From Serial to Massively Parallel Constraint Satisfaction*

Hans Werner Guesgen
Auckland University
Computer Science Report No. 69

March 1993

Abstract

Local propagation algorithms such as Waltz filtering and Mackworth's AC-x algorithms have been successfully applied in AI for solving constraint satisfaction problems (CSPs). It has been shown that they can be implemented in parallel very easily. However, algorithms like Waltz filtering and AC-x are not complete. In general, they can only be used as preprocessing methods as they do *not* compute a globally consistent solution for a CSP; they result in local consistency also known as arc consistency.

In this paper, we introduce extensions of local constraint propagation to overcome this drawback, i.e. to compute globally consistent solutions for a CSP. The idea is to associate additional information with the values during the propagation process so that global relationships among the values are maintained. The result are algorithms that are complete and for which there are straightforward, parallel and massively parallel implementations.

1 Introduction

Constraint satisfaction algorithms have been applied successfully in many subfields of AI, such as circuit analysis [21], computer vision [23], diagnosis [3, 8, 4], logic programming [14], and planning [22]. The realizations of constraint satisfaction algorithms lie in a broad spectrum, having serial implementations at the one end and massively parallel ones at the other. This paper describes a family of constraint satisfaction algorithms for which serial, parallel, and massively parallel implementations exist. Unless other approaches, the algorithms are easy to compare, since they are all based on the same principle: They apply local propagation combined with techniques that associate additional information with the values of the constraint variables.

The paper may be viewed as part three of a trilogy, the first part of which [13] describes how constraints can be used with other knowledge representation formalisms such as frames, rules, logic, etc. and the second part of which [10] introduces a constraint language and an interpreter for this language. In part three, we will report about the

*The author performed part of this work while at the German National Research Center for Computer Science (GMD) in St. Augustin, Germany, and the International Computer Science Institute in Berkeley, California. At the GMD he was supported by the German Federal Ministry for Research and Technology within the joint projects TEX-B (grant ITW8506D) and TASSO (grant ITW8900A7).

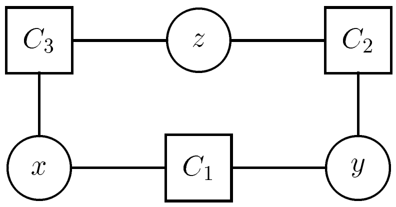

Figure 1: Graph of the constraint network N_1. The variables are represented by circles and the constraints by rectangles. An edge between a circle and a rectangle means that the corresponding variable belongs to the constraint represented by the rectangle.

realization of constraint satisfaction algorithms. Although this part is closely connected to the other two parts, it is a report of its own and should be understandable without knowing its predecessors.

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.

Consider, for example, the following coloring problem: Three fields of a map are to be colored with either *red* or *green* in such a way that adjacent fields have different colors. This problem can be represented in a constraint network N_1, consisting of the constraints C_1, C_2, and C_3, and the variables x, y, and z (cf. figure 1). C_1, C_2, and C_3 are binary constraints over the domain $D = \{red, orange, yellow, green, blue, purple\}$, each constraint realizing the relation $\{(red, green), (green, red)\}$.

A brute-force approach to finding a solution for a constraint network is to use a backtracking algorithm. However, a distributed implementation of backtracking requires decomposability of the given CSP. Beyond that, Dechter & Meiri [5] have shown that backtracking alone is inefficient for many CSPs. To improve backtracking, a variety of algorithms has been developed which may be viewed as preprocessing methods and which achieve several kinds of consistency for a constraint network such as arc consistency, path consistency, etc. (see [17]). For example, initializing x, y, and z with D, respectively, arc consistency algorithms such as Mackworth's AC-x would reduce D to $\{red, green\}$.

The key idea of the AC-x algorithms is local constraint propagation: A constraint is evaluated and the result is propagated to its direct neighbors in the network. It has been shown that there are parallel versions of local propagation algorithms. Kasif [15], Rosenfeld [19], and Samal & Henderson [20] have introduced such algorithms. Moreover, there are also massively parallel algorithms: AC Chip which can be implemented directly in VLSI and which computes an arc-consistent solution to CSPs almost instantaneously, and ACP which has been designed for SIMD computers like the Connection Machine (see [2] as reference for both algorithms). They are closely related to Mohr and Henderson's AC-4 algorithm [18], which is optimal for doing arc consistency on single-processor machines.

The purpose of this paper is to show how algorithms based on local propagation (such as the AC-x algorithms) can be used to compute a global solution for a CSP (rather than only as a preprocessing method resulting in arc consistency) and how these algorithms can be realized in a serial, parallel, and massively parallel manner. Our approach 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.

2 The Tagging Method

The tagging method is based on the idea of maintaining global relationships among the values during the propagation process. Since in a typical local propagation algorithm the variables are associated with value *sets* rather than single values, it does not make sense to describe dependencies among variables as in, for example, reason maintenance systems. For the constraint network of figure 1, e.g., it is inadequate to propose that x depends on y. Instead, it is more appropriate to state that the value *red* of x depends on the value *green* of y, and vice versa, as constraint networks are not directed in most cases.

So the question is: How can relationships among values be represented in an undirected way so that they can be handled efficiently during local propagation? One answer is to construct higher-order constraints until an m-ary constraint (m being the number of variables in the network) is obtained that represents the relation of the network [7]. The solution presented here is to tag values during the propagation process, assigning identical tags to values whose combination satisfies the constraints.

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 integers, whereas the tags used in constraint networks, i.e. the tags that are propagated among the constraints, are more complex. Suppose that the network consists of n constraints[1]. Then, every tag is an n-tuple where the ith value in the tuple is the tag assigned by the ith constraint. For example, $red_{(2,1,3)}$ means that C_1, C_2, and C_3 assigned the tags 2, 1, and 3, respectively. The advantage of using tuples as tags in constraint networks is obvious: Each subtag in the tuple can be uniquely mapped to a constraint of the network which facilitates their handling, especially in the case of hierarchical constraint networks.

Before a constraint is evaluated, the tags of the values of its corresponding variables are simplified. Suppose that the ith constraint of the network is to be evaluated, then only the ith position in the tag is of interest. Hence, every value is simplified before evaluation, replacing its tag by the ith position of the tag. For example, the value $red_{(2,1,3)}$ is simplified to red_1 when C_2 is to be evaluated. The projection assures that a subordinated constraint can only manipulate the part of the tag that is related to the constraint.

After the tags have been simplified, a standard algorithm for constraint evaluation is applied. Such an algorithm can be formulated as follows:[2] 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 tags of each

[1]Each of these constraints may also be a subordinated constraint network, i.e. the tagging algorithm is not restricted to flat networks but may also be applied to hierarchies of constraint networks.

[2]In [10], a more efficient way to evaluate constraints is discussed. However, we do not apply it here for reasons of simplicity and clarity.

tuple of the Cartesian product. This is done in the following way. First, the common tag is computed, which is a new tag if all values of the tuple are untagged. If some values are already provided with tags, and if all of them are identical, then the common tag is determined by this tag; otherwise it is undefined. For example, the common tag of red_2 and $green$ is 2, whereas the common tag of red_2 and $green_3$ is undefined. If the common tag of a tuple is undefined, the tuple represents an invalid value combination with respect to the tags, and therefore is deleted from the set of permitted tuples. In the other case, the tags of the values in the tuple are updated by the common tag.

After the evaluation of a constraint, the projection procedure described above is executed in the opposite way. For that purpose, the subtags, i.e. the tags which result from the evaluation process, are merged with the original tags. After that, tags which are merged with the same subtag are unified. Unification in this context means that the components of the tags are compared, and if the known components are equal, a common tag exists and the unknown components are substituted by this tag. In the case where unification is not possible, the corresponding values are deleted.

For example, let $red_{(1,2,-)}$ and $green_{(-,-,3)}$ be values for the variables y and z, respectively. When C_2 is activated, the tuple $(red_2, green)$ is matched with the constraint relation, resulting in $(red_2, green_2)$. The new tags are merged with the original ones (which is trivial for the value red):

$$\left.\begin{array}{l} red_2 \\ red_{(1,2,-)} \end{array}\right\} \longmapsto red_{(1,2,-)}$$
$$\left.\begin{array}{l} green_2 \\ green_{(-,-,3)} \end{array}\right\} \longmapsto green_{(-,2,3)}$$

Since the subtags with which the original tags are merged are both identical, the resulting tags must be unified:

$$\begin{array}{lcl} red_{(1,2,-)} & \longmapsto & red_{(1,2,3)} \\ green_{(-,2,3)} & \longmapsto & green_{(1,2,3)} \end{array}$$

3 Application of the Tagging Method to N_1

In the following, we will illustrate the tagging method by applying it to the constraint network N_1 whose graph is shown in figure 1. The initial variable coverings are:

$$x = y = z = \{red, orange, yellow, green, blue, purple\}$$

C_1 removes the values *orange*, *yellow*, *blue*, and *purple* from x and y and provides *red* and *green* with tags:

$$\begin{array}{lcl} x & = & \{red_1, green_2\} \\ y & = & \{red_2, green_1\} \end{array}$$

Then the merging/unifying step computes the following:

$$\begin{array}{lcl} x & = & \{red_{(1,-,-)}, green_{(2,-,-)}\} \\ y & = & \{red_{(2,-,-)}, green_{(1,-,-)}\} \\ z & = & \{red, orange, yellow, green, blue, purple\} \end{array}$$

Now either C_2 or C_3 may be evaluated. Let us suppose that C_2 is the one selected to be evaluated first. Then the tags of y (z does not yet have any tags) are simplified such that each tag tuple is replaced by the second element of the tuple. The resulting values are constrained by C_2 as follows:

$$\left.\begin{array}{rcl} y &=& \{red, green\} \\ z &=& \{red, green\} \end{array}\right\} \xrightarrow{\text{evaluation}}$$

$$\left.\begin{array}{rcl} y &=& \{red_3, green_4\} \\ z &=& \{red_4, green_3\} \end{array}\right\} \xrightarrow{\text{merging/unifying}}$$

$$\begin{array}{rcl} x &=& \{red_{(1,-,-)}, green_{(2,-,-)}\} \\ y &=& \{red_{(2,3,-)}, green_{(1,4,-)}\} \\ z &=& \{red_{(1,4,-)}, green_{(2,3,-)}\} \end{array}$$

The evaluation of C_3 detects that there is no global solution for N_1:

$$\left.\begin{array}{rcl} x &=& \{red, green\} \\ z &=& \{red, green\} \end{array}\right\} \xrightarrow{\text{evaluation}}$$

$$\left.\begin{array}{rcl} x &=& \{red_5, green_6\} \\ z &=& \{red_6, green_5\} \end{array}\right\} \xrightarrow{\text{merging}}$$

$$\left.\begin{array}{rcl} x &=& \{red_{(1,-,5)}, green_{(2,-,6)}\} \\ y &=& \{red_{(2,3,-)}, green_{(1,4,-)}\} \\ z &=& \{red_{(1,4,6)}, green_{(2,3,5)}\} \end{array}\right\} \xrightarrow{\text{unifying}}$$

$$\begin{array}{rcl} x &=& \emptyset \\ y &=& \{red_{(2,3,-)}, green_{(1,4,-)}\} \\ z &=& \emptyset \end{array}$$

The first subtag of $red_{(1,-,5)}$ is unequal to the first subtag of $green_{(2,3,5)}$. The same holds for $green_{(2,-,6)}$ and $red_{(1,4,6)}$. Thus, unification results in empty variable coverings for x and z.

After activating the first constraint once more, the inconsistency in the network is completely detected and the empty set is associated with each variable:

$$\begin{array}{rcl} x &=& \emptyset \\ y &=& \emptyset \\ z &=& \emptyset \end{array}$$

In general, not all values are removed from the variable coverings. The remaining values specify the solution(s) of the CSP, i.e. it is admissible to combine those values to solution tuples that have the same tag.

Consider, for example, the following modifications to the example, resulting in the constraint network N_2:

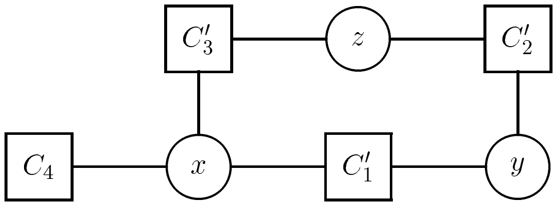

Figure 2: Graph of the constraint network N_2.

1. C_1, C_2, and C_3 are extended to C_1', C_2', and C_3', respectively, by adding *yellow*, i.e. the relations of C_1', C_2', and C_3' are identical and equal to:

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

2. The constraint C_4 is inserted into the network, constraining the variable x to be *red* (cf. figure 2).

The algorithm would assign quadruples as tags, the fourth position corresponding to the constraint C_4. Assuming that this constraint is evaluated first, followed by an evaluation of the constraints C_1', C_2', and C_3' as in the example above, the result would be:

$$\begin{array}{rcl} x & = & \{red_{(2,4,6,1)}, red_{(3,5,7,1)}\} \\ y & = & \{green_{(2,4,6,1)}, yellow_{(3,5,7,1)}\} \\ z & = & \{yellow_{(2,4,6,1)}, green_{(3,5,7,1)}\} \end{array}$$

These variable coverings state that there are two solutions for the constraint network N_2, namely (*red*, *green*, *yellow*) and (*red*, *yellow*, *green*).

4 Parallel Implementation

In the previous section, we illustrated the tagging method by embedding it in a serial local propagation algorithm.[3] As other authors have shown, local propagation can be implemented on parallel machines in a variety of different ways [2, 15, 19, 20]. In principle, the tagging algorithm works like a common local propagation algorithm. The difference is only in the values that are propagated, which, in our case, contain more information than just possible values of the variables, namely their relationships to other values. It is thus possible to implement the tagging algorithm on a parallel computer.

In the following, the rough sketch of a simple parallel local propagation algorithm is given:

1. Assign a processor to each variable and each constraint. Communication is allowed only between different types of processors, i.e. between constraints and their corresponding variables.

[3] We implemented this algorithm in Lisp in a straightforward manner.

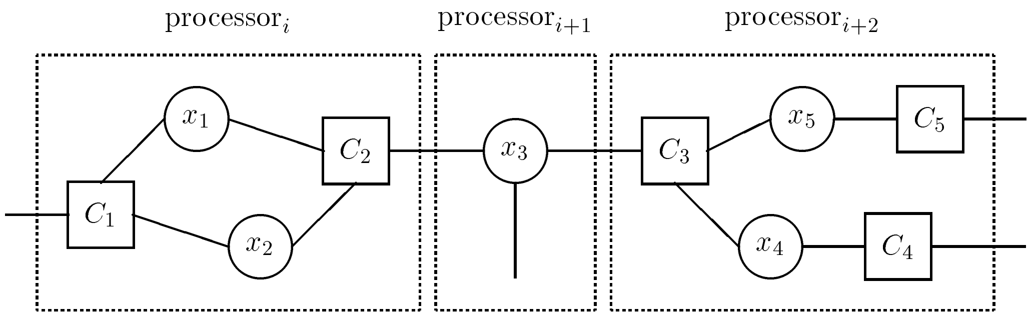

Figure 3: Parallel implementation with a limited number of processors.

2. Activate the processors. The constraints receive the current values of their variables, evaluate their relations, and send the results back to the variables which intersect the incoming values.

3. If the variables do not receive any new messages, i.e. if the network is stable, then:

   (a) Deactivate the processors.

   (b) Return the variable coverings.

Details about the parallel implementation of the tagging algorithm can be found in [12].

In the case where the number of available processors is not necessarily equal to or greater than the number of variables plus the number of constraints in the network, the above scheme can be extended as follows. Let p be the number of available processors, m be the number of variables, and n be the number of constraints. If $p \geq m + n$, then each variable and each constraint can be associated with its own processor. If $p < m + n$, then some processors obtain subnetworks, performing local propagation among these (cf. figure 3).

5 Connectionist Constraint Satisfaction

We will now illustrate how connectionist networks may be used for constraint satisfaction. Similar to [2], the nodes used in our networks are in accordance with the unit/value principle (cf. [6]): A separate connectionist node is dedicated to each value of each variable and each tuple of each constraint of the constraint network. To compute global consistency (rather than arc consistency as in [2]), we apply a technique that is closely related to the tagging method.

Let V be the set of variables on which the constraints are defined, and let D be their domains. We represent each variable-value pair $\langle x, a \rangle$, $x \in V$, $a \in D$, by a connectionist node (v-node) and denote such a node by $v\langle x, a \rangle$.

We restrict ourselves here to binary constraints, i.e. constraints consisting of sets of pairs, each pair representing a consistent value assignment for the variables. We introduce a connectionist node (c-node) for each quadruple $\langle x, y, a, b \rangle$ with $x, y \in V$ and $a, b \in D$,

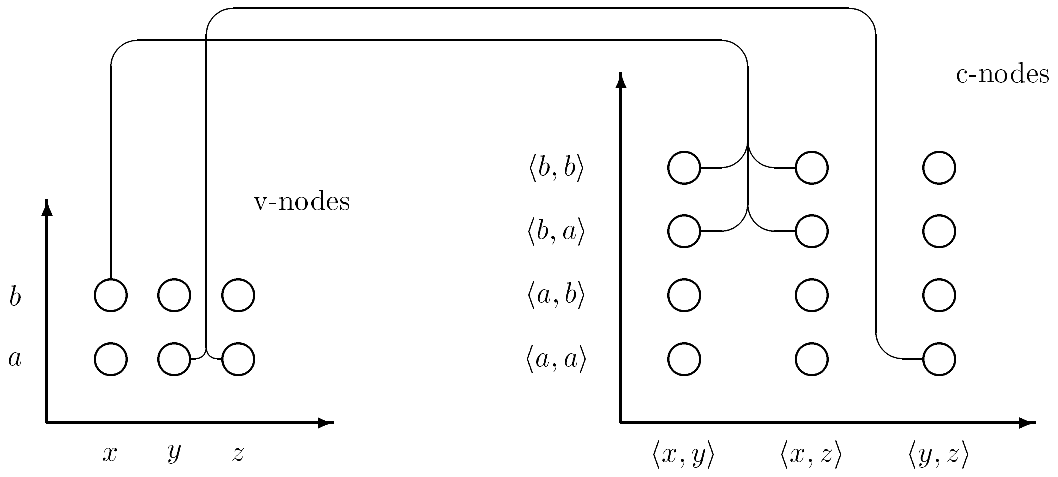

Figure 4: Scheme of a connectionist network for constraint satisfaction.

and denote such a node by $c\langle x,y,a,b\rangle$. Because of symmetry, $c\langle x,y,a,b\rangle$ and $c\langle y,x,b,a\rangle$ denote the same node.

This representation corresponds directly to the one in [2]. As it is shown there, v-nodes and c-nodes can be connected in such a way that arc consistency is computed (cf. figure 4). For that purpose, the nodes are initialized as follows:

- Each v-node obtains potential 1.
- A c-node $c\langle x,y,a,b\rangle$ obtains potential 1, if $\langle a,b\rangle$ is an admissible assignment of values for $\langle x,y\rangle$; else it obtains potential 0.

Since it is more convenient, we will use $v\langle x,a\rangle$ for referring to the potential of a node as well as for denoting the node itself.

A v-node is reset to 0 if one cannot find at least one v-node for every other variable such that the constraint between this v-node and the given v-node is satisfied. This rule is called the arc consistency label discarding rule:

$$\mathrm{reset}(v\langle x,a\rangle) = \neg \bigwedge_{y\in V} \bigvee_{b\in D} (v\langle y,b\rangle \wedge c\langle x,y,a,b\rangle)$$

This is equivalent to:

$$v\langle x,a\rangle = \begin{cases} 1 & \text{if } \forall y \in V \exists b \in D : v\langle y,b\rangle \wedge c\langle x,y,a,b\rangle \\ 0 & \text{else} \end{cases}$$

We will show in the next section how the connectionist networks sketched above can be extended such that global consistency is computed rather than arc consistency. The approach described in that section may be compared with the one in [16], where signatures are used to maintain variable bindings.

6 Towards Global Consistency

The idea is to use the potential of a v-node to encode information about how the variable/value pair contributes to a solution (and not only whether or not it does so). The information is composed from codings that are associated with the c-nodes. In particular, we encode each c-node by a prime number and denote this encoding by a function $e : V^2 \times D^2 \rightarrow P$ from the set of c-nodes to the set of prime numbers.

For example, let $V = \{x, y\}$ and $D = \{\mathit{red}, \mathit{green}\}$, then e may be defined as follows:

$$\begin{array}{ll}
e\langle x, x, \mathit{red}, \mathit{red}\rangle = 2 & e\langle x, y, \mathit{red}, \mathit{red}\rangle = e\langle y, x, \mathit{red}, \mathit{red}\rangle = 11 \\
e\langle x, x, \mathit{green}, \mathit{green}\rangle = 3 & e\langle x, y, \mathit{red}, \mathit{green}\rangle = e\langle y, x, \mathit{green}, \mathit{red}\rangle = 13 \\
e\langle y, y, \mathit{red}, \mathit{red}\rangle = 5 & e\langle x, y, \mathit{green}, \mathit{red}\rangle = e\langle y, x, \mathit{red}, \mathit{green}\rangle = 17 \\
e\langle y, y, \mathit{green}, \mathit{green}\rangle = 7 & e\langle x, y, \mathit{green}, \mathit{green}\rangle = e\langle y, x, \mathit{green}, \mathit{green}\rangle = 19
\end{array}$$

Nodes such as $c\langle x, x, \mathit{red}, \mathit{green}\rangle$ do not make sense and therefore are omitted in the coding.

We will again use the same notation for a node and its potential, i.e. the term $c\langle x, y, a, b\rangle$ may denote the connectionist node representing the tuple $\langle a, b\rangle$ of the constraint between x and y, or may denote the potential of that node. It is determined by the context which meaning is intended.

The initial potentials of c-nodes are the same as in [1]. A c-node $c\langle x, y, a, b\rangle$ is assigned the potential 1, if there is a pair $\langle a, b\rangle$ in the constraint between x and y; else it is 0. The initial potential of a v-node $v\langle x, a\rangle$ is determined by the product of the codes of all c-nodes except those that refer to the same variable as the given v-node but to a different value for that variable (i.e. a factor $e\langle x, .., b, ..\rangle$ with $b \neq a$ does not occur in the product):

$$\prod_{\substack{y \in V \\ b \in D}} e\langle x, y, a, b\rangle \prod_{\substack{y_1, y_2 \in V \setminus \{x\} \\ b_1, b_2 \in D}} \sqrt{e\langle y_1, y_2, b_1, b_2\rangle}$$

The square root is due to the fact that $c\langle y_1, y_2, b_1, b_2\rangle$ and $c\langle y_2, y_1, b_2, b_1\rangle$ are identical nodes.

Unlike computing arc consistency (in which a v-node's potential is reset to 0 if it is inconsistent), we will perform here what is called graceful degradation:

1. A c-node receives the potentials of its v-nodes and computes their greatest common divisor (gcd).

2. The gcd is returned to the v-nodes if the c-node has potential 0; else 1 is returned.

3. A v-node computes the least common multiples (lcm) of data coming in from c-nodes that refer to the same variables and combines these by computing their gcd.

The idea is that the potentials of v-nodes shall reflect paths in the network that correspond to solutions of the constraint satisfaction problem. A v-node may be on the same path as another v-node if the c-node between them has potential 1. We start with allowing all paths among the v-nodes. Whenever a part of path is determined that is not admissible, i.e. the corresponding c-node has potential 0, the path is deleted.

This means that global information about solution paths is held locally in the v-nodes of the network. To keep this information consistent, the c-nodes compute the gcd of the potentials of neighboring v-nodes. The gcd reflects that piece of information neighboring

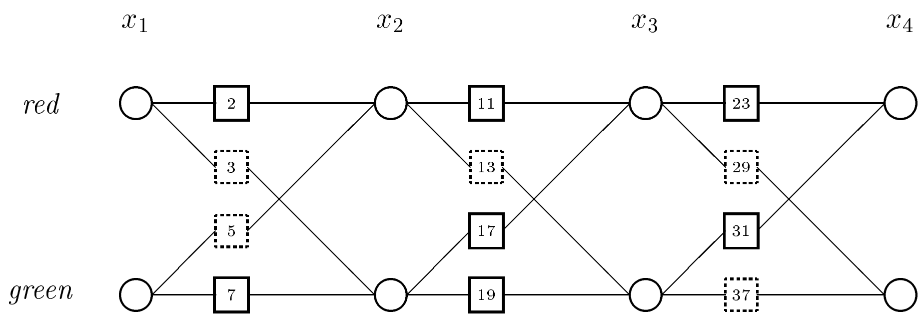

Figure 5: Connectionist network for a simple constraint satisfaction problem.

v-nodes can agree on. In order to consider alternatives, the v-nodes compute the lcm of data that comes in from c-nodes connecting to the same variable, and combine the results by applying the gcd operator. The alternation between the application of gcd and lcm directly corresponds to the semantics of constraints and their constituting tuples: A constraints network can be viewed as a conjunction of constraints (therefore gcd) whereas a constraint can be viewed as disjunction of tuples (therefore lcm).

More formally, the degradation rule can be denoted as follows:

$$v\langle x, a\rangle \leftarrow \operatorname{gcd}_{y \in V} \operatorname{lcm}_{b \in D} (\operatorname{out}(c\langle x, y, a, b\rangle))$$

with

$$\operatorname{out}(c\langle x, y, a, b\rangle) = \begin{cases} \gcd(v\langle x, a\rangle, v\langle y, b\rangle) & \text{if } c\langle x, y, a, b\rangle = 1 \\ 1 & \text{else} \end{cases}$$

Since the degradation rule is monotonous and discrete, the network finally settles down. After that, the potentials of the v-nodes characterize the set of solutions of the given constraint satisfaction problem. In particular, a solution is given by a subset of v-nodes, W, for which the following holds:

1. Every variable occurs exactly once in W.
2. The potentials of the v-nodes in W are divisible by p, where:

$$p = \prod_{v\langle x,a\rangle, v\langle y,b\rangle \in W} \sqrt{e\langle x, y, a, b\rangle}$$

(Again, the square root is due to the fact that $c\langle x, y, a, b\rangle$ and $c\langle y, x, b, a\rangle$ are identical nodes.)

7 Illustration of the Connectionist Approach

We will now illustrate our approach by a small example. Figure 5 shows a part of a connectionist network for a constraint problem with variables x_1, x_2, x_3, and x_4, which are constrained by binary constraints according to the following table:

Variables	Constraint
x_1, x_2	$\{(red, red), (green, green)\}$
x_2, x_3	$\{(red, red), (green, green), (green, red)\}$
x_3, x_4	$\{(red, red), (green, red)\}$

Here, we use circles for the representation of v-nodes, boxes with solid boundary lines for c-nodes that have potential 1, and boxes with dashed boundary lines for c-nodes that have potential 0. For the sake of simplicity, c-nodes that correspond to universal constraints, i.e. constraints with the whole Cartesian product as relation, have been omitted. This simplification is admissible since all v-nodes are already connected by nonuniversal constraints, guaranteeing that inconsistent codes are removed from the potentials of the v-nodes.

Figures 6 and 7 show a sequence of tables in which listings of v-node potentials and c-node outputs alternate. It illustrates how the network is initialized and how it settles down.

The example suggests that nodes in the center of the network ($v\langle x_2, red\rangle$, $v\langle x_2, green\rangle$, $v\langle x_3, red\rangle$, and $v\langle x_3, green\rangle$) settle down faster than nodes at the periphery ($v\langle x_1, red\rangle$, $v\langle x_1, green\rangle$, $v\langle x_4, red\rangle$, and $v\langle x_4, green\rangle$). This, however, is only because we simplified the example and left out some connections. In a network with full connectivity, there is no distinction between center nodes and peripheral nodes; therefore, these networks settle down in a more uniform way.

8 Summary

In this paper, we introduced a class of constraint satisfaction algorithms that can be characterized as follows:

- The algorithms use local propagation to interchange information in a constraint network.
- Additional data in form of tags or Gödel numbers is associated with the values of the constraint variables to maintain dependencies.
- They compute global consistency rather than arc consistency, i.e. they determine the solutions of a given CSP.

We described serial, parallel, and massively parallel realizations of the algorithms and illustrated them by means of examples. A discussion of the complexity, soundness, and completeness of the algorithms can be found in [9] and [11].

9 Acknowledgements

Thanks to the members of the TEX-B project and the XPS research group at the GMD who gave comments on earlier versions, especially Manfred “Manilac” Fidelak (who is now with the University of Koblenz), Joachim Hertzberg, and Marc Linster. I am also

Initial State:

V-Node	Potential
$v\langle x_1, red\rangle$	$2 \cdot 3 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_1, green\rangle$	$5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_2, red\rangle$	$2 \cdot 5 \cdot 11 \cdot 13 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_2, green\rangle$	$3 \cdot 7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_3, red\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 23 \cdot 29$
$v\langle x_3, green\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 13 \cdot 19 \cdot 31 \cdot 37$
$v\langle x_4, red\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_4, green\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 29 \cdot 37$

C-Node	Output
$c\langle x_1, x_2, red, red\rangle$	$2 \cdot 11 \cdot 13 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$c\langle x_1, x_2, red, green\rangle$	1
$c\langle x_1, x_2, green, red\rangle$	1
$c\langle x_1, x_2, green, green\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$c\langle x_2, x_3, red, red\rangle$	$2 \cdot 5 \cdot 11 \cdot 23 \cdot 29$
$c\langle x_2, x_3, red, green\rangle$	1
$c\langle x_2, x_3, green, red\rangle$	$3 \cdot 7 \cdot 17 \cdot 23 \cdot 29$
$c\langle x_2, x_3, green, green\rangle$	$3 \cdot 7 \cdot 19 \cdot 31 \cdot 37$
$c\langle x_3, x_4, red, red\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$c\langle x_3, x_4, red, green\rangle$	1
$c\langle x_3, x_4, green, red\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 13 \cdot 19 \cdot 31$
$c\langle x_3, x_4, green, green\rangle$	1

2nd State:

V-Node	Potential
$v\langle x_1, red\rangle$	$2 \cdot 11 \cdot 13 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_1, green\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_2, red\rangle$	$2 \cdot 11 \cdot 23 \cdot 29$
$v\langle x_2, green\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_3, red\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$v\langle x_3, green\rangle$	$3 \cdot 7 \cdot 19 \cdot 31$
$v\langle x_4, red\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_4, green\rangle$	1

C-Node	Output
$c\langle x_1, x_2, red, red\rangle$	$2 \cdot 11 \cdot 23 \cdot 29$
$c\langle x_1, x_2, red, green\rangle$	1
$c\langle x_1, x_2, green, red\rangle$	1
$c\langle x_1, x_2, green, green\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$c\langle x_2, x_3, red, red\rangle$	$2 \cdot 11 \cdot 23$
$c\langle x_2, x_3, red, green\rangle$	1
$c\langle x_2, x_3, green, red\rangle$	$7 \cdot 17 \cdot 23$
$c\langle x_2, x_3, green, green\rangle$	$7 \cdot 19 \cdot 31$
$c\langle x_3, x_4, red, red\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$c\langle x_3, x_4, red, green\rangle$	1
$c\langle x_3, x_4, green, red\rangle$	$3 \cdot 7 \cdot 19 \cdot 31$
$c\langle x_3, x_4, green, green\rangle$	1

Figure 6: Sequence of v-node potentials and c-node outputs.

3rd State:

V-Node	Potential
$v\langle x_1, \mathit{red}\rangle$	$2 \cdot 11 \cdot 23 \cdot 29$
$v\langle x_1, \mathit{green}\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_2, \mathit{red}\rangle$	$2 \cdot 11 \cdot 23$
$v\langle x_2, \mathit{green}\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_3, \mathit{red}\rangle$	$2 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$v\langle x_3, \mathit{green}\rangle$	$7 \cdot 19 \cdot 31$
$v\langle x_4, \mathit{red}\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_4, \mathit{green}\rangle$	1

C-Node	Output
$c\langle x_1, x_2, \mathit{red}, \mathit{red}\rangle$	$2 \cdot 11 \cdot 23$
$c\langle x_1, x_2, \mathit{red}, \mathit{green}\rangle$	1
$c\langle x_1, x_2, \mathit{green}, \mathit{red}\rangle$	1
$c\langle x_1, x_2, \mathit{green}, \mathit{green}\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$c\langle x_2, x_3, \mathit{red}, \mathit{red}\rangle$	$2 \cdot 11 \cdot 23$
$c\langle x_2, x_3, \mathit{red}, \mathit{green}\rangle$	1
$c\langle x_2, x_3, \mathit{green}, \mathit{red}\rangle$	$7 \cdot 17 \cdot 23$
$c\langle x_2, x_3, \mathit{green}, \mathit{green}\rangle$	$7 \cdot 19 \cdot 31$
$c\langle x_3, x_4, \mathit{red}, \mathit{red}\rangle$	$2 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$c\langle x_3, x_4, \mathit{red}, \mathit{green}\rangle$	1
$c\langle x_3, x_4, \mathit{green}, \mathit{red}\rangle$	$7 \cdot 19 \cdot 31$
$c\langle x_3, x_4, \mathit{green}, \mathit{green}\rangle$	1

Final State:

V-Node	Potential
$v\langle x_1, \mathit{red}\rangle$	$2 \cdot 11 \cdot 23$
$v\langle x_1, \mathit{green}\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_2, \mathit{red}\rangle$	$2 \cdot 11 \cdot 23$
$v\langle x_2, \mathit{green}\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_3, \mathit{red}\rangle$	$2 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$v\langle x_3, \mathit{green}\rangle$	$7 \cdot 19 \cdot 31$
$v\langle x_4, \mathit{red}\rangle$	$2 \cdot 7 \cdot 11 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_4, \mathit{green}\rangle$	1

Figure 7: Sequence of v-node potentials and c-node outputs (continued).

grateful to Jerry Feldman and Peter Ladkin for their contributions to this paper. Last but not least I would like to thank Alix Collison and Renee Reynolds who struggled with my English.

References

[1] P.R. Cooper. Parallel object recognition from structure (the tinkertoy project). Technical Report 301, University of Rochester, Computer Science Department, Rochester, New York, 1989.

[2] 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.

[3] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[4] J. de Kleer and B.C. Williams. Diagnosing multiple faults. In *Proc. AAAI-86*, pages 132–139, Philadelphia, Pennsylvania, 1986.

[5] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proc. IJCAI-89*, pages 271–277, Detroit, Michigan, 1989.

[6] J.A. Feldman and D.H. Ballard. Connectionist models and their properties. *Cognitive Science*, 6:201–254, 1982.

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

[8] H. Geffner and J. Pearl. An improved constraint-propagation algorithm for diagnosis. In *Proc. IJCAI-87*, pages 1105–1111, Milan, Italy, 1987.

[9] H.W. Guesgen. A tagging method for distributed constraint satisfaction. Technical Report TR-89-037, ICSI, Berkeley, California, 1989.

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

[11] H.W. Guesgen. A connectionist approach to symbolic constraint satisfaction. Technical Report TR-90-018, ICSI, Berkeley, California, 1990.

[12] 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.

[13] H.W. Guesgen, U. Junker, and A. Voß. Constraints in a hybrid knowledge representation system. In *Proc. IJCAI-87*, pages 30–33, Milan, Italy, 1987.

[14] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Conference Record of the 14 th Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, 1987.

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

[16] T.E. Lange and M.G. Dyer. High-level inferencing in a connectionist network. Technical Report UCLA-AI-89-12, University of California, Los Angeles, California, 1989.

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

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

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

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

[21] R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[22] M. Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16:111–140, 1981.

[23] D.L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI-TR-271, MIT, Cambridge, Massachusetts, 1972.