ELSEVIER

Artificial Intelligence 136 (2002) 101–144

Artificial Intelligence

www.elsevier.com/locate/artint

Multi-agent oriented constraint satisfaction

Jiming Liu *, Han Jing, Y.Y. Tang

Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong

Received 31 May 2001; received in revised form 2 August 2001

Abstract

This paper presents a multi-agent oriented method for solving CSPs (Constraint Satisfaction Problems). In this method, distributed agents represent variables and a two-dimensional grid-like environment in which the agents inhabit corresponds to the domains of the variables. Thus, the positions of the agents in such an environment constitute the solution to a CSP. In order to reach a solution state, the agents will rely on predefined local reactive behaviors; namely, *better-move*, *least-move*, and *random-move*. While presenting the formalisms and algorithm, we will analyze the correctness and complexity of the algorithm, and demonstrate the proposed method with two benchmark CSPs, i.e., n-queen problems and coloring problems. In order to further determine the effectiveness of different reactive behaviors, we will examine the performance of this method in deriving solutions based on behavior prioritization and different selection probabilities. © 2001 Published by Elsevier Science B.V.

Keywords: Constraint satisfaction; Multi-agent; Reactive moving behaviors; Behavior prioritization; Behavior selection; Experimental validation

1. Introduction

1.1. CSPs

Many problems in Artificial Intelligence (AI) as well as in other areas of computer science and engineering can be translated into a certain type of *constraint satisfaction problem* (CSP) [19,29]. Some examples of such problems include: spatial and temporal planning, qualitative and symbolic reasoning, diagnostics, decision support, computational

* Corresponding author.
E-mail address: jiming@comp.hkbu.edu.hk (J. Liu).

0004-3702/01/$ – see front matter © 2001 Published by Elsevier Science B.V.
PII: S0004-3702(01)00174-6

linguistics, scheduling, resource allocation and planning, graph problems, hardware design and verification, configuration, real-time systems, and robot planning.

Definition 1.1. A *constraint satisfaction problem* (*CSP*) consists of:

(1) A finite set of variables, $\boldsymbol{X} = \{X_1, X_2, \ldots, X_n\}$.
(2) A domain set, containing a finite and discrete domain for each variable:

$$\boldsymbol{D} = \{D_1, D_2, \ldots, D_n\}, \quad \forall i \in [1, n],\ X_i \in D_i.$$

(3) A constraint set, $\boldsymbol{C} = \{C(R_1), C(R_2), \ldots, C(R_m)\}$, where each R_i is an ordered subset of the variables, and each constraint $C(R_i)$ is a set of tuples indicating the mutually consistent values of the variables in R_i.

Definition 1.2. The solution, S, for a CSP is an assignment to all variables such that the assignment satisfies all given constraints. Specifically,

(1) S is an ordered set, $S = \langle v_1, v_2, \ldots, v_n\rangle$, $S \in D_1 \times D_2 \times \cdots \times D_n$.
(2) $(\forall j \in [1, m])\ (\exists S' \subseteq S) \wedge (S' \in C(R_j))$ is `true`. S' is also an ordered set.

In this paper, we will focus our discussion on *binary CSPs* where each constraint is either unary or binary [19]. A binary CSP has the same definition as Definition 1.1, except $R_i = D_{i1} \times D_{i2}$. It is possible to convert a CSP with n-ary constraints to an equivalent binary CSP [19,34].

Let us now take a look at two typical CSP examples as follows:

Example 1.1. The n-queen problem is a classical CSP. It is generally regarded as a benchmark for testing algorithms and has attracted a lot of attentions in the CSP community [38]. This problem requires one to place n queens on an $n \times n$ chessboard, so that no two queens are in the same row, the same column, or the same diagonal. There exist solutions for the n-queen problems with n greater than or equal to 4 [2,38] (see Fig. 1). The equivalent CSP can be stated as follows:

$$\begin{aligned}
&\boldsymbol{X} = \{X_1, X_2, \ldots, X_n\}.\\
&\boldsymbol{D} = \{D_1, D_2, \ldots, D_n\}, \quad \forall i,\ D_i = [1, n].\\
&\boldsymbol{C} = \big\{C(R_u) \mid \forall i, j \in [1, n], C(R_u) = \{\langle b, c\rangle \mid b \in D_i,\ c \in D_j,\ b \neq c,\\
&\qquad\qquad i - j \neq b - c,\ i - j \neq c - b\}\big\}.
\end{aligned}$$

	O		
			O
O			
		O	

O: queen

Fig. 1. A solution for a 4-queen problem.

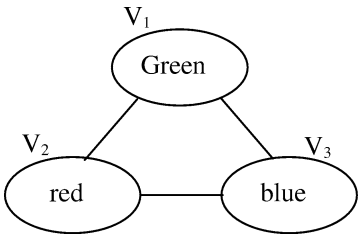

Fig. 2. An example solution for a coloring problem.

Example 1.2. The (vertex) coloring problem that is found in a variety of applications can readily be modeled as a CSP. In this problem, we need to color each vertex or node of a graph by a certain color (from a set of colors, suppose to be m colors), such that no two nodes incident to any edge have the same color. The equivalent CSP will represent each of the nodes in the graph into a variable. The domain of the variable corresponds to the given set of m colors. For each pair of nodes incident to an edge, there is a binary constraint between the corresponding variables that disallows identical assignments to these two variables. Here is an example: $\boldsymbol{X} = \{V_1, V_2, V_3\}$, $D_1 = \{green, red\}$, $D_2 = \{red, blue\}$, $D_3 = \{blue\}$, where constraints are: $V_1 \neq V_2$, $V_1 \neq V_3$, and $V_2 \neq V_3$, so $\boldsymbol{C} = \{\{\langle green, red\rangle, \langle green, blue\rangle, \langle red, blue\rangle\}, \{\langle green, blue\rangle, \langle red, blue\rangle\}, \{\langle red, blue\rangle\}\}$. Fig. 2 presents a possible solution to this problem.

1.2. Related work

General methods for solving CSPs include *generate-and-test* (GT) and *backtracking* (BT) methods [19]. GT generates each possible combination of the variables systematically and then checks to see whether it is a solution, i.e., whether it satisfies all the constraints. One limitation of this method is that it has to consider all instances of the Cartesian product of all the variable domains. In this respect, BT is more efficient than GT, as it assigns values to variables sequentially and then checks constraints for each variable assignment. If a partial assignment does not satisfy any of the constraints, it will backtrack to the most recently assigned variable and repeat the process again. Although this method eliminates a subspace from the Cartesian product of all the variable domains, its computational complexity for solving most nontrivial problems remains to be exponential.

Many studies have been conducted to investigate various ways of improving the above-mentioned BT method. In order to avoid *thrashing* [15,19] in BT, *consistency* techniques (Arc Consistency and k-Consistency) have been developed by Mackworth and other researchers [6,17,19,25,28], which are able to remove inconsistent values from the domains of the variables. In order to avoid both thrashing and *redundant-work* [19] in BT, a *dependency-directed backtracking* scheme and its improvements have been proposed [3, 19,33,39]. Other ways of increasing the efficiency of BT include the use of *search order* for variables, values, and consistency check [32]. Nevertheless, even with such improvements, BT is still unable to solve nontrivial large-scale CSPs in a reasonable runtime.

For the GT method, there have been some research efforts on making the solution generator smarter. The representatives of such efforts are *stochastic* and *heuristic* algorithms. Along this direction, one of the most popular ideas is to perform *local search* [16,19]. For large-scale n-queen CSPs, it gives better results than a complete, or even incomplete, systematic BT method. There are three key elements in local search [1], they are:

(1) Configuration: one possible assignment of all variables, not required to be a solution.
(2) Evaluation value: the number of unsatisfied constraints.
(3) Neighbor: the configuration obtained by changing one variable's assignment in the current configuration.

Local search generates an initial configuration and then incrementally uses "repair" or "hill climbing" to modify the inconsistent configuration to move to a neighborhood configuration that has the best or better evaluation value among its neighbors, until a solution is found. In order to avoid falling into *local optima*, it sometimes performs *random-walk* and *tabu search* [12]. As related to the idea of local search, other heuristics have also been developed, such as *hill-climbing* [1], *min-conflicts* [27], *MCRW* (Min-Conflicts-Random-Walk) [41], and *GSAT* (GSAT is a randomized local search procedure for solving propositional satisfiability problems) [36,37].

1.2.1. Min-conflicts heuristics

Repair-based heuristics were originally used in AI problem-solving systems to debug and modify initial solutions. Minton et al. [27] extended this approach to solving large-scale constraint satisfaction problems, and proposed a value-ordering heuristic, called *min-conflicts* heuristic. The *min-conflicts* heuristic attempts to select a new value that minimizes the number of outstanding constraint violations after each step.

They argued that the effectiveness of the *min-conflicts* heuristic is largely due to the repair of a complete but inconsistent assignment that is more informative in guiding search than an incrementally constructed partial assignment as in the traditional backtracking methods [27]. They also noted that the performance of this sequentially executed heuristic is remarkably comparable to that of a parallelly implemented Guarded Discrete Stochastic (GDS) network for solving constraint satisfaction problems (e.g., the Hubble Space Telescope scheduling problem). The two implementations employed the same heuristic (in fact, as stated in [27], the *min-conflicts* approach was intended to replicate the behavior of the GDS network): the network reassigns a value for a variable by choosing the value that violates the fewest constraints (i.e., flipping the neuron whose output is most inconsistent with its current input).

Our multi-agent approach utilizes the idea of inconsistency reduction over a complete initial assignment. However, our approach differs from the *min-conflicts* approach in a number of ways. For instance, our approach explores heuristics in addition to violation *minimization*, and relays on the combination of prioritized heuristics in order to improve computational efficiency. A more detailed discussion on the distinctions between the two approaches is provided in Section 5.

Other methods for solving CSPs have been based on Neural Networks [26] and Genetic Algorithms [18].

The above-mentioned methods and techniques have their advantages and drawbacks, and no single algorithm has been found to be suited to solving all CSPs. For small-size problems, we may use BT to readily find a solution, whereas for large-scale problems we may use local search. The efficiency of local search for solving n-queen problems is reported to be very efficient, among other algorithms [38]. However, we cannot prove that it can find solutions for every case every time as it is stochastic in nature, while on the other hand BT's performance is more stable and complete. Furthermore, local search is not suitable for problems other than n-queen. It requires that the problems have a clear neighborhood structure.

1.3. Multi-agent systems

Agent-based computation has been studied for some 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. Agents in such systems may be homogeneous or heterogeneous, and may have common goals or distinct goals [21]. Previous work on multi-agent systems has generally focused on areas such as simulations of social and biological systems, problem solving, communication, collective robotics, and electronic commerce on the Internet.

1.3.1. Distributed constraint satisfaction

A distributed constraint satisfaction problem (distributed CSP) is a constraint satisfaction problem in which variables and constraints are semantically partitioned (or *distributed*) into *sub-problems*, each of which is to be solved by an agent. When multiple agents are involved in solving a distributed CSP, the agents have to comply with certain constraints among them. Thus, finding a solution to a distributed CSP requires that all agents find the values for their variables that satisfy not only their own constraints but also interagent constraints. Examples of distributed CSP research efforts include distributed scheduling, planning, and reasoning [5,7,31,35].

Yokoo et al. [42–45] have made significant contributions in the area of distributed CSP. They developed an algorithm called *asynchronous backtracking* that guarantees the completeness, and then later extended this algorithm into a more efficient *asynchronous weak-commitment search* algorithm, by introducing dynamic ordering among agents. Furthermore, they also proposed a *multi-agent real-time-A* algorithm with selection* to solve an n-puzzle problem [45]. In those algorithms, the agents are individual solvers for obtaining partial solutions.

1.3.2. Swarm-like systems

In addition to the above-mentioned distributed constraint satisfaction approaches, it is worth mentioning a special instance of multi-agent systems for applications in computation and simulation; namely, *swarm* [40].

Swarm is a formulation for simulating distributed multi-agent systems, which involves three key concepts: living environment, agents with reactive rules, and a schedule serving

as a timetable to update the changes and dispatch agents' actions. Based on this idea, Liu et al. [23,24] developed an evolutionary autonomous agent system to adaptively extract image features and segments. Recently, Liu and Han [22] proposed an energy-based artificial-life model for solving n-queen problem.

1.4. The proposed approach

As inspired by the previous models of swarm, in this paper we will present a new approach called ERA (i.e., Environment, Reactive rules, and Agents) to solving CSPs. The CSPs considered here will not be limited to distributed CSPs. This approach is intended to provide an alternative, multi-agent formulation that can be used to handle general CSPs and to find approximate solutions without too much computational cost. The key idea behind ERA lies in a distributed multi-agent system, having the same architecture as swarm, i.e., *an environment*, *agents with moving behaviors*, *reactive rules*, and *a schedule*. This system self-organizes itself, when each individual agent follows its behavioral rules, and gradually evolves toward a global solution state.

From the point of view of solving CSPs, the proposed approach may be regarded as an extended GT approach, somewhat like local search. However, the main difference between ERA and local search is that the evaluation value of ERA is not the number of unsatisfied constraints for the whole assignment as in local search, but the number of unsatisfied constraints for the value of each variable—these numbers constitute an *environment* in the ERA system.

If there exists a consistent solution, the ERA system will eventually find it. On the other hand, if there is no complete solution, the ERA system can still generate an approximate solution. As to be presented in this paper, our approach can solve both n-queen problems and coloring problems. Furthermore, our experiments will show that ERA is efficient in finding exact as well as approximate solutions to CSPs in few time steps. Generally speaking, it is more efficient than the BT algorithms and more readily to solve different CSPs than the local search algorithm.

1.5. Organization of the paper

The remainder of this paper is organized as follows: Section 2 describes the basic ideas behind this distributed multi-agent oriented method. Section 3 discusses how to use this method to find an approximate solution. Section 4 describes several experiments and observations. Section 5 discusses the features of the proposed ERA approach and compare them with the existing major approaches in the field. Finally, Section 6 concludes the paper by highlighting the contribution of this work and some future extensions.

2. The multi-agent model

In this section, we will describe the basic formulation and algorithm for our proposed multi-agent model. Specifically, we will provide the definitions as well as formalisms for

agent environment, major policies for agent-environment interaction, reactive behaviors, and a basic ERA algorithm.

2.1. ERA fundamentals

Problem solving is an area that many multi-agent-based applications are concerned with. It includes the following subareas: *distributed solutions to problems*, *solving distributed problems*, and *distributed techniques for problem solving* [10,21]. In this paper, we will introduce an application of distributed techniques for solving CSPs. In our case, the domain of a CSP is represented into a multi-agent environment. Thus, the problem of finding a solution to the CSP is reduced to that of local behavior-governed moves within such an environment.

Specifically, the notions of agent and multi-agent system can be defined as follows:

Definition 2.1. An agent, a, is a virtual entity that essentially has the following properties:

(1) Be able to live and act in the environment.
(2) Be able to sense its local environment.
(3) Be driven by certain objectives.
(4) Have some reactive behaviors.

Definition 2.2. A multi-agent system is a system that contains the following elements:

(1) An environment, E, a space in which the agents live.
(2) A set of reactive rules, R, governing the interaction between the agents and their environment. They are the laws of the agent universe.
(3) A set of agents, $A = \{a_1, a_2, \ldots, a_n\}$.

The goal of this work is to examine how exact or approximate solutions to CSPs can be self-organized by a multi-agent system, consisting of $\{E, R, A\}$.

2.1.1. Overview of the ERA multi-agent formulation

The ERA method is meant to be a framework for interacting agents to achieve a global solution state. In ERA, the environment records the number of constraint violations of the current state for each value in the domains of all variables. Each agent represents a variable and the position of the agent corresponds to the value of the respective variable. The agent can move locally within a row and has its own reactive moving behaviors. Its objective is to move to a position whose constraint violation number is *zero*, we call it *zero-position* (for detail see Definition 2.3(2)). An exact solution state in ERA is reached when every agent (variable) finds its *zero-position*. The reactive rules correspond to the schedules for dispatching agents and updating the environment.

In this paper, we will first present the basic formulation and algorithm for the ERA method, and then focus on the effectiveness of some extended ERA techniques that utilize combined reactive behaviors as well as different selection probabilities.

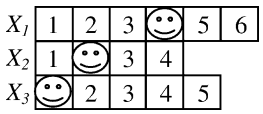

Fig. 3. An illustration of agent model for Example 2.1.

In the following paragraph, we will use an example to illustrate how a CSP can be translated into an ERA multi-agent system.

Example 2.1. A CSP is given as follows:

$$\boldsymbol{X} = \{X_1, X_2, X_3\}, \quad n = 3.$$
$$\boldsymbol{D} = \{D_1, D_2, D_3\},$$
$$D_1 = \{1, 2, 3, 4, 5, 6\}, \qquad D_2 = \{1, 2, 3, 4\}, \qquad D_3 = \{1, 2, 3, 4, 5\}.$$
$$\boldsymbol{C} = \{X_1 \neq X_2, X_1 > X_3\}.$$

Example 2.1 can be modeled as a multi-agent system as follows: The lattices represent an environment, where each row corresponds to a variable's domain and the length of each row is equal to the domain size. In each row, there exists only one agent. In this case, the horizontal coordinate of the agent represents the corresponding variable's value. As in Fig. 3, there are three agents all residing at *zero-positions*. The numbers that these agents occupy correspond to the values within the domains of the three variables. Fig. 3 shows a solution state of $S = \langle 4, 2, 1\rangle$.

2.1.2. Environment

An environment, E, has n rows corresponding to the number of variables. For all $i \in [1, n]$, row_i has $|D_i|$ columns. It records two kinds of values: the domain value and the violation value.

Definition 2.3. The data structure of E can be defined as follows:

(1) Size
- n rows $\Leftrightarrow$ n variables. $E = \langle row_1, row_2, \ldots, row_n\rangle$.
- $\forall i \in [1, n]$,
  $row_i \Leftrightarrow$ domain of $X_i \Leftrightarrow D_i$, so row_i has $|D_i|$ columns.
  $row_i = \langle lattice_{1i}, lattice_{2i}, \ldots, lattice_{|Di|i}\rangle$.
- E is an array of size $\sum |D_k|$. $e(i, j)$ refers to the position of $lattice_{ij}$.

(2) Values
- Domain value: $e(i, j).value$ records the ith value of domain D_j.
- Before we introduce $e(i, j).violation$, let us first define the notion of 'attack' between position$_1$ and position$_2$.
  We use (x_1, y_1) to represent position$_1$ and (x_2, y_2) to represent position$_2$. So,

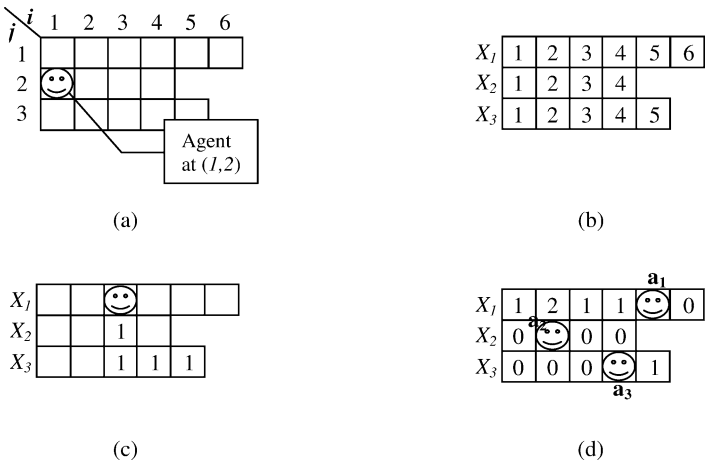

Fig. 4. An illustration of the agent environment. (a) the position of an agent, (b) the representation of domain values, and (c)–(d) violation numbers marked in the environment.

$$\text{Attack}\big((x_1, y_1), (x_2, y_2)\big) = \begin{cases} \text{true}, & \text{if there is constraint } C(R_t) \text{ between } X_{y_1} \text{ and } X_{y_2} \\ & \text{and } \langle e(x_1, y_1).value, e(x_2, y_2).value\rangle \notin C(R_t), \\ \text{false}, & \text{otherwise.} \end{cases} \quad (1)$$

- Violation number: $e(i, j).violation$ records in the current state how many agents whose positions *attack* position (i, j), i.e., $e(i, j).violation = m$ means there are m agents whose assignments dissatisfy the assignment of $X_j = e(i, j).value$. The $e(i, j).violation$ values are dynamically modified since the agents keep on moving and their corresponding state is changing. The violation numbers will be updated by applying an *updating-rule*, which will be described in Section 2.1.5.
- *zero-position*: position (i, j), in which $e(i, j).violation = 0$. That means all other agents agree on $X_j = e(i, j).value$, according to constraints related to X_j.

Fig. 4(a) presents the position of an agent at $(1, 2)$. Fig. 4(b) shows the domain value of each lattice. row_1 contains values in domain $D_1 = \{1, 2, 3, 4, 5, 6\}$, row_2 represents $D_2 = \{1, 2, 3, 4\}$, and row_3 represents $D_3 = \{1, 2, 3, 4, 5\}$. Fig. 4(c) shows that if agent a_1 stays at $(3, 1)$, meaning $X_1 = 3$, according to the constraints of $X_1 \neq X_2$ and $X_1 > X_3$, it will violate $X_2 = 3$, $X_3 = 1$, $X_3 = 2$, and $X_3 = 3$. Therefore, it will contribute 1 to the violation number at position $(3, 2)$, $(3, 3)$, $(4, 3)$, and $(5, 3)$. Fig. 4(d) presents a snapshot for the state of the system with the violation numbers. Since all agents are at *zero-positions*, the state corresponds to an exact solution.

2.1.3. Agents

All agents inhabit in an environment, in which their positions indicate values of certain variables. During the operation of the system, the agents will keep on moving, based on certain reactive moving behaviors. At each time step, the positions of the agents provide a

consistent or inconsistent assignment for all variables. The agents are trying to find better positions that can lead them to a solution state.

Here is a summary of some major policies for agent-environment interaction in the ERA model:

(1) $\forall i \in [1,n]$, a_i represents X_i. As shown in Fig. 4(d), three agents, a_1, a_2, and a_3 represent X_1, X_2, and X_3, respectively.
(2) Agent a_i moves locally in row_i. It can only move to its right or left, but not up and down. $a_i.x$ represents its x-coordinate. So the position of a_i can be denoted as $(a_i.x, i)$.

Example 2.2. In Fig. 4(d), a_2 lives in row_2, and it can move freely to position (1, 2), (2, 2), (3, 2), or (4, 2) in one step, but not to other positions.

In this paper, we use function ψ to define an agent's move.

Definition 2.4. $\psi : [1,n] \times [1,|D_i|] \to [1,|D_i|]$. $\psi(x,y)$ gives the x-coordination of the new position of agent a_i, after it moves from position (x,y). So the new position can be represented as $(\psi\ (x,y), y)$.

(3) In any state of the system, the positions of all agents form an assignment for all variables. $\forall j \in [1,n]$, $X_j = e(a_j.x, j).value$. It may not be a consistent assignment, i.e., not an exact solution.
If an assignment satisfies all the constraints, i.e., $\forall j \in [1,n]$, $e(a_j.x, j).violation = 0$, it is an exact solution, $S = \langle e(a_1.x,1).value, e(a_2.x,2).value, \ldots, e(a_n.x,n).value\rangle$.
(4) Agent a_i is able to 'perceive' the violation number for each lattice in row_i. Here, we define a function $\varphi(i)$ for returning a position (x-coordination) with the minimum violation number in row_i.

Definition 2.5. A *minimum-position* is the position of (x,j) such that $j \in [1,n] \wedge (\forall i \in [1,|D_j|])$, $e(x,j).violation \leqslant e(i,j).violation$.

Definition 2.6. The function for finding the *first minimum-position* for agent a_i in row_i is defined as follows:
$\varphi : [1,n] \to [1, \max(|D_i|)]$, $\varphi(i) = x \mid (x,i)$ is a *minimum-position* $\wedge\ (\forall j \in [1,x))$ (j,i) is not a *minimum-position*.

(5) In order to achieve a goal state, each agent uses its local reactive behaviors. Agents attempt to move toward *zero-positions* at each time step. But, in most cases, they cannot, or only some lucky agents can, find *zero-positions*, simply because some rows do not contain such positions at a certain time step. In such cases, the agents will have to perform other behaviors.

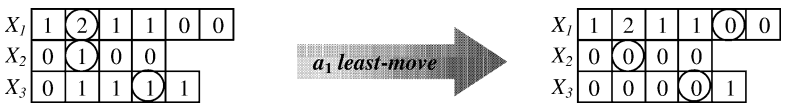

Fig. 5. The violation numbers are updated, when agent a_1 moves to a new position by executing a *least-move* behavior.

2.1.4. Local reactive behaviors

In order to reach a solution state, the agents will select and execute some predefined local reactive behaviors, namely, *better-move*, *least-move*, and *random-move*. Later in Section 4, we will investigate the effectiveness of these reactive behaviors by examining the performance of the ERA system with behavior prioritization and/or different selection probabilities.

2.1.4.1. least-move. An agent moves to a *minimum-position* with a probability of *least-p*. If there exists more than one *minimum-position*, we let the agent choose the first one on the left of the row. The *least-move* behavior can be expressed as follows:

$$\psi_{-l}(x, y) = \varphi(y). \tag{2}$$

Note that in this function, the result will not be affected by the current x, and the number of computational operations to find the position for each i is $|D_i| - 1$.

Example 2.3. Fig. 5 shows that when agent a_1 performs a *least-move*, it will first compute $\psi_{-l}(2, 1) = \varphi(1) = 5$, and thereafter move to (5, 1).

2.1.4.2. better-move. An agent moves to a position that has a smaller violation number than its current position with a probability of *better-p*. It will randomly select a position and then compare its violation number to decide whether or not it should move to this position. We use function *Random*(k) to get a random number of uniform distribution between 1 and k. This behavior can be defined using function ψ_{-b}:

$$\psi_{-b} = \begin{cases} x, & \text{when } e(\mathit{Random}(|D_y|), y).\mathit{violation} \geqslant e(x, y).\mathit{violation}, \\ \mathit{Random}(|D_y|), & \text{when } e(\mathit{Random}(|D_y|), y).\mathit{violation} < e(x, y).\mathit{violation}. \end{cases} \tag{3}$$

Although it may not be the best choice for the agent, the computational cost required for this behavior is much less than that of *least-move*. Only two operations are involved for deciding this move, i.e., producing a random number and performing a comparison. This behavior can readily find a position to move to especially when the agent is currently at a larger violation position.

As will be shown in Section 4, the *better-move* behavior plays an important role in bringing down the number of global constraint violations in a few time steps.

Example 2.4. Fig. 6 shows that when agent a_1 performs a *better-move*, it will compute $\psi_{-b}(2, 1)$. Suppose that $\mathit{Random}(6) = 3(|D_1| = 6)$. Thus, $\psi_{-b}(2, 1) = 3$, since $e(2, 1).\mathit{violation} > e(3, 1).\mathit{violation}$. The new assignment will become $\langle 3, 2, 4\rangle$. Although

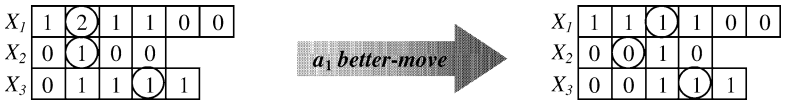

Fig. 6. The violation numbers are updated, when agent a_1 moves to a new position by executing a *better-move* behavior.

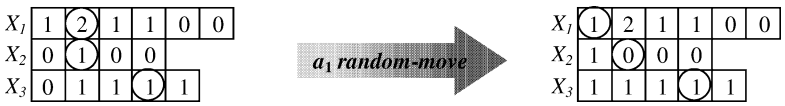

Fig. 7. The violation numbers are updated, when agent a_1 moves to a new position by executing a *random-move* behavior.

this assignment is not an exact solution, it is a better approximate solution than the assignment of $\langle 2, 2, 4\rangle$ as in Fig. 5, because the new state has only one constraint, $X_1 > X_3$, unsatisfied.

2.1.4.3. random-move. An agent moves randomly with a probability of *random-p*. *random-p* will be relatively smaller than the probabilities for selecting *better-move* and *least-move* behaviors. It is somewhat like a *random-walk* in local search. For the same reason as in local search, *random-move* is necessary because without randomized moves the system will get stuck in *local-optima*, that is, all the agents are at *minimum-positions*, but not all of them at *zero-positions*. In the state of *local-optima*, no agent will move to a new position if using the behaviors of *better-move* and *least-move* alone. Thus, the agents will lose their chance of finding a solution if without any techniques to avoid getting stuck in *local-optima*.

random-move can be defined using function ψ_{-r}:

$$\psi_{-r}(x, y) = \mathit{Random}(|D_y|). \tag{4}$$

Example 2.5. Fig. 7 shows that when agent a_1 performs a *random-move*, it will randomly produce a number. If $\mathit{Random}(6) = 1$, it will move to $(1, 1)$. If $\mathit{Random}(6) = 3$, it will move to $(3, 1)$.

2.1.5. System schedule

The multi-agent system proposed in this paper is concurrent and discrete in nature, with respect to its space, time, and state space. In the present *simulated* implementation, the system will use a *discrete clock* to synchronize its operations, as shown in Fig. 8. It works as follows:

- *time step* $= 0$: The system is initialized. We place n agents into the environment, a_1 in row_1, a_2 in $row_2, \ldots, a_n$ in row_n. The simplest way to place the agents is to randomly select positions. That is, for a_i, we set a position of $(\mathit{Random}(|D_i|),\ i)$.
- *time step* $\leftarrow$ *time step* $+ 1$: For each time step, which means one unit increment of the system clock, all agents will have a chance to decide their moves, that is, *whether to move or not and where to move, and then move synchronously*.

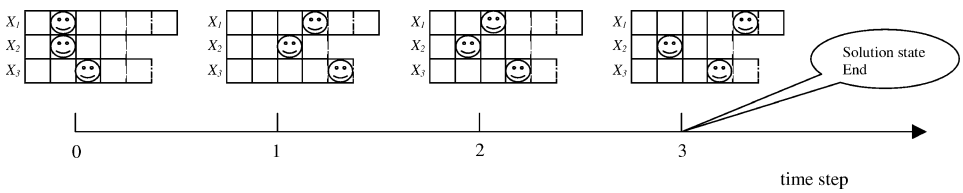

Fig. 8. Distributed agent-environment interaction at different time steps.

It should be pointed out that in the simulation, the multi-agent system dispatches the agents one by one. The order of dispatching is based on a random or a predefined sequence.

After the move of an agent from (x_1, y) to (x_2, y), the violation number of the environment will be updated according to the following two *update-rule*s:

(1) *update-rule* 1: Remove from (x_1, y)

$$(\forall x' \in [1, |D_y|])\ (\forall y' \in [1, n])\ \big(Attack\big((x_1, y), (x', y')\big)\big),$$

$$\text{execute } e(x', y').violation \leftarrow e(x', y').violation - 1.$$

(2) *update-rule* 2: Add to (x_2, y)

$$(\forall x' \in [1, |D_y|])\ (\forall y' \in [1, n])\ \big(Attack\big((x_2, y), (x', y')\big)\big),$$

$$\text{execute } e(x', y').violation \leftarrow e(x', y').violation + 1.$$

- *End*: After the moves of agents at each time step, the system will check whether all agents are at *zero-positions* and whether its clock exceeds a time threshold (i.e., time allowed). If one of these conditions is `true`, the system will stop its operations and return either an exact or an approximate solution.

Another way to terminate the operations is when q agents are staying at *zero-positions*.

2.2. The basic ERA algorithm

Now let us consider the basic ERA algorithm, from which a number of ERA properties and extended methods will be derived in the following sections.

Fig. 9 presents a function for initializing individual agents and then adding them to the environment randomly. This function also initializes the probabilities for *better-move*, *least-move*, and *random-move*. Fig. 10 provides the *RemoveFrom* function that updates environment numbers, according to *update-rule 1* when removing an agent from the environment. Fig. 11 shows the function of *AddTo* that updates violation numbers, according to *update-rule 2* when adding an agent to the environment. Function *SelectBehavior* in Fig. 12 selects a reactive behavior, according to the probabilities for various behaviors.

The complete listing of the basic ERA algorithm is given in Fig. 13.

Input: a_i
Output: the probabilities of a_i.
1. $a_i.random\text{-}p = p_1$;
2. $a_i.least\text{-}p = p_2$;
3. $a_i.better\text{-}p = p_3$;
4. $a_i.x = Random(|d_i|)$;
5. $AddTo(a_i.x, i)$;

Fig. 9. Function *Initialize*.

Input: position (x, y)
Output: updating violation numbers according to *update-rule 1* while removing a from (x, y).
1. **For** all position $(i, j) \in environment$ **do**
2. **If** $Attack((i, j), (x, y))$ **then** $e(i, j).violation \leftarrow e(i, j).violation - 1$;
3. **End for**

Fig. 10. Function *RemoveFrom*.

Input: position (x, y)
Output: updating violation numbers according to *update-rule 2* while adding a to (x, y).
1. $a_i.x = x'$;
2. **For** all position $(i, j) \in environment$ **do**
3. **If** $Attack((i, j), (x, y))$ **then** $e(i, j).violation \leftarrow e(i, j).violation + 1$;
4. **End for**

Fig. 11. Function *AddTo*.

Input: i
Output: ψ
1. $p = Random(a_i.random_p + a_i.least_p + a_i.beter_p)$;
2. **if** ($p \leq a_i.random\text{-}p$) **then return** ψ_r;
3. **if** ($p \leq a_i.random\text{-}p + a_i.least\text{-}p$) **then return** ψ_l;
4. **return** ψ_b;

Fig. 12. Function *SelectBehavior*.

2.3. Properties of the basic ERA algorithm

2.3.1. Termination

After each move, the ERA system will check whether all agents stay at *zero-positions*. Generally speaking, the termination condition for an exact solution can be stated as follows:

condition-1: $(\forall a_i \in A)\ e(a_i.x, i).violation = 0.$

For an approximate solution, we can employ certain termination conditions as mentioned in Section 2.1.5, such as a threshold of time step. In this case, the algorithm will terminate if the clock exceeds $t\text{-}max$. Thus, the termination condition for an approximate solution can be stated as follows:

condition-2: time step $t\text{-}max$.

Input: n variables, domains of variables, and constraints.
Output: an (approximate) solution.

Section-1. Initialization of the system:
1. *time step* = *0*;
2. **For** all position $(i, j) \in e$ **do**
3. $e(i, j).value$ = the corresponding i[th] value of domain D_j;
4. $e(i, j).violation$=*0*;
5. **End for**
6. **For** all $a_i \in A$ **do**
7. *Initialize*(a_i);
8. **End for**

Section-2.Running of the system:
9. **While** (***true***) **do**
10. **For** all $a_i \in A$ **do**
11. ψ = *SelectBehavior*(*Random-Move*, *Least-Move*, *Better-Move*);
12. New position $(x', i) = (\psi(a_i.x, i), i)$;
13. **If** current-position $(a_i.x, i) = (x', i)$ **then** stay
14. **Else**
15. *RemoveFrom*($a_i.x$, i);
16. *AddTo*(x', i);
17. **If** current-state *can satisfy us* **GoTo** 21;
18. **End for**
19. *time step* ++;
20. **End while**

Section-3. Output solution:
21. **For** all $a_i \in A$ **do**
22. $X_i = e(a_i.x, i).value$;
23. **End for**

Fig. 13. The basic ERA algorithm for solving CSPs. It should be noted that the parallel operations of distributed agents are here simulated by means of sequentially dispathing agents and allowing them to sense the present state of their environment, containing violation numbers, and then decide where to move. In so doing, the movements of the agents will not interfere with each other. That is, their decisions will be independent. This is because in our implementation the sequentially updated violation numbers as a result of each agent's movement will be copied as the next state of the environment only after all the agents have been given the chance to move.

2.3.2. Correctness

We now give the correctness theorems for the basic ERA algorithm. Detailed proofs for the theorems can be found in Appendix A.

Note that when the system terminates at *condition-1*, all agents are at *zero-positions*.

Theorem 2.1. *If* (x, y) *is a zero-position, i.e.,* $e(x, y).violation = 0$*, the following assertion is* `true`: $(\forall a_i \in A, i \neq y)\ (\exists t \in [1, m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow \langle e(a_i.x, i).value, e(x, y).value \rangle \in C(R_t)$.

Theorem 2.2. *The assignment of* $S = \langle X_1, X_2, \ldots, X_n \rangle$, $X_i = e(a_i.x, i).value$*, is an exact solution when the system terminates at* condition-1.

2.3.3. Complexity

Now let us discuss the complexity of the basic ERA algorithm.

Theorem 2.3. *The space complexity of the basic ERA algorithm is* $O(\sum |D_i|)$.

Proof. The main contribution to the space complexity is from the storage for environment e. e has n rows, each row_i has $|D_i|$ lattices. So the total number of lattices in e is $\sum|D_i|$. For each lattice, it records 2 values, i.e., the domain value and the violation value. It requires $2\sum|D_i|$ units to record environment e. Another contribution to the space is from the storage for n agents. For each agent, it records the current x-coordinate and three probabilities for *better-move*, *least-move*, and *random-move*, respectively. So it requires $4n$ space units in total for n agents. If all the agents have the same probabilities for *better-move*, *least-move*, and *random-move*, it needs only $n+3$ units in total for n agents. In conclusion, the space complexity is $O(\sum|D_i|)$. □

Theorem 2.4. *The time complexity of the initialization is* $O(\sum|D_i|)$.

Proof. For lines 2–5 in Fig. 13, there are in total $\sum|D_i|$ lattices to be initialized with the domain values and the violation numbers. So it needs $2\sum|D_i|$ operators. And for lines 6–8, there are n agents to be initialized. For each agent, there are 3 operators for initializing the probabilities of three behaviors. There are in total $3n$ operations. So the complexity of initialization is $O(\sum|D_i|)$. □

Theorem 2.5. *The time complexity of each time step in the basic ERA algorithm is bounded by* $O(n\sum|D_i|)$ *in the worst case.*

Proof. The main contribution to the time complexity is from the agent's move, including the modification of violation numbers in the functions of *RemoveFrom* and *AddTo*, and the checking of the solution state. The number of operations in *RemoveFrom* and *AddTo* is $\sum|D_i|$ in the worst case that there exists a constraint between every two agents. And for the checking of a solution state, it needs n checks in the worst case. Now for each agent's move, the total operation number is $\sum|D_i|+n$. So the complexity is bounded by $O(\sum|D_i|+n)$ in the worst case. In conclusion, for n agents, in the worst case, the time complexity is bounded by $O(n\times(\sum|D_i|+n)) = O(n\sum|D_i|+n^2)) = O(n\sum|D_i|)$. □

3. Approximate solution

One of the major motivations for developing the ERA multi-agent method is to be able to find an *anytime* solution, although it may be approximate, within the time allowed. In this section, we will discuss some properties of the basic ERA method, with respect to the goal of deriving approximate solutions:

(1) Each state represents an approximate solution.
In the BT method, variables are assigned with values sequentially. Unless the first k variables' assignments satisfy constraints, the $(k+1)$th variable's assignment will not be considered. Thus, we cannot get an assignment for all variables when the solution is not found. That is to say, in the process of BT, we cannot get an approximate solution. However, in the ERA method, every state, including the initial state, represents an

assignment to all variables, even though it may not be an exact solution. Therefore, ERA is able to provide an approximate solution at *anytime*.
This property is useful for real-time systems that require a solution within a fixed time interval while being not so demanding on the optimality of the solution.

(2) The system always evolves toward a better state in which more and more constraints are satisfied.
Note that *random-p* is much smaller than *better-p* and *least-p* in most situations. Thus, agents will have a greater chance to choose either *better-move* or *least-move*, in order to effectively reduce the number of unsatisfied constraints. In the ERA method, $e(a_i.x, i).violation$ records the violation number for the position at which agent a_i resides.

Definition 3.1. λ (s) represents in the current state s the sum of the violation numbers for those positions at which agents reside, so $\lambda(s) = \sum_{i\in[1,n]} e(a_i.x, i).violation$.

If state s is not a solution state, that is, the positions of some agents are not *zero-positions*, then $\lambda(s) > 0$. Otherwise, $\lambda(s) = 0$ for solution state s.

Theorem 3.1. *For state s,* $\lambda(s) = 0 \Leftrightarrow$ *s is a solution state.*

Proof. Because $\forall i \in [1,n]$, $j \in [1,|D_i|]$, $e(a_i.x, i).violation \geqslant 0$, we have $\lambda(s) = 0 \Leftrightarrow \forall i \in [1,n]$, $e(a_i.x, i).violation = 0 \Leftrightarrow s$ is a solution state (based on Theorem 2.2). □

Now we can see that the process of finding a solution is essentially a process of minimizing the value of $\lambda(s)$. In initial state s_0, $\lambda(s_0) > 0$ for most situations. Gradually, as the system keeps on dispatching agents to move to smaller violation-number positions, the value of $\lambda(s)$ will get minimized. When $\lambda(s)$ reaches *zero*, an exact solution is found.

Theorem 3.2. *When agent* a_i *moves from position* (x_1, y) *to position* (x_2, y)*, the variation of* λ *can be computed as follows*: $\Delta_\lambda = 2\times(q_2 - q_1)$ *where* $q_1 = e(x_1, y).violation$, $q_2 = e(x_2, y).violation$.

Proof. We use $\Delta_{\lambda 1}$ to denote the variation after picking up a_i from (x_1, y), $\Delta_{\lambda 2}$ to denote the variation after placing a_i to (x_2, y). When a_i is at position (x_1, y), there will be q_1 agents attacking a_i. When we pick up a_i from (x_1, y), a_i's contribution to λ is *zero*, and all these q_1 agents' violation numbers are reduced by 1. Now $\Delta_{\lambda 1} = -q_1 - q_1 \times 1 = -2 \times q_1$. Then we place a_i to (x_2, y), a_i's contribution to λ is q_2, and there will be q_2 agents attacking a_i. The violation numbers of all these q_1 agents are increased by 1, i.e., $\Delta_{\lambda 2} = q_2 + q_2 \times 1 = 2 \times q_2$. So, we have $\Delta_\lambda = \Delta_{\lambda 1} + \Delta_{\lambda 2} = -2 \times q_1 + 2 \times q_2 = 2 \times (q_2 - q_1)$. □

Theorem 3.3. *After agent* a_i *performs better-move or least-move,* $\Delta_\lambda \leqslant 0$.

Proof. Because $\psi_{-l}(x,\ y) = \varphi(y)$,

$$e(x, y).violation \geqslant e\big(x, \varphi(y)\big).violation \quad \text{(based on Definitions 2.5 and 2.6)}, \tag{5}$$

$$\Delta_\lambda = 2 \times \big(e\big(x, \varphi(y)\big).violation - e(x, y).violation\big) \quad \text{(based on Theorem 3.2)}. \tag{6}$$

Thus, for *least-move*, $\Delta_\lambda \leqslant 0$.
And since

$$\psi_{-b} = \begin{cases} x, & \text{when } e(Random(|D_y|), y).violation \\ & \geqslant e(x, y).violation \\ Random(|D_y|), & \text{when } e(Random(|D_y|), y).violation \\ & < e(x, y).violation \end{cases}$$

we have

$$e(x, y).violation \geqslant e\big(x, \psi_{-b}(x, y)\big).violation, \tag{7}$$

$$\Delta_\lambda = 2 \times \big(e\big(x, \psi_{-b}(x, y)\big).violation - e(x, y).violation\big) \quad \text{(based on Theorem 3.2)}. \tag{8}$$

Thus, for *better-move*, $\Delta_\lambda \leqslant 0$. □

In the process of dispatching, agents will have a much higher probability to perform either a *better-move* or a *least-move*. So after each move, $\Delta_\lambda \leqslant 0$, which means that λ is decreasing and the system is improving the solution. However, $\lambda(s_1) < \lambda(s_2)$ does not always indicate that state s_1 is better than state s_2. For instance, if s_1 is a local optimum state, which means all agents are at *minimum-positions*, but not all of them are at *zero-positions*, it is hard for s_1 to move to a new state except using a *random-move*. So, λ is just one of the important criteria.

(3) After a few steps, the assignments of most variables will satisfy constraints.
Because we randomly place the agents at the initialization step, they will seldom be placed right at good positions. In other words, they will most likely be placed at positions that have large violation numbers. After one time step, many agents that apply the behavior of *better-move* or *least-move* will move to positions with smaller violation numbers, which means $\Delta_\lambda < 0$. The improvement achieved at the first time step will be very substantial because $|\Delta_\lambda|$ is large. While at the following time step, the chance for finding a smaller violation position is becoming less as many agents are already at the *minimum-positions* of their respective rows. After r time steps (to different problem r is different), the variation of λ will fluctuate up and down around a fixed value. In this state, the agents usually stay at their original positions and only the behavior of *random-move* will make them move to other positions. In such states, λ is very small and the corresponding solution can satisfy most of the constraints. So, if we do not require an exact solution, we may stop the system and get an approximate solution from the current state. This phenomenon can readily be observed in the experiments of solving n-queen problems and coloring problems as to be discussed in the next section.

4. Empirical studies on extended ERA methods with behavior prioritization and different selection probabilities

The preceding sections have presented the basic ERA formulation and algorithm, and discussed some of their key properties. In this section, we will further examine the extensions and performance of the ERA approach under various behavioral settings. Specifically, the goal of this section is threefold:

(1) It presents several empirical results on solving different n-queen and coloring problems.
(2) It discusses how to apply and implement this approach by choosing the probabilities of *least-move* and *random-move*.
(3) It examines the effectiveness of prioritizing agent behaviors in order to efficiently derive an approximate solution.

In the experiments, we will initialize all agents with the same set of parameters, i.e., $\langle \textit{better-p}, \textit{least-p}, \textit{random-p}\rangle$. Specifically, $\forall i \in [1,n], a_i.\textit{better-p} = \textit{better-p}, a_i.\textit{least-p} = \textit{least-p}, a_i.\textit{random-p} = \textit{random-p}$.

4.1. n-queen problem

An n-queen problem has been stated in Example 1.1. It is required to place n queens on an $n \times n$ chessboard so that no two queens are in the same row, or the same column, or the same diagonal. This problem is a good benchmark because its problem size n can vary from 4 to a very large number. Also the solution to an n-queen problem can find many practical applications [38].

An n-queen problem can be translated into a binary CSP as described in Example 1.1 of Section 1.1:

- *n queens*: $X = \{X_1, X_2, \ldots, X_n\}$.
- $n \times n$ *chessboard*: $\boldsymbol{D} = \{D_1, D_2, \ldots, D_n\},\ \forall i,\ D_i = \{1, 2, \ldots, n\}$.
- *Placement requirement*:

$$\boldsymbol{C} = \big\{C(R_u) \mid \forall i, j \in [1,n], C(R_u) = \{\langle b, c\rangle \mid b \in D_i,\ c \in D_j,\ b \neq c,\ i - j \neq b - c,\ i - j \neq c - b\}\big\}.$$

In this CSP, each variable has the same domain $[1,n]$ and there is a constraint between every two variables. In the following paragraphs, we will show how to apply the basic ERA algorithm to solve this problem.

First, we use n agents to represent n queens (variables), a_i represents variable X_i, which is a queen in $\textit{row}_i$. Second, we model the domains, i.e., the chessboard, as the environment of the agents (see Fig. 14(a)). Fig. 14(b) presents an example of the multi-agent system for a 4-queen problem.

At the initialization step, the domain values will be recorded as $e(i,j).\textit{value}$ (see Fig. 14(a)) and the violation numbers for all positions will be set to *zero* (see Fig. 15(a)).

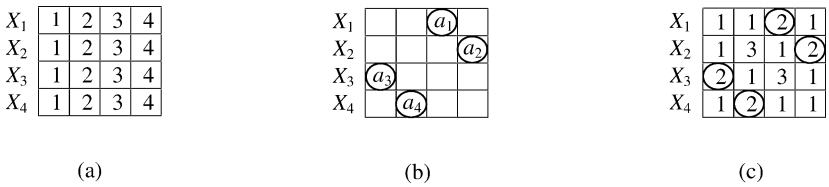

Fig. 14. (a) The representation of domain values for a 4-queen problem. (b) Four agents dispatched into the 4-queen environment. (c) Updated violation numbers corresponding to the positions of the four agents.

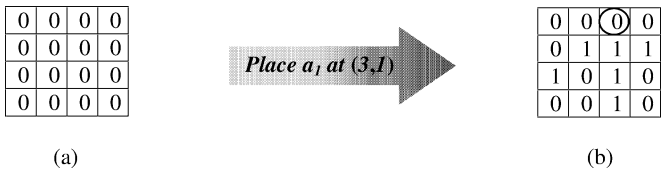

Fig. 15. (a) Violation numbers at the initialization step. (b) Violation numbers updated having placed a_1 at (3, 1).

After that, agents will be randomly placed into different rows. For instance, if agent a_1 is placed at position (3, 1), the violation numbers in the environment will be updated accordingly, as shown in Fig. 15(b).

Fig. 16 presents a series of snapshots from an 8-queen problem experiment. Here each circle signifies an agent. The number on the lattice gives the corresponding violation number. The darker the color of a lattice, the larger the violation number of that position will be. First, in the initialization of time step 0 in Fig. 16(a), eight agents are randomly placed onto the rows. In this particular case, none of the agents is at a *zero-position*. Five of them are at the positions of *violation* = 3. Two agents are at the positions of *violation* = 2. One agent is at the position of *violation* = 1. Obviously, the assignment according to this state is not a solution. For agent a_1 at position (4, 1), we can observe that agent a_2 at (3, 2) and agent a_3 at (2, 3) are both in the same diagonal as a_1, and agent a_4 is in the same column as a_1. So the position where a_1 stays has the violation number of 3. In this state of s_0, $\lambda\ (s_0) = 3+3+3+3+3+2+2+1 = 20$.

Between time step 0 and time step 1, most agents can move to a better position that reduces the violation number. At time step 1, eight agents have moved to a better position. Now we have $\lambda\ (s_1) = 1+1+0+1+1+0+0+0 = 4$. In this assignment, four variables (i.e., X_3, X_6, X_7, and X_8) satisfy all the constraints applicable to them. Two pairs, $\langle X_1, X_5\rangle$ and $\langle X_2, X_4\rangle$, cannot satisfy the constraints: X_1 and X_5 are in the same diagonal, and X_2 and X_4 are in the same column. Obviously, the assignment in state s_1 is much better than the assignment in state s_0.

From time step 1 to time step 2, the following moves have occurred: $s_1 \Rightarrow a_4$ stays, a_5 *least-moves* to (6, 5), a_3 stays, a_6 stays, a_2 stays, a_7 stays, a_1 stays, and a_8 stays $\Rightarrow s_2$. Now, $\lambda(s_2) = 0+1+0+1+0+0+0+0 = 2$. In this assignment, six variables (i.e., X_1, X_3, X_5, X_6, X_7, and X_8) satisfy all the constraints related to them, while the pair of $\langle X_2, X_4\rangle$ cannot satisfy each other.

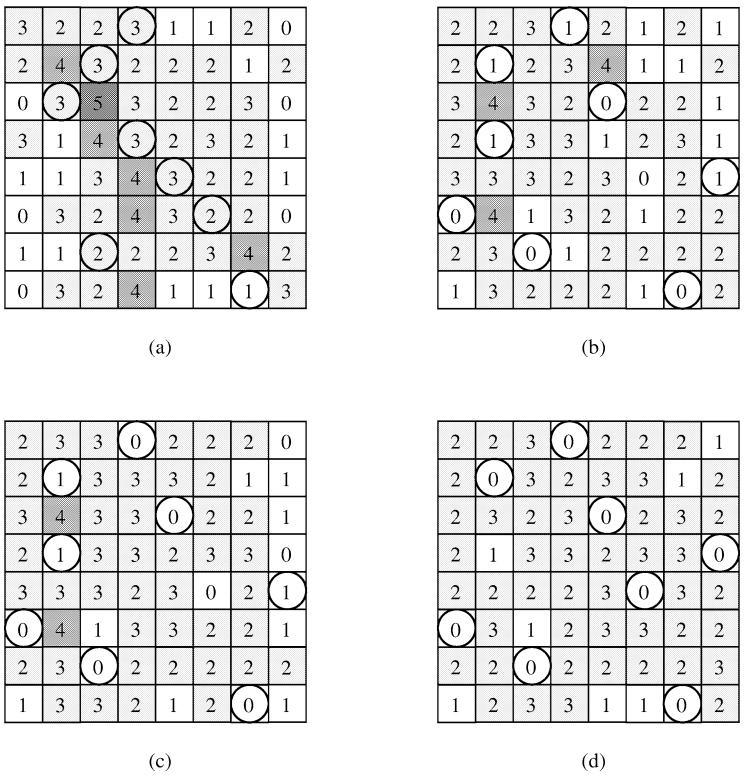

Fig. 16. (a) s_0 at time step 0 (initialization). (b) s_1 at time step 1. (c) s_2 at time step 2. (d) s_3 at time step 3, which is an exact solution state.

From time step 2 to time step 3, the moves can be summarized as follows: $s_2 \Rightarrow a_4$ *least-moves* to $(8, 4)$ but all other agents remain at the same positions $\Rightarrow s_3$. Now, $\lambda(s_3) = 0+0+0+0+0+0+0+0+0 = 0$. An exact solution state is reached.

4.1.1. The effects of the least-p/random-p ratio and behavior prioritization

Having illustrated the process of the basic ERA system in solving an n-queen problem, let us now consider the effects of behavior selection probabilities and behavior prioritization on the efficiency of finding an exact solution. In the experiments, we will let the system run until an exact solution is found. For the ease of comparison, we will record the average runtime required for generating the solution. Based on the observations made from the experiments, we will empirically show how the basic ERA method can be improved by adjusting *random-p* and by prioritizing behaviors.

4.1.1.1. The least-p/random-p ratio. From the above sections, we know that besides *better-move* and *least-move*, *random-move* is also necessary. If there is no *random-move*, i.e., *random-p* $= 0$, the system may get stuck in a local optimum and cannot find a solution. Now, the question that remains is how to set the probability ratio, *least-p*/*random-p*. It is

Table 1
Average runtime for different ratios of *least-p* to *random-p*

average runtime (s)		***least-p* : *random-p***			
		$0.5n$	n	$1.5n$	$2n$
n	1000	17.13	12.13	**8.63**	15.75
	2000	91.38	35	**31.13**	46.88
	3000	124.88	120.75	**79.75**	113.13
	4000	270.88	151.88	**150.25**	187.25
	5000	598.25	451.13	**370.75**	530.38
	6000	641.38	722.88	496.75	**426.38**
	7000	3478.75	1476.5	**1011.5**	1971.5

not so intuitive to see the most effective setting for solving a problem. In the following experiment, we will try to empirically determine a good *least-p*/*random-p* ratio.

Experiment 4.1. $n = \{1000, 2000, 3000, 4000, 5000, 6000, 7000\}$, *least-p*/*random-p* $= \{0.5n, n, 1.5n, 2n\}$, *type* = LR (10 runs). Note that the reason that we test n-queen problems up to 7000 queens is because beyond that number, the memory limitation of our computer becomes a problem. The behavioral type of LR indicates that in this experiment, the agents will only use the *least-move* and *random-move* behaviors.

Observation 4.1. The bold and underlined numbers in Table 1 correspond to the shortest ones among all the ratios for each n. As shown in the table, for all n except $n = 6000$, the ratio of *least-p* to *random-p* that results in the shortest runtime is $1.5n$. Therefore, we can obtain an empirical rule for setting the ratio of *least-p*/*random-p*: For n-queen problems, the good ratio is $1.5n$.

4.1.1.2. The high-priority better-move. Behaviors *better-move* and *least-move* are similar: move to a position based on the violation number. At each time step, it would be much easier for an agent using *better-move* to find a better position to move to than for the one using *least-move*. This is because *least-move* checks *all* the positions in its row, whereas *better-move* checks only *one* position. Therefore, the time complexity of *better-move* is much less than that of *least-move*.

In order to take the advantage of *better-move*, we decide to set this behavior to the highest priority, which means that an agent will first use *better-move* to compute its new position. If it fails to find a better position to move to, the agent will then turn to *least-move*. We call this behavior *better-least-move*, the probability of which will be the same as that of *least-p*.

Initially, most agents are at the positions with large violation numbers. The chance of successfully finding a position to move to with *better-move* is quite high. Thus at the first step, most agents will perform a *better-move* instead of a *least-move*.

Further to the above prioritized *better-move*, the next question is whether more *better-move* attempts will be helpful (since their complexity is low—a comparison operation). If so, how many *better-move* attempts will be most effective? In order to examine these issues, we will introduce another *better-move* right after the first *better-move* fails in finding a better position. For example, if a_i at position (x, y) performs a *better-move*, it will compute

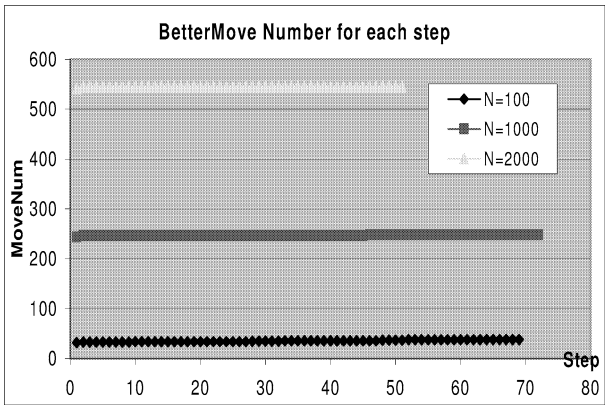

Fig. 17. Total *better-move* number vs. clock step.

$\psi_{-b}(x, y)$. If $\psi_{-b}(x, y) = x$, then it will compute $\psi_{-b}(x, y)$ again. If at this time $\psi_{-b}(x, y) = x$, it will then turn to *least-move*. We call this behavior BBLR (i.e., one *better-move* and if it cannot find a better position, it will perform BLR) type of behavior. Extending this concept, with the probability of *least-p*, the agent may perform at most r times *better-move* until it finds a better position to move to. We denote this behavior prioritization rBLR and its complexity is r.

4.1.1.3. The number of better-move at each step. Now, let us examine how many *better-move* attempts are needed in order to find a better position at each time step of the system. We will test and record the total number of *better-move* for all agents successfully finding a better position with *better-least-move* and *random-move* (this type of combined behaviors is called BLR). In this experiment, $n = 100$, 1000, and 2000, respectively.

Experiment 4.2. $n = \{100, 1000, 2000\}$, *least-p*/*random-p* $= n$, *type* $=$ BLR.

Observation 4.2. From Fig. 17, we note that except at time step 1, the lines for $n = 100$, 1000, and 2000 go very smoothly without so much changes. That means *better-move* enables most agents to find better positions at the first step.

Specifically, 32 agents in the 100-queen problem, 247 agents in the 1000-queen problem, and 541 agents in the 2000-queen problem can move to a better position by using *better-move* at the first time step. And after the first step, very few agents can find a better position with *better-move*. Based on the observation, we may change the type of behavior prioritization into FBLR, which means agents will perform BLR *only at the first* (F) *time step* and then perform LR at the following steps. So, the type of combined behaviors changes during the process.

4.1.1.4. rBLR. Agents will apply this type for all steps. We will test the runtime for $n = 1000$ with $r = 1$, 2, 3, and 5, respectively.

Experiment 4.3. $n = 1000$, *least-p*/*random-p* $= n$, *type* $=$ {BLR, 2BLR, 3BLR, 5BLR} (10 runs).

Table 2
Average runtime of $n = 1000$ with rBLR

	BLR	2BLR	3BLR	5BLR
average runtime (s)	6.13	4.5	5.25	10.25

Table 3
Average runtime of $n = 1000$ with LR, rBLR, and FrBLR

average runtime (s)		**type**		
		LR	***r*BLR**	**F*r*BLR**
r	**1**	12.25	6.13	5.25
	2		4.5	**3.63**
	3		5.25	4.75
	5		10.25	8.25

Observation 4.3. From Table 2, we note that the system has the best performance when $r = 2$. That means 2BLR will create more chance for agents to find a better position than BLR. The runtime complexity of 2BLR is less than 3BLR and 5BLR. So, 2BLR will be the best setting if we want to successfully find better positions, and at the same time, have less runtime complexity.

4.1.1.5. LR *vs.* rBLR *vs.* FrBLR. The following experiment compares the average runtime among the prioritization types discussed above. LR means that an agent only has two moving behaviors: *least-move* (with the probability of *least-p*) and *random-move* (with the probability of *random-p*). FrBLR means that the agent will perform rBLR behavior at the first step and then perform LR at all the following steps.

Experiment 4.4. $n = 1000$, *least-p*/*random-p* $= n$, *type* $=$ {LR, BLR, 2BLR, 3BLR, 5BLR, FBLR, F2BLR, F3BLR, F5BLR} (10 runs).

Observation 4.4. The results given in Table 3 show that FBBLR has the best runtime performance among all the types. For $\forall r \in \{1, 2, 3, 5\}$, the runtime of F$r$BLR is less than that of rBLR. So, we conclude that FrBLR is better than rBLR, and rBLR is better than LR.

4.1.2. Approximate solution

In this section, we will study the performance of the system in finding an approximate solution by tracking at each time step the total number of agent moves, the number of agents at *zero-positions*, and the total number of *zero-positions*. The results of our experiments consistently indicate that $\forall n \in [100, 7000]$, the system will converge as follows:

- After 1 step, 80% agents are at *zero-positions*.
- After 2 steps, $n - c_1$ (where c_1 is a constant, $c_1 \approx 25$) agents are at *zero-positions*.
- After 3 steps, $n - c_2$ (where c_2 is a constant, $c_2 \approx 7$) agents are at *zero-positions*.

This property enables the system to efficiently find an approximate solution: 3 steps are needed in order to find an approximate solution, in which about $n - 7$ queens will not attack each other.

4.1.2.1. Is the system stable? In the following paragraphs, we will track the system step by step and show how the system performs. The measures to be considered are as follows:

- *Move-Num* (number of moves): First, we observe the total number of moves that all agents have performed, from the beginning of the system till the current time step (i.e., accumulative number). Before an exact solution is found, the agents in the system should move to other positions to improve the current assignment. If no agent moves, the system will not improve itself. Thus, the number of moves at one time step measures the *improvement speed* of the system.
- *Zero-Agent-Num* (number of agents at *zero-positions*): This measure expresses how good the assignment in the current state is. If *Zero-Agent-Num* $= n$, the current state is an exact solution state. A larger *Zero-Agent-Num* means more variables satisfy the constraints.
- *Zero-Position-Num* (*zero-position* number in the environment): If there are a lot of *zero-positions* in the environment, the agents will have a good chance to find and move to *zero-positions*. So, the *Zero-position-Num* implies the degree of difficulty to find a zero-position to move to, and the difficulty for the system to improve itself.

Now let us examine the cases of $n = 100$, 1000, and 2000.

Experiment 4.5. $n = \{100, 1000, 2000\}$, *least-p*/*random-p* $= n$, *type* $=$ `2BLR`.

Observation 4.5.

(1) *Move-Num*: As shown in Figs. 18 and 19, especially in large problem sizes such as $n = 1000$, the agents move more drastically at the first time step, and thereafter the total number of moves increases slowly. This means that the agents can easily find a better position to move to at the first step, and then the chance of finding a better position decreases as time goes by.
(2) *Zero-Position-Num*: As also shown in Figs. 18 and 19, except in small problem sizes such as $n = 100$, the number of *zero-positions* drops sharply at the first 3 steps and then fluctuates at the following steps. These may explain why *Move-Num* increases at the first 3 steps and then remains unchanged. More *zero-positions* means more chance for agents to find a better position.
(3) *Zero-Agent-Num*: The plots as shown in Figs. 18–19 all increase at the first 3 steps, nearly reaching n, and then slightly fluctuate. This means that the system improves itself quickly during the first 3 steps.

4.1.2.2. Approximate solution: the first 3 *steps.* Since we have observed that the system will converge and its improvement will slow down after 3 steps, we can just let the system

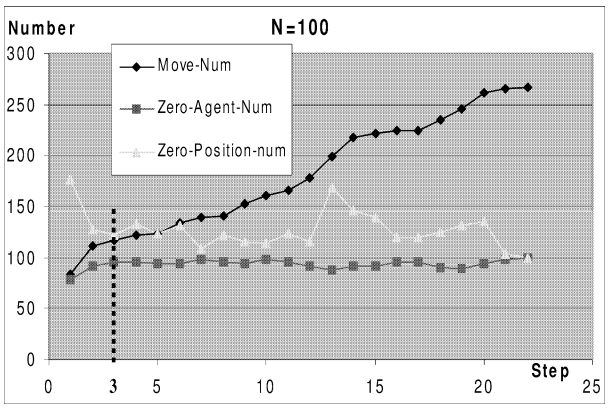

Fig. 18. *Move-Num*, *Zero-Agent-Num*, and *Zero-Position-Num* ($n = 100$).

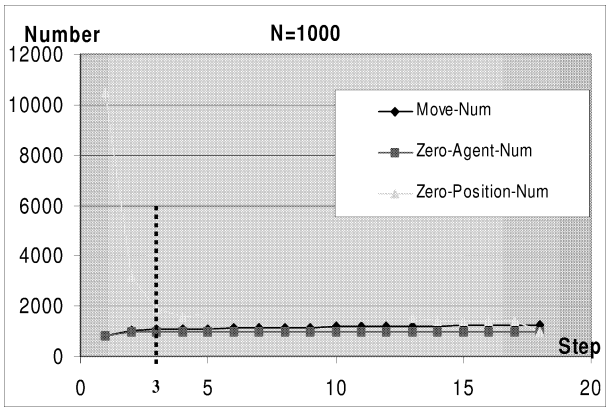

Fig. 19. *Move-Num*, *Zero-Agent-Num*, and *Zero-Position-Num* ($n = 1000$).

run for 3 steps and then get an approximate solution. Experiment 4.6 examines how well the system does at the first 3 steps.

Experiment 4.6. $n = [100, 7000]$, $\Delta n = 100$, *least-p*/*random-p* $= n$, *type* $=$ `F2BLR` (10 runs).

Observation 4.6.

(1) Our experiments have shown that after the initialization, nearly 10% agents stay at *zero-positions*.
(2) After the 1st step, nearly 80% $\times$ n agents stay at *zero-positions* when n is larger than 1000, as shown in Fig. 20. That means when $n > 1000$, there are 80% variables' assignments can satisfy the constraints. This result is desirable because it is obtained by just one step.

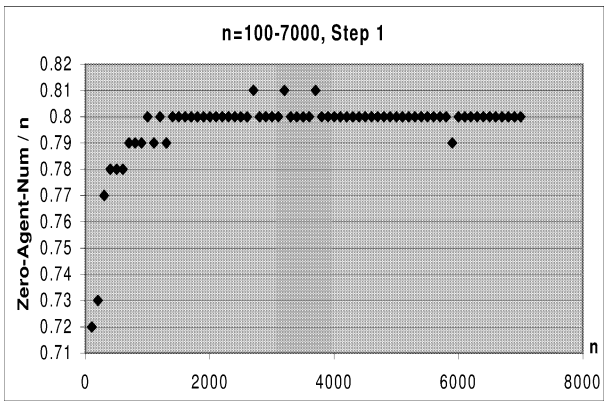

Fig. 20. Average *Zero-Agent-Num*/n of the 1st step for $n = 100$ to 7000, $\Delta n = 100$.

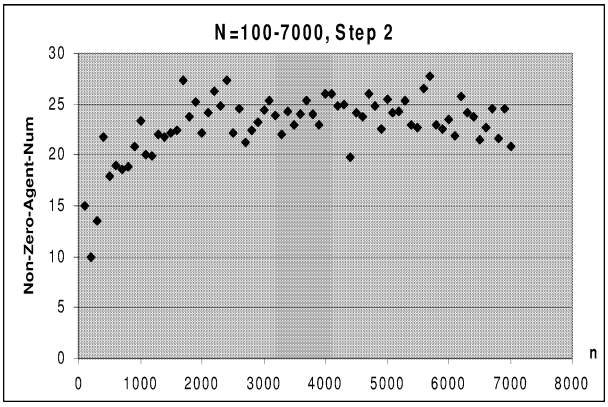

Fig. 21. Average *Non-Zero-Agent-Num* of the 2nd step for $n = 100$ to 7000, $\Delta n = 100$.

(3) After the 2nd step, nearly $n - 25$ agents stay at *zero-positions* when n is larger than 1000, as shown in Fig. 21. That means when $n > 1000$, there are about $n - 25$ variables' assignments can satisfy the constraints. On the other hand, there are about 25 variables' assignments cannot satisfy the constraints. This result is obtained by just two steps no matter how large n is.
(4) After the 3rd step, nearly $n - 7$ agents stay at *zero-positions* when n is larger than 1000, as shown in Fig. 22. That means when $n > 1000$, there are about $n - 7$ variables' assignments can satisfy the constraints. This is a good approximate solution obtained in just three steps.

Based on Theorem 2.5, we know that the complexity for the first time step is $O(n \sum |D_i|)$. Thus, we can tell that the complexity for the approximate solution will be bounded by $O(n \sum |D_i|)$. For n-queen problems, $|D_i| = n$, the complexity will be bounded

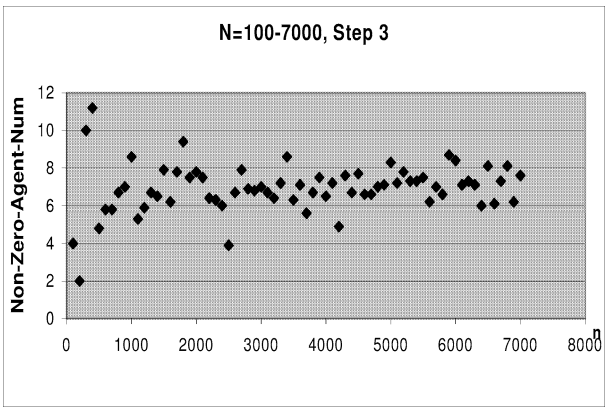

Fig. 22. Average *Non-Zero-Agent-Num* of the 3rd step for $n = 100$ to 7000, $\Delta n = 100$.

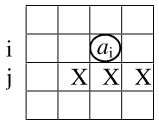

Fig. 23. Attacking 3 positions at most.

by $O(n^3)$. In fact, for such problems, the single time step complexity of the ERA method is $O(n^2)$.

Theorem 4.1. *The single time step complexity of the ERA method in solving n-queen problems is* $O(n^2)$.

Proof. The total number of operations in *RemoveFrom* and *AddTo* is $6n$ in the worst case. This is because between agent a_i and all other agents in row_j, there exist at most 3 positions in row_j (see Fig. 23), in which the agents will attack a_i. So the computation of *RemoveFrom* and *AddTo* is $3n$, in total $6n$. As for checking a solution state, it needs n tests in the worst case. Now for each agent's move, the total number of operations is $7n$. So the complexity for one time step is $7n^2$, bounded by $O(n^2)$, and the complexity for 3-step approximate solution finding is also $O(n^2)$. □

4.1.2.3. Runtime for finding an approximate solution. Now let us take a look at the runtime in finding an approximate solution in 3 steps.

Experiment 4.7. $n = \{100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000\}$, *least-p/random-p* $= n$, *type* $=$ `F2BLR` (10 runs for 3 steps each).

Observation 4.7. Fig. 24 shows that the empirical results are consistent with Theorem 4.1. To find an approximate solution in 3 time steps, in which nearly $n - 7$ variables satisfy the constraints, the runtime complexity is $O(n^2)$.

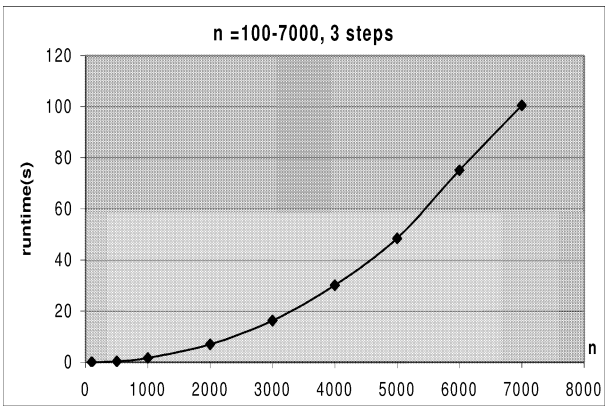

Fig. 24. Average runtime (second) for 3 steps ($n = 100 \sim 7000$).

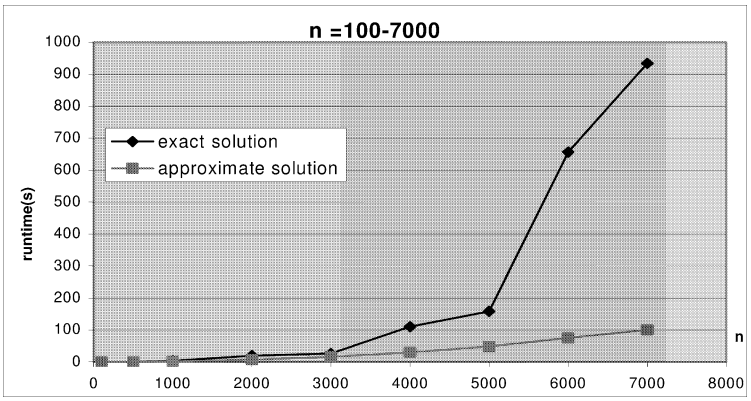

Fig. 25. Average runtime (second) for finding an exact solution and an approximate solution ($n = 100 \sim 7000$).

We have also compared the runtime for deriving an exact solution with that for finding an approximate solution. The results are presented in Fig. 25.

4.2. Coloring problem

A coloring problem has been defined in Example 1.2. Many problems of practical interest can be modeled as coloring problems, such as time tabling and scheduling [20], frequency assignment [13], register allocation [4], printed circuit board testing [14], and pattern labeling [30].

Given an undirected graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, an m-coloring problem can be translated into a binary CSP as follows:

- *n nodes*, $\boldsymbol{V}$: $\boldsymbol{X} = \{X_1, X_2, \ldots, X_n\}$, X_i represents $v_i \in \boldsymbol{V}$.
- *m colors*: $\boldsymbol{D} = \{D_1, D_2, \ldots, D_n\}$, $\forall i$, $D_i = \{1, 2, \ldots, m\}$.

- *Coloring requirement*:

$$\boldsymbol{C} = \big\{C(R_u) \mid \forall i, j \in [1, n]\ \langle v_i, v_j\rangle \in \boldsymbol{E},\ C(R_u) = \{\langle b,\ c\rangle \mid b \in D_i,\ c \in D_j, b \neq c\}\big\}.$$

In this case, all variables have the same domain, $[1, m]$, and there is a constraint between two nodes incident to an edge. An example coloring problem is given in Fig. 26. This instance can be colored by using three colors. Now, let us see how the ERA method works in solving this problem.

First, we use four agents to represent four nodes (variables). a_i represents node v_i (variable X_i). Second, we model the domains as the environment of the agents (see Fig. 27).

Initially, the domain values (labels of color) will be recorded as $e(i, j).value$ (see Fig. 27(a)) and the violation numbers for all positions will be set to *zero*. After that, we will randomly place the agents onto different rows. For instance, if we place agent a_1 at position (2, 1), the violation numbers will be updated accordingly as shown in Fig. 28.

In what follows, we will examine the performance of the ERA method in solving a set of large-scale coloring problems from DIMACS (the Center for Discrete Mathematics and

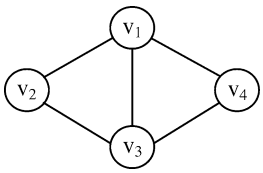

Fig. 26. A coloring problem.

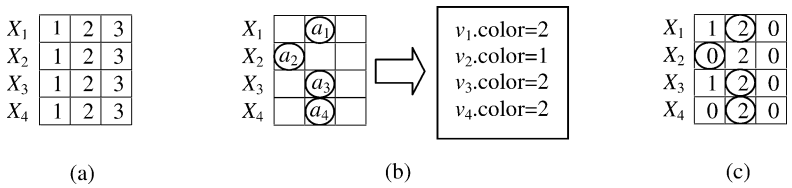

Fig. 27. (a) The representation of domain values into the environment of a multi-agent system for the coloring problem of Fig. 26. (b) Four agents dispatched into the environment, the positions of which correspond to a specific color assignment. (c) Violation numbers updated.

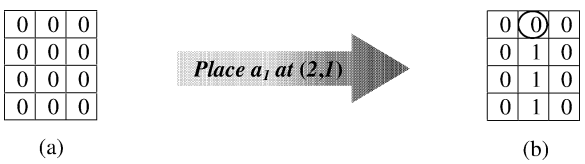

Fig. 28. (a) Violation numbers at the initialization step. (b) Violation numbers updated having placed a_1 at (2,1).

Theoretical Computer Science) [46].[1] In particular, we will use the test problems from *Donald Knuth's Stanford GraphBase* [47].

Each problem includes the information of (nodes, edges), number of optimal colors, and source. Thus, each problem has n variables, where n is the number of nodes, k values for each domain, where k is the number of optimal colors, and m constraints, where m is the number of edges. In the following listing, the source of SGB (from Michael Trick (trick@cmu.edu)) refers to Donald Knuth's Stanford GraphBase:

(1) *miles250.col* (128,387), 8, SGB
(2) *miles500.col* (128,1170), 20, SGB
(3) *miles750.col* (128,2113), 31, SGB
(4) *miles1000.col* (128,3216), 42, SGB
(5) *miles1500.col* (128,5198), 73, SGB
(6) *anna.col* (138,493), 11, SGB
(7) *david.col* (87,406), 11, SGB
(8) *huck.col* (74,301), 11, SGB
(9) *jean.col* (80,254), 10, SGB
(10) *games120.col* (120,638), 9, SGB
(11) *inithx.i.1.col* (864,18707), 54, REG

In the above problems, *miles graphs* are similar to geometric graphs in that the nodes are placed in space with two nodes connected if they are close enough. The nodes represent a set of United States cities and the distance between them corresponds to the road mileage recorded in 1947. Book graphs are created where each node represents a character and two nodes are connected by an edge if the corresponding characters encounter each other in the book. The book graphs were created by *Anna*, *David*, *Huck* and *Jean*. *games120* from Knuth represents the 1990 college football season. In *games120*, the nodes represent college teams and two teams are connected by an edge if they played against each other during the season. *inithx.i.1* is a problem based on register allocation (named REG) as contributed by Gary Lewandowski (gary@cs.wisc.edu).

4.2.1. Approximate solution

In the previous experiments on n-queen problems, we have shown that the system can get a good approximate solution after 3 steps. Similarly, for coloring problems, we will also test the performance of 3 steps. We will examine the runtime for 3 steps and the number of agents at *zero-positions* (*Zero-Agent-Num*) at each time step.

Experiment 4.8. *least-p*/*random-p* $= n$, *type* $=$ `BBBLR` (10 runs for 3 steps each).

Observation 4.8. The system can quickly find an approximate solution. As shown in Table 4, the average runtime measurement for all problems are close to *zero*.

[1] More information can be found from the OR-Library [48] for test data sets for a variety of Operations Research (OR) problems.

Table 4
Zero-Agent-Num for coloring problems after 3 steps

measurements	*miles 250*	*miles 500*	*miles 750*	*miles 1000*	*miles 1500*	*anna*	*david*	*huck*	*jean*	*games 120*	*inithx.i.1*
runtime (s)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
zero-agent-num / nodes	124 / 128	124 / 128	122 / 128	122 / 128	120 / 128	134 / 138	86.8 / 87	74 / 74	80 / 80	120 / 120	604.2 / 864
zero-agent-num (%)	0.97	0.97	0.95	0.95	0.94	0.97	1	1	1	1	0.7

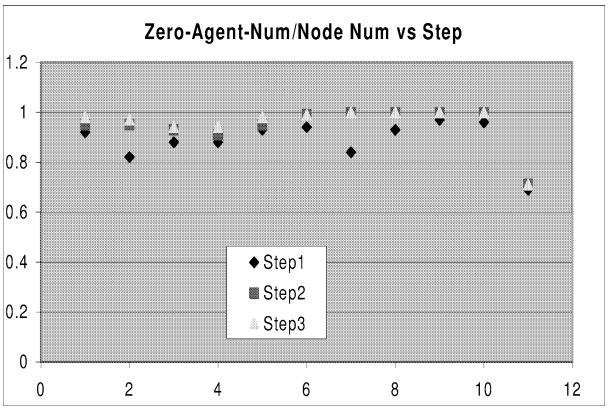

Fig. 29. Zero-Agent-Num (%) at each time step for 11 coloring problems. The number on the horizontal axis corresponds to an individual problem: (1) *miles250*, (2) *miles500*, (3) *miles750*, (4) *miles1000*, (5) *miles1500*, (6) *anna*, (7) *david*, (8) *huck*, (9) *jean*, (10) *games120*, and (11) *inithx.i.1*.

Observation 4.9. After the first step, the system can find an approximate solution for all problems except *inithx.i.1*, with more than 80% variables satisfying the constraints. For the *miles* problems, almost 95% variables (nodes) can satisfy the constraints after 3 steps. For the problems of *david*, *huck*, *jean*, and *games120*, the system can find an exact solution within 3 steps. In fact, in this experiment, they need only 2 time steps to find an exact solution (see Fig. 29). An exception is the problem of *inithx.i.1*, where about 70% variables can satisfy all the constraints.

4.2.1.1. Zero-Agent-Num at each time step. In order to study the performance of this system, we will record *Zero-Agent-Num* at each time step. The problems of *david*, *huck*, *jean*, and *games120* will be excluded from the following experiment, since the system can find an exact solution for these problems within 2 steps.

Experiment 4.9. *least-p*/*random-p* = *n*, *type* = `FBBLR` (1 run).

Observation 4.10. The results from this experiment are shown in Fig. 30, where the *Zero-Agent-Num* curves fluctuate around 120 after 3 steps. The value at the last time step corresponds to an exact solution found. This observation has been quite consistent,

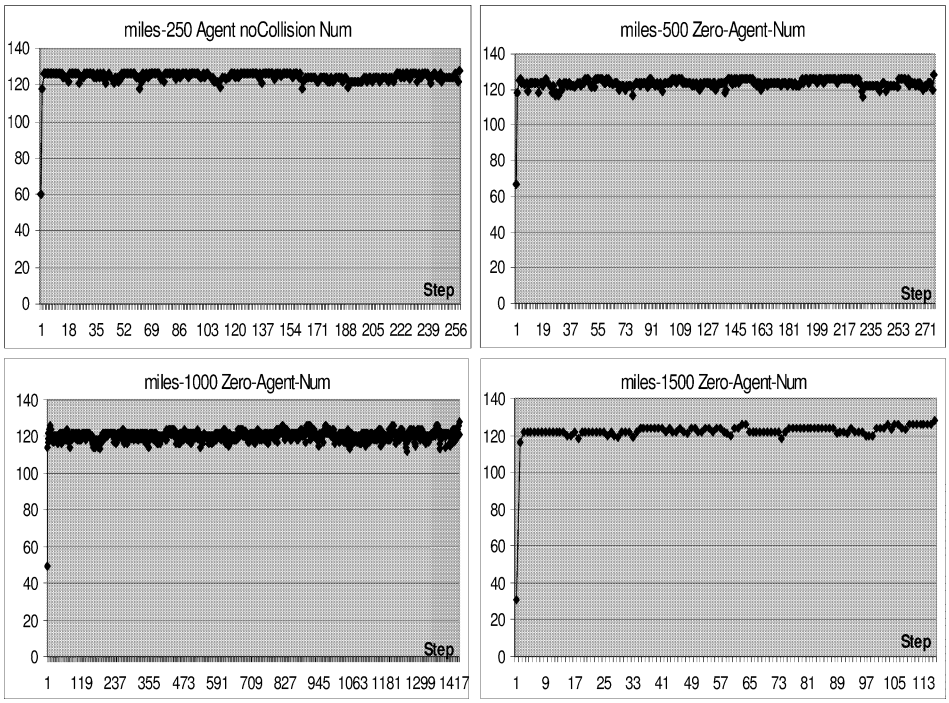

Fig. 30. *Zero-Agent-Num* at different time steps, obtained from the experimental studies on the *miles* problems.

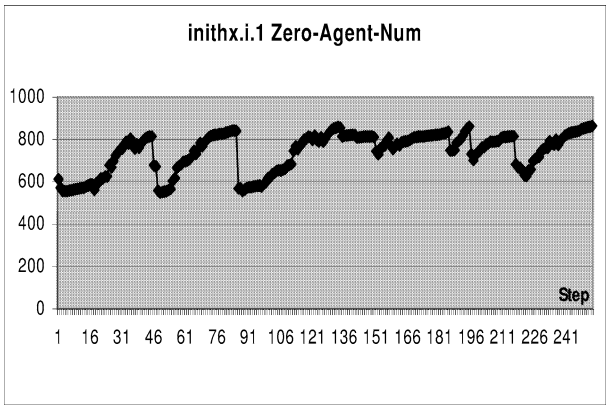

Fig. 31. *Zero-Agent-Num* at each time step in the case of *inithx.i.1*.

meaning that we can get an approximate solution with about 120 (95%) variables satisfying the constraints at anytime after 3 steps.

Observation 4.11. In Fig. 31, *Zero-Agent-Num* increases during the first 45 steps and then falls down. This process is repeated for several times before it finally reaches an exact

solution state. The speed of convergence is not as high as that in other problems. This indicates that not all the problems can converge to 80% at the first step and to 95% at the third step, while using the ERA method. The performance is, to some extent, affected by the structure of the problem.

5. Discussion

In this section, we will compare the basic as well as extended ERA approach to the existing heuristic and distributed approaches, and discuss their distinct features and advantages.

5.1. Comparison with min-conflicts heuristics

The proposed ERA approach differs from the min-conflicts approach in the following aspects:

(1) In the min-conflicts hill-climbing system reported in [27], the system chooses a variable at each step that is currently in conflict and reassign its value by searching the space of possible assignments and selecting the one with the minimum total conflicts. The hill-climbing system can get trapped in a local minimum (note that the same phenomenon can also be observed from the GDS network for constraint satisfaction). On the other hand, in our approach, an agent is given a chance to select a *random-move* behavior according to its probability, and hence it is capable of escaping from a local trap. In our present work, we also note that the extent to which the agents can most effectively avoid the local minima and improve their search efficiency is determined by the probabilities (i.e., behavior selection probabilities) of the *least-move* (as well as *better-least-move*) and *random-move* behaviors.

(2) Another system introduced in [27] is called *informed backtracking*. It arguments a standard backtracking method with the *min-conflicts* ordering of the variables and values. This system attempts to find a sequence of repairs, such that no variable is repaired more than once. If there is no way to repair a variable without violating a previously repaired variable, the algorithm backtracks. It incrementally extends a consistent partial assignment in the same way as a constructive backtracking program, however, it uses information from the initial assignment to guide its search.
The key distinction between this approach and ours is that our approach does not require backtracking. As stated by Minton et al. [27], their system trades search efficiency for completeness; for large-scale problems, terminating in a no-solution report will take a very long time.

(3) In both *min-conflicts* hill-climbing and informed backtracking systems proposed in [27], the key is to compute and *order* the choice of variables and values to consider. It requires to test all related constraints for each variable and to test all its possible values.
This step is similar to the *RemoveFrom* and *AddTo* operations in our approach, except that we only test a selected position (one value for each variable) and do not sort the

variables. The use of the ordering heuristic can lead to excessive assignment evaluation preprocessing and therefore will increase the computational cost at each step.

(4) In our present approach, we examine the use of a *fewer-conflicts* repair, by introducing the *better-move* behavior, that requires only one violation number evaluation for each variable. The empirical evidence has shown that the use of the high-priority *better-move* when combined with other behaviors can achieve more efficient results. We believe that the reason that using the currently-available min-conflicts value at each step can compromise the systems performance is because the min-conflicts values quickly reduce the number of inconsistencies for some variables but at the same time also increase the difficulties (e.g., local minima) for other variables.

5.2. Comparison with Yokoo et al.'s distributed constraint satisfaction

Our multi-agent approach has several fundamental distinctions from Yokoo et al.'s distributed constraint satisfaction approach, as listed below:

(1) Yokoo et al.'s approach does not require a global broadcasting mechanism or data structure. It allows agents to communicate their constraints to others by sending and receiving messages such as *ok?*, and *nogood*. In other words, their methods handle the violation checking among agents (variables) through agent-to-agent message exchanges, such that each agent knows all instantiated variables relevant to its own variable.

In our approach, the notion of agent-to-agent communication is implicit—we assume that for violation updating, each agent (representing the value of a variable) is 'informed' about the values from relevant agents (representing the values of relevant variables) either by means of accessing an $n \times n$ look-up memory table or via pairwise value exchange—both implementations enable an agent to obtain the same information, but the latter can introduce significant communication overhead costs (i.e., longer *cycles* required [43][2]) to the agents.

As the communication in Yokoo et al.'s approach is quite distributed, we believe that their approach will work well under a large number of constrained conditions.

(2) In the *asynchronous weak-commitment search* algorithm developed by Yokoo et al. [42, 43], a consistent partial solution is incrementally extended until a complete solution is found. When there exists no value for a variable that satisfies all the constraints between the variables included in the partial solution, this algorithm abandons the whole partial solution and then constructs a new one. Although *asynchronous weak-commitment search* is more efficient than *asynchronous backtracking*, abandoning partial solutions after one failure can still be costly. In the case of the ERA approach, the high-level control mechanism for maintaining or abandoning consistent partial solutions does not exist.

Yokoo et al. [42] have also developed a non-backtracking algorithm called *distributed breakout*, which provides a distributed implementation for the conventional breakout.

[2] As stated in [43], "one drawback of this model is that it does not take into account the costs of communication".

Table 5
Comparison (in averaged number of cycles) between ERA and Yokoo et al.'s distribution constraint satisfaction in solving benchmark n-queen problems

n	Asynchronous backtracking	Asynchronous backtracking with min-conflicts heuristic	Asynchronous weak-commitment	ERA
100	510	504	51	22
1,000	–	324	30	18
2,000	–	–	–	30

(3) In *asynchronous weak-commitment search*, each agent utilizes the *min-conflicts* heuristic as mentioned in Section 5.1 to select a value from those consistent with the *agent_view* (those values that satisfy the constraints with variables of high-priority agents, i.e., value-message senders).
On the other hand, the ERA approach utilizes a combination of value-selection heuristics that involves a *better-move* behavior for efficiently finding *fewer-conflicts* repairs.
(4) As related to the above two remarks, the *asynchronous weak-commitment search* and *asynchronous backtracking* algorithms are designed to achieve completeness and thus the steps of backtracking and incremental solution constructing/abandoning are necessary, whereas the ERA approach is aimed at more efficiently finding an *approximate* solution, which is useful when the amount of time available for an exact solution is limited.
(5) Last but not the least, we have also systematically compared the performance of the ERA system with that of Yokoo et al.'s algorithms, namely, asynchronous backtracking, asynchronous backtracking with min-conflicts heuristic, and asynchronous weak-commitment, in solving benchmark n-queen problems where $n = 100, 1000, 2000$, respectively [43]. The averaged numbers of cycles used in each case are summarized in Table 5. We can establish that as demonstrated in solving the benchmark n-queen problems, EAR is an effective approach and the number of of cycles used in the ERA system is competitive with those by Yokoo et al.'s approach, given that our formulation utilizes different behavior prioritization and violation checking schemes. Note that the '–' symbol in the table indicates that the data item is presently unavailable.

In summary, as complementary to each other, *both Yokoo et al.'s asynchronous approach and the ERA approach can be very efficient and robust when applied in the right context.* For instance, in some practical applications such as distributed telecommunication networks, Yokoo et al.'s formulation involving agent information exchange offers a natural way of modeling and solving the distributed CSP, whereas in CSPs that do not lend themselves so well to partitioning variables and constraints into sub-problems, the ERA formulation becomes straightforward to implement and execute. A distinct feature of Yokoo et al.'s asynchronous approach is, like other standard backtracking techniques, its completeness, whereas the feature of the EAR approach lies in its efficiency and robustness in obtaining an approximate solution within a few time steps (although it empirically always produces an exact solution when enough time steps are allows). The ERA approach

is not guaranteed to be complete since it involves random moves. Another feature of the ERA approach is that its strategies are quite easy to implement.

5.3. Remarks on partial constraint satisfaction

Partial constraint satisfaction is a very desirable way of solving CSPs that are either overconstrained or too difficult to solve [41]. It is also extremely useful in situations where we want to find the best solution obtainable within fixed resource bounds or in real-time. Freuder and Wallace [11] are the pioneers in systematically studying the effectiveness of a set of partial constraint satisfaction techniques using random problems of varying structural parameters. The investigated techniques included basic branch and bound, backjumping, backmarking, pruning with arc consistency counts, and forward checking. Based on the measures of constraint checks and total time to obtain an optimal partial solution, forward checking was found to be the most effective. Also of general interest is that their work has offered a model of partial constraint satisfaction problems (PCSPs) involving a standard CSP, a partially ordered space of alternative problems, and a notion of distances between these problems and the original CSP.

Our present work attempts to develop, and empirically examine, an efficient technique that is capable of generating partial constraint satisfaction solutions. This work shares the same motivation as that of Freuder and Wallace's work [11,41], and also emphasizes that the costs of calculating (communicating) and accessing constraint violation information should be carefully considered in developing a practically efficient technique. That is also part of the reason why much attention in our work has been paid to (1) the use of environmentally updated and recorded violation numbers (without testing from the scratch for each variable) and (2) the effectiveness of the *better-move* behavior in finding an approximate solution.

5.4. Remarks on agent information and communication for conflict-check

Yokoo et al.'s approach and the ERA approach have a common thread; both formulations employ multiple agents that reside in an *environment* of variables and constraints (although in the ERA approach, the environment also contains violation information, which is analogous to the 'artificial pheromone' in an *ant system* [8,9]) and make their own decisions in terms of how the values of certain local variables should be searched and selected in the process of obtaining a globally consistent solution.

Nevertheless, it should be pointed out that the present implementations of the two approaches differ from each other in the way in which the agents record and access their conflict-check information. The former utilizes a sophisticated communication protocol to enable the agents representing different groups of variables and constraints to exchange their values. By doing so, the agents are capable of evaluating constraint conflict status with respect to other relevant agents (variables). On the other hand, our implementation utilizes a feature of agent current-value broadcast to enable other agents to compare with their values and to update the violation numbers in their local environment. Although the formulations may seem different, the objectives as well as effects of them are fundamentally similar. The reasons that we decided to use value broadcast and sharing are threefold: First, the

implementation can make use of a common storage space of complexity O(n) where n corresponds to the number of variables and by doing so avoid introducing the same space requirement to every agent; secondly, it can reduce the overhead costs incurred during the pairwise information exchange, which can be quite significant; and thirdly, since our ERA method extensively uses *fewer-conflicts* moves, such behaviors can be triggered based on only one possible violation evaluation instead of n assignment evaluations, and hence the access to such a broadcast information source is not demanding.

5.5. Remarks on sequential-iteration implementation

Theoretically, the ERA method using swarm-like agents can be implemented in a parallel fashion, however, in light of our resource limitation, we used a sequential computation implementation to simulate the multi-agent concurrent or synchronous actions and to test the effectiveness of our approach.

Our sequential simulation utilizes a global simulated clock, called *time step*. The state of the environment as well as the agents (i.e., the positions of the agents) will be changed only at each discrete time step. In order to simulate the concurrent or synchronous actions of the agents at time step k, we let the individual agents perform their cycles of behavior selection, value selection, and violation updating. In so doing, the agents are dispatched in a sequential fashion. Once this is completed, the state of the system will then be refreshed with the new positions of the agents corresponding to the newly-selected values, and thereafter, the simulated clock will be incremented to $k + 1$.

Here, it is worth mentioning that apart from the fact that our implementation simulates the operations of a parallel system, the empirical results of our sequential ERA implementation are still comparable to those reported in [27,43], if we evaluate the performance using the measures of number of constraint checks, as introduced by Freuder and Wallace [11], and space complexity.

6. Summary

In this paper, we have described a multi-agent oriented approach to solving constraint satisfaction problems, such as n-queen problems and coloring problems. The key ideas behind this approach rest on three notions: Environment, Reactive rules, and Agents (ERA). Each agent can only sense its local environment (i.e., violation numbers) and apply some behavioral rules for governing its value-selection moves. The environment records and updates the local values that are computed and affected according to the moves of individual agents (analogous to the idea of laying 'artificial pheromone' in an *ant system* [8,9]).

In solving a CSP with either a basic or an extended ERA method, each agent represents a variable and its position corresponds to a value assignment for the variable. The environment for the whole multi-agent system contains all the possible domain values for the problem, and at the same time, it also records the *violation numbers* for all the positions. An agent can move within its row, which represents its domain. So far, we have

introduced three reactive behaviors: *better-move*, *least-move*, and *random-move*. The move of an agent will affect the violation numbers of other rows in the environment.

While formally describing the ERA approach as well as its properties, we have also presented several empirical studies that examine how the basic algorithm can be extended to effectively find a solution to an n-queen or a coloring problem. Some practical rules for behavioral settings have been established following our observations from the experiments.

We have compared the ERA approach with some of the existing heuristic or distributed approaches in order to highlight their main features and limitations. Some features of the ERA approach include:

(1) The move of an agent will affect the whole environment. Thus, the interaction among agents is indirectly carried out through the medium of their environment. In this sense, we may regard that the agents can self-organize themselves in finding a solution.
(2) The performance of this approach in solving CSPs for approximate solutions is efficient. It can find a reasonably good solution in just few steps. This property is quite useful in situations where a solution is required with a hard deadline.
(3) The ERA approach is also open and flexible: We may set or combine various reactive behaviors for each agent, or modify their parameter settings.

Although we have tested several n-queen problems and coloring problems in our present work, we hope that in our future work we will be able to further improve the ERA approach by explicitly introducing communication and/or cooperation mechanisms and to discover new properties of this approach in solving other types of CSPs.

Acknowledgements

We would like to thank the reviewers and associate editor for their valuable comments and suggestions, which have led us to examine several related issues and at the same time make the paper easier to follow. We also want to thank Xiadong Jin for proof-reading our paper. Our work has been partially supported by an HKBU FRG research grant.

Appendix A

In what follows, we will present the proofs for the ERA correctness theorems. In the proofs, we will adopt symbol '$\rightarrow$' to represent 'such that' and symbol '$\Rightarrow$' to denote 'imply'.

A.1. Proof for Theorem 2.1

Proof. There two sections of the algorithm that will have an effect on the violation numbers: Initialization (Section 1 in Fig. 13) and the running (Section 2 in Fig. 13). Therefore, our proof can be divided into two sections.

(1) Stage 1: Initialization

(a) Lines 1–5 in Fig. 13.

Before we place the agents into the environment, because there is no agent in it, there will be no assignment. In such a case, there is no constraint for each domain value. The precondition $(\forall a_i \in A, i \neq y)\ (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y)$ is *false*, so $(\forall a_i \in A, i \neq y, a_i\ one)\ (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow \langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t)$ is `true`.

(b) Lines 6–8.

To each agent, the *Initialize* function is defined in Fig. 9, and the operations on violation are in the *AddTo* function. After adding a_i to position (g,i), if position (x,y) is still a *zero-position*, that means

$$
\begin{aligned}
& \mathit{Attack}((g,i),(x,y)) \text{ is } \mathit{false} \\
\Rightarrow\ & \nexists t\ C(R_t) \text{ between } X_i \text{ and } X_y \vee \\
& \exists t\ C(R_t) \text{ between } X_i \text{ and } X_y \wedge \\
& \langle e(g,i).value, e(x,y).value\rangle \in C(R_t). \\
\Rightarrow\ & \big(\exists t\ C(R_t) \text{ between } X_i \text{ and } X_y \rightarrow \\
& \langle e(g,i).value, e(x,y).value\rangle \in C(R_t)\big). \\
\Rightarrow\ & (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow \\
& \langle e(g,i).value, e(x,y).value\rangle \in C(R_t).
\end{aligned}
$$

After all agents are placed onto the environment, if $\exists(x,y) \in$ environment, position (x,y) is still a *zero-position*, that means:

$$
\begin{aligned}
& (\forall a_i \in A, i \neq y)\ ((\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow \\
& \langle e(g,i).value, e(x,y).value\rangle \in C(R_t)) \\
\Rightarrow\ & (\forall a_i \in A, i \neq y)\ (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow \\
& \langle e(g,i).value, e(x,y).value\rangle \in C(R_t).
\end{aligned}
$$

So Theorem 2.1 is `true` in this section.

(2) Stage 2: Running

We need to prove: If Theorem 2.1 is `true` before line 15, Theorem 2.1 is still `true` after line 15–16. We will prove it in two steps: (1) Theorem 2.1 is `true` before line 15 $\rightarrow$ Theorem 2.1 is `true` after line 15; (2) Theorem 2.1 is `true` after line 15 $\rightarrow$ Theorem 2.1 is `true` after line 16.

[Theorem 2.1 is `true`.]

(a) Line 15 *RemoveFrom*$(a_i.x,i)$.

After remove a_i from $(a_i.x,i)$, if $\exists(x,y) \in$ environment, (x,y) is still a *zero-position*, that means:

$$
\begin{aligned}
& \mathit{Attack}((a_i.x,i),(x,y)) \text{ is } \mathit{false} \\
\Rightarrow\ & \nexists t\ C(R_t) \text{ between } X_i \text{ and } X_y \vee \\
& \exists t\ C(R_t) \text{ between } X_i \text{ and } X_y \wedge \\
& \langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t). \\
\Rightarrow\ & \big(\exists t\ C(R_t) \text{ between } X_i \text{ and } X_y \rightarrow \\
& \langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t)\big). \\
\Rightarrow\ & (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow \\
& \langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t). \qquad [-\mathrm{I}]
\end{aligned}
$$

Because during this process, other agents do not move, and Theorem 2.1 is `true` before this process. So all these agents

$$\begin{aligned}&(\forall a_j \in A,\ j \neq y \wedge j \neq i)\ (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y)\\ \rightarrow\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t) \text{ is } \texttt{true}.\end{aligned}$$

Combining with [–I], we have:

$$\begin{aligned}&(\forall a_i \in A, i \neq y)\ \big((\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow\\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t)\big).\\ \Rightarrow\ &(\forall a_i \in A, i \neq y)\ (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow\\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t).\end{aligned}$$

[Theorem 2.1 is `true`.]

(b) Line 16 $AddTo(a_i.x, i)$.
After adding a_i to $(a_i.x, i)$, if $\exists(x, y) \in$ environment, (x, y) is still a *zero-position*, that means:

$$\begin{aligned}&Attack\big((a_i.x,i),(x,y)\big) \text{ is } false\\ \Rightarrow\ &\nexists t\ C(R_t) \text{ between } X_i \text{ and } X_y \vee\\ &\exists t\ C(R_t) \text{ between } X_i \text{ and } X_y \wedge\\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t).\\ \Rightarrow\ &\big(\exists t\ C(R_t) \text{ between } X_i \text{ and } X_y \rightarrow\\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t)\big).\\ \Rightarrow\ &(\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow\\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t). \qquad [-\mathrm{I}]\end{aligned}$$

Because during the process of $AddTo(a_i.x, i)$, other agents do not move, and Theorem 2.1 is `true` before this process. So all other agents

$$\begin{aligned}&(\forall a_j \in A,\ j \neq y \wedge j \neq i)\ (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y)\\ \rightarrow\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t) \text{ is } \texttt{true}.\end{aligned}$$

Combining with [–I], we get:

$$\begin{aligned}&(\forall a_i \in A, i \neq y)\ \big((\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow\\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t)\big).\\ \Rightarrow\ &(\forall a_i \in A, i \neq y)\ (\exists t \in [1,m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_y) \rightarrow\\ &\langle e(a_i.x,i).value, e(x,y).value\rangle \in C(R_t).\end{aligned}$$

[Theorem 2.1 is `true`.]

So during the running process, Theorem 2.1 is `true`.

In conclusion, Theorem 2.1 is `true`. □

A.2. Proof for Theorem 2.2

Proof. We will prove it based on Definition 1.2 and Theorem 2.1.

(1) $X_i = e(a_i.x, i).value \in D_i$, so S is an n-tuple that $S \in D_1 \times D_2 \times \cdots \times D_n$.

(2) For each $u \in [1, m]$, $C(R_u) \in \boldsymbol{C}$, suppose $\exists g, h \in [1, n]$ that $R_u = D_g \times D_h$,

$$\begin{aligned} & e(a_g.x, g).violation = 0 \\ \Rightarrow\ & \big((\forall a_i \in A, i \neq g)\ (\exists t \in [1, m])\ (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_g) \rightarrow \\ & \langle e(a_i.x, i).value, e(a_g.x, g).value\rangle \in C(R_t)\big), \end{aligned}$$

(based on Theorem 2.1)

then

$$\begin{aligned} & (C(R_u) \in \boldsymbol{C}) \wedge \big((\forall a_i \in A, i \neq g)\ (\exists t \in [1, m]) \\ & (C(R_t) \in \boldsymbol{C}) \wedge (R_t = D_i \times D_g) \rightarrow \\ & \langle e(a_i.x, i).value, e(a_g.x, g).value\rangle \in C(R_t)\big) \\ \Rightarrow\ & \langle e(a_h.x, h).value, e(a_g.x, g).value\rangle \in C(R_t). \end{aligned}$$

We have:

$$\begin{aligned} & (C(R_u) \in \boldsymbol{C}) \wedge e(a_g.x, g).violation = 0 \rightarrow \\ & \langle e(a_h.x, h).value, e(a_g.x, g).value\rangle \in C(R_t). \end{aligned}$$

So for

$$\begin{aligned} & \forall C(R_u) \in \boldsymbol{C}, R_u = D_g \times D_h, \\ & \quad (\langle e(a_h.x, h).value, e(a_g.x, g).value\rangle \subseteq S) \wedge \\ & \quad \big(\langle e(a_h.x, h).value, e(a_g.x, g).value\rangle \in C(R_u)\big)\ \text{is } \texttt{true}. \end{aligned}$$

So the assignment of $S = \langle X_1, X_2, \ldots, X_n\rangle$, $X_i = e(a_i.x, i).value$, is an exact solution when the system terminates at *condition-1*: $(\forall a_i \in A) e(a_i.x, i).violation = 0$. □

References

[1] R. Barták, Heuristics and stochastic algorithms, http://ktilinux.ms.mff.cuni.cz/~bartak/constraints/stochastic.html.

[2] J.R. Bitner, E. Reingold, Backtrack programming techniques, Comm. ACM 18 (11) (1991) 651–656.

[3] M. Bruynooghe, Solving combinatorial search problems by intelligent backtracking, Inform. Process. Lett. 12 (1) (1981) 36–39.

[4] F.C. Chow, J.L. Hennessy, The priority-based coloring method to register allocation, ACM Trans. Programming Languages and Systems 12 (4) (1990) 501–536.

[5] B. Clement, E. Durfee, Scheduling high-level tasks among cooperative agents, in: Proceedings of the Third International Conference on Multi-Agent Systems, Paris, 1998.

[6] M.C. Cooper, An optimal k-consistency algorithm, Artificial Intelligence 41 (1989) 89–95.

[7] K. Decker, J. Li, Coordinated hospital patient scheduling, in: Proceedings of the Third International Conference on Multi-Agent Systems, Paris, 1998.

[8] M. Dorigo, G. Di Caro, L.M. Gambardella, Ant algorithms for discrete optimization, Artificial Life 5 (2) (1999) 137–172.

[9] M. Dorigo, V. Maniezzo, A. Colorni, The ant system: An autocatalytic optimizing process, Technical Report No. 91-016 Revised, Politecnico di Milano, Italy, 1991.

[10] J. Ferber, Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence, Addison-Wesley, New York, 1999, pp. 31–35.

[11] E.C. Freuder, R.J. Wallace, Partial constraint satisfaction, Artificial Intelligence 58 (1–3) (1992) 21–70.

[12] P. Galinier, J.-K. Hao, Tabu search for maximal constraint satisfaction problems, in: Proceedings of CP-97, Linz, Austria, 1997, pp. 196–208.

[13] A. Gamst, Some lower bounds for a class of frequency assignment problems, IEEE Trans. Vehicular Technology 35 (1) (1986) 8–14.
[14] M.R. Garey, D.S. Johnson, H.C. So, An application of graph coloring to printed circuit testing, IEEE Trans. Circuits Systems CAS-23 (1976) 591–599.
[15] J. Gaschnig, Performance measurement and analysis of certain search algorithm, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1979.
[16] J. Gu, Efficient local search for very large-scale satisfiability problem, SIGART Bull. 3 (1992) 8–12.
[17] C.C. Han, C.H. Lee, Comments on Mohr and Henderson's path consistency algorithm, Artificial Intelligence 36 (1988) 125–130.
[18] A. Homaifar, J. Turner, S. Ali, The n-queen problem and genetic algorithms, in: Proceedings of IEEE SOUTHEASTCON-92, Birmingham, AL, 1992, pp. 262–267.
[19] V. Kumar, Algorithm for constraint satisfaction problem: A survey, AI Magazine 13 (1) (1992) 32–44.
[20] F.T. Leighton, A graph coloring algorithm for large scheduling problems, J. Res. Nat. Bureau Standards 84 (1979) 489–506.
[21] J. Liu, Autonomous Agents and Multi-Agent Systems: Explorations in Learning, Self-Organization, and Adaptive Computation, World Scientific, Singapore, 2001.
[22] J. Liu, J. Han, ALIFE: A multi-agent computing paradigm for constraint satisfaction problems, Internat. J. Pattern Recogn. Artificial Intell. 15 (3) (2001) 475–491.
[23] J. Liu, Y.Y. Tang, Adaptive segmentation with distributed behavior-based agents, IEEE Trans. Pattern Anal. Machine Intell. 21 (6) (1999) 544–551.
[24] J. Liu, Y.Y. Tang, Y. Cao, An evolutionary autonomous agents approach to image feature extraction, IEEE Trans. Evolutionary Comput. 1 (2) (1997) 141–158.
[25] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1) (1977) 99–118.
[26] J. Mandziuk, Solving the n-queen problem with a binary Hopfield-type network: Synchronous and asynchronous models, Biol. Cybernet. 72 (1995) 439–446.
[27] S. Minton, M.D. Johnston, A.B. Philips, P. Laird, Minimizing conflicts: A heuristic repair method for constraint-satisfaction and scheduling problems, Artificial Intelligence 58 (1992) 161–205.
[28] R. Mohr, T.C. Henderson, Arc and path consistency revisited, Artificial Intelligence 28 (1986) 225–233.
[29] B. Nadel, Some applications of the constraint satisfaction problem, Technical Report CSC-90-008, Computer Science Department, Wayne State University, Detroit, MI, 1990.
[30] H. Ogawa, Labeled point pattern matching by Delaunay triangulation and maximal cliques, Pattern Recognition 19 (1) (1986) 35–40.
[31] S. Park, E. Durfee, W. Birmingham, Emergent properties of a market based digital library with strategic agents, in: Proceedings of the Third International Conference on Multi-Agent Systems, Paris, 1998.
[32] V.G. Paul, Introduction to constraint satisfaction, Lecture 3: Search Order in CSPs and Solution Synthesis, 1997.
[33] W. Rosiers, M. Bruynooghe, On the equivalence of constraint satisfaction problem, in: Proceedings of AIMSA-86, North-Holland, Amsterdam, 1989.
[34] F. Rossi, C. Petrie, On the equivalence of constraint satisfaction problems, Technical Report ACT-AI-222-89, MCC, Austin, TX, 1989.
[35] F. Seghrouchni, S. Haddad, A recursive model for distributed planning, in: Proceedings of the Second International Conference on Multi-Agent Systems, Kyoto, Japan, 1996.
[36] B. Selman, H. Kautz, B. Cohen, Local search strategies for satisfiability testing, in: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, American Mathematical Society, Providence, RI, 1996, pp. 521–532.
[37] B. Selman, H. Kautz, An empirical study of greedy local search for satisfiability testing, in: Proceedings of AAAI-93, Washington, DC, MIT Press, Cambridge, MA, 1993, pp. 46–51.
[38] R. Sosic, J. Gu, Efficient local search with conflict minimization: A case study of the n-queen problem, IEEE Trans. Knowledge and Data Engineering 6 (5) (1994) 661–668.
[39] R. Stallman, G.J. Sussman, Forward reasoning and dependency directed backtracking, Artificial Intelligence 9 (2) (1977) 135–196.
[40] Swarm Development Group, Swarm simulation system, http://www.swarm.org/index.html.
[41] R. Wallace, Analysis of heuristic methods for partial constraint satisfaction problem, in: Principles and Practice of Constraint Programming (CP-1996), Cambridge, MA, 1996, pp. 482–496.

[42] M. Yokoo, K. Hirayama, Algorithms for distributed constraint satisfaction: A review, Autonomous Agents and Multi-Agent Systems 3 (2000) 185–207.
[43] M. Yokoo, E.H. Durfee, T. Ishida, K. Kuwabara, The distributed constraint satisfaction problem: Formalization and algorithms, IEEE Trans. Knowledge and Data Engineering 10 (5) (1998) 673–685.
[44] M. Yokoo, K. Hirayama, Distributed constraint satisfaction algorithm for complex local problems, in: Proceedings of the Third International Conference on Multi-Agent Systems, Paris, 1998, pp. 372–379.
[45] M. Yokoo, Y. Kitamura, Multi-Agent real-time-a* with selection: Introducing competition in cooperative search, in: Proceedings of the Second International Conference on Multi-Agent Systems, Kyoto, Japan, 1996.
[46] http://dimacs.rutgers.edu/Challenges/Seventh/#PC.
[47] http://mat.gsia.cmu.edu/COLOR/instances.html.
[48] http://mscmga.ms.ic.ac.uk/jeb/orlib/colourinfo.html.