Distributing Constraints by Sampling in Non-binary CSPs

Miguel A. Salido*, Adriana Giret†, Federico Barber†
* *Dpto. Ciencias de la Computación e I.A.*
Universidad de Alicante
Alicante, Spain
† Dpto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Valencia, Spain
{msalido,agiret,fbarber}@dsic.upv.es

Abstract.

In constraint satisfaction, a general rule is to tackle the hardest part of a search problem first. Many Constraint Satisfaction Problems (CSPs) are solved using search algorithms, which require an order in which variables and values should be considered. Choosing the right order of variables and values can noticeably improve the efficiency of constraint satisfaction. The order in which constraints are studied can also improve efficiency, particularly in problems with non-binary constraints.

In this paper, we present a distributed model for solving non-binary CSPs, in which agents are committed to sets of constraints. A preprocessing agent is committed to ordering the constraints by a sample in finite population so that the tightest constraints are studied first. This preprocessing agent is only applied in problems where constrainedness cannot be known in advance. Then, a set of agents are incrementally and concurrently committed to building partial solutions until a problem solution is found. This constraint ordering, as well as value and variable ordering, can improve efficiency because inconsistencies can be found earlier and the number of constraint checks can be significantly reduced.

1 Introduction

Nowadays, many real problems in Artificial Intelligence (AI) as well as in other areas of computer science and engineering can be efficiently modeled as Constraint Satisfaction Problems (CSPs) and solved using constraint programming techniques. Some examples of such problems include: spatial and temporal planning, qualitative and symbolic reasoning, diagnosis, decision support, scheduling, hardware design and verification, real-time systems and robot planning. Some of these problems can be modeled naturally using non-binary (or n-ary) constraints. The need to address issues regarding non-binary constraints has recently started to be widely recognized in the constraint satisfaction literature. However, researchers have traditionally focused on binary constraints [15]. Thus, defining models to solve non-binary CSPs becomes relevant.

General methods for solving CSPs include Backtracking-based search algorithms. While the worst-case complexity of backtrack search is exponential, several heuristics to reduce its average-case complexity have been proposed in the literature [3]. For instance, some algorithms incorporate features such as variable ordering which have a substantially better performance than a simpler algorithm without this feature [10], and yet the two share the same worst-case complexity.

Many works have investigated various ways of improving the backtracking-based search algorithms. In order to avoid *thrashing* [11] in Backtracking, *consistency* techniques, such as *arc-consistency* and *k-consistency*, have been developed by many researchers. These techniques are able to remove inconsistent values from the domains of the variables. Other ways of increasing the efficiency of Backtracking include the use of *search order* for variables and values. Thus, some heuristics based on *variable ordering* and *value ordering* [13] have been developed, due to the additivity of the variables and values. However, constraints are also considered to be *additive*, that is, the order of imposition of constraints does not matter; all that matters is that the conjunction of constraints be satisfied [1].

In spite of the additivity of constraints, only some works have be done on binary constraint ordering mainly for arc-consistency algorithms and for problems whose constrainedness is known in advance [16, 9], but little work has be done on non-binary constraint ordering (for instance in disjunctive constraints [14]), and on problems whose constrainedness cannot be known in advance. Only some heuristic techniques classify the non-binary constraints by means of the arity. However, less arity does not imply a tighter constraint. Moreover, when all non-binary constraints have the same arity, or these constraints are classified as hard and soft constraints, these techniques are not useful.

In this paper, we propose a distributed model in which the constraints are ordered and partitioned into a set of subproblems and solved by search algorithms. Thus, these subproblems are classified so that the tightest subproblems are studied first. This is based on the *first-fail* principle, which can be explained as

"To succeed, try first where you are more likely to fail"

This classification is straightforward if the constrainedness is known in advance. However in the general case, where constrainedness cannot be known in advance, this classification must be obtained in a preprocessing step in which an agent called *preprocessing agent* carries out a sample in finite population, as in statistics, in which a well-distributed sample of states from the search space represents the entire search space. This sample is checked with all the constraints in order to classify them from the tightest constraints to the loosest constraints. Afterwards, constraints are partitioned in k blocks. Each block contains different number of constraints (distributed by a geometric progression) such that the first block maintains very few constraints but they are the tightest constraints and the last block maintains many constraints but they are the loosest constraints. Each block of constraints will be studied by agents called *block agents*. However, as in statistics, although our objective is to select a well-distributed sample, an incorrect constraint classification may be obtained, so a repair method is dynamically carried out to classify the constraints in the appropriate order. Thus, constraints are labelled to identify the number of violated tuples.

Block agent 1 works on the tightest constraints, that is, the constraints that are more likely to fail, and so, inconsistent tuples can be found earlier. If *block agent* 1 finds a solution to its partial problem, then *block agent* 2 begins to study the second set of tightest constraints using the consistent partial state generated by *block agent* 1. Concurrently, *block agent* 1 continues studying its subproblem to obtain another consistent partial state, and so on. Finally, *block agent* k, using the variable assignments of the previous agents, attempts to find a problem solution with its group of constraints (the loosest ones). This model allows agents to run concurrently to achieve partial solutions, and it removes the drawbacks of synchronous backtracking algorithms [19].

In the following section, we formally define a constraint satisfaction problem and describe well-known ordering algorithms. Section 3 describes the multi-agent model. Section 4 presents the computational complexity. We present the results of the evaluation in section 5 and finally, we present our conclusions in section 6.

2 Definitions and Algorithms

In this section, we review some basic definitions as well as basic heuristics for CSPs.

2.1 Definitions

CSP: Generally, a constraint satisfaction problem (CSP) consists of:

- a set of variables $X = \{x_1, x_2, ..., x_n\}$
- a set of domains $D = \{D_1, D_2, ..., D_n\}$, where each variable $x_i \in X$ has a set D_i of possible values
- a finite collection of constraints $C = \{c_1, c_2, ..., c_p\}$ restricting the values that the variables can simultaneously take.

State: one possible assignment of all variables; the number of states is equal to the product of the domain size.

Partition : A partition of a set C is a set of disjoint subsets of C whose union is C. The subsets are called the blocks of the partition.

Distributed CSP: A distributed CSP is a CSP in which the variables and constraints are distributed among automated agents [19].

Each agent has some variables and attempts to determine their values. However, there are interagent constraints and the value assignment must satisfy these interagent constraints. In our model, there are k agents $1, 2, ..., k$. Each agent knows a set of constraints and the domains of variables involved in these constraints.

Objective in a CSP: A *solution* to a CSP is an assignment of values to all the variables so that all constraints are satisfied. The objective in a CSP may be to determine:

- whether a solution exists, that is, if the CSP is consistent.
- all solutions, many solutions, or only one solution, with no preference as to which one.
- an optimal, or a good solution by means of an objective function defined in terms of certain variables.

In some real problems, it is desirable to find all solutions in order to give the user the ability to search the design space for the best solution, particularly when various parameters are difficult to model [4]. Some techniques such as value ordering are not valid to solve this type of problems. In this case, it is necessary to be able to efficiently find the *dead-ends* in order to reduce the search tree.

Two ordering algorithms are analyzed in [13, 1]: variable ordering and value ordering. Let's briefly look at these two algorithms.

2.2 Variable Ordering

The experiments and analyses by various researchers have shown that the ordering in which variables are assigned during the search may have substantial impact on the complexity of the search space explored. The ordering may be either a static ordering or a dynamic ordering. Examples of static ordering heuristics are *minimum width* [5] and *maximum degree* [2], in which the order of the variables is specified before the search begins and is not changed thereafter. An example of a dynamic ordering heuristic is *minimum remaining values* [10], in which the choice of the next variable to be considered at any point depends on the current state of the search.

Dynamic ordering is not feasible for all search algorithms. For example, with simple backtracking, there is no extra information available during the search that could be used to make a different choice of ordering from the initial ordering. However, with forward checking, the current state includes the domains of the variables as they have been pruned by the current set of instantiations. Therefore, it is possible to base the choice of the next variable on this information.

2.3 Value Ordering

The order in which values are considered during the search can have substantial impact on the time necessary to find the first solution. There exist some algorithms for value ordering [7, 6]. The basic idea behind value ordering algorithms is to select the value for the current variable which is most likely to lead to a solution. However, if all solutions are required or the problem is not consistent, then the value ordering does not make any difference. A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than a branch which leads to a dead-end. For example, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking.

Suppose we have selected a variable to instantiate: how should we choose which value to try first? It may be that none of the values will succeed. In that case, every value for the current variable will eventually have to be considered and the order does not matter. On the other hand, if we can find a complete solution based on the past instantiations, we want to choose a value which is likely to succeed and unlikely to lead to a conflict.

2.4 Constraint Ordering

Comparatively little work has been done on constraint ordering. In spite of the additivity of constraints, only some works have be done on constraint ordering. Heuristics of making a choice that minimises the constrainedness of the resulting subproblem can reduce search over standards heuristics [8].

Wallace and Freuder initiated a systematic study to identify factors that determine the efficiency of constraint propagation that achieve arc consistency [16].

Gent et al. proposed a new constraint ordering heuristic in AC3, where the set of choices is composed by the arcs in the current set maintained by AC3 [9]. They considered the remaining subproblem to have the same set of variables as the original problem, but with only those arcs still remaining in the set.

Walsh studied the constrainedness ”knife-edge” in which he measured the constrainedness of a problem during search in several different domains [17]. He observed a constrainedness ”knife-edge” in which critically constrained problems tend to remain critically constrained. This knife-edge is predicted by a theoretical lower-bound calculation.

Many of these algorithms focus their approximate theories on just two factors: the size of the problems and the expected number of solutions. However, the expected number of solutions is not easy to estimate in many real problems.

However, few work has be done on non-binary constraint ordering for general CSPs and only some heuristics classify the non-binary constraints by means of the arity. We will focus on this point, where the non-binary constraints will be classified from the tightest one to the loosest one, in problems where tightnesses cannot be known in advance.

3 The Multi-Agent Model

Agent-based computation has been studied for several years in the field of artificial intelligence and has been widely used in other branches of computer science. Multi-agent systems are computational systems in which several agents interact or work together to achieve goals. Agents in such systems may be homogeneous or heterogeneous and may have common goals or distinct goals [12].

As we pointed out in the above definitions in section 2.1, a distributed constraint satisfaction problem (distributed CSP) is a constraint satisfaction problem in which variables and constraints are semantically partitioned (or distributed) into subproblems, each of which is to be solved by an agent.

In this section, we will provide the definitions and specifications of the different agents involved in these models and the formulation for our proposed multi-agent model.

Definition 1: A *block agent* a_j is a virtual entity that essentially has the following properties: autonomy, social ability, reactivity and pro-activity [18].

Block agents are autonomous agents. They operate their subproblems without the direct intervention of any other agent or human. *Block agents* interact with each other by sending messages to communicate consistent partial states or to exchange constraints. They perceive their environment and changes in it, such as new partial consistent states, and react, if possible, with more complete consistent partial states. *Block agents* take the initiative by evicting constraints that are tightest than others sending them to or exchanging them with previous *block agents*.

Definition 2: A *multi-agent system* is a system that contains the following elements:

1. An environment in which the agents live (variables, domains, constraints and consistent partial states).
2. A set of reactive rules, governing the interaction between the agents and their environment (constraint exchange rules, communication rules, etc).

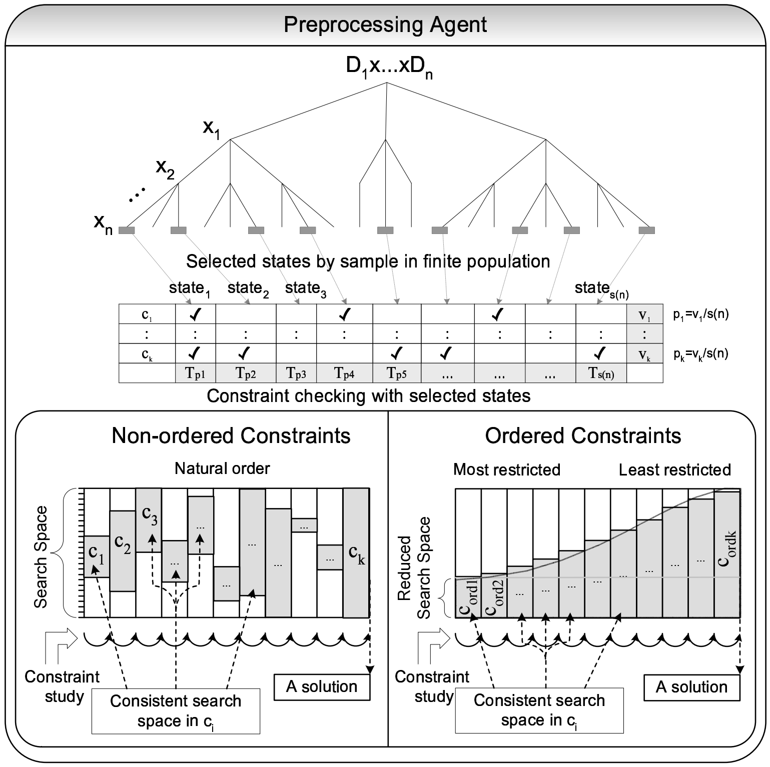

Figure 1: The preprocessing agent

3. A set of agents, $A = \{a_1, a_2, ..., a_k\}$.

3.1 The Preprocessing Agent

The *preprocessing agent* carries out a preprocessing step based on the sampling in finite population, as in statistics, where there is a target population and a sampled population is chosen to represent this population. In our context, the population is made up of the states generated by means of the product of variable domains. The *preprocessing agent* chooses a sampled population composed by $s(n)$ states of the target population (s is a polynomial function). These states are well distributed in order to represent the target population. As in statistics, the user may select the size of the sample $s(n)$ as well as the distribution function. Figure 1 represents the preprocessing agent. With the selected sample of states $s(n)$, the *preprocessing agent* checks how many states $v_i : v_i \leq s(n)$ satisfy each constraint c_i. Thus, each constraint c_i is labelled with pr_i: $c_i(pr_i)$, where $pr_i = v_i/s(n)$ represents the probability that c_i satisfies the whole problem. Thus, the *preprocessing agent* classifies the constraints in ascending order of the labels pr_i. Therefore, the *preprocessing agent* translates the initial non-binary CSP into an ordered one so that it can be studied by a CSP solver. In Figure 1, it can be observed that each constraint is checked with each selected state. Furthermore, each state of the sample might store the evaluation value T_i to be used by a stochastic local search algorithm to restart the search.

The ordered constraints must be partitioned in k blocks of constraints to be managed by agents called *block agents* such that all *block agents* work equally. To this end, the *preprocessing agent* must carry out a balanced partition such that a *block agents* with tight constraints must manage few constraints and *block agents* with loosest constraints must manage many constraints. Thus, *block agent* 1 is committed to solving the most restricted subproblem and *block agent* k is committed to solving the least restricted subproblem.

To obtain an appropriate number of *block agent* k, we will distribute the number of constraints by a geometric progression, that is, by a sequence of number such that the quotient of any two successive members of the sequence is a constant. Thus, the sum of terms of this geometric progression is a geometric series whose sum must be equal to the number of constraints p. The sum of a geometric series can be computed quickly with the formula

$$\sum_{h=1}^{k} x^k = \frac{x^{k+1} - x}{x - 1} = p \tag{1}$$

$$x^{k+1} - x = (x-1)p \Rightarrow log x^{k+1} = log((x-1)p + x)$$

$$\Rightarrow (k+1) log x = log((x-1)p + x) \Rightarrow$$

$$k + 1 = \frac{log((x-1)p + x)}{log x} \Rightarrow k = \left\lfloor \frac{log((x-1)p + x)}{log x} \right\rfloor - 1$$

For instance, let's suppose a problem with 65 constraints (p=65) and a geometric progression with common quotient 2 ($x = 2$), we obtain that the number of block k is:

$$k = \left\lfloor \frac{log(65+2)}{log 2} \right\rfloor - 1 = 5$$

Thus, there are 5 blocks of constraints so that the *block agents* 1,2,3,4 and 5 maintain 2, 4, 8, 16, 35 constraints, respectively. It can be observed that the last *block agent* maintains 2^5 constraints plus the remaining constraints ($2^5 + 3$) to complete the total number of constraints. Thus, the first *block agent* is committed to the smaller set of constraints, but they are the tightest constraints, while the last *block agent* is committed to the greater set of constraints, but they are the loosest constraints. In this way, all *block agents* work equally.

Nevertheless, the user may select the number of blocks of constraints or the common quotient for the geometric progression in order to solve the problem.

3.2 The Block Agents

Block agents are agents committed to solving subproblems. As we pointed out in definition 1, an agent has a set of properties. Following, we present the behavior and characteristics of *block agents* (Figure 2):

- Each *block agent* a_j has an identifier j.

- There is a partition of the set of constraints $C \equiv \bigcup_{i=1}^{k} C(i)$ and each *block agent* a_j is committed to the block of constraints $C(j)$. Each constraint is labelled with the number of violated constraints.

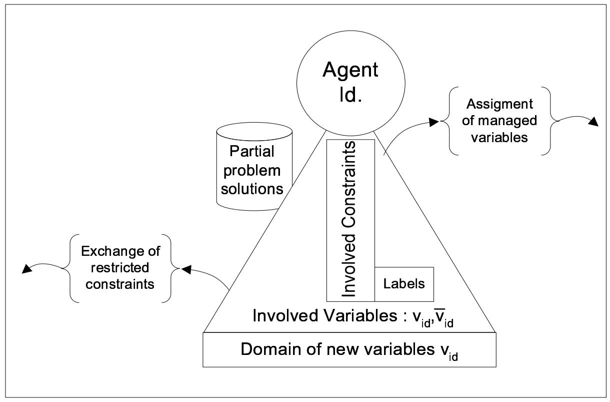

Figure 2: Properties and characteristics of *block agents*.

- Each *block agent* a_j has a set of variables V_j involved in its block of constraints $C(j)$. These variables fall into two different sets: *used variables* set ($\overline{v}_j$) and *new variables* set (v_j), that is: $V_j = \overline{v}_j \cup v_j$.
- The domain D_i corresponding to variable x_i is maintained in the first *block agent* a_t in which x_i is involved, (i.e.), $x_i \in v_t$.
- Each *block agent* a_j assigns values to variables that have not been assigned yet, that is, a_j assigns values to variables $x_i \in v_j$, because variables $x_k \in \overline{v}_j$ have already been assigned by previous *block agents* $a_1, a_2, ..., a_{j-1}$.
- Each *block agent* a_j maintains a storage of partial problem solutions generated by the previous *block agents* $a_1, a_2, ..., a_{j-1}$. Thus, *block agent* a_j maintains assignments of variables included in sets: $\overline{v}_1, \overline{v}_2, ..., \overline{v}_{j-1}$.
- Each *block agent* a_j can send constraints with the highest labels to the previous *block agent* a_{j-1} to be managed. In this case, the *new variables* involved in these constraints are also sent to the previous *block agent* with their corresponding variable domains.

Thus, these *block agents* are committed to solving CSPs that represent subproblems of the main CSP. These *block agents* must cooperate with each other by sending messages with consistent partial states or exchanging constraints.

In what follows, we present an overview of our multi-agent formulation in which we analyze the relationship among all agents.

3.3 Overview of the Multi-Agent Formulation

In the specialized literature, there are many works about distributed CSPs. In [19], Yokoo et al. present a formalization and algorithms for solving distributed CSPs. These algorithms can be classified as either centralized methods, synchronous backtracking or asynchronous backtracking [20].

Our model can be considered as a synchronous model. It is meant to be a framework for interacting agents to achieve a consistent state. The main idea of our multi-agent model is based on partitioning the problem constraints in k groups called *blocks* of constraints so that the tightest constraints are grouped and studied first by autonomous agents. To this end, a *preprocessing agent* carries out a partition of the constraints, similar to a sample in finite population, with the objective of classifying the constraints in k groups, from the tightest ones to the loosest ones. As we pointed out in Figure 1, each selected state of the sample can store the evaluation value T_i to be used by a stochastic local search algorithm in order to restart the search. Note that if some state has an evaluation value of zero, ($T_i = 0$), (the state does not violate any constraint), then a solution is found.

Once the constraints are divided into k blocks by the *preprocessing agent*, a group of *block agents* concurrently manages each block of constraints. Each *block agent* is in charge of solving its own subproblem by means of a search algorithm. Each *block agent* is free to select any algorithm to find a consistent partial state. It can select a local search algorithm, a backtracking-based algorithm, or any other, depending on the problem topology. In any case, each *block agent* is committed to finding a solution to its particular subproblem. This subproblem is composed by its CSP subject to the variable assignment generated by the previous *block agents*. Thus, *block agent* 1 works on the most restricted block of constraints. If *block agent* 1 finds a solution to its subproblem, then it sends the consistent partial state to *block agent* 2, and both they work concurrently to solve their specific subproblems; *block agent* 1 tries to find other solution and *block agent* 2 tries to solve its subproblem knowing that its *used variables* have been assigned by *block agent* 1. Thus, *block agent* j, with the variable assignments generated by the previous *block agents*, works concurrently with the previous *block agents*, and tries to find a more complete consistent state using a search algorithm. Finally, the last *block agent* k, working concurrently with *block agents* $1, 2, ...(k-1)$, tries to find a consistent state in order to find a problem solution. Note that as the *block agent* identifier gets higher, the number of *new variables* gets lower. Therefore, the set of *new variables* in *block agents* with a high identifier *(k-2,k-1,k)* may be empty. In this case, these *block agents* need only check their constraints with the states sent by the previous *block agents*.

3.3.1 Dynamic repair method to exchange constraints:

The *preprocessing agent* may not correctly classify the constraints from the tightest one to the loosest one. This is due to the fact that the size of the sample is not appropriate or the sample has not been correctly selected. In this case, *block agents* can apply a dynamic repair method to exchange constraints with each other. Each constraint is labelled to identify the number of violated tuples. Each *block agent* maintains an upper bound of the label value. If the upper bound is reached, *block agent i* and *block agent i-1* negotiate the exchange of the highest label constraints of *block agent i* for the lowest label constraints of *block agent i-1*. This way, the sampled population grows more and more and the constraint ordering becomes more exact.

Figure 3 shows the multi-agent model, in which the *preprocessing agent* carries out a constraint ordering and the *block agents* (a_i) are committed to concurrently finding partial problem solutions (s_{ij}). Each *block agent* sends the partial problem solutions to the following *block agent* until a problem solution is found (by the last *block agent*). For example, state: $s_{11} + s_{21} + ... + s_{k1}$ is a problem solution. The concurrence can be seen in Figure 3 in *Time step* 6 in which all *block agents* are concurrently working. Each *block agent* maintains the corresponding domains for its *new variables*. The *block agent* must assign values to its *new variables* so that the block of non-binary constraints is satisfied. When a *block agent* finds

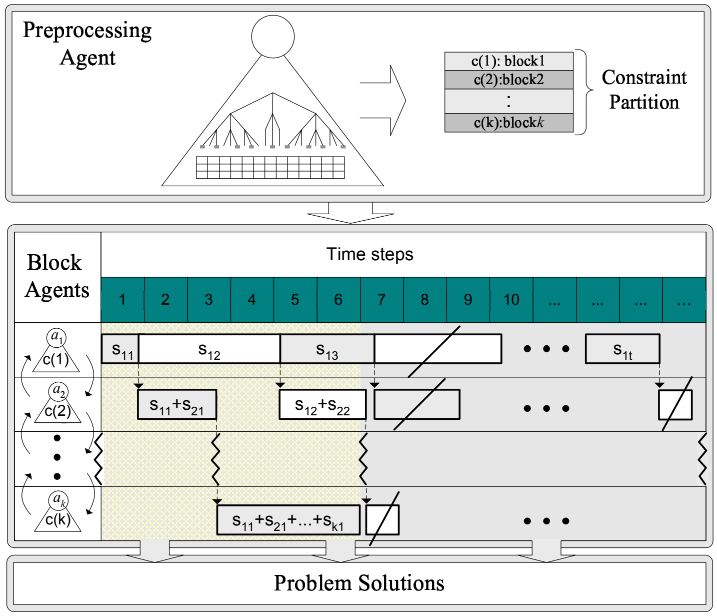

Figure 3: Multi-agent model

a value for each *new variable*, it then sends the consistent partial state to the next *block agent*. When the last *block agent* assigns values to its *new variables* satisfying its block of constraints, then a solution is found. The dynamic repair method can also be applied if the constraint ordering has not been correctly carried out.

Example: Let's look at a similar example that is presented in [19]. There are three variables, x_1, x_2, x_3, with variable domains $\{1,2,3\}, \{1,2\}, \{1,2,3\}$, respectively, and constraints $c_1 : x_1 \neq x_2$ and $c_2 : x_2 = x_3$ (see Figure 4).

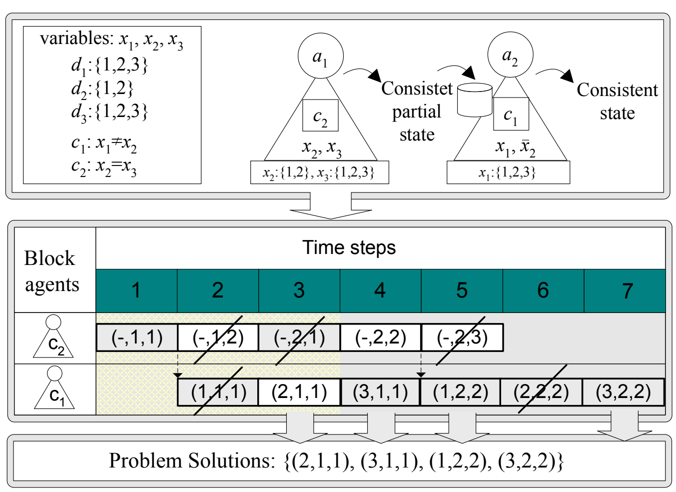

Figure 4: CSP solved by our model

As there are only two constraints, the constraint partition is straightforward. Therefore, there are only two blocks with one constraint in each block to consider. The first block is composed of constraint c_2 and the second block is composed of constraint c_1. This is due to the fact that constraint c_2 is tightest than constraint c_1 as c_2 maintains two valid tuples:

$(-, 1, 1)$ and $(-, 2, 2)$, while c_1 maintains four valid tuples: $(1, 2, -)$, $(2, 1, -)$, $(3, 1, -)$ and $(3, 2, -)$. *Block agent* a_1 manages constraint 1 and *block agent* a_2 manages constraint 2. It can be observed that variables x_2 and x_3 are *new variables* in a_1, and x_1 is a *new variable* in a_2, while x_2 is a *used variable* in a_2. Thus, domains of variable x_2 and x_3 are known by a_1, and the domain of x_1 is known by a_2. Furthermore, a_1 is responsible for assigning values to x_2 and x_3 using a search algorithm, and a_2 is responsible for assigning values to x_1. Figure 4 shows the behavior of our distributed model using GT. It can be observed that *Time step* 1 is only used by a_1 to generate a consistent partial state (-,1,1). Thus, a_1 sends a message to a_2 with the consistent partial state $(-, 1, 1)$. In *Time step* 2, both a_1 and a_2 work concurrently to find a consistent partial state for their own problems. a_2 tests state (1,1,1) which is not a solution and simultaneously a_1 tests partial state (-,1,2) which is not a consistent partial state. In *Time step* 3, a_2 tests state (2,1,1) which is a consistent state, that is, a solution. Meanwhile a_1 tests partial state (-,2,1). If only one solution is required, the process is halted. If more solutions are required, the model continues with steps 4,5,6 and 7.

Example (The 5-Queens Problem): This well-known problem is an example of a discrete problem with five variables and eleven constraints.

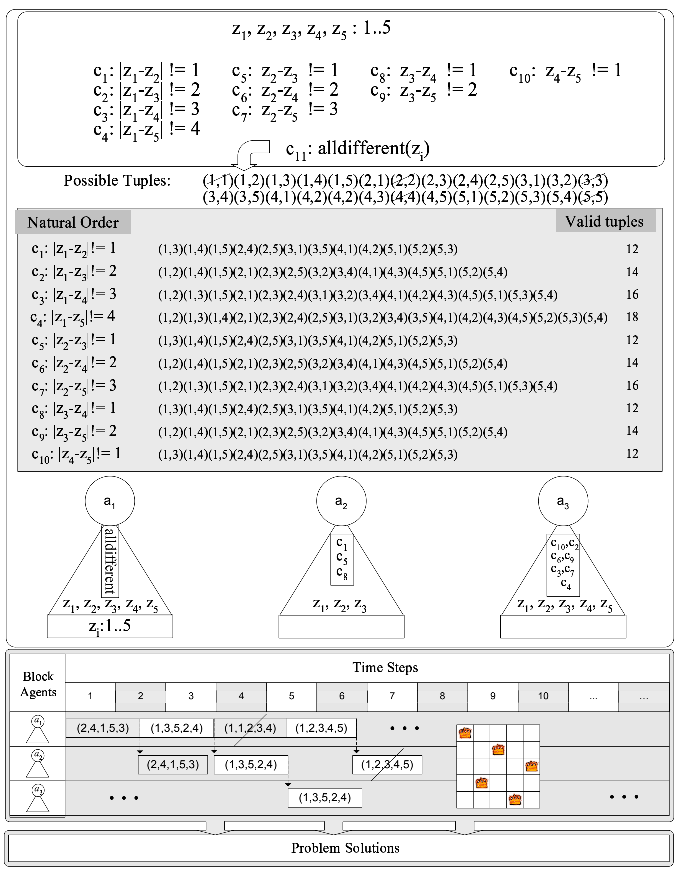

Figure 5: The 5-queens problem using our multi-agent model

Figure 5 shows the initial CSP and the number of valid tuples in each disequation. The *preprocessing agent* generates 3 blocks of constraints corresponding to a geometric progression with common quotient 3. The first *block agent* has only one constraint *all-different* constraint, the second *block agent* has 3 constraints and the third *block agent* has 7 constraints.

Thus, each *block agent* is committed to its block of constraints and solves its subproblem by means of a search algorithm. It can be observed that all *block agents* can work concurrently when the last *block agent* receives a consistent (partial) state. In section 5, we present an exhaustive evaluation of the n-queens problem in which we can observe the constraint check saving.

4 Analysis of Our Distributed Model

In this section, we only evaluate the computational cost of the *preprocessing agent* because each *block agent* can use any search algorithm with the corresponding computational complexity. The *preprocessing agent* selects a sample composed of $s(n)$ points, so the spatial cost is $O(s(n))$. The *preprocessing agent* checks the consistency of the sample with each non-binary constraint, so its temporal cost is $O(ks(n))$. Then, the *preprocessing agent* classifies the set of constraints in ascending order. Its temporal complexity is $O(klogk)$. Thus, the temporal complexity of the *preprocessing agent* is $O(max\{ks(n), klogk\})$.

5 Evaluation of Our Model

In this section, we compare the performance of our model with some well-known CSP solvers: Chronological Backtracking (BT), Generate&Test (GT), Forward Checking (FC) and Real Full Look Ahead (RFLA), because they are the most appropriate techniques for observing the number of constraint checks. Furthermore, we compare the performance of our model with Hill-Climbing because it is a well-known local search algorithm for analyzing the number of restart savings.

This empirical evaluation was carried out with two different types of problems: benchmark problems and random problems.

Benchmark Problems

The n-queens problem is a classical search problem in the artificial intelligence area. The 5-queens problem was studied in the previous section. The 5-queens problem is an instance of the n-queens problem with 10 possible solutions. The problem is to place five queens on a 5×5 chessboard so that no two queens can capture each other. That is, no two queens are allowed to be placed on the same row, the same column, or the same diagonal. In the general n-queens problem, a set of *n* queens is to be placed on a $n \times n$ chessboard so that no two queens attack each other.

In Table 1, we present the amount of constraint check saving in the n-queens problem using GT with our model (Mod+GT), BT with our model (Mod+BT), Forward Checking with our model (Mod+FC) and Real Full Look Ahead with our model (Mod+RFLA). Here, our objective is to find all solutions. The results show that the amount of constraint check saving was significant in Mod+GT and Mod+BT due to the fact that our model classifies the constraints in ascending order (see Figure 5), and inconsistent tuples were discarded earlier. Furthermore, the amount of constraint check saving was also significant in Mod+FC and Mod+RFLA in spite of being more powerful algorithms than BT and GT.

The percentage of restart savings using Hill-Climbing with our model is also presented in Table 2. Hill-Climbing uses the sample and selects the states with the lowest labels (T_i) to

Table 1: Amount of constraint check saving using our model with GT and BT in the n-queens problem.

	Mod+GT	**Mod+BT**	**Mod+FC**	**Mod+RFLA**
queens	*Constraint Check Saving*	*Constraint Check Saving*	*Constraint Check Saving*	*Constraint Check Saving*
5	2.1×10^4	2.4×10^2	150	110
10	4.1×10^{11}	3.9×10^7	1.4×10^5	9.3×10^4
20	1.9×10^{26}	3.6×10^{18}	9.6×10^{14}	6.03×10^{11}
50	2.4×10^{70}	3.6×10^{52}	3.1×10^{44}	1.6×10^{32}
100	2.1×10^{143}	2.1×10^{106}	4.5×10^{93}	1.8×10^{66}
150	5.2×10^{219}	3.7×10^{161}	6.8×10^{142}	2.1×10^{100}
200	9.4×10^{295}	8.7×10^{219}	9.9×10^{198}	2.2×10^{134}

Table 2: Percentage of restart savings using Hill-Climbing *(Max-Flip=n)* in the n-queens problem.

	Mod+Hill-Climbing
queens	*Percentage of Restart Savings*
4	57.62%
5	70.41%
6	75.84%
7	80.04%
8	83.32%
9	85.51%
13	91.50%

restart the search. Here, our objective is to find only one solution. It can be observed that the percentage was high in the 4-queens problem where the number of restarts was reduced by 57.62%. In the 13-queens problem, the percentage of restarts was reduced by 91.5%.

Random Problems

Benchmark sets are used to test algorithms for specific problems. However, in recent years, there has been a growing interest in the study of the relation among the parameters that define an instance of CSP in general (i.e., the number of variables, constraints, domain size, arity of constraints, etc). Therefore, the notion of randomly generated CSPs has been introduced to describe the classes of CSPs. These classes are then studied using empirical methods.

In our empirical evaluation, each set of random constraint satisfaction problems was defined by the 3-tuple $< n, c, d >$, where n was the number of variables, c the number of constraints and d the domain size. The problems were randomly generated by modifying these parameters. We considered all constraints as global constraints, that is, all constraints had maximum arity. Thus, Tables 3 and 4 sets two of the parameters and varies the other one in order to evaluate the algorithm performance when this parameter increases. We evaluated 100 test cases for each type of problem and each value of the variable parameter.

The number of constraint checks using BT filtered by *arc-consistency* (as a preprocessing) (BT-AC) and BT-AC using our model (Mod+BT-AC) is presented in Table 3. We present the number of constraint checks in problems where the number of constraints was increased from 3 to 15 and the number of variables and the domain size were set at 5 and 10, respectively: $< 5, c, 10 >$. The results show that the number of constraint checks were reduced in all cases.

Table 3: Number of constraint checks using BT filtered with Arc-Consistency in problems classes $< 5, c, 10 >$.

	BT-AC	**Mod+BT-AC**
problems	*constraint checks*	*constraint checks*
$< 5, 3, 10 >$	2275.5	798.5
$< 5, 5, 10 >$	14226.3	2975.2
$< 5, 7, 10 >$	35537.4	5236.7
$< 5, 9, 10 >$	50315.7	5695.5
$< 5, 11, 10 >$	65334	5996.3
$< 5, 13, 10 >$	80384	6283.5
$< 5, 15, 10 >$	127342	8598.6

Table 4: Number of constraint checks using BT filtered with Arc-Consistency in problems classes $< 3, 5, d >$.

	BT-AC	**Mod+BT-AC**
problems	*constraint checks*	*constraint checks*
$< 3, 5, 5 >$	78.9	17.7
$< 3, 5, 10 >$	150.3	33.06
$< 3, 5, 15 >$	196.3	41.26
$< 3, 5, 20 >$	260.5	55.1
$< 3, 5, 25 >$	344.8	68.9
$< 3, 5, 30 >$	424.6	85.9
$< 3, 5, 35 >$	550.4	110.1

Table 4 presents the number of constraint checks in problems where the domain size was increased from 5 to 35 and the number of variables and the number of constraints were set at 3 and 5, respectively: $< 3, 5, d >$. The results were similar and the constraint checks were also reduced in all cases.

6 Conclusion and Future work

In this paper, we present a distributed model for solving non-binary CSPs, in which a *preprocessing agent* is committed to ordering and partitioning the constraints so that the tightest constraints are studied first. This preprocessing agent is only applied in problems where constrainedness cannot be known in advance. Then, a set of *block agents* are incrementally and concurrently committed to building partial solutions until a global solution is found. Thus, inconsistent tuples can be found earlier with the corresponding savings in constraint checking. Also, hard problems can be solved more efficiently overall in problems where all solutions are required.

As future work, we are working on a distributed model in which the *preprocessing agent* can be removed due to the fact that *block agents* can dynamically exchange constraints. Thus, the set of constraints are initially partitioned in k randomly groups. Each of them will be managed by a *block agent*. They will be able to exchange constraints depending on the constrainedness and the relationship among variables in the constraint network. Furthermore, the number of *block agents* can be enlarged or reduced depending on the problem topology.

References

[1] Bartk, R. 1999. Constraint programming: In pursuit of the holy grail. *in Proceedings of WDS99 (invited lecture), Prague, June*.

[2] Dechter, R., and Meiri, I. 1994. Experimental evaluation of preprocessing algorithms for constraints satisfaction problems. *Artificial Intelligence* 68:211–241.

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

[4] Denk, T., and Parhi, K. 1998. Exhaustive scheduling and retiming of digital signal processing systems. *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing* 45:821–837.

[5] Freuder, E. 1982. A sufficient condition for backtrack-free search. *Journal of the ACM* 29:24–32.

[6] Frost, D., and Dechter, R. 1995. Look-ahead value orderings for constraint satisfaction problems. *In Proc. of IJCAI-95* 572–578.

[7] Geelen, P. 1992. Dual viewpoint heuristic for binary constraint satisfaction problems. *In proceeding of European Conference of Artificial Intelligence (ECAI'92)* 31–35.

[8] Gent, I.; MacIntyre, E.; Prosser, P.; and Walsh, T. 1996a. The constrainedness of search. *In Proceedings of AAAI-96* 246–252.

[9] Gent, I.; MacIntyre, E.; Prosser, P.; and Walsh, T. 1997b. The constrainedness of arc consistency. *Principles and Practice of Constraint Programming* 327–340.

[10] Haralick, R., and G., E. 1980. Increasing tree efficiency for constraint satisfaction problems. *Artificial Intelligence* 14:263–314.

[11] Kumar, V. 1992. Algorithms for constraint satisfaction problems: a survey. *Artificial Intelligence Magazine* 1:32–44.

[12] Liu, J. 2001. Autonomous agents and multi-agent systems: Explorations in learning, self-organization, and adaptative computation. *World Scientific, Singapore*.

[13] Sadeh, N., and Fox, M. 1990. Variable and value ordering heuristics for activity-based jobshop scheduling. *In proc. of Fourth International Conference on Expert Systems in Production and Operations Management* 134–144.

[14] Salido, M., and Barber, F. 2003. A Polynomial Algorithm for Continuous Non-binary Disjunctive CSPs: Extended DLRs. *Knowledge Based Systems. Ed. Elsevier Science* 16:277–285.

[15] Tsang, E. 1993. *Foundation of Constraint Satisfaction*. London and San Diego: Academic Press.

[16] Wallace, R., and Freuder, E. 1992. Ordering heuristics for arc consistency algorithms. *In Proc. of Ninth Canad. Conf. on A.I.* 163–169.

[17] Walsh, T. 1998. The constrainedness knife-edge. *In Proceedings of the 15th National Conference on AI (AAAI-98)* 406–411.

[18] Wooldridge, M., and Jennings, R. 1995. Agent theories, arquitectures, and lenguajes: a survey. *Intelligent Agents* 1–22.

[19] Yokoo, M.; Durfee, E.; Ishida, T.; and Kuwabara, K. 1998. The distributed constraint satisfaction problem: formalization and algorithms. *In IEEE Transactions on Knowledge and Data Engineering* 10:673–685.

[20] Yokoo, M. 1995. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. *Proc. of the First International Conference on Principles and Practice of Constraint Programming* 88–102.