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 \(i\)th constraint \(C_i\) of \(C\) is posed on a subset of variables \(V_{C_i} = \{X_{i_1}, X_{i_2}, \ldots, X_{i_{\ell_i}}\}\) to limit the values they can take on. In other words, \(C_i\) represents a relation on \(D_{i_1} \times D_{i_2} \times \ldots \times D_{i_{\ell_i}}\). 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 \(V_i = \{X_{i_1}, \ldots, X_{i_{\ell_i}}\} \subset V_{C_i}\), there is a constraint \(C_{i_k}\) on \(V_i\) induced by \(C_i\) such that the relation on
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 \(< V, У, С >\) is a CSP with no redundant constraints and no isolated constraints in C, the algorithm CDGT is described as follows:

\[
\text{CDGT}(\langle V, У, С >, \text{Sol}) : \\
1. \text{BEGIN} \\
2. \text{WHILE } |С| > 1 \text{ DO} \\
3. \text{BEGIN} \\
4. \text{select } C_i \text{ and } C_j \text{ from } С \text{ such that } V_{C_i} \cap V_{C_j} = V_{X} \neq \emptyset ; \\
5. \text{compute } C' = \{ \text{c}_k \in С |V_{C_k} \subset V_{C_i} \cup V_{C_j}\}; \\
6. \text{join-test}(C_i, C_j, V_{X'}, C', C_{ij}); \\
7. \text{delete } C_i, C_j \text{ from } С ; \\
8. \text{subtract } C' \text{ from } С ; \\
9. \text{add } C_{ij} \text{ to } С ; \\
10. \text{END WHILE}; \\
11. \text{Sol} \leftarrow C_i \in С ; \\
12. \text{END} \\
\]

Let \(c_{ij} = [V_{C_i} \cup V_{C_j}].\) The procedure \text{join-test}(C_i, C_j, V_{X'}, C', C_{ij}) \) generates a c_{ij,arity} constraint on \(V_{C_i} \cup V_{C_j}\) by enumerating those c_{ij,arity} tuples (from all c_{ij,arity} tuples in \(C_i\) and c_{ij,arity} tuples in \(C_j\)) that satisfy all constraints in \(C'.\) In terms of relational algebra, \(C_{ij}\) is the result relation of performing a \text{JOIN} operation on relations \(C_i\) and \(C_j\) with the condition that \(c_{ij} \in С\) must be satisfied.

The \text{join-test} algorithm is described as follows:

\[
\text{join-test}(C_i, C_j, V_{X'}, C', C_{ij}) : \\
1. \text{BEGIN} \\
2. \text{FOR each tuple } t_i \in C_i \\
3. \text{BEGIN} \\
4. \text{FOR each } t_j \in C_j \\
5. \text{IF } \text{proj}(t_i, V_{X}) = \text{proj}(t_j, V_{X'}) \\
\]
6. BEGIN  
7. \( \text{join}(t_{c_1}, t_{c_2}, t_{c_3}) \) 
8. IF \( \text{test}(t_{c_1}, C') \) THEN add \( t_{c_2} \) to \( C_{ij} \);  
9. END_IF;  
10. END_FOR  
11. END

The function \( \text{proj}(t_{c_1}, V_X) \) returns an \( |V_X| \)-ary tuple which is the projection of the \( |V_C| \)-ary tuple \( t_{c_1} \) on the variable subset \( V_X \). For example, let \( t_{c_1} = (3, 6, 8, 2, 4) \) 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 \( t_{c_1} \) and \( t_{c_2} \) be assignments of values to variables in \( V_{C_1} \) and \( V_{C_2} \) respectively. If \( V_{C_1} \cap V_{C_2} = V_X \neq \emptyset \) and \( \text{proj}(t_{c_1}, V_X) = \text{proj}(t_{c_2}, V_X) \), then the procedure \( \text{join}(t_{c_1}, t_{c_2}, t_{c_3}) \) returns an \( |C_{ij}| \)-ary tuple \( t_{c_3} \) such that \( \text{proj}(t_{c_3}, V_{C_1}) = t_{c_1} \) and \( \text{proj}(t_{c_3}, V_{C_2}) = t_{c_2} \). For example, if \( t_{c_1} = (3, 6, 8, 2, 4) \) is a 5-ary tuple from \( D_1 \times \ldots \times D_5 \) and \( t_{c_2} = (2, 4, 9, 5) \) is a 4-ary tuple from \( D_2 \times D_4 \times D_5 \times D_7 \) then the procedure \( \text{join}(t_{c_1}, t_{c_2}, t_{c_3}) \) will return a 7-ary tuple \( t_{c_3} = (3, 6, 8, 2, 4, 9, 5) \) which is in \( D_1 \times \ldots \times D_7 \).

The function \( \text{test}(t_{c_2}, C') \) returns \( \text{true} \) if the tuple \( t_{c_2} \) satisfies all the constraints in \( C' \). It is defined as follows:

\[
\text{test}(t_{c_2}, C')
\]

1. BEGIN  
2. FOR each \( C_h \) in \( C' \)  
3. IF \( \text{proj}(t_{c_2}, V_{C_h}) \neq C_h \) THEN RETURN \( \text{false} \);  
4. RETURN \( \text{true} \);  
5. END

So far we have been assuming constraints such as \( C_i \) posed on a subset of variables \( V_{C_i} = \{ X_{i_1}, X_{i_2}, \ldots, X_{i_n} \} \) are given in the form of a relation on \( D_{i_1} \times D_{i_2} \times \ldots \times D_{i_n} \). 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 \( \text{generate}(C^*, C_i) \) is needed to generate a relational constraint \( C_i \) on \( D_{i_1} \times D_{i_2} \times \ldots \times D_{i_n} \) from the given non-relational constraint \( C^*_i \) on \( V_{C_i} = \{ X_{i_1}, X_{i_2}, \ldots, X_{i_n} \} \). We can modify the CDGT algorithm to deal with non-relational constraints by preprocessing the constraints \( C^* \) using the procedure \( \text{generate} \) 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 \( \text{test}(t_{c_2}, C') \) should also be changed to “\( \text{IF \ pro j}(t_{c_2}, V_{C_h}) \) is not consistent with \( C_h \) THEN RETURN \( \text{false} \)”.

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 a 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

\[
C_{12} = C_{23} = C_{34} = \{ (13), (14), (24), (31), (41), (42) \}
\]

\[
C_{13} = C_{24} = \{ (12), (14), (21), (23), (32), (34), (42), (43) \}
\]

\[
C_{14} = \{ (12), (33), (21), (23), (24), (31), (32), (34), (42), (43) \}
\]

We select \( C_{12} \) and \( C_{23} \) which are connected, and we have \( C' = \{ C_{12} \} \).

Performing \( \text{join}_{\text{test}}(C_{12}, C_{23}, V_{C_{12}} \cap V_{C_{23}}, C', C_{12}) \), 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


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 \(\text{join} test(C_{123}, C_{34}, V_{C_{123}} \cap V_{C_{34}}, C', C_{1234})\), we get
\[C_{1234} = \{(2413), (3412)\}\.\]

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 \(n\text{-ary}\) 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 \(\{C_i\}\) \(n\text{-ary}\) constraint satisfies all the constraints on the variable subset \(V_h \subseteq V_{C_i} \cup V_{C_j}\). This is guaranteed by performing \text{test} at line 8 in procedure \text{join} test.

To prove the completeness of CDGT is to prove that every solution is included as a tuple in the final \(n\text{-ary}\) 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 \(\{C_i\}\) \(n\text{-ary}\) constraint. This is guaranteed by performing \text{join}, which enumerates all possible combinations of tuples in \(C_i\) and in \(C_j\) at line 7 in procedure \text{join} test.

4 Complexity of CDGT

For analyzing the complexity of CDGT, we define a \textit{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 \textit{CDGT Tree} to be a directed graph obtained from a \textit{SCN} by eliminating all the dashed lines and those nodes connected only with dashed lines.

The \textit{CDGT SCN} and the \textit{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.
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 \( i \)th level there are \( \left\lceil \frac{n}{2^i} \right\rceil \) nodes and each node at \( i \)th level corresponds to a \( (2^i - 1) \)-ary constraint.

When synthesizing a \( k \)-ary constraint from two \( \left\lceil \frac{n}{2^i} \right\rceil \)-ary constraints, CDGT has to consider each combination of these two \( \left\lceil \frac{n}{2^i} \right\rceil \)-ary constraints. The number of tuples contained in a \( \left\lceil \frac{n}{2^i} \right\rceil \)-ary constraint is \( \mathcal{O}(a^{\left\lfloor \frac{k}{2^i} \right\rfloor}) \) in the worst case. So the time complexity of synthesizing a \( k \)-ary constraints is \( \mathcal{O}(a^{k+1}) \).

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 \( \mathcal{O}(a^{k+1}) \). So the time complexity of CDGT is \( \sum_{i=0}^{\log_2 n + 1} \left\lceil \frac{n}{2^i} \right\rceil a^{2^i} \), which is \( \mathcal{O}(a^n) \).

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 \( n \)th level of consistency ultimately obtained by FA contains all the solution of the given problem. The basic idea of
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 embedding a learning mechanism to learn heuristics to improve efficiency seems to be interesting and challenging problems for future study.

References


