Autonomous Agents and Multi-Agent Systems (2021) 35:24
https://doi.org/10.1007/s10458-021-09511-z

Utility distribution matters: enabling fast belief propagation for multi-agent optimization with dense local utility function

Yanchen Deng[1] · **Bo An**[1]

Accepted: 25 May 2021

Abstract
Belief propagation algorithms including Max-sum and its variants are important methods for multi-agent optimization. However, they face a significant scalability challenge as the computational overhead grows exponentially with respect to the arity of each utility function. To date, a number of acceleration algorithms for belief propagation algorithms were proposed. These algorithms maintain a lower bound on total utility and employ either a domain pruning technique or branch and bound to reduce the search space. However, these algorithms still suffer from low-quality bounds and the inability of filtering out suboptimal tied entries. In this paper, we first show that these issues are exacerbated and can considerably degenerate the performance of the state-of-the-art methods when dealing with the problems with dense utility functions, which widely exist in many real-world domains. Built on this observation, we then develop several novel acceleration algorithms that alleviate the effect of densely distributed local utility values from the perspectives of both bound quality and search space organization. Specifically, we build a search tree for each distinct local utility value to enable efficient branch and bound on tied entries and tighten a running lower bound to perform dynamic domain pruning. That is, we integrate both search and pruning to iteratively reduce the search space. Besides, we propose a discretization mechanism to offer a tradeoff between the reconstruction overhead and the pruning efficiency. Finally, a K-depth partial tree-sorting scheme with different sorting criteria is proposed to reduce the memory consumption. We demonstrate the superiorities of our algorithms over the state-of-the-art acceleration algorithms from both theoretical and experimental perspectives.

Keywords DCOP · Inference · Belief propagation · Max-sum · Domain pruning

This paper is an extension to our IJCAI paper [9]. Beside additional examples, experiments and proofs, we also present a K-depth partial tree-sorting scheme reducing the memory consumption by limiting the depth of search trees, which is not included in the IJCAI paper.

✉ Yanchen Deng
ycdeng@ntu.edu.sg

Bo An
boan@ntu.edu.sg

[1] School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

Published online: 02 June 2021

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) [32] are a fundamental model for multi-agent optimization and coordination, in which agents cooperatively find assignments to optimize a global objective. DCOPs have been successfully applied to model many real-world problems where information and controls are inherently distributed among multiple agents, such as distributed scheduling [15, 27], smart-grids [11] and radio frequency allocation [33].

Complete algorithms for DCOPs [6, 16, 28, 32, 34, 40, 41, 49] aim to find the optimal solution but incur exponential coordination overheads since solving DCOPs is NP-Hard. In contrast, incomplete algorithms [19, 30, 35–37, 53] trade the solution quality for smaller computational efforts and can scale up to large problems. Max-sum and its variants [4, 10, 43, 56] are popular incomplete algorithms built upon the Generalized Distributive Law (GDL) [1] and have been applied to many real-world domains [10, 22, 29]. However, these algorithms face a significant scalability challenge. In more detail, Max-sum implements belief propagation on a factor graph [26] by optimizing the sum of local utility functions and corresponding query messages. As a result, the computational effort grows exponentially with respect to the arity of each utility function, which prohibits Max-sum from scaling up to the problems with high-arity factors.

Therefore, a number of acceleration algorithms for belief propagation algorithms were proposed to improve their scalability and can be generally divided into BnB-based and sorting-based algorithms. BnB-based algorithms [5, 29, 46] construct an estimation for each partial assignment and employ branch and bound to reduce the search space. Differently, Generic Domain Pruning (GDP) [21] technique is a sorting-based approach that performs a one-shot pruning on a completely sorted local utility list.

However, a key issue of the existing methods is the bound quality. In more detail, BnB-based algorithms alphabetically exhaust the whole search space without using any *a priori* knowledge. As a result, the algorithms may not be able to find a high-quality lower bound promptly when utility functions are highly structured, leading to a poor acceleration performance. On the other hand, although GDP attempts to find a more efficient bound by sorting local utility values, the one-shot nature still cannot guarantee the quality of the lower bound (cf. Sect. 4).

The issue could be amplified when solving the problem with dense utility functions. Consider a utility function where the entries with high utility value are sparsely distributed (or, equivalently, the entries with low utility value are densely distributed). BnB-based algorithms may need to search a considerably large proportion of search space before finding a high-quality lower bound. Similarly, when the entries with high utility value are densely distributed, the pruned range returned by GDP may contain many entries whose utility value is no less than the one-shot lower bound. Even worse, GDP cannot prune at all if the entries have the same utility value. In fact, GDP relies on a linearly structured space (i.e., the sorted local utility list) to perform pruning and thus cannot discard the suboptimal tied entries in advance.

Unfortunately, densely distributed utility functions are ubiquitous in real-world scenarios. For example, in a NetRad system [22], multiple radars coordinate their scanning strategy to maximize the total utility of scanning weather phenomena. In such scenario, the utility of scanning a weather phenomenon does not necessarily grow linearly with respect to the scanning quality. Some phenomena may require high-quality observations and the entries with low utility value are very densely distributed in the corresponding utility

functions. While some phenomena require less joint efforts and the corresponding utility functions have many entries with high utility value.

Against the background, in this paper we aim to cope with dense local utility functions from the perspectives of both bound quality and search space organization and develop more efficient acceleration algorithms for Max-sum and its variants. Specifically, our main contributions are listed as follows.

- We propose a Generic Dynamic Domain Pruning (GD^{2}P) technique to ensure the bound quality. Instead of performing a one-shot pruning with the initial bound in GDP, we iteratively reduce the search space by continuously tightening a running lower bound. We further enhance GD^{2}P by merging the entries with the same utility value into a search tree to enable efficient branch and bound on tied entries. Therefore, the search space is collectively represented by a sorted array of search trees and the scheme is referred as Search Tree-based GD^{2}P (ST-GD^{2}P).
- We propose a discretization mechanism for ST-GD^{2}P to offer a tradeoff between the reconstruction overhead and the domain pruning efficiency. That is, we eliminate small search trees in ST-GD^{2}P by discretizing the utility range into slots according to a fixed step size and building a search tree for each slot. In this way, we can reduce both the preprocessing and reconstruction overheads when the utility functions are extremely dense.
- We introduce a K-depth Partial Tree-sorting Scheme (PTS) with different sorting criteria to reduce the memory consumption of ST-GD^{2}P. Given a utility function and a total ordering on the involved variables, the scheme builds a sorted array of search trees to represent the search space of the first K variables according to certain sorting criterion. For the remaining variables, the vanilla branch and bound is carried out to exhaust the search space. Finally, we introduce a built-in termination detection mechanism to allow domain pruning in PTS.
- We theoretically show that the proposed algorithms are correct and both GD^{2}P and ST-GD^{2}P outperform GDP in terms of pruning capability. Our empirical evaluations indicate that the proposed algorithms significantly outperform the state-of-the-art in various benchmarks.

The rest of the paper is organized as follows. We review related work in Sect. 2. Backgrounds and preliminaries can be found in Sect. 3. We motivate our research in Sect. 4. In Sect. 5, we detail the proposed GD^{2}P, ST-GD^{2}P and discretization mechanism. K-depth sorting scheme is presented in Sect. 6. We present empirical evaluations in Sect. 7 and conclude the paper in Sect. 8.

2 Related work

Over the past decade, many algorithms for DCOPs have been proposed and can be generally classified as complete and incomplete according to whether they guarantee to find the optimal solution. While search-based complete algorithms like SynchBB [16], ADOPT [32], AFB [13], BnB-ADOPT [51], ConFB [34], and PT-FB [28] perform distributed backtrack search to systematically explore the whole solution space, inference-based complete algorithms including DPOP [40], MB-DPOP [41], Action-GDL [49] and RMB-DPOP [6] perform dynamic programming to backup utility tables. Besides, some hybrid schemes like

ADOPT-BDP [2], DJAO [24] and HS-CAI [3] attempt to combine the advantages of both search and inference. However, due to the NP-hardness of solving a DCOP, complete algorithms incur exponential coordination overheads and cannot scale up to large problems.

Local search algorithms including DSA [53] and MGM [30] are typical incomplete algorithms, which iteratively optimize the solution via local moves. In those algorithms, agents keep exchanging self states (i.e., assignments or gains) with their neighbors, and determine whether to replace and how to replace their assignment based on the received states from their neighbors. Different algorithms adopt different assignment replacement strategies. For example, agents in DSA stochastically replace their assignment in every iteration, while only the agents who hold the maximum gain among their neighbors can replace their assignment in MGM. Recently, GDBA [36] adapts DBA [17], which is designed to solve Distributed Constraint Satisfaction Problems (DisCSPs) [52], to solve the general-valued DCOPs. To improve the quality of local convergence, KOPT [20] coordinates the decisions of all agents within the K-size coalition and converges to a K-optimal [38] state ensuring that the solution quality cannot be improved if K agents or fewer change their assignment.

Sampling-based techniques including DUCT [37] and D-Gibbs [35] are the emerging incomplete methods for DCOPs, which perform sequential sampling on a pseudo tree [12]. Given a context a, an agent in DUCT first constructs a confidence bound for each value k in its domain, which is an optimistic estimation of the optimal cost for its subtree under context a and value k, and samples the value with the lowest bound. However, the memory requirement per agent in DUCT is exponential in the number of agents, which makes it unsuitable for large real applications. Differently, D-Gibbs solves a DCOP by mapping it to a Maximum Likelihood Estimation (MLE) problem and using Gibbs sampling to find a solution, which requires a linear-space of memory. However, D-Gibbs requires all agents to be organized into a pseudo tree which usually leads to low concurrency in large-scale systems. Besides, D-Gibbs offers no quality guarantee before it converges, which is not suitable for safety critical applications such as disaster response, surveillance, etc.

Max-sum [10] is an important GDL-based incomplete algorithm which performs loopy belief propagation to accumulate utility values through the whole factor graph. Specifically, each agent in Max-sum maintains belief about the global utility for each possible assignment, and keeps updating its belief based on the messages received from its neighbors. To make a decision, an agent in Max-sum chooses the assignment with the highest utility under the current belief. Unfortunately, Max-sum only guarantees to converge in cycle-free problems which are very rare in realistic applications.

Bounded Max-sum [43] attempts to overcome the non-convergence of Max-sum by removing edges from a cyclic factor graph to make it acyclic. Specifically, the algorithm first relaxes a cyclic factor graph into a tree-structured graph by shrinking the binary dependencies between variable nodes and function nodes which have the least impacts on the solution quality to unary functions via a minimization operation. Then standard Max-sum is used to solve the relaxed problem and provides a bound on the approximation of the optimal solution. To provide a tighter approximation ratio, Improved Bounded Max-sum (IBMS) [44] concurrently solves a minimized and a maximized relaxed problem, and selects the best result to calculate the approximation ratio. Nonetheless, all BMS algorithms introduce errors in their relaxation phase. Thus, ED-IBMS [45] attempts to alleviate the pathology by decomposing binary dependencies into unary functions.

Different than removing edges in BMS algorithms, Max-sum_AD [56] makes a factor graph acyclic by strictly controlling the direction of message-passing. That is, instead of sending messages to every neighbor in Max-sum, nodes in Max-sum_AD only send

messages to ones ordered after them. And the message-passing direction is alternated after the algorithm converges. Further, Max-sum_ADVP [56] enhances Max-sum_AD by implementing a value propagation mechanism to enforce the cross phase convergence and monotonicity. However, the exploitative nature of Max-sum_ADVP restricts the solutions that are ultimately found. Therefore, a number of non-consecutive value propagation strategies were proposed in [4] to balance exploration and exploitation.

Damped Max-sum [7] provides an alternative way to increase the chances for convergence of belief propagation by decreasing the effect of cyclic information propagation. Instead of transforming factor graph or controlling message-passing direction, damped Max-sum manipulates the content of the messages sent from function nodes to variable nodes. That is, a message now is the weighted sum of new calculation performed in the current iteration and calculation performed in the previous iteration. Combining with anytime mechanism [55] or factor splitting [7], Damped Max-sum with a high damping factor outperforms all other versions of Max-sum, as well as local search algorithms.

Despite their significance in the field of distributed constraint reasoning, Max-sum and its variants face a huge scalability challenge. Specifically, these algorithms implement belief propagation by optimizing the sum of local utility functions and corresponding query messages. As a result, the computational effort for producing a message from a function node to a variable node grows exponentially with respect to the arity of the utility function, which prohibits Max-sum from scaling up to high-arity factors.

Therefore, a number of acceleration algorithms for belief propagation were proposed to improve their scalability and can be generally divided into BnB-based and sorting-based algorithms. BnB-MS [46] and BnB-FMS [29] are typical BnB-based algorithms which construct an estimation for each partial assignment and employ branch and bound to reduce the search space. Nevertheless, these algorithms compute estimations by either brute-force or domain-specific knowledge, which limits their generality. Recently, FDSP [5] was proposed to implement generic branch and bound by using dynamic-programming to construct domain-agnostic estimations.

On the other hand, sorting-based algorithms including G-FBP [23] and GDP [21] require (partially) sorted local utility list to perform acceleration. Specifically, G-FBP only sorts for top $cd^{\frac{n-1}{2}}$ values of the search space and presumes that the highest utility can be found in the ranges. Here, c is a constant, d is the maximal domain size and n is the arity of a utility function, respectively. However, the algorithm has to perform an exhaustive traverse when the assumption fails. In contrast, GDP constructs a completely sorted local utility list for each assignment of each variable. Then it uses the entry with the highest local utility value to compute a one-shot lower bound to prune suboptimal entries.

Orthogonally, Tarlow et al. [48] studied the problems with Tractable Higher Order Potentials (THOPs) and pointed out that the computational complexity of Max-sum on these problems can be reduced to polynomial time. For example, given a binary problem with cardinality factors where each variable can only take either 0 (off) or 1 (on) and the value of a utility function depends solely on the number of on variables, Max-sum can be efficiently implemented in $O(n \log n)$ per function node by using sorting and dynamic programming. For some specific types of THOPs, the belief propagation can even be implemented in linear time [14, 25, 42].

However, a major issue of THOP-based methods is the limited reprentational capability of THOPs, which disallows the use in general problems. In fact, since the value of a cardinality factor is determined only by the number of on variables, it fails to capture the complex utility structures such as preferences. Besides, applying THOP-based algorithms could be laborous: for each utility function, one needs to first identify its pattern and then design a specified

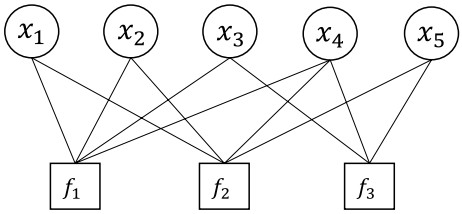


Fig. 1 A factor graph

algorithm for it. Therefore, in this work, we are interested in developing general acceleration algorithms for Max-sum.

3 Backgrounds

In this section, we review preliminaries including DCOPs, Max-sum, GDP and FDSP.

3.1 Distributed constraint optimization problems

A Distributed Constraint Optimization Problem (DCOP) [32] can be defined by a tuple $\langle A, X, D, F\rangle$ where $A = \{a_1, \ldots, a_p\}$ is the set of agents, $X = \{x_1, \ldots, x_q\}$ is the set of variables, $D = \{D_1, \ldots, D_q\}$ is the set of discrete domains and $F = \{f_1, \ldots, f_m\}$ is the set of utility functions. Each function $f_j : \mathbf{x_j} \rightarrow \mathbb{R}_{\geq 0}$ specifies a utility for each possible combination of involved variables $\mathbf{x_j} \subseteq X$. For the sake of simplicity, we assume that each agent controls a variable (i.e., $p = q$). The objective of a DCOP is to find an assignment for each variable to maximize the global utility. That is,

$$X^* = \arg\max_X \sum_{f_j \in F} f_j(\mathbf{x_j}).$$

3.2 Max-sum

Max-sum [10] is a GDL-based incomplete message-passing algorithm for DCOPs operating on a factor graph. A factor graph [26] is a bipartite graph representation of a DCOP, which consists of variable nodes representing variables and function nodes representing utility functions in the DCOP, respectively. Figure 1 presents a factor graph consisting of 3 function nodes and 5 variable nodes.

Starting from arbitrarily initialized messages, Max-sum implements belief propagation via query and response messages. A query message is sent from a variable node to a neighboring function node, which is computed by summing up the latest messages received from the neighboring function nodes except the target function node. Formally, the query message from variable node $x_i \in \mathbf{x_j}$ to function node f_j in iteration m is given by

$$q^m_{x_i \rightarrow f_j}(x_i) = \sum_{f_k \in N_{x_i} \setminus \{f_j\}} r^{m-1}_{f_k \rightarrow x_i}(x_i) - \alpha, \tag{1}$$

where N_{x_i} is the set of neighboring function nodes of x_i, $r^{m-1}_{f_k \rightarrow x_i}(x_i)$ is the response message from f_k in the previous iteration and α is a normalization term to control the magnitude of

each entry in the message. A response message is sent by a function node to a neighboring variable node, which contains the best utility value for each value in the domain of target variable under current belief. Formally, the response message from function node f_j to variable node $x_i \in \mathbf{x_j}$ in iteration m is given by

$$r^m_{f_j \to x_i}(x_i) = \max_{\mathbf{x_j} \setminus \{x_i\}} \left(f_j(\mathbf{x_j}) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q^{m-1}_{x_k \to f_j}(x_k) \right). \tag{2}$$

A variable node x_i makes a decision in iteration m[1] by choosing the assignment with the highest utility under the current belief. That is,

$$x_i^* = \arg\max_{v_i \in D_i} \sum_{f_k \in N_{x_i}} r^m_{f_k \to x_i}(v_i).$$

3.3 Generic domain pruning

It can be seen that the computational overhead of Eq. (2) is exponential in the arity of utility function f_j, which prohibits Max-sum from solving the problems with high-arity factors. Generic Domain Pruning (GDP) [21] is a state-of-the-art acceleration algorithm operating on completely sorted utility lists. In more detail, for each variable $x_i \in \mathbf{x_j}$ and each value $v_i \in D_i$, f_j sorts the space corresponding to $\mathbf{x_j} \wedge \{x_i = v_i\}$ into $SortedEntries^i_j(v_i)$ according to the utility value, where each entry $e \in SortedEntries^i_j(v_i)$ is an assignment to $\mathbf{x_j}$ such that x_i takes v_i. When computing the response utility for $x_i = v_i$, f_j constructs a one-shot lower bound lb according to the entry with the highest local utility value. That is,

$$lb = f_j(e_0) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(e_0[x_k]) - \max_{v_k \in D_k} q_{x_k \to f_j}(v_k),$$

where e_0 is the first element in $SortedEntries^i_j(v_i)$ and $e_0[x_k]$ is the assignment of x_k in entry e_0. Clearly, an entry e is proven to be suboptimal if its local utility $f_j(e)$ is lower than lb, since its total value is lower than the one produced by e_0 even if the maximum query message utility is attained. Therefore, GDP performs binary search on $SortedEntries^i_j(v_i)$ to find the maximum index q such that $f_j(e_q) \geq lb$, and returns all entries between e_0 and e_q (both are inclusive) as the pruned range.

Figure 2 illustrates the trace of GDP when computing response utility for $x_1 = F$. Starting with a preprocessing phase, which sorts the utility function in descending order, GDP uses the first entry in the sorted utility list $\{F, T, T\}$ to compute an initial value 10. Then the value is shifted by subtracting the maximum message utility of each non-target variable, resulting in the lower bound of 5. Finally, any entry with local utility value smaller than 5 is pruned and the algorithm returns $\{F, T, T\}$ and $\{F, F, F\}$ as the pruned range.

[1] We hereafter drop the notion of m for sake of simplicity.

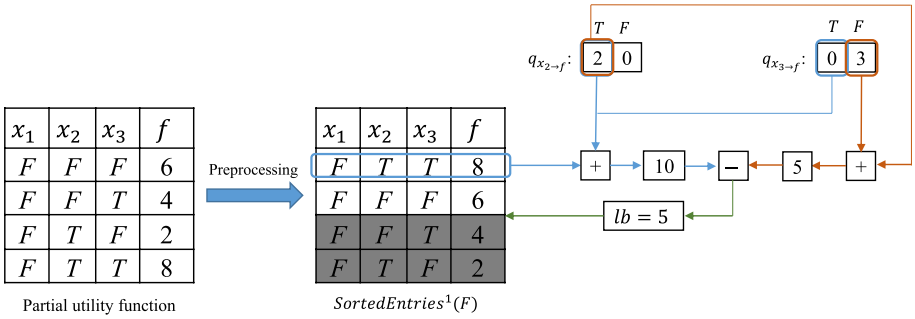


Fig. 2 The trace of GDP

3.4 Function decomposing and state pruning

A prominent drawback of GDP is the prohibitively expensive sorting overhead. In fact, for each utility function f_j, sorting for a variable requires $O(nd^n)$ operations, where $n = |\mathbf{x_j}|$ and $d = \max_{x_i \in \mathbf{x_j}} |D_i|$. Therefore, Function Decomposing and State Pruning (FDSP) [5] was proposed to reduce the search space by directly performing branch and bound without sorting the utility values.

Specifically, FDSP assumes a static ordering over the involved variables of a utility function, e.g., a lexicographical order. When f_j is computing a response utility for a target assignment $\mathbf{x_{j,i}} = v$ where $1 \le i \le |\mathbf{x_j}|$, it performs backtrack search by maintaining a partial assignment PA. Pruning happens whenever the upper bound of PA is smaller than the known lower bound. Here, the upper bound is given by

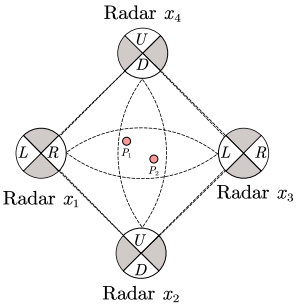
$$ub^{PA} = \sum_{1 \le k \le l, k \ne i} q_{\mathbf{x_{j,k}} \to f_j}(PA[\mathbf{x_{j,k}}]) + \sum_{l < k \le |\mathbf{x_j}|, k \ne i} \max_{v_k} q_{\mathbf{x_{j,k}} \to f_j}(v_k) + FunEst_l(PA), \tag{3}$$

where i is the index of the target variable, l is the index of the last assigned variable of PA in $\mathbf{x_j}$ and $FunEst_l(PA)$ is the most optimistic estimation on how much local utility we will get given prefix PA. In particular, FDSP considers two types of function estimation. When $i < l$, the estimation is nothing but a maximization to f_j over all unassigned variables. Otherwise, the estimation is enhanced by considering the assignment of the target variable $\mathbf{x_{j,i}}$. That is,

$$FunEst_l(PA) = \begin{cases} \max\limits_{z = \{\mathbf{x_{j,k}} | l < k \le |\mathbf{x_j}|\}} f_j(PA, z) & i < l \\ \max\limits_{z = \{\mathbf{x_{j,k}} | l < k \le |\mathbf{x_j}| \wedge k \ne i\}} f_j(PA, \mathbf{x_{j,i}} = v, z) & i > l \end{cases}. \tag{4}$$

To compute the estimations efficiently, FDSP performs dynamic programming (i.e., the so-called "function decomposing") to back up the maximum local utility values in preprocessing phase, which incurs a much smaller overhead than GDP.

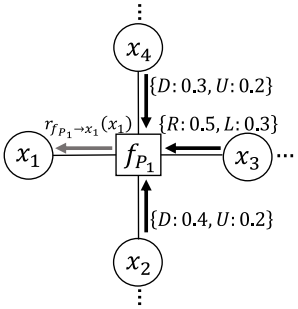
Fig. 3 The NetRad system example



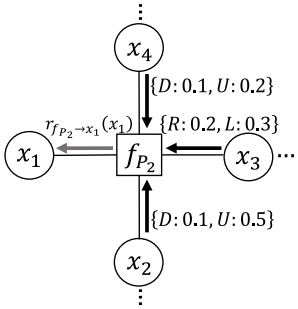
(a) the NetRad system

x_1	x_2	x_3	x_4	f_{P_1}	f_{P_2}
R	D	R	D	0.98	0.14
R	D	R	U	0.97	0.13
R	D	L	D	0.99	0.15
R	D	L	U	0.98	0.14
R	U	R	D	0.99	0.15
R	U	R	U	0.98	0.14
R	U	L	D	1	1
R	U	L	U	0.99	0.15

(b) partial utility tables



(c) local interactions of f_{P_1}



(d) local interactions of f_{P_2}

3.5 Search trees

Search trees [8, 31] are a powerful tool for representing search spaces. In a search tree, each level is labelled with a variable and each node corresponds to a possible assignment the variable can take. Therefore, a path from a root to a leaf uniquely determines a full assignment, and enumerating all possible solutions of the search space is equivalent to performing a travers of the search space. For example, Fig. 4b presents a search tree over variables x_2, x_3 and x_4 where x_2 and x_4 can take either D or U while x_3 takes a value from $\{R, L\}$. A path $D - R - D$ corresponds to an assignment $\{x_2 = D, x_3 = R, x_4 = D\}$.

Naturally, the search space of Eq. (2) can be represented by a search tree where each level is assigned to a non-target variable and each node is the value the associated variable can take. In fact, FDSP implicitly exhausts the search tree by branching on each non-target variable and discards subtrees when the corresponding partial assignments are proven to be suboptimal.

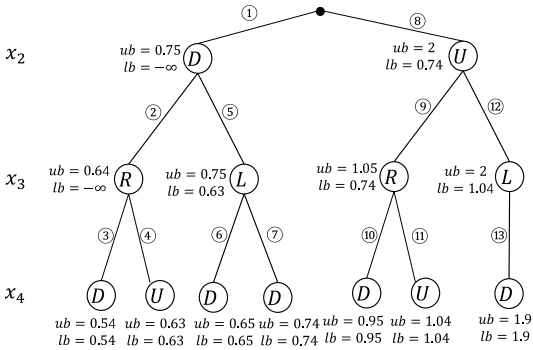
4 Motivation

In this section, we demonstrate that the existing acceleration algorithms could perform poorly when solving problems with dense utility functions. Consider a part of a NetRad system shown in Fig. 3 consisting of four radars and two weather phenomena P_1, P_2. For simplicity, assume that each radar can only scan one of two sectors (i.e., $\{R, L\}$ for x_1 and

x_1	x_2	x_3	x_4	f_{P_1}	
R	U	L	D	1	e_0
R	D	L	D	0.99	
R	U	R	D	0.99	
R	U	L	U	0.99	
R	D	R	D	0.98	
R	D	L	U	0.98	
R	U	R	U	0.98	
R	D	R	U	0.97	

$lb = 0.6$

(a) the trace of GDP on f_{P_1}



(b) the trace of FDSP on f_{P_2}

Fig. 4 Traces of GDP and FDSP on the NetRad system example

x_3, $\{D, U\}$ for x_2 and x_4). Phenomenon P_1 requires less joint observation efforts and the utility is high even if there is only one radar scanning P_1, while observing P_2 is demanding and all radars must scan simultaneously to obtain a high utility.

Figure 3b–d give the partial utility functions (w.r.t. $x_1 = R$) and local interactions for P_1 and P_2, respectively. We now aim to compute the response utility value for $x_1 = R$ from function node f_{P_1} and f_{P_2}. When applying GDP to f_{P_1}, a one-shot lower bound is constructed according to the entry $e_0 = \{R, U, L, D\}$ since it has the highest local utility value. That is,

$$lb = 1 + 0.2 - 0.4 + 0.3 - 0.5 + 0.3 - 0.3 = 0.6.$$

Therefore, any entry with local utility value less than 0.6 is filtered out. Unfortunately, as demonstrated in Fig. 4a, since all entries are densely distributed in the range of [0.97, 1], GDP actually cannot prune any combination from the search space in this case and fails to accelerate Max-sum.

On the other hand, when applying FDSP to f_{P_2}, the algorithm sequentially extends the partial assignment and performs branch and bound to reduce the search space. Figure 4b presents the trace of FDSP on f_{P_2} when computing response utility for $x_1 = R$ where the number labelled in each edge denotes the sequence of exploration. The algorithm explores the search space in a depth-first fashion by extending a partial assignment. After 3 steps of extension, FDSP reaches the first full assignment $\{R, D, R, D\}$[2] and updates the lower bound by

$$lb = 0.14 + 0.1 + 0.2 + 0.1 = 0.54.$$

Unfortunately, the lower bound fails to reduce the search space in this case. In fact, it can be clearly seen that the pruning only happens after visiting $\{R, U, L, D\}$. Therefore, in this case FDSP needs to explore a considerably large proportion of search space to obtain a high-quality lower bound. In fact, FDSP almost degenerates to vanilla Max-sum and only

[2] We have omitted $x_1 = R$ from the search tree.

prunes one partial assignment in this case. That is because it sequentially explores the search space, and high utility values are sparsely distributed in the utility function.

In fact, dense utility values are ubiquitous in real-world scenarios due to the law of diminishing marginal utility. Therefore, we aim to develop a series of efficient techniques for accelerating belief propagation algorithms by alleviating the negative effect of dense utility values.

5 Dynamic domain pruning techniques

In this section, we present our dynamic domain pruning techniques for accelerating Max-sum. We begin with introducing GD^{2}P, a variant of GDP that integrates both search and pruning to iteratively reduce the search space. To prune entries with the same utility value effectively, we then present ST-GD^{2}P which merges the tied entries to search trees to enable branch and bound. Finally, we propose a discretization mechanism to offer a trade-off between the reconstruction overhead and the domain pruning efficiency. We conclude by discussing the modifications when applying our methods to other versions of belief propagation.

5.1 GD^{2}P

As demonstrated earlier, the quality of the lower bound is the key of efficient pruning, especially when local utility values are dense. Therefore, we propose to integrate both search and pruning and iteratively reduce the search space by continuously tightening a running lower bound. Thus, the pruning is no longer a one-shot procedure and the scheme is referred as Generic Dynamic Domain Pruning (GD^{2}P). Alg.1 presents the sketch of GD^{2}P.

Similar to GDP, GD^{2}P also begins with complete sorting of each utility function (line 1-3). When computing a response message for a variable node $x_i \in \mathbf{x_j}$, it searches for the highest utility for each assignment $v_i \in D_i$ by exhausting the sorted entries whose local utility value is no less than the running lower bound lb in a sequential order (line 8–13). Here, the lower bound is updated whenever a higher utility is found (line 10–11).

Table 1 The trace of GD^{2}P on f_{P_1}

e	$f_{P_1}(e)$	*util*	*util**	*lb*
$\{R, U, L, D\}$	1	1.8	1.8	0.6
$\{R, D, L, D\}$	0.99	1.99	1.99	0.79
$\{R, U, R, D\}$	0.99	1.99	1.99	0.79
$\{R, U, L, U\}$	0.99	1.69	1.99	0.79
$\{R, D, R, D\}$	0.98	2.18	2.18	0.98
$\{R, D, L, U\}$	0.98	1.88	2.18	0.98
$\{R, U, R, U\}$	0.98	1.88	2.18	0.98
$\{R, D, R, U\}$	0.97	Pruned	2.18	0.98

Algorithm 1: GD^{2}P for function node f_j

When `Initialization`**:**
1 **foreach** $x_i \in \mathbf{x_j}$ **do**
2 **foreach** $v_i \in D_i$ **do**
3 $SortedEntries_j^i(v_i) \leftarrow$ arg sort $f_j(x_i = v_i, \cdot)$ in non-increasing order
When *computing a response message for* $x_i \in \mathbf{x_j}$**:**
4 $msgUB \leftarrow \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k)$
5 **foreach** $v_i \in D_i$ **do**
6 $e \leftarrow$ the first element in $SortedEntries_j^i(v_i)$
7 $u \leftarrow f_j(e)$, $util^* \leftarrow -\infty, lb \leftarrow -\infty$
8 **while** $e \neq NIL$ *and* $u \geq lb$ **do**
9 $util \leftarrow u + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(e[x_k])$
10 $util^* \leftarrow \max(util^*, util)$
11 $lb \leftarrow util^* - msgUB$
12 $e \leftarrow$ the next element in $SortedEntries_j^i(v_i)$
13 $r_{f_j \to x_i}(v_i) \leftarrow util^*$
14 **return** $r_{f_j \to x_i}(x_i)$

To look deeper into how dynamic domain pruning works, consider the instance in Fig. 3c. The upper bound of query messages' utility is given by

$$msgUB = \max\{0.4, 0.2\} + \max\{0.5, 0.3\} + \max\{0.3, 0.2\} = 1.2.$$

Then GD^{2}P sequentially explores the rows of sorted utility list (i.e., Fig. 4a) and updates the lower bound until the current local utility value is less than the lower bound. Table 1 presents the detailed trace of GD^{2}P.

Next, we theoretically show its correctness and superiority over GDP.

Theorem 1 *GD^{2}P guarantees the optimality of Eq.* (2).

Proof Assume that the optimal entry e^* is pruned by GD^{2}P when $x_i = v_i$. According to line 4, 8 and 11, we have

$$f_j(e^*) < lb = util^* - msgUB$$

where $util^*$ is the highest utility returned by GD^{2}P. Combing with query messages, we have

$$f_j(e^*) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(e^*[x_k]) < util^* - msgUB + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(e^*[x_k]).$$

Since

$$\sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(e^*[x_k]) \leq \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k) = msgUB,$$

it must be the case that

$$f_j(e^*) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(e^*[x_k]) < util^*,$$

which results in e^* is not an optimal entry, and is contradictory to the assumption. Therefore, GD^{2}P must preserve entry e^*. According to the assumption that e^* is the optimal solution to Eq. (2) and the fact that $util^*$ is the maximum value of all visited entries (line 9–10), GD^{2}P must return the value corresponding to e^* and therefore guarantees the optimality of Eq. (2). □

Theorem 2 *GD^{2}P never explores the entries pruned by GDP.*

Proof Assume that GD^{2}P explores an entry e which is pruned by GDP at iteration m. Denote the lower bound constructed by GDP as lb^{GDP} and the current lower bound of GD^{2}P as $lb^{GD^2P}(m)$. According to line 8, since e is explored by GD^{2}P but pruned by GDP, it must be the case that

$$lb^{GDP} > f_j(e) \geq lb^{GD^2P}(m). \tag{5}$$

W.l.o.g., assume that sorting is stable, i.e., both GDP and GD^{2}P have the same sorted local utility list and construct the initial lower bound using the same entry. Therefore, $lb^{GDP} = lb^{GD^2P}(0)$, where $lb^{GD^2P}(0)$ is the lower bound of GD^{2}P in iteration 0. Since $lb^{GD^2P}(m)$ is non-decreasing (i.e., line 10-11), we have

$$lb^{GD^2P}(m) \geq lb^{GD^2P}(0) = lb^{GDP},$$

which is contradictory to Eq. (5). □

The preprocessing overhead of GD^{2}P is the same as that of GDP. Formally, we have the following proposition.

Proposition 1 *GD^{2}P requires $O(n^2d^n)$ operations and $O(nd^n)$ space to preprocess a utility function f_j, where $n = |\mathbf{x_j}|$ is the arity of the function and $d = \max_{x_i \in \mathbf{x_j}} |D_i|$ is the maximum domain size of the involved variables.*

Proof Since the function has $O(d^n)$ utility values, for each involved variable and each value it requires $O(d^{n-1} \log d^{n-1}) = O((n-1)d^{n-1})$ operations to perform sorting (line 1-3).

Therefore, GD^{2}P needs $O((n-1)d^n)$ operations for each variable and $O(n^2d^n)$ operations in total to perform preprocessing.

For space complexity, we first note that an assignment to the involved variables takes $O(n)$ space and thus storing all combinations of the involved variables requires $O(nd^n)$ space for a variable. Since we could share these assignments across different variables via pointers in practice, it takes $O(d^n)$ space for each variable to store the addresses of the data structures corresponding to the assignments. Therefore, the overall space complexity of GD^{2}P is $O(nd^n + nd^n) = O(nd^n)$. □

To implement such sharing mechanism, we maintain an array of length $\Pi_{x_i \in \mathbf{x_j}} |D_i|$, in which each element is the address of the data structure corresponding to an assignment to all involved variables. To lookup the address of a specific assignment, we first convert the assignment into an index according to a pre-defined dimension order, then get the corresponding address by indexing the array. Therefore, the extra overhead of the mechanism lies in computing the index for each assignment, which requires $O(n)$ operations. Given n variables and $O(d^n)$ assignments for each variable to be processed, the total extra overhead is in $O(n^2d^n)$, which does not increase the overall time complexity of GD^{2}P.

5.2 ST-GD^{2}P

As demonstrated by Fig. 4a and Table 1, both GDP and GD^{2}P perform poorly when there are ties in utility functions since they use a linear structure to organize the search space. Consequently, they have to exhaust all the tied entries whose local utility is no less than the lower bound. Thus, we propose to organize tied entries into a search tree so as to enable efficient branch and bound when the domain pruning technique fails to reduce the search space. The scheme is referred as Search Tree-based GD^{2}P (ST-GD^{2}P) and Alg.2 presents the sketch.

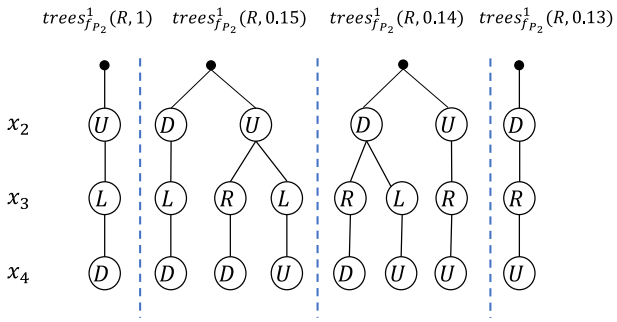


Fig. 5 Search trees for $x_1 = R$ when applied to Fig. 3d

Algorithm 2: ST-GD^{2}P for function node f_j

When `Initialization`**:**
1 **foreach** $x_i \in \mathbf{x_j}$ **do**
2 **foreach** $v_i \in D_i$ **do**
3 $SortedUtil^i_j(v_i) \leftarrow$ all distinct utility values of $f_j(x_i = v_i, \cdot)$
4 sort $SortedUtil^i_j(v_i)$ in descending order
5 **foreach** $u \in SortedUtil^i_j(v_i)$ **do**
6 $t \leftarrow$ create an empty tree
7 **foreach** *entry* e *such that* $e[x_i] = v_i$ *and* $f_j(e) = u$ **do**
8 `Insert`(t, e)
9 $tree^i_j(v_i, u) \leftarrow t$
When *computing a response message for* $x_i \in \mathbf{x_j}$**:**
10 $msgUB \leftarrow \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k)$
11 **foreach** $v_i \in D_i$ **do**
12 $u \leftarrow$ the first element in $SortedUtil^i_j(v_i)$
13 $util^* \leftarrow -\infty, lb \leftarrow -\infty$
14 **while** $u \neq NIL$ *and* $u \geq lb$ **do**
15 $t \leftarrow trees^i_j(v_i, u)$
16 $util \leftarrow$ `Branch-and-bound`$(t, util^*)$
17 $util^* \leftarrow \max(util^*, util)$
18 $lb \leftarrow util^* - msgUB$
19 $u \leftarrow$ the next element in $SortedUtil^i_j(v_i)$
20 $r_{f_j \to x_i}(v_i) \leftarrow util^*$
21 **return** $r_{f_j \to x_i}(x_i)$
Function `Insert`(t, e)**:**
22 $n \leftarrow$ the root of t
23 **foreach** $x_i \in \mathbf{x_j}$ **do**
24 **if** *there is a child* c *of* n *such that* $val(c) = e[x_i]$ **then**
25 $n \leftarrow c$
26 **else**
27 $c \leftarrow$ create a new node
28 $val(c) \leftarrow e[x_i]$, mark c as a child of n
29 $n \leftarrow c$

Different than completely sorting a utility function in GDP and GD^{2}P, ST-GD^{2}P only sorts for *distinct* local utility values for each assignment of each variable in the preprocessing phase (line 1–4). After that, it groups the entries with the same local utility value

(line 5–9, 22–29) by inserting these entries to a search tree (line 22–29). In other words, instead of maintaining a linear list in GDP and GD^{2}P, ST-GD^{2}P uses a sorted array of search trees to collectively represent the search space. Note that the construction of search trees can be done by a sequential iteration over the utility function as the trees can be built incrementally. Similar to GD^{2}P, it maintains a running lower bound *lb* and terminates whenever the local utility value is less than the lower bound (line 12–19).

It is noteworthy that instead of iterating over the sorted entries in GDP and GD^{2}P, ST-GD^{2}P iterates over the distinct sorted utility values in descending order and performs branch and bound to reduce the search space of tied entries. In more detail, for each distinct local utility u, ST-GD^{2}P exhausts the corresponding search tree $tree_j^i(v_i, u)$ in a depth-first fashion. For a node in the search tree, the path from the root to that node corresponds to a *partial assignment*. Therefore, by constructing a upper bound ub^{PA} for each partial assignment PA, we can efficiently explore the search tree by discarding the subtree of a node whenever the upper bound of corresponding partial assignment is less than the known highest utility $util^*$ (line 15–16). Formally, the upper bound is given by

$$ub^{PA} = u + \sum_{x_k \in PA} q_{x_k \to f_j}(PA[x_k]) + \sum_{x_k \notin PA} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k). \tag{6}$$

Take Fig. 3d as an example. Given $x_1 = R$, ST-GD^{2}P first sorts distinct local utility values $\{1, 0.15, 0.14, 0.13\}$ and builds a search tree for each of them. Figure 5 presents the search trees. When computing the response utility for $x_1 = R$, ST-GD^{2}P first performs branch and bound to exhaust the search tree $tree_{f_{P_2}}^1(R, 1)$. It can be concluded that the algorithm terminates after reaching the first full assignment $\{R, U, L, D\}$ since it finds the highest utility 1.9, which results in a pruned rate of 87.5%.

We now show its correctness and its superiority over GD^{2}P.

Theorem 3 *ST-GD^{2}P guarantees the optimality of Eq.* (2).

Proof There are two lines that prune the search space, i.e., line 14 and line 16. We have shown that line 14 cannot prune the highest utility in Theorem 1. Thus, we only need to show that line 16 never prunes the optimal assignment. Assume that the optimal full assignment A^* is pruned by line 16. Thus, it must exist a prefix $PA \subset A^*$ such that

$$ub^{PA} < util^*,$$

where $util^*$ is the highest utility found by ST-GD^{2}P. In fact,

$$\begin{aligned} ub^{PA} &= f_j(A^*) + \sum_{x_k \in PA} q_{x_k \to f_j}(PA[x_k]) + \sum_{x_k \notin PA} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k) \\ &\geq f_j(A^*) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A^*[x_k]). \end{aligned}$$

Thus, we have

$$f_j(A^*) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A^*[x_k]) < util^*,$$

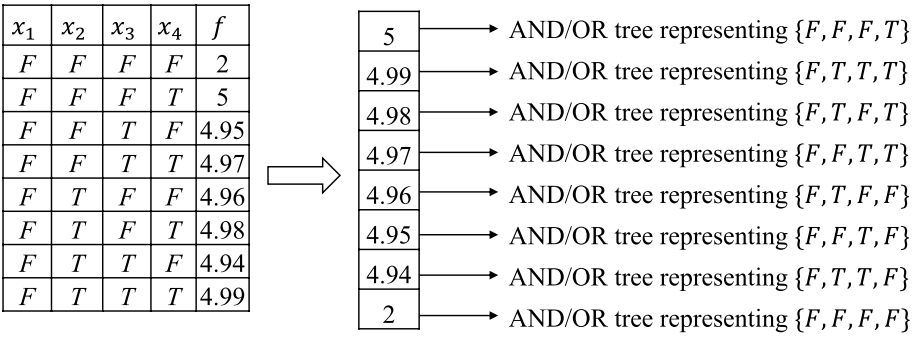


Fig. 6 ST-GD^{2}P operating on extremely dense utilities

implying that there is another full assignment $A' \neq A^*$ yielding higher utility than A^*, which is contradictory to the assumption that A^* is optimal. The optimality is hereby guaranteed. □

Theorem 4 *ST-GD^{2}P never explores the assignments pruned by GD^{2}P.*

Proof Assume that ST-GD^{2}P explores one full assignment A which is pruned by GD^{2}P. Denote the lower bound of ST-GD^{2}P when exploring A as lb^{ST} and the lower bound of GD^{2}P when pruning A as lb^{GD^2P}, respectively. It must be the case that

$$lb^{GD^2P} > f_j(A) \geq lb^{ST}. \tag{7}$$

Since GD^{2}P sequentially exhausts the search space, it must exist an assignment $A' \prec A$ such that

$$\begin{aligned} lb^{GD^2P} &= f_j(A') + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A'[x_k]) - \max_{v_k \in D_k} q_{x_k \to f_j}(v_k) \\ &> f_j(A), \end{aligned} \tag{8}$$

which indicates that $f_j(A') > f_j(A)$. On the other hand, ST-GD^{2}P explores distinct local utility values in a descending order (line 4, 12 and 19 of Alg. 2). According to Eqs. (7–8), A' produces a higher utility than A. Thus, ST-GD^{2}P must have explored an assignment A'' such that

$$f_j(A'') + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A''[x_k]) \geq f_j(A') + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A'[x_k]),$$

which implies

$$lb^{ST} \geq f_j(A') + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A'_{x_k}) - \max_{v_k \in D_k} q_{x_k \to f_j}(v_k) = lb^{GD^2P}. \tag{9}$$

Therefore, Eq. (9) is contradictory to Eq. (7) and the theorem is concluded. □

The additional overheads of ST-GD^{2}P mainly lie in creating and maintaining the sorted array of search trees. Specifically, we have the following result.

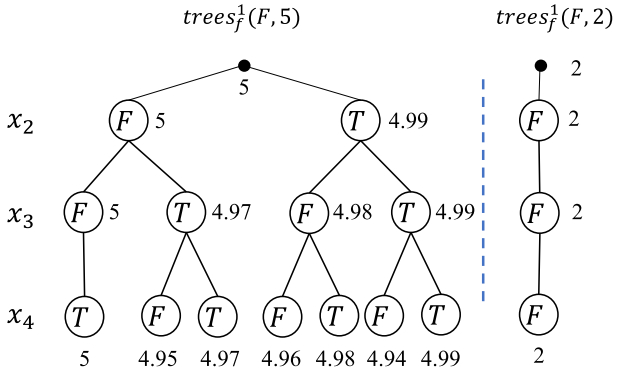


Fig. 7 Discretization mechanism operating on extremely dense utilities

Proposition 2 *For each utility function, ST-GD^{2}P requires $O(n^2d^n)$ operations and space to perform preprocessing.*

Proof For a utility function with n variables, inserting an assignment to a search tree requires $O(n)$ operations (line 22–29). Therefore, building search trees for each variable requires $O(nd^n)$ operations, and sorting these trees requires $O((n-1)d^n)$ operations in the worst case where all utility values are different. Given n variables to be preprocessed, the overall time complexity is $O(n^2d^n + (n^2-n)d^n) = O(n^2d^n)$.

Similarly, in the worst case each search tree corresponds to a single assignment and takes $O(n)$ space. Therefore, ST-GD^{2}P needs $O(nd^n)$ space for a variable and $O(n^2d^n)$ space in total in order to perform preprocessing. □

5.3 Discretization mechanism

Since ST-GD^{2}P builds a search tree for each distinct local utility value, each tree would correspond to a small search space when the local utility values are extremely dense, which would incur unnecessary solution reconstructions. Consider the example in Fig. 6. Assume that all the assignments with prefix $\{F,T\}$ are pruned by line 16 and the highest utility is attained on $\{F,F,F,F\}$. Obviously, ST-GD^{2}P has to visit the suboptimal partial assignment $\{F,T\}$ for four times in this case. Besides, storing small search trees is also memory-consuming due to the repetitive substructures. In this example, since each search tree only represents an assignment, substructures like $\{F,F,F\}$, $\{F,F,T\}$, $\{F,T,F\}$ and $\{F,T,T\}$ repeat twice, which results in the worst-case memory consumption. Finally, a large number of unique utility values would also incur a high sorting overhead.

We overcome the issues by introducing a discretization mechanism. That is, instead of sorting and building search trees for distinct local utility values directly, we first group the utility values into several discrete slots. In more detail, given a step size $\eta > 0$, each utility value u of a utility function is transformed to a discretized value $s = \eta\lceil u/\eta \rceil$, where $\lceil \cdot \rceil$ is the ceiling operation, which returns the least integer greater than or equal to a number. Then we sort all the distinct discretized values into a list $S = \{s_1, \ldots, s_{|S|}\}$ such that $s_i > s_j, \forall 1 \le i < j \le |S|$. For each $s \in S$, we merge all the entries in the utility function whose discretized value is s into a search tree $tree(v_i, s)$, and update the discretized value s

to the maximum local utility value in the search space specified by $tree(v_i, s)$. Again, note that since search trees can be built incrementally, we only need a single scan to the utility function in order to perform preprocessing.

When computing a response message, we sequentially explore the discretized value list S, perform branch and bound on the corresponding search trees, and terminate if the current discretized value s is smaller than the running lower bound. It is worth noting that since discretized value s could be larger than the utility value of an entry in the search space specified by $tree(v_i, s)$, directly plugging s to Eq. (6) may not result in a tight upper bound and would jeopardize the performance of branch and bound. Therefore, for each node in the search tree $tree(v_i, s)$, we maintain an estimation $est_s(\cdot)$ which is the maximum local utility achieved by any full assignment derived from that node. Now, the upper bound of a partial assignment PA is given by[3]

$$ub^{PA} = est_s(PA) + \sum_{x_k \in PA} q_{x_k \to f_j}(PA[x_k]) + \sum_{x_k \notin PA} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k). \tag{10}$$

Figure 7 presents the search trees after performing discretization with a step size of 1 on the instance in Fig. 6, where the number associated with each node is the estimation. It can be seen that both the number of search trees and total memory consumption are reduced significantly. In fact, if we directly apply ST-GD^{2}P on Fig. 6, it would require 24 units of memory to store 8 search trees. While we only need 16 units of memory to maintain 2 search trees after discretization, since all small search trees with utility value greater than 4.94 are merged into a single large search tree (i.e., $tree_f^1(F, 5)$).

It is noteworthy that beside helping to reduce the preprocessing overhead by restricting the number of discretized utility values, discretization mechanism also offers a trade-off between domain pruning efficiency and reconstruction overhead. In more detail, a small η produces fine-grained slots and could prune the suboptimal slots promptly (line 12 of Alg.2). In contrast, a large η tends to build search trees corresponding to large search spaces, which reduces the reconstruction overhead since we are less likely to revisit the same prefix when the search trees are large enough.

We now show the soundness and the complexity of discretization mechanism.

Theorem 5 *ST-GD^{2}P with discretization mechanism guarantees the optimality of Eq.* (2).

Proof We first show that each discretized value s is no less than the local utility value of any entry in the space specified by $tree(v_i, s)$. Assume by contradiction that an entry e of utility function f_j belongs to the space specified by $tree(v_i, s)$ but $f_j(e) > s$. Its discretized value s_e is given by

$$s_e = \eta \lceil \frac{f_j(e)}{\eta} \rceil \geq f_j(e) > s,$$

which implies $s \neq s_e$ and thus e cannot belong to the space specified by $tree(v_i, s)$.

[3] Recall that a path from the root to an internal node in a search tree specifies a partial assignment. We therefore slightly abuse $est_s(PA)$ to denote the estimation of PA in $tree(v_i, s)$ which is stored in the node that corresponds to PA.

Assume that the optimal entry e^* belongs to a search tree which is pruned by the domain pruning part of the algorithm under the discretization mechanism. Since the discretized value is used to perform domain pruning, it must be the case that

$$f_j(e^*) \leq s_{e^*} < util^* - msgUB,$$

where s_{e^*} is the discretized value for entry e^*. Applying the same argument in the proof of Theorem 1, we conclude that e^* cannot be optimal and the domain pruning part preserves the optimality.

Similarly, assume that the optimal assignment A^* is pruned by the branch and bound part of the algorithm under the discretization mechanism. It must exist a prefix $PA \subset A^*$ such that

$$ub^{PA} = est_{s_{A^*}}(PA) + \sum_{x_k \in PA} q_{x_k \to f_j}(PA[x_k]) + \sum_{x_k \notin PA} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k) < util^*,$$

where s_{A^*} is the discretized value for assignment A^*. Since by definition $f_j(A^*) \leq est_{s_{A^*}}(PA)$, we have

$$\begin{aligned} ub^{PA} &\geq f_j(A^*) + \sum_{x_k \in PA} q_{x_k \to f_j}(PA[x_k]) + \sum_{x_k \notin PA} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k) \\ &\geq f_j(A^*) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A^*[x_k]), \end{aligned}$$

and

$$f_j(A^*) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A^*[x_k]) < util^*.$$

Applying the same argument in the proof of Theorem 3, we conclude that A^* cannot be optimal and the branch and bound part preserves the optimality. Therefore, ST-GD^{2}P with discretization mechanism guarantees the optimality of Eq. (2). □

Proposition 3 *Given a utility function f_j, let $\delta_j = \max_e f_j(e) - \min_e f_j(e)$ be the difference between the maximum and minimum utility values, and $\theta_j = \min_{e,e':f_j(e)\neq f_j(e')} |f_j(e) - f_j(e')|$ be the minimum difference between two distinct utility values. ST-GD^{2}P with discretization mechanism requires $O(n^2 d^n)$ operations and $O(\frac{\delta_j}{\eta} n^2 + nd^n)$ space to store search trees if $\eta > \theta_j$.*

Proof We first note that ST-GD^{2}P with discretization mechanism degenerates to ST-GD^{2}P if step size η is small enough, i.e., $\eta \leq \theta_j$. Hereafter we will focus on the case $\eta > \theta_j$.

Like ST-GD^{2}P, ST-GD^{2}P with discretization mechanism also needs to insert $O(d^n)$ entries into search trees for each variable, which requires $O(n^2 d^n)$ operations in total. Since the number of search trees in discretization mechanism is bounded by δ_j/η, the overhead of sorting search trees is $O(n\frac{\delta_j}{\eta} \log \frac{\delta_j}{\eta})$, which is bounded by $O(nd^n \log d^n) = O(n^2 d^n)$. Therefore, the overall time complexity is $O(n^2 d^n)$.

For space complexity, note that these search trees span the total search space. Therefore, for each variable there are $O(d^n)$ leaves. On the other hand, since there are at most δ_j/η search trees according to the discretization mechanism, any partial assignment can only appear at most δ_j/η times (i.e., the case that every search tree contains the partial assignment). Since there are $n-2$ variables[4] in a partial assignment in the worst case, ST-GD^{2}P requires $O(\frac{\delta_j}{\eta}(n-2)+d^n) = O(\frac{\delta_j}{\eta}n + d^n)$ space for a variable and $O(\frac{\delta_j}{\eta}n^2 + nd^n)$ space for all involved variables. □

5.4 Discussion

We note that our proposed algorithms can be easily adapted to the variants of Max-sum and other versions of belief propagation. For example, we could adapt our algorithms to speed up Max-product [50], which is an important instantiation of GDL to find maximum *a posteriori* (MAP) assignment of a probabilistic graphic model, by changing the summation operations to the product operations (i.e., line 4, 9 and 11 of Alg.1, line 10, 14 and 16 of Alg.2). That is, *msgUB* now is the product of maximum value in the query messages from non-target variables, the result is computed by multiplying the function value with the corresponding value of query messages, and *lb* is computed by the best result found so far dividing *msgUB*. Alternatively, one can cast Max-product belief propagation to Max-sum and directly use our methods by operating on the log space of the problem.

When combining with Max-sum variants like damped Max-sum [7], bounded Max-sum [43] and Max-sum_AD [56], our algorithms require few modifications since these variants still use Eq. (2) to compute response messages. However, there are variants which do not exactly follow Eq. (2). For example, a function node in Max-sum_ADSSVP [4] needs to consider the assignments of some neighbors when computing the maximum utility for Eq. (2). In this case, an additional verification procedure needs to be introduced to filter out the incompatible entries when performing branch and bound on a search tree. That is, when visiting a node whose variable needs to be fixed, we first check its compatibility before adding it to the partial assignment. If the assignment of the node is not consistent with the given assignment, we discard all the subsequent search space by backtracking to its parent.

6 Partial tree-sorting scheme

In this section, we present a K-depth partial tree-sorting scheme to reduce the preprocessing overhead of the proposed ST-GD^{2}P. We begin with the motivation of using limited-depth tree-sorting scheme. Then we detail the proposed K-depth partial tree-sorting scheme. Besides, we introduce several sorting criteria for ranking search spaces. Finally, we propose a built-in termination detection mechanism to avoid unnecessary enumeration.

[4] Note that the target variable is omitted from the search tree.

6.1 Motivation

A major drawback of ST-GD^{2}P is the high preprocessing overhead incurred by creating and maintaining the full search trees. Although discretization mechanism attempts to alleviate the issue by merging small search trees, the complexities still depend on the step size and the gap between the maximum utility and the minimum utility which varies in different utility functions. Moreover, even in the best case, i.e., there is only one search tree for each variable-assignment pair, ST-GD^{2}P still requires $O(nd^n)$ memory per function node to maintain all trees, which is undesirable in memory-limited scenarios such as coordinating low-power embedded devices [10].

Therefore, we consider to overcome the defect by limiting the depth of the search trees. That is, given a utility function with the arity of n, instead of maintaining the full search trees, we only consider the search trees with the depth of $K < n$, resulting the worest-case memory overhead of $O(nKd^K)$ which allows the tradeoff between the memory consumption and orderliness of search space. However, since each search tree now only specifies a subspace, several key issues arise when realizing such K-depth tree sorting scheme:

- How to implement branch and bound on limited-depth search trees?
- How to rank search trees when each search tree specifies a subspace rather than a concrete utility value?
- How to enforce domain pruning technique if the search trees are not ranked according to the maximum utility value in their subspace?

We address the first issue in Sect. 6.2, by presenting a general framework called PTS. Then we present several sorting criterion to rank search trees to solve the second issue in Sect. 6.3. Finally, we propose a built-in termination detection mechanism to enforce the domain pruning technique in Sect. 6.4.

6.2 *K*-depth partial tree-sorting scheme

In this section, we aim to develop a configurable K-Depth Partial Tree-sorting Scheme (PTS) to provide a tight bound on the memory consumption, and perform preprocessing phase within a user-specified memory budget.

Algorithm 3: K-depth PTS for function node f_j

When `Initialization`**:**
1 **foreach** $x_i \in \mathbf{x_j}$ **do**
2 **foreach** $v_i \in D_i$ **do**
3 **foreach** $e \in \Pi_{x_k \in SV_j^i} D_k$ **do**
4 $w_e \leftarrow$ digest the subspace $f_j(e, x_i = v_i, \cdot)$ by using a **sorting criterion** γ
5 insert e to $Tree_j^i(v_i, w_e)$
6 $SortedTree_j^i(v_i) \leftarrow$ sort $Tree_j^i(v_i, \cdot)$ by w in decreasing order
When *computing a response message for* $x_i \in \mathbf{x_j}$**:**
7 $msgUB \leftarrow \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} \max_{v_k \in D_k} q_{x_k \to f_j}(v_k)$
8 **foreach** $v_i \in D_i$ **do**
9 $t \leftarrow$ the first element in $SortedTree_j^i(v_i)$
10 $util^* \leftarrow -\infty, lb \leftarrow -\infty$
11 **while** ***termination condition*** *is not met* **do**
12 **if** $est(t) \geq lb$ **then**
13 $util \leftarrow$`Branch-and-bound`$(t, util^*)$
14 $util^* \leftarrow \max(util^*, util)$
15 $lb \leftarrow util^* - msgUB$
16 $t \leftarrow$ the next element in $SortedTree_j^i(v_i)$
17 $r_{f_j \to x_i}(v_i) \leftarrow util^*$
18 **return** $r_{f_j \to x_i}(x_i)$

Alg.3 presents the sketch of the partial tree-sorting scheme. Instead of sorting and maintaining full search trees over $n-1$ variables for each variable-assignment pair in ST-GD^{2}P, PTS only considers the combinations of the first K non-target variables. Specifically, given the lexicographical ordering of the involved variables of a utility function f_j, PTS first computes sorting variables SV_j^i for target variable $x_i \in \mathbf{x_j}$ according to

$$SV_j^i = \begin{cases} \{\mathbf{x_{j,k}} | 1 \leq k \leq K\} & x_i \succ \mathbf{x_{j,K}} \\ \{\mathbf{x_{j,k}} | 1 \leq k \leq K+1 \wedge \mathbf{x_{j,k}} \neq x_i\} & \text{Otherwise} \end{cases},$$

where $x_i \succ \mathbf{x_{j,K}}$ means x_i is ordered after $\mathbf{x_{j,K}}$. Intuitively, SV_j^i is the first K variables in the scope of f_j, except target variable x_i. Then the algorithm iterates over all possible combinations of SV_j^i to build search trees (line 3–5). For each combination e, a *sorting criterion* γ is applied to assess the quality of the combination by summarizing the utility values in the subspace of $\mathbf{x_j}$ given the prefix of $e \cup \{x_i = v_i\}$ into a scalar weight w_e, which will be detailed in Sect. 6.3. Optionally, like discretizing utility values in ST-GD^{2}P, the weight also can be discretized by a step size of η. Then the combination is merged into the search tree which has the same weight (or same discretized weight if discretization is used). After exhausting the combinations of SV_j^i, PTS sorts all the search trees by their weight (line 6). Like in the discretization mechanism, when building search trees we also maintain an estimation for each node in a search tree for efficient branch and bound.

It is noteworthy that both weight and estimation are indispensable for efficient branch and bound. Horizontally, we sort the partial search trees by their weight for obtaining high-quality lower bound promptly; vertically, estimations are used to compute a tight upper bound for a partial assignment so as to discard suboptimal solutions as soon as possible.

When computing a response message for variable node $x_i \in \mathbf{x_j}$, PTS sequentially explores the sorted search trees by performing branch and bound to reduce the search space until a *termination condition* is reached (line 9–17). Since we only build search trees for the first K variables, the procedure `Branch-and-bound` (line 13) needs to be specialized to handle two halves of search spaces. Specifically, when performing branch and bound on a search tree (i.e., the first half of the search space), we follow exactly the same method in discretization mechanism and use Eq. (10) to compute upper bound for a partial assignment.

After reaching a leaf of a search tree, the procedure `Branch-and-bound` needs to exhaust the remaining subspace (i.e., the second half of the search space). In fact, the path from the root to a leaf of a search tree now only corresponds to a prefix rather than a concrete full assignment. In other words, we still need to solve a smaller optimization problem of Eq. (2) defined over $\mathbf{x_j} \backslash (SV_j^i \cup \{x_i\})$.

Technically, all the existing approaches (e.g., exhaustive enumeration, GDP, GD^{2}P, FDSP, etc.) can be used to exhaust the subspace. In our implementation, we use FDSP due to its lower computational overhead. To this end, we maintain function estimations [5] for the remaining $n - K - 1$ variables that do not appear in the search trees (i.e., the variables in $\mathbf{x_j} \backslash (SV_j^i \cup \{\mathbf{x_i}\})$). When exhausting the remaining subspace, the upper bound of a partial assignment is computed by querying the corresponding function estimation [i.e., Eqs. (3–4)].

It is noteworthy that our PTS requires less space than FDSP in the worst case when K is small. Formally, we have the following results.

Proposition 4 *Given a utility function, PTS requires $O(nKd^K)$ space to store search trees.*

Proof Since PTS only builds search trees for the first K variables, a path from a root to a leaf corresponds to an assignment of length K. The total number of combinations of K variables is $O(d^K)$. Therefore, PTS requires $O(Kd^K)$ space for one variable in the worst case, and $O(nKd^K)$ to store all search trees. □

Corollary 1 *PTS' worst-case memory consumption is less than FDSP's worst-case memory consumption when $K < \frac{1+nd}{n+d}$.*

Proof For the remaining $n - K - 1$ variables that do not appear in the search trees, PTS and FDSP require the same space as both of them need to store function estimations for each of these variables. For the first K variables, FDSP needs at least $O([1 + d(n - K)]d^K)$ space since maintaining function estimations for variable $\mathbf{x_{j,k}}$ requires $O([1 + d(n - k)]d^k)$ space [5]. Letting $nKd^K < [1 + d(n - K)]d^K$, we arrive at $K < \frac{1+nd}{n+d}$. □

6.3 Sorting criteria

In ST-GD^{2}P, each path from a root to a leaf in an search tree uniquely determines a full assignment to the involved variables of a utility function, and hence the utility values can be directly used to rank these paths/assignments. However, since only a subset of variables are sorted in PTS, a path now could correspond to a partial assignment and specify a *subspace* rather than a concrete utility value. Therefore, we propose to rank a partial assignment by a scalar weight that summarizes the utility values in the corresponding subspace

0	1	2	3	3	4	4	5	5	6	7	7	8	9	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig. 8 Sorted utility list $\mathcal{U}$

(i.e., line 4, 6 of Alg.3). Specifically, given a utility function f_j and a partial assignment e of length K and the target variable-assignment pair $x_i = v_i$, we consider the following criteria to compute the weight for summarizing subspace $f_j(e, x_i = v_i, \cdot)$.

- **Maximum utility value**. The most straightforward way to describe the subspace is the maximum utility value in the subspace. Formally,

$$\gamma_{\max}(f_j(e, x_i = v_i, \cdot)) = \max_z f_j(e, x_i = v_i, z),$$

  where z is the remaining unassigned variables that do not appear in the prefix $e \cup \{x_i = v_i\}$. Computing the maximum utility value requires $O(1)$ space to store the result and $O(d^{n-K-1})$ operations to exhaust the subspace.
- **Mean utility value**. Concentrating on the maximum utility value solely is not desirable when the maximum utility value is sparsely distributed in the subspace. Therefore, this criterion tries to remedy the issue by considering the average value of the subspace. That is,

$$\gamma_{\text{mean}}(f_j(e, x_i = v_i)) = \frac{\sum_z f_j(e, x_i = v_i, z)}{|f_j(e, x_i = v_i, \cdot)|}.$$

  The criterion has the same complexities as maximum utility value criterion, since both of them need to exhaust the subspace.
- Q_3 **value**. This criterion considers the third quartile of the utility values in the subspace. Denote the sorted utility values (in ascending order) of $f_j(e, x_i = v_i, \cdot)$ as $\mathcal{U}^e_{f_j}$. Q_3 value is given by

$$\gamma_{Q_3}(f_j(e, x_i = v_i, \cdot)) = \mathcal{U}^e_{f_j}[m] + \alpha(\mathcal{U}^e_{f_j}[m+1] - \mathcal{U}^e_{f_j}[m]),$$

  where $m = \lfloor 0.75 \times |f_j(e, x_i = v_i, \cdot)| \rfloor$ and $\alpha = 0.75 \times |f_j(e, x_i = v_i, \cdot)| - m$. The criterion needs to sort the utility values in the subspace, which requires $O((n-K)d^{n-K-1})$ operations and $O(d^{n-K-1})$ space.
- **H-utility value**. Analogous to the H-index [18] measuring the impact of a scholar, this criterion attempts to capture simultaneously the quality and quantity of utility values. Formally, given sorted utility values $\mathcal{U}^e_{f_j}$, the H-position of subspace $f_j(e, x_i = v_i, \cdot)$ is given by

$$h = \max\left\{ i \in \mathbb{N}^+ | 1 - \frac{i}{|f_j(e, x_i = v_i, \cdot)|} \geq \frac{\mathcal{U}^e_{f_j}[i] - f_j^-}{\delta_j} \right\},$$

  where δ_j is the difference between the maximum and minimum utility values of f_j, and $f_j^- = \min_{\mathbf{x_j}} f_j(\mathbf{x_j})$. Intuitively, an H-position of h means that there are at least $\frac{\mathcal{U}^e_{f_j}[h] - f_j^-}{\delta_j} \times 100\%$ entries in $f_j(e, x_i = v_i, \cdot)$ whose utility value ranks at the top $(1 - \frac{\mathcal{U}^e_{f_j}[h] - f_j^-}{\delta_j}) \times 100\%$ of the utility range. Then H-utility is defined as

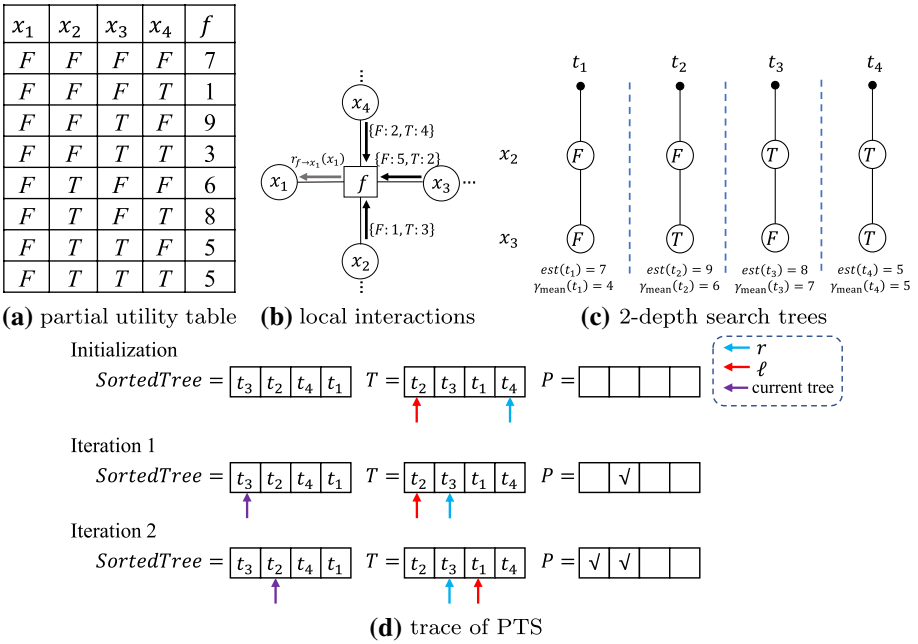


Fig. 9 The trace of PTS

$$\gamma_{\text{H-utility}}(f_j(e, x_i = v_i, \cdot)) = \mathcal{U}^e_{f_j}[h].$$

Like Q_3 value criterion, this criterion also operates on the sorted utility list, which requires $O((n-K)d^{n-K-1})$ operations and $O(d^{n-K-1})$ space.

As an example, consider the following utility values.

$$0 \quad 1 \quad 3 \quad 5 \quad 8 \quad 10 \quad 2 \quad 6 \quad 4 \quad 7 \quad 9 \quad 4 \quad 5 \quad 7 \quad 3$$

The maximum utility value criterion returns $\gamma_{\max} = 10$, while the mean utility value is $\gamma_{\text{mean}} = 4.93$. For Q_3 value and H-utility criteria, we need to first sort the utility values. Figure 8 gives the sorted utility list $\mathcal{U}$.

The third quartile index $m = \lfloor 15 \times 0.75 \rfloor = 11$ and $\alpha = 15 \times 0.75 - 11 = 0.25$. Therefore, $\gamma_{Q_3} = 7 + 0.25 \times (7-7) = 7$. For H-utility criterion, assume that $\delta = 10$ and $f^- = 0$. We iterate over the sorted utility list and find the H-position $h = 7$ since $1 - 7/15 > 4/10$ and $1 - 8/15 < 5/10$. Therefore, $\gamma_{\text{H-utility}} = 4$.

6.4 Built-in termination detection mechanism

Since the search trees now are not necessarily ranked according to the maximum utility value in their subspace, we cannot directly perform domain pruning and discard the remaining search space when the estimation of current search tree is less than the lower bound like ST-GD^{2}P (line 11 of Alg.2). In fact, when detected a suboptimal search tree, the only thing we can do in PTS is to simply skip the tree and continue to explore the next search tree (line 12, 16 of Alg.3). In other words, to compute a response utility for a

particular assignment, we must visit all search trees related to the assignment, which is not desirable when the number of trees is large.

Therefore, we develop a termination detection mechanism to enable domain pruning with few extra overheads. Alg.4 presents the sketch of the proposed mechanism.

Algorithm 4: Termination detection mechanism for K-depth PTS

Input: Target variable $x_i \in \mathbf{x_j}$, assignment $v_i \in D_i$
When `Initialization`:
1 $T \leftarrow$ sort the search trees in $SortedTree_j^i(v_i)$ according to their estimation
2 $P \leftarrow$ an array of length $|T|$
When *starting to compute the response utility*:
3 $\ell \leftarrow 1, r \leftarrow |T|$
4 clear the flags in P
When *an search tree t is exhausted*:
5 $i \leftarrow$ the index of t in T
6 mark $P[i]$ as visited
7 **while** $P[\ell]$ *is visited* **do**
8 $\ell \leftarrow \ell + 1$
When *lower bound lb is updated*:
9 $r \leftarrow$`Binary-search`(lb, ℓ, r)
When *querying termination condition*:
10 **if** $\ell > r$ **then**
11 **return** *"Terminate"*

The general idea behind the mechanism is keeping track of the set of visited search trees. If all search trees with estimation higher than current lower bound are visited, then PTS can terminate safely. To do this efficiently, we need to maintain another ordered search tree list T in which trees are sorted by their estimation (line 1), and a record array P. When PTS is invoked to compute a response utility, the mechanism clears the content of record array P, and initializes two pointers ℓ, r to the first position and the last position of T, respectively (line 3–4). After exhausting a search tree t, we mark the corresponding position as visited (line 5–6), and update the pointer ℓ to exclude all visited search trees (line 7–8). When the lower bound lb is updated, we update the pointer r by finding the last search tree in T whose estimation is no less than lb via binary search (line 9). Finally, PTS can terminate if $\ell > r$ (line 10–11).

Consider the example shown in Fig. 9. Assume that we use 2-depth partial tree-sorting scheme with sorting criterion γ_{mean} to compute response utility for $x_1 = F$. Figure 9c, d give 2-depth search trees and sorted result (i.e., *SortedTree*), respectively. The algorithm begins with sorting the search trees by their estimation (i.e., T) and constructing an empty array P (the first row of Fig. 9d). After exhausting the first search tree t_3, PTS updates lower bound $lb = 8$ (line 13–15 of Alg.3). Therefore, termination detection mechanism marks t_3 as visited (line 5–6), and finds a new right pointer $r = 2$ which corresponds to the last search tree with estimation no less than 8 (line 9). The results are presented in the second row of Fig. 9d. Finally, after exhausting t_2, the mechanism marks the corresponding position as visited and updates the left pointer $\ell = 3$ (line 5–8). Since $\ell > r$, the algorithm terminates and prunes both t_1 and t_4.

We now are ready to show the correctness of PTS.

Theorem 6 *PTS guarantees the optimality of Eq.* (2).

Proof There are three lines that prune the search space, i.e., line 11–13 of Alg.3. Since we have already proved that line 12 and line 13 do not affect the optimality of Eq. (2) in Theorem 5, we only need to show that line 11 does not prune the optimal assignment.

Assume that the optimal assignment A^* belongs to a search tree t^* which is pruned by line 11 of Alg.3. Denote the index of t^* in T as i^*. According to line 10-11 of Alg.4, it must be the case that $i^* > r$ by the end of execution since otherwise PTS must have visited t^* (line 6–8 of Alg.4), which implies that the estimation of t^* is less than lb (line 9 of Alg.4). That is,

$$f_j(A^*) \leq est(t^*) < lb.$$

According to line 15 of Alg.3, there must be an assignment $A' \neq A^*$ such that

$$\begin{aligned} f_j(A^*) &< f_j(A') + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A'[x_k]) - \max_{v_k \in D_k} q_{x_k \to f_j}(v_k) \\ &< f_j(A') + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A'[x_k]) - q_{x_k \to f_j}(A^*[x_k]) \end{aligned},$$

which implies

$$f_j(A^*) + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A^*[x_k]) < f_j(A') + \sum_{x_k \in \mathbf{x_j} \setminus \{x_i\}} q_{x_k \to f_j}(A'[x_k]).$$

Therefore, A^* cannot be an optimal assignment and PTS guarantees to return the optimal utility values. □

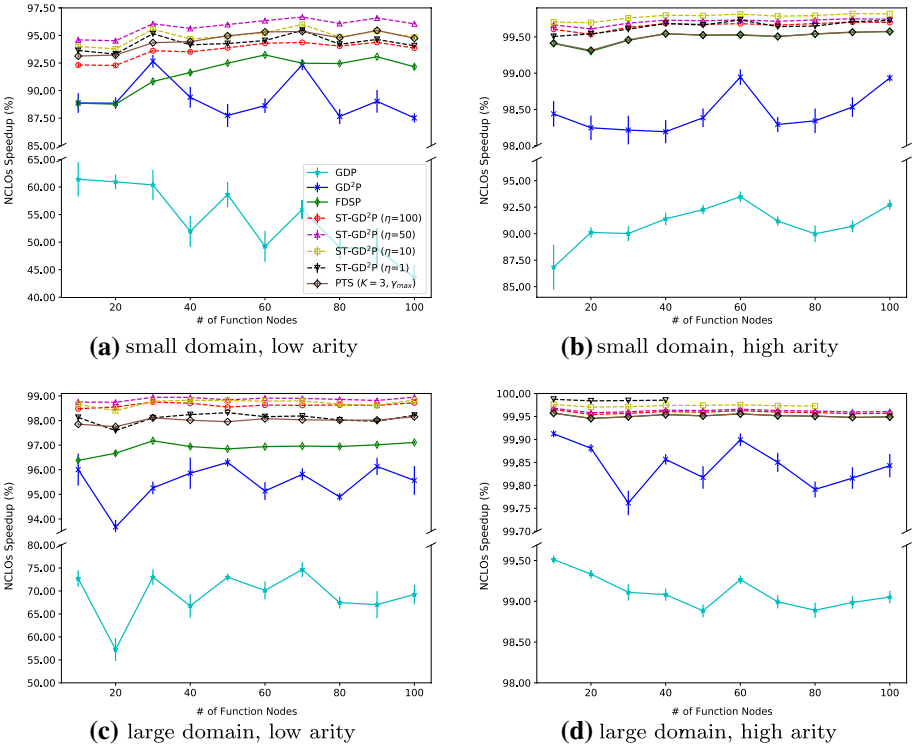
7 Empirical evaluations

In this section, we empirically evaluate the proposed algorithms when accelerating Max-sum on various standard benchmark problems. We first present the results on random n-ary DCOPs to show our superiorities on general problems. To examine the performance on the problems with dense local utility values, we conduct experiments on the problems in which utility values follow a power-law distribution. Finally, we present the results on realistic benchmarks including NetRad systems and channel allocation problems.

7.1 Experimental configurations

We benchmark algorithms with three types of problems, i.e., random n-ary DCOPs, NetRad system problems and channel allocation problems.

- *Random n-ary DCOPs* are the general form of the n-ary distributed constraint optimization problems. In the experiments, we vary the number of function nodes from 10 to 100. For each function node, we randomly sample the arity n from either [2,5] (low arity problems) or [5,8] (high arity problems), and connect the function node to n variable nodes. The total number of variable nodes is specified by variable tightness [5], which is randomly picked from either [0.1,0.5] (sparse problems) or



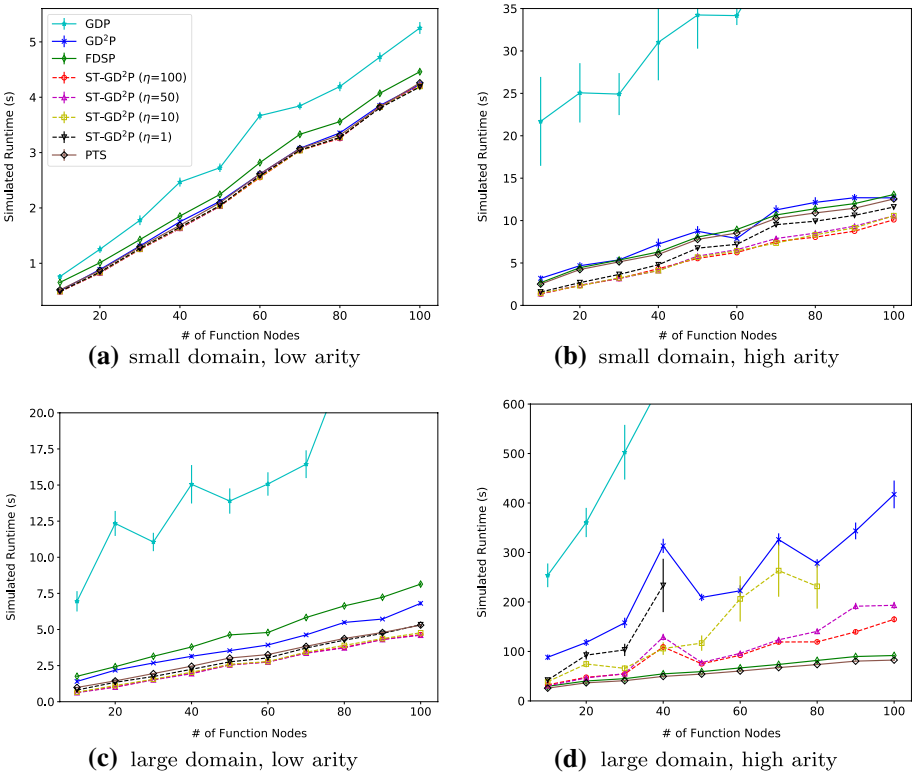
(a) small domain, low arity **(b)** small domain, high arity

(c) large domain, low arity **(d)** large domain, high arity

Fig. 10 NCLOs speedup of different algorithms on sparse random *n*-ary DCOPs

[0.5,0.9] (dense problems). Further, we generate large domain problems and small domain problems by uniformly selecting a domain size from [5,8] and [2,5] for each variable, respectively. Finally, for each function node, the utility values are sampled either uniformly or according to a power-law from the range of [0, 1000].

- *NetRad system problems* [22] aim to find a joint scanning strategy to maximize the total utility of scanning weather phenomena. Specifically, each radar is modelled as a variable whose assignment represents its scanning strategy (i.e., any combination of *E,N,W,S*). Weather phenomena with different sizes, weights and types are randomly generated across grids. Each phenomenon can be sensed by a subset of radars, and the scanning utility is determined by the joint strategy of the radars. In our experiments, the utility functions are defined according to [39]. We consider the NetRad systems with 48 and 96 radars which are arranged into 6×8 and 8×12 grids, respectively. In this set of experiments, the maximal utility of a function is 1, the maximal arity is 4 and the domain size of a variable can go up to 15.
- *Channel allocation problems* [37] try to coordinate a set of access points (APs) to minimize the radio interference. Let x_i be the channel allocated to AP i, P_i be the signal strength of i at the source, d_{ij} be the distance between AP i and j, and $Ap_i = \{j | P_j > N_b d_{ij}^2\}$ be the set of APs which can cause interference to the signal of i with background noise N_b. Then the utility function for AP i is modelled as



(a) small domain, low arity **(b)** small domain, high arity

(c) large domain, low arity **(d)** large domain, high arity

Fig. 11 Simulated runtime of different algorithms on sparse random n-ary DCOPs

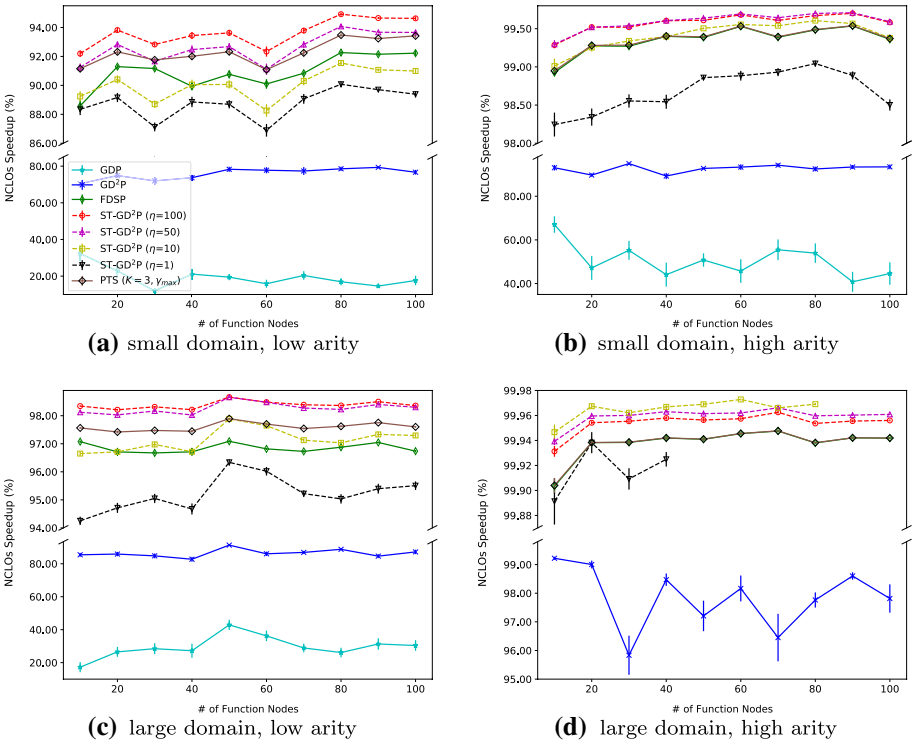
$$f_i(\mathbf{x_i}) = W \log_2 \left(1 + \frac{P_i}{1 + \sum_{j \in Ap_i} \frac{P_j}{d_{ij}^2} \mathbb{I}_{|x_i - x_j| \leq 3}} \right), \tag{11}$$

where W is a constant and $\mathbb{I}$ is the indicator function. In our experiments, we consider a 300×300 map, 10 available channels, and vary the number of APs from 60 to 100. For each AP i, we uniformly pick its power P_i from the range [490,510]. Finally, we set $W = 20$, and generate sparse problems and dense problems by setting $N_b = 1$ and $N_b = 0.5$, respectively.

The baselines we consider are GDP and FDSP since they are domain-independent state-of-the-art acceleration methods for belief propagation algorithms. Since channel allocation problems is a type of cardinality factors, we also consider THOP-based method [48] as an additional baseline.[5]

The performance metrics we consider include NCLOs [34] speedup, pruned rate [21] and simulated runtime [47]. We consider a logical operation to be an access to a query

[5] The implementation of compared baselines and our proposed algorithms can be found in https://github.com/dyc941126/ARTGD2P.



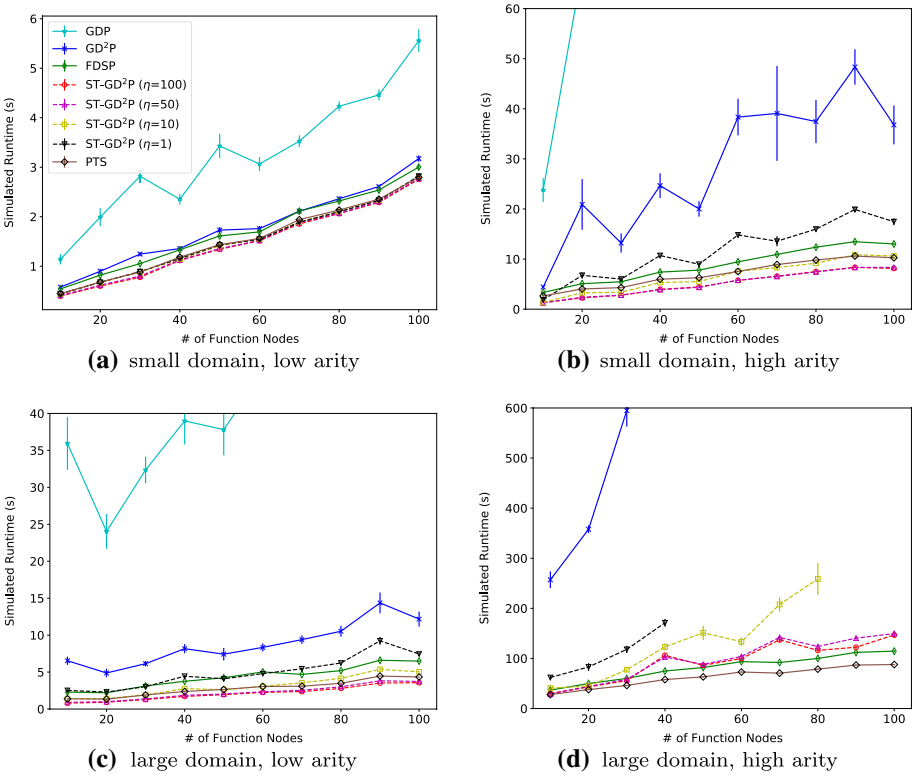
(a) small domain, low arity **(b)** small domain, high arity

(c) large domain, low arity **(d)** large domain, high arity

Fig. 12 NCLOs speedup of different algorithms on dense random n-ary DCOPs

message, and compute NCLOs speedup of an acceleration algorithm by dividing its total number of non-concurrent logical operations in an execution by the number of non-concurrent logical operations in vanilla Max-sum running for the same iterations. Then the NCLOs speedup is the remaining after subtracting the quotient from 1. This metric reflects the general acceleration capability of an algorithm in a distributed environment. Besides, we also consider the percentage of search space pruned (i.e., pruned rate) which is the improvement in terms of the total number of message accesses incurred in an execution. This metric drops the factor of concurrency and assesses purely the pruning capability of an algorithm. Finally, the simulated runtime directly reflects the overall performance of an acceleration algorithm in a distributed environment.

For each experiment, we generate 50 random instances and terminate algorithms after 2000 iterations with the timeouts of 1800s (in terms of simulated runtime), reporting the averaged metrics and standard errors as the results and the confidence intervals, respectively. For each instance, we host each function node randomly in one of its involved agents. All the experiments are conducted on a 32-cores Linux workstation with 384 GB memory. Finally, to avoid the potential precision loss incurred by floating-point numbers during an execution, we cast the utility values to big integers by multiplying by 10^6. That is important for a fair comparison since it ensures that every algorithm has the same query messages in every iteration and solves exactly the same optimization problem of Eq. (2) when computing response messages.



(a) small domain, low arity **(b)** small domain, high arity

(c) large domain, low arity **(d)** large domain, high arity

Fig. 13 Simulated runtime of different algorithms on dense random n-ary DCOPs

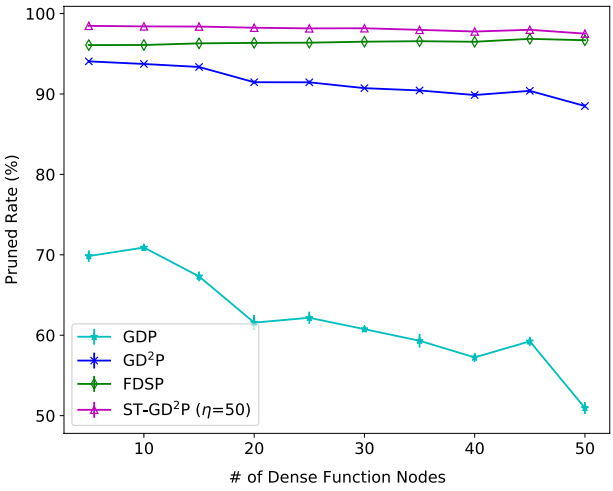
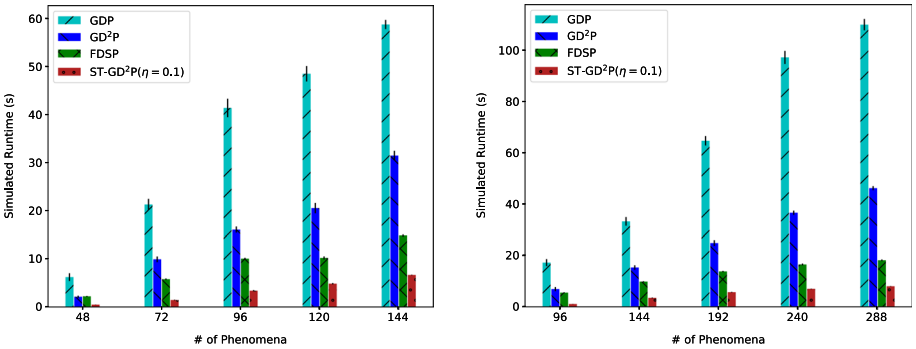


Fig. 14 Performance comparison on the problems with dense utility functions



(a) 48 radars **(b)** 96 radars

Fig. 15 Simulated runtime of different algorithms on NetRad system problems

7.2 Evaluations of dynamic domain pruning techniques

Figures 10 and 11 present the results when solving sparse random n-ary DCOPs. It can be seen that FDSP performs worse than ST-GD^{2}P in terms of NCLOs speedup. That is due to the fact that FDSP performs a depth-first search on a huge search tree without any *a priori* knowledge. On the other hand, our ST-GD^{2}P performs branch and bound on a sorted array of search trees and can find an efficient lower bound more promptly, reducing about 95% - 99.9% of operations, which demonstrates great superiorities over the other competitors. It can be clearly seen from Fig. 11 that the simulated runtime of all ST-GD^{2}P variants is lower than the one of FDSP, except the problems with large domain and high arity due to the excessive preprocessing overhead of building search trees. Nonetheless, combining with K-depth partial tree-sorting scheme, our proposed PTS requires the lowest runtime on the problems with large domain and high arity.

Besides, our GD^{2}P maintains a running lower bound and significantly improves the performance of GDP, which highlights the importance of tight lower bounds in domain pruning techniques. In fact, GD^{2}P requires runtime lower than FDSP when solving the problems with low arity (i.e., Fig. 11a, c). That might be due to the extra overhead FDSP needs to query function estimation to compute the upper bound for each partial assignment. Finally, step size plays an important role in the overall performance of ST-GD^{2}P. In more detail, ST-GD^{2}P with an appropriate step size (e.g., $\eta = 50$) has a higher NCLOs speedup than other versions of ST-GD^{2}P. That is because the algorithm with a small step size (e.g., $\eta = 1$) tends to build small search trees which lead to frequently solution reconstruction, while a large step size (e.g., $\eta = 100$) would produces coarse utility slots, preventing the algorithm from pruning suboptimal solutions promptly. Besides, a small step size can also incur a large preprocessing overhead. In fact, as shown in Figs. 10d and 11d, ST-GD^{2}P with $\eta = 1$ and $\eta = 10$ have significantly higher simulated runtime than other variants, and run out of memory when solving the large-domain, high-arity problems with more than 40 and 80 function nodes, respectively.

Figures 12 and 13 present the results when solving dense random n-ary DCOPs. Compared to the ones in sparse problems, the variable nodes in these problems are over-constrained due to high variable tightness. Specifically, each variable node in this set of problems connects to 4 function nodes on average. As a result, query messages are amplified quickly due to the excessive cyclic information propagation in dense problems [54] as

Table 2 Pruned rate (%) of PTS ($\eta = 0.1$) with different sorting criteria and depths on sparse channel allocation problems

# of access points		60	70	80	90	100
Average arity		3.24	3.28	3.29	3.39	3.51
$\gamma_{\max}$	$K = 1$	99.27 ± 0.212	99.294 ± 0.284	99.361 ± 0.239	99.717 ± 0.076	99.747 ± 0.077
	$K = 2$	99.396 ± 0.165	99.422 ± 0.216	99.462 ± 0.186	99.746 ± 0.066	99.779 ± 0.063
	$K = 3$	99.35 ± 0.178	99.349 ± 0.261	99.424 ± 0.192	99.723 ± 0.072	99.764 ± 0.064
γ_{mean}	$K = 1$	99.343 ± 0.191	99.354 ± 0.258	99.42 ± 0.219	99.737 ± 0.073	99.778 ± 0.067
	$K = 2$	99.41 ± 0.164	99.427 ± 0.215	99.47 ± 0.185	99.75 ± 0.067	99.786 ± 0.061
	$K = 3$	99.347 ± 0.18	99.34 ± 0.264	99.413 ± 0.197	99.717 ± 0.074	99.761 ± 0.065
γ_{Q_3}	$K = 1$	99.341 ± 0.193	99.353 ± 0.258	99.414 ± 0.225	99.735 ± 0.074	99.777 ± 0.068
	$K = 2$	**99.417 ± 0.163**	**99.436 ± 0.212**	**99.476 ± 0.183**	**99.751 ± 0.066**	**99.789 ± 0.06**
	$K = 3$	99.348 ± 0.181	99.346 ± 0.262	99.421 ± 0.193	99.721 ± 0.073	99.765 ± 0.064
$\gamma_{\text{H-utility}}$	$K = 1$	99.303 ± 0.209	99.313 ± 0.286	99.395 ± 0.232	99.731 ± 0.076	99.772 ± 0.07
	$K = 2$	99.229 ± 0.245	99.242 ± 0.347	99.368 ± 0.238	99.721 ± 0.078	99.765 ± 0.07
	$K = 3$	98.974 ± 0.329	98.96 ± 0.513	99.189 ± 0.303	99.64 ± 0.104	99.699 ± 0.092
FDSP		99.171 ± 0.257	99.148 ± 0.383	99.29 ± 0.29	99.702v0.083	99.733 ± 0.086
ST-GD^{2}P ($\eta = 0.1$)		97.8 ± 0.549	98.471 ± 0.286	98.744 ± 0.112	99.013 ± 0.11	99.17 ± 0.073

Bold values indicate the best results

Max-sum proceeds, which may widen the difference between the maximum message utility and the message utility corresponding to the assignment with the highest local utility value. Therefore, GDP needs to explore more entries to cover the gap when solving a dense problem, and fails to solve the large-domain, high-arity problem in reasonable time. In fact, comparing to Fig. 10 we can clearly see that the performance of GDP degenerates significantly and is strictly dominated by other competitors. In contrast, our dynamic domain pruning techniques are more robust to the high variable tightness due to the continuously tightening lower bound. Besides, it is interesting to find that different from the results on sparse problems, ST-GD^{2}P with small step size (e.g., $\eta = 1$) is dominated by FDSP when solving high-arity problems in terms of simulated runtime (i.e., Fig. 13b, d). In addition to high preprocessing overhead, another possible reason for this could be that FDSP performs a depth-first search on search trees, which significantly reduces solution reconstructions. However, by limiting the depth of sorted tree, our proposed PTS reduces both preprocessing overhead and solution reconstruction overhead, which demonstrates its merits on general n-ary DCOPs.

Table 3 Pruned rate (%) of PTS ($\eta = 0.1$) with different sorting criteria and depths on dense channel allocation problems

# of access points		60	70	80	90	100
Average arity		3.64	3.69	3.95	4.14	4.13
$\gamma_{\max}$	$K = 1$	99.559 ± 0.272	99.824 ± 0.106	99.879 ± 0.031	99.89 ± 0.027	99.878 ± 0.046
	$K = 2$	99.639 ± 0.208	99.85 ± 0.087	99.893 ± 0.027	99.903 ± 0.022	99.893 ± 0.038
	$K = 3$	99.653 ± 0.199	99.856 ± 0.082	99.894 ± 0.024	99.906 ± 0.02	99.896 ± 0.034
γ_{mean}	$K = 1$	99.57 ± 0.273	99.842 ± 0.097	99.891 ± 0.028	99.903 ± 0.024	99.894 ± 0.038
	$K = 2$	99.625 ± 0.225	99.854 ± 0.087	99.896 ± 0.027	**99.909 ± 0.021**	99.9 ± 0.035
	$K = 3$	99.644 ± 0.206	99.853 ± 0.086	99.892 ± 0.027	99.905 ± 0.021	99.897 ± 0.033
γ_{Q_3}	$K = 1$	99.578 ± 0.266	99.842 ± 0.097	99.891 ± 0.029	99.903 ± 0.024	99.894 ± 0.039
	$K = 2$	99.651 ± 0.202	99.857 ± 0.084	**99.897 ± 0.027**	**99.909 ± 0.021**	**99.902 ± 0.033**
	$K = 3$	**99.655 ± 0.199**	**99.857 ± 0.082**	99.893 ± 0.026	99.906 ± 0.021	99.898 ± 0.032
$\gamma_{\text{H-utility}}$	$K = 1$	99.552 ± 0.287	99.835 ± 0.104	99.89 ± 0.029	99.902 ± 0.025	99.89 ± 0.042
	$K = 2$	99.542 ± 0.295	99.835 ± 0.105	99.89 ± 0.029	99.903 ± 0.024	99.891 ± 0.041
	$K = 3$	99.416 ± 0.388	99.799 ± 0.133	99.874 ± 0.032	99.89 ± 0.028	99.875 ± 0.048
FDSP		99.464 ± 0.349	99.804 ± 0.129	99.877 ± 0.032	99.886 ± 0.031	99.875 ± 0.045
ST-GD^{2}P ($\eta = 0.1$)		99.263 ± 0.211	99.346 ± 0.142	99.573 ± 0.067	99.566 ± 0.07	–

Bold values indicate the best results

To examine the general pruning capability on the problems with dense utility functions, we consider the problems with 100 function nodes, low arity, large domain size and the variable tightness of 0.5. For each problem, there are a number of dense function nodes whose utility values are selected according to a power-law distribution. More specifically, the probability of selecting a utility $u \in (0, 1000)$ is proportional to $(1000 - u)^{-\alpha}$. In our experiments, we set $\alpha = 1.1$. Figure 14 presents the performance comparison in terms of the pruned rate.

It can be seen that the performance of GDP decreases by near 20% w.r.t. growing dense function nodes, which indicates that GDP is sensitive to dense utility functions. That is due to the fact that the pruned range could contain many entries since the utility values are close to each other in dense utility functions. On the other hand, although GD^{2}P also relies on domain pruning to reduce the search space, it is much more robust to the presence of dense function nodes due to the iteratively tighten lower bound. In fact, with the growing of dense function nodes its performance only drops by 4%. Finally, dense function nodes can hardly deteriorate the performance of ST-GD^{2}P since it can still perform

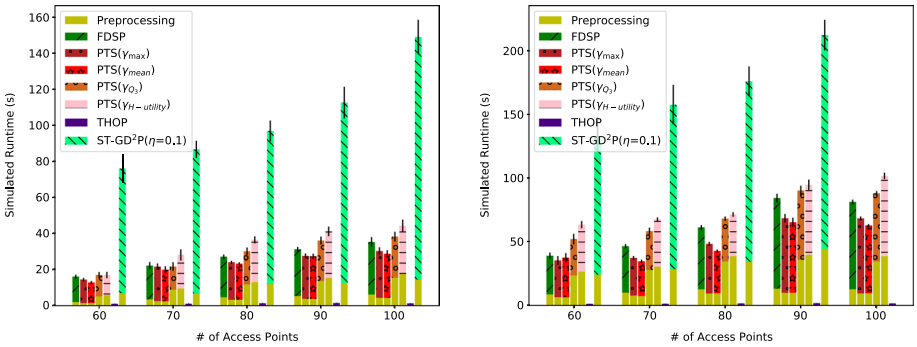


Fig. 16 Simulated runtime of PTS ($\eta = 0.1$) different sorting criteria on channel allocation problems

effective branch-and-bound according to the query messages when local utility values are not distinguishable.

Figure 15 presents the results on NetRad system problems. It can be concluded that GDP performs poorly and the runtime grows quickly w.r.t. the number of phenomena in both cases. In contrast, our proposed GD^{2}P and ST-GD^{2}P significantly outperform GDP, only requiring about a half and one-tenth of its runtimes, respectively. On the other hand, although FDSP outperforms domain pruning variants, it is still dominated by ST-GD^{2}P. This is due to the fact that with the sorted search trees our proposed ST-GD^{2}P can find a high-quality lower bound quickly despite of highly-structured utility functions, which highlights the importance of search space sorting.

7.3 Evaluations of partial tree-sorting scheme variants

Table 2 and 3 tabulate the pruned rate of PTS with different sorting criteria and sorting depths on channel allocation problems. Compared to NetRad system problems, this set of problems are more challenging due to higher arity and more structured utility functions. In fact, both GDP and GD^{2}P fail to terminate in reasonable time. Therefore, we only present the results of PTS and FDSP. It can be concluded that both sorting depth K and sorting criterion γ can significantly affect the pruned rate. Specifically, PTS with a small K still can outperform FDSP. That is because FDSP sequentially explores all possible combinations starting from an all-one assignment, which is very inefficient in channel allocation problems since the first 4^{n-1} assignments correspond to low local utility values according to Eq. (11). On the other hand, our PTS tries to improve by ranking the most promising search trees at the top according to their weights, so as to find a high-quality lower bound more promptly. Therefore, a small permutation (e.g., $K = 1$) can greatly reduce the search effort, and the benefit increases as the preprocessing budget K is increased. Besides, ST-GD^{2}P fails to solve the dense problems with 100 APs since it runs out of memory, and exhibits the worst performance on all remaining test cases. That might due to the large number of search trees after partitioning the search space of large factors. Consequently, solution reconstruction happens frequently and offsets the advantage of completely sorting.

For sorting criterion, it can be seen that directly using the maximum utility value to rank the search trees performs poorly when the sorting depth is small. That is not surprising since $\gamma_{\max}$ completely ignores the quantity of utility values in the remaining search space.

However, the performance of $\gamma_{\max}$ is generally improved when K is large, especially on dense problems. That is because the size of the remaining search space is reduced if the search trees include more variables, and therefore the effect of quantity on pruning efficiency is reduced. Differently, γ_{mean} takes quantity into consideration, achieving better performance when K is small. However, its performance does not necessarily grow as the sorting depth increases. In fact, γ_{mean} is inferior to $\gamma_{\max}$ on the most of test cases when $K = 3$. Quartile criterion γ_{Q_3} considers the fine-grained statistical characteristics of utility value distribution, and is superior to all compared criteria as well as FDSP when $K = 2$. Finally, although $\gamma_{\text{H-utility}}$ tries to capture simultaneously the quality and quantity of utility values, it is strictly dominated by γ_{mean}. A possible reason could be that $\gamma_{\text{H-utility}}$ assumes a uniform distribution of utility values in calculating H-position, which may not be applicable to the highly-structured utility function of channel allocation problems.

Figure 16 presents simulated runtime of PTS with different sorting criteria on channel allocation problems when $K = 2$. Notably, THOP-based method exhibits the lowest runtime on all test cases. That is due to the fact that it only requires several linear scans over involved variables to perform sorting and dynamic programming when computing a response message. However, as we discussed earlier, THOP-based methods have a limited generality since it requires a utility function can be represented by a specific type of THOPs. Arguably, applying THOP-based methods also requires laborous expertise to identify the pattern of each utility function to implement a specific THOP-based method. Instead, our proposed algorithms make no assumption about the utility function and thus can be applied to general settings. It can be seen that compared to FDSP, PTS with $\gamma_{\max}$ and γ_{mean} has a smaller preprocessing overhead. That is because the calculation of maximum and mean utility value only requires linear time to scan entries, while FDSP needs to perform multiple phases of dynamic programming to compute the function estimations for each involved variables. On the other hand, the preprocessing phase of PTS with γ_{Q_3} and $\gamma_{\text{H-utility}}$ takes more time than FDSP. This is due to the higher complexity of utility sorting. In addition to lower preprocessing overhead, it is noteworthy that our proposed PTS with $\gamma_{\max}$ and γ_{mean} also significantly outperforms FDSP in terms of total runtime. In fact, our PTS is about 11% and 16% faster than FDSP on sparse problems when using $\gamma_{\max}$ and γ_{mean}, respectively.

8 Conclusion

Max-sum and its variants are important belief propagation algorithms for solving DCOPs but suffer from a high computational complexity. In this paper, we demonstrate that the state-of-the-art algorithms for accelerating Max-sum could perform poorly when dealing with the problems with dense utility functions. To alleviate the negative effect of densely distributed utility values, we propose to iteratively update the lower bound and organize the tied entries to search trees. Further, we present a discretization mechanism to offer a tradeoff between the reconstruction overhead and domain pruning efficiency. Finally, we propose a K-depth partial tree-sorting scheme with different sorting criteria to reduce the preprocessing overhead. We theoretically show the correctness and superiority of our proposed algorithms. The extensive empirical evaluations confirm the advantages over the state-of-the-art methods on both synthetic and realistic benchmarks.

We note that all the existing acceleration methods are based on static knowledge and fail to exploit the characteristics of GDL algorithms or runtime information. For example,

query messages in Damped Max-sum could change slowly when damping factor is high. Therefore, one can leverage such similarity between two consecutive iterations and give priority to search the neighborhood of optimal assignment found in the last iteration. In our future work, we plan to utilize the algorithmic characteristics or running history to further improve the performance of our algorithms.

Acknowledgements This research was supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG-RP-2019-0013), National Satellite of Excellence in Trustworthy Software Systems (Award No: NSOE-TSS2019-01), and NTU.

References

1. Aji, S. M., & McEliece, R. J. (2000). The generalized distributive law. *IEEE Transactions on Information Theory, 46*(2), 325–343.
2. Atlas, J., Warner, M., & Decker, K. (2008). A memory bounded hybrid approach to distributed constraint optimization. In *Proceedings of 10th international workshop on distributed constraint reasoning* (pp. 37–51).
3. Chen, D., Deng, Y., Chen, Z., Zhang, W., & He, Z. (2020). HS-CAI: A hybrid DCOP algorithm via combining search with context-based inference. In *AAAI* (pp. 7087–7094).
4. Chen, Z., Deng, Y., Wu, T., & He, Z. (2018). A class of iterative refined max-sum algorithms via non-consecutive value propagation strategies. *Autonomous Agents and Multi-Agent Systems, 32*(6), 822–860.
5. Chen, Z., Jiang, X., Deng, Y., Chen, D., & He, Z. (2019). A generic approach to accelerating belief propagation based incomplete algorithms for DCOPs via a branch-and-bound technique. In *AAAI* (pp. 6038–6045).
6. Chen, Z., Zhang, W., Deng, Y., Chen, D., & Li, Q. (2020). RMB-DPOP: Refining MB-DPOP by reducing redundant inference. In *AAMAS* (pp. 249–257).
7. Cohen, L., Galiki, R., & Zivan, R. (2020). Governing convergence of max-sum on DCOPs through damping and splitting. *Artificial Intelligence, 279,* 103212.
8. Dechter, R., & Mateescu, R. (2007). AND/OR search spaces for graphical models. *Artificial Intelligence, 171*(2–3), 73–106.
9. Deng, Y., & An, B. (2020). Speeding up incomplete GDL-based algorithms for multi-agent optimization with dense local utilities. In *IJCAI* (pp. 31–38).
10. Farinelli, A., Rogers, A., Petcu, A., & Jennings, N.R. (2008). Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *AAMAS* (pp. 639–646).
11. Fioretto, F., Yeoh, W., Pontelli, E., Ma, Y., & Ranade, S. J. (2017). A distributed constraint optimization (DCOP) approach to the economic dispatch with demand response. In *AAMAS* (pp. 999–1007).
12. Freuder, E. C., & Quinn, M. J. (1985). Taking advantage of stable sets of variables in constraint satisfaction problems. *IJCAI, 85,* 1076–1078.
13. Gershman, A., Meisels, A., & Zivan, R. (2009). Asynchronous forward bounding for distributed COPs. *Journal of Artificial Intelligence Research, 34,* 61–88.
14. Givoni, I. E., & Frey, B. J. (2009). A binary variable model for affinity propagation. *Neural Computation, 21*(6), 1589–1600.
15. Hirayama, K., Miyake, K., Shiotani, T., & Okimoto, T. (2019). DSSA+: Distributed collision avoidance algorithm in an environment where both course and speed changes are allowed. *International Journal on Marine Navigation and Safety of Sea Transportation, 13*(1), 117–123.
16. Hirayama, K., & Yokoo, M. (1997). Distributed partial constraint satisfaction problem. In *CP* (pp. 222–236).
17. Hirayama, K., & Yokoo, M. (2005). The distributed breakout algorithms. *Artificial Intelligence, 161*(1–2), 89–115.
18. Hirsch, J. E. (2005). An index to quantify an individual's scientific research output. *Proceedings of the National academy of Sciences, 102*(46), 16569–16572.
19. Hoang, K.D., Fioretto, F., Yeoh, W., Pontelli, E., & Zivan, R. (2018). A large neighboring search schema for multi-agent optimization. In *CP* (pp. 688–706).
20. Katagishi, H., & Pearce, J.P. (2007). KOPT: Distributed DCOP algorithm for arbitrary k-optima with monotonically increasing utility. In *DCR workshop*.

21. Khan, M.M., Tran-Thanh, L., & Jennings, N.R. (2018). A generic domain pruning technique for GDL-based DCOP algorithms in cooperative multi-agent systems. In *AAMAS* (pp. 1595–1603).
22. Kim, Y., Krainin, M., & Lesser, V. (2011). Effective variants of the max-sum algorithm for radar coordination and scheduling. In *WI/IAT* (pp. 357–364).
23. Kim, Y., & Lesser, V. (2013). Improved max-sum algorithm for DCOP with n-ary constraints. In *AAMAS* (pp. 191–198).
24. Kim, Y., & Lesser, V. (2014). DJAO: A communication-constrained DCOP algorithm that combines features of ADOPT and Action-GDL. In *AAAI* (pp. 2680–2687).
25. Komodakis, N., & Paragios, N. (2009). Beyond pairwise energies: Efficient optimization for higher-order mrfs. In *CVPR* (pp. 2985–2992). IEEE.
26. Kschischang, F. R., Frey, B. J., Loeliger, H. A., et al. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory, 47*(2), 498–519.
27. Li, S., Negenborn, R. R., & Lodewijks, G. (2016). Distributed constraint optimization for addressing vessel rotation planning problems. *Engineering Applications of Artificial Intelligence, 48,* 159–172.
28. Litov, O., & Meisels, A. (2017). Forward bounding on pseudo-trees for DCOPs and ADCOPs. *Artificial Intelligence, 252,* 83–99.
29. Macarthur, K.S., Stranders, R., Ramchurn, S.D., & Jennings, N.R. (2011). A distributed anytime algorithm for dynamic task allocation in multi-agent systems. In *AAAI* (pp. 701–706).
30. Maheswaran, R. T., Pearce, J. P., & Tambe, M. (2004). Distributed algorithms for DCOP: A graphical-game-based approach. In *ISCA PDCS* (pp. 432–439).
31. Marinescu, R., & Dechter, R. (2009). AND/OR branch-and-bound search for combinatorial optimization in graphical models. *Artificial Intelligence, 173*(16–17), 1457–1491.
32. Modi, P. J., Shen, W. M., Tambe, M., & Yokoo, M. (2005). ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence, 161*(1–2), 149–180.
33. Monteiro, T. L., Pujolle, G., Pellenz, M. E., Penna, M. C., & Souza, R. D. (2012). A multi-agent approach to optimal channel assignment in WLANs. In *WCNC* (pp. 2637–2642).
34. Netzer, A., Grubshtein, A., & Meisels, A. (2012). Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence, 193,* 186–216.
35. Nguyen, D. T., Yeoh, W., Lau, H. C., & Zivan, R. (2019). Distributed Gibbs: A linear-space sampling-based DCOP algorithm. *Journal of Artificial Intelligence Research, 64,* 705–748.
36. Okamoto, S., Zivan, R., & Nahon, A. (2016). Distributed breakout: Beyond satisfaction. In *IJCAI* (pp. 447–453).
37. Ottens, B., Dimitrakakis, C., & Faltings, B. (2017). DUCT: An upper confidence bound approach to distributed constraint optimization problems. *ACM Transactions on Intelligent Systems and Technology, 8*(5), 69:1-69:27.
38. Pearce, J.P., & Tambe, M. (2007). Quality guarantees on k-optimal solutions for distributed constraint optimization problems. In *IJCAI* (pp. 1446–1451).
39. Pepyne, D., Westbrook, D., Philips, B., Lyons, E., Zink, M., & Kurose, J. (2007). *A design for distributed collaborative adaptive sensing of the atmosphere*. Tech. rep. University of Massachusetts.
40. Petcu, A., & Faltings, B. (2005). A scalable method for multiagent constraint optimization. In *IJCAI* (pp. 266–271).
41. Petcu, A., & Faltings, B. (2007). MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *IJCAI* (pp. 1452–1457).
42. Pujol-Gonzalez, M., Cerquides, J., Meseguer, P., Rodríguez-Aguilar, J. A., & Tambe, M. (2013). Engineering the decentralized coordination of uavs with limited communication range. In *CAEPIA* (pp. 199–208).
43. Rogers, A., Farinelli, A., Stranders, R., & Jennings, N. R. (2011). Bounded approximate decentralised coordination via the max-sum algorithm. *Artificial Intelligence, 175*(2), 730–759.
44. Rollon, E., & Larrosa, J. (2012). Improved bounded max-sum for distributed constraint optimization. In *CP* (pp. 624–632). Springer.
45. Rollon, E., & Larrosa, J. (2014). Decomposing utility functions in bounded max-sum for distributed constraint optimization. In *CP* (pp. 646–654). Springer.
46. Stranders, R., Farinelli, A., Rogers, A., & Jennings, N. R. (2009). Decentralised coordination of mobile sensors using the max-sum algorithm. In *IJCAI* (pp. 299–304).
47. Sultanik, E. A., Lass, R. N., & Regli, W. C. (2008). DCOPolis: A framework for simulating and deploying distributed constraint reasoning algorithms. In *AAMAS* (pp. 1667–1668).
48. Tarlow, D., Givoni, I., & Zemel, R. (2010). HOP-MAP: Efficient message passing with high order potentials. In *AISTAT* (pp. 812–819).
49. Vinyals, M., Rodriguez-Aguilar, J.A., & Cerquides, J. (2009). Generalizing DPOP: Action-GDL, a new complete algorithm for DCOPs. In *AAMAS* (pp. 1239–1240).

50. Weiss, Y., & Freeman, W. T. (2001). On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs. *IEEE Transactions on Information Theory, 47*(2), 736–744.
51. Yeoh, W., Felner, A., & Koenig, S. (2010). BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research, 38,* 85–133.
52. Yokoo, M., Durfee, E. H., Ishida, T., & Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on knowledge and data engineering, 10*(5), 673–685.
53. Zhang, W., Wang, G., Xing, Z., & Wittenburg, L. (2005). Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence, 161*(1–2), 55–87.
54. Zivan, R., Lev, O., & Galiki, R. (2020). Beyond trees: Analysis and convergence of belief propagation in graphs with multiple cycles. In *AAAI* (pp. 7333–7340).
55. Zivan, R., Okamoto, S., & Peled, H. (2014). Explorative anytime local search for distributed constraint optimization. *Artificial Intelligence, 212,* 1–26.
56. Zivan, R., Parash, T., Cohen, L., Peled, H., & Okamoto, S. (2017). Balancing exploration and exploitation in incomplete min/max-sum inference for distributed constraint optimization. *Autonomous Agents and Multi-Agent Systems, 31*(5), 1165–1207.