Emergence from Local Evaluation Function

Han Jing
Cai Qingsheng

SFI WORKING PAPER: 2002-08-036

SFI Working Papers contain accounts of scientific work of the author(s) and do not necessarily represent the views of the Santa Fe Institute. We accept papers intended for publication in peer-reviewed journals or proceedings volumes, but not papers that have already appeared in print. Except for papers by our external faculty, papers must be based on work done at SFI, inspired by an invited visit to or collaboration at SFI, or funded by an SFI grant.

©NOTICE: This working paper is included by permission of the contributing author(s) as a means to ensure timely distribution of the scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the author(s). It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author’s copyright. These works may be reposted only with the explicit permission of the copyright holder.

www.santafe.edu
Emergence from Local Evaluation Function

Han Jing and Cai Qingsheng
Department of Computer Science and Technology
University of Science and Technology of China
hanjing@mail.ustc.edu.cn

This paper presents a new look on emergence from the aspect of using local evaluation functions to solve traditional computer problems. We first translate the Constraint Satisfaction Problem (CSP) to a multi-agent system, and then show how a global solution emerges from the system in which every agent uses a local evaluation function to decide its action, while comparing with other traditional algorithms, such as Local search and Simulated Annealing which use global evaluation functions. We also give some computer experimental results on large-scale N-queen problems and \( k \)-Coloring problems, and show that emergence only depends on problem instance, not details of agent settings, i.e. in some CSPs, the system can self-organize to a global solution, but can not in some other CSPs no matter what settings the agents are.

1. Motivation

Emergence is regarded as a significant property of Complex Systems. People find emergent complex behaviors from local interactions in nature and human society. In man-made systems, such as the computer programs, we can also find emergent behaviors that exceed what the programmer has expected, such as *Game of Life* \[1\] and *Boid* \[2\]. Computer scientists have tried to use the emergent computational ability to solve problems. Some examples are the Genetic Algorithm \[3\], and Ant Colony Optimization \[4\] algorithms.

So, what is the relationship between computer problems and complex systems? Can we translate a computer problem to a complex system, and how? What is emergence in Problem solving in computer science? How can we construct a system that will emerge useful properties? Can we design an algorithm that only have local rules but can lead the system to evolve to a global solution? What affect that the system self-organize to a global solution?

This paper tries to answer the above questions by showing a simple and easy-understood case. We design an algorithm in which agent’s behaviors only depend on the local evaluation functions (evaluation function sometimes is called objective function, fitness function, cost function or energy \( E \)) and use its emergent property to solve a class of computer problems (CSPs), comparing to the traditional methods that use global evaluation functions. We first translate a CSP to a multi-agent system,
2 Emergence from Local Evaluation Function

and then design a set of rules (algorithm) to fit the concept of Complex System and emergence: gives each agent some simple strategies based on a local evaluation function, as well as the running schedule of the whole system.

The original idea of this paper is from our Alife model for solving the N-queen problem [5], we then extract the basic multi-agent model from the Alife model, called AER(Agent-Environment-Rule) Model for solving binary CSPs [6]. In this paper, we find out the most essential attribute of the Alife&AER Model by comparing to other algorithms, and try to extend it to a more general rule for algorithm design — Local Evaluation Function leads to global solution.

The outline of this paper is as follows. In section 2 we introduce CSP and the solving computer algorithms, and then describe the method of modeling a CSP to a multi-agent system. In Section 3, we discuss how to design the strategy of each agent based on the local evaluation function, comparing to global evaluation function. We classify the current basic Single-solution algorithms into two classes according to the locality of their evaluation functions. By giving some computer experiments, section 4 validates that the system using local evaluation function can self-organize to a global solution of large-scale N-queen problems and k-coloring problems, and shows that it only depends on the problem instances. Section 5 is the conclusions.

2. MODELING CSPS TO MULTI-AGENT SYSTEMS

2.1. CSPs

Problem Solving is an old and important topic in Artificial Intelligence (AI) of computer science. People hope to use the cheap and high performance computing resource of computers to solve hard problems, such as the combinatorial problems: scheduling, planning, design, graph problems, diagnostic, computational linguistics, configuration, decision support, hardware verification, molecular biology, qualitative reasoning, real-time systems, resource allocation, robotics, temporal reasoning, vision, visualization, user interfaces, etc. All of them require finding a solution from all of the possible combinations of values. Some of them need a best solution, that called Optimization Problems; and the others need a solution or prove that no solution exists, called Decision problem or Constraint Satisfaction Problems (CSPs) [7].

A constraint satisfaction problem, $P$, consists of
1. A finite set of variables, $X$, identified by the natural numbers, $1,2,...,n$, $X = \{x_1, x_2, ..., x_n\}$.
2. A domain set, containing a finite and discrete domain for each variable. $D = \{D_1, D_2, ..., D_n\}$, for all $i \in [1, 2, ..., n]$, $x_i \in D_i$.
3. A constraint set, $C = \{C(R_1), C(R_2), ..., C(R_m)\}$. $R_i$ is the $i$th constraint. 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 for the variables in $R_i$.
4. $P$ is a finite discrete CSP, if all $D_i$ in $D$ are discrete and finite.
5. $P$ is a binary CSP, if each constraint is either unary or binary. It is possible to convert a CSP with n-ary constraints to another equivalent binary CSP[7]).

A solution, $S$, is a complete assignment to all variables such that the assignment satisfies all the constraints. (Any complete assignment can be regarded as an approximate solution.)
1. $S$ is an ordered set, $S = \langle v_1, v_2, \ldots, v_n \rangle$, $S \in D_1 \times D_2 \times \ldots \times D_n$.

2. $(\forall j \in [1, m])((\exists S' \subseteq S) \land (S' \in C (R_j)))$ is true. $S'$ is also an ordered set.

3. If a solution $S$ exists for the CSP, it is called satisfiable.

This paper focuses on the finite discrete binary CSP. $\langle X_i, d \rangle$ denotes an assignment of $X_i$, $X_i = d$, $d \in D_i$.

**$k$-Coloring Problem** is the (vertex) coloring problem, which is useful in various applications, such as time tabling and scheduling, frequency assignment, register allocation [8]. In this problem, we need to color each vertex of the graph (by a 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 has variables for each of the nodes of the graph. The domain of each variable is the given set of $m$ colors. For each pair of nodes linked by an edge, there is a binary constraint between the corresponding variables that disallows identical assignments. Here is an example: $X = \{ V_1, V_2, V_3, V_4 \}$, $D_1 = D_2 = D_3 = D_4 = \{ \text{green, red, blue} \}$, constraints are $V_1 \neq V_2$, $V_1 \neq V_3$, $V_1 \neq V_4$ and $V_2 \neq V_3$. Figure 1 is an example of a solution for a 3-coloring problem.

$k$-Coloring problem is NP-hard, which means it will be impossible to find an algorithm to solve the $k$-Coloring problem in polynomial time.

**N-queen Problem** is another classical benchmark of CSPs. It requires us to place $n$ indistinguishable queens on an $N \times N$ chessboard so that no two queens are on the same row, or the same column, or the same diagonal. There exist solutions for the N-queen problem with $n$ greater than or equal to 4 [9] (see Fig.2). The equivalent CSP is $X = \{ X_1, X_2, \ldots, X_n \}$. $X_i$ refers to the position of the queen in the $i$\textsuperscript{th} row.

$D = \{ D_1, D_2, \ldots, D_n \}$, $\forall i, D_i = [1, n]$, $C = \{ C (R_u) \mid \forall i, j \in [1, n], C (R_u) = \{ \langle s, c \rangle \mid b \in D_i, c \in D_j, b \neq c, i-j \neq b-c, i-j \neq c-b \} \}$

Although N-queen problem is not NP-hard, it is still an attractive problem because it is a scalable problem that good for testing algorithm performance and has many applications [9], such as VLSI testing, scheduling, resource management, etc.

### 2.2. Algorithms to CSPs

There are several basic algorithms and a lot of improved methods for solving combinatoral problems (including CSPs). In this paper, we will only focus on the main basic algorithms in this area, while improvement or variance of the basic algorithm will not be discussed because its main idea is as same as the basic one.

- According to how many (approximate) solutions the system seeking concurrently, we can divide the current algorithms into two classes:
4 Emergence from Local Evaluation Function

**Single-solution Algorithms**: the system is searching for only one (approximate) solution in a time. Examples are Backtracking [10], Local Search [11-15,20-21], Simulated Annealing [16], Extremal Optimization (EO) [17], Alife&AER model, etc. We can very easy to include them in a simple multi-agent system. Therefore, this paper will only focus on the one-copy algorithms.

**Multi-solution Algorithms**: the system is parallel searching for several (approximate) solutions, which means that there are several copies of the system. Examples are Genetic Algorithm and Ants Colony Optimization. If considering non-basic algorithms, we can also put those hybrid algorithms in this category, such as Portfolio of Algorithms [18] and Cooperative Search [19].

- From the viewpoint of the style of search, there are two types: Systematic search and Generate—test [7]. Backtracking and Ants Colony Optimization belong to the former, while Local Search, Simulated Annealing, Genetic Algorithm, Extremal Optimization, Alife&AER model fall into the latter.

**Backtracking** (depth-first search) 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 go back to the most recently assigned variable and perform the process again. Backtracking cannot solve large-scale problems because its searching space increases sharply with the problem size and it is not an efficient algorithm. We will not discuss Backtracking in the following sections in this paper because it is not a multi-agent system like algorithm.

**Generate—test (GT)** generates a possible combination of all variables (an approximate solution) and then check if it satisfies all the constraints, i.e., check if it is a solution. The simplest way to generate a complete assignment to all variables is randomly select a value to each variable. However, this is a very low efficient way. There have been some research efforts on making the generator smarter. The most popular idea is Local Search [9]. For many large-scale CSPs, it always gives better results than systematic Backtracking paradigm. It generates an initial complete (but possibly inconsistent) assignment to all variables and then incrementally uses hill climbing [20] or min-conflict [11] to move to a solution with better evaluation value among its current (approximate) solution’s neighborhood (see detail in 3.1), until a solution is found. The earliest Local Search algorithm, GSAT [20], use Hill-climbing heuristic. To avoid falling into the local-optima, it sometimes performs stop-and-restart, random walk [21] and Tabu search [14-15]. Simulated Annealing [16], a famous algorithm inspired by the roughly analogous physical process of heating and then slowly cooling a substance to obtain a strong crystalline structure, is a Monte Carlo approach for combinatorial problems. It can be regarded as one heuristic of the stochastic Local Search. EO and Alife&AER model also generate a random initial complete assignment, and then change one variable’s value in each improvement until a solution is found or reach to maximum tried step number. However, they improve the assignment of a variable based on some local evaluation functions, which is the main difference from local search and Simulated Annealing.

All the above algorithms have their advantages and drawbacks, and no algorithm can solve all CSPs efficiently. For the small size problem or systematic problems, we can use backtracking to find a solution easily while in large-scale hard problems often use generate-test. Generate-test algorithms are not complete, i.e., can not prove there is no solution for a CSP and even worse, sometimes can not find a solution when the CSP is satisfiable. Even that, people still found that it would be better to find a solution quickly by an efficient generate-test algorithms than using backtracking to prove the satisfiability of a CSP.
FIGURE 3 Classifications of basic algorithms of solving CSPs. This paper will focus on the intersection of single-solution and Generate-test (the gloom area algorithms).

2.3. MODELING A CSP TO A MULTI-AGENT SYSTEM

The concept of “agent” as a computational entity that is autonomous has been studied for some years in artificial intelligence and more widely used in computer science. Multi-agent systems are computational systems in which several agents interact or work together to reach goals. Agents in these systems may be homogeneous or heterogeneous, and may have common goals or distinct goals. Research on these systems generally focuses on areas such as simulations of social and biological systems, problem solving, communication, collective robotics, electronic commerce on Internet, etc. The cellular automata [22] and Ants Colony Optimization can be regarded as a multi-agent system, by regarding the cell and ant as an agent. Multi-agent Systems focus on the complexity of a group of agents, not an individual agent’s function. This embodies the idea of ‘more is different’ [23].

The concept of agent and multi-agent system can be defined as follows.

Agent, \( a \), is a virtual entity that basically has the following properties:

1. Be able to act and get information from the system.
2. Finite possible states of Agent.
3. Be driven by some objectives.
4. Have some behavioral strategies.

Multi-agent system (MAS) is a system that including the following elements:

1. A set of Agents, \( A = \{a_1, a_2, \ldots, a_n\} \).
2. Interaction between agents.
3. A set of rules, \( R \), for the interactions between the agents.

FIGURE 4 A Multi-agent System (right) for a 3-coloring problem (left).
Emergence from Local Evaluation Function

It is straightforward to construct a Multi-agent System (MAS) for a CSP:

\[
\begin{align*}
\text{CSP} & \iff \text{MAS} \\
\text{Variable } X_i & \iff \text{Agent } a_i \\
\text{Domain } D_i \text{ of a variable } X_i & \iff \text{Possible states of agent } a_i \\
\text{Assignment of variable } < X_i, d > & \iff \text{State of agent } < a_i, \text{state}, s_i > \\
\text{Constraint between two variables} & \iff \text{Interaction between two agents} \\
\text{Complete assignment} & \iff \text{Current state of the whole system } S_{\text{sys}}=<s_1, ..., s_n> \\
\text{Solution} & \iff \text{The solution state of the whole system} \\
\text{Process of finding a solution by Generate-test algorithm} & \iff \text{Evolution of MAS} \\
\text{Heuristic of search} & \iff \text{Strategy of Agent move from one state to another state}
\end{align*}
\]

Let's use the example of 4-queen Problem to illustrate how a CSP can be translated to a multi-agent system (see Figure 5):

\[X_i\text{, position of queen in the }i\text{th row, is represented by an agent } a_i \text{ (or represented by several agents in the Alife Model)[5]); So state of an agent is equal to the X-position of the queen, and the possible states of each agent are the same:}\{1,2,3,4\}, \text{which are possible positions of each queen. When } a_i \text{ in a position of the }i\text{th row and } s_i\text{th column, denoted as } (i, s_i), \text{the current state of } a_i \text{ is } s_i, \text{equally means } X_i=s_i.

Interaction exists in each two agents because there are constraints between each two variables. (But in the coloring problems, it is not that case: only those agents that corresponding to linked nodes will have interaction. See Figure 4.)

Figure 5 represents a complete assignment of \{<X_1, 3>, <X_2, 4>, <X_3, 1>, <X_4, 2>)\}, which is not an exact solution because there are some violated constraints: \(X_1\) conflicts \(X_2\), \(X_1\) conflicts \(X_3\), \(X_3\) conflicts \(X_4\), \(X_2\) conflicts \(X_4\).

Thus, the system needs to evolve, and improve the current approximate solution. Agent will change from its current state to another state that will improve the whole solution. The objective of the whole system is to get a solution state, which requires each agent in a state that can satisfy all related constraints:

The process of evolution of the system can be regarded as the process of find a solution. According to the first step of Generate-test algorithms, generating the initial complete assignment to all variables, all agents in the multi-agent system will get their initial states: placed in a position on the chessboard. Then in the following time steps, one or several agents will move to another position (change to another state) by some strategies. Once the system reaches to a solution state that all agents are in non-conflict positions, a solution is found (see Figure 6): The solution state of the system: \(S_{\text{sys}}=<3, 4, 1, 2>\).

\[\text{FIGURE 5} \quad \text{Multi-agent model for 4-queen problem. Current complete assignment is } <3, 4, 1, 2>.
\]

\[\text{FIGURE 6} \quad \text{Solution state of the multi-agent system for 4-queen problem. Current complete assignment is } <3, 4, 1, 2>.
\]
How does the system evolve to the solution state? This is a key point of problem solving and MAS. While regarding the current Generate-test algorithms as a MAS, there exist some heuristics for agent’s movement from one state to another state (assignment of variable changes). However, what heuristics can lead the whole system to a global solution state? Can we define some selfish or greedy strategies of a single agent? Can we define some local interaction between agents and let them self-organize to a solution state? Can we use ‘emergence’ of the MAS to solve a problem? These are what we are going to talk in the next section.

3. AGENT STRATEGY: LOCAL EVALUATION FUNCTION

Once we model the CSP to be a multi-agent system, we need to design a set of rules for the evolution, which is called ‘algorithm’ in computer science. The core of the algorithm is the strategies of how the whole system (complete assignment) change from one state to another state, i.e., how variables change their assigned values.

In the Generate-test style, there are two kinds of methods: use the global evaluation function to select the best system state in its current neighborhood, or use the local evaluation function to determine movement of a single agent (change of variable assignment). We call the former GEF methods that are centralized and from-top-to-bottom, while the latter LEF methods focus on the individual (a single agent) that is decentralized and from-bottom-to-top. LEF is a way for the system to self-organize to a solution state without the high-level control and guide. It is an emergence of the MAS.

This section will describe GEF methods (Local Search and Simulated Annealing) and LEF methods (EO and Alife&AER Model) in the language of MAS. We also compare the main difference of them by showing the case of N-queen problem.

3.1. TRADITIONAL METHODS: GLOBAL EVALUATION FUNCTIONS

The idea behind GEF of Generate-test style search methods (Local Search and Simulated Annealing) are as follows:

1. Each agent picks a random state from its possible state set.
2. Choose a neighbor state and the system moves to that state:
   - Compute the evaluation value of all neighbors of the current system state by the global evaluation function, and then select the best state;
   - Alternatively, performs random walk with some probability.
3. Repeat 2 through 3 until a solution is found or reach the max. tries.
4. Return the current state as the (approximate) solution.

**Neighborhood** is the search space of the current point (state). There are varieties of neighborhood structure. For example, one simple method to construct the neighbor is to change any one variable’s assigned value. Therefore, the difference between the current state and its neighbors are one variable’s assigned value. By this way, we will get $\sum_i(|D_i|-1)$ neighbors for each state. As a case, one neighbor of a state showed in Figure 7(a) of N-queen problem is showed in Figure 7(b). We will use this neighborhood structure in the following discussion (please note that this is not the most efficient way for solving N-queen problem, but the easiest way to show the basic idea and easy-understood).

The neighbor structure will affect the efficiency of the algorithm [24]: the larger the neighborhood, the better is the quality of the local optimal solutions, but longer
Emergence from Local Evaluation Function

time it takes to search the neighborhood. However, this is not in the scope of this paper and will not be discussed here.

**Global Evaluation Function** evaluates how good the current system state $S_{sys}$ (approximate solution) is. It is used to rate the potential of each move to lead to a global solution. In Simulated Annealing, it is the energy or cost function of the current system state. It guides search in the solving process and points out how good of each neighbor and only the best/better system state can be accepted to replace the current system state. Undoubtedly, evaluation function plays an important role in the algorithm and a good definition of it will increase the performance of the algorithm.

There are a lot of ways to define the evaluation function. One of the general ways for binary CSPs is to count the number of dissatisfied constraints in the current complete assignment. It is called global evaluation function because it consider the whole system including all agents. The number of dissatisfied constraints for a solution state is zero: $E_{global}(solution\ state) = 0$. In language of physics, the solution state is the ground state of the n-body system. Minimizing the evaluation value is the goal of the solving process. For N-queen problem, the evaluation function can be defined as (1) and Figure 7 shows a movement in some case.

$$E_{global}(S_{sys}) = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \left[ s_i = s_j \lor s_i - s_j = i - j \lor s_i - s_j = j - i \right] \quad ---- (1)$$

For the k-Coloring problem, the evaluation function can be defined as (2):

$$E_{global}(S_{sys}) = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \left[ s_i = s_j \land \text{linked}(node_i, node_j) \right] \quad ---- (2)$$

where \( \text{linked}(node_i, node_j) \) returns true if \( node_i \) links to \( node_j \), which means there exists a constraint between two variables $X_i$ and $X_j$.

---

1 Sometimes the solution state corresponds to the maximal value. It depends on how the evaluation function is defined. In this paper, the minimum corresponds to the solution state.

2 Please note that a state with better evaluation value does not always mean that it is more close to the solution state.
Centralized control in GEF methods is obvious: in each step, the system checks all neighbors and selects the one who has the best evaluation value as the new state. Therefore, no agents here are autonomous that they all obey the choice of the centralized controller, and they are selfless that they will give up their movement if there is some other agent’s movement can make the global solution better. This is a typical programming manner of from-up-to-bottom. This requires the programmer knows the problem very well and efficiency depends on the intelligence of the designer. There is no emergence and all are in controlled.

3.2. LOCAL EVALUATION FUNCTION METHODS

The idea behind LEF methods of Generate-test style search algorithms (EO and Alife&AER Model) are as follows:

1. Each agent picks a random state from its possible state set.
2. In a step, for each agent \( a_i \), do the followings parallel or sequentially:
   - Choose a new state and \( a_i \) move to that state:
     - Compute the evaluation value of all/some states of \( a_i \) by the local evaluation function, and then select the best one;
     - Alternatively, performs random walk with some probability.
3. Repeat 2 through 3 until a solution is found or reach the max. tries.
4. Return the current state as the (approximate) solution.

Local Evaluation Function here expresses the object of a single agent. The local evaluation functions focus on a single agent and return a better value if the agent is in a better state that can satisfy more related constraints of it. Therefore, for each agent, it has its own local evaluation function and it wants to minimize the evaluation function value, i.e., maximize its own profit, without considering the whole system (the overall profit of the system).

On the other hand, it is natural to think about the MAS as a physical system: the interactions between agents force agents switching from one state to another state. If there is a constraint between two variables \( X_i \) and \( X_j \), agent \( a_i \) and \( a_j \) will receive a force from each other if their states cannot satisfy the constraint between them. In the LEF methods, force just likes the repulsion that makes these two agents to move to another place. For example, in the N-queen problem, if two agents stand in a same column, each of them will receive a conflict force, and this force makes them to move to another place with less conflict force. While in the \( k \)-Coloring problems, two agents (linked nodes) in the same state (color) will receive the conflict force from each other.

The following is one possible definition of the local evaluation function for each agent. Formula (3) is for agent \( a_i \) in the N-queen problem and formula (4) is for the \( k \)-Coloring problem. Hence, all agents try to move to another state for minimizing the evaluation function. When all of them get to the minimum state (here the value is zero), i.e., no conflict state, a solution is found.

\[
E_{\text{local}}(S_{\text{sys}}^{(i)}) = \frac{1}{2} \sum_{j=1}^{N} [s_i = s_j \lor s_i - s_j = i - j \lor s_i - s_j = j - i] \quad ---- (3)
\]

\[
E_{\text{local}}(S_{\text{sys}}^{(i)}) = \frac{1}{2} \sum_{j=1}^{N} [s_j = s_j \land \text{linked}(\text{node}_i, \text{node}_j)] \quad ---- (4)
\]

where \( S_{\text{sys}}^{(i)} = \{s_j | j \in \{1,2,\ldots,n\} \land j \neq i \land \text{a constraint between } X_i \text{ and } X_j \} \).
Emergence from Local Evaluation Function

They are different from the *global evaluation functions* of formula (1) and formula (2). The relationship between local and global evaluation functions is:

\[ E_{\text{global}}(S_{\text{sys}}) = \sum_{i=1}^{N} E_{\text{local}}(S_{\text{sys}}^{(i)}) \] ---- (5)

So, when all agents in the minimum state, we have \( \forall i \ E_{\text{local}}(S_{\text{sys}}^{(i)}) = 0 \Rightarrow E_{\text{global}}(S_{\text{sys}}) = 0 \). This is one of the preconditions that prove those agents using *local evaluation functions* can achieve a global goal.

**Agent Strategies**: for each agent, it will have some strategies based on the *local evaluation function*: Least-move, Better-move, Random-move, etc. Agents will select one of them in one step with probabilities of least-p, better-p and random-p.

- **Least-move**: the agent moves to a state with minimal evaluation function value. The least-move behavior can be expressed as follows:
  \[ a_i.\text{state} = s_i = d, \text{ where } d \in |D_i|, \forall d' \in D_i \land d' \neq d \ E_{\text{local}}(S_{\text{sys}}^{(i)} d) \leq E_{\text{local}}(S_{\text{sys}}^{(i)} d'). \]

- **Better-move**: the agent randomly select a state and moves to it if it is better than the current state. Otherwise, stay.
  \[ d = \text{random}(D_i), \text{ where } \text{random}(D) \text{ returns a random item in } D. \]
  \[ a_i.\text{state} = s_i = \begin{cases} d, & \text{if } E_{\text{local}}(S_{\text{sys}}^{(i)} d) < E_{\text{local}}(S_{\text{sys}}^{(i)} s_i) \\ s_i, & \text{otherwise}. \end{cases} \]

- **Random-move**: the random walk, to avoid stuck into the *local-optima*.
  \[ a_i.\text{state} = s_i = \text{random}(D_i). \]

Figure 8 shows the movements (by random-move, better-move and least-move) of the 4-queen problem in a step. Number in position \((i, j)\) is the value of formula (3) \( E_{\text{global}}(S_{\text{sys}}^{(i)} s_i') \), where \( s_i' \) is obtained by moving agent \( a_i \) to position \((i, j)\) from the current state \( s_i \) while all the other agents stay still. Here (b) is the same state as Figure 7(a), but they have different behaviors according to different schedules and evaluation functions.

\[ (a) \ t_0: \text{after Initialization} \quad (b) \ t_1: \text{after } a_3 \text{ random-move} \]

\[ (c) \ t_1: \text{after } a_1 \text{ better-move} \quad (d) \ t_1: \text{after } a_4 \text{ least-move} \]

**FIGURE 8** Moving strategies of agents in 4-queen Problem.

All agents move in a time step \( t_1 \).

---

3 \( S_{\text{sys}}^{(i)} d \) means in the state subset of agent \( a_i, S_{\text{sys}}^{(i)}, a_i.\text{state} \) is replaced by \( d \).
Decentralized control in LEF methods is also obvious: in each step, the system dispatches all agents parallel or sequentially (see Figure 8), while in EO algorithm, the system dispatches the agent who in a worst state (state with a maximal evaluation function value). Agents are autonomous and decide its behavior based on its local evaluation function that only considers its own profit. Therefore, all agents here are selfish, or called greedy, nearsighted. If the whole system can achieve a global goal in the process of all agents pursuing their own objective, we call emergence happen and the system self-organize to a global goal because there are no centralized control. This is a typical programming manner of from-bottom-to-up.

![Diagram](image)

**FIGURE 9** Relationship between GEF and LEF, and emergence.

### 4. EXPERIMENTAL RESULTS OF LEF METHODS

This section we present experimental results on a set of N-queen problems and k-coloring problems (benchmarks from DIMACS, the Center for Discrete Mathematics and Theoretical Computer Science [http://mat.gsia.cmu.edu/COLOR/instances.html](http://mat.gsia.cmu.edu/COLOR/instances.html)). Results in section 4.1 show that LEF methods (Alife&AER Model) can self-organize to a global solution and efficient for both N-queen problems and k-coloring problems. We also further examine what affect the system self-organize to a solution from the following aspects:

1. Does the settings of agent will affect the performance?
2. If the system under a certain setting can solve a problem \( P \), can the system under different settings solve \( P \)?
3. If the system under a certain setting cannot solve a problem \( P \), can the system under different settings solve \( P \)?

In the experiments, we will initialize all agents with the same set of parameters, i.e., \( \langle \text{least-p}, \text{better-p}, \text{random-p} \rangle \). Specifically, \( \forall i \in [1, 2, \ldots, n], \langle a_i.\text{least-p}, a_i.\text{better-p} : a_i.\text{random-p} = \text{least-p} : \text{better-p} : \text{random-p} \rangle. \)

If not specified, experiments will use the FBLR mode for agent strategies and middle-first dispatch order.

Demo of Alife Model for solving N-queen problems can be downloaded from [http://www.santafe.edu/~hanjing/dl/nq.exe](http://www.santafe.edu/~hanjing/dl/nq.exe). For more experiments on algorithmic techniques, please read our papers [4][5].

---

4 We will elucidate the strategy mode and dispatch order in section 4.2.
12 Emergence from Local Evaluation Function

4.1. Runtime Distribution

First, we examine performance of finding a solution. We first give the CPU runtime of finding an exact solution and then we measure the runtime as operation counts[25], here, that is the number of dispatch step of finding a solution. We gives the distribution of steps for finding a solution through thousands runs.

Experiment 1: (N-queen problem). \( n = \{100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000\} \), least-p: random-p = 1: \( n \). 10 runs. CPU: PII.

Experiment 2: 1000-queen problem. least-p: random-p = 1: 1.5\( n \). 10000 runs.

Experiment 3: (\( k \)-coloring problems) least-p: random-p = 1: 1.5\( n \). 10 runs. CPU: PIII

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Problem (node, link), optimal-color, source} & \text{Runtime (second)} \\
\hline
\text{fpsol2.1.1.col (496,11654),65, REG} & \text{Avg.} 0.10 & \text{Max.} 1 & \text{Min.} 0 \\
\hline
\end{array}
\]
<table>
<thead>
<tr>
<th>File Path</th>
<th>Avg. Runtime (seconds)</th>
<th>REG</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpsol2.i.2.col (451,8691), 30, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>fpsol2.i.3.col (425,8688), 30, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>inithx.i.1.col (864,18707), 54, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>inithx.i.2.col (645,13979), 31, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>inithx.i.3.col (621,13969), 31, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>mulsol.i.1.col (197,3925), 49, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>mulsol.i.2.col (188,3885), 31, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>mulsol.i.3.col (184,3916), 31, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>mulsol.i.4.col (185,3946), 31, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>mulsol.i.5.col (186,3973), 31, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>zeroin.i.1.col (211,4100), 49, REG</td>
<td>0.10</td>
<td>1</td>
</tr>
<tr>
<td>zeroin.i.2.col (211,3541), 30, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>zeroin.i.3.col (206,3540), 30, REG</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>anna.col (138,493), 11, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>david.col (87,406), 11, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>homer.col (561,1629), 13, SGB</td>
<td>0.10</td>
<td>1</td>
</tr>
<tr>
<td>huck.col (74,301), 11, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>le450_5b.col (450,5734), 5, LEI</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>jean.col (80,254), 10, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>games120.col (120,638), 9, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>miles1000.col (128,3216), 42, SGB</td>
<td>5.40</td>
<td>13</td>
</tr>
<tr>
<td>miles1500.col (128,5198), 73, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>miles250.col (128,387), 8, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>miles500.col (128,1170), 20, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>miles750.col (128,2113), 31, SGB</td>
<td>1.40</td>
<td>4</td>
</tr>
<tr>
<td>queen5_5.col (25,160), 5, SGB</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>queen6_6.col (36,290), 7, SGB</td>
<td>0.10</td>
<td>1</td>
</tr>
<tr>
<td>queen7_7.col (49,476), 7, SGB</td>
<td>0.10</td>
<td>1</td>
</tr>
<tr>
<td>queen8_12.col (96,1368), 12, SGB</td>
<td>3.90</td>
<td>11</td>
</tr>
<tr>
<td>queen8_8.col (64,728), 9, SGB</td>
<td>1.60</td>
<td>5</td>
</tr>
<tr>
<td>queen9_9.col (81,2112), 10, SGB</td>
<td>4.00</td>
<td>7</td>
</tr>
<tr>
<td>queen11_11.col (121,3960), 11, SGB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>miyel3.col (11,20), 4, MYC</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>miyel4.col (23,71), 5, MYC</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>miyel5.col (47,236), 6, MYC</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>miyel6.col (95,755), 7, MYC</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>miyel7.col (191,2360), 8, MYC</td>
<td>0.00</td>
<td>0</td>
</tr>
</tbody>
</table>

TABLE 1  Avg. runtime (second) of k-coloring problems in Experiment 3

5 '-' means the problem can not be solved in an hour.
Emergence from Local Evaluation Function

Experiment 4: ($k$-coloring problems). Anna and Miles1500. least-$p$: random-$p = 1$: $1.5n$. 1000/5000 runs.

From Experiments 1&3, we note that the Alife&AER Model can solve large-scale computational problems, such as N-queen problems and $k$-Coloring Problems in a reasonable CPU time. The runtime of N-queen problem increases polynomially with $n$. However, we also find that some problem instances of $k$-Coloring problem is very easy to solve by Alife&AER Model in less than 1s, while some hard problem (queen11_11.col and le450_5b.col) can not be solved in an hour. We will further examine what affect the system self-organize to a solution in the following section.

Because random walk is one strategy of agents, Alife&AER Model is a randomized algorithm. We try to find out the distributions in Experiments 2&4. Figure 11 shows that the total dispatch step number of finding a solution in 1000-queen problem obeys an exponential distribution, while in $k$-coloring problems, different problem instances have different pattern in their distributions.

Now we make a conclusion that LEF methods can self-organize to a global solution, as same as the system using GEF.

4.2. SYSTEM DETAILS VS. PROBLEM INSTANCE

In this section, we will show what happen if we change the details of the system by computer experiments.

- $M$ Agent/variables
  
  If we introduce the rule of ‘survival of the fitness’ to basic AER model, say, the Alife model, $m$ agents are used to represent a variable and wash out those bad strategies agents by the competition (details see our paper [5]).

Experiment 5: (N-queen problem) $n = \{100, 150, 200, 250, \ldots, 600\}$, $m = 3$, 10 run. CPU: PIV.

From Figure 14, we note that the system is still able to find a solution. But the runtime is longer than $m=1$ (Figure 10). This will not surprise us because it needs more computation in a step when $m$ is larger. The purpose of setting $m>1$ is trying to
let the system itself to evolve to a good setting for agents if we don’t know the problem so well.

![Figure 14](image.png)

**FIGURE 14** Result of Experiment 5. The runtime(s) and dispatch step number in the process of finding the first solution for N-queen problems.

- **Probability of Random walk**
  How randomness of the system should be? In each step, an agent will select a movement from *random-move*, *least-move* (and *better-move*) based on its corresponding probabilities. We test the performance under different ratio of probability of *random-move* to probability of *least-move* of agents.

**Experiment 6:** (N-queen problem) \( n\{1000, 2000, 3000, 4000, 5000, 6000, 7000\} \), *least-p : random-p* = \[0, 0.5n, n, 1.5n, 2n, \infty\], LR, 10 runs. CPU: PII

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 0 )</th>
<th>0.5N</th>
<th>N</th>
<th>1.5N</th>
<th>2N</th>
<th>( \infty )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>-</td>
<td>17.13</td>
<td>12.13</td>
<td><strong>8.63</strong></td>
<td>15.75</td>
<td>-</td>
</tr>
<tr>
<td>2000</td>
<td>-</td>
<td>91.38</td>
<td>35</td>
<td><strong>31.13</strong></td>
<td>46.88</td>
<td>-</td>
</tr>
<tr>
<td>3000</td>
<td>-</td>
<td>124.88</td>
<td>120.75</td>
<td><strong>79.75</strong></td>
<td>113.13</td>
<td>-</td>
</tr>
<tr>
<td>4000</td>
<td>-</td>
<td>270.88</td>
<td>151.88</td>
<td><strong>150.25</strong></td>
<td>187.25</td>
<td>-</td>
</tr>
<tr>
<td>5000</td>
<td>-</td>
<td>598.25</td>
<td>451.13</td>
<td><strong>370.75</strong></td>
<td>530.38</td>
<td>-</td>
</tr>
<tr>
<td>6000</td>
<td>-</td>
<td>641.38</td>
<td>722.88</td>
<td>496.75</td>
<td><strong>426.38</strong></td>
<td>-</td>
</tr>
<tr>
<td>7000</td>
<td>-</td>
<td>3478.75</td>
<td>1476.5</td>
<td><strong>1011.5</strong></td>
<td>1971.5</td>
<td>-</td>
</tr>
</tbody>
</table>

**TABLE 2** Result of Experiment 6: Avg. runtime under different ratio of *least-p* to *random-p*. CPU: PII

If the ratio of *least-p* to *random-p* is zero, *Alife&AER Model* returns to a pure *random walk* and the system is chaotic, nearly impossible to self-organize to a global solution; If the ratio is \( \infty \), it returns to a greedy algorithm and falls into the *local optimum* very soon. Between \((0, \infty)\), the system is able to self-organize to a global solution although the runtime is different. The best setting seems to be \( 1:1.5n \) in table 2.
16 Emergence from Local Evaluation Function

Experiment 7: \((k\text{-coloring problems). least-p: random-p} = \{0.5n, n, 1.5n, 2n\}, 10\) runs. CPU: PII

<table>
<thead>
<tr>
<th>Avg. runtime (s)</th>
<th>Mile s500</th>
<th>Mile s750</th>
<th>Mile s100</th>
<th>Mile s150</th>
<th>Anna</th>
<th>Dav id</th>
<th>Inith x.i.1</th>
<th>queen 11_11</th>
<th>le45 0_5b</th>
</tr>
</thead>
<tbody>
<tr>
<td>least-p: random-p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.5 N</td>
<td>0.00</td>
<td>0.75</td>
<td>1.50</td>
<td>0.00</td>
<td>0.00</td>
<td>2.38</td>
<td>0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1N</td>
<td>0.00</td>
<td>0.50</td>
<td>1.13</td>
<td>0.00</td>
<td>0.00</td>
<td>0.13</td>
<td>0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1.5 N</td>
<td>0.00</td>
<td>0.20</td>
<td>0.75</td>
<td>0.00</td>
<td>0.00</td>
<td>0.25</td>
<td>0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2N</td>
<td>0.00</td>
<td>0.63</td>
<td>2.50</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

TABLE 3 Result of Experiment 6: Avg. runtime under different ratio of \(\text{least-p}\) to \(\text{random-p}\).

Note that in the \(k\text{-coloring problems}, the ratio also affects the efficiency of the algorithm. But it doesn’t change whether the system can self-organize to a solution or not. A hard instance is still can not be solved (see queen11_11.col and le450_5b.col in Table 1 and Table 3).

- **Strategy Mode**
  
  Better-move is a small computation rule but the quality of search by this way is worse than least-move because it only compares two positions. What’s more, we observe that agents using better-move can always find out a better position to move only in the first few steps. So we develop some strategy modes (L—least-move; B—better-move; R—random-move):
  
  LR: no better-move in agent’s strategy.
  
  rBLR: repeat trying better-move until a better position is found or reaches \(r\) tries. If \(r = 1\), it is the normal mode of BLR.
  
  FrBLR: Better-move is only available in the first step. Agent performs \(r\)BLR in the first step and then LR in the following steps.

Experiment 8: 1000-queen problem. \(\text{least-p: random-p} = 1000, type = \{LR, BLR, 2BLR, 3BLR, 5BLR, FBLR, F2BLR, F3BLR, F5BLR\}. 10\) runs. CPU: PII.

<table>
<thead>
<tr>
<th>Avg. Runtime (second)</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r)</td>
<td>LR</td>
</tr>
<tr>
<td>1</td>
<td>12.25</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 4 Result of Experiment 8: Avg. runtime under different strategy modes.

Experiment 9: \((k\text{-coloring problems). least-p: random-p} = n, \{LR, 2BLR, 3BLR, F2BLR\}, 10\) runs. CPU: PII.
TABLE 5 Result of Experiment 9: Avg. runtime under different strategy modes.

From the above experiments, we note that the strategy mode will affect the runtime of finding a solution if the problem is easy under a certain mode, but not the self-organization: easy problem is still easy, hard problem is still hard.

- **Dispatch order**
  In each step of the system, each agent is dispatched to move once. The order of dispatching all agents can be parallel and sequential. In solving N-queen problem by the sequential way, we can dispatch agents from middle to both sides, or following the ordinal order ‘1, 2, …, n’, or random.

**Experiment 10**: (N-queen problem). n={1000, 2000}, least-p: random-p = n, 10 runs. CPU: PIII.

TABLE 6 Result of Experiment 10: Avg. runtime under different dispatch order
18 Emergence from Local Evaluation Function

As shown in Table 6, the dispatch order affects the runtime of finding a solution. In Figure 15-16, we can see different patterns of solution in different dispatch orders. So the dispatch order will decide the solution pattern too. As same as in the previous setting discussion, it will not affect the self-organization. Experiment show that the system still can not find a solution for queen11_11.col and le450_5b.col in an hour.

Now let’s finish this section by answering the questions we propose in the beginning:

First, the settings of agent will affect the performance, the runtime for finding a solution will various.

Second, if the system under a certain setting can solve a problem $P$, the system under different settings within a certain range can still solve $P$.

Finally, if the system under a certain setting cannot solve a problem $P$, the system under different settings seems also hard to solve $P$.

In conclusion, whether or not the LEF methods self-organize to a global solution depends on the problem instance, not the details of the system.

5. CONCLUSIONS

This paper analyses the relationship between the computer problems and the complex systems by translating a computer problem, CSP, to a multi-agent system. After that, we describe several famous computer algorithms, local search and Simulated Annealing, in the multi-agent framework of CSPs, and show that the traditional algorithms use a global evaluation functions (GEF, computes how good the current system state is, the total number of violated constraints). We then illustrate how the EO and Alife&AER model algorithms use local evaluation function (LEF, computes how good the agent state is, how many constraints it violates) to self-organize to a global solution state. Therefore, the notion in this paper of emergence in computer problem solving is:

If each agent in the system uses LEF, and the whole system can self-organize to a (global solution) state (that can be reached by using the GEF methods), we say emergence happens in this system.

The key point of designing an ‘emergent’ algorithm is to give each agent/variable $a_i$ a local evaluation function $E_{local}(S_{sys}(i))$ and let it evolve by minimizing (or sometimes random walk) $E_{local}(S_{sys}(i))$ in each step, but not minimizing the global evaluation function $E_{global}(S_{sys})$. An obvious precondition is that when all agents are in their minimal position of $E_{local}(S_{sys}(i))$, $E_{global}(S_{sys})$ reaches its minimum.

We use this idea to solve N-queen problem and $k$-coloring problem. Some local rules based on the local evaluation function for each agent are defined according to the problem, as well as the evolving schedule for the system. In the computer experiments, we observe that the LEF method can self-organize to the global solution state of large-scale N-queen problems ($n \leq 8500$) and $k$-coloring problems and the runtime for finding a solution is reasonable. We further explore what affect the system self-organize to a global solution and find that the details of the system only affect the efficiency of finding a solution, but no the self-organization. What decides whether the system can self-organize to a global solution is the problem instance itself. This is an interesting observation and we will study more on what problem instance will affect LEF in the future, as well as the LEF and GEF methods.
ACKNOWLEDGMENTS

This paper is supported by the International Program of Santa Fe Institute and the grant of China National Science Foundation (No.70171052).

REFERENCES


20 Emergence from Local Evaluation Function