An algorithm for solving constraint-satisfaction problems

Wanlin Pang
Scott D. Goodwin
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada S4S 0A2

Abstract

We introduce a new method, called constraint-directed-generate-and-test (CDGT), for solving constraint satisfaction problems (CSPs). Instead of appending a new instantiation of *one* variable to a partial solution, as in traditional search methods, CDGT joins two sets of partial solutions and then checks against other relevant constraints to obtain a set of higher order partial solutions. CDGT is an efficient algorithm for finding all solutions of a CSP and it can be implemented for parallel execution.

1 Introduction

Constraint satisfaction problems (CSPs) involve finding values for variables subject to constraints which permit combinations of values. Since many problems in AI and other areas of computer science can be formulated as CSPs, it has been a research subject for a long time and researchers have approached the subject in three directions: searching for the CSP's solutions from the possible solution space ([6, 4, 2]), reducing a CSP to a simpler and equivalent CSP ([8, 9, 11, 5, 1]), and synthesizing solutions from partial solutions ([3, 12, 13]).

Many specialized techniques have been developed to improve the efficiency of searching and reducing. the search. The synthesis method, which is potentially suitable for finding all solutions and for parallel implementation, seemed until now to be less attractive than search and reduction methods. In this paper, we present the preliminary version of a new synthesis method, called constraint-directed-generate-and-test method (CDGT), for solving constraint satisfaction problems. In this method, a constraint on a subset of variables is considered to represent a partial solution with respect to that subset of variables. By joining two lower arity constraints on two subsets of variables, a possible constraint of higher arity is obtained. This constraint is tightened by testing it against relevant constraints. The tightened constraint represents partial solutions w.r.t. the subset of variables which is the union of the original two variable subsets.

In the following sections, we describe the CDGT algorithm, prove its soundness and completeness, analyze its complexity and compare it with other solution synthesis algorithms.

2 Constraint-directed-generate-and-test method

2.1 Constraint-satisfaction problems

Before describing the CDGT algorithm, we first provide some necessary definitions.

A constraint satisfaction problem is a structure $< V, U, C >$ where $V = \{X_1, X_2, \ldots, X_n\}$ is a set of variables which may take on values from a set of domains $U = \{D_1, D_2, \ldots, D_n\}$, and $C = \{C_1, C_2, \ldots, C_m\}$ is a set of constraints. The ith constraint C_i of C is posed on a subset of variables $V_{C_i} = \{X_{i_1}, X_{i_2}, \ldots, X_{i_{c_i}}\}$ to limit the values they can take on. In other words, C_i represents a relation on $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{c_i}}$. The problem is to find all tuples from $D_1 \times D_2 \times \ldots \times D_n$, i.e., all possible instantiations of the variables, such that each instantiation satisfies all constraints.

We say that two constraints $C_i, C_j \in C$ are **connected** if $V_{C_i} \cap V_{C_j} \neq \emptyset$. A constraint is **isolated** in C if it is not connected to any other constraints in C. Given a constraint C_i on V_{C_i} and a subset $V_k = \{X_{ik_1}, \ldots, X_{ik_l}\} \subset V_{C_i}$, there is a constraint C_{ik} on V_k induced by C_i such that the relation on

$D_{ik_i} \times \ldots \times D_{ik_l}$ represented by C_{ik} is the **projection** of the relation represented by C_i. We call C_{ik} an **induced** constraint of C_i on V_k. Two constraints $C_i, C_j \in C$ are **consistent** if either C_i and C_j are not connected, or the induced constraints of C_i and C_j on $V_{C_i} \cap V_{C_j}$ can be satisfied simultaneously by at least one instantiation of the variables in $V_{C_i} \cap V_{C_j}$. (Clearly, if there exist $C_i, C_j \in C$ such that C_i and C_j are not consistent, then there is no solution to the given problem.) A constraint C_i is **tighter** than a constraint C_j if they are posed on the same subset of variables and the instantiation that satisfies C_i also satisfies C_j. A constraint $C_i \in C$ is **redundant** if there exists $C_j \in C$ such that $V_{C_i} \subseteq V_{C_j}$ and the induced constraint C_j on V_{C_i} is tighter than C_i. The **arity** of constraint C_i is $|V_{C_i}|$.

2.2 Basic idea of CDGT

Suppose that each constraint $C_i \in C$ on subset of the variables $V_{C_i} = \{X_{i_1}, X_{i_2}, \ldots, X_{i_{c_i}}\}$ is given as a relation, i.e., a subset of $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{ci}}$ (if not, we can always generate such a relation from C_i). A tuple t_{c_i} in C_i can be considered a consistent instantiation of variables V_{C_i}, i.e., t_{c_i} is a partial solution w.r.t. V_{C_i}. Similarly, suppose the tuple t_{c_j} is a partial solution w.r.t. V_{C_j}. Joining these two tuples we have a tuple $t_{c_{ij}}$ which is an instantiation of variables in $V_{C_i} \cup V_{C_j}$. Tuple $t_{c_{ij}}$ must be tested to see if it satisfies those constraints $C_h \in C$ such that $V_{C_h} \subset V_{C_i} \cup V_{C_j}$, i.e., if the projection of $t_{c_{ij}}$ on V_{C_h} is a member of C_h. If $t_{c_{ij}}$ satisfies all those C_h then it is a consistent instantiation, so it is a partial solution w.r.t. $V_{C_i} \cup V_{C_j}$. Repeating this *join-and-test* process for all tuples in C_i and C_j gives a higher arity constraint C_{ij} on $V_{C_i} \cup V_{C_j}$ which contains all partial solutions w.r.t. to $V_{C_i} \cup V_{C_j}$.

The basic idea of the algorithm is to incrementally (also consistently) synthesize higher arity constraints from selected lower arity constraints until an *n_ary* constraint is obtained (or we discover none exists).

2.3 CDGT algorithms

If $< V, U, C >$ is a CSP with no redundant constraints and no isolated constraints in C, the algorithm *CDGT* is described as follows:

CDGT($< V, U, C >$, *Sol*):

```
BEGIN
      WHILE |C| > 1 DO
      BEGIN
            select C_i and C_j from C such that V_{C_i} ∩ V_{C_j} = V_K ≠ ∅;
            compute C' = {C_h ∈ C | V_{C_h} ⊂ V_{C_i} ∪ V_{C_j}};
            join_test(C_i, C_j, V_K, C', C_ij);
            delete C_i, C_j from C;
            subtract C' from C;
            add C_ij to C;
      END_WHILE;
      Sol ← C_1 ∈ C;
END
```

Let $c_{ij} = |V_{C_i} \cup V_{C_j}|$. The procedure $join\_test(C_i, C_j, V_K, C', C_{ij})$ generates a c_{ij}_ary constraint on $V_{C_i} \cup V_{C_j}$ by enumerating those c_{ij}_ary tuples (from all c_i_ary tuples in C_i and c_j_ary tuples in C_j) that satisfy all constraints in C'. In terms of relational algebra, C_{ij} is the result relation of performing a *JOIN* operation on relations C_i and C_j with the condition that $C_h \in C$ must be satisfied.

The join_test algorithm is described as follows:

$join\_test(C_i, C_j, V_K, C', C_{ij})$:

```
BEGIN
      FOR each tuple t_{c_i} ∈ C_i
      BEGIN
            FOR each t_{c_j} ∈ C_j
                  IF proj(t_{c_i}, V_K) = proj(t_{c_j}, V_K)
```

```
6.                BEGIN
7.                      join(t_ci, t_cj, t_cij);
8.                      IF test(t_cij, C') THEN add t_cij to C_ij;
9.                END_IF;
10.       END_FOR
11. END
```

The function $proj(t_{c_i}, V_K)$ returns an $|V_K|$*_ary* tuple which is the projection of the $|V_{C_i}|$*_ary* tuple t_{c_i} on the variable subset V_K. For example, let $t_{c_i} = (3, 6, 8, 2, 4)$ be a 5*_ary* tuple from $D_1 \times \ldots \times D_5$, its projection on X_2, X_3, X_4 is an 3*_ary* tuple $(6, 8, 2)$ in $D_2 \times D_3 \times D_4$.

Let t_{c_i} and t_{c_j} be assignments of values to variables in V_{C_i} and V_{C_j} respectively. If $V_{C_i} \cap V_{C_j} = V_K \neq \emptyset$ and $proj(t_{c_i}, V_K) = proj(t_{c_j}, V_K)$, then the procedure $join(t_{c_i}, t_{c_j}, t_{c_{ij}})$ returns an c_{ij}*_ary* tuple $t_{c_{ij}}$ such that $proj(t_{c_{ij}}, V_{C_i}) = t_{c_i}$ and $proj(t_{c_{ij}}, V_{C_j}) = t_{c_j}$. For example, if $t_{c_i} = (3, 6, 8, 2, 4)$ is a 5*_ary* tuple from $D_1 \times \ldots \times D_5$ and $t_{c_j} = (2, 4, 9, 5)$ is a 4*_ary* tuple from $D_4 \times D_5 \times D_6 \times D_7$ then the procedure $join(t_{c_i}, t_{c_j}, t_{c_{ij}})$ will return a 7*_ary* tuple $t_{c_{ij}} = (3, 6, 8, 2, 4, 9, 5)$ which is in $D_1 \times \ldots \times D_7$.

The function $test(t_{c_{ij}}, C')$ returns *true* if the tuple $t_{c_{ij}}$ satisfies all the constraints in C'. It is defined as follows:

$test(t_{c_{ij}}, C')$

```
1. BEGIN
2.       FOR each C_h in C'
3.            IF proj(t_cij, V_Ch) ∉ C_h THEN RETURN false;
4.       RETURN true;
5. END
```

So far we have been assuming constraints such as C_i posed on a subset of variables $V_{C_i} = \{X_{i_1}, X_{i_2}, \ldots, X_{i_{ci}}\}$ are given in the form of a relation on $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{c_i}}$. To handle constraints expressed in other forms, a procedure is needed to generate the relational form from the original form. Suppose we are given a CSP problem $< V, U, C^* >$ where C^* is the set of constraints expressed in a non-relational form. A procedure $generate(C_i^*, C_i)$ is needed to generate a relational constraint C_i on $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{c_i}}$ from the given non-relational constraint C_i^* on $V_{C_i} = \{X_{i_1}, X_{i_2}, \ldots, X_{i_{ci}}\}$. We can modify the CDGT algorithm to deal with non-relational constraints by preprocessing the constraints C^* using the procedure *generate* to produce the set of relational constraints C. Or we can only generate relations from those constraints when they are selected (the fourth line in CDGT algorithm) to be joined. In the latter case, the third line in function $test(tc_{ij}, C')$ should also be changed to "IF $proj(t_{c_{ij}}, V_{C_h})$ is not consistent with C_h THEN RETURN *false*".

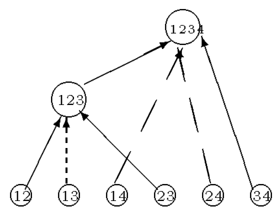
Note: If there are redundant constraints in the given set of constraints, we can remove them by a simple pre-processing procedure. If there are isolated constraints, they can be processed separately and then the separate solutions can be joined to produce global solutions by an enumerating method.

2.4 Example

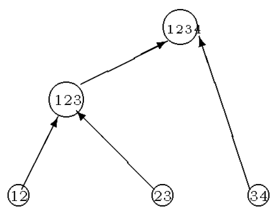
We consider the 4-queens problem where we need to place 4 queens in 4 by 4 chess board such that they do not attack each other. This can be formulated as a binary CSP with four variables $V = \{X_1, X_2, X_3, X_4\}$. Each variable corresponds to a row and its value represents which column to place a queen. The domain of each variable is $\{1, 2, 3, 4\}$. The constraints specified in the problem description exist between every pair of variables.

Suppose we have the set of constraints $C = \{C_{12}, C_{13}, C_{14}, C_{23}, C_{24}, C_{34}\}$, where
$C_{12} = C_{23} = C_{34} = \{(13), (14), (24), (31), (41), (42)\}$,
$C_{13} = C_{24} = \{(12), (14), (21), (23), (32), (34), (42), (43)\}$,
$C_{14} = \{(12), (13), (21), (23), (24), (31), (32), (34), (42), (43)\}$.
We select C_{12} and C_{23} which are connected, and we have $C' = \{C_{13}\}$.

Performing $join\_test(C_{12}, C_{23}, V_{C_{12}} \cap V_{C_{23}}, C', C_{123})$, i.e., enumerating combinations of two tuples from C_{12} and C_{23} respectively, and testing each combination to see if it satisfies C_{13} which is the only constraint in C', we get
$C_{123} = \{(142), (241), (314), (413)\}$.
After adding C_{123} to C and deleting C_{12}, C_{23}, C_{13} from C, we have



(a) A SCN for the 4-queens problem



(b) A CDGT Tree the for 4-queens problem

Figure 1: A CDGT SCN and Tree for the 4-queens problem

$C = \{C_{123}, C_{14}, C_{24}, C_{34}\}$.

In the second loop, we select C_{123} and C_{34} which are connected, and we have $C' = \{C_{14}, C_{24}\}$. Performing $join\_test(C_{123}, C_{34}, V_{C_{123}} \cap V_{C_{34}}, C', C_{1234})$, we get

$C_{1234} = \{(2413), (3142)\}$.

Then we add C_{1234} to C and delete $C_{123}, C_{14}, C_{24}, C_{34}$ from C. There is only one constraint left in C; it contains all the solutions.

Notice that in the main loop the selection of C_i and C_j, which are used to synthesize C_{ij}, is arbitrary. It may be possible to develop useful selection strategies for certain problems.

3 Soundness and completeness of CDGT

An algorithm is sound if every result returned by the algorithm is a solution. An algorithm is complete if every solution can be found by the algorithm. In this section, we show that CDGT is both sound and complete.

To prove the soundness of CDGT is to prove that every tuple in the final *n_ary* constraint satisfies all the given constraints $C = \{C_1, C_2, \ldots, C_m\}$. To prove this we need only to prove that at any stage every tuple in the joined $|c_{ij}|$*_ary* constraint satisfies all the constraints on the variable subset $V_h \subset V_{C_i} \cup V_{C_j}$. This is guaranteed by performing *test* at line 8 in procedure *join_test*.

To prove the completeness of CDGT is to prove that every solution is included as a tuple in the final *n_ary* constraint. To prove this we need only to prove that at any stage every partial solution w.r.t. variables in $V_{C_i} \cup V_{C_j}$ is included as a tuple in the synthesized $|c_{ij}|$*_ary* constraint. This is guaranteed by performing *join*, which enumerates all possible combinations of tuples in C_i and in C_j at line 7 in procedure *join_test*.

4 Complexity of CDGT

For analyzing the complexity of CDGT, we define a *Synthesized Constraint Network (SCN)* to be a directed graph which is constructed as follows:

1. The nodes of the graph are the given and the synthesized constraints.
2. There is a directed line from node C_k to C_j iff C_k is one of the constraints from which C_j is synthesized.
3. There is a directed dashed line from node C_h to C_j iff C_h is one of the constraints to be tested when C_j is synthesized.

We define a *CDGT Tree* to be a directed graph obtained from a *SCN* by eliminating all the dashed lines and those nodes connected only with dashed lines.

The *CDGT SCN* and the *CDGT Tree* corresponding to the example in previous section are shown in Figure 1.

We first consider the complexity of CDGT when it is applied to solving binary constraint problems.

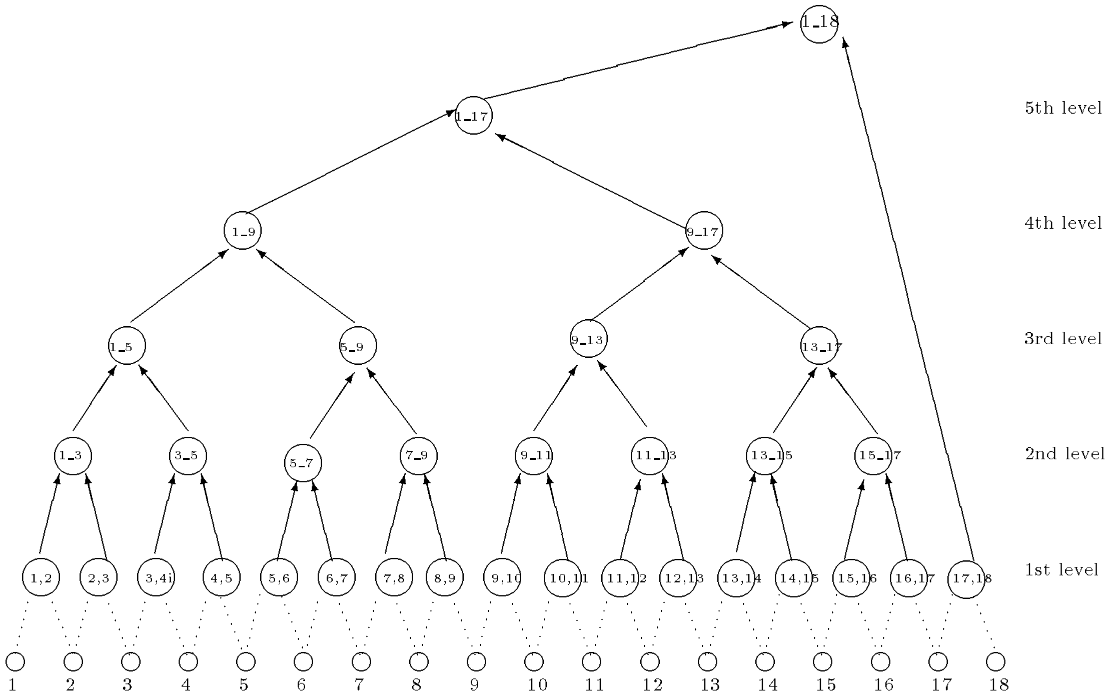


Figure 2: CDGT tree with 18 variables

Suppose that n is the number of variables in the given CSP, a is the maximum size of the domains for the variables, and every two variables are constrained (the worst case).

Using a simple strategy to select C_i, C_j (at line 4 in the algorithm) to be joined, we can make CDGT perform the synthesis process from lower level nodes to higher level nodes as shown in the *CDGT Tree* (Figure 2).

The tree has $\lceil \log_2 n + 1 \rceil$ levels. At ith level there are $\lceil \frac{n}{2^{i-1}} \rceil$ nodes and each node at ith level corresponds to a $(2^i - 1)$_*ary* constraint.

When synthesizing a k_*ary* constraint from two $\lceil \frac{k}{2} \rceil$_*ary* constraints, CDGT has to consider each combination of these two $\lceil \frac{k}{2} \rceil$_*ary* constraints. The number of tuples contained in a $\lceil \frac{k}{2} \rceil$_*ary* constraint is $O(a^{\lceil \frac{k}{2} \rceil})$ in the worst case. So the time complexity of synthesizing a k_*ary* constraints is $O(a^{k+1})$. When synthesizing a k_*ary* constraint from two constraints with arity k_i and k_j respectively, where $k_i + k_j = k + 1$, in the worst case, the time complexity is also $O(a^{k+1})$. So the time complexity of CDGT is $\sum_{k=2}^{\lceil \log_2 n+1 \rceil} \lceil \frac{n}{2^{k-1}} \rceil a^{2^k}$, which is $O(a^n)$.

For general CSPs, the CDGT Tree constructed has less than $\lceil \log_2 n + 1 \rceil$ levels, so CDGT should perform much better in practice.

If the constraints are given in a form other than relations on the domains of variables, CDGT must generate relations for the selected constraints (which will appear in the *CDGT Tree*) to be used to synthesize higher arity constraints. The generation can be done by using any generate-and-test or backtrack search method on this subset of variables.

5 Related work and comparison

In the first section, we mentioned that there are mainly three kinds of techniques for solving CSPs: *search, reduction* and *synthesis*. For a survey of these different methods, readers are referred to [7, 10]. In this section, we briefly describe the most popular synthesis algorithms, i.e., Freuder's synthesis algorithm FA [3] and Essex basic algorithm AB [13]. Then we compare their time complexity.

FA is designed to achieve any level of consistency for a given n variable CSP. The nth level of consistency ultimately obtained by FA contains all the solution of the given problem. The basic idea of

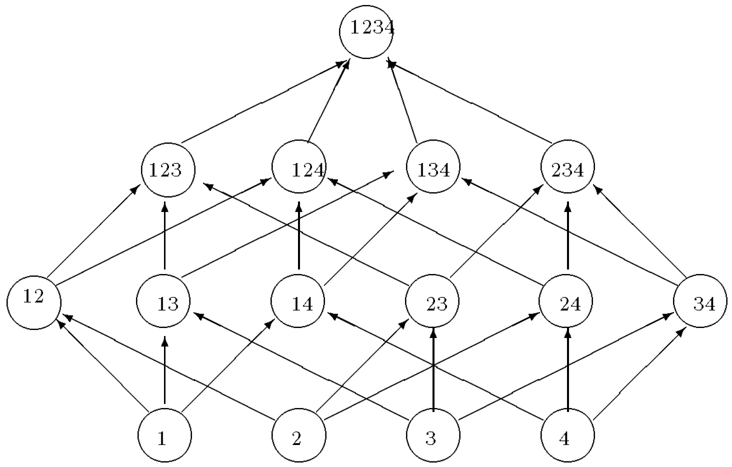


Figure 3: $FACN$ for the 4-queens problem

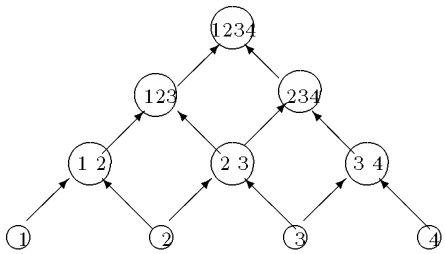


Figure 4: $ABCN$ for the 4-queens problem

FA is to incrementally construct a constraint network where each node built at step k corresponds to one of the constraints with arity k on a subset of variables $V_{IK} = \{X_{i_1}, X_{i_2}, \ldots, X_{i_k}\}$. The network at step k contains a partial solution w.r.t. the subset V_{IK}. For example, the constraint network constructed by FA for solving the 4-queens problem is shown in Figure 3.

The time complexity of FA is shown in [13] to be $O(2^n + na^{2n})$.

Similarly, the Essex basic algorithm AB constructs a constraint network incrementally, but at step k, only adjacent (according to a predefined variable partial ordering) nodes built at step $k-1$ are used to construct new nodes, so that only some of the constraints with arity k are synthesized. The constraint network constructed by AB for solving the 4-queens problem is shown in Figure 4.

The time complexity of AB is shown in [13] to be $O(a^{2n-2})$.

Compared to FA, where all the constraints with arity k for $k = 1, 2, \ldots, n$ are considered and the number of nodes in SCN constructed is $2^n - 1$, and AB, where only some of the constraints with arity k for $k = 1, 2, \ldots, n$ are considered and the number of nodes in SCN constructed is $\frac{n(n+1)}{2}$, CDGT synthesizes only necessary constraints with arity k for a few $k \in \{1, 2, \ldots, n\}$. In general, the number of nodes in SCN constructed by CDGT is $\leq 2n$. By using a different strategy to select the lower arity constraints to be joined to synthesize the higher arity constraints, different constraint networks are constructed by CDGT. One such CN constructed by CDGT for solving the 4-queens problem is shown in Figure 1(b), which is much simpler than those constructed by both FA and AB.

6 Future work and conclusion

We presented a new synthesis algorithm for CSPs which is more efficient than similar ones such as Freuder's synthesis algorithm and Essex basic algorithm AB.

The motivation for developing a new synthesis algorithm is that we intend to build an intelligent scheduling system based on CSP or CSOP for a parallel architecture machine. CDGT has inherent parallelism (e.g., simultaneously selecting different constraint-pairs C_i and C_j) so that synthesis can be processed in parallel.

CDGT also provides flexibility in that different heuristics (such as ordering heuristics) can be used to guide the synthesis process. Finding the right heuristic for a problem, or even embeding a learning mechanism to learn heuristics to improve efficiency seems to be interesting and challenging problems for future study.

References

[1] Y. Chen. Improving Han and Lee's path consistency algorithm. In *Proceedings of the 3rd IEEE International Conference on Tools for AI*, pages 346–350, San Jose, CA., Nov 1991. IEEE.

[2] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[3] E. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[4] E. Freuder. Backtrack-free and backtrack-bounded search. In L. Kanal and V. Kumar, editors, *Search in Artificail Intelligence*, pages 343–369. Springer-Verlag, New York, 1988.

[5] C. Han and C. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[6] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

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

[8] A. Mackworth. Consistency in networks of realtions. *Artificial Intelligence*, 8(1):99–118, 1977.

[9] U. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Science*, 19:229–250, 1979.

[10] P. Mesaguer. Constraint satisfaction problems: An overview. *AI Communications*, 2(1):3–17, 1989.

[11] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[12] R. Seidel. A new method for solving constraint satisfaction problems. In *Proceedings of the 7th IJCAI*, pages 338–342, 1981.

[13] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, San Diego, CA, 1993.