Available online at www.sciencedirect.com

SCIENCE DIRECT®

Cognitive Systems Research 5 (2004) 191–206

Cognitive Systems RESEARCH

www.elsevier.com/locate/cogsys

Multi-agent collaborative service and distributed problem solving

Action editor: Ron Sun

Jiming Liu[a,*], Xiaolong Jin[a], Yi Tang[b]

[a] *Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong*
[b] *Department of Mathematics, Guangzhou University, Guangzhou, China*

Received 27 August 2003; accepted 30 March 2004
Available online 17 June 2004

Abstract

In this paper, we show that collaborative services can be formulated into distributed constraint satisfaction problems. We introduce the notion of multi-agent collaborative service (MACS), and employ a distributed discrete Lagrange multipliers (DDLM) method to automatically handle an MACS task. The DDLM method is based on a distributed multi-agent system. The behaviors of agents are guided by predefined DDLM rules. In order to make it more efficient in achieving a solution state, we incorporate strategies for tuning the Lagrange multipliers. We validate the effectiveness of the DDLM method with benchmark SAT problems. Furthermore, we provide the mathematical properties of DDLM and present the corresponding DDLM algorithms.
© 2004 Published by Elsevier B.V.

Keywords: Multi-agent collaborative services; Distributed problem solving; Constraint satisfaction problems; Satisfiability problems; Lagrange multipliers

1. Introduction

Collaborative service is a process that should be taken in a broad sense. Many tasks in real-life involve collaborative services, such as on-line Web services, computational Grid services, collaborative engineering design (e.g., designing a micro satellite (Yoshida, Teduka, & Nishida, 1999) or a CD player (Mori & Cutkosky, 1998)), e-learning (Loser, Grune, & Hoffmann, 2002), and e-enterprise activity integration. A collaborative service is a complex process where each service participant is required to coordinate with other participants so as to arrange the required subservices for its users. Such a collaborative service normally involves a large number of participants, subservices, and constraints among them that need to be satisfied.

[*] Corresponding author. Fax: +852-3411-7892.
E-mail addresses: jiming@comp.hkbu.edu.hk (J. Liu), jxl@comp.hkbu.edu.hk (X. Jin), ytang@gzhu.edu.cn (Y. Tang).

1389-0417/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.cogsys.2004.03.006

The constraints can be of different types, such as costs, spatial, and temporal constraints.

In this paper, we will show that collaborative services can be formulated into *constraint satisfaction problems* (CSPs). In this case, we can utilize a multi-agent based approach to automatically achieving a collaborative service task.

1.1. Organization of the paper

The rest of the paper is organized as follows. In Section 2, we provide a formulation of collaborative services in terms of distributed constraint satisfaction problems. We also give a survey of the related work. In Section 3, we introduce and illustrate the notion of multi-agent collaborative service (MACS), and describe the general design of MACS. In Section 4, we describe a novel distributed discrete Lagrange multiplier (DDLM) method for solving MACS problems. We show several experiments for validating the DDLM method. In Section 5, we elaborate on the working mechanism and characteristics of MACS. Finally in Section 6, we conclude the paper by summarizing the key contributions of this work.

2. Formulation of a collaborative service into a CSP

A collaborative service usually involves participants, subservices, and constraints among them. In some cases, the constraints can be formulated. In the following, we will take e-learning as an example to illustrate our above statement.

E-learning is a technology used to provide learning and training contents to learners via some electronic means, such as, computers, Intranet, and Internet. Essentially, it bridges the minds of instructors and learners with IT technologies. Contents are usually modularized into different modules, called *learning objects*, at different granularities, such as, *fragments* (e.g., picture, figure, table, text), *lessons*, *topics*, *units*, *courses*, and *curriculums* (IEEE, 2001; 2002; Loser et al., 2002). The reason for doing so is to increase the interoperability, scalability (Loser et al., 2002), etc. features of an e-learning service environment. Learning objects are located on different, usually geographically distributed, computers. Several smaller granular objects can constitute a bigger granular object. Generally speaking, there are two important types of relationships among learning objects (Loser et al., 2002): *content* and *ordering* relations. A content relation describes the semantic interdependency among modules, while an ordering relation describes the sequence for accessing modules. In the e-learning service, these relations actually act as *content constraints* that should be satisfied during the service time.

In addition to the above mentioned content and ordering relations, an e-learning service can also involve *system constraints* that are related to the hardware and software of the environment. Generally speaking, more than one learner can simultaneously access either the same or different contents. Learners are also geographically distributed. Although a learning object in an e-learning environment can be accessed by one or more learners each time, the number of learners is limited because of some specific factors, such as hardware bandwidth. Therefore, these factors play the roles of system constraints.

Example 2.1. Fig. 1 shows an illustrative e-learning service scenario, where four learners, A, B, C, and D are accessing the e-learning environment via clients connected to the environment. The learners need to learn three knowledge modules, i.e., m_1, m_2, and m_3, provided by three geographically distributed computers. The constraints in the environment are:

(1) *Content constraint:* To learn Modules m_2 and m_3, Module m_2 must be learned before Module m_3.
(2) *System constraint:* One module can serve at most two learners at the same time.

In order to successfully provide services for the above learners simultaneously, the distributed clients should collaborate with one another so as to determine a service sequence for each learner.

This collaborative service can be formulated as follows: For each learner $L \in \{A, B, C, D\}$, its client should coordinate with those of other learners to generate a combination $\{S_L^1, S_L^2, S_L^3\}$ of $\{1, 2, 3\}$ ($\forall i$, $S_L^i \in \{1, 2, 3\}$), which will be used as the sequence

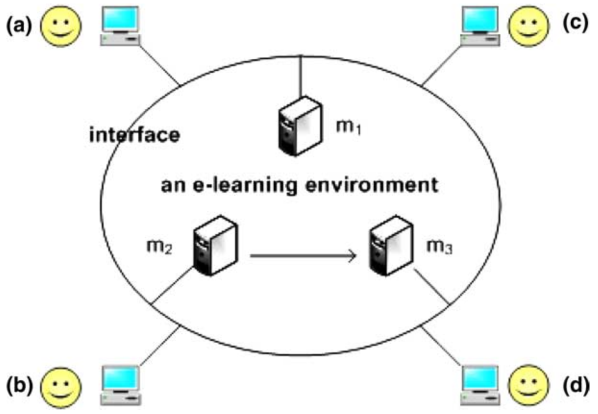

Fig. 1. A scenario where an e-learning environment provides services for four learners simultaneously. All learners need learn three modules, where Module m_1 is independent, while Modules m_2 and m_3 are dependent: Module m_2 has to be learned before Module m_3. At any time, each module can only be accessed by at most two learners.

in which Learner L accesses the knowledge modules. For example, if Learner A is assigned a combination $\{2, 3, 1\}$, it means Learner A should access the three modules in a sequence of 2, 3, and 1. Therefore, the above mentioned constraints are accordingly formulated as:

(1) $\forall L \in \{A, B, C, D\}$, $S_L^i \in \{1, 2, 3\}$, and for $i \neq j$, $S_L^i \neq S_L^j$.
(2) *Content constraint:* $\forall L \in \{A, B, C, D\}$, if $S_L^i = 2$ and $S_L^j = 3$, then it should guarantee $i < j$.
(3) *System constraint:* $\forall i \in \{1, 2, 3\}$ and $\forall j \in \{1, 2, 3\}$, $\sum_{L \in \{A,B,C,D\}} T(S_L^i = j) \leqslant 2$, where $T(\cdot)$ is a boolean function to test whether or not an input proposition is true.

By doing so, we have actually formulated the collaborative service into a distributed constraint satisfaction problem (CSP), where distributed clients are responsible for assigning values to their learners' variables.

In general, if all constraints in a collaborative service are written into certain formal expressions, the service has been formulated into a CSP. The obtained CSP is distributed in nature. It usually consists of:

(1) A finite set of variables, $X = \{x_1, x_2, \ldots, x_N\}$, where $\forall i \in [1, N]$, $x_i \in \mathbf{D}_i$.
(2) A constraint set, $C = \{C(R_1), C(R_2), \ldots, C(R_m)\}$, where R_i is an ordered subset of X, and each constraint $C(R_i)$ is a set of tuples indicating the mutually consistent values of the variables in R_i.

Moreover, in the obtained CSP, variables in X are divided into different groups. Some constraints consist of variables in one variable group while the others involve variables in different variable groups.

After a collaborative service is transformed into a distributed CSP, the process to arrange services for different users becomes a search process in a search space specified by the Cartesian product of all variables, namely, $\mathbf{D}_1 \times \mathbf{D}_2 \times \cdots \times \mathbf{D}_n$. In this Cartesian product, each element is a 'possible solution' of the corresponding CSP. It assigns values to all variables in X, but it does not guarantee all constraints are satisfied. If a 'possible solution' can satisfy all constraints, it is actually a solution of the CSP in a conventional sense. It is also a solution of the corresponding service. To achieve a service task, it is to search for such an element in the search space.

In a collaborative service, because each service participant (e.g., a client in Example 2.1) is responsible for coordinating with others so as to arrange services for its user, the constraints among the services to be arranged are transferred to participants. These constraints can be categorized into two types: *intra-participant-constraints* and *inter-participant-constraints*. An intra-participant-constraint is a constraint caused by the services for a specific user. An inter-participant-constraint is a constraint between services for different users. If a collaborative service is formulated into a CSP, each service participant will be responsible for a subgroup of variables in the CSP (e.g., in Example 2.1, the client of Learner A is responsible for variable group $\{S_A^1, S_A^2, S_A^3\}$). In this case, if a constraint in the CSP is based on variables in the same subgroup, it is called an *intra-constraint*. If a constraint is based on variables in different subgroups, it is called an *inter-constraint*. In the formulation of Example 2.1, the constraints contained in Items 1 and 2 are *intra-constraints*, while those in Item 3 are *inter-constraints*. As we have seen in Example 2.1, service participants are usually distributed, therefore the corresponding CSP is distributed in nature.

2.1. Related work on solving CSPs

The methods for solving CSPs can be classified into two types: *complete* and *incomplete*. Complete methods can detect whether or not a CSP has a solution and can find all solutions if there exist. But these methods are computationally expensive, and may not be applied to large-scale problems. On the other hand, incomplete methods can work much faster but cannot determine whether or not solving a CSP is feasible.

Local search is one of the popular *incomplete* methods. It starts with a randomly initialized assignment and performs a search by changing to the neighbor of the current assignment. The changes often depend on some search strategies.

Recently, some distributed CSP solving methods have been proposed, some of which make use of multi-agent systems (Liu, Jing, & Tang, 2002; Yokoo, Durfee, Ishida, & Kuwabara, 1998). In these methods, each agent governs multiple variables and deals with a subproblem. An agent will decide the values of the variables it governs, based on the condition of its local environment. Although the behaviors are local, the group of agents can be cooperatively organized to emerge a coherent behavior that leads to a solution state.

In solving a CSP with distributed agents, the system is required to reach a solution state as efficiently as possible. In this paper, we propose a distributed CSP solving method based on DLM. We view a CSP problem as an optimization problem, and try to solve this optimization problem with distributed agents based on DLM rules. In order to speed up problem solving, we use Lagrange multipliers to influence the local decision-making of individual agents. At the same time, we also use a run-time adaptive strategy to update the multipliers.

2.2. Related work on solving satisfiability problems

Hard satisfiability (SAT) problems constitute a special class of well-defined CSPs, which can be found in various theoretical as well as applied computer science studies. In fact, it has been proven that SATs and CSPs can be transformed to each other. A typical proposition SAT instance is composed of m clauses cl_i on N boolearn variables x_j. The clauses are formulated into a formula in conjunctive normal form (CNF), i.e., $\bigwedge_i cl_i$, where each clause cl_i has the form of $\bigvee_k l_k$, and each literal l_k is either a variable or its negation. The goal is to find an assignment to all variables, which makes the formula *true*, or to determine that no such assignment exists. SAT is NP-complete. A SAT instance is called a k-SAT, if each clause has k literals.

Local search is one of the commonly used methods for solving SAT (Hoos, 1999; Selman, Levesque, & Mitchell, 1992). Although it cannot determine whether or not a SAT instance is satisfiable, i.e., it is *incomplete*, it can find a solution very efficiently (Schuurmans & Southey, 2001). A general local search starts with a randomly initialized assignment, and then performs a search by flipping a variable of the current assignment based on a predefined evaluation function.

It is generally considered that GSAT was the first SAT solver based on the notion of local search (Selman et al., 1992). Since then, several local search methods for solving SAT have been introduced, including WSAT (McAllester, Selman, & Kautz, 1997; Selman, Kautz, & Cohen, 1994), Novelty (McAllester et al., 1997), and Novelty+ (Hoos, 1999), etc. These methods are efficient in finding a satisfied assignment for large-scale SAT problems. When the search reaches a local minimum, a strategy called *restart* will be applied to make the system escape from the local trap. In (Schuurmans & Southey, 2001), another local search method, called SDF, is proposed by identifying three measurable characteristics of local search behavior, i.e., *restart*, *mobility*, and *coverage*. The SDF method can be regarded as one of the most efficient local search methods in solving SAT problems.

The discrete Lagrange multiplier (DLM) method falls into global search methods (Shang & Wah, 1998; Wu, 2001). It can also be used to solve SAT problems. The difference between DLM and local search methods lies in the strategy for escaping from a local minimum. When the search gets stuck in a local minimum, the Lagrange multiplier can provide a strategy to lead the search out of the local minimum. The typical algorithms include *DLMA*$_2$ (Shang & Wah, 1998), *DLM_BASIC_SAT*, and *DLM_Distance_Penalty_SAT* (Wu, 2001). Although DLM is *incomplete*, it performs better than many existing methods.

3. Multi-agent collaborative service

The previous section has shown how to formulate a collaborative service into a CSP. In this section, we propose an approach, called *multi-agent based collaborative service* (MACS), to automatically achieve a given collaborative service that has been transformed into a CSP.

In the classical formulation of a distributed CSP, N variables are partitioned to n groups, and n agents, $agent_1, agent_2, \ldots, agent_n$ are used to govern these groups of variables, respectively. In other words, a multi-agent system for a CSP in a distributed manner consists of:

(1) A group of agents $\{agent_1, agent_2, \ldots, agent_n\}$.
(2) A set of variable groups $\{X_1, X_2, \ldots, X_n\}$ where $\sum_{i=1}^{n} |X_i| = N$. $agent_i$ governs the variables in X_i for $i \in \{1, 2, \ldots, n\}$. $X_i \cap X_j = \phi$ for any $i, j \in \{1, 2, \ldots, n\}$ and $i \neq j$.
(3) A set of domains $\{\mathbf{D_1}, \mathbf{D_2}, \ldots, \mathbf{D_n}\}$ that are corresponding to the variable groups.
(4) A set of constraints $\{cst_1, cst_2, \ldots, cst_m\}$ among variables.

A solution to a CSP is an assignment to all variables, which satisfies all constraints.

For $i \in \{1, 2, \ldots, n\}$, domain $\mathbf{D_i}$ is a vector space whose size is $|\mathbf{D_i}|$. $agent_i$ will move in this space – the *static part* of the agent environment. The position where an agent stays indicates a value vector of its governed variables. When performing a search in this environment, $agent_i$ will select a value vector $\mathbf{x}_i \in \mathbf{D}_i$ based on some predefined reactive behaviors (Liu et al., 2002). We refer to this as $agent_i$ moving to position $\mathbf{x}_i$. When the system reaches a solution state, the positions of different agents constitute a satisfied assignment for the distributed CSP.

Given a collaborative service, we assume it can be and has been transformed into a CSP. Thus, we can utilize the above multi-agent model to formulate it as follows:

(1) Each participant is represented by an agent.
(2) Variables that belong to a certain participant are assigned to the corresponding agent.
(3) An intra-constraint becomes an *intra-agent-constraint* belonging to the corresponding agent.
(4) An inter-constraint becomes an *inter-agent-constraint* belonging to the corresponding agents.

Given the above formulation to a multi-agent system, the question is how to automatically achieve the service by this multi-agent system. Since each agent represents a subgroup of variables, in order to solve the CSP, agents must assign values to their variables such that all constraints are satisfied. In other words, if all variables are assigned values and all constraints are satisfied, the original service task is finished.

For each agent, in order to assign 'appropriate' values to its variables, the agent should search in the Cartesian product of its variable domains. The Cartesian product can be regarded as an environment where the agent resides and moves around. Each element in the product denotes a position where the agent can stay. It also indicates a value combination of variables belonging to this agent. When an agent stays at a position, it indicates the agent assigns its variables with values that correspond to this position. When an agent moves from one position to another, it indicates at least one of its variables has been assigned a new value.

According to the above description, we can see that at any time, if we combine the positions of all agents, we can get a possible solution of the CSP.

Specifically, the multi-agent system works as follows: Initially, all agents are randomly placed on one of positions in their respective environments. After that, agents will coordinately move in their environments so as to explore their positions. The system is synchronized by a discrete timer. At each time step, each agent has a chance to decide whether or not it moves to a new position. The multi-agent system stops until a solution state emerges or a predefined time threshold is reached. A solution state means under this state a solution of the CSP is found.

4. The DDLM method for handling an MACS task

As shown in the above multi-agent distributed SAT solving, reactive agents interact with their local environment to form a global trend toward a solution. In order to automatically handle an MACS task in an adaptive fashion, in what follows we will describe a multi-agent method based on distributed discrete Lagrange multipliers, called DDLM. When solving a given MACS problem with the DDLM method, associated Lagrange multipliers are used to control the local behaviors of individual agents, which in turn improve the global performance of finding a solution. We will conduct experiments on benchmark 3-SAT instances to demonstrate the effectiveness of this method.

4.1. The Lagrange multiplier method

The Lagrange multiplier method is a traditional technique for solving continuous constraint optimization problems (Bertsekas, 1982; Chong & Żak, 1996). In (Shang & Wah, 1998), a global search method, called DLM (discrete Lagrange multiplier method), is presented for solving SAT problems. The main advantage of the DLM method lies in its strategy for escaping from a local minimum. When the search gets stuck in a local minimum, a local search method often uses a random flip strategy, whereas in the DLM method the power for getting out of a local minimum is provided by the multiplier. Although DLM falls into the category of *incomplete* methods, it demonstrates a competitive performance in solving certain benchmark problems (Wu, 2001).

4.2. The DDLM method for solving SAT problems

To apply the Lagrange multiplier method to distributed SAT solving, we need to define a class of evaluation functions such that the original problem can be transformed into an optimization problem subject to some constraints. A simple evaluation function is a mapping from constraints to a *violation* number (Liu et al., 2002). The *violation* number is the number of unsatisfied constraints, and can be classified into two types: local and global.

A local *violation* number of *agent*$_i$ is the number of unsatisfied constraints associated with that agent, whereas a global one is the sum of all unsatisfied constraints in the system. When solving a CSP using a distributed multi-agent system like ERA (Liu et al., 2002), an agent will select *least-move* or *better-move* that can reduce its local *violation* number based on its current state. However, this reduction does not mean the reduction of global violation, and may lead several agents to move back and forth periodically. A low-probability, reactive behavior called *random-move* will be used to help the system escape from such a local minimum.

Although the *random-move* like strategy is an efficient method for escaping from a local minimum, the randomness may undermine the current running results. This will in turn make such a system execute extra movements, and thus become more time-consuming.

It is interesting to note that most of the best-known *incomplete* methods, e.g., DLM (Shang & Wah, 1998; Wah & Shang, 1997; Wu, 2001), ESG (Schuurmans, Southey, & Holte, 2001), and SAPS (Hutter, Tompkins, & Hoos, 2002), are associated with the Lagrange multiplier method. This is because the multipliers play an effective role in bringing the search out of a local minimum. By extending this advantage, we hope to construct a distributed multi-agent method that can solve distributed SAT problems more efficiently. We refer to the Lagrange multiplier method for distributed discrete SAT solving as the *distributed discrete Lagrange multiplier* (DDLM) method.

4.2.1. The discrete Lagrange multiplier method

This section will briefly introduce the discrete Lagrange multiplier method for solving discrete equality constrained optimization problems (Shang & Wah, 1998; Wah & Shang, 1997).

Generally speaking, a discrete equality constrained optimization problem can be formulated as follows:

$$\begin{array}{ll} \min_{\mathbf{x}\in \mathbf{D}_1\times\mathbf{D}_2\times\cdots\times\mathbf{D}_n} & f(\mathbf{x}), \\ \text{subject to} & g(\mathbf{x}) = 0, \end{array} \tag{1}$$

where $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)^{\mathrm{T}}$, $g(\mathbf{x}) = (g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots, g_m(\mathbf{x}))^{\mathrm{T}}$, $\mathbf{x}_i \in \mathbf{D}_i$, and $\mathbf{D}_i$ is a domain space, $i \in \{1, 2, \ldots, n\}$. The corresponding Lagrange function is given as follows:

$$L(\mathbf{x},\lambda) = f(\mathbf{x}) + \sum_{j=1}^{m} \lambda_j \cdot g_j(\mathbf{x}) = f(\mathbf{x}) + \lambda^T \cdot g(\mathbf{x}), \tag{2}$$

where $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_m)^{\mathrm{T}}$.

Next, we present the definitions of a *saddle point* of the Lagrange function $L(\mathbf{x}, \lambda)$ and a *local minimum* of the problem in (1). Further, we describe a theorem about the relationship between them.

Definition 4.1 (Shang and Wah, 1998). A saddle point $(\mathbf{x}^*, \lambda^*)$ of function $L(\mathbf{x}, \lambda)$ is the point that satisfies the following:

$$L(\mathbf{x}^*, \lambda) \leqslant L(\mathbf{x}^*, \lambda^*) \leqslant L(\mathbf{x}, \lambda^*),$$

for all λ sufficiently close to λ^* and for all $\mathbf{x}$ that differ from x^* in only one dimension by a magnitude of 1.

Definition 4.2 (Shang and Wah, 1998). A local minimum of the problem (1) is a feasible point, $\mathbf{x}^*$, which satisfies $f(\mathbf{x}^*) \leqslant f(\mathbf{x})$ for any feasible point $\mathbf{x}$, and $\mathbf{x}^*$ and $\mathbf{x}$ differ in only one dimension by a magnitude of 1.

Theorem 4.1 (Shang and Wah, 1998). *A point* $\mathbf{x}^*$ *is a local minimum solution to the discrete optimization problem in* (1) *if there exists* λ^* *such that* $(\mathbf{x}^*, \lambda^*)$ *constitutes a saddle point of the corresponding Lagrange function* $L(\mathbf{x}, \lambda)$.

The discrete Lagrange multiplier method is actually to find a saddle point of the Lagrange function through interactions of the following two iterative equations (Shang & Wah, 1998):

$$\mathbf{x}^{t+1} = \mathbf{x}^t - D(\mathbf{x}^t, \lambda^t), \tag{3}$$

$$\lambda^{t+1} = \lambda^t + g(\mathbf{x}^t), \tag{4}$$

where t is the iteration step and $D(\mathbf{x}^t, \lambda^t)$ is a certain heuristic descent direction for updating λ. According to Theorem 4.1, if we find a saddle point of $L(\mathbf{x}, \lambda)$, it means that we have found a local minimum of the problem in (1).

It should be pointed out that the above method can only find an local minimum of the problem in (1).

4.2.2. The general DDLM method

The DDLM method is actually to present a distributed method, where a group of agents determine $D(\mathbf{x}^t, \lambda^t)$ in (3) in a distributed fashion. Before further description, let us define three important notions at first.

Definition 4.3. A neighborhood of $\mathbf{x}$ in $\mathbf{D}_i$ is a set $N_i(\mathbf{x})$,

$$N_i(\mathbf{x}) = \{\mathbf{y} |\ \text{for } i,\ |\mathbf{x}_i - \mathbf{y}_i| = 1 \\ \text{and for all } j \neq i,\ \mathbf{x}_j = \mathbf{y}_j\}.$$

Definition 4.4. The steepest difference gradient of $L(\mathbf{x}, \lambda)$ in $\mathbf{D}_i$ is defined as follows:

$$\Delta_{\mathbf{x}} L_i(\mathbf{x}, \lambda) = \mathbf{x} - \mathbf{y}, \tag{5}$$

where $\mathbf{y} = arg \min_{\mathbf{x}' \in N_i(\mathbf{x}) \cup \{\mathbf{x}\}} L(\mathbf{x}', \lambda)$.

Definition 4.5. The k-step quasi-steepest difference gradient of $L(\mathbf{x}, \lambda)$ is denoted as $\Delta_{\mathbf{x}} L^k(\mathbf{x}, \lambda)$, where $1 \leqslant k \leqslant n$. It is calculated as follows: select any k different numbers $i_1, i_2, \ldots, i_k$ from $\{1, 2, \ldots, n\}$, then calculate $\Delta_{\mathbf{x}} L_{i_j}(\mathbf{x}, \lambda)$ in a sequence for $j = 1, 2, \ldots, k$ by (5), the last value we obtain is $\Delta_{\mathbf{x}} L^k(\mathbf{x}, \lambda)$.

Gradient descent is a commonly used search technique that evaluates the gradient of given functions (Chong & Źak, 1996). Based on the above definitions, we can readily formulate a gradient descent iteration in (3) and (4) as follows: Assume $\mathbf{x^0}$ and λ^0 be any starting point,

$$\mathbf{x}^{t+1} = \mathbf{x}^t - \Delta_{\mathbf{x}} L^k(\mathbf{x}^t, \lambda^t), \tag{6}$$

$$\lambda^{t+1} = \lambda^t + g(\mathbf{x}^t), \tag{7}$$

where t denotes the tth iteration step, and $k \in \{1, 2, \ldots, n\}$ denotes the number of agents contributing to generate the descent direction, $\Delta_{\mathbf{x}} L^k(\mathbf{x}^t, \lambda^t)$. It should be noted that the change of $\mathbf{x}^t$ depends on the currently activated agents. In each iteration, if there are k activated agents, those k agents can contribute to such a change. Each agent $agent_i$ will determine its contribution using (5). The system will update the global parameters λ by using (7), after all k decisions are made.

Theorem 4.2. *A saddle point* $(\mathbf{x}^*, \lambda^*)$ *of* $L(\mathbf{x}, \lambda)$ *is reached iff the iteration* (6), (7) *terminates at* $(\mathbf{x}^*, \lambda^*)$.

Proof. $\Leftarrow$: If (6) terminates at $(\mathbf{x}^*, \lambda^*)$, then $\Delta_{\mathbf{x}} L_i(\mathbf{x}^*, \lambda^*) = 0$ for any i. It means that $L(\mathbf{x}^*, \lambda^*) \leqslant L(\mathbf{x}, \lambda^*)$ for any $\mathbf{x} \in N_i(\mathbf{x}) \cup \{\mathbf{x}\}$ where $i \in \{1, 2, \ldots, n\}$ by Definition 4.4. On the other hand, when (7) terminates at $(\mathbf{x}^*, \lambda^*)$, we have $g(\mathbf{x}^*) = 0$ and it means that $L(\mathbf{x}^*, \lambda) = L(\mathbf{x}, \lambda^*)$. Therefore $(\mathbf{x}^*, \lambda^*)$ is a saddle point.

$\Rightarrow$: According to Theorem 4.1, if $(\mathbf{x}^*, \lambda^*)$ is a saddle point, then $\mathbf{x}^*$ is a local minimum of the problem in (1). This indicates $g(\mathbf{x}^*) = 0$. We also have $L(\mathbf{x}^*, \lambda^*) \leqslant L(\mathbf{x}, \lambda^*)$ for all $\mathbf{x}$ that differ from $\mathbf{x}^*$ in one dimension by a magnitude of 1, it means that $\Delta_{\mathbf{x}} L_i(\mathbf{x}^*, \lambda^*) = 0$ for any i. Thus the iteration (6), (7) terminates at a saddle point $(\mathbf{x}^*, \lambda^*)$. $\square$

4.2.3. The DDLM SAT solver

Now, we turn to distributed SAT solving and consider only 3-SAT problems.

Assume the clauses related to $agent_i$ is a set $CL_i = CL_{i,1} \cup CL_{i,2} \cup CL_{i,3}$, where $CL_{i,k}$ denotes the constraints (i.e., clauses) whose variables are governed by $k-1$ other agents besides $agent_i$. Obviously, $CL_{i,1}$ contains the intra-agent constraints of $agent_i$, while $CL_{i,2}$ and $CL_{i,3}$ contain the inter-agent constraints. We formulate the local *violation* function of $agent_i$ as follows:

$$f_i(\mathbf{x}) = \sum_{k=1}^{3} \sum_{cl_j \in CL_{i,k}} c_j(\mathbf{x}), \text{ and } c_j(\mathbf{x}) = \prod_{i=1}^{N} a(x_i), \tag{8}$$

where x_i denotes a boolean variable, and

$$a(x_i) = \begin{cases} (1 - x_i)^2 & \text{if } x_i \text{ in } cl_j, \\ x_i^2 & \text{if } \bar{x}_i \text{ in } cl_j, \\ 1 & \text{otherwise.} \end{cases}$$

For a certain $c_j(\mathbf{x})$, if there exists an assignment $\mathbf{x}$ that makes $c_j(\mathbf{x}) = 0$, it means this assignment satisfies constraint cl_j. If this assignment also makes $f_i(\mathbf{x}) = 0$, it means all the constraints associated with $agent_i$ are satisfied. Hence, a distributed 3-SAT problem can be transformed into the following optimization problem:

$$\begin{array}{ll} \min f(\mathbf{x}), \text{or} \min \sum_{i=1}^{n} f_i(\mathbf{x}), \\ \text{subject to} \qquad C(\mathbf{x}) = 0, \end{array} \tag{9}$$

where $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)^{\mathrm{T}}$, $C(\mathbf{x}) = (c_1(\mathbf{x}), c_2(\mathbf{x}), \ldots, c_m(\mathbf{x}))^{\mathrm{T}}$, $\mathbf{x}_i \in \mathbf{D}_i$, $i \in \{1, 2, \ldots, n\}$, and the Lagrange function is as follows:

$$L(\mathbf{x}, \lambda) = \sum_{k=1}^{n} f_k(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \cdot c_i(\mathbf{x}). \tag{10}$$

We define the evaluation function of $agent_i$ as follows:

$$h_i(\mathbf{x}, \lambda) = \sum_{k=1}^{3} \sum_{cl_j \in CL_{i,k}} (1 + \lambda_j) \cdot c_j(\mathbf{x}). \tag{11}$$

Let $\Delta h_i(\mathbf{x}, \lambda) = \mathbf{x} - \mathbf{y}$, where $\mathbf{y} = arg\min_{\mathbf{x}' \in N_i(\mathbf{x}) \cup \{\mathbf{x}\}} h_i(\mathbf{x}', \lambda)$. It is easy to show that $\Delta h_i(\mathbf{x}, \lambda)$ is equal to the distributed steepest difference gradient of (10) at $\mathbf{D}_i$, i.e., $\Delta h_i(\mathbf{x}, \lambda) = \Delta L_i(\mathbf{x}, \lambda)$. Therefore, we can denote the distributed k-step quasi-steepest difference gradient of (10) as $\Delta h^k(\mathbf{x}, \lambda)$ where $1 \leqslant k \leqslant n$.

The DDLM iteration for SAT solving can thus be updated as follows:

$$\mathbf{x}^{t+1} = \mathbf{x}^t - \Delta h^k(\mathbf{x}^t, \lambda^t), \tag{12}$$

$$\lambda^{t+1} = \lambda^t + C(\mathbf{x}^t), \tag{13}$$

where $k \in \{1, 2, \ldots, n\}$, and t is the iteration step. Specifically, if we start this iteration system with $\lambda = 0$, we have:

Theorem 4.3. *A satisfied assignment* $\mathbf{x}^*$ *to problem* (9) *is found iff the iteration* (12), (13) *terminates.*

Proof. $\Leftarrow$: If the iteration terminates, then there exist $\mathbf{x}^*$ and λ^* s.t. $C(\mathbf{x}^*) = 0$. This implies that $\mathbf{x}^*$ is a feasible point to (9) and $f(\mathbf{x}^*) = 0$. Since $f(\mathbf{x}) \geqslant 0$ for any valid $\mathbf{x}$, $\mathbf{x}^*$ is a minimum point to (9) and hence $\mathbf{x}^*$ is a satisfied assignment.

$\Rightarrow$: If $\mathbf{x}^*$ is a satisfied assignment to (9), $f(\mathbf{x}^*) = 0$. For any $\mathbf{x}$ and $i \in \{1, 2, \ldots, n\}$, $c_i(\mathbf{x}) \geqslant 0$, since $\lambda^0 = 0$, by (13) we have $\lambda_i^k \geqslant 0$ for any k and i. Choosing a vector λ_i^* from $\{\lambda_i^k\}$, we have $L(\mathbf{x}^*, \lambda^*) = L(\mathbf{x}^*, \lambda)$ and $L(\mathbf{x}^*, \lambda^*) \leqslant L(\mathbf{x}, \lambda^*)$. Hence, $(\mathbf{x}^*, \lambda^*)$ is a saddle point. By Theorem 4.2, the iteration (12), (13) terminates. □

According to Theorem 4.3, if a SAT instance is feasible, the running of the iteration 12, 13 from $\lambda = 0$ can lead to a solution state.

4.2.4. The DDLM algorithm

The main procedure of the DDLM algorithm that demonstrates distributed SAT solving can be given as follows:

Algorithm 1. main-procedure

```
 1. While neither a solution nor time-out do
 2.    For all randomly selected agent $a_i$ do
 3.       compute new-position by (12);
 4.       If new-position = current-position then
 5.          stay;
 6.       Else
 7.          move to new-position;
 8.       EndIf
 9.    EndFor
10.    If current state is a solution state then goto 15;
11.    If time-out then goto 16;
12.    If condition for updating $\lambda$ is satisfied then
13.       call $\lambda$-update-procedure;
14. EndWhile
15. output a solution;
16. Return
```

4.2.5. Using the Lagrange multiplier to control local behaviors

When *agent*$_i$ is dispatched, it will evaluate each position in its local environment by computing (11). The constraints associated with *agent*$_i$ and the multipliers associated with these constraints constitute the *dynamic part* of *agent*$_i$'s environment. Based on this local information, *agent*$_i$ always tries to move to the position with a minimum evaluation value. But, it may not move to any positions if it stays in *zero-position*, i.e., $f_i(\mathbf{x}) = 0$, or gets stuck in a local minimum, i.e., $h_i(\mathbf{x})$ already has the minimum value in the current assignment but $f_i(\mathbf{x}) \neq 0$. We call the former *non-null-movement* (zero position) and the latter *null-movement* (local minimum). Correspondingly, when *agent*$_i$ finishes moving to a new position, we say that it finishes an *agent-movement*.

In multi-agent based problem solving, we cannot control the running sequence of distributed agents or the efficiency of a single agent. However, we can change the Lagrange multiplier to influence the local environment of each agent, and hence the evaluation values in different positions. Thus, acting as a class of global parameters, the Lagrange multiplier can provide a global level control to each agent's local reactive behavior.

4.2.6. Strategies for updating λ in distributed SAT solving

From the viewpoint of pure mathematics, λ denotes the change ratio of corresponding constraints. When used in problem solving (Shang & Wah, 1998), it may be viewed as the weights of constraints. However, in a distributed multi-agent system, we will consider it as a virtual global constraint that decides the moving directions of agents.

Now, the questions that remain are: when and how to update λ in distributed SAT solving? The formula given in (13) will allow λ to increase without an upper bound if some constraints remain unsatisfied. This may make it difficult to escape from a local minimum (Wu, 2001). We need some strategies to avoid λ becoming too big. The DLM implementation in Wu (2001) adopts a strategy to reduce all λ periodically, while the ESG algorithm in Schuurmans et al. (2001) adopts multiplicative updates and then smoothes λ based on a population average. The strategy we adopt is to reduce all non-zero λ periodically.

Three factors should be considered (there need some tradeoffs among them):

(1) Only actual agent-movements may speed up SAT solving; we need to improve the ratio of agent-movements to λ-updates.
(2) The core of the algorithm lies in the gradient descent step that leads agents to search in a narrow area when the direction is erroneous.
(3) As the smoothing technique is verified by using many instances (Schuurmans & Southey, 2001; Schuurmans et al., 2001), we should search slowly in a feasible area.

4.2.7. An adaptive strategy for updating λ

In this work, we adopt a run-time adaptive strategy to update λ. We first define an equation for reducing λ. It is based on the idea of reducing all non-zero λ periodically.

$$\lambda^{t+1} = \lambda^t - \omega(\lambda^t), \tag{14}$$

where $\omega(\lambda) = (\omega(\lambda_1), \omega(\lambda_2), \ldots, \omega(\lambda_m))^{\mathrm{T}}$ and

$$\omega(\lambda_j) = \begin{cases} 1 & \text{if } \lambda_j \neq 0, \\ 0 & \text{if } \lambda_j = 0. \end{cases}$$

We then set *counter*$_1$ to record the sum of the numbers of *agent-movements* and *null-movements* (local minimum), *counter*$_2$ to record the number of λ updates. Two threshold functions (*threshold*$_1(\mathbf{x}, \lambda)$ and *threshold*$_2(\mathbf{x}, \lambda)$) and two variables (internal variable *inc* and external parameter *dec*) are used to decide when to update λ. We assume that *counter*$_1$, *counter*$_2$, and *inc* are initialized with 0, 0, and 1 in the **main-procedure**, respectively, and *dec* is the parameter of this procedure.

What follows is **λ-update-procedure** that we adopt in this work.

Algorithm 2. λ-update-procedure

```
1.  If $a_i$ encounters an agent-movement then
2.      $inc++$;
3.  EndIf
4.  If $a_i$ encounters a non-null-movement (zero position) then
5.      $counter_1$++;
6.      If $counter_1 \% (threshold_1(\mathbf{x}, \lambda)/inc) = 0$ then
7.          $counter_2$++;
8.          If $counter_2 \% (threshold_2(\mathbf{x}, \lambda)/dec) = 0$ then
9.              update $\lambda$ by (14);
10.         Else
11.             update $\lambda$ by (13);
12.         EndIf
13.         If $(dec < threshold_1(\mathbf{x}, \lambda)/2)$ then
14.             dec++;
15.         EndIf
16.     EndIf
17. EndIf
18. Return
```

4.3. Validations

In this subsection, we will through experiments validate our proposed DDLM SAT solver, as well as compare it with other solvers on different benchmark problems.

4.3.1. Experiments

We have conducted several experiments to examine the DDLM algorithm using benchmark 3-SAT problems (Satlib, 2000). We set the two threshold functions to *agent-number* (i.e., the number of agents). In running each instance, we initialize all x and all λ to 0, and equally partition the variables into some groups in a sequence. The number of groups is equal to *agent-number*. For example, if we distribute 100 variables to 28 agents, we will distribute the first 64 variables to 16 agents and the latter 36 to 12 agents. Further, we let *agent*$_1$ manage the first four variables, *agent*$_2$ manage the second four, ..., and *agent*$_{28}$ manage the last three.

4.3.2. Comparisons with the DLM-based algorithm in solving small 3-SAT problems

In Table 1, we compare the performance of four *aim*-100-2_0 problems with another DLM-based SAT solver, DLM_BASIC_SAT. These four problems are artificially generated random 3-SAT (Satlib, 2000). Each of them has 100 variables and 200 clauses. The results of DLM_BASIC_SAT are from (Wu, 2001). In our experiments, we set *agent-number* to 70. The results show that in solving these four 3-SAT problems, our algorithm performs better than DLM_BASIC_SAT.

Table 1
Comparisons with another DLM method in solving small 3-SAT problems

Problem ID	Success ratio	DLM_BASIC_SAT	DDLM
aim-100-2_0-yes1-1	10/10	9,460	2,342
aim-100-2_0-yes1-2	10/10	9,473	1,681
aim-100-2_0-yes1-3	10/10	5,077	1,574
aim-100-2_0-yes1-4	10/10	7,797	1,808

The *aim* problems are artificially generated random 3-SAT. Each problem has 100 variables and 200 clauses. The results of DLM_BASIC_SAT are the average number of flips, and those of DDLM are the average number of movements. In our experiments, *agent-number* is 70.

4.3.3. Comparisons with the DLM-based algorithm in solving large 3-SAT problems

Table 2 shows comparisons with DLM_BASIC_SAT in solving 3-SAT problems with 200 variables. These problems are also from (Satlib, 2000). The results of DLM_BASIC_SAT are from (Wu, 2001). In our experiments, we set *agent-number* to 150 in aim-200-1_6-yes1-1 and that in the rest to 56. The result shows that our algorithm performs better in solving *aim*-200 problems.

4.3.4. Comparisons with local search methods

In order to compare with other best-known local search methods, we have run a set of experiments using satisfiable benchmark 3-SAT test-sets: uf150, uf200, and uf250 (Satlib, 2000). Each test-set has 100 satisfiable uniform random 3-SAT instances. There exist 150 variables and 645 clauses in each uf150 instance, 200 variables and 860 clauses in each uf200 instance, and 250 variables and 1075 clauses in each uf250 instance. We run each instance 100 times and average the agent-movements after finishing each test-set. If a run requires more than 500,000 movements to reach a solution state, we label it as fail. The experimental results are summarized in Table 3.

Table 3 shows that our DDLM method performs better than the existing best-known local search methods, when comparing the number of flips (movements), and that there is a slight difference between ESG_h and DDLM.

4.4. Discussions

This subsection will discuss the influence of two important parameters, i.e., *agent-number* and *the Lagrange multiplier*, on the performance of the DDLM SAT solver.

4.4.1. Effects of agent-number

In solving a given SAT problem, an agent will determine its local reactive behavior using (6). After running some steps, various local reactive behaviors will be aggregated and force the system to tune the global parameters, i.e., the Lagrange multiplier. In solving SAT problems, we use (13) and (14) to update the Lagrange multiplier. The updating depends on two threshold functions. In our experiments, we set the two functions as a constant that is equal to *agent-number*. Table 4 shows that various *agent-numbers* can affect the performance and the optimal agent-number setting can vary in different problems.

Table 2
Comparisons with another DLM method in large 3-SAT problems

Problem ID	Success ratio	DLM_BASIC_SAT	DDLM
aim-200-1_6-yes1-1	10/10	80,877	26,978
aim-200-2_0-yes1-1	10/10	174,356	50,524
aim-200-3_4-yes1-1	10/10	99,393	12,163
aim-200-6_0-yes1-1	10/10	894	435

Each problem has 200 variables. The results of DLM_BASIC_SAT are the average number of flips, and those of DDLM are the average number of movements. In our experiments, we set *agent-number* to 56, except that we set it to 150 in aim-200-1_6-yes1-1.

Table 3
Comparisons with other best-known local search methods using uf test-sets: uf150, uf200, and uf250

Problem ID	Algorithm ID	Average Flips	Fail (%)
uf150	Novelty+(0.5, 0.01)	8,331	0.03
	DLM(pars4)	3,263	0
	SDF(.00065)	3,312	0
	ESG_h(1.15, 0.01, 0.001)	2,649	0
	DDLM (2, 50)	2,523	0
uf200	Novelty+(0.5, 0.01)	28,529	2.3
	DLM(pars4)	12,215	0.08
	SDF(.0003)	14,962	0.44
	ESG_h(1.23, 0.01, 0.003)	10,360	0.18
	DDLM (2, 66)	10,548	0.11
uf250	Novelty+(0.5, 0.01)	31,560	2.2
	DLM(pars4)	22,635	0.3
	SDF(.0002)	18,905	0.26
	ESG_h(1.15, 0.01, 0.003)	13,529	0.14
	DDLM (3, 83)	12,438	0.14

Our results are the average movements from running each instance 100 times and the algorithm ID is labeled as DDLM (*dec*, *agent-number*). Other results are obtained from (Schuurmans & Southey, 2001; Schuurmans et al., 2001).

Table 4
Effects of different agent-numbers

Agent-number	Success ratio	aim-100-2_0-yes1-1	aim-100-2_0yes1-2
20	10/10	103,201	8,429
30	10/10	2,607	5,189
40	10/10	2,694	3,272
50	10/10	3,929	2,482
60	10/10	2,935	1,850
70	10/10	2,342	2,679
80	10/10	3,713	2,634
90	10/10	3,720	1,811
100	10/10	2,455	3,373

4.4.2. Global parameter: The Lagrange multiplier

In practice, it is important to pay attention to the global parameter, i.e., the Lagrange multiplier in SAT solving. Both the agent-number and the current state of the system can affect the multipliers. That is why we use two functions, whose variables are $\mathbf{x}$ and λ, to denote the thresholds in updating the Lagrange multiplier.

As discussed in Shang and Wah (1998), the multipliers can be viewed as clause weights in a SAT problem, but we rather view them as the virtual global constraints during problem solving. In case that an agent movement might destroy the global behavior for achieving a solution state, the multipliers can limit this local behavior before the agent makes such a decision. In fact, we combine the associated multipliers to an agent's local decision. The use of the Lagrange multiplier for local decisions will give us a more global-level feedback in problem solving (Nareyek, 2001).

Fig. 2 shows the relationship between agent-movements and multiplier-updates. When a SAT problem can be solved with a small number of agent-movements, the ratio will be small, which means the number of multiplier-updates will be much smaller. As the number of agent-movements in finding a solution increases, the ratio will increase and then become stabilized around 0.5–0.55. This means that we need more multiplier-updates to help agents escape from a local minimum.

Fig. 3 shows the log–log plot of complement-to-one of the cumulative distribution for the number of multiplier-updates in the case of 10,000 runs with the $uf100$ test-set. The linear nature of the tail reveals the heavy-tailed behavior of *multiplier-updates*.

5. The working mechanism of MACS

The preceding section has provided and validated the DDLM method for achieve a MACS task that has been transformed to a CSP. This section will discuss several important issues about MACS. To better understand the working mechanism of MACS, we should clarify the following three questions: In a multi-agent system for solving a CSP in a distributed fashion,

(1) How does an agent evaluate its positions?
(2) How does an agent move in its environment?
(3) How does an agent coordinate with the others?

5.1. Position evaluation

To explore its positions, an agent need a certain evaluation criteria to evaluate them. With an evaluation criterion, an agent can judge which position is more 'appropriate' than the others.

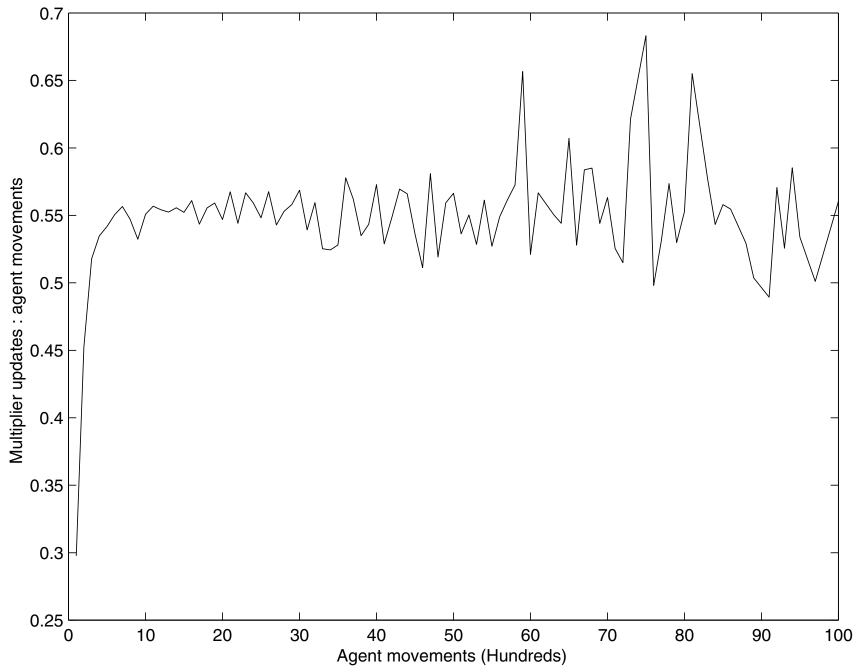

Fig. 2. Ratios between the number of *multiplier-updates* and *agent-movements*. After running the *uf*-100 instances for 10,000 times, we partition the running results into groups (each with a hundred movements). We calculate the ratios between the number of multiplier-updates and that of agent-movements in each group.

Generally speaking, an evaluation criterion is given in the form of an evaluation function. The ultimate goal of the multi-agent system is to satisfy all constraints. Therefore, as we have seen, MACS uses (11) as the evaluation function for each agent. Specifically, (11) reflects how many constraints related to $agent_i$ are satisfied. In this case, to evaluate its positions, an agent is actually to check how many constraints are satisfied if it stays at them. For the evaluation function in (11), the lower the evaluation value, the more the constraints are satisfied.

5.2. Information exchanging mechanism

For an intra-agent-constraint of an agent, it is easy to test whether or not it is satisfied, because the agent knows the values of all variables in the constraint. For an inter-agent-constraint, it is hard to test. To evaluate an inter-agent-constraint, an agent must get the value information of other variables in the constraint, which do not belong to itself.

There are different methods to get variable values of other agents. For example, we can allow agent–agent communications to exchange value information between related agents. But, this method is time-consuming, especially when the constraint involves many variables that belong to different agents. In this case, an agent must communicate with many other agents to get values of their variables. MACS uses a different method without explicit agent–agent communications. A shared blackboard is provided as a medium for all agents to share the value information of their variables. In so doing, the agent applies the following two actions:

- **Write**: An agent writes the values of all its variables onto the shared blackboard;
- **Read**: An agent reads the values of the variables it needs from the shared blackboard.

With this mechanism, an agent can get the values of all other variables. Thus, it can test all constraints.

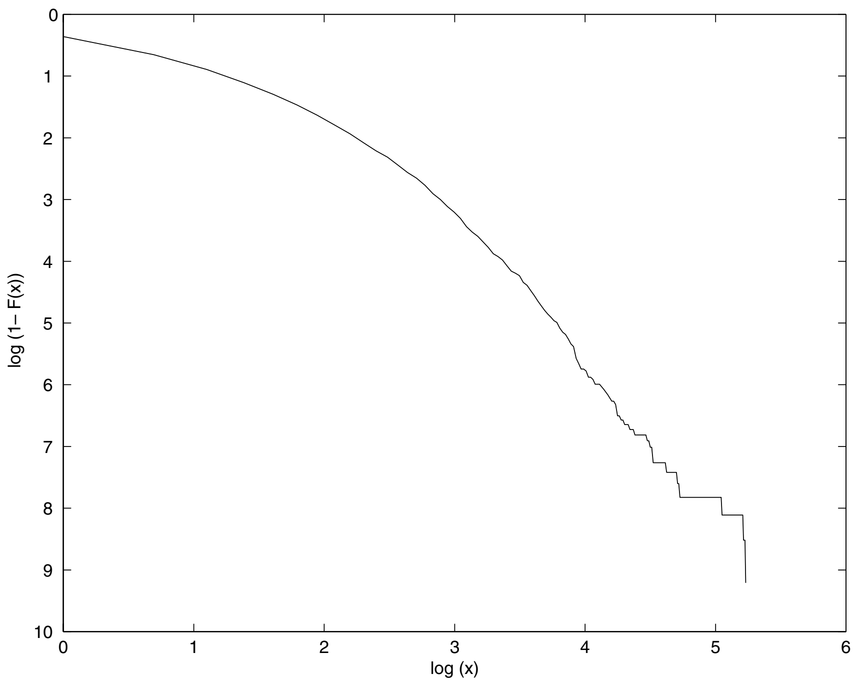

Fig. 3. A log–log plot of the heavy-tailed behavior for updating the Lagrange multiplier in solving the *uf*-100 problems. We choose x as the number of multiplier-updates, and $F(x) = P(X \leqslant x)$. The figure is plotted using the results of 10,000 runs in the *uf*-100 problems.

5.3. Coordination protocol

As DDLM is a multi-agent based method, a big problem is how agents coordinate in DDLM so as to solve the problem in hand. This subsection will show the coordination protocol between agents in MACS. MACS uses a blackboard as a medium to share information of agents rather than the way of direct agent–agent communications. Therefore, the coordination in MACS is implicit. The following is the protocol used by agents at each time step to behave and coordinate with the others.

(1) **Read information:** The agent reads the value information of other variables from the blackboard. On the blackboard, all agents have written the value information of their variables corresponding to their positions in the previous time step. At different positions, an agent will write different value information onto the blackboard. So, from the value information written by an agent, other agents will know whether or not this agent has moved to a new position.
(2) **Evaluation positions:** The agent evaluates all its positions after it has gotten the value information of other necessary variables. Obviously, different value information of other variables leads to different evaluation values even for the same position of a certain agent. In this sense, if an agent moves to a new position, its movement will affect the others in the next time step. The coordination in MACS just materializes in this way. After the evaluation, each position will have a corresponding evaluation value.
(3) **Find a new position:** In this step, the agent will find one of its positions with the lowest evaluation value. If the position with the lowest evaluation value is exactly its current positions, the agent will stay. If there are several positions with the lowest evaluation value, it will randomly select one. Through this step, an agent can move to a position with the smallest number of violated constraints. Because all agents have the same strategies, the system will gradually evolve to the state with less and less constraints violated.

(4) **Move to the selected position:** As a result of the previous behaviors, at this step, the agent moves to the selected new position.
(5) **Write information:** After an agent moves to the selected new position, the values of its variables will be changed. In this case, the agent will rewrite the changed value information on the blackboard. At the next time step, other agents will read the new value information of this agent. The new value information will affect the evaluation of other agents. So, it will indirectly affect the behaviors of others.
(6) **System evaluation:** This step is executed by the multi-agent system. Through this step, the system will check whether or not a solution state emerges (i.e., whether or not all constraints are satisfied) or the time is out. If the former holds, the system will stop and output the solution found. The the latter holds, it means the system fails to find a solution in the given time period. The system will also check whether or not the λ updating condition is satisfied. If so, it will update the value of λ according to **λ-update-procedure**.

6. Conclusion

This paper is concerned with collaborative services as they are formulated into CSPs. In this case, a multi-agent system can be employed to automatically achieve collaborative services. For this purpose, we proposed an approach, called multi-agent collaborative service (MACS) where each agent corresponds to a participant and the intra- or inter-participant-constraints become the corresponding intra- or inter-agent-constraints. Besides providing the notion and formulation of MACS, we have also discussed several important practical issues on how to implement the multi-agent system in a collaborative service environment. Specifically, we presented the information exchanging mechanism and the protocol used by agents to behave and coordinate with others.

We have described and demonstrated a DDLM method for handling distributed MACS tasks. This method belongs to incomplete methods, and can be applied in a distributed multi-agent environment. We have also provided a set of definitions and theorems as the theoretical foundation for this method.

In the DDLM distributed problem solving, an agent may manage multiple variables and determine its own local reactive behavior in an environment. However, this behavior will be limited by the Lagrange multiplier. In our method, we have used Lagrange multipliers to control the local reactive behaviors of agents to efficiently establish their global convergence toward a solution state. The multipliers can be viewed as the virtual global constraints that will guide the agents to form a coherent behavior in a global sense. In order to speed up their SAT solving process, we have adopted an adaptive strategy to update the multipliers. At the same time, the local search environment of each agent is a multi-variable space, this offers us a new formulation for local search. We have validated this method using some benchmark 3-SAT problems and have obtained a better performance as compared to the DLM-based methods and some well-known local search methods.

Acknowledgements

The authors acknowledge the support for this research project as provided in part by a HKBU FRG grant and in part by a HK RGC grant.

References

Bertsekas, D. P. (1982). *Constrained optimization and Lagrange multiplier methods*. Academic Press.

Chong, E. K. P., & Źak, S. H. (1996). *An introduction to optimization*. John Wiley & Sons.

Hoos, H. H. (1999). On the run-time behavior of stochastic local search algorithms for SAT. In *Proceedings of the 16th national conference on artificial intelligence (AAAI'99)* (pp. 661–666).

Hutter, F., Tompkins, D., & Hoos, H. H. (2002). Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the 18th international conference on principles and practice of constraint programming, LNCS 2470* (pp. 233–248).

Liu, J., Jing, H., & Tang, Y. Y. (2002). Multi-agent oriented constraint satisfaction. *Artificial Intelligence, 136*, 101–144.

Loser, A., Grune, C., & Hoffmann, M. (2002). A didactic model, definition of learning objects and selection of metadata for an online curriculum. Available: http://www.ibi.tu-berlin.de/diskurs/onlineduca/onleduc02/.

McAllester, D., Selman, B., & Kautz, H. (1997). Evidence for invariants in local search. In *Proceedings of the 14th national conference on artificial intelligence (AAAI'97)* (pp. 321–326).

Mori, T., & Cutkosky, M. R. (1998). Agent-based collaborative design of parts in assembly. In *Proceedings of the 1998 ASME design engineering technical conferences*.

Nareyek, A. (2001). Using global constraints for local search, constraint programming and large scale. In *DIMACS* (pp. 9–28).

Satlib (2000). Available: http://www.satlib.org.

Schuurmans, D., & Southey, F. (2001). Local search characteristics of incomplete SAT procedures. *Artificial Intelligence, 132*(2), 121–150.

Schuurmans, D., Southey, F., & Holte, R. C. (2001). The exponentiated subgradient algorithm for heuristic boolean programming. In *Proceedings of the 19th international joint conference on artificial intelligence (IJCAI'01)* (pp. 334–341).

Selman, B., Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proceedings of the 9th national conference on artificial intelligence (AAAI'92)* (pp. 440–446).

Selman, B., Kautz, H., & Cohen, B. (1994). Noise strategies for improving local search. In *Proceedings of the 11th national conference on artificial intelligence (AAAI'94)* (pp. 337–343).

Shang, Y., & Wah, B. W. (1998). A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization, 12*(1), 61–99.

IEEE (2001). Draft standard for information technology, learning technology glossary. Technical Report P1484.3/D3, IEEE.

IEEE (2002). Standard for learning object metadata. Technical Report P1484.12.1, IEEE.

Wah, B. W. & Shang, Y. (1997). Discrete lagrangian-based search for solving MAX-SATproblems. In *Proceedings of the 15th international joint conference on artificial intelligence (IJCAI'97)* (pp. 378–383).

Wu, Z. (2001). *The theory and applications of discrete constrained optimization using Lagrange multiplier*. Ph.D. Thesis, Department of Computer Science, University of Illinois.

Yokoo, M., Durfee, E. H., Ishida, T., & Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering, 10*(5), 673–685.

Yoshida, T., Teduka, T. & Nishida, S. (1999). Facilitate agreement in cooperative design by detecting irrational claims in negotiation. In *Proceedings of the first asia-pacific conference on intelligent agent technology* (pp. 64–73).