# Stochastic Local Search for Distributed Constraint Satisfaction Problems

**Miguel A. Salido**[*], **Federico Barber**[†]

[*]Dpto. Ciencias de la Computación e Inteligencia Artificial. Universidad de Alicante
Campus de San Vicente, Ap. de Correos: 99, E-03080, Alicante, Spain

[†]Dpto. de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia
Camino de Vera s/n, 46071, Valencia, Spain
{msalido, fbarber}@dsic.upv.es

## Abstract

Nowadays, many real problems can be solved using local search strategies. These algorithms incrementally alter inconsistency value assignments to all the variables using a *repair* or *hill climbing* metaphor to move towards more and more complete solutions. Furthermore, if the problem can be modeled as a distributed problem, the advantages can be even greater.

This paper presents a distributed model for solving Constraint Satisfaction Problems (CSPs), in which agents are committed to sets of constraints. The problem constraints are ordered and partitioned, by a preprocessing step, so that the most restricted constraints are studied first. Thus, each agent solves a subproblem by means of a stochastic local search algorithm. 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). Some examples of such problems include: spatial and temporal planning, qualitative and symbolic reasoning, diagnosis, decision support, scheduling, real-time systems and robot planning.

General methods for solving CSPs include *Generate and test* [Kumar, 1992] and *Backtracking* [Kumar, 1987] algorithms. Many works have been carried out to improve the *Backtracking* method. One way of increasing the efficiency of *Backtracking* includes the use of *search order* for variables and values. Some heuristics based on *variable ordering* and *value ordering* [Sadeh, 1990] have been developed, because of 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 [Bartak, 1999].

In spite of the additivity of constraints, little work has been done over constraint ordering. In this paper, we propose a distributed model in which a preprocessing step classifies the constraints in $k$ sets, so that the most restricted constraints (included in set 1) are studied first (by agent 1). This is based on the *first-fail* principle, which can be explained as

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

Our model manages CSPs in a distributed way so that each agent is committed to a set of constraints. The constraints that are more likely to fail are studied first using some local search algorithm. In this way, inconsistent tuples can be found earlier. It must be taken into account the difference between our block structure, to solve classical CSPs, and a hierarchical CSP. Our model internally uses a block structure towards helping search in problems where all constraints are required and no preferences among constraints are defined, whereas a hierarchical CSP is focused on representing preferences defined by the user. However, our model internally transforms any CSP into a hierarchical one by means of an ordered block structure [Salido, 2003].

## 2 Definitions and Algorithms

In this section, we relate some basic definitions as well as basic algorithms for solving CSPs.

### 2.1 Definitions

***CSP:*** Briefly, 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.

***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 [Yokoo *et al.*, 1998].

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 maintains a set of constraints and the 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. A problem with a solution is termed *satisfiable* or *consistent*. 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.

***Terms in local search methodology***:

- **state**: one possible assignment of all variables; the number of states is equal to the Cartesian product of the domain size
- **evaluation value**: the number of constraint violations of the state
- **neighbor**: the state which is obtained from the current state by changing only one variable value
- **local-minimum**: the state that is not a solution and the evaluation values of all its neighbors are larger than or equal to the evaluation value of this state.

## 2.2 Algorithms

There are several algorithms and a lot of improved methods for solving CSPs. According to how many solutions the system seeks concurrently, we can divide the current algorithms in *Single-solution Algorithms* where the system is searching for only one solution at a time and *Multi-solution Algorithms* where the system is parallel searching for several solutions. If considering non$-$basic algorithms, we can also put those hybrid algorithms in this category, such as *Portfolio of Algorithms* and *Cooperative Search* [Hogg, 1993].

However, cooperative search can also be applied to find a single solution. In this way, CSPs can be modeled as distributed CSPs in order to improve efficiency. Every agent is committed to finding a consistent partial state to its own problem and cooperates with the other agents to find a problem solution. Each agent can use any search method for solving CSPs and finding a consistent partial state to its partial problem. Furthermore, a single solution from the first agent can lead to many solutions at the last agent.

From the viewpoint of the search style, algorithms can be classified in two types: *Systematic search* (Backtracking) and *Generate&test* [Kumar, 1992].

***Backtracking*** assigns values to variables sequentially and then checks constraints for each variable assignment. If a partial state does not satisfy any of the constraints, it will go back to the most recently assigned variable and perform the process again. Backtracking cannot solve largescale problems because its search space increases sharply with the problem size and it is not an efficient algorithm. We will use this technique in the evaluation section in order to analyze the behavior of our distributed model.

***Generate&Test*** generates a state and then checks whether it satisfies all the constraints, i.e., checks if it is a solution. The simplest way to generate a state is to randomly select a value for each variable. However, this is a less efficient way. There have been some research efforts to make the generator smarter. The most popular idea is Local Search [Sosic, 1994]. For many large CSPs, it always gives better results than the systematic Backtracking paradigm. It generates an initial (but possibly inconsistent) state and then incrementally uses hill climbing [Selman, 1992] or min-conflict [Minton *et al.*, 1992] to move to a solution with a better evaluation value among its current solution neighborhood, until a solution is found.

Let's look at some well-known local search techniques summarized in [Bartak, 1999]:

- **Hill-Climbing**

  Hill-climbing is probably the most well-known algorithm for local search. The idea of hill-climbing is: to start at randomly generated state; move to the neighbor with the best evaluation value; and if a strict local-minimum is reached then restart at another randomly generated state. This procedure repeats till the solution is found. Generally, this algorithm maintains a parameter called (*Max-Flips*), that is used to limit the maximum number of moves between restarts which helps us to leave non-strict local-minimums. The fact that the hill-climbing algorithm has to explore all the neighbors of the current state before choosing the move must be taken into account. This can take a lot of time.

- **Min-Conflicts**

  To avoid exploring all neighbors of the current state, some heuristics were proposed to find a next move. A Min-conflicts heuristic [Minton *et al.*, 1992] randomly chooses any conflicting variable, that is, the variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints. If no such value exists, it randomly picks a value that does not increase the number of violated constraints. Note, that the pure min-conflicts algorithm is not able to leave local-minimum. In addition, if the algorithm achieves a strict local-minimum it does not perform any move at all and, consequently, it does not terminate with the global solution.

- **Tabu-Search**

  Tabu search (TS) [Glover, 1986] is another method to avoid cycles and getting trapped in local minimums. It is based on the notion of tabu list, which is a special short term memory that maintains a selective history, composed of previously encountered configurations or more generally pertinent attributes of such configurations. A simple TS strategy consists of preventing configurations of tabu list from being recognized for the next $t$ iterations ($t$, called tabu tenue, is the size of tabu list). Such a strategy prevents Tabu from being trapped in short term cycling and allows the search process to go beyond local optima.

- **Simulated Annealing**

Simulated Annealing [Kirkpatrick *et al.*, 1983] is a Monte Carlo approach for combinatorial problems. It is a famous algorithm inspired by the roughly analogous physical process of heating and then slowly cooling a substance to obtain a strong crystalline structure. It can be regarded as a heuristic of the stochastic *Local Search*.

## 3 Our Distributed 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 in order to achieve goals. In the specialized literature, there are many works about distributed CSP. In [Yokoo *et al.*, 1998], Yokoo et al. present a formalization and algorithms for solving distributed CSP. These algorithms can be classified as centralized methods, as synchronous backtracking and as asynchronous backtracking [Yokoo, 1995].

Our model can be considered as a synchronous model. It is meant to be a framework for interacting agents to achieve a global solution state. The main idea of our multi-agent model is based on carrying out a partition of the problem constraints, in $k$ groups called *blocks* of constraints, so that the most restricted constraints are grouped and studied by autonomous agents. To this end, a preprocessing step carries out a partition of the constraints, similar to a sample in finite population, in order to classify the constraints from the most restricted ones to the least restricted ones. Then, a group of *block agents* manages concurrently each block of constraints, generated by the preprocessing step. Each *block agent* is in charge of solving its partial problem by means of a stochastic local search algorithm. Thus, finding a solution to a distributed CSP requires that all *block agents* find a incremental consistent partial state for their own partial problem, that is, the solution is incrementally generated from the first *block agent* to the last *block agent*.

Figure 1 shows the multi-agent model, in which consistent partial states ($s_{ij}$) are concurrently generated by each *block agent* ($a_i$) and sent to the following *block agent* until a consistent state is found (for example, state: $s_{11} + s_{21} + s_{k1}$). Each *block agent* maintains the corresponding domains for its new variables and must assign values to its new variables so that the block of constraints is satisfied. When a *block agent* finds a value for each new variable, then it communicates with the next *block agent* by sending the consistent partial state. Thus, when the last *block agent* assigns values to the new variables that satisfy its block of constraints, then a consistent state is found.

### 3.1 Preprocessing Step

In this section, we present the preprocessing step that classifies the constraints, so that the most restricted constraints are studied first.

As we pointed out above, the preprocessing step carries out a sample from a finite population in statistics, where there is a population, and a sample is chosen to represent this population. In our context, the population is composed by the states generated by means of the Cartesian Product of variable domain bounds and the sample is composed by $s(n)$ random

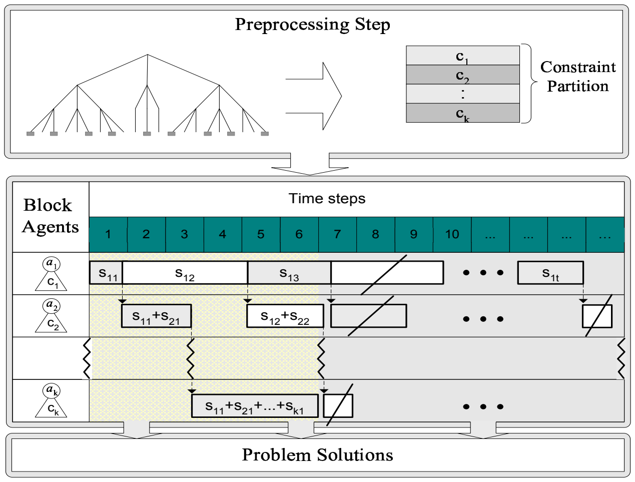
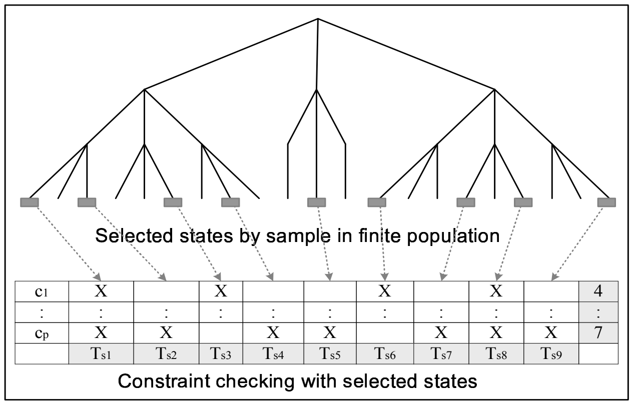


Figure 1: Multi-agent model

and well distributed states ($s$ is a polynomial function) in order to represent the entire population. As in statistic, the user selects the size of the sample ($s(n)$). Without loss of generality, we suppose a sample of ($n^2$) states. The preprocessing step studies how many states $st_i : st_i \leq n^2$ satisfy each constraint $c_i$. Thus, each constraint $c_i$ is labeled with $p_i$: $c_i(p_i)$, where $p_i = st_i/n^2$ represents the probability that $c_i$ satisfies the whole problem. Thus, in the preprocessing step the constraints are classified in ascending order of the labels $p_i$. Therefore, in the preprocessing step, the initial CSP is translated into an ordered CSP so that, it can be divided into a set of subproblems. Furthermore, this sample will be used by the stochastic local search algorithms to restart the search. Thus, the random states with lower evaluation value $T_{si}$ are firstly selected to restart the search.



Selected states by sample in finite population
c1
:
cp
X
4
7
Ts1
Ts2
Ts3
Ts4
Ts5
Ts6
Ts7
Ts8
Ts9
Constraint checking with selected states

Figure 2: Preprocessing step

The stochastic local search algorithm moves to the neighbor and restarts at the following lower evaluation value state when a local-minimum or a number of iterations (*Max-Flips*) is reached. In Figure 2, an example of the preprocessing step is shown. It can be observed that a sample of states is selected from the spanning tree. Each state is checked with the problem constraints and the evaluation value $T_{si}$ is stored to be used by the stochastic local search algorithms. Furthermore, each constraint $c_i$ is labeled in order to be classified from the most restricted one to the least restricted one. Thus, as we pointed in Figure 1, these ordered constraints are partitioned in $k$ blocks to divide the problem in $k$ interdependent subproblems. Each problem will be solved by an agent, called *block agent*, using a stochastic local search algorithm and the information derived from the preprocessing step.

## 3.2 Block Agent

A *block agent* is a cooperating agent with a set of properties. Without loss of generality, we make the following assumptions (Figure 3):

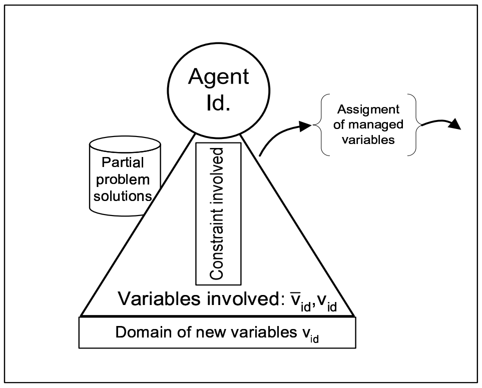

Agent Id.
Partial problem solutions
Constraint involved
Assigment of managed variables
Variables involved: $\overline{v}_{id}, v_{id}$
Domain of new variables $v_{id}$

Figure 3: Block agent

- There is a partition of the set of constraints $C \equiv \bigcup_{i=1}^{k} C_i$ generated by the preprocessing step, and each *block agent* $a_j$ has a block of constraints $C_j$.
- Each *block agent* $a_j$ knows 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 (by a stochastic local search algorithm) 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 agents $a_1, a_2, ..., a_{j-1}$.
- Each *block agent* $a_j$ knows the consistent partial states generated by the previous agents $a_1, a_2, ..., a_{j-1}$. Thus, agent $a_j$ knows assignments of variables included in sets: $\overline{v}_1, \overline{v}_2, ..., \overline{v}_{j-1}$.

*Block agents* cooperate to achieve a consistent state. *Block agent* 1 tries to find a consistent state of its partial problem. When it has a consistent partial state, it communicates this partial state to *block agent* 2. *Block agent* 2 studies the second set of more restricted constraints using the variable assignments generated by *block agent* 1. Meanwhile, agent 1 tries to find any other consistent partial state. So, each *block agent* $j$, using the variable assignment of the previous block agents $1, 2, .., j-1$, tries to find concurrently a more complete assignment. A consistent state is obtained when the last *block agent* $k$ finds a complete variable assignment.

**Example**: Let's look at the following example, (similar to the one presented in [Yokoo *et al.*, 1998]). 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).

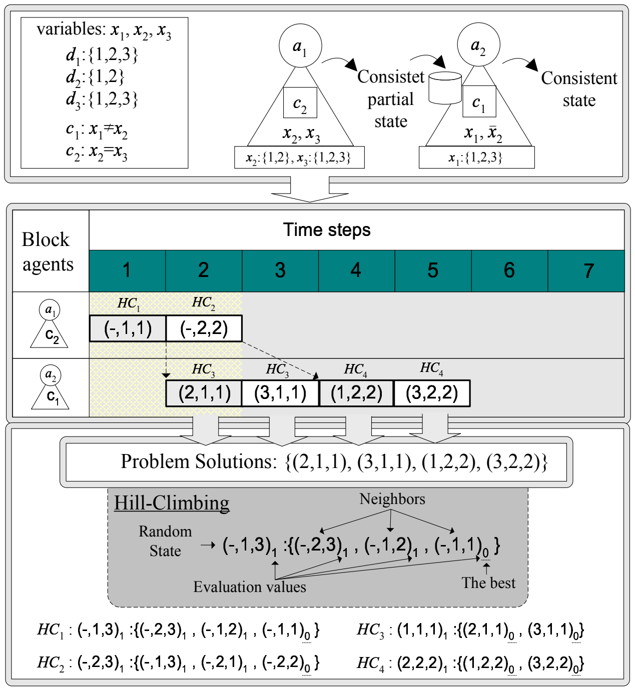


Figure 4: CSP solved by our model using Hill-Climbing.

Only two constraints are involved, so the constraint partition is straightforward, and only two blocks, with only one constraint, are considered. The first block is composed by constraint $c_2$ and the last block is composed by constraint $c_1$. This is due to the fact that constraint $c_2$ is more restricted than constraint $c_1$ because $c_2$ maintains two valid tuples: $(-,1,1)$ and $(-,2,2)$, while $c_1$ maintains four valid tuples: $(1,2,-),(2,1,-),(3,1,-)$ and $(3,2,-)$. So, *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 stochastic local search algorithm, and $a_2$ is responsible for assigning values to $x_1$. Figure 4 shows the behavior of our distributed model using Hill-Climbing. It can be observed that the time step 1 is only used by $a_1$ to generate a consistent partial state (-,1,1). Hill-Climbing starts at randomly generated state (-,1,3) with evaluation value (1) in $HC_1$. Its neighbors are (-,2,3),(-,1,2) and (-,1,1), with evaluation value

(1),(1) and (0), respectively. So, the neighbor with the best evaluation value is (-,1,1), which is a consistent partial state. Thus, $a_1$ sends a message to $a_2$ containing the consistent partial state $(-,1,1)$. In time step 2, both $a_1$ and $a_2$ work for finding a consistent state for their own problems. $a_2$ tries to find a value to $x_1$, knowing that $x_2$ and $x_3$ are fixed to (1,1). Hill-Climbing starts at randomly generated state (1,1,1) with evaluation value (1) in $HC_3$. Its neighbors are (2,1,1) and (3,1,1), with evaluation value (0) and (0), respectively. Furthermore, in time step 2, $a_1$ tries to find another consistent partial state for $x_2$ and $x_3$. Thus, in time step 2, $a_2$ finds a value to $x_1$, and a consistent state (2,1,1) is reached, while $a_1$ finds another consistent partial state $(-,2,2)$ in $HC_2$. If only one solution is required, the process is halted. However, if more solutions are required, the process continues in time steps 3,4 and 5 using Hill-Climbing $HC_3$ and $HC_4$. It can be observed (in time step 2) that our model allows agents to run concurrently to achieve consistent partial states.

Let's suppose that the domain of $x_1$ is $d_1 : \{1\}$. Then, the first consistent partial state, generated by $a_1$ (in time step 1), is (-,1,1). It does not go through a consistent state, because there is no value to $x_1$ (in time step 2) to satisfy the constraint $c_1$. Thus, the state $(1,1,1)$ is a local-minimum because this state is not a solution and the evaluation values of all its neighbors are larger than or equal to the evaluation value of this state. Then, $a_1$ restarts at another randomly generated state (-,2,3), (in time step 2), in which Hill-climbing finds a consistent partial state (-,2,2), that will go through a consistent state (1,2,2) by $a_2$ in the time step 3.

### 3.3 Application to Problems with *Hard* and *Soft* Constraints

The proposed distributed model can be also applied to problems with *hard constraints* and *soft constraints*. *Hard constraints* are conditions that must be satisfied, *soft constraints* however may be violated, but should be satisfied as much as possible.

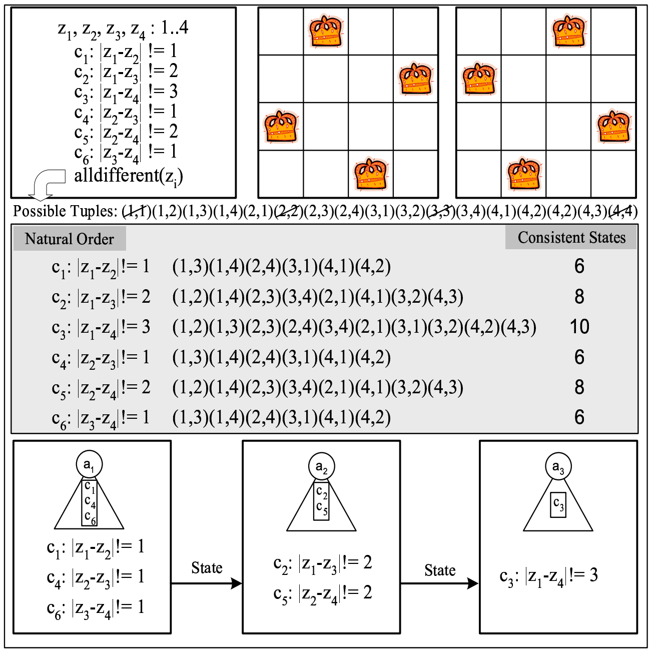
This type of problems can be easily managed as following:

- First, the preprocessing step studies normally the *hard constraints*, that is, it classifies the *hard constraints* so that the most restricted set of *hard constraints* are studied first.
- Later, the preprocessing step studies the *soft constraints*. However, in this case, it classifies the *soft constraints* so that the least restricted set of *soft constraints* are studied first.

In this way, the priority of constraints is: The most restricted *hard constraints*, the least restricted *hard constraints*, the least restricted *soft constraints* and the most restricted *soft constraints*. Thus, *hard constraints* must be satisfied and as many *soft constraints* as possible.

## 4 Evaluation

The n-queens problem is a classical search problem in the artificial intelligence area. The 4-queens problem is internally managed in Figure 5.



| Natural Order | | Consistent States |
|---|---|---|
| $c_1: \|z_1-z_2\|!= 1$ | (1,3)(1,4)(2,4)(3,1)(4,1)(4,2) | 6 |
| $c_2: \|z_1-z_3\|!= 2$ | (1,2)(1,4)(2,3)(3,4)(2,1)(4,1)(3,2)(4,3) | 8 |
| $c_3: \|z_1-z_4\|!= 3$ | (1,2)(1,3)(2,3)(2,4)(3,4)(2,1)(3,1)(3,2)(4,2)(4,3) | 10 |
| $c_4: \|z_2-z_3\|!= 1$ | (1,3)(1,4)(2,4)(3,1)(4,1)(4,2) | 6 |
| $c_5: \|z_2-z_4\|!= 2$ | (1,2)(1,4)(2,3)(3,4)(2,1)(4,1)(3,2)(4,3) | 8 |
| $c_6: \|z_3-z_4\|!= 1$ | (1,3)(1,4)(2,4)(3,1)(4,1)(4,2) | 6 |

Figure 5: The 4-queens problem in our distributed model.

Figure 5 shows the initial CSP and the possible solutions (2,4,1,3) and (3,1,4,2). The preprocessing step checks how many partial states (from a given sample: 16 tuples $\{(1,1),(1,2),\cdots,(4,3),(4,4)\}$) satisfy each constraint and classifies them afterwards. It can be observed that some constraints are more restricted than others. Constraints $c_1, c_4, c_6$ only satisfy 6 partial states, while constraints $c_2, c_5$ satisfy 8 partial states and constraint $c_3$ satisfies 10 partial states. Thus, the preprocessing step classifies the most restricted constraints first $c_1, c_4, c_6$. These constraints are enough to obtain both solutions:

$$(2,4,1,3) \leftarrow c_1 : (2,4), c_4 : (4,1), c_6 : (1,3)$$
$$(3,1,4,2) \leftarrow c_1 : (3,1), c_4 : (1,4), c_6 : (4,2)$$

In the 4-queens problem, the usefulness of our proposal becomes apparent when a stochastic local search algorithm is carried out to find a solution. The algorithm starts at a randomly generated state and the algorithm studies the evaluation value for each neighbor and constraint. In our distributed model, the evaluation value is only calculated in the first block of constraints, with the corresponding constraint check saving.

Following, two tables are presented to evaluate our distributed model in the $n-$queens problem. In Table 1, our objective is to find only one solution using Hill-Climbing. In Table 2, our objective is to find all solutions using *Generate&Test* and *Backtracking*.

The percentage of restart savings in the $n-$queens problem using Hill-Climbing with *Max-Flips=n* is presented in Table 1. It can be observed that the percentage is high for $n = 4$, that is, in the 4-queens problem the number of restarts is reduced by 57.62%. In the 13-queens problem the percentage of restarts is reduced by 91.5%. This restart saving is due to the random states with lower evaluation value $T_{si}$ are firstly selected to restart the search. Thus, Our model does not select random states but it selects the most appropriate ones.

Table 1: Percentage of restart savings in the $n-$queens problem using Hill-Climbing.

| Hill-Climbing | |
|---|---|
| *n-queens* | *Percentage of restart savings* |
| 4 | 57.62% |
| 5 | 70.41% |
| 6 | 75.84% |
| 7 | 80.04% |
| 8 | 83.32% |
| 9 | 85.51% |
| 10 | 87.27% |
| 13 | 91.50% |

The amount of constraint check savings in the n-queens problem using our distributed model with *Generate&Test* (GT+Dis) and *Backtracking* (BT+Dis) is presented in Table 2. In this case, our objective is to obtain all solutions in the $n-$queens problem. The results show that the amount of constraint check savings is significant in GT+Dis and BT+Dis. This is due to the fact that the preprocessing step classifies the constraints in ascending order (see Figure 5), so that the most restricted constraints have been checked first, and inconsistent states have been discarded earlier.

Table 2: amount of constraint check savings in the $n-$queens problem.

| | | **GT+Dis** | **BT+Dis** |
|---|---|---|---|
| *n-queens* | *Solutions* | *Constraint Check Savings* | *Constraint Check Savings* |
| 4 | 2 | 731 | 22 |
| 5 | 10 | 21336 | 240 |
| 6 | 4 | 446474 | 2406 |
| 7 | 40 | 14039727 | 24408 |
| 8 | 92 | $40.6 \times 10^7$ | 267982 |
| 9 | 352 | $13.6 \times 10^8$ | 3120302 |
| 10 | 724 | $402.1 \times 10^9$ | $3.9 \times 10^7$ |

## 5 Conclusions and future work

In this paper, we propose a distributed method for solving constraint satisfaction problems in which agents are committed to solving their partial problems by means of stochastic local search algorithms. They communicate partial solutions to other agents so that the most restricted set of constraints are studied first. Thus, inconsistencies can be found earlier and the number of constraint checks can be significantly reduced. In this way, hard problems can be solved more efficiently, especially problems where the number of constraints is large.

As future work, we are working on a distributed model in which *block agents* can dynamically interchange constraints, depending on the evaluation values, so that the preprocessing step can be removed and block agents can carry out this constraint partition.

## References

[Bartak, 1999] Roman Bartak. Constraint Programming: In Pursuit of the Holy Grail. In *Proceedings of WDS99 (invited lecture), Prague, June 1999.*

[Glover, 1986] Fred Glover. *Future paths for Integer Programming and Links to Artificial Intelligence*. Computers and Operations Research, 5:533–549, 1986.

[Hogg, 1993] Tad Hogg, and Colin P. Williams. *Solving the Really Hard Problems with Cooperative Search*. Proc. of AAAI93, 231–236, 1993.

[Kirkpatrick *et al.*, 1983] Scott Kirkpatrick, C. Gelatt, and M. Veechi. *Optimization by Simulated Annealing*. Science, 220:671–681, 1983.

[Kumar, 1987] Vipin Kumar. *DepthFirst Search*. In Encyclopedia of Artificial Intelligence, 2:1004-1005, 1987.

[Kumar, 1992] Vipin Kumar. *Algorithms for Constraint Satisfaction Problems: a Survey*. Artificial Intelligence Magazine, 1:32–44, 1992.

[Minton *et al.*, 1992] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 52:161-205, 1992.

[Sadeh, 1990] Nomal Sadeh. 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, 1990.

[Salido, 2003] Miguel A. Salido, and Federico Barber. A Constraint Ordering Heuristic For Scheduling Problems. *To appear in Proc. of 1st International Conference on Scheduling : Theory and Applications*, 2003.

[Selman, 1992] Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, 440–446, 1992.

[Sosic, 1994] Rok Sosic, Jun Gu. Efficient local search with conflict minimization: A case study of the Nqueen problem. *IEEE Transactions on Knowledge and Data Engineering*, 6:661-668, 1994.

[Yokoo, 1995] Makoto Yokoo. 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, 1995.

[Yokoo *et al.*, 1998] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, Kazuhiro Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *Knowledge and Data Engineering*, 10:673–685, 1998.