Available online at www.sciencedirect.com

SCIENCE DIRECT

Artificial Intelligence 161 (2005) 89–115

Artificial Intelligence

www.elsevier.com/locate/artint

The distributed breakout algorithms

Katsutoshi Hirayama [a,*], Makoto Yokoo [b]

[a] *Faculty of Maritime Sciences, Kobe University, 5-1-1 Fukaeminami-machi, Higashinada-ku, Kobe 658-0022, Japan*

[b] *Faculty of Information Science and Electrical Engineering, Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka 812-8581, Japan*

Received 3 April 2003; accepted 31 August 2004

Abstract

We present a new series of distributed constraint satisfaction algorithms, the *distributed breakout algorithms*, which is inspired by local search algorithms for solving the constraint satisfaction problem (CSP). The basic idea of these algorithms is for agents to repeatedly improve their tentative and flawed sets of assignments for variables simultaneously while communicating such tentative sets with each other until finding a solution to an instance of the distributed constraint satisfaction problem (DisCSP). We introduce four implementations of the distributed breakout algorithms: SINGLE-DB, MULTI-DB, MULTI-DB$^+$, and MULTI-DB^{++}. SINGLE-DB is a distributed breakout algorithm for solving the DisCSP, where each agent has a single local variable and its related constraints. MULTI-DB, on the other hand, is another distributed breakout algorithm for solving the distributed SAT (DisSAT) problem, where each agent has multiple local variables and their related clauses. MULTI-DB$^+$ and MULTI-DB^{++} are stochastic variations of MULTI-DB. In MULTI-DB$^+$, we introduce a technique called *random break* into MULTI-DB; in MULTI-DB^{++}, we introduce a technique called *random walk* into MULTI-DB$^+$. We conducted experiments to compare these algorithms with the asynchronous type of distributed constraint satisfaction algorithm. Through these experiments, we found that SINGLE-DB, MULTI-DB, and MULTI-DB$^+$ scale up better than the asynchronous type of distributed constraint satisfaction algorithms, but they sometimes show very poor performance. On the other hand, we also found that MULTI-DB^{++}, which uses random walk, provides a clear performance improvement.
© 2004 Elsevier B.V. All rights reserved.

* Corresponding author.
E-mail addresses: hirayama@maritime.kobe-u.ac.jp (K. Hirayama), yokoo@is.kyushu-u.ac.jp (M. Yokoo).

0004-3702/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2004.08.004

Keywords: Distributed constraint satisfaction; Local search; SAT; Coordination

1. Introduction

The *distributed constraint satisfaction problem* (DisCSP) [23,25] is a constraint satisfaction problem (CSP) in which variables and constraints are distributed among multiple agents. Even though the definition of CSP is very simple, a surprisingly wide variety of problems in computer science can be formalized as CSPs. Similarly, various application problems in Multi-Agent Systems (MAS) that are concerned with finding a consistent combination of agent actions (e.g., the distributed resource allocation problem [4], the distributed scheduling problem [21], the distributed interpretation task [14], and the multi-agent truth maintenance task [11]) can be formalized as DisCSPs. Therefore, we have considered an efficient distributed algorithm for solving the DisCSP as an important infrastructure in MAS.

The authors have previously presented two distributed algorithms for solving the DisCSP, called the *asynchronous backtracking algorithm* (ABT) and the *asynchronous weak-commitment search algorithm* (AWC) [25]. These algorithms are similar in their basic operations: in both algorithms, a priority order is defined among agents, and agents exchange their current assignments for variables and change their assignments concurrently and asynchronously so that the assignments are consistent with those of higher-priority agents. A major difference between ABT and AWC is the operation at dead-ends, that is, the operation invoked when an agent cannot find a consistent assignment for its variable. In ABT, an agent backtracks at dead-ends by sending a *nogood*, a combination of value assignments that cannot be a part of a solution, to a higher-priority agent to request that it change an assignment. On the other hand, in AWC, an agent uses a technique called *weak-commitment*, where an agent gives up the attempt to satisfy its constraints and delegates them to other agents by raising its own priority. While doing this, an agent can send nogoods to other agents so that they will not take the value assignments specified in the nogoods. Experimental evaluation shows that AWC greatly outperforms ABT in finding solutions to some hard DisCSP instances [25].

In ABT and AWC, *nogood learning* plays an important role in the search performance. A nogood learning technique specifies how an agent generates/stores nogoods. By making each agent generate/store all nogoods, both ABT and AWC are guaranteed to be complete, and, moreover, the communication cost of AWC can be dramatically reduced [8]. However, especially when solving critically hard DisCSP instances, both algorithms are likely to produce a huge number of nogoods, and some agents may hence consume a lot of memory to store these nogoods as well as a lot of computation to check them. This problem can be serious, especially when an agent has to solve a DisCSP instance when it's permitted to use only a limited amount of memory.

In this paper, we introduce a new series of distributed constraint satisfaction algorithms called the *distributed breakout algorithms*. Since these algorithms do not have to rely on nogood learning, they can operate in a situation where each agent has a limited amount

of memory. These algorithms are inspired by local search algorithms for the (centralized) CSP, such as the heuristic repair method [15], the breakout algorithm [16], and GSAT [17]. The basic idea of the distributed breakout algorithms is that the agents repeatedly improve their tentative and flawed sets of assignments for variables simultaneously while communicating such tentative sets with each other until finding a solution to a DisCSP instance. To realize this, each agent first sets an initial set of assignments for its variables and exchanges the set with its *neighbors*, then alternates as follows until finding a solution to a DisCSP instance.

(1) Each agent searches for the candidate for the next set of assignments that would reduce a *cost* and exchanges information on the candidate with its neighbors to resolve potential conflicts.
(2) Each agent sets the candidate as a new set of assignments if the candidate still remains valid after the conflict resolution process. Then, it exchanges a set of assignments with its neighbors.

In this procedure, the agents can sometimes be trapped in a *local minimum*, where no agent can reduce a cost while some agent has a flawed set of assignments, on the way to a solution to a DisCSP instance. To escape from such local minima, the procedure adopts a simple escaping technique called *breakout at quasi-local minima*.

We introduce four implementations of the distributed breakout algorithms: SINGLE-DB [24], MULTI-DB [9], MULTI-DB$^+$, and MULTI-DB^{++}. SINGLE-DB is a distributed breakout algorithm that is basically designed for the DisCSP where each agent has a single local variable and its related constraints. Also, it uses a simple and deterministic conflict resolution technique when selecting valid candidates for the next set of assignments for variables. MULTI-DB is a distributed breakout algorithm that solves the distributed SAT (DisSAT) problem where each agent has multiple local variables and their related clauses. Moreover, it uses a sophisticated and deterministic conflict resolution technique that allows agents to perform more simultaneous assignment changes leading to a rapid cost decrease. Both MULTI-DB$^+$ and MULTI-DB^{++} are extensions of MULTI-DB, and they both basically follow the same procedure as MULTI-DB does, but they are extended to behave in a stochastic manner. More specifically, we devise two stochastic techniques called *random break* and *random walk*; we introduce random break into MULTI-DB and call the resultant algorithm MULTI-DB$^+$ and random walk into MULTI-DB$^+$, calling the resultant algorithm MULTI-DB^{++}.

The remaining parts of this paper are organized as follows. First, in Section 2, we give the background of this work, including the definition of DisCSP and the outline of a local search algorithm for the CSP. Next, after describing the generic distributed breakout algorithm in Section 3, we present a series of implementations: SINGLE-DB in Section 4 and MULTI-DB and its stochastic variations in Section 5. We then experimentally evaluate these implementations in Section 6 and finally conclude this work in Section 7.

2. Background

In this section, as background of this work, we first introduce the definition of DisCSP with two illustrative examples: the distributed graph coloring problem and the DisSAT problem, then give the outline of a local search algorithm for the CSP.

2.1. DisCSP

The CSP consists of n variables $x_1, x_2, \dots, x_n$, whose values are taken from finite and discrete domains $D_1, D_2, \dots, D_n$, respectively, and a set of constraints on their values. A constraint can be described as a *nogood*, i.e., a set of values for some variables that are prohibited for the variables. A nogood is violated when its corresponding variables actually take the values appearing in the nogood. A solution to the CSP is an assignment of values for all of the variables whereby no nogood is violated. The problem of finding a solution to the CSP is known to be NP-complete.

The DisCSP is a CSP in which variables and constraints are distributed among multiple agents. It consists of the following:

- a set of *agents*, $1, 2, \dots, k$,
- a set of CSPs, $P_1, P_2, \dots, P_k$, such that P_i belongs to agent i and consists of
  - a set of *local variables* whose values are controlled by agent i,
  - a set of *intra-agent constraints*, each of which is defined over agent i's local variables,
  - a set of *inter-agent constraints*, each of which is defined over agent i's local variables and other agents' local variables.

A solution to the DisCSP is a set of solutions to all of the agents' CSPs, i.e., a state where all of the agents find sets of assignments of values for their local variables whereby no intra/inter-agent constraint is violated. Obviously, the problem of finding a solution to the DisCSP is NP-complete.

Fig. 1 illustrates an example of a DisCSP, the distributed graph coloring problem. The graph coloring problem is a problem that requires finding a color (among available colors) for each node of a given graph such that no adjacent pair of nodes has the same color. By considering a node as a variable and a link as a constraint, the graph coloring problem can be mapped into the CSP. The distributed graph coloring problem involves nodes and links that are distributed among agents such that each agent has some nodes and all of the links that are connected to the nodes. In Fig. 1, there are three agents, 1, 2, 3, each of which has nodes in the corresponding ellipse and links that are connected to the nodes. For example, agent 1 has the nodes n_1 and n_2 and the links l_{12}, l_{13}, and l_{26}. In other words, agent 1 has a CSP instance consisting of local variables derived from n_1 and n_2, intra-agent constraints derived from l_{12}, and inter-agent constraints derived from l_{13} and l_{26}.

Next, we introduce the DisSAT problem as another example of the DisCSP. The propositional satisfiability (SAT) problem is the problem of finding a *model* for a propositional formula, i.e., a truth assignment for variables in a formula that makes the formula true. A formula is typically described in *Conjunctive Normal Form* (CNF), and we hence call it

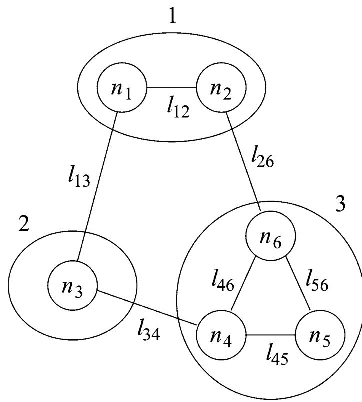

Fig. 1. Distributed graph coloring problem.

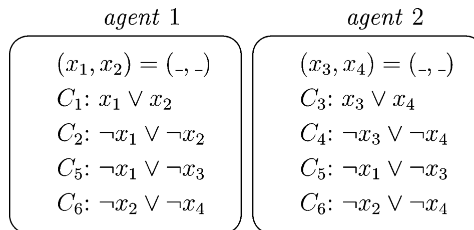

Fig. 2. DisSAT problem.

a CNF formula. A CNF formula consists of a set of clauses, where a *clause* is a disjunction of a number of literals and a *literal* is a variable or its negation. Given a CNF formula consisting of a set of clauses $C_1, C_2, \ldots, C_m$ on the variables $x_1, x_2, \ldots, x_n$, the problem is to determine whether the formula $C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is satisfiable. This decision problem was one of the first problems shown to be NP-complete. The SAT problem has attracted considerable attention recently in the AI community since many AI tasks, such as planning [12], theorem proving, and etc., can be encoded into it.

The DisSAT problem is the problem of finding models for formulae of multiple agents. Each agent in the DisSAT problem has its own formula and tries to find a model for it. Each agent's formula is defined on its local variables and some other agents' local variables and consists of a number of intra-agent clauses and a number of inter-agent clauses. An intra-agent clause is defined on only local variables, while an inter-agent clause is defined on both local variables and non-local variables. Fig. 2 illustrates a DisSAT problem instance, where there are two agents each having its own formula. Agent 1, for example, has the local variables x_1, x_2 and the clauses C_1, C_2, C_5, C_6. Since the clauses C_1, C_2 are defined on only agent 1's local variables, they are intra-agent clauses. On the other hand, since the clauses C_5, C_6 include not only agent 1's local variables x_1, x_2 but also agent 2's local variables x_3, x_4, they are inter-agent clauses.

2.2. Local search algorithm for the CSP

In the early 1990s, several researchers demonstrated that local search algorithms could successfully solve very large instances of various NP-complete problems that had been considered computationally expensive for traditional complete search algorithms [7,15, 17]. Although complete search algorithms have recently extended their reach and some of them can perform as well or better than local search algorithms [2], local search algorithms have still attracted plenty of attention because they are at least comparable to state-of-the-art complete search algorithms.

Local search algorithms for the CSP basically follow a similar procedure: an algorithm starts with an initial flawed "solution" and then repeats making local changes that reduce the *cost*, the total number of constraint violations, until finding a solution. However, one drawback of these algorithms is the possibility of getting stuck at a locally optimal point, a *local minimum*, where no local change can reduce the cost while there still exists at least one constraint violation. Various techniques have been proposed for escaping from local minima [7,16–19]. *Breakout* is one such technique that has been proposed by Morris [16]. The characteristics of the *breakout algorithm*, i.e., the local search algorithm that incorporates the breakout technique, are summarized as follows.

- A *weight* is associated with each constraint. For a *state* (a complete set of assignments for the variables), a cost is measured as the sum of the weights of violated constraints. The weights have 1 as their initial value.
- The local search algorithm proceeds as usual until a local minimum is reached.
- At a local minimum, the weights of constraints violated in the current state are increased so that the cost of the current state becomes larger than those of the neighboring states. The local search algorithm then resumes.

Similarly to other local search algorithms, this algorithm is incomplete; namely, it cannot prove the unsatisfiability of a problem explicitly, and, furthermore, it may fail to find a solution to a problem even if the problem is satisfiable. Therefore, for practical usage, we may need to set an upper bound of the number of repetitions to halt the procedure.

3. The distributed breakout algorithms

Recently, several researchers have developed *distributed constraint satisfaction algorithms* for solving the DisCSP [1,3,5,20,22,24,25]. In distributed constraint satisfaction algorithms, all of the agents perform their search procedures concurrently while communicating information on their search processes with each other. In this paper, we introduce a new series of distributed constraint satisfaction algorithms called *distributed breakout algorithms*. This is inspired by the breakout algorithm for the (centralized) CSP. This section gives a macroscopic view of the distributed breakout algorithms by describing three functionally divided operations: *core operation*, *breakout operation*, and *termination detection operation*.

3.1. Core operation

A macro-level behavior of the distributed breakout algorithms is that all of the agents repeatedly make local changes concurrently while coordinating their actions through a communication protocol. A major characteristic is that search and coordination are separated, i.e., all of the agents alternate search and coordination synchronously. Before describing the core operation of the distributed breakout algorithms, we first define the term *neighbors* as follows.

Definition 1 (*Neighbors*). For each agent i, i's neighbors are a subset of the agents that i has to contact in order to examine whether its inter-agent constraints are violated.

For example, in Fig. 1, agent 1's neighbors consist of agents 2 and 3 since agent 1 has inter-agent constraints (derived from l_{13} and l_{26}) that include variables belonging to agents 2 and 3.

The core operation of the distributed breakout algorithms is as follows: each agent first sets an initial set of assignments for variables and exchanges the set with its neighbors, then repeats the following until a solution to a DisCSP instance is found or a predetermined upper bound is reached. We call one such cycle a *round*.

(1) Each agent performs search to find a set of local changes that would reduce the cost (the sum of the weights of violated constraints) and then exchanges information on the set of local changes with its neighbors to identify potential conflicts.
(2) Each agent makes all of the local changes in the set found in the above step if they do not involve any potential conflict; otherwise, the agent makes all/some of the local changes in the set if they are still valid after the conflict resolution with its neighbors. Then, the agent exchanges a set of assignments for variables with its neighbors.

The details of identifying and resolving potential conflicts are specified in our implementations described in Sections 4 and 5.

Using this core operation, the agents can make multiple local changes concurrently in one round, where two types of messages are exchanged in turn. The first message is for notifying information on a set of local changes, called the *improve* message; the second is for notifying a set of assignments for variables, called the *ok?* message.

3.2. Breakout operation

Similar to local search algorithms for the CSP, the distributed breakout algorithms have the drawback of possibly getting stuck at local minima. In the distributed breakout algorithms, the local minimum is defined as follows.

Definition 2 (*Local minimum*). A state is called local minimum if some of the agents are violating constraints and no subset of the agents can make local changes resulting in a state with a lower cost.

Detecting the fact that the agents as a whole are in a local minimum requires global communication among agents. In a distributed environment, however, such global communication is usually expensive. Therefore, we introduce a weak notion of local minimum, a *quasi-local minimum*, which is detectable by local communication among agents.

Definition 3 (*Quasi-local minimum*). A state is called quasi-local minimum if there exists an agent that violates some of its constraints, and neither this agent nor some of its neighbors can make local changes resulting in a state with a lower cost.

We should note that if a state is a local minimum, that state is also a quasi-local minimum; on the other hand, if a state is a quasi-local minimum, it is not necessarily a local minimum since, in the state, there may exist some agents that can make local changes resulting in a state with a lower cost.

To escape from local minima, the distributed breakout algorithms use a technique called *breakout at quasi-local minima*, where an agent increases the weights of constraints that are known to be violated at a quasi-local minimum. Note that, in the distributed breakout algorithms, a weight is associated with each constraint, and each agent measures its cost as the sum of the weights of violated constraints.

3.3. Termination detection operation

A distributed constraint satisfaction algorithm has to be terminated when all of the agents obtain solutions to their local problems. To achieve this, each agent maintains a counter called *t_counter* in the distributed breakout algorithms. In each round agent i updates the counter, which is initialized to zero, as follows.

(1) If having constraint violations, agent i sets the value of its t_counter to zero; otherwise, it keeps the current value of its t_counter. Then, agent i sends the value of its t_counter to its neighbors.
(2) After receiving t_counters from all of its neighbors, agent i sets the value of its t_counter to the minimum value of i's and the neighbors' t_counters. Then, if neither agent i nor some of its neighbors have constraint violations, agent i increases the value of its t_counter by 1.

With these operations, we can ensure the following.

Theorem 1. *If the value of agent i's t_counter is d, every agent whose distance from agent i is within d obtains a solution to its local CSP.*

Note that the distance between agents is measured using the concept of neighbors. That is, if agent i has agent j among its neighbors, the distance from agent i to agent j is one; if agent i does not have agent j in its neighbors but one of agent i's neighbors does, the distance is two ($i \neq j$); generally, if agent i can reach agent j via at least $d - 1$ agents, the distance is d.

A proof of this theorem is as follows.

Proof. We can prove this inductively. When the value of agent i's t_counter is 1, the theorem obviously holds since the value of the counter is increased from 0 to 1 when neither agent i nor some of its neighbors have constraint violations. Next, we assume that the theorem holds when d is up to some specific value, say d_u. According to the operation for updating the counter, the value of agent i's counter increases from d_u to $d_u + 1$ if and only if neither agent i nor some of its neighbors have constraint violations and the value of their counters is equal to or larger than d_u. By the assumption, this is when every agent whose distance from agent i is within $d_u + 1$ obtains a solution to its local CSP. This means that the theorem also holds when d is $d_u + 1$. Therefore, the theorem is proved by induction. □

According to this theorem, if the value of some agent's t_counter reaches a distance that covers all of the agents, then all of the agents obtain solutions to their local CSPs, i.e., the DisCSP is solved. It may be difficult to know the exact value of the counter that can cover all of the agents, but fortunately it is sufficient to know an upper-bound value of it. Such an upper-bound value could be the diameter of the agent network in the DisCSP.

Fig. 3 summarizes the distributed breakout algorithms, where the above three operations are merged into one procedure.

```
randomly set an initial set of assignments for variables;
set the weight of all constraints to one;
t_counter ← 0;
round ← 0;
send a set of assignments for variables to neighbors;
while t_counter does not reach a specified upper-bound value do
  round ← round + 1;
  collect messages from neighbors;
  If there are constraint violations then
    t_counter ← 0;
  end if;
  LC ← a set of local changes that would reduce the cost;
  send t_counter and the information on LC to neighbors;
  collect messages from neighbors;
  t_counter ← minimum value of i's and neighbors' t_counters;
  if neither i nor some of its neighbors have constraint violations then
    t_counter ← t_counter + 1;
  end if;
  if i gets stuck at a quasi local minimum then
    increase the weights of violated constraints;
  end if;
  if LC does not involve any potential conflict then
    make all of the local changes in LC;
  else
    make all/some of the local changes in LC that are still valid after conflict resolution;
  end if;
  send a set of assignments for variables to neighbors;
end do;
```

Fig. 3. Distributed breakout algorithms (sketch of the procedure for agent i).

4. SINGLE-DB

SINGLE-DB, formerly called the distributed breakout algorithm in our previous paper [24], is one implementation of the distributed breakout algorithms. It is basically designed for the DisCSP where each agent has a single local variable and its related constraints. In this section we provide the basic ideas and the details of SINGLE-DB followed by an illustration of the solution process.

4.1. Basic ideas

Since an agent has only one local variable, searching for a set of local changes that would reduce the cost is equivalent to selecting an assignment for the variable that would reduce the cost. In SINGLE-DB, by following the *min-conflict heuristic* [15], we have an agent select an assignment for the variable that would maximally reduce the cost. Although this requires an agent to sweep all values in a variable domain, we can generally expect the size of a variable domain not to be so large.

To avoid a potential conflict among local changes, we allow an agent to make a local change if it would reduce the cost by more than any of its neighbors would; otherwise we make an agent withdraw a local change. Ties are broken deterministically such that, given that each agent has a unique ID number, we give priority to the agent with the smaller ID number if a pair of neighboring agents has the same degree of cost reduction. By resolving a potential conflict in this way, no pair of neighboring agents make their local changes simultaneously. On the other hand, if two agents are not neighboring, it is possible for them to make local changes simultaneously. This means that we can eliminate an oscillation among multiple states that might be typically caused by simultaneous local changes made by neighboring agents. To realize this, before making a local change, an agent needs to send its neighbors the degree of cost reduction that would be achieved by the local change, called the *improve*, as the information on the local change.

4.2. Details

The details of SINGLE-DB are illustrated in Figs. 4–6. Each agent follows these procedures, each of which is summarized as follows.

- In the MAIN procedure in Fig. 4, an agent sends an initial assignment for the variable to its neighbors via ok? messages (step 06) and repeats calling WAIT_OK and WAIT_IMPROVE until a solution is found or a predetermined upper bound of rounds, *Maxrounds*, is reached (step 07–10).
- In the WAIT_OK procedure in Fig. 5, an agent waits for all of the ok? messages issued by its neighbors and invokes SEND_IMPROVE (step 10 in WAIT_OK), where the agent selects an assignment for its variable that would give the maximal cost reduction (step 03 in SEND_IMPROVE) and sends improve messages, which include quadruples: variable, improve, current cost, and t_counter, to its neighbors (step 17 in SEND_IMPROVE).

```
      procedure Main
(01)    neighbors_i := number of i's neighbors;
(02)    val_i := randomly chosen assignment for variable x_i;
(03)    set the weight of all constraints to 1;
(04)    t_counter_i := 0;
(05)    Newweight := null;
(06)    send ok?(x_i, val_i, Newweight) to neighbors;
(07)    for round := 1 to Maxrounds do
(08)      Wait_ok;
(09)      Wait_improve;
(10)    end do
```

Fig. 4. MAIN of SINGLE-DB.

```
      procedure Wait_ok
(01)    counter := 0;
(02)    Agent_view_i := null;
(03)    wait_ok_mode := TRUE;
(04)    while wait_ok_mode do
(05)      when i receives ok?(x_j, val_j, Newweight) from j do
(06)        counter := counter + 1;
(07)        add (x_j, val_j) to Agent_view_i;
(08)        update the weights of constraints based on Newweight;
(09)        if counter = neighbors_i then
(10)          Send_improve;
(11)          wait_ok_mode := FALSE;
(12)        end if
(13)      end do
(14)    end do

      procedure Send_improve
(01)    cost_i := sum of weights of i's violated constraints under Agent_view_i;
(02)    improve_i := possible maximal degree of cost reduction;
(03)    new_val_i := assignment that would give the maximal degree of cost reduction;
(04)    if cost_i = 0 then
(05)      consistent_i := TRUE;
(06)    else
(07)      consistent_i := FALSE;
(08)      t_counter_i := 0;
(09)    end if
(10)    if improve_i > 0 then
(11)      canmove_i := TRUE;
(12)      quasi_lm_i := FALSE;
(13)    else
(14)      canmove_i := FALSE;
(15)      quasi_lm_i := TRUE;
(16)    end if
(17)    send improve(x_i, improve_i, cost_i, t_counter_i) to neighbors;
```

Fig. 5. WAIT_OK of SINGLE-DB.

```
     procedure WAIT_IMPROVE
(01)   counter := 0;
(02)   wait_improve_mode := TRUE;
(03)   while wait_improve_mode do
(04)     when i received improve(x_j, improve_j, cost_j, t_counter_j) do
(05)       counter := counter + 1;
(06)       t_counter_i := min(t_counter_i, t_counter_j);
(07)       if improve_j > improve_i then
(08)         canmove_i := FALSE;
(09)         quasi_lm_i := FALSE;
(10)       end if
(11)       if improve_j = improve_i and j < i then
(12)         canmove_i := FALSE;
(13)       end if
(14)       if cost_j > 0 then
(15)         consistent_i := FALSE;
(16)       end if
(17)       if counter = neighbors_i then
(18)         SEND_OK;
(19)         wait_improve_mode := FALSE;
(20)       end if
(21)     end do
(22)   end do
     procedure SEND_OK
(01)   Newweight := null;
(02)   if consistent_i then
(03)     t_counter_i := t_counter_i + 1;
(04)     if t_counter_i = maxdistance then
(05)       broadcast that a solution has been found;
(06)       terminate all procedures;
(07)     end if
(08)   end if
(09)   if quasi_lm_i then
(10)     for each violated constraint C do
(11)       increase the weight of C by one;
(12)       add (C, the new weight) to Newweight;
(13)     end do
(14)   end if
(15)   if canmove_i then
(16)     val_i := new_val_i;
(17)   end if
(18)   send ok? (x_i, val_i, Newweight) to neighbors;
```

Fig. 6. WAIT_IMPROVE of SINGLE-DB.

- In the procedure WAIT_IMPROVE in Fig. 6, an agent waits for all of the improve messages issued by its neighbors and invokes SEND_OK (step 18 in WAIT_IMPROVE), where, depending on the state the agent is in, the agent increases the value of t_counter by 1 (step 03 in SEND_OK), increases the weights of violated constraints by 1 (step 11 in SEND_OK), or makes a local change (step 16 in SEND_OK). It then sends its neighbors ok? messages, which include triplets: its variable, a current assignment of its variable, and information on constraints whose weights are updated (if weight up-

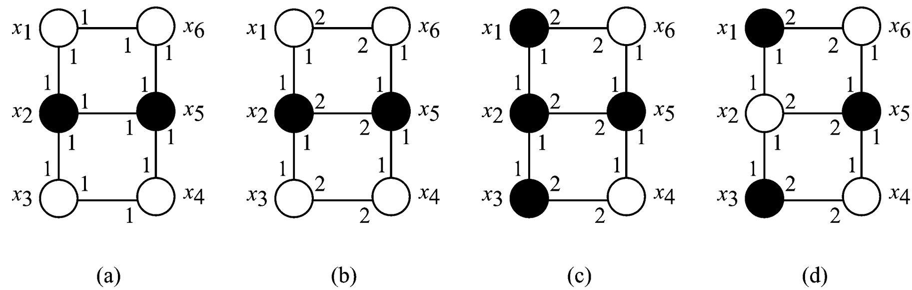

Fig. 7. Snapshots of the solution process of SINGLE-DB.

date occurs) (step 18 in SEND_OK). The current round ends at this step, and agent i then turns to the next round, where it starts again with the WAIT_OK procedure to wait for the ok? messages issued by its neighbors in the previous round.

We show snapshots of the solution process of SINGLE-DB in Fig. 7. This illustrates an instance of the distributed graph coloring problem, where agent i having the node x_i, whose possible colors are black and white, tries to find a color so that its related constraints, derived from links connected to x_i, are satisfied. We assume that initial assignments are chosen as in Fig. 7(a). Each agent communicates the initial assignment with its neighbors via ok? messages. After receiving ok? messages from all of its neighbors, each agent identifies a local change by selecting an assignment for a node that would achieve the maximal cost reduction under its current view and exchanges improve messages with its neighbors. Since the weight of all constraints is 1, no agent has a positive improve in this state. Therefore, agents increase weights of violated constraints, $x_1 =$ white and $x_6 =$ white, $x_2 =$ black and $x_5 =$ black, and $x_3 =$ white and $x_4 =$ white, by 1 as in Fig. 7(b). Then, the improve of agents 1, 3, 4, and 6 becomes 1, since each of them can reduce its cost from 2 to 1 by changing its assignment from white to black. On the other hand, those of agents 2 and 5 are still not positive. Through conflict resolution, both agents 1 and 3 are selected as the agents that have the right to make local changes, since each one of them has the largest improve among itself and its neighbors and has a smaller ID number than its competitor (agent 6 for agent 1 and agent 4 for agent 3). Each of them thus makes a local change from white to black as in Fig. 7(c). Next, the improve of agent 2 is 4, while those of the other agents are not positive. Therefore, only agent 2 makes a local change, and all of the constraints are satisfied as in Fig. 7(d).

5. MULTI-DB

MULTI-DB [9] is another implementation of the distributed breakout algorithms. A notable feature of MULTI-DB is that it can solve the DisSAT problem where each agent has multiple local variables and their related clauses. In this section we present the basic ideas and the details of MULTI-DB followed by its stochastic variations.

5.1. Basic ideas

To find a set of local changes that would reduce the cost, an agent in SINGLE-DB simply selects an assignment for its variable from the variable domain that would give the maximal degree of cost reduction. However, in MULTI-DB, since an agent has multiple local variables, it has to search in the space of combinations of assignments for the multiple local variables. Such a search space is usually much larger than the domain of one local variable. Therefore, we make each agent run a local search algorithm for a certain number of steps to find a set of local changes that would reduce the cost.

We use a variant of WalkSAT [18], which is known to be one of the most efficient local search algorithms for the SAT problem, as a local search algorithm for each agent. In each round of MULTI-DB, each agent starts from a current set of assignments for its local variables and searches for the set of assignments that would reduce the cost as follows: repeat the following procedure *Maxflips* (given as a parameter) times.

(1) Randomly select one of the clauses that is violated under a set of assignments for its local variables and a set of the most recently notified assignments for its neighbors' variables.
(2) In the selected clause, pick up one local variable to *flip* (change an assignment from true to false or vice versa) such that: if there are local variables in the clause that can be flipped without violating other clauses, pick up one of them randomly; otherwise, with probability p pick up any local variable in the clause randomly and with probability $1 - p$ pick up a local variable that minimizes the sum of weights of clauses that are currently satisfied but would be violated if the variable were flipped. Then, perform the flip to *virtually* change an assignment for its local variables.

After finishing the procedure, the agent identifies the best set of assignments (in terms of the degree of cost reduction) among those found during the repetitions to obtain a set of flips, called *Possflips*, whereby it can turn the initial set of assignments that our WalkSAT variant starts from into the best set of assignments.

Our WalkSAT variant uses the techniques called *sideway rule* and *tabu list* for efficiency. In the sideway rule, when choosing the best set of assignments, each agent breaks ties in favor of the one with the largest hamming distance from the initial set of assignments that our WalkSAT variant starts from. On the other hand, each agent maintains a tabu list that keeps the history of the sets of assignments for its local variables that have been sent to its neighbors in the latest TL rounds. The agent is prohibited from taking the sets of assignments in its tabu list during the procedure.

As described earlier, SINGLE-DB uses a simple method for resolving potential conflicts among local changes of neighboring agents. Using this method, no pair of neighboring agents make their local changes simultaneously. Although we could apply this method for the DisSAT problem where each agent has multiple local variables, such an approach may not be a good idea, since it misses the opportunity to make more local changes in parallel. We therefore introduce a more sophisticated conflict resolution method, that allows neighboring agents to make local changes simultaneously while ensuring that the total cost is reduced by their local changes. In this method, each agent exchanges *Possflips*, a set of

possible flips the agent is planning to perform, with its neighbors, and looks ahead to reason what the state in the next round would be like. Then, each agent identifies a *conflict among possible flips*, which is defined as follows.

Definition 4 (*Conflict among possible flips*). Two (or more) possible flips conflict with each other if (1) they belong to different agents, and (2) in the next round they would violate a clause that is currently satisfied.

In other words, two possible flips conflict with each other if they make a "fallacy of composition", that is, each of the possible flips contributes to reducing the cost if the other is not performed, but their composition accidentally contributes to non-reducing the cost.

If an agent detects that no possible flip in its own *Possflips* conflicts with those in its neighbors' *Possflips*, the agent can perform all of the flips in the *Possflips*. On the other hand, if an agent detects that one of the possible flips in its own *Possflips* conflicts with those in its neighbors' *Possflips*, the agent resolves the conflict by withdrawing the possible flip if it has the lowest improve, i.e., the lowest degree of cost reduction, among conflicting agents. Ties are broken deterministically such that we make an agent having a larger ID number withdraw its possible flip when two conflicting agents have the same improve.

After this conflict resolution process, there may be a case in which an agent's *Possflips* are partially withdrawn, i.e., some elements are withdrawn and the others are not. To deal with a partially withdrawn *Possflips*, one option would be that we make an agent withdraw it because we cannot say for certain whether the partially withdrawn *Possflips* can reduce the cost. However, in MULTI-DB, since an agent can flip (or keep the value of) any variable associated with a partially withdrawn *Possflips* without causing any conflict with its neighbors' possible flips, we make an agent execute the WalkSAT variant again over the variables associated with the partially withdrawn *Possflips* to find *Backupflips*, a subset of the partially withdrawn *Possflips* that can certainly reduce the cost. Since any possible flip in *Backupflips* obviously does not conflict with its neighbors' possible flips, an agent can perform all of the possible flips in *Backupflips* immediately.

5.2. Details

Details of MULTI-DB are illustrated in Figs. 8–12. MULTI-DB also consists of five procedures: MAIN, WAIT_OK, SEND_IMPROVE, WAIT_IMPROVE, and SEND_OK.

MULTI-DB and SINGLE-DB have almost the same main procedure. Fig. 8 shows that agent i starts MULTI-DB by randomly determining a set of assignments for its variables (step 03) and sending it to the neighbors via *ok?* messages (step 07). Then, agent i repeats WAIT_OK and WAIT_IMPROVE until a solution to a DisSAT problem instance is found or a predetermined upper bound of rounds, *Maxrounds*, is reached (steps 08–11).

In the WAIT_OK procedure in Fig. 9, agent i collects ok? messages from its neighbors while constructing $Currentview_i$, which records a set of current assignments for all of i's and its neighbors' variables (step 09). When ok? messages come from all of its neighbors, agent i calls SEND_IMPROVE in Fig. 10.

In SEND_IMPROVE, agent i makes *Possflips*, a set of possible flips that agent i can perform under $Currentview_i$, and sends such a set to its neighbors via improve messages.

```
     procedure MAIN
(01)   neighbors_i := number of i's neighbors;
(02)   Tabu_i := null;
(03)   Values_i := a set of randomly chosen assignments;
(04)   set the weight of all clauses to 1;
(05)   t_counter_i := 0;
(06)   Newweight := null;
(07)   send ok?(i, Values_i, Newweight) to neighbors;
(08)   for round := 1 to Maxrounds do
(09)     WAIT_OK;
(10)     WAIT_IMPROVE;
(11)   end do
```

Fig. 8. MAIN of MULTI-DB.

```
     procedure WAIT_OK
(01)   counter := 0;
(02)   Currentview_i := null;
(03)   Improveview_i := null;
(04)   add Values_i to Currentview_i;
(05)   wait_ok_mode := TRUE;
(06)   while wait_ok_mode do
(07)     when i receives ok?(j, Values_j, Newweight) from j do
(08)       counter := counter + 1;
(09)       add Values_j to Currentview_i;
(10)       update the weights of clauses based on Newweight;
(11)       if counter = neighbors_i then
(12)         SEND_IMPROVE;
(13)         wait_ok_mode := FALSE;
(14)       end if
(15)     end do
(16)   end do
```

Fig. 9. WAIT_OK of MULTI-DB.

More specifically, agent i first measures $cost_i$, the sum of weights of violated clauses under $Currentview_i$ (step 01). If $cost_i = 0$, agent i makes the state variable $consistent_i$ true (step 05), prohibits flips for all of its variables in the current round (step 32), and sends improve messages to its neighbors (step 33). On the other hand, if $cost_i \neq 0$, after making $consistent_i$ false and $t_counter_i$ zero (steps 07, 08), agent i performs local search (steps 09–26) to make *Possflips* (step 31), prohibits any other flips except for *Possflips* in the current round (step 32), and sends improve messages (step 33).

In WAIT_IMPROVE shown in Fig. 11, agent i collects improve messages from all of its neighbors while updating the state variables and the views. The views updated here are $Nextview_i$ (step 07) and $Improveview_i$ (step 08). $Nextview_i$ records a set of possible assignments for all of i's and its neighbors' variables in the next round; namely, it indicates what the state in the next round would be like. $Improveview_i$, on the other hand, records improves, i.e., the degrees of cost reduction, of agent i and its neighbors. When improve messages come from all of its neighbors, agent i calls SEND_OK shown in Fig. 12.

```
     procedure SEND_IMPROVE
(01)  cost_i := sum of weights of i's violated clauses under Currentview_i;
(02)  n := cost_i;
(03)  Nextview_i := Currentview_i;
(04)  if cost_i = 0 then
(05)    consistent_i := TRUE;
(06)  else
(07)    consistent_i := FALSE;
(08)    t_counter_i := 0;
(09)    T := Currentview_i;
(10)    for f = 1 to Maxflips do
(11)      v := i's variable selected by variable selection rule;
(12)      T := T with v's value flipped;
(13)      if i's variable values in T ∉ Tabu_i then
(14)        e := weighted sum of i's violated clauses under T;
(15)        if e < n then
(16)          n := e;
(17)          Nextview_i := T;
(18)        end if
(19)        if e = n then
(20)          update Nextview_i by sideway rule;
(21)        end if
(22)        if e = 0 then
(23)          break;
(24)        end if
(25)      end if
(26)    end do
(27)  end if
(28)  improve_i := cost_i − n;
(29)  add improve_i to Improveview_i;
(30)  Nextvalues_i := i's variable values in Nextview_i;
(31)  Possflips := values in Nextvalues_i that are different from those in Values_i;
(32)  prohibit any other flips except for Possflips;
(33)  send improve(i, Possflips, improve_i, cost_i, t_counter_i) to neighbors;
```

Fig. 10. SEND_IMPROVE of MULTI-DB.

In SEND_OK, agent i detects the termination condition (steps 03–05), increases the weights of violated clauses (steps 08–12), or performs variable flips (steps 14–35) depending on the situation in the current round, and then sends ok? messages to its neighbors (step 40). In detecting the termination condition, an agent in SINGLE-DB and MULTI-DB follows the same procedure. In increasing the weights of violated clauses, agent i first checks whether its $Nextview_i$ and $Currentview_i$ are the same (step 07). If this is true, it means that neither agent i nor its neighbors have a possible flip, and thus agent i detects a quasi-local minimum. When detecting a quasi-local minimum, an agent in SINGLE-DB and MULTI-DB basically follows the same procedure.

On the other hand, in performing variable flips, agent i proceeds as follows. For each clause that is not violated in the current round but would be violated in the next round, agent i identifies the possible flips (*Culprit_flips*) that would cause the violation and the agents (*Culprit_ag*) who plan to perform those flips (steps 15, 16); agent i also checks whether the following three conditions hold (step 17): (1) agent i is responsible for the

```
     procedure WAIT_IMPROVE
(01)   counter := 0;
(02)   wait_improve_mode := TRUE;
(03)   while wait_improve_mode do
(04)     when i receives improve(j, Possflips, improve_j, cost_j, t_counter_j) from j do
(05)       counter := counter + 1;
(06)       t_counter_i := min(t_counter_i, t_counter_j);
(07)       update Nextview_i by Possflips;
(08)       add improve_j to Improveview_i;
(09)       if cost_j > 0 then
(10)         consistent_i := FALSE;
(11)       end if
(12)       if counter = neighbors_i then
(13)         SEND_OK;
(14)         wait_improve_mode := FALSE;
(15)       end if
(16)     end do
(17)   end do
```

Fig. 11. WAIT_IMPROVE of MULTI-DB.

violation, (2) the violation is caused by at least two agents, and (3) agent i has the lowest *improve* among *Culprit_ag*. Ties in the third condition are broken deterministically by comparing agent ID numbers, i.e., the third condition holds when agent i has the largest ID number among agents with the same lowest improve. If the first two conditions hold, it is clear that *Culprit_flips* are in conflict with each other; in other words, this clause would be accidentally violated in the next round by the flips simultaneously performed by agents in *Culprit_ag*. To avoid this, an agent that meets the third condition withdraws one of its flips in *Culprit_flips* (step 18) (ties are broken randomly), thereby resolving a conflict in *Culprit_flips*.

As a result of the above procedure (steps 14–19), agent i sometimes withdraws some of its possible flips. However, when agent i does not have to withdraw any of its possible flips, this means that it can perform all of them without causing any accidental new clause violation. In this case, agent i performs those flips (step 21) and sends ok? messages to its neighbors (step 40). On the other hand, when agent i withdraws some of its possible flips, it performs local search again (steps 23–33) over the *flippable* variables, meaning the variables whose flips are not prohibited or withdrawn. The flips obtained from this local search obviously do not conflict with those of other agents, and agent i can therefore perform these flips immediately (step 34) to send ok? messages to its neighbors (step 40).

Fig. 13 depicts snapshots of a typical solution process of MULTI-DB for the DisSAT problem instance shown in Fig. 2. As shown at the top of Fig. 13, we assume that both agents 1 and 2 assign TRUE for all of their variables and exchange them via ok? messages. As a result, each finds that the current cost is 3 because agent 1 violates the clause C_2, C_5, C_6 and agent 2 violates the clause C_4, C_5, C_6 (we assume that the weight of all clauses is one). Then, each performs local search to find *Possflips*. Agent 1 finds that flipping x_1 can reduce the cost from 3 to 1 and agent 2 also finds that flipping x_3 can do this. These possible flips, x_1 by agent 1 and x_3 by agent 2, are exchanged via improve

```
      procedure SEND_OK
(01)  Newweight := null;
(02)  if consistent_i then
(03)     t_counter_i := t_counter_i + 1;
(04)     if t_counter_i = maxdistance then
(05)        broadcast that a solution has been found; terminate all procedures; end if
(06)  else
(07)     if Nextview_i = Currentview_i then
(08)        for each violated clause C under Currentview_i do
(09)           increase the weight of C by one;
(10)           if C is an inter-agent clause then
(11)              add (C, the new weight) to Newweight; end if
(12)        end do
(13)     else
(14)        for each clause not violated under Currentview_i but violated under Nextview_i do
(15)           Culprit_flips := flips causing such violation;
(16)           Culprit_ag := agents planning to perform Culprit_flips;
(17)           if (i ∈ Culprit_ag) ∧ (|Culprit_ag| ≥ 2)
                  ∧ (i has the lowest improve among Culprit_ag) then
(18)              withdraw one of i's flips in Culprit_flips; end if
(19)        end do
(20)        if i hasn't withdrawn any flip then
(21)           Values_i := Nextvalues_i;
(22)        else
(23)           T := Currentview_i; n := cost_i; Nextview_i := T;
(24)           for f = 1 to Maxflips do
(25)              v := i's flippable variable selected by variable selection rule;
(26)              T := T with v's value flipped;
(27)              if i's variable values in T ∉ Tabu_i then
(28)                 e := weighted sum of i's violated clauses under T;
(29)                 if e < n then n := e; Nextview_i := T; end if
(30)                 if e = n then update Nextview_i by sideway rule; end if
(31)                 if e = 0 then break; end if
(32)              end if
(33)           end do
(34)           Values_i := i's variable values in Nextview_i;
(35)        end if
(36)     end if
(37)  end if
(38)  if |Tabu_i| = TL then delete the oldest element in Tabu_i; end if
(39)  add Values_i to Tabu_i;
(40)  send ok(i, Values_i, Newweight) to neighbors;
```

Fig. 12. SEND_OK of MULTI-DB.

messages. After exchanging the possible flips, each checks whether there is a conflict among them. Agent 1, for example, can reason that in the next round the state would be $\{x_1 = \mathrm{F}, x_2 = \mathrm{T}, x_3 = \mathrm{F}, x_4 = \mathrm{T}\}$, which violates only C_6. Accordingly, agent 1 finds that flipping x_1 does not conflict with flipping x_3, since they will not cause accidental new clause violation, and hence performs the flip of x_1. Agent 2, on the other hand, follows the same and performs the flip of x_3. After flipping these variables, both exchange their new values with each other and go to the second round.

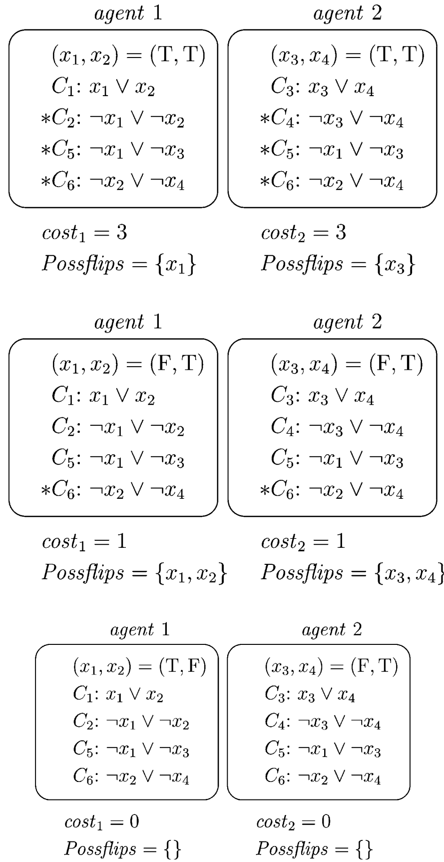

Fig. 13. Snapshots of the solution process of MULTI-DB (the clauses marked by $*$ are violated).

In the second round shown in the middle of Fig. 13, since each agent violates C_6, it performs local search to find possible flips. As a result, agent 1 finds that flipping x_1 and x_2 leads to no violation, while agent 2 also finds that flipping x_3 and x_4 leads to no violation. Then, they exchange these possible flips. After exchanging them, both agents now find that these possible flips are in conflict with each other, since these flips would make the state in the next round $\{x_1 = \mathrm{T}, x_2 = \mathrm{F}, x_3 = \mathrm{T}, x_4 = \mathrm{F}\}$, where C_5 is accidentally violated. Therefore, agent 2 withdraws the flip of x_3 in this case because agent 2's improve is one, which ties with agent 1's improve, and its ID number is larger than agent 1's ID number. After the withdrawal, agent 2 reconsiders whether x_4 should be flipped. More specifically, agent 2 performs local search again over x_4, the only flippable variable, to determine an assignment for x_4. In this case, agent 2 determines that an assignment for x_4 should be TRUE. To sum up, in the second round agent 1 flips x_1 and x_2 just as planned, while agent 2, on the other hand, flips no variable. After such flipping, they exchange their new values and reach the state shown at the bottom of Fig. 13. This state is obviously a solution to this problem instance, which is to be detected by the agents within a few more rounds.

5.3. Stochastic variations

The original version of MULTI-DB used *random restart*, where the agents simultaneously reinitialize a set of assignments for variables if a solution has not been discovered after a fixed number of rounds [9]. This is a simple and effective method for agents to avoid stagnation of the search caused by a bad set of initial assignments for variables. Indeed, we observed that MULTI-DB was improved by random restart with a carefully chosen cutoff round. However, it is difficult to set an appropriate cutoff round at which the agents restart randomly, since the search performance is very sensitive to the cutoff round. Moreover, when restarting, the agents waste all the effort that they have put into searching for a solution and restart their search from scratch. Clearly, this process is wasteful. Therefore, we introduce new methods for agents to avoid stagnation of the search: *random break* and *random walk*.

Random break is similar in its basic idea to DBA(wp) [27]. DBA(wp) is a stochastic variation of SINGLE-DB, which proceeds as SINGLE-DB, except that when two neighboring agents have the same improve, they make local changes probabilistically (both of them may or may not change, or just one of them may change). DBA(wp) adds some randomness to SINGLE-DB but does not always ensure cost reduction. However, random break ensures cost reduction in a non-deterministic way. In this method, each agent keeps a random variable whose value is randomly chosen in each round and sends this value to its neighbors via improve messages. When two neighboring agents have the same improve, they break the tie by giving the right to make changes to the agent with the smaller value for the random variable instead of the agent with the smaller ID number. Note that, in this method, a tie break does not always occur only in one direction, since a value for the random variable varies in each round.

Random walk is a method that allows randomized up-hill moves with a fixed probability [10,18]. With a fixed probability rw, this method works in the following way: (1) select a currently violated clause randomly, (2) select one of the variables in the clause randomly, and (3) flip the variable. This is called a *random walk step*. We introduce random walk into our algorithm by having each agent perform a random walk step with a fixed probability. More specifically, an agent proceeds as usual, except that just before flipping some variables, with a fixed probability rw it replaces those variables with a variable determined by the random walk step and flips it. Obviously, since this new variable flip is selected with no regard as to how it contributes to the current cost, it may increase the cost.

We first combine random break with MULTI-DB and call the resultant algorithm MULTI-DB$^+$. Then, we add random walk to MULTI-DB$^+$ and call the resultant algorithm MULTI-DB^{++}.

6. Evaluation

We evaluated the performance of SINGLE-DB and the family of MULTI-DB through experiments using satisfiable problem instances from the *uniform random 3-SAT* in SATLIB (http://www.satlib.org/). The *uniform random 3-SAT* is generally considered to be one of the hardest classes of the 3-SAT problem. In these experiments, to convert a SAT

problem instance into a DisSAT problem instance, we evenly partitioned n variables of a SAT problem instance among k agents and assigned each clause to all of the agents having variables in the clause.

To apply SINGLE-DB to a DisSAT problem instance in which each agent has multiple local variables and their related clauses, we used the following method: first, introduce additional *virtual* agents to distribute multiple local variables so that each resulting agent will have exactly one variable, then run SINGLE-DB for the given instance involving these virtual agents.

We compared our algorithms with MULTI-AWC [26]. MULTI-AWC is one of the most efficient algorithms for solving the DisCSP where each agent has multiple local variables and their related constraints. Just as with AWC, the performance of MULTI-AWC can be enhanced by employing an appropriate nogood learning technique. Nogood learning, however, generally requires agents to have a lot of extra memory. For example, AWC with full learning demands extra memory during algorithm execution, and such memory demand grows exponentially in the worst case. In addition, agents have to check whether the nogoods that have been learned are violated, which also requires a lot of computation to be performed. Accordingly, MULTI-AWC used in the experiments did not employ nogood learning for a fair comparison with our algorithms, which requires agents to have very little memory.

In order to implement a distributed algorithm, we have to determine an underlining distributed system on which the algorithm is executed. Although the assumption of a communication model for our algorithm is so common that we can implement the algorithm on any type of distributed system, for simplicity we used the *synchronous distributed system* in our experiments. The synchronous distributed system is a distributed system in which all of the agents repeat the *cycle* of communication and computation simultaneously [13]. One cycle consists of the following three steps: (1) all of the agents read incoming messages that were issued in the previous cycle, (2) all of the agents perform their local computation, and (3) all of the agents send messages to other agents. We implemented all of the algorithms on a simulator of the synchronous distributed system and measured the following as their communication and computation costs, respectively.

#cycles: the number of cycles consumed until the agents find one solution to a DisCSP instance. Since agents communicate with each other in every cycle, the number of cycles increases with the amount of communication among agents. Thus, we regard one cycle as the unit of communication cost and used the number of cycles as the communication cost of an algorithm. Note that one round in the distributed breakout algorithms, in which the agents perform one series of WAIT_OK and WAIT_IMPROVE, corresponds to two cycles on this simulator.

#flips: the total sum of the maximal number of flips over the agents at each cycle until the agents find one solution. More specifically, we calculate such a measure like this: at each cycle we first identify the *bottleneck agent*, which performed the maximal number of flips in its local computation, and sum all of the maximal numbers of flips over all consumed cycles. Although the amount of computation at each cycle varies among the agents, the total amount of computation is dominated by the bottleneck agents. This measure can thus be considered the computation cost

of an algorithm. Note that for MULTI-AWC, we measured the total sum of the maximal number of visited search nodes (instead of the maximal number of flips) over the agents at each cycle until the agents found one solution.

We set the upper bound of the number of cycles to $5000n$, where n is the total number of variables, and cut off a run if it exceeded the upper-bound cycle in order to finish our experiments within a reasonable amount of time. For a run cut off, we used #cycles and #flips at the time the run was cut off.

We set the parameters in the MULTI-DB family as follows.

- $\mathit{Maxrounds} = 2500n$. Since we cut off a run at $5000n$ cycles, we set *Maxrounds* in this way.
- $\mathit{Maxflips} = n/k$, $p = 0.3$, $TL = 5$. These are the parameters for the local search procedure, the WalkSAT variant, each agent performs. *Maxflips* specifies the number of flips each agent performs at each call of the local search procedure. Since n is the total number of variables and k is the number of agents, setting *Maxflips* in this way allows each agent to perform flips at most the number of times that corresponds to the average number of variables of each agent. The parameter p is the probability used in the WalkSAT variant, with which a variable to flip is selected randomly from a selected violated clause. The parameter TL is the length of the tabu list also used in the WalkSAT variant.
- $rw = 1/(5k)$. This is a parameter only for MULTI-DB^{++} that specifies the probability with which each agent performs a random walk step. By setting rw in this way, we can expect that one agent performs a random walk step for every five rounds.

In the uniform random 3-SAT in SATLIB, there are 100 satisfiable problem instances for each n (1000 for $n = 100$). We gave one randomly chosen initial set of assignments for variables for each problem instance of each combination of (n, k) and made each algorithm run. Fig. 14 indicates the mean #cycles and the mean #flips over 100 (or 1000) runs for each combination of (n, k). Table 1 indicates the ratio of runs that were successfully completed within the upper bound of the number of cycles. Note that the results of SINGLE-DB do not depend on k, since SINGLE-DB converts an original problem instance involving k agents into the one where each virtual agent has exactly one variable. From these results, we can observe the following.

On comparing MULTI-DB and MULTI-AWC, MULTI-DB is better than MULTI-AWC in all cases in terms of both the mean #cycles and the mean #flips. Moreover, the differences become greater as the number of variables increases, and MULTI-DB achieves at least one order of magnitude improvement in many cases. On the other hand, MULTI-DB obtains a lower success ratio in the following four cases: $(n, k) = (100, 5)$, $(100, 10)$, $(100, 20)$, and $(125, 5)$. These results indicate that although MULTI-DB scales up better than MULTI-AWC, it sometimes shows very poor performance regardless of the problem size. We conjecture that this poor performance of MULTI-DB is caused by its lack of randomness. Indeed, except for each agent making *Possflips* by using the WalkSAT variant, which involves some randomness, the search process of MULTI-DB is inherently deterministic.

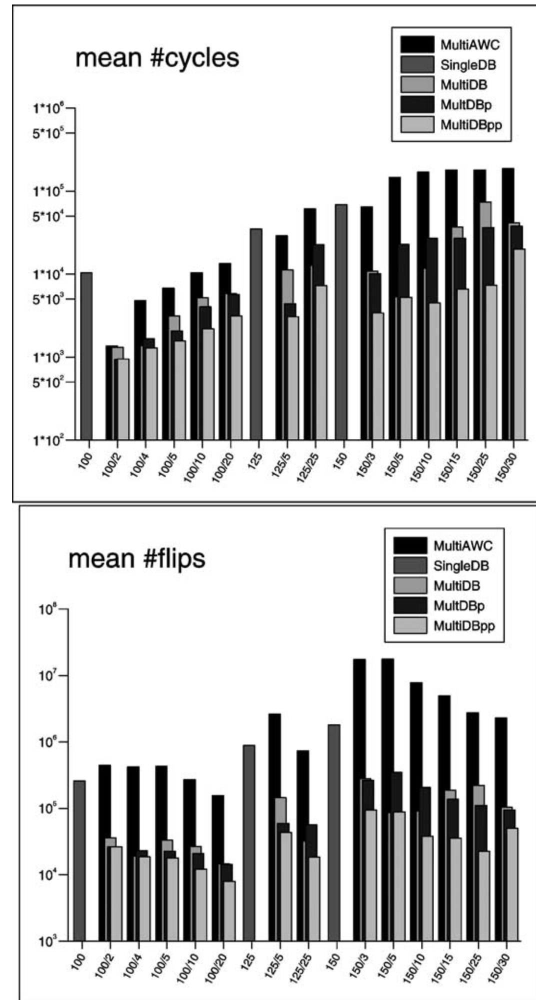

Fig. 14. Mean costs of algorithms (showing MultiAWC, MultiDB, MultiDBp, and MultiDBpp from the back to the front at every n/k and SingleDB separately from the rest at every n).

On comparing MULTI-DB^{+} and MULTI-DB, we can see that MULTI-DB^{+} is slightly better for some cases but not for others. These results are not so impressive, so it does not follow that adding only random break is effective.

On the other hand, on comparing MULTI-DB^{++} and MULTI-DB, MULTI-DB^{++} shows very clear performance improvement. We should point out that MULTI-DB^{++} successfully completes its run within the upper bound of the number of cycles ($5000n$ cycles) in all cases. Furthermore, for almost all cases, MULTI-DB^{++} has fewer mean #cycles and mean #flips, with the only exception being the mean #flips for $(n, k) = (150, 5)$, although we can say that the difference is relatively minor. However, although the results are not shown, we also observed that in MULTI-DB^{++} the median #cycles and #flips slightly in-

Table 1
Success ratio on uniform random (Dis)3-SAT where n variables are divided by k agents

n	k	SINGLE-DB	MULTI-AWC	MULTI-DB	MULTI-DB$^+$	MULTI-DB^{++}
100	2	0.991	1.000	1.000	1.000	1.000
	4		1.000	1.000	0.999	1.000
	5		1.000	0.998	1.000	1.000
	10		0.999	0.996	0.997	1.000
	20		0.998	0.995	0.996	1.000
125	5	0.98	1.00	0.99	1.00	1.00
	25		0.97	1.00	0.97	1.00
150	3	0.95	0.98	0.99	0.99	1.00
	5		0.91	1.00	0.98	1.00
	10		0.90	1.00	0.97	1.00
	15		0.87	0.96	0.97	1.00
	25		0.83	0.92	0.98	1.00
	30		0.90	0.96	0.97	1.00

crease in many cases. These results suggest that the stagnation of the search processes of multiple agents, which is observed in a very few runs of MULTI-DB, can be avoided by adding random walk at the cost of slightly distracting their search processes.

On comparing SINGLE-DB and MULTI-DB, MULTI-DB is better in almost all cases in terms of both the mean #cycles and the mean #flips. The reason would be that in SINGLE-DB a *real* agent, which originally owns multiple local variables, fails to make better use of the knowledge of its local problem. By introducing additional virtual agents and distributing its multiple local variables so that each resulting agent will have one variable, a real agent obtains the applicability of SINGLE-DB but loses quick access to the knowledge of its local problem. This increases the amount of communication among the agents and thus SINGLE-DB results in deteriorated performance.

Recently, some researchers have investigated the distributed stochastic algorithm (DSA) [5,6,28]. In DSA, agents sometimes act *incoherently* in such a way that at a certain probability each agent i makes a local change without caring the possibility that its local change would conflict with those of neighboring agents. Among some variations of DSA, Fitzpatrick and Meertens have reported that the CFP algorithm shows the best performance on satisfiable instances of the distributed k-coloring problem [6]. In CFP, each agent acts as follows. It first randomly chooses an assignment for its variable and sends the assignment to its neighbors. Then, it repeats a sequence of steps until a termination condition is met. Each agent collects the assignments of neighbors at each step; if there are constraint violations, with probability α the agent chooses an assignment giving the largest cost reduction, and with probability $1 - \alpha$ it keeps its current assignment, after which the agent sends an assignment to its neighbors if the assignment is new. Zhang and Xing show experimental results to compare DSA and SINGLE-DB on the distributed scan scheduling problem, which can be formulated as the distributed graph coloring problem, in terms of solution quality and communication cost [28]. Their conclusion is that CFP is superior to SINGLE-DB in terms of both solution quality and communication cost. However, the distributed scan scheduling problem is an optimization problem whose goal is to minimize the total weight of violated (soft) constraints. For a decision problem whose goal is to completely

Table 2
Success ratio on uniform random (Dis)3-SAT where n variables are divided by k agents (Note: first 100 instances are tried in the $n = 100$ case)

n	k	CFP $\alpha = 0.3$	CFP $\alpha = 0.5$	CFP $\alpha = 0.7$
100	10	0.03	0.09	0.11
	20	0.01	0.00	0.03
125	25	0.00	0.00	0.03
150	15	0.00	0.00	0.05
	25	0.01	0.01	0.04
	30	0.00	0.00	0.03

satisfy all of the constraints, the conclusion must be different because CFP can reach a sub-optimal solution very quickly but has no explicit technique for escaping from local minima. In fact, we adapted the CFP algorithm to the DisSAT problem and tested its performance on some sets of instances in the uniform random (Dis)3-SAT. Table 2 indicates the success ratios of the CFP algorithm within the upper bound of $5000n$ cycles when we control α over 0.3, 0.5, and 0.7. From this, it is clear that the CFP algorithm rarely reaches a solution. Recall that the success ratio of MULTI-DB^{++} is 1 in all cases.

7. Conclusions

We have presented the distributed breakout algorithms along with four implementations: SINGLE-DB, MULTI-DB, MULTI-DB^{+}, and MULTI-DB^{++}. SINGLE-DB is a distributed breakout algorithm for solving the DisCSP, where each agent has only one local variable and its related constraints. MULTI-DB, on the other hand, is another distributed breakout algorithm for solving the DisSAT problem, where each agent has multiple local variables and their related clauses. MULTI-DB^{+} and MULTI-DB^{++} are stochastic variations of MULTI-DB, where we introduce *random break* to MULTI-DB to make MULTI-DB^{+} and *random walk* to MULTI-DB^{+} to make MULTI-DB^{++}. According to our experimental evaluation, SINGLE-DB, MULTI-DB, and MULTI-DB^{+} scale up better but show very poor performance in a few cases. On the other hand, MULTI-DB^{++}, which uses random walk, shows remarkable performance improvement.

References

[1] A. Armstrong, E. Durfee, Dynamic prioritization of complex agents in distributed constraint satisfaction problems, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97), Nagoya, Japan, 1997, pp. 620–625.

[2] R.J. Bayardo Jr., R.C. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), Providence, RI, 1997, pp. 203–208.

[3] C. Bessière, A. Maestre, P. Meseguer, Distributed dynamic backtracking, in: Proceedings of the IJCAI-01 Workshop on Distributed Constraint Reasoning, Seattle, WA, 2001, pp. 9–16.

[4] S.E. Conry, K. Kuwabara, V.R. Lesser, R.A. Meyer, Multistage negotiation for distributed constraint satisfaction, IEEE Trans. Systems Man Cybernet. 21 (6) (1991) 1462–1477.
[5] M. Fabiunke, Parallel distributed constraint satisfaction, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 1999, pp. 1585–1591.
[6] S. Fitzpatrick, L. Meertens, An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs, in: Proceedings of the First Symposium on Stochastic Algorithms: Foundations and Applications, 2001, pp. 49–64.
[7] J. Gu, Efficient local search for very large-scale satisfiability problems, SIGART Bull. 3 (1) (1992) 8–12.
[8] K. Hirayama, M. Yokoo, The effect of nogood learning in distributed constraint satisfaction, in: Proceedings of the 20th IEEE International Conference on Distributed Computing Systems, 2000, pp. 169–177.
[9] K. Hirayama, M. Yokoo, Local search for distributed SAT with complex local problems, in: Proceedings of the First International Joint Conference on Autonomous Agents & Multi-Agent Systems, 2002, pp. 1199–1206.
[10] H.H. Hoos, On the run-time behaviour of stochastic local search algorithms for SAT, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), Orlando, FL, 1999, pp. 661–666.
[11] M.N. Huhns, D.M. Bridgeland, Multiagent truth maintenance, IEEE Trans. Systems Man Cybernet. 21 (6) (1991) 1437–1445.
[12] H.A. Kautz, B. Selman, Pushing the envelope: planning, propositional logic, and stochastic search, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), Portland, OR, 1996, pp. 1194–1201.
[13] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann, San Mateo, CA, 1996.
[14] C. Mason, R. Johnson, DATMS: a framework for distributed assumption based reasoning, in: L. Gasser, M. Huhns (Eds.), Distributed Artificial Intelligence, vol. 2, Morgan Kaufmann, San Mateo, CA, 1989, pp. 293–318.
[15] S. Minton, M.D. Johnston, A.B. Philips, P. Laird, Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems, Artificial Intelligence 58 (1–3) (1992) 161–205.
[16] P. Morris, The breakout method for escaping from local minima, in: Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93), Washington, DC, 1993, pp. 40–45.
[17] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA, 1992, pp. 440–446.
[18] B. Selman, H.A. Kautz, B. Cohen, Noise strategies for improving local search, in: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), Seattle, WA, 1994, pp. 337–343.
[19] Y. Shang, B.W. Wah, A discrete Lagrangian-based global-search method for solving satisfiability problems, J. Global Optim. 12 (1998) 61–99.
[20] M.C. Silaghi, D. Sam-Haroud, B. Faltings, Asynchronous search with aggregations, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), Austin, TX, 2000, pp. 917–922.
[21] K.P. Sycara, S. Roth, N. Sadeh, M. Fox, Distributed constrained heuristic search, IEEE Trans. Systems Man Cybernet. 21 (6) (1991) 1446–1461.
[22] W.E. Walsh, M. Yokoo, K. Hirayama, M.P. Wellman, On market-inspired approaches to propositional satisfiability, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01), Seattle, WA, 2001, pp. 1152–1158.
[23] M. Yokoo, E.H. Durfee, T. Ishida, K. Kuwabara, Distributed constraint satisfaction for formalizing distributed problem solving, in: Proceedings of the 12th IEEE International Conference on Distributed Computing Systems, 1992, pp. 614–621.
[24] M. Yokoo, K. Hirayama, Distributed breakout algorithm for solving distributed constraint satisfaction problems, in: Proceedings of the Second International Conference on Multi-Agent Systems, 1996, pp. 401–408.
[25] M. Yokoo, E.H. Durfee, T. Ishida, K. Kuwabara, The distributed constraint satisfaction problem: formalization and algorithms, IEEE Trans. Knowledge Data Engrg. 10 (5) (1998) 673–685.
[26] M. Yokoo, K. Hirayama, Distributed constraint satisfaction algorithm for complex local problems, in: Proceedings of the Third International Conference on Multi-Agent Systems, 1998, pp. 372–379.
[27] W. Zhang, L. Wittenburg, Distributed breakout revisited, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02), Edmonton, AB, 2002.
[28] W. Zhang, Z. Xing, Distributed breakout vs. distributed stochastic: a comparative evaluation on scan scheduling, in: Proceedings of the Third International Workshop on Distributed Constraint Reasoning, 2002, pp. 192–201.