Solving CSPs with Naming Games

Stefano Bistarelli[1,2,3] and Giorgio Gosti[3,4]

[1] Dipartimento di Scienze, Università "G. d'Annunzio" di Chieti-Pescara, Italy
bista@sci.unich.it
[2] Institute of Informatics and Telematics (IIT-CNR) Pisa, Italy
stefano.bistarelli@iit.cnr.it
[3] Dipartimento di Matematica e Informatica, Università degli Studi di Perugia,
giorgio.gosti@dipmat.unipg.it.
[4] Institute for Mathematical Behavioral Sciences, University Of California, Irvine.

Abstract. Constraint solving problems (CSPs) represent a formalization of an important class of problems in computer science. We propose here a solving methodology based on the naming games. The naming game was introduced to represent N agents that have to bootstrap an agreement on a name to give to an object. The agents do not have a hierarchy and use a minimal protocol. Still they converge to a consistent state by using a distributed strategy. For this reason the naming game can be used to untangle distributed constraint solving problems (DCSPs). Moreover it represents a good starting point for a systematic study of DCSP methods, which can be seen as further improvement of this approach.

1 Introduction

The goal of this research is to generalize the naming game model in order to define a distributed method to solve CSPs. In the study of this method we want to exploit the power of distributed calculation, by letting the CSP solution emerge, rather than being the conclusion of a hierarchical search.

In DCSP protocols we design a distributed architecture of processors, or more generally a group of agents, to solve a CSP instantiation. In this framework we see the problem as a dynamic system and we set the stable states of the system as one of the possible solutions to our CSP. To do this we design each agent so that it will move towards a stable local state. The system is called "self-stabilizing" whenever the global stable state is obtained starting from a local stable state [4]. When the system finds the stable state the CSP instantiation is solved. A protocol designed in this way is resistant to damage and external threats because it can react to changes in the problem instance.

It is important to notice the fundamental difference with the Distributed CSP Algorithms designed by Yokoo [13], and Sipper [11]. Yokoo addresses three fundamental types of distributed CSP Algorithms: Asynchronous Backtracking, Asynchronous Weak-commitment Search, and Distributed Breakout Algorithm. Although these algorithms share the propriety of being asynchronous they require a pre-agreed agent/variable ordering. The algorithm that is presented in this paper does not need this initial condition, and despite the analogies with a distributed local search algorithm, the solution search is based on the competition between the domains of agents, in which by domain we define a set of agents in a local solution. Furthermore, the domain competition dynamics is driven by the agents change in the local solution and by the agent option to belong to more then one partial solution. Sipper defines the Non-uniform Cellular Automata as a distributed agent system immersed in a discrete time and space lattice, in which each agent is able to evolve a specific deterministic behavior. His approach to distributed CSP "[...] is one in which a grid of rules locally co-evolves to perform a given task", such as finding the solution. Besides, in our algorithm the agents have a predefined behavior, although this behavior is determined by random variables, such as the drawn speaker and the drawn assignment.

The algorithm described in this paper uses a central scheduler that randomly draws the speakers, this may be interpreted as a "central orchestrator" scheme, although we evince that this central scheduler has no information on the CSP instance, and has no pre-determined agent/variable ordering. Moreover, if we consider the case in which there is no central scheduler, and the agents have a fix probability P to speak at a certain turn t, then we see that if this probability satisfies the relation $P << 1/N$, the probability that there be more then one speaker at the time t will be approximately equal to zero. Under these conditions our system can be seen as an approximation of a distributed system with no central scheduler.

In Section 2 we illustrate the naming game formalism and we make some comparisons with the distributed CSP (DCSP) architecture. Then we describe the language model that is common to the two formalizations and introduce an interaction scheme to show the common framework. At last we state the definition of Self-stabilizing system [4].

In Section 3 we explicitly describe our generalization and formalize the protocol that our algorithm uses and test it on different CSPs. Moreover, for particular CSPs instantiations we analytically describe the multi-agent algorithm evolution that makes the system converge to the solution.

2 Background

2.1 The distributed constraint satisfaction problem (DCSP)

In CSPs we consider a set of N variables $x_1, x_2, \ldots, x_N$, their definition domains $D_1, D_2, \ldots, D_N$ and a set of constraints on the values of these variables. Solving the CSP means finding a at least one of the CSP solutions. A CSP solution is a particular assignment tuple $\overline{X} = (\overline{x_1}, \overline{x_2}, \ldots, \overline{x_N})$ for the variables $x_1, x_2, \ldots x_N$ that satisfy all the constraints C.

In the DCSP [13], the variables of the CSP are distributed among the agents. These agents are able to communicate between themselves and know all the constraint predicates that are relevant to their own variables. The agents through interaction find the appropriate values to assign to the variables and solve the CSP.

2.2 Introduction to Naming Games

The naming games [10, 1, 9, 6] describe a set of problems in which a number N of agents bootstrap a commonly agreed name for one or more objects. Each naming game is defined by an interaction protocol. There are two important aspects of the naming game: the agents randomly interact and use simple deterministic or probabilistic rules to update their state; the agents converge to a consistent state in which all the objects of the set have a uniquely assigned name, by using a distributed social strategy.

Generally, two agents are randomly extracted at each turn to perform the role of the speaker and the listener (or hearer as used in [10, 1]). The interaction between the speaker and the listener determines the agents' update of their internal state. DCSP and the naming game share a variety of common features [5], moreover, we will show in Section 3 that the naming game can be seen as a set of particular DCSP instances.

2.3 The Communication Model

In this framework we define a general model that describes the communication procedures between agents both in naming games and in DCSPs. The communication model consists of N agents (also called processors) arranged in a network.

We will use a central scheduler that at each turn randomly extracts the agents that will be interacting. Two agents connected by an edge in the network are neighbors. We define a broadcast register in which only the speaker i can write and can be read by all the neighboring listeners. At each interaction the speaker broadcasts the same variable assignment (word) b_s to all the neighbors by assignings the value b_s to the broadcast register. For each edge of the *communication graph* we allocate a register on which the listener can upload the communication outcome feedback f_{ij} using a predetermined signaling system.

The interaction scheme can be represented in three steps:

1. *Broadcast* The speakers broadcast information related to the proposed assignment for the variable;
2. *Feedback* The listeners feedback the interaction outcome expressing some information on the speaker assignment by using a standardized signal system;
3. *Update* The speakers and the listeners update their state regarding the overall interaction outcome.

In this scheme we see that at each turn the agents update their state. The update reflects the interaction they have experienced. We have presented the general interaction scheme, wherein each naming game and DCSP algorithm has its own characterizing protocol.

2.4 Self-Stabilizing Algorithms

A self-stabilizing protocol [4] has some important properties. First, the global stable states are the wanted solutions to our problem. Second, the system configurations are divided into two classes: legal associated to solutions and illegal associated to non-solutions. We may define the protocol as self-stabilizing if in any infinite execution the system finds a legal system configuration that is a global equilibrium state. Moreover, we want the system to converge from any initial state. These properties make the system fault tolerant and able to adapt its solutions to changes in the environment.

To make a self-stabilizing algorithm we program the agents of our distributed system to interact with the neighbors. The agents through these interactions update their state trying to find a stable state in their neighborhood. Since the algorithm is distributed many legal configurations of the agents' states and its neighbors' states start arising sparsely. Not all of these configurations are mutually compatible and so form inconsistent legal domains. The self-stabilizing algorithm must find a way to make the global legal state emerge from the competition between these domains. Dijkstra [4] and Collin [3] suggest that an algorithm designed in this way can not always converge and a special agent is needed to break the system symmetry. In this paper we will show a different strategy based on the concept of random behavior and probabilistic transition function that we will discuss in the next sections.

3 Generalization of the naming game to solve DCSP

In the naming game, the agents want to agree on the name given to an object. This can be represented as a DCSP, where the name proposed by each agent is the assignment of the CSPs variable controlled by the agent, and where an equality constraint connects all the variables. On the other hand, we can generalize the naming game to solve DCSPs.

We attribute an agent to each variable of the CSP as in [13]. Each agent $i = 1, 2, \dots N$, names its own variable x_i in respect to the *variable domain* D_i. We restrict the constraints to binary relation C_{ij} between variable x_i and x_j. If $x_i C_{ij} x_j$ is true, then the values of the variables x_i and x_j are consistent. We define two agents as neighbors if their variables are connected by a constraint.

The agents have a *list*, which is a continuously updated subset of the domain elements. The difference between the *list* and the domain is that the domain is the set of values introduced by the problem instance, and the *list* is the set of variable assignments that the agent subjectively forecasts to be in the global solution, on the basis of its past interactions. When the agent is a speaker, it will refer to this *list* to choose the value to broadcast and when it is a listener, it will use this *list* to evaluate the speaker broadcasted value.

At turn $t = 0$ the agents start an empty *list*, because they still do not have information about the other variable assignments. At each successive turn $t = 1, 2, \dots$ an agent is randomly drawn by the central scheduler to cover the role of the speaker, and all its neighbors will be the listeners. The communication between the speaker s and a single listener l can be a *success*, a *failure*, or a *consistency failure*. At the end of the turn all the listeners feedback to the speaker the *success*, the *failure*, or the *consistency failure* of the communication.

If all the interaction sessions of the speakers with the neighboring listeners are successful, we will have a *success update*. If there was one or more *consistency failure* we will have a *consistency failure update*. If there was no *consistency failure* and just one or more *failures*, there will be a *failure update*.

The interaction, at each turn t, is represented by this protocol:

1. *Broadcast*. If the speaker *list* is empty it extracts an element from its *variable domain* D_s, puts it in its *list* and communicates it to the neighboring listeners. Otherwise, if its *list* is not empty, it randomly draws an element from its *list* and communicates it to the listeners. We call this the broadcast assignment d_s.
2. *Feedback*. Then the listeners calculate the consistent assignment subset K and the consistent domain subset K':
   - *Consistency evaluation*. Each listener uses the constraint defined by the edge, which connects it to the speaker, to find the consistent elements d_l to the element d_s received from the speaker. The elements d_l that it compares with d_s are the elements of its *list*. These consistent elements form the *consistent elements subset* K. We define $K = \{d_l \in \mathit{list} | d_s C_{sl} d_l\}$. If K is empty the listeners compare each element of its variable domain D_l with the element d_s, to find a *consistent domain subset* K'. We define $K' = \{d_l \in D_l | d_s C_{sl} d_l\}$.

   The *consistent elements subset* K and the *consistent domain subset* K' determine the following feedback:
   - *Success*. If the listener has a set of elements d_l consistent to d_s in its *list* (K is not empty), there is a *success*.
   - *Consistency failure*. If the listener does not have any consistent elements d_l to d_s in its *list* (K is empty), and if no element of the listener *variable domain* is consistent to d_s (K' is empty), there is a *consistency failure*.
   - *Failure*. If the listener does not have any d_l consistent elements to d_s in its *list* (K is empty), and if a non empty set of elements of the listener *variable domain* are consistent to d_s (K' is not empty), there is a *failure*.
3. *Update*. Then we determine the overall outcome of the speaker interaction on the basis of the neighbors' feedback:
   - *Success update*. This occurs when all interactions are successful. The speaker and the neighbors cancel all the elements in their *list* and update it in the following way: the speaker stores only the successful element d_s and the listener stores the consistent elements in K.
   - *Consistency failure update*. This occurs when there is at least one *consistency failure* interaction. The speaker must eliminate the element d_s from its *variable domain* (this can be seen as a step of local consistency pruning). The listeners do not change their state.
   - *Failure update*. This occurs in the remaining cases. The speaker does not update its list. The listeners update their *lists* by adding the set K' of all the elements consistent with d_s to the elements in the *list*.

We can see that in the cases where the constraint $x_i C_{ij} x_j$ is an equality, the subset of consistent elements to x_i is restricted to one assignment of x_j. For this assignment of the constraint $x_iC_{ij}x_j$ we obtain the naming game as previously described. Our contribution to the interaction protocol is to define K and K'. As a matter of fact, in the naming game the consistent listener assignment d_l to the speakers assignment d_s is one and only one ($d_l = d_s$), unlike the CSP instances, in which there may be more consistent listener assignment d_l for each speakers assignment d_s. This observation is fundamental to solve general CSP instances. Moreover, in the naming game there is only one speaker and one listener at each turn, but we observed that under this hypothesis the agents were not always able to enforce local consistency (e.g.: graph coloring of a completely connected graph). Thus we had to extend the interaction to all the speaker neighbors and let all the neighbors be listeners.

At each successive turn the system evolves through the agents' interactions in a global equilibrium state. In the equilibrium state all the agents have only one element for their variable and this element must satisfy the constraint $x_iC_{ij}x_j$ with the element chosen by the neighboring agents. Clearly, this state is a solution of the CSP instance, and we call this a *global consensus state*. Once in this state the interactions are always successful. The probability to transit to a state different from the *global consensus state* is zero, for this reason the *global consensus* state is referred to as an absorbing state. We call the turn at which the system finds global consensus *convergence turn* t_{conv}.

Simple Algorithm Execution The N-Queens Puzzle is the problem of placing N queens on a $N \times N$ chessboard without having them be mutually capturable. This means that there can not be two queens that share the same row, column, or diagonal. This is a well known problem and has been solved linearly by specialized algorithms. Nevertheless it is considered a classical benchmark and we use it to show how our algorithm can solve different instances. To reduce the search space we assign the variables of a different column to each queen. We can do this because, if there were more then one queen in a column, they would have been mutually capturable. In this way each of our agents will have as its domain the values of a distinct column and all the agents will be mutually connected by an edge in the graph representing all the constraints. In the example we show a N-Queens Puzzle with $N = 4$. Each agent (queen) is labeled after its column with numbers from zero to three from left to right. The rows are labeled from the bottom with numbers from zero to three. In Figure 1 we show how the algorithm explores the solution space randomly and how it evolves at each turn t. We write the speaker s that is extracted at each turn and its broadcasted value b. Then we write the listeners l, their respective K, K', and the feedbacks. At the end we write the updates. The picture represents graphically the evolution of the agents' list at the end of each turn.

At turn $t = 1$ (see Fig.1) speaker $s = 3$ is randomly drawn. The *variable* controlled by this speaker is the position of the queen on the last column of the chessboard. The speaker has an empty *list*, hence it draws from its *variable domain* the element $d_s = 3$ which corresponds to the highest row of its column. The speaker add this new element to its *list*. Since all the agents are connected, all

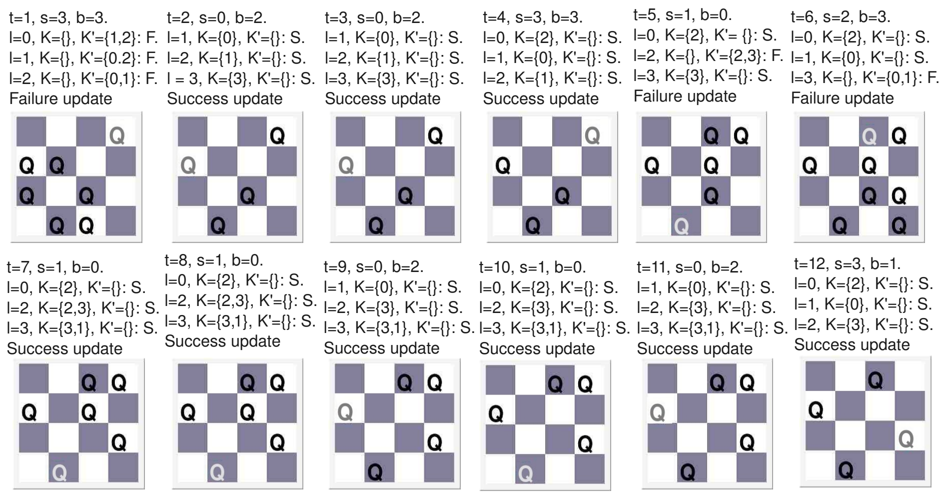

Fig. 1. *Single algorithm run for the N-Queens Puzzle with $N = 4$.*

the agents apart from the speaker are listeners. Their *lists* are empty therefore K is empty. Thus they compute K' from the *variable domain*. The listeners feedback failure, thus the speaker replies with a failure update. The listeners add the elements of their respective K' to their *lists*. The picture in fig.1 shows the elements in the agents' *lists* at the end of the turn. At turn $t = 2$ speaker $s = 0$ is drawn and it broadcasts the element $d_s = 2$. All the listeners have a consistent element in their list, therefore, their Ks are not empty, and they feedback a success. The listeners delete their *lists* and add the elements in their K. At turn $t = 3$ the speaker $s = 0$ speaks again and broadcasts the same element $d_s = 2$. Therefore, the listener computes the same Ks as before, and feedbacks a success. Thus we have a success update, but since the Ks are the same the system does not change. At turn $t = 4$ the speaker $s = 0$ is drawn and broadcasts the same variable $d_s = 3$ that it had broadcasted at the first turn. Since all the elements in the listeners' *lists* are still consistent to this broadcast, the algorithm has a success update and the agents' *lists* remain the same. At turn $t = 5$ a new speaker is drawn $s = 1$, it broadcasts $d_s = 0$. The listeners zero and three have a consistent element to this broadcast, therefore their K is not empty. Furthermore, listeners two has no consistent elements to put in K, and finds the rows two and three from its *variable domain* to be consistent to this broadcast. The overall outcome is a failure and thus we have a failure update. The listeners zero and two have empty K's so they do not change their *lists* and listener two adds two new elements to its lists. At turn $t = 6$ agent two speaks and broadcasts the element $d_s = 3$. Agent three does not have consistent elements to this broadcast and thus feedbacks a failure. Then we have a failure update and agent two adds two elements to its *list*. At turn $t = 7$ agent one speaks and broadcasts the element $d_s = 0$. All the listeners have consistent elements therefore their Ks are not empty. We get a success update. The agents two and three both delete an element from their *lists* which is not consistent to the speaker broadcast. At turn $t = 8$ agent one speaks again and broadcasts the same element $d_s = 0$. The system is unchanged. At turn $t = 9$ agent zero speaks and broadcasts the element $d_s = 2$. All listeners have consistent elements, therefore, there is a success update. Listener two deletes an element that was not consistent with the speaker broadcast. At turn $t = 10$ agent one speaks and broadcasts the element $d_s = 0$. All listeners have consistent elements to this broadcast, there is a success update, and the system is unchanged. At turn $t = 11$ agent zero speaks and broadcasts the element $d_s = 2$. All listeners have consistent elements to this broadcast, there is a success update, and the system is unchanged. At turn $t = 12$ agent three speaks and broadcasts the element $d_s = 1$. All listeners have consistent elements to this broadcast, there is a success update. Since the speaker had a different element in its list from the broadcasted element $d_s = 1$, he deletes this other element from its *list*. At this point all the elements in the agents *lists* are mutually consistent. Therefore, all the successive turns will have success updates and the system will not change any more. The system has found its global equilibrium state that is a solution of the puzzle we intended to solve.

3.1 Difference with Prior Self-Stabilizing DCSPs

In the prior DCSPs the agent is a finite state-machine. Furthermore, its evolution in time is represented by a *transition function*, which depends on its state and its neighbors' states in the current turn. Let the local state s_i be the union of the agent state a_i and its neighbors' state. The *transition function* determines the communication outcome, and the agents states update once we know the local state s_1. Moreover, the *transition function* of an agent, which state is a_i, in the local state s_i is one and one only, and we can forecast exactly its next state a_{i+1} and the next local state s_{i+1}.

Let uniform protocols be distributed protocols, in which all the nodes are logically equivalent and identically programmed. It has been proved that in particular situations uniform self-stabilizing algorithms can not always solve the CSPs ([3]), especially if we consider the ring ordering problems. In ring ordering problems we have N numbered nodes $\{n_1, n_2, \ldots, n_N\}$ ordered on a cycle graph. Each node has a variable, the variable assignment of the $i+1$-th node n_{i+1} is the consecutive number of the variable assignment of the i-th node n_i in modulo N. The *variable domain* is $\{0, 1, \ldots, N-1\}$ and every link has the constraint $\{n_i = j, n_{i+1} = (j+1) \mod N | 0 \leq j \leq N\}$. Dijkstra [4] and Collin [3] propose dropping the uniform protocol condition to make the problem solvable.

Our protocol overcomes this by introducing a random behavior *probabilistic transition function* $T(P_s, P_l, P_b)$. In our algorithm the agent state is defined by an array that attributes a zero or a one to each element of the agent domain. The array element will be zero if the element is not in the *list*, and one if the element is in the list. This array determines a binary number a_i, which defines the state of the agent. If we know the states of all the agents, the transition of each agent from state a_i to a_j is uniquely determined once we know the agent that will be the speaker, the agents that will be the listeners, and the element that will be broadcasted by the speaker (Fig.2(a)).

Since the speaker will be chosen randomly, we can compute the probability for each agent being a speaker P_s. From this information, since we know the underlying graph and that all its neighbors will be listeners, we can compute the probability for each agent to be a listener P_l. Knowing the speaker state, we can compute the probability for each element to be broadcast P_b. At this point we may be able to compute the *probabilistic transition function* $T(P_s, P_l, P_b)$, which will depend on the probabilities that we have just defined (Fig.2(b)).

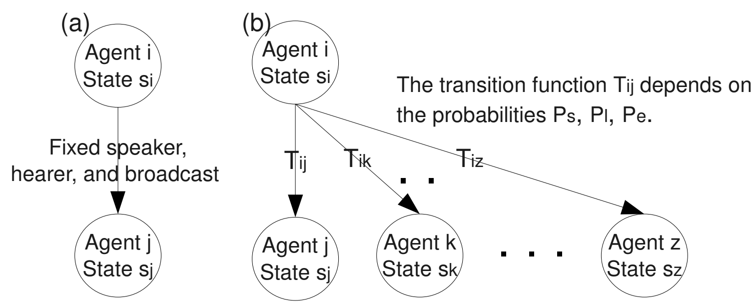

Fig. 2. *(a) Shows that once we determine the speaker state, the broadcast, and the listener state we are able to determine the speaker and listeners' transitions. (b) Shows that since we have the speaker probability P_s, the broadcast probability P_b, and the listener probability P_l we can determine the* probabilistic transition function $T(P_s, P_l, P_b)$.

In this setting the agent state a_t at turn t can now be represented by a discrete distribution function and the *transition function* is now a Markovian Chain, the arguments of which are the transition probabilities p_j between the local states s_i and s_j. Thus we speak of a *probabilistic transition function*, which represents the probability of finding the system in a certain state s_j starting from s_i at time t. This behavior induces the algorithm to explore the state space randomly, until it finds the stable state that represents our expected solution. In the following plot we show the *convergence turn* t_{conv} scaling with the size N of the ring ordering problem. We average the *convergence turn* t_{conv} on ten runs of our algorithm for a set size N. Then we plot this point in a double logarithmic scale to evince the power law exponent of the function. We found that $t_{conv} \propto N^{3.3}$.

3.2 Analytical Description

In this section we are going probabilistically analyze how our algorithm solves graph coloring problems for the following graph structures: path graph and completely connected graph. These are simple limiting cases that help us to picture

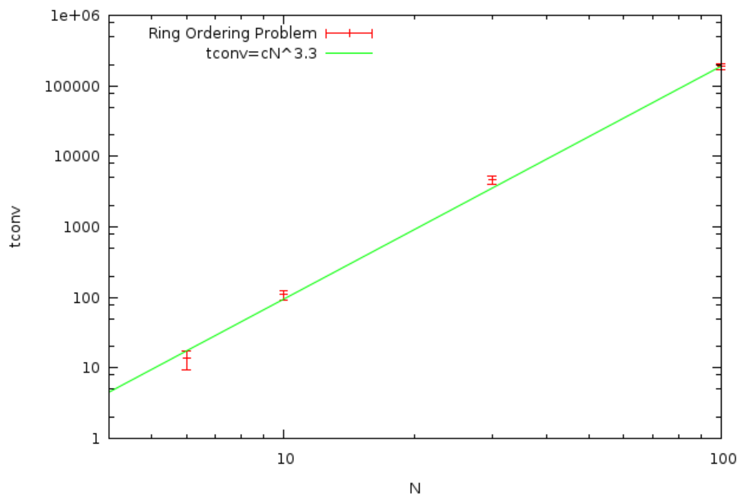

Fig. 3. *The plot shows the ring ordering problem with N nodes, we see for the convergence turn t_{conv} the scaling proportion: $t_{conv} \propto N^{3.3}$.*

how our algorithm evolves in more general cases. In graph coloring the variable domain elements represent colors, and the edges between nodes represent inequality constraints.

Path graph The way our algorithm solves a path graph coloring instance can be described through analytical consideration; similar observations can then be extended to the cycle graph. The system dynamics are analogous to the naming game on a one dimensional network [2]. To each node of the path graph we attribute a natural number in increasing order, from the first node of the path, to which we attribute 1, to the last node, to which we attribute N. The path graph coloring needs minimum two colors to be solved, therefore we imposed the agents' variable domain to two colors. We see that there are only two solutions: with odd number nodes of one color and even number nodes of the second color in one solution; and inverted in the other solution. At the beginning, when $t < N/3$, the system is dominated by new local consistent nodes' neighborhoods, which emerge sparsely and propagate to the connected nodes. At each turn a speaker is randomly drawn. This speaker has an empty *list* and it has to draw the assignment from the two element *variable domain*. In this way it selects one of the two solutions to which it starts to belong. Broadcasting its assignment to its neighbors it makes them converge on the same solution. In this way a small consistent domain of three agents that agree on the final state emerges.

Since the speakers are chosen by the scheduler randomly, after some time, $t > N/3$, all the agents have been speakers or listeners at least once. Thus we find approximately $N/3$ domains dispersed in little clusters of generally three agents. Each of these domains belong to one of the two final solutions.

At this point the domains start to compete. Between two domains we see an overlapping region appear. This region is constituted by agents that have more than one element in their lists. We can refer to them as undecided agents that belong to both domains, since the agents are on a path graph this region is linear. By probabilistic consideration we can see that this region tends to enclose less than two agents. For this reason we define the region that they form as a *border*, for a path graph of large size N the border width is negligible. So we approximate that only one agent is within this *border*. Under this hypothesis we can evaluate the evolution of the system as a diffusion problem, in which the borders move in a random walk on the path graph. When a domain grows over another domain, the second domain disappears. Thus the relation between the cluster growth and time is $\Delta x \propto (\frac{t}{\xi})^{1/2}$, where ξ is the time needed for the random walk to display a deviation of ± 1 step from its position. The probability that the border will move one step right or left on the path graph is $\propto 1/N$, proportional to the probability that an agent on the border or next to the border is extracted. Thus we can fix the factor $\xi \propto 1/N$. Since the lattice is N steps long we find the following relation for the average convergence turn $t_{conv} \propto N^3$. The average convergence turn t_{conv} is the average time in which the system finds global consistency. Let the average convergence turn t_{conv} be defined as the sum of the weighted convergence turns of all the possible algorithm runs, where the weights are the probabilities of the particular algorithm run.

Completely connected graph Since all the variables in the graph coloring of a completely connected graph are bound by a inequality constraint, these variables must all be different. Thus the agent domain must have N elements.

At the beginning all the agents' lists are empty. The first speaker chooses a color, and since all the agents are neighbors, it communicates with all of them. The listeners place in their *lists* the colors from the *variable domain* consistent with the color picked by the speaker. In the following turns the interactions are always successful. Two cases may be observed: the speaker has never spoken so it selects a color from its *list* and it broadcasts it, and deletes all the other elements, while the listeners cancel the broadcasted color from their lists; or the speaker has already spoken once, so there are no changes in the system because it already has only one color and all the other agents have already deleted this assignment. Since at each turn only one agent is a speaker, to let the system converge all the agents have to speak once.

Let N be the number of agents and t_{conv} the system convergence time. Let t_i be the turn t in which i agents have spoke one or more times, and $N - i$ have never spoke. Let s_i be the waiting time between the turn t_{i-1} and the turn t_i. It is useful to notice that we can represent t_{conv} by $t_{conv} = \sum_{i=1}^{N} s_i$ and s_i is a geometric random variable. We use this probability to compute the weighted average turn at which the system converges. This will be the weighted average convergence turn t_{conv}. Moreover, we use the property that the expected value of the sum of two random variables is equal to the sum of the respective expected values, $E[t_{conv}] = \sum_{i=1}^{N} E[s_i]$. Since s_i is a geometric random variable, $E[s_i] = i/(N-i)$. Thus:

$$t_{conv} = N \sum_{i=1}^{N-1} \frac{1}{N-i} = N \sum_{j=1}^{N-1} \frac{1}{j} \sim N \log(N-1) \tag{1}$$

This, as we stated above corresponds to the time of convergence of our system, when it is trying to color a completely connected graph.

3.3 Algorithm Test

We have tested the above algorithm in the classical CSP problems, graph coloring. We plotted the graph of the convergence turn t_{conv} scaling with the number N of the CSP variables, each point was measured by ten runs of our algorithm. We considered four types of graphs for the graph coloring: path graphs, cycle graphs, completely connected graphs, and Mycielsky graphs.

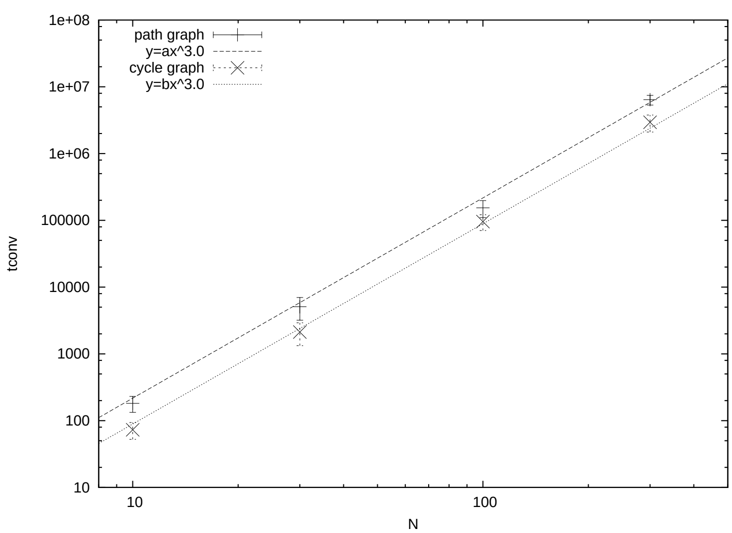

Fig. 4. *The plot shows the graph coloring in the case of path graphs and special 2-colorable cycle graphs with 2 colors. The convergence turn t_{conv} of the path graphs and cycle graphs exhibit a power law behavior $t_{conv} \propto N^{3.0}$. The cycle graph exhibits a faster convergence. The points on this graph are averaged on ten algorithm runs.*

Graph Coloring In the study of graph coloring we presented four different graph structures:

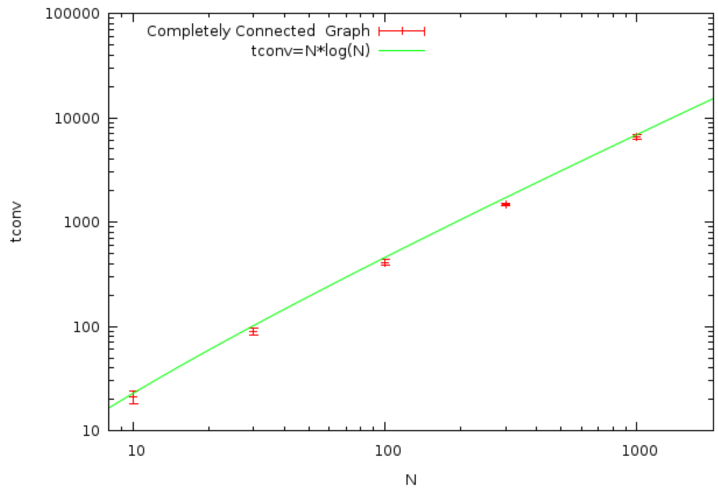

Fig. 5. *The plot shows the graph coloring in the case of a completely connected graph with N colors: in this case we find that the convergence turn is $t_{conv} \sim N\log(N)$. The points on this graph are averaged on ten algorithm runs.*

– path graphs
– cycle graphs
– completely connected graph
– Mycielsky graphs.

In the study of the path graph and the cycle graph we have restricted ourselves to the $2-chromatic$ cases: all the path graphs and only the even number node cycle graphs. Thus we imposed the agent variable domain to two colors. In this context the convergence turn t_{conv} of the path graph and the cycle graph exhibit a power law behavior $t_{conv} \propto N^{3.0}$. The cycle graph exhibits a faster convergence (Fig. 4). We see from these measurements that the power law of the convergent time scaled with the number of nodes N is compatible with our analytical considerations.

The graph coloring in the case of a completely connected graph always needs at least N colors: in this case we find that the convergence turn is $t_{conv} \propto N\log(N)$ (Fig. 5).

The Mycielski graph [8] of an undirected graph G is generated by the Mycielski transformation on the graph G and is denoted as $\mu(G)$. Let the N number of nodes in the graph G be referred to as $v_1, v_2, \ldots, v_N$. The Mycielski graph is obtained by adding to graph G $N+1$ nodes: N of them will be named $u_1, u_2, \ldots, u_N$ and the last one w. We will connect with an edge all the nodes $u_1, u_2, \ldots, u_N$ to w. For each existing edge of the graph G between two nodes v_i and v_j we include an edge in the Mycielski graph between v_i and u_j and between u_i and v_j.

M_i	N	E	k optimal coloring	t_{conv}
M_4	11	20	4	32 ± 2
M_5	23	71	5	170 ± 20
M_6	47	236	6	3300 ± 600
M_7	95	755	7	$(1.1 \pm 0.2) \cdot 10^6$

Table 1. Convergence turn t_{conv} of the Mycielski graph coloring. M_i is the Mycielski graph identification, N is the number of nodes, E is the number of edges, k the optimal coloring, and t_{conv} the convergence turn.

The Mycielski graph of graph G of N nodes and E edges has $2N + 1$ nodes and $2E + N$ edges.

Iterated Mycielski transform applications starting from the null graph, generates the graphs $M_i = \mu(M_{i-1})$. The first graphs of the sequence are M_0 the null graph, M_1 the one node graph, M_2 the two connected nodes graph, M_3 the five nodes cycle graph, and M_4 the Grötzsch graph with 11 vertices and 20 edges (see Fig. 6). The number of colors k needed to color a graph M_i of the Mycielski sequence is, $k = i$ ([8]).

These graphs are particularly difficult to solve because they do not possess triangular cliques, moreover, they have cliques of higher order and the coloring number increases each Mycielski transformation ([7]). We ran our algorithm to solve the graph coloring problem with the known optimal coloring. Table 1 shows for each graph of the Mycielski sequence M_i, the number of nodes N, the number of edges E, the minimal number of colors needed k and the convergence turn t_{conv} of our algorithm.

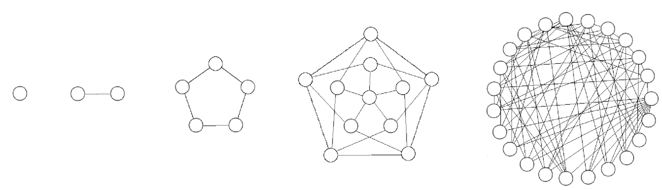

Fig. 6. *Mycielski graph sequence M_1, M_2, M_3, M_4, and M_5[12].*

4 Conclusions and Future Work.

Our aim is to develop a probabilistic algorithm able to find the solution of a CSP instance. In the study of this method we are trying to fully exploit the power of distributed calculation. To do this we generalize the naming game algorithm, by letting the CSP solution emerge, rather than being the conclusion of a sequence of statements. As we saw in Subsection 3.2, our algorithm is based on the random exploration of the system state space. Our algorithm travels through the possible states until it finds the absorbing state, where it stabilizes. These ergodic features guarantee that the system has a probability equal to one to converge for long times. Unfortunately this time, depending on the particular CSP instance, can be too long for practical use. This goal where achieved through the union of new topics addressed in statistical physics (the naming game), and the abstract framework posed by constraint solving.

In future work we will test the algorithm on a uniform random binary CSP to fully validate this method. We also expect to generalize the communication model to let more then one agent speak at the same turn. Once we have done this we can let the agents speak spontaneously without a central scheduler.

References

1. Baronchelli, A., Felici, M., Caglioti, E., Loreto, V., Steels, L.: Sharp Transition Toward Shared Vocabularies in Multi-Agent Systems. In: Journal of Statistical Mechanics, (2006).
2. Baronchelli, A., Dall'Asta, L., Barrat, A. and Loreto., V.: Topology Induced Coarsening in Language Games Language. In: Phys. Rev., E 73, (2006).
3. Collin, Z., Dechter, R., and Katz, S.: On the Feasibility of Distributed Constraint Satisfaction. In: Proceedings of the Twelfth International Joint Conference of Artificial Intelligence (IJCAI-91), (1991).
4. Dijkstra E.W.: Self-Stabilizing Systems in Spite of Distributed Control. In: Communications of the ACM, (1974).
5. Gosti, G.: Resolving CSP with Naming Games. In: 24th International Conference on Logic Programming, (2008).
6. Komarova, N. L, Jameson, K. A., Narens, L.: Evolutionary Models of Color Categorization Based on Discrimination. In: Journal of Mathematical Psychology, (2007)
7. Trick M.: Network Resources for Coloring a Graph. http://mat.gsia.cmu.edu/COLOR/color.html.
8. Mycielski, J.: Sur le coloriage des graphes. In: Colloq. Math. 3, (1955).
9. Nowak, M.A., Plotkin, J.B., Krakauer, J.D.: The Evolutionary Language Game. In: Journal of Theoretical Biology, (1999).
10. Steels, L.: Self-Organizing Vocabularies. In: Artificial Life V: Proceeding of the Fifth International Workshop on the Synthesis and Simulation of Living Systems (1996).
11. Sippers, M.: Co-evolving Non-Uniform Cellular Automata to Perform Computations. In: Physica D, 92:193-208, (1996).
12. Weisstein, E. W.: Mycielski Graph. In: MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/MycielskiGraph.html.
13. Yokoo, M., Katsutoshi H.: Algorithms For Distributed Constraint Satisfaction: A Review. In:Autonomous Agents and Multi-Agent Systems, Vol.3, No.2, pp.198-212, (2000).