# Constraint Programming: In Pursuit of the Holy Grail

*Roman Barták*

*Charles University, Faculty of Mathematics and Physics, Department of Theoretical Computer Science*
*Malostranské námûstí 2/25, 118 00 Praha 1, Czech Republic*
*e-mail: bartak@kti.mff.cuni.cz*

**Abstract:** Constraint programming (CP) is an emergent software technology for declarative description and effective solving of large, particularly combinatorial, problems especially in areas of planning and scheduling. It represents the most exciting developments in programming languages of the last decade and, not surprisingly, it has recently been identified by the ACM (Association for Computing Machinery) as one of the strategic directions in computer research. Not only it is based on a strong theoretical foundation but it is attracting widespread commercial interest as well, in particular, in areas of modelling heterogeneous optimisation and satisfaction problems.

In the paper, we give a survey of constraint programming technology and its applications starting from the history context and interdisciplinary nature of CP. The central part of the paper is dedicated to the description of main constraint satisfaction techniques and industrial applications. We conclude with the overview of limitations of current CP tools and with outlook of future directions.

**Keywords:** constraint satisfaction, search, inference, combinatorial optimisation, planning, scheduling

## 1 Introduction

In last few years, Constraint Programming (CP) has attracted high attention among experts from many areas because of its potential for solving hard real-life problems. Not only it is based on a strong theoretical foundation but it is attracting widespread commercial interest as well, in particular, in areas of modelling heterogeneous optimisation and satisfaction problems. Not surprisingly, it has recently been identified by the ACM (Association for Computing Machinery) as one of the strategic directions in computer research. However, at the same time, CP is still one of the least known and understood technologies.

Constraints arise in most areas of human endeavour. They formalise the dependencies in physical worlds and their mathematical abstractions naturally and transparently. A *constraint* is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. The constraint thus restricts the possible values that variables can take, it represents partial information about the variables of interest. Constraints can also be heterogeneous, so they can bind unknowns from different domains, for example the length (number) with the word (string). The important feature of constraints is their declarative manner, i.e., they specify what relationship must hold without specifying a computational procedure to enforce that relationship.

We all use constraints to guide reasoning as a key part of everyday common sense. “I can be there from five to six o’clock”, this is a typical constraint we use to plan our time. Naturally, we do not solve one constraint only but a collection of constraints that are rarely independent. This complicates the problem a bit, so, usually, we have to give and take.

*Constraint programming* is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (requirements) about the problem area and, consequently, finding solution satisfying all the constraints.

“Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.” [E. Freuder]

## 2 The interdisciplinary origins

The earliest ideas leading to CP may be found in the **Artificial Intelligence** (AI) dating back to sixties and seventies.

The *scene labelling* problem [41] is probably the first constraint satisfaction problem that was formalised. The goal is to recognise the objects in a 3D scene by interpreting lines in the 2D drawings, First, the lines or edges must be labelled, i.e., they are categorised into few types, namely convex (+), concave (-) and occluding edges (<). In more advanced systems, the shadow border is recognised as well.

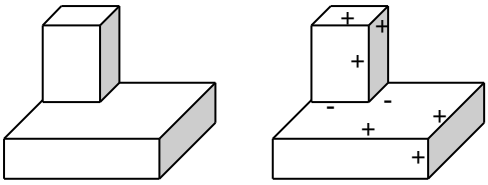


Figure 1
Scene labelling

There are a lot of ways how to label the scene (exactly $3^n$, where n is a number of edges) but only few of them has any 3D meaning. The idea how to solve this combinatorial problem is to find legal labels for junctions satisfying the constraint that the edge has the same label at both ends. This reduces the problem a lot because there is only a limited number of legal labels for junctions.

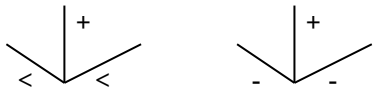


Figure 2
Available labelling for junction

The main algorithms developed in those years (like Waltz labelling algorithm [41]) were related to achieving some form of consistency.

Another application for constraints is *interactive graphics* where Ivan Sutherland's Sketchpad [36], developed in early 1960s, was the pioneering system. Sketchpad and its follower, ThingLab [5] by Alan Borning, were interactive graphics applications that allowed the user to draw and manipulate constrained geometric figures on the computer's display. These systems contribute to developing local propagation methods and constraint compiling.

The main step towards CP was achieved when Gallaire [16], Jaffar & Lassez [20] noted that logic programming was just a particular kind of constraint programming. The basic idea behind **Logic Programming** (LP), and declarative programming in general, is that the user states *what* has to be solved instead of *how* to solve it, which is very close to the idea of constraints. Therefore the combination of constraints and logic programming is very natural and *Constraint Logic Programming* (CLP) makes a nice declarative environment for solving problems by means of constraints. However, it does not mean that constraint programming is restricted to CLP. Constraints were integrated to typical imperative languages like C++ and Java as well.

The nowadays real-life applications of CP in the area of planning, scheduling and optimisation rise the question if the traditional field of **Operations Research** (OR) is a competitor or an associate of CP. There is a significant overlap of CP and OR in the field of NP-Hard combinatorial problems. While the OR has a long research tradition and (very successful) method of solving problems using linear programming, the CP emphasis is on higher level modelling and solutions methods that are easier to understand by the final customer. The recent advance promises that both methodologies can exploit each other, in particular, the CP can serve as a roof platform for integrating various constraint solving algorithms including those developed and checked to be successful in OR.

As the above paragraphs show, the CP has an inner interdisciplinary nature. It combines and exploits ideas from a number of fields including Artificial Intelligence, Combinatorial Algorithms, Computational Logic, Discrete Mathematics, Neural Networks, Operations Research, Programming Languages and Symbolic Computation.

## 3 Solving Technology

Currently, we see two branches of constraint programming, namely constraint satisfaction and constraint solving. Both share the same terminology but the origins and solving technologies are different.

*Constraint satisfaction* deals with problems defined over finite domains and, currently, probably more than 95% of all industrial constraint applications use finite domains. Therefore, we deal with constraint satisfaction problems mostly in the paper.

*Constraint solving* shares the basis of CP, i.e., describing the problem as a set of constraints and solving these constraints. But now, the constraints are defined (mostly) over infinite or more complex domains. Instead of combinatorial methods for constraint satisfaction, the constraint solving algorithms are based on mathematical techniques such as automatic differentiation, Taylor series or Newton method. From this point of view, we can say that many famous mathematicians deal with whether certain constraints are satisfiable (e.g. recently proved Fermat's Last Theorem).

### 3.1 Constraint Satisfaction

Constraint Satisfaction Problems [37] have been a subject of research in Artificial Intelligence for many years. A *Constraint Satisfaction Problem* (CSP) is defined as:

- a set of *variables* $X=\{x_1,...,x_n\}$,
- for each variable $x_i$, a finite set $D_i$ of possible values (its *domain*), and
- a set of *constraints* restricting the values that the variables can simultaneously take.

Note that values need not be a set of consecutive integers (although often they are), they need not even be numeric.

A *solution to a CSP* is an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied at once. We may want to find:

- just one solution, with no preference as to which one,
- all solutions,
- an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables.

Solutions to a CSP can be found by searching (systematically) through the possible assignments of values to variable. Search methods divide into two broad classes, those that traverse the space of partial solutions (or partial value assignments), and those that explore the space of complete value assignments (to all variables) stochastically.

### 3.1.1 Systematic Search

From the theoretical point of view, solving CSP is trivial using systematic exploration of the solution space. Not from the practical point of view where the efficiency takes place. Even if systematic search methods (without additional improvements) look very simple and non-efficient they are important because they make the foundation of more advanced and efficient algorithms.

The basic constraint satisfaction algorithm, that searches the space of complete labellings, is called *generate-and-test* (GT). The idea of GT is simple: first, a complete labelling of variables is generated (randomly) and, consequently, if this labelling satisfies all the constraints then the solution is found, otherwise, another labelling is generated.

The GT algorithm is a weak generic algorithm that is used if everything else failed. Its efficiency is poor because of non-informed generator and late discovery of inconsistencies. Consequently, there are two ways how to improve efficiency of GT:

- The generator of valuations is smart (informed), i.e., it generates the complete valuation in such a way that the conflict found by the test phase is minimised. This is a basic idea of stochastic algorithms based on local search that are discussed later.
- Generator is merged with the tester, i.e., the validity of the constraint is tested as soon as its respective variables are instantiated. This method is used by the backtracking approach.

*Backtracking* (BT) [31] is a method of solving CSP by incrementally extending a partial solution that specifies consistent values for some of the variables, towards a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution

As mentioned above, we can see BT as a merge of the generating and testing phases of GT algorithm. The variables are labelled sequentially and as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial solution violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. Consequently, backtracking is strictly better than generate-and-test, however, its running complexity for most nontrivial problems is still exponential.

There are three major drawbacks of the standard (chronological) backtracking:

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

Methods for solving the first two drawbacks were proposed, namely backjumping and backmarking (see below), but more attention was paid to detect the inconsistency of partial solution sooner using consistency techniques.

### 3.1.2 Consistency Techniques

Another approach to solving CSP is based on removing inconsistent values from variables' domains till the solution is got. These methods are called consistency techniques and they were introduced first in the scene labelling problem [41]. Notice that consistency techniques are deterministic, as opposed to the non-deterministic search.

There exist several consistency techniques [23,25] but most of them are not complete. Therefore, the consistency techniques are rarely used alone to solve CSP completely.

The names of basic consistency techniques are derived from the graph notions. The CSP is usually represented as a *constraint graph* (network) where nodes correspond to variables and edges are labelled by constraints [29]. This requires the CSP to be in a special form that is usually referred as a *binary CSP* (contains unary and binary constraints only). It is easy to show that arbitrary CSP can be

transformed to equivalent binary CSP [2], however, in practice the binarization is not likely to be worth doing and the algorithms can be extended to tackle non binary CSP as well.

The simplest consistency technique is referred to as a *node consistency* (NC). It removes values from variables' domains that are inconsistent with unary constraints on respective variable.

The most widely used consistency technique is called *arc consistency* (AC). This technique removes values from variables' domains that are inconsistent with binary constraints. In particular, the arc $(V_i,V_j)$ is arc consistent if and only for every value $x$ in the current domain of $V_i$ which satisfies the constraints on $V_i$ there is some value $y$ in the domain of $V_j$ such that $V_i=x$ and $V_j=y$ is permitted by the binary constraint between $V_i$ and $V_j$.

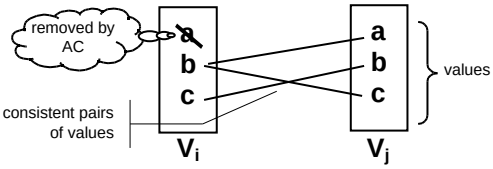


Figure 3
Arc-consistency removes local inconsistencies

There exist several arc consistency algorithms starting from AC-1 and concluding somewhere at AC-7. These algorithms are based on repeated revisions of arcs till a consistent state is reached or some domain becomes empty. The most popular among them are AC-3 and AC-4. The AC-3 algorithm performs re-revisions only for those arcs that are possibly affected by a previous revision. It does not require any special data structures opposite to AC-4 that works with individual pairs of values to remove potential inefficiency of checking pairs of values again and again. It needs a special data structure to remember pairs of (in)consistent values of incidental variables and, therefore, it is less memory efficient than AC-3.

Even more inconsistent values can be removed by *path consistency* (PC) techniques. Path consistency requires for every pair of values of two variables X, Y satisfying the respective binary constraint that there exists a value for each variable along some path between X and Y such that all binary constraints in the path are satisfied. It was shown by Montanary [29] that CSP is path consistent if and only if all paths of length 2 are path consistent. Therefore path consistency algorithms can work with triples of variables (paths of length 2). There exist several path consistency algorithms like PC-1 and PC-2 but they need an extensive representation ({0,1}-matrix) of constraints that is memory consuming. Less interesting ratio between complexity and simplification factor and the modifications to the connectivity of the constraint graph by adding some edges to the graph are other disadvantages of PC.

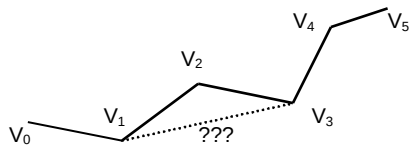


Figure 4
Path-consistency checks constraints along the path only.

All above mentioned consistency techniques are covered by a general notion of K-consistency [12] and strong K-consistency. A constraint graph is *K-consistent* if for every system of values for K-1 variables satisfying all the constraints among these variables, there exits a value for arbitrary K-th variable such that the constraints among all K variables are satisfied. A constraint graph is *strongly K-consistent* if it is J-consistent for all J≤K. Visibly:

- NC is equivalent to strong 1-consistency,
- AC is equivalent to strong 2-consistency,
- PC is equivalent to strong 3-consistency.

Algorithms exist for making a constraint graph strongly K-consistent for K>2 but in practice they are rarely used because of efficiency issues. Although these algorithms remove more inconsistent values than any arc-consistency algorithm they do not eliminate the need for search in general.

Clearly, if a constraint graph containing N nodes is strongly N-consistent, then a solution to the CSP can be found without any search. But the worst-case complexity of the algorithm for obtaining N-consistency in an N-node constraint graph is exponential. Unfortunately, if a graph is (strongly) K-consistent for K<N, then, in general, backtracking (search) cannot be avoided, i.e., there still exist inconsistent values.

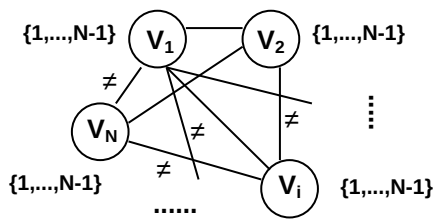


Figure 5
Strongly (N-1)-consistent graph still has no complete labelling.

Because most consistency techniques (NC, AC, PC) are not complete, i.e., there still remain some inconsistent values, the restricted forms of these algorithms takes attention as they remove similar amount of inconsistencies but they are more efficient. For example *directional arc consistency* (DAC) revises each arc only once (it works under given ordering of variables) and, thus, it requires less computation than AC-3 and less space than AC-4. Nevertheless, DAC is still able to achieve full arc consistency in some problems (e.g., tree constraint graphs).

It is also possible to weaken the path-consistency in a similar way. The resulting consistency technique is called *directional path consistency* (DPC) which is again computationally less expensive than achieving full path consistency.

Half way between AC and PC is Pierre Berlandier's *restricted path consistency* (RPC) [4] that extends AC-4 algorithm to some form of path consistency. The algorithm checks path-consistency along path X, Y, Z if and only if some value of the variable X has only one supporting value from the domain of incidental variable Y. Consequently, the RPC removes at least the same amount of inconsistent pairs of values as AC and also some pairs beyond.

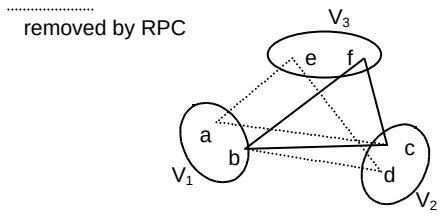


Figure 6
RPC removes more inconsistencies than AC

### 3.1.3 Constraint Propagation

Both systematic search and (some) consistency techniques can be used alone to solve the CSP completely but this is rarely done. A combination of both approaches is a more common way of solving CSP.

The **Look Back** schema uses consistency checks among already instantiated variables. BT is a simple example of this schema. To avoid some problems of BT, like thrashing and redundant work, other look back schemas were proposed.

*Backjumping* (BJ) [15] is a method to avoid thrashing in BT. The control of backjumping is exactly the same as backtracking, except when backtracking takes place. Both algorithms pick one variable at a time and look for a value for this variable making sure that the new assignment is compatible with values committed to so far. However, if BJ finds an inconsistency, it analyses the situation in order to identify the source of inconsistency. It uses the violated constraints as a guidance to find out the conflicting variable. If all the values in the domain are explored then the BJ algorithm backtracks to the most recent conflicting variable. This is a main difference from the BT algorithm that backtracks to the immediate past variable.

Another look back schemas, called *backchecking* (BC) and *backmarking* (BM) [18], avoid redundant work of BT. Both backchecking and its descendent backmarking are useful algorithms for reducing the number of compatibility checks. If the algorithm finds that some label Y/b is incompatible with any recent label X/a then it remembers this incompatibility. As long as X/a is still committed to, the Y/b will not be considered again.

*Backmarking* is an improvement over backchecking that avoids some redundant constraint checking as well as some redundant discoveries of inconsistencies. It reduces the number of compatibility checks by remembering for every label the incompatible recent labels. Furthermore, it avoids repeating compatibility checks which have already been performed and which have succeeded.

All look back schemas share the disadvantage of late detection of the conflict. In fact, they solve the inconsistency when it occurs but do not prevent the inconsistency to occur. Therefore **Look Ahead** schemas were proposed to prevent future conflicts [30].

*Forward checking* (FC) is the easiest example of look ahead strategy. It performs arc-consistency between pairs of not yet instantiated variable and instantiated variable, i.e., when a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. Therefore, FC maintains the invariance that for every unlabelled variable there exists at least one value in its domain that is compatible with the values of instantiated/labelled variables. FC does more work than BT when each assignment is added to the current partial solution, nevertheless, it is almost always a better choice than chronological backtracking.

Even more future inconsistencies are removed by the *Partial Look Ahead* (PLA) method. While FC performs only the checks of constraints between the current variable and the future variables, the partial look ahead extends this consistency checking even to variables that have not direct connection with labelled variables, using directional arc-consistency.

The approach that uses full arc-consistency after each labelling step is called *(Full) Look Ahead* (LA) or *Maintaining Arc Consistency* (MAC). It can use arbitrary AC algorithm to achieve arc-consistency, however, it should be noted that LA does even more work than FC and partial LA when each assignment is added to the current partial solution. Actually, in some cases LA may be more expensive than BT and, therefore FC and BT are still used in applications.

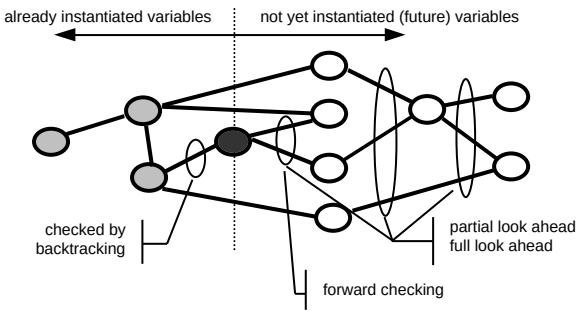


Figure 7
Comparison of propagation techniques

### 3.1.4 Stochastic and Heuristic Algorithms

Till now, we have presented the constraint satisfaction algorithms that extend a partial consistent labelling to a full labelling satisfying all the constraints. In the last few years, greedy local search strategies have become popular, again. These algorithms alter incrementally inconsistent value assignments to all the variables. They use a "repair" or "hill climbing" metaphor to move towards more and more complete solutions. To avoid getting stuck at "local minimum" they are equipped with various heuristics for randomising the search. Their stochastic nature generally voids the guarantee of "completeness" provided by the systematic search methods.

*Hill-climbing* is probably the most famous algorithm of local search [31]. It starts from a randomly generated labelling of variables and, at each step, it changes a value of some variable in such a way that the resulting labelling satisfies more constraints. If a strict local minimum is reached then the algorithm restarts at other randomly generated state. The algorithm stops as soon as a global minimum is found, i.e., all constraints are satisfied, or some resource is exhausted. Notice, that the hill-climbing algorithm has to explore a lot of neighbours of the current state before choosing the move.

To avoid exploring the whole state's neighbourhood the *min-conflicts* (MC) heuristic was proposed [27]. This heuristic chooses randomly any conflicting variable, i.e., the variable that is involved in any unsatisfied constraint, and then picks a value which minimises the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of violated constraints (the current value of the variable is picked only if all the other values increase the number of violated constraints).

Because the pure min-conflicts algorithm cannot go beyond a local-minimum, some noise strategies were introduced in MC. Among them, the *random-walk* (RW) strategy becomes one of the most popular [33]. For a given conflicting variable, the random-walk strategy picks randomly a value with probability *p*, and apply the MC heuristic with probability *1-p*. The random-walk heuristic can be applied to hill-climbing algorithm as well and we get the Steepest-Descent-Random-Walk (SDRW) algorithm.

*Tabu search* (TS) is another method to avoid cycling and getting trapped in local minimum [17]. It is based on the notion of *tabu list*, that is a special short term memory that maintains a selective history, composed of previously encountered configurations or more generally pertinent attributes of such configurations. A simple TS strategy consists in preventing configurations of tabu list from being recognised for the next *k* iterations (*k*, called tabu tenure, is the size of tabu list). Such a strategy prevents algorithm from being trapped in short term cycling and allows the search process to go beyond local optima.

Tabu restrictions may be overridden under certain conditions, called *aspiration criteria*. Aspiration criteria define rules that govern whether next configuration is considered as a possible move even it is tabu. One widely used aspiration criterion consists of removing a tabu classification from a move when the move leads to a solution better than that obtained so far.

Another method that searches the space of complete labellings till the solution is found is based on *connectionist approach* represented by GENET algorithm [42]. The CSP problem is represented here as a network where the nodes correspond to values of all variables. The nodes representing values for one variable are grouped into a cluster and it is assumed that exactly one node in the cluster is switched on that means that respective value is chosen for the variable. There is an inhibitory link (arc) between each two nodes from different clusters that represent incompatible pair of values according to the constraint between the respective variables.

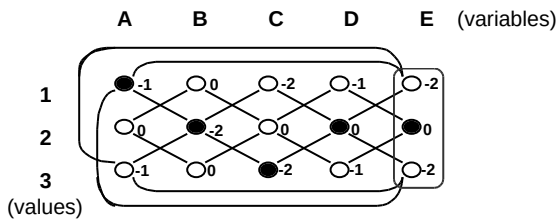


Figure 8
Connectionist representation of CSP

The algorithm starts with random configuration of the network and re-computes the state of nodes in cluster repeatedly taking into account only the state of neighbouring nodes and the weights of connections to these nodes. When the algorithm reaches a stable configuration of the network that is not a solution of the problem, it is able to recover

from this state by using simple learning rule that strengthens the weights of connections representing the violated constraints. As all above mentioned stochastic algorithms, the GENET is incomplete, for example it can oscillate.

### 3.2 Constraint Optimization

In many real-life applications, we do not want to find any solution but a good solution. The quality of solution is usually measured by an application dependent function called *objective function.* The goal is to find such solution that satisfies all the constraints and minimise or maximise the objective function respectively. Such problems are referred to as Constraint Satisfaction Optimisation Problems (CSOP).

A *Constraint Satisfaction Optimisation Problem (CSOP)* consists of a standard CSP and an optimisation function that maps every solution (complete labelling of variables) to a numerical value [37].

The most widely used algorithm for finding optimal solutions is called *Branch and Bound* (B&B) [24] and it can be applied to CSOP as well. The B&B needs a heuristic function that maps the partial labelling to a numerical value. It represents an under estimate (in case of minimisation) of the objective function for the best complete labelling obtained from the partial labelling. The algorithm searches for solutions in a depth first manner and behaves like chronological BT except that as soon as a value is assigned to the variable, the value of heuristic function for the labelling is computed. If this value exceeds the bound, then the sub-tree under the current partial labelling is pruned immediately. Initially, the bound is set to (plus) infinity and during the computation it records the value of best solution found so far.

The efficiency of B&B is determined by two factors: the quality of the heuristic function and whether a good bound is found early. Observations of real-life problems show also that the "last step" to optimum, i.e., improving a good solution even more, is usually the most computationally expensive part of the solving process. Fortunately, in many applications, users are satisfied with a solution that is close to optimum if this solution is found early. Branch and bound algorithm can be used to find sub-optimal solutions as well by using the second "acceptability" bound. If the algorithm finds a solution that is better than the acceptability bound then this solution can be returned to the user even if it is not proved to be optimal.

### 3.3 Over-Constrained Problems

When a large set of constraints is solved, it appears typically that it is not possible to satisfy all the constraints because of inconsistency. Such systems, where it is not possible to find valuation satisfying all the constraints, are called *over-constrained.*

Several approaches were proposed to handle over-constrained systems and among them the Partial Constraint Satisfaction and Constraint Hierarchies are the most popular.

*Partial Constraint Satisfaction* (PCSP) by Freuder & Wallace [13] involves finding values for a subset of the variables that satisfy a subset of the constraints. Viewed another way, some constraints are "weaken" to permit additional acceptable value combinations. By weakening a constraint we mean enlarging its domain. It is easy to show that enlarging constraint's domain covers also other ways of weakening a CSP like enlarging a domain of variable, removing a variable or removing a constraint.

Formally, PCSP is defined as a standard CSP with some evaluation function that maps every labelling of variables to a numerical value. The goal is to find labelling with the best value of the evaluation function.

The above definition looks similar to CSOP but, note, that now we do not require all the constraints to be satisfied. In fact, the global satisfaction of constraints is described by the evaluation function and, thus, the constraints are used as a guide to find an optimal value of the evaluation function. Consequently, in addition to handle over-constrained problems, PCSP can be seen as a generalisation of CSOP. Many standard algorithms like backjumping, backmarking, arc-consistency, forward checking and branch and bound were customised to work with PCSP.

*Constraint hierarchies* by Alan Borning et all [6] is another approach of handling over-constrained problems. The constraint is weakened explicitly here by specifying its strength or preference. It allows one to specify not only the constraints that are required to hold, but also weaker, so called soft constraints. Intuitively, the hierarchy does not permit to the weakest constraints to influence the result at the expense of dissatisfaction of a stronger constraint. Moreover, constraint hierarchies allow "relaxing" of constraints with the same strength by applying, e.g., weighted-sum, least-squares or similar methods.

Currently two groups of constraint hierarchy solvers can be identified, namely refining method and local propagation. While the *refining methods* solve the constraints starting from the strongest level and continuing to weaker levels, the *local propagation* algorithms gradually solve constraint hierarchies by repeatedly selecting uniquely satisfiable constraints. In this technique, a single constraint is used to determine the value for a

variable. Once this variable's value is known, the system may be able to use another constraint to find a value for another variable, and so forth. This straightforward execution phase is paid off by a foregoing planning phase that chooses the order of constraints to satisfy.

Note finally, that PCSP is more relevant to satisfaction of constraints over finite domains, whereas constraint hierarchy is a general approach that is suitable for all types of constraints.

## 4 Applications

Constraint programming has been successfully applied to many different problem areas as diverse as DNA structure analysis, time-tabling for hospitals or industry scheduling. It proves itself to be well adapted to solving real-life problems because many application domains evoke constraint description naturally.

*Assignment problems* were perhaps the first type of industrial application that were solved with the constraint tools. A typical example is the stand allocation for airports, where aircraft must be parked on the available stand during the stay at airport (Roissy airport in Paris) [10] or counter allocation for departure halls (Hong Kong International Airport) [8]. Another example is berth allocation to ships in the harbour (Hong Kong International Terminals) [32] or refinery berth allocation.

Another typical constraint application area is *personnel assignment* where work rules and regulations impose difficult constraints. The important aspect in these problems is the requirement to balance work among different persons. Systems like Gymnaste [7] were developed for production of rosters for nurses in hospitals, for crew assignment to flights (SAS, British Airways, Swissair) or stuff assignment in railways companies (SNCF, Italian Railway Company) [11].

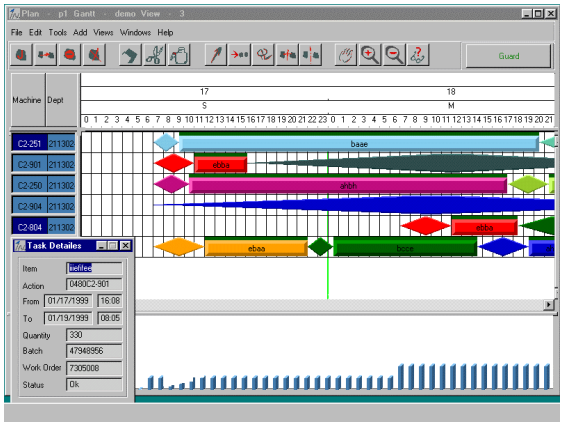


Figure 9
Schedule description using Gantt charts

Probably the most successful application area for finite domain constraints are *scheduling problems*, where, again, constraints express naturally the real-life limitations. Constraint based software is used for well-activity scheduling (Saga Petroleum) [22], forest treatment scheduling [1], production scheduling in plastic industry (InSol) [44] or for planning production of military and business jets (Dassault Aviation) [3]. The usage of constraints in Advanced Planning and Scheduling systems even increases due to current trends of on-demand manufacturing.

Another large area of constraint application is *network management and configuration*. These problems include planning of cabling of the telecommunication networks in the building (France Telecom) or electric power network reconfiguration for maintenance scheduling without disrupting customer services (Ether) [9]. Another example is optimal placement of base stations in wireless indoor telecommunication networks [14].

There are many other areas than have been tackled using constraints. Recent applications include computer graphics (expressing geometric coherence in the case of scene analysis, drawing programs, user interfaces), natural language processing (construction of efficient parsers), database systems (to ensure and/or restore consistency of the data), molecular biology (DNA sequencing, chemical hypothesis reasoning), business applications (option trading), electrical engineering (to locate faults), circuit design (to compute layouts), transport problems etc.

## 5 Limitations

Extensive application usage of constraint programming in solving real-life problems uncovers a number of limitations and shortcomings of the current tools.

As many problems solved by CP belong to the area of NP-hard problems, the identification of restrictions that make the problem tracktable is very important both from the theoretical and the practical points of view. However, as with most approaches to NP-hard problems, efficiency of constraint programs is still unpredictable and the intuition is usually the most important part of decision when and how to use constraints. The most common problem stated by the users of the constraint systems is stability of the constraint model. Even small changes in a program or in the data can lead to a dramatic change in performance. Unfortunately, the process of performance debugging for a stable execution over a variety of input data, is currently not well understood.

Another problem is choosing the right constraint satisfaction technique for particular problem.

Sometimes fast blind search like chronological backtracking is more efficient than more expensive constraint propagation and vice versa.

A particular problem in many constraint models is the cost optimisation. Sometimes, it is very difficult to improve an initial solution, and a small improvement takes much more time than finding the initial solution. There is a trade off between “anytime” solution and “best” solution.

Constraint programs are incremental in some sense (they can add constraints dynamically) but they have no support for online constraint solving that is required in current changing environment. Most of the time, the constraint systems produce plans that are then executed but, the machines break down, planes are delayed and new orders come at the worst possible time. This requires fast rescheduling or upgrading the current solution to absorb unexpected events. Again, there is trade off between optimality of the solution that usually means tight schedule and less optimal but stable solution that absorbs small deviations.

## 6 Trends

The shortcomings of current constraint satisfaction systems mark the directions for the further development. Among them, *modelling* looks one of the most important. The discussions started about using of global constraints that encapsulate primitive constraints into a more efficient package (e.g., all-different constraint). A more general question concerns modelling languages to express constraint problems. Currently, most CP packages are either extensions of a programming language (CLP) or libraries that are used with conventional programming languages (ILOG Solver) [43]. Constraint modelling languages similar to algebraic description are introduced to simplify description of constraints (Numerica) [40] or visual modelling languages are used to generate constraint programs from visual drawings (VisOpt VML) [44].

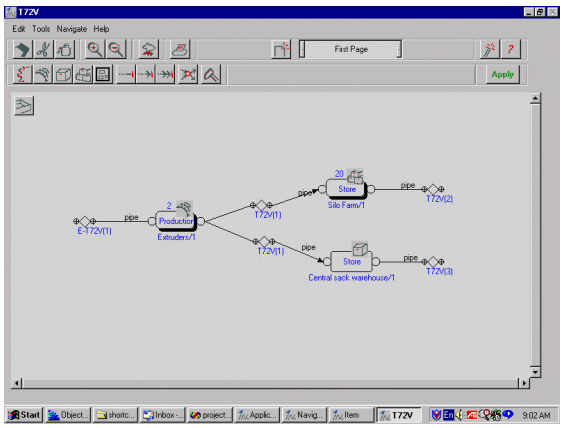


Figure 10
Visual Modelling Language in VisOpt

From the lower level point of view the visualisation techniques for *understanding search* becomes more popular as they help to identify the bottlenecks of the system [34]. Controlling search is probably one of the least developed parts of the constraint programming paradigm and in may problems the choice of search routine is done an ad-hoc basic.

The study of interactions of various constraint solving methods is one of the most challenging problems. The *hybrid algorithms* combining various constraint solving techniques are one of the results of this research [19].

Another interesting area of study is *solver collaboration* [28] and correlative combination of theories in prove theory. The combination of constraint satisfaction techniques with traditional OR methods like integer programming is another challenge of current research.

Last but not least, parallelism and concurrent constraint solving (cc) [39] are studied as methods for improving efficiency. In these systems, *multi-agent technology* looks very promising.

## 7 Summary

In the paper we give a survey of basic solving techniques behind constraint programming. In particular we concentrate on constraint satisfaction algorithms that solve constraints over finite domains. We also overview the main techniques of solving constraint optimisation problems and over-constrained problems. Finally, we list key application areas of constraint programming, describe current shortcomings and present some ideas of future development.

## 8 References

[1] Adhikary, J., Hasle, G., Misund, G.: Constraint Technology Applied to Forest Treatment Scheduling, in: Proc. of Practical Application of Constraint Technology (PACT97), London, UK, 1997

[2] Barták, R.: On-line Guide to Constraint Programming, Prague, 1998, http://kti.mff.cuni.cz/~bartak/constraints/

[3] Bellone, J., Chamard, A., Pradeless, C.: PLANE: - An Evolutive Planning System for Aircraft Production, in. Proc. of Practical Application of Prolog (PAP92), London, UK, 1992

[4] Berlandier, P.: Deux variations sur le theme de la consistance d’arc: maintien et renforcement, RR-2426, INRIA, 1994

[5] Borning, A.: The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory, in: ACM Transactions on Programming Languages and Systems 3(4): 252-387, 1981

[6] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, K., Wolf, M.: Constraint Hierarchies, in. Proc. ACM Conference on Object Oriented Programming Systems, Languages, and Applications, ACM, 1987

[7] Chan, P., Heus, K., Weil, G.: Nurse Scheduling with Global Constraintsa in CHIP: GYMNASTE, in: Proc of Practical Application of Constraint Technology (PACT98), London, UK, 1998

[8] Chow, K.P., Perett, M.: Airport Counter Allocation using Constraint Logic Programming, in: Proc. of Practical Application of Constraint Technology (PACT97), London, UK, 1997

[9] Creemers, T., Giralt, L.R., Riera, J., Ferrarons, C., Rocca, J., Corbella, X.: Constraint-Based Maintenance Scheduling on an Electric Power-Distribution Network, in Proc. of Practical Application of Prolog (PAP95), Paris, France, 1995

[10] Dincbas, M., Simonis, H.: APACHE – A Constraint Based, Automated Stand Allocation Systems, in: Proc. of Advanced Software Technology in Air Transport (ASTAIR91), London, UK, 1991

[11] Focacci, F., Lamma, E., Mello, P., Milano, M.: Constraint Logic Programming for the Crew Rostering Problem, in: Proc. of Practical Application of Constraint Technology (PACT97), London, UK, 1997

[12] Freuder, E.C.: Synthesizing Constraint Expressions, in: Communications ACM 21(11): 958-966, ACM, 1978

[13] Freuder, E.C., Wallace, R.J.: Partial Constraint Satisfaction, in: Artificial Intelligence, 1-3(58): 21-70, 1992

[14] Frühwirth, T., Brisset, P.: Optimal Placement of Base Stations in Wireless Indoor Telecommunication, in: Proc. of Principles and Practices of Constraint Programming (CP98), Pisa, Italy, 1998

[15] Gaschnig, J.: Performance Measurement and Analysis of Certain Search Algorithms, CMU-CS-79-124, Carnegie-Mellon University, 1979

[16] Gallaire, H., Logic Programming: Further Developments, in: IEEE Symposium on Logic Programming, Boston, IEEE, 1985

[17] Glover, F., Laguna, M.: Tabu Search, in: Modern Heuristics for Combinatorial Problems, Blackwell Scientific Publishing, Oxford, 1993

[18] Haralick, R.M., Elliot, G.L.: Increasing tree search efficiency for constraint satisfaction problems, in: Artificial Intelligence 14:263-314, 1980

[19] Jacquet-Lagreze, E.: Hybrid Methods for Large Scale Optimisation Problems: an OR Perspective, in: Proc. of Practical Application of Constraint Technology (PACT98), London, UK, 1998

[20] Jaffar, J. & Lassez J.L.: Constraint Logic Programming, in Proc. The ACM Symposium on Principles of Programming Languages, ACM, 1987

[21] Jaffar, J. & Maher, M.J.: Constraint Logic Programming – A Survey, J. Logic Programming, 19/20:503-581, 1996

[22] Johansen, B.S., Hasle, G.: Well Activity Scheduling – An Application of Constraint Reasoning, in: Proc. of Practial Applications of Constraint Technology (PACT97), London, UK, 1997

[23] Kumar, V.: Algorithms for Constraint Satisfaction Problems: A Survey, AI Magazine 13(1): 32-44,1992

[24] Lawler, E.W., Wood, D.E.: Branch-and-bound methods: a survey, in: Operations Research 14:699-719, 1966

[25] Mackworth, A.K.: Consistency in Networks of Relations, in: Artificial Intelligence 8(1): 99-118, 1977

[26] Marriot, K. & Stuckey. P.: Programming with Constraints: An Introduction, The MIT Press, Cambridge, Mass., 1998

[27] Minton, S., Johnston, M.D., Laird, P.: Minimising conflicts: a heuristic repair method for constraint satisfaction and scheduling problems, in: Artificial Intelligence 58(1-3):161-206, 1992

[28] Monfroy, E.: Solver Collaboration for Constraint Logic Programming, PhD Thesis, l'Universite Henri Poincare-Nancy I, 1996

[29] Montanary, U.: Networks of constraints fundamental properties and applications to picture processing, in: Information Sciences 7: 95-132, 1974

[30] Nadel, B.: Tree Search and Arc Consistency in Constraint Satisfaction Algorithms, in: Search in Artificial Intelligence, Springer-Verlag, New York, 1988

[31] Nilsson, N.J.: Principles of Artificial Intelligence, Tioga, Palo Alto, 1980

[32] Perett, M.: Using Constraint Logic Programming Techniques in Container Port Planning, in: ICL Technical Journal, May:537-545, 1991

[33] Selman, B., Kautz, H.: Domain-independent extensions to GSAT: Solving Large Structured Satisfiability Problems, in: Proc. IJCAI-93, 1993

[34] Simonis, H., Aggoun, A.: Search Tree Debugging, Technical Report, COSYTEC SA, 1997

[35] Smith, B.M.: A Tutorial on Constraint Programming, TR 95.14, University of Leeds,1995

[36] Sutherland I., Sketchpad: a man-machine graphical communication system, in: Proc. IFIP Spring Joint Computer Conference, 1963

[37] Tsang, E.: Foundations of Constraint Satisfaction, Academic Press, London, 1995

[38] van Hentenryck, P.: Constraint Satisfaction in Logic Programming, The MIT Press, Cambridge, Mass., 1989

[39] van Hentenryck, P., Saraswat, V., Deville, Y.: Design, Implementations and Evaluation of the Constraint Language cc(FD), Tech. Report CS-93-02, Brown University, 1992

[40] van Hentenryck, P., Michel, L., Deville, Y.: Numerica: A Modeling Language for Global Optimization, The MIT Press, Cambridge, Mass., 1997

[41] Waltz, D.L.: Understanding line drawings of scenes with shadows, in: Psychology of Computer Vision, McGraw-Hill, New York, 1975

[42] Wang, C,J., Tsang, E.P.K.: Solving constraint satisfaction problems using neural-networks, in: Proc. Second International Conference on Artificial Neural Networks, 1991

[43] ILOG, France, http://www.ilog.fr/

[44] InSol Ltd., Israel, http://www.insol.co.il/