Extract concerning Constraint Programming from

Constraint Relaxation Techniques

&

Knowledge Base Reuse,

A Ph.D. Thesis

by

Tomas Eric Nordlander

If you need to reference this document, please use this:

Nordlander, T.E., (2004) 'Constraint Relaxation Techniques & Knowledge Base Reuse', University of Aberdeen, PhD Thesis, pp. 246.

Table of Contents

Table of Figures

Table of Equations

Table of Codes

Table of Tables

List of Acronyms & Abbreviations

AC	Arc-Consistency
AI	Artificial Intelligence
BC	Back Checking
BJ	Back Jumping
BM	Back Marking
BT	Standard Back Tracking Algorithm
CBR	Case Base Reasoning
CLP	Constraint Logic Programming
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
FC	Forward Checking
FL	Full Look Ahead
GT	Generate and Test
KB	Knowledge Base

KBS	Knowledge Based System
MAC	Maintaining Arc Consistency
NC	Node-Consistency
OR	Operational Research
PC	Path-Consistency
PLA	Partial Look Ahead

'Many learned persons have read themselves stupid.'

Arthur Schopenhauer

1 Literature Review

1.1 Constraint Programming

Constraint programming (CP) has successfully been applied to many real-world problems since these problems can easily be modelled in terms of constraints, such as: scheduling, planning, configuration, layout, resource allocation, and decision support [100, 130]. Other areas where CP is used are: Concurrent computing, database systems, graphical interfaces, hardware verification, operations research and combinatorial optimisation [8, 45, 58, 107, 130]. In the eighties, constraint logic programming (CLP) appeared; the first general-purpose computational framework based on combining constraints and logic programming [58].

1.1.1 Constraint Satisfaction

Constraint Satisfaction techniques attempt to find solutions to constraint satisfaction problems (CSPs) [7, 127]. There are a number of efficient toolkits and languages available, for instance ILOG and SICStus [59, 113], especially designed to handle these problems.

1.1.1.1 CSP Definition

The definition of a Constraint Satisfaction Problem (CSP) is:

- 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 $C_{<j>} \subseteq D_{j1} \times D_{j2} \times \ldots \times D_{jt}$, restricting the values that subsets of the variables can take simultaneously.

A solution to a CSP is the assignment of a value from its domain to every variable, in such a way that all constraints are satisfied. The main CSP solution technique interleaves consistency enforcement [42], in which unfeasible values are removed from the problem through reasoning about the constraints, and various forms of backtracking search. The same approach also serves to identify unsolvable problems. Formulating the problem as a Constraint Satisfaction Problem tends to be less complicated than traditional Operational Research (OR) techniques (e.g. [119]). Though sometimes when fine-tuning the search for a CSP it requires remodelling in a more complicated fashion than the less expressive [97] OR techniques. In CSP variables and domain correlate directly to the problem entities and the values they can take. In some cases constraint satisfaction techniques may give a solution faster than OR techniques such as integer linear programming [8, 57, 118, 119].

1.1.1.2 Search Cost

Solving a CSP may be intended to achieve one of the following goals:

- demonstrate there is no solution;
- find any solution;
- find all solutions;
- find an optimal, or at least a good, solution given some objective evaluation function.

According to Freuder and Wallace [44], a standard measure of effort for a CSP algorithm is the number of constraint checks. Other properties such as time, backtracking, and resumption are also commonly used to measure the cost of the search, which depends on the following CSP properties:

- The structure of the problem; how the constraints interact to rule out assignments.
- The individual constraints; some constraints are cheap to test/propagate, while others are expensive. Some constraints even push the problem into areas where there are no efficient solving methods [26].
- The number of solutions in a best-solution search or in an all-solution search.

1.1.1.3 Constraint Arity

The constraint arity is the number of variables that the constraint is connected to. A 'unary constraint' constrains one single variable while a constraint that constrains two variables is called a 'binary constraint'. A commonly used notation of arity is a constraint that constrains the values of N variables and is a 'N-ary constraint'. Below (Table 2-1) are some example of constraints and their arity[1].

Constraint	Arity	Name
X_1#\=0	1	Unary constraint
X_1#\=X_2	2	Binary constraint
all_different (X_1,X_2,X_3)	3	Non-binary constraint with a arity of 3
…	…	…
X_1#\=X_2#\=…X_n	n	n-ary constraint

Table 2-1. Constraints with Different Arity

One of the reasons that researchers in the last 10 years, have mainly worked with binary constraints is that all constraints of an arity greater than 2 can be reformulated[2] and represented with binary constraints [5, 7]. For instance, the arity 3 constraint 'all_different(X_1,X_2,X_3)' can be reformulated to the following three binary constraints 'X_1#\=X_2, X_2#\=X_3, X_3#\=X_1. For more information about this binary representation of a non-binary constraint, see [5]. I have reservations about the practice of using solely binary constraints [5, 32, 31, 61, 102] and in section 4.1.2, I present the shortcomings of this practice and argue for introducing a mixture of different constraint arities.

1.1.2 Constraint Graph & Constraint Hyper-graph

[1] I used SICStus not-equal sign '#\=' from its constraint library over finite domains.
[2] Even though in practice this transformation is not likely to be worth doing.

Graphical representations of the binary CSPs are normally done with Constraint Graphs (left graph in Figure 2-1). The nodes of the graph represent the variables and the constraints between them are represented by the edges joining two of the nodes. Graphical representations of the non-binary CSPs are normally done with a Constraint Hyper-graph, where the nodes represent the variables and the constraint is circled around the variables that are involved in the constraint (right graph in Figure 2-1).

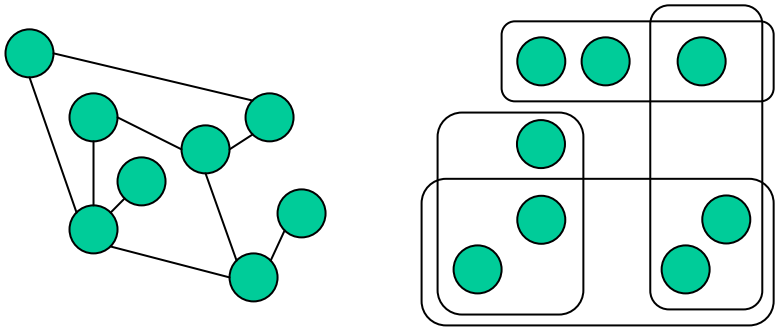


Figure 2-1. Constraint Graph & Constraint Hyper-Graph.

1.1.3 Search Methods

The majority of search algorithms systematically assign possible values to the variables. Although these types of algorithms are guaranteed to find existing solutions, they have the drawback of sometimes requiring a lot of time for the search. The effectiveness of an algorithm is normally judged by its time complexity; how long it takes to find the solution. Note that search is also commonly referred to as labelling.

One of the earlier systematic search algorithms is Generate and Test (GT) that starts with randomly generating a value for each variable and checking if the set is consistent with the existing constraints. The instantiation and checking procedure iterates until a solution is found or until all possible instantiations have been tried. The advantage of this algorithm is that it is easy to implement; however it also has two major drawbacks: *firstly*, the run-time complexity of the algorithm is exponential $O(\max(|D_i|)^n)$, where n is the number of variables and D is the domain size used. *Secondly*, the algorithm is rather inefficient because the algorithm does not memorise

previous inconsistent variable instantiations and it will continue to instantiate the same inconstant values to the variables.

The Standard Backtracking algorithm (BT) is a more commonly used systematic search algorithm and can be seen as a modified GT algorithm that surmounts the last shortcoming of the GT. After the first domain-value is instantiated to one of the variables, the Standard BT algorithm continues to instantiate another variable and checks for consistency against the first instantiation (partial solution). If consistent, it will extend the partial solution with the domain-value-instantiation and continue by instantiating the next variable, checking for consistency against previous partial solutions. This process will iterate until either a complete consistent assignment is found or no solution is found. A solution is detected when a complete consistent assignment is found, while no solution exists if no complete consistent assignment is found. If during this iterative process an inconsistency is detected in the BT instantiation process, it will ignore all further instantiations containing that partial solution and backtrack to the last successful variable instantiation and re-assign it with a new domain-value. This means that the algorithm avoids some of the inconsistent search space the GT would examine. If BT can find a solution without any backtracking its run-time complexity becomes linear. This is seldom the case as the most non-trivial problems require backtracking and the worst time complexity then becomes exponential $O(d^n m)$ and the space complexity linear $O(dn)$. In order to reduce the amount of backtracking it is possible to implement search heuristics (see section 2.2.5) that consider the ordering of variables and values in the instantiation process.

Standard BT has three drawbacks that affect its run-time complexity: *firstly*, Thrashing; which is the failure of BT to detect the actual variable that makes the partial instantiation test inconsistent. For example, if X_1 is instantiated with value D_a and the search continues instantiating values on variables X_2, X_3,…, X_n without realising that it is impossible to find any consistent assignment on these variables as long as D_a is instantiated to X_1. *Secondly*, the algorithm does redundant work: even when the reason for inconsistency is correctly detected, the reason would be forgotten when an identical inconsistency occurs in the iterative process. *Lastly*, the late inconsistency detection; the algorithm would only detect inconsistency after all variables in the partial assignments have been instantiated. Intelligent BT algorithms have been developed to

overcome the drawbacks of the standard BT, for example, Back-jumping (BJ), Back-marking (BM), or Back-checking (BC). These are all Intelligent BT 'Look Back algorithms' that, by using consistency to check among the assigned variables, can overcome the first two limitations of standard BT. When backtracking takes place, this algorithm can identify the source of inconsistency and backtrack to the place where the inconsistent variables were assigned. In spite of the fact that these algorithms normally perform better than standard BT, they still suffer from the drawback of only detecting inconsistency after the assignment has been made. Algorithms that manage to overcome the third weakness of standard BT enforce consistency techniques (see section 2.2.4) during search, to avoid any inconsistent domain sets values before the instantiation is done. Several of these so-called Intelligent BT 'Look Ahead algorithms' have been proposed, see section 2.2.4.5.

Even though search algorithms such as standard BT are guaranteed to find any existing solution and its run-time complexity becomes linear if no backtracking is needed, this is hardly ever the case, for the most non-trivial problems backtracking is needed and the run-time complexity becomes exponential. Intelligent BT algorithm bridges the three inadequacies of the standard BT which affect its run-time complexity, but they are sometimes so costly to apply that standard BT is preferred. For further information on search algorithms see [46, 67, 105, 118].

1.1.4 Consistency Algorithms

Consistency techniques were first introduced for picture recognition programs [132] and later successfully applied on different hard search problems [46]. Consistency techniques try to detect and remove inconsistent values from the domain sets of a variable but can seldom discard all inconsistent domain values for a problem. Because consistency algorithms do not remove any values that would take part in any solutions, they can be considered to transform the original CSP to an equivalent one. Note that although Consistency Algorithms are often called discrete relaxation algorithms, they are completely different from the relaxation algorithms I have introduced (See section 4.2).

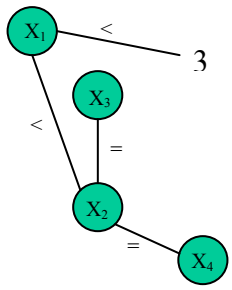
The effectiveness of the consistency algorithms is normally judged by how long it takes to find the solution (time complexity) as well as how much memory is

needed to perform the search (space complexity). Because these algorithms can not demonstrate consistency (are incomplete), they are more frequently used either interleaved with the search or before the search as a preparation phase to remove redundant domain values that might have been detected several times and thus slow down the search. Researchers have long worked under the assumption that consistency checks before the search are always valuable. It should be noted that empirical results [106] have shown that the consistency checking before the search can interfere with the interleaved checking inside a search algorithm, making the search with pre-processing consistency checks more costly.

1.1.4.1 Node-Consistency

Node-consistency algorithms check that each variable (nodes) connected to unary constraints are consistent. Node-consistency algorithms locate variables that are constrained with unary constraints. When such a variable is found, the algorithm checks each of the domain values of the variable against the unary constraint and removes those that violate the constraint. A variable is Node Consistent (NC) if all its domain size values satisfy the unary constraint and a CSP is considered NC if all the variables connected with a unary constraint are NC.

Figure 2-2 shows a CSP example where X_1 is the only variable with the unary constraint. After the algorithm locates X_1 it examines domain values of X_1 to see if any of the domain values violate the unary constraint. In the example X_1 domain values are reduced from {1,2,3,4,5,6,7,8,9,10} to {1,2}, because {3,4,5,6,7,8,9,10} violates the unary constraint that states the X_1 can only take a value less than 3.



Variable	Domain Size	Domain Size after Node-Consistency
X_1	{1,2,3,4,5,6,7,8,9,10}	{1,2}
X_2	{1,2,3,4,5,6,7,8,9,10}	{1,2,3,4,5,6,7,8,9,10}
X_3	{1,2,3,4,5,6,7,8,9,10}	{1,2,3,4,5,6,7,8,9,10}
X_4	{1,2,3,4,5,6,7,8,9,10}	{1,2,3,4,5,6,7,8,9,10}

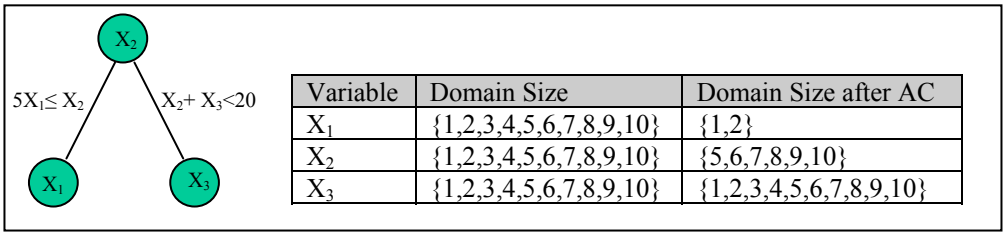
Figure 2-2. Reduction in Domain Size with Node-Consistency

The node-consistency algorithm can in some exceptional cases identify inconsistent CSPs but can never demonstrate a CSP to be consistent. If the algorithm removes all values in one of the variable domain sets, the CSP will then be inconsistent. Because the algorithm is incomplete it does not eliminate the need for search and should be used normally in a pre-processing phase to reduce the search. The node-consistency algorithm has a linear time-complexity of O(dn), where d is the maximum size of the domains and n is the number of variables to be examined.

1.1.4.2 Arc-Consistency

Arc-consistency is the most frequently used consistency technique; it checks the consistency for two variables connected with a constraint, and removes the domain values from the variables that violate the constraint. Many different Arc-consistency algorithms have been put forward; such as, AC-1 to AC-7 as well as variations of them (AC+3_d [12]). Even though most of the proposed algorithms are only applicable on binary CSPs, corresponding non-binary algorithms (e.g. NAC4 and GAC4 [26]) have been presented as well.

The procedure of making the CSP Arc Consistent (AC) is the iterative process of making the variable of each binary constraint consistent. Figure 2-3, shows an example where domain values that violate the binary constraints are detected and removed: X_1 and X_2 with the constraint $5X_1 \leq X_2$ is made AC by removing 3 to 10 from X_1 and 1 to 5 from X_2. None of the domain values for the X_3 and X_2 variables violate the constraint $X_2+X_3<20$, so it is already AC. Because all the CSPs constraints in Figure 2-3 are now AC, the whole CSP is AC.



Variable	Domain Size	Domain Size after AC
X_1	{1,2,3,4,5,6,7,8,9,10}	{1,2}
X_2	{1,2,3,4,5,6,7,8,9,10}	{5,6,7,8,9,10}
X_3	{1,2,3,4,5,6,7,8,9,10}	{1,2,3,4,5,6,7,8,9,10}

Figure 2-3. Reduction in Domain Size with Arc-Consistency

The arc-consistency can identify inconsistent CSPs only if one of its domain sets becomes empty during the consistency process. In addition, if every domain set only contains one value after the AC, the CSP is consistent and a solution is found. In spite of the fact that AC in some cases can identify CSPs as consistent or inconsistent, this is not the normal case [67]. More often, some domain sets will contain more than one value (e.g. example in Figure 2-3) after the CSP is made AC, which makes it impossible to demonstrate the CSP either consistent or inconsistent without a search; the arc-consistency algorithm is incomplete. Because AC does not normally eliminate the need for search it is used in the pre-processing phase to ease the search or is interweaved with the search.

Table 2-2 shows the cost in time and memory that the different arc-consistency algorithms have, where n is the number of nodes, d domain size, and e is the number of constraints. The best arc-consistency algorithm is now generally assumed to be AC-2001 [9, 16].

Arc-consistency Algorithm	Worst Time Complexity	Space Complexity
AC-1	$O(n^3d^3)$	O(e+nd)
AC-2	$O(ed^3)$	$O(n^2d^2)$
AC-3	$O(ed^3)$	O(e+nd)
AC-4	$O(ed^2)$	$O(ed^2)$
AC-6	$O(ed^2)$	O(ed)
AC-7	$O(ed^2)$	O(ed)
AC-2001	$O(ed^2)$	O(ed)

Table 2-2. Time- and Space-Complexity for Arc-consistency Algorithms [11, 12, 15, 16, 26, 67, 118]

1.1.4.3 **Path-Consistency**

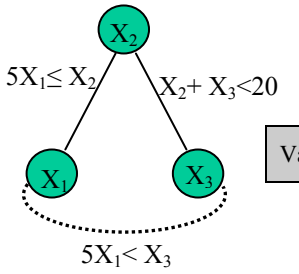
Path-consistency algorithms examine the consistency for three of the variables connected with two constraints and removes inconsistent domain values from the variables. A binary CSP is only Path Consistent (PC), if every possible path in the CSP is PC.

> *'A binary CSP is path-consistent, if for any path in its constraint graph it holds that if the assignments of the starting and ending variables are*

consistent, then this can be extended to a consistent partial instantiation by assigning values to the remaining variables along the path.' [105]

Several path-consistency algorithms (PC-1 to PC-5) have been proposed, which are normally enforced after making the problem AC. Figure 2-3 in the previous section, shows that the domain values are first reduced to make the CSP AC. From that example it is possible to see that even though the problem is made AC, it can still have inconsistent domain values; for example, there exists no consistent D_{X2} if X_1 is assigned the value 2 at the same time as the value 10 is assigned to X_3. This inconsistent assignment can be detected and discarded during the process of making the problem path consistent.

The path-consistency algorithm identifies all inconsistent instantiation of X_1 and X_3. In my example it is only $D_{X1}\{2\}$ and $D_{X3}\{10\}$ that causes an inconsistency. This means that there is an implicit constraint between X_1 and X_3 that forbids this tuple to be instantiated with these values at the same time. By adding a constraint equivalent to the implicit constraint to the constraint graph as shown in Figure 2-3, and thereafter enforcing AC on the three variables, the path is made path consistent. With the new constraint, the arc-consistency algorithm would now detect and remove {2} from X_1 domain set as well as {1,2,3,4,5} from X_3 domain set. After this procedure, the path of the triple X_1, X_2, and X_3 is PC. Because there is only one triple in my example the whole CSP has also become PC. If one of the domain sets becomes empty during the path-consistency then the CSP is inconsistent. If each domain set is left with only one value the CSP is consistent and the instantiation is the only solution. If some of the variable domain lists have more than one value, like example Figure 2-4, this does not normally demonstrate a CSP consistent or inconsistent, but in this case it does, due to n-consistency which is explained in section 2.2.4.



Variable	Domain Values	Domain Size After AC	Domain Size after PC

X_1	{1,2,3,4,5,6,7,8,9,10}	{1,2}	{1}
X_2	{1,2,3,4,5,6,7,8,9,10}	{5,6,7,8,9,10}	{5,6,7,8,9,10}
X_3	{1,2,3,4,5,6,7,8,9,10}	{1,2,3,4,5,6,7,8,9,10}	{6,7,8,9,10}

Figure 2-4. Example of Path-Consistency, partly based on [118]

In practice, while it is usually valuable to enforce arc-consistency, path-consistency is often not worth applying for three reasons. *Firstly*, even though the path-consistency algorithms remove more inconsistent values than arc-consistency algorithms the time-complexity is much worse (compare Table 2-2 with Table 2-3). The most efficient path-consistency algorithm has a worst case time-complexity of $O(d^3n^3)$ [67, 118]. The main reason for the large difference in time-complexity is that the number of possible triples in a CSP to investigate is much larger than the CSP's number of constraints. *Secondly*, the constraints are rarely expressed in allowed tuples, which makes it complicated to remove individual values in order to tighten a binary constraint [118]. *Thirdly*, because the algorithms add extra constraints into the constraint graphs (see Figure 2-3) there is a huge memory requirement even for small problems [8]. Although path-consistency removes more inconsistent values than any arc-consistency algorithm, it is incomplete and normally does not eliminate the need for search.

Path-consistency Algorithm	Worst Time Complexity	Space Complexity
PC-1	$O(d^5n^5)$	$O(d^2n^3)$
PC-2	$O(d^5n^3)$	$O(d^2n^3 + n^2)$
PC-3	$O(d^5n^3)$	$O(d^2n^3 + n^2)$
PC-4	$O(d^3n^3)$	$O(d^3n^3)$
PC-5	$O(d^3n^3)$	$O(d^3n^3)$

Table 2-3. Time- and Space-Complexity for Path-Consistency Algorithm [66, 125]

1.1.4.4 Obtaining n- and k-Consistency

Freuder states [42] that a CSP is k-consistent, if any set of k-1 variables surrounding constraints are consistent and there exists domain values for the k-th variable that makes all k variables consistent. In addition, he states that a CSP is said to be strongly k-consistent if it is [1-(k-1)]-consistent. The k-consistency and

the strong k-consistency definition allow a general notation for the three different consistency algorithms I have explained: node-consistency becomes strong 1-consistency, arc-consistency becomes strong 2-consistency, and path-consistency becomes strong 3-consistency. The time-complexity of the algorithm k-consistency is $O(dn^k)$. The maximum k value for k-consistency is the CSP's number of variables (n), which allows a CSP with n variables to become n-consistent. If a problem containing n nodes is strongly n-consistent, then a solution to the CSP can be found without any search [42, 67]: this happened in the example in Figure 2-4 which was made strongly n-consistent because the CSP only had 3 variables (k = n = 3). But if the problem is strongly k-consistent with k<n, undetected redundant domain values can still exist, and so search is needed. The time-complexity of the algorithm for obtaining n-consistency is $O(dn^n)$. For more information about consistency algorithms I recommend the following papers [12, 15, 42, 67, 71].

1.1.4.5 Search Algorithms that use Consistency Techniques

One of the weaknesses of standard BT Search algorithms is the late detection of inconsistency; the algorithm would only detect an inconsistency after all variables in the partial assignments have been instantiated. By combining Standard BT with consistency algorithms this limitation can be overcome by enforcing consistency techniques (see section 2.2.4) during search, to avoid any inconsistent domain set values before the instantiation is done. Several types of the latter algorithms, so-called *Look Ahead algorithms* have been proposed such as, Forward Checking (FC) Partial Look Ahead (PLA), Full Look (FL) and Maintaining Arc Consistency (MAC) etc. The two most frequently used are FC and MAC, which differ in the amount of consistency they enforce during search [106]; MAC enforces full arc-consistency [47, 106] while FC enforces a limited form of arc-consistency [55, 106]. Initial research made the wrong assumption that *Look Ahead algorithms* would do best in only enforcing a limited arc-consistency [67]; Sabin and Freuder [106] showed that establishing and maintaining full arc-consistency during search (MAC) was in many cases more efficient than only implementing partial arc-consistency (FC). Research [13, 52, 106] has now shown that different types of MAC-algorithms in general perform much better than FC; MAC is significantly better than FC around the transition phase (see section 2.2.7) but inferior in the clearly over- and under-constrained area. Note that although

the improved BT algorithms mentioned above manage to bridge the three inadequacies of the standard BT they are sometimes very costly to apply.

For example, the MAC algorithm uses backtracking interleaved with polynomial consistency techniques that have E combinations to consider; its worst time complexity becomes $E\times(O(edn^k))\times\max(|D_i|)^n$. If I take the MAC-4 algorithm that uses full AC-4 consistency checking, the total time complexity becomes $\sim k\times(O(edn^2))\times\max(|D_i|)^n$; $O(ed^2)$ for AC-4 and $\max(|Di|)^n$ for the time complexity of search. To guarantee completeness, you need n-consistency $O(dn^n)$, which is the same worst-case complexity as MAC.

1.1.4.6 Approaches that Reduce the Time Complexity

My research aim is to quickly identify inconsistent CSPs. A possible approach that reduces the time complexity involved, is to use the latest search and higher order of consistency algorithms described above, which have better worst time-complexity than those commonly used in constraint packages (e.g. [59, 113]). For the following reasons I am not interested using this approach:

1. The time-complexity of enforcing consistency algorithms is very high, and unless n-consistency is enforced it does not guarantee the detection of inconsistent CSPs. Achieving n-consistency can often be even more expensive than simple backtracking [67].

2. My research aim is to help KBS people examine if existing KBs can be reused. I can not assume that these people have the necessary knowledge to implement the latest search and higher k-consistency algorithms on the CSPs. Therefore standard CP toolkits are used. These toolkits have good search algorithms but normally can only enforce 1 and 2-consistency.

3. Enforcing higher k-consistency is complicated: real-world constraints are rarely expressed in allowed tuples (see section 4.1.3.1) and to enforce higher k-consistency the algorithms need to remove individual parts of values in order to tighten a binary constraint [118].

4. Because the higher k-consistency algorithms adds extra constraints into the constraint graphs they have substantial space requirements even for small problems [8].

5. Even the latest most efficient search algorithms such as MAC are sometimes very costly to apply.

My approach uses constraint relaxation strategies to quickly identify inconsistent CSPs. Through empirical investigation I have created constraint relaxation strategies, which relax the CSP by carefully removing constraints to create relaxed CSPs that are easier to demonstrate inconsistent. If the relaxed CSP is inconsistent the original CSP can be discarded without performing expensive search; empirical time-complexity reduction. This means the two approaches do not compete, because different search and consistency algorithms can and are used to demonstrate the relaxed CSP inconsistent. Consequently, my approach can be seen as a contribution and a complement to the existing search and consistency algorithms.

1.1.5 Search Heuristics.

When the search algorithm starts to instantiate the variables, it must know the order in which variables are to be considered. This so-called 'Variable Ordering' can either be static or dynamic. When using static ordering the variable order is decided before the search starts. The dynamic ordering starts either with an order list that might change depending on the state of the search or it starts by computing the next variable afresh each time. A common ordering is 'smallest domain' which starts working with variables with the smallest domain size. Other common ordering is 'minimum width ordering' and 'minimum conflict first'. After the search algorithm chooses a variable to instantiate another search heuristic comes in to play. This is the so-called 'Value Ordering'—the choice of domain value with which to start instantiating the variable.

One set of heuristics is not always better than another. The heuristics are chosen depending on the problem and search algorithm. These heuristic choices are essential for the performance of the search and demand expertise. Smith [118] used a small Cryptarithmetic puzzle to demonstrate that a specific heuristic can drastically

improve the result. Some current research is investigating how novices could get help choosing search heuristics from a Case Base Reasoning (CBR) system [48]. For more information on search heuristics see [46, 105, 118].

1.1.6 Random Generated CSPs

Ideally, researchers would use real-world problems for empirical analyses of different constraint satisfaction algorithms, such as search, relaxation, and consistency algorithms etc. However, it is hard to find sufficient numbers of analogous real-world problems to statistically verify the results of the algorithms, the usage of a randomly generated CSPs have been used as a substitute in the constraint community.

The main benefits of using random binary CSPs as a test-bed, is the large numbers of analogous problems that can easily be generated, which allows the researchers to statistically verify the results of their algorithms. Another benefit is a test-bed which produces examples with specific properties; for example, the possibility to generate a large number of problems close to the solution transition phase (for more information about the transition phase, see next section), where the problems are known to be harder [25, 96]. Hard areas like these are particularly suited for comparing the performance of different algorithms [17]. In addition, Bessière has highlighted the test-beds inter-changeability as one often forgotten advantage of using random CSPs. No particular domain knowledge is needed to understand the problems and neither do the problems contain sensitive or classified information. The above reasons make it relatively easy for researchers to replicate the problems, for example, when comparing the effectiveness of different algorithms. However, although I list benefits obtained by using random CSPs as test-beds, it should not be forgotten that the reason of random CSP existence is to substitute real-world problems.

Random binary CSPs are normally generated according to one of four common models; A, B, C, and D [70]. These are normally described by a 4-tuple **<n,m,p_1,p_2>,** where **n** is the number of variables and **m** is the number of values in

each domain[3], $\mathbf{p_1}$ is the density of the constraint graph (the proportion of constraints used in the CSP relative to the maximum number possible), and $\mathbf{p_2}$ is the tightness (the proportion of forbidden tuples in each constraint, see section 4.1.1). The four models differ in how the CSP constraints are created and chosen. Both A and C use probability $\mathbf{p_1}$ in selecting each one of the **(n(n-1)/2)** possible constraints, while B and D uniformly select exactly $\mathbf{p_1(n(n-1)/2)}$ constraints. Model A and D use probability $\mathbf{p_2}$ to select each of the $\mathbf{m^2}$ forbidden tuples, while B and C uniformly select exactly $\mathbf{p_2m^2}$ pairs as forbidden tuples.

Recently the problem class notation **<n,m,c,t>** is more frequently used: in this notation $\mathbf{p_2}$ is changed to **t** but continues to represent the number of forbidden tuples in each constraint (a measure of problem tightness). More important is that $\mathbf{p_1}$ (density) is replaced with **c** (the number of constraints in the CSP). Density ($\mathbf{p_1}$) describes how dense the CSP is with constraints; adding more constraints makes the CSP denser. Density is normally calculated as a percentage by Equation 2-1 (page 25), often this leads to working with a rounded off percentage which could lead to misinterpretations in the number of constraints used when the problem is recreated. The reason for the change to **c** in notation is practical—working with an integer is more exact then dealing with probabilities and proportions. For example, if the choice is made to create a CSP with 40 variables (n = 40) and 503 constraints (c = 503), the equation would give the CSP a density of ~0.6448718. Let us say that the density is then rounded down to 0.64 and placed in the problem class description <40**,m,**0.64**,**$\mathbf{p_2}$>. If the problem class needs to be replicated then 0.64 is used as the density. Equation 2-1. would be used to calculate the number of constraints required. With 0.64 the equation would then wrongly suggest 499 when the original problem class had 503 constraints. Working with a percentage here would create a completely new problem class when trying to replicate an old problem class. It is due to this reason I use the number of constraints **c** (integer value) as a density measurement in the problem class notation in this thesis.

Recent additions to these four models have been introduced after Achiloptas et al. [1] showed that all four models can generate flawed variables; a variable is

[3] Normally a random generated CSP uses the same domain size for all its variables.

flawed if every one of its domain values are unsupported. ‘A value for a variable is unsupported if, when the value is assigned to the variable, there exists an adjacent variable in the constraint graph that cannot be assigned a value without violating constraint’ [70]. Flawed variables make it easy to demonstrate the CSP inconsistent: no search is needed as a simple arc-consistency algorithm would find the CSP inconsistent. As my CSP-Suite is based on model B, a deeper discussion on how the findings of Achiloptas et al. influence the results of my experiments, is put forward in section 4.1.7.

Surprisingly little attention has been given to modifying the conventional CSPs (random binary CSPs with fixed internal tightness) to better simulate real-world problems. Although real-world problems involve different types of non-binary constraints with different arity, most CSP Generators, described in the literature, only work with binary constraints. Another important shortcoming of the conventional CSP Generators is that they fail to embody the constraint’s diversity in tightness that occurs in a real-world problem. An implementation of non-binary constraints of different arity and allowing implementation of different statistical tightness distribution into the CSP Generator would create CSP test-beds that more accurately simulate real-world problems. In section 4.1.5.2 I argue for the importance of implementing different binary and non-binary constraints into my CSPs generator. In section 4.1.2 I highlight the reasons, as well as explaining how different tightness distributions were implemented in my CSP Generator. One might argue that few parameters associated with conventional CSPs give the user a beneficial controlled test environment. My thesis will demonstrate that even when implementing non-binary constraints with different statistical internal tightness, the user is still very much in control of the test environment. The rationale of striving to implement real-world properties in a CSP Generator comes from the very purpose of random CSPs test-beds, that is, to substitute for real-world problems. I believe the conventional CSP test-bed still retains its purpose, when comparing a new algorithm with an earlier one. If an earlier algorithm cannot be applied to problems with non-binary CSP and different tightness distributions, the performance comparison should be conducted on the conventional CSP test-bed. If the algorithms cannot work on test-beds that better simulate real-world problems,

would they be applicable on real-world problems? If not, what is the reason behind their creation, what contribution do they make? Maybe it is time to update these algorithms so they can handle properties from real-world problems. I argue that the algorithms should evaluated, if possible, on Random CSP with properties as close to real-world problem before applying them on real-world problems.

1.1.7 Phase Transition Behaviour & Hardness Peak

The phase transition was first identified by [19, 38, 39] and has in recent years received considerable attention [25, 50, 69, 95, 96, 98, 139]. To study this phase transition phenomena [96], researchers generated large numbers of problem classes with fixed **n** and **m** but varying **c** and **t**. Within each of the problem classes a large number of CSPs are generated with identical parameters <**n,m,c,t**>. By measuring the number of CSPs in each problem class that have a solution it is possible to calculate the probability of finding a solution for a specific problem class. Figure 2-5 shows the phase transition, where the rapid change in the probability of finding a solution occurs when density and tightness parameters are varied on problem class 30<20,10,Stepped,Stepped>.

The transition phenomena occurs between an area where CSPs have many solutions and are easy to demonstrate consistent, and a region where most CSPs are inconsistent and it is relatively easy to verify that there is an inconsistency (many constraints allow effective pruning of domain values). In the phase transition, the probability of finding a solution drastically shifts from 100% to 0%. When working with a problem class close to the transition phase, small changes to some of the control parameters of the problem class such as adding constraints to the CSP (Density increase) or tightening the constraints of the CSPs, can push the CSP over the phase transition from consistent to inconsistent. Note that if several CSPs from the same problem class are generated in the middle of the transition phase, some CSPs would turn up inconsistent while others would be consistent, which makes it possible to calculate the probability of finding a solution for that problem class.

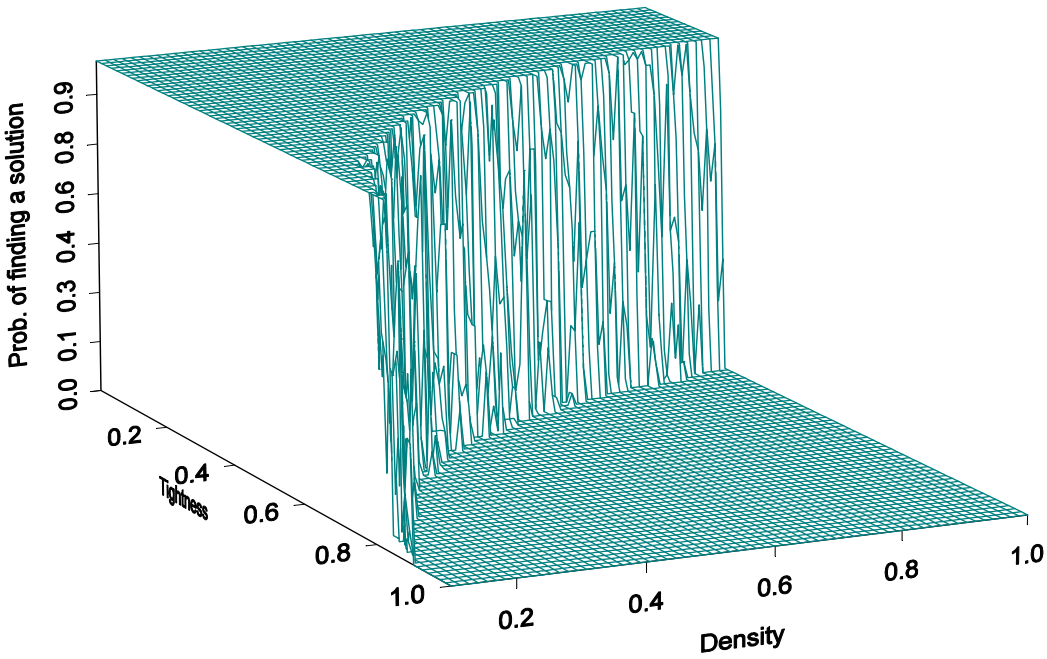


Figure 2-5. Solution Transition Phase for 30<20,10,Stepped,Stepped>

In Figure 2-6 instead of measuring the probability of finding a solution I have measured the average search effort (problem class hardness) when density and the tightness parameters are varied. The figure shows how hard CSPs from certain problem classes are, and research has shown [25, 96] that the hardness peak coincides with the transition phase, i.e. CSPs close to or on the solution transition phase are in general much harder to solve than others. This transition phase area of very hard CSPs is of special interest to the constraint community, as it supports empirical work, such as algorithm comparison [17].

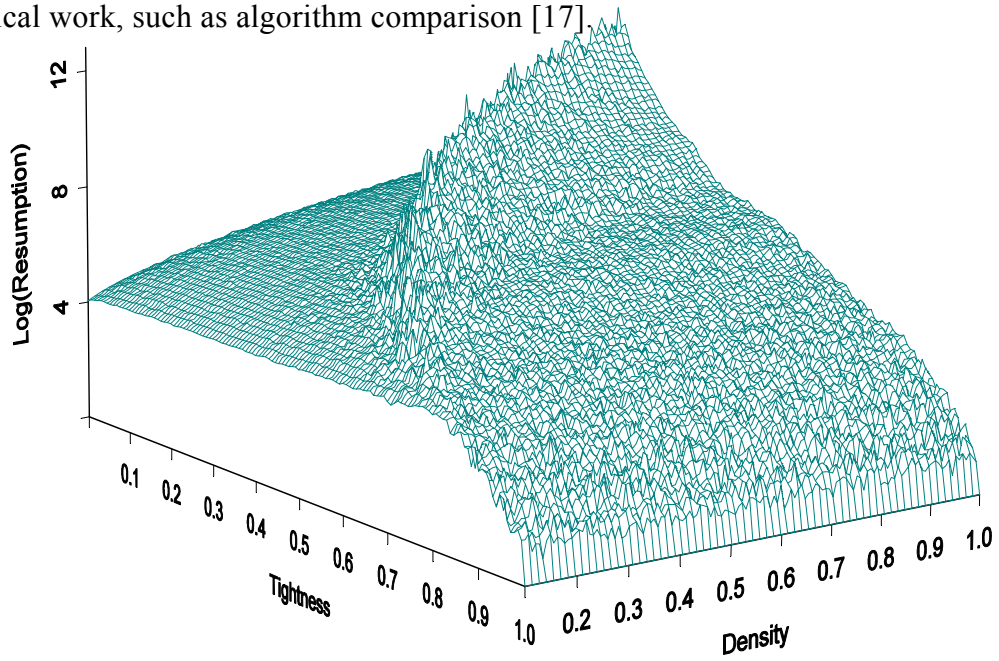


Figure 2-6. Hardness Peak for 30<20,10, Stepped,Stepped>

Figure 2-7 shows slices taken out of the 3D graph in Figure 2-6 and Figure 2-5, when the tightness is fixed to 0.45. Figure 2-7 shows that the transition phase coincides with the hardness peak and that the inconsistent CSP tends to become somewhat harder than the consistent CSP. The aim of my relaxation strategies is to detect these inconsistent CSPs using less search effort than is used by a normal search.

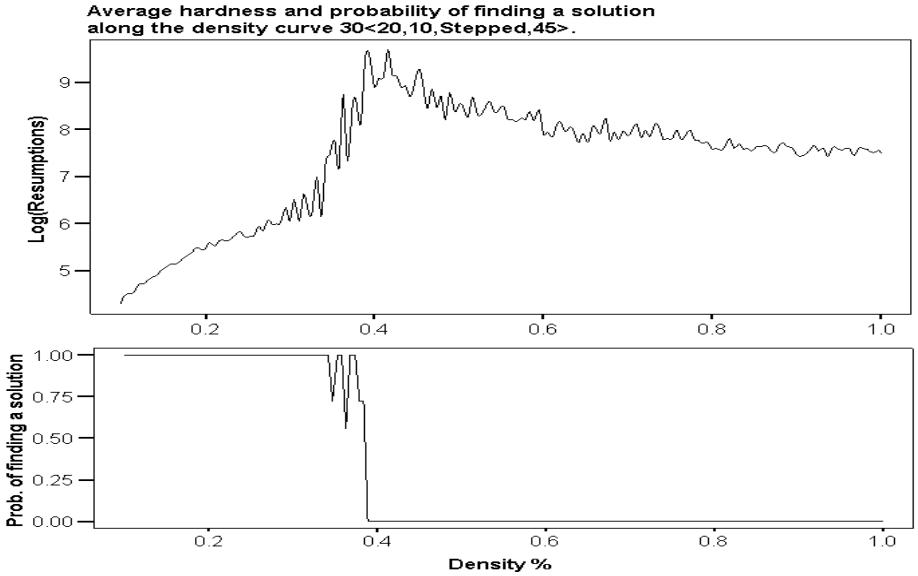


Figure 2-7. Density & Solution Transition Graph for 30<20,10,Stepped,0.45>

Research that tries to characterise problem hardness, for instance [103], and understand the behaviour of the phase transition is important in my research because my relaxation strategies (described in section 4.2) are only profitable when constraints are removed from an inconsistent CSP in such a way that new inconsistent CSP's, which are easier to solve, are produced. Due to the CSPs hardness peak at the transition phase (transition phase behaviour) this is a difficult task.

Gent et al. [50] presented Equation 2-2 that predicts the location of the phase transition. The relaxation strategies can use the equation when deciding the number of constraints to remove, to reduce the possibility of passing over the consistent side of the transition phase, i.e. the relaxed CSP will be inconsistent. Equation 2-2 predicts the location of the phase transition, where *k* is a measurement of *constrainedness* of the problem. Because I use the <n,m,c,t> notation while Gent et al. used the somewhat older notation <n,m,p_1,p_2> I replace p_1 in Equation 2-2 with the density Equation 2-1. Using this substitution, I get the phase transition

prediction equation (Equation 2-3) for the notation <n,m,c,t>. Gent et al. [50] have shown that for problem classes with <20,10, Step, Step> the phase-transition occurs between $0.75 \leq k \leq 1$. In my empirical chapter (section 5.2) there is a more detailed discussion of the transition phase behaviour influencing my research.

$$p_1 = \frac{c}{\left(\frac{(n(n-1))}{2}\right)}$$

Equation 2-1. Density Calculation

$$k = \frac{n-1}{2} p_1 \log_m\left(\frac{1}{1-t}\right)$$

Equation 2-2. Constrainedness

$$k = \frac{c}{n} \log_m\left(\frac{1}{1-t}\right)$$

Equation 2-3. Constrainedness using Number of Constraint instead of Density

1.1.8 Formulating the CSP

A real-world problem can be formulated as a CSP, in many different ways. To represent the problem as a CSP in an efficient way is difficult but important to get an efficient search. It has been shown [118] that a simple change in representation can dramatically improve the performance of the search algorithm. Ruttkay [105] claims that there are exceptionally complicated problems where reformulation has led to a solution. Evidently, the constraint programmers need knowledge and experience of CSP formulation. Until now, only experienced constraint programmers have sufficient knowledge to efficiently represent problems as CSPs. There is interesting ongoing research into easing the formulation problem for novice users with the help of the Case Based Systems approach [48]. Even though representation is difficult in constraint programming it is still considered easier than for OR techniques [97].

1.1.9 Examples of CSPs

There are many different types of combinatorial problem, which can be represented as a CSP. I will now describe three classical combinatorial problems and show how these can be formulated as CSPs. The problems are the Cryptarithmetic Puzzle, Graph Colouring, and a small scheduling problem. My formulations of the problems are written in SICStus Prolog, using its library over finite domains [21] and these CSPs are available online [89].

1.1.9.1 Cryptarithmetic Puzzles

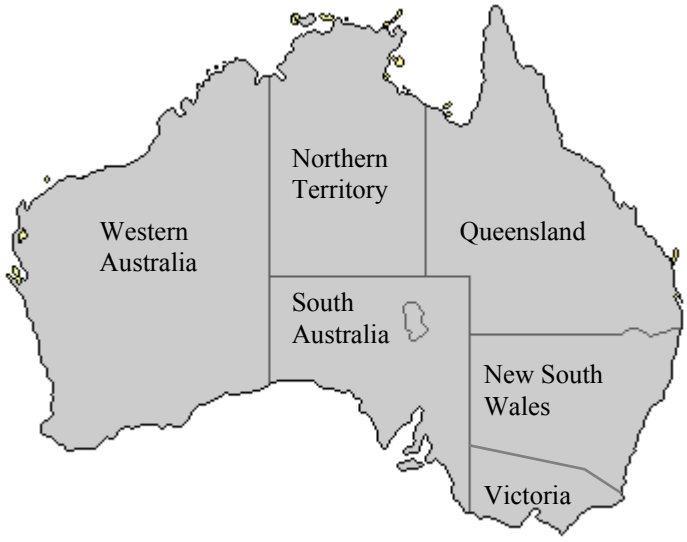
Cryptarithmetic Puzzles is a mathematical problem in which the digits are replaced by letters of the alphabet or other symbols. Solving the problem involves assigning each letter a digit, in such a way that the resulting mathematical calculation is correct. Code 2-2 shows the classical SEND+MORE=MONEY problem. Other possibilities are: SEVEN-NINE = EIGHT, CROSS + ROADS = DANGER, etc.

```
:- use_module(library(clpfd)).

mm([S,E,N,D,M,O,R,Y],Type) :-
  domain([S,E,N,D,M,O,R,Y],0,9),          % Variables and their domain size
  S#>0, M#>0,                             % Constraint
  all_different([S,E,N,D,M,O,R,Y]),       % Constraint
  sum(S,E,N,D,M,O,R,Y),                   % Call Equation Constraint

  labeling([],[S,E,N,D,M,O,R,Y]).         % Assign values to the variable

sum(S,E,N,D,M,O,R,Y) :-                   % Equation Constraint
1000*S+100*E+10*N+D+1000*M+100*O+10*R+E#=10000*M+1000*O+100*N+10*E+Y.

% End of file

| ?- mm([S,E,N,D,M,O,R,Y],Type).
D = 7,
E = 5,
M = 1,
N = 6,
O = 0,
R = 8,
S = 9,
Y = 2 ?
yes
| ?-
```

Code 2-1. Cryptarithmetic Puzzle 'Send+more=money', written in SICStus Prolog

1.1.9.2 Graph Colouring

Given a map and a number of different colours, the graph colouring problem is solved by colouring the map so that no regions sharing a boundary line have the same colour. Code 2-3 shows an example, in SICStus Prolog, of the colouring problem on the different states in Australia (See Figure 2-8).




```
:- use_module(library(clpfd)).

solve_AUSTRALIA(WA,NT,Q,SA,NSW,V):-

domain([WA,NT,Q,SA,NSW,V], 1, 3),          % Variables & their domain size colour 1=blue, 2=red
                                              or 3= green.

WA#\=NT,                                   % Constraints
WA#\= SA,                                  % WA abbreviation of Western Australia
NT#\= SA,                                  % NT abbreviation of Northern Territory
NT#\= Q,                                   % Q abbreviation of Queensland
SA#\=Q,                                    % SA abbreviation of South Australia
SA#\=NSW,                                  % NSW abbreviation of New South Wales
SA#\=V,                                    % V abbreviation of Victoria
Q#\=NSW,
NSW#\=V,

labeling([],[WA,NT,Q,SA,NSW,V]).           % assign values to the Variables

%% End of file

| ?- solve_AUSTRALIA(WA,NT,Q,SA,NSW,V).
Q = 1,
V = 1,
NT = 2,
SA = 3,
WA = 1,
NSW = 2 ?
yes
| ?-
```

1.1.9.3 Scheduling

Code 2-2. Graph Colouring Problem for the States of Australia, written in SICStus Prolog

Since my relaxation approach is exemplified on a production schedule problem, I show here an example of a scheduling problem. The problem below is taken from the SICStus manual [114]. The scheduling problem below aims to minimize the completion time for seven tasks without exceeding the resource capacity of 13. Each task has its specific duration, and resource needs. If this problem is put in a real-world context (See Code 2-4) with real-world scheduling variable names it could look like this: A manufacturing line is divided into seven workstations and has access to 13 workers. At each of the seven workstations a task is performed which has a minimum need of workers (resources) and will take a certain amount of time (duration). The manager would like to find the most efficient scheduling with regard to time and resources.

```
*/ TASK    DURATION          RESOURCE
  ====     ========          ========
  t1       16                2
  t2       6                 9
  t3       13                3
  t4       7                 7
  t5       5                 10
  t6       18                1
  t7       4                 11        /*

:- use_module(library(clpfd)).
:- use_module(library(lists), [append/3]).

schedule(Ss, Rs, End) :-
   length(Ss, 7),
   Ds = [16,6,13,7,5,18,4],
   Rs = [2,9,3,7,10,1,11],
   domain(Ss,1,30),
   domain([End],1,50),
   after(Ss, Ds, End),
   cumulative(Ss, Ds, Rs, 13),
   append(Ss, [End], Vars),
   labeling([minimize(End)], Vars).

after([], [], _).
after([S|Ss], [D|Ds], E) :-
E #>= S+D, after(Ss, Ds, E).

%% End of file



| ?- schedule(Ss,Rs,End).
Rs = [2,9,3,7,10,1,11],
Ss = [1,17,10,10,5,5,1],
End = 23 ?
yes
| ?-
```

Code 2-3. Simple Scheduling Program, written in SICStus Prolog [114]

'All things of this world are nothing, unless they have reference to the next.'

Spanish Proverb

Bibliography

1. Achiloptas, D., L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and C. Stamation, (1997) 'Random Constraint Satisfaction: A more accurate picture.' in *Third International Conference on Principles and Practice of Constraint Programming (CP97)*, Schloss Hagenberg, Austria: Springer Verlag, pp. 107-120

2. ACM, (1998) 'The ACM Computing Classification System', [WWW] Available from: http://www.acm.org/class/1998/ccs98.html [Accessed 5 August 2004]

3. AKT, (2003) 'Reuse Knowledge', The Advanced Knowledge Technologies project (AKT) [WWW] Available from: http://www.aktors.org/publications/reuse/ [Accessed 10 June 2004]

4. Anon, (2001) 'Timeline of the history of Artificial Intelligence', [WWW] Available from: Http://web.mit.edu/STS001/www/Team7/timeline.html [Accessed 7 August 2001]

5. Bacchus, F., X. Chen, P.v. Beek, and T. Walsh, (2002) 'Binary vs. non-binary constraints', *Artificial Intelligence*, Volume 140, Issue 1-2. September, pp. 1-37

6. Barker, V.E., D.E. O'Connor, J. Bachant, and E. Soloway, (1989) 'Expert systems for configuration at Digital: XCON and beyond', *Communications of the ACM*, Volume 32, Issue 3. pp. 298-318

7. Barták, R., (1998) 'Online Guide to Constraint Programming', Roman Barták [WWW] Available from: http://kti.ms.mff.cuni.cz/~bartak/constraints/binary.html [Accessed March 2003]

8. Barták, R., (1999) 'Constraint Programming: In Pursuit of the Holy Grail' in *Eigth Annual Conference of Doctoral Students (WDS'99)*, Prague, Czechoslovakia: MatFyzPress, pp. 555-564

9. Barták, R., (2004) 'Propagating Deletions in Tabular Constraints' in *Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming (CSCLP -2004)*, Lausanne, Switzerland, pp. 1-11

10. Bennett, J. and R. Engelmore, (1983) 'Experience using EMYCIN. In Rule-Based Expert Systems', E. Shortliffe, Editor, Addison-Wesley, London, pp. 314-328

11. Bessière, C., (1994) 'Arc-consistency and arc-consistency again', *Artificial Intelligence*, Volume 65, Issue 1. pp. 179-190

12. Bessière, C., E.C. Freuder, and J.-C. Regin, (1995) 'Using inference to reduce arc consistency computation.' in *Eleventh International Joint Conference on Artificial Intelligence (IJCAI'95)*, Montreal Quebec: Morgan Kaufmann Publishers, Inc, pp. 592--598

13. Bessière, C., (1996) 'MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems' in *Second International Conference on Principles and Practice of Constraint Programming (CP96)*, Cambridge, Massachusetts, USA: Springer-Verlag, pp. 61-75

14. Bessière, C., (1999) 'Non-binary constraints' in *Fifth International Conference on Principles and Practice of Constraint Programming (CP99)*, Alexandria VA, pp. 24-27

15. Bessière, C., E.C. Freuder, and J.-C. Regin, (1999) 'Using constraint metaknowledge to reduce arc consistency computation' in *Fifteen International Joint Conference on Artificial Intelligence (IJCAI'99)*, Montreal Quebec: Morgan Kaufmann Publishers, Inc, pp. 125-148

16. Bessière, C. and J.-C. Regin, (2001) 'Refining the basic constraint propagation algorithm' in *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, Seattle, WA, pp. 309-315

17. Bessière, C., (2004) 'Random Uniform CSP Generators', [WWW] Available from: http://www.lirmm.fr/~bessiere/generator.html [Accessed 3 July 2004]

18. Bistarelli, S., U. Montanari, and F. Rossi, (1995) 'Constraint Solving over Semi-rings' in *IJCAI*, pp. 624-630

19. Bollobás, B. and A. Thomason, (1987) 'Threshold functions', *Combinatorica*, Volume 7, Issue 1. January, pp. 35-38

20. Burke, P. and P. Prosser, (1991) 'A Distributed Asynchronous System for Predictive and Reactive Scheduling', *The International Journal for Artificial Intelligence in Engineering*, Volume 6, Issue 3. pp. 106-124

21. Carlsson, M., G. Ottosson, and B. Carlsson, (1997) 'An Open-Ended Finite Domain Constraint Solver' in *Programming Languages: Implementations, Logics, and Programs.*, pp. 345-381

22. Carlsson, M., (2004) 'Re: Run time complexity for the global constraints in SICStus', Google Newsgroups: comp.lang.prolog [WWW] Available from: http://groups.google.com/groups?selm=77bbf36a.0402030748.70045c4c%40posting.google.com [Accessed 2 January 2004]

23. Chandrasekaran, B., J.R. Josephson, and V.R. Benjamins, (1999) 'What Are Ontologies, and Why Do We Need Them?' *IEEE Intelligent Systems*, Volume 14, Issue 1. pp. 20-26

24. Chaudhri, V.K., M.E. Stickel, J.F. Thomere, and R.J. Waldinger, (2000) 'Using Prior Knowledge: Problems and Solutions' in *Seventeenth National Conference on Artificial Intelligence and Twelfth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI'00)*, Austin, Texas, USA: AAAI Press/MIT Press, pp. 437

25. Cheeseman, P., B. Kanefsky, and W.M. Taylor, (1991) 'Where the really hard problems are' in *Seventh International Joint Conference on Artificial Intelligence (IJCAI'91)*, Sydney, Australia, pp. 331-337

26. Chopra, R., R. Srihari, and A. Ralston, (1995) 'Hyper-Arc Consistency and Expensive Constraints', State University of New York at Buffalo, Technical Report: 95-21, pp. 1-6

27. Clarke, R., (1992) 'Fundamentals of 'Information Systems', [WWW] Available from: http://www.anu.edu.au/people/Roger.Clarke/SOS/ISFundas.html [Accessed 11 March 2004]

28. Collinot, A., C.L. Pape, and G. Pinoteau, (1988) 'SONIA --- a knowledge-based scheduling system.' *Artificial Intelligence in Engineering*, Volume 3, Issue 4. pp. 86-94

29. Compton, P. and R. Jansen, (1989) 'A philosophical basis for knowledge acquisition' in *Third European Workshop on Knowledge Acquisition for Knowledge-Based systems (EKAW'89)*, Paris, pp. 75-89

30. Davis, R., H. Shrobe, and P. Szolovits, (1993) 'What is a Knowledge Representation?' *AI Magazine*, Volume 14, Issue 1. pp. 17-33

31. Dechter, R. and J. Pearl, (1989) 'Tree clustering for constraint networks (research note)', *Artificial Intelligence*, Volume 38, Issue 3. April, pp. 353-366

32. Dechter, R., (1990) 'On the expresiveness of networks with hiden variables' in *Eighth National Conference on Artificial Intelligence (AAAI-90)*, Boston: MA, pp. 556-562

33. Devedzic, V., (2001) ' Knowledge modeling--State of the art', *Integrated Computer-Aided Engineering*, Volume 8, Issue 3. pp. 257-281

34. Dieng, R., O. Corby, A. Giboin, and M. Ribière, (1998) 'Methods and Tools for Corporate Knowledge Management' in *11th Banff Workshop on Knowledge Acquisition, Modelling and Management (KAW'98)*, pp. 42

35. Drucker, P.F., (1989) 'The new realities : in government and politics, in economics and business, in society and world view', New York: Harper & Row, pp. 276

36. Eisenstadt, M. and M. Brayshaw, (1990) 'Build your own knowledge engineering toolkit', [WWW] Available from: http://kmi.open.ac.uk/people/marc/mike/mike2.03/how-mike-works.htm [Accessed 17 March 2004]

37. Elleby, P., H.E. Fargher, and T.R. Addis, (1988) 'A Constraint-Based Scheduling System for VLSI Wafer Fabrication', Department of Computer Science, University of Reading, Technical Report: 1988b

38. Erdös, P. and A. Rényi, (1959) 'On random graphs', *Math. Debrecen*, Volume 6. pp. 290-297

39. Erdös, P. and A. Rényi, (1960) 'On the evolution of random graphs', *Publ. Math. Inst. Hungarian Acad. Sci.*, Volume 5. pp. 17-61

40. Feigenbaum, E., P. McCorduck, and H.P. Nii, (1988) 'The rise of the expert company', New york: MIT press

41. Fox, M.S., B.P. Allen, S.F. Smith, and G.A. Strohm, (1983) 'ISIS: A Constraint-Directed Reasoning Approach to Job Shop Scheduling System Summary', Robotics Institute, Carnegie Mellon University, Technical Report: CMU-RI-TR-83-08

42. Freuder, E., (1978) 'Synthesizing constraint expressions' in *Communication of the ACM (CACM)*, pp. 958-966

43. Freuder, E. and R. Wallace, (1992) 'Partial Constraint Satisfaction', *Artificial Intelligence*, Volume 58. pp. 2170

44. Freuder, E. and R. Wallace, (1992) 'Partial Constraint Satisfaction', *Artificial Intelligence*, Volume 58, Issue 1-3. December, pp. 21-70

45. Freuder, E.C., (1997) 'In Pursuit of the Holy Grail.' *Constraints*, Volume 2. pp. 57-61

46. Fruhwirth, T., A. Herold, V. Kuchenhoff, T.L. Provost, P. Lim, E. Monfroy, and M. Wallace, (1993) 'Constraint Logic Programming: An informal introduction', European Computer-Industry Research Centre, Technical Report: ECRC-93-5

47. Gashing, J., (1979) 'Performance measurement and analysis of certain search algorithms', Carnegie-Mellon University, PhD Thesis

48. Gebruers, C., A. Guerri, B. Hnich, and M. Milano, (2004) 'Making Choices at the Instance Level with a Case Based Reasoning Framework' in *Proceedings of the 1st International Conference on the Integration of AI and OR Technologies*, Nice, France: Springer

49. Gennari, J.H., M.A. Musen, R. Fergerson, and . (2003) 'The Evolution of Protégé: An Environment for Knowledge-Based Systems Development.' *International Journal of Human-Computer Studies*, Volume 58, Issue 1. pp. 89-123

50. Gent, I., E. MacIntyre, P. Prosser, and T. Walsh, (1996) 'The Constrainedness of Search' in *Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 246-252

51. Graner, N. and D. Sleeman, (1993) 'MUSKRAT: A Multistrategy Knowledge Refinement and Acquisition Toolbox.' in *Proceedings of the Second International Workshop on Multistrategy Learning*, Harpers Ferry, West Vergina, USA, pp. 107-119

52. Grant, S.A. and B.M. Smith, (1995) 'The phase transition behaviour of maintaining arc consistency', School of Computer Studies, University of Leeds, Technical Report: 95:25

53. Hadavi, K., W. Hsu, T. Chen, and C. Lee, (1992) 'An Architecture for Real-Time Distributed Scheduling', *AI Magazine*, Volume 13, Issue 3. pp. 46-56

54. Hameed, A., D. Sleeman, and A. Preece, (2001) 'Detecting Mismatches Among Experts' Ontologies Acquired through Knowledge Elicitation' in *21 st SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence*, Cambridge: Springer Verlag, pp. 9-22

55. Haralick, R.M. and G.L. Elliott, (1980) 'Increasing tree search efficiency for constraint satisfaction problems', *Artificial Intelligence*, Volume 14, Issue 3. October, pp. 263-313

56. Hayes-Roth, F., D.A. Waterman, and D.B. Lenat, (1983) 'Building Expert Systems'. Teknowledge series in knowledge engineering ; v. 1, Reading, Mass.: Addison-Wesley, pp. 444

57. Hentenryck, P.V. and J.P. Carillon, (1988) 'Generality versus specificity: An experience with AI and OR techniques.' in *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI'90)*, pp. 660-664

58. Hooker, J., (2000) 'Logic-based methods for optimization: combining optimization and constraint satisfaction', New York, pp. 209

59. ILOG Solver, (2003) Version: 5.3, from ILOG Inc., [WWW]: http://www.ilog.com/

60. SPSS for Windows, (2003) Version: 11, [WWW]: http://www.spss.com/

61. Janssen, P., P. Jegou, B. Jougier, and M.C. Vilarem, (1989) ' A filtering process for general constraint satisfaction problems: achieving pairewise-consistency using an associated binary representation.' in *IEEE-89 Workshop on Tools for Artificial Intelligence*, Ottawa, Ontario, Canada, pp. 420-427

62. Kalfoglou, Y., T. Menzies, E. Motta, and K.-D. Althoff, (2000) 'Metaknowledge in systems design: panacea...or undelivered promise', *The Knowledge Engineering Review*, Volume 15, Issue 4. pp. 381-404

63. Kiziltan, Z., P. Flener, and B. Hnich, (2001) 'Towards inferring labelling heuristics for CSP application domains' in *Advances in Artificial Intelligence, Joint German/Austrian Conference on AI (KI/ÖGAI)*, Vienna, Austria: Springer, pp. 275-289

64. Kodratoff, Y., D. Sleeman, M. Uszynski, K. Causse, and S. Craw, (1992) 'Building a Machine Learning Toolbox', in *Enhancing the knowledge engineering process : contributions from ESPRIT*, European Strategic Programme of Research and Development in Information Technology., Editor, North-Holland, Amsterdam ; New York, pp. 81-108

65. Kolodner, J.L., (1993) 'Case-Based Reasoning', San Mateo, CA: Morgan Kaufmann Publishers, pp. 668

66. Kreuger, P. and M. Bohlin, (2002) 'Introduction to constraint programming Lecture IV', SICStus Swedish Institute of Computer Science [WWW] Available from: http://www.idt.mdh.se/phd/courses/constraints/slides/cp-slides-IV.pdf [Accessed 13 July 2004]

67. Kumar, V., (1992) 'Algorithms for constraint satisfaction problems: a survey', *Artificial Intelligence Magazine*, Volume 13, Issue 1. pp. 32-44

68. Leeuwen, P.v., H.H. Hesselink, and J.H.T. Rohling, (2002) 'Scheduling Aircraft Using Constraint Satisfaction' in *11th International Workshop on Functional and (Constraint) Logic Programming (WFLP'2002)*, Grado, Italy, pp. 1-17

69. Lozinskii, E.L., (2004) 'Another look at the phenomenon of phase transition', [WWW] Available from: http://www.cs.huji.ac.il/~lozinski/tran.ps [Accessed 1 August 2004

70. MacIntyre, E., P. Prosser, B. Smith, and T. Walsh., (1998) 'Random Constraint Satisfaction: Theory meets Practice' in *Fourth International Conference on Principles and Practice of Constraint Programming (CP98)*, pp. 325-339

71. Mackworth, A.K. and E.C. Freuder, (1985) 'The complexity of some polynomial network consistency algorithms for constraint satisfaction problems [AC1-3]', *Artificial Intelligence*, Volume 25, Issue 1. pp. 65-74

72. Maple, (2001) Version: 7.00, from Waterloo Maple Inc, [WWW]: http://www.maplesoft.com/

73. S-Plus 2000, (1999) Version: 1, from MathSoft, Inc., [WWW]: http://www.mathsoft.com/

74. McCarthy, J., (2003) 'Applications of AI', [WWW] Available from: http://www-formal.stanford.edu/jmc/whatisai/node3.html [Accessed 16 March 2004]

75. MINITAB Statistical Software, (2003) Version: 13.31, from Minitab Inc, [WWW]: http://www.minitab.com/

76. Musen, M.A., N. Fridman, and M. Crubezy, (1999) 'Reusable Problem-Solving Methods', [WWW] Available from: http://smi-web.stanford.edu/courses/mis230/week8/index.htm [Accessed 17 March 2004]

77. Newman, D.R., (1998) 'Advantages of KBS', [WWW] Available from: http://www.qub.ac.uk/mgt/intsys/kbsadvan.html [Accessed 12 March 2004]

78. Newman, D.R., (1998) 'What are KBS used for', [WWW] Available from: http://www.qub.ac.uk/mgt/intsys/kbsused.html [Accessed 14 March 2004]

79. Nonaka, I. and H. Takeuchi, (1995) 'The knowledge-creating company : how Japanese companies create the dynamics of innovation', New York: Oxford University Press, pp. 284

80. Nordlander, T., (2001) 'AI Surveying: Artificial Intelligence in Business', Del Montfort University, MSc Thesis

81. Nordlander, T., (2002) 'First Year Report', University of Aberdeen, Report, pp. 10

82. Nordlander, T., D. Sleeman, and K. Brown, (2002) 'Manipulation of Constraints within the MUSKRAT framework', University of Aberdeen, Internal Report, pp. 14

83. Nordlander, T., K. Brown, and D. Sleeman, (2003) 'Constraint Relaxation Techniques to Aid the Reuse of Knowledge Bases and Problem Solvers' in *The Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, Cambridge: Springer Verlag, pp. 323-336

84. Nordlander, T., K. Brown, and D. Sleeman, (2003) 'Identifying Inconsistent CSPs by Relaxation', University of Aberdeen, Technical Report: TR0304

85. Nordlander, T., K. Brown, and D. Sleeman, (2003) 'Identifying inconsistent CSPs by Relaxation' in *Ninth International Conference on Principles and Practice of Constraint Programming (CP03)*, Cork, Ireland: Springer Verlag, pp. 987

86. Nordlander, T., D. Sleeman, and K. Brown, (2003) 'Exploring Relaxation Strategies in: Random Binary Constraint Satisfaction Problems', University of Aberdeen, Internal Report, pp. 1-18

87. Nordlander, T., D. Sleeman, and K. Brown, (2003) 'Exploring Relaxation Strategies in: Random Binary Constraint Satisfaction Problems with Internal Random Tightness', University of Aberdeen, Internal Report, pp. 1-19

88. Prototyp of a Mobile Phone Manufacturing Scheduling System, (2004) Version: 1.1, from University of Aberdeen, [WWW]: http://www.csd.abdn.ac.uk/~tnordlan/Proglog%20programs

89. Examples of CSPs, (2004) from University of Aberdeen, [WWW]: http://www.csd.abdn.ac.uk/~tnordlan/Proglog%20programs

90. Data from the Knowledge Base Survey, (2004) from University of Aberdeen, [WWW]: http://www.csd.abdn.ac.uk/~tnordlan/Proglog%20programs

91. Constraint Tightness Test Program, (2004) Version: 2.40, from University of Aberdeen, [WWW]: http://www.csd.abdn.ac.uk/~tnordlan/Proglog%20programs

92. Pape, C.L., (1994) 'Constraint-Based Programming for Scheduling: a Historical Perspective.' in *Working Notes of the Operations Research Society Seminar on Constraint Handling Techniques*, London, UK, pp. 12

93. Park, J.Y., J.H. Gennari, and M.A. Musen, (1998) 'Mappings for Reuse in Knowledge-based Systems' in *11 th Workshop on Knowledge Acquisition, Modelling and Management (KAW)*, Calgary Canada: Springer, pp. 1-21

94. Parunak, H.V.D., (1987) 'Manufacturing experience with the contract net.' in *Distributed Artificial Intelligence*, I.M.N. Huhns, Editor, Pitman, London, pp. 285-310

95. Pemberton, J.C. and W.X. Zhang, (1996) 'Epsilon-transformation: Exploiting phase transitions to solve combinatorial optimization problems', *Artificial Intelligence*, Volume 81, Issue 1-2. pp. 297- 325

96. Prosser, P., (1994) 'Binary constraint satisfaction problems: Some are harder than others' in *Eleventh European Conference on Artificial Intelligence (ECAI-94)*, Amsterdam, the Netherlands, pp. 95-99

97. Prosser, P. and I. Buchanan, (1994) 'Intelligent scheduling: past, present and future', *Intelligent system engineering*, Volume 3, Issue 2. Summer, pp. 67-78

98. Prosser, P., (1996) 'An empirical study of phase transitions in binary constraint satisfaction problems', *Artificial Intelligence*, Volume 81, Issue 1-2. March, pp. 81-109

99. Puppe, F., (1998) 'Knowledge Reuse among Diagnostic Problem Solving Methods in the Shell-Kit D3', *International Journal of Human-Computer Studies*, Volume 49, Issue 4. pp. 627-649

100. Purvis, L. and P. Jeavons, (1999) 'Constraint Tractability Theory And Its Application to the Product Development Process for a Constraint-Based Scheduler' in *Proceedings of the 1st International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP'99)*, London, United Kingdom, pp. 63-79

101. Rajpathak, D., E. Motta, and R. Roy, (2001) 'The Generic Task Ontology For Scheduling Applications' in *Proceedings of the International Conference on Artificial Intelligence (IC-AI'2001)*, Las Vegas, USA

102. Rossi, F., V. Dahr, and C. Petrie, (1990) 'On the equivalence of constraint satisfaction problems.' in *Nineth European Conference on Artificial Intelligence (ECAI-90)*, Stockholm, pp. 550-556

103. Ruan, Y., H.A. Kautz, and E. Horvitz:, (2004) 'The Backdoor Key: A Path to Understanding Problem Hardness.' in *Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI-04)*, San Jose, California, USA.: The MIT Press, pp. 12-130

104. Runcie, T., (2004) 'Reuse of Knowledge Bases and Problem Solvers Explored in the VT Domain', University of Aberdeen, Thesis Proposal Report, pp. 22

105. Ruttkay, Z., (1998) 'Constraint Satisfaction - a Survey', *CWI Quarterly*, Volume 11. pp. 123-161

106. Sabin, D. and E.C. Freuder, (1994) 'Contradicting Conventional Wisdom in Constraint Satisfaction' in *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming (PPCP-94)*, Washington, USA, pp. 10-20

107. Saraswat, V. and P.V. Hentenryck, (1995) 'Principles and practice of constraint programming: the Newport papers', Cambridge, Mass.: The MIT Press, pp. 475

108. Sauer, J. and H.-J. Appelrath, (1997) 'Knowledge-Based Design of Scheduling System' in *World Manufacturing Congress (WMC-97), International Symposium on Manufacturing Systems*, Auckland: ICSC Academic Press, pp. 247-252

109. Sauer, J. and R. Bruns, (1997) 'Knowledge-Based Scheduling Systems in Industry and Medicine', *IEEE-Expert*, Volume 12, Issue 1. pp. 24-31

110. Schiex, T., H. Fargier, and G. Verfaillie, (1995) 'Valued Constraint Satisfaction Problems: hard and easy problems' in *IJCAI*, pp. 631-637

111. Schreiber, G., H. Akkermans, A. Anjewierden, R.d. Hoog, N. Shadbolt, W.V.d. Velde, and B. Wielinga, (1999) 'KNOWLEDGE ENGINEERING AND MANAGEMENT The CommonKADS Methodology', MIT press, pp. 91

112. Shortliffe, E.H., (1976) 'Computer-based medical consultations, MYCIN'. Artificial Intelligence Series, New York, USA: Elsevier, pp. 264

113. SICStus Prolog, (2001) Version: 3.10.0, from Swedish Institute of Computer Science, [WWW]: http://www.sics.se/sicstus/

114. SICStus, (2002) 'Constraint Logic Programming over Finite Domains', in *SICStus Prolog User's Manual*, R. 3.9.1, Editor, Intelligent Systems Laboratory Swedish Institute of Computer Science, Kista, pp. 357-359

115. Sleeman, D. and S. White, (2002) 'Uses of a Grammar-Driven Case Acquisition Tool', University of Aberdeen, Technical Report: AUCS/TR0202

116. Sleeman, D., (2003) 'Knowledge Acquisition/Capture 'TYPES of KNOWLEDGE'', [WWW] Available from: http://www.csd.abdn.ac.uk/~sleeman/abdn.only/teaching/CS4021/info-LecturesandPracticals/Lectures/kt/KA-3-actual.ppt [Accessed 17 March 2004]

117. Sleeman, D., Y. Zhang, and W. Vasconcelos, (2003) 'Characterisation of Knowledge Bases' in *The Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, Cambridge: Springer Verlag, pp. 235-246

118. Smith, B.M., (1995) 'A Tutorial on Constraint Programming', University of Leeds, Technical Report: 95.14

119. Smith, B.M., S.C. Brailsford, P.M. Hubbard, and H.P. Williams, (1995) 'The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared', University of Leeds, Technical Report: 95.8

120. Smith, S.F., N. Muscettola, D.C. Matthys, P.S. Ow, and J.-Y. Potvin, (1990) 'OPIS: An Opportunistic Factory Scheduling System.' in *Third International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-90)*, Knoxville, Texas, pp. 268-274

121. Smith, S.F., (1991) 'Knowledge-Based Production Management: Approaches, Results and Prospects', Robotics Institute at Carnegie Mellon University, Technical Report: CMU-RI-TR-91-21

122. Smith, S.F. and M.A. Becker, (1997) 'An Ontology for Constructing Scheduling Systems' in *Fourteenth National Conference on Artificial Intelligence:Spring Symposium on Ontological Engineering (AAAI'97)*, Providence, Rhode Island, USA: AAAI Press, pp. 1-10

123. SPSS for Windows, (2003) Version: 11, [WWW]: http://www.spss.com/

124. Stergiou, K. and T. Walsh, (1999) 'The Difference All-Difference Makes' in *Fifteen International Joint Conference on Artificial Intelligence (IJCAI'99)*, Montreal Quebec: Morgan Kaufmann Publishers, Inc, pp. 414-419

125. Sukpan, A., (2004) 'A Survey on Constraint Satisfaction Problems', Department of computing science at University of Regina [WWW] Available from: http://www2.cs.uregina.ca/~sukpan1a/csp/csp.htm#2.2.3%20Path%20Consistency [Accessed 13 July 2004]

126. Szykman, S. and R.D. Sriram, (2001) 'The role of knowledge in next-generation product development systems', *Journal of Computation and Information Science in Engineering*, Volume 1, Issue 1. March, pp. 3-11

127. Tsang, E., (1993) 'Foundations of Constraint Satisfaction', Academic Press, London & San Diego, pp. 421

128. Uschold, M., P. Clark, M. Healy, K. Williamson, and S. Woods, (1998) 'An Experiment in Ontology Reuse' in *11th Banff Knowledge Acquisition Workshop (KAW'98)*, Calgary Canada: SRDG Publications, pp. 33

129. Vasconcelos, W.W. and M.A.T. Aragao, (2000) 'Slicing Knowledge-Based Systems: Techniques and Applications', *Knowledge-Based Systems Journal.*, Volume 13, Issue 4. pp. 177-198

130. Wallace, M., (1996) 'Practical application of constraint programming', *Constraints*, Volume 1, Issue 1-2. pp. 139-168

131. Walsh, T., (2001) 'Search on high degree graphs' in *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, Seattle, WA, pp. 266-274

132. Waltz, D., (1972) 'Generating semantic description from drawing of scenes with shadows', MIT, Technical Report: AI271

133. White, S. and D. Sleeman, (1998) 'Providing Advice on the Acquisition and Reuse of Knowledge Bases in Problem Solving' in *11th Banff Knowledge Acquisition Workshop (KAW'98)*, Calgary Canada: SRDG Publications, pp. 21

134. White, S. and D. Sleeman, (1999) 'A Constraint-Based Approach to the Description of Competence' in *Workshop on Knowledge Acquisition, Modelling, and Management (EKAW'99)*, Dagstuhl Castle, Germany: Springer Verlag, pp. 291-308

135. White, S., (2000) 'Enhancing Knowledge Acquisition with Constraint Technology', University of Aberdeen, PhD Thesis, pp. 345

136. White, S. and D. Sleeman, (2000) 'A Constraint-Based Approach to the Description & Detection of Fitness-for-Purpose', *Electronic Transactions on Artificial Intelligence (ETAI)*, Volume 4. pp. 155-183

137. White, S. and D. Sleeman, (2001) 'A Grammar-Driven Knowledge Acquisition Tool that incorporates Constraint Propagation' in *Proceedings of the international conference on Knowledge capture (KCAP-01)*, Victoria, British Columbia, Canada: ACM Press, New York, USA, pp. 187-193

138. Wielinga, B.J., A.T. Schreiber, and J.A. Breuker, (1992) 'KADS: a modelling approach to knowledge engineering.' *Knowledge Acquisition*, Volume 4. pp. 5-53

139. Xu, K. and W. Li, (2000) 'Exact Phase Transitions in Random Constraint Satisfaction Problems', *Journal of Artificial Intelligence Research*, Volume 12. pp. 93-103