A Distributed Cooperative Search Algorithm using Multiple Contexts and Pruning

Toshihiro Matsui and Hiroshi Matsuo
Nagoya Institute of Technology
Gokiso-cho, Showa-ku, Nagoya, Aichi, 466-8555, Japan
{matsui.t, matsuo}@nitech.ac.jp

Abstract
Distributed constraint optimization problems (DCOPs) have been studied as a fundamental framework of cooperative problem solving in multiagent systems. Exact solvers based on tree search and dynamic programming have been proposed for DCOPs. Based on tree searches, the solvers perform iterative processing that depends on message communication. Therefore the overhead of the communication affects the execution time of the search. On the other hand, solvers that are completely based on dynamic programming require no iterative processing among agents. However, its memory and message size is exponential to the induced width of pseudo-trees. Although memory-bounded solvers that employ both methods have been proposed, several solvers search for one solution at a time [3]. Other solvers employ no pruning based on global cost values [5, 6]. Especially, when the communication overhead is relatively large, simultaneously expanding multiple solutions and transferring them by the same message are reasonable. In this study, we present the basic algorithms of memory-bounded solvers that employ multiple search points.

1 Introduction

Distributed constraint optimization problems (DCOPs) [2, 3, 4, 10] have been studied as a basic framework of cooperative problem solving in multiagent systems. With DCOPs, the states of agents and the relationships between agents are formalized into a constraint optimization problem. The problem is solved by distributed search algorithms. These studies focus on the optimization problems and the distributed search algorithms that are essentially contained in cooperative protocols of the multiagent systems. Several cooperative problems including distributed resource scheduling and sensor networks are represented as DCOPs [1, 10]. Exact solvers based on tree search and dynamic programming have been proposed for DCOPs. Based on tree searches, the solvers perform iterative processing that depends on message communication. Therefore the overhead of the communication affects the execution time of the search.

On the other hand, solvers that are completely based on dynamic programming [4] require no iterative processing among agents. However, memory and message size is exponential to the induced width of the pseudo-trees. Although memory-bounded solvers that employ both methods have been proposed, several solvers search for one solution at a time [3]. Other solvers employ no pruning based on global cost values [5, 6]. Especially, when the communication overhead is relatively large, simultaneously expanding multiple solutions and transferring them by the same message are reasonable. In this study, we present the basic algorithms of memory-bounded solvers that employ multiple search points. Such methods resemble the execution of overlapped multiple search processes. Basically, the proposed methods are generalizations of previous studies. In search processing, the solver recomposes the solution sets that are currently expanded. The solution spaces of the sub-problems are considered to recompose a set of solutions. For the set of solutions, boundaries of the global cost values are simultaneously computed. By the multiple search points and the pruning based on the boundaries reduce iterative search among agents. The algorithm can also be considered as the basis of solvers that employ a mixed search strategy.

The rest of our paper is organized as follows. In Section 2, the background of the study is shown. Our proposed methods are shown in Section 3, 4 and 5. The methods are experimentally evaluated in Section 6. In Section 7, we conclude our study.

2 Backgrounds

2.1 Distributed constraint optimization problem

In this study, we use a fundamental definition of distributed constraint optimization problems. A problem is defined by set $A$ of agents, set $X$ of variables, set $C$ of binary constraints, and set $F$ of binary functions. Agent $i$ has its own variable $x_i$ that takes a value from discrete finite domain $D_i$. The value of $x_i$ is controlled by agent $i$. Constraint $c_{i,j}$ represents the relationship between $x_i$ and $x_j$. The cost of assignment $\{(x_i, d_i), (x_j, d_j)\}$ is defined by binary function
2.2 Pseudo-trees

A pseudo-tree [4, 7], which is a graph structure that defines a partial order on variables, is based on a spanning tree of the constraint network. A typical pseudo-tree is generated using a depth-first traversal of a constraint network. For example, the pseudo-tree in Figure 1 (b) is generated from the constraint network in Figure 1 (a). In the pseudo-tree, the edges of the original constraint network are categorized into either tree edges or back edges. The tree edges are the edges of the spanning tree. The other edges are back edges. The tree edges represent the partial order relation between the two variables. We consider the tree edges of the pseudo-tree edges of the corresponding spanning tree. Also, vertices, variables, and agents may not be strictly distinguished. The following notations are used:

prnt\(_i\): parent variable of \(x\).

Chld\(_i\): set of child variables of \(x\).

Nbr\(_i\): partial set of ancestor variables of \(x\). The variables in Nbr\(_i\) are related to \(x\) by constraints. The variables are called pseudo-parents.

Ast\(_i\): partial set of ancestor variables of \(x\). Let \(x_k\) denote a variable in Ast\(_i\). For at least one variable \(x_j\) that is contained in the pseudo-tree rooted at \(x_i\), \(x_k\) has relationship \(x_k \in Nbr_j\).

No back edge exists between different sub-trees. By employing this property, search processing can be performed in parallel.

2.3 Computation of cost values

We outline cost computation using pseudo-trees [3, 4]. Below, we assume that agents have already received both variables’ values and cost values from other agents. Agent \(i\)’s computation is based on partial solution \(s_i\) of Ast\(_i\). \(s_i\) is called context.

Local cost \(\delta_i(s_i \cup \{(x_i, d)\})\) for context \(s_i\) and value \(d\) of variable \(x_i\) is defined as follows.

\[
\delta_i(s_i \cup \{(x_i, d)\}) = \sum_{(x_j, d_j) \in s_i, j \in Nbr_i} f_{i,j}(d, d_j) \tag{1}
\]

Figure 1: Pseudo-tree and messages

\(f_{i,j}(d_i, d_j) : D_i \times D_j \rightarrow \mathbb{N}_0\). The goal is to find global optimal solution \(A\) that minimizes the global cost function:

\[
\sum_{f_{i,j} \in F, \{(x_i, d_i), (x_j, d_j)\} \subseteq A} f_{i,j}(d_i, d_j).
\]

Agent \(i\) knows the constraints and the cost functions that are related to \(x_i\). The search process to find the optimal solution is represented as a distributed algorithm based on message communication between agents.

2.4 Previous works

A number of exact search methods of DCOPs are performed based on the pseudo-trees. These algorithms are based on dynamic programming and branch-and-bound. Dynamic programming [4] computes the optimal cost values of sub-trees from leaf agents to a root agent. Then the optimal assignments are decided from a root agent to leaf agents. There is no iterative processing. However, the size of the memory and the messages are exponential to induced agents. There is no iterative processing. However, the size of the memory and the messages are exponential to induced width of the pseudo-trees because each agent \(i\) simultaneously computes \(g^*(s_i)\) for all assignments of the variables contained in Ast\(_i\). Although computation of the optimal cost values can be divided into iterative processing [5, 6], pruning methods that employ global cost values have not been discussed. In memory-bounded methods based on dynamic programming and branch-and-bound [3], one current context \(s_i\) is processed in each agent \(i\). If the overhead of the message communication is relatively large, it is reasonable to expand multiple solutions at the same time and to send them in one message because the number of iterations that contain the message communication is reduced. In previous studies, several efficient methods have been proposed [8, 9]. However, the methods that simultaneously expand multiple solutions have not been adequately discussed.

3 Multiple contexts search — basic solver

In this study, we generalize the exact methods of DCOPs to handle multiple current contexts. As a basic
method, we present an algorithm that does not employ pruning based on global cost values. The computation essentially resembles a memory bounded dynamic programming [6]. The method employs two types of messages (VALUES and COSTS in Figure 1(c)). As shown in Figure 2(a), multiple contexts are expanded using VALUES messages that are propagated based on the tree edges of a pseudo-tree in a top-down manner. In this example, three contexts are simultaneously expanded. On the other hand, cost values for current contexts are computed using COSTS messages that are propagated in a bottom-up manner. The processing is repeated until the upper and lower bounds of the global cost values reach identical values in the root agent. Then the optimal assignment of root variable is determined. Similarly, the optimal solution for the rest of the problem is recursively determined. In the following, we describe the components of the algorithm and show several explanations about important points.

Agent \( i \) uses following the variables and the data structures in addition to the representations of the DCOPs and the pseudo-trees:

\( S_i \): a set of current contexts for the variables contained in \( Ast_i \). The contexts are received from the parent of \( i \).

\( U_i \): a set that represents a table of the cost values for \( S_i \). Element \( u \) of \( U_i \) consists of \((a(u), lb(u), ub(u))\). \( a(u) \) is a partial solution. \( lb(u) \) and \( ub(u) \) represent the upper and lower bounds of the cost value of \( a(u) \). \( U_i \) is sent to the parent of \( i \).

\( sAct_i \): a state of \( S_i \). When new contexts are received from the parent of \( i \), \( sAct_i \) is set to true. As mentioned below, when \( closed(U_i) \) is true, \( sAct_i \) is set to false.

\( sOpt_i \): a state of \( S_i \), which is received from parent of \( i \). When \( S_i \) contains an optimal solution, \( sOpt_i \) is true. Otherwise, it is false.

\( opt_i \): a state of \( i \). When the optimal assignment of \( x_i \) has been determined, \( opt_i \) is true. Otherwise, it is false.

\( D_i^j \): a domain that contains only one optimal assignment of \( x_i \).

\( R_j^i \): a set of contexts for \( Ast_j \) of child \( j \), which is computed from \( S_i \). Let \( R_j \) denote set \(\{s|s' = s \cup \{(x_i, d), s \in S_i, d \in D_i\}\}\). An element of \( R_j^i \) is recomposed from an element of \( R_j \) by eliminating the assignments of variables that are not contained in \( Ast_j \). An element of \( R_j^i \) corresponds to at least one element of \( R_j \).

\( S_j^i \): a subset of \( R_j^i \), which is sent to child \( j \) as a set of current contexts.

\( U_j^i \): a set that represents a table of the cost values for \( R_j^i \).

The elements of \( U_j^i \) resemble elements of \( U_i \), and are updated using \( U_j \) received from child \( j \).

\( sFil_j^i \): If all \( S_j^i \) have been sent to child \( j \) for all elements of \( R_j^i \), \( sFil_j^i \) is true. Otherwise, it is false.

The messages transfer the following information:

VALUES: agent \( i \) sends \( S_j^i \) and \( sOpt_i \) to child \( j \) using

```
Main(){
    Initialize().
    until(forever){
        until(receive loop is broken){ receive messages. }
        if(Si ≠ φ ∧ sActi){ Maintain(). } }
    Initialize()
    Si ← φ. sActi ← false. sOpti ← false. opti ← false.
    Di^j ← φ.
    foreach j s.t. xj ∈ Chld,{
        (Sj^i, Uj^i, Rj, sFilj^i) ←(φ, φ, φ, false).}
    if(xi is root){ send (VALUES, {φ}, true) to i. }
    Receive VALUES, Si, sOpti{
        Si ← S. sActi ← true. sOpti ← sOpti.
    foreach j s.t. xj ∈ Chld,{
        reset child states (Sj^i, Uj^i, Rj, sFilj^i). } }
    Receive COSTS, j, Uj^i{
        if(compatible(U,Sj^i)) { merge U to Uj^i. }
    Maintenance(){
        update Ui and test optimal condition.
        if(opti ∧ (Di has not been replaced by Di^j)){
            replace Di by Di^j.
    foreach j s.t. xj ∈ Chld,{
        reset child states (Sj^i, Uj^i, Rj, sFilj^i). }
    update Ui and test optimal condition. }
    MaintainAndSendChildrensContexts().
    if(xi is root) {send (VALUES, Si, Uj) to i s.t. xj = prnt. }
    if(closed(Uj)) { sActj ← false. }
    MaintainAndSendChildrensContexts(){}
    foreach j s.t. xj ∈ Chld,{
        (−contexts of j are active) ∧ ¬sFilj^i){
            set next contexts Sj^i.
    if(Sj^i is last part of Rj)(sFilj^i ← true.)
    send (VALUES, Sj^i, optj) to j. }
```

Figure 3: Multiple contexts search

VALUES messages.

COSTS: agent \( i \) sends \( U_i \) that corresponds to \( S_i \), i.e. \( S_i^j \), to parent \( j \) by COSTS messages.

3.1 Reduced contexts in branching

Agent \( i \) sends the current sets of contexts to its child \( j \) using VALUES messages. The current set of contexts is a partial set of \( R_j^i \) that is a set of partial solutions for variables contained in \( Ast_j \). Assignments of other variables are eliminated because the cost values of the subtree rooted at \( j \) only depend on \( Ast_j \). This reduces the size and the number of partial solutions to be expanded. Additionally, the assignments of \( x_j \)’s ancestor variables that are not contained in \( Ast_i \) are not leaked to agent \( i \). When the cost values of the current context converge, the contexts \( S_j^i \) are updated to the next part of \( R_j^i \) that are sub-problems of \( S_i \).

When \( opt_i \) is false (i.e. the optimal assignments have not been determined), \( S_j^i \) is updated as follows. Let \( R_j^i^+ \) denote set \( R_j^i \backslash \{a(u) | u \in U_j^i \} \). \( S_j^i \) is updated to satisfy condition \( S_j^i \subseteq R_j^i^+ \land |S_j^i| \leq L \). Here \( L \) is the maximum number of current contexts. Then initial values \( \{u|a(u) \in S_j^i, lb(u) = 0, ub(u) = ∞\} \) are added to \( U_j^i \). On the other hand, when \( opt_i \) is true, \( S_j^i \) is updated so that \( S_j^i \) contains
only one optimal element. If $U^j_i$ contains no element $u$ such that $a(u) = r$ for $r \in R^j_i$, $(r, 0, \infty)$ is added to $U^j_i$.

3.2 Asynchronous computation of cost values

Agent $i$ receives the upper and lower bounds $U_j$ of the cost values of $R^j_i$ from its child $j$ using COSTS messages. Then the cost values are stored in $U^j_i$. For current contexts $S_i$, cost table $U_i$ of the subtree rooted at $i$ is computed based on all $U^j_i$ and the local costs of $i$. For all elements $s$ of $S_i$, $U_i$’s elements $u_s$ such that $u_s = (s, lb(u_s), ub(u_s))$ are computed. $lb(u_s)$ and $ub(u_s)$ are computed based on the computation of $U^j_i$ shown in Expressions 2 and 3. First, for each $d$ in $D_i$, lower bound cost $lb_{s \cup \{(x_i, d)\}}$ of $s \cup \{(x_i, d)\}$ is computed using local cost $\delta(s \cup \{(x_i, d)\})$ and all $lb(u')$ such that $a(u') \subseteq s \cup \{(x_i, d)\}$, $u' \in U^j_i, x_i \in \text{Child}_i$. Then $lb(u_s)$ is computed as $\min_{d \in D_i} lb_{s \cup \{(x_i, d)\}}$. Similarly, $ub(u_s)$ is computed using upper bound cost values. If $(s(u'), lb(u'), ub(u'))$ are not contained in $U^j_i$, the values of $lb(u')$ and $ub(u')$ are 0 and $\infty$.

i repeatedly sends COSTS messages to its parent until $lb(u) = ub(u)$ for all elements $u$ of $U_i$. Note that the boundaries $lb(u)$ and $ub(u)$ cannot be identical in the first step of the computation because they are partially computed. On the other hand, when the boundaries of the cost values are converged in all $U^j_i$’s elements that corresponds to $S^j_i$, agent $i$ asynchronously updates $S^j_i$ to the next set of contexts. Then $S^j_i$ is sent to $j$ by a VALUES message to $i$’s children. Even if COST messages for the previous contexts are received from the children, the old messages are ignored.

3.3 Details of the basic solver

Details of the basic solver is shown in Figure 3. The following are the procedures and functions in the figure:

- **reset child states** ($S^j_i, U^j_i, R^j_i, S^j_i, \text{FILL}^j_i$): For child $j$, ($S^j_i, U^j_i, \text{FILL}^j_i$) $\leftarrow$ ($\phi, \phi$, false). Then $R^j_i$ is updated.

- **compatible($U_i, S_i$)**: If $S$ contains $a(u)$ for all $u$ in $U$, the return value is true. Otherwise, the return value is false.

- **merge $U_i$ to $U^j_i$**: For all elements $u$ of $U_i$, $U_i$’s elements $u'$ such that $a(u') = a(u)$ are updated as follows. $lb(u') \leftarrow \max(lb(u'), lb(u))$, $ub(u') \leftarrow \min(ub(u'), ub(u))$.

- **update $U_i$ and test optional condition**: $U_i$ is updated as shown in Section 3.2. In the computation of $U_i$, a condition of the optional solution is also tested. If $s_{\text{Opt}_i} \land \exists u \in U_i, lb(u) = ub(u) \land \neg \text{opt}_i$, then $i$ determines the optimal solution as follows: $\text{opt}_i \leftarrow \text{true}$, $D^i_{\text{Opt}} \leftarrow \{d^*\}$. In the above, the upper bound cost of $a(u) \cup \{(x_i, d^*)\}$ equals to $lb(u)$ and $ub(u)$. The algorithm selects one optional solution. Therefore, when $s_{\text{Opt}_i}$ is true, $|S^j_i| = |U_i| = 1$.

- **closed($U_j$)**: If, for all $u$ in $U_j$, $lb(u) = ub(u)$, then the return value is true. Otherwise, the return value is false.

- **set next contexts** $S^j_i$: $S^j_i$ is updated to the next contexts as shown in Section 3.1.

**contexts of $j$ are active**: For at least one element $s$ of $S^j_i$ and $U^j_i$’s element $u$ such that $a(u) = s$, if $lb(u) \neq ub(u)$, then the return value is true. Otherwise, the return value is false.

4 Pruning and reusing cost values

In this section, we apply efficient methods to the basic algorithm shown in Section 3. The most important point is the pruning based on global cost values. To compute the upper and lower bounds of the global cost values, the cost values of ancestors and other subtrees are computed in a top-down manner as shown in Figure 2(b). In this example, cost values of the partial network except for the subtree rooted at $x_j$ are aggregated via $x_i$. Now VALUE messages transfer cost values of contexts. As the summations of the top-down costs of $x_j$ and the costs of $x_j$’s subtree, the global upper and lower bounds of cost values are obtained. The global upper bounds improve the best cost value. On the other hand, the global lower bounds are evaluated to prune partial solutions. New top-down message GUB is introduced to propagate the globally best cost value(Figure 1(c)). Moreover, the computation results are reused to reduce redundant searches. Below we mainly describe the difference from the basic methods. The variables and the data structures of agent $i$ are modified as follows:

- **$S_i$**: Each partial solution is integrated with the upper and lower bounds of the cost values that are computed in a top-down manner. Similar to $U_i$, element $s$ of $S_i$ consists of $(a(s), lb(s), ub(s))$. $a(s)$ is a partial solution. $lb(s)$ and $ub(s)$ are upper and lower bounds of cost values for ancestors and other subtrees.

- **$R^j_i$**: Similar to $S_i$, the elements of $R^j_i$ consist of a partial solution and the related upper and lower bounds. The $lb(r)$ of $R^j_i$’s element $r$ is computed as a summation of the following cost values:
  - $lb(s)$ of $S_i$’s element $s$ such that $(a(r) \setminus \{(x_i, d)\}) \subseteq s$,
  - local cost $\delta(s \cup \{(x_i, d)\})$,
  - all $lb(u')$ such that $a(u') \subseteq s \cup \{(x_i, d)\}, u' \in U^j_i, x_i \in \text{Child}_i \setminus \{x_j\}$.

  Similarly, $ub(r)$ is computed using the upper bounds of the cost values. $S_i$ can contain multiple elements $s$ such that $(a(r) \setminus \{(x_i, d)\}) \subseteq s$. In such cases, minimum $lb(r)$ and maximum $ub(r)$ are used as $lb(r)$ and $ub(r)$.

- **$S^j_i$**: The elements of $S^j_i$ is resemble the elements of $S_i$ and $R^j_i$.

- **$U^j_i$**: $U^j_i$ represents the upper and lower bound costs for $R^j_i$, which is also employed as a cache memory. When $S_i$ is updated to different set of contexts, the elements of $U^j_i$ are not immediately erased. If element $u$ of $U^j_i$ corresponds to the element of current $R^j_i$ (for $S_i$), then $u$ is reused. If $|U^j_i| \leq K$ is not satisfied when a new element is added to $U^j_i$, then an element is erased based on LRU. Here, $K \geq |R^j_i|$.
In addition, the following elements are employed:

- \( gUB_i \): Upper bound of global cost values. \( gUB_i \) represents the currently minimal cost value of complete solutions.
- \( sCls_i \): a state of \( S_i \). \( sCls_i \) represents the value of \( cls_j \) of parent \( j \).
- \( cls_i \): represents that the upper and lower bounds of the top-down costs are closed in \( i \). If \( sCls_i \) is true and \( lb(s) = ub(s) \) is true for all children \( j \) and all elements \( s \) of \( S_j \), then its value is true. Otherwise, its value is false.
- \( sChg_i \): represents that \( S_i \) or the state of \( S_i \) have been just changed when its value is true.
- \( sFin_i \): a state of \( S_i \). The initial value of \( sFin_i \) is false. When root agent \( r \) determines the optimal assignment of \( x_r \), \( sFin_i \) is set to true. Then the value of \( sFin_i \) is propagated to all agents. This changes the pruning condition.

There are no modifications in \( U_i \), \( sOpt_i \), \( opt_i \), \( D_i \), \( sAct_i \), \( sFil_i \). The following are the modifications of the messages:

**VALUES:** The structures of \( S_j \) and the related states are modified. VALUES messages that concern current \( S_j \) are repeatedly sent until the boundaries of the costs of \( S_j \) are closed.

**GUB:** The value of \( gUB_i \) is sent to the children by the GUB messages.

### 4.1 Top-down computation of cost values

In this method, the top-down computation of the cost values for ancestors and other subtrees, and the bottom-up computation of the optimal cost are performed at the same time. By combining the top-down and bottom-up costs, more precise boundaries of the global costs can be computed. To compute the top-down costs, elements \( s \) of current contexts \( S_i \) that are sent from \( i \)’s parent to agent \( i \) consist of partial solution \( a(s) \), upper bound \( ub(s) \), and lower bound \( lb(s) \) of the cost value. The cost value of each element of \( R_i \) that is a set of partial solutions for child \( j \) of agent \( i \) is computed by adding the costs of \( S_i \), the local cost of \( i \), and the costs of \( U_k \) for all children \( k \) except for the child \( j \) (Figure 2(b)). Note that the multiple elements of \( S_i \) can correspond to an element of \( R_i \) when the assignments of the contexts are reduced as shown in Section 3.1. In such a case, the widest boundaries are used so that the boundaries do not exceed the true cost value.

Current set \( S_i \) of the contexts for \( R_j \) is sent from \( i \) to \( j \) by VALUES messages. If there are other subtrees of agent \( i \), the boundaries of the subtrees’ costs may not be closed immediately after \( S_i \) is updated. Therefore, until the boundaries are closed, the VALUES messages are repeatedly sent. The top-down computation depends on current \( S_i \). However, the computation is independent from the bottom-up computation of the optimal cost. Its convergence is managed using \( cls_i \) and \( sCls_i \).

### 4.2 Pruning using global cost values

Agent \( i \) minimizes upper bound \( gUB_i \) of the global cost value. \( gUB_i \) is sent to the other agents by GUB messages. By comparing \( gUB_i \) and the lower bound of the costs of the current solution, \( i \) judges whether the search can be pruned. Pruned partial solutions are not sent to child agents. When the optimal assignment of the root variable has not been determined yet, the search is pruned if the lower bound exceeds or equals to the global upper bound. After the optimal assignment of the root variable is determined, the search is pruned if the lower bound exceeds the global upper bound. To switch the pruning condition, \( sFin_i \) is propagated.

### 4.3 Reusing cost values

Agent \( i \) stores the information of the cost values received from child \( j \) into \( U_j \). When current contexts \( S_i \) are updated, elements \( u \) of \( U_j \) are not immediately erased. To limit \( |U_j| \), the elements of \( U_j \) are managed by LRU. However, \( u \), which corresponds to \( R_j \) (and current \( S_i \)), must not be erased from \( U_j \) while the costs for \( S_i \) are being computed. Therefore, to avoid erasing these elements, when \( S_i \) is updated, the reusable elements of \( U_j \) are moved to the end of the LRU queue.

### 4.4 Details of the efficient solver

Details of the efficient algorithm is shown in Figure 4. The following are the procedures and functions in the figure:

- **reset child states** \((S_j, U_j, R_j, sFil_j, sChg_j)\): For child \( j \), \((S_j, sFil_j, sChg_j)\) is sent to \( i \). Then \( R_j \) is updated based on current \( S_i \) and \( D_j \). All \( U_j \)’s elements \( u \) such that \( \exists r \in R_j, a(r) = a(u) \) are moved to the end of the LRU queue.

- **compatible** \((U, S)\): If \( S \) contains \( s \) such that \( a(s) = a(u) \) for all elements \( u \) of \( U \), the return value is true. Otherwise, the return value is false.

- **update** \((U_i, gUB_i)\) and **test optimal condition**:
  - There are no modifications in the computation of \( U_i \). Upper bound \( gub_s \) of the global cost value is computed for each element \( s \) of \( S_i \). First, for each \( s \) and \( d \in D_i \), upper bound \( gub_{a(s),a\{x_i,d\}} \) is computed as a summation of the following cost values:
    - \( ub(s) \).
    - Local cost \( \delta_i(a(s) \cup \{(x_i, d)\}) \).
    - All \( ub(u') \) such that \( a(u') \subseteq a(s) \cup \{(x_i, d)\}, u' \in U_j, x_j \in Chld_i \).

  Then \( gub_s \) is computed as \( \min_{d \in D_i} gub_{a(s) \cup \{(x_i, d)\}} \). \( gub_s \) is used for two computations:
    - \( gUB_i \) is updated so that \( gUB_i \leftarrow \min(gub_s, gUB_i) \).
    - As shown in the following, the condition of the optimal solution contains \( gub_s \).
Main() {
  Initialize().
  until(forever) {
    for (x is root) {
      if (cond) {
        ...}
      else {
        ...}
    }
  }
}

until(receive loop is broken) {
  receive messages.
  for each receive message {
    if (is root) {
      ...}
    else {
      ...}
  }
}

foreach j s.t. j ∈ Child{(...)}
foreach j s.t. j ∈ Child{
  if (is root) {
    ...}
  else {
    ...}
}

Figure 4: Multi-solution search with efficient methods

If $s_{Opt_1} \land ((\exists u \in U_i, lb(u) = ub(u)) \lor (\exists s \in S_i, gb_{u} = gUB_i)) \land \neg opt_1$, then an optimal solution is determined. Here, if $\exists s \in S_i, gb_{u} = gUB_i$, then $d^s$ such that $gb_{u}(s) = (x, d^s) = gb_{u}$ is the optimal assignment of $x_i$. Otherwise, the computation of the optimal assignment is the same as the basic method.

closed($U_i$): In addition to the original condition, lower bound $gb_{u}$ of the global cost for element $s$ of $S_i$ is employed. The computation of $gb_{u}$ is the same as the computation of $gb_{u}$ except that the lower bounds are used instead of the upper bounds. Here, define condition $cond_{gb_{u}}$ that depends on $sFin_i$:

$$cond_{gb_{u}} = \begin{cases} gb_{u} > gUB_i & sFin_i \\ gb_{u} \geq gUB_i & otherwise \end{cases}$$

If $lb(u) = ub(u) \lor (cond_{gb_{u}}$ where $a(s) = a(u))$ for all elements $u$ of $U_i$, then the return value of closed($U_i$) is true. Otherwise, the return value is false.

set next contexts $S_j^+$: When $opt_1$ is false, then $S_j^+$ is updated as follows. Let $R_j^+$ denote a subset of $R_j^+$, $R_j^+$ does not contain elements that have been already searched or pruned. $S_j^+$ is updated as a subset of $R_j^+$. If $U_j^+$ does not contain $u$ such that $a(s) = a(u)$ for an element $s$ of $S_j^+$, then $(a(s), 0, \infty)$ is added to $U_j^+$. The following computation generates $R_j^+$ from $R_j^+$. First, for each element $r$ of $R_j^+$, lower bound $lb_{sF}$ = $lb(r) + lb(u)$ of the cost is computed. Here, $u$ such that $a(r) = a(u)$ is an element of $U_j^+$. Using condition $cond_{gb_{u}}$, that resembles Expression 4, element $r$ is evaluated. If $lb(u) = ub(u) \lor cond_{gb_{u}}$, then $r$ is not contained in $R_j^+$. When $opt_1$ is true, the computation is the same as the basic method.

update $R_j^+$: $R_j^+$ is updated according to current $S_j^+$.

update current contexts $S_j^+$: For all elements $s$ of $S_j^+$, $lb(s)$ and $ub(s)$ are updated based on current $R_j^+$.

all children’s contexts are closed: If $lb(s) = ub(s)$ for all children $j$ and all elements $s$ of $S_j^+$, then the return value is true. Otherwise, the return value is false.

contexts of $j$ are active: If $lb(u) \neq ub(u) \land \neg cond_{lb_{u}}$ for at least one element $s$ of $S_j^+$, the return value is true. Here, $u$ such that $a(u) = a(s)$ is an element of $U_j^+$. Otherwise, the return value is false.

There are no modifications in ‘merge $U$ to $U’’ and COSTS messages.

4.5 Extra messages

In the algorithms shown in Figure 3 and 4, identical messages are often repeatedly sent. Such redundant sending of messages can be blocked. However, for notification of a change of state, messages are sent when the VALUES messages are received (and when $sChg$ is true in Figure 4).

5 Correctness

The essential computation of the algorithms shown in this paper resembles conventional algorithms. The bottom-up computation of optimal cost and the top-down computation of global cost depend on current contexts. When the current contexts have not changed, the boundaries of the cost values are monotonously narrowed. The current contexts of each agent eventually reflect the last decision of the ancestor agents. Therefore, if pruning is not applied, the boundaries converge to an optimal value. Pruning does not destroy the boundaries because it only skips
the search. Moreover, pruning is only judged based on the current states of each agent. The upper bound of the global cost value is monotonously decreased and immediately propagated. Switching of the pruning condition is also immediately propagated using $sF in_i$. Therefore, situations where pruning is excessive eventually disappear and the search processing converges.

6 Evaluation

As the first result, we experimentally evaluated the effects of our proposed methods, which were applied to two types of problems that consist of $n$ ternary variables and $l \cdot n$ binary constraints/functions. Similar classes of the problems are used to evaluate DCOP solvers in several studies. Here parameters $l = 1.2, 1.5, 2, 3$ were used.

**max-csp-gcl**: the problems resemble maximum constraint satisfaction problems for vertex coloring. If $x_i = x_j$, then the value of $f_{i,j}(x_i, x_j)$ is 1. Otherwise, the cost value is 0.

**cop**: $f_{i,j}(x_i, x_j)$ takes integer values between 0 and 10. The values are randomly set based on uniform distribution.

The results are average of twenty instances. Actually, the results are significantly affected by the instances whose induced width is large. b (Figure 3), c (b with reusing $U_j^i$), ppgc (Figure 4), ppg (ppgc without reusing of $U_j^i$), pp (pp without $yU_B_j$ in the test of optimal condition), and pl (pp without top-down costs) were compared. Solutions are expanded in a depth-first manner based on a fixed order of variables/values. Maximum number $L = 1, 3, 9, 27$ of $|S_j^i|$ and maximum size $K = L \cdot |D_i|$ of $|U_j^i|$ are used. The number of message cycles was evaluated. In each message cycle, agents receive messages from their receive-queue, perform its processing and send messages to their send-queue if necessary. Then the messages in the queues are exchanged. The experiment was stopped at the $10^5$ cycle. In that case, the number of message cycles was considered to be $10^5$.

The number of message cycles in max-csp-gcl is shown. Figure 5 shows the case of $l = 1.5, L = 27$. In the problems of $n = 50, 100$, both pruning and reusing $U_j^i$ are effective. Induced-width tends to grow with the increase of $n$ in these problems. Therefore, the effect of c, which only employs the reuse of $U_j^i$, is small when $n$ is relatively large. Figure 6 shows the case of $n = 20, l = 2$. In these results, the increment of $L$ is relatively effective in the methods that employ pruning. Figure 7 shows the case of $n = 20, L = 27$. The results resemble Figure 5 because the induced width tends to grow with the increase of $l$. When pruning methods of pl, pp and ppg are effective, they complementarily reduce the number of message cycles. The number of message cycles in cop is shown in Figure 8. Increasing $L$ reduced the number of message cycles. However, the difference of ppgc and c is relatively small even if dependence on pruning is large ($L = 1, 3$). In the results, only reusing $U_j^i$ is relatively effective. Actually, ppgc is less effective than c in several results. This drawback might be caused by the pruning that affects the order
of the cached elements.

The size of the pseudo-trees is shown in Table 1. The induced width of the pseudo-trees grow with the increase of $f$ and $n$. Therefore, maximum size $s_z$ of the partial problems increases. Table 2 shows the number of messages per message cycle. The number of pairs of agents (source, destination) that transferred at least one message in one message cycle is also shown. When no messages are received, the agents are in quiescence. Redundant sending of messages is blocked, as shown in 4.5. Therefore, the frequency of message communication is relatively small.

In the algorithms shown in Sections 3 and 4, current set $S_i^j$ of the contexts is updated to subsequent contexts when all boundaries of all contexts are closed. However, when the boundaries of one element $s$ of $S_i^j$ are closed, $s$ can be immediately replaced with the next context. We applied the above methods to $c$ and $ppgc$. Figure 9 shows the results of the methods. $cus$ and $ppgcus$ are the modified algorithms. The number of message cycles of $c$ was relatively decreased in several cases. On the other hand, for $ppgc$ that employs pruning, the effect of the modification was small.

### 7 Conclusion

In this study, we present the basic algorithms of memory-bounded solvers that employ multiple search points. Such methods resemble the execution of overlapped multiple search processes. In search processing, the solver recomposes the sets of solutions that are currently expanded. As a result, iterative search that depends on message communication is reduced. Basically, the proposed methods are generalizations of previous studies. To clarify the essence of the processing, we obviously used several fundamental computations in the algorithms. The solution spaces of the sub-problems are considered when sub-solutions are expanded. The boundaries of global cost values are exactly and separately computed. Such methods can be considered as the basis of solvers that employ mixed search strategies. Several experimental results show that the proposed method efficiently works with pruning and reusing results. Detailed investigation of the trade-off between computation and communication, multiple search strategies, and other additional efficient methods are future works.

### Acknowledgements

This work was supported in part by a Grant-in-Aid for Young Scientists (B), 22700144 and the Kayamori Foundation of Informational Science Advancement.

### References