Constraint Satisfaction from a Deductive Viewpoint*

by
Wolfgang Bibel

Technical Report 87-40

December 1987

Constraint Satisfaction from a Deductive Viewpoint*

W. Bibel

University of British Columbia and
Canadian Institute for Advanced Research

Abstract

This paper reports the result of testing the author's proof techniques on the class of constraint satisfaction problems (CSP). This experiment has been successful in the sense that a completely general proof technique turns out to behave well also for this special class of problems which itself has received considerable attention in the community. So at the same time the paper happens to present a new (deductive) mechanism for solving constraint satisfaction problems that is of interest in its own right. This mechanism may be characterized as a bottom-up, lazy-evaluation technique which reduces any such problem to the problem of evaluating a database expression typically involving a number of joins. A way of computing such an expression is proposed.

1 Introduction

Many tasks in Intellectics[1] can be seen as constraint satisfaction problems (CSP). Usually such problems are represented using constraint satisfaction networks [16]. For this kind of representation algorithms have been developed [10] that provide a solution to such problems without suffering from the maladies inherent in straightforward backtracking techniques. Although these algorithms are quite satisfactory in many applications it is known that the class of constraint satisfaction problems is NP-complete so that a polynomial solution might not exist for the general case [12].

*Submitted to the Artificial Intelligence Journal

[1]the field of Artificial Intelligence and Cognitive Science [4]

Basically, constraint satisfaction is a deductive problem of a very special form [10]. Therefore one might expect that the algorithms developed for the network representation may be incorporated into existing deductive mechanisms in order to speed–up their performance for such special cases. This has recently been explored in [8].

From the viewpoint of research in deductive mechanisms it would, however, be even more satisfactory if the same performance experienced with these special algorithms would come along as an aside with the improvement of the general deductive mechanisms. This has been attempted in [7] with the general goal in mind to eliminate the redundancies experienced in the backtracking behavior of deductive systems like PROLOG. A recent experimental study [14] compares the resulting behavior with that of the specialized algorithms mentioned above.

Rather than putting more intelligence into backtracking, [6] proposes a different approach whereby the redundancy is eliminated by way of transforming a given problem with a number of preprocessing operations. This technique was developed independent of any particular application (such as constraint satisfaction) in mind. Because of the importance of constraint satisfaction problems, we were interested to see how this technique would cope with them. This note reports the result of this experiment.

The most important aspect of this result is the observed fact that a completely general proof technique turns out to behave well also for the particular and important case of constraint satisfaction problems which is demonstrated in detail in the paper. This is by far not an obvious result since there are numerous examples in the literature where this desired behavior could not be achieved (e.g. the case of "sort"–problems [6]). As a side–result, but obviously of importance in its own right, the instantiation of our proof technique to the special case of CSP may be regarded as a new mechanism for solving constraint satisfaction problems. Finally, our technique brings together three different areas, namely logic along with its deductive machinery, CSP, and database technology. It therefore opens a wide field for further analysis and comparison.

As stated before, we approach the problem of constraint satisfaction from the deductive side which starts with a logical representation of any particular CSP. This representation models the claim that for a given world description (consisting merely of facts in the case of CSP) there exist values satisfying the constraints. In order to reduce the redundancy inherent in this kind of representation, all facts in the world description, that involve the same predicate, are combined into a single fact with an argument that ranges over the entries of a database table defined by the variety of these facts. The deductive task remaining after this reduction consists of a linear sequence of resolution steps with no search involved since the burden is shifted into the unificational part.

Namely, unification now needs to take into account database operations. The solution may first be computed symbolically as an expression in the relations (represented by the database tables) and a number of *join*–operations that characterizes the set of solutions. Query optimization techniques allow then for a relatively efficient computation of such an expression. We indicate one such way that may be viewed as an association list technique — or as a non–first–normal–form representation of the relations to be joined together, to phrase it in dababase terminology.

The whole approach might be characterized as a bottom–up, lazy–evaluation technique. In so far it radically differs from the intelligent backtracking techniques, that are top–down oriented. The differences and similarities of our method in comparison with the techniques used in the network approach are less obvious. While here, under the control of the goal statement, we manipulate the statements describing the world under consideration, those operate on the network that combines the information from the goal statement along with the domain information. [11] mentions, however, that arc consistency is a particular sequence of semi–joins. A detailed comparison with the network approach is beyond the scope of this note.

As indicated earlier, this paper continues our line of work which attempts to enhance the performance of deductive systems by way of an elaborate and fast preprocessing component. Such a component is supposed to eliminate much of the redundancy that, from the viewpoint of current deductive procedures, is inherent in most problems. After this preparatory operation only a small search space is left over for the main deductive process in most cases. This basic philosophy is successfully realized in a system PROTHEO [1]. Further it has been shown in [6] that the same approach allows the efficient treatment of another class of problems. These problems deal with different sorts (or types) of objects in a way which again causes serious maladies for the usual backtracking mechanisms. In [2] the treatment of recursion is approached in this way. With the present paper we show, as its main result, that the feature appropriate for constraint satisfaction is already present in this uniform framework, a fine confirmation, we think, for our general approach to theorem proving.

In more general terms, the emerging generation of deductive systems features a refined structure. Instead of a single deductive strategy on the one hand and the unification procedure on the other they combine several more parts of similar importance. One is this preprocessing component with a special unification that involves database operations. Another is the special treatment of recursive loops in the set of possible connections (that define the deductive structure). Yet another is the integration of whole theories into the unification part that would previously have caused a heavy load for the deductive search

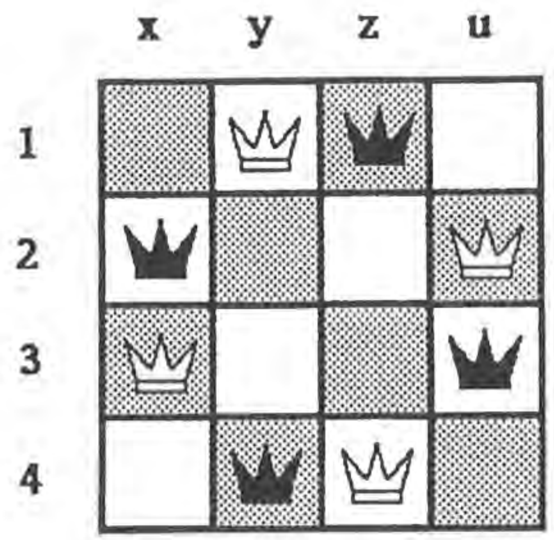

Figure 1: Two solutions for the 4–queens problem

part. A system, that would combine these and other parts in a well–balanced way, for a number of reasons is not yet under construction anywhere.

After stating the problem of constraint satisfaction in the following section the procedure for obtaining a solution to a given CSP is presented thereafter. In the final section a number of computational aspects are discussed.

2 The logical form of constraint satisfaction

The standard form of the constraint satisfaction problem, particularly for vision applications, is as follows. Let $V = \{v_1, \cdots, v_n\}$ be a set of variables. With each variable v_i a set L_i of values (or labels) is associated. Further, binary predicates P_{ij} are considered such that $P_{ij}(k, \ell)$ expresses that the assignment of label k to the variable v_i is compatible with the assignment of label ℓ to the variable v_j. With these notions the *Constraint Satisfaction Problem* (CSP_ℓ) is defined as the problem of finding an assignment of labels to the variables that does not violate the constraints given by P_{ij}.

Let us illustrate this definition with the well–known 4–queens problem as an example. The task consists in positioning four queens on a 4×4 chess board, so that none is attacked by any other. The board in figure 1 illustrates the two possible solutions.

The variables $V = \{x, y, z, u\}$ will be associated with the board's columns. $P_{ij}(k, \ell)$ holds if positioning a queen in row k at column i is safe for another queen in row ℓ and column j. The solution shown in the figure with the black queens is represented by the substitution $\{x\backslash 2, y\backslash 4, z\backslash 1, u\backslash 3\}$.

In order to state the 4–queens problem in a logical form, we use P as short for P_{12}, Q for P_{13}, R for P_{14}, S for P_{23}, T for P_{24}, and W for P_{34}, merely to improve readability. Hence $P_{12}(1,3)$ now briefly reads $P(1,3)$ which once more is abbreviated as $P13$. Again, this literal expresses that a queen in row 1 and column 1 is compatible with another queen in row 3 and column 2. In the following formula F each line expresses the possible alternatives for some pair of columns.

$$\begin{aligned}
&P13 \wedge P14 \wedge P24 \wedge P31 \wedge P41 \wedge P42 \wedge \\
&Q12 \wedge Q14 \wedge Q21 \wedge Q23 \wedge Q32 \wedge Q34 \wedge Q41 \wedge Q43 \wedge \\
&R12 \wedge R13 \wedge R21 \wedge R23 \wedge R24 \wedge R31 \wedge R32 \wedge R34 \wedge R42 \wedge R43 \wedge \\
&S13 \wedge S14 \wedge S24 \wedge S31 \wedge S41 \wedge S42 \wedge \\
&T12 \wedge T14 \wedge T21 \wedge T23 \wedge T32 \wedge T34 \wedge T41 \wedge T43 \wedge \\
&W13 \wedge W14 \wedge W24 \wedge W31 \wedge W41 \wedge W42
\end{aligned}$$

The task consists in determining a position in each column of the board so that for each pair of columns these positions are compatible. As a logical formula this task description altogether reads

$$F \rightarrow \exists x, y, z, u\, (Pxy \wedge Qxz \wedge Rxu \wedge Syz \wedge Tyu \wedge Wzu)$$

F formalizes the world description, the existential formula, let us call it G, states the goal to be achieved. A *Constraint Satisfaction Problem* (in the proof–theoretic version) is any problem that has exactly this form.

More formally, a *CSP* is a formula (without function symbols) of the form $F \rightarrow G$; the world description F is a conjunction of ground literals (listing all facts in this world); the goal G is an existential formula with a conjunction of literals (the *constraint*). A ***solution*** to a CSP consists in a constructive proof of the formula's validity (that automatically yields a substitution for the existential variables).

Lemma. Any CSP_ℓ may be transformed into a CSP in linear time, and vice versa.

The easy proof, illustrated by the example above, is left to the reader.

On the basis of this lemma the distinction between the two isomorphic versions may be abandoned. While in our definition the arity of the predicates is arbitrary, it is well known that we may restrict ourselves to the case of unary and binary predicates [10,13]. Moreover, the unary ones may be handled easily [10] so that in our discussions we will mostly focus on the binary ones although all results hold for the general case of arbitrary arities, as will be briefly discussed in the final section.

3 A deductive CSP procedure

The way it is stated in the previous section, solving a CSP means proving the formula's validity. Those readers familiar with the basic concepts in Automated Theorem Proving will recall that syntactically this in turn means, we need determine a spanning and unifiable set of connections [5,3]. In [6] we presented a more elaborate way of controlling the actions of a theorem prover to obtain a better performance. This general control turns out to be perfectly suitable for CSP as we are now going to demonstrate. For this purpose, consider again the previous formula for the 4–queens problem.

In order to eliminate the redundancy inherent in its presentation we (and the theorem prover) apply a reduction rule, introduced in [6] and briefly recalled here as follows. In any set–theoretic environment the following equivalence obviously is a valid one.

$$Pr \leftarrow r \in \{c_1, \ldots, c_k\} \leftrightarrow Pc_1 \wedge \ldots \wedge Pc_k$$

The substitution, in a logical formula, of the right side of this equivalence by the left side will be called a *DB–reduction*. Note that the literal $r \in \{c_1, \ldots, c_k\}$ for given values of r may be tested for its truth–value by database operations. For that reason we will use the convention to drop such a literal from this kind of logic program and treat the domain, that it defines, separately (or even implicitly). In fact, we take the view that this domain is indeed represented as a database table. So in short, instead of listing for each predicate all the facts associated with it separately, they are combined in a single literal with a special variable that ranges over all possible arguments collected in a database table.

Consider, for instance, the predicate P that occurs in the facts $P13, P14, P24, P31, P41, P42$. Let p be the variable that ranges over the values $(1,3)$, $(1,4)$, $(2,4)$, $(3,1)$, $(4,1)$, $(4,2)$ which we assume to be stored in a database table identified by p. Then the conjunction of all these facts reduces to the single literal Pp. Similar tables are created for Q, R, S, T, W in accordance with the formula F above in an obvious way. These tables and the corresponding variables are now tacitly assumed. The formula then reads as follows thereby using a familiar PROLOG–like notation.

$$\begin{tikzcd}[row sep=tiny]
Pp. \arrow[ddddddr, dash, bend left] & \\
Qq. & \\
Rr. & \\
Ss. & \\
Tt. & \\
Ww. & \\
& \leftarrow\ Pxy\,,\ Qxz\,,\ Rxu\,,\ Syz\,,\ Tyu\,,\ Wzu\,?
\end{tikzcd}$$

In this representation we have a very simple form of a PROLOG program in which for each predicate symbol there is exactly one head literal which moreover is simply a fact. Note that this property is not specific to our current example but holds for any CSP in general. In such a case another reduction rule introduced in [6] applies which is now briefly recalled. A connection is called *isolated* if none of its literals does occur in any other connection in the formula. For instance, the depicted connection $\{Pp, Pxy\}$ in the program above is obviously isolated in this sense. In fact, all connections in this program are isolated. Resolution upon an isolated connection and deletion of the parent clauses is called *ISOL–reduction.*

The definition of isolated connections may be liberalized [6] by allowing literals to occur in more than one connection if they are ground literals in unit clauses (i.e. facts). This is because the logic would not change if each connection uses its own instance of the ground literal as depicted in the following program.

$$\begin{tikzcd}[row sep=tiny]
Pp. \arrow[ddr, dash, bend left, "{Pyz}"'] & & \\
Pp. \arrow[dr, dash, bend left] & & \\
& \leftarrow\ Pxy\,,\ Pyz\,? &
\end{tikzcd}$$

Performing ISOL–reduction upon the connection depicted in the previous program above leads to the following remaining program.

$$\begin{tikzcd}[row sep=tiny]
Qq. \arrow[dddddr, dash, bend left] & \\
Rr. & \\
Ss. & \\
Tt. & \\
Ww. & \\
& \leftarrow\ Qxz\,,\ Rxu\,,\ Syz\,,\ Tyu\,,\ Wzu\,,\ (x,y) \in P\,?
\end{tikzcd}$$

The unification of the pair (x, y) with p resulted in the restriction of (x, y) to the entries in the database table for the predicate P (i.e. the domain of p) which is formally expressed in the newly added last literal of the goal clause. Note that we use P also to denote the database containing the p–values, a practice that we will follow similarly in the sequel. The next ISOL–reduction, i.e. resolution step upon the depicted Q–connection leads to

$$\begin{tikzcd}
Rr. \arrow[ddddr, bend left, no head] & \\
Ss. & \\
Tt. & \\
Ww. & \\
& \leftarrow \; \boldsymbol{Rxu\,,\, Syz\,,\, Tyu\,,\, Wzu\,,\; (x,y,z) \in P \bowtie Q\,?}
\end{tikzcd}$$

Here $\bowtie$ denotes the natural equi–join from relational algebra [15]. It takes the cross–product of P and Q, but restricts the result to those elements with equal values in the two x–positions and deletes the second (now redundant) x–position. Whenever we want to explicitly show the condition that is used in the join, we use a functional notation like $join(P,\, P.1 = Q.1,\, Q)$ instead of $P \bowtie Q$ in the present case. In this sort of expression we allow more than one such equality condition.

Let PQ denote $P \bowtie Q$ and PQR denote $PQ \bowtie R$. The next steps then yield the following sequence of intermediate results obtained in exactly the same way as the previous two just explained.

$$\begin{tikzcd}
Ss. \arrow[dddr, bend left, no head] & \\
Tt. & \\
Ww. & \\
& \leftarrow \; \boldsymbol{Syz\,,\, Tyu\,,\, Wzu\,,\; (x,y,z,u) \in PQ \bowtie R\,?}
\end{tikzcd}$$

$$\begin{tikzcd}
Tt. \arrow[ddr, bend left, no head] & \\
Ww. & \\
& \leftarrow \; \boldsymbol{Tyu\,,\, Wzu\,,} \\
& \boldsymbol{(x,y,z,u) \in join(PQR,\, PQR.2 = S.1 \wedge PQR.3 = S.2,\, S)\,?}
\end{tikzcd}$$

$$\begin{tikzcd}
Ww. \arrow[dr, bend left, no head] & \\
& \leftarrow \; \boldsymbol{Wzu\,,} \\
& \boldsymbol{(x,y,z,u) \in join(PQRS,\, PQRS.2 = T.1 \wedge PQRS.4 = T.2,\, T)\,?}
\end{tikzcd}$$

$$\leftarrow \quad \boldsymbol{(x,y,z,u) \in join(PQRS,\, PQRST.3 = W.1 \wedge PQRST.4 = W.2,\, W)\,?}$$

As the reader may have noticed, $PQRS$ abbreviates $join(PQR,\, PQR.2 = S.1 \wedge PQR.3 = S.2,\, S)$; similarly for $PQRST$ and $PQRSTW$. The last line gives all the values for x, y, z, u that are possible under the constraints in G and therefore constitute a solution for this CSP; they are represented as a relation $PQRSTW$ computed from the relation tables P, Q, R, S, T, W by altogether five join operations. In our example, $PQRSTW$ contains the two tuples $(2,4,1,3)$ and $(3,1,4,2)$ corresponding to the two configurations shown in figure 1 as we will show in detail in the next section.

No constants occurred in any of the goal literals of our present example. Assume there would be literals with constants as in the following slightly modified goal formula.

$$\leftarrow \quad P2y, Qx1, R2u, Syz, Tyu, Wzu\,?$$

Then obviously we could proceed as before. Namely, whenever a constant occurs in the literal the associated table would first be reduced to those of its elements that match with this constant in the appropriate position (called selection in databases) before continuing with a join if there are still variables left. In the presence of unary predicates we end up with doing a similar thing. Their connections would be treated prior to the remaining ones so that the (domain) variables ending up in the constraint literals by unification would then reduce the associated tables similarly as just described. This way we actually handle node inconsistency from the network approach. Altogether we obtain the following general procedure for solving a CSP.

CSP–Procedure

STEP1. Represent the CSP in logical form and apply DB–reduction.

STEP2. Resolve on all (isolated) connections (prefer unary predicates, otherwise apply any order), each resulting in a (further) restriction on the variables determined by database operations (such as the join).

STEP3. Compute the relation defined by the expression resulting in STEP 2 (see the next section).

Theorem. The final restriction resulting from performing STEP1 and STEP2 for a CSP determines a relation that contains exactly the solutions of the CSP.

Proof. The special logical form ensures that all connections must be isolated. Further there are no more such connections than literals in the goal. The procedure thus must terminate successfully if a solution exists.

From the correctness of the reduction operations from [6] it is then obvious that the execution of the resolution steps upon these connections yields the relation as described in the theorem.

4 Computational aspects

The mechanism reported in this paper has not yet been integrated into our current deductive system PROTHEO, a task which in a first straight–forward approach simply requires a suitable interface with an adequate database system. In fact the deductive part of the mechanism (STEP2) has become so simple for the special case of CSP that we do not even need much of the deductive power at all. Rather, the main computational problem has now been shifted into STEP3 which is its appropriate place (i.e. we consider this a particular virtue of our technique). The task to be performed there, in database terms, amounts to executing a query. As the expression resulting from STEP2 may be quite complicated, we have to be concerned about the computational needs for this part. This is a concern well–known in CSP and the database field, in the latter treated under the label of query–optimization [15]. On the basis of the results known from there we can provide a first approximation for the complexity of the behavior of our procedure.

The number of join–operations needed for obtaining the solution is bounded by the number of constraint literals in the constraint formula. More precisely, this number depends upon the goal's structure in the following way. Let us call two goal literals L_1, L_n *linked together* if there is a variable x contained in L_1 and L_n and a sequence of goal literals $L_1, L_2, \ldots, L_n$ such that any pair of subsequent literals share at least one variable. The transitive closure of this binary relation on the goal literals partitions the set of all goal literals from G into disjoint subsets $G_1, \ldots, G_m$ (in our 4–queens example we have $m = 1$). Then a solution of the CSP requires $|G_1| + \cdots + |G_m| - m$ join operations. Note that the solution for each subset may be obtained independently and is an independent part of the final solution for G. So from a complexity point of view we may restrict our attention to the case $m = 1$.

Let k denote the number of literals in G and n the maximal number of entries in each table (i.e. the cardinality of the relation). Then a join in the worst case produces $n \times n$ elements in the new table. The time complexity for executing the join is $o(n \log n)$ [15]. So in this crude way of estimation the total time complexity becomes $o(n^{k+1})$.

The behavior can be improved if hashing–techniques are used. Also, one might select an execution order for the sequence of joins that gives a preference to the smallest table among those to be treated or, more generally, to a sequence that minimizes the intermediate tables. In addition, an alternate way is provided by the use of a different represention of the tables that eliminates the redundancy of storing the information and takes account for the fact that none of the intermediate relations — PQ, PQR, ... in this note's standard example — has actually to be computed explicitly. We will briefly illustrate such a possibility that is related with association graphs used in AI and with non–first–normal–form representations used in databases.

The first step thus consists of a transformation of the representation of each relation as follows. While relations are usually represented by storing all its tuples explicitly (first normal form representation), we prefer for our purposes a representation that mentions in each column (i.e. attribute) any occurring value only once, whence we might speak of a value–oriented representation. For illustration recall the relation P from Section 2. It consists of the following table.

1	3
1	4
2	4
3	1
4	1
4	2

We define an operation ρ that, applied to P, yields the following representation of the same information.

1_1	1_{34}
2_2	2_4
3_3	3_1
4_4	4_{12}

That is, for any value in one of the columns, e.g. value 1 in the left column, we select any new index which only by coincidence here is 1 again. This index is assigned to each associated value in the second column of the original table. P can be reconstructed from ρP in the obvious way. Let us now assume we have done this for Q, R, S, T in a similar way.

Starting to work on the expression $PQRSTW$ from the previous section leads us first to joining P and Q whereby we use this representation, which is illustrated in the following table.

P		Q		$P \bowtie Q$		
1_1	1_{34}	1_5	1_{68}	1_1	1_{34}	1_{24}
2_2	2_4	2_6	2_{57}	2_2	2_4	2_{13}
3_3	3_1	3_7	3_{68}	3_3	3_1	3_{24}
4_4	4_{12}	4_8	4_{57}	4_4	4_{12}	4_{13}

The first columns of both relations are to be identified. So for any value occurring in both columns, e.g. for the value 1, this is achieved simply by renaming its index all over in one of the two predicates. The renaming was done in Q above, i.e. 1_5 became 1_1, 2_{57} became 2_{17}, and 4_{57} changed into 4_{17}. Doing this with all four values in the example and removing the obsolete second instance of the first column gives the result. The formal definition of this "merging" operation, denoted by $\bowtie_\rho$, derives from that of the usual join by way of the following equation.

$$\rho P \bowtie_\rho \rho Q = \rho(P \bowtie Q)$$

This is why we may actually ignore the distinction between $\bowtie$ and $\bowtie_\rho$. The following table shows the next step for our example which is of the same kind as the previous one.

$P \bowtie Q$			R		$(P \bowtie Q) \bowtie R$			
1_1	1_{34}	1_{24}	1_5	1_{67}	1_1	1_{34}	1_{24}	1_{23}
2_2	2_4	2_{13}	2_6	2_{578}	2_2	2_4	2_{13}	2_{134}
3_3	3_1	3_{24}	3_7	3_{568}	3_3	3_1	3_{24}	3_{124}
4_4	4_{12}	4_{13}	4_8	4_{67}	4_4	4_{12}	4_{13}	4_{23}

The next step, illustrated in the following table, is slightly more complicated.

$(P \bowtie Q) \bowtie R$				S		$PQRS$			
1_1	1_{34}	1_{24}	1_{23}	1_5	1_{78}		1_3	1_2	1_{23}
2_2	2_4	2_{13}	2_{134}	2_6	2_8	2_2			2_3
3_3	3_1	3_{24}	3_{124}	3_7	3_5	3_3			3_2
4_4	4_{12}	4_{13}	4_{23}	4_8	4_{56}		4_2	4_3	4_{23}

According to the definition of $PQRS$ the second and third column of PQR have to be matched with the two columns in S. For instance, consider the value 1 in what might be called the "match column". In S it is paired with the values 3 and 4 by way of the index 5 in what might be called the "constraint column". With this information let us look into the corresponding pairing in $PQRS$. Via index 4, 1 is paired with 1 there which does not match with any of those two pairs in S. So the index 4 in 1_{24} will be removed. Similarly with index 3 in 2_{13}. Performing this match with each of the values results in the vector $(1_2, 2_\emptyset, 3_\emptyset, 4_3)$ for the constraint column of $PQRS$. As we see, it does no more contain the indices 1 and 2 so that these two can now be discarded from the whole matrix which leads to the result shown above. In order to complete the example, joining in T removes the index 3 in 1_{23} and the index 2 in 4_{23}, while joining in W then does the same with the remaining index in both these entries. Hence the final matrix just consists of the two possible solutions left over.

This exercise is meant to illustrate the sort of possibilities for improvement in calculating the final expression. It is not meant to give a full account of this technique which would be beyond the scope of this note. So we just add a few general remarks.

The technique is polynomial in time and linear in space, as can be easily seen. In fact, it is pretty efficient. It is not covering the general case, though. One problem that can immediately arise is illustrated by the following example of two simple relations to be joined on the b, c–columns.

$$\begin{array}{cc} a_1 & b_1 \\ & c_1 \end{array} \qquad \begin{array}{cc} b_1 & d_{12} \\ c_2 & e_1 \end{array}$$

The matching operation described above runs into trouble because the indices in the second column are not independent in contrast to those in the third one that is to be matched. The easiest way out of this problem is a recoding of the first relation by making the second column the primary one.

$$\begin{array}{cc} a_{12} & b_1 \\ & c_2 \end{array} \qquad \begin{array}{cc} b_1 & d_{12} \\ c_2 & e_1 \end{array}$$

Now there is no problem to proceed as described above. General wisdom from the database field teaches us that even with this possibility of recoding (which again can be done in polynomial time) we are covering the case of the "loss–less join" but still not the general one. But since the problem is NP–hard there may not be much further hope anyway.

In our whole presentation we have focused on the case of binary relations. This was only to keep things simple for the reader. The CSP–procedure from the previous section generalizes to the case of arbitrary predicates. With the restrictions just mentioned the same is true for the technique outlined in the present section.

The potential for parallel treatment is limited as was shown in [9]. In any case, each of the subsets G_j may be treated independently, as we mentioned above, hence they may be dealt with in parallel. For each of them the sequence of join operations may be executed in any order. Hence, by using the bi–section technique parallelism may be exploited so that maximally $\log k$ join operations will have to be performed in sequence (instead of the k mentioned above). The combination of these two possibilities of exploiting parallelism should additionally lead to a significant improvement in practice.

Acknowledgements. I am particularly indebted to A. Mackworth, who taught me many details of CSP and helped me overcome wrong perceptions. He, P. Gilmore, H.J. Ohlbach, R. Reiter, J. Schumann, and the referees provided many helpful and greatly appreciated comments that gradually improved previous versions of this paper.

References

[1] S. Bayerl, E. Eder, F. Kurfess, R. Letz, and J. Schumann. An implementation of a PROLOG–like theorem prover based on the connection method. In Ph. Jorrand and V. Sgurev, editors, *AIMSA '86, Artificial Intelligence — Methodology, Systems, Applications*, pages 29–36, North–Holland, Amsterdam, 1987.

[2] W. Bibel. Advanced topics in automated deduction. In R. Nossum, editor, *Fundamentals of Artificial Intelligence II*, Springer, Berlin, 1988.

[3] W. Bibel. *Automated Theorem Proving*. Vieweg Verlag, Braunschweig, second edition, 1987.

[4] W. Bibel. "Intellektik" statt "KI" — Ein ernstgemeinter Vorschlag. *Rundbrief der Fachgruppe Künstliche Intelligenz in der Gesellschaft für Informatik*, 22:15–16, December 1980.

[5] W. Bibel. Matings in matrices. *Comm. ACM*, 26:844–852, 1983.

[6] W. Bibel, R. Letz, and J. Schumann. Bottom–up enhancements of deductive systems. In I. Plander, editor, *Proceedings of 4th International Conference on Artificial Intelligence and Information–Control Systems of Robots*, North–Holland, Smolenice, CSSR, October 1987.

[7] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J. A. Campbell, editor, *Implementations of PROLOG*, pages 194–215, Horwood, Chichester, England, 1984.

[8] P. Van Hentenryck. *Consistency Techniques in Logic Programming*. PhD thesis, University Namur, Belgium, July 1987.

[9] S. Kasif. On the parallel complexity of some constraint satisfaction problems. In *AAAI–86*, pages 349–353, Kaufmann, Palo Alto CA, 1986.

[10] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[11] A. K. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211, J. Wiley and Sons, New York NY, 1987.

[12] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence Journal*, 25:65–74, 1985.

[13] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto CA, 1980.

[14] W. Rosier and M. Bruynooghe. *Empirical Study of Some Constraint Satisfaction Algorithms.* Report CW 50, Katholieke Universiteit Leuven, 1986.

[15] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville MD, 1982.

[16] P. H. Winston. *Artificial Intelligence.* Addison Wesley, Reading MA, 1984.

December 4, 1987