Sixth International Workshop on Distributed Constraint Reasoning, 2005.

Reducing Redundant Messages in the Asynchronous Backtracking Algorithm

Hong Jiang and José M. Vidal

Computer Science and Engineering
University of South Carolina
Columbia, SC 29208, USA
jiangh@engr.sc.edu, vidal@sc.edu

Abstract. We show how the Asynchronous Backtracking Algorithm, a well known distributed constraint satisfaction algorithm, produces unnecessary messages. Our new optimized algorithm reduces the number of messages by implementing message management mechanism. Tests show our algorithm significantly reduces the total number of messages sent and drastically reduces the number of cycles used when solving instances of the graph coloring problem.

1 Introduction

The Asynchronous BackTracking (ABT) algorithm is a well known algorithm for solving distributed constraint satisfaction problems. Many problems that arise in multiagent systems can be reduced to a distributed constraint satisfaction problem and this approach has lead to successful multiagent applications. However, our analysis of the ABT algorithm has revealed some room for improvement. Specifically, we have found that ABT sends needless messages. In this paper we present our improved ABT algorithm along with tests that show how our improved version can reduce the number of messages sent to about 67% and the number of cycles to about 25% of those in the original algorithm.

Section 2 starts with the formal description of a constraint satisfaction problem and its distributed variation. Section 3 presents the ABT algorithm and Section 4 gives our analysis of it. Section 5 then presents our improved version of ABT. We show the test results in Section 6, the related work is in Section 7 and our conclusion is given in Section 8.

2 The CSP Problem

In a Constraint Satisfaction Problem (CSP) the goal is to find a consistent assignment of values for a set of variables [8]. Formally, a CSP consists of n variables x_1, x_2, ..., x_n, whose values are taken from finite, discrete domains $D_1, D_2, \ldots, D_n$, respectively, and a set of constraints on their values. A constraint is defined by a predicate. The constraint $p_k(x_{k_1}, x_{k_2}, \ldots, x_{k_j})$ is a predicate that is defined on the Cartesian product $D_{k_1} \times \cdots \times D_{k_j}$. In general, there is no restriction about the form of the predicate. It can be a logical or mathematical formula, or any arbitrary relation defined by a tuple of variable values. We will sometimes also refer to these constraints as nogoods. In CSP the predicate is true if and only if the value assignment of these variables satisfies the constraint. Solving a CSP is equivalent to finding an assignment of values for all variables such that all constraints are satisfied.

2.1 Distributed CSP

A distributed CSP is a CSP in which the variables and constraints are distributed among automated agents [10]. For the agents, it assumes the following communication model:

– Agents communicate by sending messages. An agent can send messages to other agents if and only if the agent knows the addresses of the agents.
– The delay in delivering a message is finite, though random. For the transmission between any pair of agents, messages are received in the order in which they were sent.

In this model, the physical communication network may not be fully connected. In other words, the topology of the physical communication network doesn't play an important role here, it assumes the existence of a reliable underlying communication structure among the agents and does not care about the implementation of the physical communication network.

Every agent owns some variables and it tries to determine their values. However, there exist inter-agent constraints which must be satisfied. Formally, there exist m agents $1,2,\ldots,m$. Each variable x_j belongs to one agent i, which could be represented as $belongs(x_j,i))$. Constraints are also distributed among agents. The fact that an agent l knows a constraint predicate p_k is represented as $known(p_k,l)$.

A Distributed CSP is solved if and only if the following conditions are satisfied: for any i and x_j where $belongs(x_j,i)$, the value of x_j is d_j, and for any l and p_k where $known(p_k,l)$, p_k is true under the assignment $x_j = d_j$.

3 The ABT Algorithm

The ABT algorithm is a distributed, asynchronous version of a backtracking algorithm used in solving CSP. The basic idea for backtracking algorithm is to construct a partial solution first, which is a value assignment to a subset of variables that satisfies all of the constraints within the subset, and then to expand the partial solution by adding new variables, until it becomes a complete solution. In the asynchronous backtracking algorithm, agents act asynchronously and concurrently based on their local knowledge without any global control, while the completeness of the algorithm is guaranteed. That is, if there is a solution they will find it. On the other hand, the priority order of agents is determined and each agent tries to find a value satisfying the constraints with the variables of higher priority agents. When an agent sets a variable value, the agent is strongly committed to the selected value, i.e., the selected value will not be changed unless an exhaustive search is performed by lower priority agents. Among the search, the backtracking is achieved by checking the consistency of the variable, if not consistent, then using hyper-resolution rule to generate new nogoods and communicating them to higher priority variables. Therefore, in large-scale problems, a single mistake in the selection of values becomes fatal since such an exhaustive search can be virtually impossible for large m. This drawback is common to all backtracking algorithms.

```
HANDLE-OK-MSG(“ok?”, $(x_j, d_j)$)
1  add $(x_j, d_j)$ to local-view
2  CHECK-LOCAL-VIEW
```

```
HANDLE-NOGOOD-MSG(“nogood”, $x_j$, nogood)
1  record nogood as a new constraint
2  for each agent $x_k$ in nogood that is not its neighbor
3        do request $x_k$ to add $x_i$ as a neighbor
4           add $x_k$ to its neighbors
5           add $(x_k, d_k)$ to local-view
6  old-value $\leftarrow$ current-value
7  CHECK-LOCAL-VIEW
8  if old-value $=$ current-value
9     then send (“ok?”, $(x_i$, current-value$)$) to $x_j$
```

```
CHECK-LOCAL-VIEW
1  if local-view and current-value are not consistent
2     then if no value in $D_i$ is consistent with local-view
3              then BACKTRACK
4              else  select $d \in D_i$ where local-view and $d$ are consistent
5                    current-value $\leftarrow d$
6                    send (“ok?”, $(x_i, d)$) to neighbors
```

```
BACKTRACK
1  nogoods $\leftarrow \{V | V =$ inconsistent subset of local-view by using
                    hyper-resolution-rule$\}$
2  if an empty set is an element of nogoods
3     then broadcast to other agents that there is no solution
4           terminate this algorithm
5  for each $V \in$ nogoods
6        do select $(x_j, d_j)$ where $x_j$ has the lowest priority in $V$
7           send (“nogood”, $x_i, V$) to $x_j$
8           remove $(x_j, d_j)$ from local-view
9  CHECK-LOCAL-VIEW
```

Fig. 1. ABT algorithm

The original Asynchronous Backtracking Algorithm is described as in [9,10,11]. There are slight differences between the different papers. Figure 1 shows a combined version whose correctness we tested.

4 ABT Algorithm Analysis

4.1 Link between non-neighbor agents

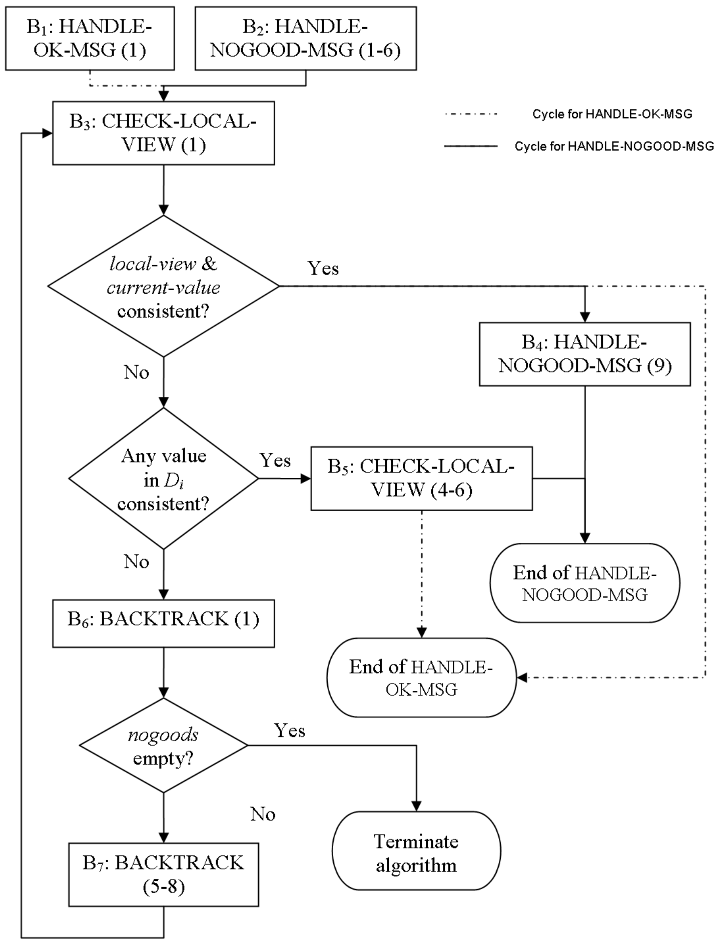


Fig. 2. Flowchart of original asynchronous back tracking algorithm

We believe that lines 3 and 4 in the procedure HANDLE-NOGOOD-MSG are not necessary. That is, it is not necessary to add a link between non-neighbor agents. These additional links increase the number of messages sent. We can imagine that in some extreme case these links might keep increasing until there is a link between every pair of agents. In this case, sending one message amounts to a broadcast. Thus the algorithm would be greatly slowed down. We now explain why lines 3 and 4 are not necessary.

Figure 2 shows a flowchart that represents the handling of an *ok* message (dotted line) and the handling of a *nogood* message (solid line). The numbers in parenthesis in the process blocks correspond to the code line number of their procedure.

By following the arrows, a cycle of handling an *ok* message goes from $B_1 \to B_3 \to$ End of the cycle, $B_1 \to B_3 \to B_5 \to$ End of the cycle, $B_1 \to B_3 \to B_6 \to$ Terminate with failure, or $B_1 \to B_3 \to B_6 \to B_7 \to B_3$ which has a loop. For a cycle of handling a *nogood* message, it goes from $B_2 \to B_3 \to B_4 \to$ End of the cycle, $B_2 \to B_3 \to B_5 \to$ End of the cycle, $B_2 \to B_3 \to B_6 \to$ Terminate with failure, or $B_2 \to B_3 \to B_6 \to B_7 \to B_3$ with a loop. The above process sequences depend on different conditions.

We now check the data flow of the variable *neighbors* to see in what kind of circumstance the non-neighbor agent is added to *neighbors* and how it is used. The variable *neighbors* appears in the algorithm twice: one is at line 3 and 4 of procedure HANDLE-NOGOOD-MSG for adding neighbor and the other is at line 6 of procedure CHECK-LOCAL-VIEW for using *neighbors*.

First, let us consider when will the non-neighbor agent be added as a neighbor. The codes are in procedure HANDLE-NOGOOD-MSG, thus there should be at least one *nogood* message to handle. On the other hand, in the beginning state, we only have *ok* messages. So, we start from checking how HANDLE-OK-MSG could produce a *nogood* message. From the above analysis we know that there are 4 kind of process sequences for the cycle of handling an *ok* message, and the first 3 finish without involving sending out any *nogood* messages, therefore the only possible process sequence is $B_1 \to B_3 \to B_6 \to B_7 \to B_3$.

Assume there is a *nogood* message from x_i to x_j with information of a non-neighbor agent x_k to x_j. According to line 8 in procedure BACKTRACK (B_7), we know that there is no information about x_j in the local view of x_i. So far, we have that B_2, x_k and x_j become neighbors of each other, and the information of x_k is added to the local view of x_j.

Let us consider the 4 possible process sequences for a *nogood* message handling cycle:

1. For $B_2 \to B_3 \to B_4 \to$ End of the cycle, the variable *neighbors* is not used;
2. For $B_2 \to B_3 \to B_5 \to$ End of the cycle, the variable *neighbors* is used, and the new value of x_j is sent to x_k. Obviously, this *ok* message will bring the information of x_j to the local view of x_k. Then, let us check if this information valuable to x_k. We know that the first usage of local view is to check if it is consistent with the current value, and this check is based on the constraints related to the agent. Originally, there are no constraints between non-neighbor agents. However, once an agent handles a *nogood* message, non-neighbor agents may become a constraint for the agent. In the given case, x_j and x_k are originally not neighbors, and by handling *nogood* message, x_j adds x_k to its new constraint, but so far, x_j is not x_k's constraint yet. So, when x_k checks the consistence, the information of x_j in the local view is useless. In other words, x_j don't have to send its information to the new added neighbor x_k.

3. For $B_2 \rightarrow B_3 \rightarrow B_6 \rightarrow$ Terminate with failure, the variable *neighbors* is not used, and the whole algorithm is terminated by reporting no solution.
4. For $B_2 \rightarrow B_3 \rightarrow B_6 \rightarrow B_7 \rightarrow B_3$, the variable *neighbors* is not used in this single sequence, but because there involves a loop, it has potential possibility to be used. Let us analyze this potential possibility. We can start from checking B_7. For x_j there are new constraints involving x_k to be added and the local view of x_j is updated according to the handled *nogood* message. By using hyper-resolution rule, there must be at least one nogood in the *nogoods* set involving x_k. Disregard those nogoods without x_k, let's focus on the nogood with x_k. If x_k has the lowest priority in the nogood, then x_j will send this nogood to x_k, and remove x_k's information from its local view. Based on the hyper-resolution rule, this nogood to x_k has no information about x_j, that is, x_j will not become x_k's new constraint. At the same time, x_k's information loses its worth to x_j since x_k is already deleted from x_j's local view. If x_k is not the one with the lowest priority in the nogood then we could ignore analyzing the code at line 7 and 8 in B_7, because nothing related to the relationship between x_j and x_k. For checking local view B_3 again, we can follow above analysis about the 4 possible process sequences.

Based on above analysis we conclude that the original algorithm does produce unnecessary messages, and that we can delete line 3 and 4 in the procedure HANDLE-NOGOOD-MSG in order to simplify the algorithm without affecting the algorithm's correctness.

4.2 Local View Updating

In line 5 of procedure HANDLE-NOGOOD-MSG, we can see that only the information for non-neighbor agent is recorded to the local-view and the information from other nodes is ignored. However, based on the communication model assumed in this algorithm, the more recently received message carries new updated information of the agent. That is, when handling nogood messages the above algorithm doesn't update the information of the neighbors in the local view, it always uses the old information for the neighbors and new information for the non-neighbors. Obviously, we should use new information since it reflects the current agents' values more closely. This could also influence the efficiency of the implementation of the algorithm. In our algorithm, we correct this by updating all agents information in the nogood to local-view.

4.3 Ways to Improve Efficiency

There are many issues which can be considered in order to improve efficiency for a specific case, for example, the order of selecting variables and values. In the algorithm, the order of selecting agents or variables is determined by the priority, and the priority is determined in advance. Since a variable's value will not be changed unless an exhaustive search is performed by lower priority agents, the priority should be determined very carefully, otherwise the exhaustive search might need a long time. However, in different cases, the method to determine the priority might be different. Therefore, we just focus on how to improve efficiency for the general case.

For the general case, the first thing we could do is to eliminate the unnecessary messages and update local view in time. Just as we discussed in Section 4.1 and 4.2, this keeps the message queue slim and updated.

Another method is the addition of a message management mechanism for the message queue. The basic idea is still to maintain the message queue slim. Specifically, the mechanism tries to keep the message queue slim, update local view in time, and reduce times to call procedure CHECK-LOCAL-VIEW. In our earlier tests with the ABT algorithm we found that it spends most of the time in checking local view. If we could reduce the number of calls to this procedure then the efficiency of the implementation should be also improved. The next section shows how we implemented our message management mechanism.

5 Message Management ABT

Based on the above analysis, we already have a general idea about the message management mechanism. In this section, we will show exactly how it works with our Message Management ABT (MMABT).

As we have shown, the goal for this message management mechanism is to keep the message queue slim, update local view in time, and reduce times to call procedure CHECK-LOCAL-VIEW. Practically, we remove the unnecessary messages as analyzed in above section and we update local view not only for non-neighbor nodes but also for neighbor nodes, by these we keep the local view more updated. On the other hand, by handling several messages together we reduce times CHECK-LOCAL-VIEW is called and keep *msg-queue* from growing too fast.

It is possible to either handle each message as it arrives or to handle groups of messages in one step, that is, handle more than one message and then call BACKTRACK. We add a handling size to limit the number of messages collected and handled together. On the other hand, if the message queue has messages less than the handling size, we just collect those messages from the message queue and handle them. By this way, we don't have to wait until we get full size of the messages. When the handling size is set to 1 this algorithm is equivalent to the original algorithm which handles messages one by one as they arrive.

Another strategy we use for keeping the message queue slim is to remove redundant messages in *msg-queue*. For example, assume that we have message queue $\{\ (ok?,(x_1,1))$ $(ok?,(x_3,2))\ (ok?,(x_1,2))\ \}$ where messages are shown in arrival order. In this case the first message is redundant to the third one. Since we believe the later message always carries agent's information more close to current status, we could simply delete the redundant messages in the message queue to avoid handling the out of date information.

Another technique to keep message queue slim involves the details about how to use hyper-resolution rule. However, the usage of the hyper-resolution rule in the ABT is not clearly described in the related papers. The technique we use is to minimize constrains set and generate nogood based on the local view.

Our MMABT Algorithm is shown in Figure 3.

The procedures CHECK-LOCAL-VIEW and BACKTRACK are kept the same with the original algorithm, and are ignored here.

```
MSG-MANAGE($msg\text{-}queue, handling\text{-}size$)
counter ← 0
while counter < handling-size
      do if msg-queue is empty
            then counter ← handling-size
            else  retrieve one message from msg-queue
                  UPDATE-LOCAL-VIEW(message)
                  counter ← counter + 1
HANDLE-MSGS
```

```
UPDATE-LOCAL-VIEW(message)
if message is (“ok?”, $(x_j, d_j)$)
    then add $(x_j, d_j)$ to local-view
if message is (“nogood”, $x_j$, nogood)
    then add $x_j$ to nogood-senders
          record nogood as a new constraint
          For each agent $x_k$ in nogood
               do add $(x_k, d_k)$ to local-view
```

```
HANDLE-MSGS
old-value ← current-value
CHECK-LOCAL-VIEW
if old-value = current-value
    then send(“ok?”, $(x_i, current\text{-}value)$) to nogood-senders
reset nogood-senders to be empty
```

Fig. 3. Message Management ABT algorithm.

5.1 Example

To show how our algorithm works we use a small example shown in Figure 4.

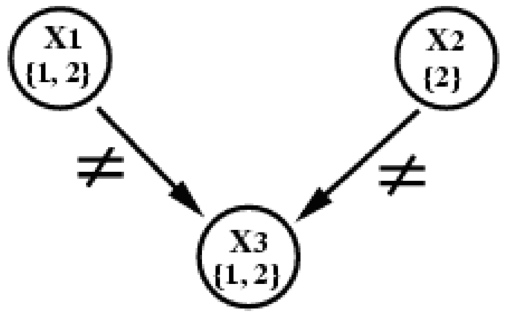


Fig. 4. Example

The constraint in this case is that the neighbor nodes couldn't be the same value, which can be represented as in Table 1. In this table the constraints are preceded by a number which indicated the time at which they were added.

x_1	x_2	x_3
$0:(x_1=1\vee x_1=2)$	$0:(x_2=2)$	$0:(x_3=1\vee x_3=2)$
$0:\neg(x_1=1\wedge x_3=1)$	$0:\neg(x_2=2\wedge x_3=2)$	$0:\neg(x_1=1\wedge x_3=1)$
$0:\neg(x_1=2\wedge x_2=2)$	$1:\neg(x_1=1\wedge x_2=2)$	$0:\neg(x_1=2\wedge x_3=2)$
$2:\neg(x_1=1)$		$0:\neg(x_2=2\wedge x_3=2)$

Table 1. Constraints for example.

We assume that the priority of the nodes here is decreasing with the alphabet order of the nodes id. That is, x_1 has the highest priority and x_3 has the lowest. We assume that the original value for x_1 is 1, x_2 is 2 and x_3 is 2. The initial local view for all nodes is empty, but of course, they know their own current value.

At first, the nodes with higher priority send *ok*? messages to their lower priority neighbor nodes. Thus, x_1 sends $(ok?,(x_1,1))$ to x_3, and x_2 sends $(ok?,(x_2,2))$ to x_3. When x_3 accepts those messages it handles them together and updates its local view to $\{(x_1,1),(x_2,2)\}$. Then x_3 checks its local view. It finds that the current value is not consistent with the current local view (constraint $\neg(x_2=2\wedge x_3=2)$ is not satisfied) so it checks if there is some other value in the domain that is consistent. However, if x_3 equals 1, then constraint $\neg(x_1=1\wedge x_3=1)$ is not satisfied. So, there is no value in the domain which could make it consistent. Therefore, x_3 has to backtrack. In order to generate nogoods set, x_3 first filters and groups the inconsistent constraints which are $\{\neg(x_1=1\wedge x_3=1),\neg(x_2=2\wedge x_3=2)\}$. By using the hyper-resolution rule, we get the nogoods set $\{\neg(x_1=1\wedge x_2=2)\}$ which has only one nogood. For this nogood, x_3 sends that to x_2, the lowest priority node in the nogood.

After that, x_3 deletes x_2's information from the local view, and changes it to $\{(x_1,1)\}$. Then, x_3 has to check local view again. Finally, because $x_1 = 1$ and $x_3 = 2$, which is consistent, x_3 stops working. For x_2, once it accepts the nogood message from x_3, it updates the local view to $\{(x_1,1)\}$, and also adds the constraint to its constraints set as shown in table 1. It checks the local view, and finds an inconsistency. Since x_2 could be 2 only, it has to backtrack. By using hyper-resolution value, it gets nogoods set $\{\neg(x_1 = 1)\}$, and sends the nogood to x_1. Also x_2 removes x_1's information from the local view. Once x_1 gets the nogood message from x_2, it updates the constraints set. Since $\{x_1 = 1\}$ is a subset of $\{x_1 = 1, x_3 = 1\}$ we can simply delete $\neg(x_1 = 1 \wedge x_3 = 1)$ and add $\neg(x_1 = 1)$, since the later is the sufficient condition for the former. By checking local view we determine that the current value $x_1 = 1$ is not consistent, while we could change the value to 2 to make it consistent (note that we don't have x_3's information in local view yet) and send $(ok?,(x_1,2))$ to x_3. x_3 update local view from $\{(x_1,1)\}$ to $\{(x_1,2)\}$, and then check local view again, this time, current value 2 is not consistent, so that we can change the value to 1 to make it consistent. Note that we don't have x_2's information in the local view here, since the message is sent by x_2.

So far, we reach the consistent status for all 3 nodes. We could report success now. In some other case, if we use hyper-resolution rule and get an empty set we could simply report failure and halt.

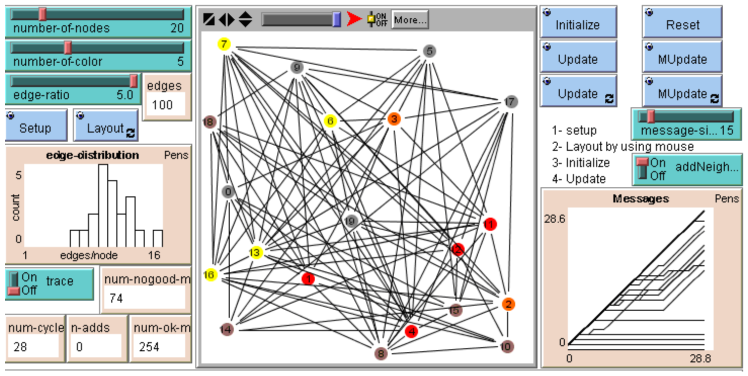
5.2 MMABT Analysis

The soundness and completeness for the original asynchronous backtracking algorithm is already described as in [10]. For our MMABT algorithm, we just need to show that this algorithm is actually using same principle and achieves the same result.

Based on the analysis in Section 4, it is clear that the changes to the original algorithm are removing unnecessary messages and making the local view of each agent up to date, which didn't change the searching method in the algorithm. In other words, the new algorithm is still a kind of depth-first search algorithm. For the worst case, this algorithm still has to do the exhaustive search and there is no much difference between the original algorithm and the new one. On the other hand, we add a message management mechanism and simplify the using of the hyper-resolution rule, which are used to improve the efficiency in general. Our changes do not change the algorithm's soundness and completeness.

6 Test Results

Without losing generality, we apply MMABT to graph coloring problem. The algorithm was implemented using NetLogo [7], a screenshot is shown in Figure 5. In our tests we generate graphs and then make both algorithms work on the same graph with same initial colors. In general, our results show that both algorithms achieve the correct result and the


number-of-nodes 20
number-of-color 5
edge-ratio 5.0
edges 100
num-nogood-m 74
num-cycle 28
n-adds 0
num-ok-m 254
message-si... 15

1- setup
2- Layout by using mouse
3- Initialize
4- Update

Fig. 5. NetLogo implementation of ABT and MMABT.

only difference is that our algorithm works faster. Details on MMABT's performance are presented in this section.

We evaluate the efficiency of the algorithms by comparing their performance on the graph coloring problem. We test both algorithms on identical graphs: with the same vertexes, nodes, and initial colors.

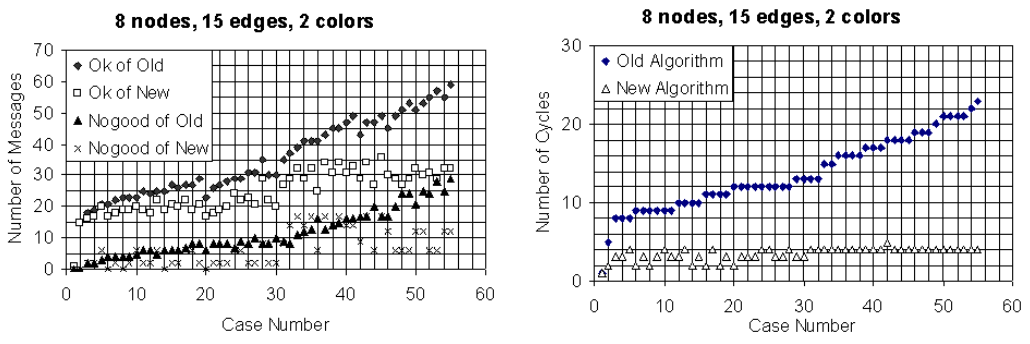


Fig. 6. Comparison of the number of messages and the number of cycles.

Our first set of tests simply verified the correctness of MMABT. We have run both algorithms on over 1,000 randomly generated graphs and in all cases our algorithm finds the same solution as ABT.

We then tested them on one randomly generated graph with 8 nodes, 15 edges, and 2 colors. The graph has a coloring that does not violate any constraints. We ran both algorithms on this graph for 100 times, each time starting with a different randomly generated initial coloring. Figure 6, on the left, shows the number of *ok*? and *nogood* messages sent by both algorithms for all runs. The runs are sorted by the time they took to finish. We note that

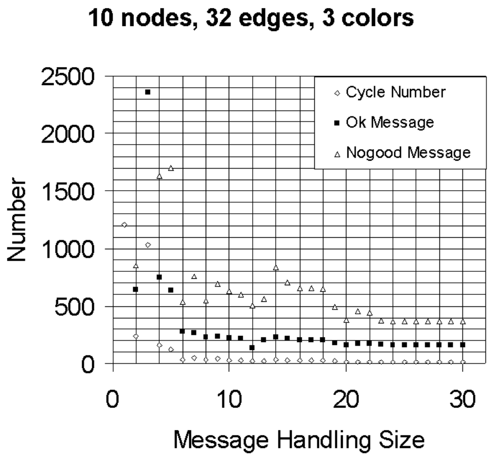
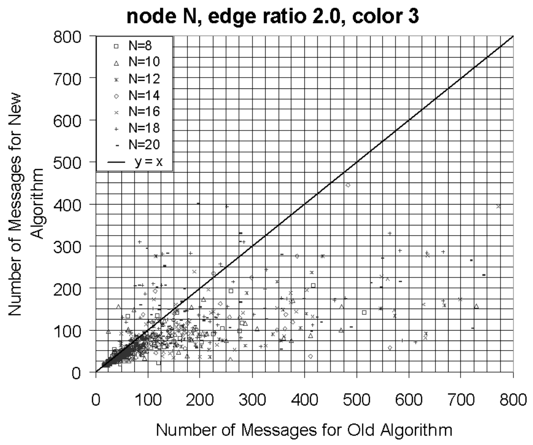


Fig. 7. Comparison of message handling size

in every instance MMABT sends fewer *ok*? messages than ABT, and that in over 80% of the cases MMABT sends fewer *nogood* messages than ABT. Figure 6, on the right, shows the number of execution cycles for both algorithms. We define a "cycle" to be the number of CHECK-LOCAL-VIEW calls made by the algorithm since that is by far the most compute-intensive procedure. Note that the number of cycles for MMABT is always smaller than for ABT. An analysis of these results lets us determine that MMABT sends 67% of the messages sent by ABT and MMABT uses 25% of the cycles used by ABT.

However, note that this is the ideal case which assumes that all available messages are handled at each time step. In practice, an implementation must set a limit on the number of messages it handles at each time step, we call this limit the message handling size. Figure 7 shows that the number of the cycles, the number of *ok*? messages, and the number of *nogood* messages are reduced as message handling size increased from 1 to 30. The results are for a graph of 10 nodes, 32 edges, and 3 colors. This reduction is due to the fact that as the message handling size is increases the local view is kept more up to date. However, in some cases the cycles may increase when the handling size increases, which is a result of the difference between the local view with the current values. In other words, though the local view of the nodes reflects the current values more closely when message handling size increases, there is still some difference between the local view and the current values which makes the search order a little difference. Sometimes this difference will result a worse search but, on average, the number of messages does decrease.

We also tested our algorithm on randomly generated graphs with varying number of nodes, 3 colors, and an edge ratio of 2. Specifically, we let the number of nodes N be 8, 10, 12, 14, 16, 18, and 20 and for each value of N we generated 100 random graphs with N nodes and $2N$ edges. We then compared the performance of both algorithms on each graph. Figure 8 shows the total number of messages (left) and total cycles (right) used by



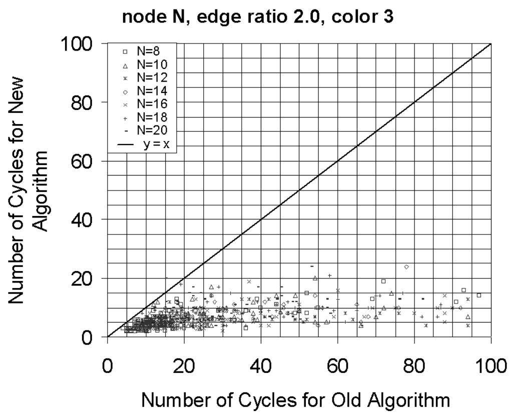


Fig. 8. Number of cycles (left) and messages (right) by ABT and MMABT for 7 sets of 100 randomly generated graphs. *N* is the number of nodes.

N	Num. cases reduce msg by MMABT / Num. total cases	Num. msgs. sent by MMABT / Num. msgs. sent by ABT
8	.90	.65
10	.89	.56
12	.92	.52
14	.83	.61
16	.92	.53
18	.80	.61
20	.81	.67

Table 2. Performance Evaluation on Message Reduced by MMABT.

both algorithms for all these tests. We can see that MMABT sends fewer messages and uses less cycles for most cases. Table 2 gives the fraction of the number of total cases that is with messages reduced by MMABT, and the fraction of messages from ABT that MMABT sends. If we average over all these number we find that around 87% of the random cases could have messages reduced by MMABT and MMABT sends around 60% of the messages sent by ABT, which include the worse cases.

MMABT is able to reduce the number of messages via its message management mechanism which keeps the agents' local view more closely synchronized to reality. Unfortunately, in some cases this message management leads to changes in the search order which might result in the algorithm going down a path in the search space that is a dead end. In these cases MMABT performs worse than ABT. Luckily, our tests show that these cases are rare for randomly generated graphs, and even in those worse cases, we still manage to reduce the number of messages sent.

MMABT is able to reduce the number of cycles used because it handles all available messages at a time instead of only one at a time, as ABT does. However, if there are a lot of messages in the queue it might be impossible to handle all of them in one step. As such, we also compared a variation of MMABT where the algorithms uses a fixed *message handling size* which we define as the number of messages that MMABT handles at each time step. That is, a message handling size of 1 means that we handle only one message each time, the same as ABT. Above all, we could see that by removing unnecessary messages we can reduce message numbers in some case, but the improvement is not very obvious; By using message management mechanism, it reduces both circles and messages number greatly. It works great for average cases. In the MMABT algorithm, in general case, the cycle number, ok message number and nogood message number will decrease with increasing of the message handling size.

7 Related Work

These are many different versions of ABT along with extensions to original ABT. For example, in [3] the authors propose a backtracking algorithm which makes use of some of the good properties of centralized dynamic backtracking, and [13] gives an asynchronous version of dynamic backtracking. [5] applies ABT to a distributed constraint network and present a generic distributed method for computing any variable ordering heuristic. In [6] the authors show how an algorithm for maintaining consistency during distributed asynchronous search can be designed by expanding on ABT. In [2,1] they propose an asynchronous backtracking algorithm ABT_{not} which does not need to add links between initially unconnected agents. We found no paper that notices the redundancy problem in the original ABT. The algorithms in [2,1] come close—the authors notice that an ABT algorithm without adding links between initially unconnected agents will also works correctly. However, they did not mention that the links were unnecessary and did not give a proof of this. According to their test result and our analysis, the messages reduced by the algorithm without adding links are not obvious. Our paper points out the unnecessariness and gives a proof.

Rather than simply remove the redundant links, we also improve the efficiency of the original ABT by proposing a new algorithm, MMABT, with a message management mechanism. It involves some idea of processing messages by packets, the benefit of which is shown in [4,12]. Differently, instead of reading all messages, we set *handling-size* to control the size of the packets, which limits the number of messages collected and handled together. By this way, we avoid that some agents keep working hard in handling a lot of messages while some other keep idle. Meanwhile, the *handling-size* is an upper bound, such that in a time period, if some agents do not have messages collected to be full size, they do not have to wait until they get the full size, which avoid that some agents keep waiting too long to get enough messages to handle.

8 Conclusion

We analyzed the original asynchronous backtracking algorithm and determined that it produces unnecessary messages. We showed an optimized asynchronous backtracking algorithm, MMABT, which achieves more efficiency by incorporating an extra message management mechanism to remove unnecessary messages, keep the message queue updated, handling several messages together, and controlling the number of messages to be handled by a handling-size. Our tests show that these changes improve the efficiency of the original ABT algorithm greatly. On a set of randomly generated graphs, the MMABT algorithm uses around 20% of the number of cycles and 60% of the number of messages used by ABT. Since our proposed changes are simple to implement and the gains are significant, we consider MMABT is a significant improvement on the standard ABT.

References

1. Christian Bessiére, Ismel Brito ans Arnold Maestre, and Pedro Meseguer. The asynchronous backtracking family. *Technical Report 03139*, 2003.
2. Christian Bessiére, Arnold Maestre, Ismel Brito, and Pedro Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence*, 161:7–24, 2005.
3. Christian Bessiére, Arnold Maestre, and Pedro Meseguer. Distributed dynamic backtracking. *Notes of the IJCAI'01 Workshop on Distributed Constraint Reasoning, Seattle, WA*, pages 9–16, 2001.
4. Ismel Brito and Pedro Meseguer. Synchronous, asynchronous and hybrid algorithms for DisCSP. In Mark Wallace, editor, *Principles and Practice of Constraint Programming*. 2004.
5. Youssef Hamadi, Christian Bessiére, and Joël Quinqueton. Backtracking in distributed constraint networks. In *Proceedings of the European Conference on Artificial Intelligence*, pages 219–223, Brighton, UK, 1998.
6. Marius-Călin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Consistency maintenance for ABT. In *Proceedings 7th National Conference on Principles and Practicd of Constraint Programming, CP'01, Paphos, Cyprus*, pages 271–285, 2001.
7. Uri Wilensky. NetLogo: Center for connected learning and computer-based modeling, Northwestern University. Evanston, IL, 1999. `http://ccl.northwestern.edu/netlogo/`.
8. Michael Wooldridge. *Introduction to MultiAgent Systems*. John Wiley and Sons, 2002.
9. Makoto Yakoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. *12th IEEE International Conference on Distributed Computing Systems '92*, pages 614–621, 1992.
10. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.
11. Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.
12. Roie Zivan and Amnon Meisels. Synchronous vs asynchronous search on DisCSPs. In *Proceedings of the First European Workshop on Multi-Agent Systems (EUMA)*, 2003.
13. Roie Zivan and Amnon Meisels. Concurrent dynamic backtracking for distributed CSPs. In *Proceedings Constraint Programming*, pages 782–787, 2004.