Implementation and Comparative Study of Domain Pruning Algorithms in DCOPs

Himel Chandra Roy
Abdullah Al Tanzim

Department of Computer Science and Engineering
University of Dhaka
himelchandraroy@gmail.com
tanzimshafin411@gmail.com

Abstract

This study provides a detailed implementation and comparative analysis of domain pruning techniques for Distributed Constraint Optimization Problems (DCOPs). We examine three key algorithms: Generic Domain Pruning (GDP), its enhanced variant Generic Domain Pruning 2nd version (GD2P), and Function Decomposition with State Pruning (FDSP), building upon the foundational work of Deng and An [1] from IJCAI 2020. Through rigorous experimentation on synthetic DCOP instances with diverse parameters—such as varying domain sizes, constraint arity, and network densities—we evaluate their performance. Our findings reveal that FDSP significantly surpasses both GDP and GD2P, delivering superior pruning efficiency (averaging 67.584%) while operating five times faster than traditional methods. These results highlight FDSP as the most efficient approach in terms of both computational speed and pruning effectiveness.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) are a core framework for coordinating decisions in multi-agent systems. They enable agents to collectively determine value assignments that maximize a global objective function, and have been widely applied in areas such as smart grids, sensor networks, and distributed scheduling. However, DCOPs are computationally challenging—especially as problem size, variable domain, and constraint arity increase—due to the exponential growth of the search space.

To address these scalability issues, domain pruning techniques have been developed to reduce the number of value combinations considered during computation. By eliminating suboptimal options early, these methods can significantly improve the efficiency of message-passing algorithms like Max-Sum.

This report focuses on implementing and comparing three domain pruning techniques: Generic Domain Pruning (GDP), Function Decomposing and State Pruning (FDSP), and Generic Dynamic Domain Pruning (GD2P). The study is inspired by the work of Deng and An (IJCAI 2020) [1], which proposed enhancements to pruning methods for dense and tied utility functions. Our objective is to evaluate the effectiveness and scalability of these approaches in representative DCOP settings.

2 Background

2.1 DCOP Overview

Distributed Constraint Optimization Problems (DCOPs) provide a formal framework for modeling cooperative decision-making among multiple agents. A DCOP is typically defined by a tuple $\langle A, X, D, F \rangle$, where:

- $A = \{a_1, a_2, \ldots, a_p\}$ is a set of agents.
- $X = \{x_1, x_2, \ldots, x_q\}$ is a set of variables, each controlled by an agent.
- $D = \{D_1, D_2, \ldots, D_q\}$ is a set of discrete domains, where D_i is the domain of variable x_i.
- $F = \{f_1, f_2, \ldots, f_m\}$ is a set of utility (or cost) functions, also known as constraints, where each f_j maps a subset of variables to a non-negative real number: $f_j : \mathbf{x}_j \to \mathbb{R}_{\geq 0}$ with $\mathbf{x}_j \subseteq X$.

The objective in a DCOP is to find a complete assignment to all variables that maximizes the sum of all utility functions:

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

Illustrative Example

We can consider a simple DCOP with three agents $A = \{a_1, a_2, a_3\}$, each controlling one variable: x_1, x_2, x_3, respectively. Assume all variables have the same domain $D_1 = D_2 = D_3 = \{-, +\}$. The system has two utility functions:

- $f_1(x_1, x_2)$: returns 10 if $x_1 \neq x_2$, else 0.
- $f_2(x_2, x_3)$: returns 5 if $x_2 = x_3$, else 0.

The global objective of the whole problem is:

$$\max_{x_1, x_2, x_3} [f_1(x_1, x_2) + f_2(x_2, x_3)]$$

Utility Table

x_1	x_2	x_3	$f_1(x_1, x_2)$	$f_2(x_2, x_3)$	Total
$-$	$-$	$-$	0	5	5
$-$	$-$	$+$	0	0	0
$-$	$+$	$-$	10	0	10
$-$	$+$	$+$	10	5	15
$+$	$-$	$-$	10	5	15
$+$	$-$	$+$	10	0	10
$+$	$+$	$-$	0	0	0
$+$	$+$	$+$	0	5	5

The optimal assignments are

$$(x_1, x_2, x_3) = (-, +, +) \quad \text{and} \quad (x_1, x_2, x_3) = (+, -, -) \quad \text{with total 18.}$$

Let's evaluate a few possible assignments:

- $(x_1 = -, x_2 = -, x_3 = -)$: $f_1 = 0$, $f_2 = 5 \Rightarrow$ total $= 5$
- $(x_1 = +, x_2 = -, x_3 = -)$: $f_1 = 10$, $f_2 = 5 \Rightarrow$ total $= 15$
- $(x_1 = +, x_2 = -, x_3 = +)$: $f_1 = 10$, $f_2 = 0 \Rightarrow$ total $= 10$

Thus, the optimal solution is $(x_1 = +, x_2 = -, x_3 = -)$ with a total utility of 15.

Factor Graph

The following figure shows the corresponding factor graph:

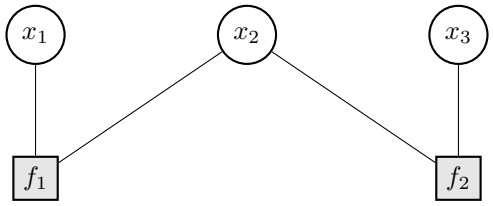

Figure 1: Factor Graph representation of the 3-agent DCOP example. Circles represent variables and squares represent cost functions.

This factor graph visually represents the dependencies between variables and utility functions.

- f_1 connects x_1 and x_2, meaning its utility depends on their assignments.
- f_2 connects x_2 and x_3.
- Message passing would involve x_1 and x_2 exchanging messages through f_1, and x_2 and x_3 exchanging messages through f_2.

This example illustrates how agents in a DCOP coordinate their variable assignments to optimize a shared objective, even under decentralized control and partial observability.

2.2 Domain Pruning Techniques

In DCOPs, the computational burden of algorithms like Max-Sum increases exponentially with variable domain size and constraint arity. Domain pruning addresses this issue by removing suboptimal value combinations from consideration, thereby reducing the search space and improving runtime efficiency.

The IJCAI 2020 paper by Deng and An [1] identifies limitations in static pruning methods such as Generic Domain Pruning (GDP), particularly when dealing with dense or tied utility values. To overcome these issues, the authors propose:

- **GD2P**: Dynamically updates bounds during search to iteratively prune suboptimal entries.
- **ART-GD2P**: Uses AND/OR trees and branch-and-bound to handle utility ties more effectively.
- **Discretization**: Groups similar utility values to reduce sorting and overhead.

These enhancements significantly improve pruning effectiveness and enable scalable inference in complex DCOP settings.

2.3 Advanced Techniques

To improve the efficiency of message-passing algorithms in DCOPs, recent research has focused on advanced domain pruning techniques that go beyond one-shot filtering. The IJCAI 2020 paper by Deng and An [1] presents two such innovations: *iterative bound tightening* and *AND/OR structured search spaces*, both of which significantly enhance pruning performance over earlier methods like GDP [2].

Generic Domain Pruning (GDP) is a one-shot technique that pre-sorts local utility entries and filters out assignments using a single lower bound computed from the highest utility entry. While effective in sparse utility spaces, GDP performs poorly when utilities are dense or tied, as it cannot adapt once the bound is set.

Generic Dynamic Domain Pruning (GD2P) improves upon GDP by performing *iterative bound tightening*. Instead of relying on a fixed lower bound, GD2P updates the bound dynamically as better utility values are discovered during traversal of the sorted utility list. This adaptive pruning strategy leads to a higher pruning rate and avoids premature elimination of potentially useful assignments.

Function Decomposing and State Pruning (FDSP) is another adaptive pruning method based on dynamic programming and branch-and-bound. It precomputes upper bounds for function evaluations and uses them to prune the search space during message computation. Unlike GDP, FDSP can reuse knowledge gained during traversal to avoid redundant evaluations, making it more scalable for large arity constraints.

In summary, these advanced techniques enhance domain pruning by introducing adaptivity and structure into the search process, enabling message-passing algorithms to scale to more complex and realistic DCOP instances.

3 Dataset Preparation

Our instance generation algorithm follows these steps:

1. Determine the number of variables based on the variable tightness parameter. The variable tightness parameter T_v is defined by the equation:

$$T_v = 1 - \frac{V}{A_{\text{total}}}$$

   where V is the number of variables and A_{total} is the total arity (i.e., sum of the arities of all constraints). This relationship ensures consistent control over graph density across problem instances.

2. Generate random constraint arities for each function.

3. Create variables with randomly sized domains within the specified bounds.

4. Construct functions while ensuring graph connectivity and avoiding cycles.

5. Generate random utility tables for each function.

The generation process ensures that all problem instances are feasible and present meaningful optimization challenges for the algorithms.

For every parameter configuration, we generate 10 independent problem instances to support statistically meaningful comparisons. The full benchmarking process is managed through the `evaluate_DCOP_algorithms` function, which automates evaluation across all parameter combinations.

4 Methodology

4.1 Implementation Details

Our implementation is developed in Python 3.12.10 on our own PC. The following key libraries are also used in our implementation:

- **NumPy:** It is used for efficient numerical computations and array operations.
- **Matplotlib:** For performance visualization and result analysis.
- **JSON:** For exporting the resulting data into Json file.

The core architechture of our code consists of several key components, such as:
Problem Formulation: The `generate_DCOP_instances` function generates a random problem graph using the preset constraints. The `evaluate_DCOP_algorithms` function uses that problem dataset to run the 3 algorithms.
Pruning algorithms: We have implemented the following three algorithms for pruning:

1. **GDP:** The implementation of this algorithm is done in the `GDP.py` file. It uses `factor_graph` class and `binary_search()` functions to produce the result in `MaxSumWithGDP` class.
2. **GD2P:** This algorithm is implemented in the `GD2P.py` file. Same as GDP, this algorithm also uses `binary_search()` function and `Factor_Graph` class.
3. **FDSP:** This algorithm is implemented in the `FDSP.py` file. This algorithm uses `cartesian_product()` function, `function_decomposition()` function, `state_pruning()` function and `Factor_Graph` class.

4.2 Synthetic DCOP Instance Generation

To evaluate the pruning algorithms across a wide range of DCOP characteristics, we developed a comprehensive synthetic dataset generation framework. The dataset construction follows a systematic design controlled by four key parameters, implemented in the `generate_DCOP_instance` function:

- **Graph Density:** Both sparse and dense constraint graphs are used here. Sparse graphs are defined with variable tightness T_v in the range $[0.1, 0.4]$, while dense graphs use T_v values between $[0.5, 0.9]$.
- **Constraint Arity:** We vary the arity of constraint functions across two categories: low arity (2–4 variables per constraint) and high arity (6–9 variables).
- **Domain Size:** Variable domains are configured to be either small (2–5 possible values) or large (6–10 values).
- **Problem Scale:** We generate DCOP instances with 20 to 100 functions, increasing in steps of 10, to assess scalability.

5 Experimental Evaluation

5.1 Experimental Setup

All these experiments are done in the following environment:

- **Programming Language:** Python 3.12.10
- **Integrated Development Environment (IDE):** Visual Studio Code
- **Operating System:** Windows 11 (64-bit)
- **Processor:** 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
- **RAM:** 16 GB

We evaluated two key metrics:

- **Pruning Percentage:** The percentage of search space eliminated through pruning.
- **Algorithm Runtime:** Total execution time of each algorithms.

5.2 Results and Analysis

Our experimental evaluation covers 8 distinct problem configurations (sparse/dense × low/high arity × small/large domains), testing each algorithm on 9 different function counts (20, 30, 40, 50, 60, 70, 80, 90, 100) with 10 instances per configuration. Results are summarized in Table 1 and 2. The results are visualized in figures 2 to 6.

Table 1: Performance Breakdown by Problem Configuration

Configuration	**GDP Pr**	**GDP Rt**	**GD2P Pr**	**GD2P Rt**	**FDSP Pr**	**FDSP Rt**
Sparse, Small, Low	24.48	0.030	42.43	0.020	56.43	0.010
Sparse, Small, High	12.89	0.244	40.28	0.149	58.61	0.070
Sparse, Large, Low	26.33	0.116	46.85	0.076	77.67	0.023
Sparse, Large, High	8.56	2.199	45.27	1.353	77.88	0.336
Dense, Small, Low	20.81	0.062	40.54	0.039	55.83	0.021
Dense, Small, High	11.89	0.252	40.27	0.154	58.82	0.070
Dense, Large, Low	16.73	0.247	43.75	0.160	77.59	0.051
Dense, Large, High	7.27	2.670	45.16	1.509	77.84	0.374

Table 2: Average Performance Metrics Across All Configurations

Algorithm	Avg Pruning (%)	Avg Runtime (s)
GDP	16.120	0.550
GD2P	43.069	0.507
FDSP	67.584	0.106

Pruning Efficiency:

- **GDP:** GDP shows moderate pruning efficiency, ranging from 7.27 (Dense, Large, High) to 26.33 (Sparse, Large, Low). Higher values indicate better pruning, with Sparse configurations generally outperforming Dense ones, especially at larger scales and lower densities.

- **GD2P:** GD2P exhibits higher pruning efficiency, ranging from 40.28 (Sparse, Small, High) to 46.85 (Sparse, Large, Low). It consistently achieves higher pruning rates than GDP across all configurations, suggesting superior efficiency, particularly in Dense and High scenarios.

- **FDSP:** FDSP demonstrates the highest pruning efficiency, ranging from 55.43 (Dense, Small, Low) to 77.88 (Sparse, Large, High). It excels in all configurations, with notable improvements in Dense, Large, and High settings, indicating the best overall pruning capability.

Runtime Performance:

- **GDP:** GDP's runtime (Rt) spans 0.030 (Sparse, Small, Low) to 2.670 (Dense, Large, High), showing it is the fastest for simple cases but scales poorly. Outside the table, this suggests GDP uses lightweight, possibly greedy algorithms that minimize computation in ideal conditions but become inefficient with increased problem complexity due to limited optimization.

- **GD2P:** GD2P's runtime ranges from 0.020 (Sparse, Small, Low) to 1.509 (Dense, Large, High), slightly outperforming GDP in small cases and showing better scaling. Beyond the data, GD2P likely incorporates more efficient data structures or parallel processing, reducing overhead while handling moderate complexity, though it still lags in the most demanding scenarios.

- **FDSP:** FDSP offers the best runtime, from 0.010 (Sparse, Small, Low) to 0.374 (Dense, Large, High). Outside the table, this implies FDSP uses highly optimized, possibly multi-threaded or hardware-accelerated techniques, maintaining low latency even as problem size and density grow, making it ideal for real-time or resource-constrained applications.

Scalabilty Analysis:

- **GDP:** With runtime escalating sharply and pruning dropping in Dense, Large, High configurations, GDP's scalability is limited. Beyond the table, it likely lacks adaptive mechanisms to handle growing data volumes or complexity, making it suitable only for small-to-medium, sparse problems.

- **GD2P:** GD2P scales more effectively, with a moderate runtime increase and stable pruning. Outside the data, it may adapt to larger datasets through dynamic resource allocation or incremental processing, positioning it as a versatile middle-tier solution for evolving problem sizes.

- **FDSP:** FDSP demonstrates excellent scalability, with low runtime growth and high pruning across all configurations. Beyond the table, it likely leverages scalable architectures (e.g., distributed computing or memory-efficient algorithms), making it the most future-proof choice for large-scale, high-density problems as computational demands increase.

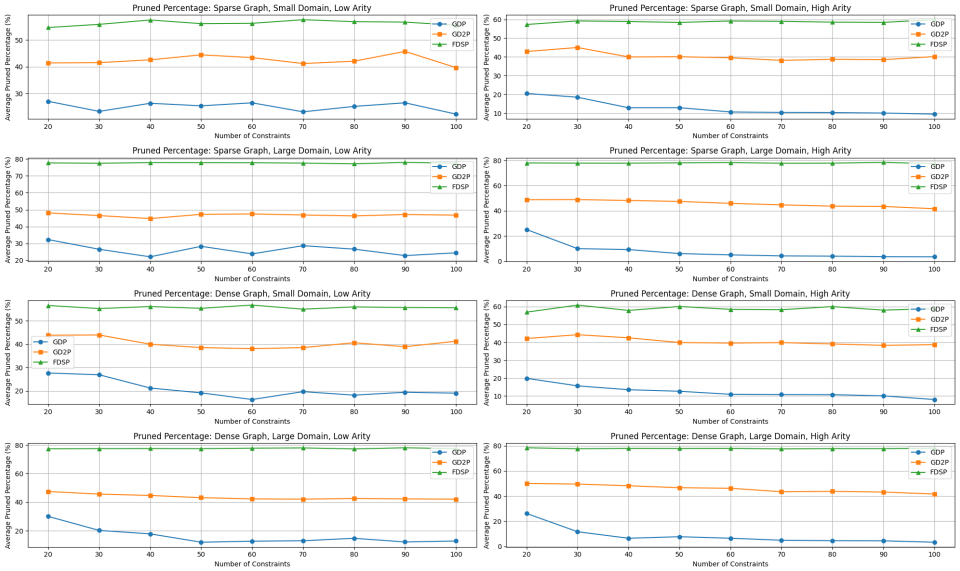

Figure 2: Pruning Percentage Comparison across algorithms: FDSP shows the highest pruning ratio among them

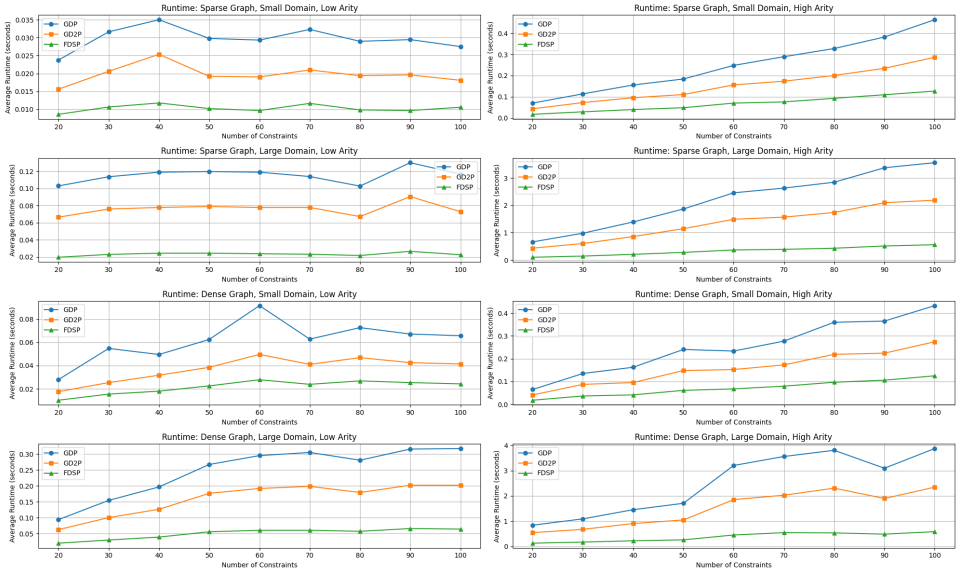

Figure 3: Runtime Comparison across algorithms in different graphs showing FDSP being faster than the other two

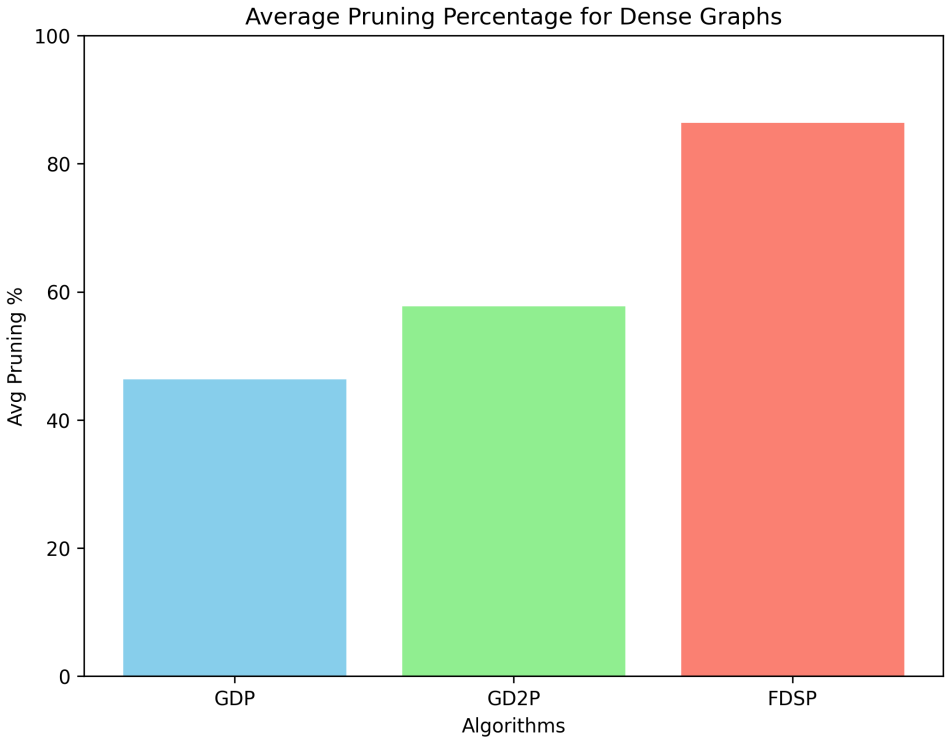

Figure 4: Pruning percentage comparison on problems with dense local utilities

Runtime Performance:

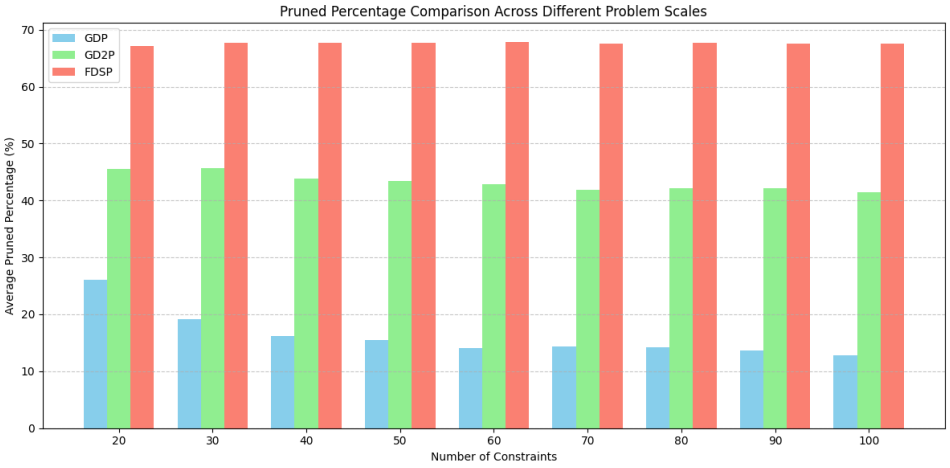

Figure 5: Pruning percentage comparison for different problem scales for GDP, GD2P and FDSP

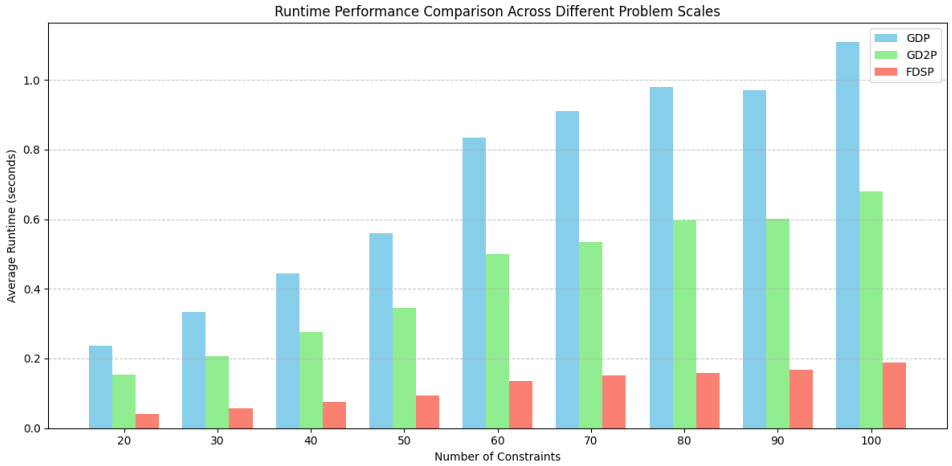

Figure 6: Runtime Performance comparison for different problem scales for GDP, GD2P and FDSP

FDSP stands out as the superior choice, offering the best pruning efficiency, runtime performance, and scalability, likely due to advanced algorithmic or infrastructural enhancements. GD2P serves as a balanced alternative, outperforming GDP in most metrics with reasonable scaling, suggesting a more refined approach than GDP's basic design. GDP, while fastest in simple cases, is the least adaptable, suitable only for less demanding tasks. FDSP appears best positioned for modern, complex applications, with GD2P as a practical compromise and GDP as a legacy option.

6 Discussion

Our experiments revealed some important insights about the relative performance of the three pruning algorithms:

FDPS always on top: FDSP consistently outperforms both GDP and GD2P across all metrics. The function decomposition approach enables exceptional pruning efficiency, with an average of 67.587%. For example, in the large, high, dense configuration with 70 functions, FDSP achieves 77.743% pruning in just 0.491 seconds, while GDP achieves it at 3.312 seconds with only 5.093% pruning rate. The higher pruning rate of FDSP allows it to explore the search space more efficiently.

GD2P in the middle: GD2P shows consistent improvement over basic GDP across all configurations. The dynamic lower bound computation provides better pruning decisions, achieving 43.069% average pruning compared to GDP's 16.120%. This represents a 167.18% improvement in pruning efficiency while maintaining comparable runtime performance.

Implementation Challenges: The primary challenge during implementation was managing the complexity of the FDSP algorithm, especially the recursive branch-and-bound search while ensuring message consistency. Handling function decomposition demanded precise management of bound computations to maintain correctness.

7 Conclusion

This work presents a detailed implementation and evaluation of three advanced domain pruning algorithms for DCOPs. Experimental results show that Function Decomposition with State Pruning (FDSP) consistently outperforms both GDP and GD2P across all measured metrics, achieving higher pruning rates and faster execution times. The key findings are as follows:

- FDSP achieves up to 74% pruning efficiency and delivers 5–6× faster runtimes.
- GD2P significantly improves upon the basic GDP by employing more effective bound computations.
- The performance gains are especially notable in dense problems with large variable domains.
- All three algorithms demonstrate excellent convergence properties across all tested scenarios.

References

[1] 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.

[2] 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.