INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

On the Relationship between Synthesizing and Tagging

Hans Werner Guesgen*

TR-94-022

May 1994

Abstract

During recent years, various constraint satisfaction algorithms have been developed. Among them are Freuder's synthesizing algorithm and our tagging method. We will compare the two approaches in this paper and work out commonalities and differences.

The purpose of this paper is to give a deeper insight into existing methods (rather than introducing new ones). Although the algorithms we chose for our investigation might not be the most valuable ones from the viewpoint of applications, they illustrate important and interesting principles of constraint satisfaction.

*On leave from the Computer Science Department at the University of Auckland, Private Bag 92019, Auckland, New Zealand, email: hans@cs.auckland.ac.nz. The author has been supported by the University of Auckland Research Fund under the grant numbers A18/XXXXX/62090/3414014 and A18/XXXXX/62090/F3414025.

1 Introduction

Unlike many other papers in the area of constraint satisfaction, this paper does not introduce any new method, technique, or algorithm. On the contrary, it takes two existing constraint satisfaction algorithms and discusses their common aspects and differences. The two algorithms are:

Synthesizing Algorithm
: In [2], Freuder introduced an approach to constraint satisfaction based on generating higher order constraints. The algorithm starts with a binary constraint network. It successively computes 3-ary, 4-ary, etc. constraints and adds them to the network until a n-ary constraint is computed, where n is the number of variables in the network. This n-ary constraint represents the solution of the problem represented by the constraint network.

Tagging Method
: In [7], we suggested a method which extends a standard filtering algorithm like the Waltz algorithm (i.e., an algorithm usually computing arc consistency) such that it can be used to compute globally consistent solutions. The idea is to associate tags with the values propagated in the network. The tags maintain the global relationships among the values.

We assume that Freuder's synthesizing algorithm is well-known in the constraint satisfaction community, and therefore, only a short review of this algorithm is given in this paper. The tagging method, on the other hand, might not be as well-known as Freuder's algorithm. Therefore, we will give a more detailed (but not exhaustive) description of the tagging method in the following.

The paper is organized as follows. We will start with a brief clarification of the terminology and notation used in this paper. Then we will sketch Freuder's synthesizing algorithm and illustrate it by a small coloring problem. We will continue with an introduction to the tagging method and show how the example is handled by this approach. We will finish the paper with a comparison of the two algorithms.

2 Terminology and Notation

Throughout this paper, we will view a constraint as consisting of a set of variables and a relation on these variables. Networks of constraints are obtained by sharing variables among constraints. A constraint satisfaction problem (CSP) can be defined as follows: Given a constraint network and an initial assignment of possible values to its variables, find one or more tuples of values that satisfy the constraint network, i.e., that are elements of the relation represented by the network.

We denote variables by x, y, z, etc. and constraints by $C(x, y)$, $C(x, y, z)$, etc. Where it does not cause confusion, we use the same notation for a constraint and its relation.

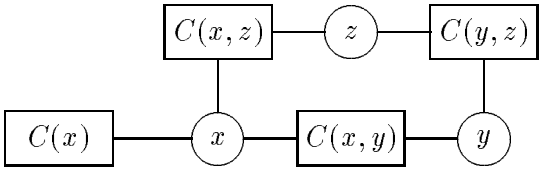

Figure 1: Graph of the constraint network N. The variables are represented by circles and the constraints by rectangles. An edge between a circle and a rectangle means that the corresponding variable belongs to the constraint represented by the rectangle.

For example, let us consider the following coloring problem. A fully-connected graph with three nodes x, y, and z is to be colored such that the following requirements are fulfilled:

1. The only colors available are *red*, *yellow*, and *green*.
2. x must be *red*.
3. Two adjacent nodes must not have the same color.

This problem can be represented by a constraint network N as shown in figure 1.

The constraint network consists of constraints $C_(x)$, $C(x, y)$, $C(x, z)$, and $C(y, z)$ on variables x, y, and z, each variable ranging over the domain $D = \{red, yellow, green\}$. The constraint relations are as follows:

1. The binary constraints, $C(x, y)$, $C(x, z)$, and $C(y, z)$, have identical relations equal to

$$\begin{array}{l}\{(red, yellow), (red, green),\\ \ (yellow, red), (yellow, green),\\ \ (green, red), \ \ (green, yellow)\}\end{array}$$

2. The relation of the unary constraint $C(x)$ is equal to $\{red\}$

The corresponding CSP has two solutions:

$$\begin{array}{l}(x, y, z) = (red, green, yellow)\\ (x, y, z) = (red, yellow, green)\end{array}$$

Since finding a solution for a given CSP is NP-hard in general, we can expect a complete constraint satisfaction algorithm to be exponential. This is also the case for the algorithms we are discussing in this paper: Freuder's synthesizing algorithm and our tagging method. The synthesizing algorithm, for example, suffers from exponential complexity because it successively adds constraints to the network. The following two sections provide an introduction to the synthesizing algorithm, respectively the tagging method.

3 The Synthesizing Algorithm

The idea of Freuder's algorithm is to add higher-order constraints to a network of binary constraints. These higher-order constraints are computed from constraints with a lesser arity. For example, if there are constraints $C(x, y)$ and $C(y, z)$ in the initial network, we can generate a constraint $C(x, y, z)$ whose relation is the join of the relations of $C(x, y)$ and $C(y, z)$. For the coloring problem represented by the constraint network N, the relation of the resulting constraint $C(x, y, z)$ contains the solutions of the corresponding CSP (and nothing else), i.e.:

$$C(x, y, z) = \{(red, green, yellow),\ (red, yellow, green)\}$$

In general, Freuder's synthesizing algorithm computes not only 3-ary constraints but also 4-ary, 5-ary, etc. constraints until it reaches an arity that is equal to the number of variables in the network. Since our example network contains only three variables, the algorithm terminates in this case after reaching the arity of three.

Before we discuss further details of the synthesizing algorithm, we will switch to another algorithm which uses the tagging method to compute the solutions of a given CSP. The next section is dedicated to this algorithm.

4 The Tagging Method

The tagging method is supposed to be embedded in an algorithm that is based on local propagation: A constraint of the given constraint network is evaluated and the result is propagated to its direct neighbors in the network. Examples of local propagation algorithms are the well-known AC-x algorithms [11, 12], (massively) parallel versions of which can be found in [1, 10, 13, 14].

The basic idea of tagging is to provide the domain values with tags and to apply the local propagation algorithm to the tagged values. The tags, which are tuples of indices in the serial and parallel case and Gödel numbers or bit vectors in the massively parallel one, maintain the information that is usually lost during local propagation, i.e., the global relationships among the values. We will focus on the tuple variant in the following; details of the Gödel number variant can be found in [5].

We distinguish between two types of tags: Those that are assigned to the values when single constraints are evaluated and those that are used in constraint networks. The tags that are assigned when a constraint is evaluated are called subtags, and we use integers to represent them. The tags used on the level of constraint networks for propagation purposes are called full tags. They are represented by tuples of subtags.

Suppose the network consists of n constraints. Then, every full tag is an n-tuple where the ith value in the tuple is the tag assigned by the ith constraint. For example, assuming the order $C(x, y) < C(y, z) < C(x, z) < C_(x)$ among the constraints of network N, $red_{(2,4,6,1)}$ means that $C(x, y)$, $C(y, z)$, $C(x, z)$, and $C_(x)$ assigned the tags 2, 4, 6, and 1, respectively.

The advantage of using tuples as full tags is obvious: Each subtag in the tuple can be uniquely mapped to a constraint of the network which facilitates their handling.

Before a constraint is evaluated, the full tags of the values of its variables are simplified. Suppose that the ith constraint of the network is to be evaluated, then the the full tag of each value is replaced with its ith subtag, as this is the only subtag of interest when the ith constraint is evaluated. For example, the value $red_{(2,4,6,1)}$ is simplified to red_4 when the second constraint, $C(y,z)$, is to be evaluated.

After the tags have been simplified, a standard algorithm for constraint evaluation is applied. Such an algorithm can be formulated as follows:[1] Compute the Cartesian product of the constraint variables and intersect this set with the constraint relation. In addition to such an evaluation algorithm, the tagging algorithm matches the subtags of each tuple of the Cartesian product by first computing the common subtag and then either replacing each subtag in the tuple by the common subtag, if the common subtag exist, or deleting the tuple from the Cartesian product, otherwise. The common subtag is a new tag if all values of the tuple are untagged. If some values are already provided with subtags, and if all of them are identical, then the common subtag is determined by this subtag; otherwise it is undefined. For example, the common subtag of red_2 and $green$ is 2, whereas the common subtag of red_2 and $green_3$ is undefined.

After the evaluation of a constraint, the projection procedure described above is executed in the opposite way. For that purpose, the subtags resulting from the evaluation process are merged with the original full tags. After that, full tags which are merged with the same subtag are unified. Unification in this context means that the common subtag is computed for each component of the full tags. If a common subtag exists for each component, the full tags are updated by the common subtags; otherwise, the corresponding values are deleted.

For example, let $green_{(2,4,-,-)}$ be a value for the variable y and $yellow_{(-,-,6,-)}$ be a value for the variable z. When the second constraint, $C(y,z)$, is evaluated, the tuple $(green_4, yellow\_)$ is matched with the constraint relation, resulting in the $(green_4, yellow_4)$. The subtags are then merged with the original ones:

$$\left.\begin{array}{l} green_4 \\ green_{(2,4,-,-)} \end{array}\right\} \longmapsto \quad green_{(2,4,-,-)}$$

$$\left.\begin{array}{l} yellow_4 \\ yellow_{(-,-,6,-)} \end{array}\right\} \longmapsto \quad yellow_{(-,4,6,-)}$$

Since the subtags with which the full tags have been merged are identical, the resulting full tags must be unified:

$$\begin{array}{lcl} green_{(2,4,-,-)} & \longmapsto & green_{(2,4,6,-)} \\ yellow_{(-,4,6,-)} & \longmapsto & yellow_{(2,4,6,-)} \end{array}$$

To summarize: Tagging during local propagation successively fills positions in initially empty full tags until all positions are filled and incompatible values are removed. In the case of the example network N, the algorithm has to fill four positions in each full tag.

[1] In [4], a more efficient way to evaluate constraints is discussed. However, we do not apply it here for reasons of simplicity and clarity.

Assuming that subtags are generated in increasing order and that the constraint $C(x)$ is evaluated first, followed by an evaluation of the constraints $C(x,y)$, $C(y,z)$, and $C(x,z)$, the result would be:

$$x \in \{red_{(2,4,6,1)}, red_{(3,5,7,1)}\}$$
$$y \in \{green_{(2,4,6,1)}, yellow_{(3,5,7,1)}\}$$
$$z \in \{yellow_{(2,4,6,1)}, green_{(3,5,7,1)}\}$$

Values with identical full tags represent a solution of N. There are two solutions for N, namely $(red, green, yellow)$ and $(red, yellow, green)$.

5 Comparison of the Algorithms

This section will look at some of the commonalities and differences of the synthesizing algorithm and the tagging method. The selection of issues discussed here is somewhat arbitrary and not exhaustive. It should be viewed as a basis for further research rather than a complete analysis. Let us start with the complexity of the algorithms:

Complexity

Both algorithms are exponential in the worst case (which is what we expect from a complete constraint satisfaction algorithm). The synthesizing algorithm constructs higher-order constraints until it gets to a constraint whose arity is identical with the number of variables in the network. In the worst case, this may take exponential time.

The tagging method avoids the construction of higher-order constraints, but has to cope with the problem of reproducing values. When a constraint is evaluated, values matching several relation elements must be duplicated, because for each match a different subtag is assigned. This means that the algorithm results in an exponential number of values in the worst case.

The previous observation leads us to an interesting duality between the two algorithms:

Value–Constraint Duality

Freuder's synthesizing algorithm maintains global information in the form of additional constraints. These constraints are of a higher order, since binary constraints are, in a certain sense, too weak to carry such information. The highest order needed is identical with the number of variables in the constraint network.

The tagging method maintains the global information in the form of full tags. Here the values rather than the constraint are extended, as they alone are not capable of carrying the global information. The dimension of the extension, i.e., the positions in each full tag, is identical with the number of variables in the constraint network.

Although the algorithms differ in how the solutions of a given CSP are computed, they don't differ in how many solutions they compute:

Exhaustive Search

Usually, a backtracking algorithm leaves it open how many solutions are to be computed. It may terminate after the first solution has been computed, or it may continue until all solutions have been determined. As opposed to backtracking, synthesizing and tagging always produce all solutions of a given CSP, even if one solution is sufficient. Although this is a disadvantage in general, there are many situations in which all solutions are requested, for example:

1. The CSP is expected to be inconsistent. In this case, it doesn't matter if the algorithm seeks one, a few, or all solutions of a given CSP, as there isn't any solution at all.
2. The CSP is an optimization problem. In this case, the optimal solution is to be computed for a given CSP. A straightforward (although often not efficient) approach to finding the best solution is first generating all solutions and then selecting the best one.

We will now turn to issues that are more in favor of the two algorithms than the issues above. After all, why should one be interested in synthesizing or tagging when at the same time a simple backtracking approach may be even more efficient. The answer lies in the following:

Parallel Implementation

Although it is possible to implement backtracking on a parallel computer, the implementation isn't as straightforward as the parallel implementation of synthesizing and tagging. The latter, for example, can be implemented in a straightforward way on a parallel computer by using the optimistic discrete relaxation scheme described in [8]. The idea of this scheme is to organize the full tags in a lattice structure. Parallel tagging can proceed as long as full tags are filled according to the lattice structure. Whenever the lattice structure is violated (which actually doesn't occur very often), certain computation steps must be repeated.

See [9] for details of how the synthesizing algorithms can be implemented in parallel.

Let us now look at the arity of the constraints in the network. So far, we have assumed that a binary constraint network be given, i.e., a network whose constraints have at most an arity of two. This assumption is not necessary:

Non-Binary Constraint Networks

Although originally designed for binary constraint networks, there is no reason why the synthesizing algorithm can't be applied to networks of constraints of any arity. The same holds for the tagging method. This method even works for hierarchical constraint networks, i.e., constraint networks whose components may be other (hierarchical) constraint networks [3].

Last but not least, a remark is in order regarding finite versus infinite relations:

Infinite Constraints

Both algorithms work with networks of finite constraints, i.e., constraints whose relations have a finite number of elements. The tagging method can also be used with infinite relations or, to be more precise, with a certain type of infinite relations, called pattern-characterizable relations. A pattern-characterizable relation is a possibly infinite relation whose elements can be specified by a finite number of patterns. We implemented pattern-characterizable relations in the constraint satisfaction system CONSAT [3], and we incorporated them into the tagging method.

6 Conclusion

Although the synthesizing algorithm and the tagging method were developed at different places, at different times, and independently of each other, they have many things in common. Even features like adding constraints versus adding values, that on first sight seem to be different, can be interpreted as commonality by focusing on the kind of information rather than the location where it is stored: To compute the solution of a constraint network with n variables, n-tuples are required, may they be n-ary constraints or n-ary full tags.

To make this point even stronger, neglect for a moment that there is a difference between variables and constraints. This can be achieved by using the concept of dynamic constraints as introduced in [6]. Dynamic constraints play the role of both variables and constraints. When formulating a given CSP as a dynamic CSP, the difference between synthesizing and tagging almost boils down to a syntactic difference only.

The paper certainly doesn't provide an exhaustive comparison of synthesizing and tagging. For example, we haven't compared synthesizing and tagging when applied to identical test data (like the test data used in [8] to evaluate the performance of tagging in parallel). This might give further insights into the two approaches and is certainly worth to be investigated in future research.

References

[1] P.R. Cooper and M.J. Swain. Parallelism and domain dependence in constraint satisfaction. Technical Report 255, University of Rochester, Computer Science Department, Rochester, New York, 1988.

[2] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, 1978.

[3] H.W. Guesgen. *CONSAT: A System for Constraint Satisfaction.* Research Notes in Artificial Intelligence. Morgan Kaufmann, San Mateo, California, 1989.

[4] H.W. Guesgen. A universal constraint programming language. In *Proc. IJCAI-89*, pages 60–65, Detroit, Michigan, 1989.

[5] H.W. Guesgen. Relational connectionist networks. In *Proc. ANNES-93*, pages 23–28, Dunedin, New Zealand, 1993.

[6] H.W. Guesgen and J. Hertzberg. *A Perspective of Constraint-Based Reasoning*. Lecture Notes in Artificial Intelligence 597. Springer, Berlin, Germany, 1992.

[7] H.W. Guesgen, K. Ho, and P.N. Hilfinger. A tagging method for parallel constraint satisfaction. *Journal of Parallel and Distributed Computing*, 16:72–75, 1992.

[8] K. Ho, P.N. Hilfinger, and H.W. Guesgen. Optimistic parallel discrete relaxation. In *Proc. IJCAI-93*, pages 268–273, Chambéry, France, 1993.

[9] W. Hower. Constraint satisfaction via partially parallel propagation steps. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, pages 234–242. Springer Verlag, Berlin, Germany, 1992.

[10] S. Kasif. Parallel solutions to constraint satisfaction problems. In *Proc. KR-89*, pages 180–188, Toronto, Canada, 1989.

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

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

[13] A. Rosenfeld. Networks of automata: Some applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 5:380–383, 1975.

[14] A. Samal and T.C. Henderson. Parallel consistent labeling algorithms. *International Journal of Parallel Programming*, 16:341–364, 1987.