each value)
      if not CONSISTENT$(c_m, x_1, \ldots, x_{i-1}, x_i = a, x_k = b)$ **then**
        remove b from $d(k)$
    end foreach
    if $d(k)$ is empty **then** (some future variable has now empty domain)
      $empty\text{-}domain =$ true
      break
  end foreach
  if $empty\text{-}domain$ **then**
    reset each $d(k), i < k \leq n$ to its state before a was selected
  else
    return a
end while
return null (no consistent value found)

Figure 3.4: The SELECT-VALUE function for backtracking with forward checking.

values from the domain of current variable x_i (according to the value ordering heuristic). For each value $a \in d(i)$, all values of future variables that are inconsistent with the above mentioned constraints are removed. The function CONSISTENT checks whether the current partial assignment to $x_1, \ldots, x_{i-1}$, extended by $x_i = a, x_k = b$ satisfies the constraint c_m. If not, the value b may be removed. If a domain is emptied, the current value a can not be part of a solution and another one must be selected. If all values of the current variable x_i are removed, the search must backtrack to the previous variable and hence null is returned.

3.4 Incomplete Search

Incomplete search methods do not explore the whole search space. They search the space either non-systematically or in a systematic manner, but with a limit on some resource. They may not provide a solution but their computational time is reasonably reduced. They can not be applied to find all solutions or to prove that no solution exits. However, they may be sufficient when just some solution is needed. Another application is to seek a feasible solution of an optimization problem (see the next chapter).

For example, the search may be limited by a maximal allowed computational time. When this time is exceeded, the search is terminated. The limited discrepancy search may be limited by the number of iterations, i.e., by the maximal allowed discrepancy limit. Or the number of

backtracks of a tree search algorithm may be recorded and terminate the search when it exceeds a specified limit. This method is referred to as *bounded backtrack search* [15].

Another incomplete tree search method is so called *credit search* [15]. It starts from the root with a certain amount of credits. This credit is split among the child nodes, their credit between their child nodes, and so on. A single unit of credit can not be split any further: subtrees provided with only a single credit are not allowed any nondeterministic choices, only one path through these subtrees can be explored. Subtrees for which no credit is left are not visited. Therefore the number of nondeterministic choices throughout the search is limited by the given credit.

All these incomplete variants of tree search algorithms may be easily implemented in CLP. Very promising incomplete search methods, not based on tree search, are so called *local search* methods. They will be introduced in the next chapter because they behave generally better for optimization problems [20].

Chapter 4

Algorithms for Optimization Problems

In this chapter, the basic algorithms for optimization problems are presented. They seek not one or all solutions, but the best solution according to a criterion. There is a variety of optimization algorithms but still no one is efficient enough. In the following sections the algorithms for both CSOPs and VCSPs are introduced. In the end of this chapter, the algorithms for VVCSPs are discussed, since the next part of the thesis is devoted to an algorithm for this kind of problem.

4.1 Algorithms for Constraint Satisfaction Optimization Problems

There are two basic approaches to solving CSOPs. The *constructive algorithms* gradually extend a partial solution to a complete one. They are based on tree search algorithms for satisfaction problems and may benefit from constraint propagation. The *iterative repair algorithms* start with an initial solution and iteratively alter its values to improve the current cost. Some hybrid algorithms trying to get advantages of both approaches have been proposed recently. They are described in the last section.

4.1.1 Constructive Algorithms

The basic constructive algorithm for optimization problems is a so called *branch and bound (B&B)* algorithm. It iteratively seeks a solution applying chronological backtracking with constraint propagation, or any other tree search algorithm. It records the best so far found solution and its cost (called the *upper bound*). Initially, the upper bound is $\top$ (the maximal possible value).

In each iteration, the B&B tries to find a solution with a better cost than the upper bound. It employs a heuristic function evaluating each partial solution. This function represents an under estimate (in case of minimization) of the cost function for the best solution that may be obtained from the current partial solution. This function is named a *lower bound*.

Each time a value is assigned to the variable, the lower bound for the current assignment is computed. Clearly, if the value exceeds the upper bound, the subtree under the current partial assignment can not contain a solution with a better cost. It is immediately pruned. Naturally, the subtree which contains no solution is pruned as well due to constraint propagation.

A possible implementation of B&B is shown in Figure 4.1. The domain variable $cost$ is constrained to hold the cost of the solution. Therefore the heuristic function is covered by constraint propagation into the domain of $cost$. The lower bound is equal to the minimal element in the domain of $cost$. First solution is found without any upper bound. The subsequent solutions must have smaller cost than the value of the variable $bound$. This is repeated until such a better solutions are being found.

Function BRANCH-AND-BOUND
Input: A CSP (X, D, C), a variable $cost$
Output: Either a best found solution or fail.

$bound \leftarrow \top$
$solution \leftarrow$ fail
repeat
  $prevsolution \leftarrow solution$
  $solution \leftarrow$ SOLVE$(X, D, C \cup \{cost < bound\})$
  $bound \leftarrow$ value of $cost$ in solution (if any)
until $solution =$ fail
return $prevsolution$

Figure 4.1: The branch and bound algorithm.

Figure 4.2 shows a possible implementation in a CLP environment. First a domain variable $Cost$ is bounded by the specified bound. The initial value of $Bound$ may be specified by a user. It is usually sup, as mentioned above. The predicate findall/3 [17] instantiates Variables and Cost with the help of labeling/1. It saves their values into Solution and FoundCost, respectively, and leaves them unassigned. The cut operator ensures that only the first solution is found. If no solution exists, the predicate tries to unify an empty list with the third argument. This results in a fail. If a solution has been found, it is saved employing a retract/assert mechanism. Then the predicate branch_and_bound/3 is called recursively for the newly found bound. If no solution is found, the second clause of branch_and_bound/3 is called. It returns the best found solution and its cost.

```
branch_and_bound( Bound, Variables, Cost ) :–
        Cost #< Bound,
        findall( [ Variables-Cost ], ( labeling( Variables ), ! ), [ Solution-FoundCost ] ),
        !,
        retract( solution( _, _ ) ),
        assert( solution( Solution, FoundCost ) ),
        branch_and_bound( FoundCost, Variables, Cost ).
branch_and_bound( _, Variables, Cost ) :–
        solution( Variables, Cost ), !.
```

Figure 4.2: CLP implementation of branch and bound.

Obviously, the quality of the heuristic function is very important and significantly influences the efficiency of the search. Also the closer the initial solution is to the optimal, the quicker the optimal or a good enough solution is found.

To ensure reasonable computation time, branch and bound is often limited by a time-out. Also there may be a requirement that the cost of a solution must be less than the upper bound by at least some minimum to be accepted. Therefore the algorithm more quickly approaches the optimum. More about B&B and its extensions can be found in [14].

4.1.2 Iterative Repair Algorithms

Iterative repair methods start with an initial solution, found by some other approach. Or it may be found randomly. They incrementally alter the values to get a "better" one, and eventually an optimal or at least a good enough solution. Their non-systematic nature generally voids the guarantee of "completeness", but they are often able to get quickly close to the optimal solution and overcome the algorithms based on tree search [20].

These algorithms may be applied to satisfaction problems, as well. The search will move throughout all assignments and the quality of the assignment will be determined by the number of violated constraints.

Important iterative repair methods are the so called *local search methods* [28, 39]. Local search is based on making small (local) changes in assignments to variables. A way of moving from one solution to another is called a *move*. A move is problem-specific. A set of all solutions that differ from the current one in only one move is called a *neighborhood*. In each iteration, a solution is selected from the current neighborhood (typically a better one). And it becomes the current solution. If the algorithm selected only better solutions all the time, it could get entrapped in so called *local optimum*. It is a solution whose neighborhood contains only solutions with equal or worse cost, and it is not the optimal solution (*global optimum*). To avoid this state, local search is often equipped with various meta-heuristics randomizing the search, i.e., allowing also worse neigbors to be selected under certain conditions The algorithm is usually bounded by some stopping criterion. It may be based on computational time, number of moves or some other indicator.

The implementation of local search is complicated in CLP environments, because it implicitly does not support the change of a value as soon as the variable has once been instantiated. CLP tools may contain a mechanism for this change, but it is still not so efficient as it would be in an imperative language and a special library needs to be employed, e.g., the *repair* library of ECL^{i}PS^e [15].

There are many local search algorithms, like *hill climbing*, *GRASP*, *tabu search*, *simulated annealing*, *evolutionary algorithms*, or *ant colony systems*. The more detailed survey of local search can be found in above mentioned [39].

Simulated Annealing

Simulated annealing was first introduced in [21]. It imitates the annealing process in which a material is first heated to a high temperature near melting point, and then slowly cooled down so that it will crystallize into a highly ordered state. The time spent at each temperature must be sufficiently long to allow a thermal equilibrium to be approached. The algorithm operates by allowing cost degrading moves but with decreasing probability of their acceptance. It picks a random neighbor solution and accepts it with the following probability (let the cost of the candidate solution be worse than the current cost by d):

1 if $d \leq 0$ (non-degrading move);
$e^{-\frac{d}{T}}$ if $d > 0$ (degrading move).

As can be seen, non-degrading moves are always accepted. Degrading moves are accepted with a probability which increases with a decrease in cost difference d, and a parameter T. The parameter T is known as the *temperature* and it controls the way the search operates. For large T, cost degrading moves have a higher chance of acceptance. For low T, the probability is lower and only moves that slightly degrade the cost have real chance of acceptance. The idea is to begin the search with a high temperature T and gradually lower (cool) T as the search progresses. As a consequence, the search is able to roam freely near the beginning; but later on, when the temperature is low, the search makes only non-degrading moves.

A simulated annealing is influenced by two factors. The first is how the temperature is reduced as the search progresses. A common method is to reduce the temperature after each move according to the rule $T \leftarrow T * factor$, where $0 \leq factor < 1$. Its value should be close to 1 to reduce more slowly, e.g., 0.99. As long as there are enough moves, a slower reduction of the temperature tends to result in better solutions. The second factor which controls the search is the initial temperature. If it is set too high, degrading moves will be accepted for a very long time, and consequently it will take long to produce a good solution. If the initial temperature is set too low, the search may get quickly entrapped in a local minimum. The example value is 1.0. A possible implementation of simulated annealing is shown in Figure 4.3.

Function SIMULATED-ANNEALING
Input: A CSP (X, D, C), a cost function $cost$, initial temperature T and cooling factor $factor$, number of moves max_moves
Output: A solution and its cost

generate initial solution A_0
$p \leftarrow 0, r \leftarrow 0$ (initialize variables)
for $i \leftarrow 1$ **to** max_moves **do**
  while $r \geq p$ **do**
    generate random neighbor solution A (choose candidate solution)
    $d \leftarrow cost(A) - cost(A_0)$ (compute its cost degradation)
    if $d \leq 0$ **then break** (non-degrading move)
    $p \leftarrow e^{-\frac{d}{T}}$ (compute probability of degrading move)
    generate random number $0 \leq r < 1$
  end while
  $T \leftarrow T * factor$ (cool temperature)
  $A_0 = A$
end for
return (solution A, $cost(A)$)

Figure 4.3: The simulated annealing algorithm.

Rearranging the probability acceptance equation, $T = -\frac{d}{\ln p(accept)}$ is obtained. Therefore if a cost degradation of d should be accepted with probability $p(accept)$, at the beginning of search, the initial value of T should be set to $-\frac{d}{\ln p(accept)}$. Normally, meta-heuristics require some tuning for the problem being solved. These parameters for simulated annealing are examples of what to tune. Another important parameter is the number of moves.

4.1.3 Hybrid Search

Constructive algorithms suffer from the disadvantage of tree search. If a wrong decision is made early in the search tree, it is necessary to undone it by backtracking. Many so far assigned values are thrown away. Look-back and look-ahead schemes try to reduce the importance of this problem, but they can not solve it completely. Also constructive algorithms have to heuristically estimate the final cost.

The local search algorithms decide upon complete solutions and the absence of systematicity allows them to modify assignments in any order. However, when a problem is tightly constrained, a local search algorithm may not get to a solution. Either the initial solution may be too far from it, or it may get entrapped in a local optimum.

While algorithms based on tree search are due to constraint propagation effective at finding solutions for tightly constrained problems with complex and overlapping constraints, the local search methods can be superior at optimization problems that are loosely constrained [20].

The above mentioned considerations lead to conclusion that the different characteristics of tree search and local search methods are complementary. Therefore promising hybrid algorithms trying to get the advantages of both approaches have been proposed recently. They can be found, e.g., in [19, 33, 20].

The following categories of hybrid approaches can be found in the literature [19]:

- performing a local search before or after a tree search method;
- performing a tree search algorithm improved with a local search at some point of the search: at each leaf of the tree (i.e., over complete assignments) or also nodes in the search tree (i.e., on partial assignments);
- performing an overall local search, and using a tree search algorithm either to select a candidate neighbor or to prune the search space.

Some of them can not be easily implemented in a CLP environment because the change of already instantiated variables is not supported by standard CLP libraries. A special library is needed, e.g., the *repair* library of ECL^{i}PS^e.

4.2 Algorithms for Valued Constraint Satisfaction Problems

Most of the current algorithms for VCSP are extensions of the branch and bound algorithm. The simplest lower bound can be computed as the cost of constraints violated by the current partial assignment. This value is called the *distance* of the current partial assignment from a total assignment satisfying all constraints. The extensions try to propagate soft constraints and improve this bound. The soft constraint propagation techniques look at the bound from local (e.g., *partial forward checking* [8] or *reversible directed arc consistency* [24]) or global (*Russian Doll search* [37]) point of view. Most of them are based on classical constraint propagation. They are usually, for the sake of simplicity, proposed for binary weighted CSP. But they can be easily extended for non-binary problems [27] and more general valuation structures [4]. Unfortunately, there is a lack of CLP tools that support soft constraints. The available libraries either do not extend a classic CLP(FD) library, like `clp(FD,S)` [10], or they support only a restricted number of soft constraints (see Chapter 7).

Another class of algorithms that can be applied to VCSP are local search methods. They may move throughout all assignments and minimize the valuation. To the author's knowledge, there are no improvements to local search that would deal with soft constraints and possibly propagate them. It can be guided only on the basis of the final costs of neighbor solutions.

In this section, a basic soft constraint propagation technique is introduced. It is based on forward checking described in Section 3.3.2. This technique was employed in Purdue University timetabling problem described in Chapter 7.

4.2.1 Partial Forward Checking

For the sake of simplicity, the following description is given for weighted CSPs. But it can be easily generalized to a VCSP. Partial forward checking maintains so called *inconsistency count (IC)* for each value of a future variable (not yet assigned variable). Inconsistency count for a value a of a future variable x, denoted $ic_x(a)$, is the cost of all constraints which refer only to x and past variables (already instantiated variables), and would be violated by the value a. In other words, x

is the only future variable in the scope. The lower bound is computed as the sum of the distance and the minimal inconsistency counts of values for each future variable.

Function SELECT-VALUE
while $d(i)$ is not empty
  select according to value ordering heuristic element $a \in d(i)$ and remove it
  $empty\text{-}domain \leftarrow$ false
  $new\text{-}distance \leftarrow distance + ic_i(a)$ (compute new distance)
  $lower\text{-}bound \leftarrow new\text{-}distance + \sum_{i<k\leq n} \min_{j\in d(k)} ic_k(j)$
  if $lower\text{-}bound \geq upper\text{-}bound$ **then** (lower-bound exceeds upper-bound)
    continue (restart while loop with another value)
  foreach constraint c_m that refers to x_i and only one future variable $x_k (i < k \leq n)$
    foreach value $b \in d(k)$ (check each value)
      if not CONSISTENT$(c_m, x_1, \ldots, x_{i-1}, x_i = a, x_k = b)$ **then**
        $ic_k(b) \leftarrow ic_k(b)+$ cost of c_m (increase incon. count for inconsistent value)
      if $new\text{-}distance + ic_k(b) \geq upper\text{-}bound$ **then**
        remove b from $d(k)$ (value has too many inconsistencies)
    end foreach
    if $d(k)$ is empty **then** (some future variable has now empty domain)
      $empty\text{-}domain =$ true
      break
  end foreach
  if $empty\text{-}domain$ **then**
    reset each $d(k), i < k \leq n$ to its state before a was selected
  else
    $distance \leftarrow new\text{-}distance$ (update distance)
    return a
end while
return null (no applicable value found)

Figure 4.4: The SELECT-VALUE function for the B&B with partial forward checking.

One iteration of branch and bound with partial forward checking is an extension of backtracking with constraint propagation. It differs in the implementation of the function SELECT-VALUE. It is shown in Figure 4.4. Another difference is the initialization of new auxiliary variables. In the beginning of each iteration, the value of $distance$ must be reset to zero. And all inconsistency counts must be reset to the costs of unary constraints violated by the current value. If no such unary constraint exists, they will be equal to zero. The function operates as follows. It successively selects (and removes) values from the domain of current variable x_i (according to value ordering heuristic). For each value $a \in d(i)$, new distance is computed and stored into the variable $new\text{-}distance$. Then the lower bound corresponding to the new partial assignment is computed. It is the sum of the current distance and minimal inconsistency counts for each future variable x_k. If the computed lower bound exceeds the upper bound, a is skipped. Otherwise the inconsistency count of each future variable b is updated. If the sum $new\text{-}distance + ic_k(b)$ is then greater or equal to the current upper bound, the value b can not be selected in the future and therefore it can be removed. If a domain of some future variable is emptied, i.e., all its values had too large inconsistency counts, the current value a is not applicable and another one must be selected. Otherwise the current distance is updated and the value a is returned as a selected value. If the value a has been found unsuitable and no another value exists in the domain of x_i, the search must backtrack to the previous variable and hence null is returned.

Inconsistency counts can be used not only to prune the subtrees that can not improve the current best solution but also to state the value ordering heuristic. Values having smaller counts are tried first, since they are more likely to be included in an optimal solution than values with greater counts.

There are also further improvements of the lower bound that try to include also inconsistencies among future variables, like the above mentioned reversible directed arc consistency or Russian Doll search.

4.2.2 Algorithms for Variable Valued Constraint Satisfaction Problems

Most papers consider the problem of finding a total assignment satisfying hard constraints and minimizing its valuation. Not many works are devoted to the problem of finding a maximal partial solution satisfying all constraints. To the author's knowledge, the only papers that deal with this problem are [37] and [25]. Both describe the same application of VVCSP to the daily management of an earth observation satellite. The proposed idea is as follows: the VVCSP is transformed into an equivalent VCSP as described in Section 2.2.2 and a method for solving VCSPs is applied. They employ the Russian Doll search algorithm with the following value ordering heuristic: the rejection value is tried last and the values from the so far best solution are tried first. Clearly, besides soft constraint propagation algorithms, also a local search method may be applied after the transformation.

A trivial and not very effective approach to VVCSP is to employ an incomplete version of a tree search algorithm 3.4 and record the best found partial solution. But because the VVCSP is over-constrained, this may fail during the initial constraint propagation prior to the search or during the labeling. Consequently, no solution is found.

The rejection value can not be efficiently implemented in the current CLP(FD) libraries [17, 15]. They do not support efficient extension of a domain by a value that would be excluded from all constraints. A promising algorithm that tries to find a maximal partial solution and can be easily implemented in a CLP environment is the main subject of this thesis as is proposed in the next chapter.

Chapter 5

Limited Assignment Number Search Algorithm

In this chapter, a *limited assignment number (LAN) search algorithm* is introduced. It is an iterative repair search algorithm based on a special incomplete version of backtracking with constraint propagation. It is devoted to tightly constrained problems where it is difficult or impossible to find a solution. The heuristics may not be good enough, constraint propagation may be weak, or the problem may be over-constrained. It tries to cope with the situation when a wrong decision is made early and the conflict is encountered too deep in the search tree.

Example 5.1 Let the variables X, Y have domains $\{2, 3, 4, 5\}$, $\{3, 4, 6, 7\}$, resp. It may happen that the relation $X = Y$ holds in all possible solutions of the problem which contains these variables. However, it can be discovered too late. If the search starts with the assignment $X = 2$, it is obviously a wrong decision. The search continues and the relation $X = Y$ is derived. Unfortunately, it may be a reason for an expensive backtracking. For a large scale problems, it may even mean that a solution can not be found in a reasonable time. In this situation, LAN search would be able to return a partial solution with the variable Y unassigned. Certainly, it is a better result than no solution at all.

Other possibility is that the problem may be over-constrained. Let the domain of X be $\{1, 2\}$. Here it is even impossible to return any complete solution. A partial solution could be the only result if the problem is not treated as valued constraint satisfaction problem.

Standard backtracking would throw away many values during the backtracking to the variable X. LAN search utilizes the found values in the next iteration to improve the last found partial solution. It develops its own ordering heuristics based on the previous results. They are applied in subsequent iterations to improve the current partial solution. These heuristics can be influenced after each iteration by a user who can direct continuation of the search. Another possibility is to relax some constraints in the problem and continue with the relaxed problem in subsequent iterations. The advantage of user advice was discussed in Section 2.2.2.

An importance of this algorithm consists in its natural and easy implementation in a CLP language which already provides built-in consistency techniques and the backtracking mechanism. While CLP environments usually can not cope with over-constrained problems, LAN search is capable also of solving such problems. When a problem is too tightly constrained, a local search algorithm may not get to a solution, if the initial solution is too far from it, and may get entrapped in a local optimum. The LAN search algorithm tries to avoid it employing constraint propagation to guide the search.

5.1 One Iteration

One iteration of LAN search is a special, incomplete version of chronological backtracking with constraint propagation. The algorithm maintains for each variable a count of how many times a value has been assigned to it. A *limit* is set on this count. If the limit is exceeded, the variable is left unassigned and the search continues with the other variables. Such a variable will be denoted as a *repair variable*. Clearly, the variables x_j, x_k in example 5.1 would be repeatedly unsuccessfully instantiated, changing values of a few previous variables by backtracks. Eventually, their number of assignments would exceed the limit and they would be declared repair variables. Labeling of repair variables is not processed even after backtracking. These variables are skipped. As a result of this search, a partial assignment of variables is obtained together with the set of the remaining unassigned (repair) variables.

Limiting the number of attempts to assign a value to each variable ensures the finiteness of this incomplete search. As each of n variables can be tried l times, one iteration of LAN search is of *linear time complexity* $\mathcal{O}(ln)$.

Function LAN-SEARCH-ITERATION
Input: A CSP (X, D, C), a limit l
Output: Indices of unassigned variables and a partial solution, or index of a conflicting variable

```
d(i) ← D_i for 1 ≤ i ≤ n                                   (copy all domains)
c_i ← 0 for 1 ≤ i ≤ n                                      (initialize assignment counter)
j ← 0                                                      (initialize direction variable)
i ← 1                                                      (initialize variable counter)
while 1 ≤ i ≤ n do
    reorder variables x_i, ..., x_n according to variable ordering heuristic   (select variable)
    if c_i ≥ l then
        i ← i + j                                          (skip repair variable)
    else
        c_i ← c_i + 1                                      (increase assignment counter)
        x_i ← SELECT-VALUE                                 (select consistent value)
        if x_i is null then                                (no value was returned)
            if i > 1 then
                reset each d(k), k ≥ i, to its state before x_{i-1} was last instantiated
            i ← i − 1;  j ← (−1)                           (backtrack)
        else
            i ← i + 1;  j ← (+1)                           (step forward)
end while
if i = 0 then
    E ← min_s{(c_s < l) ∧ (‖d(s)‖ > 1)}
    return (E, "no solution")                              (return conflicting variable)
else
    R ← {r|(‖d(r)‖ > 1)}
    return (R, assigned values of X − R)                   (return partial solution and
                                                            repair variables)
```

Figure 5.1: One iteration of the LAN search algorithm.

An implementation of one iteration of LAN search is shown in Figure 5.1. The input of the algorithm consists of a CSP (X, D, C) and a limit l. In the beginning, the algorithm creates auxiliary value domains and copies the values of the original domains into them. Next the algorithm goes

through all variables. In each step, it first reorders the remaining variables according to the (dynamic) variable ordering heuristics. Thereby the variable that should be tried next is placed to the i-th position. If the limit of the variable has been exceeded, the variable is skipped. There are now two possibilities. Either the algorithm has just backtracked, or the repair variable has just been selected to be labeled. In the former case, the backtracking goes to the previous variable. In the latter case, the search continues to the following variable. The current direction of movement over variables is hold in the variable j. Its value is 1, when the algorithm is moving forward. When the value is (-1), the algorithm is moving backwards. If the limit of a variable has not yet been exceeded, the counter is increased by one and a value is selected. The function SELECT-VALUE behaves identically to the same function in backtracking with constraint propagation (see Section 3.3.2). If a value is successfully selected and propagated, the algorithm continues with the next variable. Otherwise it resets the auxiliary value domains, and goes to the previous variable (it backtracks). In both cases, it sets properly the value of j. The algorithm can possibly backtrack to $i = 0$ (no value was assigned). The only result of such a search is a so called conflicting variable (see Section 5.3). However, the algorithm usually ends with $i = n + 1$ (all variables were processed). Here, the output is a partial solution and a list of unassigned variables. Not all variables which exceeded the limit are repair variables in the end. Some may be instantiated because of constraint propagation.

5.2 Ordering Heuristics

Partial assignment computed in each iteration is used to define new variable and value ordering heuristics. They are used in subsequent iterations. The developed variable and value ordering heuristics are based on the following ideas:

- values of successfully assigned variables are preferred in the next iteration — once a suitable value for a variable has been found, it remains a promising assignment;
- unsuccessfully attempted values for any repair variable are demoted in the ordering so that they will be tried *last* in the subsequent iteration — a suitable value is more likely to occur among those that have not been tried;
- all repair variables are labeled *first* in the subsequent iteration — it may be difficult to assign a value to this variable, therefore, it should be given preference in labeling.

In the first iteration of the algorithm (*initial search*), there is a choice of applying either problem-specific heuristics or standard heuristics, such as first-fail (see Section 3.2.2) for variable ordering. In successive iterations (*repair searches*), heuristics based on the previous iteration are primarily used. Any ties are broken in favor of the initial search heuristics.

5.3 Conflicting Variable

There are two cases when LAN search does not help and all variables remain unassigned. Both occur only when the problem is over-constrained. First, the initial constraint propagation prior to labeling may cause a fail and a labeling (LAN search) is not processed at all. Here the user should relax some constraints and restart the algorithm. Second, a fail may also occur during LAN search. It happens if the domain of a variable x_s is emptied before its limit is exceeded and there is no variable to backtrack to. It means that all previous variables x_i are either repair variables ($c_i = l$) or they have been instantiated by constraint propagation, not by assignment ($\|d(i)\| = 1$). The variable x_s is denoted as a *conflicting variable*. In this case, the algorithm in

Figure 5.1 terminates with $i = 0$. The problem is again over-constrained. No value can be assigned to the conflicting variable without violating some constraint — there is no variable assigned by labeling, i.e., no extra constraint was created by a value assignment.

There are two alternatives to cope with the conflicting variable:

1. some constraints can be relaxed based on user input and the iteration can be restarted;
2. the conflicting variable may be moved to further place in the variable ordering heuristic and the iteration can be restarted.

The first option is aimed to make the problem solvable. The second option possibly allows computing of a partial solution.

5.4 LAN Search and CLP

LAN search may be easily implemented in CLP environment, which already provides built-in consistency techniques and the backtracking mechanism. Also while CLP environments usually can not cope with over-constrained problems, LAN search is capable also of solving such problems.

```
labeling( [], _, [] ).
labeling( Variables, Limit ) :-
        select_variable( Variables, Var, Rest ),
        get_and_increment_counter( Var, Counter ),
        ( Counter < Limit ->
            select_value( Var, Value ),
            (
                Var #= Value, Rest1 = Rest
            ;
                Var ## Value, Rest1 = Variables
            )
        ;
            Rest1 = Rest,
        ),
        labeling( Rest1, Limit ).
```

Figure 5.2: CLP implementation of LAN search

Figure 5.2 shows how it is possible to extend a classical CLP labeling (written in boldface). The code includes one iteration of the algorithm. The predicate get_and_increment_counter/2 is used to maintain a counter for each variable. The other predicates are the same as in backtracking (see Figure 3.2). All counters must be initialized to zero before the labeling is processed. They can be implemented with the help of assert/retract mechanism, blackboard or any other primitives which allow in CLP to preserve values even during backtracking.

5.5 Possibility of User Interaction

The user may also manually modify the results after each iteration to influence the behavior of the heuristics, relax constraints to eliminate contradictory requirements, or change the problem definition. The options available for continuing the search are as follows:

1. processing the automated search as proposed above;
2. defining other values to be tried first or last based on user input, and process the repair search directed by the updated value ordering heuristics;
3. relaxing some hard constraints based on user input, and processing the initial search or the repair search;
4. addition or deletion of variables or constraints based on user input, and applying the repair search to reuse results of the former solution.

The first possibility is aimed at automated generation of a better assignment. Approach 2 allows the user to direct the search into parts of the search space where a complete assignment of variables might more easily be found. This approach can be easily included into the developed variable and value ordering heuristics. Step 3 can be useful if the user discovers a conflict among some hard constraints based on a partially generated output. The repair search can reuse most of the results from the former search and permute only some parts of the last partial assignment. It should be noted that there is often an advantage to a user directed search (e.g., in timetabling problems), since the user may be able to detect inconsistencies or propose a suitable assignment based on a partially generated solution (timetable). See, for example, the description of the Purdue University Timetabling Problem solver in Chapter 7. The last possibility introduces a new direction for the development of the algorithm aimed at incorporating changes to the problem input. This should be studied in detail as a part of future work.

5.6 Extensions of the Algorithm

Two extensions that were aimed to improve the behavior of the algorithm were proposed. Both try to reflect results of all previous iterations to the developed ordering heuristics, not only of the last one.

5.6.1 Variable Ordering

The idea of this extension is as follows. A counter for each variable is maintained which stores how many times it was among repair variables throughout all iterations. These counters help to develop more accurate variable ordering heuristic: the higher the counter is, the more critical the variable probably is, and the more preferred it should be. The repair variables from the last iteration should be still given preference, because a new solution will be close to the last one and the successfully assigned variables are not critical for the moment.

The maintained counter may be used in two ways: either only the repair variables from the last iteration should be sorted according to it, or also the successfully assigned variables with greater counter should be preferred. Ties are still broken by the initial heuristic.

5.6.2 Value Ordering

This extension has similar idea to the previous one, only it considers the value ordering. For each variable two lists of values are maintained: a list of all values from its domain that have been successfully assigned in any iteration, and a list of all unsuccessfully tried values in any iteration. The values from the former list are more promising than the values from the latter. As a consequence, the modified value ordering heuristic is obtained. The promising values derived from the last iteration will still be preferred. But ties will be broken according to the newly maintained

lists and then by initial heuristic. The values from the former list will be preferred, the values from the latter list will be discouraged.

Chapter 6

Experiments on Random Problems

First experiments with LAN search were processed on n-queens problem. They were not very successful. If the problem is solvable, it can be solved by backtracking with constraint propagation and LAN search application does not bring any improvement. If it is over-constrained, most of the variables are left unassigned. This problem is loosely constrained, i.e., there are not many local inconsistencies. Any wrong decision is quickly recognized applying a simple constraint propagation.

Next, the algorithm was experimentally evaluated on proposed random problems. Two different types of experiments were done. The first type was aimed at better understanding of the algorithm itself. Various variants of the algorithm were compared. The second type of experiments compared LAN search with other search algorithms that can be applied to VVCSP. In this chapter, the random problems are defined and stated as a CSP. The instances on which the experiments were accomplished are introduced. Finally, the experiments are described and the results are presented and discussed.

6.1 Random Placement Problem

This section includes the description of the random placement problem and its instances. In is divided into three subsections that contain the definition of the problem, its statement as a CSP, and generated instances, resp.

6.1.1 Description

The *random placement problem (RPP)* seeks to place a set of randomly generated rectangles (called *objects*) of different sizes into a larger rectangle (*placement area*) in such a way that no objects overlap and all object borders are parallel to the border of the placement area. In addition, a set of allowable placements can be randomly generated for each object.

The ratio between the size of the placement area and the total area of all objects will be denoted as the *filled area ratio*. The higher the filled area ratio is, the more constrained (possibly over-constrained) the problem is. For problems with the filled area ratio greater than one (or any over-constrained problem instance with a smaller filled area ratio), a solution with the maximum placed objects should be found.

The RPP was proposed because it allows to generate various instances of the problem similar to a trivial timetabling problem and LAN search was originally developed for this kind of problem. The correspondence is as follows: the object corresponds to a course to be timetabled — the x-coordinate to its time, the y-coordinate to its classroom. For example, a course taking three hours corresponds to an object with dimensions 3x1 (course should be taught in one classroom only).

Each course can be placed only in a classroom of sufficient capacity — each object will have a randomly generated lower y-bound.

6.1.2 Solution Approach

The RPP can be simply expressed as a CSP. Each object is represented by a pair of variables — x-coordinate and y-coordinate in the placement area. Each variable is given a domain according to the size of the placement area and in case of y-variable also according to the y-lower bound. The only necessary constraint is a global constraint ensuring that objects will not overlap. The suitable constraint is for example the *disjoint2* constraint from the CLP(*FD*) library of SICStus Prolog [6].

If the RPP is over-constrained, it can be expressed as a VVCSP, i.e., the aim is to maximize the number of assigned variables. It could be expressed also as a VCSP, making the global constraint soft and minimizing the number of overlapping objects. However, LAN search is not designed for general optimization problems, only for VVCSP. Therefore the first variant was chosen.

6.1.3 Problem Instances

Most experiments were accomplished on 8 sets by 50 problems, each concerning 200 objects. The sets are distinguished by the filled area ratio of problems in the set. It ranges from 75 %, 80 %, 85 %, . . . , to 110 %. Clearly all problems in the last two sets are over-constrained. However, this may also be true for other problems with the filled area ratio "close" to 100 %. Table 6.1 shows the ratio of different sizes of objects in a generated RPP. Table 6.2 shows the ratio of

size	2x1	3x1	4x1	6x1
ratio (%)	80.42	16.63	2.52	0.43

Table 6.1: The ratio of different sizes of objects.

lower bound	0	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.91
ratio (%)	31	8	4.4	7	20	10.3	6	2	11.3

Table 6.2: The ratio of different lower bounds of objects.

different lower bounds (as a portion of a maximal y-coordinate). The given ratios were chosen to correspond to the real-life timetabling problem described in the next chapter. The placement area size was the same for each problem in the same set and was chosen to meet the required filled area ratio. It is shown in Table 6.3.

fi lled area ratio (%)	75	80	85	90	95	100	105	110
placement area	40x15	40x14	38x14	35x14	36x13	35x13	33x13	33x12

Table 6.3: The size of placement area for different sets.

Some conclusions were verified on a set of 50 problems whose size was enlarged to have double sized domain. The average filled area ratio of these problems was 90 %. The size of the placement area was 70x28 and the problems included 800 objects.

All generated data instances together with the problem generator written in SICStus Prolog [17] are available on the enclosed CD (see Appendix A).

6.2 Accomplished Experiments

In this section, the accomplished experiments are described and the results given. First the experiments evaluating the LAN search algorithm itself are presented. Afterwards, the comparison with other algorithms is mentioned. The presented results were accomplished under Linux on a PC with a Pentium III 1 GHz processor with 1 GB of memory.

6.2.1 Different Parameters of LAN Search

The aim of these experiments on RPPs was to evaluate the behavior of LAN search, and to determine the best parameters of the algorithm. The algorithm was implemented in the CLP(FD) library of SICStus Prolog [6, 17] version 3.9.1.

The following variable and value ordering heuristics were used:

- variables were ordered according to the first-fail principle, breaking any ties by the order of variables (variables were ordered in a sequence of the pairs of object's variables, one after another);
- values were ordered in ascending order, because the objects have lower bound and therefore the lower the object is, the more space is left for other objects.

These initial heuristics were chosen as promising representatives of the standard heuristics used for CSPs.

Each problem was solved applying 50 iterations. In all experiments, changes of the average percentage of assigned variables (y-coordinates) wrt. the iteration number (x-coordinates) were explored. The average run time did not exceed a few seconds per iteration.

Limit on Number of Assignments

Figure 6.1 shows the runs of algorithm with different limits for the data set with 90 % filled area ratio. As expected, the amount of assigned variables increased with the iteration number. It was interesting to find out that quite small limits could give the best results. The greater limits made slightly better improvements at the beginning (see limit 300). However, smaller limits (see limit 5) were able to introduce better improvements and, in the end, find a slightly better solution. As can be seen in Figure 6.2, there is a lower bound on limit below which the results were not so good.

These conclusions were successfully verified on the set with double sized domain (results are presented in Figure 6.3) and also on different limits.

Number of Iterations

Figure 6.4 shows that 50 iterations are enough for the comparison. After around 35th iteration, no further significant progress was made. Also the lower limits did not overcome the limit of 5 even after 100 iterations.

Hardness of Problems

Other experiments on random problems were aimed at the analysis of the algorithm wrt. hardness of the problem and existence of the problem solution. Figure 6.5 shows the average percentage of assigned variables after each iteration on different problem sets. The complete assignment of variables was found for 49, 50, 47, 23, and 6 problems with the filled area ratio 75, 80, 85, 90, and 95 %, resp. No problem was completely solved in the data sets with the higher filled area ratio.

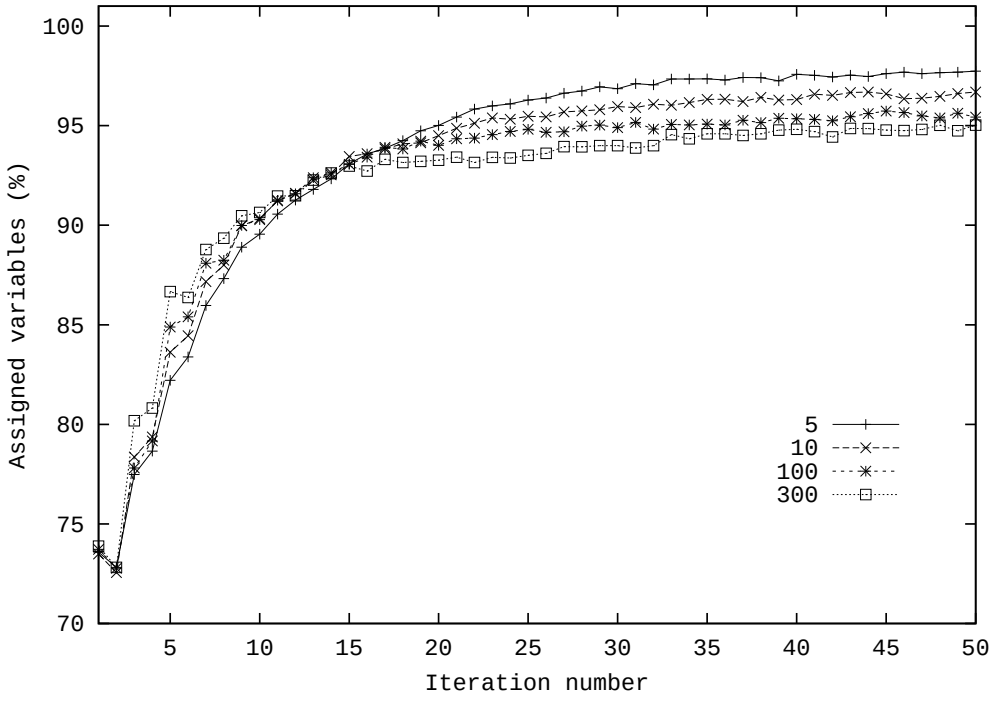

Figure 6.1: Comparison of different limits

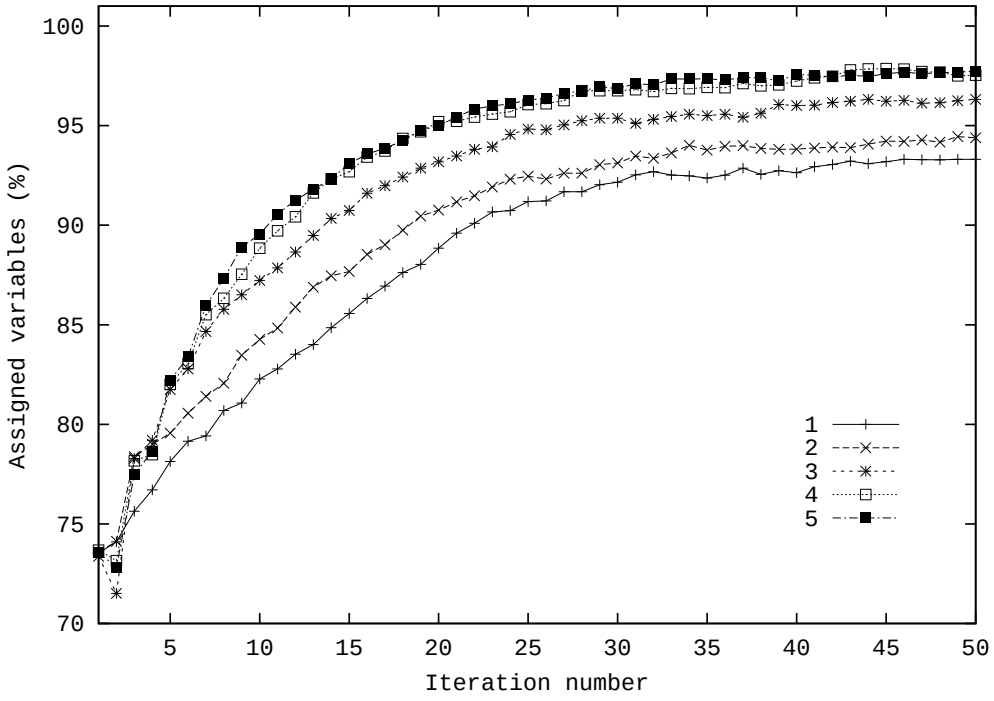

Figure 6.2: Comparison of different low limits

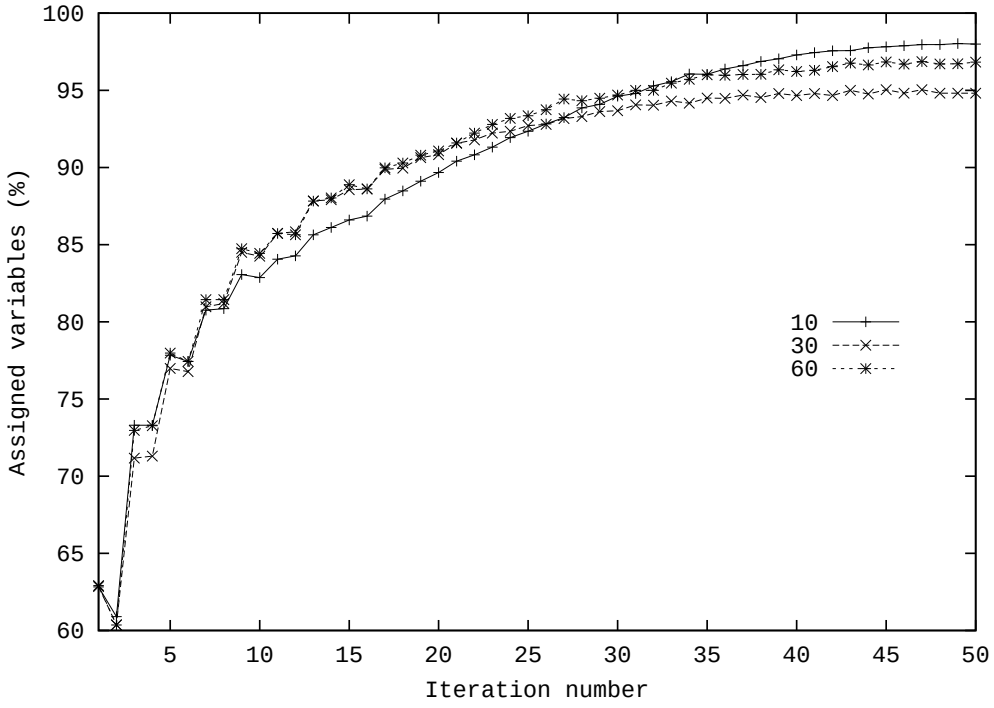

Figure 6.3: Different limits for double-sized domains

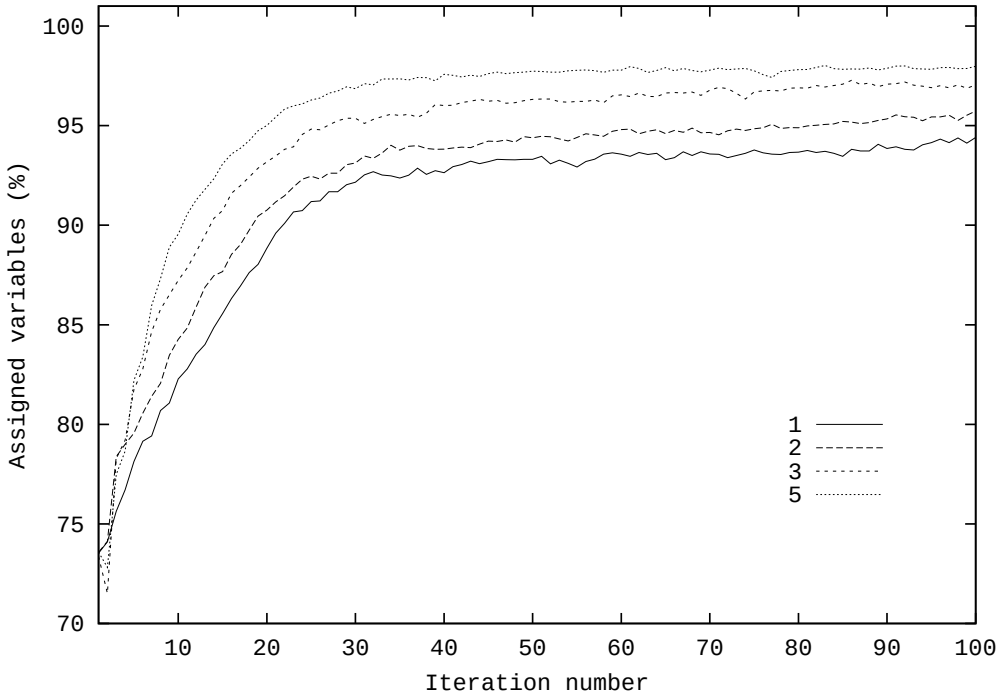

Figure 6.4: High number of iterations for some limits

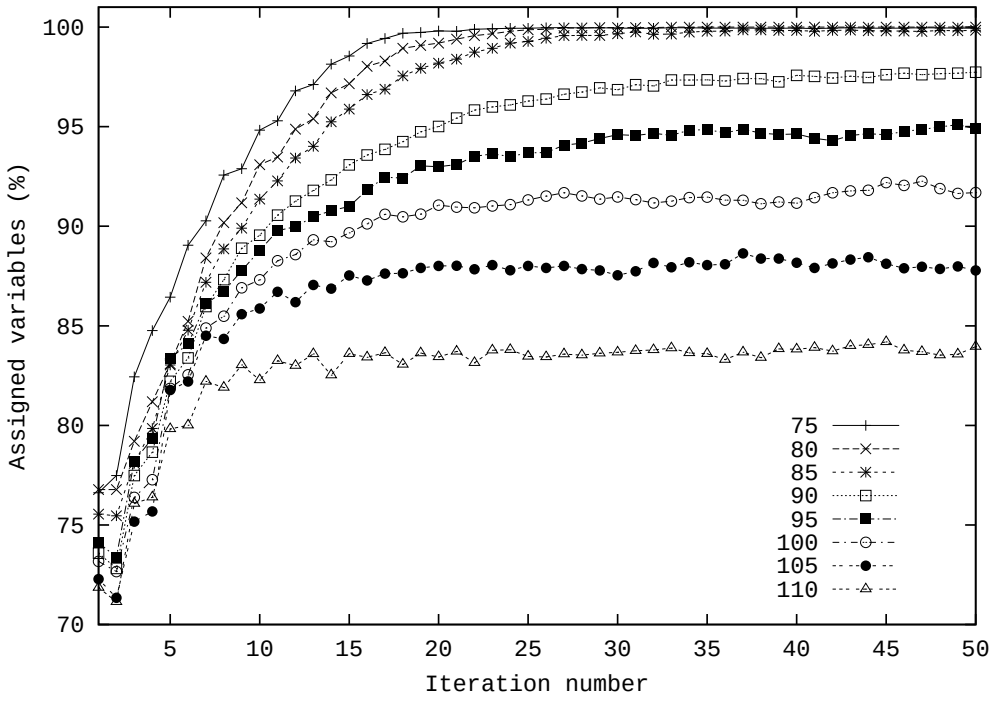

Figure 6.5: Problems with different filled area ratio

Results were not surprising: the more constrained the problem was, the less complete solution was found. At the same time, it is clear that it is not possible to assign all variables for many problems (at least 9 % of the variables can not be assigned for the problem with the filled area ratio of 110 %).

Extended Ordering Heuristics

Figure 6.6 shows the comparison of the original variable ordering heuristic and the two possible extensions mentioned in Section 5.6. The application of the extension to the ordering of all variables was significantly behind the other two variants. The explanation is simple. Many variables variables that are difficult to assign may not be among repair variables only thanks to the fact that they have been chosen preferentially because of the first-fail principle. Applying the extension to all variables would place them to the last positions (they were never among repair variables) and therefore they would get unassigned.

The application to the ordering of repair variables made almost no difference against the original variable ordering heuristic due to the same reason. Here the preference of variables that were more times unassigned is more appropriate. Still it did not prove to make any noticeable improvement. For some problems, the results were slightly better, for some worse.

The extension of the value ordering heuristic slightly worsened the results, as can be seen in Figure 6.7. The values that were successfully selected in any of the previous iterations, may not be proper for the current partial solution.

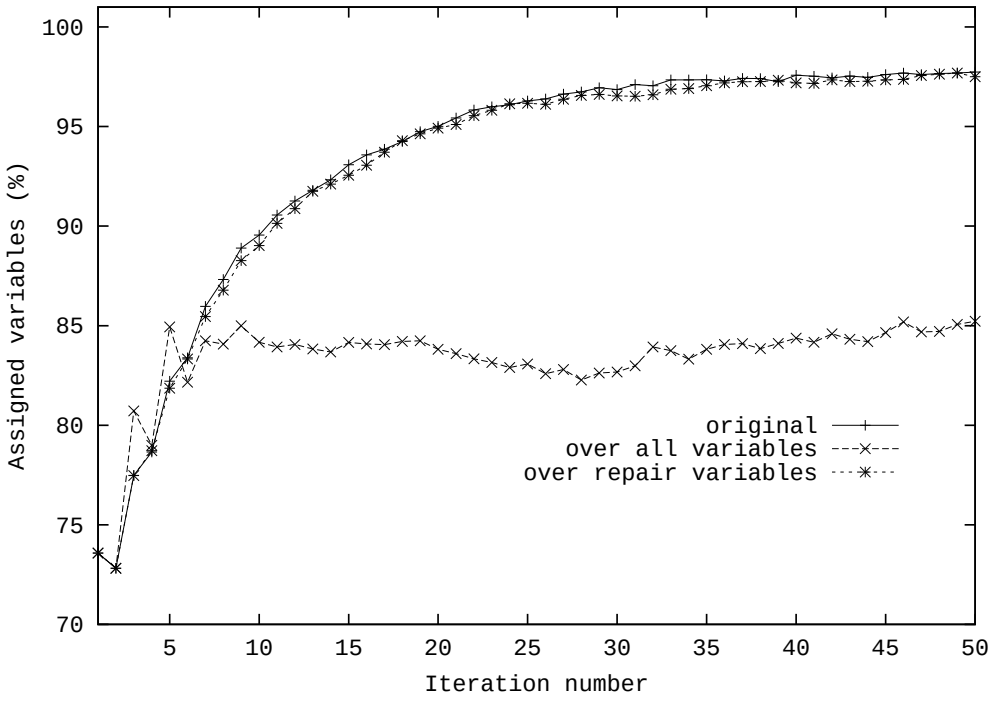

Figure 6.6: Comparison of variable ordering heuristics

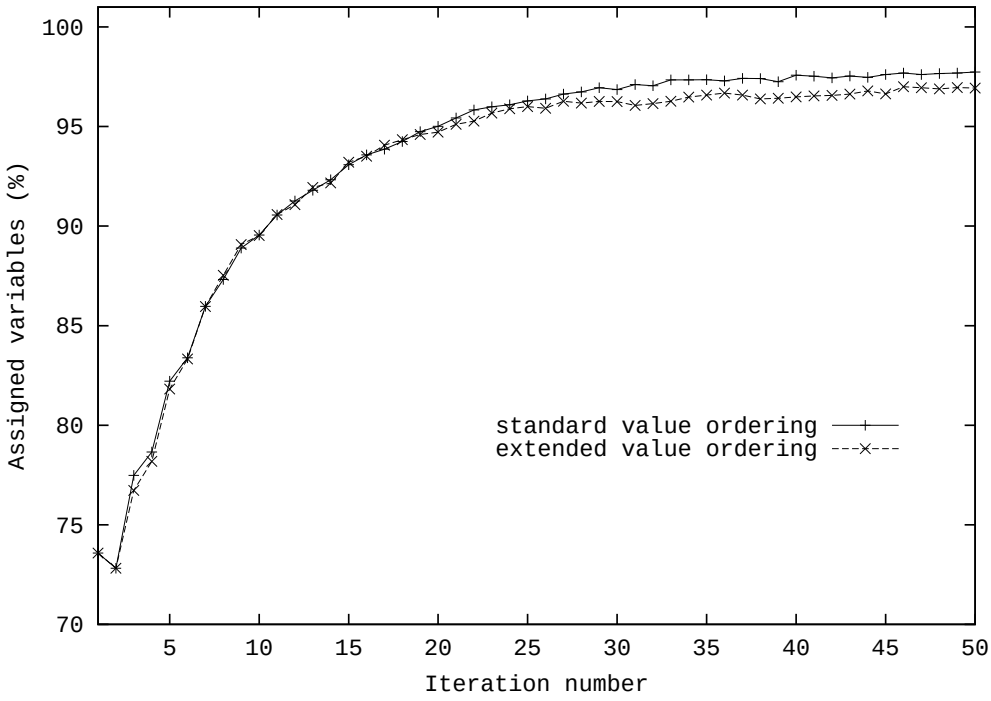

Figure 6.7: Comparison of value ordering heuristics

6.2.2 Comparison with Other Algorithms

In this subsection, the comparison of LAN search with some other algorithms that can be applied to find a maximal partial solution is presented. To compare with LAN search the following algorithms were chosen:

- backtracking with constraint propagation and maximal time of computation limited by five hours (BT);
- limited discrepancy search with constraint propagation and maximum of two iterations, because three iterations exceeded 10 hours (LDS);
- simulated annealing with maximum of $10,000$ iterations (more iterations did not made any substantial progress), initial temperature of 1.0 and cooling factor of 0.99 (SA).

The first two algorithms represent the tree search algorithms with constraint propagation. The incomplete versions (see Section 3.4) were applied, otherwise the time of computation would be too long. The last algorithm is the representative of iterative repair methods. The variable and value ordering heuristics for LDS and BT correspond with the initial heuristics of LAN search (see Section 6.2.1). LAN search, BT and LDS sought for the maximal partial solution. SA looked for the solution minimizing the number of overlapping objects. To enable the comparison of the results, the solution found by SA had to be expressed as a number of assigned variables. An object with the most overlaps was selected (ties broken randomly) and removed. Both its variables were declared repair. This was repeated until there were no overlapping objects. The initial solution of SA was generated with random x-coordinates and minimal y-coordinates. A neighbor assignment was any assignment that differed in a value of exactly one variable.

The incomplete versions of the tree search search algorithms were implemented in the same environment as LAN search, i.e., in the CLP(FD) library of SICStus Prolog [6] version 3.9.1. The simulated annealing algorithm was implemented in C++ for the purpose of an easier implementation (see Section 4.1.2).

All compared algorithms were used to solve problems of all eight standard sets of instances of the RPP. Figure 6.8 compares the achievement of individual algorithms. Both BT and LDS did not solve any problem. Numbers of problems that were completely solved by simulated annealing and LAN search can be found in Table 6.4. The average times in minutes taken by LAN search, LDS and SA are shown in Table 6.5.

Filled area ratio (%)	75	80	85	90	95	100	105	110
LAN	49	50	47	23	6	0	0	0
SA	47	44	40	22	13	6	0	0

Table 6.4: Number of problems solved by LAN search and SA.

Filled area ratio (%)	75	80	85	90	95	100	105	110
LAN	1.5	2.4	2.4	3	2	3	4	5
LDS	51	52	56	38	43	38	39	38
SA	0.15	0.25	0.43	1	1.7	2	1	0.8

Table 6.5: Average times (in minutes) taken by LAN search, LDS and SA.

As can be seen, LAN search keeps almost as well as SA. SA shows to be slightly better for most RPPs. Only for the less constrained LAN search overcomes it. However, LAN search significantly improves over backtracking and it also outperforms limited discrepancy search. LDS would need much better ordering heuristics to improve significantly. This would help to BT, as well. Also neither BT nor LDS are developed for over-constrained problems.

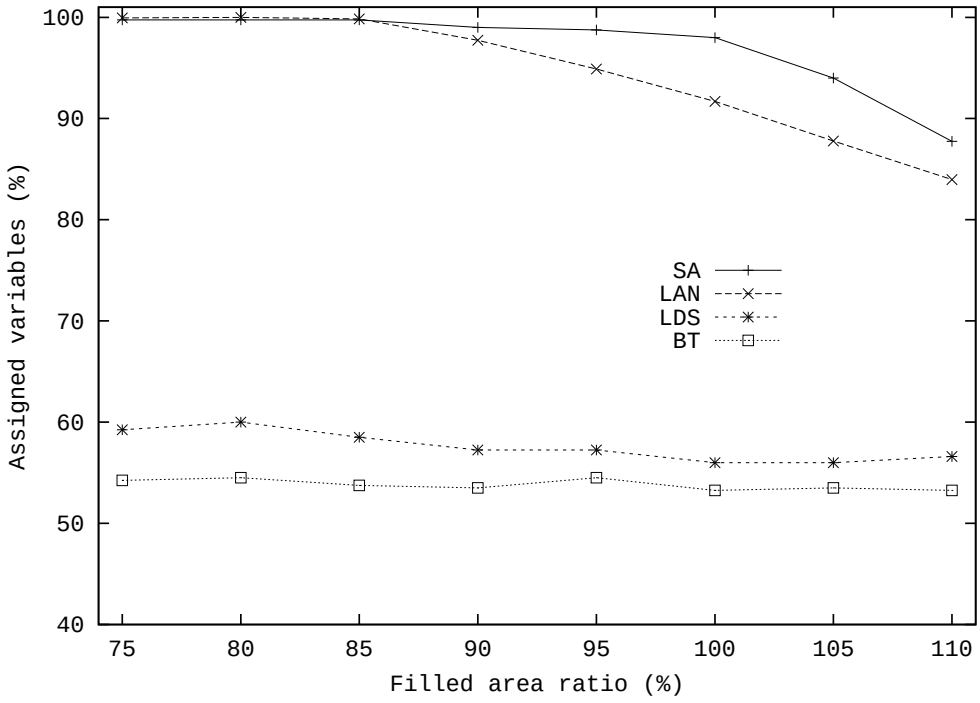

Figure 6.8: Comparison of SA, LAN search, LDS and BT.

6.3 Discussion

This section summarizes achieved results and discusses their impact on the settings of the algorithm and the area of its applicability.

As expected, the amount of assigned variables increased with the iteration number. It was interesting to find out that quite small limits could give the best results. The greater limits made slightly better improvements at the beginning but were overcome by the smaller limits in the end. However, there is a lower bound on limit below which the behavior is again not so good.

The most significant progress was made in first iterations. After a certain number of iterations, no further improvement was achieved. It was shown that this behavior is very similar for different filled areas ratio and even for over-constrained problems. However, for the most hard problems the algorithm is not very effective. Too many variables were left unassigned.

The attempt to develop the heuristics on the basis of all previous iterations did not bring any improvement in general. However, the application of the extended variable ordering only to the repair variables may achieve slightly better results for a particular problem. It was successfully implemented in the real-life timetabling problem (see Section 7.3).

As an incomplete version of backtracking, LAN search seems to be more useful for hard or over-constrained problems where algorithms aimed at generation of a complete solution do not help and where a sufficiently good partial solution can be an interesting output. Its importance lies in solving tightly constrained problems where constraint propagation is not strong enough. For such problems, standard tree search methods may become lost in the search space because of an early made wrong decision. LAN search returns at least a partial solution and tries to improve it in subsequent iterations. However, when the problem is too loosely constrained (few local inconsistencies) or the constraint propagation is strong enough, e.g., n-queens problem, LAN search makes no effect. Here, any wrong decision is quickly recognized and cause fail. If the problem is

solvable, it can be solved by backtracking with constraint propagation and LAN search application does not bring any improvement. If it is over-constrained, most of the variables are left unassigned.

For proposed random problems, LAN search outperformed significantly limited discrepancy search and backtracking. Both would need much better ordering heuristics. LAN search started with the same initial heuristics and developed its own better heuristics that finally lead to much better solutions. Local search behaves better for more constrained problems while LAN search outperforms it a bit in the number of completely solved problems. It is in correspondence with the usual behavior of local search and tree search algorithms. Local search algorithms are generally better in optimization problems, while backtracking-based algorithms are better for satisfaction problems, where local search is more likely to get entrapped in a local optimum (see Section 4.1.3).

Chapter 7

Real-life Application of LAN search

The LAN search algorithm was originally developed for a real-life large scale application from the area of course timetabling. The requirements of this problem were so hard that backtracking with constraint propagation was not able to find a solution even after 10 hours. It was even not guaranteed that the constraints are not contradictory. Therefore an algorithm that would find at least a partial timetable was needed. This would help the operator to place some classes manually and/or identify the constraints that should be relaxed.

Here the problem is briefly specified and the original implementation of the solver is described. Afterwards, some modifications that could improve the behavior of the solver are proposed. Finally, the modifications are experimentally evaluated.

7.1 Purdue University Timetabling Problem (PUTP)

At Purdue University (USA), the timetabling process consists of timetabling approximately 750 classes attended by almost 29,000 students into 41 large lecture rooms with capacities up to 474 students. The classrooms are variously equipped and some classes need to be timetabled only to a classroom with the specific equipment. The classes are taught several times a week, resulting in around 1,600 meetings (objects in RPP) to be timetabled. The space covered by all meetings fills approximately 85 % of the total available space (corresponds to filled area ratio of RPP). Special meeting patterns defined for classes direct possible time and location placement, e.g., all meetings of the same class must be taught in the same classroom and hour, valid combinations of days are given, etc. The timetable must respect the requirements and preferences of faculties and instructors on both times and locations.

The initial requirements for the timetable may be so hard that it may be difficult to find a timetable that would satisfy all of them. They may be even contradictory because each faculty and instructor has many demands. Therefore the major objective in developing an automated system is to maximize the number of timetabled classes. If some classes are left untimetabled, the operator will look at the timetable and try to place them manually. If this is not possible, he or she will utilize the partial timetable to identify the requirements that has to be relaxed or stated as preferences. And the search process will be repeated. The second major objective is to minimize the number of potential conflicts between classes of each student. In the third place, the timetable should satisfy as many preferences of the instructors and faculties as possible. A full description of the problem can be found in [32].

7.2 Original Implementation

The PUTP was originally implemented as follows. Each meeting of a class is assigned two domain variables — a *time variable* (time in a week) and a *classroom variable*. The hard constraints secure that two classes will not be taught in the same classroom at once. They also removes values banned by meeting patterns and by the requirements of instructors and faculties. But in addition to this, it is necessary to evaluate also the second and third objective. These are expressed with the help of soft constraints. The cost of a soft constraint ensuring that two classes with common students will not overlap is equal to the number of common students. The preferences of instructors and faculties are expressed applying soft unary constraints; the preference of a certain value is expressed as a cost of the constraint demanding that the value will be assigned. The cost of such preferences ranges from 0 to 29. Therefore the number of common students is more important than the most strong preferences, if there are more than 30 common students. More details about applied hard and soft constraints are given in [32].

Clearly the PUTP can be expressed as a VCSP with a rejection value, i.e., evaluating also partial solutions. A partial solution with more timetabled classes is more preferred. Ties are broken by the cost of violated soft constraints.

The problem solver was implemented in SICStus Prolog [17]. The standard CLP(FD) library [6] had to be extended. The extension was named *soft CLP(FD)* [30]. It currently defines necessary soft constraints and supports their propagation applying partial forward checking (see Section 4.2.1).

Different initial variable and value ordering heuristics were tried. But always a variation of first-fail was applied for variable ordering and the most preferred values were chosen first to satisfy as many soft constraints as possible.

The search process consists of two parts. Time variables are labeled first, followed by the classroom variables. This ordering corresponds to the relative importance of the preferences over these variables. Attempts to label all variables together consumed to much memory. There are two cost functions. One evaluates the time variables, and the other id defined over the room variables. Currently a new version of the solver is being developed which tries to label all variables at once. It is based on a new version of SICStus Prolog.

The main structure of the solver is shown in Figure 7.1. All soft constraints must be stated before any hard constraint is activated. Hard constraints may cause a violation of some soft constraints which could not be captured in a cost function otherwise. Within the predicates add_repair_preferences/1 and remove_repair_preferences/1, the developed value ordering heuristic is implemented. They will be described below. One iteration of LAN search is applied to find a partial assignment of time variables. A branch and bound algorithm is then used to find an optimal solution over classroom variables (classes without instantiated time variable are skipped). The branch and bound algorithm can be applied due to the hard constraints that ensure that when the labeling of time variables is finished, there are enough rooms of sufficient capacities for the classes in each time slot. If no solution for classroom variables is found within the time limit, the predicate time_out/2 fails and all room variables are left unassigned. The information about unassigned classroom variables reflects back to the corresponding time variables. It is difficult to find a suitable classrooms for these classes, or such a classrooms are not available. To reflect this fact, the value assigned to a time variable with a corresponding unassigned classroom variable is discouraged by developed value ordering heuristic in the subsequent search. Once a partial assignment is generated by this process, the search for a complete solution may continue by repeating these steps. The user may also provide input to influence the behavior of the search (see Section 5.5). The time limit of the branch and bound was set to 15 seconds, and the limit of LAN search to the value of 50.

```
solver_iteration( MaxTime, Limit, TimeVariables, RoomVariables, TimeCost, RoomCost ) :–
        declare_variables( TimeVariables, TimeCost, RoomVariables, RoomCost ),
        post_soft_constraints( TimeVariables, RoomVariables ),
        post_hard_constraints( TimeVariables, RoomVariables ),
        add_repair_preferences( TimeVariables ),
        labeling( TimeVariables, TimeCost, RoomVariables, RoomCost, Limit, MaxTime ),
        delete_repair_preferences( TimeVariables ).

labeling( TimeVariables, TimeCost, RoomVariables, RoomCost, Limit, MaxTime ) –>
        lan_search( TimeVariables, Limit ),
        (
            time_out( MaxTime, branch_and_bound( sup, RoomVariables, RoomCost ) )
        ;
            true
        ), !.
```

Figure 7.1: The structure of the PUTP solver.

The value ordering heuristic developed by LAN search for time variables is slightly modified to reflect more the student conflicts. It is implemented with the help of soft unary constraints. The predicate add_repair_preferences/1 posts for each preferred or discouraged value a soft unary constraint that requires this value. Its cost is 30 or -30, resp. Therefore it is is more weighty than the preferences of faculties and instructors. However, the soft constraints concerning student conflicts have greater weight if they refer to more than 30 students. The preferences added by add_repair_preferences must be removed after the labeling process, otherwise they will be included in the final cost. This is done by the predicate remove_repair_preferences/1.

7.3 Proposed Modifications

A few extensions and modifications were proposed to possibly improve the behavior of the solver. Most of them are based on the experiments on random placement problems.

```
labeling( TimeVariables, TimeCost, RoomVariables, RoomCost, Limit, MaxTime ) –>
        lan_search( TimeVariables, Limit ),
        (
            time_out( MaxTime, branch_and_bound( sup, RoomVariables, RoomCost ) )
        ;
            lan_search( RoomVariables, Limit )
        ), !.
```

Figure 7.2: The improved predicate labeling.

First modification refers to the situation when the branch and bound algorithm fails. Instead of leaving all classroom variables unassigned, one iteration of LAN search can be applied. Therefore at least some classroom variables are instantiated. The modified predicate labeling/6 is shown in Figure 7.2. The information about unassigned classroom variables still reflects back to time

variables. It is difficult to find a suitable classrooms for the classes in the given time, or such a classroom is not available. To reflect this fact, the value assigned to a time variable with a corresponding unassigned classroom variable is discouraged in the subsequent search.

Another modification implements the extended version of the developed variable ordering heuristics (see Section 5.6). Experiments on random problems showed that the variant when the counter is utilized only for the repair variables may bring some improvement for a particular problem (see Section 6.3).

Next extension develops a value ordering heuristic also for classroom variables. Originally no heuristics were developed for them. When a class has been successfully timetabled, the value of its time variable is preferred by LAN search, but the value of its room variable has still the same preference. The value ordering heuristic for classroom variables is constructed in the same manner as for LAN search, i.e., successfully assigned values are preferred, unsuccessfully tried values are discouraged. Clearly, this heuristics can be applied only when the class has been assigned the same time as in the last iteration.

The last proposal took into consideration the fact that various limits gave slightly different results in experiments with LAN search (see Section 6.3). Besides the original value of 50, also the values 5, 20, and 100 were tried.

7.4 Achieved Results

All experiments were accomplished on a PC with a AMD Athlon/850 MHz processor and 128 MB of memory. The solver was applied to the data set from fall semester 2001. It can be downloaded in the form of Prolog facts from `http://www.fi.muni.cz/~hanka/purdue_data`. In the future, also data from other semesters will be added.

One iteration took around 2–3 minutes for time labeling. One step of branch and bound search for classroom variables took 1–2 seconds. The time labeling took longer due to the preference propagation and more complex constraints posted on time variables.

The original version of the solver [31] was able to timetable 744 out of 747 classes. This solution was found in 5 iteration steps. Subsequent iterations did not find a better solution. After the implementation of the first two proposals, the solver was able to timetable all classes but one in 8 iterations. The last class was timetabled with user help (see Section 5.5). This version is described in [32]. The development of value ordering heuristics also for classroom variables brought even further improvement. All the classes were successfully timetabled without a user assistance within 10 iterations.

Numbers of untimetabled classes in each iteration for the three above described versions are shown in the Figure 7.3. The increase in the number of unassigned classes during the second iteration of the improved versions occurs as a results of the branch and bound search over classroom variables being replaced by LAN search. As can be seen in Figure 7.4, other limits did not bring any improvement. The best results were achieved with the original limit of 50.

Run	1	2	3	4	5	6	7	8	9	10
Student conflicts (%)	1.54	1.63	2.12	2.17	2.04	2.09	2.11	2.03	2.07	2.10
Preferred times (%)	83.78	81.50	79.83	82.08	81.99	81.46	79.40	80.06	79.78	79.78
Discouraged times (%)	4.15	4.40	4.24	4.53	4.68	4.67	4.47	4.61	4.48	4.47
Preferred classrooms (%)	74.17	91.16	97.56	49.66	84.77	45.64	45.03	45.03	45.03	45.03

Table 7.1: The results of particular iterations.

In Table 7.1, the amounts of student conflicts and satisfied preferences of faculties and instructors for particular iterations of the current implementation are given. It shows that the final solution

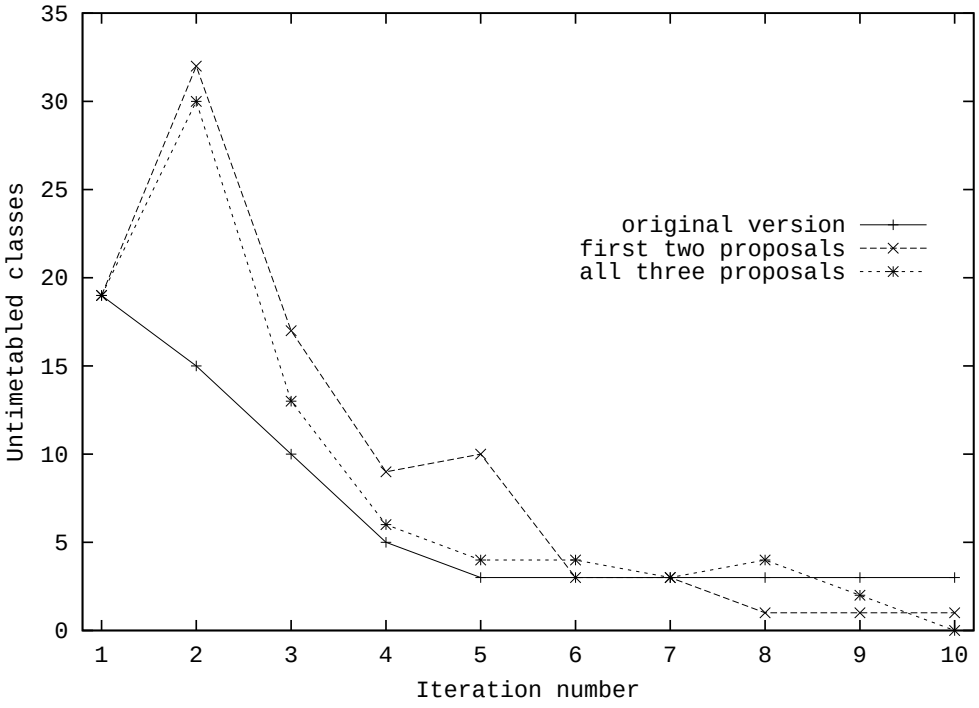

Figure 7.3: Comparison of the original and improved versions.

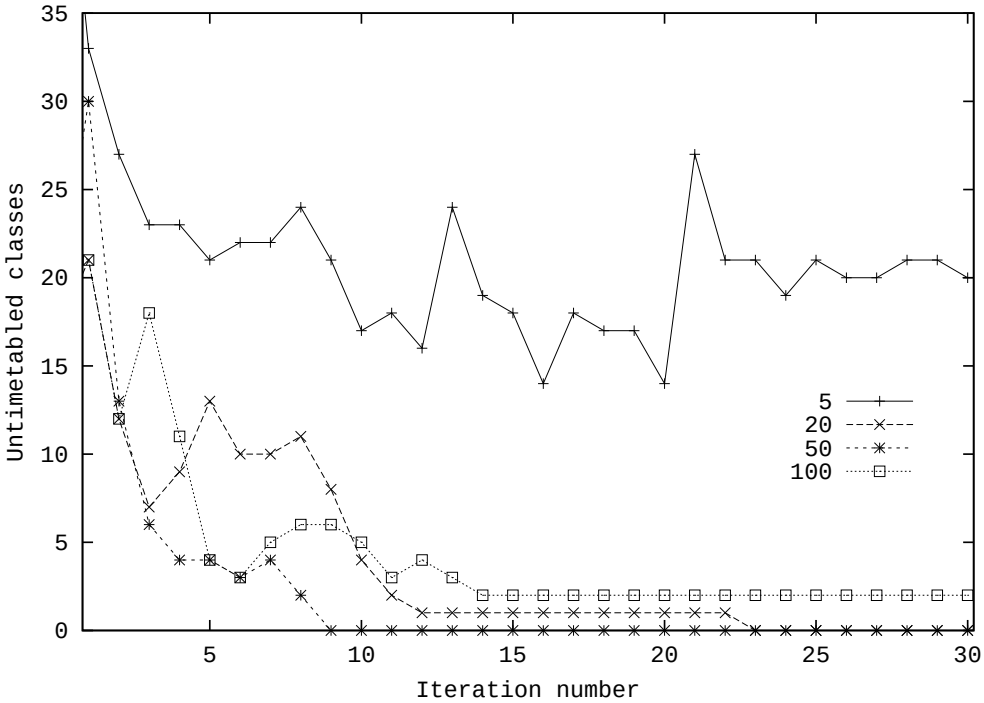

Figure 7.4: Comparison of different limits.

was able to satisfy 97.9 % of the student requirements from course pre-enrollment (in other words, the classes of 2.1 % of students are conflicting). 79.78 % of classes were assigned at the preferred times while 4.47 % classes must be taught at discouraged times. A classroom preferences were satisfied up to 45.03 %. The results of the previous versions can be found in the above cited [31] and [32]. They are very similar. The repair runs of LAN search did not significantly influence the amount of satisfied preferences, but were able to find a complete solution.

Chapter 8

Conclusions

A classification of various algorithms to solve constraint satisfaction problems is proposed, including a formal description of some algorithms of each approach. This overview is given separately for satisfaction and optimization problems. A special section is devoted to the algorithms which try to find a maximal partial solution. The search algorithm that is the main part of the thesis has been developed for such problems. The classification will be utilized as a studying material for a new course *Constraint Programming* at the Faculty of Informatics. Unfortunately, there is a lack of good textbooks in the area of Constraint Programming and a uniform description of the algorithms can be helpful in this context.

The structure of this part results from long discussions. Various papers about particular algorithms were studied together with available textbooks [36, 26, 14, 28]. All presented algorithms are given in a uniform, non-recursive pseudo-code. It is based on the algorithm of chronological backtracking with constraint propagation given in [7]. A possible implementation in a CLP(FD) environment is always discussed. Algorithms based on depth-first search can be simply implemented in CLP. For these algorithms also a CLP code is given. On the other hand, the implementation of some algorithms is more complicated.

The algorithm outlined in [31] is given in a revised and formalized form. It was named the limited assignment number search algorithm. The implementation in both an imperative and a CLP paradigm is presented. A potential problem with a so called conflicting variable was identified. Possible methods how to cope with the conflicting variable are discussed and were successfully employed in experiments with the real-life problem. Some extensions, concerning developed value and variable ordering heuristics, were proposed.

A number of experiments have been done to evaluate the algorithm and its extensions. LAN search was also compared with some other algorithms. A suitable random problems for these experiments were proposed. The most important experiments are presented and discussed. Most of the extensions showed to be ineffective. One of them, however, made a useful improvement in the real-life application (see Section 7.3).

The real university timetabling problem for which LAN search was originally proposed is introduced. The original implementation was extended to reflect the achieved knowledge about the algorithm. Among others, the value ordering heuristic is now developed for all variables, not only for the time variables. While the original version left at least three classes unscheduled, the extended version was able to create a complete timetable.

A first attempt to formalize the algorithm together with some computational results has been presented at *Student Research Forum* of *SOFSEM 2002* [38]. A paper about the final version has been submitted to the *Ninth International Conference on Principles and Practice of Constraint Programming (CP'2003)*.

Possible extensions of this work include a verification on other types of problems and comparison with another algorithms. This would help to specify more explicitly the areas of its applicability. Mainly, the intelligent backtracking algorithms, could by considered. The algorithm might be extended towards problems where all variables remain unassigned. A cooperation with soft constraints propagation as applied for the real life problem could be improved.

Another part of future work will include a possible application to a minimal perturbation problem. Here the original problem is slightly modified or extended and a solution of a new problem should be found, minimizing the differences from the original solution.

Bibliography

[1] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 311–318, Menlo Park, 26–30 1998. AAAI Press. 2.1, 3.1

[2] Roman Barták. On-line Guide to Constraint Programming. Available from `http://kti.mff.cuni.cz/~bartak/constraints`, 1998. 1

[3] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of Workshop of Doctoral Students'99*, 1999. 3.1

[4] Stephano Bistarelli, Rosella Gennari, and Francesca Rossi. Constraint propagation for soft constraint satisfaction problems: Generalization and termination conditions. In *Principles and Practice of Constraint Programming — CP'00*, pages 83–97. Springer-Verlag LNCS 1894, 2000. 4.2

[5] Ivan Bratko. *Prolog Programming for AI*. Addison-Wesley, third edition, 2001. 2.3

[6] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Programming Languages: Implementations, Logics, and Programming*. Springer-Verlag LNCS 1292, 1997. 6.1.2, 6.2.1, 6.2.2, 7.2

[7] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002. 3.3.1, 8

[8] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992. 4.2

[9] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Principles and Practice of Constraint Programming*, pages 179–193, 1996. 3.2.2

[10] Y. Georget and P. Codognet. Compiling Semiring-based constraints with `clp(FD,`*S*`)`. In Michael Maher and Jean-François Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 205–219. Springer-Verlag LNCS 1520, 1998. 4.2

[11] M. Haralick and G.L. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. In *Artificial Intelligence 14*, pages 263–313, 1980. 3.3.1, 3.3.2

[12] William D. Harvey. *Nonsystematic Backtracking Search*. Phd thesis, CIRL, University of Oregon, 1269 University of Oregon; Eugene, OR USA 97403-1269, 1995. 3.2.3

[13] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–615. Morgan Kaufmann, 1995. 3.2.3

[14] John Hooker. *Logic-based methods for Optimization: combining optimization and constraint satisfaction*. Wiley-Interscience, 2000. 1, 4.1.1, 8

[15] IC-Parc. *ECL^{i}PS^e User Manual*, 2002. `http://www.icparc.ic.ac.uk/eclipse`. 2.3, 3.3.1, 3.4, 4.1.2, 4.2.2

[16] ILOG S.A. *ILOG Solver 5.1 User's Manual*, 2001. `http://www.ilog.com`. 2.3, 1

[17] Intelligent Systems Laboratory, Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 2002. 2.3, 3.2.3, 3.3.1, 4.1.1, 4.2.2, 6.1.3, 6.2.1, 7.2, 1

[18] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19:503–581, 1994. 2.3

[19] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, 2002. 4.1.3

[20] Olli Kamarainen and Hani El Sakkout. Local probing applied to scheduling. In Pascal Van Hentenryck, editor, *Proceedings of Principles and Practice of Constraint Programming, 8th International Conference, CP 2002, Itahca, NY, USA*, pages 155–171. Springer-Verlag LNCS 2470, 2002. 3.4, 4.1.2, 4.1.3

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983. 4.1.2

[22] Richard E. Korf. Improved limited discrepancy search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96) and the Eighth Conference on Innovative Applications of Artificial Intelligence (IAAI-96)*, pages 286–291. Menlo Park, Calif.: AAAI Press. (http://www.aaai.org), 1996. 3.2.3

[23] Vipin Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992. 3.1

[24] Javier Larossa, Pedro Meseguer, and Thomas Schiex. Maintaining reversible DAC for MAX-CSP. *Artificial Intelligence*, 107(1):149–163, 1999. 4.2

[25] M. Lemaître and G. Verfaillie. Daily management of an earth observation satellite : comparison of ILOG solver with dedicated algorithms for Valued constraint satisfaction problems. In *Proceedings of Third ILOG International Users Meeting*, Paris, France, July 1997. 2.2.2, 4.2.2

[26] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998. 1, 8

[27] Pedro Meseguer, Javier Larrosa, and Martì Sánchez. Lower bounds for non-binary constraint optimization problems. In Toby Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2001. 4.2

[28] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer-Verlag, 2000. 1, 4.1.2, 8

[29] Hana Rudová. *Constraint Satisfaction with Preferences*. PhD thesis, Faculty of Informatics, Masaryk University, 2001. See `http://www.fi.muni.cz/~hanka/phd.html`. 2.2.2, 2.2.2

[30] Hana Rudová. Soft CLP(*FD*). In Susan Haller and Ingrid Russell, editors, *Proceedings of the 16th International Florida Artificial Intelligence Symposium, FLAIRS-03*. AAAI Press, 2003. Accepted for publication. 7.2

[31] Hana Rudová and Keith Murray. University course timetabling with soft constraints. In Edmund Burke and Patrick De Causmaecker, editors, *PATAT 2002 — Proceedings of the 4th international conference on the Practice And Theory of Automated Timetabling*, pages 73–89, 2002. 1, 7.4, 7.4, 8

[32] Hana Rudová and Keith Murray. University course timetabling with soft constraints. In Edmund Burke and Patrick De Causmaecker, editors, *Practice And Theory of Automated Timetabling, Selected Papers*. Springer-Verlag LNCS, 2003. Accepted for publication. 7.1, 7.2, 7.4, 7.4

[33] W. Ruml. Incomplete tree search using adaptive probing. In *Proceedings of Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001. 4.1.3

[34] Zsófia Ruttkay. Constraint satisfaction — a survey. *CWI Quaterly*, 11(2&3):123–161, 1998. 2.1, 3.1

[35] Peter van Beek and Rina Dechter. Constraint tightness and looseness versus local and global consistency. *Journal of the ACM*, 44(4):549–566, 1997. 2.1

[36] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989. 1, 2.1, 3.1, 3.2.2, 8

[37] G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search for solving constraint optimization problems. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96) and Eighth Conference on Innovative Applications of Artificial Intelligence (IAAI-96)*, pages 181–187, Portland, OR, USA, 1996. 4.2, 4.2.2

[38] K. Veřmiřovský and H. Rudová. Limited assignment number search algorithm. In *Student Research Forum of the 29th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'2002)*, 2002. This paper is also available from `http://www.fi.muni.cz/~hanka/publications`. 8

[39] Stefan Voß. Meta-heuristics: State of the art. In Alexander Nareyek, editor, *Local search for planning and scheduling: revisited papers*, pages 1–23. Springer-Verlag LNCS 2148, 2001. 4.1.2

[40] Toby Walsh. Depth-bounded discrepancy search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97-Vol2)*, pages 1388–1395. Morgan Kaufmann, 1997. 3.2.3

Appendix A

Contents of the CD

This thesis is enclosed with a CD[1]. It is structured as web pages, starting with *index.html* in the root directory. It contains this document in *portable data format (pdf)* and *postscript (ps)*. Next the LaTeX source files of the thesis are embodied, including the *makefile*. The rest of the CD consists of the material related to the experiments on random placement problems.

The specification of RPP is given as described in Section 6.1.1. The application employed for the generation of problem instances is present. It is written in SICStus Prolog [17]. It consists of the main file *generator.pl*, the self explanatory file *parameters.pl*, and the license file.

Next the CD contains all data instances utilized for the experiments and the best found solutions. That is, all eight standard sets and one set of the problems with double sized domain are present (see Section 6.1.3). Each instance is defined in one file *genX.pl* with Prolog syntax. Its complete or partial solution is given in the corresponding file *genX.solution* or *genX.partial*, resp. The syntax of these files is described in detail on the CD.

Finally three solvers that can be applied to RPPs are included. All solvers are given in one tar gzipped file. This file consists of the solver files, usage information in the file *README* and the license in *LICENSE* (all can be freely distributed or modified). The *LAN/LDS/BT RPP Solver* is written in SICStus Prolog [17]. It implements LAN search, limited discrepancy search and chronological backtracking with constraint propagation. The *SA RPP Solver* is written in ANSI C++ and it implements the simulated annealing algorithm. The last solver was not utilized for the experiments. It is present for possible future use. It is named the *LAN/LDS/BT RPP ILOG Solver*. It is written in C++ with the help of ILOG Solver library [16] and it implements the same algorithms as the above mentioned LAN/LDS/BT RPP Solver.

[1] The CD is mirrored at `http://www.fi.muni.cz/~hanka/students/xvermir`.