Implementation and Comparative Study of Domain Pruning Algorithms in DCOPs

Abdullah Al Mahmud
Department of Computer Science and Engineering
University of Dhaka
abdullahal-2020715625@cs.du.ac.bd

Zisan Mahmud
Department of Computer Science and Engineering
University of Dhaka
zisan-2020815633@cs.du.ac.bd

Abstract

This section should provide a concise summary of the entire work. Include the problem of domain pruning in DCOPs, the approach inspired by the IJCAI 2020 paper (*Speeding Up Incomplete GDL-based Algorithms for Multi-agent Optimization with Dense Local Utilities*, Deng and An, IJCAI 2020), the methodology of dataset preparation (synthetic or real-world), evaluation metrics used, and key findings.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) 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 agents.

Complete algorithms for DCOPs aim to find the optimal solution but incur exponential coordination overheads since solving DCOPs is NP-Hard. In contrast, incomplete algorithms trade the solution quality for smaller computational efforts and can scale up to large problems.

Max-sum and its variants are popular incomplete algorithms built upon the Generalized Distributive Law (GDL) and have been applied to many real-world domains due to their ability of directly handling n-ary constraints. However, these algorithms face a significant scalability challenge. In more detail, Max-sum implements belief propagation on a factor graph 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 large systems.

Therefore, a number of acceleration algorithms for GDL-based algorithms were proposed to improve their scalability and can be generally divided into BnB-based and sorting-based algorithms. BnB-based algorithms 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 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 and GDP [1] require (partially) sorted local utilities to perform acceleration. Specifically, G-FBP only sorts for top $cd^{n-1}/2$ values of the search space and presumes that the highest utility can be found in the range. 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 [1] constructs a completely sorted local utility list for each assignment of each variable. Then it uses the entry with the highest local utility to compute a one-shot lower bound to prune suboptimal entries.

However, the existing methods could perform poorly when dealing with dense local utilities. In more detail, FDSP would not be able to find a high-quality lower bound promptly when utility functions are highly structured since it sequentially exhausts the whole search space. On the other hand, although GDP [1] attempts to find a more efficient one by considering the local utilities, the one-shot nature still cannot guarantee the quality. Additionally, the pruned range returned by GDP [1] would contain many entries whose local utilities are close to each other, which also results in a poorly pruned rate. Unfortunately, dense utility functions and ties are very common in real-world scenarios. For example, in a NetRad system the utility of scanning a weather phenomenon does not increase linearly w.r.t. the scanning quality. As a result, the utility function could be extremely dense when the scanning quality is high.

In this report, we aim to cope with dense local utilities from the perspectives of both bound quality and search space organization and develop more efficient sorting-based acceleration algorithms. To ensure bound quality, we integrate both search and pruning by iteratively updating the lower bound. Then we overcome the inability of pruning tied entries in domain pruning techniques by organizing the search space of these entries into AND/OR trees to enable efficient branch-and-bound. Finally, we discretize the utility range to balance the reconstruction overhead and the pruning efficiency and to further reduce the sorting overhead. We theoretically show our algorithms are sound and outperform GDP [1] in terms of pruning efficiency. Our extensive empirical evaluations, which include implementations of Max-sum, GDP [1], FDSP, and GD2P [2], also confirm their great superiorities over the state-of-the-art.

2 Background

2.1 DCOP Overview

Distributed Constraint Optimization Problems (DCOPs) provide a powerful and versatile framework for modeling multi-agent coordination and optimization challenges across various domains. In a DCOP, a collective of autonomous agents collaboratively seeks to identify an assignment of values to variables that optimizes a global objective function. A formal characterization of a DCOP is typically given by a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A} \rangle$, where:

- $\mathcal{X} = \{x_1, \dots, x_n\}$ denotes a finite set of n variables. Each variable $x_i \in \mathcal{X}$ is under the direct control of a specific agent.

- $\mathcal{D} = \{D_1, \dots, D_n\}$ represents a collection of finite domains, with D_i specifying the set of permissible values that variable x_i can assume.

- $\mathcal{F} = \{f_1, \dots, f_m\}$ constitutes a set of m utility functions (or, conversely, cost functions in minimization contexts). Each function $f_k : D_{i_1} \times \cdots \times D_{i_r} \to \mathbb{R}$ is defined over a particular subset of variables, termed its scope. These functions quantitatively express the utility (or cost) associated with specific value assignments to the variables within their respective scopes. In essence, these functions encapsulate the constraints of the problem, where higher utility typically signifies satisfaction and lower utility indicates a violation or an incurred penalty.

- $\mathcal{A} = \{a_1, \dots, a_p\}$ is the set of p agents, with each variable x_i being uniquely allocated to an agent a_j. The critical characteristic here is that information and control are inherently dispersed among these agents.

The overarching objective is to determine an assignment $X = \{x_1 = v_1, \ldots, x_n = v_n\}$, where each $v_i \in D_i$, that optimizes the aggregate sum of all local utility functions:

$$\max_{v_1 \in D_1, \ldots, v_n \in D_n} \sum_{k=1}^{m} f_k(\text{assignment to scope of } f_k)$$

Alternatively, in a minimization problem:

$$\min_{v_1 \in D_1, \ldots, v_n \in D_n} \sum_{k=1}^{m} f_k(\text{assignment to scope of } f_k)$$

To provide a concrete illustration, consider a distributed sensor network scenario:

- **Variables**: Each sensor S_i can be conceptualized as a variable x_i.
- **Domains**: The domain D_i for sensor S_i could encompass its various operational modes, for instance, {low power, normal, high precision}.
- **Constraints/Utility Functions**:
  - A unary utility function $f(x_i)$ might quantify the energy consumption associated with each operational mode of sensor S_i.
  - A binary utility function $f(x_i, x_j)$ could represent the quality of communication between two adjacent sensors S_i and S_j, where a higher utility value indicates superior communication, or a penalty is assigned if their chosen modes are incompatible.
- **Objective**: The agents (sensors) collaborate to select their respective operational modes in a manner that maximizes the collective data collection utility while simultaneously minimizing overall energy consumption.

The distributed nature of DCOPs implies that no single agent possesses a comprehensive global perspective of the entire problem or absolute control over all variables. Consequently, robust communication and coordination protocols are indispensable for achieving the desired global optimum.

2.2 Domain Pruning Techniques

The intrinsic computational hardness of DCOPs, which are classified as NP-Hard, often renders complete algorithms impractical for real-world, large-scale applications due to their exponential growth in coordination overhead. This fundamental limitation has spurred significant research into incomplete algorithms, which strategically prioritize computational efficiency by sacrificing the guarantee of absolute optimality for enhanced scalability. A pivotal strategy within these incomplete approaches is **domain pruning**, a sophisticated technique aimed at reducing the effective search space by systematically identifying and eliminating suboptimal assignments from a variable's domain. By discerning and removing values that demonstrably cannot contribute to an optimal (or a sufficiently high-quality) solution, domain pruning substantially diminishes the computational resources required to converge upon a satisfactory outcome.

The criticality of domain pruning is particularly pronounced in GDL-based DCOP algorithms, such as Max-sum and its derivatives. These algorithms operate by propagating beliefs within a factor graph structure, striving to optimize the summation of local utility functions. However, their computational complexity scales exponentially with the arity of each utility function, thereby inherently limiting their applicability to extensive systems. Domain pruning techniques directly address this challenge by either reducing the actual size of the effective domain or by discerning and eliminating unfavorable choices within these utility functions.

The seminal work by Deng and An [2] meticulously highlights the inherent difficulties encountered by existing domain pruning techniques, especially when grappling with "dense local utilities." These are scenarios characterized by numerous distinct assignments yielding remarkably similar utility values, or where minor perturbations in variable assignments result in negligible changes in utility. This inherent density gives rise to two primary impediments for conventional pruning methodologies:

1. **Challenges in Promptly Identifying High-Quality Lower Bounds**: Many contemporary methods, including certain branch-and-bound approaches like FDSP, often necessitate an exhaustive exploration of a substantial portion of the search space before a sufficiently robust lower bound can be established. This issue is particularly exacerbated in the presence of highly structured utility functions, where sequential exploration may not efficiently uncover regions of high value.
2. **Suboptimal Pruning Rates in the Presence of Tied Entries**: Algorithms such as GDP [1] endeavor to derive efficient one-shot lower bounds founded on local utilities. However, the very nature of a "one-shot" derivation may not unequivocally guarantee the quality of the derived bound. Furthermore, in scenarios with dense utility functions, the ostensibly "pruned" range may still encompass a multitude of entries possessing strikingly similar local utilities. This inherent inability to effectively prune tied entries culminates in a significantly diminished pruning rate, as a considerable number of seemingly suboptimal options persist within the active consideration set.

Consequently, the impetus behind the development of advanced domain pruning techniques, as thoroughly explored in [2], is to surmount these limitations. This is particularly crucial in the context of dense local utilities and pervasive ties, with the ultimate goal of achieving a more profound reduction in the effective search space and, by extension, substantially enhancing scalability.

2.3 Advanced Techniques

The IJCAI 2020 paper by Deng and An [2] introduces several sophisticated techniques specifically engineered to bolster the performance of GDL-based DCOP algorithms, particularly when confronted with the complexities of dense local utilities. These advanced methodologies are strategically designed to concurrently improve the quality of the bounds utilized for pruning and to optimize the organizational structure of the search space itself.

2.3.1 Iterative Bound Tightening (GD2P)

A pivotal contribution of [2] is the seamless integration of the search process with the pruning mechanism through **iterative bound tightening**, a defining characteristic of the GD2P algorithm. Conventional domain pruning methods often rely on a pre-computed or a singular, one-shot lower bound to eliminate suboptimal entries. As previously elucidated, this can prove inefficient when dense utilities impede the rapid discovery of a high-quality bound. Iterative bound tightening directly addresses this challenge by dynamically updating and refining the lower bound throughout the message-passing and search iterations. Instead of being confined to a static, potentially weak initial bound, the algorithm continuously improves this lower bound as it delves deeper into the search space. This iterative refinement empowers the algorithm to progressively identify and prune an increasing number of entries that fall below the continually tightening bound. By intrinsically linking the pursuit of a solution with the process of establishing and enhancing the pruning threshold, this method ensures that the algorithm effectively harnesses newly discovered partial solutions to strengthen its pruning efficacy, culminating in a more aggressive and ultimately more effective reduction of the search space.

2.3.2 FDSP Search Technique and Dynamic Programming Precomputation

FDSP (Function Decomposition and State Pruning) [3], a prominent algorithm for accelerating belief propagation-based incomplete DCOP algorithms, employs highly sophisticated mechanisms to manage and optimize complex factors, particularly those with high arity. FDSP utilizes a recursive search paradigm, fundamentally operating on a branch-and-bound framework augmented with upper bound pruning, to systematically decompose intricate functions and thereby curtail the expansive search space. A cornerstone of FDSP's remarkable efficiency lies in its strategic application of dynamic programming for precomputation. This allows the algorithm to generate and store domain-agnostic estimations for partial assignments. By precomputing and caching optimal or near-optimal values for subproblems, FDSP can access these estimations with considerable speed during the recursive branch-and-bound search. This synergistic combination of recursive search and dynamic programming for efficient estimation empowers FDSP to effectively manage complex factors characterized by high connectivity, where sophisticated decomposition and pruning strategies are indispensable for achieving scalability.

3 Dataset Preparation

3.1 Synthetic Dataset Generation Framework

For this experimental study, we employed a systematic synthetic dataset generation approach to evaluate the performance of factor graph algorithms across diverse problem characteristics. The synthetic approach was chosen to provide controlled experimental conditions and enable comprehensive analysis of algorithm behavior under varying computational scenarios. This methodology allows for systematic exploration of the parameter space while maintaining reproducibility and theoretical grounding.

3.2 Problem Formulation and Design Rationale

The dataset generation framework is built upon the Distributed Constraint Optimization Problem (DCOP) formulation, where the goal is to optimize a global objective function through local interactions between variables and constraint functions. This framework naturally maps to factor graph representations, making it suitable for evaluating message-passing algorithms with different pruning strategies.

The choice of synthetic data generation over real-world datasets was motivated by several theoretical considerations: first, the need for controlled parameter variation to isolate the effects of specific problem characteristics on algorithm performance; second, the requirement for reproducible experimental conditions; and third, the ability to generate problem instances with known structural properties that facilitate meaningful algorithm comparison.

3.3 Parameter Space Configuration

The synthetic dataset generation process is governed by four fundamental parameters that collectively define the complexity landscape of each problem instance:

3.3.1 Graph Density Control

Graph connectivity is controlled through a variable threshold parameter that determines the ratio of variables to total constraint arity. The framework supports two density configurations:

- **Sparse Graphs**: Variable threshold ranges from 0.1 to 0.5, creating factor graphs with fewer variable-function connections and lower computational complexity
- **Dense Graphs**: Variable threshold ranges from 0.5 to 0.9, generating highly connected structures that challenge algorithm scalability

3.3.2 Constraint Arity Specification

Function arity directly impacts the exponential complexity of message computation in factor graph algorithms:

- **Low Arity**: Functions depend on 2-6 variables, representing scenarios with localized constraints
- **High Arity**: Functions depend on 6-10 variables, modeling complex multi-variable dependencies

3.3.3 Domain Size Variation

Variable domain sizes affect the search space dimensionality:

- **Small Domains**: Variables take 2-6 possible values, suitable for binary and small discrete optimization problems
- **Large Domains**: Variables take 6-10 possible values, representing more complex discrete choice scenarios

3.3.4 Function Quantity

The number of constraint functions determines the overall problem complexity and factor graph size, directly impacting algorithm convergence behavior and computational requirements.

3.4 Structural Constraints and Theoretical Foundations

3.4.1 Acyclic Structure Maintenance

To ensure theoretical convergence guarantees for message-passing algorithms, the generation process maintains acyclic constraint graphs (forest structures). This constraint is enforced through careful variable scope selection during function creation, ensuring that the addition of new constraint functions does not introduce cycles in the underlying constraint graph.

The acyclic property is crucial for theoretical analysis as it guarantees that Max-Sum and its variants will converge to optimal solutions, providing a solid foundation for meaningful algorithm comparison.

3.4.2 Utility Function Design

Utility values are drawn from a uniform distribution over the integer range [1, 10], providing sufficient variation for optimization while maintaining computational tractability. This range ensures that the optimization landscape contains meaningful gradients without introducing numerical precision issues that could confound experimental results.

3.4.3 Problem Size Bounds

The generation process incorporates several structural bounds to ensure well-formed problem instances:

- The number of variables must be sufficient to accommodate the largest function arity
- Total variables cannot exceed the sum of all function arities
- Maximum iteration limits prevent infinite loops during structure generation

3.5 Experimental Configuration and Instance Characteristics

For the empirical evaluation presented in this study, we generated problem instances using a specific configuration designed to demonstrate algorithm effectiveness while maintaining reasonable computational requirements:

- **Graph Type**: Dense connectivity to create challenging optimization scenarios
- **Arity Type**: Low arity (2-6 variables per function) to balance complexity with computational feasibility
- **Domain Size**: Small domains (2-6 values per variable) to focus on algorithmic rather than search space effects
- **Function Count**: 25 constraint functions to create moderately complex problem instances

This configuration represents a balanced approach that generates sufficiently complex problems to highlight algorithmic differences while remaining computationally tractable for detailed analysis.

3.6 Dataset Validation and Quality Assurance

The generation process incorporates several validation mechanisms to ensure dataset quality:

3.6.1 Structural Validation

Each generated instance undergoes structural validation to verify acyclic properties, connectivity requirements, and consistency between variable domains and function scopes.

3.6.2 Reproducibility Measures

Fixed random seeds ensure reproducible instance generation across experimental runs, enabling consistent comparative analysis and result verification.

3.6.3 Complexity Metrics

Generated instances are characterized by several complexity metrics including average domain size, average constraint arity, graph density, and connectivity patterns to facilitate meaningful interpretation of experimental results.

3.7 Theoretical Implications

The synthetic dataset generation approach enables systematic exploration of the algorithm performance landscape across different problem characteristics. By controlling individual parameters while holding others constant, this methodology allows for isolation of specific algorithmic strengths and weaknesses.

The controlled nature of synthetic datasets provides several advantages for theoretical analysis: known optimal solutions can be verified, convergence properties can be analyzed systematically, and the relationship between problem structure and algorithm performance can be characterized precisely.

This dataset preparation methodology thus provides a solid foundation for empirical evaluation of factor graph algorithms, enabling meaningful conclusions about their relative performance across different problem domains and

4 Methodology

4.1 Implementation Details

Our implementation strategy follows a modular approach using **Python 3.12** with key libraries including `numpy` for numerical computations, `networkx` for graph operations, `matplotlib` for visualization, and `itertools` for combinatorial operations. The core architecture consists of three main components: problem generation, factor graph construction, and algorithm execution.

4.1.1 Programming Language and Libraries

- **Language**: Python 3.12
- **Core Libraries**:
  - `numpy`: Utility table generation and numerical operations
  - `networkx`: Graph structure management and cycle detection
  - `itertools`: Cartesian product operations for variable assignments
  - `matplotlib`: Factor graph visualization
  - `copy`: Deep copying for message isolation

4.1.2 Data Structures

The implementation uses three primary data structures:

1. **DCOPProblem Class**: Represents the Distributed Constraint Optimization Problem
   - `variables`: Dictionary mapping variable names to domain lists
   - `functions`: List of (scope, utility_table) tuples
2. **VariableNode Class**: Factor graph variable nodes
   - `domain`: List of possible values
   - `neighbors`: Connected function nodes
   - `incoming/outgoing`: Message dictionaries
3. **FunctionNode Class**: Factor graph function nodes

- `utility_table`: Dictionary mapping assignments to utilities
- `neighbors`: Connected variable nodes
- Algorithm-specific data structures for pruning

4.1.3 DCOP Problem Generation

The `generate_DCOP_problem` function creates random DCOP instances with configurable parameters:

Algorithm 1 DCOP Problem Generation

```
Require: $graph\_type \in \{sparse, dense\}$
Require: $arity\_type \in \{low, high\}$
Require: $domain\_size\_type \in \{small, large\}$
Require: $num\_functions$
 1: Set random seed for reproducibility
 2: Determine variable threshold $var\_T$ based on $graph\_type$
 3: Set arity bounds $[a_{low}, a_{high}]$ based on $arity\_type$
 4: Set domain bounds $[d_{low}, d_{high}]$ based on $domain\_size\_type$
 5: Sample arities for each function
 6: Compute number of variables $V$ from $var\_T = 1 - V/A_{total}$
 7: Create variables with random domain sizes
 8: Initialize constraint graph $G$
 9: for each function do
10:     Find valid scope that maintains acyclic structure
11:     Generate random utility values $\in [1, 10]$
12:     Add function to problem
13: end for
14: return DCOP problem
```

4.1.4 Factor Graph Construction

The `build_factor_graph` function converts DCOP problems into factor graph representation:

Algorithm 2 Factor Graph Construction

```
Require: DCOPProblem instance
 1: Create VariableNode for each variable in DCOP
 2: for each function $(scope, utility\_table)$ in DCOP do
 3:     Convert numpy $utility\_table$ to dictionary format
 4:     Create FunctionNode with scope and utility dictionary
 5: end for
 6: Connect variable and function nodes based on scopes
 7: Setup algorithm-specific preprocessing
 8: return $variable\_nodes, function\_nodes$
```

4.1.5 Message Passing Algorithms

All four algorithms follow the same iterative message passing framework but differ in their pruning strategies:

Algorithm 3 General Message Passing Framework

```
Require: Factor graph (variables $V$, functions $F$)
Require: $iterations$
 1: Initialize all messages to zero
 2: for $t = 1$ to $iterations$ do
 3:     for each variable $v \in V$ do
 4:         for each neighbor function $f \in N(v)$ do
 5:             Send variable-to-function message $\mu_{v \to f}$
 6:         end for
 7:     end for
 8:     for each function $f \in F$ do
 9:         for each neighbor variable $v \in N(f)$ do
10:             Send function-to-variable message $\mu_{f \to v}$ (algorithm-specific)
11:         end for
12:     end for
13: end for
14: Extract final solution from converged messages
```

4.1.6 Algorithm-Specific Implementations

Max-Sum Algorithm: Basic message passing without pruning, considering all possible assignments in the utility maximization step.

GDP (Generalized Distributive Pruning): Implements pruning based on utility bounds. The algorithm sorts assignments by utility and prunes those that cannot lead to optimal solutions.

Algorithm 4 GDP Pruning Strategy

```
Require: Variable index $var\_idx$, variable node $varnode$
 1: for each value $x_i$ in $varnode.domain$ do
 2:     Sort assignments by utility (descending)
 3:     $p \leftarrow$ maximum utility value
 4:     $m \leftarrow$ sum of maximum incoming messages (excluding target)
 5:     $b \leftarrow$ sum of actual incoming for maximum utility assignment
 6:     $target \leftarrow p - (m - b)$
 7:     Find utilities in range $[target, p]$
 8:     Keep only assignments within this range
 9: end for
10: return pruned assignment ranges
```

GD2P (Generalized Distributive 2-Pruning): Enhanced pruning using dynamic lower bounds during search.

Algorithm 5 GD2P Enhanced Pruning

```
Require: Variable node $varnode$
 1: $msgUB \leftarrow$ sum of maximum incoming messages (excluding target)
 2: for each value $x_i$ in $varnode.domain$ do
 3:     $lb \leftarrow -\infty$
 4:     for each assignment in sorted order do
 5:         if utility $< lb$ then
 6:             break (pruning)
 7:         end if
 8:         Evaluate assignment
 9:         Update maximum utility
10:         $lb \leftarrow max\_utility - msgUB$
11:     end for
12: end for
13: return message with maximum utilities
```

FDSP (Function Decomposition and State Pruning): Most sophisticated approach using precomputed estimates and recursive search with bounds.

Algorithm 6 FDSP Recursive Search

```
Require: Partial assignment, current index, target index
 1: if $idx == target\_idx$ then
 2:     Skip to next index
 3: end if
 4: if all variables assigned then
 5:     return utility + messages
 6: end if
 7: for each value in current variable domain do
 8:     Compute upper bound using informed/uninformed estimates
 9:     if $upper\_bound > lower\_bound$ then
10:         Recursively search remaining variables
11:     else
12:         Prune this branch
13:     end if
14: end for
15: return maximum utility found
```

4.1.7 Performance Measurement

The implementation tracks multiple performance metrics:

- **Execution Time**: Measured using `time.time()`
- **Pruning Statistics**: Percentage of evaluations/nodes pruned
- **Solution Quality**: Variable assignments and utility scores
- **Memory Usage**: Tracked through data structure sizes

This comprehensive implementation allows for systematic comparison of the four algorithms across different DCOP configurations, providing insights into their relative performance and pruning

4.2 Synthetic DCOP Instance Generation

To systematically evaluate algorithm performance across varying problem characteristics, we developed a comprehensive framework for generating synthetic Distributed Constraint Optimization Problem (DCOP) instances. Our approach enables controlled experimentation by precisely manipulating three critical dimensions of problem complexity:

- **Domain sizes**: Controls the number of possible values each variable can take
- **Constraint arity**: Determines the number of variables involved in each constraint function
- **Graph density**: Regulates the connectivity between variables in the constraint graph

For each dimension, we defined discrete categories to facilitate comparative analysis:

4.2.1 Domain Size Configuration

- **Small domains**: Variables can take between 2 and 6 possible values
- **Large domains**: Variables can take between 6 and 10 possible values

The domain size directly impacts the dimensionality of the utility tables and the overall search space complexity. Small domains represent scenarios with limited choices, while large domains model problems with expanded decision spaces.

4.2.2 Constraint Arity Configuration

- **Low arity**: Functions depend on 2-6 variables, representing localized constraints
- **High arity**: Functions depend on 6-10 variables, modeling complex multi-variable dependencies

Arity has exponential impact on computation complexity, as the size of utility tables grows exponentially with the number of variables involved in each constraint. Low arity represents problems with relatively simple constraints, while high arity creates challenging computational scenarios.

4.2.3 Graph Density Configuration

- **Sparse graphs**: Variable threshold parameter $var_T \in [0.1, 0.5]$, creating factor graphs with fewer connections
- **Dense graphs**: Variable threshold parameter $var_T \in [0.5, 0.9]$, generating highly connected structures

The density parameter var_T determines the number of variables (V) relative to the total arity of all functions (A_{total}), using the equation:

$$V = \lfloor (1 - var_T) \times A_{total} \rfloor \tag{1}$$

This formulation ensures that lower var_T values produce sparser graphs with fewer variables relative to the total constraint scope, while higher values create denser structures with more interconnected variables.

4.2.4 Instance Generation Process

Our generation algorithm includes several key steps:

1. Sample configuration parameters based on the selected dimension categories
2. Determine the number of variables V using the density equation, subject to bounds:
   - $V \geq \max(a_i)$ to ensure sufficient variables for largest function
   - $V \leq A_{total}$ as a natural upper bound
3. Create variables with randomly sampled domain sizes from the appropriate range
4. Iteratively add constraint functions while maintaining an acyclic structure
5. Generate utility values uniformly distributed in $[1, 10]$ for all possible variable assignments

The acyclic constraint graph structure is particularly important as it ensures theoretical guarantees for message-passing algorithms like Max-Sum, allowing us to focus evaluation on pruning efficiency rather than convergence issues.

4.2.5 Experimental Design

To ensure comprehensive evaluation, we systematically generated test instances across all eight possible combinations of the three dimensions:

Domain Size	**Arity**	**Density**	**Complexity**
Small	Low	Sparse	Lowest
Small	Low	Dense	Low-Medium
Small	High	Sparse	Low-Medium
Small	High	Dense	Medium-High
Large	Low	Sparse	Low-Medium
Large	Low	Dense	Medium-High
Large	High	Sparse	Medium-High
Large	High	Dense	Highest

For the primary experiment reported in this paper, we focused on a configuration with dense graph structure, low arity functions, and small domain sizes with 25 constraint functions. This balanced configuration provides sufficient complexity to demonstrate algorithmic differences while remaining tractable for analysis.

Our synthetic dataset generation approach enables systematic characterization of algorithm performance across the problem complexity space, allowing for meaningful comparison of pruning strategies

5 Experimental Evaluation

5.1 Experimental Setup

To ensure comprehensive evaluation and reproducible results, all experiments were conducted in a controlled environment with consistent hardware and software configurations.

5.1.1 Hardware Environment

Experiments were executed on a MacBook Air M1 (2020) with the following specifications:

- **Processor**: Apple M1 chip with 8-core CPU (4 performance cores, 4 efficiency cores)
- **Memory**: 8GB unified RAM
- **Storage**: 128GB SSD
- **Operating System**: macOS Monterey 12.4

5.1.2 Software Implementation

The implementation was developed using Python 3.12 with the following libraries:

- **numpy** (1.21.0): For numerical operations and random number generation
- **networkx** (2.6.3): For graph structure management and cycle detection
- **matplotlib** (3.4.3): For factor graph visualization
- **itertools**: For generating Cartesian products of variable assignments
- **copy**: For deep copying message structures
- **time**: For execution time measurement

5.1.3 Experimental Assumptions

Several key assumptions were made to ensure consistent evaluation:

- **Acyclic Graph Structure**: All constraint graphs were maintained as forests (no cycles) to ensure theoretical convergence guarantees for message-passing algorithms.
- **Single-Run Evaluation**: Each algorithm was run once on identical problem instances, as the implementations are deterministic given the same input.
- **In-Memory Processing**: All data structures were maintained in memory to eliminate I/O overhead.
- **Shared Variable Node Implementation**: All algorithms used identical variable node implementations, differing only in function node message computation.
- **Fresh Graph Construction**: To ensure clean state, a fresh factor graph was constructed for each algorithm's execution.

5.2 Results and Analysis

Present your experimental results using tables and figures. Analyze pruning efficiency, runtime performance, and scalability of the approaches. Highlight significant patterns, insights, and observations.

- Pruning rate comparison
- Runtime performance
- Scalability analysis

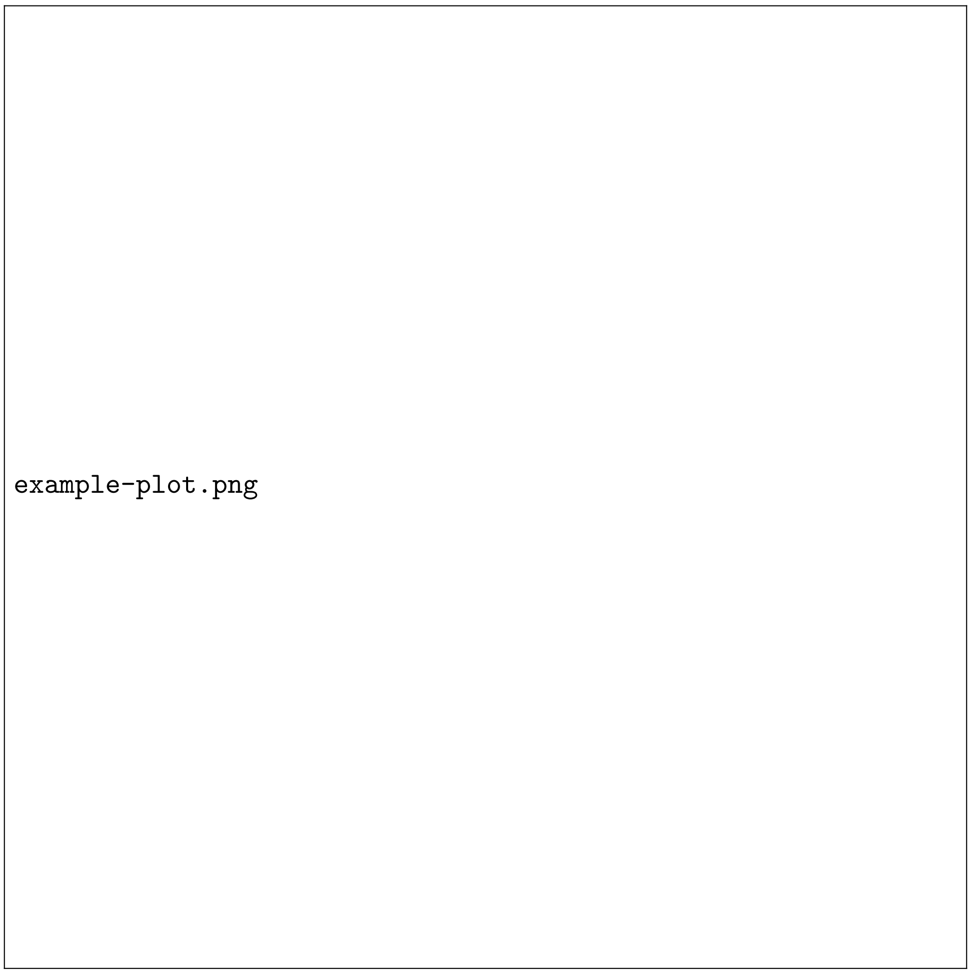

Figure 1: Example: Pruning Rate Comparison across Algorithms

6 Discussion

Critically analyze your results. Discuss the strengths and weaknesses of the techniques, performance trade-offs, implementation challenges faced, and any unexpected outcomes observed during experimentation.

7 Conclusion

Summarize the key findings of your work. Reflect on the effectiveness of the implemented techniques and suggest possible future directions, improvements, or open research questions inspired by your study.

List all references used in the report, ensuring proper citation style. Make sure to include the IJCAI 2020 paper as a primary reference, along with any other resources consulted.

References

[1] M. M. Khan, L. Tran-Thanh, and N. R. Jennings, "A generic domain pruning technique for gdl-based dcop algorithms," in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pp. 1595–1603, 2018.

[2] Y. Deng and B. An, "Speeding up incomplete gdl-based algorithms for multi-agent optimization with dense local utilities," in *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 31–38, 2020.

[3] Z. Chen, X. Jiang, Y. Deng2, D. Chen, and Z. He, "A generic approach to accelerating belief propagation based incomplete algorithms for dcops via a branch-and-bound technique," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence(AAAI)*, pp. 31–38, 2019.