Completeness Matters: Towards Efficient Caching in Tree-Based Synchronous Backtracking Search for DCOPs

Jie Wang[1]
College of Computer Science, Chongqing University, China

Dingding Chen[2]
College of Computer Science, Chongqing University, China

Ziyu Chen[3]
College of Computer Science, Chongqing University, China

Xiangshuang Liu
College of Computer Science, Chongqing University, China

Junsong Gao
College of Computer Science, Chongqing University, China

Abstract

Tree-based backtracking search is an important technique to solve Distributed Constraint optimization Problems (DCOPs), where agents cooperatively exhaust the search space by branching on each variable to divide subproblems and reporting the results to their parent after solving each subproblem. Therefore, effectively reusing the historical search results can avoid unnecessary resolutions and substantially reduce the overall overhead. However, the existing caching schemes for asynchronous algorithms cannot be applied directly to synchronous ones, in the sense that child agent reports the lower and upper bound rather than the precise cost of exploration. In addition, the existing caching scheme for synchronous algorithms has the shortcomings of incompleteness and low cache utilization. Therefore, we propose a new caching scheme for tree-based synchronous backtracking search, named Retention Scheme (RS). It utilizes the upper bounds of subproblems which avoid the reuse of suboptimal solutions to ensure the completeness, and deploys a fine-grained cache information unit targeted at each child agent to improve the cache-hit rate. Furthermore, we introduce two new cache replacement schemes to further improve performance when the memory is limited. Finally, we theoretically prove the completeness of our method and empirically show its superiority.

2012 ACM Subject Classification Computing methodologies → Cooperation and coordination

Keywords and phrases DCOP, Cache, Any-space Algorithms, Complete Search Algorithms

Digital Object Identifier 10.4230/LIPIcs.CP.2022.39

Supplementary Material *Software (Source Code)*: `https://github.com/czy920/RS`

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) [15, 11] are a popular framework for multi-agent systems (MAS) where agents need to cooperate with each other to optimize a global objective. Owing to their excellent modeling ability, DCOPs have been widely applied in many real-world problems such as sensor network [9], task scheduling [18, 27], smart grid [12] and so on.

[1] Equal contribution.
[2] Equal contribution.
[3] Corresponding author.

Incomplete algorithms for DCOPs [31, 17, 10, 23, 22] aim to rapidly find an acceptable solution at the expense of sacrificing optimality, while complete algorithms ensure the optimal solution and can be generally divided into inference-based and search-based algorithms. DPOP [24] and Action_GDL [28] are typical inference-based complete algorithms which employ dynamic programming to solve DCOPs. However, they require a linear number of messages of exponential size which bring an excessive network load. Whereupon, ODPOP [25], MB-DPOP [26] and RMB-DPOP [6] were proposed to trade the number of messages for smaller memory consumption by propagating the dimension-limited utilities with the corresponding contexts iteratively. Besides, DPOP with function filtering [2] proposed to exploit utility bounds to reduce the size of messages.

Search-based complete algorithms perform distributed backtracking search to exhaust the search space and have a linear size of messages. Among them, tree-based complete search algorithms have attracted wider attention due to their high concurrency and can further be classified into synchronous and asynchronous. Given a context, tree-based complete synchronous search algorithms like NCBB [3], PT-FB [16] and HS-CAI [5] require each agent to report its search result after it has thoroughly explored its subproblem, while asynchronous ones like ADOPT [20] and BnB-ADOPT [29] allow each agent to report its search results solely based on its local view of its subproblem. To accelerate the search process, these algorithms utilize the upper and lower bounds to prune the search space. Accordingly, many pruning techniques like soft arc consistency [14], forward bounding procedure [16] and inference-based estimation [8, 1, 5] have arisen to tighten the lower bounds. On the other side, tree-based backtracking search requires agents to report their search results regarding their subproblems given a context. When receiving the same context, agents can effectively reuse the historical search results to avoid unnecessary resolutions and substantially reduce the overall overhead. Accordingly, any-space algorithms using historical exploration results to speed up the search process have been proposed. Matsui et al [19] proposed a caching scheme for ADOPT, named any-space ADOPT, where each agent caches the upper and lower bounds of its subproblem given a context. Yeoh et al [30] further improved the scheme by introducing three cache replacement schemes to boost the cache utilization. However, these caching schemes for asynchronous algorithms are not suitable for synchronous algorithms since synchronous algorithms need to report the optimal cost for each subproblem rather than the bounds. Although tree-based complete synchronous search algorithms have been widely studied, there are few studies on cache schemes for them. Any-space NCBB [4], the only caching scheme for synchronous algorithms now, presented a caching scheme for NCBB where a caching unit is introduced to store the search result regarding a given context for each agent. Unfortunately, the scheme fails to consider the impact of pruning on cached results and leads to the incompleteness. And also, the scheme matches by comparing all the separators of local agent, which brings the low cache-hit rate. Therefore, the cache scheme in any-space NCBB cannot be applied to existing tree-based complete synchronous search algorithms. To this end, we present a complete and effective caching scheme for tree-based complete synchronous search algorithms and our main contributions are listed as follows:

- We systematically analyze the cause for the incompleteness of the existing synchronous caching scheme, and provide a solution which compares the historical upper bound with the current upper bound to determine whether the historical results can be reused.
- To improve the cache-hit rate, we introduce a fine-grained cache information unit targeted at each child agent which allows each agent to independently reuse the historical results of subproblem rooted at its child. Along with the solution to the incompleteness, we propose our Retention Scheme (RS) accordingly, which is suitable for all tree-based complete synchronous search algorithms.

- In addition, to improve the performance of our RS when the memory is limited, we propose two heuristic cache replacement schemes which consider the characteristics of the cached information units and synchronous algorithms, respectively.
- We theoretically show the completeness of RS, and the experimental results demonstrate it improves state-of-the-art tree-based complete synchronous search algorithms on all the metrics in most cases.

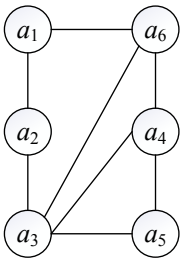
2 Background

In this section, we first introduce the preliminaries including DCOPs, pseudo tree, tree-based complete synchronous search algorithms and the caching scheme in any-space NCBB.

2.1 Distributed Constraint optimization Problems

A distributed constraint optimization problem [20] can be defined by a tuple $\langle A, X, D, F\rangle$ where

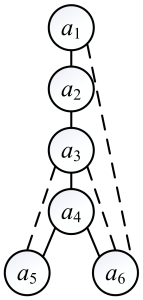
- $A = \{a_1, a_2, \cdots, a_n\}$ is a set of agents.
- $X = \{x_1, x_2, \cdots, x_m\}$ is a set of variables.
- $D = \{D_1, D_2, \cdots, D_m\}$ is a set of finite variable domains, each variable x_i taking a value in D_i.
- $F = \{f_1, f_2, \cdots, f_q\}$ is a set of constraints, each of which $f_i : D_{i_1} \times D_{i_2} \times \cdots D_{i_k} \to \mathbb{R}_{\geqslant 0}$ denotes the non-negative cost for each assignment combination of $x_{i_1}, x_{i_2}, \cdots x_{i_k}$.



(a) Constraint graph.

x_i \ x_j	0	1
0	7	1
1	4	5

(b) Constraint matrix.



(c) Pseudo tree.

Figure 1 An example of a DCOP and its pseudo tree.

For the sake of simplicity, the paper assumes that each agent holds exactly one variable (i.e., $n = m$) and all constraints are binary relations (i.e., $f_{ij} : D_i \times D_j \to \mathbb{R}_{\geqslant 0}$). Thus, the term "agent" and "variable" could be used interchangeably. Without loss of generality, a DCOP seeks an assignment to all the variables that minimizes the total cost. Formally,

$$X^* = \underset{d_i \in D_i, d_j \in D_j}{\arg\min} \sum_{f_{ij} \in F} f_{ij}\,(x_i = d_i, x_j = d_j)$$

In general, a DCOP can be visualized by a constraint graph where vertexes represent the agents and edges represent the constraints, respectively. Fig.1(a) presents a DCOP with six variables and eight constraints. For simplicity, the domain size of each variable is two and all constraints are identical as shown in Fig.1(b) where $i < j$.

2.2 Pseudo Tree

A pseudo tree [13, 7] is a partial ordering among agents, which can be generated by depth-first search (DFS) traversal to a constraint graph. It has the property that different branches are independent, and categorizes its constraints into tree edges and pseudo edges (i.e., non-tree edges). Fig.1(c) presents a possible pseudo tree deriving from Fig.1(a) where tree edges and pseudo edges are denoted by solid and dashed lines, respectively. For an agent a_i, its neighbors can be categorized into its parent $P(a_i)$, children $C(a_i)$ and pseudo parents $PP(a_i)$, according to their positions in the pseudo tree and the type of edges they connect through. Each agent directs its message passing and traverses the solution space through such a (pseudo) parent-child relationship. More precisely, they can be formally defined as follows:

- $P(a_i)$ is the ancestor connecting with a_i through a tree edge (e.g., $P(a_4) = a_3$ in Fig.1(c)).
- $C(a_i)$ is the set of descendants connecting with a_i through tree edges (e.g., $C(a_4) = \{a_5, a_6\}$ in Fig.1(c)).
- $PP(a_i)$ contains the ancestors that connect with a_i through pseudo edges (e.g., $PP(a_5) = \{a_3\}$ in Fig.1(c)).

For succinctness, we also adopt the following notations.

- $AP(a_i)$ is the set of all (pseudo) parents of a_i. i.e., $AP(a_i) = PP(a_i) \cup \{P(a_i)\}$ (e.g., $AP(a_5) = \{a_3, a_4\}$ in Fig.1(c)).
- $Anc(a_i)$ is the set of ancestors of a_i (e.g., $Anc(a_4) = \{a_1, a_2, a_3\}$ in Fig.1(c)).
- $Desc(a_i)$ is the set of descendants of a_i (e.g., $Desc(a_3) = \{a_4, a_5, a_6\}$ in Fig.1(c)).
- $Sep(a_i)$ [11] is the separator set of a_i, comprising the ancestors that are constrained with agents in $\{a_i\} \cup Desc(a_i)$ (e.g., $Sep(a_4) = \{a_1, a_3\}$ in Fig.1(c)), i.e.,

$$Sep(a_i) = \{a_j \in Anc(a_i) \,|\exists a_k \in \{a_i\} \cup Desc(a_i), s.t., a_j \in AP(a_k)\}$$

2.3 Tree-based Complete Synchronous Search Algorithms

Tree-based complete synchronous search algorithms perform a branch-and-bound search on a pseudo tree to exhaust the search space. Specifically, each agent in the algorithms obtains the optimal cost of subproblem rooted at itself under the current partial assignment, and prunes to avoid unnecessary exploration by exploiting the bounds including the lower and upper bounds of its subproblem. In fact, most of the existing synchronous algorithms (e.g., NCBB, PT-FB and HS-CAI etc.) can be regarded as variants of SBB [15] on a pseudo tree, which utilize different techniques to obtain a tighter lower bound to improve pruning. Algorithm 1 presents the sketch of the implementation of SBB on a pseudo tree (named TreeBB) to describe the general framework of tree-based complete synchronous search algorithms.

In TreeBB, each agent a_i needs to maintain the following data structures.

- Cpa_i refers to the current partial assignment which contains all the assignments to $Anc(a_i)$.
- ub_i is the upper bound of its subproblem under Cpa_i.
- $opt_i^c(d_i)$ is the optimal cost of its child $a_c \in C(a_i)$ for $d_i \in D_i$, which is set to infinite if $Cpa_i \cup \{x_i, d_i\}$ is infeasible under ub_i.
- $sendub_c(d_i)$ is the upper bound sent to its child a_c for $d_i \in D_i$, i.e.,

$$sendub_c(d_i) = ub_i - \sum_{a_j \in AP(a_i)} f_{ij}(d_i, Cpa_i(a_j)) - \sum_{a_j \in C(a_i) \wedge opt_i^j \neq null} opt_i^j(d_i) \quad (1)$$

Algorithm 1 TreeBB for a_i.

```
   When Initialization ():
1  |  if a_i is the root agent then
2  |  |  InitializeVariables()
3  |  |  d_i ← the first element in D̃_i^c, Cpa_i ← {(x_i, d_i)}
4  |  |  send CPA(Cpa_i, ∞) to ∀a_c ∈ C(a_i)
   When received CPA (Cpa_i, ub_i) from P(a_i):
5  |  store {Cpa_i, ub_i}
6  |  if a_i is a leaf agent then
7  |  |  SendBacktrack ()
8  |  else
9  |  |  InitializeVariables()
10 |  |  ExploreValue (a_c), ∀a_c ∈ C(a_i)
   When received BACKTRACK (opt*) from a_c ∈ C(a_i):
11 |  d_i ← SrchVal_i^c, D̃_i^c ← D̃_i^c\{d_i} ,opt_i^c(d_i) ← opt*
12 |  if a_i has received all BACKTRACK messages from C(a_i) for d_i then
13 |  |  ub_i ← min(ub_i, lb_i(d_i))
14 |  ExploreValue (a_c)
   Function InitializeVariables ():
15 |  D̃_i^c ← D_i, ∀a_c ∈ C(a_i)
16 |  opt_i^c(d_i) ← null, ∀a_c ∈ C(a_i), d_i ∈ D_i
   Function ExploreValue (a_c):
17 |  d_i ← NextFeasibleAssignment (a_c)
18 |  if D̃_i^c = ∅, ∀a_c ∈ C(a_i) then
19 |  |  if a_i is the root agent then
20 |  |  |  Algorithm terminates.
21 |  |  else
22 |  |  |  SendBacktrack ()
23 |  else
24 |  |  SrchVal_i^c ← d_i, Cpa_c ← Cpa_i ∪ {(x_i, d_i)}
25 |  |  send CPA(Cpa_c, sendub_c(d_i)) to a_c
   Function NextFeasibleAssignment (a_c):
26 |  d_i ← the first element in D̃_i^c
27 |  while d_i ≠ null ∧ lb_i(d_i) ⩾ ub_i do
28 |  |  D̃_i^c ← D̃_i^c\{d_i}, opt_i^c(d_i) ← ∞, d_i ← the first element in D̃_i^c
29 |  return d_i
   Function SendBacktrack ():
30 |  opt* ← min_{d_i ∈ D_i} lb_i(d_i)
31 |  Send BACKTRACK(opt*) to P(a_i)
```

- $lb_i(d_i)$ is its lower bound for $d_i \in D_i$, i.e.,

$$lb_i(d_i) = \sum_{a_j \in AP(a_i)} f_{ij}(d_i, Cpa_i(a_j)) + \sum_{a_c \in C(a_i) \wedge opt_i^c \neq null} opt_i^c(d_i) \tag{2}$$

- opt^* is the current optimal cost of its subproblem under Cpa_i, i.e.,

$$opt^* = \min_{d_i \in D_i} lb_i(d_i) \tag{3}$$

TreeBB begins with the root agent sending the first value in its domain to its children (line 1-4). When an agent a_i receives a CPA message from its parent, it first stores the partial assignment Cpa_i and the upper bound ub_i (line 5), then initializes the search domain $\tilde{D}_i^c$ and opt_i^c (line 9,15-16). Next, a_i finds the first feasible assignment, i.e., the first assignment d_i such that $lb_i(d_i) < ub_i$ (line 26-29) and explores the corresponding subproblem (line 10, 17). If such an assignment exists, a_i sends a CPA message together with $sendub_c(d_i)$ calculated by Eq.(1) to its children (line 23-25). Otherwise, it sends a BACKTRACK message with the optimal cost opt^* to its parent (line 18-22, line 30-31).

When a_i receives a BACKTRACK message for d_i from its child a_c, it updates the corresponding optimal cost $opt_i^c(d_i)$ with the actual cost opt^* reported by a_c and continues to explore the subproblem corresponding to its next feasible assignment (line 11, 14). If a_i

has received all the BACKTRACK messages for d_i from its children, it updates the current upper bound for its subproblem (line 12-13). Finally, TreeBB terminates after the root agent exhausts its search domain and a global optimal cost will be got (line 18-20).

2.4 Caching Scheme in Any-space NCBB

To better utilize the historical search results, any-space NCBB proposed that agent a_i caches the historical cost and the corresponding assignments of $Sep(a_i)$, called $context_i$. To this end, any-space NCBB constructs a caching information unit I, a map set $\langle context_i, result \rangle$, to store these historical costs and *contexts* for future use. Here, we refer to such a caching scheme as OCS. Taking Fig.1(c) for example, assume that a_4 has obtained the current optimal cost opt^* of its subproblem. The opt^* is valid as long as a_1 and a_3 do not change their assignments. Accordingly, a_4 could directly get the opt^* for its subproblem from the cache and backtracks since re-exploring is unnecessary.

We next illustrate how OCS works when applied to TreeBB. There are two modifications to the original for implementing the scheme. First, before a_i sends a BACKTRACK message (line 22), it stores the current optimal cost opt^* along with the corresponding $context_i$ into its cache. Second, before exploring its subproblem (line 10), a_i looks up whether the current *context* is already in the cache. If it is, a_i sets the search domain $\tilde{D}_i^c$ to *null* and sends a BACKTRACK message with the opt^* in the cache to its parent. Unfortunately, such a scheme could lead to the incompleteness, which will be illustrated in detail in Subsection 3.1.

3 Proposed Method

In this section, we present a new caching scheme, named Retention Scheme (RS). We begin with the motivation of our work and then present the details. Finally, we give two cache replacement schemes.

3.1 Motivation

Actually, in OCS, agent a_i caches the optimal cost opt^* under a given ub_i. However, when pruning happens with ub_i, agent a_i could carry out an incomplete exploration, and caches the opt^* and its $context_i$. Clearly, such a cost cannot guarantee to be optimal since its search subspace is not exploited thoroughly yet. Considering a condition where pruning is not carried out, a_i could find a better opt^* by exploring the search space pruned before. Unfortunately, such a condition could happen since ub_i calculated by Eq.(1) may increase with the different assignments of $Anc(a_i)$. Note that, in the same $context_i$, the upper bound for pruning may be different since $Sep(a_i)$ is only a subset of $Anc(a_i)$. As a result, if OCS directly uses the opt^* obtained from a pruned search space from its cache, the completeness of the algorithm can not be guaranteed.

To illustrate the issue, based on TreeBB, we take Fig.1 as an example. Here, we mainly focus on agent a_4 and compare the difference when OCS is applied. Assume that the upper bound for $x_1 = 0$ is 20, thus a_4 could receive a CPA message containing a current partial assignment $Cpa_4 = \{(x_1, 0), (x_2, 0), (x_3, 0)\}$ and an upper bound $ub_4 = 6$ as illustrated in Fig.2(a). Next, since $lb_4(0) = 7$ calculated by Eq.(2) is greater than its $ub_4 = 6$, a_4 prunes the corresponding search space and sets the related cost to infinity, i.e., $opt_4^5(0) = opt_4^6(0) = \infty$. Subsequently, a_4 chooses its next feasible assignment $x_4 = 1$, and stores the best costs from its children into $opt_4^5(1)$ and $opt_4^6(1)$, respectively. Then, the opt^* is calculated according to Eq.(3) and caches together with the corresponding $context_4$ as $\langle \{(x_1, 0), (x_3, 0)\}, 14 \rangle$, shown in

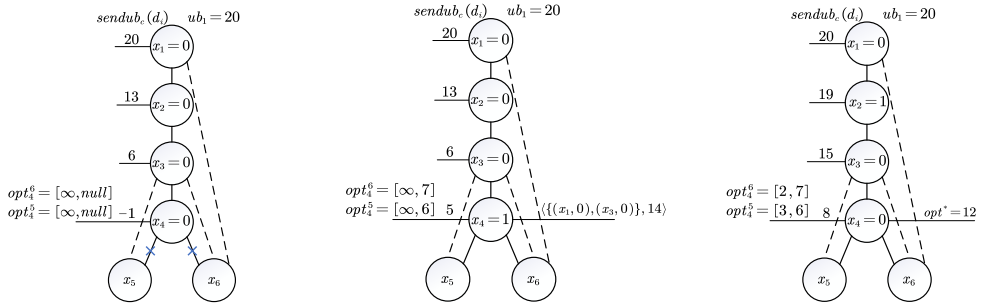


Figure 2 An example of OCS-based algorithm to show its incompleteness.

Fig.2(b). Now, if a_4 receives a new CPA message containing $Cpa_4 = \{(x_1, 0), (x_2, 1), (x_3, 0)\}$, in OCS, a_4 could directly use the historical result in the cache since the $context_4$ is identical even in difference Cpa_4. However, the better cost for the current Cpa_4 is 12 which is depicted in Fig.2(c), since a_4 explores the search space pruned before under the larger $ub_4 = 15$. As a result, the completeness could be impaired when OCS is applied.

Additionally, in OCS, the cache is accessed only when the current $context_i$ is identical to the items that has already stored in the information unit I. In the worst cases, the cache does not work in synchronous algorithms if $|Anc(a_i)| = |Sep(a_i)|$, as the traversal for the combinations of $context_i$ is ordered. As a result, the next $context_i$ is impossible to be identical to the cached items. However, for each child a_c, its corresponding $context_c$ is only the subset of that of its parent $context_i$, which means the cached costs may also be valid under a different $context_i$ for a_c. Taking a_4 in Fig.2 as an example, the $opt_4^5(1) = 6$ is valid as long as a_3 does not change its assignment, regardless of the assignment of a_1. Therefore, the OCS makes little use of the historical results and leads to a low cache utilization.

3.2 Retention Scheme

In order to solve the issues mentioned above, we propose the Retention Scheme (RS) to ensure the completeness and improve the cache utilization.

For the first issue, given a $context_i$ we additionally record the current upper bounds a_i received into the information unit I for judging whether the cached item is reliable. Specifically, if a newly received ub_i is greater than the stored one under the same $context_i$, the cached opt^* could be unreliable as pointed out in Subsection 3.1, and re-exploration should be carried out. Specially, if the current opt^* is obtained by exhausting the search space of its children, such opt^* is sure to be reliable. Here, we denote such opt^* as the $realcost$ under its corresponding $context_i$. Obviously, if a_i is the leaf agent, the opt^* is the $realcost$. Besides, if the opt^* is less than the received ub_i, the opt^* is the $realcost$, see Lemma 1. Thus, once the current opt^* is detected to be the $realcost$, we set the stored ub_i to be ∞ to avoid unnecessary re-exploration. So far, we have already guaranteed the completeness since only the $realcost$ can be obtained for use.

Next, we illustrate how we solve the second issue. Here, we adopt a fine-grained cache strategy by splitting the $context_i$ in OCS into $context_c$ for $\forall a_c \in C(a_i)$, where $context_c$ only includes the assignments of $Sep(a_c)$ for each a_c. Eq.(4) gives the relationship between $Sep(a_c)$ and $Sep(a_i)$.

$$\bigcup_{a_c \in C(a_i)} Sep\,(a_c) \setminus \{a_i\} \;\subseteq Sep\,(a_i) \tag{4}$$

So, different from OCS, we propose to use I to only store the information map related to its children, i.e., $\langle context_c, \langle opt_c, sub_c \rangle \rangle$, where opt_c and sub_c is the best cost reported by its child a_c and the corresponding upper bound for each child under the current $context_c$, respectively.

Algorithm 2 Retention Scheme for a_i.

When `Initialization`**:**
1 allocate memory space k for each child by Eq.(5)
Function `InitVariable` ()**:**
2 **foreach** $a_c \in C(a_i)$ **do**
3 **foreach** $d_i \in D_i$ **do**
4 $context_c \leftarrow Cpa_i(Sep(a_c)) \cup \{(x_i, d_i)\}$
5 **if** $IUCache_i^c\,(context_c) \neq null$ **then**
6 $\{opt_i^c(d_i), sub_i^c(d_i)\} \leftarrow IUCache_i^c\,(context_c)$
7 **else**
8 $opt_i^c(d_i) \leftarrow null, sub_i^c(d_i) \leftarrow 0$
9 $\tilde{D}_i^c \leftarrow \{d_i | d_i \in D_i, sub_i^c(d_i) \neq \infty\}$
Function `NextFeasibleAssignment` (a_c)**:**
10 $d_i \leftarrow$ the first element in $\tilde{D}_i^c$
11 **while** $(d_i \neq null \wedge lb_i\,(d_i) \geqslant ub_i) \vee (d_i \neq null \wedge sendub_c(d_i) \leqslant sub_i^c(d_i))$ **do**
12 **if** $lb_i\,(d_i) \geqslant ub_i$ **then**
13 $\{opt_i^c(d_i), sub_i^c(d_i)\} \leftarrow \{\infty, 0\}$
14 $\tilde{D}_i^c \leftarrow \tilde{D}_i^c \backslash \{d_i\}$
15 $d_i \leftarrow$ the first element in $\tilde{D}_i^c$
16 **return** d_i
Function `Backtrack` ()**:**
17 `FillIUCache` ()
18 $opt^* \leftarrow \min_{d_i \in D_i} lb_i\,(d_i)$
19 **if** opt^* *is a realcost* **then**
20 `send BACKTRACK` (opt^*, ∞) to $P(a_i)$
21 **else**
22 `send BACKTRACK` (opt^*, ub_i) to $P(a_i)$
Function `FillIUCache` ()**:**
23 **foreach** $a_c \in C(a_i)$ **do**
24 **if** $IUCache_i^c\,(context_c) \neq null$ **then**
25 **foreach** $d_i \in D_i$ **do**
26 **if** $sub_i^c(d_i) > 0$ **then**
27 $IUCache_i^c\,(context_c) \leftarrow \{opt_i^c\,(d_i)\,, sub_i^c\,(d_i)\}$
28 **else if** $mem_i^c > 0$ **then**
29 $IUCache_i^c\,(context_c) \leftarrow \{opt_i^c\,(d_i)\,, sub_i^c\,(d_i)\}\,, \forall d_i \in D_i$
30 $mem_i^c \leftarrow mem_i^c - |D_i|$
31 **else**
32 call certain page replacement procedure

Besides, some modifications should be done to implement the strategy appropriately. Specifically, each agent a_i additionally attaches the upper bound to the BACKTRACK message to its parent agent and maintains a list $sub_i^c(d_i), \forall d_i \in D_i, a_c \in C(a_i)$ to record the upper bound sent from a_c and update it if needed. Moreover, for each child a_c, a_i also maintains some additional data structures, which includes an $IUCache_i^c$ to store all information units and a mem_i^c to record the remaining memory space.

Next, we will detail how to implement the RS in TreeBB. Algorithm 2 presents the sketch of RS, and we only describe the difference from TreeBB. Specifically, each agent a_i starts by allocating memory for each child. Here, we set the maximum memory space with a parameter k for each agent, which means a_i can cache $|D_i|^k$ information units I at most and allocate the memory for each child a_c according to Eq.(5) if a_i has more than one child (line 1). The maximum memory function $|D_i|^k$ is widely used in some memory-bounded algorithms, and other functions (e.g., a constant value or $\rho * |D|^{|Sep(a_i)|}$ where $\rho \in (0, 1]$) can also be adopted.

As can be seen from Eq.(5), agent a_i allocates as much memory space as possible for a_c with small $|Sep(a_c)|$, but the initialized memory space mem_i^c should not exceed the maximum memory requirement, i.e.,$|D_i|^{|Sep(a_c)|}$. It is based on an intuitive idea that agent a_i with small $|Sep(a_c)|$ has fewer combinations for $context_c$, which could lead to a higher cache-hit rate.

$$mem_i^c \leftarrow \min\left(\frac{\left(\sum_{a_{c'}\in C(a_i)}|Sep(a_{c'})|-|Sep(a_c)|\right)*|D_i|^k}{(|C(a_i)|-1)*\sum_{a_{c'}\in C(a_i)}|Sep(a_{c'})|}, |D_i|^{|Sep(a_c)|}\right), \forall a_c \in C(a_i) \quad (5)$$

Next, the execution phase of TreeBB starts. When a_i receives a CPA message from its parent, for each child a_c, it judges whether the current $context_c$ is already in the cache (line 2–4). Specifically, if it hits, a_i obtains the corresponding historical results in the information unit cache $IUCache_i^c$ and initializes opt_i^c and sub_i^c (line 5–6). Otherwise, a_i sets $sub_i^c(d_i)$ to 0 and explores its child a_c when $sendub_c(d_i) > 0$ (line 7–8). Then, when the obtained $sub_i^c(d_i)$ is ∞, which means a $realcost$ for d_i has already been obtained, then d_i is removed from $\tilde{D}_i^c$ (line 9). After that, when a_i selects the next feasible assignment to explore for child a_c, our RS gives an additional pruning judgement (line 11–15). That is, a value d_i is removed from the search domain $\tilde{D}_i^c$ if a new $sendub_c(d_i) \leqslant sub_i^c(d_i)$ since a_i still cannot get a better cost under the $sendub_c(d_i)$.

Before a_i sends a BACKTRACK message, for each child a_c, it stores the current optimal cost $opt_i^c(d_i)$ and upper bound $sub_i^c(d_i)$ with the corresponding $context_c$ into its cache (line 17). Specifically, if the $context_c$ already exists in $IUCache_i^c$, a_i updates the cached results directly (line 23–27). Here, to make the cached results more effective, a_i performs the update only when $sub_i^c(d_i) > 0$. If the $context_c$ is not in the $IUCache_i^c$ and the remaining memory space mem_i^c is greater than 0, a_i stores the corresponding results as an information unit I and reduces the remaining memory space mem_i^c (line 28-30). If a_i does not have enough cache capacity to store a new information unit, it will perform certain cache replacement procedure (line 31–32). Then, if the current optimal cost opt^* is identified as the $realcost$, a_i replaces the upper bound ub_i with infinity in BACKTRACK messages to avoid unnecessary re-exploration (line 19–20). Otherwise, a_i needs to report the current upper bound (line 21–22).

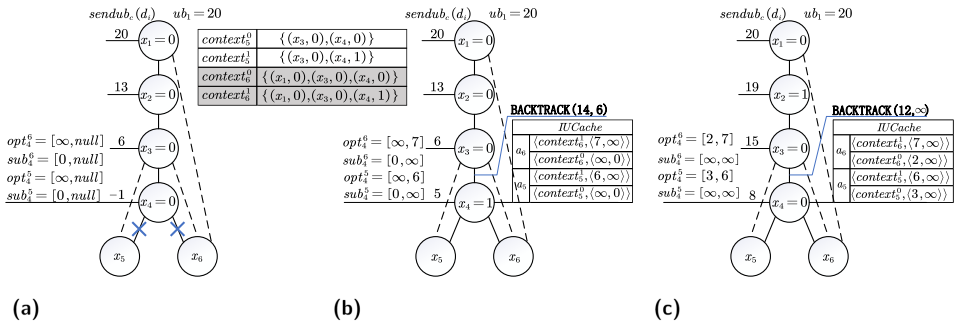


Figure 3 An example of RS-based algorithm to show its completeness.

To illustrate the effect of RS, we take Fig.1 as an example. Here, we mainly focus on agent a_4 and points out the difference when RS is applied based on Fig.3. Similar to the example in Fig.2(a), we assume the upper bound for $x_1 = 0$ is 20, a_4 receives the same CPA message and prunes the search space corresponding to 0, as illustrated in Fig.3(a). However, different from OCS, the RS additionally constructs sub_4^6 and sub_4^5 and set $sub_4^5(0) = sub_4^6(0) = 0$ (line 12–13). Then, a_4 explores its next feasible assignment $x_4 = 1$ and stores the best costs and

upper bound sent from its children, respectively. Note that, a_4 sets $sub_4^5(1) = sub_4^6(1) = \infty$ since a_5 and a_6 are leaf agents and the *realcosts* ($opt_4^5(1) = 6$ and $opt_4^6(1) = 7$) are reported (line 19–20). After that, a_4 caches these results together with the corresponding *contexts* (shown in the table) as information units, and sends a BACKTRACK message including $opt^* = 14$ and $ub_i = 6$ to its parent a_3, shown in Fig.3(b). Next, when a_4 receives a new CPA message containing $Cpa_4 = \{(x_1, 0), (x_2, 1), (x_3, 0)\}$ like the example in Fig.2(c), different from OCS which just uses the cached cost and sends a BACKTRACK message with the suboptimal cost, a_4 re-explores the search space corresponding to $x_4 = 0$ since $sendub_c(0) = 8$ is greater than $sub_4^5(0) = sub_4^6(0) = 0$ (line 11), and for $x_4 = 1$, it directly reuses the *realcost*s stored in cache before. Then, a_4 obtains the *realcosts* for its children by exhausting the search space and updates *IUCache* accordingly (line 17, 23-27), shown as Fig.3(c). At last, a_4 calculates $opt^* = 12$ which is a *realcost* and sends it with an infinite upper bound through a BACKTRACK message to its parent a_3 (line 19–20). It can be seen that the RS successfully guarantees the completeness and brings higher cache utilization compared to OCS.

3.3 Cache Replacement Scheme

To better make use of the cached items with limited memory, a cache replacement scheme is necessary since such a good scheme can often bring higher cache utilization and improve the performance of algorithms. Yeoh et al [30] proposed three cache replacement schemes for asynchronous algorithms, including MaxPriority, MaxEffort and MaxUtility Scheme. All these schemes allow each agent to make decisions according to heuristics constructed by reordering the cached information units. However, none of them can be applied to synchronous algorithms directly due to the fact that the valid information to cache is actually different between synchronous and asynchronous algorithm. Besides, for synchronous algorithms, there is no other relevant research on cache replacement schemes reported before. Based on the background, we introduce two feasible cache replacement schemes including UB and SYS to match our RS for synchronous algorithms.

Specifically, we construct a heuristic scheme named UB in which a_i sorts the cached information units according to sub_c, which allows new information units to preempt the unit with the smallest sub_c in the cache. It is because the larger the sub_c, the more likely the opt_c regarding it is the *realcost*.

In addition, we propose another cache replacement scheme named SYS by carrying out the FIFO (First-In-First-Out) scheme only when a specific agent changes its value. It gives full play to the advantages of the FIFO scheme in synchronous algorithms, and avoids the disadvantages caused by frequent cache replacement. Before introducing the SYS scheme, we first give some notations as follows.

- $H(a_i)$ is the height of a_i on the pseudo tree. In particular, the height of root agent is 0.
- $SepQ_i^c$ is a queue containing all the $Sep(a_c)$ agents for $a_c \in C(a_i)$, which is sorted by the height in a reverse order, i.e., $H\left(SepQ_i^c\left(m\right)\right) > H\left(SepQ_i^c\left(m+1\right)\right)$ where m is the index of $SepQ_i^c$. Taking Fig.1(c) as an example, $SepQ_4^6 = \{a_4, a_3, a_1\}$.

In synchronous algorithms, each agent changes its assignment orderly according to its height on a pseudo tree. That is, the root agent changes its value far less than the leaf agents, which also means an agent with larger index in $SepQ_i^c$ changes its assignment less frequently. Hence, it seems that the traditional FIFO scheme is suitable for synchronous algorithms. Consider a_4 in Fig.1(c), when a_1 changes to its next assignment, i.e., $x_4 = 1$, the

items with $context_6 = \{(x_1, 0), (x_3, d_3), (x_4, d_4)\}, \forall d_3 \in D_3, d_4 \in D_4$ cached in information unit cache $IUCache$ actually expire, since the next $context_6$ is no longer similar to that a_4 cached before for a_6. Therefore, if a_4 adopts the FIFO scheme, it only needs $|D_3||D_4|$ memory for a_6 to cache the reported results. However, naively using the FIFO scheme might bring some issues in some situations. Still take a_4 in Fig.1(c) as an example and assume the allocated memory mem_4^5 for its child a_5 is limited to $|D_4|$. If a_4 directly uses the FIFO scheme, it starts to replace the cached items under $x_3 = 0$ once a_3 changes to 1. However, it is clear that when a_2 changes to its next value, the items a_4 cached under $x_3 = 0$ is still valid, but they have already been replaced by that under $x_3 = 1$. As a result, according to the FIFO scheme, a_4 may replace the cached items frequently, leading to the fact that items always are unused before replaced. Therefore, the SYS scheme aims to overcome the issue by finding the first agent a_j satisfying Eq.(6) for each child a_c and then discarding the search results directly if $a_j = null$ or the assignment of a_j in the current $context_c$ is consistent with that in the cached items. Otherwise, a_i utilizes the FIFO scheme to replace the cached items.

$$a_j = SepQ_i^c(m) \qquad s.t.\ P(SepQ_i^c(m-1)) \neq SepQ_i^c(m), m \geqslant \lfloor \log_{|D_i|} mem_i^c \rfloor \tag{6}$$

Still take Fig.1(c) as an example and assume the initialized memory space $mem_4^5 = mem_4^6 = |D_4|$. Different from that in the FIFO scheme where a_4 always replaces the new $context_i$ for the cached ones when the $IUCache$ is full, in SYS a_4 does not replace the cached items and directly discards the current search results for a_5 since $a_j = null$. Similarly, for child a_6, a_4 does not replace either until $a_j(a_1)$ changes its assignment.

4 Theoretical Results

In this section, we prove the completeness of the Retention Scheme (RS), and give the complexity analysis of the proposed method.

▶ **Lemma 1.** *The optimal cost opt^* reported by a_i is the realcost if $opt^* < ub_i$.*

Proof. Assuming that the opt^* reported by a_i is not the *realcost*, there must exist a *realcost*, called $cost^*$, which satisfies $cost^* < opt^*$. Therefore, the $cost^*$ must have been pruned since only the current best cost opt^* found so far is reported. However, in tree-based complete synchronous search algorithms, only when $cost^* \geqslant ub_i$, pruning is carried out, which leads to a contradiction to the condition $opt^* < ub_i$ in lemma 1. Thus, Lemma 1 is proved. ◀

▶ **Lemma 2.** *Given a same $context_i$, assume that a_i gets a suboptimal cost under ub_i. If the newly-received upper bound ub_i' for a_i is less than ub_i, i.e.,$ub_i' < ub_i$, a_i still gets a suboptimal cost.*

Proof. According to Lemma 1, the suboptimal cost opt^* should satisfy $opt^* \geqslant ub_i$ owing to opt^* is not a *realcost*. Therefore, we have $opt^* \geqslant ub_i > ub_i'$, which indicates that a *realcost* can not be obtained by exploring with ub_i' either. Thus, Lemma 2 is proved. ◀

▶ **Theorem 3.** *The RS is complete.*

Proof. According to Lemma 1 and Lemma 2, the RS only utilizes the *realcosts* and the additional pruning judgement it adopts does not affect the exploration for such *realcosts*. Thus, the RS is complete. ◀

It is worth noting that the completeness of the RS will not be affected by the arity of constraint functions as it is related only to the cached results of historical exploration. Besides, the cache replacement schemes do not hurt the completeness of the RS as they only determine which results explored should be stored in the cache.

4.1 Complexity

When applied to existing tree-based complete synchronous search algorithms, the RS does not introduce any new messages, and it only attaches the current upper bound to a BACKTRACK message which only requires linear memory.

For each agent a_i, assuming it has enough memory to store all the search results, it requires $|D_i|^{|Sep(a_c)|}$ memory to cache the items for its child $a_c \in C(a_i)$. Therefore, the overall consumption is $O\left(|C\left(a_i\right)||D_i|^{Sep}\right)$, where $Sep = \max_{a_c \in C(a_i)} |Sep(a_c)|$. On the other hand, for each agent a_i, it traverses the information unit cache $IUCache$ to map for its child $a_c \in C(a_i)$, which requires a linear time complexity. So, the entire time complexity is $O(|C(a_i)|)$.

Besides, for the cache replacement schemes, UB requires $O\left(|C(a_i)||D_i|^{Sep} \log |D_i|^{Sep}\right)$ time complexity to perform quick sort for information units, while for SYS it only needs $O(1)$ to calculate the specific agent by Eq.(6).

5 Empirical Evaluation

In this section, we apply the RS to state-of-the-art tree-based complete synchronous search algorithms, and compare them against the originals. Then, when the memory is limited, we investigate the effects of different cache replacement schemes on the RS-based algorithms.

5.1 Experimental Configuration

In order to evaluate the effect of RS, we apply it to tree-based complete synchronous algorithms including TreeBB, PT-FB, HS-CAI and PT-ISBB (a variant of PT-ISABB [8] in DCOP settings), and name the corresponding RS-based version as TreeBB+RS, PT-FB+RS, HS-CAI+RS and PT-ISBB+RS. Specially, to ensure the completeness of OCS when applied to TreeBB for fairness, we also append the upper bound to the information unit and name it as TreeBB+OCS. Besides, we uniformly use the SYS scheme to manage memory when the cache is full. In our experiments, we will compare these RS-based algorithms with their originals on random DCOPs and weighted graph coloring problems. For random DCOPs, we set the graph density to 0.2, the domain size to 3 and vary the agent number from 16 to 28 as the sparse configuration, and the graph density to 0.5, the domain size to 3 and the number of agents varying from 14 to 24 as the dense configuration. For weighted graph coloring problems, similar to the sparse configuration in random DCOPs, we set the graph density to 0.2, the domain size to 3 and vary the agent number from 16 to 28. Besides, we choose $k = 4$ and $k = 8$ as the low and high memory budget for HS-CAI, PT-ISBB and RS, respectively. Furthermore, we compare the performance of different cache replacement schemes including FIFO (First In First Out), FILO (First In Last Out), LRU (Least Recently Used) and LFU (Least Frequently Used), UB, SYS and ORI (an original scheme which discards the next results when the cache is full.) when they are applied to TreeBB+RS, on the dense configuration for random DCOPs and low memory budget. We use the number of messages (Msgs) to measure the communication overheads, and the NCLOs [21] to measure the hardware-independent runtime where the logical operations in the inference and the search are accesses to utilities and constraint checks, respectively. For each experiment, we randomly generate 50 instances with the integer constraint costs in the range of 0 to 100, and report the average over all instances. The experiments are conducted on an i7-7820x workstation with 32GB of memory, and we set the timeout to 30 minutes for each algorithm.

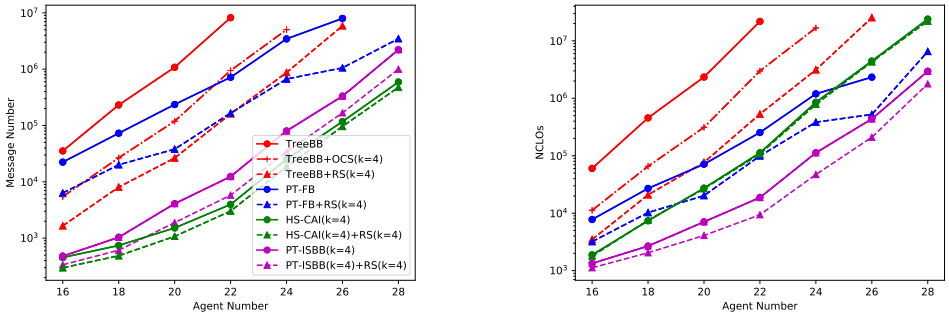
5.2 Experimental Results

Table 1 gives the detailed results of the improvement of the RS-based algorithms over their originals including both memory budget $k = 4$ and $k = 8$, where the numbers greater than zero are shown in bold. It can be seen that the RS-based algorithms outperform the originals on both metrics under both low and high memory budget, i.e., $k = 4$ and $k = 8$. Besides, the improvement under $k = 8$ is slightly higher than that under $k = 4$ in most cases since with larger memory, the RS-based algorithms can cache more *realcosts* for reuse. However, since the number of valid *realcost* is limited, such an improvement fades away. Next, Fig.4 to Fig.6 give the experimental results for the sparse random DCOPs, dense random DCOPs and weighted graph coloring problems with $k = 4$, and we do not present the results with $k = 8$ since they are of the same trend as that with $k = 4$.

Table 1 The improvement of the RS-based algorithms over their respective originals.

Configuration	k	TreeBB+OCS		TreeBB+RS		PTFB+RS		PT-ISBB+RS		HS-CAI+RS	
		Msgs(%)	NCLOs(%)	Msgs(%)	NCLOs(%)	Msgs(%)	NCLOs(%)	Msgs(%)	NCLOs(%)	Msgs(%)	NCLOs(%)
Sparse DCOPs	4	**84∼88**	**81∼86**	**95∼98**	**94∼97**	**71∼86**	**58∼77**	**29∼57**	**16∼52**	**18∼34**	**3∼8**
	8	**85∼89**	**81∼87**	**96∼99**	**95∼98**	**72∼87**	**59∼78**	**30∼49**	**1∼17**	**20∼40**	0∼5
Dense DCOPs	4	**45∼50**	**40∼44**	**72∼76**	**65∼71**	**19∼40**	**13∼42**	**9∼23**	**9∼34**	**2∼8**	0∼2
	8	**46∼50**	**41∼45**	**74∼80**	**69∼75**	**20∼42**	**14∼43**	**8∼15**	0∼12	**2∼15**	0∼6
weighted graph coloring	4	**39∼50**	**45∼49**	**41∼68**	**49∼60**	**39∼52**	**25∼47**	**34∼51**	**19∼40**	**21∼45**	**1∼9**
	8	**40∼54**	**46∼53**	**41∼71**	**49∼61**	**40∼53**	**26∼47**	**34∼48**	**1∼13**	**35∼50**	0∼5

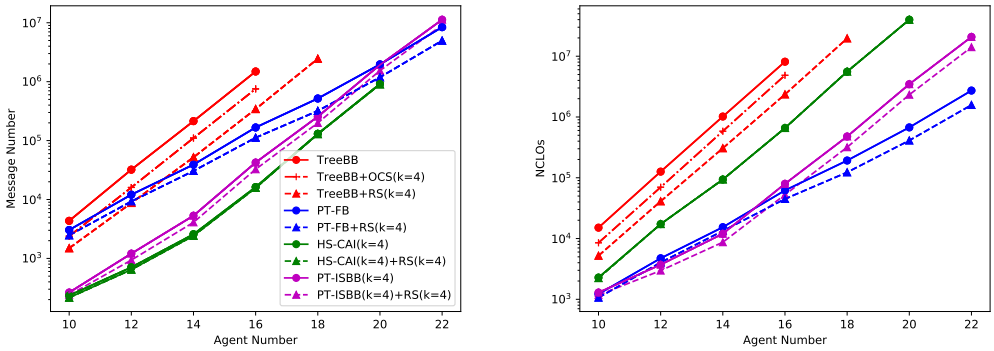
Fig.4 presents the experimental results for the number of messages (a) and NCLOs (b) under different agent numbers on the sparse configuration for random DCOPs, and the corresponding improvement over the originals is displayed in the first row of Table 1. It can be seen that the RS-based algorithms exhibit a great advantage on both metrics over their originals. Especially for TreeBB and PT-FB, the RS improve them more. It is due to the fact that compared to PT-ISBB and HS-CAI, TreeBB and PT-FB perform less pruning due to their less tight lower bounds. Therefore, the cached results could be more likely to be the *realcost* and reused more frequently. Besides, it can be seen from Table 1 that TreeBB+RS is superior to TreeBB+OCS by about 10% on the the sparse problems, which indicates the fine-grained caching could indeed boost the cache-hit rate.



(a) Number of Messages. **(b)** NCLOs.

Figure 4 Performance comparison under different agents on sparse DCOPs.

We can also see from Fig.4 that with the help of the RS, TreeBB and PT-FB solved larger problems, scaling up to the problems with 26 agents and with 28 agents, respectively.



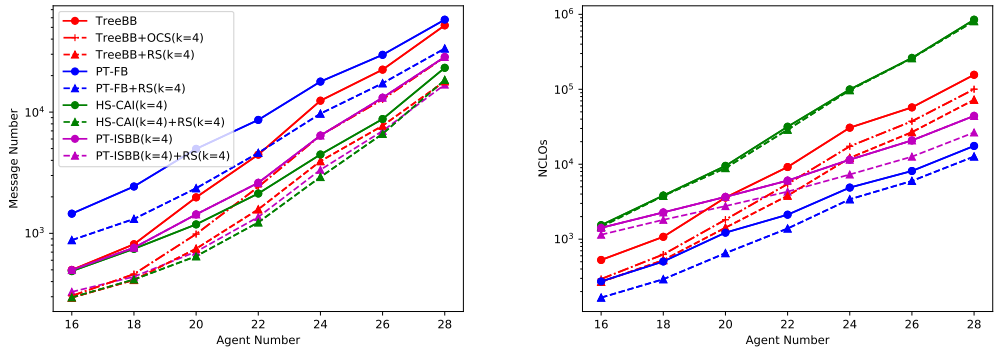
(a) Number of Messages. **(b)** NCLOs.

Figure 5 Performance comparison under different agents on dense DCOPs.

Fig.5 presents the experimental results under different agent numbers on the dense configuration for random DCOPs, and the third row of Table 1 shows the corresponding improvement over the originals. It can be seen that the RS-based algorithms also perform better than their originals in terms of both the number of messages and NCLOs. However, the improvement of RS decreases compared to that on the sparse configuration. It is because $|Sep(a_c)|$ for each agent a_i is larger on the dense configuration. Such large $Sep(a_c)$ will lead to more combinations for the $context_c$ and thus bring down the cache-hit rate. In addition, a_i may not be able to cache all the $context_c$ for large $Sep(a_c)$ with its limited memory, which would lead to the fact that the historical results for some $context_c$ would never be reused. Moreover, it can be seen from Table 1 that the outperformance of TreeBB+RS over TreeBB+OCS increases from about 10% on sparse problems to about 25% on dense problems. It is because under larger $|Sep(a_i)|$, it is more difficult for TreeBB+OCS to match a new $context_i$ to the cached items, while for TreeBB+RS it is much easier since the latter only needs to match the subset of $|Sep(a_i)|$ for its child.

Fig.6 presents the experimental results under different agent numbers on weighted graph coloring problems, and the corresponding improvement over the originals can be found in the fifth row of Table 1. It can be seen that our RS can greatly improve the performance of the originals on both metrics, which is similar to the results on random DCOPs. It is worth noting that, when $k = 4$, the performance of TreeBB+RS is better than all other competitors without RS on the number of messages, which verifies that a cache for reusing historical results has a more significant role than providing a tighter lower bound on weighted graph coloring problems.

Table 2 presents the experimental results for TreeBB+RS with different cache replacement schemes. We can find that our proposed cache replacement schemes perform better than other competitors in most cases. It is because ORI, FIFO, FILO, LRU, LFU are all model-free cache replacement schemes which ignore the structure of the problems, while our proposed methods consider the related information such as the cached upper bounds (UB) and the characteristics of synchronous algorithms (SYS). Besides, the SYS is better than UB in this case, as the SYS aims at the characteristics of sequential assignments in synchronous algorithms and adjusts itself by the initialized memory.



(a) Number of Messages. **(b)** NCLOs.

Figure 6 Performance comparison under different agents on weighted graph coloring problems.

Table 2 Performance comparison under different cache replacement schemes on dense DCOPs.

Scheme	Metrics	Agent Number					Metrics	Agent Number				
		10	12	14	16	18		10	12	14	16	18
ORI	Msgs	1463	10841	84208	569718	3890905	NCLOs	5133	49443	449313	3626525	28718461
FIFO		1489	8717	52018	342777	2449699		5214	40978	306201	2353110	19653448
FILO		1466	10748	83715	566028	3825017		5135	48909	445818	3597353	28243741
LRU		1480	8705	52057	342724	2449808		5215	40971	306211	2352837	19655744
LFU		1466	10744	83521	565403	3840109		5137	48864	444652	3592756	28323712
UB		1481	8705	52003	342767	2449665		5215	40972	306185	2353149	19653238
SYS		**1433**	**8619**	**51412**	**340792**	**2437553**		**5002**	**40666**	**303945**	**2347033**	**19578876**

6 Conclusion

To overcome the shortcomings of OCS, we propose a new caching scheme named RS, which can be deployed to all tree-based complete synchronous search algorithms with minor modifications. It ensures the completeness of the algorithms by appending the upper bound to the information unit and further improves the cache utilization by adopting a fine-grained cache information unit. Besides, we also propose two cache replacement schemes UB and SYS to improve the performance of RS when the memory is limited. Finally, we give a theoretical proof for the completeness of RS, and our empirical evaluation shows the superiority of the RS-based algorithms over their originals and the advantage of our cache replacement schemes over the traditional ones. In the future, we will devote to further optimizing the cache information units and designing more appropriate cache replacement schemes for RS.

References

1 James Atlas, Matt Warner, and Keith Decker. A memory bounded hybrid approach to distributed constraint optimization. In *Proc. 10th International Workshop on Distributed Constraint Reasoning*, pages 37–51, 2008.

2 Ismel Brito and Pedro Meseguer. Improving DPOP with function filtering. In *AAMAS*, volume 1435, pages 141–148, 2010.

3 Anton Chechetka and Katia Sycara. No-commitment branch and bound search for distributed constraint optimization. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1427–1429, 2006.

4 Anton Chechetka and Katia P Sycara. An Any-space Algorithm for Distributed Constraint Optimization. In *AAAI Spring Symposium: Distributed Plan and Schedule Management*, pages 33–40, 2006.

5 Dingding Chen, Yanchen Deng, Ziyu Chen, Wenxing Zhang, and Zhongshi He. HS-CAI: A hybrid DCOP algorithm via combining search with context-based inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7087–7094, 2020.

6 Ziyu Chen, Wenxin Zhang, Yanchen Deng, Dingding Chen, and Qiang Li. RMB-DPOP: Refining MB-DPOP by Reducing Redundant Inference. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pages 249–257, 2020.

7 Rina Dechter, David Cohen, et al. *Constraint processing*. Morgan Kaufmann, 2003.

8 Yanchen Deng, Ziyu Chen, Dingding Chen, Xingqiong Jiang, and Qiang Li. PT-ISABB: A Hybrid Tree-based Complete Algorithm to Solve Asymmetric Distributed Constraint Optimization Problems. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1506–1514, 2019.

9 Alessandro Farinelli, Alex Rogers, and Nick R Jennings. Agent-based decentralised coordination for sensor networks using the max-sum algorithm. *Autonomous agents and multi-agent systems*, 28(3):337–380, 2014.

10 Alessandro Farinelli, Alex Rogers, Adrian Petcu, and Nicholas R Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 639–646, 2008.

11 Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698, 2018.

12 Ferdinando Fioretto, William Yeoh, Enrico Pontelli, Ye Ma, and Satishkumar J Ranade. A distributed constraint optimization (DCOP) approach to the economic dispatch with demand response. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 999–1007, 2017.

13 Eugene C Freuder and Michael J Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *IJCAI*, volume 85, pages 1076–1078. Citeseer, 1985.

14 Patricia Gutierrez and Pedro Meseguer. BnB-ADOPT+ with Several Soft Arc Consistency Levels. In *ECAI*, pages 67–72, 2010.

15 Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *International conference on principles and practice of constraint programming*, pages 222–236. Springer, 1997.

16 Omer Litov and Amnon Meisels. Forward bounding on pseudo-trees for DCOPs and ADCOPs. *Artificial Intelligence*, 252:83–99, 2017.

17 Rajiv T Maheswaran, Jonathan P Pearce, and Milind Tambe. A family of graphical-game-based algorithms for distributed constraint optimization problems. In *Coordination of large-scale multiagent systems*, pages 127–146. Springer, 2006.

18 Rajiv T Maheswaran, Milind Tambe, Emma Bowring, Jonathan P Pearce, and Pradeep Varakantham. Taking dcop to the real world: Efficient complete solutions for distributed multi-event scheduling. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 310–317, 2004.

19 Toshihiro Matsui, Hiroshi Matsuo, and Akira Iwata. Efficient Methods for Asynchronous Distributed Constraint Optimization Algorithm. In *Artificial Intelligence and Applications*, pages 727–732, 2005.

20 Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180, 2005.

21 Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence*, 193:186–216, 2012.

22 Duc Thien Nguyen, William Yeoh, Hoong Chuin Lau, and Roie Zivan. Distributed gibbs: A linear-space sampling-based DCOP algorithm. *Journal of Artificial Intelligence Research*, 64:705–748, 2019.

23 Brammert Ottens, Christos Dimitrakakis, and Boi Faltings. DUCT: An upper confidence bound approach to distributed constraint optimization problems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(5):1–27, 2017.

24 Adrian Petcu and Boi Faltings. DPOP: A scalable method for multiagent constraint optimization. In *IJCAI 05*, pages 266–271, 2005.

25 Adrian Petcu and Boi Faltings. ODPOP: An algorithm for open/distributed constraint optimization. In *AAAI*, volume 6, pages 703–708, 2006.

26 Adrian Petcu and Boi Faltings. MB-DPOP: A New Memory-Bounded Algorithm for Distributed Optimization. In *IJCAI*, pages 1452–1457, 2007.

27 Evan Sultanik, Pragnesh Jay Modi, and William C Regli. On Modeling Multiagent Task Scheduling as a Distributed Constraint Optimization Problem. In *IJCAI*, pages 1531–1536, 2007.

28 Meritxell Vinyals, Juan A Rodriguez-Aguilar, Jesús Cerquides, et al. Generalizing DPOP: Action-GDL, a new complete algorithm for DCOPs. In *AAMAS (2)*, pages 1239–1240, 2009.

29 William Yeoh, Ariel Felner, and Sven Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.

30 William Yeoh, Pradeep Varakantham, and Sven Koenig. Caching schemes for DCOP search algorithms. In *AAMAS (1)*, pages 609–616, 2009.

31 Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1-2):55–87, 2005.