

INTELLIGENT SOLUTION SPACE SEARCH AND REDUCTION USING
TREE-BASED LEARNING METHODS

by

Fulya Terzi

B.S., Industrial Engineering, Middle East Technical University, 2012

M.S., Industrial Engineering, Boğaziçi University, 2017

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Industrial Engineering
Boğaziçi University

2025

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Prof. Necati Aras and Assoc. Prof. Mustafa Gökçe Baydoğan, for their guidance and support throughout my entire graduate study. Their expertise and insightful feedback have been instrumental in shaping this thesis. I am truly grateful for their patience, dedication, and unhesitating belief in my work.

I also want to thank Prof. Taner Bilgiç, Prof. Mehmet Gönen, Prof. Refik Güllü, and Assoc. Prof. Mert Edalı for taking part in my thesis committee and for their valuable suggestions.

I would like to thank my parents, my brother and my sister for their support and love. I am thankful for trusting me and encouraging me throughout my life.

I would like to thank my loving husband with all my heart. I am sure things would be much harder without his support and love.

I also thank TUBITAK for their financial support during my graduate study. I also thank YÖK for their financial support during my graduate study.

ABSTRACT

INTELLIGENT SOLUTION SPACE SEARCH AND REDUCTION USING TREE-BASED LEARNING METHODS

This thesis presents the development and evaluation of three innovative algorithms aimed at improving the solution performance of optimization problems through the integration of tree-based machine-learning techniques. The first algorithm, “Remove-Add Algorithm Augmented by Population Management,” combines machine learning and optimization within a population-based heuristic framework. Extensive computational results on the p -median problem and Quadratic Assignment Problem (QAP) demonstrate its superior performance compared to traditional methods. The second algorithm, “Variable Fixing Algorithm,” utilizes rule sets for variable fixing generated by regression trees that utilize active learning and worst leaf node path split decisions. Extensive computational results on the QPLIB instances highlight the approach’s effectiveness in enhancing solution diversity, preventing premature convergence, and ensuring computational efficiency. The third algorithm, “Learning-Based Bound Generating Algorithm,” uses historical data and regression tree models to generate lower and upper bounds for decision variables that guide the optimization process, effectively narrowing the search space and accelerating convergence. Experimental results show that deeper regression trees improve the quality of generated bounds, particularly for complex optimization landscapes. Through comprehensive experiments and comparative analyses, this research demonstrates the effectiveness of these algorithms in improving solution quality and computational efficiency. The findings underscore the ability of tree-based learning to effectively differentiate between high-quality and low-quality solutions, guiding the algorithm toward highly promising regions of the solution space.

ÖZET

AĞAÇ TABANLI ÖĞRENME YÖNTEMLERİ KULLANARAK ÇÖZÜM UZAYININ AKILLI ŞEKİLDE ARANMASI VE DARALTILMASI

Bu tez, ağaç tabanlı makine öğrenimi tekniklerinin entegrasyonu yoluyla optimizasyon problemlerinin performansını artırmaya yönelik üç yenilikçi algoritmanın geliştirilmesini ve değerlendirilmesini sunmaktadır. İlk algoritma olan “Popülasyon Yönetimi Eklenmiş Çıkar-Ekle Algoritması,” veri madenciliği ve optimizasyonu popülasyon tabanlı sezgisel bir çerçevede birleştirir. p -median problemi ve Kareleme Atama Problemi (QAP) üzerindeki kapsamlı hesaplamalı sonuçlar, bu algoritmanın geleneksel yöntemlere kıyasla üstün performansını göstermektedir. İkinci algoritma olan “Değişken Sabitleme Algoritması,” aktif öğrenme ve en kötü yaprak düğümü yol bölme kararlarını kullanarak değişken sabitlemek için regresyon ağaç kuralları setlerini kullanır. QPLIB örnekleri üzerindeki kapsamlı hesaplamalı sonuçlar, bu yaklaşımın çözüm çeşitliliğini artırmada, erken yakınsamayı önlemede ve hesaplama verimliliğini sağlamada etkinliğini vurgulamaktadır. Üçüncü algoritma olan “Öğrenme Tabanlı Değişken Kısıtları Üretme Algoritması,” optimizasyon sürecini yönlendiren sınırları oluşturmak için önceki iterasyonlardaki verileri ve regresyon ağaç modellerini kullanır, böylece arama alanını etkili bir şekilde daraltır ve yakınsamayı hızlandırır. Deneysel sonuçlar, daha derin regresyon ağaçlarının, özellikle karmaşık çözüm uzayı için üretilen sınırların kalitesini artırdığını göstermektedir. Kapsamlı deneyler ve karşılaştırmalı analizler yoluyla, bu araştırma bu algoritmaların çözüm kalitesini ve hesaplama verimliliğini artırmadaki etkinliğini göstermektedir. Bulgular, ağaç tabanlı öğrenmenin yüksek kaliteli ve düşük kaliteli çözümler arasında etkili bir şekilde ayırım yapma ve algoritmayı daha kaliteli çözüm alanlarına yönlendirme yeteneğini vurgulamaktadır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET	v
LIST OF FIGURES	x
LIST OF TABLES	xv
LIST OF SYMBOLS	xix
LIST OF ACRONYMS/ABBREVIATIONS	xx
1. INTRODUCTION	1
2. LITERATURE REVIEW	6
2.1. Using Machine Learning for Initialization	8
2.2. Using Machine Learning to Set the Parameters	9
2.3. Using Machine Learning Methods in Population Management	10
2.4. Using Machine Learning Methods During the Evaluation	11
2.5. Application of Machine Learning Methods Within Operators	14
2.6. Using Machine Learning Methods in Local Search	14
2.7. Using Machine Learning Methods to Support Optimization Solvers	15
3. REMOVE-ADD ALGORITHM AUGMENTED BY POPULATION MANAGEMENT	17
3.1. General Scheme	17
3.1.1. Feasibility and Constraints in Solution Representation	18
3.1.2. Handling Complex Constraints	19
3.1.3. Solution Representation	20
3.1.4. Algorithm Workflow and Visual Explanation	20
3.1.5. Overview of the Population Management Integrated Add-Remove Algorithm Flowchart	28
3.1.6. Pseudo-Codes of RegTree Algorithm	32
3.1.7. Conceptual Comparison: Naive Approach vs. Regression Tree Integration	41

3.2. Illustrative Examples	45
3.2.1. p -median Problem	45
3.2.1.1. Mathematical Model	46
3.2.2. Quadratic Assignment Problem	46
3.3. Computational Results for p -median Problem	48
3.3.1. Sensitivity Analysis for Population Size	49
3.3.2. RegTree Design Requirements for p -median	54
3.3.3. Diversification Analysis with Three Techniques	54
3.3.4. Comparison of RegTree and Base methods with Equal Number of Iterations	57
3.3.5. Comparison of RegTree method with MDM-GRASP	64
3.3.6. Adding new problem-specific features	67
3.4. Computational Studies for QAP	70
3.4.1. RegTree Design Requirements for QAP	70
3.4.2. Comparison of RegTree, Base, and RegTree added Three Fea- tures methods	72
3.4.3. Q-Matrix Calculation	73
3.4.3.1. Calculation Steps	73
3.4.3.2. Properties of the Q-Matrix	74
3.4.3.3. Application in QAP	74
3.4.3.4. Q-Matrix Density	74
3.4.4. Results and Analysis of QAP Instances	75
4. VARIABLE FIXING ALGORITHM	78
4.1. General Scheme	78
4.1.1. Feasibility in the Binary Domain	78
4.1.2. Solution Space Restriction via Regression Trees	79
4.1.3. Applications and Effectiveness	80
4.2. System Architecture	80
4.2.1. Solution Representation	81
4.2.2. Population Initialization and Regression Tree Construction	81
4.2.3. Rule Set Generation	82

4.2.4.	Variable Fixing Procedure	83
4.2.5.	Population Update Procedure	85
4.2.6.	Detailed Algorithm Steps	86
4.2.7.	Pseudo-Codes and Detailed Explanations	88
4.3.	Illustrative Examples	96
4.3.1.	Quadratic Programming Problems	97
4.4.	Numerical Results	97
4.4.1.	QPLIB Benchmark Instances Trial Results	98
5.	LEARNING BASED BOUNDS GENERATING ALGORITHM	106
5.1.	General Scheme	106
5.1.1.	Preserving Feasibility	108
5.1.2.	Scope of Problems Solved	109
5.2.	System Architecture	109
5.2.1.	Solution Representation	110
5.2.2.	Population Initialization and Regression Tree Construction . . .	110
5.2.3.	Bound Generation Procedure	111
5.2.4.	Population Update Procedure	117
5.3.	Pseudo-Code for Algorithms	118
5.4.	Illustrative Examples	128
5.4.1.	Preliminary Analysis: Regression Trees for Continuous Solution Space Division	128
5.4.1.1.	Univariate Split	129
5.4.1.2.	Multivariate Split	131
5.4.2.	BBOB Test Functions	133
5.4.2.1.	Characteristics of BBOB Test Functions	133
5.4.2.2.	Notable BBOB Test Functions	134
5.4.3.	Importance of BBOB Test Functions	135
5.5.	Numerical Results	135
5.5.1.	Iterative Solution Space Partitioning and Optimization Using Regression Trees	137
5.5.2.	BBOB Test Functions-Metric Evaluations	142

5.5.3. BBOB Test Functions Trial Results-Minimum Distance to Optimal Solution	149
5.5.4. BBOB Test Functions Comparison with Original-Minimum Distance to Optimal Solution	150
5.5.5. BBOB Test Functions Graphical Comparison Based on Selected Best Scenarios	151
6. CONCLUSIONS	162
REFERENCES	167
APPENDIX A: THE REMOVE-ADD ALGORITHM AUGMENTED BY POPULATION MANAGEMENT	177
A.1. Analysis of RegTree algorithm with respect to different techniques . . .	177
A.2. RegTree comparison with Base at $PopAddRem = 2$	179
A.3. RegTree comparison with Base at $PopAddRem = 4$	182
A.4. Paper [20] Results	186
A.5. RegTree added Three New Problem Specific Features	188
A.6. QAP Results for thirteen selected problem instances	191
APPENDIX B: VARIABLE FIXING ALGORITHM	193
B.1. Selected QPLIB Instances Properties	193
B.2. The Results of the scenarios	194
APPENDIX C: GENERATING VARIABLE BOUNDS ALGORITHM	197
C.1. Categorization of BBOB Test Functions by Specialties	197
C.2. Results of the Variable Bounds Generation Algorithm Trials	199
C.2.1. The percentage shrinkage rates	199
C.2.2. The Number of Out-of-bound Variable	201
C.2.3. The Leaf Node Order of the Optimal Solution	203
C.2.4. The distance of the best solution to the optimal solution	205
C.2.5. Minimum Distance to Optimal Solution Across Five Replication Results: All Scenarios	222
C.2.6. A Comparative Analysis of Scenario-Based Models vs. Original Bound Constraints	231
APPENDIX D: FIGURE PERMISSION	239

LIST OF FIGURES

Figure 1.1.	A general taxonomy of machine learning-supported metaheuristics [6].	3
Figure 2.1.	Using representatives for the evaluation process through clustering.	13
Figure 3.1.	Regression Tree Example.	22
Figure 3.2.	Regression Tree Example with the Worst Leaf Node.	23
Figure 3.3.	Regression Tree Example with the Best Leaf Node.	24
Figure 3.4.	Histogram of decisions in solutions of the Worst Leaf Node.	25
Figure 3.5.	Relative frequencies obtained from frequencies of solutions in the Worst Leaf Node.	26
Figure 3.6.	The determination of the remove decision.	26
Figure 3.7.	Histogram of decisions in solutions of the Best Leaf Node.	27
Figure 3.8.	Relative frequencies of the Best Leaf Node solutions.	27
Figure 3.9.	Flow Chart for Population Management Integrated Add Remove Algorithm Using Regression Tree.	31
Figure 3.10.	Selection Phase of the Algorithm.	41
Figure 3.11.	Improvement Phase of the Algorithm.	42

Figure 3.12. Population Management Phase of the Algorithm.	43
Figure 3.13. Whole Picture of the Algorithm.	44
Figure 3.14. Mean GAP (%) by Population Size.	52
Figure 3.15. Mean Runtime by Population Size.	53
Figure 3.16. Trade-Off: GAP (%) vs. Runtime for Different Population Sizes. .	53
Figure 3.17. Average GAP(%) - Best and SecBest Techniques.	56
Figure 3.18. Average GAP(%) - Best and MaxVar Techniques.	56
Figure 3.19. Average GAP(%) - SecBest and MaxVar Techniques.	57
Figure 3.20. Best GAP(%) - RegTree and Base methods, $PopAddRem=2$	59
Figure 3.21. Average GAP(%) - RegTree and Base methods, $PopAddRem=2$. .	60
Figure 3.22. Solution Time - RegTree and Base methods, $PopAddRem=2$	60
Figure 3.23. Number of Iterations - RegTree and Base methods, $PopAddRem=2$.	61
Figure 3.24. Best GAP(%) - RegTree and Base methods, $PopAddRem=4$	62
Figure 3.25. Average GAP(%) - RegTree and Base methods, $PopAddRem=4$. .	63
Figure 3.26. Solution Time - RegTree and Base methods, $PopAddRem=4$	63
Figure 3.27. Number of Iterations - RegTree and Base methods, $PopAddRem=4$.	64

Figure 3.28. Best GAP(%) - RegTree and MDM-GRASP.	66
Figure 3.29. Average GAP(%) - RegTree and MDM-GRASP.	66
Figure 3.30. Best GAP(%) - RegTree and RegTree+3Features.	69
Figure 3.31. Average GAP(%) - RegTree and RegTree+3Features.	69
Figure 3.32. Solution Representation Design for QAP.	71
Figure 3.33. Best GAP(%) - RegTree, Base and RegTree+3Features.	77
Figure 3.34. Average GAP(%) - RegTree, Base and RegTree+3Features.	77
Figure 4.1. Initial Set of Solutions.	82
Figure 4.2. Rule Set Generation.	83
Figure 4.3. Variable Fixing Iterations.	84
Figure 4.4. The Architecture of Algorithm.	86
Figure 4.5. The comparison with Gurobi for scenario AL_AllPop_3Stop.	99
Figure 4.6. The comparison with Gurobi for scenario AL_HalfPop_3Stop.	101
Figure 4.7. The comparison with Gurobi for scenario WL_AllPop_3Stop.	103
Figure 4.8. The comparison with Gurobi for scenario WL_HalfPop_3Stop.	104
Figure 5.1. Regression Tree Example.	111

Figure 5.2.	Decision boundaries, leaf node ranking, and peak-valley scores from regression tree partitioning.	113
Figure 5.3.	Bound Generation Example.	116
Figure 5.4.	New Solutions Generation.	117
Figure 5.5.	Univariate Split Methodology.	129
Figure 5.6.	Solution space evolution for univariate split iterations (Iterations 1 to 10). Each panel shows the solution space refinement at the corresponding iteration.	130
Figure 5.7.	Multivariate Split Methodology.	131
Figure 5.8.	Solution space evolution for multivariate split iterations (Iterations 1 to 10). Each panel shows the solution space refinement at the corresponding iteration.	132
Figure 5.9.	f5-Linear Slope-Regression Tree on Solution Space-Initial and Iterative.	138
Figure 5.10.	f8-Rosenbrock-Regression Tree on Solution Space-Initial and Iterative.	139
Figure 5.11.	f16-Weierstrass-Regression Tree on Solution Space-Initial and Iterative.	141
Figure 5.12.	BOBYQA-max_depth=8-pair_wise=0.3.	153
Figure 5.13.	COBYLA-max_depth=8-pair_wise=0.0.	154

Figure 5.14. PRAXIS-max_depth=8-pair_wise=0.3.	155
Figure 5.15. NELDERMEAD-max_depth=8-pair_wise=0.3.	156
Figure 5.16. SBPLX-max_depth=8-pair_wise=0.3.	157
Figure 5.17. DIRECT-max_depth=8-pair_wise=0.0.	157
Figure 5.18. CRS2_LM-max_depth=8-pair_wise=0.0.	159
Figure 5.19. ISRES-max_depth=8-pair_wise=0.3.	160
Figure D.1. Permission received from [6] for Figure 1.1: "A general taxonomy of machine learning-supported metaheuristics."	239

LIST OF TABLES

Table 3.1.	Solution Representation of a Solution.	20
Table 3.5.	Pairwise comparisons of population sizes using Tukey's post hoc analysis.	50
Table 3.6.	Population size statistics: GAP(%) and Runtime.	50
Table 4.1.	Solution Representation of a Solution.	81
Table 5.1.	Solution Representation of a Solution	110
Table 5.2.	Summary of Best Scenarios for Each Solver.	149
Table A.1.	GAP (%) according to p/n_{loc} ratio for Three Techniques.	178
Table A.2.	Best and Average GAP (%) - RegTree and Base, PopAddRem=2.	179
Table A.3.	Time and NumOfIt - RegTree and Base, PopAddRem=2.	181
Table A.4.	Best and Average GAP (%) - RegTree and Base, PopAddRem=4.	183
Table A.5.	Time and NumOfIt for RegTree and Base, PopAddRem=4.	184
Table A.6.	Best and Average GAP(%) Values for article Plastino2011.	187
Table A.7.	RegTree, Base, and RegTree+3Features Best GAP (%) Values.	188
Table A.8.	RegTree, Base, and RegTree+3Features Average GAP (%) Values.	190

Table A.9.	QAP Results for thirteen selected problem instances	192
Table B.1.	QPLIB Problem Instances.	193
Table B.2.	QPLIB instances and comparisons for AL_AllPop_3Stop.	194
Table B.3.	QPLIB instances and comparisons for AL_HalfPop_3Stop.	195
Table B.4.	QPLIB instances and comparisons for WL_AllPop_3Stop.	195
Table B.5.	QPLIB instances and comparisons for WL_HalfPop_3Stop.	196
Table C.1.	Function Codes, Names, and Groups in the BBOB Test Suite.	197
Table C.2.	The (%) shrinkage rate values, max_depth=4 (mean).	200
Table C.3.	The (%) shrinkage rate values, max_depth=8 (mean).	201
Table C.4.	Out-of-bound variables count, max_depth=4 (mean).	202
Table C.5.	Out-of-bound variables count, max_depth=8 (mean).	203
Table C.6.	Optimal solution leaf node order, max_depth=4 (mean).	204
Table C.7.	Optimal solution leaf node order, max_depth=8 (mean).	205
Table C.8.	BOBYQA best-to-optimal distance, max_depth=4 (mean).	206
Table C.9.	BOBYQA best-to-optimal distance, max_depth=8 (mean).	207
Table C.10.	COBYLA best-to-optimal distance, max_depth=4 (mean).	208

Table C.11.	COBYLA best-to-optimal distance, max_depth=8 (mean).	209
Table C.12.	PRAXIS best-to-optimal distance, max_depth=4 (mean).	210
Table C.13.	PRAXIS best-to-optimal distance, max_depth=8 (mean).	211
Table C.14.	NELDERMEAD best-to-optimal distance, max_depth=4 (mean).	212
Table C.15.	NELDERMEAD best-to-optimal distance, max_depth=8 (mean).	213
Table C.16.	SBPLX best-to-optimal distance, max_depth=4 (mean).	214
Table C.17.	SBPLX best-to-optimal distance, max_depth=8 (mean).	215
Table C.18.	DIRECT best-to-optimal distance, max_depth=4 (mean).	216
Table C.19.	DIRECT best-to-optimal distance, max_depth=8 (mean).	217
Table C.20.	CRS2_LM best-to-optimal distance, max_depth=4 (mean).	218
Table C.21.	CRS2_LM best-to-optimal distance, max_depth=8 (mean).	219
Table C.22.	ISRES best-to-optimal distance, max_depth=4 (mean).	220
Table C.23.	ISRES best-to-optimal distance, max_depth=8 (mean).	221
Table C.24.	BOBYQA best-to-optimal distance across scenarios (min).	222
Table C.25.	COBYLA best-to-optimal distance across scenarios (min).	223
Table C.26.	PRAXIS best-to-optimal distance across scenarios (min).	224

Table C.27. NELDERMEAD best-to-optimal distance across scenarios (min).	225
Table C.28. SBPLX best-to-optimal distance across scenarios (min).	226
Table C.29. DIRECT best-to-optimal distance across scenarios (min).	227
Table C.30. CRS2_LM best-to-optimal distance across scenarios (min).	228
Table C.31. ISRES best-to-optimal distance across scenarios (min).	229
Table C.32. Optimality gap (min over 5 runs) for solvers w/ original bounds.	230
Table C.33. Summary of Best Scenarios for Each Solver.	230
Table C.34. BOBYQA-Binary comparisons with original bounds (min).	231
Table C.35. COBYLA-Binary comparisons with original bounds (min).	232
Table C.36. PRAXIS-Binary comparisons with original bounds (min).	233
Table C.37. NELDERMEAD-Binary comparisons with original bounds (min).	234
Table C.38. SBPLX-Binary comparisons with original bounds (min).	235
Table C.39. DIRECT-Binary comparisons with original bounds (min).	236
Table C.40. CRS2_LM-Binary comparisons with original bounds (min).	237
Table C.41. ISRES-Binary comparisons with original bounds (min).	238

LIST OF SYMBOLS

$ARSolns$	The number of solutions on which remove-add operations to be applied
MSL	The minimum number of samples required to be at a leaf node, which is given as a parameter to the decision tree regressor
MSS	The minimum number of samples required to split an internal node, which is given as a parameter to the decision tree regressor
n_{cust}	Number of customer in the problem instance
n_{est}	Number of regression tree to be used in Random Forest
n_{loc}	Number of possible facility locations in the problem instance
NIT	The maximum number of successive iteration no improvement
NPM	The maximum number of population management phase
p	The number of facilities to be opened
P_t	Population size in iteration t
$PopAddRem$	The number of solutions to be removed and added to population
$PopSize$	Population Size abbreviation

LIST OF ACRONYMS/ABBREVIATIONS

ANN	Artificial Neural Network
CSP	Constrained Satisfaction Problem
CTAP	Constrained Task Allocation Problem
DM	Data Mining
DMDTS	Data Mining Driven Tabu Search
EC	Evolutionary Computation
FIM	Frequent Itemset Mining
GA	Genetic Algorithm
GAAR	Genetic Algorithm with Association Rule
GRASP	Greedy Randomized Adaptive Search Procedure
JSS	Job Shop Scheduling
KNN	K-nearest-neighbor
MBMP	Matrix Bandwidth Minimization Problem
Meta-RaPS	Metaheuristic for Randomized Priority Search
NS	Neighborhood Search
PSO	Particle Swarm Optimization
RLFAP	Radio Link Frequency Assignment
SA	Simulated Annealing
SVM	Support Vector Machines
TS	Tabu Search
VFA	Variable Fixing Algorithm
VNS	Variable Neighborhood Search
VND	Variable Neighborhood Descent
VRP	Vehicle Routing Problem

1. INTRODUCTION

Heuristic methods are used to solve computationally difficult optimization problems for which exact solution methods may fail to generate solutions of sufficient quality in a reasonable amount of time. Over the years, various heuristics have been designed and successfully applied to diverse optimization problems. Heuristic methods have been observed to exhibit varying performances depending on the type of problem and even on different instances within the same problem type. Metaheuristics are search methodologies that provide guidelines for heuristic optimization [1]. There are various types of metaheuristics, such as Genetic Algorithm (GA), Tabu Search (TS), Simulated Annealing (SA), Greedy Randomized Adaptive Search Procedure (GRASP), and others, that can be applied to real-life problems in various contexts [2]. Some metaheuristics are based on maintaining a population of solutions; therefore, they are called population-based metaheuristics. Others use Neighborhood Search (NS) to move from one solution to the next in the solution space, such as Variable Neighborhood Search (VNS) and Tabu Search. The neighborhood of a solution can be defined by the move mechanisms applied to transform one solution into another. The simplest form of NS is local search, which accepts moves with a better objective function value. The search ends when no improving solution can be found and the resulting solution is called a local optimum. Metaheuristic methodologies involve strategies that aim to escape from local optima and improve the best solution found during the search.

The development of hybrid metaheuristics is a hot search area [3] where hybrid methods are created by either (a) combining concepts and procedures from different existing metaheuristics, or (b) integrating metaheuristics with methodologies from other areas to perform specific tasks that enhance the method. This thesis focuses on the latter approach, specifically on hybridizing a heuristic method with a machine learning (ML) (also called as data analytics or statistical learning) technique. During the execution of the iterations of a heuristic algorithm, valuable data is generated. Using these data for guidance in the metaheuristic algorithm has recently been an active research

area. ML algorithms can generate information from these data that may help in the search process.

Today's technology enables extensive data to be collected and stored. There is an enormous amount of data, and the current data processing needs are beyond the computing capacity of traditional computational models. Hence, ML methods have been developed to obtain invaluable knowledge from continuous and exponentially growing data [4]. The extracted knowledge, expressed in terms of patterns or rules, represents important features of the data set. Learning can be categorized into four types: supervised, semi-supervised, unsupervised, and reinforcement learning.

In the ML field, the main goal is to develop models that can accurately estimate output (also called target, response, dependent) variables based on input (also called predictor, explanatory, independent) variables. Output estimation is carried out primarily for two main reasons: prediction and inference. In prediction, the objective is to estimate the output as accurately as possible using techniques that minimize reducible error. In contrast, inference focuses on understanding the relationship between the input dataset and the output. This understanding allows us to discover the critical factors that influence the prediction, improving our understanding of the underlying processes. Additionally, some techniques combine prediction and inference to improve the accuracy and interpretability of the results [5].

Tree-based methods partition the feature space into hyperrectangles and fit a simple model, such as a constant, within each partition. Despite their conceptual simplicity, these methods are powerful and versatile, making them highly effective for both classification and regression tasks. Tree-based techniques encompass decision trees, bagging, random forests, and boosting. Their popularity stems from their ability to handle complex, non-linear relationships and interactions between features, making them a popular choice for various predictive modeling tasks [5].

The integration of ML techniques into heuristic algorithms opens new pathways for enhancing their performance on combinatorial optimization problems. By using the vast amounts of data generated during the iterations of heuristic algorithms, we can extract valuable patterns and insights that guide the search process more effectively. This study investigates how incorporating an ML method into a heuristic algorithm affects its performance in solving combinatorial optimization problems. To further explore the synergy between ML and optimization methods, it is essential to consider Talbi's taxonomy [6], which outlines three methods for integrating ML into metaheuristics (MHs) (see Figure 1.1). This taxonomy differentiates between problem-level, high-level, and low-level integrations of ML into MHs.

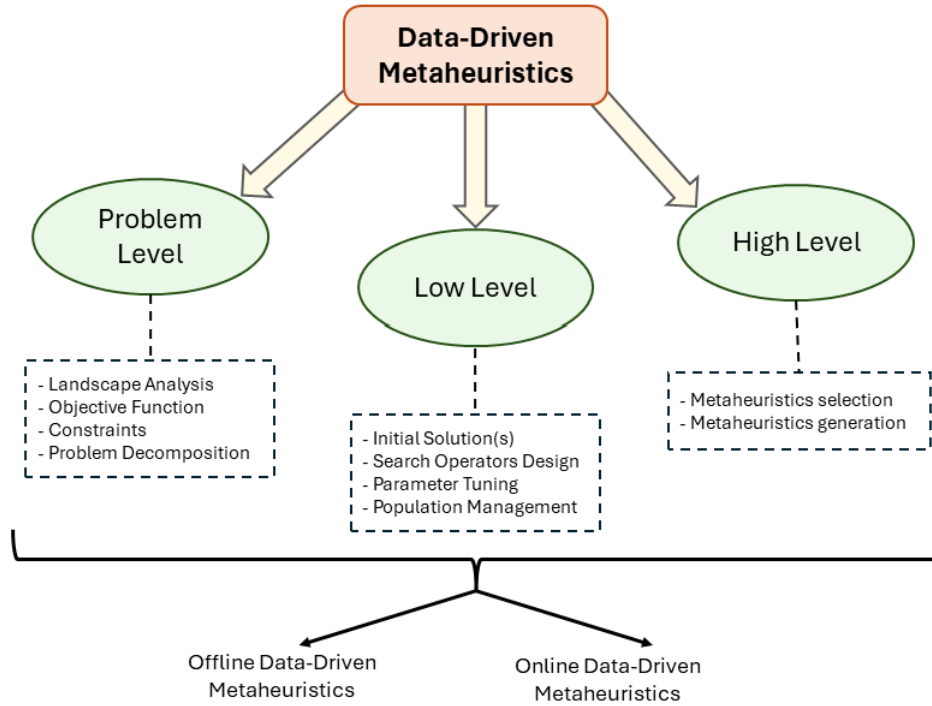


Figure 1.1. A general taxonomy of machine learning-supported metaheuristics [6].

At the problem level, ML aids in modeling the optimization problem itself, which includes defining the objective function, setting constraints, conducting landscape analysis, and decomposing the problem into manageable sub-problems. This integration level helps create a more accurate and efficient representation of the problem that

the metaheuristic aims to solve [7–9]. At the high level, ML contributes to the selection, creation, and design of hybrid MHs. This involves generating new metaheuristic algorithms or selecting the most appropriate ones based on the problem at hand. High-level ML-supported MHs focus on strategic decisions that influence the overall search process, allowing for the creation of more sophisticated and effective hybrid algorithms [10–12]. At the low level, ML enhances specific components of MHs. This includes guiding the initialization of solutions, designing search operators (such as mutation and crossover in evolutionary algorithms), tuning various metaheuristic parameters, and managing populations of solutions. Low-level ML-MHs aim to improve the performance and efficiency of individual components within the metaheuristic framework. Additionally, Talbi’s taxonomy distinguishes between offline and online data-driven MHs. Offline methods utilize pre-collected data to inform the optimization process, making decisions based on a static dataset. In contrast, online methods continuously integrate new data during the optimization process, allowing for dynamic adjustments and improvements based on real-time information.

Our research is concentrated on low-level, online data-driven ML-MHs, where ML optimizes individual components of the algorithms dynamically. By continuously integrating new data, our approach allows for real-time adjustments to the metaheuristic’s local search, population management, and reduction to promising regions, thereby enhancing the effectiveness and efficiency of the search process.

Firstly, it introduces a novel approach for integrating machine learning techniques, specifically regression trees, into population-based metaheuristics. We incorporate ML techniques into the algorithm’s local search and population management phases. Specifically, we use a regression tree in a population-based heuristic to determine the removal and addition of solutions in the local search phase and to generate new solutions in the population management phase when there is no improvement in the current best solution. The ML method extracts patterns from both high-quality and low-quality solutions to guide the search through the solution space and find more effective solutions. We aim to solve an optimization problem more efficiently and effec-

tively by using ML techniques within the heuristic algorithm. This integration not only improves the algorithm’s ability to adaptively explore and exploit the solution space, but also reduces the reliance on handcrafted rules or random operations traditionally used in metaheuristics.

In addition, we develop an algorithm to shrink the solution space for exact methods (i.e., commercial solvers). We utilize split decisions of the regression tree to fix decision variables or to generate lower and upper bounds of the decision variables. By doing this, we reduce the solution space, thus helping exact methods in finding optimal solutions more efficiently. This approach not only benefits exact solution methods but also enhances the performance of any kind of heuristic. By limiting the search space to more promising regions, we enable faster convergence to high-quality solutions, ultimately improving the overall effectiveness of the optimization process.

Together, these contributions represent a significant advancement in addressing combinatorial optimization and continuous nonlinear problems, which are notoriously difficult to solve due to their complex search spaces and vast number of feasible solutions. We develop and propose a robust, tree-based framework that is applicable to any optimization algorithm, whether exact or heuristic. This framework enhances the efficiency and effectiveness of solving complex optimization problems, providing an adaptable tool for both theoretical research and practical applications.

The content of this thesis is organized as follows: Chapter 2 presents the most relevant works in the literature. We develop three different algorithms to improve the quality of solutions to various optimization problems solved by heuristic techniques or commercial solvers. Chapter 3 includes the algorithm used to solve the p -median problem, where a regression tree is used in the local search and population management phases of the algorithm. Chapter 4 describes the way in which regression tree split decisions can be used to reduce the solution space by fixing decision variables in quadratic binary problems. Chapter 5 makes use of a regression tree to solve optimization problems with continuous decision variables and nonlinear objectives.

2. LITERATURE REVIEW

The research stream most relevant to this study is the application of ML algorithms to assist exact or heuristic methods in solving optimization problems. In fact, the reverse has been addressed more frequently in the literature, where optimization methods, such as heuristic algorithms, are used to enhance the performance of ML algorithms [13–16].

In the context of data-driven metaheuristics, Talbi’s taxonomy, presented in Chapter 1, provides a comprehensive classification framework that identifies three levels of integration: problem level, low level, and high level. This taxonomy categorizes the use of ML techniques within metaheuristics, facilitating a structured approach to improving optimization algorithms through data-driven insights. As depicted in Figure 1.1, the framework delineates how ML can affect various stages of the optimization process, from initial solution generation to metaheuristic selection and tuning.

A structured approach to categorizing the integration of ML techniques into MHs considers three properties: the type of information, the purpose, and the localization, as outlined in [17].

- There are two types of information: Apriori Information, which is acquired beforehand, and Dynamic Information, which is extracted or discovered during the search process.
- Another useful criterion for classifying algorithms is the purpose of hybridization. Cooperation can reduce computational time through speed-up techniques, such as simplifying the fitness function or significantly narrowing the search space, for instance, by directing the metaheuristic towards a promising area. Alternatively, cooperation can enhance search quality by incorporating knowledge into operators or other components of the metaheuristic. Integrating ML methods often accelerates the metaheuristic and improves solution quality.

- The final criterion for distinguishing integrations is to identify which stage of the metaheuristic incorporates ML methods, a process known as localization. Hybridization can occur in various parts of the MH, such as evaluation, parameters, encoding, initialization, population management, operators, and local search.

This study aims to improve the solution quality of a metaheuristic algorithm and its computation time by enhancing it through ML methods. One purpose of using ML in metaheuristics is to discover relevant patterns in the best solutions found so far, and the search process may be directed to the promising regions of the search space. An application could be its usage for population management in population-based MHs. Population-based MHs, which use evolutionary computation in nature, are based on maintaining a population of solutions. Evolutionary computation (EC) is inspired by the mechanisms of biological evolution and the behavior of living organisms [18]. In EC algorithms, ML methods can be incorporated for population reproduction and variation, with the aim of reducing the size of the problem, learning the structure of the problem, maintaining population diversity, and predicting promising regions of the solution space. Another application might be to use ML methods to decide whether to add or drop in a local search phase. An ML algorithm can be applied in the local search phase in all MHs.

To illustrate how learning techniques are integrated with other methods, we describe an example from [19], which demonstrates how the Probabilistic Graphical Model (PGM), an ML algorithm, can enhance the learning process of Estimation of Distribution Algorithms (EDAs). In this approach, a population-based heuristic is employed, in which the fitness values of the solutions in the population are first evaluated. Then, a PGM is constructed by modeling the dependencies among solution dimensions, capturing complex interrelations. This model is subsequently used to generate new candidate solutions by sampling from the learned distribution, facilitating a guided search toward improved solutions. The incorporation of the PGM allows the algorithm to balance exploration and exploitation more effectively by preserving learned structural knowledge across generations.

In recent years, there has been an increase in the use of hybrid approaches that combine ML and heuristic methods to solve optimization problems [20–23]. While it is common to use MHs for knowledge discovery, it is less common to use knowledge discovery to improve the performance of MHs [17]. This study aims to explore the latter approach, using ML techniques to improve the effectiveness and efficiency of heuristic algorithms in solving optimization problems.

There is an increasing interest in the combination of ML and heuristic methods. However, using MHs for knowledge extraction can be seen more frequently than the other way, which consists of using knowledge discovery to improve metaheuristics. The literature, including ML integrated with MHs, is reviewed with respect to the localization of the ML method integration.

2.1. Using Machine Learning for Initialization

The initial solution is often generated randomly or using construction heuristics in MHs. A better initial solution can guide the MHs toward a more promising region, which can be advantageous. [24] addresses a dynamic job shop scheduling problem by integrating the K -means algorithm with the VNS method. The authors employ clustering to group similar jobs based on their processing times, thus obtaining prior knowledge. The goal of this integration is to enhance the quality of the solution. The K -means algorithm is applied during the initialization phase.

In the context of population-based heuristics/MHs, [25] proposes an ML-based approach to generate an improved initial population to solve the job-shop scheduling problem (JSSP). This approach aims to uncover the rules behind optimal or near-optimal JSSP schedules. To achieve this, they use a combination of attribute-oriented induction and association rule mining techniques. The Assignment Procedure method proposed by the authors is then applied to heuristically solve the JSSP using the extracted rules. This methodology allows for the generation of numerous solutions that serve as the initial population for population-based heuristics. The effectiveness of the

proposed method is evaluated using GA and particle swarm optimization (PSO). The results demonstrate that the method significantly improves the performance of both GA and PSO, producing better solutions in less computational time.

2.2. Using Machine Learning to Set the Parameters

Every metaheuristic requires the setting of specific parameters that are crucial to its performance and effectiveness. For example, the number of iterations is a common parameter across various metaheuristics. In population-based metaheuristics, such as GA, parameters include population size, crossover rate, mutation rate, and selection method. Other MHs have their own unique parameters, such as tabu tenure in Tabu Search and the cooling factor in Simulated Annealing. Properly tuning these parameters is essential, as they can significantly influence the algorithm's ability to find optimal or near-optimal solutions.

Al-Douli and Rabadi [26] focus on using an Inductive Decision Tree (IDT) to perform online tuning of the parameters in the Metaheuristic for Randomized Priority Search (Meta-RaPS). They apply this idea to the Capacitated Vehicle Routing Problem (CVRP) and demonstrate that the hybrid Meta-RaPS outperforms the original Meta-RaPS in all but four instances out of 14. An ML method is applied to obtain dynamic information about the parameters. The parameters they attempt to tune are $p\%$ (priority percentage), $r\%$ (restriction percentage), $i\%$ (improvement percentage), and the category (or label) of the solution. They construct IDT to find favorable parameters using the knowledge gained from previous Meta-RaPS iterations. Their goal is to improve solution quality.

Lessmann et al. [27] proposed an algorithm to tune the parameters of Particle Swarm Optimization (PSO) using regression models. For each particle, data are collected from successful PSO moves. A regression model is established for each parameter. The independent variables of the regression model are the position vector, flying velocity vector, and best position vector. The target variables are PSO parameters,

estimated using the following forecasting methods: linear regression, Support Vector Machines (SVM), and random regression forests (REGFOR). Among the nonlinear regression models, the most promising one is REGFOR, which yields better results for tuning parameters. They aim to improve solution quality by utilizing dynamic knowledge.

2.3. Using Machine Learning Methods in Population Management

A number of studies, [21, 28–34], suggest that Frequent Itemset Mining (FIM) methods can be used to extract patterns, which represent characteristics of sub-optimal solutions of an optimization problem, to speed-up and improve quality. The study procedure is closely similar to the methods used by Plastino et al. [20] in their study, which applied GRASP, FIM-integrated GRASP (DM-GRASP), and multi-data mining GRASP (MDM-GRASP) on the p -median problem. The modifications mentioned, such as hybridizing GRASP with Variable Neighborhood Descent (VND) and the application of Multi-DM, suggest an adaptation of these previous methods to potentially improve performance and solution quality in optimization problems. The procedure called DM-GRASP consists of two phases. In the first phase, the Elite Set Generation Phase, a random solution is generated, and a local search method is applied to the solution for n iterations. As a result, a solution set with a size of n is obtained. The d best solutions from this set form an elite set. After this first phase, the FIM procedure is applied to extract a set of solutions that occur in $s\%$ of the elite solutions, i.e., with support $s\%$. In their procedure, they use dynamic knowledge. The method allows for the speed-up of the algorithm's convergence and the improvement of the robustness of the GRASP. This hybridized algorithm helps better population management since a solution set is kept at each iteration. In these studies, this approach is applied to different kinds of problems, such as Set Packing Problem, Maximum Diversity Problem, Efficient Server Replication for Reliable Multicast Problem, p -median Problem, 2-Path Network Design Problem and the One-Commodity Pickup-and-Delivery Traveling Salesman Problem (1-PDTSP). In most cases, the hybrid approach achieves better solution quality and faster convergence than the original strategies.

The FIM methodology is combined with population-based MHs to manage the population of solutions. In a study by [35], a hybridized methodology is proposed and applied to the constraint satisfaction problem, specifically the Binary Max-CSP and Radio Link Frequency Assignment Problem (RLFAP). Their hybridization methodology is similar to those in the aforementioned articles. They compare GA and GAAR (Genetic Algorithm with Association Rule) in terms of mean objective values and solution time using different instances of Max-CSP and RLFAP. The results show that in Binary Max-CSP instances, the hybrid method outperforms the simple GA in almost all cases except for one distribution. In RLFAP, the GAAR algorithm demonstrates greater robustness than the standard GA and the breakout GA. Additionally, the hybrid algorithm spent only a small percentage (less than 21%) of the total time on the association rule mining algorithm. In summary, the methodology improves solution quality and speed.

[36] integrates a straightforward clustering approach using the K -means algorithm with an evolutionary path relinking method to address Quadratic Unconstrained Binary Optimization (QUBO) problems. The method involves obtaining an elite set of solutions through GA iterations, partitioning the solutions into clusters, and then selecting pairs of solutions based on their cluster membership to generate smaller numbers of paths with useful features. This approach aimed to improve the efficiency and effectiveness of the evolutionary path relinking method by focusing on generating fewer, but more promising, paths that exploit useful features identified through clustering.

2.4. Using Machine Learning Methods During the Evaluation

In some cases, calculating the fitness function can be computationally expensive, necessitating a reduction in the number of evaluations. By employing ML techniques, it is possible to obtain accurate approximations of the fitness function, thereby reducing computational effort and improving efficiency. This surrogate modeling approach enables the algorithm to prioritize more promising solutions without repeatedly incurring the cost of full evaluations.

Kusiak et al. [37] incorporate four ML techniques—classification and regression tree (C&R Tree), boosting tree regression, support vector machine (SVM), and neural network (NN)—into a GA to solve the power optimization problem of wind turbines. Their model is represented as $\max_x f(x, v), \quad x \in S$, where they aim to identify a function that captures the process in a steady state over time. In the model, the non-controllable variables v , the underlying function $f(\cdot)$, and the feasible search space S are time-dependent. After identifying a functional mapping between a wind turbine’s power output and both controllable and non-controllable variables, they applied the GA to determine the optimal control settings for the problem. Once the optimal vector x_{opt} is determined, it is applied to the wind turbine to achieve maximum power output. They applied all four algorithms and compared their performances, ultimately selecting NN to construct $f(\cdot)$. They used dynamic knowledge by continuously updating the model with data obtained from the GA algorithm.

In [38], the authors incorporate an artificial neural network (ANN) with a multiple-objective GA. They evaluate a large part of the population by approximating it with an ANN, and a small part is still simultaneously evaluated with the original function. The results of tests applied on benchmark instances show that the method approximates the fitness function and can save a computational time ranging from 13% to about 30% in several benchmark problems.

In addition, some authors apply the approximation only within operators such as initialization, mutation, crossover, and selection. This approach mainly avoids fitness calculations for bad-quality individuals, and that will not be kept in the population. For example, in [39], the authors use approximations to improve operators based on the fitness calculation. They generate several possible new individuals and then choose the best ones with respect to a simplified model. The idea is to maintain a large sample of the individuals encountered during the course of the optimization divided into dynamic clusters and periodically compute least squares approximations for the measure of merit and the constraint violations for each large cluster as well as the whole sample. To calculate the approximate fitness function of individuals, the individual is first

classified using a distance weighted K -nearest-neighbor (KNN) to decide either feasible, infeasible-evaluable, or unevaluable. Then, depending on the classification and cluster of the individual, the appropriate approximation functions are used to approximate the fitness. They demonstrate that informed operators significantly improve their performance.

Another way to speed up the MH is to reduce the number of calls to the fitness function. In this kind of approach, a set of individuals who are similar to each other are grouped. The group is considered to have a fixed fitness value equal to a representative. The representative can be a reference individual, medoid, centroid, or mean, etc. Clustering techniques are often used to determine the different sets. Therefore, in [40–42], they use clustering algorithms to deal with large population sizes. For example, [42] proposes the application of a *fuzzy c-means* algorithm to divide the population and select a representative of each cluster. Only representatives are computed, which allows for the reduction of evaluation costs. The fitness value of an individual of a cluster is estimated relative to its associated representative (Figure 2.1). The results from several benchmark problems show that the proposed algorithm has almost the same solution quality in less computational time.

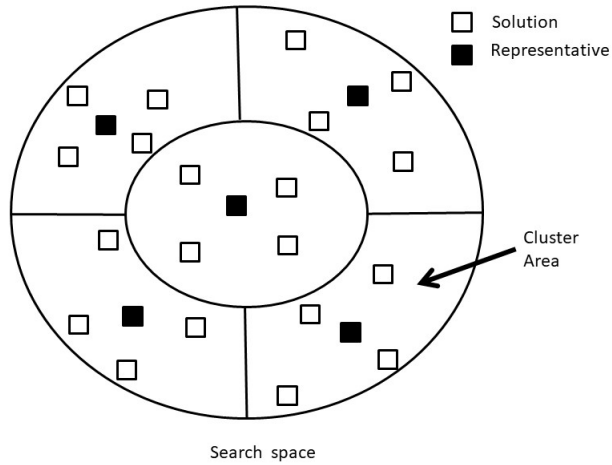


Figure 2.1. Using representatives for the evaluation process through clustering.

2.5. Application of Machine Learning Methods Within Operators

The literature includes several examples of the incorporation of knowledge derived from ML methodologies into operators of MH algorithms.

An Association Rule Mining method is proposed to generate rules between decision variables and objective values in the GA technique [43]. This approach is applied to three weighted constrained satisfaction problems: graph coloring, max-cut, and radio link frequency assignment (RLFAP). These rules are applied in two scenarios. The first scenario involves population management, while in the second scenario, the rules are incorporated into the genetic operators to generate new individuals. A probability distribution, which accounts for the frequency of values in the generated patterns, is computed. The hybrid method usually achieves statistically better results than the simple genetic algorithm for the selected instances. The GAARm (used in the mutation operator) algorithm performs better than the GAARs (applied to individuals) hybrid version for nearly all instances of graph coloring and RLFAP problems.

A co-evolutionary GA is proposed that incorporates a decision tree algorithm (C4.5) and the CN2 algorithm to discover schemata in the crossover process [44]. This method is applied to the Multidimensional Knapsack Problem and Constraint Satisfaction Problems. Findings suggest that integrating evolutionary computation with machine learning techniques enhances the search capability of conventional coevolutionary GA.

2.6. Using Machine Learning Methods in Local Search

ML methods can be effectively hybridized with MHs during the local search stage, and the literature is rich with examples demonstrating this approach. The integration of ML techniques helps to escape local optima and enhances both intensification and diversification processes. By benefiting from ML to analyze and interpret patterns in the search space, MHs can dynamically adjust their strategies, leading to improved per-

formance and more robust solutions in complex optimization problems. This synergy allows the algorithm to focus its search efforts more intelligently, reducing unnecessary exploration of unpromising regions.

The See5 decision tree methodology is applied to learn from the population of good solutions generated by a GA in the Job Shop Scheduling (JSS) problem [45]. Prior knowledge is effectively utilized by generating a decision tree from the attributes of known problem instances. This hybrid approach results in better solution quality.

A hybrid methodology incorporating a Hyper-Plane Based Classifier into the TS algorithm is proposed to guide local search and help escape local optima [46]. This approach involves learning rules from a training set of problem instances and applying these rules as an escape procedure to new instances, thereby utilizing prior knowledge. The selected problem instances are from the Constrained Task Allocation Problem (CTAP) and Matrix Bandwidth Minimization Problem (MBMP). The performance of the proposed method is compared with existing solutions, including the Dynamic Memory Tabu Search (DMDTS) algorithm. Although some results from the literature, such as VNS by Lusa and Potts (2008), outperform DMDTS, others, such as HTS by Chen and Lin (2000), indicate that DMDTS outperforms HTS.

Simple decision trees, random forests, and linear SVMs are incorporated into the Meta-RaPS algorithm for the Vehicle Routing Problem to guide the search process by distinguishing between good and suboptimal solutions [22]. This approach measures a solution's structure and an instance's characteristics using problem-specific metrics. The findings indicate that understanding the structural characteristics of good solutions leads to better outcomes, which ultimately improves the quality of the solutions.

2.7. Using Machine Learning Methods to Support Optimization Solvers

ML has emerged as a powerful tool for improving traditional optimization solvers, offering innovative approaches to improve efficiency, adaptivity, and scalability in solv-

ing complex combinatorial and nonlinear optimization problems. [47] explores three primary ways in which ML is integrated with combinatorial optimization algorithms. First, in end-to-end learning, ML models, such as pointer networks, are trained to directly output solutions to problems such as the Traveling Salesman Problem (TSP) using supervised or reinforcement learning techniques [48]. Second, in order to learn the best configuration of algorithms, ML provides valuable insight to optimize algorithm parameters, such as improving decomposition and linearization strategies in mixed-integer programming [49,50]. Lastly, ML models are used in a branch-and-bound framework to make critical decisions in the tree, such as selecting branching variables [51] and predicting the effectiveness of heuristics to streamline search trees [52]. Together, these approaches demonstrate the versatility and effectiveness of ML in solving combinatorial optimization problems by either directly generating solutions or enhancing traditional optimization methods.

Despite several successes, integrating ML into optimization still faces challenges, particularly in addressing nonlinearities and computational inefficiencies in complex systems. [53] demonstrates the potential of ML to overcome these issues by employing a random forest regressor to dynamically improve the relaxation parameter (ω^o) in the Picard iterative solver. Two approaches are applied: In the first approach, the random forest predicts the optimal relaxation parameter directly, while in the second approach it predicts the number of inner nonlinear iterations to guide adjustments to the relaxation parameter. Both approaches leverage dimensionless physical parameters and generalize across reservoir scenarios, achieving a 24% reduction in nonlinear iterations while maintaining accuracy. This highlights ML's effectiveness in enhancing nonlinear solvers.

3. REMOVE-ADD ALGORITHM AUGMENTED BY POPULATION MANAGEMENT

The purpose of this chapter is to explore our tree-learning integrated heuristic algorithm, called as the “Remove-Add Algorithm Augmented by Population Management,” which we refer to as *RegTree* for ease of comparison with other methods. Our proposed methodology comprises five key components: initialization, parameter settings, selection of solutions for improvement, local search, and population update.

The initial population is generated randomly, while parameter settings are optimized through trial and error. A regression tree is employed to navigate the solution space effectively, focusing on selecting solutions with potential for improvement. By utilizing decisions derived from poor solutions, as identified by the regression tree, we aim to enhance these selected solutions. Specifically, the regression tree helps us identify decisions that typically degrade good solutions, allowing us to reverse-engineer these to achieve improvements. Given the limited exploration capabilities inherent in certain stages, we implement additional operations during the population update phase. This involves generating new high-quality solutions through the application of a regression tree. Notably, the local search and population management stages can operate independently due to the modular and component-based design of our algorithm.

3.1. General Scheme

This study aims to enhance solution quality and analyze the impact of integrating machine learning (ML) techniques into heuristic algorithms. Following the taxonomy proposed by [17], we employ dynamic knowledge to improve solution quality and efficiency in solving combinatorial optimization problems. Specifically, ML methods are incorporated into the local search and population management phases, enabling the algorithm to navigate the solution space more effectively. Our approach utilizes a regression tree within a population-based heuristic to guide decision-making during the

solution refinement process. The regression tree identifies patterns from previous iterations and provides insights into where improvements can be made. For instance, in the local search phase, the regression tree is employed to determine suitable decisions for removal and addition operations, which are performed using a greedy algorithm. This framework also aids in the selection of candidate solutions for improvement, as demonstrated in the p -median problem, where binary decisions (e.g., facility opening decisions) are used to maintain feasibility. The algorithm transitions to the population management phase if no improvement is observed in the best solution after a predefined number of successive iterations. During this phase, new solutions are generated by utilizing the regression tree to replace underperforming solutions in the population. This integration of ML techniques within the metaheuristic framework facilitates the extraction of valuable insights from sub-optimal or poor solutions, which guide the search process in subsequent iterations. By identifying and employing patterns, the algorithm strategically navigates the combinatorial solution space to achieve better results.

3.1.1. Feasibility and Constraints in Solution Representation

In optimization problems with specific decision constraints, feasibility can often be ensured directly during the solution generation process by embedding constraints into the solution representation. This approach minimizes the need for repair mechanisms and guarantees that solutions are feasible by design. For example:

- In the p -median problem, the constraint of opening exactly p facilities is satisfied during solution generation by ensuring p decisions are made during initialization. This ensures feasibility from the outset, as the total number of facilities adheres to the problem's requirements.
- In the Quadratic Assignment Problem (QAP), the constraint that each facility must be assigned to one location (and each location can contain only one facility) is incorporated into the solution representation. This is achieved by designing a binary column for each location, where each column contains exactly one value

of 1, and the position of this 1 indicates the facility assigned to that location. This representation ensures a one-to-one correspondence between locations and facilities, maintaining feasibility. For N locations, this results in N binary features (not an $N \times N$ matrix).

By embedding problem-specific constraints into the solution representation, feasibility is inherently maintained, eliminating the need for additional repair mechanisms.

3.1.2. Handling Complex Constraints

Certain constraints, which cannot be directly incorporated into the solution representation or calculated trivially based on the solution itself, require repair mechanisms to ensure feasibility. For problems where solution representation or decision-based calculations are not inherently constraint-aware, either solving the problem becomes infeasible, or a problem-specific repair mechanism must be implemented. Repair mechanisms are typically applied:

- After executing a move operator in the local search phase, ensure that the updated solution adheres to constraints.
- After generating new solutions during population updates, to correct any infeasibilities introduced in the process.

By incorporating repair mechanisms where necessary, the algorithm maintains feasibility while addressing problems with more complex or dynamic constraints. However, this study focuses on problems with static decision structures, where feasibility can be preserved during solution generation without the need for sequential decisions or complex repair processes.

3.1.3. Solution Representation

The algorithm is designed to solve problems that can be represented using binary solution formats. Converting solutions into a binary format enables efficient handling in subsequent steps, such as regression tree-based decision-making. Once binary decisions are made (e.g., 1s and 0s), the objective value can be calculated based on a pre-calculated matrix, such as a pairwise distance matrix in the p -median problem. While this approach is effective for static decision structures, the algorithm is not designed to handle sequential decision-making or dynamic constraints.

The solution representation is given in Table 5.1. In our algorithm, we use binary solution representation. The binary decision variable y_i , facility is opened in location i or not, is represented. The decision variable x_{ij} is calculated according to the decision of y_i variable by taking the minimum distance of possible routings from facility i to customer j . After that, according to these decisions, we calculate objective function value by summing up distances from opened facilities to assigned customers.

Table 3.1. Solution Representation of a Solution.

1	2	3	...	n	n+1	...	$n_{loc}-1$	n_{loc}				
0	1	0	.	.	.	1	0	.	.	.	1	1

legend:

$i \leftarrow location$

y_i

3.1.4. Algorithm Workflow and Visual Explanation

In this study, we incorporate the regression tree algorithm in local search and population management phases to show the usability in metaheuristics. Before explaining, a graphical representation shows how the remove-add decisions are made.

This visualization helps to present the underlying concept of the proposed algorithm, making it easier to understand the integration process and its benefits.

Initially, a population of solutions is generated randomly based on the solution representation outlined in Table 5.1. Each row represents a solution, and each column corresponds to a decision, represented as a binary variable (e.g., 1 indicating inclusion and 0 indicating exclusion). The last column contains the objective function values (targets) for each solution, which measure the quality of the solution. For example, in the p -median problem, 1 may indicate the inclusion of a facility in the solution, while 0 indicates its exclusion. This approach ensures that the representation is flexible and can be adapted to different combinatorial optimization problems.

Figure 3.1 illustrates this process. The top section of the figure displays a sample population of solutions, with n columns representing the binary decision variables and the last column showing the target objective values. Below this, the regression tree constructed from this dataset is shown. The regression tree partitions the solution space based on the decision variables. Each internal node represents a decision rule (e.g., “ $V_{39} < 0.5$ ”), splitting the population based on whether a specific decision is included or excluded. The leaf nodes contain the predicted target values (objective function values) and the number of solutions (n) falling into each partition. This structure enables the algorithm to identify patterns and relationships between decision variables and their impact on the objective function. The tree highlights groups of solutions with similar characteristics and objective function values, supporting a more focused and strategic search process.

	1	2	...	18	...	39	40	...	n-1	n	Target (Objective Value)
1	0	0	...	1	...	0	1	...	0	1	8752
2	1	0	...	1	...	0	0	...	1	0	9572
⋮	⋮	⋮		⋮		⋮		⋮	⋮		⋮
PopSize	1	1	...	0	...	1	0	...	0	0	3458

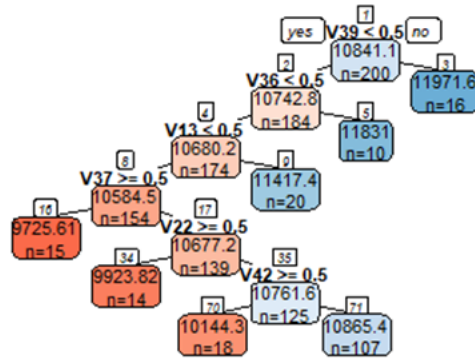


Figure 3.1. Regression Tree Example.

After constructing the regression tree, the worst leaf node is identified based on the optimization objective. For minimization problems, the worst leaf node corresponds to the one with the largest predicted target value (objective function value), whereas for maximization problems, it is the one with the smallest predicted target value. This node represents the subset of solutions with the poorest objective values according to the optimization goal. In Figure 3.2, the worst leaf node is highlighted in the darkest blue within the red circle. This node has a predicted target value of 11,971.6 and contains a sample size of $n = 16$. The path leading to this node is defined by a series of decision rules (e.g., “ $V39 \geq 0.5$ ”), which progressively partition the solution space to isolate the least-performing group of solutions. Identifying the worst leaf node provides insights into low-quality solutions, enabling the algorithm to refine the population by targeting these solutions for improvement during the Remove-Add phase.

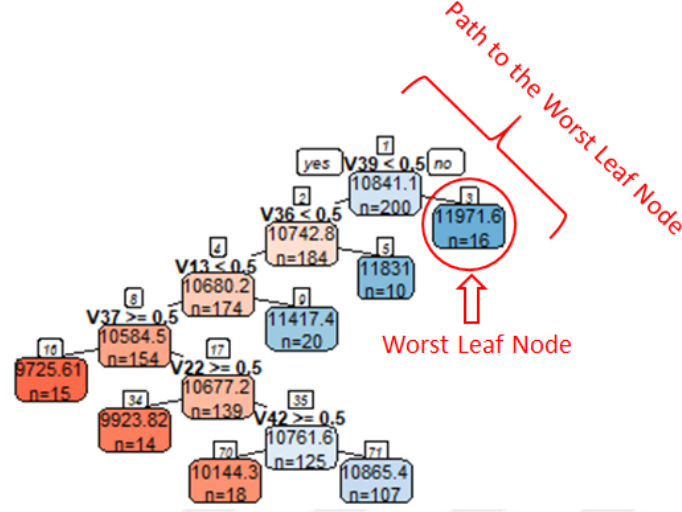


Figure 3.2. Regression Tree Example with the Worst Leaf Node.

From the same regression tree, the best leaf node is also identified, reflecting the optimization objective. For minimization problems, the best leaf node is the one with the smallest predicted target value (objective function value), while for maximization problems, it is the one with the largest predicted target value. This node represents the subset of solutions with the most favorable objective values according to the optimization goal. In Figure 3.3, the best leaf node is highlighted in the darkest red within the red circle. This node has a predicted target value of 9,725.61 and contains a sample size of $n = 15$. The path leading to this node is defined by a series of decision rules (e.g., “ $V39 < 0.5$, $V36 < 0.5$, $V13 < 0.5$, and $V37 \geq 0.5$ ”), which progressively partition the solution space to isolate the best-performing group of solutions. Identifying the best leaf node allows the algorithm to leverage high-quality solutions in the Population Management phase, where new solutions are generated by prioritizing frequently occurring features within this group.

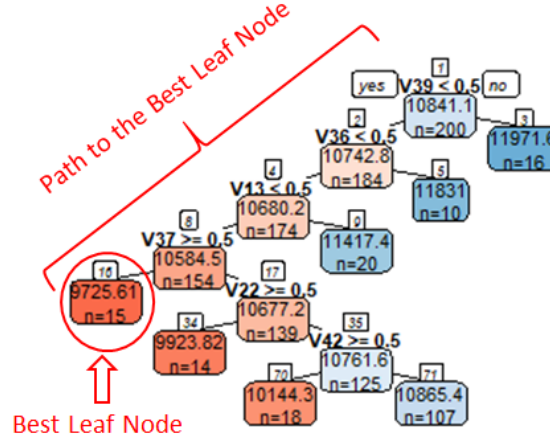


Figure 3.3. Regression Tree Example with the Best Leaf Node.

To enhance the effectiveness of the remove-add operations in our algorithm, we utilize the worst leaf node of the regression tree. The Remove-Add phase of the algorithm is applied to iteratively refine and improve each solution in the best leaf node. For each solution, a decision to remove is determined based on the information from the worst leaf node. Decisions that frequently appear in the low-quality solutions within the worst leaf node are considered candidates for removal, as they are likely to contribute to poor solution quality.

After determining the decision to remove, the decision to add is selected using a greedy algorithm. This algorithm identifies the candidate that provides the greatest improvement to the objective function when incorporated into the solution. This systematic approach ensures that each solution is refined by removing less effective decisions and replacing them with more promising ones.

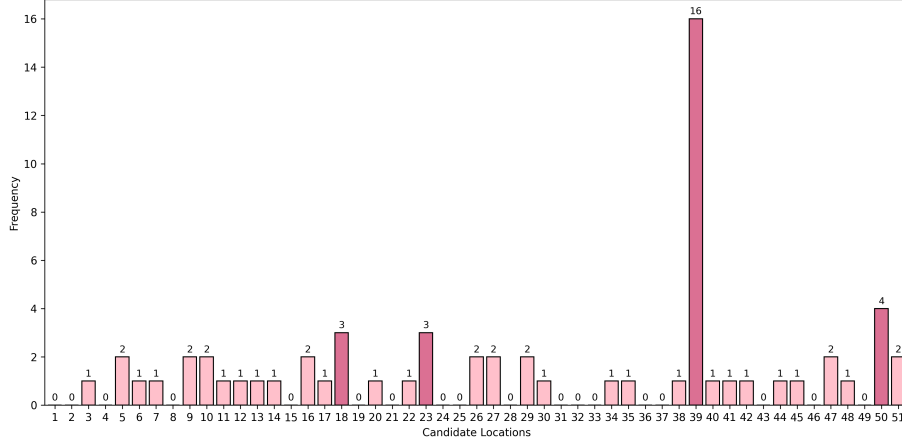


Figure 3.4. Histogram of decisions in solutions of the Worst Leaf Node.

To support this process, the occurrences of each candidate decision within the worst leaf node are counted and normalized to produce relative frequencies, highlighting the prominence of each decision in the poor-quality solutions. By utilizing these frequencies, informed decisions are made about which decisions to remove. Removing high-frequency decisions ensures the elimination of components that consistently degrade solution quality, thereby improving the overall performance of the population in subsequent iterations. The histogram in Figure 3.4 illustrates the frequencies obtained from the solutions in the worst leaf node shown in Figure 3.2.

To calculate the relative frequencies, the total occurrences of each candidate decision in the solutions within the worst leaf node are counted (as shown in the histogram in Figure 3.4). These counts are then normalized by dividing each count by the total number of occurrences across all decisions in the worst leaf node. Mathematically, the relative frequency r_i for a candidate decision i is calculated as:

$$r_i = \frac{f_i}{\sum f_i}, \quad (3.1)$$

where f_i represents the frequency of candidate decision i , and $\sum f_i$ is the total frequency of all candidate decisions in the worst leaf node.

1	2	3		37	38		49	50	51
0	0	0.0625	...	0	0.0625	...	0	0.25	0.125

Figure 3.5. Relative frequencies obtained from frequencies of solutions in the Worst Leaf Node.

These relative frequencies, presented in Figure 3.5, are used to identify the remove decisions when applying the Remove-Add move operator to each solution in the best leaf node, to improve the high-quality solutions further to reach optimal. This process aims to refine these high-quality solutions further, improving them and guiding the search toward even better solutions, potentially reaching the optimal solution. The process for determining remove decisions is demonstrated in Figure 3.6.

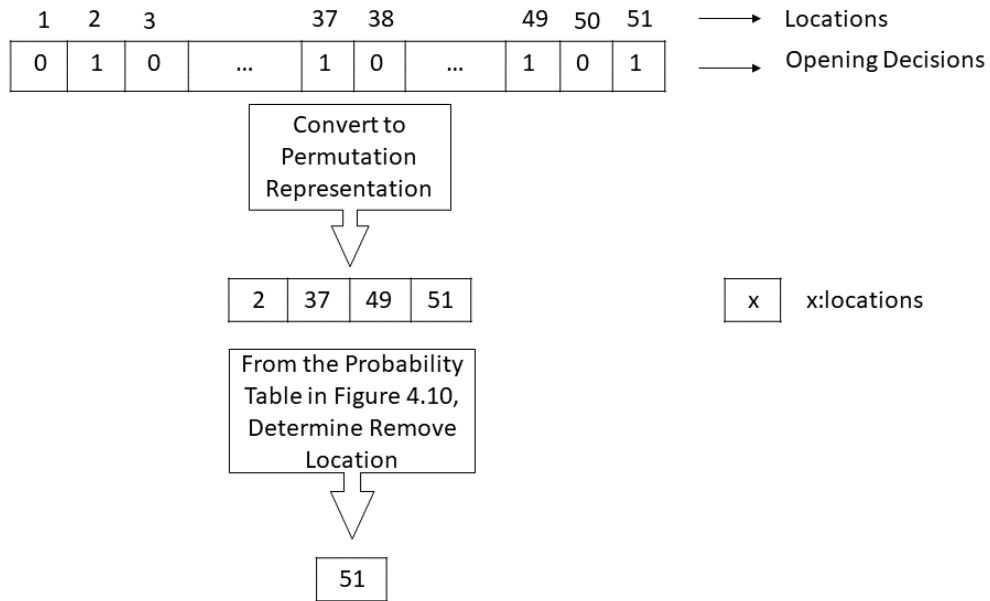


Figure 3.6. The determination of the remove decision.

A frequency analysis is performed by examining the solutions in the best leaf node. This frequency indicates how often each candidate decision appears in the solutions within the best leaf node. By analyzing these frequencies, we can identify the decisions that are the most frequently included in high-quality solutions. The histogram that illustrates these frequencies, obtained from the best leaf node shown in Figure 3.3, is provided in Figure 3.7.

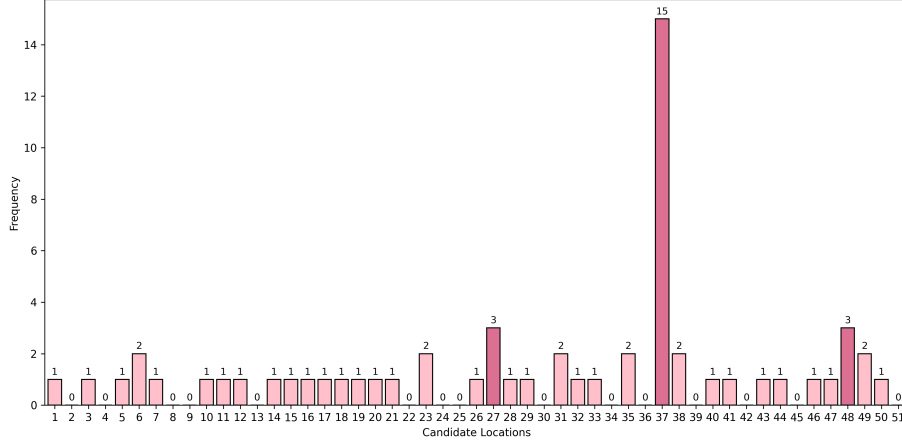


Figure 3.7. Histogram of decisions in solutions of the Best Leaf Node.

In the Population Management phase of the algorithm, after removing the k worst solutions from the population, we generate new solutions using the relative frequencies derived from the best leaf node. These relative frequencies allow us to prioritize candidate decisions that frequently appear in the best-performing solutions, ensuring that the newly generated solutions are likely to perform well.

To calculate the relative frequencies, we first count the total occurrences of each candidate decision in the solutions within the best leaf node. These counts are then normalized by dividing each count by the total number of occurrences across all decisions in the best leaf node. Mathematically, if f_i represents the frequency of candidate decision i and $\sum f_i$ is the total frequency of all decisions in the best leaf node, the relative frequency r_i is calculated as shown in Equation 3.1. The resulting relative frequencies, illustrated in Figure 3.8, represent the likelihood of each candidate decision being included in a high-quality solution. These probabilities are then used to guide the generation of new solutions.

1	2	3	...	37	38	...	49	50	51
0.067	0	0.067	...	1	0.134	...	0.134	0.067	0

Figure 3.8. Relative frequencies of the Best Leaf Node solutions.

Using the relative frequencies derived from the best leaf node, new solutions are generated by probabilistically selecting candidate decisions. Decisions with higher

relative frequencies are more likely to be included in the newly generated solutions. This approach ensures that the solutions added to the population during the Population Management phase are influenced by the patterns observed in the best performing solutions, promoting convergence toward high-quality regions of the solution space. By prioritizing frequently occurring decisions from the best leaf node, the Population Management phase effectively benefits from the regression tree's insights to improve the overall quality of the population in subsequent iterations.

In summary, within the Population Management phase of the algorithm, newly added solutions are determined by using the frequencies obtained from the solutions of the best leaf node in each iteration. During the Remove-Add phase, the decisions to be removed are identified based on the frequencies calculated from the solutions of the worst leaf node in each iteration.

3.1.5. Overview of the Population Management Integrated Add-Remove Algorithm Flowchart

This section presents the flowchart and a detailed explanation of the Population Management Integrated Add-Remove Algorithm, which incorporates a regression tree for iterative solution refinement. The algorithm combines population management techniques with a structured approach to identify and improve high-quality solutions while addressing poor-performing ones.

The flow chart in Figure 3.9 illustrates the steps of the augmented population management remove-add algorithm, which combines regression tree analysis and population update techniques which are the remove-add move operator and to iteratively improve the quality of solutions. Below is a detailed explanation of the algorithm's process:

The algorithm begins with an initial population of solutions. The iteration counter t and the non-improvement counter NoImpt are initialized to zero.

A regression tree is constructed using the values of the objective function of the current population as the dependent variable (target) and the solution representation (e.g., decision variables) as the independent variables (features). This tree partitions the solution space into leaf nodes, with each node containing a subset of the population and its corresponding objective values.

From the constructed regression tree, the best leaf node is identified as the node containing the highest-quality solutions. For minimization problems, this is the node with the smallest average predicted objective value, while for maximization problems, it is the node with the largest. These solutions are selected as *SelectedLeafNodeSolutions*.

The *SelectedLeafNodeSolutions* parameter is made parametric to allow for experimental trials. Instead of always using the best leaf node, alternatives such as the second-best leaf node or the maximum variance leaf node (characterized by high variability in objective values) are also considered. This flexibility provides additional diversity and robustness to the algorithm, especially during experimentation.

The Remove-Add operation is applied to the *SelectedLeafNodeSolutions* as follows:

- (i) A decision is removed based on the worst leaf node of the regression tree. The worst leaf node contains solutions with the poorest objective values, making it a reliable indicator for identifying decisions to remove.
- (ii) A new decision is added using a greedy algorithm. This algorithm selects the candidate decision with the highest potential to improve the objective value when incorporated into the solution.

After the Remove-Add operation, the algorithm evaluates the improvement in the objective value of the best solution. If no improvement is observed ($\text{Improvement} = 0$), the counter NoImpt is incremented. If NoImpt exceeds a predefined threshold (N_T), the algorithm moves to the population management phase.

In the population management phase, the k worst solutions in the population are removed. Simultaneously, k new solutions are generated based on the relative frequencies of candidate decisions derived from the best leaf node. These frequencies reflect the likelihood of each decision being part of a high-quality solution. The newly generated solutions are added to the population, ensuring continued refinement and exploration of the solution space.

If frequency differences (FreqDiffSum) between iterations converge to zero, indicating stagnation, the *SelectedLeafNodeSolutions* are updated to include either the second-best leaf node or the maximum variance leaf node. This step promotes diversification and avoids premature convergence by exploring different parts of the solution space.

The algorithm continues to iterate until the maximum number of iterations (T) is reached. Once $t = T$, the process terminates, and the final population containing the optimized solutions is returned.

Key features and benefits of the algorithm include:

- Utilization of the regression tree to partition the solution space effectively, identifying high-quality and poor-quality solution sets.
- Parametric selection of leaf nodes, allowing for experimentation with alternatives such as the second-best or maximum variance leaf nodes to enhance robustness.
- Application of Remove-Add operations to iteratively refine solutions by removing low-quality features and adding promising ones.
- Population management to balance exploration (diversification) and exploitation (intensification) by removing poor solutions and generating new ones based on high-performing solution patterns.
- Adaptability to ensure robustness by switching to alternative leaf nodes when stagnation is detected.

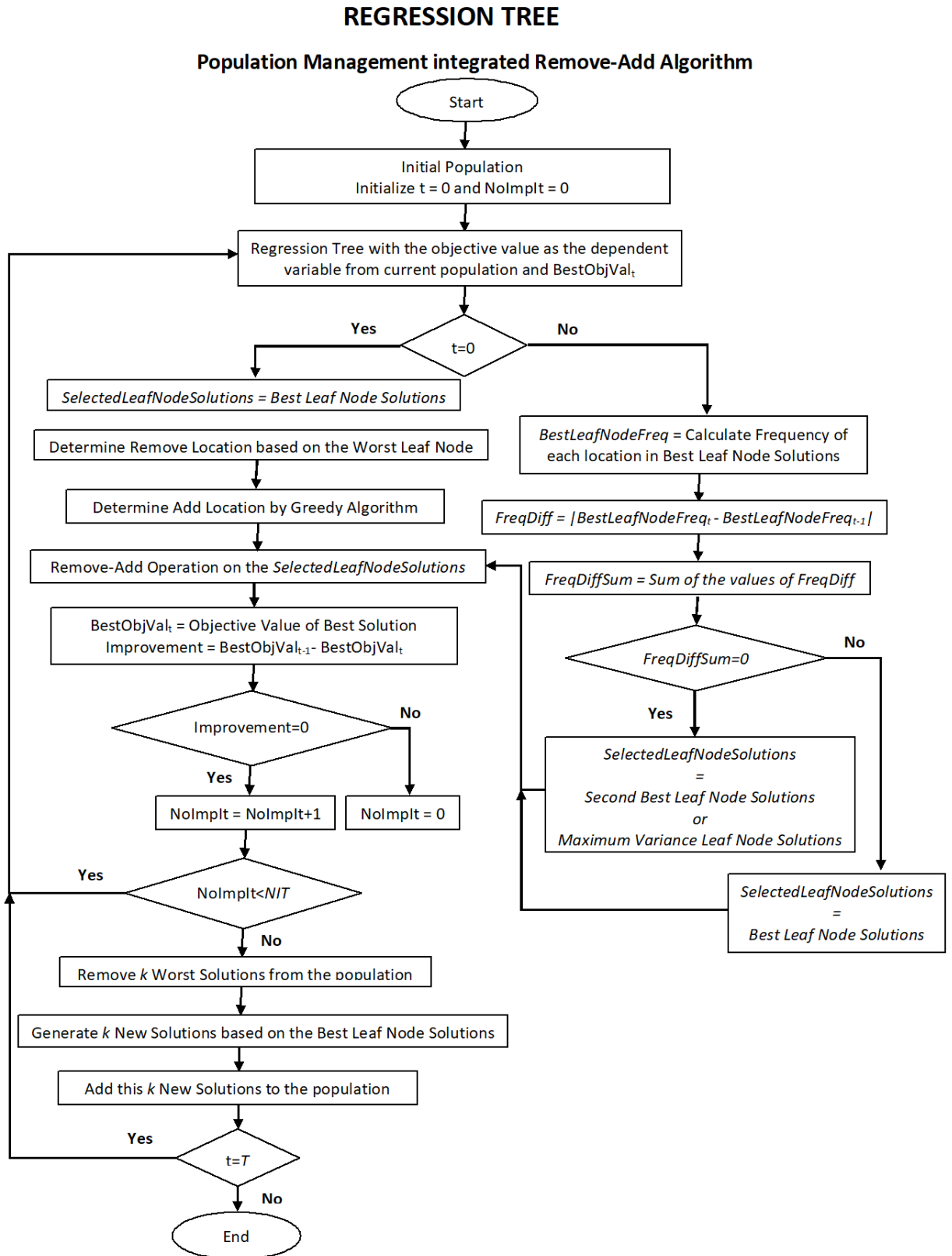


Figure 3.9. Flow Chart for Population Management Integrated Add Remove Algorithm Using Regression Tree.

3.1.6. Pseudo-Codes of RegTree Algorithm

The pseudo-codes presented in this section are written under the assumption that the objective function of the problem is a minimization problem. All operations, such as selecting the best solutions and worst solutions and calculating improvements, are defined accordingly. If the objective function were to be maximized, the logic for these operations would need to be adjusted appropriately (e.g., selecting the maximum objective values instead of the minimum).

The main algorithm iteratively manages a population of solutions, aiming to find the best solution for a given optimization problem. The process begins by initializing a population (P_t) of size $PopSize$, where solutions are randomly generated from the search space (S), and their corresponding objective values ($O(P_t)$) are calculated. The algorithm also initializes several key parameters, including the maximum number of population management steps (NPM), the maximum number of iterations without improvement (NIT), the number of solutions selected for Remove-Add operations ($n_{ARSolns}$), and the number of solutions to be removed and added during population updates ($n_{PopAddRem}$).

During each iteration, a regression tree ($\mathcal{T}$) is fitted to the population using the objective values as the target variable. The tree is then used to identify promising subsets of solutions from specific leaf nodes based on a given criterion, such as selecting the best-performing solutions ($\mathcal{S}_{best}$) or the worst-performing solutions ($\mathcal{S}_{worst}$). Relative frequencies ($RelFreq$) are computed from these subsets, guiding the Remove-Add operation, which updates the population by modifying selected solutions. The best solution ($BestSoln_t$) and its objective value ($BestObj_t$) are tracked throughout the iterations. If no improvement is observed over a specified number of iterations, the population is updated by removing the worst solutions and introducing new solutions generated from selected subsets of the population.

Algorithm 1 Main Algorithm

```

1: Init:
2: Initialize  $PopSize$ ,  $MSL$ ,  $MSS$ ,  $n_{ARSolns}$ ,  $n_{PopAddRem}$ ,  $NPM$ ,  $NIT$ 
3:  $P_t, O(P_t) \leftarrow \text{InitializePopulation}(PopSize, S)$ 
4:  $t \leftarrow 0$ 
5:  $BestObj_t \leftarrow \min(O(P_t))$ ,  $BestSoln_t \leftarrow \arg \min(O(P_t))$ 
6: for  $i \leftarrow 1$  to  $NPM$  do
7:    $NoImpIt \leftarrow 0$ 
8:   while  $NoImpIt \leq NIT$  do
9:     Fit a regression tree  $\mathcal{T}$  on  $P_t$  with  $O(P_t)$ 
10:     $\mathcal{S}_{\text{remove\_add}} \leftarrow \text{GetSolutionsFromLeafNode}(\mathcal{T}, O(P_t), \text{criteria} = \text{"best"},$ 
       $n_{ARSolns})$ 
11:     $\mathcal{S}_{\text{worst}} \leftarrow \text{GetSolutionsFromLeafNode}(\mathcal{T}, O(P_t), \text{criteria} = \text{"worst"})$ 
12:     $t \leftarrow t + 1$ 
13:     $RelFreq \leftarrow \text{CalculateRelativeFrequencies}(\mathcal{S}_{\text{worst}})$ 
14:     $P_t, O(P_t) \leftarrow \text{ApplyRemoveAdd}(\mathcal{S}_{\text{remove\_add}}, RelFreq, P_t, O(P_t))$  (Alg. 5)
15:     $BestObj_t \leftarrow \min(O(P_t))$ ,  $BestSoln_t \leftarrow \arg \min(O(P_t))$ 
16:     $\text{Improvement} \leftarrow BestObj_t - BestObj_{t-1}$ 
17:    if  $\text{Improvement} \geq 0$  then
18:       $NoImpIt \leftarrow NoImpIt + 1$ 
19:    else
20:       $NoImpIt \leftarrow 0$ 
21:    end if
22:  end while
23:   $P_t, O(P_t) \leftarrow \text{PopulationUpdate}(P_t, O(P_t), n_{PopAddRem})$ 
24: end for
25:  $BestObj_t \leftarrow \min(O(P_t))$ ,  $BestSoln_t \leftarrow \arg \min(O(P_t))$ 
26: Output: Best solution  $BestSoln_t$  with objective value  $BestObj_t$ 

```

Algorithm 2 InitializePopulation

- 1: **Input:** $PopSize$, S
 - 2: $P \leftarrow$ Generate $PopSize$ random individuals from S
 - 3: $O(P) \leftarrow$ Calculate objective values for P
 - 4: **Output:** Population P , Objective values $O(P)$
-

The Algorithm 2, InitializePopulation function, is responsible for generating the initial population of solutions. Given the population size ($PopSize$) and the search space (S), this function randomly generates $PopSize$ solutions. For each solution, the corresponding objective value is calculated, resulting in a population (P) and its associated objective values ($O(P)$). The output of this function serves as the starting point for the iterative optimization process. Although the initial population is generated randomly, its diversity is crucial for the optimization process, as it effectively influences the algorithm's ability to explore the search space and avoid premature convergence.

The Algorithm 3, GetSolutionsFromLeafNode function, dynamically selects subsets of solutions from specific leaf nodes of a regression tree ($\mathcal{T}$) fitted to the population. The selection is guided by a specified criterion, such as “best”, “worst”, or “max_var”. For each leaf node, the function calculates the mean objective value ($\bar{O}_l$) and variance (σ_l^2) of the solutions within that node. These statistics are used to identify subsets like the best leaf node solutions ($\mathcal{S}_{\text{best}}$), the second-best leaf node solutions ($\mathcal{S}_{\text{sec_best}}$), the worst-performing solutions ($\mathcal{S}_{\text{worst}}$), and the solutions within the highest variance leaf node ($\mathcal{S}_{\text{max_var}}$). The function outputs a subset ($\mathcal{S}_{\text{selected}}$) based on the provided criterion, optionally limiting the number of selected solutions to n_{ARSolns} .

Algorithm 3 GetSolutionsFromLeafNode

1: **Input:**

- Regression tree $\mathcal{T}$ fitted on population P_t
- Objective values $O(P_t)$
- Selection criteria, *criteria* (e.g., “best”, “worst”, “max_var”)
- Number of solutions to select, n_{ARSolns} (optional)

2: **Output:** $\mathcal{S}_{\text{selected}}$: Selected solutions from the leaf node based on *criteria*

3: Extract leaf node IDs L_i for each solution S_i

4: **for** each leaf node $l \in \mathcal{T}$ **do**

5: Calculate the mean objective value $\bar{O}_l$ for leaf node l

6: Calculate the variance of objective values σ_l^2 for leaf node l

7: **end for**

8: Sort leaf nodes by mean objective values $\bar{O}_l$

9: Initialize a mapping $\mathcal{S}_{\text{all}}$:

10: $\mathcal{S}_{\text{all}}[\text{“best”}] \leftarrow$ Solutions from the leaf node with $\min(\bar{O}_l)$

11: $\mathcal{S}_{\text{all}}[\text{“sec_best”}] \leftarrow$ Solutions from the leaf node with the second $\min(\bar{O}_l)$

12: $\mathcal{S}_{\text{all}}[\text{“worst”}] \leftarrow$ Solutions from the leaf node with $\max(\bar{O}_l)$

13: $\mathcal{S}_{\text{all}}[\text{“max_var”}] \leftarrow$ Solutions from the leaf node with $\max(\sigma_l^2)$

14: Select the subset based on the criteria:

15: $\mathcal{S}_{\text{selected}} \leftarrow \mathcal{S}_{\text{all}}[\textit{criteria}]$

16: **if** n_{ARSolns} is provided **then**

17: $\mathcal{S}_{\text{selected}} \leftarrow$ Select the first n_{ARSolns} solutions from $\mathcal{S}_{\text{selected}}$

18: **end if**

19: **Return:** $\mathcal{S}_{\text{selected}}$

Algorithm 4 CalculateRelativeFrequencies

 1: **Input:**

- $\mathcal{S}_{\text{leaf}}$: Solutions from a selected leaf node

 2: **Output:**

- Decision frequencies, $DecFreq$
- Relative Frequencies, $RelFreq$

 3: Initialize $LocFreq \leftarrow 0$ for all decisions

 4: **for** each solution $S_i \in \mathcal{S}_{\text{leaf}}$ **do**

 5: **for** each location j **do**

 6: **if** $S_i[j] = 1$ **then**

 7: $DecFreq[j] \leftarrow DecFreq[j] + 1$

 8: **end if**

 9: **end for**

 10: **end for**

 11: $n_{\text{solns}} \leftarrow |\mathcal{S}_{\text{leaf}}|$

 12: $RelFreq \leftarrow \frac{DecFreq}{n_{\text{solns}}}$

 13: **Return** $RelFreq$

The Algorithm 4, CalculateRelativeFrequencies function, computes the relative frequency ($RelFreq$) of each decision being active in a given subset of solutions ($\mathcal{S}_{\text{leaf}}$). For each decision j , the function calculates the number of times j is active ($S_i[j] = 1$) across all solutions $\mathcal{S}_{\text{leaf}}$, resulting in the decision frequency ($DecFreq$). The relative frequency for each decision is then obtained by dividing its frequency by the total number of solutions (n_{solns}) in $\mathcal{S}_{\text{leaf}}$. These frequencies are subsequently used to guide the generation of new solutions.

A move operator is a fundamental concept in heuristic optimization, defining a neighborhood function that guides transitions from one solution to another within the search space. By applying local changes to a solution, the move operator explores the space of candidate solutions to identify promising regions. In this algorithm, we integrate a data-driven approach into local search by using a regression tree to inform

the move operator and guide solution updates. The 5, ApplyRemoveAdd function, performs the Remove-Add operation to update the population. For each solution in the set of solutions selected for modification ($\mathcal{S}_{\text{remove_add}}$), the function identifies the most frequently active decision ($RemDec_k$) to remove based on the relative frequencies ($RelFreq$). A new decision ($AddDec_k$) is then determined using a greedy algorithm, and the solution is updated accordingly. The modified solution (S_{new}) is added to the population if it is not already present. The updated population (P_t) and objective values ($O(P_t)$) are returned as the output.

Algorithm 5 ApplyRemoveAdd

```

1: Input:
2:  $\mathcal{S}_{\text{remove\_add}}$ : Solution set for Remove-Add
3:  $RelFreq$ : Relative Frequencies of Decisions calculated based on the worst leaf node
4:  $P_t$ : Current population
5:  $O(P_t)$ : Objective values of the current population
6: Output:
7: Updated population  $P_t$  and objective values  $O(P_t)$ 
8: for  $k \leftarrow 0$  to  $|\mathcal{S}_{\text{remove\_add}}| - 1$  do
9:    $RemDec_k \leftarrow \arg \max_{j \in \mathcal{S}_{\text{remove\_add}}} RelFreq[j]$ 
10:   $AddDec_k \leftarrow$  Determine using Greedy Algorithm
11:   $S_{\text{new}}[RemDec_k] \leftarrow 0$ 
12:   $S_{\text{new}}[AddDec_k] \leftarrow 1$ 
13:  if  $S_{\text{new}} \notin P_t$  then
14:     $P_t \leftarrow P_t \cup \{S_{\text{new}}\}$ 
15:     $O(P_t) \leftarrow O(P_t) \cup O(S_{\text{new}})$ 
16:  end if
17: end for
18: Return: Updated  $P_t$  and  $O(P_t)$ 

```

Our trials with the Remove-Add algorithm, which exclusively utilizes regression trees during the Remove-Add phase, indicate a necessity for diversification within the population after several iterations. Consequently, we introduce population updates through our proposed Population Management algorithm (Algorithm 6), as illustrated in the flowchart presented in Figure 3.9. These updates are triggered after a specified number of successive iterations without any improvement in the best objective value.

Initially, we do not remove any solutions; instead, we apply the remove-add operation to enhance the objective value. If the objective value improves, we incorporate the new, enhanced solution into the population without discarding the existing solution, recognizing its value. Subsequently, when there is no improvement in the best objective value over a predefined number of successive iterations, we update the population using the Algorithm 6, PopulationUpdate function.

Algorithm 6 PopulationUpdate

1: **Input:**

- P_t : Current population
- $O(P_t)$: Objective values of the current population
- $n_{\text{PopAddRem}}$: Number of solutions to remove and add

2: **Output:** Updated population P_t and objective values $O(P_t)$

3: Fit a regression tree $\mathcal{T}$ on P_t with $O(P_t)$

4: $P_t, O(P_t) \leftarrow \text{RemoveWorstSolutions}(n_{\text{PopAddRem}}, P_t, O(P_t))$

5: $\mathcal{S}_{\text{pop_update}} \leftarrow \text{GetSolutionsFromLeafNode}(\mathcal{T}, O(P_t), \text{criteria} = \text{"best"})$

6: $\text{RelFreq} \leftarrow \text{CalculateRelativeFrequencies}(\mathcal{S}_{\text{pop_update}})$

7: $S_{\text{new}}, O(S_{\text{new}}) \leftarrow \text{NewSolutionsGeneration}(n_{\text{PopAddRem}}, \text{RelFreq})$

8: $P_t \leftarrow P_t \cup \{S_{\text{new}}\}$

9: $O(P_t) \leftarrow O(P_t) \cup O(S_{\text{new}})$

10: **Return:** Updated P_t and $O(P_t)$

The Algorithm 6, the PopulationUpdate function, plays a critical role in re-inspecting the population by removing poor solutions and introducing new promising ones. The process begins by fitting a regression tree ($\mathcal{T}$) to the current population (P_t) and its objective values ($O(P_t)$). The regression tree captures the relationship between solutions and their objective values, enabling a data-driven approach to selecting candidate solutions for updates. Once the tree is fitted, the worst-performing solutions ($n_{\text{PopAddRem}}$) are removed from the population based on their objective values. This removal creates space in the population for new solutions to be added. The solutions of the best leaf node ($\mathcal{S}_{\text{pop_update}}$) of the regression tree are then selected, representing high-quality solutions that guide the generation of new candidates. Relative frequencies (RelFreq) of features are calculated from the selected solutions ($\mathcal{S}_{\text{pop_update}}$). These frequencies provide probabilistic insights on promising solution features and serve as the basis for generating $n_{\text{PopAddRem}}$ new solutions. The newly generated solutions (S_{new}) and their corresponding objective values ($O(S_{\text{new}})$) are added to the population, resulting in an updated population (P_t) and objective values ($O(P_t)$). This iterative process ensures that the population evolves toward better regions of the search space while maintaining diversity. By balancing exploration and exploitation, this approach effectively navigates the search space, enhancing the likelihood of finding globally optimal solutions.

Algorithm 7 RemoveWorstSolutions

1: **Input:**

- Population of solutions $P = \{S_1, S_2, \dots, S_{n_{\text{Pop}}}\}$ with objective values $O(P) = \{O(S_1), O(S_2), \dots, O(S_{n_{\text{Pop}}})\}$
- Number of solutions to remove, $n_{\text{PopAddRem}}$

2: **Procedure:**

3: Sort P in descending order according to $O(P)$, such that $O(S_i) \geq O(S_{i+1}) \forall i$

4: $P_{\text{updated}} \leftarrow P \setminus \{S_1, S_2, \dots, S_{n_{\text{PopAddRem}}}\}$

5: **Output:**

6: Updated population P_{updated} and Updated objective values $O(P_{\text{updated}})$

The Algorithm 7, the RemoveWorstSolutions function, eliminates the $n_{\text{PopAddRem}}$ worst solutions of the population. The solutions are sorted in descending order according to their objective values ($O(P)$), and the top $n_{\text{PopAddRem}}$ solutions are removed. The function returns the updated population (P_{updated}) and the corresponding objective values ($O(P_{\text{updated}})$).

Algorithm 8 NewSolutionsGeneration

```

1: Input:
    • Number of solutions to generate,  $n_{\text{PopAddRem}}$ 
    • RelFreq: Relative Frequencies for each decision
2: Output:
    • Newly generated solution set  $\mathcal{S}_{\text{new}}$ 
3: Initialize  $\mathcal{S}_{\text{new}} \leftarrow \{\}$ 
4: for  $k \leftarrow 1$  to  $n_{\text{PopAddRem}}$  do
5:   Generate a new solution  $S_{\text{new}}$  based on RelFreq
6:    $\mathcal{S}_{\text{new}} \leftarrow \mathcal{S}_{\text{new}} \cup \{S_{\text{new}}\}$ 
7: end for
8: Return  $\mathcal{S}_{\text{new}}, O(\mathcal{S}_{\text{new}})$ 

```

The Algorithm 8, NewSolutionsGeneration function, generates new solutions to diversify the population. Using the relative frequencies (*RelFreq*) calculated from selected solutions, the function probabilistically activates decisions in each new solution (S_{new}). A total of $n_{\text{PopAddRem}}$ solutions are generated, and their corresponding objective values ($O(S_{\text{new}})$) are computed. The function outputs the set of new solutions ($\mathcal{S}_{\text{new}}$) and their objective values, which are added to the population. The new solutions are then added to the population, resulting in an updated P_t and $O(P_t)$. This iterative approach ensures that the population evolves dynamically, balancing intensification (by focusing on promising regions) and diversification (by introducing novel solutions). By integrating the PopulationUpdate function into the algorithm, the search process remains adaptive and helps avoid premature convergence.

3.1.7. Conceptual Comparison: Naive Approach vs. Regression Tree Integration

The *Base* method refers to the naive implementation of the algorithm, which operates without utilizing the regression tree method. In this naive approach, the algorithm processes the entire solution space without any guidance from predictive models. The detailed framework for both the *RegTree* and *Base* methods is presented in Figure 3.13. This figure illustrates the key differences between the two methods, highlighting how the *RegTree* method enhances the algorithm with data-driven decision-making.

In the Selection Phase, the naive *Base* method selects the best solutions from the current population, referred to as *BestN*. In contrast, the *RegTree* method selects solutions from the best leaf node of the generated regression tree, denoted as $\mathcal{S}_{\text{best}}$. The selection process for both methods is illustrated in Figure 3.10, where the *Base* method extracts top-performing solutions directly, while the *RegTree* method uses the regression tree structure to identify the best-performing subset.

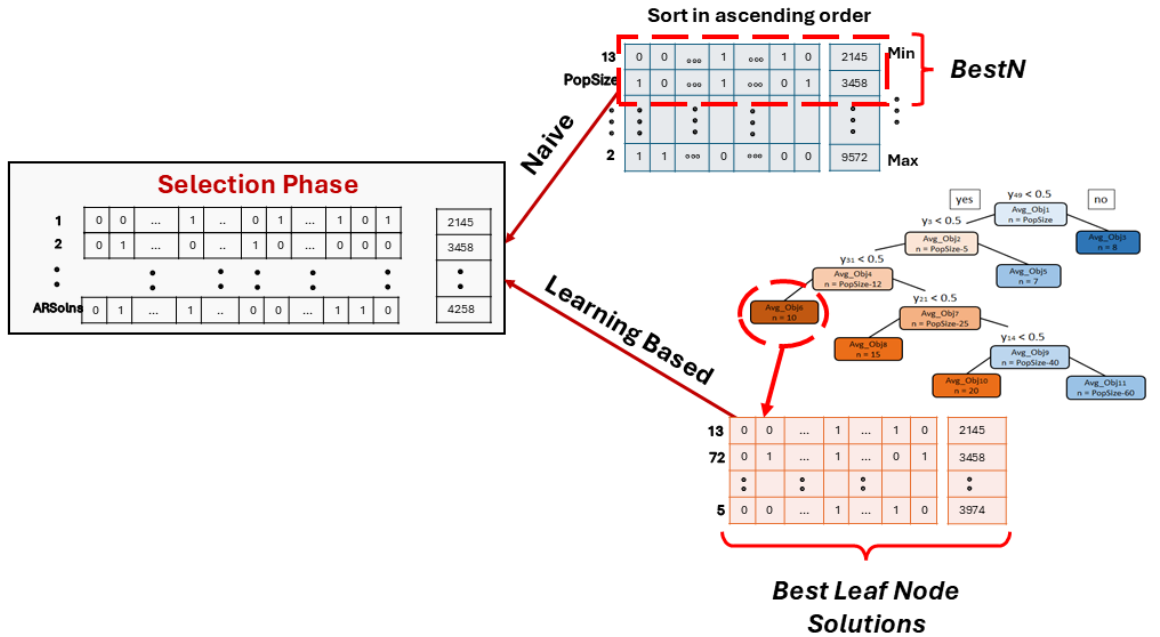


Figure 3.10. Selection Phase of the Algorithm.

In the Improvement Phase, the *Base* method identifies decisions to be removed using the worst solutions from the population, referred to as *WorstN*. The *RegTree* method, however, utilizes the worst leaf node solutions from the regression tree, denoted as $\mathcal{S}_{\text{worst}}$, to determine the decisions to remove. This process is illustrated in Figure 3.11, where the *Base* method eliminates the lowest-ranked solutions, while the *RegTree* method removes solutions from the leaf node having the worst average object value.

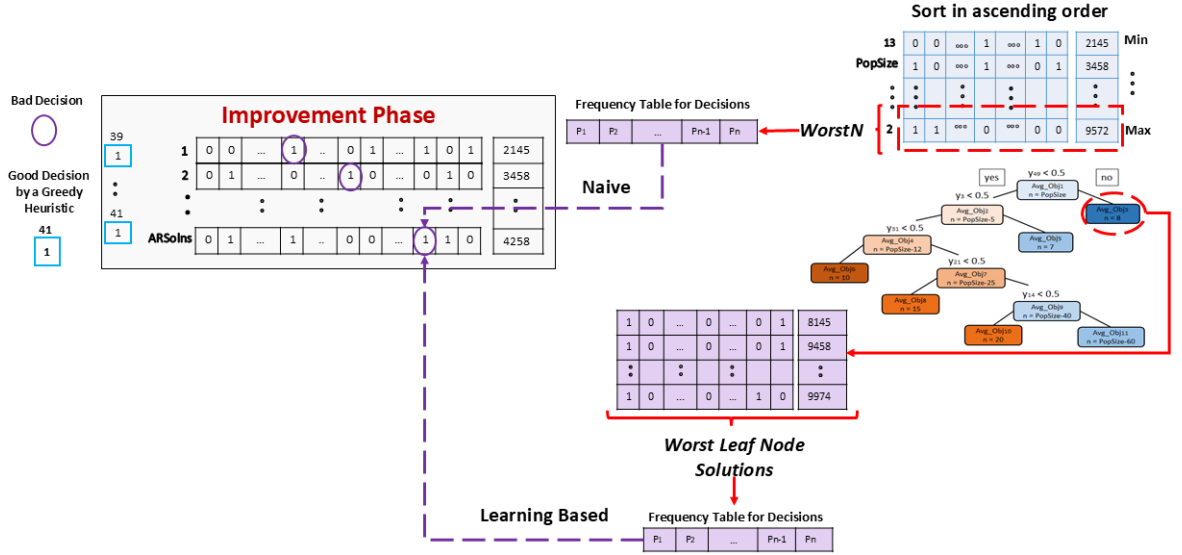


Figure 3.11. Improvement Phase of the Algorithm.

If no improvement is observed after a predetermined number of successive iterations, the algorithm transitions to the Population Management Phase. In this phase, both methods remove underperforming solutions from the population and introduce new solutions. The *Base* method generates new solutions using frequency information derived from the best solutions in the population. Conversely, the *RegTree* method generates new solutions using frequency information derived from the best leaf node solutions of the updated regression tree. This process is visualized in Figure 3.12, where both methods aim to enhance population diversity by replacing low-quality solutions with newly generated high-quality ones.

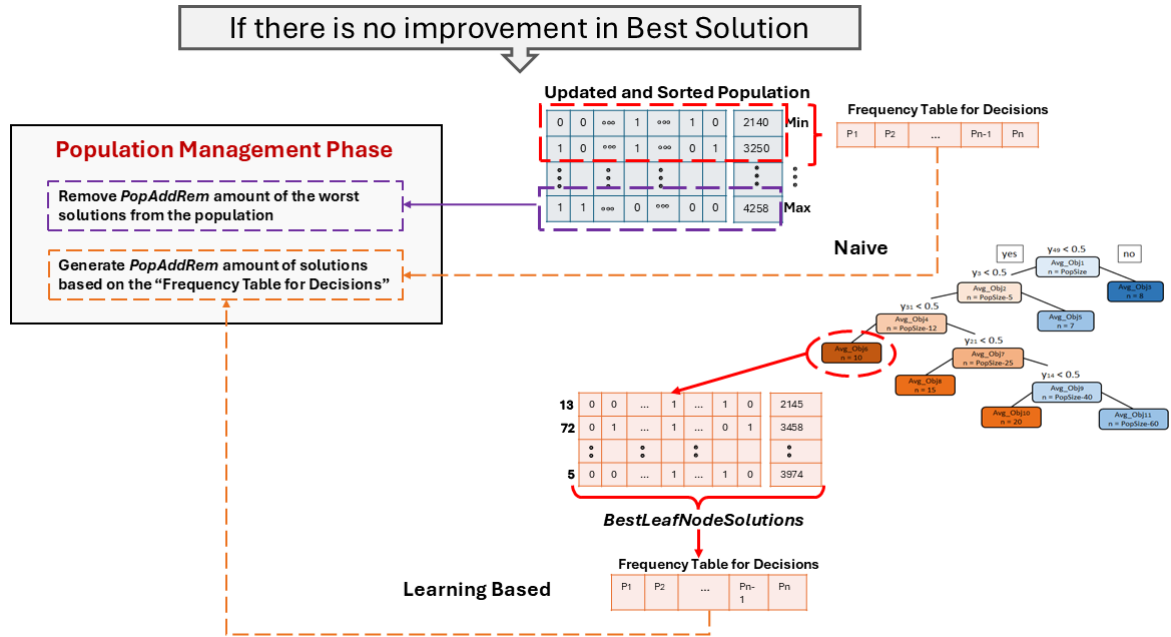


Figure 3.12. Population Management Phase of the Algorithm.

This comparison underscores how the *RegTree* method introduces a structured, data-driven approach to guide the search process, offering a more targeted exploration of the solution space compared to the naive *Base* method.

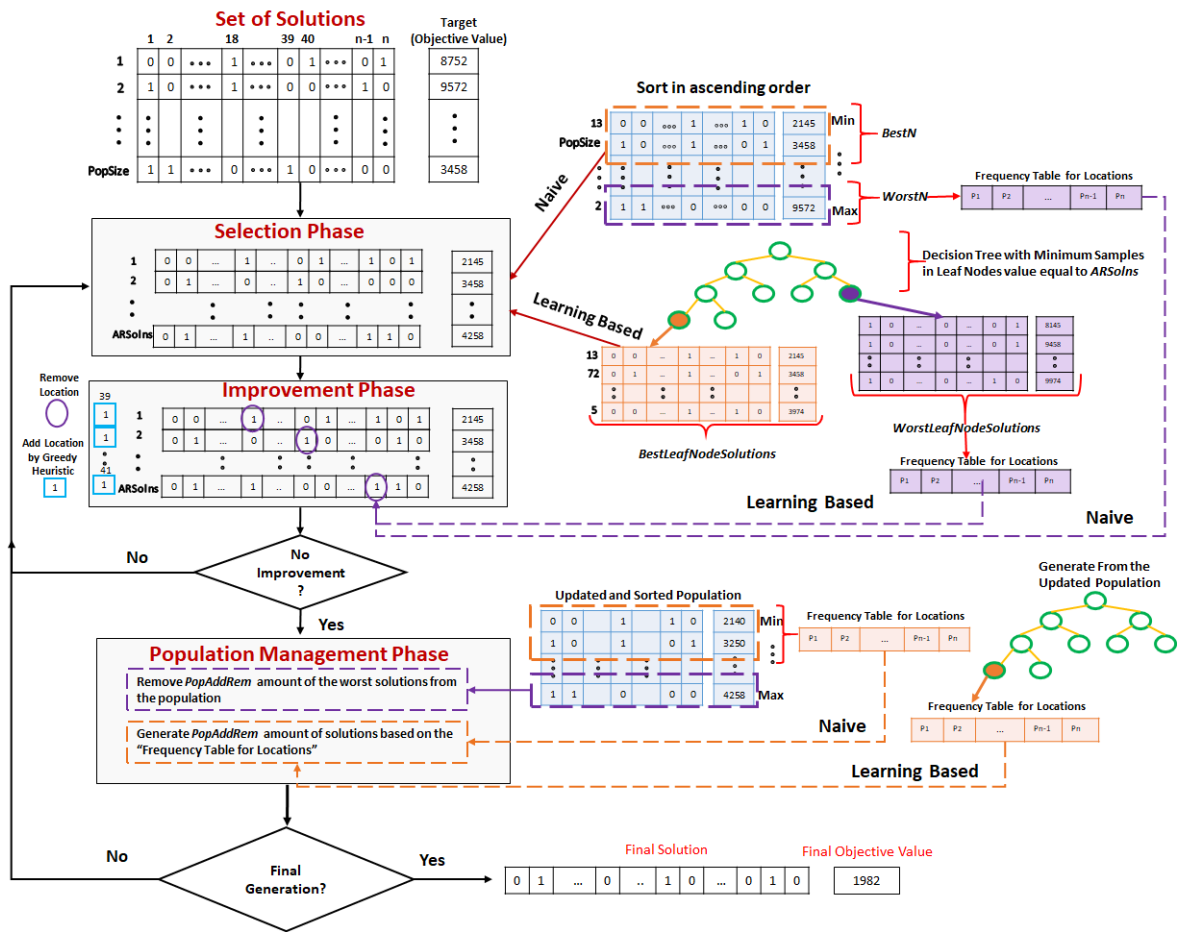


Figure 3.13. Whole Picture of the Algorithm.

3.2. Illustrative Examples

To clearly understand the applications and implications of the methodologies discussed, we will examine two illustrative examples: the p -median problem and the Quadratic Assignment Problem (QAP).

3.2.1. p -median Problem

Facility Location Problems (FLP) aim to determine optimal facility locations to ensure that all customer points are served by facilities chosen among potential locations in the most cost-effective manner. These problems can be classified as capacitated or uncapacitated, depending on whether the facilities have capacity constraints. Capacitated FLP requires modeling both demand satisfaction and capacity constraints, as customers must have specific demands. In contrast, uncapacitated problems do not impose capacity limits on facilities. One of the simplest and most well-known versions of FLP is the p -median problem, first introduced by [54]. This problem is a specific type of discrete location model that seeks to locate p facilities to minimize the total distance between demand nodes and their assigned facilities. In this study, we focus on the uncapacitated p -median problem. Additionally, in our model, there is no explicit customer demand; instead, facility opening decisions are made based on the calculated distances between locations. The mathematical model for the p -median problem is presented in this section.

Sets:

I : the set of candidate locations

J : the set of customers

Parameters:

c_{ij} : unit cost of satisfying customer j from facility i

p : the number of facilities to locate

Decision Variables:

y_i : 1, if a facility is located at candidate site i ; 0, otherwise

x_{ij} : the fraction of the demand of customer j that is supplied from facility i

3.2.1.1. Mathematical Model. With this notation, we can formulate the p -median problem as follows:

$$\text{minimize } z = \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \quad (3.2)$$

The constraints of the problem are as follows:

$$\sum_{i \in I} x_{ij} = 1 \quad j \in J \quad (3.3)$$

$$\sum_{i \in I} y_i = p \quad (3.4)$$

$$x_{ij} \leq y_i \quad i \in I, j \in J \quad (3.5)$$

$$y_i \in \{0, 1\}, \quad i \in I \quad (3.6)$$

$$x_{ij} \geq 0 \quad i \in I, j \in J \quad (3.7)$$

The objective function (3.1) minimizes the total distance. Constraints (3.2) mean that all demands of customer j must be satisfied. Constraints (3.3) require exactly p facilities to be located. Constraints (3.4) state that customers can only be assigned to open facilities. Constraints (3.5) stipulate that the location variables must be integer and binary. Finally, constraints (3.6) state that the assignment variables must be non-negative.

3.2.2. Quadratic Assignment Problem

The quadratic assignment problem (QAP) was introduced by [55]. It is a model for many practical problems like backboard wiring [56], campus and hospital layout [57]

and [58], typewriter keyboard design, and scheduling. The QAP can be described as the problem of assigning a set of facilities to a set of locations with given distances between the locations and flows between the facilities. The goal is to find a minimal assignment in terms of the sum of the products of the flows and distances.

More formally, we are given a set $K = 1, \dots, n$ of n facilities and a set $I = 1, \dots, n$ of n locations, two $n \times n$ matrices $D = [d_{ij}]$ and $F = [f_{kl}]$, where d_{ij} is the distance between locations i and j and f_{kl} is the flow between facilities k and l , then the NP-Hard QAP can be formulated as follows:

$$\text{minimize } z = \sum_{i \in I} \sum_{j \in I} \sum_{k \in K} \sum_{l \in K} d_{ij} f_{kl} x_{ik} x_{jl} \quad (3.8)$$

The constraints of the problem are as follows:

$$\sum_{i \in I} x_{ik} = 1 \quad k \in K \quad (3.9)$$

$$\sum_{k \in K} x_{ik} = 1 \quad i \in I \quad (3.10)$$

$$x_{ik} \in \{0, 1\} \quad i \in I, k \in K \quad (3.11)$$

The QAP has numerous applications across different fields, including:

- Facility Layout: Arranging departments or machines within a manufacturing plant to minimize the material handling cost.
- Keyboard Design: Optimizing the arrangement of keys on a keyboard to reduce the finger travel distance for typists.
- Electronic Circuit Design: Assigning components to positions on a circuit board to minimize the total wire length or signal delay.
- Hospital Layout: Allocating different departments within a hospital to minimize the transportation cost of patients, staff, and materials.
- Data Analysis: QAP can be used for the seriation problem, which involves re-ordering rows and columns of a matrix to reveal underlying structures.

3.3. Computational Results for p -median Problem

The algorithm was implemented using Python 3.9. Development and experimentation were conducted in both Jupyter Notebook and PyCharm environments. The experiments' hardware includes a computer equipped with an Intel® Core™ i7-4510U CPU at 2.00GHz and 8GB of RAM. The operating system installed on this machine is Microsoft Windows 10.

We conducted experiments using our *RegTree* on the PMed1-40 instances, which are p -median problems from the OR-Library. The initial population size was set to 100, as determined by the sensitivity analysis for population sizes (Section 3.3.1), which examined configurations of 50, 100, 200, and 300. The results of this analysis demonstrated that a population size of 100 achieved the best balance between solution quality and computational efficiency. Throughout the iterations, the population size dynamically increased as the objective value of a solution improved, with newly generated solutions being added to the population accordingly.

To evaluate the performance of the regression tree, we use the Mean Squared Error (MSE) as an error metric, which is a widely used measure of the average squared difference between predicted and actual values. It provides insight into the model's accuracy by indicating how well the regression tree captures the underlying patterns in the data. A lower MSE suggests better model performance, as it reflects smaller prediction errors.

The mathematical formulation for the MSE is as follows:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (3.12)$$

where n is the total number of data points, y_i is the actual value of the i -th data point, and $\hat{y}_i$ is the predicted value for the i -th data point.

Additionally, the minimum number of samples in a leaf node is set to 5% of the initial population size, corresponding to the size of the first N solutions selected for

the *Base* algorithm. Thus, the value of *ARSolns* is 5% of the initial population size in these experiments.

To evaluate the performance of the proposed approach, the $GAP(\%)$ metric is used to quantify the relative difference between the obtained objective value and the best-known or optimal objective value. A lower $GAP(\%)$ indicates better performance, with a value of 0% signifying an optimal solution. The $GAP(\%)$ is calculated as

$$GAP(\%) = \left(\frac{Obj_{sol} - Obj_{opt}}{Obj_{opt}} \right) \times 100 \quad (3.13)$$

where Obj_{sol} represents the objective value obtained by the algorithm, and Obj_{opt} represents the best-known or optimal objective value.

3.3.1. Sensitivity Analysis for Population Size

Population size plays a critical role in determining the performance of optimization algorithms, as it directly influences solution quality, convergence speed, and computational efficiency. To investigate the sensitivity of the algorithm to changes in population size, we conducted a detailed analysis by comparing groups with initial population sizes of 50, 100, 200, and 300. This comparison was based on key performance metrics such as GAP (solution quality) and Runtime (execution time).

To assess whether the observed differences across groups were statistically significant, we performed an Analysis of Variance (ANOVA) test. Following a significant result in the ANOVA test, we conducted Tukey's post hoc analysis to identify specific pairwise differences between population size groups. Table 3.5 summarizes the results of the Tukey analysis, presenting mean differences, adjusted p-values, confidence intervals, and whether the null hypothesis of no difference was rejected. For example, the comparison between population sizes 50 and 300 yielded a mean difference of 0.0086 with a p-value of 0.010, and the null hypothesis was rejected, indicating a statistically significant difference. Similarly, the comparison between sizes 100 and 300 showed a

mean difference of 0.0139 with a p-value of 0.000, and the null hypothesis was also rejected. In contrast, comparisons such as 50 vs. 100 and 50 vs. 200 had p-values of 0.2212 and 0.9391, respectively, and the null hypothesis was not rejected, indicating no significant differences. These findings provide valuable insights into the impact of population size on the algorithm's performance and guide the identification of an optimal configuration for balancing solution quality and computational cost.

Table 3.5. Pairwise comparisons of population sizes using Tukey's post hoc analysis.

Group1	Group2	Mean Difference	p-adj	Lower CI	Upper CI	Reject
50	100	-0.0053	0.2212	-0.0124	0.0018	False
50	200	0.0016	0.9391	-0.0055	0.0087	False
50	300	0.0086	0.0100	0.0015	0.0157	True
100	200	0.0069	0.0615	-0.0002	0.0140	False
100	300	0.0139	0.0000	0.0068	0.0210	True
200	300	0.0070	0.0536	-0.0001	0.0141	False

Population size plays a critical role in the performance of optimization algorithms, influencing both solution quality and computational efficiency. To assess the impact of population size on the algorithm, we analyzed key metrics, including Mean GAP (%), Std GAP (%), Mean Runtime (minutes), and Std Runtime (minutes), for four population sizes: 50, 100, 200, and 300. Table 3.6 presents a summary of these statistics, while Figures 3.14, 3.15, and 3.16 provide visual representations of the results.

Table 3.6. Population size statistics: GAP(%) and Runtime.

Population Size	Mean GAP(%)	Std GAP(%)	Mean Runtime	Std Runtime
50	2.24	3.24	0.749	0.616
100	1.71	1.85	0.896	0.777
200	2.40	2.58	1.059	0.746
300	3.10	3.14	1.547	1.015

The analysis revealed that the lowest Mean GAP (%) was achieved with a population size of 100, at 1.71%, compared to 2.24% for size 50, 2.40% for size 200, and 3.10% for size 300. This indicates that a population size of 100 provides the best solution quality among the tested configurations. Additionally, the Std GAP (%), which reflects variability in solution quality, was also the lowest at size 100 (1.85%), compared to 3.24% for size 50, 2.58% for size 200, and 3.14% for size 300. The combination of low Mean GAP (%) and Std GAP (%) for size 100 suggests that this population size yields both high-quality and consistent solutions.

In terms of solution time, the analysis shows a steady increase as population size grows. At size 50, the Mean Runtime was 0.75 minutes, while at size 100, it increased moderately to 0.90 minutes. However, further increases in population size led to significant runtime growth, with 1.06 minutes for size 200 and 1.55 minutes for size 300. This trend underscores the computational cost associated with larger populations. The Std Runtime followed a similar pattern, with size 100 showing moderate variability (0.78 minutes) compared to the higher variability observed at size 300 (1.02 minutes). These results highlight the computational efficiency of size 100, achieving reasonable runtimes without sacrificing solution quality.

Figure 3.14 illustrates the Mean GAP (%) for each population size, with error bars representing Std GAP (%). The chart confirms that size 100 achieves the lowest mean and variability, while sizes 200 and 300 show increasing GAP values and variability. Figure 3.15 visualizes the Mean Runtime (minutes), where size 100 is evident as a computationally efficient choice with low variability. Figure 3.16 provides a trade-off analysis between GAP (%) and runtime, clearly showing that size 100 achieves the best balance, minimizing both GAP and runtime compared to larger sizes.

Based on these findings, a population size of 100 was chosen for subsequent analyses. This configuration strikes an optimal balance between solution quality and computational efficiency, achieving the lowest Mean GAP (%) with minimal variability while maintaining manageable runtime. Larger population sizes, such as 200 and 300, offer

no substantial improvement in solution quality and incur significantly higher computational costs and variability. Thus, size 100 represents a practical and efficient choice for the algorithm.

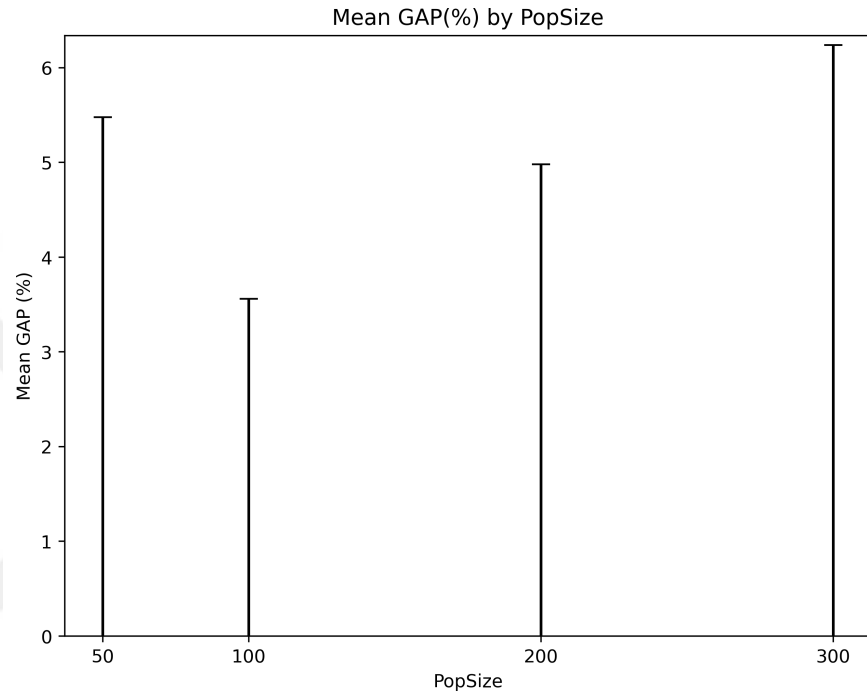


Figure 3.14. Mean GAP (%) by Population Size.

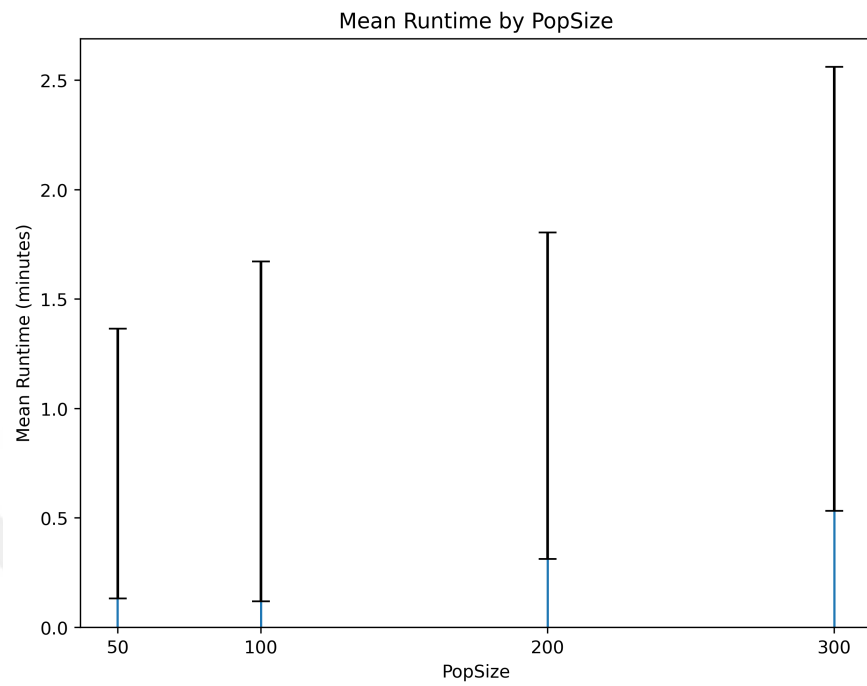


Figure 3.15. Mean Runtime by Population Size.

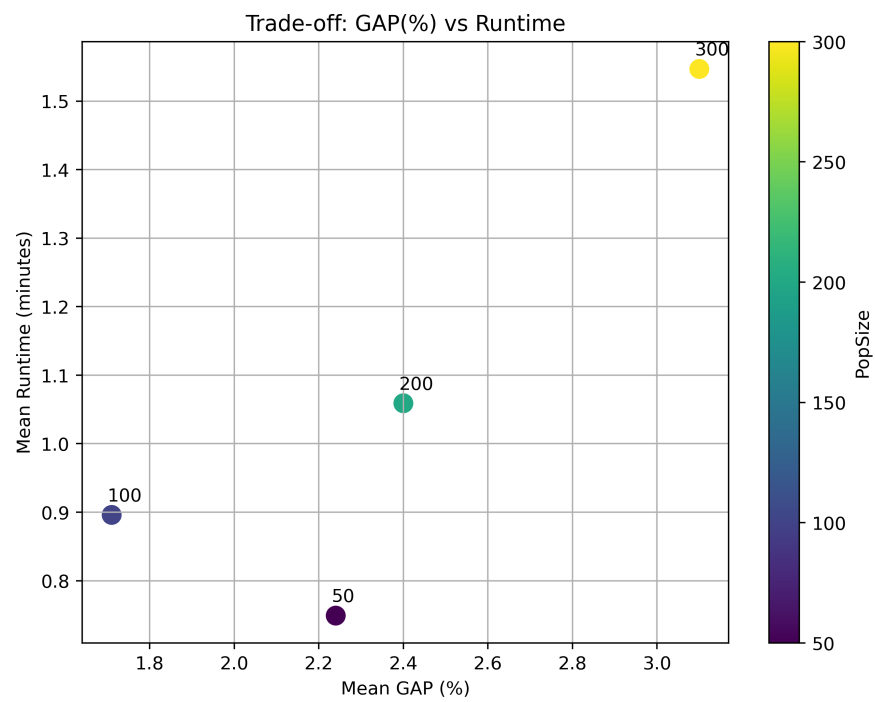


Figure 3.16. Trade-Off: GAP (%) vs. Runtime for Different Population Sizes.

3.3.2. RegTree Design Requirements for p-median

The p -median problem employs a permutation-based solution representation. However, for the proposed framework, binary features are required. To address this, the candidate locations where a facility is opened are represented as binary features, with a value of 1 if a facility is opened at a location and 0 otherwise. Consequently, there are n_{loc} features in total. The objective value of the solution serves as the target variable. Once y_i values are fixed (i.e., facilities are determined), the assignment x_{ij} can be computed by selecting the facility i that offers the minimum demand-weighted cost for each customer j . The demand-weighted cost matrix helps efficiently determine the optimal assignments given the opened facilities.

A key problem-specific design consideration is the incorporation of a move operator during the local search stage. To explore the neighborhood space effectively, a single remove-add move operator is applied. For a given solution selected for improvement, the remove location is identified as described in Section 3.1.4 and Algorithm 5. After identifying the location to be removed, a new location is added using a greedy heuristic.

During the population update stage, new solutions are generated following Algorithm 8. To maintain the population size, the worst-performing solutions (based on their objective values) are removed in equal numbers to the newly generated solutions. This approach ensures continuous refinement of the solution set while balancing exploration and exploitation in the search process.

3.3.3. Diversification Analysis with Three Techniques

Firstly, we aim to investigate how the utilization of regression trees in a population-based heuristic varies with problem size. To this end, we solve the PMed1-40 problems using our algorithm, distinguishing the solutions applied to remove-add operations. If the frequencies of opened locations in the best leaf node solutions remain consistent

across two successive iterations, we then select solutions from either the second best leaf node (*SecBest*) or the leaf node exhibiting the maximum variance in objective values (*MaxVar*). In this context, our proposed *RegTree* algorithm, referred to as *Best*, is differentiated from *SecBest* and *MaxVar*.

The *MaxVar* technique enhances population diversity by applying remove-add operations to solutions with the greatest variety in objective values. For this experiment, we set the algorithm parameters *NIT* and *NPM* to 7 and 50, respectively. For all instances, the initial population size (*PopSize*) and the population remove-add size (*PopAddRem*) are determined as 100 and 2, respectively. The dots are coded with color that if p/n_{loc} ratio is less than and equal to 0.1, it is orange; if larger than 0.1, it is dark green.

As the ratio of p to n_{loc} decreases, indicating an increase in sparsity, the need for diversification becomes more critical. Consequently, the *MaxVar* technique yields better results, as anticipated. The comparison of the three techniques is illustrated in Figures 3.17, 3.18, and 3.19. These figures show that the *MaxVar* technique outperforms the other two techniques as the p/n_{loc} ratio decreases and sparsity increases. This highlights the necessity of thoroughly exploring sparse problems. The *MaxVar* technique facilitates exploration by selecting solutions for the remove-add operation of the leaf node with the highest variance in objective values.

However, in general, the *Best* technique provides superior results. The average GAP(%) values for all problem instances are 1.63, 1.90, and 2.35 for *Best*, *SecBest*, and *MaxVar*, respectively.

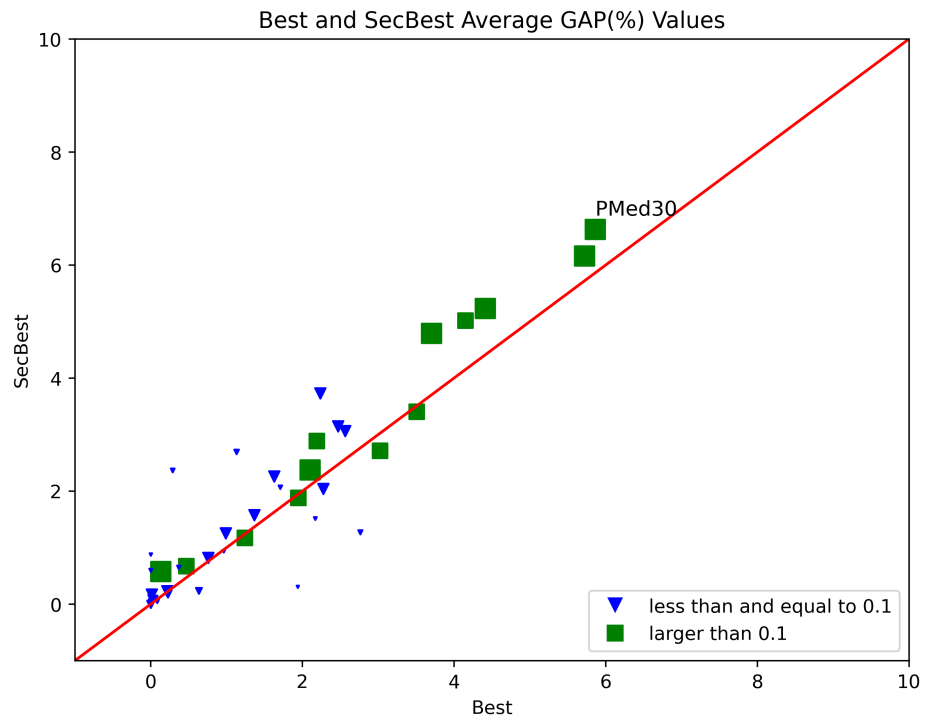


Figure 3.17. Average GAP(%) - Best and SecBest Techniques.

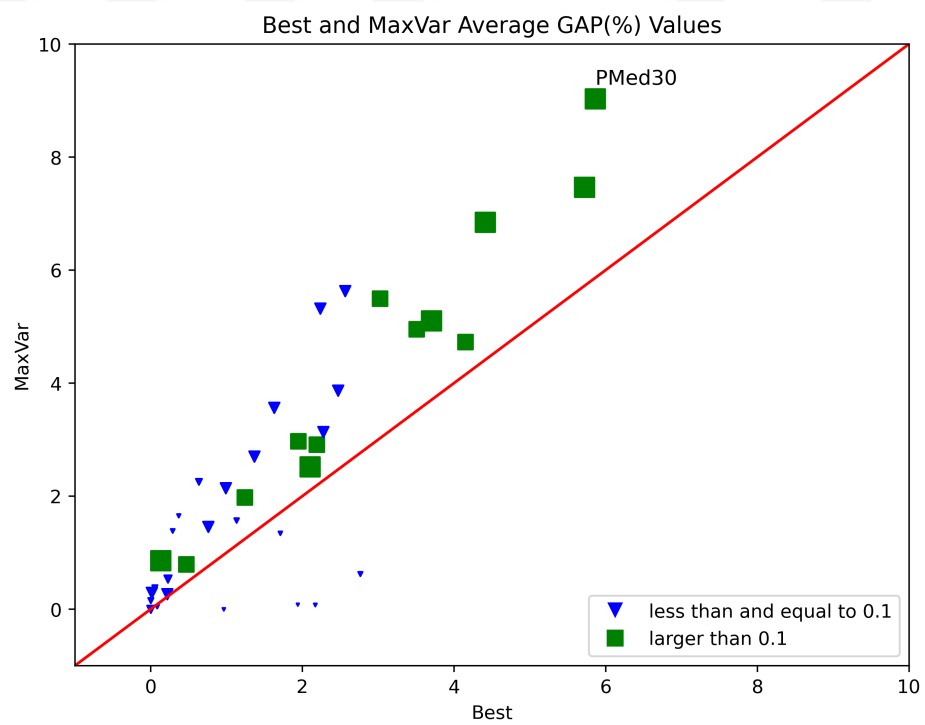


Figure 3.18. Average GAP(%) - Best and MaxVar Techniques.

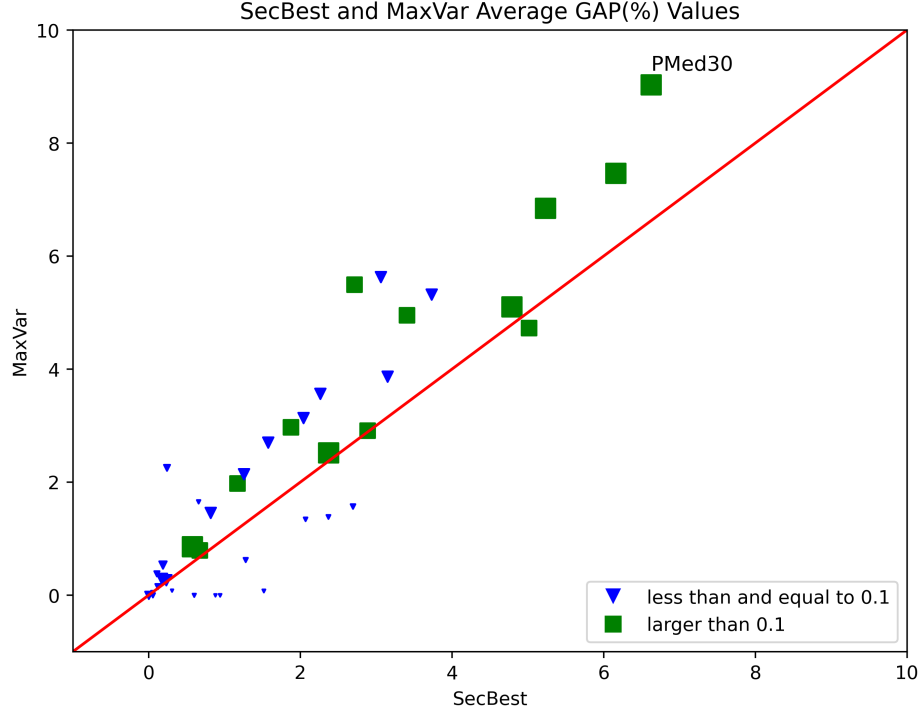


Figure 3.19. Average GAP(%) - SecBest and MaxVar Techniques.

3.3.4. Comparison of RegTree and Base methods with Equal Number of Iterations

To evaluate the performance of the proposed *RegTree*, we compare it with *Base* to highlight its advantages. The experiments are carried out using specific parameter settings: the no-improvement-in-successive-iterations (*NIT*) parameter is set to 7, and the number-of-times-population-management (*NPM*) parameter is set to 50. In addition, two different configurations of the *PopAddRem* parameter are tested, 2 and 4. For each problem instance, five independent replications are performed to ensure robust evaluation. This comparative study aims to analyze how the integration of regression trees into a population-based heuristic impacts performance, particularly with respect to variations in problem size.

The color-coded and size-scaled markers in Figures 3.20 and 3.21 represent the p/n_{loc} ratio. Triangular markers (dark purple) indicate problem instances with $p/n_{loc} \leq 0.1$, while square markers (light purple) indicate $p/n_{loc} > 0.1$. Furthermore, the size of

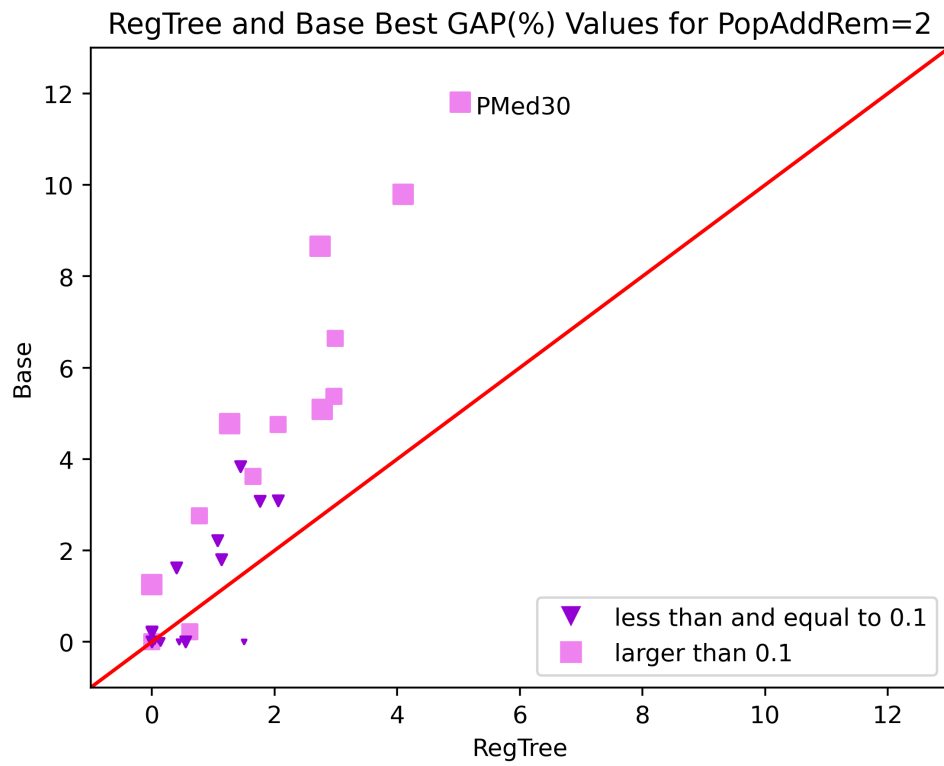
the markers is proportional to the p/n_{loc} ratio, with larger shapes representing higher values of p/n_{loc} . These figures compare the performance of *RegTree* and *Base* for the best and average GAP(%) values in the scenario with $\text{PopAddRem}=2$.

In terms of Best GAP(%) values (Figure 3.20), the *RegTree* algorithm generally performs better than the *Base* method across most instances, especially as the p/n_{loc} ratio increases, as shown by the larger square markers. This indicates that *RegTree* performs particularly well in scenarios with lower sparsity, where the problems are more challenging. In contrast, some instances of sparse problems (represented by smaller triangular markers) show that *Base* performs better. The summarized results for Best GAP(%) values can be found in Table A.2. Comparison of average GAP (%) values (Figure 3.21) reveals similar trends. For higher p/n_{loc} ratios (indicated by larger squares), *RegTree* maintains an advantage, while for sparser problems (indicated by smaller triangles), *Base* performs better in certain cases. The corresponding average GAP(%) results are also detailed in Table A.2.

Figures 3.22 and 3.23 provide insights into the solution time and the number of iterations, respectively. In these graphs, yellow triangles represent instances where $p/n_{\text{loc}} \leq 0.1$, while green squares correspond to $p/n_{\text{loc}} > 0.1$. The size of the markers reflects the p/n_{loc} ratio, with larger markers indicating higher values. As shown in Figure 3.22, the *Base* algorithm consistently requires less solution time than the *RegTree* algorithm for all instances. This is attributed to the additional computational cost of constructing regression trees in *RegTree*. Furthermore, Figure 3.23 demonstrates that the number of iterations required for *Base* to converge is consistently lower than that for *RegTree*. This indicates that *Base* converges to a local optimum more quickly. The solution time and the number of iterations for both algorithms are summarized in Table A.3.

The *RegTree* method, however, introduces diversification into the search process by incorporating the variability between the best and worst solutions, which contributes to its superior performance in more challenging scenarios with higher p/n_{loc} ratios.

This approach helps escape local optima by leveraging diverse regions of the solution space, ultimately enhancing the algorithm's robustness. Furthermore, the regression tree facilitates a learning-based mechanism to guide the search adaptively, allowing the algorithm to dynamically balance exploration and exploitation. It should be noted that the solution times, and number of iterations in Figures 3.22 and 3.23 are averages across multiple replications.



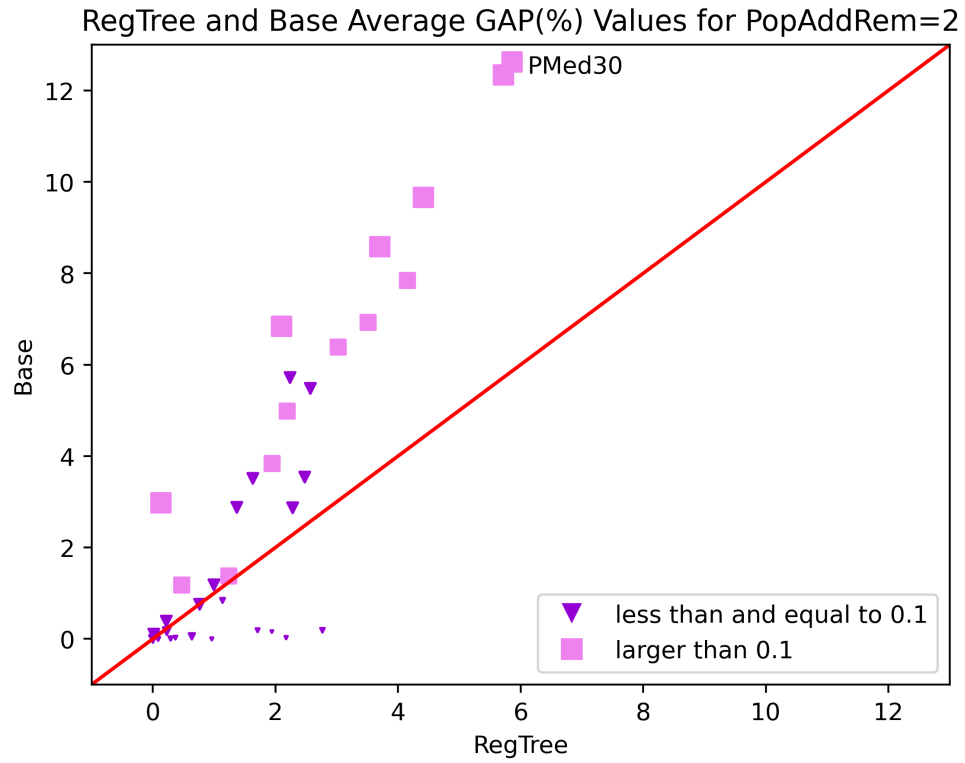


Figure 3.21. Average GAP(%) - RegTree and Base methods, $PopAddRem=2$.

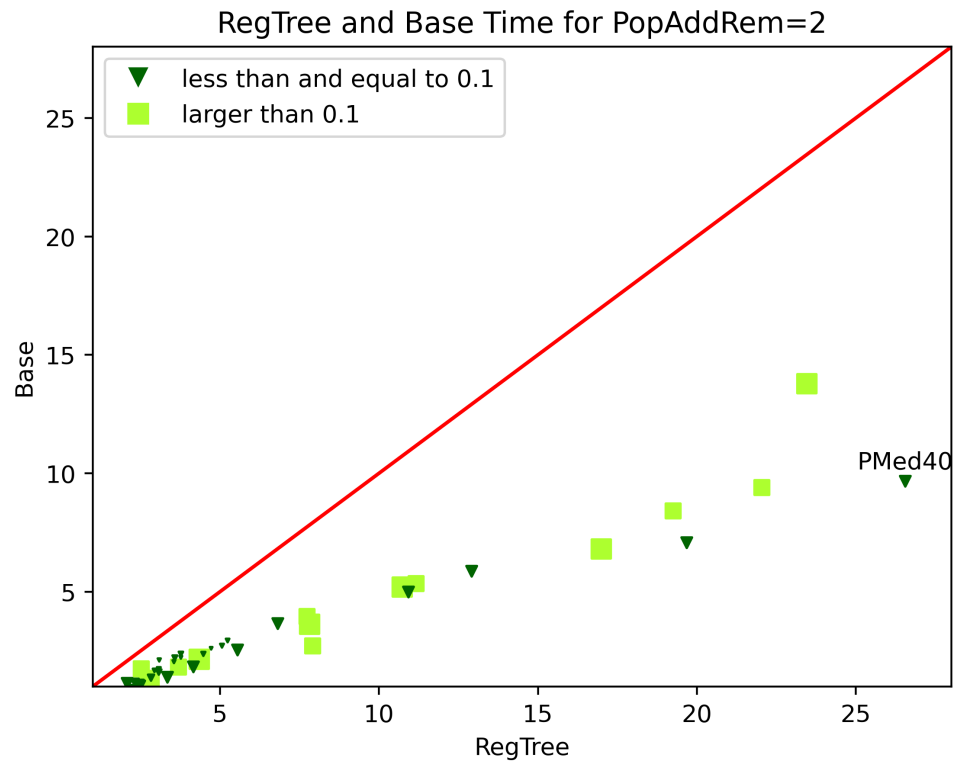


Figure 3.22. Solution Time - RegTree and Base methods, $PopAddRem=2$.

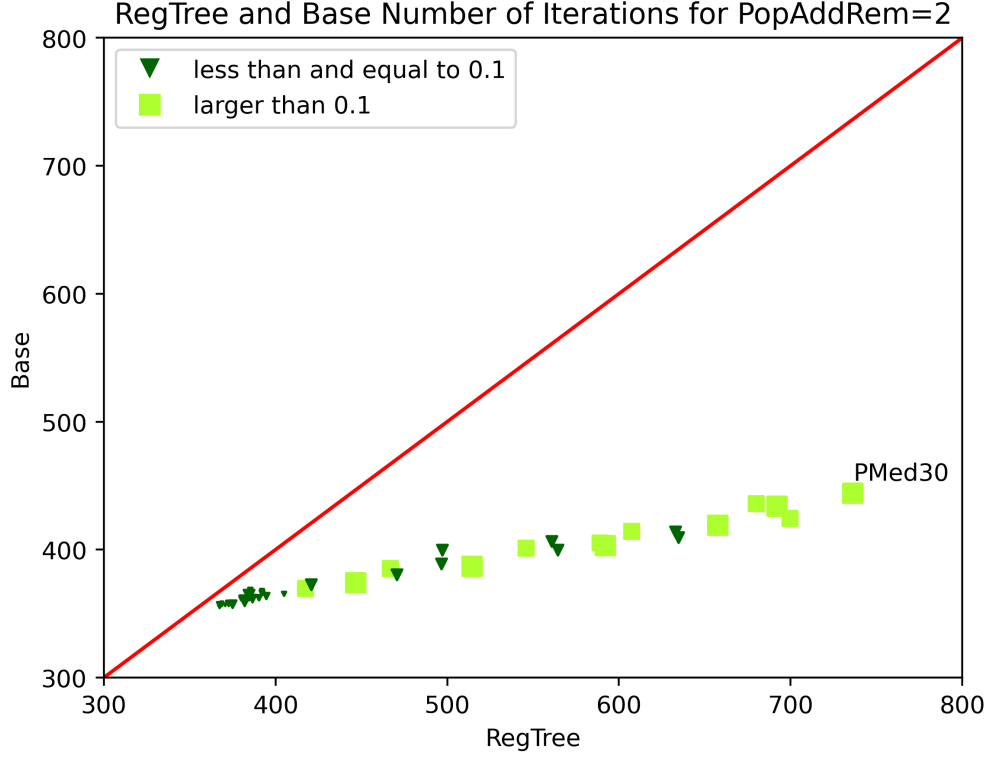


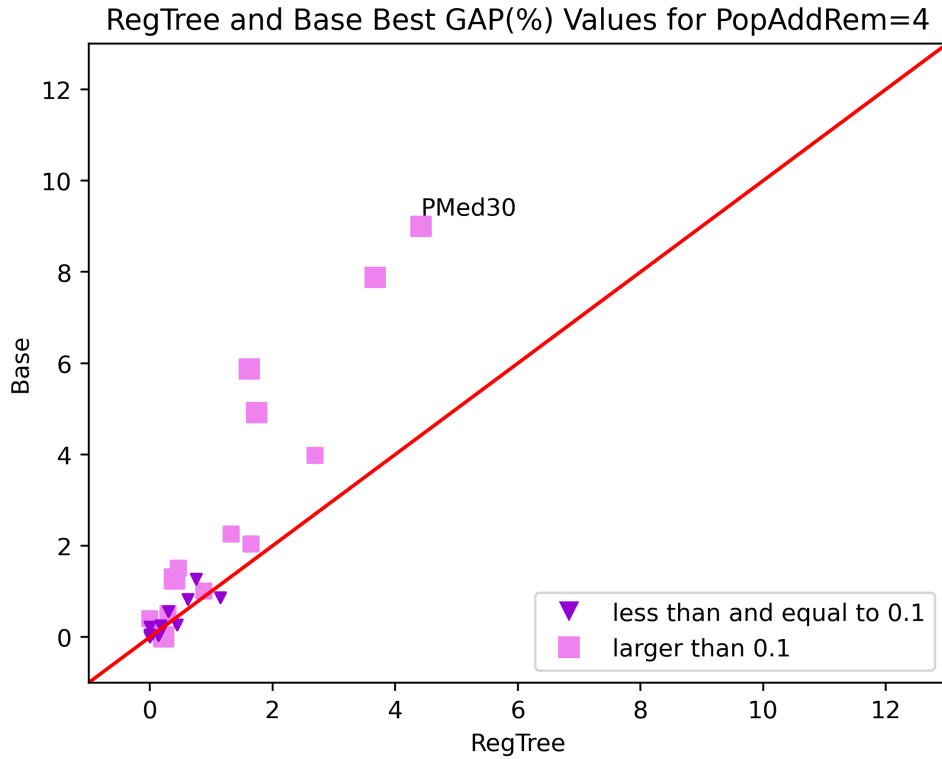
Figure 3.23. Number of Iterations - RegTree and Base methods, $PopAddRem=2$.

To address the diversification issue in more sparse and relatively easy problems, trials with $PopAddRem=4$ were also conducted. The $GAP(\%)$ values improved for both *RegTree* and *Base* methods. Even though the proposed *RegTree* algorithm performs slightly worse than the *Base* method in problem instances with higher sparsity (lower p/n_{loc} ratio), the *RegTree* method shows better performance overall with the setting $PopAddRem=4$ compared to $PopAddRem=2$. According to the Best $GAP(\%)$ values, the *RegTree* method performs fairly well, except for some problem instances (e.g., PMed5, PMed7, PMed27, and PMed40) with minor differences, as shown in Figure 3.24. As highlighted in Figure 3.25, according to the Average $GAP(\%)$ values, the suggested *RegTree* method significantly outperforms the *Base* method in problem instances with higher p/n_{loc} ratios, which are also relatively difficult problems. The results are summarized in Table A.4.

In these trials, the overall solution time of the *RegTree* algorithm decreases, while the solution time of the *Base* method increases, as seen in Figure 3.26 and summarized

in Table A.5 . This improvement in the solution time for the *RegTree* method is due to the adjustment of the *PopAddRem* parameter to 4, where solutions are added using the best leaf node solutions. This leads to faster convergence to the best solution, which is also reflected in the number of iterations. The average number of iterations is presented in Figure 3.27. Notably, the number of iterations for the *RegTree* method is consistently higher than for the *Base* method across all instances, as shown in Figure 3.27. It should be noted that Figures 3.26 and 3.27 provide the average values of multiple replications.

As the solution time decreases and solution quality improves for the suggested *RegTree* algorithm, the setting *PopAddRem*=4 is preferable for addressing problems with higher p/n_{loc} ratios.



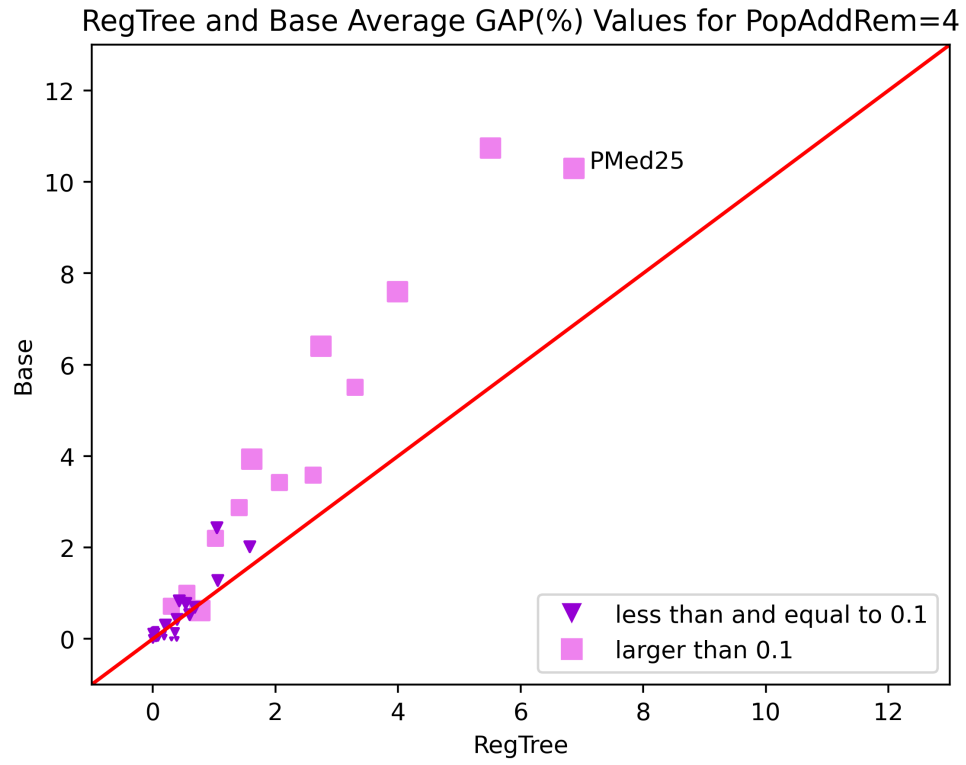


Figure 3.25. Average GAP(%) - RegTree and Base methods, *PopAddRem*=4.

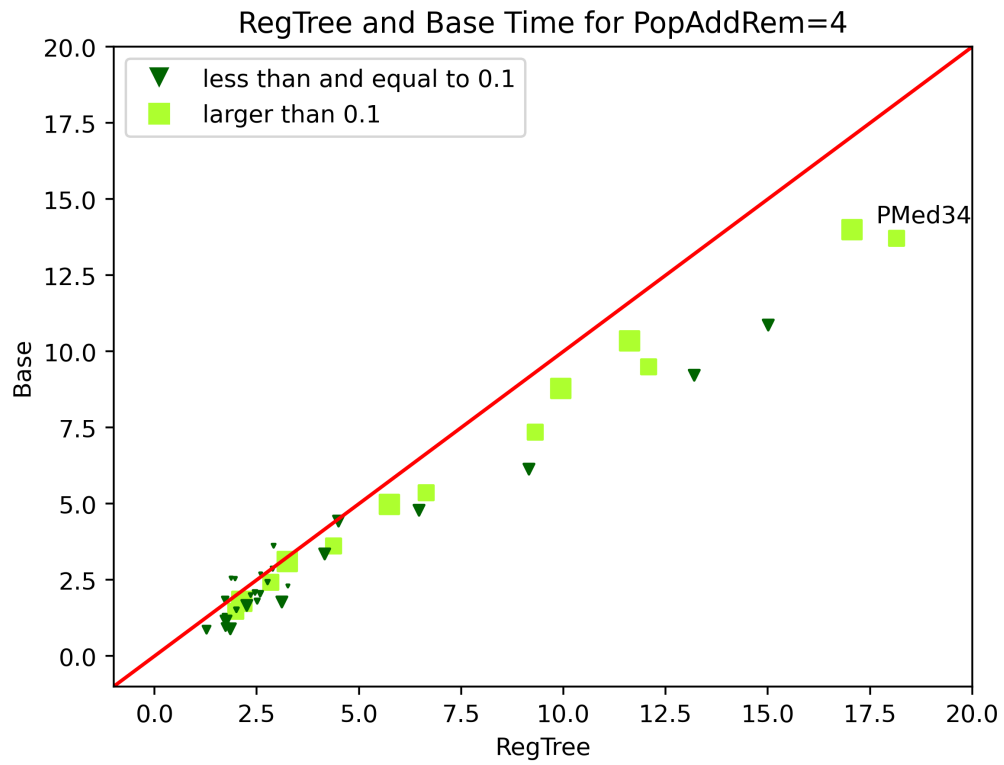


Figure 3.26. Solution Time - RegTree and Base methods, *PopAddRem*=4.

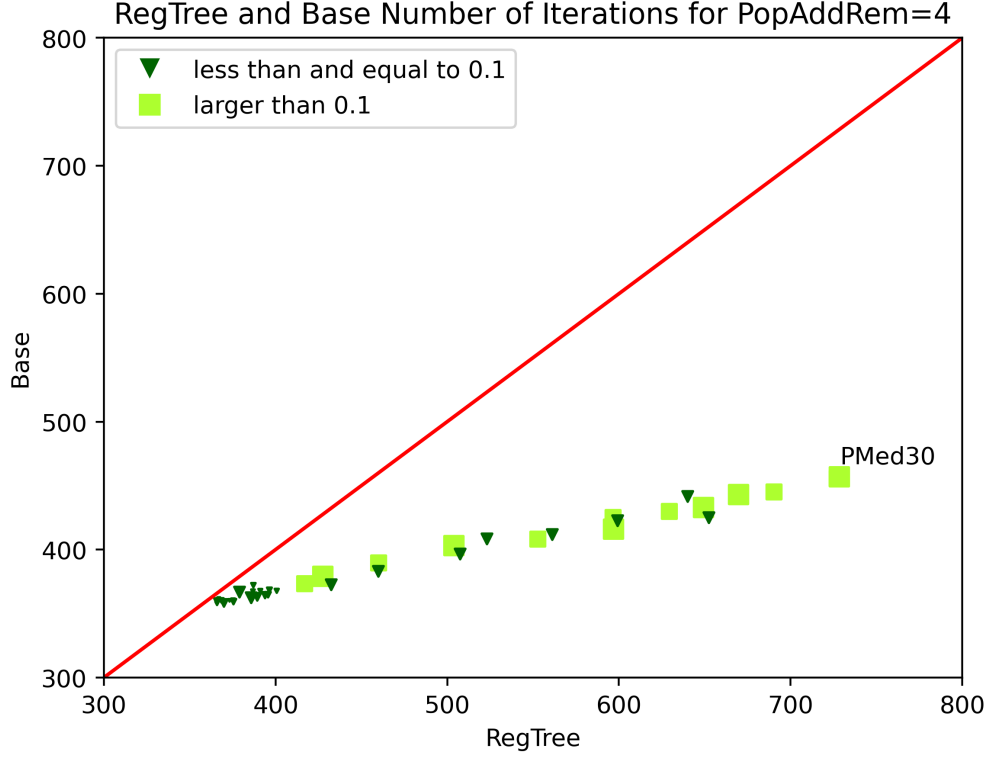


Figure 3.27. Number of Iterations - RegTree and Base methods, $PopAddRem=4$.

3.3.5. Comparison of RegTree method with MDM-GRASP

After establishing that the *RegTree* algorithm significantly outperforms the Base algorithm, we expanded our analysis comparing it with established methods from the literature. In the study by Plastino et al. [20], three approaches — GRASP, FIM-integrated GRASP (DM-GRASP), and multi-data mining GRASP (MDM-GRASP) — were applied to the p -median problem. These methods were evaluated on the PMed1-40 benchmark instances from the OR Library. The summarized results of the algorithms in their study, including the Best and Average GAP(%) values for each method, are provided in Table A.6. To assess the performance of our algorithm, we conducted a detailed comparison of our results with those reported by Plastino et al. [20]. Among these methods, MDM-GRASP achieved the best results in terms of both Best and Average GAP(%) values, making it the most effective among the three methods. However, DM-GRASP showed superior performance for specific instances, outperforming MDM-GRASP for PMed4, PMed8, and PMed13 in terms of Best GAP(%) values and

for PMed6 in terms of Average GAP(%) values. Consequently, we focus our comparison on evaluating the performance of our proposed RegTree algorithm against MDM-GRASP.

Given the prominence of MDM-GRASP as a benchmark method and its demonstrated effectiveness on PMed instances, we selected it as the primary point of comparison for our proposed *RegTree* algorithm. This decision allows us to rigorously assess the capability of *RegTree* in achieving high-quality solutions for the p -median problem relative to one of the most competitive algorithms available. Our comparison focuses on both Best and Average GAP(%) metrics to ensure a comprehensive evaluation of solution quality across a range of problem instances.

The comparisons of Best GAP(%) and Average GAP(%) values for the PMed1-40 problem instances are presented in Figures 3.28 and 3.29, respectively. We selected *PopAddRem* as 4 for these comparisons, as the *RegTree* method performs better in terms of both solution time and quality with this setting, as discussed earlier.

As shown in Figure 3.28, the *RegTree* algorithm outperforms MDM-GRASP across most problem instances in terms of Best GAP(%) values. The *RegTree* algorithm achieves better results in all but a few instances (e.g., PMed7, PMed27, and PMed30), where MDM-GRASP performs slightly better. However, the differences in these cases are minor, and *RegTree* remains competitive. For Average GAP(%) values (Figure 3.29), the *RegTree* algorithm demonstrates a significant advantage, especially for instances with higher difficulty and less sparsity. In these challenging cases, *RegTree* consistently produces solutions closer to the optimal.

These results highlight the robustness of the *RegTree* algorithm and its ability to effectively refine the solution space to produce high-quality solutions. The consistent performance improvement under the *PopAddRem*=4 configuration further emphasizes the adaptability of the method in addressing challenging combinatorial optimization problems.

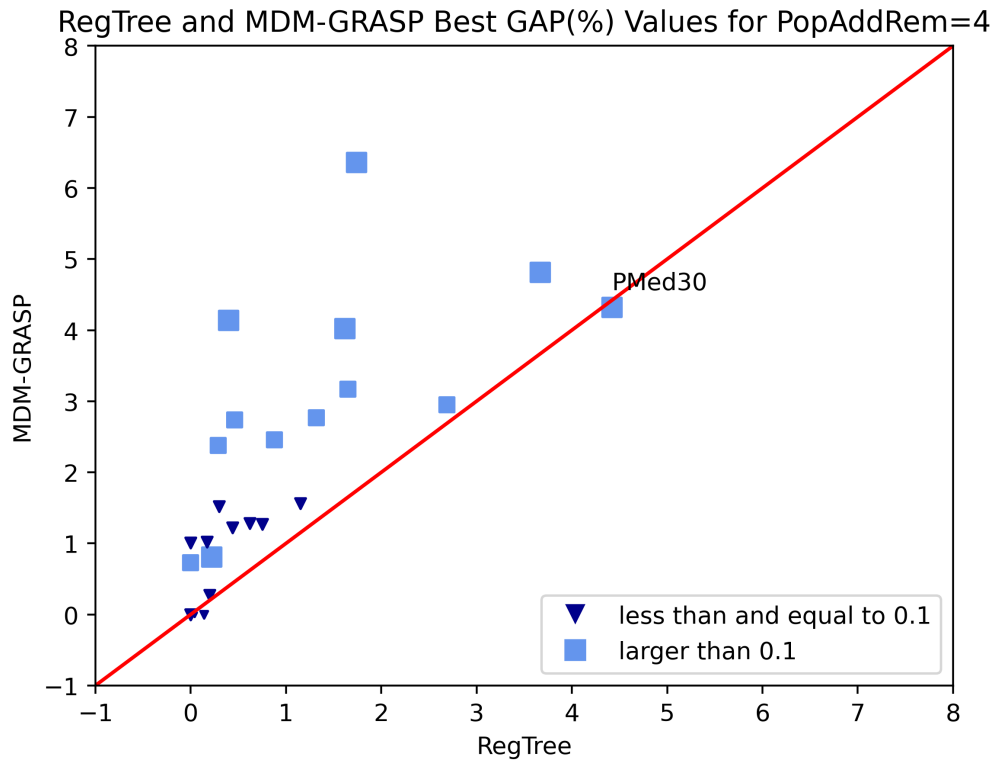


Figure 3.28. Best GAP(%) - RegTree and MDM-GRASP.

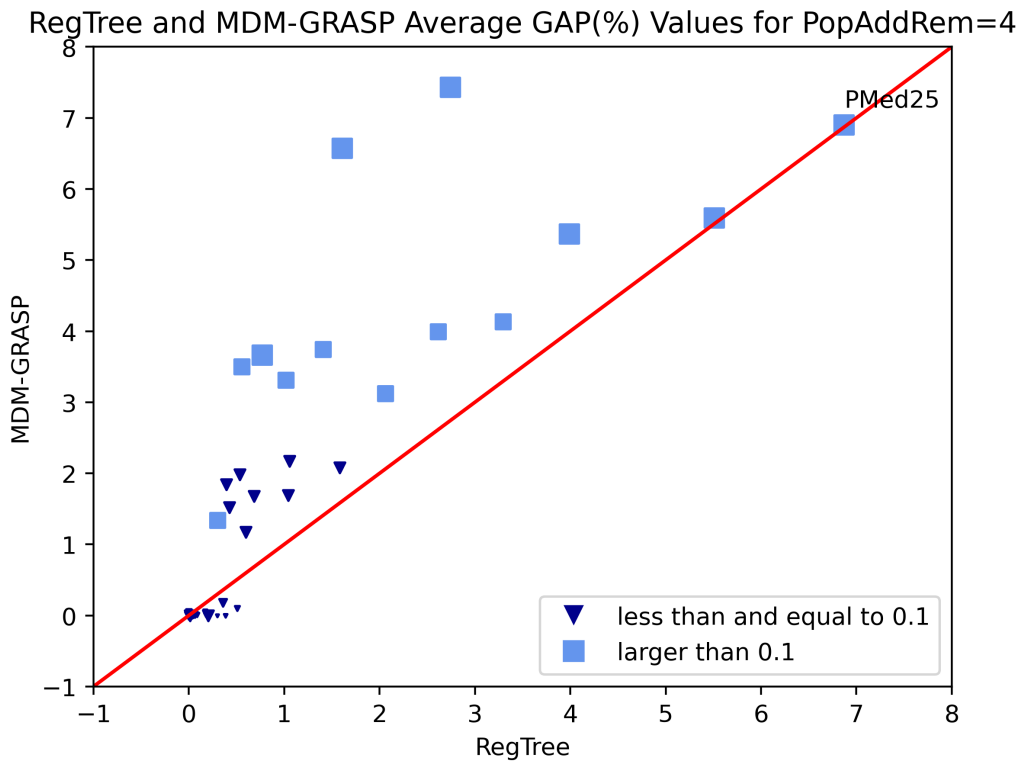


Figure 3.29. Average GAP(%) - RegTree and MDM-GRASP.

3.3.6. Adding new problem-specific features

We analyze the performance of our proposed *RegTree* algorithm by comparing it with the *Base* algorithm and MDM-GRASP as reported in [20] in Sections 3.3.4 and 3.3.5, respectively. In these computational studies, we initially use only binary decisions as features in *RegTree*. However, representing solutions with only binary features is rather basic. Therefore, we decided to enhance the representation of solutions by incorporating problem-specific features.

We enhance the representation of solutions by incorporating the following three p -median problem-specific features:

- (i) Minimum value of the pairwise demand-weighted distances among the facility-opening locations as

$$\text{MinDist} = \min_{i,k \in I, i \neq k} \left(\sum_{j \in J} d_j \cdot c_{ij} \cdot c_{kj} \right). \quad (3.14)$$

- (ii) Mean of the pairwise demand-weighted distances among the facility-opening locations as

$$\text{MeanDist} = \frac{1}{\binom{p}{2}} \sum_{\substack{i,k \in I \\ i \neq k}} \left(\sum_{j \in J} d_j \cdot c_{ij} \cdot c_{kj} \right). \quad (3.15)$$

- (iii) Variance in the number of customers assigned to the opened facilities as

$$\text{VarCust} = \frac{1}{p} \sum_{i \in I} \left(\sum_{j \in J} x_{ij} - \frac{1}{p} \sum_{i' \in I} \sum_{j \in J} x_{i'j} \right)^2. \quad (3.16)$$

In these equations, d_j represents the demand of the customer j . Note that in our uncapacitated p -median problem, $d_j = 1$ for all $j \in J$ since every customer must be assigned to exactly one facility.

For this experiment, we set the algorithm parameters *NIT* and *NPM* to 7 and 50, respectively. For all instances, the initial population size (*PopSize*) and the population

remove-add size ($PopAddRem$) are set to 100 and 2, respectively. The data points are color-coded: if the p/n_{loc} ratio is less than or equal to 0.1, they are light; if it is greater than 0.1, they are dark.

The comparison between the enhanced *RegTree* algorithm with three additional features (*RegTree+3Features*) and the basic *RegTree* algorithm in terms of Best GAP(%) and Average GAP(%) is provided in Figures 3.30 and 3.31, respectively. In Figure 3.30, the number of data points is less than the number of problem instances because the Best GAP(%) value equals 0 in 18 problem instances. Adding three new problem-specific features improves the Average GAP(%) values in 21 instances and remains the same in 7 instances, as shown in Table A.8. As depicted in Figure 3.31, *RegTree+3Features* performs better in problem instances where the p/n_{loc} ratio is greater than 0.1. These problem instances have relatively less sparse datasets and are more challenging to solve. This suggests that adding new features enhances the algorithm's learning capability, leading to better exploitation of the solution space.

RegTree and RegTree+3Features Best GAP(%) Values for PopAddRem=2

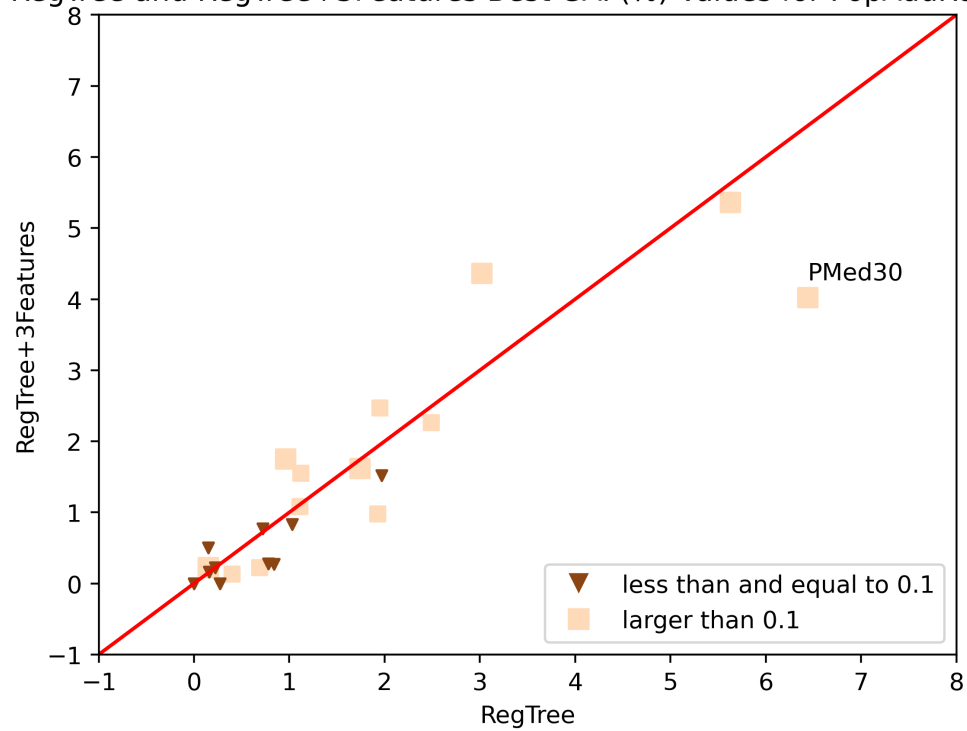


Figure 3.30. Best GAP(%) - RegTree and RegTree+3Features.

RegTree and RegTree+3Features Average GAP(%) Values for PopAddRem=2

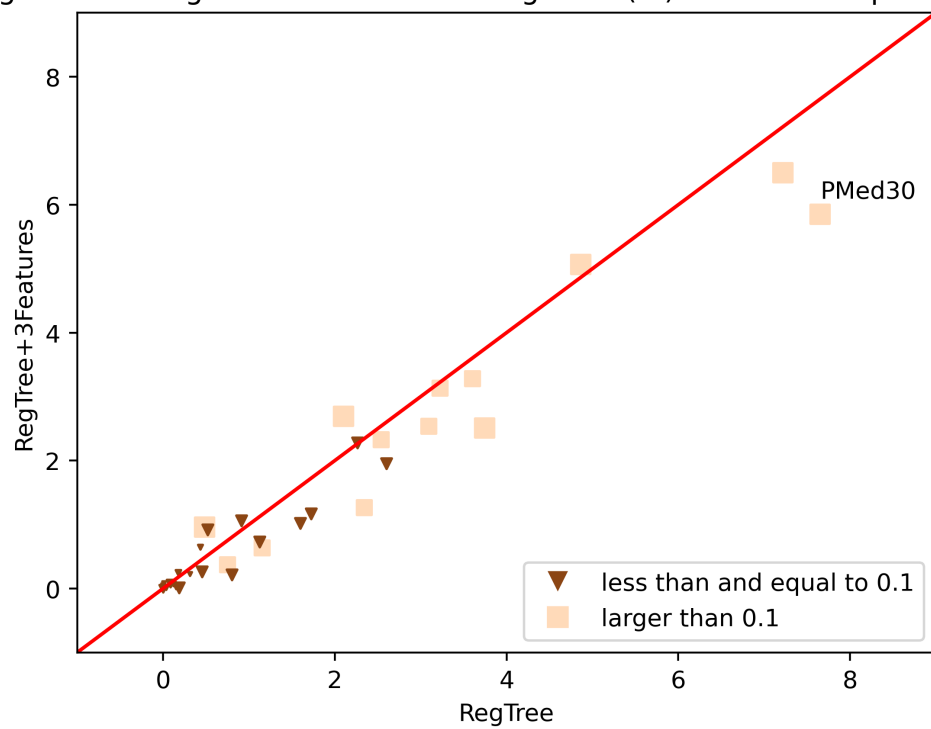


Figure 3.31. Average GAP(%) - RegTree and RegTree+3Features.

3.4. Computational Studies for QAP

We now apply the general *RegTree* approach to the Quadratic Assignment Problem (QAP) and compare its results with the *Base* method. Both the *RegTree* and *Base* methods, as explained in the previous sections, were implemented using Python in Jupyter Notebook and PyCharm environments. The experiments were conducted on a computer with an Intel Core i7-4510U CPU running at 2.00GHz, 8GB of RAM, and a Microsoft Windows 10 operating system.

We conducted trials of the *RegTree* algorithm on thirteen QAP instances from QAPLIB. The initial population size was set to 100. Throughout the iterations, the population size increased whenever the objective value of a solution improved, leading to the addition of a new solution to the population. The minimum samples in the leaf node were determined as 5% of the initial population size, which is also the size of the first N solutions selected for the *Base* algorithm. Therefore, the value of *ARSolns* was equal to 5% of the initial population size in these experiments. In the trials, MSE, defined in Equation (3.12), is used as a metric to evaluate the performance of the regression tree.

For this experiment, we selected the algorithm parameters *NIT* and *NPM* as 7 and 50, respectively. For all instances, the initial population size (*PopSize*) and the population remove-add size (*PopAddRem*) were set to 100 and 2, respectively. Additionally, we conducted five replications for each problem instance to ensure the robustness of the results.

3.4.1. RegTree Design Requirements for QAP

In general, the permutation solution representation is used for QAP. For our proposed framework, we convert this representation to a binary solution representation. Given a QAP instance with n locations and n facilities, we create $n \times n$ binary variables. Each set of n variables represents one location, with exactly one variable set

to 1, indicating the assigned facility. The order of these ones determines the facility assignment, as illustrated in Figure 3.32. Hence, we have $n \times n$ features to construct a regression tree. The target value is the objective value. Another problem-specific aspect that we consider in design requirements is the move operator. In the local search stage, we need to make moves to search neighborhood space. We apply a 1-swap move operator for QAP. To improve a selected solution, we determine one facility to swap using Algorithm 1: Determination of Bad Decision Algorithm. We decide on another facility to be swapped with this determined facility by a greedy heuristic.

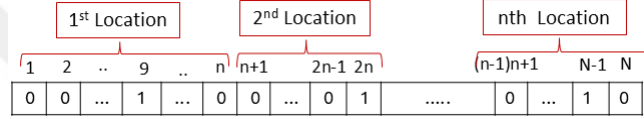


Figure 3.32. Solution Representation Design for QAP.

One of the important design choices for QAP is in the population update stage. We generate new solutions using Algorithm 2 with some corrections. Because of the binary solution representation, we calculate the frequency of the facility assignment for each location separately. For each location, we obtain an n -size table containing the frequency of each facility and thus generate the facility assignment decision for each location separately using the frequency tables of each location. This method leads to duplication, with some facilities assigned to multiple locations. We apply a QAP-specific greedy heuristic to eliminate duplication, the Maximum Flow to Minimum Distance Heuristic. The facilities assigned to multiple locations are eliminated by looking at the frequency value. The facility stays in the location with the maximum frequency value. We match the remaining locations with non-matched facilities using the Maximum Flow to Minimum Distance Heuristic algorithm. Lastly, we remove the worst solutions with respect to objective values as many as we generate.

3.4.2. Comparison of RegTree, Base, and RegTree added Three Features methods

We also analyze our developed *RegTree* algorithm by applying it to the Quadratic Assignment Problem (QAP). In this analysis, we compare the performance of three methods: *RegTree*, *Base*, and *RegTree* with three additional features specific to QAP, referred to as *RegTree+3Features*.

The three features added to the binary features are specific to QAP. Representing the solutions using only binary features is quite basic. Therefore, we decided to enhance the representation of solutions by incorporating problem-specific features.

We enhance the representation of solutions by incorporating the following three QAP-specific features:

- (i) Desirability Ratio: Facilities are sorted in descending order based on flows, and locations are sorted in ascending order based on distances. For each facility-location match decision, the ratio is calculated by dividing the order of the location by the order of the facility. The desirability ratio is the average of these ratios as

$$\text{Desirability Ratio} = \frac{1}{n} \sum_{k \in K} \frac{\text{Order of Location } i_k}{\text{Order of Facility } k}. \quad (3.17)$$

- (ii) Flow of the Facility Assigned to the Median Location: Determined by sorting locations in ascending order as

$$\text{Flow at Median Location} = f_{k_m, l_m} \quad \text{where } i_m = \left\lfloor \frac{n+1}{2} \right\rfloor. \quad (3.18)$$

- (iii) Distance Order of the Location to which the Facility with the Maximum Flow is Assigned as

$$\text{Distance Order of Location with Max Flow} = \text{Order of Location } i_{k^*}. \quad (3.19)$$

where $k^* = \arg \max_{k \in K} f_{kl}$.

3.4.3. Q-Matrix Calculation

The Q -matrix in the Quadratic Assignment Problem (QAP) is a mathematical representation that combines the distance matrix (D) and the flow matrix (F) to form a unified representation of the cost structure. The Q -matrix is defined as follows:

$$Q_{(i,k),(j,l)} = d_{ij} \cdot f_{kl}, \quad \forall i, j, k, l \in \{1, \dots, n\} \quad (3.20)$$

where d_{ij} is the distance between locations i and j , and f_{kl} is the flow or interaction cost between facilities k and l , and $Q_{(i,k),(j,l)}$ is the entry in the Q -matrix corresponding to the assignment of facility-location pairs (i, k) and (j, l) .

The dimensions of the Q -matrix are $n^2 \times n^2$, where n is the number of locations or facilities, and its rows and columns represent all possible facility-location pairs.

The Q -matrix is constructed as follows:

$$Q = \begin{bmatrix} d_{11} \cdot F & d_{12} \cdot F & \dots & d_{1n} \cdot F \\ d_{21} \cdot F & d_{22} \cdot F & \dots & d_{2n} \cdot F \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} \cdot F & d_{n2} \cdot F & \dots & d_{nn} \cdot F \end{bmatrix}$$

Each submatrix $d_{ij} \cdot F$ is an $n \times n$ matrix resulting from the element-wise multiplication of the scalar d_{ij} with the flow matrix F .

3.4.3.1. Calculation Steps. The calculation of Q involves the following steps:

- (i) Compute the Cartesian product of all possible facility-location pairs, resulting in n^2 combinations.
- (ii) For each pair of facility-location combinations (i, k) and (j, l) , calculate the corresponding entry in the Q -matrix as

$$Q_{(i,k),(j,l)} = d_{ij} \cdot f_{kl}. \quad (3.21)$$

- (iii) Assemble the entries into a $n^2 \times n^2$ matrix, where the rows and columns correspond to the facility-location pairs.

3.4.3.2. Properties of the Q -Matrix.

- Size: The Q -matrix has dimensions $n^2 \times n^2$.
- Sparsity: The sparsity of Q depends on the sparsity of the D -matrix and F -matrix. If either matrix contains many zeros, the Q -matrix will also contain a significant number of zeros.
- Symmetry: If both D and F are symmetric matrices, the Q -matrix is also symmetric.

3.4.3.3. Application in QAP. The Q -matrix is used in the quadratic assignment problem to compute the total cost of a given assignment X of facilities to locations. The objective function can be expressed in terms of Q as:

$$z = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n \sum_{l=1}^n Q_{(i,k),(j,l)} \cdot x_{ik} \cdot x_{jl} \quad (3.22)$$

where x_{ik} is a binary decision variable that indicates whether the facility k is assigned to location i .

This compact representation simplifies the mathematical formulation of the QAP and facilitates its implementation in optimization algorithms.

3.4.3.4. Q -Matrix Density. The density of the Q -matrix is a measure of the proportion of non-zero elements in the matrix. It provides insight into the sparsity or compactness of the problem's cost structure. Mathematically, the density of the Q -matrix is defined as

$$\text{Density}(Q) = \frac{\text{Number of Non-Zero Elements in } Q}{\text{Total Number of Elements in } Q}. \quad (3.23)$$

Let Q be a matrix of size $n^2 \times n^2$. Then:

$$\text{Density}(Q) = \frac{\text{nnz}(Q)}{n^4} \quad (3.24)$$

where $\text{nnz}(Q)$ is the number of non-zero elements in the Q -matrix, and n^4 is the total number of elements in the Q -matrix, since Q has dimensions $n^2 \times n^2$.

A higher density value indicates that the Q -matrix is more compact, with fewer zero entries, while a lower density value signifies sparsity. This metric is particularly useful in analyzing the structure of the problem instances and their impact on the computational complexity of solving the QAP.

3.4.4. Results and Analysis of QAP Instances

We selected 13 problem instances from QAPLIB to evaluate the performance of the *RegTree*, *Base*, and *RegTree+3Features* methods. The comparisons with respect to Best GAP(%) and Average GAP(%) are shown in Figures 3.33 and 3.34, respectively. To enhance the interpretation of the QAP results figures, I incorporated a third dimension representing the Q -matrix density. This dimension is visualized using a color gradient, where the color of the bars corresponds to the density of the Q -matrix. This addition provides insights into the sparsity or density of the problem instance's cost structure, enabling a more comprehensive analysis of the results across different scenarios.

As illustrated in Figures 3.34 and 3.33, the *Base* method does not outperform the other methods in any of the problem instances, except for *esc32e* and *esc32g*. These instances are relatively easy, as indicated by their low density values, and all three methods achieve optimal results in all replications. The addition of problem-specific features (*RegTree+3Features*) shows noticeable improvement in specific instances, such as *chr12b*, *chr15b*, and *esc32h*, where the Q -matrix density is moderate. Additionally, the inclusion of these features helps achieve optimality in some replications for the *esc32h* instance, which neither the basic *RegTree* nor the *Base* method could reach, as depicted in the figures and detailed in Table A.9.

The Q -matrix density provides a valuable third dimension for interpreting the difficulty of problem instances. Problem instances with higher density values, such as lipa30a, lipa30b, and lipa40b, indicate less sparsity and are associated with higher performance variability. Conversely, instances with lower density values, such as esc32e and esc32g, are relatively easy to solve due to their inherent sparsity.

As the problem size increases, i.e., the number of facility-location pairs grows, the sparsity in the dataset also increases due to the binary representation, as explained in Section 3.4.1. This increased sparsity, reflected in lower density values, often degrades the performance of the regression tree and its ability to generate accurate predictions. This is particularly evident in larger instances like lipa30b and lipa40b, which are inherently more challenging due to their high density and increased complexity.

Additionally, we observe from Table A.9 that adding the three new features reduces the variance of replications. This indicates that the regression tree learns better and produces more consistent results across different replications, even though the average GAP(%) does not improve significantly. The density dimension provides further clarity, as instances with moderate density (e.g., chr15b) benefit more noticeably from these added features, improving both solution quality and consistency.

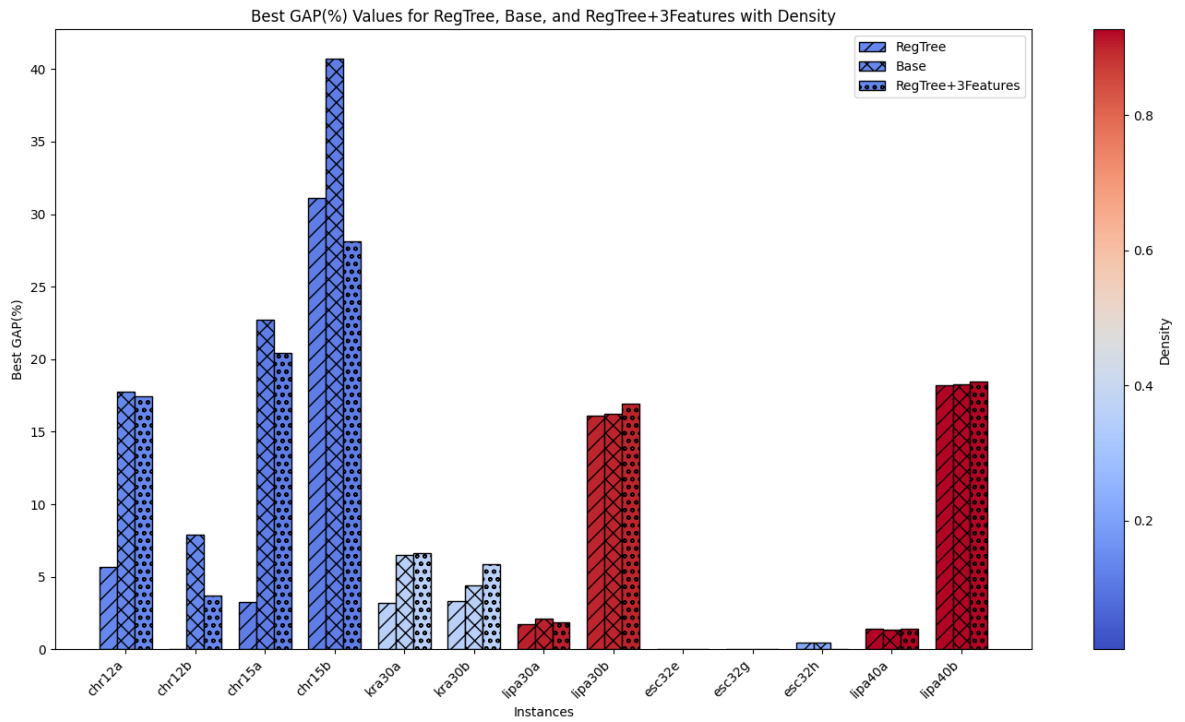


Figure 3.33. Best GAP(%) - RegTree, Base and RegTree+3Features.

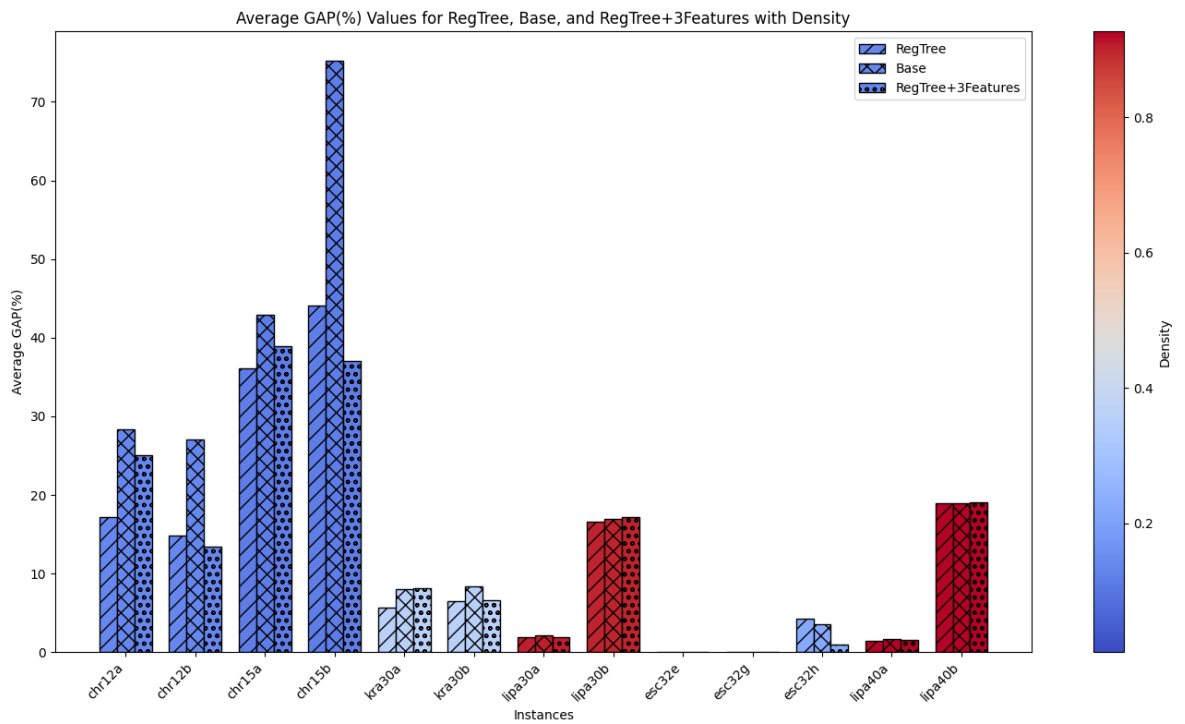


Figure 3.34. Average GAP(%) - RegTree, Base and RegTree+3Features.

4. VARIABLE FIXING ALGORITHM

This chapter introduces our tree-based Variable Fixing Algorithm (VFA), which utilizes regression trees to fix variables efficiently. By accepting the best leaf node solutions as high-quality outcomes, the algorithm benefits from split decisions on binary variables to significantly reduce the solution search space.

We begin by presenting the overall framework of the tree-based VFA, followed by a detailed exposition of its fundamental components.

4.1. General Scheme

The VFA is developed to solve discrete optimization problems involving binary decision variables with quadratic objective functions. It is particularly suited to instances from QPLIB (Quadratic Programming Library) that lack explicit constraints but require decision variables to take binary values (0 or 1). A key challenge in these problems is determining not only the optimal assignment of 0s and 1s but also the appropriate number of 1s in the solution. The algorithm benefits from regression trees to generate constraints and systematically fix the values of specific decision variables, reducing the search space and improving optimization efficiency.

4.1.1. Feasibility in the Binary Domain

Since the decision variables are binary, the algorithm inherently ensures feasibility by adhering to the 0 or 1 restriction. Initially, a set of random solutions is generated, where each solution is a binary vector corresponding to the decision variables. The algorithm does not impose a predefined ratio of 0s and 1s but instead explores the feasible space to determine the optimal distribution that maximizes the quadratic objective function. This ensures that all generated solutions are feasible by construction.

In cases where the problem includes inherent constraints on the decision variables (e.g., structural or problem-specific requirements), these constraints must be incorporated into the generated regression tree-based constraints. The algorithm accounts for such conditions during the constraint generation process, ensuring that the reduced solution space respects all problem-specific feasibility requirements.

4.1.2. Solution Space Restriction via Regression Trees

To enhance solver efficiency and reduce the complexity of the search space, the algorithm employs regression trees to generate constraints and fix the values of decision variables. The process is as follows:

- (i) Initial Solution Generation: A random set of binary solutions is generated to serve as the initial population.
- (ii) Regression Tree Construction: A regression tree is built using the generated solutions and their corresponding objective values. High-performing solutions are identified as those residing in the leaf node with the highest average objective value.
- (iii) Variable Fixing: The values of decision variables used for splits along the path leading to the best leaf node are fixed for all solutions in subsequent iterations. By fixing these variables, the regression tree focuses on the remaining decision variables, progressively narrowing the solution space.
- (iv) Constraint Generation: A collection of rules is accumulated through iterations, consisting of the variable values obtained from the paths of the regression tree. These rules are converted into inequality constraints, typically implemented as “less than or equal to” constraints in commercial solvers, to further refine the solution space.

This process effectively eliminates variables iteratively, reducing the dimensionality of the problem and improving the efficiency of the optimization process. By employing the regression tree in this manner, the VFA serves as a technique for vari-

able elimination, improving the focus and efficiency of the optimization process. This approach is particularly beneficial for reducing the combinatorial complexity of binary decision problems, enabling solvers to converge more effectively toward optimal solutions.

4.1.3. Applications and Effectiveness

The VFA is particularly effective for discrete optimization problems with the following characteristics:

- **Binary Decision Variables:** Problems where variables are restricted to binary values (0 or 1).
- **Quadratic Objective Functions:** Optimization tasks involving quadratic relationships among decision variables.
- **No Explicit Constraints:** Scenarios where traditional constraints are absent, allowing the algorithm to focus on generating effective constraints dynamically.

By iteratively fixing variable values and reducing the decision space, the algorithm ensures that computational resources are focused on the most promising regions of the solution space. The integration of regression tree-derived constraints further enhances the optimization process by guiding the solver toward high-quality solutions.

4.2. System Architecture

From the system architecture perspective, our proposed algorithm maintains a population of high-quality and low-quality tree-based mining solutions. It applies inequality constraints to a subset of decision variables based on the cumulative rule list to ensure that a certain percentage of the variables are constrained according to these rules. Characteristically, it is composed of five components: 1) a solution representation procedure (Section 4.2.1); 2) a population initialization and a regression tree construction procedure (Section 4.2.2); 3) rule set generation procedure (Section

4.2.3;4) variable fixing procedure (Section 4.2.4); and 4) population update procedure (Section 4.2.5).

4.2.1. Solution Representation

The solution representation is shown in Table 5.1. In our algorithm, we use a binary solution representation. The binary decision variable is denoted as y_i . After making these decisions, other decision variables can be calculated.

Table 4.1. Solution Representation of a Solution.

1	2	3	.	.	.	21	22	.	.	.	$n-1$	n
0	1	0	.	.	.	1	0	.	.	.	1	1

legend:

$i \longleftarrow \text{decision}$

y_i

4.2.2. Population Initialization and Regression Tree Construction

The tree-supported variable fixing algorithm starts by creating a population of randomly generated solutions, referred to as *Pop*, with a specified size *PopSize*. The structure of these solutions is illustrated in Table 4.1.

Subsequently, a regression tree is constructed using this population as the dataset. In this regression tree, the objective function values of the solutions serve as the dependent variables (responses), while the binary decision variables are treated as the independent variables (predictors). A visual representation of this process is provided in Figure 4.1.

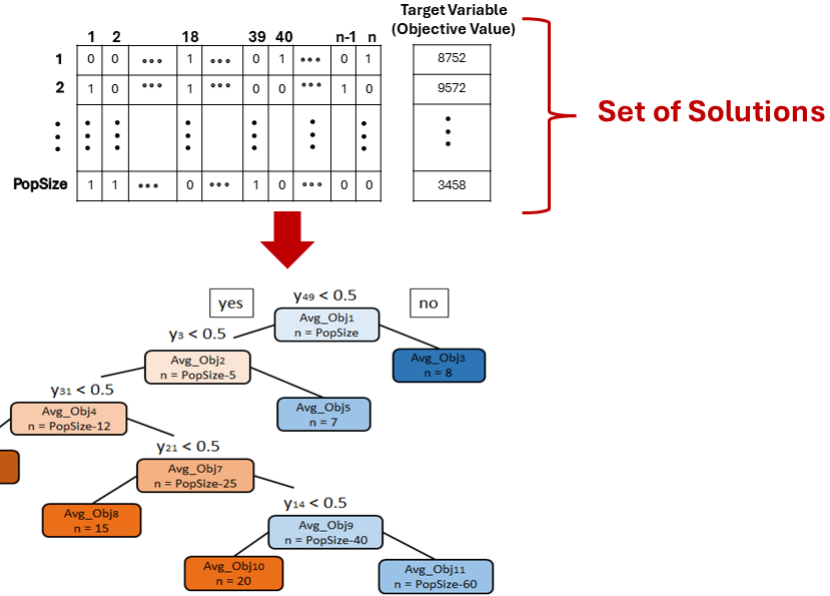


Figure 4.1. Initial Set of Solutions.

4.2.3. Rule Set Generation

Figure 4.2 illustrates the process of generating a rule set from the split decisions of a regression tree within VFA. In this tree, the nodes represent split decisions based on the binary decision variables (denoted as Y), and the leaves contain the average objective function values (denoted as Avg_Obj) along with the number of solutions that fall into each leaf (denoted as n).

The tree splits the population based on threshold values for the decision variables, guiding the solutions to different leaf nodes. The path highlighted in red indicates the route taken to reach the best leaf node, which contains the optimal average objective function value (in this case, Avg_Obj6 with $n = 10$).

The bottom section illustrates the rule set generated from the best leaf node's path. The rule set is derived from the split decisions along this path, specifying the conditions for the binary decision variables that lead to the best objective function value. In this example, the rule set indicates that for the decision variables Y_{49} , Y_3 ,

and Y_{31} , the values should be 0, 0, and 0, respectively, to achieve high-quality solutions. These conditions effectively narrow the solution space, guiding the optimization process toward regions with better performance.

This visualization clearly illustrates how our suggested algorithm utilizes regression tree analysis to generate a rule set that identifies the good configuration of decision variables for achieving the best objective function value.

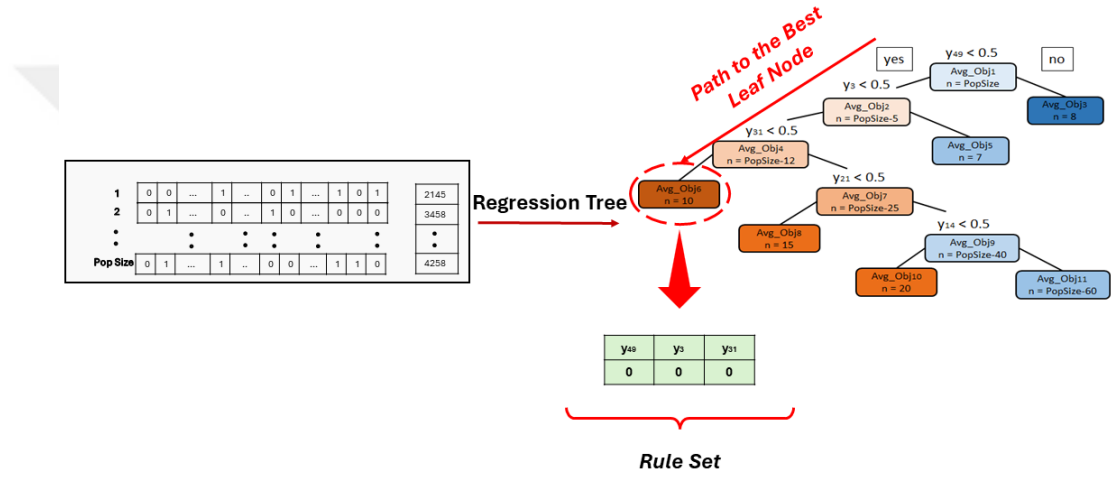


Figure 4.2. Rule Set Generation.

4.2.4. Variable Fixing Procedure

Commercial solvers often face challenges finding optimal solutions for QPLIB problem instances within a reasonable time frame. A tree-based classification method can fix variables efficiently and obtain good solutions. The figure 4.3 illustrates the process of fixing variables in a population of solutions using rules derived from regression trees, iteratively refining the population and accumulating rules to be used as constraints in a solver.

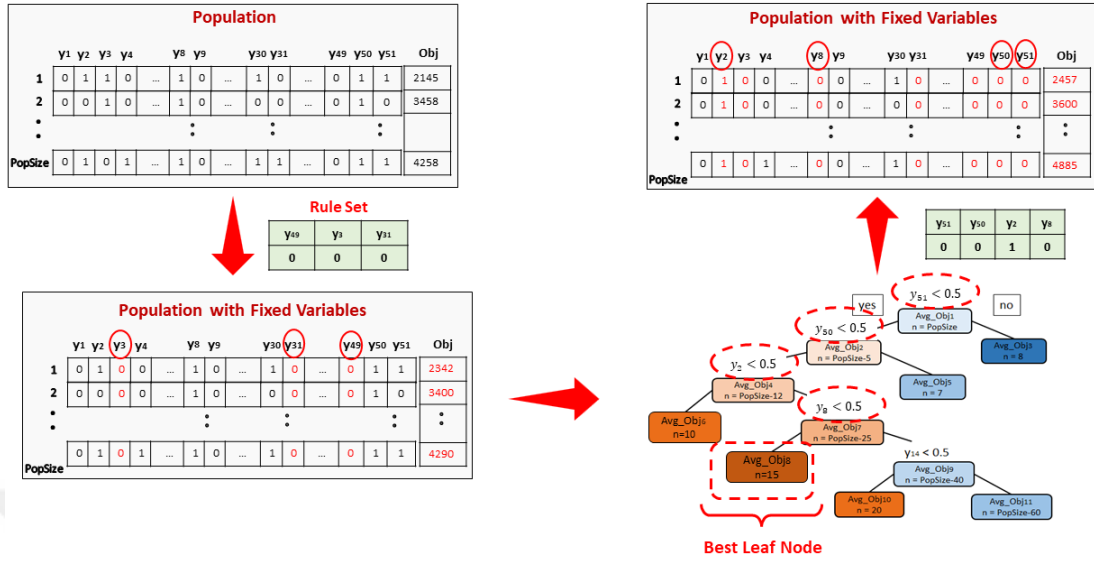


Figure 4.3. Variable Fixing Iterations.

Initially, a population of solutions is represented with various decision variables and their corresponding objective function values. The regression tree analysis, as described in Section 4.2.3, generates a rule set based on the best leaf node path. For example, the rule set obtained from the first regression tree includes fixing Y_{49} , Y_3 , and Y_{31} to 0.

This rule set is then applied to the entire population of solutions. By fixing these variables, all solutions in the population now have the same values for Y_{49} , Y_3 , and Y_{31} . This ensures that in the subsequent regression tree iterations, these fixed variables are no longer considered for splits, as they have uniform values across the dataset. Instead, the regression tree focuses on splitting the remaining decision variables, generating new rules based on the updated dataset. In the next iteration, a new regression tree is generated from the population with the fixed variables. This tree identifies the best leaf node path, resulting in a new rule set (e.g., fixing Y_{51} , Y_{50} , Y_2 , and Y_8 to specific values). This new rule set is then applied to the population, and the process repeats.

With each iteration, the rules derived from the regression trees are accumulated into a rule set list. This iterative refinement continues for a predetermined number of

iterations, progressively fixing more variables and refining the population of solutions. The accumulated rules are eventually used as constraints in the solver, reducing the problem's complexity and aiding the solver in efficiently finding optimal or near-optimal solutions. The iterative process ensures that the regression tree method systematically narrows down the solution space by fixing variables, allowing the solver to focus on the most promising regions of the solution space and enhancing the efficiency of finding high-quality solutions.

4.2.5. Population Update Procedure

The population update part of the algorithm involves refreshing the current population of solutions by incorporating new solutions obtained from a commercial solver, such as Gurobi. Additionally, new solutions are generated based on the rules derived from the split decisions leading to the worst leaf node path or by employing the active learning approach, depending on the scenario.

The steps of the first scenario in this process include:

- (i) Retrieving the solutions from the solution pool of Gurobi.
- (ii) Appending these solutions to the original population of solutions.
- (iii) Randomly generating a predetermined number of new solutions.
- (iv) Fixing the variables of these new solutions using the rules obtained from the worst leaf node with the opposite value, or generating solutions based on the active learning methodology, depending on the scenario.
- (v) Appending these newly generated solutions to the population, obtained in (ii).

This process allows for the exploration of different parts of the solution space and increases the chances of finding better solutions. The new solutions are generated using the rules from the worst leaf node to investigate the regions of the solution space that are not covered by the current population.

4.2.6. Detailed Algorithm Steps

The regression tree supported variable fixing algorithm's architecture is displayed in Figure 4.4. It outlines the key components and flow of the algorithm, showing how the regression tree is integrated into the variable fixing process to guide decision-making.

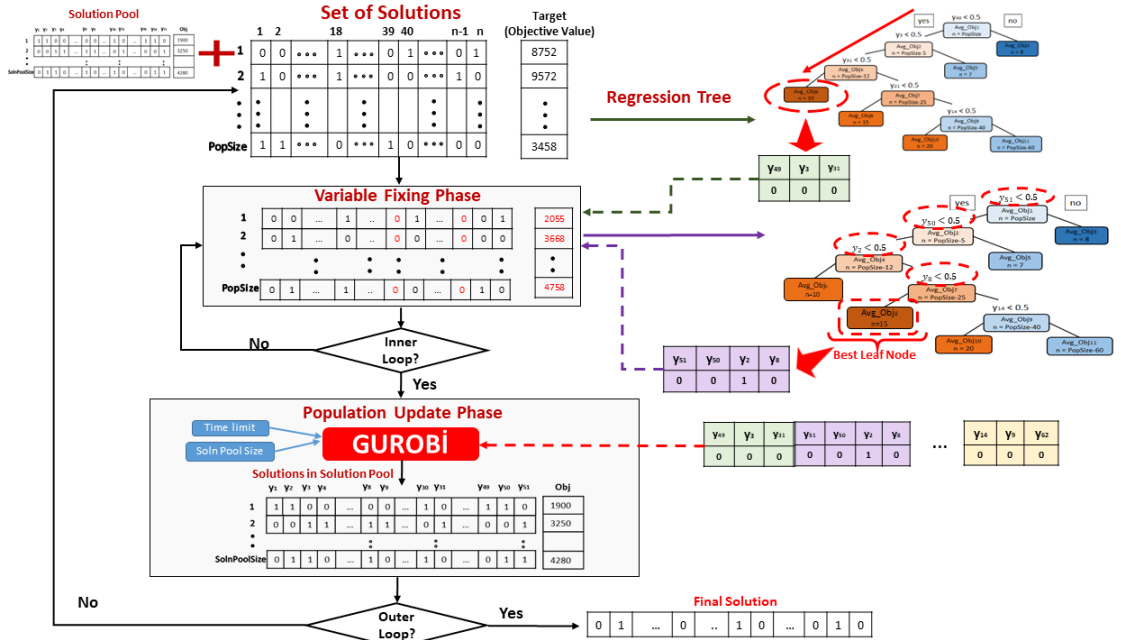


Figure 4.4. The Architecture of Algorithm.

The step-by-step explanation of the whole algorithm is as follows:

Initialization:

- Input Parameters: Number of initial solutions N , number of inner iterations I , number of outer iterations L , and time limit for Gurobi tl .
- Output: Optimal or near-optimal solution $\mathbf{x}^*$.
- Population Initialization: Randomly generate N solutions to form the initial population $\mathcal{P}$. Initialize the rule list $\mathcal{R}$ and cumulative rule list $\mathcal{R}_{cum}$.

Outer Loop Iterations ($i = 1$ to L):

- Regression Tree Generation: Generate a regression tree $\mathcal{T}$ using the current population $\mathcal{P}$.
- Leaf Node Determination: Determine the leaf nodes with average objective values $\mathcal{L}$ using the Determination of Leaf Nodes Algorithm.
- Rule Extraction: Extract rules $\mathcal{R}$ from the specified leaf nodes (best, second best, worst, second worst) using the Rule Set Generation Algorithm.

Inner Loop Iterations ($j = 1$ to I):

- Variable Fixing: Fix variables in the population $\mathcal{P}$ according to the extracted rules $\mathcal{R}$ using the Variable Fixing Algorithm.
- Regression Tree Update: Generate a new regression tree $\mathcal{T}$ using a randomly selected half of the population $\mathcal{P}$.
- Leaf Node Re-determination: Determine the new leaf nodes with average objective values $\mathcal{L}$ using the Determination of Leaf Nodes Algorithm.
- Rule Re-extraction: Extract new rules $\mathcal{R}$ from the updated leaf nodes (best, second best, worst, second worst) using the Rule Set Generation Algorithm.
- Variable Re-fixing: Fix variables in the population $\mathcal{P}$ again using the new rules $\mathcal{R}$ with the Variable Fixing Algorithm.
- Cumulative Rule List Update: Append the new rules $\mathcal{R}$ to the cumulative rule list $\mathcal{R}_{\text{cum}}$.

Gurobi Optimization:

- Solution Pool: Create a solution pool $\mathcal{SP}$ using Gurobi with constraints derived from the cumulative rule list $\mathcal{R}_{\text{cum}}$, respecting the time limit tl .
- Population Update: Update the population $\mathcal{P}$ using the Population Update Algorithm, which incorporates the best solutions from $\mathcal{SP}$ and generates new solutions based on the scenarios.

Final Solution Selection:

- Optimal Solution Identification: Identify the solution $\mathbf{x}^*$ in the final population $\mathcal{P}$ with the best objective value.
- Return Result: Return the optimal or near-optimal solution $\mathbf{x}^*$.

4.2.7. Pseudo-Codes and Detailed Explanations

In this section, we provide pseudo-codes for algorithms. The pseudo-code provides a detailed step-by-step description of the methods implemented in our research, making it easier to understand and replicate the results.

Algorithm 9 Regression Tree Processing Algorithm

1: **Input:**

- Regression tree $\mathcal{T}$ fitted on population $\mathcal{P}$
- Objective values O_i for each solution $S_i \in \mathcal{P}$

2: **Output:**

- $\mathcal{S}_{\text{best}}$: Solutions from the best leaf node
- $\mathcal{S}_{\text{sec_best}}$: Solutions from the second best leaf node
- $\mathcal{S}_{\text{worst}}$: Solutions from the worst leaf node
- $\mathcal{S}_{\text{sec_worst}}$: Solutions from the second worst leaf node

3: Extract leaf node IDs L_i for each solution S_i

4: **for** each leaf node $l \in \mathcal{T}$ **do**

5: Calculate the mean objective value $\bar{O}_l$ for leaf node l

6: **end for**

7: Sort leaf nodes in ascending order by mean objective values $\bar{O}_l$

8: $\mathcal{S}_{\text{best}} \leftarrow$ Solutions from the leaf node with $\min(\bar{O}_l)$

9: $\mathcal{S}_{\text{sec_best}} \leftarrow$ Solutions from the leaf node with the second smallest $\bar{O}_l$

10: $\mathcal{S}_{\text{worst}} \leftarrow$ Solutions from the leaf node with $\max(\bar{O}_l)$

11: $\mathcal{S}_{\text{sec_worst}} \leftarrow$ Solutions from the leaf node with the second largest $\bar{O}_l$

12: **Return** $\mathcal{S}_{\text{best}}, \mathcal{S}_{\text{sec_best}}, \mathcal{S}_{\text{worst}}, \mathcal{S}_{\text{sec_worst}}$

After creating an initial population of solutions through random generation, the regression tree processing algorithm begins. The Algorithm 9 takes in a regression tree $\mathcal{T}$ and the objective values O_i for each solution S_i in the population $\mathcal{P}$ as input and returns the best, second best, worst, and second worst leaf nodes based on their average objective values. The following steps outline the algorithm in detail:

(i) Input:

- The algorithm starts with a regression tree, denoted as $\mathcal{T}$, which is fitted on a population of solutions, $\mathcal{P}$. Each solution S_i in the population $\mathcal{P}$ is associated with a target variable, representing its objective value O_i .

(ii) Output:

- The algorithm aims to identify and return four sets of solutions:
 - $\mathcal{S}_{\text{best}}$: Solutions from the best leaf node.
 - $\mathcal{S}_{\text{sec_best}}$: Solutions from the second-best leaf node.
 - $\mathcal{S}_{\text{worst}}$: Solutions from the worst leaf node.
 - $\mathcal{S}_{\text{sec_worst}}$: Solutions from the second-worst leaf node.

(iii) Process:

- First, we extract the leaf node IDs, L_i , for each solution S_i .
- We then iterate through each leaf node l in the regression tree $\mathcal{T}$ to calculate the mean objective value $\bar{O}_l$ for that leaf node.
- After calculating the mean objective values, we sort the leaf nodes in ascending order based on these values.
- The algorithm then selects solutions from:
 - The leaf node with the smallest mean objective value and assigns them to $\mathcal{S}_{\text{best}}$.
 - The leaf node with the second smallest mean objective value and assigns them to $\mathcal{S}_{\text{sec_best}}$.
 - The leaf node with the largest mean objective value and assigns them to $\mathcal{S}_{\text{worst}}$.
 - The leaf node with the second largest mean objective value and assigns them to $\mathcal{S}_{\text{sec_worst}}$.

(iv) Return:

- Finally, the algorithm returns the sets of solutions: $\mathcal{S}_{\text{best}}$, $\mathcal{S}_{\text{sec_best}}$, $\mathcal{S}_{\text{worst}}$, and $\mathcal{S}_{\text{sec_worst}}$.

Algorithm 10 Rule Set Generation Algorithm

1: **Input:**

- Regression tree $\mathcal{T}$ fitted on population $\mathcal{P}$
- Leaf node type L_{type} (Best, Second Best, Worst, Second Worst)

2: **Output:**

- Rule set $\mathcal{R}$ derived from the specified leaf node type

3: Initialize an empty rule set $\mathcal{R}$

4: Generate Rules from Specified Leaf Node:

5: $\mathcal{L}_{\text{nodes}} \leftarrow$ Leaf nodes of type L_{type} in $\mathcal{T}$

6: **for** each variable y_j in the path to $\mathcal{L}_{\text{nodes}}$ **do**

7: **if** $y_j < 0.5$ in the path **then**

8: Add rule $y_j = 0$ to $\mathcal{R}$

9: **else**

10: Add rule $y_j = 1$ to $\mathcal{R}$

11: **end if**

12: **end for**

13: **Return** $\mathcal{R}$

The Rule Set Generation Algorithm (Algorithm 10) creates a set of rules based on the split decisions in a specified type of leaf node from a regression tree $\mathcal{T}$. Given a regression tree fitted on a population $\mathcal{P}$, the algorithm identifies leaf nodes of a specified type L_{type} (such as Best, Second Best, Worst, or Second Worst). For each variable y_j in the path to these leaf nodes, the algorithm generates a rule depending on whether the value of y_j is less than or greater than 0.5. These rules are then added to an initially empty rule set $\mathcal{R}$, which is returned as the output. This process allows for systematically extracting decision rules that characterize the solution space.

Algorithm 11 Variable Fixing Algorithm

1: **Input:**

- Population $\mathcal{P}$ with solutions S_i
- Regression tree $\mathcal{T}$ fitted on population $\mathcal{P}$
- Objective values O_i for each solution $S_i \in \mathcal{P}$
- Leaf node type L_{type} (Best, Second Best, Worst, Second Worst)

2: **Output:**

- Updated population $\mathcal{P}_{\text{fixed}}$ with fixed variables

3: $\mathcal{R} \leftarrow \text{Rule Set Generation Algorithm}(\mathcal{T}, L_{\text{type}})$

4: Apply Rules to Population:

5: **for** each solution $S_i \in \mathcal{P}$ **do**

6: **for** each rule $(y_j = v)$ in $\mathcal{R}$ **do**

7: Fix variable y_j in S_i to value v

8: **end for**

9: **end for**

10: $\mathcal{P}_{\text{fixed}} \leftarrow \mathcal{P}$

11: **Return** $\mathcal{P}_{\text{fixed}}$

The Variable Fixing Algorithm (Algorithm 11) applies the rules generated by the Rule Set Generation Algorithm to a given population $\mathcal{P}$. It takes as input the population of solutions S_i , the corresponding regression tree $\mathcal{T}$, and the objective values O_i . The specified leaf node type L_{type} determines from which leaf node the rules are extracted. For each solution S_i in the population, the algorithm fixes the variables according to the rules $(y_j = v)$ in the rule set $\mathcal{R}$. The output is an updated population $\mathcal{P}_{\text{fixed}}$ with variables set according to the derived rules, ensuring that the population adheres to the identified patterns within the solution space.

Algorithm 12 Population Update Algorithm

1: **Input:**

- Population $\mathcal{P}$
- Solution pool $\mathcal{SP}$
- Rule list $\mathcal{R}_{\text{worst}}$
- Minimum split size S_{min}

2: **Output:**

- Updated population $\mathcal{P}$

3: $\mathcal{P}_{\text{best}} \leftarrow$ Select the best k solutions from $\mathcal{SP}$

4: $\mathcal{P}_{\text{worst}} \leftarrow$ Select the worst k solutions from $\mathcal{P}$

5: **for** $i = 1$ to k **do**

6: Replace $\mathcal{P}_{\text{worst}}[i]$ with $\mathcal{P}_{\text{best}}[i]$

7: **end for**

8: Generate S_{min} new random solutions $\mathcal{P}_{\text{new}}$

9: **for** each solution $\mathbf{s}$ in $\mathcal{P}_{\text{new}}$ **do**

10: Fix variables in $\mathbf{s}$ according to $\mathcal{R}_{\text{worst}}$ by setting each variable y_i to $1 - y_i$ (Variable Fixing Algorithm)

11: **end for**

12: $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{P}_{\text{new}}$

13: **Return** $\mathcal{P}$

The Population Update Algorithm (Algorithm 12) refines the population of solutions by integrating the best solutions from a solution pool $\mathcal{SP}$ and generating new solutions based on rules derived from the worst leaf nodes. The worst solutions are replaced with the best ones, starting with selecting the best k solutions from the solution pool and the worst k solutions from the population. The algorithm then extracts a rule set from the worst leaf node splits and generates new random solutions $\mathcal{P}_{\text{new}}$. Each new solution has its variables fixed according to these rules. The updated population $\mathcal{P}$ is then formed by combining the existing population with the new solutions, maintaining diversity while ensuring adherence to the rules.

Algorithm 13 Active Learning Population Update Algorithm

1: **Input:**

- Current population $\mathcal{P}$
- Regression tree $\mathcal{T}$
- Number of new solutions N_{new}

2: **Output:**

- Updated population $\mathcal{P}$

3: $\mathcal{P}_{\text{new}} \leftarrow$ Generate N_{new} random solutions

4: $\mathcal{O}_{\text{new}} \leftarrow$ Calculate objective values for each solution in $\mathcal{P}_{\text{new}}$

5: Apply $\mathcal{P}_{\text{new}}$ to regression tree $\mathcal{T}$

6: $\mathcal{L} \leftarrow$ Find the leaf node with maximum variance in objective values

7: $\mathcal{P}_{\text{AL}} \leftarrow$ Initialize an empty set for Active Learning population

8: **for** each solution $\mathbf{s}$ in $\mathcal{P}_{\text{new}}$ **do**

9: **if** $\mathbf{s}$ falls in leaf node $\mathcal{L}$ **then**

10: Append $\mathbf{s}$ to $\mathcal{P}_{\text{AL}}$

11: **end if**

12: **end for**

13: $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{P}_{\text{AL}}$

14: **Return** $\mathcal{P}$

The Population Management based on Active Learning Methodology (Algorithm 13) involves generating a predetermined number of random solutions and calculating their corresponding objective values. These solutions are then applied to the current regression tree to determine their leaf node IDs. The algorithm identifies the leaf node with the maximum variance in objective values and selects the solutions within this node. These selected solutions are then appended to the current population, resulting in an updated population that utilizes high-variance regions for potential improvements in solution quality.

The benefits of the Active Learning Methodology are manifold. Firstly, by focusing on the leaf node with the maximum variance in objective values, the algorithm

effectively targets regions of the solution space with high potential for improvement. This approach ensures that the updated population is enriched with diverse and potentially high-quality solutions. Secondly, incorporating random solutions and evaluating them against the regression tree allows the algorithm to explore new areas of the solution space, promoting innovation and preventing premature convergence to suboptimal solutions. Finally, by iteratively refining the population based on high-variance nodes, the method balances exploration and exploitation, enhancing the overall robustness and effectiveness of the optimization process.

The Main Algorithm for Optimal or Near-optimal Solution (Algorithm 14) integrates all previously mentioned algorithms to iteratively refine the population of solutions and find the optimal or near-optimal solution. The process begins with the random generation of an initial population $\mathcal{P}$ and proceeds through L outer loop iterations. Each iteration generates a regression tree $\mathcal{T}$, and leaf nodes are identified to extract rules $\mathcal{R}$. An inner loop runs for I iterations, where variables are fixed according to the rules, and the population is updated. The pseudo-code explicitly considers a scenario where only half of the fixed population, denoted as $\mathcal{P}_{\text{half.fixed}}$, is used to construct the regression tree. However, in some cases, we utilize the entire fixed population ($\mathcal{P}_{\text{fixed}}$) to generate the tree. This flexibility allows us to explore different levels of data utilization when training the tree, potentially balancing between computational efficiency and the accuracy of partitioning. The cumulative rule list $\mathcal{R}_{\text{cum}}$ is used to constrain the solution pool generated by the Gurobi optimizer within a time limit tl . The population is then updated using the Population Update Algorithm, incorporating the best solutions from the Gurobi solution pool. The final solution $\mathbf{x}^*$, which has the best objective value, is returned as the output. This comprehensive algorithm effectively reduces the search space and enhances the quality of solutions through iterative refinement and optimization.

Algorithm 14 Main Algorithm for Optimal or Near-optimal Solution

1: **Input:**

- Number of initial solutions N
- Number of iterations: inner loop (IL) and outer loop (OL)
- Constraint threshold $cons_{threshold}$ and time limit for solver tl

2: **Output:**

- Optimal or near-optimal solution $\mathbf{x}^*$

3: **Initialize:**

- Population $\mathcal{P} \leftarrow$ Randomly generate N solutions
- $\mathcal{O}(\mathcal{P}) \leftarrow calculate_objective_values(\mathcal{P})$
- Rule lists $\mathcal{R}_{best} \leftarrow \emptyset$ and $\mathcal{R}_{worst} \leftarrow \emptyset$
- Cumulative rule lists $\mathcal{R}_{CumBest} \leftarrow \emptyset$ and $\mathcal{R}_{CumWorst} \leftarrow \emptyset$

5: **for** $i = 1$ to OL **do**

6: $\mathcal{T} \leftarrow generate_regression_tree(\mathcal{P}, \mathcal{O}(\mathcal{P}))$

7: $\mathcal{L}_{best}, \mathcal{L}_{worst} \leftarrow determine_leaf_nodes(\mathcal{T})$

8: $\mathcal{R}_{best}, \mathcal{R}_{worst} \leftarrow rule_extract(\mathcal{L}_{best}, \mathcal{L}_{worst}, \mathcal{P})$ (Algorithm 10)

9: $\mathcal{P}_{fixed} \leftarrow \mathcal{P}$

10: **for** $j = 1$ to IL **do**

11: $\mathcal{P}_{fixed} \leftarrow fix_variables(\mathcal{P}_{fixed}, \mathcal{R})$ (Algorithm 11)

12: $\mathcal{P}_{half.fixed} \leftarrow \lfloor \frac{\mathcal{P}_{fixed}}{2} \rfloor$

13: $\mathcal{T}_{fixed} \leftarrow generate_regression_tree(\mathcal{P}_{half.fixed}, \mathcal{O}(\mathcal{P}_{half.fixed}))$

14: $\mathcal{L}_{best}, \mathcal{L}_{worst} \leftarrow determine_leaf_nodes(\mathcal{T}_{fixed})$

15: $\mathcal{R}_{best}, \mathcal{R}_{worst} \leftarrow rule_extract(\mathcal{L}_{best}, \mathcal{L}_{worst}, \mathcal{P}_{fixed}, \mathcal{T}_{fixed})$ (Algorithm 10)

16: $\mathcal{R}_{CumBest} \leftarrow \mathcal{R}_{CumBest} \cup \mathcal{R}_{best}$

17: $\mathcal{R}_{CumWorst} \leftarrow \mathcal{R}_{CumWorst} \cup \mathcal{R}_{worst}$

18: **end for**

19: $\mathcal{SP} \leftarrow optimization_solver(\mathcal{R}_{CumBest}, cons_{threshold}, tl)$ (Algorithm 15)

20: $\mathcal{P} \leftarrow update_population(\mathcal{SP}, \mathcal{R}_{CumWorst})$ (Algorithm 12)

21: **end for**

22: $\mathbf{x}^* \leftarrow$ the solution in $\mathcal{P}$ with the best objective value

23: **Return** $\mathbf{x}^*$

The constraint ensures that at least a certain percentage (defined by the threshold $\text{cons}_{\text{threshold}}$) of the variables are fixed according to the rules in the cumulative rule list $\mathcal{R}_{\text{cum}}$. It is formulated as

$$\sum_{(i,v) \in \mathcal{R}_{\text{cum}}} ((1 - x_i)(1 - v) + x_i v) \geq \lfloor \text{cons}_{\text{threshold}} \cdot |\mathcal{R}_{\text{cum}}| \rfloor \quad (4.1)$$

where x_i is the decision variable corresponding to variable i , and v is the value to which x_i should be fixed (0 or 1).

Algorithm 15 Optimization Algorithm with Variable Fixing Constraints

1: **Input:**

- Cumulative rule list $\mathcal{R}_{\text{cum}}$
- Constraint threshold $\text{cons}_{\text{threshold}}$
- Time limit tl

2: **Output:**

- Solution pool $\mathcal{SP}$

3: **Initialize:**

- Model model $\leftarrow$ Create optimization model

4: Add variable fixing constraints to the model as per Equation (4.2):

$$\sum_{(i,v) \in \mathcal{R}_{\text{cum}}} ((1 - x_i)(1 - v) + x_i v) \geq \lfloor \text{cons}_{\text{threshold}} \cdot |\mathcal{R}_{\text{cum}}| \rfloor. \quad (4.2)$$

5: Solve the model within the time limit tl

6: $\mathcal{SP} \leftarrow$ Extract solution pool from the model

7: **Return** $\mathcal{SP}$

4.3. Illustrative Examples

We select the well-known and challenging Quadratic Programming(QP) problem to assess our suggested regression tree-supported variable fixing algorithm.

4.3.1. Quadratic Programming Problems

We select the instances from QPLIB (Quadratic Programming Library), a library of QP problems commonly used to test and compare the performance of different QP solvers. A QP problem consists of an objective function, a quadratic function of the decision variables, and a set of linear constraints on the decision variables. The general form of a QP problem can be written as:

$$\text{maximize} \quad x^T Q x + c^T x \quad (4.3)$$

$$\text{subject to} \quad A x \leq b \quad (4.4)$$

$$x_i \in \{0, 1\} \quad (i \in I) \quad (4.5)$$

$$x_j \in \mathbb{Z} \quad (j \in J) \quad (4.6)$$

where x is the decision variable vector, Q is the quadratic coefficient matrix, c is the linear coefficient vector, A is the constraint matrix, b is the right-hand side vector of constraints, I is the set of indices of decision variables that are required to be binary, and J is the set of indices of decision variables that are required to be integer-valued.

4.4. Numerical Results

We now apply a tree-based reduction in solution search space approach to the Quadratic Programming (QP). The suggested algorithm was implemented using Python 3.9 in Jupyter Notebook and VSCode environments. The experiments were conducted on a computer with an Intel Core i7-4510U CPU running at 2.00GHz, 8GB of RAM, and a Microsoft Windows 10 operating system. For the exact solution solver, we have selected Gurobi version 10.0.0.

The QPLIB library contains a diverse collection of Quadratic Programming (QP) problems designed to benchmark and compare the performance of various QP solvers. These problems vary widely in size, complexity, and structure, making them ideal for evaluating solver performance in different scenarios.

For our analysis, we selected 23 benchmark instances from the QPLIB library [59], specifically characterized by binary decision variables and the absence of constraints. Out of these, eight instances exhibited alternative optima, which were subsequently excluded due to their negative impact on our analysis. This exclusion is necessary because our evaluation method relies heavily on the number of matching decision variables to assess tree performance accurately. Thus, our trials focused on the remaining 15 instances. These instances feature objective functions composed of linear and quadratic terms, each with corresponding coefficients. The number of variables in these benchmark instances ranges from 120 to 1225, providing a varied dataset for robust performance evaluation.

In the trials, as discussed in Chapter 3, MSE, defined in Equation (3.12), is used as a metric to evaluate the performance of the regression tree. The regression tree parameters were configured as follows: a minimum of 10 solutions per leaf node, a minimum of 20 solutions required to perform a split and a maximum tree depth of 10. Furthermore, 63% of the total features were considered when looking for the best split at each node. The trials were carried out using all the features and the subset of features 63%, with the latter achieving better results. For the Gurobi solver, the solution pool size was set to 20, and the time limit was constrained to 10 seconds.

4.4.1. QPLIB Benchmark Instances Trial Results

In this study, the proposed algorithm is run for 5 replications. For each replication, the run time is obtained and the mean run time is computed across all replications. This mean run-time is then used as the time limit constraint for the Gurobi solver. The original results from Gurobi with this time limit are obtained and compared with the results produced by Gurobi when constraints generated by the regression tree are added. Both sets of results are compared based on the percentage deviation from the optimal objective value within the same mean solution time. The percentage deviation (*perc_dev*) from the optimal objective value is calculated using the following equation:

$$perc_dev = \left(\frac{Obj_{alg} - Obj_{opt}}{Obj_{opt}} \right) \times 100\% \quad (4.7)$$

where Obj_{alg} is the objective value obtained from the algorithm and Obj_{opt} is the optimal objective value. The higher the percentage deviation from the optimal solution, the worse the performance.

The “AL_AllPop_3Stop” scenario involves active learning-based population management, where the entire population is used to generate regression tree rule sets. The process includes a stopping condition that halts iterations after three successive iterations without improvement.

The comparison of the mean percentage deviation (%) in five replications against the percentage deviation obtained by running Gurobi with the mean runtime of these replications is presented in Figure 4.5. This comparison is provided for each selected QPLIB benchmark instance, illustrating the performance and efficiency of the AL_AllPop_3Stop scenario.

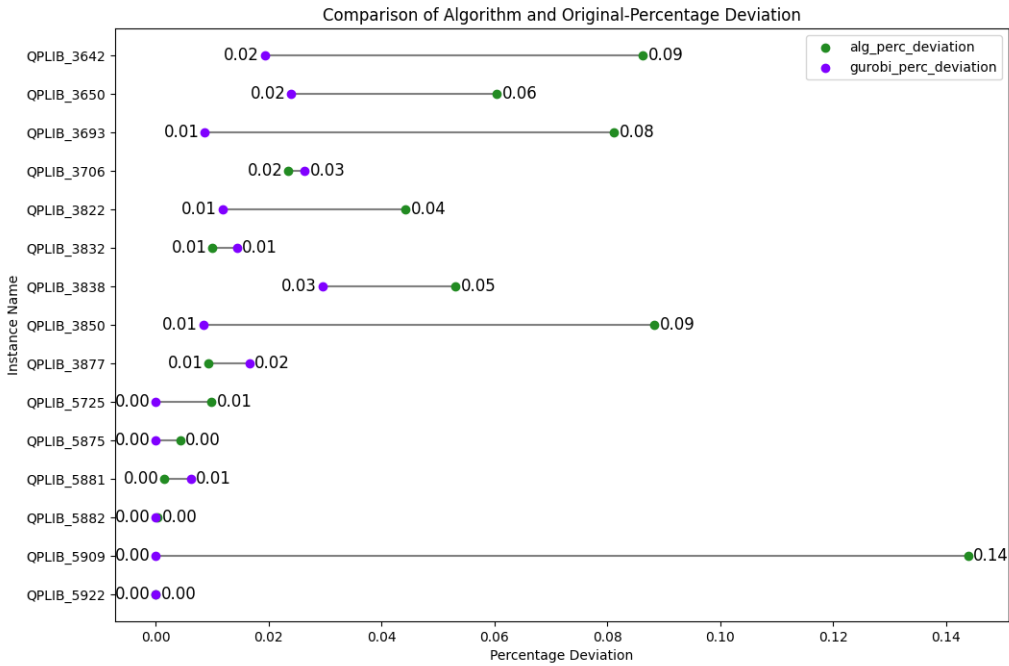


Figure 4.5. The comparison with Gurobi for scenario AL_AllPop_3Stop.

Based on the results presented in Figure 4.5 and Table B.2, we observe the comparison between our algorithm-generated constraints and Gurobi’s performance across various QPLIB benchmark instances. The figure highlights the percentage deviation from the optimal solution, with higher values indicating worse performance. Gurobi generally outperforms our algorithm, except in a few specific instances.

Table B.2 provides additional metrics, including the size of the set of common solutions with the optimal solution (`com_soln_size`) and the number of variables fixed by the regression tree (`num_fixed_by_tree`). The “`tree_matching_count_opt`” column indicates how many of the fixed variables match the optimal solution, and “`match_ratio_tree`” is the ratio of these matches to the total number fixed by the tree.

For instance, QPLIB_3642, with 1035 variables and a common solution size of 548.8 (53% of the total), shows our algorithm fixing 94 variables, 51.2 of which match the optimal solution, resulting in a match ratio of 54.5%. The percentage deviation from the optimal solution for our algorithm is 8.63%, compared to Gurobi’s 1.93%. Conversely, the “`comparison`” column indicates that instances like QPLIB_3706 and QPLIB_3832 show our algorithm performing better.

Significantly, instances such as QPLIB_5882 and QPLIB_5922 achieve a 100% match rate, meaning all variables fixed by the tree perfectly align with the optimal solution. This indicates a highly effective constraint generation in these cases. The overall statistics reveal that the mean common solution size ratio across all instances is 65.75%, with an average deviation of 4.11%. Our algorithm’s constraints fixed, on average, 65% of the variables and a mean Gurobi deviation of 1.10%.

The “AL_HalfPop_3Stop” scenario employs active learning-based population management, utilizing only half the population to generate regression tree rule sets. Iterations cease after three consecutive iterations without improvement.

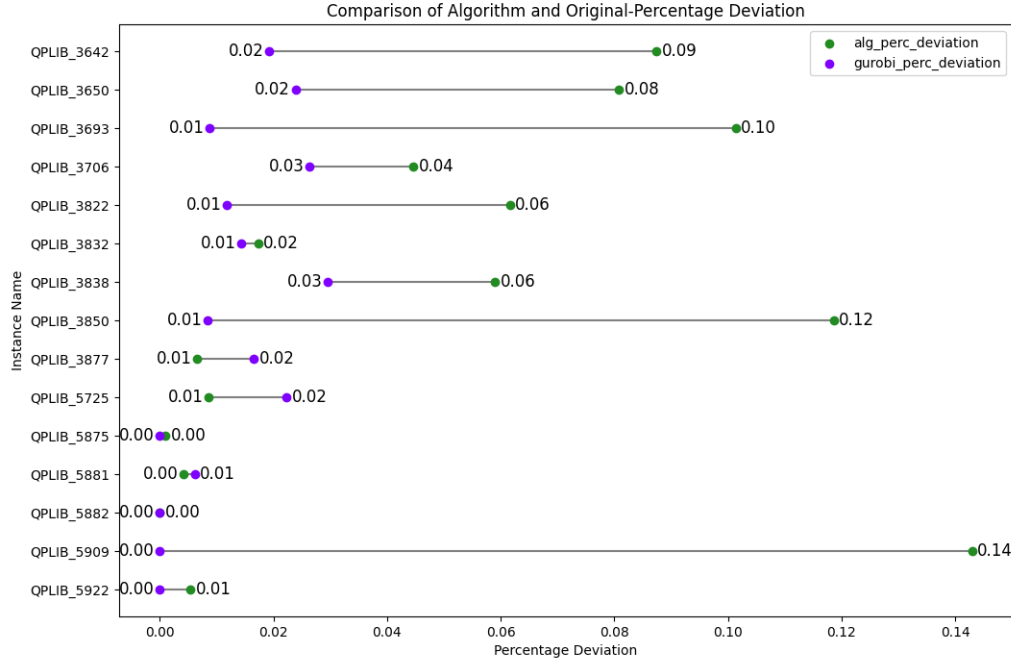


Figure 4.6. The comparison with Gurobi for scenario AL_HalfPop_3Stop.

Based on the results presented in Figure 4.6 and Table B.3, we analyze a detailed comparison between our algorithm-generated constraints and Gurobi's performance across various QPLIB benchmark instances. Gurobi generally outperforms our algorithm, except in a few specific instances. For example, QPLIB_3642, with a common solution size of 548 (53% of the total), shows our algorithm fixing 82 variables, 47.2 of which match the optimal solution, resulting in a match ratio of 58%. The percentage deviation from the optimal solution for our algorithm is 8.7%, compared to Gurobi's 1.9%. Conversely, instances like QPLIB_3877 and QPLIB_5725 show our algorithm performing better. This is evident in Figure 4.6, where the green dots (representing our algorithm) are positioned further to the left, indicating lower percentage deviations compared to Gurobi, as reflected in the `comparison` column.

Notably, instances such as QPLIB_5875, QPLIB_5881, and QPLIB_5882 achieve high match rates of 98%, 96%, and 96%, respectively, meaning a significant proportion of variables fixed by the tree align with the optimal solution. This indicates highly effective constraint generation in these cases. The overall statistics reveal that the mean

common solution size ratio across all instances is 65.41%, with an average deviation of 4.9%. Our algorithm’s constraints fixed, on average, 66% of the variables. The mean of the Gurobi deviation is 1.3%.

Comparing the two scenarios, “AL_HalfPop_3Stop” and “AL_AllPop_3Stop”, it can be seen that instances with a higher number of variables tend to reach better results under the “AL_AllPop_3Stop” scenario. Since this scenario uses the entire population to generate regression tree rule sets in the iterations of the rule set generation, it utilizes more data to potentially improve the solution quality. In contrast, “AL_HalfPop_3Stop,” which uses only half of the population, shows competitive results but generally performs slightly worse in terms of percentage deviation and match ratio. This comparison highlights the importance of utilizing a larger population for generating regression trees, especially for instances with a higher number of variables.

The “WL_AllPop_3Stop” scenario involves population management based on rule sets generated using the worst leaf node path split decisions, where the entire population is utilized to generate regression tree rule sets. The process includes a stopping condition that halts iterations after three successive iterations without improvement.

For the “WL_AllPop_3Stop” scenario, the percentage deviation is calculated as defined in Equation 4.7. A comparison of the constraints generated by our algorithm and Gurobi’s performance across various QPLIB benchmark instances is presented in Figure 4.7. Additionally, Table B.4 summarizes key metrics obtained from our algorithm, as previously discussed. For instance, in QPLIB_3642, which has a common solution size of 566 (constituting 55% of the total), our algorithm fixes 104.4 variables on average, 60.6 of which align with the optimal solution, yielding a match ratio of 58%. The percentage deviation from the optimal solution is 7.27% for our algorithm, compared to 1.93% for Gurobi. Conversely, for instances such as QPLIB_3706 and QPLIB_3877, our algorithm demonstrates superior performance. As depicted in Figure 4.7, green dots representing our algorithm are positioned further to the left, indicating lower percentage deviations relative to Gurobi, as reflected in the `comparison` column.

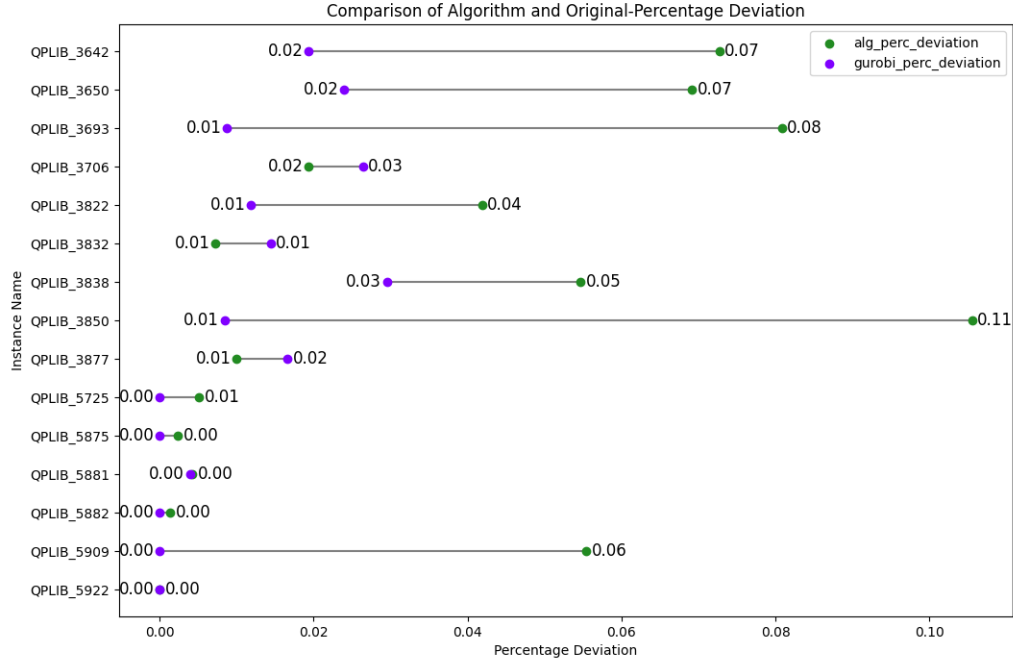


Figure 4.7. The comparison with Gurobi for scenario WL_AllPop_3Stop.

The instances such as QPLIB_5875, QPLIB_5881, and QPLIB_5882 exhibit high match ratios of 95.5%, 94.8%, and 92.5%, respectively. This suggests that a substantial proportion of the variables fixed by the regression tree align with the optimal solution, highlighting the effectiveness of the constraint generation process. Overall, the mean common solution size ratio across all instances is 64.73%, with an average deviation of 3.53%. On average, our algorithm’s constraints fix approximately 65% of the variables, while the mean deviation from the optimal solution using Gurobi remains low at 1.09%.

The “WL_HalfPop_3Stop” scenario involves population management based on rule sets generated using the worst leaf node path split decisions, where instead of using the entire population, half of the population is utilized to generate regression tree rule sets. The process includes a stopping condition that halts iterations after three successive iterations without improvement.

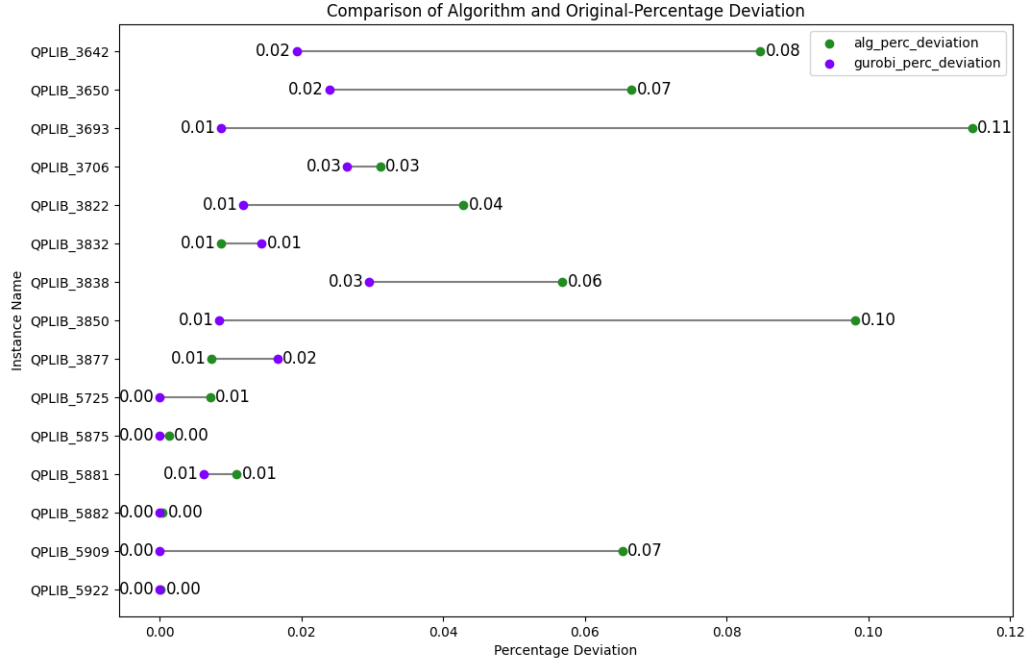


Figure 4.8. The comparison with Gurobi for scenario WL_HalfPop_3Stop.

Based on the percentage deviation values and comparison of the scenario called “WL_HalfPop_3Stop”, presented in Figure 4.8 and the evaluation metrics provided in Table B.5, generally, Gurobi outperforms our algorithm. Still, our algorithm shows competitive results in specific instances. For example, QPLIB_3642, with a common solution size of 586.8 (56.7% of the total), shows our algorithm fixing 93.8 variables, 46.4 of which match the optimal solution, resulting in a match ratio of 49.5%. The percentage deviation from the optimal solution for our algorithm is 8.5%, compared to Gurobi’s 1.9%. Conversely, instances like QPLIB_3706 and QPLIB_3877 show our algorithm performing better. This is evident in Figure 4.8, where the green dots (representing our algorithm) are positioned further to the left, indicating lower percentage deviations compared to Gurobi, as reflected in the `comparison` column.

Notably, instances such as QPLIB_5875, QPLIB_5882, and QPLIB_5922 achieve high match ratios of 96.5%, 94.2%, and 98.2% respectively, meaning a significant proportion of variables fixed by the tree align with the optimal solution. This indicates highly effective constraint generation in these cases. The overall statistics reveal that

the mean common solution size ratio across all instances is 65.7%, with an average deviation of 4.0%. Our algorithm’s constraints fixed, on average, 65.9% of the variables, with a mean Gurobi deviation of 1.10%.

Comparing the four scenarios reveals distinct differences in performance and effectiveness. The “AL_AllPop_3Stop” scenario, which uses the entire population for generating regression tree rule sets with a stopping condition of three successive iterations without improvement, outperforms Gurobi in 5 instances, demonstrating its robustness and efficiency, particularly for larger problem sizes. In the “AL_HalfPop_3Stop” scenario, where only half of the population is used, our algorithm still performs competitively, outperforming Gurobi in 4 instances, indicating that the active learning approach remains effective even with reduced data. The “WL_AllPop_3Stop” scenario, utilizing the entire population with the worst leaf node path split decisions, shows our algorithm outperforming Gurobi in 3 instances, suggesting that this method also benefits from using the full population but is slightly less effective compared to the active learning approach. Lastly, the “WL_HalfPop_3Stop” scenario, using half of the population and the worst leaf node path split decisions, outperforms Gurobi in 2 instances, highlighting the reduced effectiveness when both the population size and decision criteria are constrained. Overall, these comparisons underscore the importance of using the entire population and adopting effective decision criteria for generating high-quality regression tree rule sets, with “AL_AllPop_3Stop” showing the most consistent performance across different instances.

Despite Gurobi’s superior performance in most cases, the scenarios where our algorithm excels underscore its potential effectiveness, especially in instances with higher match ratios. This analysis highlights the intricate balance between variable fixation and solution quality in the context of QPLIB benchmarks.

5. LEARNING BASED BOUNDS GENERATING ALGORITHM

In this section, we first intend to present the general scheme of our suggested *RegTree-Bound* algorithm and then provide its underlying components.

5.1. General Scheme

We initiate the process by generating random solutions and constructing a regression tree. This tree imparts valuable insights into the rules leading to favorable solutions and those associated with solutions deviating significantly from the optimal value. Hence, the set of rules along the path leading to the best leaf node provides a valuable framework for establishing effective bounds on the solution space. This systematic approach aims to narrow the solution space and enhance the exploitation of potential global optima within the specified bounds. As described in Section 1, our framework uses a methodology to identify high-quality and low-quality solutions and uses this information to understand why certain decisions lead to high-quality solutions (inbound generation).

To elaborate further on constructing the regression tree with distinct features, we consider two scenarios: one incorporating decision variables as features and the other involving the linear regression prediction of objective values as a feature. In both instances, a single-split regression tree is generated. We exclusively focus on the best leaf node solutions to visually represent the segmentation of the solution space through the tree's splits. We isolate the best leaf node solutions in each iteration and generate a new regression tree using this refined subset. This iterative approach emphasizes the regression tree's ability to consistently partition solutions around the global optimum in each successive iteration, showcasing its efficacy in refining the understanding of the solution space.

Additionally, we adopt a strategic approach by focusing on peak-valley leaf nodes to generate new solutions during the population update step. In the context of Response Surface Methodology (RSM), peak-valley analysis involves identifying the maximum (peaks) and minimum (valleys) points on the response surface [60].

- Peaks (Maxima): These are points where the response is at a local or global maximum. Identifying peaks is important in processes where the goal is to enhance performance or yield.
- Valleys (Minima): These are points where the response is at a local or global minimum. Identifying valleys is important in processes where the goal is to reduce defects, costs, or other undesirable outcomes.

This is crucial for optimization tasks where the goal might be to maximize or minimize the response. In this scope, we select the leaf node solutions with the highest differentiability according to the objective value distance to solution distance ratio. This approach proves particularly beneficial in regions where the regression tree encounters challenges in effective learning. By identifying and integrating these peak-valley leaf nodes, the algorithm is encouraged to explore and adapt in areas where the regression tree might struggle to capture the underlying patterns or exhibit inadequate performance. This method addresses the limitations of the regression tree by actively adding solutions with significant differentiability, thereby enhancing the exploration of regions with numerous peaks and valleys in continuous functions.

Our algorithm is a versatile approach applicable to any continuous optimization problem where solutions are represented in a continuous (real-valued) framework. It is particularly effective for navigating challenging problem landscapes, such as those encountered in the BBOB (Black-Box Optimization Benchmarking) functions, which often lack derivative information, making gradient-based methods ineffective. The algorithm ensures feasibility without predefined constraints by leveraging bounds on decision variables. Split decisions, derived from the regression tree, act as limitations on the features (i.e., decision variables) and are subsequently employed as bounds

during the solution process in exact solution algorithms. This use of split decisions as bounds enhances the algorithm's efficacy in exploring and refining the continuous solution space, making it a robust method for addressing complex optimization tasks.

5.1.1. Preserving Feasibility

Feasibility is preserved through a systematic process of solution generation and refinement:

- (i) **Initial Solution Population:** The algorithm begins by generating an initial population of solutions, ensuring that all solutions are within the predefined bounds of the variables.
- (ii) **Regression Tree Construction:** A regression tree is constructed based on the generated population. This tree partitions the solution space by identifying regions associated with better-performing solutions, using their objective values as a guide.
- (iii) **Tighter Bounds Generation:** The bounds derived from the regression tree are used to tighten the variable bounds dynamically. These refined bounds remain within the original feasible range, ensuring no violation of feasibility.
- (iv) **Latin Hypercube Sampling:** During the *Population Management Phase*, the refined bounds are utilized to generate new solutions through Latin Hypercube Sampling (LHS). This process ensures that the new solutions are well-distributed within the updated bounds and remain feasible.

The algorithm maintains feasibility throughout the optimization process by iteratively refining bounds and generating new solutions, even for problems with complex and unconstrained solution spaces.

5.1.2. Scope of Problems Solved

The Variable Bound Generation Algorithm is capable of solving any type of objective function with a continuous solution space, provided that the function allows exploration through variable bounds. This includes, but is not limited to:

- Functions with Rugged, Multimodal Landscapes: Problems where traditional optimization techniques may struggle due to multiple local optima.
- Derivative-Free Problems: Problems where derivative information is unavailable or computationally prohibitive.
- General Continuous Optimization Tasks: The algorithm's generality makes it applicable to a broad range of objective functions beyond specialized test cases.

Considering the algorithm's computational complexity is crucial when selecting it for specific problems. For simpler objective functions, where direct or less complex methods suffice, the Variable Bound Generation Algorithm may introduce unnecessary overhead. It is most effective for complex problems lacking derivative information, where other methods are less suitable. Balancing capability and complexity ensures that the algorithm effectively addresses challenging problems without excessive use of computational resources.

5.2. System Architecture

Regarding system architecture, our proposed algorithm strategically manages a population comprising both high-quality and low-quality solutions, primarily for tree-based mining. Furthermore, it extracts valuable insights from the high-quality solution bounds to enhance the overall performance of the exact solution algorithm. Characteristically, it is composed of five components: 1) a solution representation procedure (Section 5.2.1); 2) a population initialization and a regression tree construction procedure (Section 5.2.2); 3) bound generation procedure (Section 5.2.3); and 4) population update procedure (Section 5.2.4).

5.2.1. Solution Representation

The solution representation is illustrated in Table 5.1. Our algorithm adopts a continuous(real-valued) solution representation, where the decision variables are expressed as continuous values. The specific decision variable is denoted as y_i .

Table 5.1. Solution Representation of a Solution

1	2	3	.	.	.	21	22	.	.	.	$n-1$	n
0.55	0.72	0.41	.	.	.	0.91	0.18	.	.	.	0.67	0.89

legend:

$i \leftarrow decision$

y_i

lower bound: 0

upper bound: 1

5.2.2. Population Initialization and Regression Tree Construction

The *RegTree-Bound* algorithm begins by generating a population of randomly generated solutions, denoted as Pop , with a size of $PopSize$. The solution representation is shown in Table 5.1.

Then, a regression tree is constructed using the population of solutions as the data set, where the objective function values of the solutions are the dependent variables (also known as the output attributes or responses) and the binary decision variables are the independent variables (also known as the features, input attributes, or predictors). The visualization of this process is shown in Figure 5.1.

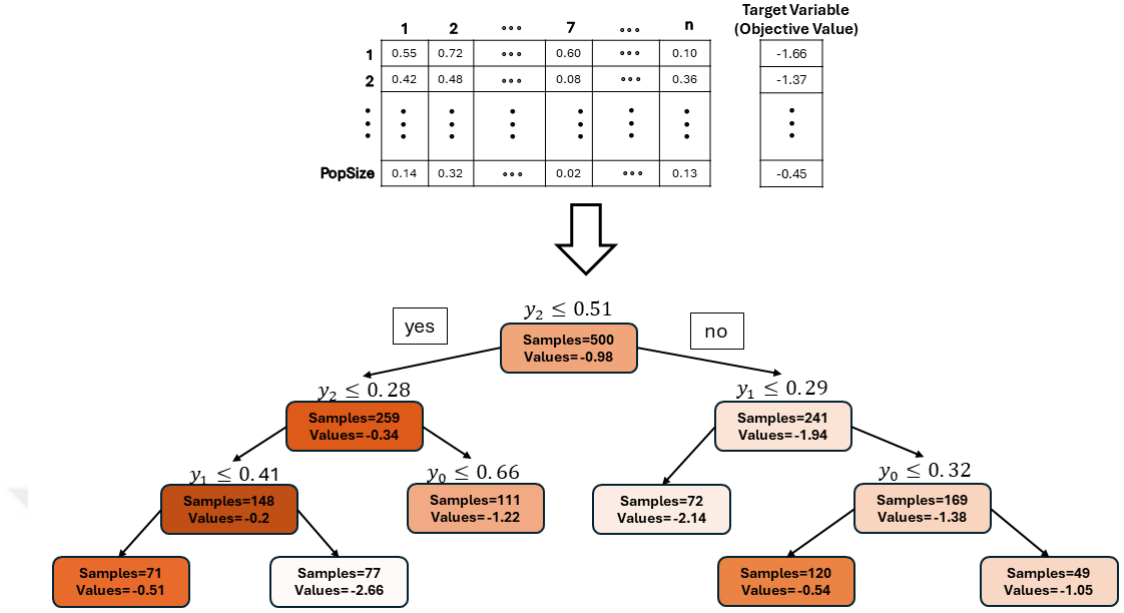


Figure 5.1. Regression Tree Example.

5.2.3. Bound Generation Procedure

Solvers addressing nonlinear problems often face challenges in efficiently identifying optimal solutions, particularly in landscapes with numerous local minima. To mitigate this, we apply tree-based classification methods to refine variable bounds, effectively reducing the solution space. Specifically, we utilize rules derived from a regression tree to set bounds for decision variables during the nonlinear problem-solving process. This approach aims to enhance the efficiency of obtaining high-quality solutions.

We extract split decisions that guide us to the *best* leaf nodes or *peak-valley* and utilize these as bounds on variables. This strategic approach is based on the premise that these splits can serve as effective bounds for achieving high-quality solutions. By narrowing the lower and upper bounds through the inclusion of split decisions along a particular path, we aim to simplify the Nonlinear Optimization (NLOpt) process, making it more conducive to finding solutions efficiently.

In this step of the algorithm, the best leaf node or the most peak-valley leaf node are found. For an optimization problem with a minimization objective, the best leaf node ℓ_{BL} of the regression tree is defined to be the one with the minimum average objective value, i.e., $\ell_{\text{BL}} = \arg \min_{\ell \in \mathcal{L}} \bar{o}_\ell$. It has to be noted that the regression tree is built to effectively differentiate between good and bad solutions, where good solutions are likely (but not necessarily) found in the best leaf node. Hence, the region of the solution space represented by the best leaf node is a promising region that is expected to contain, on average, better solutions that need to be exploited. However, this does not mean that the best solution must exist in the best leaf node ℓ_{BL} . In this step, not only the best leaf node is determined, but also the so-called most *peak-valley* node ℓ_{PV} . This node exhibits the highest number of distinct peak and valley points in the landscape of the objective function. The peaks represent local maxima where the objective function reaches its highest values relative to neighboring points, whereas the valleys denote local minima where the function attains its lowest values. This definition is grounded in the response surface methodology, in which the shape of the response surface reflects the extremum points of the function [60].

To better illustrate this process, Figure 5.2 presents two visualizations showing how the regression tree partitions the solution space during the same iteration for the same objective function. Both figures share identical decision boundaries as they are derived from the same tree. However, the color encoding differs between the two plots to emphasize different aspects of the analysis. In the figure on the left, the color gradient represents variations in the objective function values, providing a direct view of the fitness landscape. Regions with higher color diversity indicate greater variability in the objective function, reflecting areas with more dynamic changes in solution quality. The samples of solutions are shown as scattered points in space. The areas highlighted by red circles correspond to the most *peak-valley* leaf nodes, highlighting regions with significant internal fluctuations in objective values. In the figure on the right, each leaf node is numbered to indicate its order based on its average objective value, where 1 denotes the best leaf node, the partition with the lowest average objective value. This numbering helps identify which partition contains,

on average, the most promising solutions. In addition to numbering, the color of each partition represents its *peak-valley* (PV) score, which measures the internal variability of objective values within that region. Higher PV scores (warmer colors) highlight leaf nodes with more diverse solutions, indicating areas with high exploration potential. Lower PV scores (cooler colors) indicate more homogeneous regions, which are more likely to be locally converged. In both figures, the yellow star denotes the best-known solution at this stage. This dual visualization — objective function landscape on the left and peak-valley analysis with leaf ordering on the right — provides complementary insights. The left panel identifies the globally promising regions, while the right panel highlights both the leaf node ranking and the regions where further exploration could be beneficial due to high internal variability.

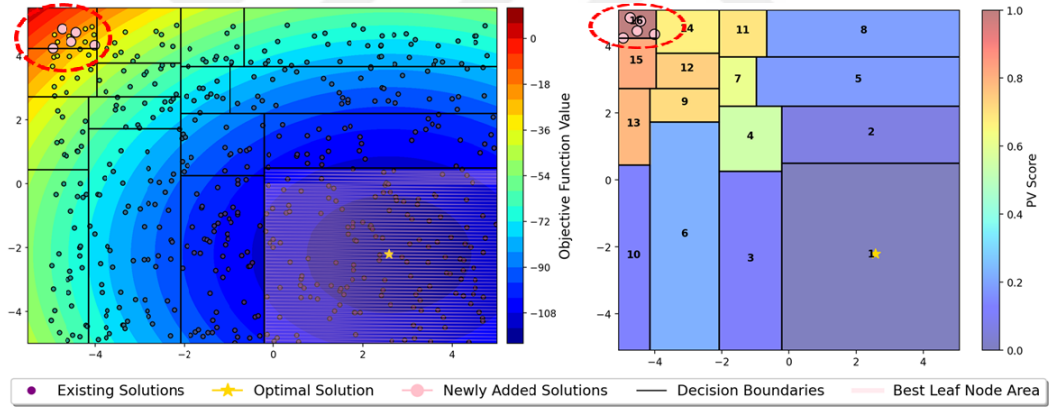


Figure 5.2. Decision boundaries, leaf node ranking, and peak-valley scores from regression tree partitioning.

Two methods are used to assess the variability in the leaf nodes to identify them as the peak-valley nodes. The first method relies on pairwise variability calculations, measuring how solutions within the same leaf node differ from each other. To formally define pairwise variability within leaf nodes, we introduce the following formulations.

First, the pairwise Euclidean distance between objective values within the same leaf node is computed as

$$D_{\text{obj}}[i][j] = \sqrt{\sum_{k=1}^{\text{len}(O_l)} (o_i[k] - o_j[k])^2}, \quad o_i \in O_l, \quad o_j \in O_l, \quad \forall l \in L. \quad (5.1)$$

Similarly, the Euclidean distance between solutions within the same leaf node is calculated as

$$D_{\text{soln}}[i][j] = \sqrt{\sum_{k=1}^{\text{len}(x_i)} (x_i[k] - x_j[k])^2}, \quad x_i \in S_l, \quad x_j \in S_l, \quad \forall l \in L. \quad (5.2)$$

The average pairwise distance within each leaf node is then computed as

$$d_{\text{obj}} = \frac{\sum D_{\text{obj}}}{\text{len}(D_{\text{obj}})} \quad (5.3)$$

$$d_{\text{soln}} = \frac{\sum D_{\text{soln}}}{\text{len}(D_{\text{soln}})}. \quad (5.4)$$

To quantify the relative variability of the objective values compared to the solution distances within a leaf node, we define:

$$r_l = \frac{d_{\text{obj}}}{d_{\text{soln}}}. \quad (5.5)$$

Finally, the most peak-valley leaf node is identified as the leaf node with the highest relative variability. The equation can be written as

$$l_{PV} = \arg \max_l r_l. \quad (5.6)$$

These equations (5.1)-(5.6) provide a systematic pairwise-based approach for detecting peak-valley nodes, ensuring that regions with significant internal variability are effectively identified.

The second method adopted the medoid-based approach, where a representative central solution is identified to represent the leaf node, and the distance between each solution in the node. While the pairwise distance-based method computes variability using all pairwise distances, the medoid-based approach provides an alternative by measuring distances relative to the most central solution within each leaf node.

To define this approach, we first determine the medoid of each leaf node. The medoid is the solution that minimizes the sum of distances to all other solutions within the leaf. It can be defined as

$$x_{\text{medoid}} = \arg \min_{x_i \in S_l} \sum_{x_j \in S_l} |x_i - x_j|. \quad (5.7)$$

Similarly, the medoid of objective values in a leaf node is defined as

$$o_{\text{medoid}} = \arg \min_{o_i \in O_l} \sum_{o_j \in O_l} |o_i - o_j|. \quad (5.8)$$

Using the medoid as the reference point, we compute the average distance to the medoid for both solutions and objective values as

$$d_{\text{obj,med}} = \frac{1}{|O_l|} \sum_{o_i \in O_l} |o_i - o_{\text{medoid}}| \quad (5.9)$$

$$d_{\text{soln,med}} = \frac{1}{|S_l|} \sum_{x_i \in S_l} |x_i - x_{\text{medoid}}|. \quad (5.10)$$

To measure relative variability, we define the medoid-based variability ratio as follows

$$r_{l,\text{med}} = \frac{d_{\text{obj,med}}}{d_{\text{soln,med}}}. \quad (5.11)$$

Finally, the most peak-valley leaf node is identified as the leaf node with the highest medoid-based variability. It can be defined as

$$l_{PV} = \arg \max_l r_{l,\text{med}}. \quad (5.12)$$

These equations (5.7)-(5.12) define the medoid-based approach for identifying peak-valley nodes. Unlike the pairwise distance method, which evaluates variability using all pairwise comparisons, this approach uses the medoid as a central reference, reducing the impact of extreme values and emphasizing the most representative variability within each leaf node.

The best leaf node ℓ_{BL} is used for the first objective and the peak-valley node ℓ_{PV} for the second, where the selection of which leaf node to use is made in a probabilistic way as follows. The peak-valley node is selected with exploration probability p_e while the best leaf node is selected with probability $1 - p_e$. New solutions are generated on the basis of the bounds provided by the path leading to the corresponding node. The region identified by the selected leaf node, whether it is ℓ_{BL} or ℓ_{PV} , can be designated by the splits at each node along the path from the root to the leaf node. These splits, which are denoted by P_{ℓ^*} , define conditions on a subset of decision variables $\{y_1, y_2, \dots, y_d\}$.

The bounds of these decision variables are updated, while variables that do not exist in the splits retain their original bounds. As provided in Figure 5.3, if the path to the best leaf node has the splits $P_{\ell^*} = \{y_2 \leq 0.51, y_2 \leq 0.28, y_1 > 0.41\}$, then the bounds of y_1 and y_2 are updated as $0.41 < y_1 \leq 1.00$ and $0.00 \leq y_2 \leq 0.28$.

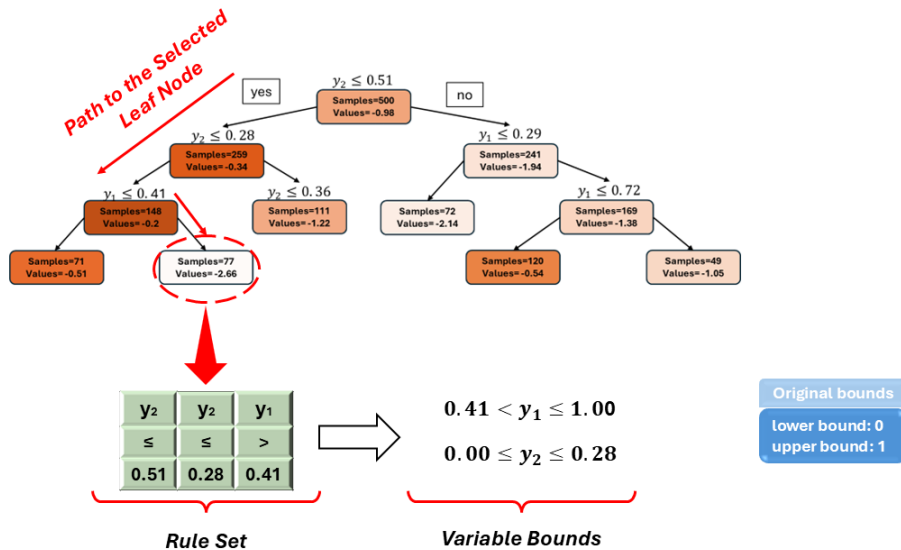


Figure 5.3. Bound Generation Example.

5.2.4. Population Update Procedure

In our population management strategy, the new solutions are generated within the bounds using the bound generation procedure outlined in Section 5.2.3. The split decisions in the path leading to the most peak-valley leaf node or the best leaf node are used to generate bounds on variables. These bounds are then employed for the Latin Hypercube sampling method to generate new solutions as seen in Figure 5.4.

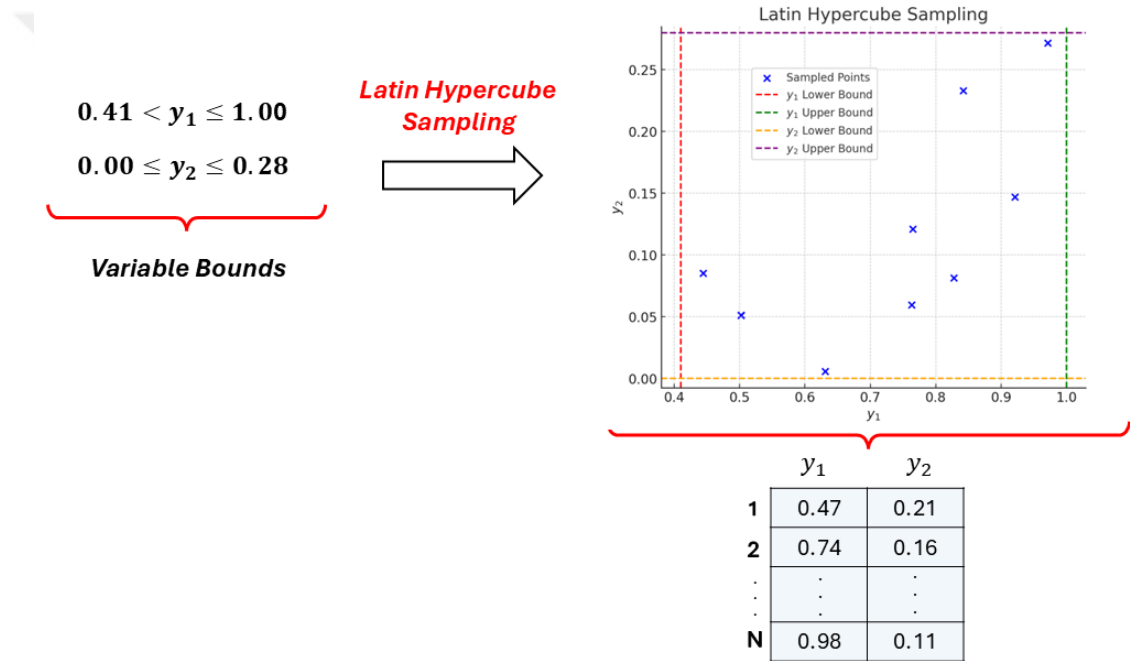


Figure 5.4. New Solutions Generation.

After generating new solutions, the next step is updating the population of solutions that make up the dataset. This is achieved by modifying the dataset used to train the regression tree by adding the new solutions generated using bounds derived from the regression tree. The split decisions in the regression tree are governed by the underlying tree-building parameters, in particular, the minimum size for a leaf node controlled by the parameter *min_samples_leaf*. The splitting process is automatically managed by the tree-building algorithm, which evaluates possible splits and ensures that no split results in a leaf node with fewer than *min_samples_leaf*. It should be

emphasized that the population update is carried out with the primary goal of maintaining diversity in the population. This is accomplished in such a way that each new solution in Ω_{new} that is randomly generated in the previous step using the bounds provided by the best leaf node or the peak-valley leaf node, is evaluated to ensure that it is not too similar to existing solutions in the population. In other words, new solutions are ensured to be different from existing solutions. To this end, the Euclidean distance of a new solution in Ω_{new} is calculated for each existing solution in the corresponding leaf node, and the minimum distance of every solution is determined. Among the new solutions, the one with the largest minimum distance is selected to be added to the population. This process continues until half of the newly generated solutions become part of the population.

5.3. Pseudo-Code for Algorithms

This section presents the pseudo-code for the algorithms used in our applications. The pseudo-code provides a detailed step-by-step description of the methods implemented in our research, making it easier to understand and replicate the results.

The Regression Tree Processing Algorithm (16) analyzes a regression tree $\mathcal{T}$ and its population of solutions $\mathcal{P}$ with objective values $\mathcal{O}$ to identify two specific leaf nodes: the Best Leaf Node (ℓ_{BL}), where solutions have the lowest average objective value, and the Most Peak-Valley Leaf Node (ℓ_{PV}), which exhibits the highest variability in objective values among its solutions. The algorithm initializes by retrieving all leaf nodes of $\mathcal{T}$ into $\mathcal{L}$ and setting ℓ_{BL} and ℓ_{PV} to *None*. It also initializes $\bar{\mathcal{O}}^{\text{min}}$ to infinity to track the minimum average objective value. For each leaf node ℓ in $\mathcal{L}$, it extracts the subset of $\mathcal{P}$ that falls into ℓ and their corresponding objective values $\mathcal{O}_\ell$. If the mean of $\mathcal{O}_\ell$ is less than $\bar{\mathcal{O}}^{\text{min}}$, it updates $\bar{\mathcal{O}}^{\text{min}}$ and sets ℓ_{BL} to ℓ . After evaluating all leaf nodes, the algorithm determines ℓ_{PV} by invoking the *most_peak_valley_leaf_node* function (Algorithm 19). Finally, it returns ℓ_{BL} and ℓ_{PV} .

Algorithm 16 Regression Tree Processing Algorithm

1: **Input:**

- Regression tree $\mathcal{T}$
- Population of solutions $\mathcal{P}$
- Objective values of the population $\mathcal{O}$

2: **Output:**

- Best leaf node ℓ_{BL}
- Most peak-valley leaf node ℓ_{PV}

3: Initialize $\mathcal{L} \leftarrow \text{get_leaf_nodes}(\mathcal{T})$

4: Initialize $\ell_{\text{BL}}, \ell_{\text{PV}} \leftarrow \text{None}$, $\text{min_avg_obj} \leftarrow \infty$

5: **for** each leaf node $\ell \in \mathcal{L}$ **do**

6: Extract population $\mathcal{P}_\ell \leftarrow \mathcal{P}[\mathcal{T}(\mathcal{P}) == \ell]$

7: Extract objective values $\mathcal{O}_\ell \leftarrow \mathcal{O}[\mathcal{T}(\mathcal{P}) == \ell]$

8: **if** $\text{mean}(\mathcal{O}_\ell) < \bar{\mathcal{O}}^{\min}$ **then**

9: $\bar{\mathcal{O}}^{\min} \leftarrow \text{mean}(\mathcal{O}_\ell)$

10: $\ell_{\text{BL}} \leftarrow \ell$

11: **end if**

12: **end for**

13: $\ell_{\text{PV}} \leftarrow \text{most_peak_valley_leaf_node}(\mathcal{L}, \mathcal{P}, \mathcal{O})$ (Algorithm 19)

14: **Return** $\ell_{\text{BL}}, \ell_{\text{PV}}$

The algorithm 17 takes a regression tree and a leaf node as input, returning a set of rules derived from the split decisions leading to the specified leaf node. The process begins by initializing an empty set of rules, denoted as $\mathcal{R}$. For each node encountered on the path from the root to the leaf node, the feature and the value used for the split at that node are added to $\mathcal{R}$. Upon reaching the leaf node, the complete set of rules is returned. This set encapsulates the conditions that define the path from the root to the chosen leaf node.

Algorithm 17 Rule Extraction Algorithm

1: **Input:**

- Regression tree $\mathcal{T}$ fitted on population $\mathcal{P}$
- Leaf node type ℓ_{type} (Best, PeakValley)

2: **Output:**

- Rule set $\mathcal{R}$ derived from the specified leaf node type

3: Initialize an empty rule set $\mathcal{R}$

4: Generate Rules from Specified Leaf Node:

5: $\ell_{\text{nodes}} \leftarrow$ leaf nodes of type ℓ_{type} in $\mathcal{T}$

6: **for** each variable V_j in the path to ℓ_{nodes} **do**

7: **if** $V_j < \text{Value}$ in the path **then**

8: Add rule $V_j < \text{Value}$ to $\mathcal{R}$

9: **else**

10: Add rule $V_j \geq \text{Value}$ to $\mathcal{R}$

11: **end if**

12: **end for**

13: **Return** $\mathcal{R}$

The Algorithm 18 is designed to generate bounds on decision variables by utilizing the rule set obtained from the algorithm 17. The algorithm applies the rules derived from the path of the regression tree to generate upper and lower bounds on decision variables. It begins by initializing empty lists for storing the lower and upper bounds. The algorithm iterates through each condition in the rule set for each variable in the set of decision variables. For each condition, it extracts the feature, operator, and value. If the feature in the solution matches the condition, the algorithm updates the bounds: if the operator is $\leq$, the upper bound is set to the minimum of the current upper bound and the condition value; if the operator is $>$, the lower bound is set to the maximum of the current lower bound and the condition value. The bounds are set to their original values if no match is found. After processing all conditions for a variable, the updated bounds are appended to the respective lists. Finally, the algorithm returns the decision variables' lower and upper bounds lists.

Algorithm 18 Bound Generation Algorithm

1: **Input:**

- Rule set $\mathcal{R}$ derived from the Rule Extraction Algorithm
- Population of solutions $\mathcal{P}$
- Original lower bound of feature, *lower_bound*
- Original upper bound of feature, *upper_bound*

2: **Output:**

- *Lower_Bounds* for variables $\mathcal{V}$
- *Upper_Bounds* for variables $\mathcal{V}$

3: Initialize *Lower_Bounds* $\leftarrow \{\}$

4: Initialize *Upper_Bounds* $\leftarrow \{\}$

5: **for** each variable V_k in $\mathcal{V}$ **do**

6: **for** each condition $Cond_j$ in $\mathcal{R}$ **do**

7: Extract feature V_k , operator *Operator*, and value *Value* from $Cond_j$

8: **if** V_k in s matches $Cond_j$ **then**

9: **if** *Operator* = $\leq$ **then**

10: $upper_bound(V_k) \leftarrow \min(upper_bound(V_k), Value)$

11: **else if** *Operator* = $>$ **then**

12: $lower_bound(V_k) \leftarrow \max(lower_bound(V_k), Value)$

13: **end if**

14: **else**

15: $lower_bound(V_k) \leftarrow$ original lower bound of V_k

16: $upper_bound(V_k) \leftarrow$ original upper bound of V_k

17: **end if**

18: **end for**

19: Append $lower_bound(V_k)$ to *Lower_Bounds*

20: Append $upper_bound(V_k)$ to *Upper_Bounds*

21: **end for**

22: **Return** *Lower_Bounds*, *Upper_Bounds*

Algorithm 19 Most Peak-Valley Leaf Node Detection Algorithm

1: **Input:**

- Set of leaf node indexes $\mathcal{L}$
- Population of solutions $\mathcal{P}$ and objective values of the population $\mathcal{O}$
- Leaf node IDs corresponding to each solution in the population ℓ_P

2: **Output:**

- Leaf node ID with the highest peak-valley ratio ℓ_{PV}

3: Initialize $division_results \leftarrow \{\}$

4: **for** each leaf node index $\ell_i \in \mathcal{L}$ **do**

5: $\mathcal{O}_{\ell_i} \leftarrow \mathcal{O}[\ell_P == \ell_i]$ and $\mathcal{P}_{\ell_i} \leftarrow \mathcal{P}[\ell_P == \ell_i]$

6: Initialize $D_{obj} \leftarrow$ empty matrix of size $(\text{len}(\mathcal{O}_{\ell_i}), \text{len}(\mathcal{O}_{\ell_i}))$

7: **for** each $o_i \in \mathcal{O}_{\ell_i}$ **do**

8: **for** each $o_j \in \mathcal{O}_{\ell_i}$ **do**

9: $D_{obj}[i][j] \leftarrow \sqrt{\sum_{k=1}^{\text{len}(o_i)} (o_i[k] - o_j[k])^2}$

10: **end for**

11: **end for**

12: $d_{obj} \leftarrow \frac{\sum D_{obj}}{\text{len}(D_{obj})}$

13: Initialize $D_{sol} \leftarrow$ empty matrix of size $(\text{len}(\mathcal{P}_{\ell_i}), \text{len}(\mathcal{P}_{\ell_i}))$

14: **for** each $x_i \in \mathcal{P}_{\ell_i}$ **do**

15: **for** each $x_j \in \mathcal{P}_{\ell_i}$ **do**

16: $D_{sol}[i][j] \leftarrow \sqrt{\sum_{k=1}^{\text{len}(x_i)} (x_i[k] - x_j[k])^2}$

17: **end for**

18: **end for**

19: $d_{sol} \leftarrow \frac{\sum D_{sol}}{\text{len}(D_{sol})}$

20: $r_{div} \leftarrow \frac{d_{obj}}{d_{sol}}$

21: $division_results \leftarrow division_results \cup \{\ell_i, r_{div}\}$

22: **end for**

23: $division_results_{sorted} \leftarrow \text{sort_values}(by = r_{div})$

24: $\ell_{PV} \leftarrow$ leaf node ID with the highest r_{div} from the sorted $division_results$

25: **Return** ℓ_{PV}

The Most Peak-Valley Leaf Node Detection Algorithm (Algorithm 19) identifies the leaf node with the highest peak-valley ratio within a regression tree. The algorithm starts by taking as input the set of leaf node indexes $\mathcal{L}$, the population of solutions $\mathcal{P}$, the corresponding objective values $\mathcal{O}$, and the leaf node IDs for each solution $\mathcal{L}_P$. For each leaf node ℓ_i in $\mathcal{L}$, it first segregates the objective values $\mathcal{O}_{\ell_i}$ and solutions $\mathcal{P}_{\downarrow}$ belonging to the leaf node. It then computes the pairwise Euclidean distances for the objective values and solutions within that leaf node, storing these distances in matrices D_{obj} and D_{sol} , respectively. The average of these distances, d_{obj} and d_{sol} , is used to calculate the peak-valley ratio $r_{\text{div}} = \frac{d_{\text{obj}}}{d_{\text{sol}}}$. These ratios are stored in a collection and then sorted to identify the leaf node with the highest peak-valley ratio, ℓ_{PV} . The algorithm returns this leaf node ID as the one exhibiting the highest degree of variability in the objective values relative to the solutions, indicating regions of the solution space with significant peaks and valleys. The pseudo-code of Algorithm 19 implements the pairwise-based approach, where variability is assessed using all pairwise Euclidean distances within each leaf node. If the medoid-based approach is selected for peak-valley leaf node detection, then instead of the pairwise calculation equations (5.1)-(5.6), the medoid-based equations (5.7)-(5.12) are applied. This alternative approach determines variability by computing distances relative to the most central solution (medoid) within each leaf node, providing a computationally efficient alternative.

Algorithm 20 Population Update Algorithm

1: **Input:**

- Regression tree $\mathcal{T}$ and the leaf node IDs ℓ_{BL}, ℓ_{PV}
- Exploration probability p_e and number of new solutions n_{new}

2: **Output:**

- Selected Solutions $\mathcal{P}_{\text{selected}}$

3: Initialize $\Omega_{\text{new}} \leftarrow \emptyset$

4: Generate a random number $r \in [0, 1]$

5: **if** $r < p_e$ **then**

6: $\mathcal{R} \leftarrow \text{rule_extract}(\ell_{PV})$ (Algorithm 17)

7: $\mathcal{S}_{\text{leaf}} \leftarrow \text{retrieve_solutions}(\mathcal{T}, \ell_{PV})$ (Algorithm 21)

8: **else**

9: $\mathcal{R} \leftarrow \text{rule_extract}(\ell_{BL})$ (Algorithm 17)

10: $\mathcal{S}_{\text{leaf}} \leftarrow \text{retrieve_solutions}(\mathcal{T}, \ell_{BL})$ (Algorithm 21)

11: **end if**

12: $\mathcal{B} \leftarrow \text{generate_bounds}(\mathcal{R})$ (Algorithm 18)

13: $\Omega_{\text{new}} \leftarrow \text{generate_solutions}(\mathcal{B}, n_{\text{new}})$ (Latin Hypercube Sampling)

14: Initialize $\text{half_}n_{\text{new}} \leftarrow \lfloor \frac{n_{\text{new}}}{2} \rfloor$

15: **while** $\text{len}(\mathcal{P}_{\text{new}}) < \text{len}(\mathcal{P}) + \text{half_}n_{\text{new}}$ **do**

16: Initialize $\mathcal{D}_{\text{min}} \leftarrow \emptyset$

17: **for** each solution $x \in \Omega_{\text{new}}$ **do**

18: $x_dist \leftarrow \text{pairwise_distances}(x, \mathcal{S}_{\text{leaf}})$ (Euclidean)

19: $d_{\text{min}} \leftarrow \min(x_dist)$

20: $\mathcal{D}_{\text{min}} \leftarrow \mathcal{D}_{\text{min}} \cup d_{\text{min}}$

21: **end for**

22: $i = \arg \max(\mathcal{D}_{\text{min}})$

23: $\mathcal{S}_{\text{leaf}} \leftarrow \mathcal{S}_{\text{leaf}} \cup \Omega_{\text{new}}[i]$

24: $\Omega_{\text{new}} \leftarrow \Omega_{\text{new}} \setminus \Omega_{\text{new}}[i]$

25: $\mathcal{P}_{\text{selected}} \leftarrow \mathcal{P}_{\text{selected}} \cup \Omega_{\text{new}}[i]$

26: **end while**

27: **Return** $\mathcal{P}_{\text{selected}}$

Algorithm 21 Retrieve Solutions from Leaf Node

```

1: Input:
    • Regression tree  $\mathcal{T}$ 
    • Leaf node ID  $L_{id}$ 

2: Output:
    • Solutions in leaf node  $\mathcal{S}_{leaf}$ 

    GetLeafNode( $\mathcal{T}, L_{id}$ )

3: Initialize current node  $\leftarrow \mathcal{T}.root$ 
4: while current node is not a leaf node do
5:   if current node.left.id ==  $L_{id}$  then
6:     current node  $\leftarrow$  current node.left
7:   else if current node.right.id ==  $L_{id}$  then
8:     current node  $\leftarrow$  current node.right
9:   else
10:    Return error "Leaf node ID not found"
11:   end if
12: end while
13: Return current node

    RetrieveSolutions( $\mathcal{T}, L_{id}$ )

14: leaf node  $\leftarrow$  GetLeafNode( $\mathcal{T}, L_{id}$ )
15: if leaf node is not a valid leaf node then
16:   Return error "Invalid leaf node"
17: end if
18:  $\mathcal{S}_{leaf} \leftarrow$  leaf node solutions
19: Return  $\mathcal{S}_{leaf}$ 
20: Call RetrieveSolutions( $\mathcal{T}, L_{id}$ )
  
```

The Population Update Algorithm (Algorithm 20) is designed to add newly generated solutions iteratively. Starting with a regression tree $\mathcal{T}$, it determines whether to explore based on a random number compared to a given exploration probability p_e . Depending on this decision, it either extracts rules and retrieves solutions from the

most peak-valley leaf node ℓ_{PV} or the best leaf node ℓ_{BL} using the Retrieve Solutions from Leaf Node Algorithm (Algorithm 21). The extracted rules are then used to generate bounds for new solutions via the Bound Generation Algorithm. These bounds guide the generation of new solutions using Latin Hypercube Sampling. The algorithm iteratively adds new solutions to the set of selected solutions, $\mathcal{P}_{\text{selected}}$, by computing pairwise distances between each new solution and the solutions in the corresponding leaf node, selecting the new solution with the maximum of minimum distances. This process continues until half of the newly generated solutions are integrated into the population. The Retrieval of Solutions from Leaf Node Algorithm (Algorithm 21) specifically retrieves solutions that reside in a given leaf node of the regression tree, ensuring an accurate and efficient selection of solutions for further processing.

The Main Algorithm for Solution Search Space Reduction (Algorithm 22) orchestrates the entire process of optimizing the population of solutions. It begins by initializing a population of solutions, $\mathcal{P}$, of size n_{pop} and calculating their respective objective values, $\mathcal{O}$. A regression tree $\mathcal{T}$ is then generated using these solutions and their objective values. ℓ_{BL} and ℓ_{PV} are determined by processing the regression tree using the Regression Tree Processing Algorithm (Algorithm 16). The algorithm then enters an iterative loop, running for a predetermined number of iterations n_{iter} . In each iteration, the population is updated by calling the Population Update Algorithm (Algorithm 20), which generates new solutions and integrates them into the population. The updated population is evaluated, and a new regression tree is generated to re-identify the best and most peak-valley leaf nodes. After completing the iterations, the algorithm extracts rules from the best leaf node, generates bounds on the variables, and applies an exact solution method within these bounds to find the final optimized solution $\mathcal{S}_{final}$. The final objective value $\mathcal{O}_{final}$ is then evaluated, and both the solution and its objective value are returned. This comprehensive algorithm effectively integrates multiple components to refine the solution space and achieve optimal results.

Algorithm 22 Main Algorithm for Solution Search Space Reduction

1: **Input:**

- Initial population size n_{pop}
- Number of iterations n_{iter}
- Exploration probability p_e

2: **Output:**

- Final optimized solution $\mathcal{S}_{final}$
- Final objective value $\mathcal{O}_{final}$

3: Initialize population $\mathcal{P} \leftarrow \text{random_init}(n_{pop})$

4: Calculate objective values $\mathcal{O} \leftarrow \text{evaluate}(\mathcal{P})$

5: Generate regression tree $\mathcal{T} \leftarrow \text{regression_tree}(\mathcal{P}, \mathcal{O})$

6: $\ell_{BL}, \ell_{PV} \leftarrow \text{process_tree}(\mathcal{T})$ (Algorithm 16)

7: **for** $i = 1 \rightarrow n_{iter}$ **do**

8: $\mathcal{P}_{new} \leftarrow \text{update_population}(\mathcal{T}, \mathcal{P}, \mathcal{O}, \ell_{PV}, \ell_{BL}, p_e, n_{new})$ (Algorithm 20)

9: $\mathcal{P} \leftarrow \mathcal{P}_{new}$

10: $\mathcal{O} \leftarrow \text{evaluate}(\mathcal{P})$

11: $\mathcal{T} \leftarrow \text{regression_tree}(\mathcal{P}, \mathcal{O})$

12: $\ell_{BL}, \ell_{PV} \leftarrow \text{process_tree}(\mathcal{T})$ (Algorithm 16)

13: **end for**

14: $\mathcal{R} \leftarrow \text{rule_extract}(\ell_{BL})$ (Algorithm 17)

15: $\mathcal{B} \leftarrow \text{generate_bounds}(\mathcal{R})$ (Algorithm 18)

16: $\mathcal{S}_{final} \leftarrow \text{exact_solution}(\mathcal{B})$

17: $\mathcal{O}_{final} \leftarrow \text{evaluate}(\mathcal{S}_{final})$

18: **Return** $\mathcal{S}_{final}, \mathcal{O}_{final}$

5.4. Illustrative Examples

In previous chapters, we have evaluated the performance of our suggested tree-supported algorithms on well-known benchmark instances of the p -median problem, the Quadratic Assignment Problem (QAP), and the Quadratic Programming (QP) problem. In this current study, we expand our evaluation to further investigate the performance and behavior of the regression tree in nonlinear continuous problems. We select well-known and challenging continuous nonlinear problems to assess the performance of our algorithm developed.

5.4.1. Preliminary Analysis: Regression Trees for Continuous Solution Space Division

To support the development of the Learning-Based Bounds Generating Algorithm, a preliminary analysis is conducted on a well-known nonlinear continuous optimization problem:

$$f(x, y) = 3(1 - x)^2 e^{-(x^2 + (y+1)^2)} - 10 \left(\frac{x}{5} - x^3 - y^5 \right) e^{-(x^2 + y^2)} - \frac{1}{3} e^{-((x+1)^2 + y^2)}. \quad (5.13)$$

The 2D peaks function, defined in Equation 5.13, exhibits complex behavior with multiple peaks and valleys, making it a suitable test case to evaluate solution space division strategies. The analysis demonstrates how regression trees can be effectively utilized to iteratively partition the solution space and remove low-quality solutions. Two approaches are applied: *a univariate split*, where a regression tree is repeatedly fitted to random solutions, and the worst leaf node solutions are removed, and *a multivariate split*, where a linear regression predicts objective values and the regression tree uses these predictions to split and prune the space over multiple iterations. By applying these learning-based methods, the solution space is progressively reduced, highlighting the utility of regression trees for bound generation and solution space refinement.

5.4.1.1. Univariate Split. This approach uses a regression tree to iteratively partition the solution space and remove low-quality solutions. The process begins by generating random solutions and calculating their corresponding objective values. A regression tree is then fitted to these data, splitting the solution space into two leaf nodes. Among these, the leaf node containing the *region with the least promising solutions* (that is, the highest objective values) is identified and the solutions within it are removed. The regression tree is subsequently refitted using the remaining solutions, and this process is repeated iteratively to progressively refine the solution space.

This method demonstrates that a simple and iterative univariate split can effectively reduce the solution space by identifying and removing the less promising region. The process is illustrated in the *methodology visualization* (Figure 5.5), which provides a step-by-step description of the univariate split approach. Furthermore, *visualizations of the solution space* (Figures 5.6) show the reduction in the solution space across iterations, highlighting how the bounds narrow over time.

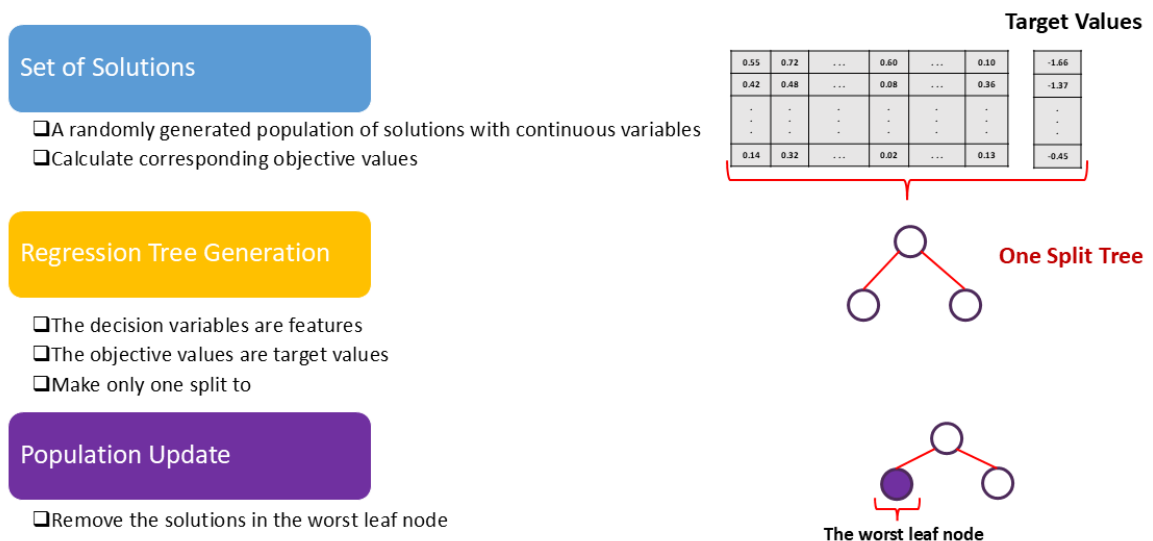


Figure 5.5. Univariate Split Methodology.

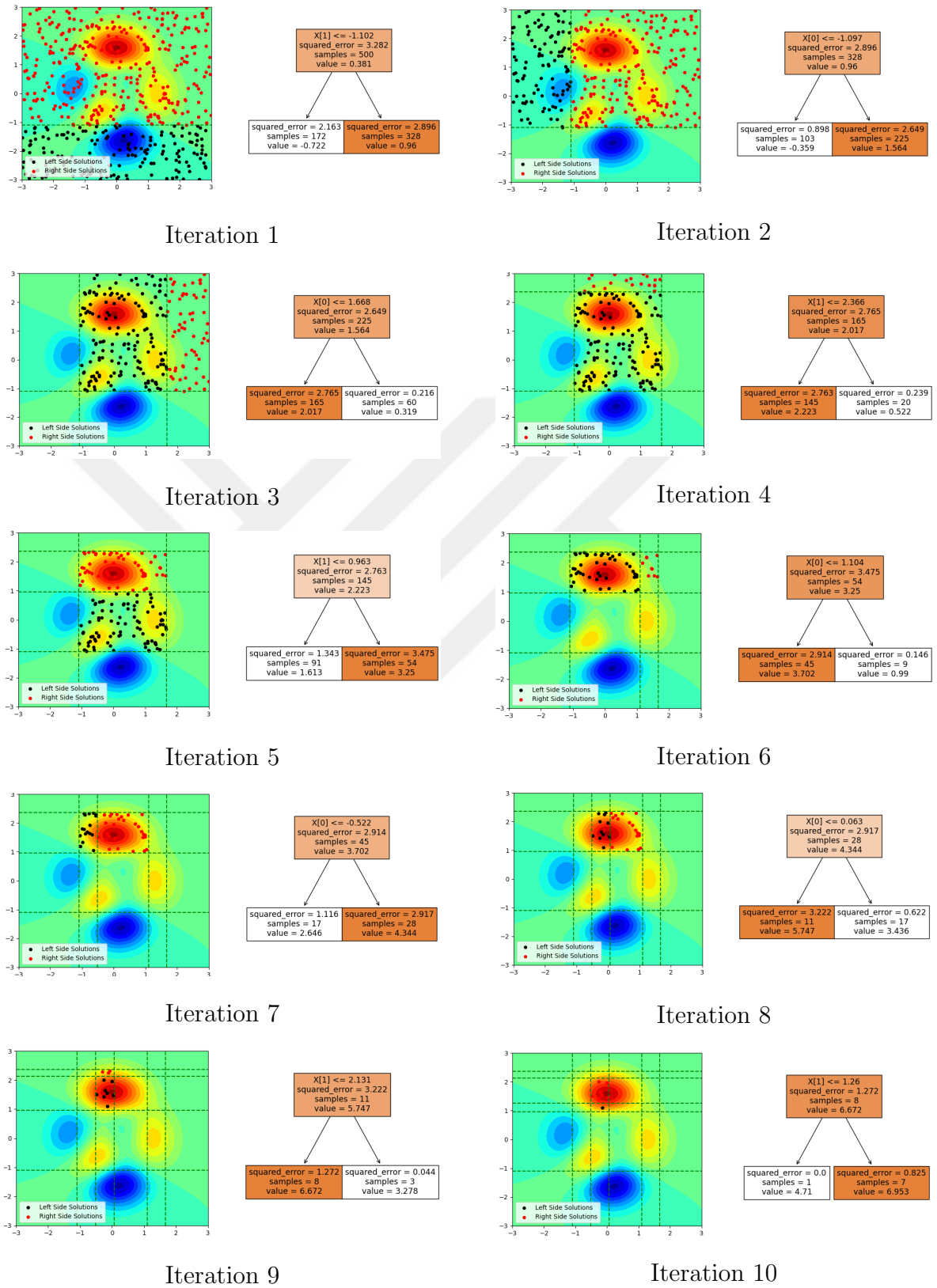


Figure 5.6. Solution space evolution for univariate split iterations (Iterations 1 to 10).

Each panel shows the solution space refinement at the corresponding iteration.

5.4.1.2. Multivariate Split. This approach combines linear regression with a regression tree to enhance the efficiency of the refinement of the solution space. First, a linear regression model is fitted to the random solutions to predict their objective values. The predicted objective values are used as features, while the real objective values serve as the target. A regression tree is then fitted to this enriched dataset, making a single split that results in two leaf nodes. The leaf node containing the *region with less promising solutions* is identified, and the solutions within it are removed. The regression tree is refitted using the remaining solutions, and this process is repeated iteratively, up to 10 iterations, until no data remains.

By integrating linear regression predictions with regression tree splits, this approach aims to improve the precision of solution space division and the efficiency of removing less promising solutions. The *methodology visualization* (Figure 5.7) illustrates the interaction between linear regression and regression tree-based splitting, while the *solution space visualizations* (Figures 5.8) depict how the solution space evolves and shrinks across iterations, demonstrating the effectiveness of the multivariate split.

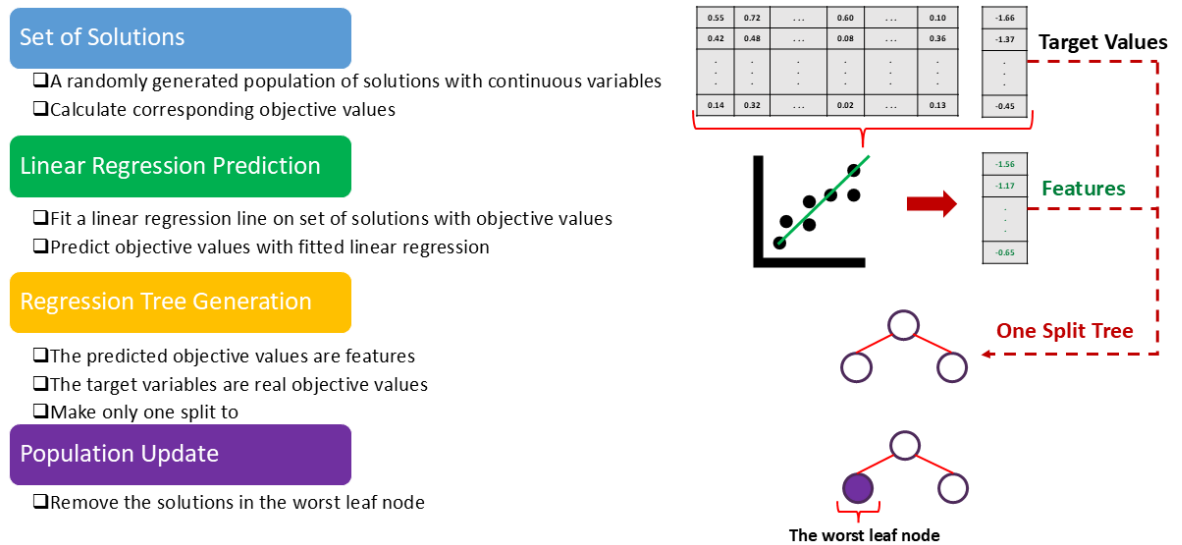


Figure 5.7. Multivariate Split Methodology.

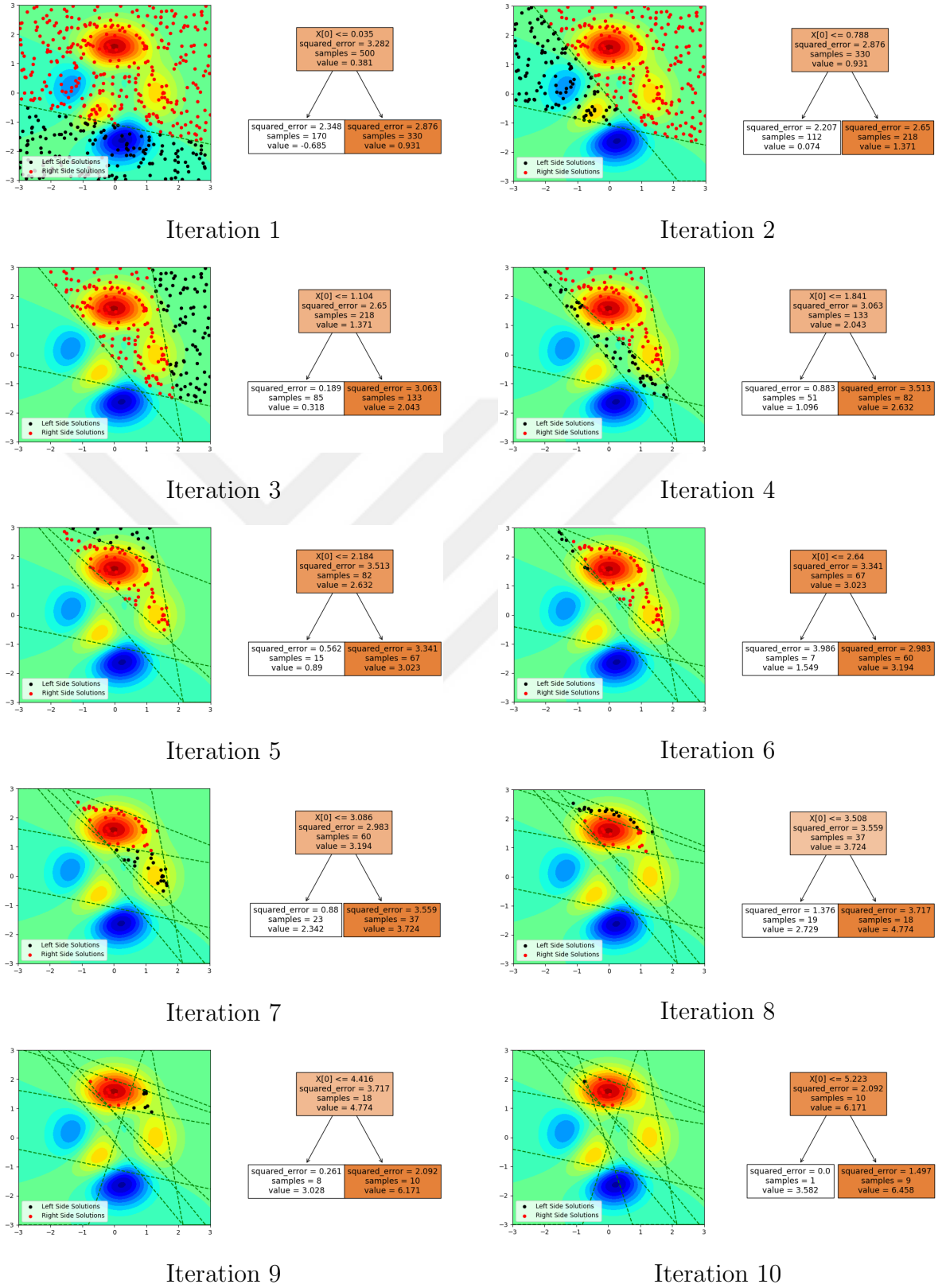


Figure 5.8. Solution space evolution for multivariate split iterations (Iterations 1 to 10). Each panel shows the solution space refinement at the corresponding iteration.

5.4.2. BBOB Test Functions

The Black-Box Optimization Benchmarking (BBOB) test functions are a comprehensive set of benchmark problems widely used to evaluate optimization algorithms. These functions are part of the COCO (COMparing Continuous Optimizers) framework, which provides a rigorous platform for assessing continuous optimization algorithms. The BBOB test suite comprises 24 noiseless test functions designed to represent various real-world optimization challenges. These functions cover different types of landscapes, including unimodal, multimodal, separable, non-separable, and those with varying conditioning levels. Each BBOB test function features a nonlinear objective, introducing inherent complexity and presenting challenges for optimization algorithms. Furthermore, the decision variables within these functions are continuous, facilitating a smooth and uninterrupted solution space. Importantly, the solution space for each function is continuous, creating a framework where variables can take any real value within a specified range. This thoughtful selection ensures diverse challenges, allowing for a comprehensive evaluation of optimization algorithms in handling nonlinearities and exploring continuous solution spaces.

5.4.2.1. Characteristics of BBOB Test Functions. The BBOB test functions are categorized into five distinct groups based on their characteristics:

- **Unimodal Functions:** These functions have a single global optimum and are used to test basic convergence, as they are typically easier for optimization algorithms. Examples include the Sphere and Ellipsoidal functions.
- **Multimodal Functions:** These functions have multiple local optima, making them challenging for optimization algorithms. They test an algorithm's ability to escape local optima and find the global optimum. Examples include the Rastrigin function and the Weierstrass function.
- **Separable Functions:** These functions can be optimized independently across dimensions and are used to test algorithms on problems with no variable interaction, such as the Sphere function.

- **Non-separable Functions:** These functions require simultaneous optimization of all dimensions due to variable interactions. They are more challenging and realistic representations of real-world problems. An example is the Rotated Ellipsoidal function.
- **Functions with Conditioning:** These functions vary in scale across directions, making them harder to optimize and useful for testing an algorithm's robustness. An example is the Ill-Conditioned Ellipsoidal function.

5.4.2.2. Notable BBOB Test Functions. The BBOB test suite comprises twenty-four distinct functions. Some of the most significant ones are described below:

(i) Sphere Function:

$$f(x) = \sum_{i=1}^n x_i^2. \quad (5.14)$$

A simple, unimodal, separable function is often used as a baseline for performance comparison.

(ii) Rastrigin Function:

$$f(x) = 10n + \sum_{i=1}^n [x_i^2 - 10 \cos(2\pi x_i)]. \quad (5.15)$$

A multimodal function with many local minima is challenging for global optimization.

(iii) Rosenbrock Function:

$$f(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]. \quad (5.16)$$

A unimodal, non-separable function that is widely used to test optimization algorithms due to its narrow, curved valley.

(iv) Weierstrass Function:

$$f(x) = \sum_{i=1}^n \left(\sum_{k=0}^{k_{\max}} [a^k \cos(2\pi b^k (x_i + 0.5))] \right). \quad (5.17)$$

A highly multimodal function with a fractal structure, differentiable nowhere, and high-frequency oscillations, making it very challenging for optimization.

5.4.3. Importance of BBOB Test Functions

BBOB test functions provide a standardized way to evaluate and compare the performance of optimization algorithms. By using a diverse set of functions that represent various real-world scenarios, researchers can gain insights into the strengths and weaknesses of different algorithms. The COCO platform also offers tools for automated benchmarking, statistical analysis, and visualization of results, facilitating rigorous and reproducible research in the field of optimization.

In summary, the BBOB test functions are a vital tool for developing and evaluating optimization algorithms. Their diverse characteristics ensure that algorithms are tested across various scenarios, thoroughly assessing their performance and robustness.

5.5. Numerical Results

We now apply a tree-based reduction in solution search space approach to the continuous functions. The suggested algorithm was implemented using Python 3.9 in Jupyter Notebook and VSCode environments. The experiments were conducted on a computer with an Intel Core i7-4510U CPU running at 2.00GHz, 8GB of RAM, and a Microsoft Windows 10 operating system. For the exact solution solver, we have selected the NLOpt open-source nonlinear solver tool to make our trials.

We selected 24 BBOB test functions for our trials, focusing on those where derivatives are unavailable or costly to compute. These functions have a defined range with lower and upper bounds of -5 and 5, respectively. We chose a dimensionality of 2 for our experiments, meaning each function has two variables. We generate bounds on variables by applying our proposed algorithms and shrink the solution search space. We support nonlinear solvers by shrinking the search space. To evaluate the effectiveness of this approach, we selected eight derivative-free solvers from the NLOpt framework, which offers a variety of optimization algorithms suitable for nonlinear problems

Among the derivative-free solvers provided in the NLOpt framework, some are more dependent on the initial solution. In contrast, others are designed to perform global optimization and are less sensitive to the starting point. The breakdown is provided:

Initial Solution Dependent (Local Optimizers): These algorithms are local optimizers, and their performance and results are significantly affected by the initial guess:

- COBYLA (Constrained Optimization BY Linear Approximations)
- BOBYQA (Bound Optimization BY Quadratic Approximation)
- PRAXIS (Principal Axis)
- Nelder-Mead Simplex Algorithm
- Sbplx (Subplex)

Less Initial Solution Dependent (Global Optimizers): These algorithms are designed for global optimization and are less sensitive to the initial guess, though it can still affect convergence speed and solution quality:

- DIRECT (DIviding RECTangles)
- CRS (Controlled Random Search)
- ISRES (Improved Stochastic Ranking Evolution Strategy)

In general, local optimizers will converge to the nearest local minimum from the initial starting point, making the choice of initial solution critical. On the other hand, global optimizers explore the search space more broadly and are less likely to be trapped in local minima, reducing the dependence on the initial guess.

5.5.1. Iterative Solution Space Partitioning and Optimization Using Regression Trees

Throughout the iterations, the split decisions of the regression tree progressively divide the solution space into distinct regions. These divisions adapt and refine over time, enabling more precise partitions that guide the generation of new solutions. To illustrate how the solution space is divided and how new solutions are logically added by utilizing these split decisions, we present Figures 5.9, 5.10, and 5.11. These figures demonstrate the regression trees' initial and iterative partitions of the solution space, highlighting the dynamic changes and improvements in the divisions. The pink shaded areas represent the best leaf nodes, while the pink symbols indicate the solution points generated using the previous iteration's tree. The yellow star indicates the optimal solution. This process demonstrates the effective adaptation and refinement of the solution space through regression tree methodologies across multiple iterations.

One of our primary aims is to detect the colorful areas representing the most peak-valley leaf nodes. Generating solutions from these leaf nodes is advantageous due to the inherent variability, which can enhance the learning process. By making further splits within these nodes, we can better differentiate between high-quality and low-quality solutions, thereby refining the optimization and improving the overall performance of the regression tree model. As shown in Figure 5.9, the pink circles denote the solutions generated from the peak-valley leaf node of the initial regression tree during Iteration 1. Up to Iteration 9, solutions are primarily generated from the pink-shaded areas, which indicate the best leaf nodes of the regression tree. In Iteration 9, however, the solutions are derived from the peak-valley leaf node identified in Iteration 8. Notably, the best leaf node changes across iterations, reflecting the dynamic adaptation and optimization of the solution space over time.

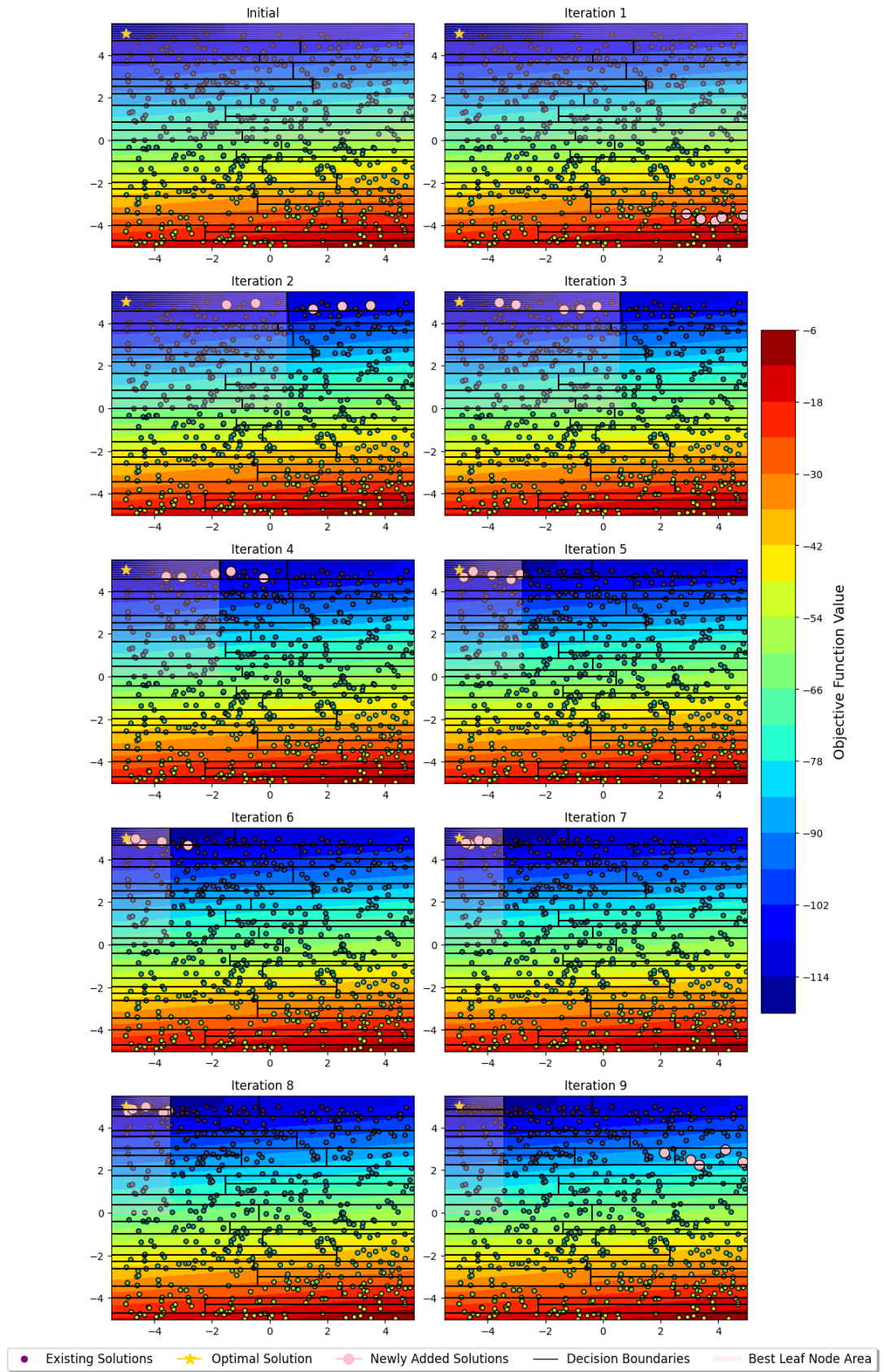


Figure 5.9. f5-Linear Slope-Regression Tree on Solution Space-Initial and Iterative.

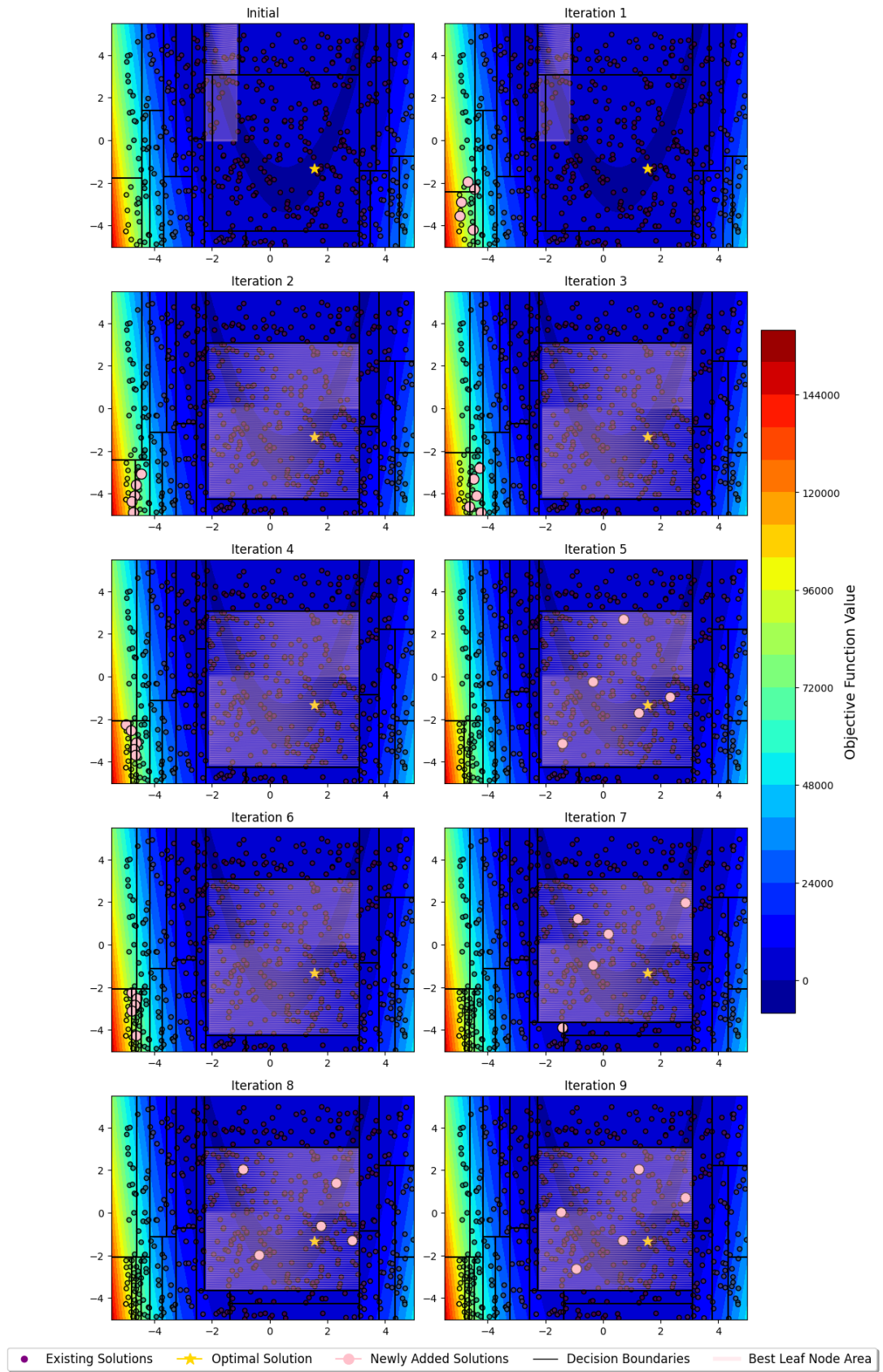


Figure 5.10. f8-Rosenbrock-Regression Tree on Solution Space-Initial and Iterative.

As shown in Figure 5.10, in the initial setup and Iteration 1, the pink-shaded area, representing the best leaf node, does not include the optimal solution. However, the solutions generated from the colorful regions—corresponding to the peak-valley leaf node—drive the pink-shaded area to shift and eventually cover the optimal solution. This process continues in subsequent iterations, progressively refining the solution space to include the optimal solution within the best leaf node. In the following iterations, solutions are alternately generated from the peak-valley leaf node and the best leaf node. This iterative refinement dynamically adapts the solution space, enabling the best leaf node to converge toward the region containing the optimal solution. In later iterations, solutions are generated solely from the best leaf node, indicating a stabilization of the process. These refinements successfully shrink the solution space while ensuring that the optimal solution is included within the bounds defined by the best leaf node. This highlights the effectiveness of the iterative approach in adapting and optimizing the solution space over time.

Figures 5.11 illustrate the evolution of the best and other leaf nodes throughout iterations. Initially, the optimal solution is contained within the best leaf node, represented by the pink-shaded area. However, in Iteration 1, as the algorithm progresses, the best leaf node shifts due to the generation of solutions from the leaf node with the most peak-valley characteristics. This shift causes the optimal solution to fall outside the best leaf node in subsequent iterations. Additionally, the best leaf node area progressively shrinks, indicating a reduction in the solution search space within the bounds of the best leaf node, but still failing to include the optimal solution. This outcome highlights the challenges posed by the complex landscape of the Weierstrass function, where the algorithm's exploration strategy struggles to adapt effectively in this instance.

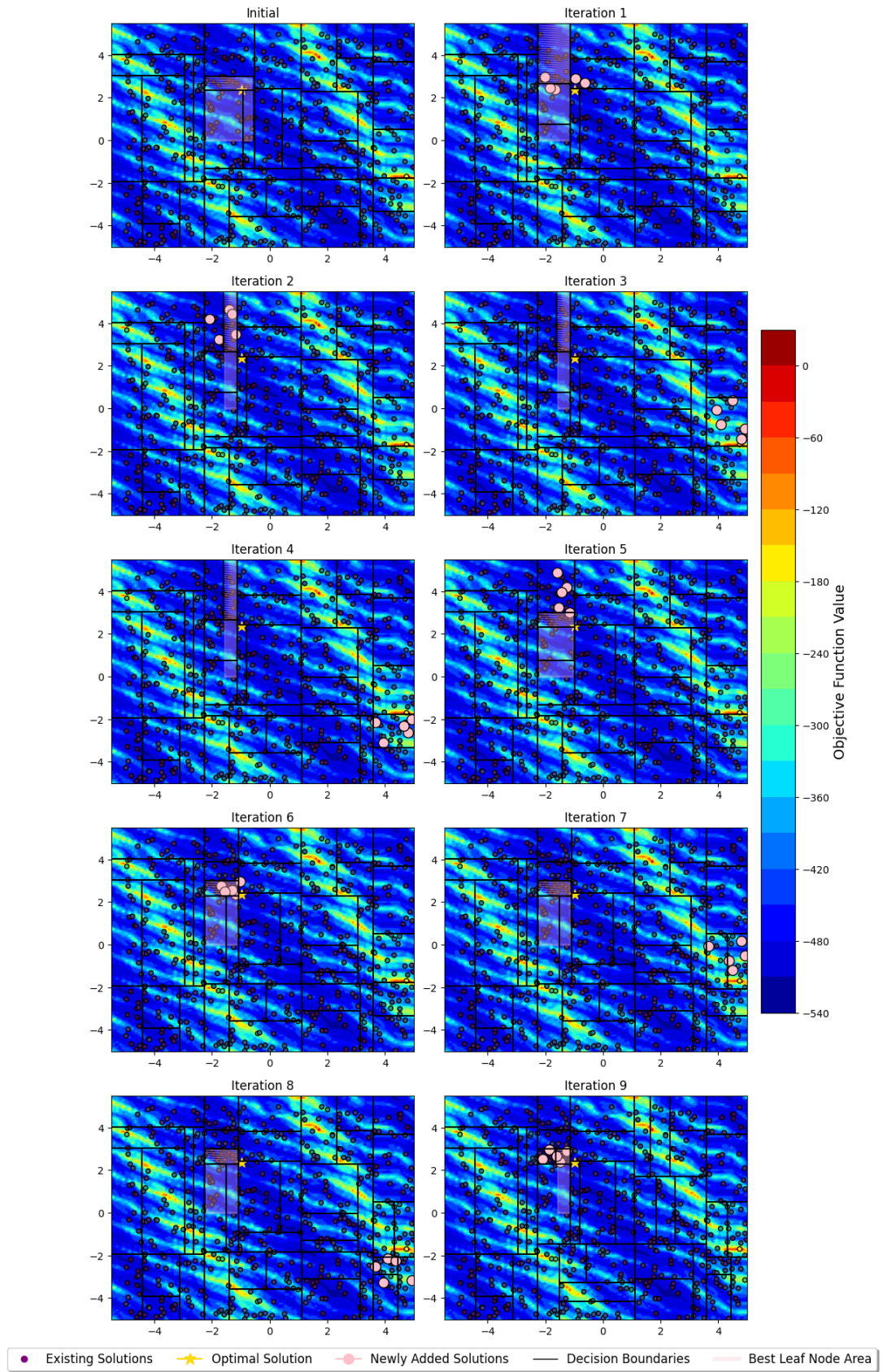


Figure 5.11. f16-Weierstrass-Regression Tree on Solution Space-Initial and Iterative.

5.5.2. BBOB Test Functions-Metric Evaluations

In this section, we analyze the proposed variable bounds generation algorithm's performance compared to the solutions with original bounds, called original, on BBOB Test Functions. As explained in Section 5.3, we generate bounds for variables benefiting from the split decisions of the regression tree.

The trials were conducted under several scenarios to evaluate the performance of the algorithm. In the *Base* scenario, a single regression tree is generated based on the *max_depth* parameters set to 4 and 8. This scenario has no population management phase; thus, the split decisions are obtained solely from the one tree generated with the initial population. Other parameters for the tree include a minimum of 10 solutions per leaf node and a minimum of 20 solutions required to perform a split.

The remaining scenarios incorporate population management and are based on the parameters, specifically for *max_depth* settings of 4 and 8. The methods compared include *medoid* and *pair-wise*, with exploration probabilities p_e set at 0.0, 0.3, and 0.5.

During the population update phase, 10 iterations are performed. Within these iterations, based on the exploration probabilities, control is implemented. If the randomly generated probability is less than the exploration probability, solutions are generated from the most peak-valley leaf node. This leaf node is determined according to either *medoid* or *pair-wise* methodologies. In each iteration, five solutions are generated and iteratively added as mentioned in Algorithm 20.

In this study, the average values of the five replications are utilized to assess the performance of outcomes in terms of these metrics. Before proceeding with the evaluation of the results, it is essential to refer to Table C.1, which provides an overview of the function codes, names, and groups in the BBOB test suite. This table categorizes the functions into groups: separable functions, functions with low or moderate conditioning, functions with high conditioning and unimodal, and multi-modal func-

tions with either adequate or weak global structure, which are mentioned in detail in Section 5.4.2. Understanding the characteristics of these function groups is important for interpreting the behavior of the solvers across different problem types.

The evaluation of the algorithm's performance is conducted with these categorizations. For instance, functions in the *separable functions* group (f1 to f5) generally exhibit simpler landscapes, which might explain why the algorithm performs exceptionally well on these functions, achieving minimal distances to the optimal solution. On the other hand, *multi-modal functions with weak global structure* (f20 to f24) present more complex landscapes with numerous local optima, posing greater challenges for the algorithm. This complexity is reflected in the results, where larger distances to the optimal solution are often observed.

By considering the function groups and their specific characteristics, we can better understand the nuances of the algorithm's performance across different scenarios. This structured approach allows for a comprehensive evaluation, highlighting both the strengths and areas for improvement in the algorithm's capability to handle various optimization landscapes.

Several metrics have been developed to measure the performance of our suggested algorithm. One such metric is the percentage shrinkage rate, which quantifies how much the generated bounds reduce the solution search space. The percentage shrinkage rate is calculated as follows.

Given the lower and upper bounds of the original hyperrectangle, we define $\mathbf{L}_{\text{org}} = (L_{\text{org},1}, L_{\text{org},2}, \dots, L_{\text{org},n})$ and $\mathbf{U}_{\text{org}} = (U_{\text{org},1}, U_{\text{org},2}, \dots, U_{\text{org},n})$, where each $L_{\text{org},i}$ and $U_{\text{org},i}$ represents the lower and upper bound, respectively, along the i -th dimension. The volume of the original hyperrectangle, V_{org} , is given by

$$V_{\text{org}} = \prod_{i=1}^n (U_{\text{org},i} - L_{\text{org},i}). \quad (5.18)$$

Similarly, for the generated bounds, we define $\mathbf{L}_{\text{path}} = (L_{\text{path},1}, L_{\text{path},2}, \dots, L_{\text{path},n})$ and $\mathbf{U}_{\text{path}} = (U_{\text{path},1}, U_{\text{path},2}, \dots, U_{\text{path},n})$. The volume of the hyperrectangle with the generated bounds, V_{path} , is calculated as

$$V_{\text{path}} = \prod_{i=1}^n (U_{\text{path},i} - L_{\text{path},i}). \quad (5.19)$$

The shrinkage rate S and the percentage shrinkage rate $S_{\%}$ are defined as

$$S = \frac{V_{\text{org}} - V_{\text{path}}}{V_{\text{org}}} \quad (5.20)$$

$$S_{\%} = S \times 100. \quad (5.21)$$

The percentage shrinkage rate measures the reduction in the search space volume. The tables present the mean percentage shrinkage rate values for different scenarios varying according to methods and parameters, specifically for *max_depth* settings of 4 and 8. The methods, i.e. to determine the most peak-valley leaf node, compared include *medoid* and *pair-wise*, with p_e set at 0.0, 0.3, and 0.5.

For *max_depth*=4 (Table C.2), the shrinkage rate for the *Base* scenario averages 63.7%. The *medoid* method slightly improves shrinkage, particularly at higher exploration probabilities, with the best performance (65.0%) observed at a probability of 0.3. The *pair-wise* method also demonstrates a consistent performance, with the highest shrinkage rate (64.7%) occurring at an exploration probability of 0.0. Notably, functions such as f5 and f16 exhibit significant improvements with both methods, achieving shrinkage rates as high as 99.4% and 99.6%, respectively.

For *max_depth*=8 (Table C.3), the shrinkage rate for the *Base* scenario is considerably higher, averaging 87.9%. The *medoid* method continues to perform well, though it shows a slight decrease in effectiveness with increasing exploration probabilities. The best performance (87.7%) is observed at an exploration probability of 0.0. The *pair-wise* method maintains a stable performance, with the highest shrinkage rate (87.3%) occurring at a probability of 0.5. Functions such as f5 and f16 again have

exceptional decreases in their hyperrectangle volumes, with shrinkage rates reaching 100% and 99.6%, respectively.

These results indicate that a larger depth of the regression tree leads to a higher percentage shrinkage rate. In addition, the results highlight the effectiveness of both the *medoid* and *pair-wise* methods in reducing the search space volume, with varying performance depending on the exploration probability and the depth of the regression tree.

Another metric to evaluate the performance of our algorithm is the number of out-of-bound variables. It measures how many variables are not covered by the path bounds. The number of out-of-bounds variables, N_{outbound} , is given by

$$N_{\text{outbound}} = \sum_{i=1}^n (\mathbf{1}\{x_i < L_{\text{pathbounds},i}\} + \mathbf{1}\{x_i > U_{\text{pathbounds},i}\}) \quad (5.22)$$

where $\mathbf{1}\{\cdot\}$ is the indicator function, which equals 1 if the condition inside the braces is true and 0 otherwise, and x_i is the value of the i -th variable.

The number of out-of-bound variables is another crucial metric for evaluating the performance of our algorithm. The values presented in Tables C.4 and C.5 are the results of the number of out-of-bound variables based on different scenarios, averaged over 5 replications.

For $\text{max_depth}=4$ (Table C.4), the *Base* scenario shows a mean value of 0.23 out-of-bound variables. The *medoid* method with an exploration probability of 0.3 has the highest mean value of 0.29, while the *pair-wise* method with exploration probabilities of 0.0 and 0.3 both show a mean value of 0.23. Function f16 consistently shows 2.0 out-of-bound variables across most scenarios, indicating a higher sensitivity to boundary conditions. This is particularly interesting because f16 also has one of the highest percentage shrinkage rates. This suggests that while the algorithm effectively shrinks the variable bounds, it generally exceeds the optimal solution bounds. Function f12 also stands out with 0.4 out-of-bound variables in certain *medoid* scenarios with explo-

ration probabilities of 0.3 and 0.5. These findings highlight the importance of carefully balancing bound tightening and solution feasibility when designing variable reduction strategies.

For $max_depth=8$ (Table C.5), the *Base* scenario exhibits a higher mean value of 0.55 out-of-bound variables, reflecting the increased complexity with greater tree depth. The *pair-wise* method with an exploration probability of 0.5 has a mean value of 0.51, while the *medoid* method with exploration probabilities of 0.0 and 0.3 shows mean values of 0.51 and 0.45, respectively. Functions such as f16, f18, and f19 exhibit higher out-of-bound variable counts, with f16 showing values up to 2.0 in various scenarios. The Weierstrass function, i.e. f16, is known for its fractal nature, which exhibits self-similarity at different scales. Trees partition the space into axis-aligned rectangles, which may not align well with the complex, repeating patterns of the Weierstrass function. Furthermore, it includes high-frequency oscillations, leading to rapid changes in the function value over very short intervals. Regression trees, which create partitions based on average values within bins, might not capture these rapid oscillations effectively, leading to inaccurate predictions and suboptimal bounds.

These results highlight how different methods and exploration probabilities affect the number of out-of-bound variables, with larger tree depths generally leading to more out-of-bound variables, thus influencing the algorithm’s performance. This trend suggests a trade-off between deeper partitioning of the solution space and maintaining tighter adherence to feasible regions.

In addition to the number of out-of-bound variables, another important metric to evaluate the performance of our algorithm is the leaf node order of the optimal solution. This metric determines the position of the optimal solution within the leaf nodes of the regression tree. Specifically, when the optimal solution is input into the regression tree, the leaf node order indicates the sequence in which the optimal solution is found within the tree’s leaf node. This metric determines the position of the optimal solution within the leaf nodes of the regression tree as

Let $\mathcal{T}$ be the regression tree, $\mathbf{x}$ be the optimal solution, and $\{L_1, L_2, \dots, L_m\}$ be the set of leaf nodes in the tree, indexed by i . Furthermore, let $\mathcal{O}_i$ be the average objective value of leaf node L_i .

We sort the leaf nodes $\{L_1, L_2, \dots, L_m\}$ according to their average objective values $\mathcal{O}_i$ in ascending order. Let the sorted order of the leaf nodes be $\{L_{(1)}, L_{(2)}, \dots, L_{(m)}\}$. We assign an order to each leaf node based on this sorting so that $L_{(j)}$ corresponds to the j -th position in the sorted list.

When the optimal solution $\mathbf{x}$ is input into the regression tree $\mathcal{T}$, it follows a series of split decisions that lead it to a specific leaf node. Let's denote this leaf node as $L_{\mathbf{x}}$.

The leaf node order $L_{\text{order}}(\mathbf{x})$ of the optimal solution $\mathbf{x}$ is given by the position of $L_{\mathbf{x}}$ in the sorted list $\{L_{(1)}, L_{(2)}, \dots, L_{(m)}\}$ as

$$L_{\text{order}}(\mathbf{x}) = j \quad \text{if} \quad L_{\mathbf{x}} = L_{(j)}. \quad (5.23)$$

Let R be the total number of replications, $L_{\text{order}}^r(\mathbf{x})$ be the leaf node order of $\mathbf{x}$ in the r -th replication, and M_r be the reciprocal of the leaf node order in the r -th replication, defined as

$$M_r = \frac{1}{L_{\text{order}}^r(\mathbf{x})}. \quad (5.24)$$

The mean of the reciprocal leaf node orders over R replications is computed as

$$\overline{M} = \frac{1}{R} \sum_{r=1}^R M_r. \quad (5.25)$$

Finally, the reciprocal of this mean, denoted as F , is given by

$$F = \frac{1}{\overline{M}} \quad (5.26)$$

where F is the final aggregated measure, based on the leaf node orders over multiple replications.

The leaf node order of the optimal solution is another key metric for evaluating the performance of our algorithm. Tables C.6 and C.7 show the mean leaf node order for various scenarios with *max_depth* settings of 4 and 8, respectively. Ideally, the leaf node order should be 1, indicating that the optimal solution is located in the best leaf node, demonstrating the algorithm’s effectiveness in differentiating and isolating the optimal solution.

For *max_depth*=4 (Table C.6), most functions exhibit a leaf node order of 1 across all scenarios, suggesting a high accuracy in identifying the best leaf node. However, function f16 stands out with higher leaf node orders, ranging from 3.1 to 6.1 across different methods and exploration probabilities. This correlates with f16 having one of the highest percentage shrinkage rates, indicating that while the bounds are significantly reduced, they sometimes extend beyond the optimal solution bounds, resulting in a higher number of out-of-bound variables.

For *max_depth*=8 (Table C.7), the base scenario and many functions still maintain a leaf node order of 1, showing consistency in identifying the optimal solution within the best leaf node. Nevertheless, function f16 again shows higher leaf node orders, reaching 8.7, which aligns with its high shrinkage rate and the higher count of out-of-bound variables. Functions like f22 and f23 also exhibit larger leaf node orders in certain scenarios, further indicating challenges in accurately bounding the optimal solution.

These results suggest that when the optimal solution resides in the best leaf node (leaf node order = 1), our algorithm is highly effective in generating bounds that closely approximate the optimal or near-optimal solutions. However, instances with larger leaf node orders tend to have higher numbers of out-of-bound variables, indicating that the bounds generated may not be as precise. This highlights the importance of achieving a leaf node order of 1 to ensure the accuracy and efficiency of the generated bounds, ultimately leading to better performance of the nonlinear solvers in finding the optimal solutions.

5.5.3. BBOB Test Functions Trial Results-Minimum Distance to Optimal Solution

Another important metric for evaluation is the distance to the optimal solution. This metric allows us to compare the effectiveness of the algorithm-generated bounds and initial solutions against the results obtained by the solver with original bounds and random initial solutions. As previously mentioned, our trials are conducted using eight nonlinear derivative-free solvers.

We conducted five replications to evaluate our algorithm's performance where variable bounds were generated and provided to a nonlinear solver. For each replication, the solver received an initial solution set at the centroid of the best leaf node from the final regression tree. Additionally, we solved the instances using the original variable bounds, each time with five different random initial solutions. This study used the minimum values obtained from the five replications to assess performance outcomes. This approach serves as a robust performance indicator, demonstrating the algorithm's potential to achieve optimal results consistently across different replications.

Table 5.2 summarizes the scenarios yielding the best total distance reductions to the optimal solution across various solvers. Each entry specifies the maximum depth (4 or 8), the method used (*pair-wise* or *medoid*), and the p_e value that led to better results, along with a *Base* scenario for each maximum depth.

Table 5.2. Summary of Best Scenarios for Each Solver.

Solvers	1			2		
	max_depth	method	p_e	max_depth	method	p_e
BOBYQA	8	pair-wise	0.3	8	pair-wise	0.0
COBYLA	8	pair-wise	0.0	8	medoid	0.0
PRAXIS	8	pair-wise	0.3	8	pair-wise	0.0
NELDERMEAD	8	pair-wise	0.3	8	pair-wise	0.0
SBPLX	8	pair-wise	0.3	8	pair-wise	0.5
DIRECT	8	pair-wise	0.0	8	base	base
CRS2_LM	8	pair-wise	0.0	8	base	base
ISRES	8	pair-wise	0.3	8	pair-wise	0.0

For the BOBYQA solver, the best scenarios involved using the *pair-wise* method with $p_e = 0.3$ at max_depth 8 and the *pair-wise* method with $p_e = 0.0$ at max depth 8 (Table C.24). Similarly, the COBYLA solver achieved optimal results with the *pair-wise* method at $p_e = 0.0$ and $p_e = 0.3$ and the *medoid* method at $p_e = 0.0$ at max_depth 8 (Table C.25). The PRAXIS solver performed best using the *pair-wise* method with $p_e = 0.3$ and $p_e = 0.0$ at max_depth 8 (Table C.26).

For the NELDERMEAD solver, the most effective scenarios were the *pair-wise* method with $p_e = 0.3$ and $p_e = 0.0$, and *Base* method at max_depth 8 (Table C.27). The SBPLX solver performed best using the *pair-wise* method with $p_e = 0.3$ and $p_e = 0.5$ at max_depth 8 (Table C.28).

The DIRECT solver's best results were obtained with the *pair-wise* method at $p_e = 0.0$ and $p_e = 0.3$, and *Base* method at max_depth 8 (Table C.29). For the CRS2.LM solver, the optimal scenarios were the *pair-wise* method at $p_e = 0.0$, *Base* method at max_depth 8, and the *pair-wise* method at $p_e = 0.3$ at max_depth 4 (Table C.30). Lastly, the ISRES solver performed best using the *pair-wise* method at $p_e = 0.3$ and 0.0 and *Base* method at max_depth 8 (Table C.31).

The analysis highlights that the *pair-wise* method with varying p_e values, especially 0.3, frequently yielded the best results across different solvers. This can be interpreted as that deeper trees with exploration by benefiting from the most peak-valley leaf node improves the solution quality. In addition, the *Base* method also shows significant improvements for several solvers, particularly at higher maximum depth parameters of the tree.

5.5.4. BBOB Test Functions Comparison with Original-Minimum Distance to Optimal Solution

In almost all scenarios, the integration of regression tree-supported variable bound generation consistently improves the performance of all solvers for more than half of the

instances. Each table, such as C.34 for BOBYQA, shows that out of the 24 instances, the number of improved instances exceeds 12 in nearly all scenarios. This demonstrates the robustness and effectiveness of the regression tree approach in enhancing solver performance.

The comparison of solvers reveals specific configurations that yield the best performance. For instance, BOBYQA shows optimal performance with the configuration of 8-pair-wise-0.3 as seen in Table C.34. Similarly, COBYLA achieves its best results with the configuration of 8-pair-wise-0.0, as indicated in Table C.35. Other solvers such as PRAXIS and NELDERMEAD also exhibit significant improvements under specific configurations, namely 8-pair-wise-0.3, as detailed in Tables C.36 and C.37 respectively.

In the case of SBPLX, two configurations, 8-pair-wise-0.3 and 4-pair-wise-0.3, are highlighted for their effectiveness, with Table C.38 providing the supporting data. DIRECT stands out with improvements observed in all scenarios, with 8-pair-wise-0.0 being the most effective configuration as shown in Table C.39. The CRS2_LM solver demonstrates optimal performance with 4-pair-wise-0.3 configuration, as seen in Table C.40, while ISRES shows significant improvements with 8-pair-wise-0.0, supported by the data in Table C.41. These tables underscore the versatility and impact of regression tree-supported variable bound generation across different solvers, highlighting specific configurations that maximize performance improvements.

5.5.5. BBOB Test Functions Graphical Comparison Based on Selected Best Scenarios

In this section, we present a series of graphs comparing the performance of various solvers with original bounds and bounds generated by our regression tree-based method. 5.2 summarizes, for each solver, the values of the parameters max_depth and p_e that yield a better solution in the largest number of test functions when the solver is called with revised bounds compared to the baseline with the original bounds. For example, setting the parameters $max_depth = 8$ and $p_e = 0.3$ provides the best results for

the BOBYQA solver, having a better solution than the baseline in 20 test functions. It can be seen that a regression tree with $max_depth = 8$ consistently yields better performance in all solvers compared to $max_depth = 4$. This can be attributed to the complexity of the test functions, which require a deeper tree for accurate predictions. Moreover, while most solvers yield better results with revised bounds when $p_e = 0.3$, the COBYLA, DIRECT, and CRS2_LM solvers achieve their best results with $p_e = 0.0$, i.e., new solutions are always generated from the best leaf node L_{best} . By highlighting the best scenario obtained for each solver, we aim to showcase the effectiveness of the regression tree-supported variable bound generation in enhancing solver performance. The graphs illustrate the distance to the optimal solution for each solver, allowing for a visual comparison of the improvements achieved with the new bounds. The purpose of these visualizations is to emphasize the potential of our approach in optimizing solver outcomes and to provide a clear and concise demonstration of its impact.

The provided figures compare the distance to the optimal solution when the solver runs with original bounds versus when it runs with bounds generated by our algorithm. Each figure includes two sets of data points: one representing the algorithm’s minimum solution distance (*rev_bound*) and the other representing the original minimum solution distance (*org_bound*). These distances are measured by making trials with twenty-four BBOB test functions, with each function represented on the y-axis and the distance to the optimal solution on the x-axis. In these figures, the legend identifies the two types of data points:

- *rev_bound* (blue dots): These points represent the minimum distance to the optimal solution obtained using our algorithm-generated bounds in the solver.
- *org_bound* (orange triangles): These points represent the minimum distance to the optimal solution obtained using the original bounds in the solver.

The y-axis lists the function codes (e.g., f1, f2, etc.) representing the BBOB test functions used in the trials. The x-axis shows the distance to the optimal solution, indicating how close each solver’s result is to the optimal solution. From the graphs,

we observe that for most functions, the algorithm-generated bounds (blue dots) result in distances to the optimal solution comparable to or better than those obtained using the original bounds (orange triangles).

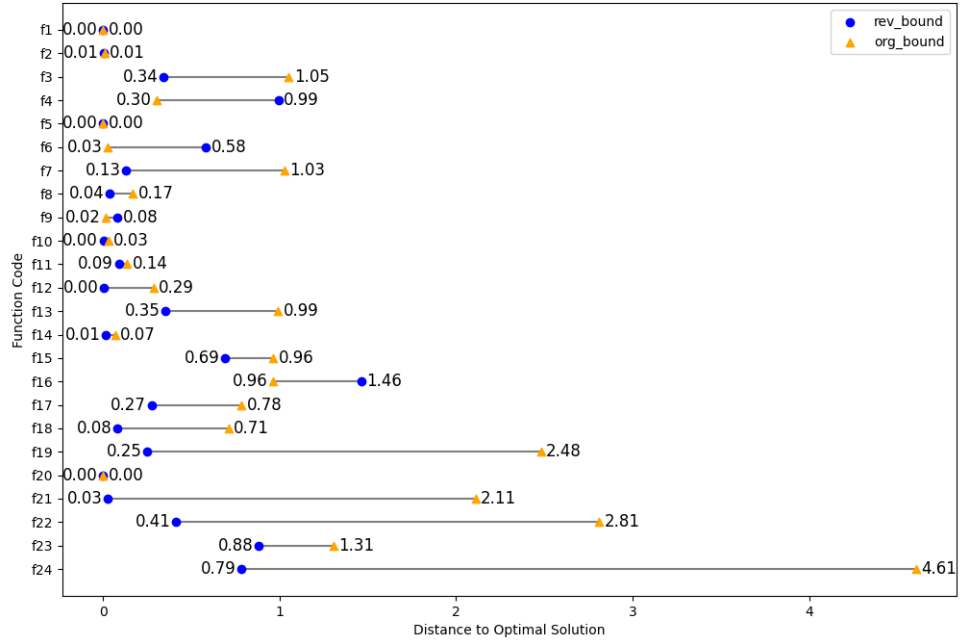


Figure 5.12. BOBYQA-max_depth=8-pair_wise=0.3.

For the BOBYQA solver (Figure 5.12), in 20 out of 24 functions, the generated bounds help improve the solver results. The functions f4, f6, f9, and f16 show a slight decline in performance with the bounds. For f16, the Weierstrass function, the average number of out-of-bounds variables is 2, indicating that in all replications, the bounds generated in this scenario do not cover the optimal solution. However, by setting $p_e = 0.0$, the performance improves, suggesting that exploration does not benefit this function. In contrast, some functions like f19, f21, f22, and f24 show significant improvement using our algorithm-generated bounds. Specifically, f24, the Katsuura function, is known for its highly rugged with many local minima and intricate landscape, making it particularly challenging for solvers. The generated bounds effectively help navigate this complex contour, leading to better optimization results.

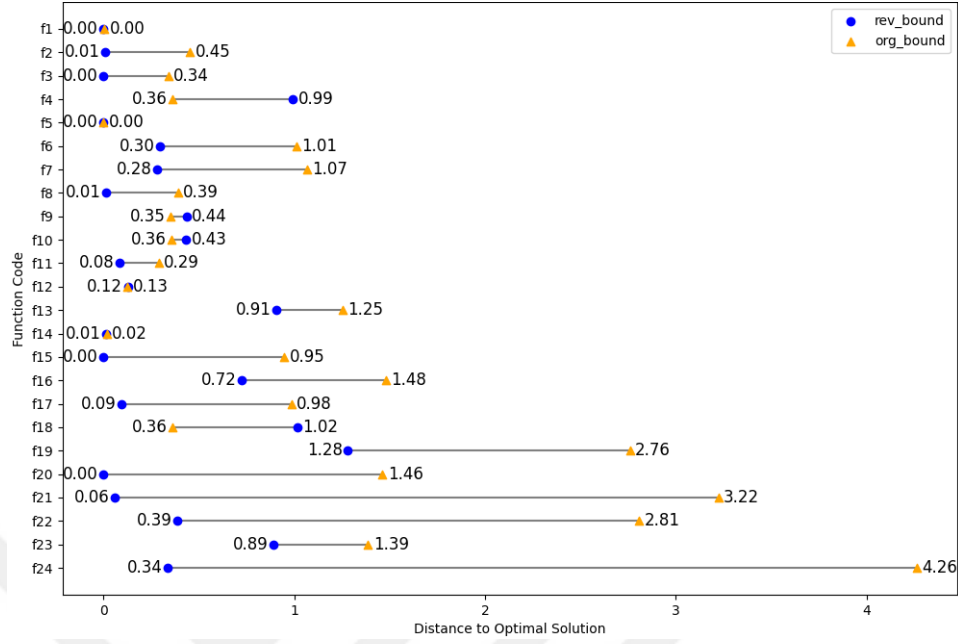


Figure 5.13. COBYLA-max_depth=8-pair_wise=0.0.

In the case of COBYLA solver (Figure 5.13), in 17 out of 24 functions, the generated bounds help improve the solver results, as indicated by the blue dots being closer to the left-hand side, compared to the orange triangles. In functions such as f20, f21, f22, f23, and f24, our algorithm significantly reduces the distance to the optimal solution, showcasing the effectiveness of our approach. However, there are a few functions where the generated bounds do not lead to improvements, such as f4, f9, f10, and f18, where the blue dots are positioned significantly worse than the orange triangles. For the f18 function, the performance improves by setting $p_e = 0.3$. The graph demonstrates that our algorithm-generated bounds enhance the COBYLA solver's performance in most functions, with notable improvements in several challenging instances. This outcome suggests that the success of the generated bounds can vary depending on the landscape complexity of each function.

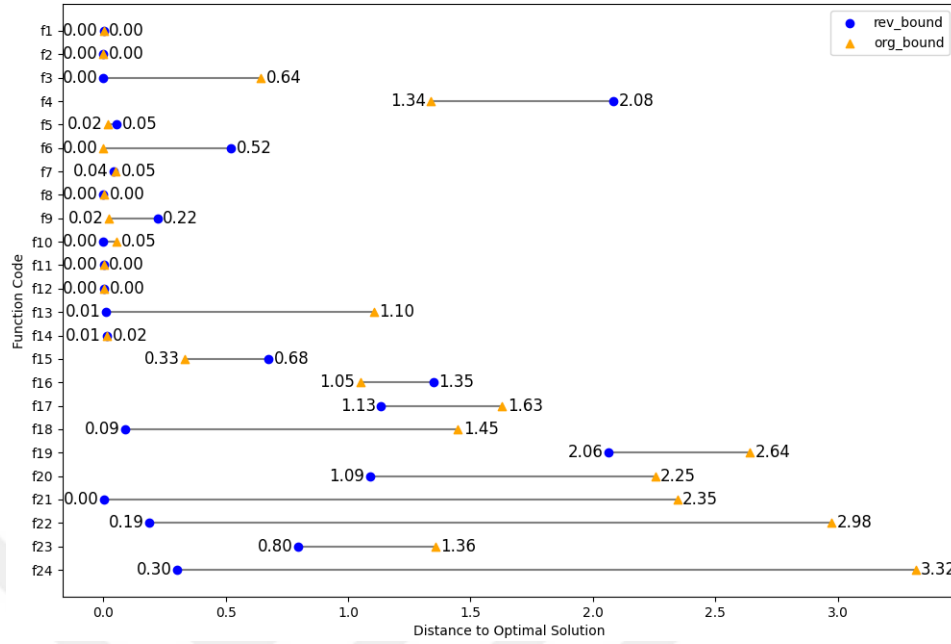


Figure 5.14. PRAXIS-max_depth=8-pair_wise=0.3.

For the PRAXIS solver (Figure 5.14), the generated bounds improve 17 out of 24 functions. This indicates that our regression tree-generated bounds help reduce the distance to the optimal solution for most functions. In functions such as f3, f6, f7, f13, and f18, there are noticeable improvements with our algorithm. However, there are functions where the generated bounds do not result in improvements, such as f4 and f16, where the blue dots are positioned further from the optimal solution than the orange triangles. Particularly, for function f16, the Weierstrass function, the blue dot is further from the optimal solution compared to the orange triangle, indicating that the regression tree-generated bounds do not cover the optimal solution effectively. However, in functions like f22, f23, and f24, our approach demonstrates significant improvements, reinforcing the effectiveness of our approach in these instances. Function f24 remains challenging, though it has noticeable improvements, but it is still relatively difficult for the solver. Hence, our algorithm-generated bounds prove effective in enhancing the solver's performance across many functions. While there are instances where the bounds do not lead to improvements, the substantial enhancements observed in several cases underscore the overall effectiveness of our approach.

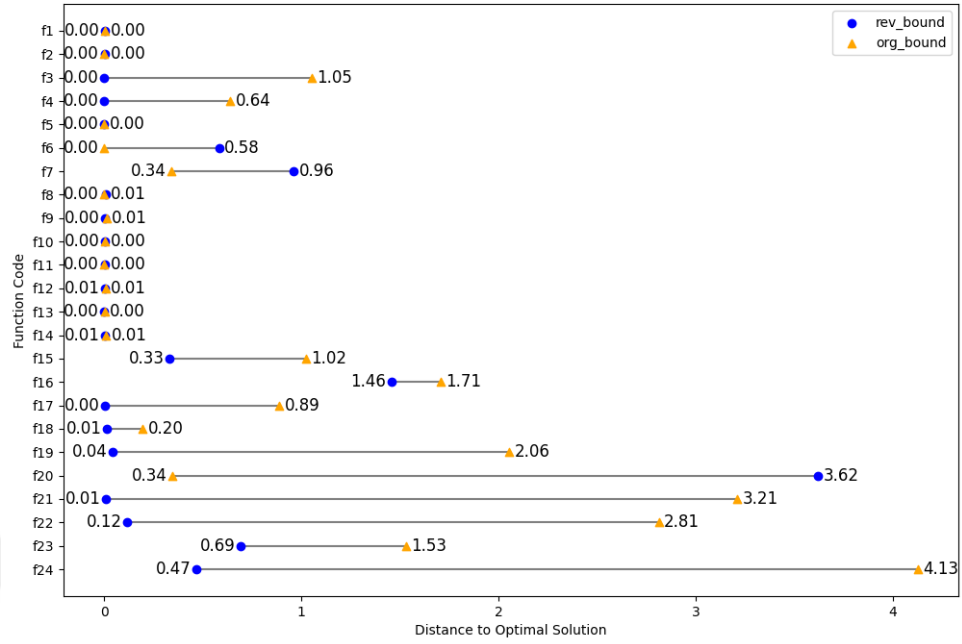


Figure 5.15. NELDERMEAD-max_depth=8-pair_wise=0.3.

For the NELDERMEAD solver (Figure 5.15), the generated bounds improve the solver's performance in 17 out of 24 functions. This is indicated by the blue dots being closer to the optimal solution than the orange triangles for these functions. An important challenge is highlighted in functions f3 and f4, where the regression tree-generated bounds enable the solver to reach optimality, which the NELDERMEAD solver alone could not achieve. This demonstrates the significant impact of the provided bounds on these particular functions. Moreover, functions like f21, f22, f23, and f24 show significant improvements, where the blue dots are noticeably closer to the optimal solution, illustrating the effectiveness of the bounds in these instances. Particularly for function f16, similar to the other solvers, the regression tree-generated bounds are ineffective, as indicated by the high number of out-of-bounds variables. In contrast, the functions f6, f7, and f20 the generated bounds do not lead to improvements. For f20, the generated bounds cover the optimal solution, and it was expected to improve the result by shrinking the search space by 79%. However, the improvement was not observed, indicating a capacity problem related to the NELDERMEAD solver.

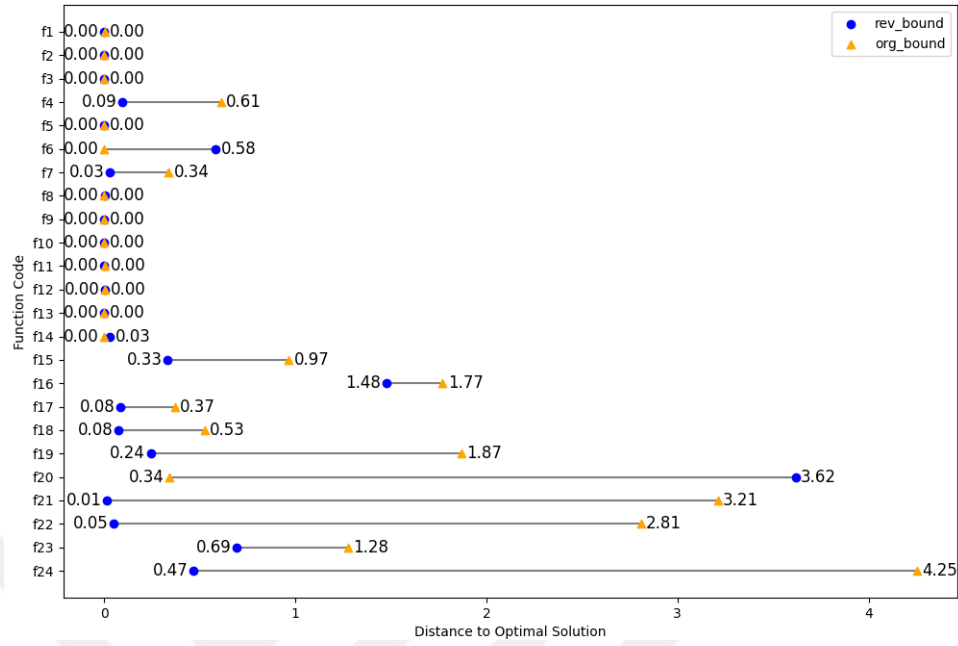


Figure 5.16. SBPLX-max_depth=8-pair_wise=0.3.

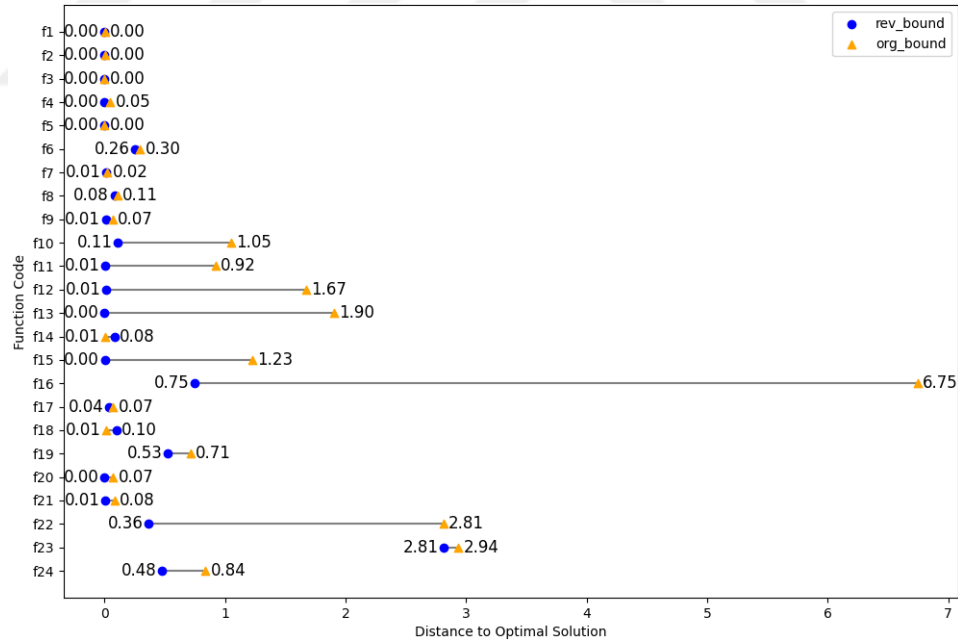


Figure 5.17. DIRECT-max_depth=8-pair_wise=0.0.

In comparing the SBPLX solver with the original and regression tree-generated bounds, we observe significant improvements in 17 out of the 24 functions, as shown in Figure 5.16. For only f6 and f20, there is a visible worse performance with the generated bounds. The remaining instances, which our algorithm does not improve, show very

close or minimally worse performance. On the other hand, f21, f22, f23, and f24 are the instances the solver has difficulty to solve are improved by the generated bounds by our algorithm. Notably, functions f3, f8, f9, f10, f11, f12, and f13 are solved to optimality with both sets of bounds, as indicated by the overlapping blue and orange triangles at zero distance. Functions like f7, f15, f17, and f18 show marked improvements, with the blue dots significantly closer to the optimal solution than the orange triangles. For instance, f17 sees a reduction from 0.37 to 0.08 and f18 from 0.53 to 0.08.

DIRECT, CRS2_LM, and ISRES are the solvers that solve to global minima. Figure 5.17 shows the comparisons made with the DIRECT solver. The graph for the DIRECT solver demonstrates a consistent performance with generated bounds. Out of 24 functions, 22 instances show improvements with the bounds generated by our regression tree algorithm. Differently from the previous solvers, DIRECT already performs well on some problem instances. For instance, function f24 shows only a slight improvement, as the DIRECT solver already performs well on this function without the generated bounds. The additional support from the bounds does help but not significantly since the solver's original performance was already strong. Function f16 stands out as particularly problematic for DIRECT. The DIRECT algorithm operates by iteratively dividing the search space into smaller rectangles and evaluating these subdivisions to guide its search. However, due to the complex and rugged landscape of f16, as mentioned in Sections 5.4.2 and 5.5.1, this method becomes inefficient. The nature of the Weierstrass function, with its numerous peaks and valleys, complicates the subdivision process, leading to poor performance. This highlights the limitation of DIRECT in handling functions with intricate landscapes. Hence, we make significant improvements to this function by providing the DIRECT with the tree-generated bounds.

Among the solvers that solve global minima, DIRECT is the least dependent on the initial solution. Consequently, the minimum and average distances to the optimal solution across replications are very close for DIRECT. When comparing the minimum distances of the replications, DIRECT's results may appear less favorable. However,

DIRECT performs well when considering the average performance, demonstrating consistent and reliable outcomes across multiple runs.

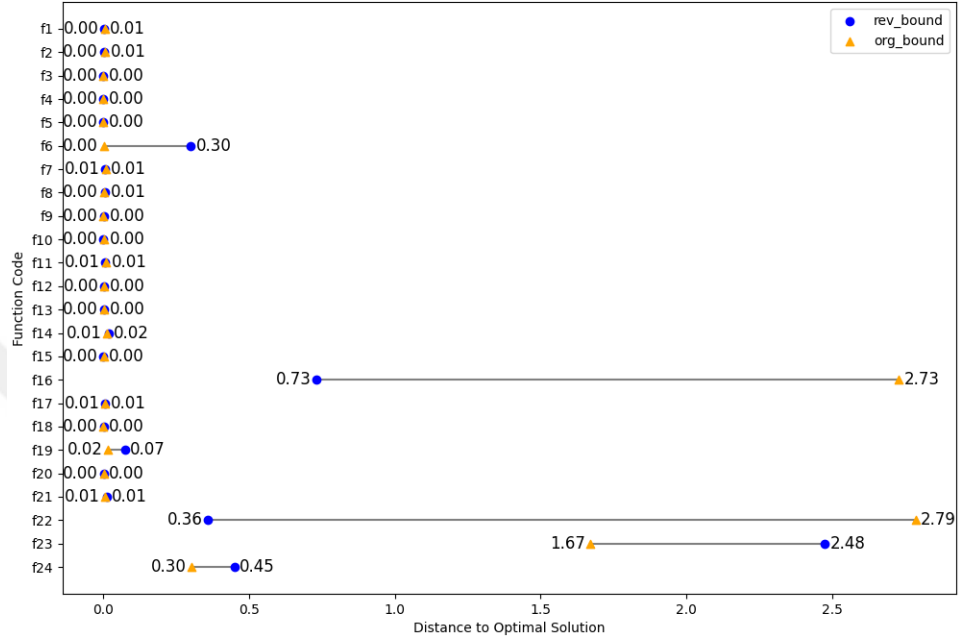


Figure 5.18. CRS2_LM-max_depth=8-pair_wise=0.0.

Figure 5.18 for the CRS2_LM solver illustrates the solver's performance with and without the generated bounds provided by our regression tree algorithm. Out of the 24 functions, 14 instances show an improvement with the generated bounds, as indicated by the blue dots being closer to the optimal solution than the orange dots. However, for functions f6, f19, f23, and f24, our algorithm-generated bounds do not improve the solver's performance; in fact, the results are slightly worse. For example, in function f6, the distance to the optimal solution increases from 0.00 (orange triangle) to 0.30 (blue dot). Similarly, for function f23, the distance increases from 1.67 to 2.48, indicating that the generated bounds were not effective in these cases. Despite these exceptions, most functions still benefit from the generated bounds. For instance, function f22 shows a clear improvement with the generated bounds, reducing the distance to the optimal solution from 2.79 to 0.36. This indicates that the algorithm-generated bounds help to refine the search space considerably, making it easier for the solver to converge towards the optimal solution. In summary, while the CRS2_LM solver benefits from the

regression tree-generated bounds across most of the test functions, certain functions like f6, f19, f23, and f24, experience a slight decline in performance. This highlights the variability in the effectiveness of the generated bounds, suggesting that further refinement may be necessary to ensure consistent improvements across all functions.

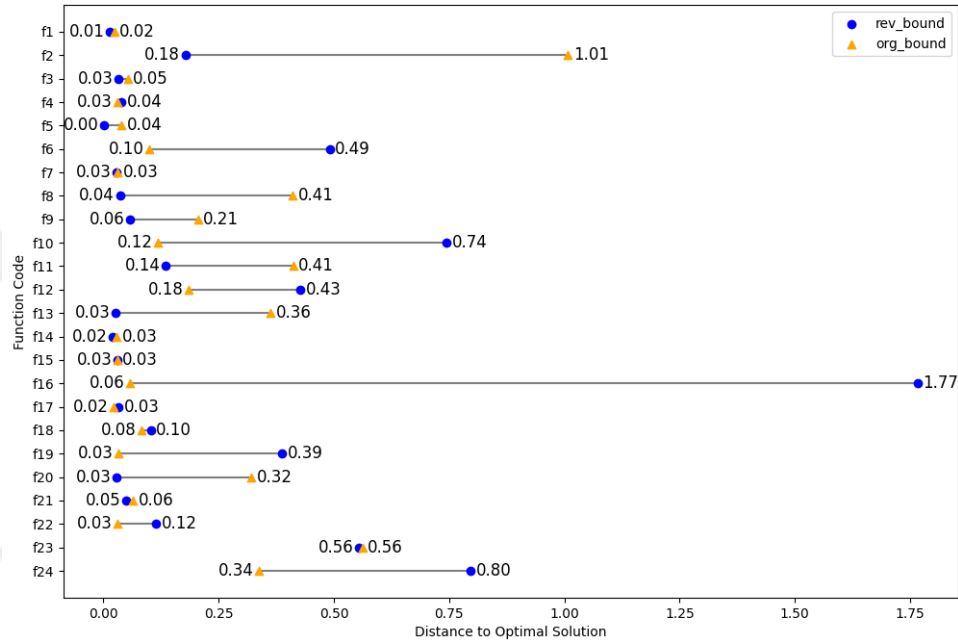


Figure 5.19. ISRES-max_depth=8-pair_wise=0.3.

The graph for the ISRES solver(5.19) demonstrates the comparative performance with and without the regression tree-generated bounds across the 24 test functions. The regression tree-generated bounds improve the performance for most functions, specifically for 14 out of the 24 functions. For functions f2, f6, f8, f9, f11, f13, and f20 the performance improves as the blue dots are closer to the optimal solution than the orange triangles, showing that our algorithm-generated bounds effectively narrowed the search space and improved the solver's performance. However, for function f16, the performance slightly deteriorates as the distance to the optimal solution increases from 0.06 (orange triangle) to 1.77 (blue dot), indicating that the generated bounds were not effective in this case because the bounds do not cover the optimal solution. A significant observation is for function f24, where the generated bounds worsen the resulting solution, increasing the distance to the optimal solution from 0.30 (orange

triangle) to 0.45 (blue dot). This is the only solver among the solvers whose result is worse with the generated bounds. In contrast, the CRS2_LM solver reaches a distance of 0.19 with our tree-generated bounds, demonstrating a better performance. Overall, the ISRES solver benefits from the regression tree-generated bounds across most functions, although there are a few instances where the bounds do not lead to improvements, indicating a need for further refinement in those cases.

To conclude, the analysis highlights that while the generated bounds often cover the optimal solution, the solver’s performance can still degrade due to its inherent ability and dynamics, as observed with function f20 in NELDERMEAD. Conversely, even when the bounds do not encompass the optimal solution, solvers like COBYLA, CRS2_LM, DIRECT, NELDERMEAD, and SBPLX can perform well by guiding the search towards promising regions for good local optima or near-optima, as seen with function f16. These findings underscore the complexity of solver behavior in response to bound constraints and the importance of understanding solver-specific dynamics for optimization.

6. CONCLUSIONS

Optimization problems can be particularly challenging due to their complex, high-dimensional search spaces, the presence of numerous local optima, and the computational intensity required to find globally optimal solutions within a reasonable time. In this study, we focused on developing and evaluating innovative algorithms for optimization problems by utilizing tree-based machine learning algorithms. Our study explored three distinct algorithms: the “Remove-Add Algorithm Augmented by Population Management,” the “Variable Fixing Algorithm,” and the “Learning-Based Bounds Generating Algorithm.” Each algorithm was designed to address specific challenges in optimization, using different methodologies and strategies to enhance solution quality and computational efficiency. Through comprehensive experiments and comparative analyses, we have demonstrated the effectiveness of these algorithms in improving optimization performance.

Our developed “Remove-Add Algorithm Augmented by Population Management” integrates data mining techniques with optimization within a population-based heuristic framework. We demonstrated the algorithm’s viability on the well-known p -median problem and the Quadratic Assignment Problem (QAP). A key innovation of our approach is the use of learning-based solution space partitioning to identify regions of high quality and high variability, which enhances the guidance provided during the search process. Specifically, we employ a regression tree not only to characterize the response surface without requiring explicit functional knowledge, but also to gain deeper insight into the structure of the discrete solution space. Unlike traditional heuristics that rely on predefined rules or random selection, our method incorporates a learning-based local search and population management framework, allowing for more intelligent and adaptive decision-making during the search process.

Parameter setting plays a crucial role in the algorithm’s performance, with the number of solutions to be removed and added, $PopAddRem = 4$, yielding better results

in a shorter time. After optimizing this parameter, we compared our results with the MDM-GRASP method presented in [20]. Extensive computational results on ORLIB benchmark instances (Pmed1-40) showed that the proposed “Remove-Add Algorithm Augmented by Population Management” significantly outperforms the Base and MDM-GRASP methods. For QAP, our algorithm performs slightly better than the Base algorithm. Incorporating problem-specific features enhances the exploitation of the solution space by learning more effectively from data generated during iterations. This improvement in learning capability translates to better solution quality for challenging problem instances, as evidenced by our computational experiments with p -median and QAP instances. Furthermore, we demonstrate that utilizing a regression tree in the population management and local search components of population-based heuristics leads to superior performance.

Beyond p -median and QAP, the proposed algorithm can be extended to other combinatorial optimization problems, such as the Knapsack Problem. Since our approach employs a binary solution representation, it can efficiently explore solution spaces by encoding items as binary variables (1 for inclusion, 0 for exclusion). However, adapting the algorithm to problems with strict feasibility constraints, such as knapsack capacity limits, requires additional considerations. To ensure feasibility, a penalty cost can be incorporated into the objective function to discourage infeasible solutions. Alternatively, a repair mechanism can be introduced to adjust solutions dynamically, ensuring constraints are satisfied while maintaining solution quality. These adaptations would enable the algorithm to handle constrained combinatorial problems effectively, expanding its applicability beyond the tested domains.

The “Variable Fixing Algorithm” utilizes regression tree rule sets to fix variables in optimization problems, focusing on active learning and worst leaf node path split decisions. A central contribution of the algorithm is the use of regression trees to gain deeper insight into the structure of the discrete solution space, allowing for more informed and data-driven variable fixing. In scenarios such as “AL_AllPop_3Stop” and “AL_HalfPop_3Stop”, the algorithm generates random solutions based on an Active

Learning methodology, evaluates them using the regression tree, and appends high-variance solutions to the population. This learning-based population refinement process effectively eliminates poor-quality solutions and guides the search toward promising regions, even in the absence of explicit knowledge about the objective function. The integration of Active Learning with regression trees supports the exploration of diverse areas within the solution space, increasing the likelihood of discovering high-quality solutions. This method enhances solution diversity and quality, promotes the exploration of new solution areas, prevents premature convergence, and improves robustness. The stopping condition halts iterations without improvement after three successive iterations, ensuring computational efficiency and effective resource utilization. The results show that making use of the entire population and active learning yields significant performance gains, particularly in “AL_AllPop_3Stop”, which outperformed Gurobi in multiple instances.

The “Learning-Based Bounds Generating Algorithm” utilizes regression tree techniques to generate bounds for optimization problems. This algorithm utilizes historical data and learning models to predict bounds that guide the optimization process. By incorporating learned bounds, the algorithm narrows the search space, reduces computational effort, and accelerates convergence to optimal or near-optimal solutions. A key strength of this method is that it does not require derivatives or explicit knowledge of the objective function, making it particularly suitable for black-box optimization problems. Moreover, to the best of our knowledge, no prior heuristic-based framework in the literature has introduced a learning-based approach for addressing multi-modal black-box test instances in derivative-free optimization, highlighting the novelty of our contribution.

Our method involves generating bounds using regression trees with varying depths and comparing their effectiveness against original bounds in optimization tasks. The results show that for many functions, particularly those with more complex landscapes, the deeper tree (max_depth=8) performs better, bringing the solutions closer to the optimal. However, the performance varies across different functions, indicating that

the effectiveness of the generated bounds depends on the specific characteristics of the optimization landscape. The analysis reveals that while the generated bounds often include the optimal solution, solver performance can still vary due to inherent solver dynamics, as seen at function f20 in NELDERMEAD. Conversely, solvers like COBYLA, CRS2_LM, DIRECT, NELDERMEAD, and SBPLX can perform well even if the bounds do not cover the optimal solution by guiding the search towards promising regions for good local optima, as observed with function f16. These findings highlight the complexity of solver behavior in response to bound constraints and emphasize the importance of understanding solver-specific dynamics for effective optimization. Overall, the deeper regression trees generally enhance the algorithm’s ability to generate more accurate bounds, demonstrating the potential of our approach in improving solution quality for complex optimization problems.

In conclusion, our research presents significant advancements in the field of optimization through the development and evaluation of three innovative algorithms: the “Remove-Add Algorithm Augmented by Population Management,” the “Variable Fixing Algorithm,” and the “Learning-Based Bounds Generating Algorithm.” Each algorithm addresses specific challenges within optimization, employing tree-based machine learning techniques to enhance solution quality and computational efficiency. Our comprehensive experiments and comparative analyses demonstrate the robustness and effectiveness of these methodologies in solving complex optimization problems, outperforming traditional methods in several instances.

Each algorithm serves as a framework that can be adapted for both well-known heuristics and exact solution methods, provided that applicability conditions are met. The flexibility of these frameworks allows them to be tailored to various problem domains, offering a scalable approach to optimization. Moreover, our study underscores the synergy between machine learning and heuristic optimization, which leverages data-driven insights to refine decision-making processes. Fundamentally, the frameworks introduced in this study are based on population-based heuristics, as training a regression tree necessitates a dataset, and in our case, the solution set itself serves as the dataset.

This self-learning mechanism allows the algorithm to iteratively refine its performance, utilizing insights gained during optimization to improve solution quality dynamically. By integrating machine learning techniques with traditional optimization frameworks, we not only enhance computational efficiency but also provide a systematic way to explore and exploit solution spaces effectively.

The findings of this research pave the way for further advancements in adaptive optimization methodologies, opening avenues for future studies to extend these algorithms to broader combinatorial and continuous optimization problems. One promising direction is the “Remove-Add Algorithm Augmented by Population Management”, which can be further explored by integrating it with well-known population-based heuristics in local search and population management phases. A dynamic mechanism for selecting solutions from different leaf nodes—such as the *Best*, *SecBest*, or *MaxVar*—can be designed to enhance performance based on problem specifications. Future work can investigate adaptive strategies to optimize this selection process dynamically. Additionally, “Variable Fixing Algorithm” warrants further investigation to better understand solver-specific dynamics and their interactions with algorithm-generated constraints. Studying how different solvers respond to fixed variables under various conditions can provide insights into optimizing computational efficiency while maintaining solution accuracy. Another potential enhancement involves “Learning-Based Variable Bound Generation Algorithm,” where adaptivity can be improved by dynamically adjusting the regression tree’s depth and structure according to the characteristics of the optimization landscape during runtime. This would allow for a more refined and flexible approach to constraint generation, potentially improving solution quality and convergence speed. Overall, advancing these methodologies through adaptive mechanisms and solver-specific optimizations presents valuable research opportunities for developing more robust and intelligent optimization techniques.

REFERENCES

1. Sorensen, K. and F. Glover, *Metaheuristics*, Encyclopedia of Operations Research and Management Science, Springer, New York, NY, 2013.
2. H. Osman, I. and G. Laporte, “Metaheuristics: A Bibliography”, *Annals of Operations Research*, Vol. 63, No. 5, pp. 513–623, 1996.
3. Talbi, E.-G., “A Taxonomy of Hybrid Metaheuristics”, *Journal of Heuristics*, Vol. 8, No. 5, pp. 541–564, 2002.
4. Cheng, S., B. Liu, T. Ting, Q. Qin, Y. Shi and K. Huang, “Survey on Data Science with Population-Based Algorithms”, *Big Data Analytics*, Vol. 1, No. 3, pp. 1–20, 2016.
5. James, G., D. Witten, T. Hastie and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*, Springer, New York, NY, 2021.
6. Talbi, E.-G., “Machine Learning into Metaheuristics: A Survey and Taxonomy”, *ACM Computing Surveys*, Vol. 54, No. 6, pp. 1–32, 2021.
7. Loshchilov, I., M. Schoenauer and M. Sebag, “Self-Adaptive Surrogate-Assisted Covariance Matrix Adaptation Evolution Strategy”, *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pp. 321–328, GECCO, Association for Computing Machinery, New York, NY, USA, 2012.
8. Wu, M., K. Li, S. Kwong, Q. Zhang and J. Zhang, “Learning to Decompose: A Paradigm for Decomposition-Based Multiobjective Optimization”, *IEEE Transactions on Evolutionary Computation*, Vol. 23, No. 3, pp. 376–390, 2018.
9. Sakamoto, N., E. Semmatsu, K. Fukuchi, J. Sakuma and Y. Akimoto, “Deep Generative Model for Non-Convex Constraint Handling”, *Proceedings of the Genetic*

- and Evolutionary Computation Conference (GECCO)*, pp. 636–644, GECCO, New York, NY, USA, 2020.
10. Bischl, B., O. Mersmann, H. Trautmann and M. Preuß, “Algorithm Selection Based on Exploratory Landscape Analysis and Cost-Sensitive Learning”, *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pp. 313–320, GECCO, New York, NY, USA, 2012.
 11. Gupta, J., R. Sexton and E. Tunc, “Selecting Scheduling Heuristics Using Neural Networks”, *INFORMS J. Comput.*, Vol. 12, No. 2, pp. 150–162, 2000.
 12. Gutierrez-Rodríguez, A., S. Conant-Pablos, J. Ortiz-Bayliss and H. Terashima-Marín, “Selecting Meta-Heuristics for Solving Vehicle Routing Problems with Time Windows via Meta-Learning”, *Expert Systems with Applications*, Vol. 118, No. March, pp. 470–481, 2019.
 13. Wicaksono, A. S. and A. A. Supianto, “Hyperparameter Optimization Using Genetic Algorithm on Machine Learning Methods for Online News Popularity Prediction”, *International Journal of Advanced Computer Science and Applications*, Vol. 9, No. 12, pp. 438–445, 2018.
 14. Aghaabbasi, M., M. Ali, M. Jasiński, Z. Leonowicz and T. Novák, “On Hyperparameter Optimization of Machine Learning Methods Using a Bayesian Optimization Algorithm to Predict Work Travel Mode Choice”, *IEEE Access*, Vol. 11, pp. 19762–19774, 2023.
 15. Bottou, L., F. Curtis and J. Nocedal, “Optimization Methods for Large-Scale Machine Learning”, *SIAM Review*, Vol. 60, No. 2, pp. 223–311, 2018.
 16. Dhaenens, C. and L. Jourdan, *Metaheuristics for Big Data*, John Wiley & Sons, London, UK, 2016.
 17. Jourdan, L., C. Dhaenens and E.-G. Talbi, “Use Data Mining Techniques to Help

- Metaheuristics: A Short Survey”, *Hybrid Metaheuristics*, pp. 57–69, 2006.
18. Fogel, D. B., *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, Wiley-IEEE Press, Piscataway, New Jersey, 1995.
 19. Lin, L. and M. Gen, “Hybrid Evolutionary Optimisation with Learning for Production Scheduling: State-of-the-Art Survey on Algorithms and Applications”, *International Journal of Production Research*, Vol. 56, pp. 1–31, 2018.
 20. Plastino, A., R. Fuchshuber, S. d. L. Martins, A. A. Freitas and S. Salhi, “A Hybrid Data Mining Metaheuristic for the P-Median Problem”, *Statistical Analysis and Data Mining: The ASA Data Science Journal*, Vol. 4, pp. 313–335, 2011.
 21. Guerine, M., I. Rosseti and A. Plastino, “Extending the Hybridization of Metaheuristics with Data Mining: Dealing with Sequences”, *Intelligent Data Analysis*, Vol. 20, No. 5, pp. 1133–1156, 2016.
 22. Arnold, F. and K. Sorensen, “What Makes a VRP Solution Good? The Generation of Problem-Specific Knowledge for Heuristics”, *Computers & Operations Research*, Vol. 106, pp. 280–288, 2019.
 23. Y. Zhou, J. H. and B. Duval, “Frequent Pattern-Based Search: A Case Study on the Quadratic Assignment Problem”, *IEEE Transactions on Systems, Man. and Cybernetics: Systems*, Vol. 52, pp. 1503–1515, 2022.
 24. Adibi, M. E. and J. Shahrabi, “A Clustering-Based Modified Variable Neighborhood Search Algorithm for a Dynamic Job Shop Scheduling Problem”, *International Journal of Advanced Manufacturing Technology*, Vol. 70, pp. 1955–1961, 2014.
 25. Nasiri, M. M., S. Salesi, A. Rahbari, N. S. Meydani and M. Abdollahi, “A Data Mining Approach for Population-Based Methods to Solve the JSSP”, *Soft Computing*, pp. 1–16, 2018.

26. Al-Duoli, F. and G. Rabadi, “Data Mining Based Hybridization of Meta-RaPS”, *Procedia Computer Science*, Vol. 36, pp. 301–307, 2014.
27. Lessmann, S., M. Caserta and I. M. Arango, “Tuning Metaheuristics: A Data Mining Based Approach for Particle Swarm Optimization”, *Expert Systems with Applications*, Vol. 38, pp. 12826–12838, 2011.
28. Ribeiro, M. H., A. Plastino and S. L. Martins, “Hybridization of GRASP Metaheuristic with Data Mining Techniques”, *Journal of Mathematical Modelling and Algorithms*, Vol. 5, pp. 23–41, 2006.
29. Santos, L. F., M. H. Ribeiro, A. Plastino and S. L. Martins, “A Hybrid GRASP with Data Mining for the Maximum Diversity Problem”, *Hybrid Metaheuristics. Lecture Notes in Computer Science*, Vol. 3636, pp. 116–127, 2005.
30. Santos, L. F., R. Milagres, C. V. Albuquerque, S. Martins and A. Plastino, “A Hybrid GRASP with Data Mining for Efficient Server Replication for Reliable Multicast”, *Proceedings of IEEE GLOBECOM 2006*, IEEE, San Francisco, California, USA, 2006.
31. Reddy, D. S., A. Govardhan and S. Sarma, “Hybridization of Neighbourhood Search Metaheuristic with Data Mining Technique to Solve P-Median Problem”, *International Journal of Computational Engineering Research*, Vol. 2, pp. 161–165, 2012.
32. Barbalho, H., I. Rosseti, S. L. Martins and A. Plastino, “A Hybrid Data Mining GRASP with Path Relinking”, *Computers and Operations Research*, Vol. 40, pp. 3159–3173, 2013.
33. Plastino, A., H. Barbalho, L. F. M. Santos, R. Fuchshuber and S. L. Martins, “Adaptive and Multi-Mining Versions of the DM-GRASP Hybrid Metaheuristic”, *Journal of Heuristics*, Vol. 20, pp. 39–74, 2014.

34. Martins, D., G. M. Vianna, I. Rosseti, S. L. Martins and A. Plastino, “Making a State-of-the-Art Heuristic Faster with Data Mining”, *Annals of Operations Research*, Vol. 263, pp. 141–162, 2018.
35. Raschip, M., C. Croitoru and K. Stoffel, “Using Association Rules to Guide Evolutionary Search in Solving Constraint Satisfaction”, *Proceedings of the 2015 IEEE Congress on Evolutionary Computation (CEC)*, pp. 744–750, IEEE, Sendai, Japan, 2015.
36. Samorani, M., Y. Wang, Z. Lv and F. Glover, “Clustering-Driven Evolutionary Algorithms: An Application of Path Relinking to the Quadratic Unconstrained Binary Optimization Problem”, *Journal of Heuristics*, pp. 1–14, 2019.
37. Kusiak, A., H. Zheng and Z. Song, “Power Optimization of Wind Turbines with Data Mining and Evolutionary Computation”, *Renewable Energy*, Vol. 35, pp. 695–702, 2010.
38. Gaspar-Cunha, A. and A. Vieira, “A Hybrid Multi-Objective Evolutionary Algorithm Using an Inverse Neural Network”, *Proceedings of the First International Workshop on Hybrid Metaheuristics (HM 2004)*, Vol. 3636 of *Lecture Notes in Computer Science*, pp. 25–30, Springer, Valencia, Spain, 2004.
39. Rasheed, K. and H. Hirsh, “Informed Operators: Speeding Up Genetic-Algorithm-Based Design Optimization Using Reduced Models”, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 628–635, GECCO, Morgan Kaufmann, Las Vegas, USA, 2000.
40. Jin, Y. and B. Sendhoff, “Reducing Fitness Evaluations Using Clustering Techniques and Neural Networks Ensembles”, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004)*, pp. 688–699, Springer, Washington, USA, 2004.

41. Kim, H.-S. and S.-B. Cho, “An Efficient Genetic Algorithm with Less Fitness Evaluation by Clustering”, *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pp. 887–894, IEEE, Seoul, Korea, 2001.
42. Yoo, S.-H. and S.-B. Cho, “Partially Evaluated Genetic Algorithm Based on Fuzzy C-Means Algorithm”, *International Conference on Parallel Problem Solving from Nature- PPSN VIII. PPSN 2004. Lecture Notes in Computer Science*, Vol. 3242, pp. 440–449, Springer, 2004.
43. Raschip, M., C. Croitoru and K. Stoffel, “Guiding Evolutionary Search with Association Rules for Solving Weighted CSPs”, *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO 2015)*, pp. 481–488, ACM, Madrid, Spain, 2015.
44. Handa, H., M. Baba, T. Horiuchi and O. Katai, “A Novel Hybrid Framework of Co-evolutionary GA and Machine Learning”, *International Journal of Computational Intelligence and Applications*, Vol. 2, No. 1, pp. 33–52, 2002.
45. Harrath, Y., B. Chebel-Morello and N. Zerhouni, “A Genetic Algorithm and Data Mining Based Meta-Heuristic for Job Shop Scheduling Problem”, *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Vol. 7, pp. 280–285, IEEE, Tunis, Tunisia, 2002.
46. Samorani, M. and M. Laguna, “Data-Mining-Driven Neighborhood Search”, *INFORMS Journal on Computing*, Vol. 24, No. 2, pp. 210–227, 2012.
47. Bengio, Y., A. Lodi and A. Prouvost, “Machine Learning for Combinatorial Optimization: A Methodological Tour D’horizon”, *European Journal of Operational Research*, Vol. 290, No. 2, pp. 405–421, 2021.
48. Vinyals, O., M. Fortunato and N. Jaitly, “Pointer Networks”, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama and R. Garnett (Editors), *Advances in Neural*

Information Processing Systems, Vol. 28, Curran Associates, Inc., 2015.

49. Kruber, M., M. E. Lübbecke and A. Parmentier, “Learning When to Use a Decomposition”, D. Salvagnin and M. Lombardi (Editors), *Integration of AI and OR Techniques in Constraint Programming*, pp. 202–210, Springer International Publishing, Cham, 2017.
50. Bonami, P., A. Lodi and G. Zarpellon, “Learning a Classification of Mixed-Integer Quadratic Programming Problems”, W.-J. van Hoes (Editor), *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 595–604, Springer International Publishing, Cham, 2018.
51. Lodi, A. and G. Zarpellon, “On Learning and Branching: A Survey”, *TOP*, Vol. 25, pp. 207–236, 2017.
52. Khalil, E. B., B. Dilkina, G. L. Nemhauser, S. Ahmed and Y. Shao, “Learning to Run Heuristics in Tree Search”, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 659–666, Melbourne, Australia, 2017.
53. Silva, V., P. Salinas, M. Jackson and C. Pain, “Machine Learning Acceleration for Nonlinear Solvers Applied to Multiphase Porous Media Flow”, *Computer Methods in Applied Mechanics and Engineering*, Vol. 384, p. 113989, 2021.
54. Hakimi, S., “Optimum Locations of Switching Centers and the Absolute Centers and Medians of a Graph”, *Operations Research*, Vol. 12, pp. 450–459, 1964.
55. Koopmans, T. C. and M. J. Beckmann, “Assignment Problems and the Location of Economic Activities.”, *Econometrica*, Vol. 25, pp. 53–76, 1957.
56. Steinberg, L., “The Backboard Wiring Problem: a Placement Algorithm”, *SIAM Review*, Vol. 3, pp. 37–50, 1961.

57. Dickey, J. W. and J. W. Hopkins, "Campus Building Arrangement Using TOPAZ", *Transportation Science*, Vol. 6, pp. 59–68, 1972.
58. Elshafei, A., "Hospital Layout as A Quadratic Assignment Problem", *Operations Research Quarterly*, Vol. 28, pp. 167–179, 1977.
59. Furini, F., E. Traversi, P. Belotti, A. Frangioni, A. Gleixner, N. Gould, L. Liberti, A. Lodi, R. Misener, H. Mittelman, N. Sahinidis, S. Vigerske and A. Wiegele, "QPLIB: A Library of Quadratic Programming Instances", *Mathematical Programming Computation*, Vol. 11, pp. 237–265, 2018.
60. Myers, R. H., D. C. Montgomery and C. M. Anderson-Cook, *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*, Wiley, Hoboken, NJ, 4th edn., 2016.
61. Kumar, L. and S. K. Rath, "Application of Genetic Algorithm as Feature Selection Technique in Development of Effective Fault Prediction Model", *2016 IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics Engineering (UPCON)*, pp. 432–437, IEEE, 2016.
62. Lipton, Z. C., "The Mythos of Model Interpretability: In Machine Learning, The Concept of Interpretability is Both Important and Slippery.", *ACM Queue*, Vol. 16, No. 3, pp. 31–57, 2018.
63. Breiman, L., "Bagging Predictors", *Machine Learning*, Vol. 24, No. 2, pp. 123–140, 1996.
64. Freund, Y. and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting", *Journal of Computer and System Sciences*, Vol. 55, No. 1, pp. 119–139, 1997.
65. Friedman, J. H., "Greedy Function Approximation: A Gradient Boosting Machine", *Annals of Statistics*, Vol. 29, No. 5, pp. 1189–1232, 2001.

66. Halim, Z., M. N. Yousaf, M. Waqas, M. Sulaiman, G. Abbas, M. Hussain, I. Ahmad and M. Hanif, “An Effective Genetic Algorithm-Based Feature Selection Method for Intrusion Detection Systems”, *Computers Security*, Vol. 110, No. C, p. 102448, 2021.
67. Tan, F., X. Fu, Y. Zhang *et al.*, “A Genetic Algorithm-Based Method for Feature Subset Selection”, *Soft Computing*, Vol. 12, No. 2, pp. 111–120, 2008.
68. Han, J., M. Kamber and J. Pei, *Data Mining: Concepts and Techniques*, Morgan Kaufmann, Burlington, MA, 2012.
69. Zhang, J., Z.-h. Zhan, Y. Lin, N. Chen, Y.-j. Gong, J.-h. Zhong, H. S. Chung, Y. Li and Y.-h. Shi, “Evolutionary Computation Meets Machine Learning: A Survey”, *IEEE COMPUTATIONAL INTELLIGENCE MAGAZINE, Review Article*, Vol. 6, No. 4, pp. 68–75, 2011.
70. Santos, H. G., L. S. Ochi, E. H. Marinho and L. M. A. Drummond, “Combining an Evolutionary Algorithm with Data Mining to Solve a Single-Vehicle Routing Problem”, *Neurocomputing*, Vol. 70, pp. 70–77, 2006.
71. Chen, T. Y. and J. H. Huang, “Application of Data Mining in a Global Optimization Algorithm”, *Advances in Engineering Software*, Vol. 66, pp. 24–33, 2013.
72. Shahzad, A. and N. Mebarki, “Data Mining Based Job Dispatching Using Hybrid Simulation-Optimization Approach for Shop Scheduling Problem”, *Engineering Applications of Artificial Intelligence*, Vol. 25, pp. 1173–1181, 2012.
73. Koonce, D. A. and S.-C. Tsai, “Using Data Mining to Find Patterns in Genetic Algorithm Solutions to a Job Shop Schedule”, *Computers & Industrial Engineering*, Vol. 38, pp. 361–374, 2000.
74. Cadenas, J. and J. Verge day, “Towards a New Strategy for Solving Fuzzy Optimization Problems”, *Fuzzy Optimization and Decision Making*, Vol. 8, p. 231–244,

2009.

- 75. Chiang, D. M.-H., C.-P. Lin and M.-C. Chen, “Data Mining Based Storage Assignment Heuristics for Travel Distance Reduction”, *Expert Systems*, Vol. 31, pp. 81–90, 2012.
- 76. Hastie, T., R. Tibshirani and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer, New York, USA, 2009.



APPENDIX A: THE REMOVE-ADD ALGORITHM AUGMENTED BY POPULATION MANAGEMENT

Appendix A presents the detailed results of our experiments, illustrating the proposed algorithm’s performance metrics under various configurations. These tables provide comprehensive insights into the impact of different parameter settings on the algorithm’s effectiveness. For a more in-depth analysis, please refer to the corresponding discussions in the main text.

A.1. Analysis of RegTree algorithm with respect to different techniques

In this section, we compare the performance of the RegTree algorithm across three different techniques: Best, SecBest, and MaxVar. The problem instances are ordered by the p/n_{loc} ratio, and the results are provided in terms of the GAP (%) values and ranks. For each instance, Table A.1 displays the GAP values for the three techniques, as well as the corresponding problem specifications (e.g., number of locations and facilities) and the ranks of each technique. The minimum GAP values for each instance are also shown, reflecting the best performance for each technique. This analysis helps evaluate each technique’s effectiveness across different problem instances.

In addition to comparing raw GAP values, the ranking information offers valuable insights into the consistency of each technique’s performance across varying problem complexities. Techniques that consistently rank higher demonstrate greater robustness and adaptability, especially as the p/n_{loc} ratio increases. The MaxVar approach, for instance, may not always yield the lowest GAP but can provide more stable results in cases with higher solution variability, suggesting its potential for maintaining solution quality in more dynamic or uncertain environments. This complementary use of GAP and rank metrics allows for a more comprehensive assessment of each technique’s strengths and limitations within the RegTree framework.

Table A.1. GAP (%) according to p/n_{loc} ratio for Three Techniques.

Problem	GAP			Problem Specs			Rank			MinGAP
	Best	SecBest	MaxVar	n_loc	p	p/n_loc	Best	SecBest	MaxVar	
PMed10	2.10	2.37	2.52	200	67	0.34	1	2	3	2.10
PMed25	5.72	6.16	7.46	500	167	0.33	1	2	3	5.72
PMed15	3.70	4.79	5.10	300	100	0.33	1	2	3	3.70
PMed30	5.86	6.63	9.03	600	200	0.33	1	2	3	5.86
PMed20	4.41	5.23	6.84	400	133	0.33	1	2	3	4.41
PMed5	0.13	0.58	0.86	100	33	0.33	1	2	3	0.13
PMed4	0.47	0.67	0.79	100	20	0.20	1	2	3	0.47
PMed9	1.24	1.17	1.98	200	40	0.20	2	1	3	1.17
PMed14	1.95	1.88	2.97	300	60	0.20	2	1	3	1.88
PMed19	2.19	2.89	2.91	400	80	0.20	1	2	3	2.19
PMed24	3.02	2.72	5.49	500	100	0.20	2	1	3	2.72
PMed29	3.51	3.40	4.95	600	120	0.20	2	1	3	3.40
PMed34	4.15	5.02	4.73	700	140	0.20	1	3	2	4.15
PMed2	0.22	0.23	0.27	100	10	0.10	1	2	3	0.22
PMed3	0.01	0.17	0.29	100	10	0.10	1	2	3	0.01
PMed8	0.76	0.82	1.46	200	20	0.10	1	2	3	0.76
PMed13	0.99	1.25	2.14	300	30	0.10	1	2	3	0.99
PMed18	1.37	1.58	2.70	400	40	0.10	1	2	3	1.37
PMed23	2.27	2.04	3.13	500	50	0.10	2	1	3	2.04
PMed28	1.63	2.26	3.57	600	60	0.10	1	2	3	1.63
PMed33	2.47	3.15	3.87	700	70	0.10	1	2	3	2.47
PMed37	2.24	3.73	5.32	800	80	0.10	1	2	3	2.24
PMed40	2.56	3.06	5.63	900	90	0.10	1	2	3	2.56
PMed1	0.00	0.00	0.00	100	5	0.05	1	1	1	0.00
PMed7	0.23	0.18	0.54	200	10	0.05	2	1	3	0.18
PMed12	0.63	0.24	2.26	300	10	0.03	2	1	3	0.24
PMed6	0.00	0.12	0.15	200	5	0.03	1	2	3	0.00
PMed17	0.05	0.11	0.38	400	10	0.03	1	2	3	0.05
PMed22	1.13	2.69	1.57	500	10	0.02	1	3	2	1.13
PMed11	0.00	0.05	0.00	300	5	0.02	1	3	1	0.00
PMed27	2.76	1.28	0.62	600	10	0.02	3	2	1	0.62
PMed32	0.29	2.37	1.39	700	10	0.01	1	3	2	0.29
PMed16	0.08	0.05	0.05	400	5	0.01	3	2	1	0.05
PMed36	1.71	2.07	1.35	800	10	0.01	2	3	1	1.35
PMed39	0.37	0.66	1.66	900	10	0.01	1	2	3	0.37
PMed21	0.00	0.60	0.00	500	5	0.01	1	3	1	0.00
PMed26	2.17	1.52	0.07	600	5	0.01	3	2	1	0.07
PMed31	0.96	0.94	0.00	700	5	0.01	3	2	1	0.00
PMed35	0.00	0.88	0.00	800	5	0.01	1	3	2	0.00
PMed38	1.94	0.31	0.08	900	5	0.01	3	2	1	0.08

A.2. RegTree comparison with Base at *PopAddRem* =2

This section presents a comparative analysis of the RegTree algorithm against the Base method, specifically focusing on scenarios where *PopAddRem* is set to 2. Tables A.2 and A.3 provide detailed results across various configurations, highlighting the performance metrics of both methods under consistent conditions. This comparison offers valuable insights into the strengths and potential limitations of the RegTree algorithm relative to the Base method, facilitating a deeper understanding of its applicability in different problem-solving contexts.

Table A.2. Best and Average GAP (%) - RegTree and Base, PopAddRem=2.

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Best	Average	Best	Average
PMed10	200	67	0.34	1.27	2.10	4.78	6.84
PMed25	500	167	0.33	4.10	5.72	9.79	12.34
PMed15	300	100	0.33	2.78	3.70	5.09	8.58
PMed30	600	200	0.33	5.03	5.86	11.81	12.63
PMed20	400	133	0.33	2.74	4.41	8.66	9.66
PMed5	100	33	0.33	0.00	0.13	1.25	2.98
PMed4	100	20	0.20	0.00	0.47	0.00	1.17
PMed9	200	40	0.20	0.62	1.24	0.22	1.38
PMed14	300	60	0.20	0.77	1.95	2.76	3.83
PMed19	400	80	0.20	1.65	2.19	3.62	4.99
PMed24	500	100	0.20	2.06	3.02	4.76	6.38
PMed29	600	120	0.20	2.97	3.51	5.37	6.92
PMed34	700	140	0.20	2.99	4.15	6.64	7.85
PMed2	100	10	0.10	0.00	0.22	0.22	0.39
PMed3	100	10	0.10	0.00	0.01	0.00	0.11
PMed8	200	20	0.10	0.00	0.76	0.20	0.76
PMed13	300	30	0.10	0.55	0.99	0.00	1.18

Table A.2. Best and Average GAP (%) - RegTree and Base,
PopAddRem=2 (cont.).

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Best	Average	Best	Average
PMed18	400	40	0.10	0.40	1.37	1.62	2.88
PMed23	500	50	0.10	1.13	2.27	1.80	2.87
PMed28	600	60	0.10	1.07	1.63	2.22	3.52
PMed33	700	70	0.10	2.06	2.47	3.09	3.54
PMed37	800	80	0.10	1.44	2.24	3.84	5.72
PMed40	900	90	0.10	1.76	2.56	3.08	5.49
PMed1	100	5	0.05	0.00	0.00	0.00	0.00
PMed7	200	10	0.05	0.14	0.23	0.00	0.17
PMed12	300	10	0.03	0.00	0.63	0.00	0.06
PMed6	200	5	0.03	0.00	0.00	0.00	0.00
PMed17	400	10	0.03	0.00	0.05	0.00	0.10
PMed22	500	10	0.02	0.44	1.13	0.00	0.85
PMed11	300	5	0.02	0.00	0.00	0.00	0.00
PMed27	600	10	0.02	1.50	2.76	0.00	0.19
PMed32	700	10	0.01	0.04	0.29	0.00	0.02
PMed16	400	5	0.01	0.00	0.08	0.00	0.00
PMed36	800	10	0.01	0.00	1.71	0.00	0.19
PMed39	900	10	0.01	0.00	0.37	0.00	0.03
PMed21	500	5	0.01	0.00	0.00	0.00	0.00
PMed26	600	5	0.01	0.00	2.17	0.00	0.03
PMed31	700	5	0.01	0.00	0.96	0.00	0.00
PMed35	800	5	0.01	0.00	0.00	0.00	0.00
PMed38	900	5	0.01	0.11	1.94	0.00	0.16

Table A.3. Time and NumOfIt - RegTree and Base, PopAddRem=2.

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Time	NumOfIt	Time	NumOfIt
PMed10	200	67	0.34	4.35	514	2.15	387
PMed25	500	167	0.33	16.99	692	6.80	434
PMed15	300	100	0.33	7.82	592	3.62	403
PMed30	600	200	0.33	23.46	736	13.78	444
PMed20	400	133	0.33	10.73	658	5.20	419
PMed5	100	33	0.33	2.76	447	1.25	374
PMed4	100	20	0.20	2.53	417	1.74	370
PMed9	200	40	0.20	3.69	467	1.83	385
PMed14	300	60	0.20	7.91	546	2.71	401
PMed19	400	80	0.20	7.74	589	3.95	405
PMed24	500	100	0.20	11.17	607	5.34	414
PMed29	600	120	0.20	19.25	700	8.41	424
PMed34	700	140	0.20	22.04	680	9.40	436
PMed2	100	10	0.10	2.43	382	1.06	360
PMed3	100	10	0.10	2.08	384	1.15	365
PMed8	200	20	0.10	3.34	421	1.40	373
PMed13	300	30	0.10	4.16	471	1.85	380
PMed18	400	40	0.10	5.55	497	2.55	389
PMed23	500	50	0.10	6.81	497	3.67	400
PMed28	600	60	0.10	10.92	564	4.99	400
PMed33	700	70	0.10	12.91	561	5.88	407
PMed37	800	80	0.10	19.67	635	7.08	409
PMed40	900	90	0.10	26.54	633	9.68	414
PMed1	100	5	0.05	2.13	375	0.65	358
PMed7	200	10	0.05	2.33	386	1.18	362
PMed12	300	10	0.03	2.83	394	1.38	364
PMed6	200	5	0.03	2.19	373	0.95	358

Table A.3. Time and NumOfIt - RegTree and Base,
PopAddRem=2 (cont.).

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Time	NumOfIt	Time	NumOfIt
PMed17	400	10	0.03	3.07	390	1.61	363
PMed22	500	10	0.02	3.58	392	2.22	366
PMed11	300	5	0.02	2.57	367	1.16	358
PMed27	600	10	0.02	3.77	385	2.39	369
PMed32	700	10	0.01	4.48	405	2.39	366
PMed16	400	5	0.01	3.07	380	1.74	362
PMed36	800	10	0.01	5.06	392	2.74	368
PMed39	900	10	0.01	5.24	387	2.95	363
PMed21	500	5	0.01	2.93	367	1.70	356
PMed26	600	5	0.01	3.08	369	2.13	357
PMed31	700	5	0.01	3.55	369	2.06	358
PMed35	800	5	0.01	3.77	371	2.23	357
PMed38	900	5	0.01	4.71	369	2.63	358

A.3. RegTree comparison with Base at *PopAddRem* =4

This section presents a comparative analysis of the RegTree algorithm against the Base method, specifically focusing on scenarios where *PopAddRem* is set to 4. Tables A.4 and A.5 provide detailed results across various configurations, highlighting the performance metrics of both methods under consistent conditions. This comparison offers valuable insights into the strengths and potential limitations of the RegTree algorithm relative to the Base method, facilitating a deeper understanding of its applicability in different problem-solving contexts.

Table A.4. Best and Average GAP (%) - RegTree and Base, PopAddRem=4.

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Best	Average	Best	Average
PMed10	200	67	0.34	0.40	1.61	1.27	3.93
PMed25	500	167	0.33	3.67	6.87	7.88	10.29
PMed15	300	100	0.33	1.74	2.74	4.92	6.41
PMed30	600	200	0.33	4.42	5.51	9.00	10.74
PMed20	400	133	0.33	1.62	3.99	5.87	7.60
PMed5	100	33	0.33	0.22	0.77	0.00	0.62
PMed4	100	20	0.20	0.00	0.30	0.40	0.71
PMed9	200	40	0.20	0.29	0.56	0.51	1.00
PMed14	300	60	0.20	0.88	1.02	1.01	2.20
PMed19	400	80	0.20	0.46	1.41	1.51	2.87
PMed24	500	100	0.20	1.32	2.06	2.26	3.42
PMed29	600	120	0.20	1.65	2.62	2.04	3.59
PMed34	700	140	0.20	2.69	3.30	3.98	5.50
PMed2	100	10	0.10	0.00	0.20	0.22	0.30
PMed3	100	10	0.10	0.00	0.01	0.00	0.11
PMed8	200	20	0.10	0.20	0.60	0.20	0.54
PMed13	300	30	0.10	0.00	0.39	0.05	0.43
PMed18	400	40	0.10	0.17	0.43	0.25	0.83
PMed23	500	50	0.10	0.30	0.54	0.56	0.78
PMed28	600	60	0.10	0.44	0.68	0.27	0.69
PMed33	700	70	0.10	0.62	1.06	0.83	1.28
PMed37	800	80	0.10	0.75	1.04	1.27	2.44
PMed40	900	90	0.10	1.15	1.58	0.86	2.02
PMed1	100	5	0.05	0.00	0.00	0.00	0.00
PMed7	200	10	0.05	0.14	0.36	0.00	0.16
PMed12	300	10	0.03	0.00	0.00	0.00	0.11
PMed6	200	5	0.03	0.00	0.00	0.00	0.00

Table A.4. Best and Average GAP (%) - RegTree and Base,
PopAddRem=4 (cont.).

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Best	Average	Best	Average
PMed17	400	10	0.03	0.00	0.04	0.00	0.20
PMed22	500	10	0.02	0.00	0.51	0.00	0.85
PMed11	300	5	0.02	0.00	0.05	0.00	0.00
PMed27	600	10	0.02	0.04	0.18	0.00	0.04
PMed32	700	10	0.01	0.00	0.08	0.00	0.02
PMed16	400	5	0.01	0.00	0.06	0.00	0.10
PMed36	800	10	0.01	0.00	0.17	0.00	0.13
PMed39	900	10	0.01	0.00	0.07	0.00	0.00
PMed21	500	5	0.01	0.00	0.39	0.00	0.00
PMed26	600	5	0.01	0.00	0.00	0.00	0.03
PMed31	700	5	0.01	0.00	0.00	0.00	0.00
PMed35	800	5	0.01	0.00	0.30	0.00	0.00
PMed38	900	5	0.01	0.00	0.00	0.00	0.22

Table A.5. Time and NumOfIt for RegTree and Base, PopAddRem=4.

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Time	NumOfIt	Time	NumOfIt
PMed10	200	67	0.34	3.25	504	3.10	403
PMed25	500	167	0.33	11.62	670	10.35	443
PMed15	300	100	0.33	5.75	597	4.98	416
PMed30	600	200	0.33	17.06	728	14.00	457
PMed20	400	133	0.33	9.93	649	8.79	433
PMed5	100	33	0.33	2.13	428	1.80	379
PMed4	100	20	0.20	1.99	417	1.47	373

Table A.5. Time and NumOfIt for RegTree and Base,
PopAddRem=4 (cont.).

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Time	NumOfIt	Time	NumOfIt
PMed9	200	40	0.20	2.84	460	2.42	390
PMed14	300	60	0.20	4.38	553	3.61	408
PMed19	400	80	0.20	6.64	597	5.37	425
PMed24	500	100	0.20	9.31	629	7.35	430
PMed29	600	120	0.20	12.09	690	9.50	445
PMed34	700	140	0.20	18.15	727	13.72	456
PMed2	100	10	0.10	1.75	379	1.15	367
PMed3	100	10	0.10	1.85	386	0.90	362
PMed8	200	20	0.10	2.26	432	1.67	373
PMed13	300	30	0.10	3.11	460	1.79	383
PMed18	400	40	0.10	4.16	507	3.35	397
PMed23	500	50	0.10	4.50	523	4.44	408
PMed28	600	60	0.10	6.47	561	4.79	412
PMed33	700	70	0.10	9.15	599	6.15	423
PMed37	800	80	0.10	13.20	652	9.22	425
PMed40	900	90	0.10	15.01	640	10.88	442
PMed1	100	5	0.05	1.26	370	0.87	358
PMed7	200	10	0.05	1.74	389	0.96	363
PMed12	300	10	0.03	1.73	394	1.85	364
PMed6	200	5	0.03	1.74	375	1.18	359
PMed17	400	10	0.03	2.58	396	2.05	365
PMed22	500	10	0.02	2.51	396	1.82	369
PMed11	300	5	0.02	1.99	368	1.53	360
PMed27	600	10	0.02	2.46	387	2.10	372
PMed32	700	10	0.01	2.76	401	2.44	368

Table A.5. Time and NumOfIt for RegTree and Base,
PopAddRem=4 (cont.).

Problem	Problem Specs			RegTree		Base	
	n_loc	p	p/n_loc	Time	NumOfIt	Time	NumOfIt
PMed16	400	5	0.01	1.72	375	1.33	360
PMed36	800	10	0.01	2.88	391	2.87	368
PMed39	900	10	0.01	2.91	387	3.62	368
PMed21	500	5	0.01	2.34	366	2.01	358
PMed26	600	5	0.01	1.96	365	2.55	361
PMed31	700	5	0.01	1.87	366	2.57	361
PMed35	800	5	0.01	3.25	367	2.30	362
PMed38	900	5	0.01	2.59	373	2.70	360

A.4. Paper [20] Results

In their study [20], they addressed the same problem instances explored in this research, employing similar methodologies that integrate ML algorithms within heuristic frameworks. Detailed descriptions of their approaches and findings are provided in the main text. For a comparative perspective, the results from this study are summarized in Table A.6.

Table A.6. Best and Average GAP(%) Values for article Plastino2011.

Problem	Problem Specs			GRASP		DM-GRASP		MDM-GRASP	
	n_loc	p	p/n_loc	Best	Average	Best	Average	Best	Average
PMed10	200	67	0.34	12.03	14.33	4.86	7.32	4.14	6.57
PMed25	500	167	0.33	11.38	13.38	7.93	9.11	4.81	6.90
PMed15	300	100	0.33	13.07	14.23	6.65	9.14	6.36	7.43
PMed30	600	200	0.33	12.62	13.36	6.84	8.12	4.32	5.59
PMed20	400	133	0.33	9.00	10.93	6.88	7.80	4.02	5.37
PMed5	100	33	0.33	4.65	7.99	3.76	5.12	0.81	3.66
PMed4	100	20	0.20	1.91	3.21	0.69	1.60	0.73	1.34
PMed9	200	40	0.20	5.38	6.32	3.95	4.85	2.38	3.50
PMed14	300	60	0.20	6.64	7.52	2.96	4.64	2.46	3.31
PMed19	400	80	0.20	5.55	6.54	3.66	4.12	2.74	3.74
PMed24	500	100	0.20	5.84	7.16	2.90	4.32	2.77	3.12
PMed29	600	120	0.20	7.55	8.21	3.89	5.23	3.17	3.99
PMed34	700	140	0.20	8.16	8.95	3.95	5.28	2.95	4.13
PMed2	100	10	0.10	0.00	0.14	0.00	0.00	0.00	0.00
PMed3	100	10	0.10	0.00	0.00	0.00	0.00	0.00	0.00
PMed8	200	20	0.10	1.46	2.35	0.07	1.38	0.27	1.17
PMed13	300	30	0.10	2.58	3.31	0.87	1.95	1.01	1.84
PMed18	400	40	0.10	1.66	2.75	1.04	1.66	1.02	1.52
PMed23	500	50	0.10	2.92	3.56	1.67	2.38	1.52	1.98
PMed28	600	60	0.10	2.80	3.49	2.13	2.41	1.22	1.68
PMed33	700	70	0.10	3.38	3.71	1.51	2.54	1.28	2.17
PMed37	800	80	0.10	2.81	3.49	1.62	2.24	1.27	1.69
PMed40	900	90	0.10	3.84	4.10	2.01	2.63	1.56	2.08
PMed1	100	5	0.05	0.00	0.00	0.00	0.00	0.00	0.00
PMed7	200	10	0.05	0.20	0.59	0.00	0.24	0.00	0.18
PMed12	300	10	0.03	0.00	0.17	0.00	0.00	0.00	0.00
PMed6	200	5	0.03	0.00	0.17	0.00	0.00	0.00	0.05
PMed17	400	10	0.03	0.00	0.21	0.00	0.06	0.00	0.00
PMed22	500	10	0.02	0.14	0.99	0.00	0.35	0.00	0.10
PMed11	300	5	0.02	0.00	0.00	0.00	0.00	0.00	0.00
PMed27	600	10	0.02	0.35	0.61	0.00	0.10	0.00	0.01
PMed32	700	10	0.01	0.73	1.26	0.00	0.13	0.00	0.01
PMed16	400	5	0.01	0.00	0.00	0.00	0.00	0.00	0.00
PMed36	800	10	0.01	0.38	0.71	0.13	0.32	0.00	0.06
PMed39	900	10	0.01	0.50	0.76	0.00	0.12	0.00	0.03
PMed21	500	5	0.01	0.00	0.00	0.00	0.00	0.00	0.00
PMed26	600	5	0.01	0.00	0.00	0.00	0.00	0.00	0.00
PMed31	700	5	0.01	0.00	0.00	0.00	0.00	0.00	0.00
PMed35	800	5	0.01	0.00	0.07	0.00	0.00	0.00	0.00
PMed38	900	5	0.01	0.00	0.04	0.00	0.00	0.00	0.00

A.5. RegTree added Three New Problem Specific Features

In this section, we compare the performance of three configurations—RegTree, Base, and RegTree enhanced with three problem-specific features—by presenting the Best and Average GAP values at $PopAddRem=2$. The inclusion of these features in RegTree aims to tailor the algorithm more closely to the problem domain, potentially improving its effectiveness. Tables A.7 and A.8 summarize the best and average GAP values across various problem instances, respectively.

Table A.7. RegTree, Base, and RegTree+3Features Best GAP (%) Values.

Instance	n_loc	p	p/n_loc	RegTree	Base	RegTree+3Features
PMed10	200	67	0.335	0.96	1.67	1.75
PMed25	500	167	0.334	5.63	7.55	5.36
PMed15	300	100	0.333	1.74	1.97	1.62
PMed30	600	200	0.333	6.44	9.05	4.02
PMed20	400	133	0.333	3.02	5.93	4.36
PMed5	100	33	0.33	0.15	0.15	0.22
PMed4	100	20	0.2	0.40	0.00	0.13
PMed9	200	40	0.2	0.69	0.22	0.22
PMed14	300	60	0.2	1.11	1.01	1.08
PMed19	400	80	0.2	1.12	2.04	1.55
PMed24	500	100	0.2	1.93	3.41	0.98
PMed29	600	120	0.2	1.95	2.31	2.47
PMed34	700	140	0.2	2.49	4.81	2.26
PMed2	100	10	0.1	0.22	0.22	0.22
PMed3	100	10	0.1	0.00	0.00	0.00
PMed8	200	20	0.1	0.27	0.00	0.00
PMed13	300	30	0.1	0.16	0.32	0.16
PMed18	400	40	0.1	0.15	0.33	0.50

Table A.7. RegTree, Base, and RegTree+3Features Best GAP (%) Values (cont.).

Instance	n_loc	p	p/n_loc	RegTree	Base	RegTree+3Features
PMed23	500	50	0.1	0.78	0.71	0.28
PMed28	600	60	0.1	0.84	0.76	0.27
PMed33	700	70	0.1	0.72	0.74	0.77
PMed37	800	80	0.1	1.03	1.27	0.83
PMed40	900	90	0.1	1.97	0.72	1.52
PMed1	100	5	0.05	0.00	0.00	0.00
PMed7	200	10	0.05	0.00	0.00	0.00
PMed12	300	10	0.033	0.00	0.00	0.00
PMed6	200	5	0.025	0.00	0.00	0.00
PMed17	400	10	0.025	0.00	0.00	0.00
PMed22	500	10	0.02	0.00	0.00	0.00
PMed11	300	5	0.017	0.00	0.00	0.00
PMed27	600	10	0.017	0.00	0.00	0.00
PMed32	700	10	0.014	0.00	0.00	0.00
PMed16	400	5	0.013	0.00	0.00	0.00
PMed36	800	10	0.013	0.00	0.00	0.00
PMed39	900	10	0.011	0.00	0.00	0.00
PMed21	500	5	0.01	0.00	0.00	0.00
PMed26	600	5	0.008	0.00	0.00	0.00
PMed31	700	5	0.007	0.00	0.00	0.00
PMed35	800	5	0.006	0.00	0.00	0.00
PMed38	900	5	0.006	0.00	0.00	0.00

Table A.8. RegTree, Base, and RegTree+3Features Average GAP (%) Values.

Instance	n_loc	p	p/n_loc	RegTree	Base	RegTree+3Features
PMed10	200	67	0.34	2.10	2.77	2.69
PMed25	500	167	0.33	7.22	9.34	6.50
PMed15	300	100	0.33	3.75	6.78	2.51
PMed30	600	200	0.33	7.65	10.19	5.85
PMed20	400	133	0.33	4.86	7.60	5.06
PMed5	100	33	0.33	0.49	0.87	0.96
PMed4	100	20	0.20	0.75	0.75	0.37
PMed9	200	40	0.20	1.15	0.89	0.64
PMed14	300	60	0.20	2.34	2.43	1.27
PMed19	400	80	0.20	2.54	2.88	2.33
PMed24	500	100	0.20	3.09	4.44	2.53
PMed29	600	120	0.20	3.22	4.21	3.13
PMed34	700	140	0.20	3.60	5.79	3.28
PMed2	100	10	0.10	0.45	0.34	0.26
PMed3	100	10	0.10	0.19	0.13	0.01
PMed8	200	20	0.10	0.80	0.28	0.22
PMed13	300	30	0.10	0.91	0.83	1.06
PMed18	400	40	0.10	0.52	0.68	0.92
PMed23	500	50	0.10	1.12	0.93	0.73
PMed28	600	60	0.10	1.60	1.02	1.02
PMed33	700	70	0.10	1.72	1.35	1.17
PMed37	800	80	0.10	2.60	2.10	1.96
PMed40	900	90	0.10	2.26	2.08	2.28
PMed1	100	5	0.05	0.00	0.00	0.00
PMed7	200	10	0.05	0.08	0.13	0.08
PMed12	300	10	0.03	0.15	0.07	0.04
PMed6	200	5	0.03	0.00	0.00	0.00
PMed17	400	10	0.03	0.18	0.37	0.25

Table A.8. RegTree, Base, and RegTree+3Features Average GAP (%) Values (cont.).

Instance	n_loc	p	p/n_loc	RegTree	Base	RegTree+3Features
PMed22	500	10	0.02	0.44	0.55	0.65
PMed11	300	5	0.02	0.00	0.00	0.00
PMed27	600	10	0.02	0.12	0.14	0.07
PMed32	700	10	0.01	0.04	0.05	0.02
PMed16	400	5	0.01	0.00	0.05	0.05
PMed36	800	10	0.01	0.31	0.23	0.23
PMed39	900	10	0.01	0.01	0.00	0.09
PMed21	500	5	0.01	0.00	0.00	0.00
PMed26	600	5	0.01	0.00	0.03	0.03
PMed31	700	5	0.01	0.00	0.00	0.02
PMed35	800	5	0.01	0.00	0.23	0.00
PMed38	900	5	0.01	0.00	0.00	0.00

A.6. QAP Results for thirteen selected problem instances

In this section, we present a comparative analysis of the RegTree algorithm, the Base algorithm, and RegTree enhanced with three problem-specific features, applied to the Quadratic Assignment Problem (QAP). The comparison focuses on instances where PopAddRem equals 2, evaluating performance based on Best GAP, Average GAP, solution time, and the number of iterations.

Table A.9: QAP Results for thirteen selected problem instances

Instance	n	RegTree				Base				RegTree+3Features			
		BestGAP	AveGAP	Time	NumOfIt	BestGAP	AveGAP	Time	NumOfIt	BestGAP	AveGAP	Time	NumOfIt
chr12a	12	5.7	17.144	0.89	386.8	17.76	28.398	0.18	360.8	17.42	25	0.82	391.6
chr12b	12	0	14.822	0.92	392.6	7.88	27.088	0.18	360.4	3.7	13.418	0.87	393
chr15a	15	3.29	36.064	1.04	399.8	22.7	42.89	0.24	362.8	20.45	38.908	0.98	397.4
chr15b	15	31.09	44.1	1.28	399.8	40.7	75.178	0.23	362.2	28.14	37.028	1.01	396.4
kra30a	30	3.23	5.644	2.95	470.2	6.48	7.998	0.85	383	6.61	8.124	3.31	463.4
kra30b	30	3.3	6.502	3.25	469.2	4.38	8.39	1.00	375.4	5.86	6.614	2.96	457.6
lipa30a	30	1.75	1.912	2.47	430	2.09	2.162	0.77	372	1.87	1.926	2.41	448.6
lipa30b	30	16.13	16.594	2.73	449.2	16.22	16.906	0.80	378	16.91	17.216	2.65	433.4
esc32e	32	0	0	4.89	356.2	0	0	2.84	353.4	0	0	4.94	357.6
esc32g	32	0	0	4.41	358.8	0	0	1.63	357.6	0	0	4.64	362
esc32h	32	0.46	4.294	3.34	401.2	0.46	3.562	1.18	376.4	0	1.004	3.39	408.6
lipa40a	40	1.4	1.496	4.37	461.6	1.35	1.71	1.41	376.4	1.41	1.53	4.02	459.4
lipa40b	40	18.24	18.918	4.55	474.4	18.27	18.976	1.53	385.6	18.46	19.068	4.21	473.4

APPENDIX B: VARIABLE FIXING ALGORITHM

B.1. Selected QPLIB Instances Properties

We present a selection of problem instances from QPLIB library and outline their key properties, including the number of variables and optimal objective values. QPLIB is a comprehensive repository of quadratic programming instances, encompassing various problems from trivial to undecidable complexities. Each instance in QPLIB is characterized by several attributes, such as the total number of variables, the number of constraints, and the optimal objective value achieved. The selected instances are without constraints.

Table B.1. QPLIB Problem Instances.

Instance	n_var	OptObj
QPLIB_3642	1035	1034
QPLIB_3650	946	920
QPLIB_3693	1128	1148
QPLIB_3706	703	682
QPLIB_3822	861	850
QPLIB_3832	561	554
QPLIB_3838	780	746
QPLIB_3850	1225	1190
QPLIB_3877	630	602
QPLIB_5725	343	33611981
QPLIB_5875	200	43757
QPLIB_5881	120	13067
QPLIB_5882	150	25388
QPLIB_5909	250	35726
QPLIB_5922	500	128339

B.2. The Results of the scenarios

This section presents a comparative analysis of the performance metrics for the selected QPLIB instances. The comparative results across the four scenarios highlight the impact of both the active learning strategy and the size of the population used for variable fixing. The full-population settings (AllPop) generally outperform their half-population counterparts (HalfPop) in terms of both solution quality and variable fixing accuracy, as reflected in higher match ratios and lower deviation metrics. These findings suggest that leveraging the full diversity of the population alongside active learning improves the robustness and effectiveness of the algorithm. Furthermore, the WL (worst leaf) strategy shows slightly lower performance in some instances compared to AL (active learning), reinforcing the benefit of using variance-based criteria to guide solution generation and selection. Overall, the results demonstrate the value of combining learning-based decision rules with population-level insights to enhance performance in solving QPLIB instances.

Table B.2. QPLIB instances and comparisons for AL_AllPop_3Stop.

Instance	com_soln _size	com_soln _size_ratio	num_fixed _by_tree	tree_matching _count_opt	match_ratio _tree	perc _dev	gurobi _dev	compa- rison
QPLIB_3642	548.8	53.0%	94	51.2	54.5%	8.6%	1.9%	0
QPLIB_3650	466.6	49.3%	99.4	44.2	44.5%	6.0%	2.4%	0
QPLIB_3693	540	47.9%	98.6	41.4	42.0%	8.1%	0.9%	0
QPLIB_3706	406.8	57.9%	97.8	57.6	58.9%	2.3%	2.6%	1
QPLIB_3822	478.8	55.6%	94.2	53	56.3%	4.4%	1.2%	0
QPLIB_3832	239.4	42.7%	98.4	43.6	44.3%	1.0%	1.4%	1
QPLIB_3838	403.8	51.8%	94.4	51.8	54.9%	5.3%	3.0%	0
QPLIB_3850	566.2	46.2%	98	37	37.8%	8.8%	0.8%	0
QPLIB_3877	331.8	52.7%	97.8	46.2	47.2%	0.9%	1.7%	1
QPLIB_5725	164.6	48.0%	98.4	48.4	49.2%	1.0%	0.0%	0
QPLIB_5875	183.6	91.8%	100.6	98.2	97.6%	0.4%	0.0%	0
QPLIB_5881	114.6	95.5%	105	100.8	96.0%	0.2%	0.6%	1
QPLIB_5882	149.2	99.5%	103.6	103	99.4%	0.0%	0.0%	0
QPLIB_5909	195.2	78.1%	102.8	95.4	92.8%	14.4%	0.0%	0
QPLIB_5922	500	100.0%	99.4	99.4	100.0%	0.0%	0.0%	1
Mean		64.7%			65.0%	4.1%	1.1%	

Table B.3. QPLIB instances and comparisons for AL_HalfPop_3Stop.

Instance	com_soln _size	com_soln _size_ratio	num_fixed _by_tree	tree_matching _count_opt	match_ratio _tree	perc _dev	gurobi _dev	compa- rison
QPLIB_3642	548	52.9%	82	47.2	58%	8.7%	1.9%	0
QPLIB_3650	512	54.1%	80.2	39.8	50%	8.1%	2.4%	0
QPLIB_3693	618.4	54.8%	83	48.2	58%	10.1%	0.9%	0
QPLIB_3706	361.2	51.4%	82.8	44	53%	4.5%	2.6%	0
QPLIB_3822	470.6	54.7%	80.8	42.6	53%	6.2%	1.2%	0
QPLIB_3832	268.8	47.9%	80	38.8	49%	1.7%	1.4%	0
QPLIB_3838	348.2	44.6%	81	32.6	40%	5.9%	3.0%	0
QPLIB_3850	584.6	47.7%	83	38	46%	11.9%	0.8%	0
QPLIB_3877	343.4	54.5%	79.8	43	54%	0.7%	1.7%	1
QPLIB_5725	189.6	55.3%	85	47.6	56%	0.9%	2.2%	1
QPLIB_5875	196	98.0%	86.6	85.2	98%	0.1%	0.0%	0
QPLIB_5881	111.8	93.2%	90.2	86.2	96%	0.4%	0.6%	1
QPLIB_5882	150	100.0%	89.8	85.8	96%	0.0%	0.0%	1
QPLIB_5909	187.8	75.1%	88.8	75.4	85%	14.3%	0.0%	0
QPLIB_5922	484.4	96.9%	81	78.8	97%	0.5%	0.0%	0
Mean		65.4%			65.8%	4.9%	1.3%	

Table B.4. QPLIB instances and comparisons for WL_AllPop_3Stop.

Instance	com_soln _size	com_soln _size_ratio	num_fixed _by_tree	tree_matching _count_opt	match_ratio _tree	perc _dev	gurobi _dev	compa- rison
QPLIB_3642	566	54.7%	104.4	60.6	58.0%	7.27%	1.9%	0
QPLIB_3650	462.4	48.9%	107	48.6	45.4%	6.91%	2.4%	0
QPLIB_3693	559.6	49.6%	110.4	59.2	53.6%	8.08%	0.9%	0
QPLIB_3706	430.8	61.3%	110.8	74.6	67.3%	1.93%	2.6%	1
QPLIB_3822	549.8	63.9%	108.2	69.6	64.3%	4.19%	1.2%	0
QPLIB_3832	279	49.7%	110.4	51.8	46.9%	0.72%	1.4%	1
QPLIB_3838	398	51.0%	108	58.2	53.9%	5.47%	3.0%	0
QPLIB_3850	557.6	45.5%	105.6	40.4	38.3%	10.56%	0.8%	0
QPLIB_3877	325	51.6%	108.2	54.6	50.5%	1.00%	1.7%	1
QPLIB_5725	95	27.7%	121.6	33.2	27.3%	0.51%	0.0%	0
QPLIB_5875	189.8	94.9%	158.4	151.2	95.5%	0.24%	0.0%	0
QPLIB_5881	114.8	95.7%	120	113.8	94.8%	0.42%	0.4%	0
QPLIB_5882	140	93.3%	149.2	138	92.5%	0.14%	0.0%	0
QPLIB_5909	209	83.6%	154	141	91.6%	5.54%	0.0%	0
QPLIB_5922	497.8	99.6%	134.2	132.8	99.0%	0.00%	0.0%	0
Mean		64.7%			65.3%	3.53%	1.1%	

Table B.5. QPLIB instances and comparisons for WL_HalfPop_3Stop.

Instance	com_soln _size	com_soln _size_ratio	num.fixed _by_tree	tree_matching _count_opt	match_ratio _tree	perc _dev	gurobi _dev	compar- ison
QPLIB_3642	586.8	56.7%	93.8	46.4	49%	8.5%	1.9%	0
QPLIB_3650	483.2	51.1%	93	51	55%	6.7%	2.4%	0
QPLIB_3693	556.8	49.4%	94.6	45.6	48%	11.5%	0.9%	0
QPLIB_3706	403.6	57.4%	95.8	50.6	53%	3.1%	2.6%	0
QPLIB_3822	496.6	57.7%	93	55.2	59%	4.3%	1.2%	0
QPLIB_3832	243.8	43.5%	99	43.6	44%	0.9%	1.4%	1
QPLIB_3838	292.4	37.5%	94.4	40.4	43%	5.7%	3.0%	0
QPLIB_3850	559	45.6%	96.8	44.4	46%	9.8%	0.8%	0
QPLIB_3877	353.4	56.1%	96.2	58.2	60%	0.7%	1.7%	1
QPLIB_5725	218.8	63.8%	108.4	68.6	63%	0.7%	0.0%	0
QPLIB_5875	195.4	97.7%	130	125.4	96%	0.1%	0.0%	0
QPLIB_5881	108	90.0%	120	106.2	89%	1.1%	0.6%	0
QPLIB_5882	148.2	98.8%	138.8	130.8	94%	0.0%	0.0%	0
QPLIB_5909	203.6	81.4%	131.8	117.8	89%	6.5%	0.0%	0
QPLIB_5922	497.8	99.6%	122	119.8	98%	0.0%	0.0%	0
Mean		65.7%			65.9%	4.0%	1.1%	

APPENDIX C: GENERATING VARIABLE BOUNDS ALGORITHM

This appendix presents the results of the proposed Generating Variable Bounds Algorithm in continuous optimization problems (BBOB test functions). The outcomes are summarized in tables, illustrating the effectiveness of the method in refining the search space and improving solution quality.

C.1. Categorization of BBOB Test Functions by Specialties

This section categorizes the BBOB test functions selected for evaluating the proposed algorithm. These benchmark functions are grouped according to their structural characteristics, such as separability, conditioning, and modality. Table C.1 provides the function codes, names, and their corresponding groups used in the experiments.

Table C.1. Function Codes, Names, and Groups in the BBOB Test Suite.

func_code	name	group
f1	Sphere Function	Separable Functions
f2	Separable Ellipsoidal Function	Separable Functions
f3	Rastrigin Function	Separable Functions
f4	Büche-Rastrigin Function	Separable Functions
f5	Linear Slope	Separable Functions
f6	Attractive Sector Function	Functions with low or moderate conditioning
f7	Step Ellipsoidal Function	Functions with low or moderate conditioning
f8	Rosenbrock Function, original	Functions with low or moderate conditioning
f9	Rosenbrock Function, rotated	Functions with low or moderate conditioning
f10	Ellipsoidal Function	Functions with high conditioning and unimodal

Table C.1. Function Codes, Names, and Groups in the BBOB Test Suite (cont.).

func_code	name	group
f11	Discus Function	Functions with high conditioning and unimodal
f12	Bent Cigar Function	Functions with high conditioning and unimodal
f13	Sharp Ridge Function	Functions with high conditioning and unimodal
f14	Different Powers Function	Functions with high conditioning and unimodal
f15	Rastrigin Function	Multi-modal functions with adequate global structure
f16	Weierstrass Function	Multi-modal functions with adequate global structure
f17	Schaffer's F7 Function	Multi-modal functions with adequate global structure
f18	Schaffer's F7 Function, moderately ill-conditioned	Multi-modal functions with adequate global structure
f19	Composite Griewank-Rosenbrock Function F8F2	Multi-modal functions with adequate global structure
f20	Schwefel Function	Multi-modal functions with weak global structure
f21	Gallagher's Gaussian 101-me Peaks Function	Multi-modal functions with weak global structure
f22	Gallagher's Gaussian 21-hi Peaks Function	Multi-modal functions with weak global structure
f23	Katsuura Function	Multi-modal functions with weak global structure
f24	Lunacek bi-Rastrigin Function	Multi-modal functions with weak global structure

C.2. Results of the Variable Bounds Generation Algorithm Trials

The results of the trials on the BBOB test functions are provided in this section. The trials are conducted for `max_depth` 4 and 8.

C.2.1. The percentage shrinkage rates

This section reports the percentage shrinkage rates of the hyperrectangle defined by the solution space, comparing the revised bounds to the original bounds. The mean shrinkage values across all test functions are reported for both tree depths (4 and 8) under different bound-generation strategies.

The shrinkage rates provide insight into how effectively the regression tree narrows the search space by identifying and discarding regions associated with poor-quality solutions. A higher shrinkage percentage implies that the revised bounds significantly reduce the feasible region, focusing the search on more promising areas. This tighter bounding is particularly beneficial for solvers in high-dimensional or complex landscapes, as it can lead to faster convergence and improved solution quality. Moreover, comparing the shrinkage results across tree depths reveals how increased model complexity (i.e., deeper trees) impacts the precision and aggressiveness of bound generation. These findings support the use of learning-based approaches in pre-processing or guiding optimization algorithms by systematically refining the solution space.

Table C.2. The (%) shrinkage rate values, max_depth=4 (mean).

func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	68.4	68.8	68.8	68.6	68.8	69.4	69.6
f2	61.6	61.8	61.8	61.8	61.8	61.8	61.8
f3	52	52	52	52	52	52	52
f4	53.4	53.4	53.4	53.4	50.8	51.2	51.2
f5	94	96	96.2	95.6	96.6	95.6	95.6
f6	70	70.4	70.4	70.4	70	70	70
f7	65.4	67	65.4	65.4	67.6	66.8	65.2
f8	37	37.2	36.2	35.6	37.2	36	36.2
f9	37.8	39.6	36.2	37.6	38.2	38	38.8
f10	51.2	53.6	51	53	53.6	51	52.4
f11	62	63.8	63.2	63.8	65.4	64.4	63.4
f12	53.2	53.2	63.2	61.8	53.2	48.6	49.2
f13	70.8	72	71.8	72.4	72.4	72	71.4
f14	45	45	46	45.4	45.8	44.6	44.6
f15	57.2	57.8	57.8	56.2	57.4	56.2	58.2
f16	98	99.4	98.8	97.6	99	97.2	99.2
f17	55	57	57.4	51.4	58.2	56	55
f18	49.4	50.6	49.4	48	50	49.4	51.4
f19	39.8	40	39.2	39.2	39.8	39.2	39.2
f20	56.6	57.6	57.6	55.2	57	53.6	54
f21	91	96	94.4	93.2	94.8	92	87.8
f22	97.4	97.8	98	97.2	97.8	98.2	97.8
f23	97.4	97.8	99	98.2	98.6	98.4	98.2
f24	65.4	63	73.2	62	65.2	57.8	68.4
Mean	63.7	64.6	65.0	64.0	64.7	63.3	63.8

Table C.3. The (%) shrinkage rate values, max_depth=8 (mean).

func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	86	89.2	87.2	87.2	89.6	87.4	88.6
f2	83	86	84.4	84	85.8	85	83.6
f3	81.6	83.2	83	83.6	82.8	83.2	83.6
f4	82.8	83.2	82.2	83.8	83.4	83.8	82.6
f5	97.6	100	99.6	99.2	100	99.8	99.4
f6	84.8	83.8	83.8	84	83.6	84	83.8
f7	84.8	86	85.8	85.4	86.6	85.4	86
f8	89.2	90.4	71.8	74.8	78.8	73.8	78.4
f9	77.6	70.6	69.8	71.6	66	66.2	72.4
f10	78	82	81.6	81	83	81	81.8
f11	85.8	87.2	86.6	86.2	86.2	85.8	85.6
f12	80.6	80	77.4	75.6	77	79.8	74.4
f13	91.2	92	92.4	93	93.8	93.6	92.2
f14	74.6	78.2	77.2	77.4	77.8	76.8	77.8
f15	81	81.4	80.2	79.6	81.6	81.4	81.2
f16	98	99.6	99.6	98.4	99.8	99.4	98.6
f17	90.4	86.6	84.4	85.4	86.6	82	82.6
f18	96.6	85.6	93.6	93	86.2	91.6	97.6
f19	97.4	83.4	91.6	85.8	83	82.6	91.8
f20	78.2	79.8	80	80.6	80.4	79.2	78
f21	97.6	99.8	99	99	99	99.2	99
f22	97.2	100	99.4	99.2	100	99.6	99.2
f23	97.4	98.8	98.8	98.6	99	98.4	98.6
f24	97.4	97	98.8	93.8	94.2	97	98
Mean	87.9	87.7	87.0	86.7	86.8	86.5	87.3

C.2.2. The Number of Out-of-bound Variable

This section presents the average number of decision variables that fall outside the bounds generated by our algorithm. The number of out-of-bound variables is a useful metric to evaluate how well the bounds capture the region containing the optimal solution. A lower count suggests that the generated bounds effectively enclose

the optimal region. Results are provided separately for trees with maximum depths of 4 and 8.

Table C.4. Out-of-bound variables count, max_depth=4 (mean).

func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f4	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f7	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f8	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f9	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f10	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f11	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f12	0.00	0.00	0.40	0.40	0.00	0.00	0.00
f13	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f14	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f15	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f16	2.00	2.00	2.00	2.00	2.00	1.60	2.00
f17	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f18	0.20	0.20	0.20	0.00	0.00	0.20	0.20
f19	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f20	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f21	0.80	1.00	1.00	1.00	1.00	0.80	0.40
f22	0.80	0.80	1.00	1.00	1.00	1.00	0.80
f23	1.20	1.00	1.40	1.60	1.20	1.20	1.40
f24	0.60	0.80	1.00	0.60	0.80	0.60	0.60
Mean	0.23	0.24	0.29	0.28	0.25	0.23	0.23

Table C.5. Out-of-bound variables count, max_depth=8 (mean).

func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f4	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	1.00	1.00	1.00	1.00	1.00	1.00	1.00
f7	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f8	1.60	1.60	0.40	0.80	0.60	0.40	0.80
f9	0.80	0.20	0.20	0.40	0.00	0.00	0.40
f10	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f11	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f12	0.40	0.00	0.00	0.00	0.00	0.00	0.00
f13	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f14	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f15	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f16	1.80	1.60	2.00	1.60	1.80	2.00	1.60
f17	1.00	0.80	0.40	0.20	0.40	0.00	0.20
f18	1.00	0.60	1.20	1.00	1.00	1.40	1.80
f19	1.80	0.80	1.60	1.00	0.40	0.80	1.40
f20	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f21	0.60	0.80	0.20	0.20	0.20	0.00	0.60
f22	0.60	1.40	1.20	1.20	1.60	0.80	1.40
f23	1.00	1.60	1.20	1.40	1.40	0.80	1.40
f24	1.60	1.80	1.40	1.60	1.60	1.60	1.60
Mean	0.55	0.51	0.45	0.43	0.42	0.37	0.51

C.2.3. The Leaf Node Order of the Optimal Solution

The leaf nodes are sorted according to their average objective values in ascending order. The order of the leaf node where the optimal solution is located in the regression tree is given.

Table C.6. Optimal solution leaf node order, max_depth=4 (mean).

func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f3	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f9	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f10	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f11	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f12	1.0	1.0	1.1	1.1	1.0	1.0	1.0
f13	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f14	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f15	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f16	4.3	6.1	5.0	4.3	5.9	3.1	5.3
f17	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f18	1.1	1.1	1.1	1.0	1.0	1.1	1.1
f19	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f20	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f21	1.3	1.8	1.7	1.6	1.8	1.3	1.1
f22	2.9	2.3	3.2	3.2	4.0	2.8	2.4
f23	5.2	4.0	6.0	5.7	4.0	4.3	4.9
f24	1.6	2.0	2.5	1.6	2.0	1.6	1.6
Mean	1.4	1.5	1.6	1.5	1.5	1.3	1.4

Table C.7. Optimal solution leaf node order, max_depth=8 (mean).

func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f3	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f6	4.1	3.7	3.7	3.7	3.7	3.7	3.7
f7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f8	1.7	1.7	1.1	1.3	1.3	1.1	1.3
f9	1.3	1.1	1.1	1.1	1.0	1.0	1.1
f10	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f11	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f12	1.1	1.0	1.0	1.0	1.0	1.0	1.0
f13	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f14	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f15	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f16	3.9	2.7	7.1	8.7	7.7	4.0	5.8
f17	1.4	1.3	1.1	1.1	1.2	1.0	1.1
f18	1.7	1.5	2.4	1.8	1.5	1.9	2.5
f19	2.5	1.6	1.8	1.5	1.4	1.6	1.8
f20	1.0	1.0	1.0	1.0	1.0	1.0	1.0
f21	1.4	1.7	1.2	1.1	1.2	1.0	1.3
f22	2.2	8.7	5.5	6.0	8.7	3.0	6.3
f23	10.5	12.2	12.2	6.6	12.2	4.0	11.5
f24	2.9	3.2	2.5	2.5	1.8	2.0	3.4
Mean	1.9	2.2	2.2	2.0	2.3	1.6	2.2

C.2.4. The distance of the best solution to the optimal solution

This section presents the Euclidean distances between the best solutions found by each solver and the known optimal solutions. These distances provide a quantitative measure of how close the algorithms come to the optimum under different bound-generation strategies. A smaller distance indicates a better approximation of

the optimal solution by the solver. The results are reported as mean values across multiple benchmark functions for two tree depths: 4 and 8.

Table C.8. BOBYQA best-to-optimal distance, max_depth=4 (mean).

	BOBYQA						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.34	0.37	0.19	0.29	0.23	0.51	0.36
f3	0.92	0.98	0.31	0.87	0.80	0.43	0.72
f4	1.25	1.46	1.43	1.08	1.46	1.25	1.25
f5	0.01	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.07	0.01	0.03	0.02	0.07	0.04	0.05
f7	0.79	0.28	1.10	0.34	0.65	0.68	1.29
f8	0.08	0.19	0.16	0.19	0.04	0.53	0.23
f9	0.69	0.62	0.34	0.36	0.35	0.36	0.38
f10	1.21	1.49	0.83	0.92	1.34	1.39	2.28
f11	0.79	0.76	0.85	0.55	0.85	0.70	0.73
f12	0.34	0.30	2.10	1.78	0.54	0.68	0.55
f13	0.26	0.31	0.28	0.15	0.13	0.31	0.28
f14	0.18	0.17	0.15	0.18	0.29	0.17	0.16
f15	1.11	0.84	1.11	2.59	1.25	1.11	0.94
f16	6.17	6.47	6.73	6.98	6.42	5.43	6.50
f17	1.12	0.53	0.44	0.67	0.48	0.42	1.03
f18	2.74	3.22	3.14	3.08	2.42	2.54	3.20
f19	1.41	1.29	0.85	1.79	1.27	0.92	1.51
f20	2.89	2.54	3.88	2.47	3.33	2.38	2.38
f21	3.80	4.61	3.81	3.49	3.84	3.07	1.84
f22	1.96	1.91	1.90	2.35	1.91	2.35	2.25
f23	3.79	4.79	6.93	5.91	6.72	5.77	5.97
f24	3.13	4.64	3.95	4.16	4.27	4.10	3.99
Mean	1.46	1.57	1.69	1.68	1.61	1.46	1.58

Table C.9. BOBYQA best-to-optimal distance, max_depth=8 (mean).

	BOBYQA						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.03	0.14	0.08	0.14	0.06	0.03	0.06
f3	1.15	1.46	0.97	1.41	1.09	1.32	1.08
f4	2.21	2.27	2.07	2.34	1.66	1.93	2.12
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.44	0.54	0.62	0.65	0.53	0.63	0.51
f7	1.59	1.45	1.47	2.16	1.51	1.11	1.12
f8	4.48	4.71	0.99	2.25	2.44	0.15	2.28
f9	2.09	0.87	0.68	1.67	0.66	1.05	1.94
f10	0.78	1.16	0.88	1.43	1.84	1.20	1.15
f11	0.42	0.67	0.61	0.61	0.85	0.41	0.72
f12	1.57	0.67	1.06	1.04	0.80	0.54	0.89
f13	1.75	1.72	1.64	1.61	1.70	1.37	1.50
f14	0.12	0.19	0.15	0.13	0.16	0.10	0.12
f15	1.00	1.17	1.22	1.18	1.07	1.50	1.40
f16	5.47	5.46	5.67	3.30	4.77	5.75	4.71
f17	2.28	2.26	1.20	1.63	1.47	0.83	1.50
f18	3.88	3.18	3.18	3.08	4.29	4.40	5.95
f19	4.61	3.63	4.68	3.38	2.82	2.69	4.15
f20	1.16	1.92	1.68	1.88	1.96	2.38	1.28
f21	0.99	0.31	0.20	0.16	0.08	0.05	0.17
f22	1.50	2.81	2.81	2.81	2.32	1.41	2.36
f23	3.53	4.82	3.86	4.05	5.44	3.30	4.28
f24	3.94	4.52	3.41	4.61	4.70	5.11	3.91
Mean	1.87	1.91	1.63	1.73	1.76	1.55	1.80

Table C.10. COBYLA best-to-optimal distance, max_depth=4 (mean).

	COBYLA						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.01	0.00	0.01	0.00	0.00	0.00
f2	0.14	0.14	0.08	0.10	0.09	0.14	0.15
f3	1.20	1.03	0.42	0.71	1.20	0.76	0.78
f4	1.29	0.93	1.16	1.22	0.65	1.36	1.04
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.02	0.01	0.01	0.01	0.00	0.01	0.00
f7	1.81	2.19	2.73	1.99	2.12	1.85	2.54
f8	1.60	2.38	2.49	1.87	2.82	1.95	2.47
f9	0.74	0.48	0.46	0.58	0.54	0.53	0.71
f10	2.34	2.62	2.47	2.37	2.49	2.51	2.59
f11	0.59	0.88	1.11	0.96	0.89	0.78	0.94
f12	1.89	1.93	2.81	3.56	1.88	1.46	1.27
f13	0.26	0.18	0.27	0.41	0.28	0.27	0.32
f14	0.24	0.21	0.23	0.16	0.21	0.23	0.21
f15	1.06	1.07	0.94	1.81	1.43	1.35	1.14
f16	6.19	6.49	6.78	6.98	6.39	5.48	6.52
f17	1.32	0.76	0.41	1.53	0.60	0.97	1.10
f18	3.33	3.11	3.57	3.18	2.96	4.07	3.01
f19	1.40	1.35	1.58	1.73	1.75	0.81	1.48
f20	2.93	2.85	3.71	2.47	3.52	2.24	2.40
f21	3.35	4.58	3.65	4.02	3.63	3.64	2.36
f22	2.36	1.89	1.90	2.36	1.91	2.36	3.15
f23	3.71	5.40	7.04	4.78	6.66	4.85	5.94
f24	3.86	5.02	4.26	4.92	4.62	5.23	4.73
Mean	1.73	1.90	2.00	1.99	1.94	1.79	1.87

Table C.11. COBYLA best-to-optimal distance, max_depth=8 (mean).

	COBYLA						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.00	0.00	0.00	0.01	0.00	0.00
f2	0.22	0.11	0.21	0.17	0.16	0.10	0.09
f3	1.04	0.98	1.03	1.49	0.81	1.21	0.90
f4	1.94	2.10	1.70	1.44	1.75	1.93	1.95
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.64	0.54	0.59	0.63	0.50	0.63	0.59
f7	1.72	0.80	1.08	1.19	1.03	1.26	1.01
f8	4.84	4.74	2.04	3.08	2.42	2.16	2.86
f9	2.65	0.57	0.57	1.59	0.50	0.59	1.46
f10	2.86	2.64	2.68	2.84	2.22	2.88	2.76
f11	1.08	1.26	0.88	0.74	0.62	1.42	1.01
f12	2.14	2.09	2.52	2.18	1.42	1.50	1.64
f13	1.91	2.08	1.40	1.63	1.90	1.73	1.98
f14	0.12	0.14	0.10	0.05	0.12	0.05	0.07
f15	1.20	0.96	1.37	1.91	0.81	1.27	1.13
f16	4.56	5.50	5.71	3.58	4.79	5.40	4.76
f17	1.95	2.24	1.32	1.85	1.54	0.94	1.22
f18	3.63	2.72	3.03	3.30	4.72	4.64	5.81
f19	4.48	3.40	4.66	3.40	2.95	3.21	3.74
f20	2.71	2.18	3.49	2.84	2.74	3.32	2.98
f21	0.93	0.23	0.08	0.07	0.08	0.07	0.16
f22	1.94	2.81	2.81	2.81	2.32	1.46	2.36
f23	3.06	4.89	3.81	4.12	4.18	3.53	3.54
f24	4.69	5.11	3.45	4.82	5.12	5.25	3.73
Mean	2.10	2.00	1.86	1.91	1.78	1.86	1.91

Table C.12. PRAXIS best-to-optimal distance, max_depth=4 (mean).

	PRAXIS						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.00	0.01	0.01	0.00	0.00	0.00
f2	0.01	0.04	0.45	0.02	0.00	0.03	0.02
f3	3.10	3.03	2.79	3.21	2.19	2.86	2.97
f4	2.17	2.85	2.68	2.69	2.27	3.24	2.83
f5	1.80	1.36	2.23	3.19	1.10	1.15	2.81
f6	0.03	0.03	0.01	0.02	0.11	0.05	0.06
f7	0.14	0.72	0.38	0.74	0.65	0.78	0.82
f8	0.03	0.07	0.02	0.04	0.08	0.05	0.02
f9	0.30	0.62	0.73	0.49	0.51	0.72	0.81
f10	0.62	0.42	0.97	0.99	0.28	1.40	0.00
f11	0.31	0.56	0.44	0.35	0.23	0.34	0.34
f12	0.04	0.18	1.20	1.22	0.12	0.02	0.03
f13	1.36	1.55	1.86	1.68	1.34	1.79	1.72
f14	0.10	0.13	0.14	0.09	0.06	0.09	0.15
f15	1.36	1.48	1.33	1.58	1.58	1.57	1.36
f16	6.15	6.64	6.81	7.17	6.53	5.46	6.61
f17	0.94	0.60	0.51	1.01	0.50	0.55	0.67
f18	2.96	2.92	2.69	2.85	2.69	2.86	3.09
f19	0.78	0.72	0.95	0.46	0.94	0.60	1.12
f20	1.75	2.27	1.45	2.01	1.64	1.13	1.55
f21	3.28	4.17	3.43	3.42	3.43	2.99	2.15
f22	1.58	1.44	1.79	1.87	1.43	1.99	2.15
f23	3.47	5.11	6.67	5.59	6.75	4.96	5.99
f24	3.71	4.13	3.89	4.40	4.51	4.48	3.81
Mean	1.50	1.71	1.81	1.88	1.62	1.63	1.71

Table C.13. PRAXIS best-to-optimal distance, max_depth=8 (mean).

	PRAXIS						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.01	0.00	0.00	0.01	0.00	0.00	0.01
f2	0.13	0.02	0.15	0.04	0.00	0.00	0.32
f3	0.94	0.92	0.75	0.80	0.86	0.77	0.69
f4	2.09	2.10	2.10	2.13	2.10	2.18	2.18
f5	1.57	0.60	0.85	0.66	0.50	0.59	0.48
f6	0.53	0.59	0.97	0.84	0.82	0.63	0.54
f7	0.93	1.65	1.41	1.76	1.37	1.04	1.56
f8	4.17	4.43	0.81	1.99	1.48	1.30	2.19
f9	2.00	0.42	0.63	1.27	0.64	0.65	1.60
f10	0.61	0.47	1.72	1.03	0.89	0.68	0.45
f11	0.36	0.15	0.45	0.18	0.72	0.16	0.22
f12	0.72	0.05	0.09	0.16	0.04	0.03	0.04
f13	2.06	1.41	1.98	1.82	1.55	1.68	1.86
f14	0.06	0.08	0.10	0.14	0.13	0.13	0.12
f15	0.48	0.37	0.85	0.93	0.72	0.96	0.91
f16	5.24	5.46	5.69	3.25	4.81	5.47	4.89
f17	2.26	1.74	1.49	1.59	1.24	1.46	1.31
f18	3.67	3.14	3.00	3.08	4.27	4.29	6.04
f19	4.53	3.04	4.30	3.84	3.29	3.52	4.09
f20	1.99	2.43	2.47	1.36	2.13	1.65	2.34
f21	0.52	0.26	0.08	0.07	0.09	0.06	0.19
f22	0.98	2.81	2.81	2.81	2.34	1.33	2.33
f23	3.76	4.85	4.09	3.69	4.86	3.17	4.31
f24	4.06	4.52	3.46	4.75	4.95	5.10	3.60
Mean	1.82	1.73	1.68	1.59	1.66	1.54	1.76

Table C.14. NELDERMEAD best-to-optimal distance, max_depth=4 (mean).

	NELDERMEAD						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.01	0.01	0.01	0.01	0.01	0.01	0.01
f2	0.01	0.01	0.00	0.01	0.00	0.00	0.00
f3	1.24	0.60	1.24	1.08	0.27	0.78	1.09
f4	1.70	1.30	1.89	0.83	0.92	0.83	1.13
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f7	1.78	1.75	2.85	1.52	1.78	1.53	2.41
f8	0.00	0.01	0.01	0.01	0.01	0.01	0.01
f9	0.02	0.10	0.01	0.01	0.02	0.03	0.02
f10	0.01	0.01	0.01	0.01	0.01	0.01	0.01
f11	0.01	0.01	0.01	0.01	0.01	0.01	0.01
f12	0.04	0.02	1.21	1.23	0.03	0.03	0.02
f13	0.01	0.01	0.01	0.01	0.01	0.01	0.01
f14	0.02	0.03	0.05	0.04	0.03	0.04	0.05
f15	0.92	0.98	0.83	0.90	0.44	0.93	1.13
f16	6.16	6.51	6.75	6.95	6.29	5.64	6.53
f17	1.12	0.69	0.63	0.88	0.65	0.42	0.99
f18	2.40	2.30	2.56	2.54	1.66	1.66	2.38
f19	1.49	1.38	0.87	1.43	1.47	0.92	1.39
f20	3.70	3.50	3.74	3.19	3.91	3.20	3.03
f21	3.32	4.58	3.62	4.05	3.63	3.37	2.15
f22	2.30	1.77	1.76	2.32	1.86	2.39	3.15
f23	4.53	5.50	7.28	4.91	6.78	4.98	6.23
f24	4.20	5.48	4.42	4.91	5.10	4.71	4.13
Mean	1.46	1.52	1.66	1.54	1.45	1.31	1.49

Table C.15. NELDERMEAD best-to-optimal distance, max_depth=8 (mean).

	NELDERMEAD						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.01	0.00	0.01	0.01	0.01	0.01	0.01
f2	0.01	0.00	0.00	0.00	0.00	0.00	0.01
f3	0.69	0.75	0.52	0.62	0.63	0.56	0.32
f4	1.93	2.09	1.67	1.52	1.62	1.66	1.89
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.64	0.54	0.59	0.62	0.50	0.63	0.60
f7	2.27	1.72	2.49	2.16	2.63	2.07	2.26
f8	4.17	4.61	0.81	1.97	1.47	1.28	2.39
f9	2.02	0.02	0.03	1.03	0.01	0.01	1.08
f10	0.01	0.01	0.02	0.01	0.54	0.01	1.49
f11	0.01	0.01	0.01	0.01	0.00	0.01	0.01
f12	0.75	0.04	0.05	0.08	0.04	0.03	0.06
f13	0.01	0.01	0.00	0.01	0.01	0.00	0.01
f14	0.06	0.04	0.05	0.06	0.06	0.06	0.03
f15	0.54	0.79	0.76	0.48	0.54	0.67	1.01
f16	4.40	5.40	5.73	3.49	4.69	5.32	4.52
f17	1.92	2.30	1.01	1.70	1.26	0.46	0.62
f18	3.75	3.23	2.82	3.32	4.17	4.21	5.71
f19	4.57	3.53	4.66	2.69	2.70	2.67	3.71
f20	2.56	2.46	3.36	3.27	3.30	3.63	3.12
f21	0.88	0.21	0.08	0.06	0.02	0.03	0.15
f22	1.85	2.81	2.82	2.81	2.32	1.30	2.36
f23	3.40	4.89	3.76	4.20	4.76	3.64	3.70
f24	3.94	4.97	3.39	4.91	4.66	5.30	3.93
Mean	1.68	1.69	1.44	1.46	1.50	1.40	1.62

Table C.16. SBPLX best-to-optimal distance, max_depth=4 (mean).

	SBPLX						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	1.10	0.60	1.29	1.08	0.41	0.78	1.09
f4	1.70	1.32	1.89	0.95	0.99	0.78	1.13
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.00	0.00	0.00	0.00	0.01	0.00	0.00
f7	1.70	1.62	2.79	1.39	1.61	1.40	2.30
f8	0.03	0.01	0.04	0.02	0.01	0.01	0.01
f9	0.01	0.01	0.01	0.01	0.01	0.02	0.01
f10	0.13	0.02	0.00	0.00	0.03	0.00	0.01
f11	0.00	0.00	0.00	0.01	0.25	0.00	0.25
f12	0.06	0.08	1.25	1.25	0.02	0.01	0.07
f13	0.06	0.06	0.17	0.05	0.07	0.01	0.11
f14	0.05	0.05	0.07	0.06	0.07	0.04	0.05
f15	0.92	0.91	0.87	0.85	0.20	0.98	1.12
f16	6.19	6.53	6.75	6.96	6.29	5.64	6.54
f17	1.08	0.98	0.63	1.14	0.67	0.49	0.65
f18	2.41	2.24	2.37	2.48	1.73	1.79	2.37
f19	1.27	1.35	0.92	1.42	1.30	0.99	1.30
f20	3.71	3.51	3.75	3.19	3.92	3.11	2.92
f21	3.31	4.58	3.62	4.03	3.61	3.37	2.13
f22	2.27	1.73	1.76	2.29	1.85	2.34	3.10
f23	4.13	5.29	7.26	4.90	6.77	4.97	6.23
f24	4.29	5.46	4.38	4.88	5.21	5.05	4.27
Mean	1.43	1.51	1.66	1.54	1.46	1.32	1.49

Table C.17. SBPLX best-to-optimal distance, max_depth=8 (mean).

	SBPLX						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.01	0.00	0.00	0.01	0.00	0.01	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.70	0.75	0.52	0.62	0.63	0.56	0.32
f4	1.93	2.09	1.67	1.52	1.62	1.68	1.89
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.64	0.54	0.59	0.63	0.50	0.63	0.59
f7	2.22	1.43	2.25	2.20	2.63	1.91	1.94
f8	4.17	4.61	0.81	1.97	1.47	1.28	2.39
f9	1.75	0.01	0.03	1.04	0.01	0.02	1.09
f10	0.04	0.00	0.04	0.00	0.46	0.01	0.00
f11	0.17	0.00	0.03	0.00	0.19	0.00	0.01
f12	0.73	0.09	0.05	0.02	0.02	0.02	0.05
f13	0.03	0.04	0.03	0.03	0.00	0.42	0.02
f14	0.07	0.07	0.06	0.08	0.08	0.05	0.08
f15	0.54	0.79	0.75	0.55	0.47	0.68	1.05
f16	4.38	5.40	5.71	3.47	4.70	5.34	4.52
f17	1.83	2.38	0.95	1.66	1.28	0.79	0.63
f18	3.77	3.04	2.83	3.28	3.93	4.23	5.73
f19	4.57	3.46	4.65	2.66	2.47	2.71	3.70
f20	2.54	2.13	3.36	3.28	3.22	3.64	3.13
f21	0.87	0.21	0.08	0.05	0.02	0.03	0.15
f22	1.72	2.81	2.81	2.81	2.32	1.26	2.31
f23	3.22	4.90	3.72	4.16	4.40	3.64	3.72
f24	4.04	5.04	3.39	4.89	4.79	5.36	3.93
Mean	1.66	1.66	1.43	1.46	1.47	1.43	1.55

Table C.18. DIRECT best-to-optimal distance, max_depth=4 (mean).

	DIRECT						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.36	0.36	0.36	0.36	0.36	0.36	0.36
f4	0.58	0.58	0.58	0.58	0.58	0.72	0.66
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.09	0.10	0.06	0.10	0.09	0.09	0.09
f7	0.18	0.02	0.34	0.18	0.04	0.16	0.34
f8	0.04	0.03	0.03	0.12	0.03	0.04	0.03
f9	0.19	0.36	0.31	0.38	0.27	0.27	0.45
f10	0.27	0.46	0.25	0.24	0.46	0.25	0.40
f11	0.49	0.70	0.43	0.52	0.32	0.79	0.80
f12	0.48	0.75	1.99	1.74	0.75	0.80	0.58
f13	0.32	0.24	0.24	0.23	0.23	0.24	0.21
f14	0.09	0.09	0.01	0.09	0.01	0.09	0.09
f15	0.40	0.40	0.40	0.30	0.67	0.40	0.20
f16	6.29	6.50	6.73	6.85	6.33	5.61	6.46
f17	0.09	0.06	0.04	0.13	0.08	0.07	0.03
f18	0.39	0.39	0.39	0.41	0.33	0.39	0.40
f19	2.49	2.19	1.96	1.96	2.49	1.96	1.96
f20	0.24	0.00	0.00	0.00	0.00	0.47	0.47
f21	3.34	4.32	3.37	3.41	3.40	2.98	1.68
f22	2.28	1.84	1.86	2.29	1.81	2.35	2.61
f23	4.07	5.71	7.65	4.87	7.60	6.25	6.91
f24	2.62	5.89	5.38	5.34	5.70	3.49	4.40
Mean	1.05	1.29	1.35	1.25	1.31	1.16	1.21

Table C.19. DIRECT best-to-optimal distance, max_depth=8 (mean).

	DIRECT						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.01	0.00	0.00	0.00	0.01	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.35	0.76	0.48	0.89	0.70	0.76	0.76
f4	0.65	0.62	0.66	1.05	0.42	0.64	0.61
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.63	0.53	0.60	0.63	0.50	0.63	0.60
f7	0.02	0.02	0.02	0.02	0.02	0.02	0.02
f8	4.18	4.67	0.93	2.02	1.55	1.35	2.25
f9	1.86	0.11	0.09	1.06	0.13	0.19	1.14
f10	0.67	0.30	0.55	0.36	0.30	0.49	0.51
f11	0.09	0.52	0.22	0.47	0.19	0.07	0.42
f12	1.32	0.30	0.74	0.73	0.64	0.46	0.60
f13	0.08	0.06	0.10	0.06	0.10	0.08	0.08
f14	0.05	0.18	0.11	0.12	0.17	0.12	0.12
f15	0.64	0.47	0.66	0.60	0.88	0.07	0.40
f16	5.38	5.31	5.70	3.18	4.61	5.76	4.52
f17	1.98	1.58	0.44	0.90	0.74	0.14	0.49
f18	2.58	1.70	1.93	2.56	3.17	3.73	5.85
f19	4.98	2.79	5.27	3.72	2.74	3.68	4.24
f20	1.72	0.98	1.72	1.20	0.98	1.95	0.46
f21	0.89	0.24	0.07	0.06	0.04	0.03	0.16
f22	1.72	2.81	2.81	2.81	2.32	1.26	2.34
f23	3.36	6.26	5.04	4.76	5.76	3.45	5.22
f24	4.07	4.75	3.17	5.05	4.64	5.10	3.52
Mean	1.55	1.49	1.30	1.34	1.27	1.25	1.43

Table C.20. CRS2_LM best-to-optimal distance, max_depth=4 (mean).

	CRS2_LM						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.02	0.01	0.01	0.02	0.01	0.01	0.01
f2	0.01	0.03	1.52	0.01	0.98	0.01	0.01
f3	0.00	0.40	0.00	0.00	0.00	0.00	0.00
f4	0.00	0.40	0.40	0.20	0.20	0.20	0.00
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.01	0.01	0.01	0.01	0.01	0.01	0.01
f7	0.02	0.02	0.03	0.02	0.03	0.03	0.03
f8	0.00	0.02	0.03	0.02	0.01	0.02	0.01
f9	0.11	0.03	0.03	0.02	0.03	0.05	0.06
f10	0.51	0.02	0.03	0.54	0.01	1.49	0.03
f11	0.01	0.01	0.17	0.01	0.03	0.04	0.01
f12	0.53	0.11	1.20	1.41	0.16	0.02	0.06
f13	0.01	0.02	0.02	0.08	0.03	0.01	0.01
f14	0.08	0.08	0.02	0.07	0.07	0.06	0.09
f15	0.00	0.60	0.00	0.22	0.20	0.00	0.80
f16	6.33	6.42	6.72	6.85	6.33	5.61	6.47
f17	0.02	0.09	0.02	0.03	0.06	0.53	0.06
f18	0.06	0.62	0.62	0.20	0.00	0.06	0.74
f19	0.13	0.72	0.54	1.32	2.11	0.54	1.73
f20	0.98	1.20	1.18	0.73	0.98	1.91	0.46
f21	3.35	4.31	3.39	3.62	3.39	2.96	1.69
f22	2.28	1.77	1.75	2.29	2.26	2.30	2.62
f23	3.44	5.20	6.51	5.64	6.71	4.71	5.36
f24	3.63	5.64	4.65	5.58	5.57	4.48	4.53
Mean	0.90	1.16	1.20	1.20	1.21	1.04	1.03

Table C.21. CRS2_LM best-to-optimal distance, max_depth=8 (mean).

	CRS2_LM						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.02	0.02	0.01	0.02	0.01	0.01	0.01
f2	0.02	0.49	0.49	0.49	0.00	0.49	0.01
f3	0.00	0.00	0.00	0.20	0.00	0.00	0.06
f4	0.00	0.20	0.06	0.00	0.40	0.00	0.00
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.64	0.53	0.60	0.63	0.50	0.63	0.56
f7	0.02	0.02	0.02	0.02	0.02	0.02	0.03
f8	4.17	4.60	0.84	1.97	1.47	1.29	2.18
f9	1.77	0.06	0.03	1.09	0.03	0.03	1.22
f10	0.04	0.55	0.03	0.93	0.13	1.43	0.59
f11	1.21	1.22	1.22	1.34	2.43	0.01	1.29
f12	1.07	0.06	0.19	0.01	0.03	0.13	1.23
f13	0.01	0.02	0.01	0.02	0.01	0.01	0.02
f14	0.05	0.01	0.04	0.05	0.05	0.03	0.11
f15	0.20	0.48	0.07	0.07	0.00	0.00	0.43
f16	5.28	5.32	5.43	3.14	4.64	5.75	4.90
f17	1.96	1.33	0.40	0.61	0.91	0.01	0.34
f18	2.72	1.72	1.69	2.18	3.19	3.84	5.63
f19	4.85	3.24	5.04	3.05	2.29	2.64	4.06
f20	0.75	1.73	0.46	2.98	0.46	0.46	0.96
f21	0.87	0.23	0.09	0.08	0.02	0.04	0.17
f22	1.69	2.80	2.80	2.81	2.33	1.31	2.34
f23	4.24	4.86	4.62	4.21	5.60	4.28	5.48
f24	3.87	4.71	3.04	4.64	4.68	4.76	3.53
Mean	1.48	1.42	1.13	1.27	1.22	1.13	1.47

Table C.22. ISRES best-to-optimal distance, max_depth=4 (mean).

	ISRES						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.07	0.06	0.07	0.05	0.08	0.05	0.05
f2	1.24	1.15	1.41	1.76	1.43	1.56	2.49
f3	0.65	0.45	0.26	0.26	0.29	0.30	0.30
f4	0.61	0.44	0.52	0.61	0.50	0.39	0.48
f5	0.10	0.10	0.06	0.08	0.01	0.09	0.07
f6	0.11	0.13	0.14	0.13	0.11	0.10	0.14
f7	0.10	0.09	0.15	0.11	0.19	0.13	0.16
f8	0.68	0.30	0.57	0.53	0.60	0.67	0.37
f9	0.57	0.32	0.24	0.58	0.45	0.42	0.38
f10	1.06	0.85	2.39	2.76	1.89	1.89	1.59
f11	2.16	1.23	1.08	2.01	0.38	1.82	2.12
f12	1.36	1.51	2.74	3.01	1.96	1.44	1.54
f13	0.52	0.22	0.32	0.44	0.21	0.31	0.55
f14	0.09	0.16	0.16	0.11	0.20	0.21	0.07
f15	0.60	0.04	0.43	0.63	0.36	0.62	0.28
f16	6.33	6.51	6.73	6.86	6.33	5.48	6.42
f17	0.29	0.22	0.15	0.38	0.11	0.26	0.20
f18	0.73	0.46	0.51	0.42	0.46	0.64	0.74
f19	0.65	0.42	1.44	0.70	1.31	1.43	1.23
f20	0.31	0.14	0.53	0.11	1.79	0.11	1.05
f21	3.38	4.32	3.40	3.38	3.39	2.96	1.70
f22	2.29	1.81	1.79	2.30	1.77	2.32	2.08
f23	2.99	5.85	7.04	5.93	6.77	5.20	5.77
f24	3.61	5.90	4.88	3.62	5.69	5.64	4.49
Mean	1.27	1.36	1.54	1.53	1.51	1.42	1.43

Table C.23. ISRES best-to-optimal distance, max_depth=8 (mean).

	ISRES						
func_code	Base	medoid_0.0	medoid_0.3	medoid_0.5	pair_wise_0.0	pair_wise_0.3	pair_wise_0.5
f1	0.03	0.05	0.03	0.05	0.04	0.03	0.03
f2	1.42	1.16	1.01	1.58	1.12	1.07	2.04
f3	0.43	0.22	0.50	0.39	0.22	0.22	0.21
f4	0.25	0.09	0.21	0.21	0.21	0.21	0.21
f5	0.06	0.01	0.01	0.03	0.01	0.01	0.02
f6	0.62	0.58	0.58	0.63	0.50	0.63	0.56
f7	0.17	0.13	0.17	0.05	0.11	0.11	0.16
f8	4.31	4.74	1.06	2.53	1.46	1.46	2.26
f9	1.94	0.45	0.09	1.27	0.33	0.33	1.32
f10	1.78	1.11	1.97	1.66	0.63	1.66	0.91
f11	1.64	0.12	0.54	0.83	0.83	1.57	0.92
f12	2.01	0.40	0.26	0.37	0.22	0.37	0.92
f13	0.18	0.14	0.12	0.12	0.22	0.22	0.01
f14	0.12	0.04	0.03	0.04	0.04	0.03	0.06
f15	0.41	0.12	0.05	0.05	0.22	0.10	0.06
f16	5.41	5.31	5.44	2.78	4.64	5.25	4.52
f17	0.64	0.53	0.40	0.61	0.55	0.27	0.34
f18	2.63	1.92	1.81	2.45	3.37	3.37	5.47
f19	4.46	3.04	4.50	3.77	3.12	2.89	3.71
f20	0.77	0.78	0.07	0.10	0.21	0.78	0.21
f21	0.88	0.26	0.09	0.08	0.04	0.03	0.15
f22	1.72	2.81	2.81	2.81	2.32	1.26	2.34
f23	4.18	4.52	3.44	4.51	5.36	3.70	3.70
f24	3.93	4.34	3.30	4.69	5.00	5.05	3.59
Mean	1.68	1.49	1.24	1.39	1.32	1.37	1.50

C.2.5. Minimum Distance to Optimal Solution Across Five Replication Results: All Scenarios

This subsection presents the minimum Euclidean distance to the optimal solution obtained across five independent replications for each scenario. Results are reported for all 24 BBOB functions using eight different solvers under various parameter settings, including tree depth (max_depth), peak-valley (PV_method), and exploration probability (p_e). The values shown in Tables C.24–C.31 help evaluate the effectiveness of each configuration in guiding the search toward high-quality solutions.

Table C.24. BOBYQA best-to-optimal distance across scenarios (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.02	0.16	0.00	0.00	0.01	0.09	0.10	0.00	0.00	0.02	0.08	0.03	0.01	0.02
f3	0.31	0.31	0.31	0.31	0.00	0.31	0.31	1.00	1.00	0.61	0.99	1.00	0.34	0.00
f4	0.00	0.99	0.99	0.31	0.99	0.10	0.99	0.61	0.99	0.99	1.04	0.99	0.99	1.89
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.02	0.00	0.01	0.00	0.01	0.00	0.01	0.02	0.30	0.51	0.58	0.30	0.58	0.41
f7	0.23	0.01	0.13	0.02	0.15	0.11	0.10	0.04	0.09	0.02	0.85	0.93	0.13	0.01
f8	0.00	0.04	0.02	0.01	0.01	0.00	0.10	0.19	0.03	0.02	0.05	0.17	0.04	0.12
f9	0.28	0.13	0.10	0.19	0.16	0.03	0.16	0.32	0.08	0.18	0.01	0.10	0.08	0.50
f10	0.02	0.04	0.01	0.18	0.03	0.01	0.23	0.16	0.29	0.02	0.35	0.27	0.00	0.10
f11	0.14	0.39	0.12	0.13	0.74	0.34	0.15	0.13	0.22	0.10	0.22	0.22	0.09	0.13
f12	0.01	0.00	0.02	0.08	0.00	0.00	0.03	0.04	0.00	0.26	0.20	0.01	0.00	0.26
f13	0.01	0.05	0.07	0.06	0.07	0.06	0.07	0.13	1.15	0.84	0.30	0.72	0.35	0.17
f14	0.04	0.05	0.04	0.04	0.12	0.05	0.05	0.00	0.06	0.04	0.03	0.03	0.01	0.00
f15	0.99	0.00	0.99	0.96	0.00	0.99	0.00	0.36	0.00	0.00	0.00	0.33	0.69	0.69
f16	5.35	5.74	5.95	6.80	5.40	0.31	5.46	0.81	0.28	1.37	1.31	0.72	1.46	1.26
f17	0.08	0.00	0.06	0.26	0.21	0.19	0.07	0.42	0.20	0.13	0.18	0.38	0.27	0.90
f18	1.81	2.41	2.69	2.48	1.37	1.73	2.70	1.47	0.17	1.08	1.43	1.34	0.08	2.50
f19	0.01	0.59	0.30	0.38	0.42	0.59	0.48	3.54	2.00	2.93	0.86	0.93	0.25	0.76
f20	1.17	1.24	2.77	1.12	1.91	1.12	1.12	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f21	0.08	0.22	0.13	0.13	0.30	0.19	0.09	0.13	0.10	0.04	0.02	0.03	0.03	0.02
f22	0.48	0.55	0.51	0.50	0.53	0.56	0.51	0.48	2.81	2.81	2.80	0.36	0.41	0.55
f23	1.93	3.04	1.59	2.77	2.74	3.09	2.49	1.93	1.98	2.37	2.86	1.93	0.88	1.15
f24	1.59	3.35	2.02	3.16	3.12	2.38	1.30	1.32	0.85	0.79	0.79	0.34	0.79	0.95
Sum	14.58	19.30	18.81	19.87	18.30	12.24	16.51	13.11	12.61	15.15	14.96	11.11	7.49	12.37

Table C.25. COBYLA best-to-optimal distance across scenarios (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.01	0.01	0.01	0.03	0.00	0.07	0.01	0.02	0.01	0.04	0.04	0.01	0.01	0.05
f3	0.31	0.31	0.00	0.00	0.31	0.00	0.00	0.99	0.34	0.99	0.99	0.00	0.64	0.34
f4	0.99	0.31	0.61	0.96	0.00	1.17	0.96	0.99	1.00	0.99	0.61	0.99	0.99	1.16
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.41	0.30	0.41	0.58	0.30	0.58	0.41
f7	1.28	2.03	1.23	1.28	1.99	1.08	1.30	0.21	0.25	0.30	0.33	0.28	0.33	0.05
f8	0.15	0.49	1.24	0.92	0.50	0.38	0.98	0.64	0.03	0.29	0.30	0.01	0.00	0.07
f9	0.40	0.40	0.40	0.41	0.49	0.45	0.40	0.74	0.47	0.40	0.45	0.44	0.47	0.46
f10	2.18	2.40	2.25	2.06	2.25	2.24	2.21	2.39	1.76	2.53	2.45	0.43	2.46	2.41
f11	0.06	0.59	0.66	0.79	0.76	0.34	0.03	0.32	0.42	0.27	0.15	0.08	0.65	0.07
f12	0.01	0.00	0.09	1.43	0.04	0.01	0.02	0.05	0.04	1.61	0.20	0.13	0.40	0.19
f13	0.00	0.01	0.00	0.03	0.08	0.08	0.10	0.64	1.56	0.32	0.15	0.91	0.40	0.65
f14	0.22	0.13	0.22	0.03	0.07	0.20	0.11	0.03	0.02	0.04	0.02	0.01	0.01	0.01
f15	0.95	0.99	0.36	0.97	1.00	1.01	0.99	0.36	0.00	0.00	0.00	0.00	0.69	0.36
f16	5.36	5.78	5.97	6.79	5.42	0.48	5.54	0.70	0.38	1.37	1.74	0.73	1.43	1.36
f17	0.03	0.04	0.01	0.00	0.01	0.01	0.13	0.39	0.06	0.48	0.89	0.09	0.03	0.64
f18	2.61	2.08	2.19	2.20	2.11	2.66	2.03	1.37	0.16	1.09	1.67	1.02	0.01	2.40
f19	0.52	0.34	0.35	0.31	0.75	0.40	0.49	3.19	2.05	2.58	1.42	1.28	2.08	0.81
f20	1.09	1.14	2.17	1.12	1.61	1.12	1.39	0.34	1.24	2.84	1.39	0.00	2.81	1.91
f21	0.04	0.08	0.07	0.06	0.07	0.06	0.06	0.10	0.01	0.03	0.00	0.06	0.02	0.01
f22	0.56	0.47	0.53	0.55	0.55	0.58	0.64	0.56	2.81	2.81	2.81	0.39	0.56	0.56
f23	0.80	3.11	2.35	1.19	2.74	2.52	2.68	0.85	2.16	1.89	2.93	0.89	0.88	1.33
f24	1.59	2.77	1.31	4.17	3.49	4.17	0.47	1.32	0.85	0.80	0.79	0.34	0.62	0.70
Sum	19.16	23.47	22.00	25.30	24.25	19.03	20.54	16.62	15.92	22.07	19.90	8.38	16.08	15.94

Table C.26. PRAXIS best-to-optimal distance across scenarios (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.64	0.31	0.00	1.20	0.00	0.31	0.00	0.31	0.61	0.00	0.00	0.31	0.00	0.00
f4	1.17	1.17	2.11	1.17	1.17	2.30	2.08	2.08	2.08	2.08	2.08	2.08	2.08	2.08
f5	0.01	0.15	1.14	0.00	0.01	0.00	0.90	0.00	0.51	0.08	0.11	0.00	0.05	0.20
f6	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.29	0.26	0.37	0.42	0.44	0.52	0.38
f7	0.01	0.04	0.03	0.24	0.01	0.08	0.04	0.27	1.16	1.34	1.26	0.36	0.04	1.20
f8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
f9	0.11	0.40	0.42	0.23	0.05	0.36	0.45	0.07	0.24	0.24	0.04	0.11	0.22	0.18
f10	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f11	0.00	0.15	0.00	0.00	0.01	0.03	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00
f12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f13	0.00	0.05	0.18	0.00	0.16	0.67	0.02	0.67	0.25	1.54	1.18	0.37	0.01	0.35
f14	0.01	0.02	0.04	0.01	0.02	0.00	0.11	0.01	0.04	0.06	0.08	0.05	0.01	0.02
f15	1.00	0.99	1.00	1.00	0.95	1.00	1.00	0.00	0.00	0.33	0.33	0.36	0.68	0.00
f16	5.36	5.74	5.93	6.67	5.49	0.39	5.51	0.57	0.33	1.41	1.28	0.69	1.35	1.11
f17	0.48	0.04	0.31	0.05	0.22	0.08	0.28	0.58	0.14	0.48	0.73	0.22	1.13	0.57
f18	2.39	2.63	1.62	2.68	1.82	2.41	2.69	1.47	0.41	0.99	1.31	1.33	0.09	2.36
f19	0.17	0.13	0.26	0.12	0.19	0.29	0.33	2.98	1.01	1.22	0.82	0.42	2.06	0.76
f20	1.09	1.12	0.00	0.00	0.34	0.00	0.00	1.12	1.09	1.39	1.09	1.12	1.09	1.09
f21	0.01	0.04	0.06	0.01	0.02	0.07	0.02	0.01	0.05	0.01	0.04	0.02	0.00	0.01
f22	0.51	0.18	0.36	0.29	0.22	0.67	0.17	0.15	2.81	2.81	2.81	0.46	0.19	0.41
f23	1.32	3.10	1.89	2.99	2.76	3.15	2.57	2.17	1.91	2.23	0.78	2.88	0.80	3.14
f24	1.59	3.43	1.44	4.17	3.49	3.16	1.30	1.32	0.80	0.89	0.79	0.79	0.30	0.70
Sum	15.90	19.67	16.79	20.82	16.94	14.97	17.48	14.09	13.73	17.48	15.14	12.03	10.64	14.56

Table C.27. NELDERMEAD best-to-optimal distance across scenarios (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.00	0.00	0.00	0.34	0.00	0.31	0.34	0.00	0.34	0.00	0.00	0.00	0.00	0.00
f4	0.99	1.04	1.04	0.00	0.00	0.00	0.00	1.17	0.99	0.99	0.31	1.17	0.00	1.00
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.41	0.30	0.41	0.58	0.30	0.58	0.41
f7	1.13	1.04	1.24	1.14	1.15	1.15	1.03	1.04	0.89	1.00	1.06	1.24	0.96	0.94
f8	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.01	0.00	0.01	0.01	0.01	0.00
f9	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.01	0.01	0.02	0.01	0.00	0.00	0.01
f10	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
f11	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00
f12	0.01	0.01	0.02	0.02	0.01	0.00	0.00	0.02	0.02	0.02	0.03	0.00	0.01	0.03
f13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f14	0.00	0.01	0.01	0.01	0.01	0.02	0.03	0.02	0.01	0.01	0.04	0.02	0.01	0.01
f15	0.36	0.36	0.36	0.00	0.00	0.36	0.95	0.00	0.00	0.00	0.00	0.00	0.33	0.36
f16	5.45	5.82	5.97	6.80	5.73	1.67	5.54	0.05	0.02	1.46	1.78	0.80	1.46	0.17
f17	0.17	0.00	0.00	0.35	0.09	0.00	0.04	0.03	0.00	0.01	0.37	0.00	0.00	0.01
f18	1.02	1.22	1.31	0.41	0.74	0.01	1.56	1.11	0.58	1.08	1.42	1.42	0.02	2.69
f19	0.05	0.85	0.00	0.70	0.75	0.00	0.28	3.16	2.21	2.92	0.09	0.05	0.04	0.87
f20	2.81	2.75	2.77	1.12	2.75	1.12	1.12	0.00	1.24	2.25	2.73	2.77	3.62	1.12
f21	0.01	0.02	0.03	0.05	0.03	0.03	0.02	0.02	0.01	0.03	0.01	0.01	0.01	0.02
f22	0.23	0.01	0.09	0.37	0.28	0.51	0.22	0.23	2.81	2.81	2.80	0.36	0.12	0.55
f23	2.72	3.43	0.89	1.48	2.78	3.11	2.42	1.16	2.32	2.33	2.96	0.78	0.69	1.15
f24	1.59	3.73	1.12	4.14	4.01	3.65	0.65	0.95	0.85	0.79	0.79	0.34	0.47	0.70
Sum	16.57	20.32	14.89	16.96	18.34	11.97	14.22	9.41	12.60	16.13	14.99	9.28	8.34	10.05

Table C.28. SBPLX best-to-optimal distance across scenarios (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.00	0.31	0.34	0.34	0.00	0.31	0.34	0.00	0.34	0.00	0.00	0.00	0.00	0.00
f4	0.99	1.04	1.04	0.31	0.31	0.31	0.00	1.17	0.99	0.99	0.31	1.17	0.09	0.99
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.41	0.30	0.41	0.58	0.30	0.58	0.41
f7	0.03	0.01	1.11	0.03	0.01	0.01	0.02	1.20	0.11	0.03	1.23	1.24	0.03	0.03
f8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.00	0.00	0.00	0.00	0.00
f9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00
f10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00
f11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f12	0.00	0.00	0.02	0.02	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.00	0.02
f13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.00
f14	0.00	0.01	0.01	0.03	0.04	0.00	0.02	0.05	0.05	0.04	0.06	0.04	0.03	0.03
f15	0.36	0.36	0.36	0.00	0.00	0.36	0.95	0.00	0.00	0.00	0.33	0.00	0.33	0.36
f16	5.49	5.87	5.97	6.80	5.73	1.66	5.55	0.05	0.03	1.36	1.75	0.83	1.48	0.22
f17	0.04	0.16	0.00	0.57	0.00	0.07	0.04	0.01	0.31	0.01	0.06	0.00	0.08	0.01
f18	1.19	1.01	0.73	0.56	0.51	0.52	1.53	1.05	0.56	1.03	1.34	0.74	0.08	2.71
f19	0.06	0.80	0.07	0.65	0.06	0.08	0.13	3.21	1.87	2.83	0.01	0.02	0.24	0.80
f20	2.81	2.75	2.77	1.12	2.75	1.12	1.12	0.00	1.09	2.25	2.74	2.31	3.62	1.12
f21	0.01	0.01	0.01	0.02	0.01	0.02	0.01	0.02	0.00	0.02	0.00	0.01	0.02	0.01
f22	0.13	0.01	0.08	0.19	0.30	0.47	0.35	0.05	2.81	2.81	2.80	0.36	0.05	0.34
f23	2.65	3.43	0.89	1.48	2.73	3.11	2.42	0.61	2.36	2.07	2.93	1.17	0.69	0.62
f24	1.59	3.82	1.12	4.14	4.01	3.65	0.65	0.95	0.85	0.79	0.79	0.34	0.47	0.70
Sum	15.36	19.62	14.53	16.26	16.47	11.67	13.13	8.81	11.73	14.67	14.95	8.54	7.81	8.37

Table C.29. DIRECT best-to-optimal distance across scenarios (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f4	0.31	0.31	0.31	0.31	0.31	0.31	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.41	0.26	0.41	0.58	0.26	0.58	0.41
f7	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02
f8	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.03	0.14	0.10	0.04	0.08	0.08	0.08
f9	0.01	0.05	0.01	0.12	0.01	0.01	0.17	0.01	0.01	0.01	0.01	0.01	0.02	0.01
f10	0.10	0.21	0.07	0.07	0.21	0.07	0.19	0.17	0.11	0.12	0.24	0.11	0.01	0.12
f11	0.03	0.25	0.03	0.03	0.01	0.13	0.66	0.02	0.06	0.02	0.02	0.01	0.02	0.06
f12	0.07	0.20	0.22	0.22	0.20	0.07	0.07	0.01	0.01	0.01	0.01	0.01	0.00	0.01
f13	0.06	0.04	0.06	0.02	0.02	0.04	0.02	0.01	0.04	0.08	0.00	0.00	0.04	0.01
f14	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.08	0.05	0.08	0.08	0.08	0.05
f15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f16	5.55	5.87	6.06	6.80	5.73	1.67	5.73	0.05	0.01	1.48	1.48	0.75	1.75	0.24
f17	0.01	0.01	0.00	0.00	0.01	0.00	0.01	0.01	0.01	0.01	0.01	0.04	0.01	0.01
f18	0.00	0.02	0.00	0.12	0.01	0.00	0.12	0.03	0.00	0.00	0.00	0.10	0.00	2.63
f19	0.96	1.33	0.96	0.96	0.96	0.96	0.96	2.36	0.01	2.73	1.85	0.53	2.52	1.85
f20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f21	0.03	0.08	0.02	0.08	0.02	0.04	0.01	0.03	0.01	0.01	0.02	0.01	0.00	0.02
f22	0.16	0.30	0.40	0.22	0.30	0.48	0.27	0.01	2.81	2.81	2.81	0.37	0.04	0.48
f23	2.08	3.76	2.34	2.54	2.87	3.11	2.88	2.08	3.80	2.37	3.59	2.81	1.31	3.14
f24	0.76	4.73	1.46	4.51	4.74	0.58	0.87	0.87	0.31	0.81	0.57	0.48	0.80	0.49
Sum	10.17	17.21	12.00	16.05	15.44	7.52	11.99	6.11	7.68	11.05	11.33	5.65	7.28	9.63

Table C.30. CRS2_LM best-to-optimal distance across scenarios (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.01	0.00
f2	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.41	0.30	0.42	0.58	0.30	0.58	0.35
f7	0.01	0.01	0.01	0.02	0.01	0.02	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.02
f8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.00
f9	0.02	0.01	0.01	0.01	0.00	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.01	0.00
f10	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.01	0.00	0.03	0.00
f11	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.01	0.01	0.00	0.01
f12	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.02	0.01	0.00	0.00	0.00	0.00	0.00
f13	0.01	0.00	0.00	0.00	0.02	0.00	0.01	0.00	0.00	0.00	0.01	0.00	0.00	0.00
f14	0.00	0.01	0.01	0.01	0.03	0.01	0.02	0.01	0.00	0.00	0.04	0.02	0.00	0.01
f15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f16	5.73	5.62	6.03	6.80	5.75	1.74	5.79	0.05	0.12	1.55	1.46	0.73	1.72	1.53
f17	0.01	0.01	0.00	0.00	0.00	0.02	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.01
f18	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.79	2.23
f19	0.05	0.03	0.02	0.02	0.09	0.02	0.02	2.11	0.03	2.76	0.02	0.07	0.43	0.12
f20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
f21	0.03	0.03	0.01	0.04	0.07	0.01	0.03	0.02	0.03	0.03	0.03	0.01	0.01	0.01
f22	0.12	0.19	0.08	0.24	0.04	0.26	0.32	0.01	2.80	2.78	2.80	0.36	0.12	0.50
f23	2.38	2.48	0.84	2.65	3.05	2.56	2.11	1.25	2.65	2.04	1.19	2.48	1.94	3.01
f24	0.57	5.53	1.00	5.45	5.45	0.35	0.35	0.66	0.35	0.45	0.70	0.45	0.19	0.20
Sum	8.95	13.92	8.04	15.24	14.50	5.01	8.68	4.56	6.32	10.05	6.87	4.46	5.86	8.03

Table C.31. ISRES best-to-optimal distance across scenarios (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	0.01	0.02	0.02	0.02	0.04	0.03	0.03	0.02	0.02	0.02	0.02	0.02	0.01	0.01
f2	0.33	0.08	0.35	0.17	0.12	0.11	0.70	0.28	0.32	0.42	0.42	0.35	0.18	0.60
f3	0.02	0.02	0.02	0.08	0.06	0.03	0.11	0.04	0.02	0.05	0.01	0.02	0.03	0.04
f4	0.29	0.06	0.04	0.02	0.05	0.02	0.01	0.04	0.02	0.03	0.05	0.03	0.04	0.04
f5	0.02	0.05	0.01	0.01	0.01	0.04	0.03	0.02	0.00	0.00	0.00	0.01	0.00	0.02
f6	0.05	0.07	0.02	0.08	0.03	0.05	0.07	0.35	0.33	0.34	0.47	0.35	0.49	0.32
f7	0.03	0.02	0.01	0.03	0.03	0.04	0.05	0.04	0.02	0.03	0.06	0.02	0.03	0.02
f8	0.27	0.03	0.20	0.22	0.22	0.34	0.02	0.08	0.40	0.16	0.51	0.04	0.04	0.02
f9	0.29	0.05	0.00	0.21	0.03	0.02	0.24	0.17	0.09	0.16	0.02	0.13	0.06	0.28
f10	0.19	0.15	0.61	0.31	0.80	0.36	0.21	0.70	0.01	1.35	0.53	0.04	0.74	0.13
f11	0.17	0.50	0.18	0.38	0.08	0.59	1.39	0.10	0.41	0.01	0.31	0.26	0.14	0.55
f12	0.13	0.61	1.50	0.15	1.13	1.06	0.90	0.38	0.64	0.19	0.20	0.04	0.43	0.36
f13	0.15	0.09	0.04	0.13	0.03	0.04	0.03	0.14	0.09	0.01	0.06	0.02	0.03	0.01
f14	0.02	0.10	0.11	0.02	0.09	0.10	0.01	0.04	0.01	0.02	0.03	0.02	0.02	0.01
f15	0.01	0.01	0.02	0.11	0.03	0.04	0.02	0.02	0.00	0.35	0.06	0.02	0.03	0.03
f16	5.63	5.91	6.07	6.79	5.73	1.82	5.55	0.11	0.01	0.38	0.09	0.73	1.77	0.28
f17	0.13	0.06	0.01	0.04	0.03	0.02	0.02	0.07	0.03	0.05	0.06	0.07	0.03	0.02
f18	0.27	0.18	0.21	0.01	0.12	0.00	0.07	0.04	0.09	0.04	0.23	0.29	0.10	2.30
f19	0.21	0.08	0.36	0.17	0.02	0.06	0.33	2.33	0.13	0.06	0.21	0.29	0.39	0.91
f20	0.06	0.01	0.10	0.03	0.13	0.05	0.05	0.02	0.04	0.03	0.02	0.03	0.03	0.01
f21	0.05	0.00	0.03	0.04	0.01	0.02	0.02	0.02	0.06	0.00	0.02	0.01	0.05	0.05
f22	0.22	0.17	0.26	0.27	0.15	0.39	0.09	0.01	2.81	2.79	2.78	0.49	0.12	0.34
f23	1.38	3.75	4.96	2.98	3.10	0.68	2.43	1.13	2.73	1.16	3.31	3.12	0.56	0.62
f24	0.62	4.76	1.12	0.21	5.48	5.22	0.35	0.64	0.51	0.44	0.57	0.15	0.80	0.46
Sum	10.54	16.79	16.21	12.46	17.52	11.11	12.72	6.76	8.78	8.10	10.04	6.52	6.10	7.40

Table C.32. Optimality gap (min over 5 runs) for solvers w/ original bounds.

Function Code	BOBYQA	COBYLA	PRAXIS	NELDERMEAD	SBPLX	DIRECT	CRS2.LM	ISRES
f1	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02
f2	0.01	0.45	0.00	0.00	0.00	0.00	0.01	1.01
f3	1.05	0.34	0.64	1.05	0.00	0.00	0.00	0.05
f4	0.31	0.36	1.34	0.64	0.61	0.05	0.00	0.03
f5	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.04
f6	0.03	1.01	0.00	0.00	0.00	0.30	0.00	0.10
f7	1.03	1.07	0.05	0.34	0.34	0.02	0.01	0.03
f8	0.17	0.39	0.00	0.00	0.00	0.11	0.00	0.41
f9	0.02	0.35	0.02	0.01	0.00	0.07	0.00	0.21
f10	0.03	0.36	0.05	0.00	0.00	1.05	0.00	0.12
f11	0.14	0.29	0.00	0.00	0.00	0.92	0.01	0.41
f12	0.29	0.12	0.00	0.01	0.00	1.67	0.00	0.19
f13	0.99	1.26	1.10	0.00	0.00	1.90	0.00	0.36
f14	0.07	0.02	0.02	0.01	0.00	0.01	0.01	0.03
f15	0.96	0.95	0.33	1.02	0.97	1.23	0.00	0.03
f16	0.96	1.48	1.05	1.71	1.77	6.75	2.73	0.06
f17	0.78	0.99	1.63	0.89	0.37	0.07	0.01	0.02
f18	0.71	0.36	1.45	0.20	0.53	0.02	0.00	0.08
f19	2.48	2.76	2.64	2.06	1.87	0.71	0.02	0.03
f20	0.00	1.46	2.25	0.34	0.34	0.07	0.00	0.32
f21	2.11	3.22	2.35	3.21	3.21	0.08	0.01	0.06
f22	2.81	2.81	2.98	2.81	2.81	2.81	2.79	0.03
f23	1.31	1.39	1.36	1.53	1.28	2.94	1.67	0.56
f24	4.61	4.26	3.32	4.13	4.25	0.84	0.30	0.34
Sum	20.85	25.70	22.61	19.96	18.35	21.62	7.57	4.55

Table C.33. Summary of Best Scenarios for Each Solver.

Solvers	1			2			3		
	max_depth	PV _{method}	p _e	max_depth	PV _{method}	p _e	max_depth	PV _{method}	p _e
BOBYQA	8	pair-wise	0.3	8	pair-wise	0.0	4	pair-wise	0.3
COBYLA	8	pair-wise	0.0	8	medoid	0.0	8	pair-wise	0.3
PRAXIS	8	pair-wise	0.3	8	pair-wise	0.0	8	medoid	0.0
NELDERMEAD	8	pair-wise	0.3	8	pair-wise	0.0	8	base	base
SBPLX	8	pair-wise	0.3	8	pair-wise	0.5	8	pair-wise	0.0
DIRECT	8	pair-wise	0.0	8	base	base	8	pair-wise	0.3
CRS2.LM	8	pair-wise	0.0	8	base	base	4	pair-wise	0.3
ISRES	8	pair-wise	0.3	8	pair-wise	0.0	8	base	base

C.2.6. A Comparative Analysis of Scenario-Based Models vs. Original Bound Constraints

This subsection provides a binary comparison between scenario-based models and the original bound constraints across all selected BBOB test functions. For each solver, a value of 1 indicates that the scenario-based method achieved a better minimum distance to the optimal solution than the original bounds. Tables C.34 to C.41 summarize the comparative results across different combinations of tree depth, peak-valley methods, and exploration probabilities.

Table C.34. BOBYQA-Binary comparisons with original bounds (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f2	0	0	1	1	0	0	0	1	1	0	0	0	1	0
f3	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f4	1	0	0	0	0	1	0	0	0	0	0	0	0	0
f5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f6	1	1	1	1	1	1	1	1	0	0	0	0	0	0
f7	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f8	1	1	1	1	1	1	1	0	1	1	1	1	1	1
f9	0	0	0	0	0	0	0	0	0	0	1	0	0	0
f10	1	0	1	0	1	1	0	0	0	1	0	0	1	0
f11	0	0	1	1	0	0	0	1	0	1	0	0	1	1
f12	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f13	1	1	1	1	1	1	1	1	0	1	1	1	1	1
f14	1	1	1	1	0	1	1	1	1	1	1	1	1	1
f15	0	1	0	1	1	0	1	1	1	1	1	1	1	1
f16	0	0	0	0	0	1	0	1	1	0	0	1	0	0
f17	1	1	1	1	1	1	1	1	1	1	1	1	1	0
f18	0	0	0	0	0	0	0	0	1	0	0	0	1	0
f19	1	1	1	1	1	1	1	0	1	0	1	1	1	1
f20	0	0	0	0	0	0	0	1	0	0	1	1	1	1
f21	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f22	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f23	0	0	0	0	0	0	0	0	0	0	0	0	1	1
f24	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Sum	16	15	17	17	15	17	15	17	16	15	16	16	20	16

Table C.35. COBYLA-Binary comparisons with original bounds (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f3	1	1	1	1	1	1	1	0	1	0	0	1	0	1
f4	0	1	0	0	1	0	0	0	0	0	0	0	0	0
f5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f6	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f7	0	0	0	0	0	0	0	1	1	1	1	1	1	1
f8	1	0	0	0	0	1	0	0	1	1	1	1	1	1
f9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f11	1	0	0	0	0	0	1	0	0	1	1	1	0	1
f12	1	1	1	0	1	1	1	1	1	0	0	0	0	0
f13	1	1	1	1	1	1	1	1	0	1	1	1	1	1
f14	0	0	0	0	0	0	0	0	1	0	1	1	1	1
f15	1	0	1	0	0	0	0	1	1	1	1	1	1	1
f16	0	0	0	0	0	1	0	1	1	1	0	1	1	1
f17	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f18	0	0	0	0	0	0	0	0	1	0	0	0	1	0
f19	1	1	1	1	1	1	1	0	1	1	1	1	1	1
f20	1	1	0	1	0	1	1	1	1	0	1	1	0	0
f21	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f22	1	1	1	1	1	1	1	1	1	1	0	1	1	1
f23	1	0	0	1	0	0	0	1	0	0	0	1	1	1
f24	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Sum	17	14	13	13	13	15	14	15	18	15	15	19	17	18

Table C.36. PRAXIS-Binary comparisons with original bounds (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	1	1	0	1	1	0	0	0	0	1	1	1	1	0
f2	0	1	0	0	0	0	1	1	0	0	0	0	1	0
f3	1	1	1	0	1	1	1	1	1	1	1	1	1	1
f4	1	1	0	1	1	0	0	0	0	0	0	0	0	0
f5	1	0	0	1	1	1	0	1	0	0	0	1	0	0
f6	0	1	1	1	1	1	1	0	0	0	0	0	0	0
f7	1	1	1	0	1	0	1	0	0	0	0	0	1	0
f8	1	1	1	1	1	1	1	0	0	1	1	1	1	1
f9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f10	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f11	0	0	0	1	0	0	0	0	0	0	1	1	0	1
f12	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f13	1	1	1	1	1	1	1	1	1	0	0	1	1	1
f14	1	1	0	1	0	1	0	1	0	0	0	0	1	0
f15	0	0	0	0	0	0	0	1	1	1	1	0	0	1
f16	0	0	0	0	0	1	0	1	1	0	0	1	0	0
f17	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f18	0	0	0	0	0	0	0	0	1	1	1	1	1	0
f19	1	1	1	1	1	1	1	0	1	1	1	1	1	1
f20	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f21	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f22	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f23	1	0	0	0	0	0	0	0	0	0	1	0	1	0
f24	1	0	1	0	0	1	1	1	1	1	1	1	1	1
Sum	17	16	13	15	15	15	14	14	13	13	15	16	17	13

Table C.37. NELDERMEAD-Binary comparisons with original bounds (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	1	1	1	1	0	1	0	1	1	1	1	1	1	1
f2	0	0	0	0	0	1	1	1	1	1	0	0	0	0
f3	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f4	0	0	0	1	1	1	1	0	0	0	1	0	1	0
f5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f8	1	0	1	0	0	0	1	0	0	0	0	0	0	0
f9	1	1	1	1	1	1	1	0	1	0	1	1	1	1
f10	1	0	0	1	0	1	1	0	0	0	1	1	0	1
f11	1	0	0	1	1	0	0	1	1	0	0	0	0	1
f12	0	0	0	0	0	1	1	0	0	0	0	1	1	0
f13	1	1	0	1	1	0	1	1	1	1	0	1	1	1
f14	1	0	1	0	1	0	0	0	1	1	0	0	1	0
f15	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f16	0	0	0	0	0	1	0	1	1	1	0	1	1	1
f17	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f18	0	0	0	0	0	1	0	0	0	0	0	0	1	0
f19	1	1	1	1	1	1	1	0	0	0	1	1	1	1
f20	0	0	0	0	0	0	0	1	0	0	0	0	0	0
f21	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f22	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f23	0	0	1	1	0	0	0	1	0	0	0	1	1	1
f24	1	1	1	0	1	1	1	1	1	1	1	1	1	1
Sum	15	11	13	14	13	16	15	14	14	12	12	15	17	15

Table C.38. SBPLX-Binary comparisons with original bounds (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	1	1	1	1	1	1	1	0	1	1	0	1	1	1
f2	1	0	1	1	1	1	1	1	1	1	0	1	1	1
f3	0	0	0	0	1	0	0	1	0	1	1	1	1	1
f4	0	0	0	1	1	1	1	0	0	0	1	0	1	0
f5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f6	0	1	0	0	0	1	1	0	0	0	0	0	0	0
f7	1	1	0	1	1	1	1	0	1	1	0	0	1	1
f8	1	0	1	0	0	1	1	0	0	1	1	1	0	0
f9	0	0	1	0	0	0	0	0	0	1	0	0	0	1
f10	0	0	0	0	0	1	0	1	1	1	1	0	0	1
f11	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f12	1	1	0	0	1	1	0	0	0	0	0	0	1	0
f13	0	0	0	0	0	0	0	0	0	0	1	0	0	0
f14	0	0	0	0	0	1	0	0	0	0	0	0	0	0
f15	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f16	0	0	0	0	0	1	0	1	1	1	1	1	1	1
f17	1	1	1	0	1	1	1	1	1	1	1	1	1	1
f18	0	0	0	0	1	1	0	0	0	0	0	0	1	0
f19	1	1	1	1	1	1	1	0	0	0	1	1	1	1
f20	0	0	0	0	0	0	0	1	0	0	0	0	0	0
f21	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f22	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f23	0	0	1	0	0	0	0	1	0	0	0	1	1	1
f24	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Sum	13	12	13	11	15	19	14	13	12	15	14	14	17	16

Table C.39. DIRECT-Binary comparisons with original bounds (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f3	1	1	1	1	1	1	1	1	1	0	1	1	1	1
f4	0	0	0	0	0	0	1	1	1	1	1	1	1	1
f5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f6	1	1	1	1	1	1	1	0	1	0	0	1	0	0
f7	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f8	1	1	1	1	1	1	1	1	0	1	1	1	1	1
f9	1	1	1	0	1	1	0	1	1	1	1	1	1	1
f10	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f11	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f12	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f13	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f14	1	1	1	1	1	1	1	0	0	0	0	0	0	0
f15	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f16	1	1	1	0	1	1	1	1	1	1	1	1	1	1
f17	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f18	1	1	1	0	1	1	0	0	1	1	1	0	1	0
f19	0	0	0	0	0	0	0	0	1	0	0	1	0	0
f20	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f21	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f22	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f23	1	0	1	1	1	0	1	1	0	1	0	1	1	0
f24	1	0	0	0	0	1	0	0	1	1	1	1	1	1
Sum	22	20	21	18	21	21	20	19	21	21	19	22	21	19

Table C.40. CRS2_LM-Binary comparisons with original bounds (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	1	1	1	0	1	1	1	1	0	0	1	1	1	1
f2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f3	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f4	1	0	1	1	1	1	1	1	1	0	1	1	1	1
f5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f6	0	0	0	1	0	1	1	0	0	0	0	0	0	0
f7	0	1	0	0	1	0	0	1	0	1	1	1	0	0
f8	1	0	1	0	1	1	1	0	0	0	0	0	0	0
f9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f10	0	1	1	1	1	1	0	0	0	0	0	1	0	0
f11	1	1	1	1	1	0	1	1	1	1	1	1	1	1
f12	1	0	0	0	1	0	0	0	0	1	1	0	1	0
f13	0	1	1	1	0	1	0	1	1	1	0	1	1	1
f14	1	1	1	1	0	1	0	0	1	1	0	0	1	1
f15	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f16	0	0	0	0	0	1	0	1	1	1	1	1	1	1
f17	1	0	1	1	1	0	1	1	1	1	1	1	1	1
f18	1	0	1	0	0	1	0	1	1	1	0	0	0	0
f19	0	0	0	0	0	1	0	0	0	0	0	0	0	0
f20	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f21	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f22	1	1	1	1	1	1	1	1	0	1	0	1	1	1
f23	0	0	1	0	0	0	0	0	0	1	0	0	0	0
f24	0	0	0	0	0	0	0	0	0	0	0	0	1	1
Sum	14	12	16	13	14	16	12	14	12	15	13	14	15	14

Table C.41. ISRES-Binary comparisons with original bounds (min).

max_depth	4							8						
PV_method	Base	medoid			pair_wise			Base	medoid			pair_wise		
p_e		0.0	0.3	0.5	0.0	0.3	0.5		0.0	0.3	0.5	0.0	0.3	0.5
f1	1	1	1	1	0	0	0	1	1	1	1	1	1	1
f2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f3	1	1	1	0	0	1	0	1	1	0	1	1	1	1
f4	0	0	0	1	0	1	1	0	1	1	0	0	0	0
f5	1	0	1	1	1	1	1	1	1	1	1	1	1	1
f6	1	1	1	1	1	1	1	0	0	0	0	0	0	0
f7	0	1	1	0	0	0	0	0	1	1	0	1	1	1
f8	1	1	1	1	1	1	1	1	1	1	0	1	1	1
f9	0	1	1	0	1	1	0	1	1	1	1	1	1	0
f10	0	0	0	0	0	0	0	0	1	0	0	1	0	0
f11	1	0	1	1	1	0	0	1	1	1	1	1	1	0
f12	1	0	0	1	0	0	0	0	0	0	0	1	0	0
f13	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f14	1	0	0	1	0	0	1	0	1	1	1	1	1	1
f15	1	1	1	0	1	0	1	1	1	0	0	1	1	0
f16	0	0	0	0	0	0	0	0	1	0	0	0	0	0
f17	0	0	1	0	0	1	1	0	0	0	0	0	0	1
f18	0	0	0	1	0	1	1	1	0	1	0	0	0	0
f19	0	0	0	0	1	0	0	0	0	0	0	0	0	0
f20	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f21	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f22	0	0	0	0	0	0	0	1	0	0	0	0	0	0
f23	0	0	0	0	0	0	0	0	0	0	0	0	1	0
f24	0	0	0	1	0	0	0	0	0	0	0	1	0	0
Sum	13	11	14	14	11	12	12	13	16	13	10	16	14	11