Agent Compromises in Distributed Problem Solving

Yi Tang[1,2], Jiming Liu[3], and Xiaolong Jin[3]

[1] Department of Mathematics, Zhongshan University, Guangzhou, China
[2] Department of Mathematics, Guangzhou University, Guangzhou, China
tyi@guangztc.edu.cn,
[3] Department of Computer Science, Hong Kong Baptist University
Kowloon Tong, Hong Kong
{jiming, jxl}@comp.hkbu.edu.hk

Abstract. ERA is a multi-agent oriented method for solving constraint satisfaction problems [5]. In this method, agents make decisions based on the information obtained from their environments in the process of solving a problem. Each agent has three basic behaviors: *least-move*, *better-move*, and *random-move*. The *random-move* is the unique behavior that may help the multi-agent system escape from a local minimum. Although *random-move* is effective, it is not efficient. In this paper, we introduce the notion of agent compromise into ERA and evaluate its effectiveness and efficiency through solving some benchmark Graph Coloring Problems (GCPs). When solving a GCP by ERA, the edges are transformed into two types of constraints: local constraints and neighbor constraints. When the system gets stuck in a local minimum, a compromise of two neighboring agents that have common violated neighbor constraints may be made. The compromise can eliminate the original violated neighbor constraints and make the two agents act as a single agent. Our experimental results show that agent compromise is an effective and efficient technique for guiding a multi-agent system to escape from a local minimum.

Keywords: Agent Compromises, Distributed Constraint Satisfaction Problem, Graph Coloring Problem, Distributed GCP Solving

1 Introduction

A distributed constraint satisfaction problem (distributed CSP) is a constraint satisfaction problem where variables and constraints are distributed to multiple agents. Several multi-agent system based problem solving methods have been proposed for solving distributed CSPs [5] [9]. In these methods, each agent deals with a subproblem that is specified by some variables and related constraints of the given problem.

ERA (Environment, Reactive rules, and Agents) is an example of this kind of methods [5]. In ERA, each agent represents several variables and stays in a

lattice-like environment corresponding to the Cartesian product of the variable domains. The agent tries to move to a certain lattice so as to make as many as possible constraints satisfied. To do so, at each step, it will select one of the basic reactive behaviors: *better-move*, *least-move*, and *random-move*.

Graph Coloring Problem (GCP) is a class of CSPs. In this paper, we will introduce the notion of agent compromise into ERA for solving GCPs. In solving a GCP by ERA , each agent represents a group of vertices in the given GCP and decides the colors of these vertices. The edges in the GCP are transformed into two types of constraints: local constraints and neighbor constraints. The agent needs to minimize the number of unsatisfied neighbor constraints in order to solve the GCP. When the system gets stuck in a local minimum, we will adopt not only the technique of *random-move* but also agent compromise to lead the system to escape from the local minimum. An agent compromise is made by two neighboring agents that have common violated neighbor constraints. It aims at eliminating the original violated neighbor constraints. When an agent compromise is made, the corresponding two agents will act as a single agent (we call it *compromise agent*) to make their decisions, i.e., decide their common local behaviors. The compromise will be abandoned after a reaction is finished.

The rest of the paper is organized as follows. In Section 2, we describe the background of the paper. In Section 3, we introduce the notion of agent compromise and integrate the agent compromise strategy to ERA. In section 4, we present our experimental results and discussions. Section 5 is the conclusion of the paper.

2 Background

2.1 Graph Coloring Problems

Graph Coloring Problems (GCPs) are NP-hard. Many practical problems can be transformed into GCPs [4]. There are several types of GCPs [10]. One of them is as follows:

Given an undirected graph $G = (V, E)$ where V is a set of vertices and E is a set of pairs of vertices called edges, and a set of available colors $S = \{1, 2, ..., k\}$. The problem is to find a mapping C: $V \rightarrow S$ such that $C(u) \neq C(v)$ if $(u, v) \in E$.

In short, a GCP is to determine whether or not it is feasible to color a graph with a certain number of colors. Fig. 1 shows a simple GCP instance. This graph contains 6 vertices and 7 edges. It can be colored with 3 colors.

2.2 Definitions

Assume that the vertices in a GCP have been divided into n groups [4]: $\{V_1, V_2, ..., V_n\}$, where $V_i \cap V_j = \phi$ and $\cup_{i=1}^{n} V_i = V$. We further assume that a set of agents: $\{agent_1, agent_2, ..., agent_n\}$ is used to represent vertex groups. Specifically, $agent_i$ represents the vertices in V_i.

[4] Each vertex can be regarded as a variable whose domain is the color set.

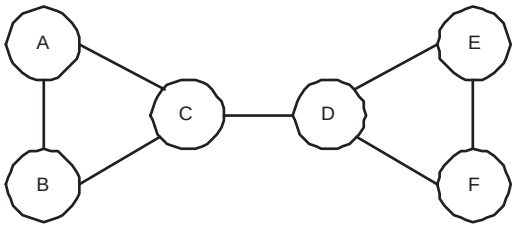

Fig. 1. A 3 colorable graph.

Definition 1. *(Local constraints, neighbor constraints, and local color vector)*

1. *Local constraints of $agent_i$ are the edges among vertices in V_i;*
2. *Neighbor constraints of $agent_i$ are the edges between vertices in V_i and those in $V - V_i$.*
3. *A local color vector of $agent_i$ is a tuple of color assignments of all vertices in V_i that satisfied all local constraints of $agent_i$.*

Definition 2. *(Agent environment)*
The environment of $agent_i$ is composed of lattices corresponding to all local color vectors of $agent_i$. The size of $agent_i$'s environment is m, if and only if $agent_i$ has m local color vectors.

An agent always stays in one of lattices in its environment. An agent stays at a lattice indicates its vertices are assigned colors corresponding to the local color vector.

For example, in Fig. 1, we can divide the vertices into two groups: {A, B, C} and {D, E, F}. Two agents, $agent_1$ and $agent_2$, are used to represent them respectively. Edges AB, BC and AC are local constraints of $agent_1$, and edge CD is its neighbor constraint with $agent_2$. If S = $\{1, 2, 3\}$ is the color set, $(1, 2, 3)$ is a local color vector of $agent_1$ which means A, B, and C are colored with colors 1, 2, and 3, respectively. The local color vectors of $agent_1$ are $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$, and $(3, 2, 1)$. The size of $agent_1$'s environment is 6.

Note that based on the above definitions, the local constraints of $agent_i$ are always satisfied in any states.

3 Distributed GCP Solving by ERA with Agent Compromises

3.1 Reactive Behaviors of Agents

In order to solve a given GCP, at each step, an agent will probabilistically select the following three predefined local reactive behaviors to decide which lattice it will move to [5]:

- *least-move*: the agent moves to a lattice where it has the minimum number of violated neighbor constraints;
- *better-move*: the agent moves to a random lattice where the number of violated neighbor constraints caused is fewer than that at its current lattice;
- *random-move*: the agent randomly moves to a lattice.

When an agent selects a *least-move* or a *better-move*, it may not move to a new lattice because at its current lattice this agent has already a minimum number of violated neighbor constraints or it is in a *zero-position* [5]. When none of the agents can move by a *least-move* or a *better-move*, it means the system gets stuck in a local minimum. The *random-move* behavior is the unique reactive behavior that an agent can use to escape from a local minimum. Different *random-move* ratios, namely, the probability ratios of p_{random_move}:p_{least_move}:p_{better_move}, may yield different performances. In order to explore the optimal *random-move* ratio, we conduct a set of experiments on an instance, *myciel*7.*col*, with different *random-move* ratios by 64 agents. . The results in Tab. 1 show that a relative small ratio is better than others. $5 : 1000 : 1000$ is the optimal one.

Table 1. Comparisons among different *random-move* ratios. The results are obtained by solving an instance, *myciel*7.*col*, with 64 agents. This instance, downloaded from [10], contains 191 vertices and 2,360 edges. It can be colored with 8 colors. We run the ERA algorithm 100 times in each case and calculate the average results.

random-move Ratio	Num. of Movements
2 : 1000 : 1000	235
3 : 1000 : 1000	252
5 : 1000 : 1000	**222**
8 : 1000 : 1000	250
10: 1000 : 1000	235
20: 1000 : 1000	259

The *random-move* behavior is simple and effective, but it is not efficient. Although we can select a suitable *random-move* ratio, we cannot slide over that the results are broken away by the randomness caused by *random-move*. The reason that we need randomness during a problem solving is that it can lead the multi-agent system to escape from a local minimum. There are other techniques to provide the function that lead the system out of a local minimum [7] [8]. In the following, we will propose a new technique for helping agents escape from a local minimum in a distGCP solving.

3.2 Agent Compromises

Why do there exist local minima when solving a given problem? An intuitional explanation is that there is a set of *primary* variables. It is hard to find suitable

values for these variables in a few steps, which satisfy all the related constraints. After assigned values for these *primary* variables, the values of other variables will be easily determined [1]. In the case of distGCP, since each vertex is represented by a certain agent, the characteristic of *primary* variables is partially demonstrated by the *hard* neighbor constraints between agents. If we can make them satisfied as early as possible, we might improve the solving. Motivated by this view, we propose an agent compromise strategy for agents. We equip the agents with the behavior of *make-compromise*.

The compromise of agents is made by two neighboring agents (i.e. the two agents have at least one common neighbor constraint) in the distGCP solving. It can be viewed as a virtual agent whose variables and constraints are inherited from the original two agents. We call the two agents as *component agent*s and the virtual agent as *compromise agent*. The local constraints of *compromise agent* are not only inherited from the local constraints of the two agents, but also from the neighbor constraints between them. Once a compromise has been made, a *compromise agent* is activated and the two *component agent*s are deactivated. The environment of the *compromise agent* is composed of those of the two *component agent*s. The same as an agent in a distGCP solving, when the *compromise agent* stays in its environment, its local constraints are satisfied. Furthermore, the *compromise agent* will inherit some reactive behaviors of the two component agents and determine its local reactive behavior based on its local environment. This implicitly makes the two component agents moving in coherence. The compromise will be abandoned after the above reaction is finished. At the same time, the two *component agent*s are recovered and they inherit the moving results of the *compromise agent*. This means that all neighbor constraints between these two *component agent*s are satisfied.

3.3 The Process of a Make-compromise

When solving a given GCP, if an agent wants to make a compromise with other neighbor agents, it will start the procedure of a *make-compromise*. And, we will call this agent *master*.

The *master* will choose one of its neighbors, called *slave*, to form a compromise agent based on a criterion that the number of violated neighbor constraints between the *master* and the *slave* is the largest. If several *master*s select the same *slave*, the one that has the largest number of violated neighbor constraints with the *slave* will win the competition and form a compromise agent with the *slave* finally. The reactive behaviors of the *compromise agent* can be *least-move*, *better-move*, and *random-move*. After finishing a selected behavior, the compromise is abandoned. The values of the variables in the compromise are returned to the *master* or the *slave*. The neighbor constraints between the *master* and the *slave* are satisfied. The state of the multi-agent system is then updated and the system will continue running from the updated state.

4 Experimental Results and Discussions

4.1 Experiments and Results

In the following experiments, we classify reactive behaviors into two types: *random* and *non-random*, and set the probability ratio of these two types behaviors as 5:2000. We set *better-move* and *least-move* with the same probability. We further introduce the *make-compromise* ratio, denoted by the probability ratio, $p_{make\_compromise} : p_{non\_random}$. For example, if this ratio is 5:100, it means that the probability ratio of the four reactive behaviors p_{random_move}:$p_{make\_compromise}$:p_{least_move}: p_{better_move} is 5:100:950:950. The vertices in each GCP instance are equally partitioned into groups according to the label sequence. A corresponding number of agents will be used to represent these vertex groups.

In the experiments, the asynchronous operations of distributed agents are simulated by means of sequentially dispatching agents and allowing them to sense the current state of their environments. All agents are dispatched randomly and are given the same *random-move* ratio. For simplicity, we assume *least-move* is the unique reactive behavior of a *compromise agent*.

1. Comparisons between different *make-compromises* ratios

   In this set of experiments, we show what is the optimal *make-compromises* ratio by solving an instance, *myciel*7.*col*, in [10]. Tab. 2 shows the results of the experiments. It shows that if choosing the ratio between 5 and 10, the performance is better.

Table 2. Comparisons between different *make-compromises* ratios. The results are obtained by solving an instance, *myciel*7.*col*, with 64 agents. The instance is from [10] and contains 191 vertices and 2,360 edges. It can be colored with 8 colors. We run the solver 100 times in each case and calculate the average number of movements and that of compromises.

make-compromise Ratio $p_{make\_compromise} : p_{non\_random}$	Num. of Movements	Num. of Compromises
-	222	-
1 : 100	182	8
2 : 100	169	13
5 : 100	**141**	**23**
8 : 100	**142**	**33**
10 : 100	**139**	**33**
15 : 100	152	59
30 : 100	146	88
40 : 100	155	110

2. Comparisons between the performances with/without *make-compromises*

   We select 8 instances from [10] to compare the performances of ERA without/with agent compromises. We run each instances in each case 100 times. In each run with compromises, we set the *make-compromise* ratio as 5:100. Tab. 3 shows the experimental results. It shows that ERA can reduce the number of movements on average 26% (ranging from 14% to 44%) if adopting *make-compromise*.

Table 3. Comparisons between different distributed GCP solving with/without *make-compromises*. The results are obtained by solving 8 instances from [10]. We run the ERA algorithm 100 times in each cases. The number of movements(with/without compromises) are on average and the reduction(%) is the percents of the movements reduced after introducing agent compromise.

Instance ID (vertices, edges, colors)	Num. of Agents	Num. of Movements (with/without compromises)	Reduction (%)
anna.col (138, 493, 11)	46	187 / 334	44
jean.col (80, 254, 10)	27	43 / 51	16
miles250.col (128, 387, 8)	43	169 / 296	43
queen5_5.col (25, 160, 5)	8	38 / 44	14
queen6_6.col (36, 290, 7)	12	1,960 / 2,291	14
queen7_7.col (49, 476, 7)	16	2,363 / 2,834	17
myciel6.col (95, 755, 7)	32	51 / 71	28
myciel7.col (191, 2360, 7)	64	141 / 222	36

4.2 Discussions

In order to help agents escape from a local minimum, we endue each agent with the behavior of *make-compromise*. An agent compromise is made by two neighboring agents that have common unsatisfied constraints. It can eliminate these constraints. Behind the introduction of agent compromise is the existence of the *primary* variables. Since these variables are *hard* to set a suitable values that satisfy all related constraints, it is one of the main reasons why the agents get stuck in local minima. The compromise between two agents gives the multi-agent system more chances to set these variables with suitable values. As compared with the *random-move* behavior, the *make-compromise* behavior is addressed the structure of a problem. This is based on that the *hard* constraints are often associated with the *primary* variables, especially when the system has been running a few time steps.

On the other hand, the behavior of a *compromise agent* can be viewed as a partially global behavior, because it is a temporary union of two neighboring agents and the corresponding two *component agent*s can make decisions in co-

herence. According to the results of our experiments, the employment of this kind of global behavior can improve the performance of ERA in solving a GCP.

However, we cannot exempt the agents from the behavior of *random-move*. Although a *compromise agent* can make the violated neighbor constraints satisfied, the variables changed might not belong to the *primary* variables. This may lead to the different reduction of agent movements. We need further study on the structure of a problem.

5 Conclusion

In this paper, we introduce the notion of agent compromise to the ERA method in distributed GCP solving. When solving a given GCP with ERA, we equip each agent with the behavior of *make-compromise*. This behavior provides another ability for agents to escape from a local minimum. As compared with the *random-move* behavior, this behavior integrates two neighboring agents and makes them. moving in coherence. We have examined the compromise strategy in distributed GCP solving. The experimental results show an obvious improvement of ERA with agent compromises in solving some benchmark GCPs.

Acknowledgements. This project is supported in part by a HKBU FRG grant and conducted at the Computer Science Department of HKBU.

References

1. Gomes, C. P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, Journal of automated reasoning, **24**, (2000), 67-100.
2. Gu, J.: Efficient local search for very large-scale satisfiability problem, SIGART Bulletin, **3**, (1992), 8-12.
3. Hoos, H.H., Stutzle, T.: Systematic vs. Local Search for SAT, Proc. of KI-99, LNAI 1701, Springer, (1999), 289-293.
4. Leighton, F.: A graph coloring algorithm for large scheduling problems, Journal of Research of the National Bureau of Standards, **84**, (1979), 489-503
5. Liu, J., Jing, H., Tang, Y.Y.: Multi-agent oriented constraint satisfaction, Artificial Intelligence. **138**, (2002) 101-144.
6. Nareyek, A.: Using global constraints for local search, Constraint Programming and Large Scale Discrete Optimization, DIMACS, **57**, (2001) 9-28.
7. Schuurmans, D., Southey, F., Holte, R. C.: In Proceedings of the 17th International Joint Conference On Artificial Intelligence (IJCAI-01), (2001) 334–341.
8. Shang, Y., Wah,B.W.: A discrete Lagrangian-based global-search method for solving satisfiability problems, Journal of global optimization. **12(1)**, (1998) 61-99.
9. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms, IEEE Transaction on Knowledge and Data Engineering. **10(5)**, (1998) 673-685.
10. http://mat.gsia.cmu.edu/COLOR02