Local Search for Distributed SAT with Complex Local Problems

Katsutoshi Hirayama
Kobe University of Mercantile Marine
5-1-1 Fukae-minami-machi, Higashinada-ku
Kobe 658-0022, JAPAN
hirayama@ti.kshosen.ac.jp

Makoto Yokoo
NTT Communication Science Laboratories
2-4 Hikaridai, Seika-cho, Soraku-gun
Kyoto 619-0237, JAPAN
yokoo@cslab.kecl.ntt.co.jp

ABSTRACT

A distributed constraint satisfaction problem(DisCSP) is a general framework that can formalize various application problems in Multi-Agent Systems. The authors have developed a series of algorithms for solving DisCSPs, including an iterative improvement algorithm called the distributed breakout (DB) algorithm. This algorithm, however, deals only with DisCSPs where each agent has exactly one local variable and the relevant constraints to the variable. In this paper, we propose a new algorithm called MULTI-DB for solving distributed SAT (DisSAT) where each agent has multiple local variables and the relevant clauses to the variables. We conduct an experiment to compare MULTI-DB with the previous algorithm called MULTI-AWC on well-known (Dis)3-SAT benchmarks. The results are very impressive since MULTI-DB has much less average communication and computation costs for almost all cases (at least an order of magnitude less for larger problems). We also identify a trade-off between communication and computation costs of algorithms when we vary the degree of decentralization.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Coherence and coordination, Multiagent systems*

General Terms

Algorithms

Keywords

distributed constraint satisfaction, propositional satisfiability, local search

AAMAS'02, July 15-19, 2002, Bologna, Italy.

1. INTRODUCTION

Distributed constraint satisfaction problems (DisCSPs) are constraint satisfaction problems (CSPs) where variables and constraints are distributed among multiple agents. Since many problems in Multi-Agent Systems (MAS) can be described as DisCSPs, a lot of studies have been made on algorithms for DisCSPs [2, 5, 12, 13, 17], complexity issues on DisCSPs [4, 7], mapping real-life MAS problems into DisCSPs [9], etc., over the past several years.

Previous distributed constraint satisfaction algorithms can be roughly divided into two groups. One is characterized by asynchronism in the value assignment to variables. This group includes the *asynchronous backtracking* (ABT) algorithm [14], the *asynchronous weak-commitment search* (AWC) algorithm [14] and its descendants [5, 16, 6, 12]. In these algorithms, agents decide the values to their variables without mutually excluding their "undesirable" simultaneous decisions. The other group is characterized by local-synchronism in the value assignment to variables. One representative algorithm in this group is the *distributed breakout* (DB) algorithm [15]. Unlike ABT or AWC, agents in DB mutually exclude undesirable simultaneous decisions only among their neighboring agents.

In ABT and AWC, nogood learning plays an important role in the search performance. By making each agent record a nogood, which is a new constraint discovered by an agent, both ABT and AWC are guaranteed to be complete, and moreover, the communication cost of AWC can be reduced even by partial nogood learning [6]. However, when solving critically hard DisCSPs, both algorithms are likely to produce a huge number of nogoods and some agents may hence consume a lot of memory to record these nogoods. This problem can be serious especially when agents have to solve a DisCSP instance where each agent is assigned a large and complex local problem while being permitted to use only a limited amount of memory.

On the other hand, since DB can solve hard DisCSPs even without a memory consumptive method like nogood learning, DB is more appropriate for situations where an agent is permitted to use only a limited amount of memory. However, DB is restricted to DisCSPs where each agent has one local variable and its relevant constraints, and hence is unable to deal with DisCSPs where each agent has multiple local variables.

We present a new algorithm called MULTI-DB, where DB is extended so that it can handle problems where each agent has multiple local variables. MULTI-DB is adapted for Dis-

tributed SAT, which is an important class of DisCSPs where variables and clauses of a propositional formula in conjunctive normal form are distributed among multiple agents. The outline of the search process in MULTI-DB is as follows. Each agent repeats (until a solution is found): performing a local search procedure (a variant of WalkSAT [11]) to find more preferable values to its variables; exchanging the possible "flips" (a flip means a change of value from true to false or vice-versa) for the variables among neighboring agents; resolving conflicts among these possible flips; flipping the variables whose flips are permitted; and exchanging new values to its variables among neighboring agents.

This paper is organized as follows. We first give the background of this research, which consists of the definition of DisCSP/DisSAT and the outline of DB in Section 2. Then, in Section 3, we present our new algorithm called MULTI-DB along with the basic idea and the detailed description of the procedure. Simple examples follow to illustrate the performance of the procedure. Next, in Section 4, we show experimental evaluations using well-known benchmark (Dis)3-SAT problems and discuss the results. Finally, we conclude this work in Section 5.

2. BACKGROUND

2.1 Distributed CSP/SAT

Propositional satisfiability (SAT) is the problem of deciding if there is an assignment for variables in a propositional formula that makes the formula true. SAT has attracted considerable attention recently within the AI community. In the field of AI planning in particular, efficient planning algorithms have been developed under the scheme of SAT-based planning where a planning problem is compiled into a propositional formula in conjunctive normal form (CNF formula) and solved by an efficient SAT solver [8]. A CNF formula is a conjunction of clauses, where a *clause* is a disjunction of literals and a *literal* is a propositional variable or its negation. The following formula over the variables $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ is an example of a CNF formula.

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3 \vee \neg x_6) \wedge (x_2 \vee \neg x_4 \vee \neg x_5) \wedge (x_3 \vee x_5 \vee x_6) \wedge (\neg x_4 \vee \neg x_5 \vee x_6) \quad (1)$$

Distributed SAT (DisSAT) is a problem where variables and clauses in a CNF formula are distributed among multiple agents. We believe that many MAS problems can be compiled into DisSAT since we have observed so many centralized problems can be compiled into SAT. Furthermore, it is important to develop an efficient algorithm on a formal model of MAS problems such as DisCSP/DisSAT, since the existence of an efficient algorithm may encourage us to compile a MAS problem into that formal model.

The DisSAT we deal with in this paper is as follows:

Component: a set of agents: $1, 2, \ldots, k$, where each is assigned a set of variables and clauses.

Variable assignment rule: variables are partitioned into all agents, i.e., no variable is shared among agents.

Clause assignment rule: a clause is assigned to all of the agents involved with the clause. In other words, an agent has all of the clauses relevant to its assigned variables.

Solution: the state where every agent finds truth values to its assigned variables that satisfy all of its assigned clauses.

Take CNF formula (1), for example. This formula consists of six variables and five clauses (we refer to them as C_1, C_2, C_3, C_4, C_5 from the left). If there exist two agents, say a_1 and a_2, where each is assigned $\{x_1, x_2, x_3\}$ and $\{x_4, x_5, x_6\}$, respectively, then the above rules says that a_1 has $\{C_1, C_2, C_3, C_4\}$ while a_2 has $\{C_2, C_3, C_4, C_5\}$. In this case, $\{C_2, C_3, C_4\}$ are assigned to both agents because each of them includes both agents' variables. Clauses that include the variables of multiple agents, such as C_2, C_3, and C_4, are called *inter-agent clauses*. On the other hand, clauses that include only the variables of one agent, such as C_1 and C_5, are called *intra-agent clauses*. An agent (say i) usually has both inter- and intra-agent clauses and each of i's inter-agent clauses includes some *external variables* that belong to other agents. We call agents that i's external variables belong to i's *neighboring agents*. We refer to the set of all of i's neighboring agents as i's *neighbors*.

2.2 Distributed Breakout

DB is a distributed constraint satisfaction algorithm for solving DisCSPs where each agent has exactly one local variable. The outline of DB is as follows [15].

Each agent (say i) first assigns a randomly chosen value to its variable and sends the value to its neighbors via *ok messages*. Upon receiving ok messages from all of its neighbors, i measures the weighted sum of violated constraints for the current value to its variable (*eval*) and the maximal possible improvement if i changes the current value to another value (*improve*); i then sends both *eval* and *improve* to its neighbors via *improve messages*.

Upon receiving improve messages from all of its neighbors, i changes the current value to its variable to the value that gives the maximum possible improvement and sends ok messages if no neighboring agent has a larger *improve* than i's *improve* (ties are broken in favor of the agent with the smaller ID); on the other hand, i keeps the current value and sends ok messages if at least one agent among its neighbors has a larger *improve*. By this simple mutual exclusion using *improve*, no two neighboring agents change the values to their variables simultaneously, and the total weighted sum of violated constraints over the agents is consistently reduced by any value change as a result.

In the above process, i checks if the following three conditions hold: 1) i's *eval*> 0, 2) i's *improve*$= 0$, and 3) no neighboring agent has a positive *improve*. If these hold, i identifies itself to be in a *quasi-local-minimum* and increases the weight of the constraints violated at that time. This is based on the *breakout strategy* [10] for centralized CSPs.

DB is an incomplete algorithm, i.e., it never finds the fact that a DisCSP instance has no solution and may fail to find a solution even if one solution exists. In many cases, however, DB finds a solution to a satisfiable DisCSP instance very quickly [15]. Furthermore, DB is not memory consumptive because an agent simply requires a memory assignment to record an originally given problem (i.e., a variable with a domain and constraints with weights) and some state variables used in the procedure.

3. MULTIPLE LOCAL VARIABLES DISTRIBUTED BREAKOUT

3.1 Basic Idea

We extend DB so that it can handle DisSAT instances where each agent has multiple local variables and call the resultant algorithm MULTI-DB. The behavior of MULTI-DB is basically similar to that of DB, but in MULTI-DB the following ideas are introduced in order to efficiently solve DisSAT instances with multiple local variables.

Local search by an agent: In DB, an agent just selects the value from its variable's domain that gives the maximal possible improvement in the weighted sum of violated constraints. In MULTI-DB, however, since an agent has multiple local variables, an agent has to search for a combination of values to its variables that gives a possible improvement in the weighted sum of violated constraints. The search space is usually much larger than the domain of one local variable. We therefore adopt a local search method for an agent to find such a combination of values. We use various well-known techniques for the local search method (random flip, sideway move, tabu list) as well.

Mutual exclusion allowing more simultaneous flips: When an agent in DB changes the value to its variable, all of its neighbors are prohibited from changing the values to their variables. For DisSAT with multiple local variables, we can make two neighboring agents flip their variables simultaneously while ensuring that the total weighted sum of violated clauses over the agents is reduced by these flips. In MULTI-DB, we introduce a new mutual exclusion method that enables agents to perform such simultaneous flips even among their neighboring agents.

Restart: In DB, once agents start their search processes with randomly chosen initial values to their variables, they commit to these initial values and never cut off their search processes to restart. In MULTI-DB, we introduce a restart mechanism where agents can restart their search processes with new randomly chosen values if a solution to a DisSAT instance has not been found within the predetermined upper bound.

3.2 Detail

Figure 1-5 describe a set of procedures for MULTI-DB. Below, we give the details of each procedure. We first show the meanings of the *global parameters* and *state variables* used in these procedures. The global parameters used in these procedures are as follows. Note that the values of these parameters are commonly known to all agents.

***Maxtries*:** the maximal number of tries. At each try, an agent (re)starts its search process with randomly chosen initial values to its variables.

***Maxrounds*:** the maximal number of times to perform a series of procedures WAIT_OK and WAIT_IMPROVE at each try. We regard performing one series of WAIT_OK and WAIT_IMPROVE as one unit and call it a *round*.

***Maxflips*:** the maximal number of flips we allow an agent to perform at one call of the procedure SEND_OK or SEND_IMPROVE.

```
    procedure MULTI-DB
(01)  Neighbors_i := number of i's neighbors;
(02)  Tabu_i :=null;
(03)  for t := 1 to Maxtries do
(04)    randomly determine Values_i;
(05)    set the weight of all clauses to 1;
(06)    Newweight := null;
(07)    send ok(i, Values_i, Newweight) to neighbors;
(08)    for r := 1 to Maxrounds do
(09)      WAIT_OK;
(10)      WAIT_IMPROVE;
(11)    end
(12)  end
```

Figure 1: Procedure Multi-DB

***Maxdistance*:** the threshold for the termination counter. When the counter $Termination_i$ of agent i increases to reach the threshold, this indicates that a solution of the DisSAT instance has been found [15]. It is sufficient to set the threshold to the maximal distance d (or more) of a graph where we regard each agent as a node and the existence of an inter-agent clause between any two agents as an edge. If such a maximal distance is not known, we can set the threshold to the number of agents k since $k > d$.

δ: the increment in the weight of a clause by which an agent increases the weight of a violated clause in a *quasi-local-minimum*.

***p*:** the noise parameter used in the local search procedure (see 3.3).

***TL*:** the size of the tabu list used in the local search procedure (see 3.3).

The state variables are given in the following. These are used by each agent (say i) during the search process.

***Values*$_i$:** the values of i's variables at the current round.

***Nextvalues*$_i$:** the possible values of i's variables at the next round.

***Currentview*$_i$:** the values of i's and its neighbors' variables at the current round.

***Nextview*$_i$:** the possible values of i's and its neighbors' variables at the next round.

***Eval*$_i$:** the weighted sum of i's clauses violated under $Currentview_i$.

***Improve*$_i$:** the possible improvement in the weighted sum of i's violated clauses after the local search procedure is performed under $Currentview_i$.

***Improveview*$_i$:** the values of i's and its neighbors' *Improve*.

***Termination*$_i$:** the counter used for detecting the fact that a solution to a DisSAT instance has been found.

***Consistent*$_i$:** a boolean variable that is set false *iff* at least one of i and its neighbors fails to get values to variables such that the weighted sum of violated clauses is 0.

***Neighbors*$_i$:** the size of i's neighbors.

***Tabu*$_i$:** i's tabu list, which records the history of $Values_i$ in the last TL rounds (see 3.3).

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

Figure 2: Procedure Wait_ok

```
    procedure SEND_IMPROVE
(01)  Eval_i := weighted sum of i's violated clauses under Currentview_i;
(02)  N := Eval_i; Nextview_i := Currentview_i;
(03)  if Eval_i = 0 then
(04)    Consistent_i := TRUE;
(05)  else
(06)    Consistent_i := FALSE; Termination_i := 0;
(07)    T := Currentview_i;
(08)    for f = 1 to Maxflips do
(09)      V := i's variable selected by variable selection rule;
(10)      T := T with V's value flipped;
(11)      if i's variable values in T ∉ Tabu_i then
(12)        E := weighted sum of i's violated clauses under T;
(13)        if E < N then
(14)          N := E; Nextview_i := T;
(15)        if E = N then
(16)          update Nextview_i by sideway rule;
(17)        if E =0 then
(18)          break;
(19)    end
(20)  Improve_i := Eval_i − N;
(21)  add Improve_i to Improveview_i;
(22)  Nextvalues_i := i's variable values in Nextview_i;
(23)  Possflips := values in Nextvalues_i different from those in Values_i;
(24)  prohibit any other flips except for Possflips;
(25)  send improve(i, Possflips, Improve_i, Eval_i, Termination_i) to neighbors;
```

Figure 3: Procedure Send_improve

Figure 1 shows the main procedure called MULTI-DB. In this procedure, agent i randomly determines the initial values to its variables (Step 04), sets the initial weight of all clauses to 1 (Step 05), and sends ok messages to its neighbors (Step 07). Then, i repeats WAIT_OK and WAIT_IMPROVE until the predetermined upper bound, *Maxrounds*, is reached (Steps 08–11). All of the above steps are repeated *Maxtries* times at most (Steps 03–12).

Figure 2 describes the WAIT_OK procedure. In this procedure, when agent i receives an ok message from agent j (Step 06), i adds $Values_j$ to $Currentview_i$ (Step 08) and updates the weights of its clauses based on *Newweight* if *Newweight* includes new clause weights (Step 09). Upon receiving ok messages from all of its neighbors, i performs SEND_IMPROVE and breaks the while loop (Steps 10–12). Note that an ok message includes a sender's ID (j), the values to the sender's variables at the current round ($Values_j$), and the new weights of the sender's clauses (*Newweight*). The value of *Newweight* is null if a sender has not updated its weights.

Figure 3 shows the SEND_IMPROVE procedure. In this procedure, agent i first measures the weighted sum of violated clauses under $Currentview_i$, $Eval_i$ (Step 01). If $Eval_i = 0$, i makes $Consistent_i$ true (Step 04), prohibits flips for all of its variables at the current round (Step 24), and sends improve messages to its neighbors (Step 25). On the other hand, if $Eval_i \neq 0$, after making $Consistent_i$ false and $Termination_i$ zero (Step 06), i performs the local search procedure (Steps 07–19). The procedure provides the possible values to i's variables that gives an improvement in the weighted sum of i's violated clauses (Step 22). The possible values may require some flips for i's variables, and hence i identifies these flips as *Possflips* (Step 23) and prohibits other flips for its variables, i.e., flips not required at the current round (Step 24). Then, i sends these possible flips (*Possflips*), the possible improvement in the weighted sum of violated clauses ($Improve_i$), $Eval_i$, and $Termination_i$ to its neighbors via improve messages (Step 25).

```
    procedure WAIT_IMPROVE
(01)  Counter := 0;
(02)  Wait_improve_mode := TRUE;
(03)  while Wait_improve_mode do
(04)    when i received improve(j, Possflips, Improve_j, Eval_j, Termination_j) do
(05)      Counter := Counter + 1;
(06)      Termination_i := min(Termination_i, Termination_j);
(07)      update Nextview_i by Possflips;
(08)      add Improve_j to Improveview_i;
(09)      if Eval_j > 0 then
(10)        Consistent_i := FALSE;
(11)      if Counter = Neighbors_i then
(12)        SEND_OK;
(13)        Wait_improve_mode := FALSE;
(14)    end
(15)  end
```

Figure 4: Procedure Wait_improve

Figure 4 describes the WAIT_IMPROVE procedure. In this procedure, when agent i receives an improve message from agent j, i updates $Termination_i$ (Step 06), $Nextview_i$ (Step 07), and $Improveview_i$ (Step 08), as shown in the figure; and makes $Consistent_i$ false if $Eval_j$ (the weighted sum of j's violated clauses at the current round) is larger than zero (Step 10). Upon receiving improve messages from all of its neighbors, i performs SEND_OK and breaks the while loop (Step 11–13).

Figure 5 shows the SEND_OK procedure. In SEND_OK, if $Consistent_i$ has not been made false, i increases $Termination_i$ by one (Step 03) and sends ok messages to its neighbors (Step 41). As we showed in [15], if $Termination_i$ reaches *Maxdistance* by this increment, this indicates that a solution to a DisSAT instance has been found, and consequently, i sends notification of this fact and can terminate (Steps 04–06).

On the other hand, if $Consistent_i$ has been made false, SEND_OK branches out in one of two directions depending on whether there exists at least one agent that can flip variables among i and its neighbors.

In one direction, where no agent among i and its neighbors can flip its variables (Step 08), i increases the weight of each clause violated under $Currentview_i$ by δ (Step 10); if a clause whose weight is increased is an inter-agent clause then i adds the clause and its new weight into *Newweight* to send to its neighboring agents who share the same clause (Steps 11, 12).

In the other direction, where some agents among i and its neighbors can flip their variables, i proceeds as follows. For each clause not violated at the current round but possibly violated at the next round (Step 15), i identifies the flips (*Culprit_flips*) causing the possible violation at the next round (Step 16) and the agents (*Culprit_ag*) planning to perform those flips (Step 17); and checks if the following three conditions hold (Step 18): 1) i is responsible for the possible violation, 2) the possible violation is caused by at least two agents, and 3) i has the least $Improve_i$ among *Culprit_ag* (ties are broken in favor of the agent with the larger ID). If

```
    procedure SEND_OK
(01)  Newweight := null;
(02)  if Consistent_i then
(03)    Termination_i := Termination_i + 1;
(04)    if Termination_i = Maxdistance then
(05)      broadcast that a solution has been found;
(06)      terminate all procedures;
(07)  else
(08)    if Nextview_i = Currentview_i then
(09)      for each violated clause C under Currentview_i do
(10)        increase the weight of C by δ;
(11)        if C is an inter-agent clause then
(12)          add the pair of C and its new weight to Newweight;
(13)      end
(14)    else
(15)      for each clause not violated under Currentview_i but violated under Nextview_i do
(16)        Culprit_flips := flips causing such violation;
(17)        Culprit_ag := agents planning to perform Culprit_flips;
(18)        if (i ∈ Culprit_ag) ∧ (|Culprit_ag| ≥ 2)
              ∧ (i has the least Improve_i among Culprit_ag) then
(19)          withdraw one of i's flips in Culprit_flips;
(20)      end
(21)      if no possible flip is withdrawn then
(22)        Values_i := Nextvalues_i;
(23)      else
(24)        T := Currentview_i; N := Eval_i; Nextview_i := T;
(25)        for f = 1 to Maxflips do
(26)          V := i's flippable variable selected by variable selection rule;
(27)          T := T with V's value flipped;
(28)          if i's variable values in T ∉ Tabu_i then
(29)            E := weighted sum of i's violated clauses under T;
(30)            if E < N then
(31)              N := E; Nextview_i := T;
(32)            if E = N then
(33)              update Nextview_i by sideway rule;
(34)            if E =0 then
(35)              break;
(36)        end
(37)        Values_i := i's variable values in Nextview_i;
(38)  if |Tabu_i| = TL then
(39)    delete the oldest element in Tabu_i;
(40)  add Values_i to Tabu_i;
(41)  send ok(i, Values_i, Newweight) to neighbors;
```

Figure 5: Procedure Send_ok

the first two conditions were to hold, the clause would be accidentally violated at the next round by simultaneous flips over multiple agents, and the total weighted sum of violated clauses over the agents would consequently increase at the next round. To avoid this, an agent that meets the third condition withdraws one of its flips in *Culprit_flips* (Step 19) (ties are broken randomly), and thereby ensures that the total weighted sum of violated clauses over the agents decreases when the flips are performed.

In the above procedure (Steps 15–20), i sometimes withdraws a part of its possible flips. However, when no possible flip is withdrawn, i can perform all of its possible flips without increasing the total weighted sum of the violated clauses over the agents. In this case, i performs the flips (Step 22) and sends ok messages to its neighbors (Step 41). On the other hand, by withdrawing some flips, can i perform the other possible flips that are not withdrawn without increasing the total weighted sum of violated clauses over the agents? The answer is unfortunately no. All flips among possible flips are usually interdependent. Even if possible flips as a whole can reduce the weighted sum of violated clauses, a part of the possible flips may increase the weighted sum of violated clauses. Therefore, i must perform the local search procedure again (Steps 24–36) using only the flippable variables that are not prohibited (Step 24 in Figure 3) or withdrawn (Step 19). The flips obtained from this "backup" local search procedure obviously do not increase the total weighted sum of violated clauses over the agents, and i can therefore perform these flips immediately (Step 37).

3.3 Local Search Techniques

In MULTI-DB, an agent performs the local search procedure in SEND_IMPROVE (Step 07–19 in Figure 3) and SEND_OK (Step 24–36 in Figure 5). We describe the techniques used in the local search procedure.

Variable selection rule For agent i to select a variable to flip (Step 09 in Figure 3, Step 26 in Figure 5), we use a technique like WalkSAT [11], where i randomly picks up one violated clause; and selects one of i's variables whose flip causes no new clause violations (breaks ties randomly); if such a variable does not exist in the clause, it randomly selects one of i's variables from the clause with probability p; otherwise, it selects one of i's variables whose flip minimizes the weighted sum of newly violated clauses (breaks ties randomly). Note that, in SEND_OK, since agent i can flip only flippable variables (variables whose flips are not prohibited or withdrawn), i randomly picks up one violated clause including flippable variables first and follows the same procedure.

Sideway rule If two sets of values to variables have the same weighted sum of violated clauses, agent i chooses the one with the larger hamming distance from the values at the current round (Step 16 in Figure 3, Step 33 in Figure 5).

Tabu list Agent i maintains a tabu list that records the history of $Values_i$ in the last TL rounds and does not select values to its variables in the tabu list.

3.4 Simple Example

Using the DisSAT instance in Section 2, we provide simple execution examples. As we mentioned before, for CNF formula (1) we assume that agent a_1 is assigned $\{x_1, x_2, x_3\}$ and $\{C_1, C_2, C_3, C_4\}$ while agent a_2 is assigned $\{x_4, x_5, x_6\}$ and $\{C_2, C_3, C_4, C_5\}$.

In the first scenario, both agents initially determine the values to their variables as follows:

$$x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 1, x_6 = 0$$

and send ok messages to each other. Upon receiving an ok message, agent a_1 knows that the weighted sum of violated clauses, C_3, is 1, discovers through the local search procedure that $\{x_1 = 1, x_2 = 1, x_3 = 0\}$ makes the weighted sum of violated clauses zero, and sends a_2 an improve message including $\{x_1 = 1, x_2 = 1\}$ as possible flips. Agent a_2, on the other hand, knows that the weighted sum of violated clauses, C_3 and C_5, is 2, discovers that $\{x_4 = 0, x_5 = 1, x_6 = 0\}$ makes the weighted sum of violated clauses zero, and sends a_1 an improve message including $\{x_4 = 0\}$ as a possible flip. Upon receiving the improve message, agent a_1 finds that its possible flips $\{x_1 = 1, x_2 = 1\}$ never cause accidental clause violations at the next round and performs these flips. Agent a_2, on the other hand, performs its possible flip $x_4 = 0$ in a similar way. Consequently, the values to their variables at the next round are:

$$x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 1, x_6 = 0$$

This is a solution to this DisSAT instance.

In the second scenario, both agents start with:

$$x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = 0$$

Agent a_1 evaluates the weighted sum of violated clauses, C_4, as 1 and realizes that flipping x_3 leads to no violation. Agent a_2 also evaluates the weighted sum of violated clauses, C_4, as 1 and realizes that flipping x_6 leads to no violation. Then, both agents exchange improve messages and discover that C_2 is not currently violated but might be violated at the next round by these flips. In this case, a_1 flips x_3, while a_2 withdraws the flip of x_6 since a_2 meets the three conditions in SEND_OK (Note: although both agent have the same *Improve*, a_2 has a larger ID than a_1). Consequently, the values to their variables at the next round are:

$$x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 0$$

This is another solution to this DisSAT instance.

4. EVALUATION

We evaluated the performance of MULTI-DB through an experiment using the *uniform random 3-SAT* in SATLIB[1]. The *uniform random 3-SAT* is generally considered to be one of the hardest classes of 3-SAT. It has been widely used to evaluate the performance of SAT solvers. A notable feature of this problem is that it includes "unforced and filtered" satisfiable instances. To generate satisfiable instances, it is important to generate instances randomly and filter unsatisfiable instances. A method like forcing an instance to have a particular solution generally produces much easier instances [1]. For the conversion of a SAT instance into a DisSAT instance, we evenly partitioned n variables of a SAT instance among k agents and assigned each clause to all agents having variables in the clause.

We compared MULTI-DB with MULTI-AWC [16]. MULTI-AWC is one of the most efficient algorithms for solving DisCSPs with multiple local variables. MULTI-AWC can employ nogood learning to reduce its communication cost. However, as we mentioned in Section 1, complete nogood learning sometimes demands huge memory resources and even partial nogood learning cannot avoid relatively high memory consumption. This can be problematic especially when agents have to solve a DisCSP instance where each agent is assigned a large and complex local problem while being permitted to use only a limited amount of memory. Accordingly, the MULTI-AWC used in this experiment did not employ nogood learning on the assumption that agents cannot demand extra memory during the search process.

We used a simulator of a *fully synchronous distributed system*. This is a typical distributed system, where all agents (nodes) repeat computation and communication simultaneously using a simulated global clock. On this simulator, we implemented MULTI-DB and MULTI-AWC and measured the following as their communication and computation costs, respectively.

cycles: the number of cycles consumed until the agents find one solution. One cycle consists of a series of simultaneous computation and communication events. In other words, every agent receives all incoming messages, performs local computation, and sends all messages in one cycle. This measure can be considered as the communication cost of an algorithm. Note that one round in MULTI-DB, in which 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 in each cycle until the agents find one solution. More specifically, in each cycle, we identify the *bottleneck agent*, which performed the most flips for its local computation, and sum up all of the maximal numbers of flips over all consumed cycles. Although the amount of computation in each cycle varies among the agents, the total amount of computation is dominated by the bottleneck agents. This measure can be thus considered as the computation cost of an algorithm. Note that for MULTI-AWC, we measured the number of visited search nodes instead of the number of flips.

We set the upper bound of the number of cycles to $500n$ and cut off a run if it exceeds the upper bound in order to finish our experiment in a reasonable amount of time. For a run cut off, we used the data from the cycles and flips at that time.

The global parameters in MULTI-DB were set as follows: *Maxtries*=1, *Maxrounds*= $250n$, *Maxflips*= n/k, $\delta = 1$, $p = 0.3$, $TL = 3$ (when $n = 20, 50, 75$), and $TL = 5$ (when $n = 100, 125, 150$). We did not set *Maxdistance* because in this experiment both MULTI-DB and MULTI-AWC are terminated immediately once all of the agents find a solution to a DisSAT instance.

Table 1 shows the results. In the uniform random 3-SAT in SATLIB, there are 100 instances for each n (1000 instances for $n = 50, 100$). We gave randomly chosen initial values to variables once for each instance of each combination of (n, k) and made each method run. This table indicates the average/median cycles and the average/median flips over 100 (or 1000) runs for each combination of (n, k). It also indicates the ratio of runs that were successfully completed within the upper bound of the number of cycles (success ratio).

As Table 1 indicates, MULTI-DB is better than MULTI-AWC in all cases in terms of both the average and median flips and in all cases for $n \geq 75$ in terms of both the average and median cycles. Moreover, the differences become greater as n increases. We would like to stress that when $n = 150$, MULTI-DB has at least an order of magnitude less average/median cycles and average/median flips at every k. Furthermore, the success ratio of MULTI-DB gets better results in all cases.

Another trend we can observe is that for each n, the average/median cycles (communication cost) increases as k (the number of agents) increases, while the average/median flips (computation cost) decreases as k increases. These are true for both MULTI-DB and MULTI-AWC. The trade-off between the communication and computation costs when we vary the degree of decentralization seems to be independent of algorithm.

Although MULTI-DB has better success ratios than MULTI-AWC in all cases, it unfortunately fails to complete a small number of runs for $n \geq 100$. One of the most simple ways of avoiding such a "heavy-tailed cost distribution" is to introduce some randomization technique including restarts [3]. Table 2 shows results for MULTI-DB with restarts (*Maxtries*=25, *Maxrounds*= $10n$, and the others are unchanged). As this table indicates, some heavy-tailed cost distributions

[1]http://aida.intellektik.informatik.tu-darmstadt.de /SATLIB/benchm.html

Table 1: Performance of Multi-DB and Multi-AWC on uniform random (Dis)3-SAT with n variables by k agents

Method	n	k	Average cycles	Median cycles	Average flips	Median flips	Success ratio
MULTI-DB	20	2	3.52×10^1	2.00×10^1	1.78×10^2	1.00×10^2	1.00
		4	5.60×10^1	3.40×10^1	1.45×10^2	8.70×10^1	1.00
	50	2	2.05×10^2	8.20×10^1	2.81×10^3	1.09×10^3	1.000
		5	2.74×10^2	1.32×10^2	1.52×10^3	7.50×10^2	1.000
		10	3.67×10^2	1.68×10^2	9.62×10^2	4.51×10^2	1.000
	75	3	6.87×10^2	2.32×10^2	9.61×10^3	3.30×10^3	1.00
		5	8.41×10^2	3.52×10^2	7.06×10^3	3.03×10^3	1.00
		15	1.07×10^3	3.92×10^2	2.75×10^3	1.03×10^3	1.00
	100	2	8.68×10^2	3.32×10^2	2.41×10^4	9.25×10^3	1.000
		4	1.53×10^3	4.68×10^2	2.13×10^4	6.86×10^3	0.996
		5	1.92×10^3	5.70×10^2	2.11×10^4	6.67×10^3	0.993
		10	2.78×10^3	1.15×10^3	1.48×10^4	6.44×10^3	0.989
		20	3.58×10^3	1.38×10^3	9.10×10^3	3.58×10^3	0.977
	125	5	3.90×10^3	1.30×10^3	5.34×10^4	1.91×10^4	0.99
		25	8.27×10^3	1.82×10^3	2.08×10^4	4.67×10^3	0.93
	150	3	3.98×10^3	8.28×10^2	1.09×10^5	2.43×10^4	0.98
		5	6.67×10^3	8.58×10^2	1.08×10^5	1.57×10^4	0.96
		10	8.20×10^3	2.40×10^3	6.49×10^4	2.02×10^4	0.97
		15	9.54×10^3	2.44×10^3	4.93×10^4	1.38×10^4	0.96
MULTI-AWC	20	2	3.04×10^1	1.30×10^1	7.48×10^2	3.00×10^2	1.00
		4	3.55×10^1	2.00×10^1	3.55×10^2	1.96×10^2	1.00
	50	2	1.07×10^2	5.50×10^1	1.05×10^4	5.04×10^3	1.000
		5	2.94×10^2	1.33×10^2	7.13×10^3	3.07×10^3	1.000
		10	3.60×10^2	1.44×10^2	3.85×10^3	1.55×10^3	1.000
	75	3	8.79×10^2	2.92×10^2	7.78×10^4	2.50×10^4	1.00
		5	1.80×10^3	5.56×10^2	7.64×10^4	2.31×10^4	0.99
		15	1.60×10^3	6.60×10^2	1.81×10^4	7.68×10^3	1.00
	100	2	1.39×10^3	4.36×10^2	4.52×10^5	1.33×10^5	0.999
		4	4.69×10^3	1.33×10^3	4.12×10^5	1.13×10^5	0.987
		5	6.10×10^3	1.73×10^3	3.87×10^5	1.05×10^5	0.976
		10	7.63×10^3	2.27×10^3	1.99×10^5	5.77×10^4	0.968
		20	8.49×10^3	2.68×10^3	9.84×10^4	3.09×10^4	0.950
	125	5	1.92×10^4	9.29×10^3	1.72×10^6	8.12×10^5	0.87
		25	2.55×10^4	1.58×10^4	3.04×10^5	1.88×10^5	0.80
	150	3	2.43×10^4	1.11×10^4	6.47×10^6	2.90×10^6	0.81
		5	3.71×10^4	2.61×10^4	4.37×10^6	2.90×10^6	0.67
		10	3.94×10^4	3.60×10^4	1.79×10^6	1.63×10^6	0.61
		15	4.23×10^4	4.17×10^4	1.16×10^6	1.15×10^6	0.61

Table 2: Performance of Multi-DB with restarts on uniform random (Dis)3-SAT with n variables by k agents

Method	n	k	Average cycles	Median cycles	Average flips	Median flips	Success ratio
MULTI-DB with restarts	100	2	8.86×10^2	3.46×10^2	2.50×10^4	9.85×10^3	0.999
		4	1.39×10^3	5.10×10^2	2.04×10^4	7.38×10^3	1.000
		5	1.64×10^3	5.70×10^2	1.93×10^4	6.85×10^3	1.000
		10	3.23×10^3	1.15×10^3	1.82×10^4	6.48×10^3	0.996
		20	3.48×10^3	1.39×10^3	9.03×10^3	3.63×10^3	0.997
	125	5	2.54×10^3	8.16×10^2	3.79×10^4	1.22×10^4	1.00
		25	6.30×10^3	2.33×10^3	1.63×10^4	6.00×10^3	1.00
	150	3	2.18×10^3	6.08×10^2	6.42×10^4	1.78×10^4	1.00
		5	3.23×10^3	1.20×10^3	5.84×10^4	2.14×10^4	1.00
		10	9.03×10^3	2.09×10^3	8.04×10^4	1.75×10^4	0.96
		15	9.85×10^3	3.85×10^3	5.56×10^4	2.15×10^5	0.98

are eliminated by the simple restart strategy. However, a few runs still remain to be completed for some combinations of (n, k).

In this experiment, we did not use nogood learning for MULTI-AWC since we assumed that memory resources are so tight that agents cannot demand extra memory during search. If the situation was different, i.e., agents had enough memory to perform nogood learning for MULTI-AWC, we might not get such significant results since the performance of MULTI-AWC can be improved by nogood learning (at the expense of memory to record nogoods and time to check nogoods). This means that the conclusion from our experimental results may be restricted to the situation where memory resources are tight. However, we believe that such a situation is not unusual.

5. CONCLUSIONS

This paper described MULTI-DB for solving DisSAT where an agent has multiple local variables and relevant clauses to the variables. In MULTI-DB, agents repeat the following steps: perform a local search procedure to find possible flips able to reduce the weighted sum of violated clauses; exchange these possible flips; resolve conflicts by withdrawing a part of the possible flips that increase the total weighted sum of violated clauses over the agents; perform the permitted flips; and exchange new values to their variables.

Our experimental results on well-known (Dis)3-SAT benchmarks indicate that for larger problems MULTI-DB has at least an order of magnitude lower average communication and computation costs than MULTI-AWC with no learning. This means that MULTI-DB performs very well even when agents are permitted to use only a limited amount of memory. Through the experiment, we also identified a trade-off between the communication and computation costs of algorithms when we vary the degree of decentralization.

6. REFERENCES

[1] D. Achlioptas, C. Gomes, H. A. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 256–261, 2000.

[2] A. Armstrong and E. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 620–625, 1997.

[3] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 431–437, 1998.

[4] Y. Hamadi. Optimal distributed arc-consistency. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, pages 219–233, 1999.

[5] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *Proceedings of ECAI-98*, pages 219–223, 1998.

[6] K. Hirayama and M. Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 169–177, 2000.

[7] K. Hirayama, M. Yokoo, and K. Sycara. The phase transition in distributed constraint satisfaction problems: First results. In *Proceedings of the International Workshop on Distributed Constraint Satisfaction*, 2000.

[8] H. A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[9] P. J. Modi, H. Jung, M. Tambe, W.-M. Shen, and S. Kulkarni. A dynamic distributed constraint satisfaction approach to resource allocation. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 685–700, 2001.

[10] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 40–45, 1993.

[11] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 337–343, 1994.

[12] M. C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 917–922, 2000.

[13] W. E. Walsh, M. Yokoo, K. Hirayama, and M. P. Wellman. On market-inspired approaches to propositional satisfiability. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1152–1158, 2001.

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

[15] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 401–408, 1996.

[16] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of the Third International Conference on Multi-Agent Systems*, pages 372–379, 1998.

[17] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-agent Systems*, 3(2):189–211, 2000.