Learning Decision Diagrams for Classification via Local Search

by

Max van Dijk

Thesis Committee:	Dr. E. Demirović	Thesis Advisor
	Dr. J. H. Krijthe	
Daily Co-Supervisor:	J. G. M. van der Linden	
Project Duration:	November 2024 – July 2025	
Faculty:	Electrical Engineering, Mathematics and Computer Science, Delft	

TUDelft

Preface

This thesis marks the end of my time at TU Delft. Over the past years, I have learned an incredible amount, both academically and personally. My time here has been challenging, rewarding, and above all, deeply fulfilling. I have had the opportunity to grow as a student, a researcher, and an individual, and this thesis is a reflection of everything I have gained and developed during my time at the university.

I would like to express my sincere gratitude to my thesis advisor, Dr. Emir Demirović, and my daily co-supervisor, Koos van der Linden, for their continuous support, valuable feedback, and guidance throughout the research process. Their insights were instrumental in shaping the direction and quality of this work.

I also want to thank Prof. Pierre Schaus for his thoughtful feedback and the inspiration he provided along the way. Additionally, I am grateful to everyone in the optimization group at TU Delft for creating such an enjoyable and motivating environment to work in.

Finally, I would like to thank my friends and family for their unwavering support and encouragement during the final phase of my studies. Your encouragement has meant a great deal to me.

Max van Dijk
Delft, July 2025

Summary

Interpretable models are essential in many machine learning applications, particularly in domains where transparency and trust are critical. Decision trees are a popular interpretable model, but their structure often leads to repeated identical subtrees and data fragmentation, which can result in large, overfit models with poor generalization. Decision diagrams could offer a more compact and less fragmented alternative by allowing sharing parts of the diagram, but constructing decision diagrams is computationally challenging and has seen limited practical adoption.

This thesis introduces a novel local search-based algorithm for learning binary decision diagrams for classification. Our approach finds a middle ground between early greedy methods and exact optimization techniques, enabling scalable construction of compact and accurate diagrams. We define a local search approach with several move operators and explore multiple metaheuristics, identifying hill climbing with information gain-based initialization as the most effective strategy. We refer to this method as Decision Diagram Local Search (DDLS).

We evaluate DDLS on 57 real-world datasets from the UCI repository and 480 synthetic datasets generated from known diagrams. Our method achieves competitive or superior accuracy compared to state-of-the-art decision tree and diagram methods on real-world datasets, while producing small models with lower fragmentation. Though challenges remain, especially on complex synthetic datasets, our results suggest that DDLS, and decision diagrams in general, hold significant untapped potential as interpretable classifiers.

Contents

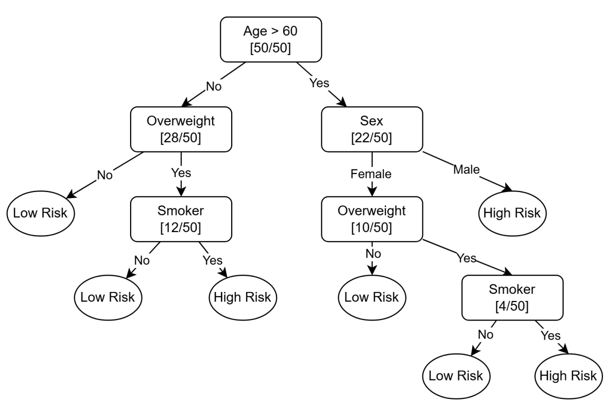
1 Introduction

Classification is an important part of machine learning and artificial intelligence, and is used in many different fields such as medicine, finance, and engineering. In diagnosing diseases, detecting fraudulent activities, or predicting customer preferences, classifiers play a critical role in the decision-making processes by enabling the automated categorization of data into labels. The success of these applications depends greatly on different factors like the accuracy, efficiency, and interpretability of the classifiers used.

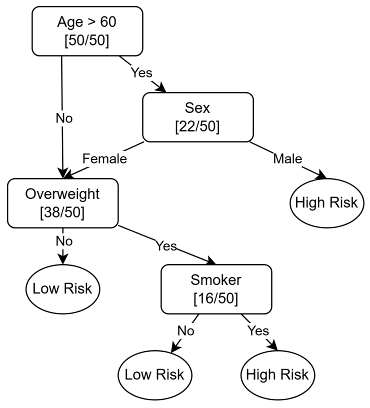
Among the different types of classifiers, interpretable models such as decision trees are highly valued. These models show their decision-making process in a transparent way, making it easy for users to understand how decisions were made and therefore trust the predictions. Additionally, decision tree-based methods are still able to outperform deep learning methods when it comes to tabular data [13, 30]. However, decision trees are not without limitations. One such limitation is that trees struggle to represent disjoint concepts like XOR without duplicating identical subtrees, which significantly increases the overall size of the tree. This leads to problems such as the replication problem, where the same subtree can be found multiple times within the decision tree. Another consequence of this is the fragmentation problem, in which the continuous splitting of the data causes the number of examples for nodes deep in the tree to decrease, making it difficult to create accurate splits [23, 31]. These problems can often lead to unnecessarily large decision trees that might not be able to perform well on unseen data.

Decision diagrams provide an alternative that addresses many of these limitations. Similar to decision trees, decision diagrams are represented as directed acyclic graphs (DAG), where nodes and edges encode the decisions that can be made. However, unlike decision trees, diagrams allow multiple nodes to point to the same child. This structure allows for the sharing of nodes across different parts of the graph, which can help prevent the replication problem. By avoiding the replication problem, the diagram does not have to grow exponentially with depth, which further helps in preventing the fragmentation problem [23]. An example of this is shown in Example 1. This compact structure aligns well with the principles of the minimum description length (MDL) [28], which states that the best description of a dataset is the description with the shortest total encoding. Additionally, the MDL directly correlates with the posterior probability of a dataset [12]. These advantages make decision diagrams theoretically well-suited for classification tasks, especially in applications where it is important to have a small model and good generalizability.

Historically, decision diagrams have seen limited use in classification. Early attempts, of which many used a greedy approach, explored their potential but were hindered by the inherent difficulty of learning decision diagrams that come from having to decide both splits and node merges. Recent advances have revisited this area, with new studies proposing exact methods for constructing optimal decision diagrams using mixed-integer linear programming (MILP) and satisfiability solvers (SAT) [4, 11, 14, 29]. These approaches are able to construct decision diagrams that are optimal in terms of accuracy on the training data, given some limitations on the size or depth of the diagram. Additionally, when compared



(a) Decision Tree



(b) Decision Diagram

Figure 1.1: A decision tree (a) and decision diagram (b) representing the same classification logic. Each decision node displays the number of data samples that pass through it.

to decision trees, several studies find that the resulting decision diagrams show a higher test accuracy compared to decision trees, showing their potential as compact and accurate classifiers [11, 14, 29]. However, these methods struggle with scalability and quickly become computationally infeasible as the problem size increases. Consequently, decision diagrams have not yet gained widespread adoption for large-scale classification tasks, leaving decision trees as the preferred interpretable model.

Example 1. Given a dataset of 50 samples, the goal is to classify any sample in the dataset as high risk when they are either a male over the age of 60, or any individual who both smokes and is overweight. Figure 1.1 shows an example of how both a decision tree and a decision diagram representing this problem could look like. The number of data samples passing through each node is also displayed in the figure for each decision node. As shown, representing this problem in a tree structure leads to a large subtree being replicated, resulting in a significantly larger final tree compared to the more compact decision diagram. This replication not only increases the size of the decision tree but also fragments the data considerably, as the replicated subtrees divide the dataset into smaller subsets. For example, the smallest number of samples passing through any decision node in the decision diagram is 16, whereas this is only four in the decision tree. Basing a decision rule on just four samples is less reliable and may not accurately represent the underlying data, highlighting the advantages of the decision diagram's compact structure.

Since both the early greedy approaches and the more recent optimal methods have significant downsides, this thesis proposes a novel local search algorithm that aims to find a middle ground between these approaches. Inspired by previous research [10], which uses local search for optimizing decision trees, we apply a similar concept to decision diagrams. Instead of attempting to construct globally optimal diagrams, which is currently a computationally intractable problem for large datasets, we design a local search algorithm that iteratively improves a diagram by making local changes. This approach aims to find a decision diagram that is close to the global optimum, without being hindered by the scalability issues that the MILP and SAT-based approaches have.

The results demonstrate that our approach achieves performance that is comparable to or better than state-of-the-art methods on real-world datasets, while producing compact models and scalable runtime. Additionally, we show that our method, and decision diagrams in general, exhibit reduced fragmentation compared to decision trees. We also identify a limitation of our approach on complex synthetic datasets and provide an initial analysis of how these can be overcome. Overall, our findings highlight the broader potential of decision diagrams as a promising direction for interpretable machine learning research.

The main contributions of this thesis are as follows:

1. We propose a novel local search-based algorithm for constructing and optimizing decision diagrams.

2. We conduct a comprehensive empirical evaluation of our local search algorithm, and compare its performance and scalability against state-of-the-art classifiers based on decision trees and decision diagrams.
3. We investigate the practical advantages of decision diagrams over decision trees to assess their effectiveness in real-world datasets and synthetic datasets.

The remainder of this paper is organized as follows. In Section 2, we discuss related work and the historical context of decision diagrams. Section 3 discusses the preliminaries and provides a formal problem definition, after which Section 4 describes our proposed method in detail. Section 5 presents experimental results and analysis. Finally, Section 6 concludes with a discussion and future work.

2 Related Work

During the early development of decision diagrams, several algorithms were proposed for their construction [16, 20, 22, 23, 25]. Most early methods were based on heuristics, starting with the construction of a decision tree followed by greedily merging nodes to create a decision diagram. Despite the similarities between decision trees and diagrams, training and optimizing decision diagrams is significantly more challenging due to the large search space for the structure of the diagram, in contrast to the fixed structure of decision trees. These characteristics have made it difficult to efficiently create decision diagrams compared to decision trees. As a result, decision diagrams have not achieved the widespread adoption that decision trees have.

2.1. Heuristic Decision Diagram Methods

One of the earliest methods for constructing decision diagrams, proposed by Mahoney and Mooney, involved identifying related subtrees in a decision tree and merging them to form a decision diagram [20]. However, they found limited success, as they found their approach to have poor generalizability and to be too computationally expensive. Oliver attempts to fix this problem by using a greedy bottom-up construction algorithm guided by the Minimum Message Length (MML) [23]. In each iteration, candidate splits are defined by individual leaf nodes, while candidate merges are determined by leaf pairs. The MML evaluates these potential modifications and selects the one resulting in the greatest improvement in message length. If the modification reduces the diagram's message length, it is applied. They reported improved classification performance over decision trees on simpler problems. However, later studies found the method ineffective for more complex cases, where the algorithm tended to make premature merges [22].

A more recent study introduces a non-greedy Tree in Tree (TnT) approach for constructing decision diagrams [33]. Inspired by concepts like Network in Network [17] and oblique trees [5], this method operates in two phases. In the growing phase, a node is replaced with a micro decision tree that uses multiple splits, allowing for more accurate decision-making. In the merging phase, the micro decision tree is integrated back into the original model, possibly resulting in a decision diagram. The process starts with a single leaf node and alternates between the growth and merging phases until a number of iterations is reached. Their results demonstrate that the TnT algorithm produces smaller models with higher accuracy, both as standalone classifiers and as part of ensemble learners. TnT has linear time complexity with respect to the number of nodes, which allows it to scale effectively to large datasets, such as MNIST [8], while maintaining a good performance. One drawback of this method is that the depth can be significantly larger compared to CART [3], resulting in an increased inference time.

Yet another paper suggests the use of an evolutionary algorithm (EA) to construct decision diagrams [19]. This method introduces a novel diagram structure that incorporates multiple root nodes and uses a majority voting mechanism through transient terminal nodes. The classification starts from one of the root nodes and follows the diagram until a transient terminal node is reached, at which point a vote is cast for the class of the node. These transient terminal nodes are connected to non-terminal nodes, allowing the classification process to continue and repeat until a fixed number of votes is reached. The evolutionary algorithm drives the generation of these diagrams, using operations like selection, crossover, and mutation to optimize accuracy. Initial results suggest improved performance compared to standard decision trees on a small number of datasets. However, further analysis is needed to fully evaluate the efficacy of this approach. Moreover, the use of a voting mechanism introduces complexity that can reduce the interpretability of the created diagrams.

2.2. Optimal Decision Diagram Methods

An alternative approach that has recently gained attention focuses on optimal methods for constructing decision diagrams. One such approach involves using a SAT-based model to find the optimal diagram [4]. This study introduces a SAT-based method for deriving the smallest Reduced Ordered Binary Decision Diagram (ROBDD) consistent with the given training data. It uses an iterative procedure where the SAT model acts as an oracle. At each iteration, a SAT problem is constructed that checks whether there exists a ROBDD of a given size N that is consistent with the data until the smallest solution is found. It further explores a heuristic approach for deriving suboptimal ROBDDs based on techniques used for logic synthesis. Results show that the SAT-based approach produces diagrams up to five times smaller than those generated using their heuristic method, with improved accuracy on unseen data. The results highlight the limited scalability of the model, as the datasets used for evaluation were limited to fewer than 100 samples.

Another approach for optimal ROBDD construction utilizes a MaxSAT model. One study proposes a MaxSAT model to learn the diagram that minimizes the classification error through soft clauses for the classification constraints while limiting the number of features used in the diagram [14]. The SAT encoding consists of a part for selecting features of the dataset into a feature ordering and a part for generating a truth table that classifies all data samples correctly using the selected feature ordering. A ROBDD can then be generated using this truth table. Because the depth of the ROBDD is limited by the number of features selected by the model, this limits the depth of the ROBDD found. This is in contrast to [4], which only limits the size of the ROBDD and does not place any restrictions on the depth. The resulting ROBDD is further refined by merging compatible subtrees to address fragmentation. Experimental results show that their method achieves competitive prediction accuracy on unseen data with significantly smaller model sizes compared to decision trees of the same depth created using MaxSAT.

Both of these models are designed for datasets with binary features and classes. Addressing this limitation, Shati, Cohen, and McIlraith [29] extend the MaxSAT approach by allowing multi-class classification and by directly encoding numerical features. They propose two methods for creating decision diagrams. In the first approach, the model learns an ordered decision tree which can be reduced to a decision diagram by merging and replacing nodes. Alternatively, the second method uses a technique similar to the TnT model, as it embeds inner decision trees within decision nodes to determine branching. Once a satisfying solution is found, the tree with inner trees is converted into a decision diagram. Results show significant accuracy and runtime improvements over [14], and were most noticeable in datasets with highly numerical features. They furthermore show that using inner trees within decision nodes allows for more expressiveness with a lower computational cost. Similar to previous methods, however, the datasets used in this study were relatively small, with the largest containing fewer than 2,000 samples.

A different perspective is offered by an MILP-based approach for training optimal decision diagrams [11], which also supports multi-class classification and continuous features. It introduces a two-step search strategy in which an initial decision diagram is generated using a greedy approach in the first step. In the second step, MILP is used to find the optimal decision diagram, using the initial decision diagram for guidance in the branch-and-bound search strategy. Similar to our proposed method, this approach requires an initial skeleton that specifies the number of nodes in each layer such that any solution is contained within the skeleton. Results demonstrate that this model produces smaller and more accurate models than optimal decision trees, particularly when multi-dimensional splits are used. However, their algorithm was only able to find the optimal decision diagram in 32% of all runs within the 600-second time limit and was able to consistently improve upon the initial heuristic solution for 33% of the evaluated datasets. Furthermore, the algorithm was only able to consistently improve upon the

Year	Authors	Method	Optimal	Binary features	Continuous features	Binary classes	Multi-class	Large dataset
2016	Mabu et al.	EA		✓		✓	✓	
2021	Zhu et al.	TnT		✓	✓	✓	✓	✓
2021	Cabodi et al.	SAT	✓	✓		✓		
2022	Hu et al.	MaxSAT	✓	✓		✓		
2023	Shati et al.	MaxSAT	✓	✓	✓	✓	✓	
2023	Florio et al.	MILP	✓	✓	✓	✓	✓	

Table 2.1: Overview of key capabilities across different decision diagram construction methods. The columns indicate whether each method: 1) finds the optimal solutions 2) supports binary features; 3) supports continuous features; 4) supports binary classification; 5) supports multiclass classification; and 6) is scalable to datasets with more than 5,000 instances.

heuristic solution for data with binary classes, showing the limitations of this approach. An overview of all discussed methods can be found in Table 2.1

2.3. Decision Tree Local Search

Our proposed method uses a local search approach using ideas inspired by a local search algorithm for learning decision trees [10]. Traditional decision tree construction methods often rely on greedy heuristics, building the tree one locally optimal split at a time. However, these local decisions can result in a final tree that is far from the global optimum. The proposed local search method improves upon this by iteratively and efficiently refining multiple trees until no further improvements are found. They demonstrate that this approach can produce decision trees that significantly outperform classical construction methods such as CART. Furthermore, the resulting decision trees show a performance comparable to random forests and boosted trees, while retaining the interpretability of a single tree. Inspired by this work, our method applies similar concepts to decision diagrams. We focus on optimizing decision diagrams by locally adjusting nodes and edges in order to find solutions close to the global optimum.

3 Preliminaries

This section provides the necessary background on supervised classification and decision trees, introduces the notation and formal definition of decision diagrams, and presents the problem formulation. It also explains the principles of local search and its associated metaheuristics, along with an overview of the framework used for the implementation.

3.1. Supervised Classification

Supervised classification starts with a labeled training dataset $X = \{(x_i, c_i)\}_{i=1}^{n}$, where each $x_i \in \mathbb{R}^p$ is the feature vector representing the i-th sample with p features, and $c_i \in \{0, 1, \ldots, k-1\}$ is the corresponding class label. The number of distinct classes is denoted by k, where $k \geq 2$. The goal of supervised classification is to learn a classification function $h : \mathbb{R}^p \to \{0, 1, \ldots, k-1\}$, such that $h(x_i)$ accurately predicts the class label c_i for each input x_i.

3.2. Decision Trees for Classification

A decision tree is a supervised machine learning model commonly used for classification and regression tasks. In classification, a decision tree partitions the input space into regions associated with different class labels by recursively splitting the dataset based on feature values. Each internal node represents a decision rule on a single feature, and each path from the root to a leaf defines a sequence of decisions that leads to the class prediction. The goal at each split is to maximize the purity of the resulting child nodes, which are typically calculated using criteria such as the Gini impurity or information gain. A sample is classified by traversing the tree from the root to a leaf according to its feature values. The overall objective is to construct a tree that can effectively separate the different classes, resulting in a high prediction accuracy.

3.2.1. Learning Decision Trees

To construct a decision tree, a splitting criterion evaluates the quality of potential splits at each internal node. One common metric that is used in decision tree algorithms such as ID3 and C4.5 is information gain (IG) [26, 27]. Information gain is a concept from information theory that is used to reason about the amount of information gained about a random variable from observing another random variable. In the context of decision trees, the information gain is used as a method to measures the reduction in impurity resulting from a split. Given a dataset X, splitting it on a feature f with threshold θ creates two subsets, X_L and X_R. The information gain is then defined as:

$$IG(X, f, \theta) = H(X) - \left(\frac{|X_L|}{|X|} H(X_L) + \frac{|X_R|}{|X|} H(X_R) \right),$$

with $H(X)$ being the entropy of a given dataset X calculated using:

$$H(X) = -\sum_{c=0}^{k-1} \left(\frac{1}{n} \sum_{i=1}^{n} \mathbb{1}(c_i = c) \right) \log_2 \left(\frac{1}{n} \sum_{i=1}^{n} \mathbb{1}(c_i = c) \right),$$

A higher information gain indicates a better split. Starting from a single root node, algorithms like ID3 work by calculating the information gain for every possible split and selecting the feature and threshold that maximize the information gain. This is recursively repeated on the created subsets until some stopping criteria are met. Leaf nodes are then labeled with the most common class of the samples in the corresponding subset. The Classification and Regression Trees (CART) algorithm uses a similar approach for constructing decision trees except that it uses Gini impurity for selecting the optimal split. Many algorithms such as CART apply pruning after creating the initial tree, which decreases the size of the tree by removing nodes and aims to improve the generalizability of the decision tree.

3.3. Binary Decision Diagrams for Classification

A binary decision diagram is a generalization of a decision tree, designed to model the classification task more flexibly by allowing nodes to have multiple parents. Formally, a decision diagram is represented as a directed acyclic graph (DAG) $D = (V, E)$, where:

- V is the set of nodes, and $E \subseteq V \times V$ is the set of directed edges.
- Internal nodes $v \in V_{internal} \subseteq V$ are associated with decision functions $d_v : \mathbb{R}^p \to \{c_{left}(v), c_{right}(v)\}$, where $c_{left}(v)$ and $c_{right}(v)$ correspond to the left and right child of v, respectively.
- Leaf nodes $V_{leaf} \subseteq V$ represent terminal states and are labeled with a class $c \in \{0, 1, \ldots, k-1\}$.
- An edge $e = (v, v') \in E$ connects a node v to one of its children v', determined by the decision function d_v.

Such a binary decision diagram can be used to represent the mapping $h : \mathbb{R}^p \to \{0, 1, \ldots, k-1\}$ in order to predict the class label of a sample $(x_i, c_i) \in X$. Given a sample $x \in \mathbb{R}^p$, represented by its feature vector, the label can be predicted by traversing the decision diagram. Starting from the root node, the diagram evaluates the decision function $d_v(x)$ at each internal node v to determine which outgoing edge to follow. The traversal proceeds through the diagram until a leaf node is reached. Once a leaf node is reached, the class label associated with the leaf is returned as the predicted class for x.

3.4. Problem Definition

The primary objective is, for a given dataset X, to construct a binary decision diagram $D = (V, E)$ that accurately learns the mapping $h : \mathbb{R}^p \to \{0, 1, \ldots, k-1\}$, and minimizes the objective function $\mathcal{L}$. This objective function attempts to balance the accuracy and the size of the model, and will be defined in Section 4. By balancing the training accuracy and the complexity, we aim to capture the underlying structure of the data and avoid overfitting on the training set.

Constructing an optimal decision diagram presents several challenges. Although binary decision diagrams offer advantages over decision trees, such as compactness and shared substructures, these benefits come at the cost of a much larger, and more complex search space of potential graph structures, making optimization computationally expensive. Consequently, the design of our approach must take into account both the quality of the resulting diagram and the computational efficiency of the optimization process.

3.5. Local Search

Local search is a heuristic optimization method aimed at finding the most optimal solution within a search space $\mathcal{S}$, where each element $s \in \mathcal{S}$ represents a candidate solution, or state. It is particularly effective for solving combinatorial optimization problems where an exact solution might be hard to obtain due to the problem's complexity. The quality of a solution is evaluated using a loss function $\mathcal{L}(s)$, which the algorithm seeks to minimize. Formally, the goal is to find an optimal state $s^* \in \mathcal{S}$ such that

$$s^* = \operatorname*{argmin}_{s \in S} \mathcal{L}(s)$$

The algorithm begins from an initial state and iteratively explores neighboring solutions by applying small changes, called moves, to the current solution. These moves define the neighborhood $\mathcal{N}(s)$ of a state s, given by:

$$\mathcal{N}(s) = \{s' \in \mathcal{S} \mid s' \text{ is reachable from } s \text{ by a single move}\}.$$

At each iteration, the local search algorithm evaluates the neighbors $s' \in \mathcal{N}(s)$ and typically selects a neighbor that reduces the loss. This way, the local search explores the search space by moving between neighboring solutions. This process continues until certain stopping criteria, such as a number of iterations without an improvement, are met. Local search is an anytime algorithm, meaning that it can return a valid solution at any point during its execution, with the solution quality typically improving the longer the algorithm runs

Local search can be categorized into various techniques depending on how the neighbors are selected and the stopping criteria. However, it typically struggles with finding globally optimal solutions because it can become stuck in a local optimum, where the current solution is better than direct neighbors, but worse compared to other regions in the search space. Local search therefore does not provide a guarantee that any given solution is optimal.

3.5.1. Metaheuristics

To address this limitation, more advanced strategies called metaheuristics are used. Metaheuristics are higher-level strategies that are designed to explore a large search space efficiently and guide a search algorithm towards a global optimal solution. They are generally not problem-specific and are flexible in tackling a variety of different optimization tasks. Metaheuristics are often employed when classical optimization techniques are not able to provide efficient solutions due to the size and complexity of the problem space. The following section introduces several widely used metaheuristics that are applied in this thesis.

Hill climbing Let $s \in \mathcal{S}$ denote the current state of the decision diagram, and let $\mathcal{N}(s) \subseteq \mathcal{S}$ be the set of neighboring states reachable from s via a single move. Starting from an initial state s_0, the algorithm iteratively explores the neighborhood $\mathcal{N}(s_t)$ at iteration t by randomly sampling a neighbor $s' \in \mathcal{N}(s_t)$. If the new cost $\mathcal{L}(s') \leq \mathcal{L}(s_t)$, the move is accepted and the algorithm proceeds to the new state $s_{t+1} = s'$; otherwise, the algorithm remains in the current state. The process continues until a certain number of iterations without improvements occurs or a maximum number of iterations is reached.

Algorithm 1 Hill Climbing (HC)

```
1: **Input:** Initial state $s_0$, maximum iterations $N$, , maximum idle iterations $N_{\text{idle}}$
2: **Output:** Best state found $s_{\text{best}}$
3: $s \leftarrow s_0$
4: $s_{\text{best}} \leftarrow s_0$
5: for iteration $i = 1$ to $N$ do
6:     Sample a neighbor $s' \in \mathcal{N}(s)$
7:     if $\mathcal{L}(s') \leq \mathcal{L}(s)$ then
8:         $s \leftarrow s'$
9:         if $\mathcal{L}(s) < \mathcal{L}(s_{\text{best}})$ then
10:            $s_{\text{best}} \leftarrow s$
11:        end if
12:    end if
13:    if no improvement for $N_{\text{idle}}$ iterations then
14:        break
15:    end if
16: end for
17: return $s_{\text{best}}$
```

Hill climbing is one of the simplest metaheuristics for local search. Unlike more advanced metaheuristic, standard hill climbing does not have any way of escaping local optima. It strictly accepts only moves that improve the objective function, which makes it prone to becoming trapped in suboptimal solutions. There are, however, variants of the hill climbing algorithm such as stochastic hill climbing and random restart hill climbing that can help overcome this problem by introducing randomness. The exact algorithm for hill climbing can be found in Algorithm 1.

Simulated annealing Simulated annealing is a stochastic optimization algorithm that extends hill climbing by allowing occasional uphill moves to escape local minima. Let $s_t \in \mathcal{S}$ be the current state at iteration t, and let $\mathcal{L}(s)$ denote the cost function. At each step, a neighbor $s' \in \mathcal{N}(s_t)$ is sampled at random. The new state is accepted with probability:

$$P(s_t \to s') = \begin{cases} 1 & \text{if } \mathcal{L}(s') < \mathcal{L}(s_t), \\ \exp\left(-\frac{\mathcal{L}(s')-\mathcal{L}(s_t)}{T_t}\right) & \text{if } \mathcal{L}(s') \geq \mathcal{L}(s_t), \end{cases}$$

where $T_t > 0$ is a temperature parameter controlling the acceptance of worse solutions. The temperature T_t decreases over time according to a cooling schedule. A commonly used schedule is exponential decay:

$$T_{t+1} = q \cdot T_t,$$

where $q \in (0, 1)$ is the cooling rate. This process continues for a fixed number of iterations or until the temperature has reached a specified value close to zero.

The algorithm balances exploration and exploitation, where during the early iterations, it explores widely, accepting not only improving solutions, but also accepting solutions with a worse objective value. During later iterations, the algorithm focuses on fine-tuning the solution in order to find a solution close to the global optimum. The full algorithm for simulated annealing is shown in Algorithm 2.

Algorithm 2 Simulated Annealing

```
1: **Input:** Initial state $s_0$, initial temperature $T_0$, cooling rate $q$, minimum temperature $T_{\text{min}}$, maximum iterations $N$
2: **Output:** Best state found $s_{\text{best}}$
3: $s \leftarrow s_0$
4: $s_{\text{best}} \leftarrow s_0$
5: $T \leftarrow T_0$
6: $i \leftarrow 0$
7: while $T > T_{\text{min}}$ and $i < N$ do
8:     Sample a neighbor $s' \in \mathcal{N}(s)$
9:     Compute cost difference $\Delta = \mathcal{L}(s') - \mathcal{L}(s)$
10:    if $\Delta < 0$ or Uniform$(0,1) < \exp(-\Delta/T)$ then
11:        $s \leftarrow s'$
12:        if $\mathcal{L}(s) < \mathcal{L}(s_{\text{best}})$ then
13:            $s_{\text{best}} \leftarrow s$
14:        end if
15:    end if
16:    $T \leftarrow q \cdot T$
17:    $i \leftarrow i + 1$
18: end while
19: return $s_{\text{best}}$
```

Iterated Local Search Iterated Local Search (ILS) tries to escape local optima by introducing perturbations. The algorithm begins by finding a locally optimal solution using a local search algorithm such as the hill climbing algorithm. Once a local optimum is reached, the solution is perturbed to create a new starting point, and the local search is applied again. This process is repeated iteratively, enabling the algorithm to explore parts of the search space that it might not have visited without the perturbation. Finding an effective perturbation algorithm is critical to the success of

ILS. A perturbation that is too small risks returning to the same local optimum, while one that is too large effectively reduces the method to a random restart, negating the benefits of the solution that was already found.

3.6. EasyLocal++

For the implementation of the local search algorithm, we use the EasyLocal++ framework [9]. EasyLocal++ has been an effective optimization tool for over 25 years across various domains, including timetabling, rostering, scheduling, and logistics [6]. It has shown to be able to produce state-of-the-art results in benchmarks and competitions, including a second-place ranking in the ITC-2021 competition, demonstrating its effectiveness and reliability.

EasyLocal++ is an object-oriented framework aimed at simplifying the development and analysis of local search algorithms. It offers a collection of abstract classes and interfaces that contain the essential components of local search and their extensions, such as tabu search and simulated annealing, by modularizing the algorithmic structure. This approach enables users to focus on problem-specific elements, such as the definition of the states and moves, while making use of reusable components for the invariant parts of the algorithm. This structure, along with built-in debugging tools and support for batch experiments, makes EasyLocal++ particularly effective for implementing these kinds of problems.

4 Method

This chapter presents a two-phase local search approach for constructing decision diagrams for classification. In the initialization phase, we first create a fixed, layered DAG structure without assigned features or thresholds, selecting from predefined architectures, as shown in Figure 4.1. Each structure specifies the number of layers and maximum nodes per layer, allowing for flexible diagram depth. The empty decision diagram is initialized using random initialization, or initialization using information gain. In the local search phase, we optimize the initialized diagram by changing the feature and threshold of decision nodes and by moving the edges between layers. Using a cost function that makes use of both the training accuracy and the size of the diagram, the output of the procedure is a decision diagram optimized for the training data, while the regularization prevents the model from overfitting.

4.1. Diagram Initialization

Before starting the local search phase, we create an initial diagram used as a starting point for the local search during the initialization phase.

4.1.1. Decision Diagram Structural Templates

The initialization phase starts by selecting a skeleton for the decision diagram, which specifies the overall structure that will be used as a starting point. A skeleton is defined by a number of layers, where each layer contains a specified number of decision nodes. The diagrams we use have a strictly layered structure, where, except for the nodes in the final layer, every node in layer d is connected exclusively to nodes in the subsequent layer $d + 1$. Since our local search optimization does not allow adding additional nodes, the selection of this skeleton is important. Figure 4.1 shows the skeleton templates that we use in our approach. These templates make it easy to adjust the depth of the diagram without needing to increase its width, allowing for smaller models compared to tree structures. Additionally, they can easily be extended to include additional templates. Besides the skeletons shown in Figure 4.1, we also test our approach on a tree structure.

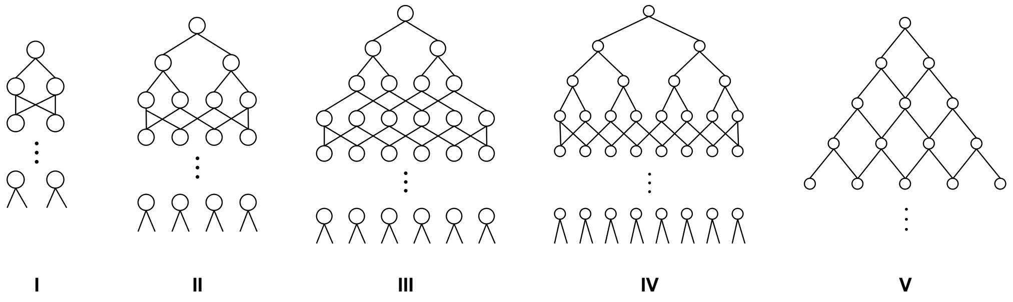


Figure 4.1: The five different structural templates for decision diagrams used in our approach.

A notable aspect of our approach is that our decision diagrams do not explicitly use leaf nodes. Instead, the outgoing edges from the nodes in the final layer of the decision diagram are directly responsible for classifying the samples that reach them, using majority voting. This design choice simplifies the structure, since leaf nodes are not essential in decision diagrams and can always be reduced to the number of classes in the dataset. By only focusing on decision nodes, we make it easier to reason about the model's complexity, especially regarding its size. However, for clarity and interpretability, we still include leaf nodes in visualizations of the diagrams.

4.1.2. Initialization of Decision Functions

After selecting the structure of the decision diagram, each internal node must be assigned an initial decision function. This function determines how samples are routed at that node. For a given node $v \in V$, we associate a feature index $f(v) \in \{1, \dots, p\}$ and a threshold value $\theta(v) \in \mathbb{R}$. For our approach, we define the decision function at node v as:

$$d_v(x) = \begin{cases} c_{left}(v), & \text{if } x_{f(v)} \leq \theta(v), \\ c_{right}(v), & \text{otherwise,} \end{cases}$$

where $x \in \mathbb{R}^p$ is the input feature vector, $x_{f(v)}$ is the value of feature $f(v)$ for x, and $c_{left}(v), c_{right}(v) \in V$ are the children of node v. We consider two strategies for initializing the decision functions:

Random Initialization. For each decision node v, we pick a feature $f(v)$ uniformly at random from the feature set $\{1, \dots, p\}$, and a threshold $\theta(v)$ by randomly selecting a value of feature $f(v)$ from the training data. This results in a random assignment of decision functions throughout the diagram, serving as a simple baseline from which we can start the local search. While this random initialization does not produce high-quality solutions on its own, it provides a starting solution for the local search optimization phase.

Information Gain Initialization. For each node v in breadth-first search order, we use the data to choose a split that maximizes information gain. Since the diagram is a DAG, a node v can have multiple parents, so we combine all incoming samples into a dataset called D_v. We then select the feature f^* and threshold θ^* that maximize the information gain on D_v. For each feature, we consider all unique candidate thresholds, and compute the information gain of each potential split. Since the children of D_v can also have incoming data from other parents, we use $D_{c_{left}(v)}$ and $D_{c_{right}(v)}$ to denote the data in its children coming from other parents combined with the data coming from v with the current feature and threshold. We then set $(f(v), \theta(v)) = (f^*, \theta^*)$, assigning the best split to node v. This process is shown in Algorithm 3 below.

4.2. Diagram Local Search Optimization

After the diagram has been initialized, we have the initial solution where each node has a decision function defined. We then continue to phase two in which we refine this solution using local search.

4.2.1. Solution Representation

Let V be the set of nodes in the decision diagram. We represent a solution by a state S, which is a mapping:

$$S : V \to \mathcal{F} \times \Theta \times V \times V,$$

where $s \in S$ is a candidate solution and $S(v) = (f(v), \theta(v), c_{left}(v), c_{right}(v))$. Here, $f(v)$ is the chosen feature at node v, $\theta(v)$ is the threshold, and $c_{\text{left}}(v), c_{\text{right}}(v)$ are the left and right children of v. By design, edges always go from layer d to $d + 1$. The neighborhood $\mathcal{N}(s)$ of a state $s \in S$ is the set of states obtained by applying a single move to s, which are defined in Section 4.2.3. We search this space with local moves that change the decision function for one node or redirect one edge.

4.2.2. Cost Function

The local search is guided by a cost function that balances the training accuracy with model complexity, thereby promoting generalization and preventing overfitting. We define $\mathcal{L}(D, X)$ to be the total cost of a

Algorithm 3 Diagram Initialization Using Information Gain

```
1: **Input:** Diagram $D(V, E)$, training data $X$
2: $D_{v_{root}} \leftarrow X$
3: **for** each $v \in V$ in BFS order **do**
4:     $H_v \leftarrow Entropy(D_v)$
5:     **if** $D_v = \emptyset$ **then**
6:         $f(v) \leftarrow -\infty$
7:     **else**
8:         $IG^* \leftarrow -\infty$
9:         **for** each $f \in \mathcal{F}$ **do**
10:            $f(v) \leftarrow f$
11:            **for** every possible threshold $\theta$ for feature $f$ **do**
12:                $\theta(v) \leftarrow \theta$
13:                $H_{left} \leftarrow Entropy(D_{c_{left}(v)})$
14:                $H_{right} \leftarrow Entropy(D_{c_{right}(v)})$
15:                $w_{left} \leftarrow \frac{|D_{c_{left}(v)}|}{|D_{c_{left}(v)}|+|D_{c_{right}(v)}|}$
16:                $w_{right} \leftarrow \frac{|D_{c_{right}(v)}|}{|D_{c_{left}(v)}|+|D_{c_{right}(v)}|}$
17:                $IG \leftarrow H_v - (w_{left} \cdot H_{left} + w_{right} \cdot H_{right})$
18:                **if** $IG > IG^*$ **then**
19:                    $IG^* \leftarrow IG$
20:                    $f^* \leftarrow f$
21:                    $\theta^* \leftarrow \theta$
22:                **end if**
23:            **end for**
24:            $f(v) \leftarrow f^*$
25:            $\theta(v) \leftarrow \theta^*$
26:        **end for**
27:    **end if**
28: **end for**
```

decision diagram D on a dataset X. This cost combines the misclassification loss with a regularization term that penalizes the number of activated decision nodes. The cost function is given by:

$$\mathcal{L}(D, X) = \frac{1}{\hat{L}} L(D, X) + \alpha \cdot complexity(D)$$

where $L(D, X)$ is the number of misclassified samples in X using diagram D, and $\hat{L}$ is the baseline classification error obtained by predicting the majority class for all inputs, which makes the effect of the regularization part independent of the dataset size. The parameter $\alpha \geq 0$ controls the trade-off between classification accuracy and model complexity. The term $complexity(D)$ counts the number of active decision nodes in D, excluding any that have been deactivated. This formulation encourages the search process to produce smaller and more interpretable diagrams without sacrificing accuracy.

4.2.3. Move Operators

We define four types of moves, each modifying one node or one of its edges in the diagram:

1. **Random feature and threshold:** This move works by first selecting a decision node v uniformly at random. Then, a feature $f(v)$ is assigned by drawing uniformly from the entire set of features $\mathcal{F}$. After selecting the feature, the threshold $\theta(v)$ is set by randomly sampling from the values of that feature in the training data.

   The purpose of this move is to promote exploration within the search space. By introducing randomness into the decision functions, it helps the algorithm escape local optima and explore diverse solutions. This is particularly useful in metaheuristics like simulated annealing, where accepting worse solutions occasionally can help find better overall results in the long run.

2. **Locally optimal threshold given random feature:** In this move, a decision node v is again selected uniformly at random, and a feature $f(v)$ is chosen randomly from $\mathcal{F}$. However, instead of assigning a random threshold, the threshold θ^* is chosen to minimize the cost function for the given node and feature.

   To efficiently determine θ^*, the data is pre-sorted by the selected feature. By scanning through the candidate thresholds in sorted order, the cost function can be incrementally updated at each potential split without recalculating from scratch. Starting from $-\infty$, which directs all samples to the left child, the threshold moves stepwise through data points, enabling fast evaluation of the optimal split. The algorithm used for this is shown in Algorithm 4.

   This move is focused on exploitation, as it refines the decision diagram by finding the locally optimal threshold for a given feature, contrasting with the previous move's emphasis on exploration.

3. **Globally optimal threshold:** This move performs a more exhaustive search for improvements by finding the threshold that reduces the cost the most, across all nodes in the diagram. It systematically evaluates all features $f \in \mathcal{F}$ and their corresponding candidate thresholds θ for every node $v \in V$ to find the split that achieves the lowest cost according to the cost function.

   The threshold search for each feature follows the efficient procedure described previously. This move aims to identify the globally optimal split at the node, making it useful for fine-tuning the model when close to an optimum. However, this also results in a large computational cost, especially on large datasets with many features.

4. **Edge redirecting:** This move allows structural changes within the decision diagram. It begins by selecting a node v from non-final layers and then randomly choosing one of its outgoing edges. The selected edge is then reassigned to point to a random node in the next layer, with the restriction that it cannot point to the same node as its other outgoing edge.

Algorithm 4 Algorithm for finding the optimal threshold in linear time

```
1:  **Input:** Current state $s \in S$, and node to optimize $v$ with $f(v) = f$ and $\theta(v) = -\infty$; a mapping $classCounts$ from final-layer edges to class-wise sample counts; the cost of the current diagram $startCost$; the current diagram accuracy $startAcc$; and the data samples reaching node $v$, sorted by feature to optimize $f$, dataset $D_v$.
2:  $cost_{best} \leftarrow startCost$
3:  $\theta^* \leftarrow \theta(v)$
4:  **for** $i = 0$ **to** $|D_v|$ **do**
5:      $x_i \leftarrow D_v(i)$
6:      $finalEdge \leftarrow classifyLeft(D, v, x_i)$    ▷ Get final-layer edge when $x_i$ is directed to $v$'s left child
7:      $classCounts(finalEdge) \leftarrow classCount(finalEdge) - 1$
8:
9:      $finalEdge \leftarrow classifyRight(D, v, x_i)$    ▷ Get final edge when $x_i$ is directed to $v$'s right child
10:     $classCounts(finalEdge) \leftarrow classCount(finalEdge) + 1$
11:
12:     **if** $i < |D_v| - 1$ **then**
13:         $\theta(v) \leftarrow \frac{x_i + x_{i+1}}{2}$
14:     **else**
15:         $\theta(v) \leftarrow \infty$
16:     **end if**
17:
18:     $cost \leftarrow compCost(s, classCounts)$    ▷ Compute new cost using $s$ and $classCounts$
19:     **if** $cost < cost_{best} \land x_i \neq x_{i+1}$ **then**
20:         $cost_{best} \leftarrow cost$
21:         $\theta^* \leftarrow \theta(v)$
22:     **end if**
23: **end for**
24: **return** $\theta^*$
```

By redirecting edges, this move enables the model to explore alternative diagram topologies. Adjusting the connections can help the decision diagram better capture data patterns and improve overall performance.

Each move generates a neighboring state by modifying either the feature or threshold of a single node, or by redirecting one of its edges. Together, these moves enable fine-tuning of the decision functions and structure within the diagram template.

An important consideration is how to select which move to apply during the search process. Since different moves contribute differently, with some promoting exploration and others focusing on exploitation, it is important to strike a balance between these two aspects. To achieve this, we employ a neighborhood search strategy that assigns specific probabilities to each move type. By tuning these probabilities, we can control how often each move is selected, enabling the search to explore new regions of the solution space while still focusing on improvements in promising areas. This should help the algorithm avoid premature convergence on local minima and improve the chances of finding high-quality solutions.

4.2.4. Local Search Metaheuristics

For both hill climbing and simulated annealing, we use the default implementations provided by EasyLocal++, as described in Section 3. In contrast, the implementation of Iterated Local Search (ILS) requires additional customization, which we describe in detail below:

Iterated Local Search ILS alternates between local search and a perturbation in order to escape local minima and explore different regions of the search space. In our case, we use hill climbing as the local search. Once hill climbing converges, a perturbation step is applied to modify the current solution. This perturbation randomly selects a fixed number of decision nodes and reassigns each of them a new feature and threshold, sampled from the dataset. The modified solution then serves as the starting point for the next round of hill climbing, allowing the algorithm to potentially escape suboptimal areas and continue improving. The full algorithm can be seen in Algorithm 5 and Algorithm 6.

4.2.5. Node Deactivation and Pruning

A distinctive feature of our method is the concept of node deactivation, which allows the model to effectively remove internal decision nodes during the local search without actually modifying the structure of the diagram. Specifically, we consider a node v to be deactivated when its threshold is set to an extreme value, either $\theta(v) = +\infty$ or $\theta(v) = -\infty$. In such cases, the decision at node v becomes trivial, routing all incoming samples exclusively to one of its children. This essentially makes the node equivalent to a passthrough.

We formally mark these nodes as deactivated, and importantly, they are excluded from the model's size when computing the complexity penalty in the objective function. This mechanism enables a form of soft regularization. Rather than explicitly removing nodes, which would complicate the structure and

Algorithm 5 Iterated Local Search (ILS)

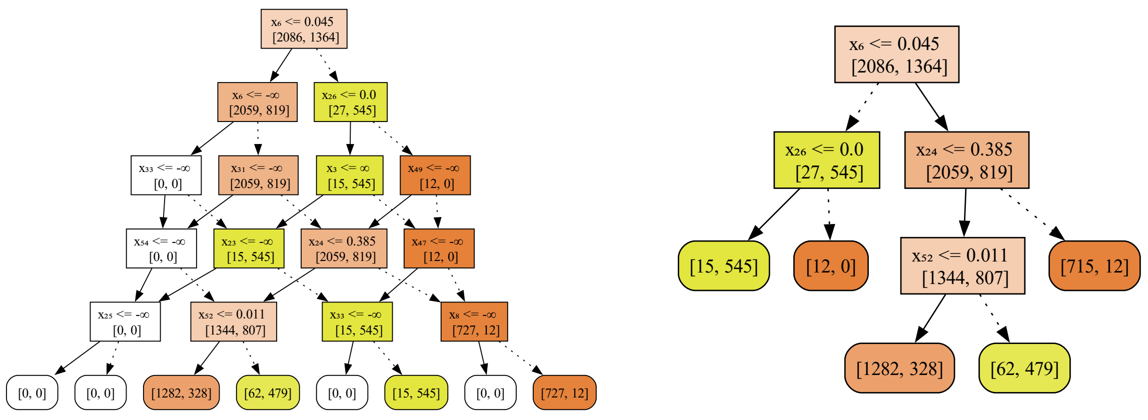
```
1:  **Input:** Initial state $s_0$, number of iterations $N$, perturbation strength $\rho$
2:  **Output:** Best state found $s_{\text{best}}$
3:  $s \leftarrow$ HillClimb($s_0$)
4:  $s_{\text{best}} \leftarrow s$
5:  **for** iteration $i = 0$ to $N$ **do**
6:      $\tilde{s} \leftarrow$ Perturb($s, \rho$)
7:      $s' \leftarrow$ HillClimb($\tilde{s}$)
8:      **if** $\mathcal{L}(s') < \mathcal{L}(s_{\text{best}})$ **then**
9:          $s_{\text{best}} \leftarrow s'$
10:     **end if**
11:     $s \leftarrow s'$
12: **end for**
13: **return** $s_{\text{best}}$
```

Algorithm 6 Perturb(s, ρ)

```
1: Input: State $s$, number of nodes to perturb $\rho$
2: Output: Perturbed state $\tilde{s}$
3: $\tilde{s} \leftarrow s$
4: Randomly select $\rho$ decision nodes $\{v_1, v_2, \dots, v_\rho\} \subseteq V$
5: for each selected node $v_i$ do
6:     Choose a feature $f \sim$ Uniform($\mathcal{F}$)
7:     Sample a threshold $\theta$ from the observed values of feature $f$
8:     Set decision function $d_{v_i}(x) = \begin{cases} c_{left}(v) & \text{if } x_f \leq \theta \\ v_{right}(v) & \text{otherwise} \end{cases}$
9: end for
10: return $\tilde{s}$
```

the optimization process, our method allows the search to implicitly simplify the model by deactivating nodes that do not contribute meaningful decisions.

Once the local search is concluded, we perform a pruning step to clean up the diagram. Each deactivated node v is removed from the graph, and its parents are reconnected directly to its only active child. This results in a cleaner, more compact decision diagram. An example of this pruning process is shown in Figure 4.2. As in the original model, class labels are determined by the outgoing edges of the final-layer nodes. The figure depicts leaf nodes for illustration purposes only. In practice, classification is handled directly via the final-layer edges.



(a) Before pruning **(b)** After pruning

Figure 4.2: Example of pruning deactivated nodes from a decision diagram. Each node contains the decision function and the number of nodes per class reaching that node. The color indicates the purity of the node.

5 Experimental Results

In this section, we present a comprehensive experimental evaluation of our proposed method to assess its performance, interpretability, and practical applicability. The evaluation is structured around the following key research questions:

1. **What are the most effective configurations for our method, and to what extent does the choice of initialization influence final performance?**
   We find that information gain-based initialization significantly improves test accuracy across all metaheuristics, with hill climbing combined with information gain yielding the best performance overall.
2. **How closely can our method approximate the optimal decision diagrams produced by an exact solver, and how does this relate to generalization on unseen data?**
   Our method reaches near-optimal training accuracy on the datasets where the MILP approach successfully found the optimal solution, while also achieving higher test accuracy in most cases.
3. **How does our approach compare to state-of-the-art methods for learning decision trees and decision diagrams, in terms of both accuracy and interpretability?**
   Our method demonstrates performance comparable to, or better than state-of-the-art interpretable approaches, achieving the best average rank and highest average test accuracy. Moreover, it consistently produces compact models while maintaining lower fragmentation.
4. **What are the advantages of using decision diagrams in our method compared to using tree-based structures, in terms of performance and interpretability?**
   Our decision diagram-based approach significantly outperforms its tree-based counterpart in terms of test accuracy. While the resulting diagrams tend to be somewhat deeper than the trees, they show substantially less fragmentation.
5. **How does our method perform on synthetic data generated from decision diagrams, and how is this affected by changes in dataset characteristics and noise levels?**
   Surprisingly, DDLS underperforms on the synthetically generated datasets, possibly due to the high complexity of the datasets. We test a random restart approach and show that this improves the performance.
6. **How does the computational efficiency of our method scale with dataset size?**
   Although slower than the other heuristic methods, DDLS scales effectively to large datasets and remains significantly more efficient than exact solvers.

5.1. Experiment Setup

To thoroughly evaluate our method, we conducted experiments on a diverse collection of 57 datasets sourced from the UCI Machine Learning Repository [15]. These datasets span a wide range of domains and vary significantly in complexity, providing a robust basis for our comparisons. The number of samples per dataset ranges from 47 to over 45,000, covering both small-scale and large-scale sce-

Dataset	n	p	k	Dataset	n	p	k
acute-inflam-nephritis	120	6	2	iris	150	4	3
acute-inflam-urinary	120	6	2	magic-telescope	19020	10	2
balance-scale	625	4	3	mammographic-mass	830	10	2
bank-marketing	45211	16	2	monks1	556	11	2
banknote-auth	1372	4	2	monks2	601	11	2
blood-transfusion	748	4	2	monks3	554	11	2
breast-cancer-diag	569	30	2	optical-recognition	5620	64	10
breast-cancer-prog	194	33	2	ozone-eighthr	1847	72	2
breast-cancer-wisc	699	9	2	ozone-onehr	1848	72	2
car-evaluation	1728	15	4	parkinsons	195	22	2
chess-kr-vs-kp	3196	37	2	pima-indians-diabetes	768	8	2
climate-simulation	540	18	2	planning-relax	182	12	2
congressional-voting	435	16	2	qsar-biodegradation	1055	41	2
connect-mines-rocks	208	60	2	seeds	210	7	3
connectionist-vowel	990	10	11	seismic-bumps	2584	20	2
contraceptive-method	1473	11	3	soybean-small	47	35	4
credit-approval	653	37	2	spambase	4601	57	2
cylinder-bands	277	484	2	spect-heart	267	22	2
dermatology	366	34	6	spectf-heart	267	44	2
dry-bean-dataset	13611	16	7	statlog-german-credit	1000	48	2
echocardiogram	61	9	2	statlog-landsat-sat	6435	36	6
fertility-diagnosis	100	12	2	teaching-assistant	151	52	3
habermans-survival	306	3	2	thoracic-surgery	470	24	2
hayes-roth	160	4	3	thyroid-ann	3772	21	3
heart-disease-cleveland	297	18	5	thyroid-new	215	5	3
hepatitis	80	19	2	tic-tac-toe	958	18	2
image-segmentation	2310	18	7	wall-following-robot-2	5456	2	4
indian-liver-patient	579	10	2	wine	178	13	3
ionosphere	351	33	2				

Table 5.1: The 57 UCI Datasets used during the experiments, with the number of samples n, features p, and classes k per dataset.

narios. In addition, the datasets show considerable variation in the number of features (ranging from two to 484 features) and the number of target classes (from binary classification up to 11 classes). This diversity ensures our evaluation reflects a wide range of real-world scenarios. A detailed summary of all datasets used, including the number of samples, features, and classes, is provided in Table 5.1. Unless otherwise specified, we use five-fold cross-validation for datasets with over 100 samples and ten-fold cross-validation for datasets with fewer than 100 samples, to minimize variance. From the training set, 25% of the data is reserved as a validation set for hyperparameter tuning. Once tuning is complete, the model is retrained on the full training set using the hyperparameters that achieved the highest validation accuracy.

Besides the real-world datasets, we also evaluate our method using synthetic data. This allows us to precisely control the characteristics of the dataset, such as the number of samples, features, and classes, as well as the degree of noise introduced in both features and class labels. We generate a total of 480 synthetic datasets based on a procedure similar to the methods described by Dunn [10], and van der Linden et al. [18]. Each dataset is derived from one of six base decision diagrams constructed using skeletons I, III, and V, for both depth five and depth ten, in order to model different complexity levels. For every internal node in the diagram, we randomly select a feature and a corresponding threshold $\theta \sim [0, 1]$. The final edges are labeled from left to right, with class labels assigned in a round-robin fashion to achieve a balanced class distribution. We ensure that every leaf node is reached by at least five samples to maintain meaningful class representation.

To create synthetic training sets, we uniformly sample n instances from $[0, 1]^p$, where p is the number of features. Feature noise is introduced by adding a perturbation sampled uniformly from the interval

$[-g, g]^p$ to each feature vector, where g is the feature noise strength. Each sample is labeled using the ground truth diagram, after which class noise is applied by flipping labels with a given probability.

For each configuration, we generate five different training and test datasets to account for variance. Corresponding test sets are constructed with size $n = complexity(D) \cdot (p - 1) \cdot 500$, where D is the decision diagram used to label the data, ensuring sufficient coverage. The default configuration uses 1,000 training samples, five features, two classes, and no noise. We systematically vary one factor at a time to analyze its effect, testing across multiple settings for the number of instances ($n = 250, 500, 1000, 5000$), features ($p = 2, 5, 15, 25$), and classes ($k = 2, 4, 6, 8$). For feature noise, we test four levels $(g = 0, 0.2, 0.6, 1)$, and for class noise we test noise rates of $0\%, 20\%, 40\%,$ and 50%.

To assess whether observed performance differences are statistically significant, we use two approaches. For pairwise comparisons, we apply the Wilcoxon signed-rank test alongside Akinshin's Gamma, a rank-based effect size measure indicating the magnitude and direction of differences [32]. For multiple method comparisons, we use the Nemenyi critical distance (CD) test when the Friedman test indicates global significant difference [1, 21]. This non-parametric test is designed for comparing multiple methods across multiple datasets and is based on average ranks rather than raw performance values, making it a robust method for measuring significance. Two methods are considered significantly different if the difference in their average ranks exceeds the critical distance. For both methods, we use a significance level of $\alpha = 0.05$. Both the Wilcoxon signed-rank test and the Nemenyi critical distance rank test are recommended for fair, statistically sound classifier evaluation [7], and form the basis of our analysis.

All experiments were conducted on an HP ZBook Studio G5 equipped with an Intel Core i7-8750H CPU (2.2 GHz, 6 cores) and 16 GB of RAM.

5.2. Method Configuration and Performance Analysis

We first describe how we configured the different settings of our approach, after which we analyze the impact of the different metaheuristics and initialization methods.

5.2.1. Hyperparameter Tuning

We begin by evaluating how different metaheuristics and initialization methods affect the performance of our approach using the UCI datasets. As previously mentioned, 25% of each training set is reserved as a validation set for hyperparameter tuning. Some hyperparameters are general and apply across all methods, while others are specific to particular metaheuristics. Due to the nature of some of these parameters, we use Bayesian optimization with a limit of 30 evaluations to tune our hyperparameters.

Three general hyperparameters are tuned for every metaheuristic: 1) the diagram structural template, selected from the five possible structures described in Section 4; 2) the depth d of the diagram, constrained to integer values within the range $1 \leq d \leq 10$, thereby limiting the maximum depth of the final diagram; and 3) the regularization parameter α, which is searched over the interval $[10^{-6}, 0.1]$ using a log-uniform distribution. Below, we provide details on how the metaheuristic-specific hyperparameters are tuned or fixed for each metaheuristic.

Hill Climbing Hill climbing requires one additional hyperparameter, which we fix to a constant value: the maximum number of idle iterations. This parameter determines how many consecutive iterations without improvement in the objective function are allowed before termination. We set this limit to 10,000, based on the observation that further improvements become unlikely after such an extended period without progress. Given that the maximum number of decision nodes in our diagrams is 63, this iteration limit makes it likely that all move operations have been applied multiple times to each node. If no improvement has been found after this extensive search, it is reasonable to assume that further progress is unlikely.

Simulated Annealing Simulated annealing introduces several hyperparameters that we tune. The first is the initial temperature T_0, which controls the level of exploration in the early search phase. We constrain this value to the range $60 \leq T_0 \leq 100$. The second is the cooling rate q, which determines how rapidly the temperature decays during the search. We restrict it to $0.8 \leq q \leq 0.995$, balancing exploration and convergence speed. The algorithm terminates once the temperature drops below a fixed minimum threshold, which we set to 0.0001 in all experiments.

Iterated Local Search For iterated local search, we tune the number of repetitions, limiting the number between two and five to balance exploration, while allowing sufficient time for optimization. The total number of allowed iterations is divided evenly among the repetitions. For example, if the overall iteration budget is 50,000 and two repetitions are used, each repetition is allowed up to 25,000 iterations. Too few repetitions may limit exploration and increase the risk of getting stuck in local optima, while too many reduce the effectiveness of each individual search phase. At the start of each repetition, the current solution is perturbed by randomly selecting a subset of decision nodes and reassigning their feature and threshold values randomly, as described in Section 4.2.4. The number of perturbed nodes is treated as a tunable parameter, selected from the range two to ten.

In addition to the hyperparameters tuned during search, we fix several general parameters: namely, the maximum number of local search iterations and the selection probabilities for each move. To determine suitable move probabilities, we included them in the Bayesian optimization process along with the other hyperparameters, using a fixed budget of 20,000 iterations. We then ran these experiments for all metaheuristics using random initialization on the 57 UCI datasets. The resulting distributions are shown in Figure 5.1. Based on these results, we set the move probabilities to 0.35 for the random threshold move, 0.6 for the optimal threshold move, and 0.05 for the edge move. This configuration aims to balance exploration and exploitation while minimizing structural changes. Limiting edge modifications helps preserve structural stability and prevents unnecessary overhead near convergence, where most moves redirecting an edge have likely already been explored. Notably, the exhaustive move that iterates over all nodes and features to find the globally best split is not included in this figure. This move proved to be too computationally expensive without yielding noticeable performance gains and was therefore excluded.

Having established these selection probabilities, we then focused on determining an appropriate maximum number of iterations, which is a critical hyperparameter for each metaheuristic. Using the fixed move selection probabilities, we varied the iteration limit and evaluated the training accuracy, test accuracy, and diagram size. For each dataset, these metrics were normalized relative to their optimal values and aggregated as percentages over the range of iterations. The results, shown in Figure 5.2, indicate that after approximately 50,000 iterations, test accuracy plateaus and further training may lead to overfitting. We therefore set the maximum number of iterations to 50,000.

Lastly, Figure 5.2 shows that diagram size decreases rapidly in early iterations, likely because disabling nodes is the optimal move at that time, which could possibly hinder the discovery of optimal solutions. To mitigate this, we experimented with gradually increasing the regularization parameter α during training. However, we did not find noticeable improvements using this approach.

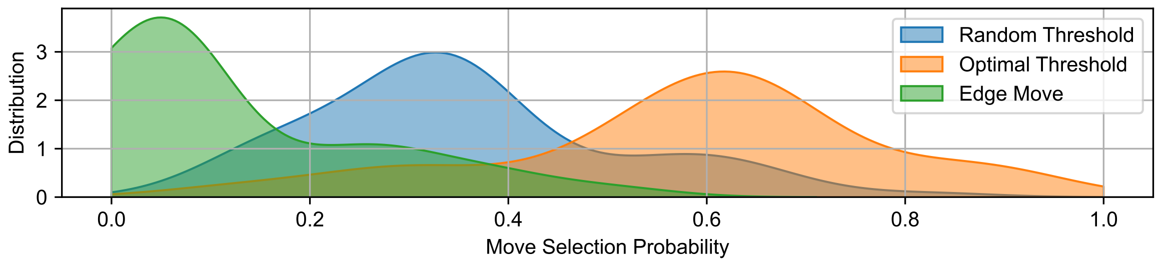


Figure 5.1: Distribution of move type frequencies across all metaheuristics on the 57 UCI datasets. Random Threshold refers to the move selecting a random node, feature, and threshold. Optimal Threshold refers to the move that selects a random node and feature, and selects the locally optimal threshold. Edge Move refers to the move selecting a node and redirecting one of its outgoing edges to a random node in the next layer.

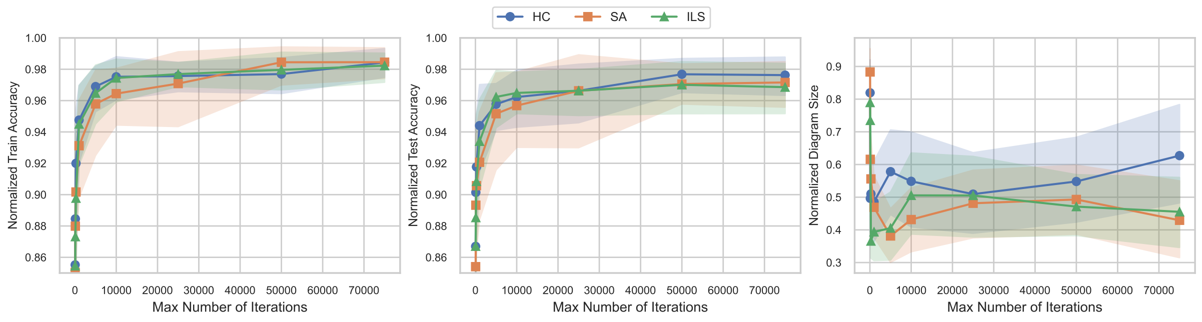
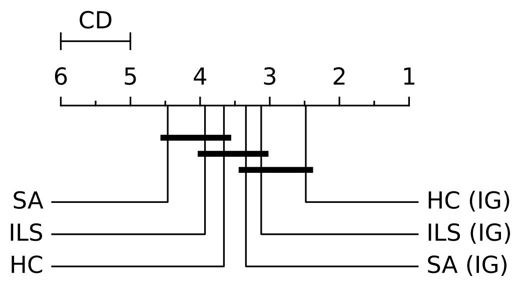
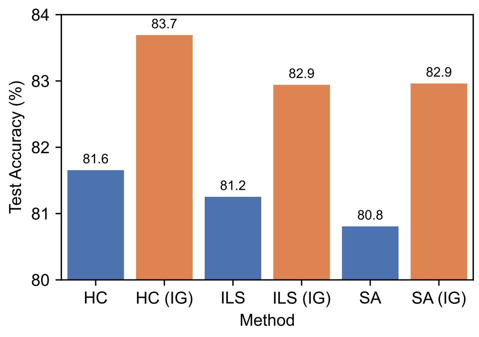


Figure 5.2: Comparison of normalized training accuracy, test accuracy, and diagram size as a function of the maximum number of iterations for each metaheuristic: Hill Climbing (HC), Simulated Annealing (SA), and Iterated Local Search (ILS), using random initialization. Error bars represent 95% confidence intervals across the evaluated datasets.



(a) Nemenyi critical distance rank test.



(b) Average test accuracy of each method.

Figure 5.3: Performance evaluation of the test accuracy for different metaheuristics and initialization methods for the 57 UCI datasets. The tested methods are Hill Climbing (HC), Simulated Annealing (SA), and Iterated Local Search (ILS). Using both random initialization and initialization using information gain (IG).

5.2.2. Performance Comparison of Metaheuristics and Initialization Methods

We conducted experiments on 57 datasets from the UCI repository and present the results in Figure 5.3. The figure displays the outcomes of the Nemenyi critical difference test alongside the average test accuracy for each metaheuristic, comparing both random and information gain (IG) initialization strategies.

The results clearly demonstrate that, across all metaheuristics, initializing the decision diagram using information gain consistently leads to a large improvement in test accuracy. This emphasizes the importance of starting from a strong initial solution, suggesting that the quality of the initial diagram has a lasting impact on the outcome of the optimization process. We also observe that the relative performance ranking of the metaheuristics remains consistent across initialization strategies, suggesting that the local search phase contributes a roughly uniform improvement relative to the quality of the initial solution.

Among the metaheuristics evaluated, hill climbing with information gain initialization (HC (IG)) emerges as the most effective approach, achieving the highest average rank and the best overall test accuracy across the benchmark datasets. In contrast, simulated annealing with random initialization (SA) performs the worst on average, indicating that its probabilistic exploration strategy may be less effective in this context. One possible reason why ILS (IG) performs worse than HC (IG) is that it may require more uninterrupted iterations to reach a high-quality solution. As shown in Figure 5.2, test accuracy for hill climbing continues to improve between 20,000 and 50,000 iterations. Since ILS splits this budget across multiple repetitions, each interrupted by random perturbations, it may struggle to make the final refinements needed for optimal performance. Given these results, we select the hill climbing method with information gain initialization as our primary algorithm for further experiments. To simplify discussion throughout the remainder of this thesis, we refer to this configuration as Decision Diagram Local Search (DDLS).

Dataset	*n*	DDLS	MILP	Dataset	*n*	DDLS	MILP
acute-inflam-nephritis	120	100	100	iris	150	100	100
acute-inflam-urinary	120	100	100	soybean-small	47	100	100
echocardiogram	61	100	100	thyroid-new	215	99.9	100
hepatitis	80	100	100	wine	178	100	100

Table 5.2: Comparison of training accuracy (%) between DDLS and MILP on the 8 UCI datasets where MILP found a provably optimal solution within a 20-minute timeout. Both methods use the same decision diagram structure (Structure II, depth five) with univariate splits and zero regularization. DDLS achieves near-optimal accuracy, closely matching MILP's performance.

5.3. Comparison Against Optimal Decision Diagrams

To assess how closely DDLS can approximate the diagrams found by exact methods, we begin by comparing the training accuracy of our method, DDLS, with an exact Mixed Integer Linear Programming (MILP) approach for decision diagrams under identical conditions [11]. Specifically, both methods are evaluated using the same decision diagram structure and objective function. Additionally, we evaluate out-of-sample performance while allowing hyperparameter tuning.

5.3.1. Comparing Training Accuracy

To evaluate how close DDLS can approximate the optimal decision diagrams produced by MILP, we compare both methods under identical conditions. Specifically, both methods are limited to using structure II with a depth of five, and we set the regularization parameter to zero for both methods. This ensures that both methods use the same objective function, which now focuses only on minimizing the number of misclassifications. To maintain comparability, and since we aim to produce interpretable decision diagrams, we configure MILP to use only univariate splits. We evaluate the performance across all 57 UCI datasets, using the full dataset (100%) for training and a timeout of 20 minutes per run.

MILP failed to improve upon its initial heuristic solution in 49 out of the 57 datasets within the timeout. As we are interested in assessing how close DDLS gets to the true optimum, we report results only for the eight datasets where MILP successfully found a provably optimal solution. The results are presented in Table 5.2. As shown, DDLS matches MILP's training accuracy on most datasets, with MILP slightly outperforming DDLS in only one case. This indicates that DDLS is effective at reaching near-optimal solutions. It is, however, worth noting that these datasets are relatively simple, with few instances and fairly low dimensionality, and therefore may not be fully representative of the performance on more complex, larger datasets.

5.3.2. Comparing Test Accuracy and Runtime

In addition to training accuracy, we also assess generalization performance by evaluating test accuracy on the same eight datasets. Unlike the training setup, test evaluation requires hyperparameter tuning. For MILP, we tune the regularization parameter α, which controls the trade-off between accuracy and model simplicity. Following the original study, we consider $\alpha \in \{0.01, 0.1, 0.2, 0.5, 1\}$. MILP also requires

Dataset	**Test Accuracy (%)**		**Runtime (s)**	
	DDLS	MILP	DDLS	MILP
acute-inflam-nephritis	98.7	**99.2**	**<1**	**<1**
acute-inflam-urinary	**100**	**100**	**<1**	1
echocardiogram	**96.9**	95.0	**<1**	**<1**
hepatitis	**86.5**	82.5	**<1**	310
iris	**94.7**	91.7	**<1**	275
soybean-small	97.5	**97.8**	**<1**	**<1**
thyroid-new	**96.3**	91.9	**<1**	546
wine	**93.0**	91.1	**<1**	261

Table 5.3: Comparison of average test accuracy and runtime between DDLS and MILP on the eight UCI datasets where MILP found provably optimal solutions. DDLS achieves higher test accuracy on most datasets with significantly lower runtime, demonstrating its efficiency and generalization performance. Bold values indicate the better result between the two methods for each metric.

a predefined skeleton that specifies the number of nodes per layer. For this, we use the skeletons from their work: (1–2–4–8), (1–2–4–4–4), (1–2–3–3–3–3–3), and (1–2–2–2–2–2–2–2), where each number indicates how many nodes are in that layer. To ensure a fair comparison, we enforce a maximum of 15 decision nodes for DDLS as well. As before, we apply a 20-minute timeout per run for MILP.

Table 5.3 compares the test accuracy and runtime of DDLS and MILP on the eight UCI datasets where MILP successfully found provably optimal solutions. The results reveal that DDLS outperforms or matches MILP on six out of eight datasets in terms of test accuracy. Interestingly, this suggests that the optimal solution obtained by MILP does not necessarily guarantee better generalization. One possible explanation lies in the way both method tune their hyperparameters. DDLS allows for more possible structures and is also more flexible in tuning the regularization parameter, in contrast to the MILP approach which only evaluates five possible values for α. The Wilcoxon signed-rank test confirms that this difference is statistically significant ($p = 0.031$), with DDLS exhibiting a higher average test accuracy with a small effect size ($\gamma = -0.36$).

In addition to accuracy, Table 5.3 reports runtime comparisons. Despite the relative simplicity of the datasets, with the largest containing only 214 samples, MILP frequently requires several minutes to complete, whereas DDLS consistently finishes in under one second. This substantial difference highlights the efficiency and scalability advantages of our method. Due to its limited scalability, we do not include MILP in any further comparisons.

5.4. Comparison Against State-of-the-Art Methods

In this section, we evaluate our algorithm against several state-of-the-art methods using the 57 datasets from the UCI repository. We assess both predictive performance and interpretability metrics, such as model size, to provide a comprehensive comparison. This evaluation helps determine not only how well our method performs on real-world data, but also whether it maintains the interpretability that makes decision diagrams appealing for machine learning.

5.4.1. Methods and Hyperparameter Tuning

We compare our method against four different approaches: 1) the TnT approach; 2) CART with cost-complexity pruning; 3) CART with cost-complexity pruning limited to a maximum depth of six; and 4) Interpretable AI (IAI) [10], a local search model for decision trees that inspired this research. For both IAI and the depth-limited CART, we set the maximum depth to six, which corresponds to the maximum number of decision nodes allowed by our method's largest possible structure, structure IV of depth ten, ensuring a fair comparison.

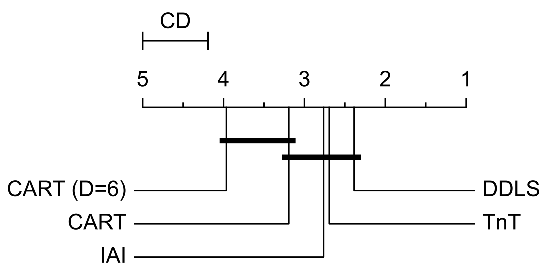
All of these methods require some form of hyperparameter tuning in order to produce the diagrams or trees. We follow the experimental setup described in Section 5.1. Below, we explain how we tune hyperparameters for each method:

TnT The TnT model requires tuning of a regularization parameter α. Following the approach in their original paper, we sample 30 values of α equally spaced on a logarithmic scale between 5×10^{-5} and 0.1. We fix the number of merging phases $N_1 = 2$ and growing phases $N_2 = 5$, as specified in their experiments. Each configuration is trained on the training set, and the best α is selected based on validation accuracy. The best α is then used to train on the entire training set, after which we use the test set to measure test accuracy.

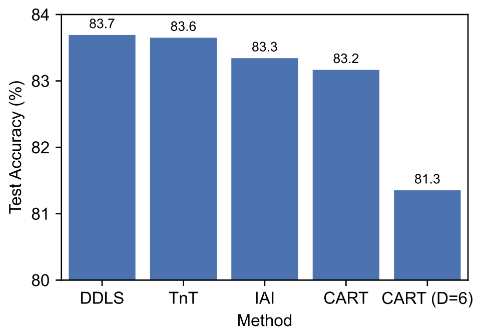
CART For CART, we use cost-complexity pruning to tune the regularization parameter α using the validation accuracy. This procedure generates a sequence of subtrees by progressively pruning branches that offer the least improvement in the cost-complexity criterion, which balances model accuracy with tree complexity. We use the best α to retrain on the full training set and report accuracy on the test set.

CART (D=6) This variant follows the same procedure as standard CART, using cost-complexity pruning to tune α, but limits the maximum tree depth to six.

IAI For IAI, hyperparameters are optimized using the method's built-in tuning routine. It uses grid search and a validation set for finding the best depth and tunes α during training. The maximum tree depth is limited to six to ensure comparability with other methods. Once the best α is



(a) Nemenyi critical distance ranking test.



(b) Average test accuracy of each method.

Figure 5.4: Test accuracy comparison across 57 UCI datasets for different methods.

identified, the model is retrained on the full training set using the found parameters, and the test accuracy is reported on the test set.

5.4.2. Model Performance Analysis

The test accuracy results are presented in Figure 5.4, where (a) displays the Nemenyi critical distance ranking test, and (b) shows the average test accuracy for each method. As illustrated, DDLS achieves the best overall performance, ranks best in the Nemenyi test and also achieves the highest average test accuracy. In contrast, CART (D=6) performs the worst, with both the highest rank and the lowest accuracy, and is significantly outperformed by all methods except the CART baseline without a depth limit. Among the remaining methods, TnT ranks second, followed closely by IAI.

Although the Nemenyi test does not indicate a statistically significant difference between DDLS and TnT, IAI, or CART, pairwise comparisons using Wilcoxon signed-rank tests reveal that DDLS achieves significantly higher test accuracy than IAI ($p = 0.044$, $\gamma = -0.036$). However, the effect size is negligible, suggesting that the practical differences are small despite being statistically significant.

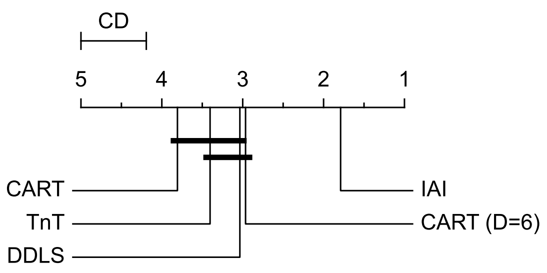
A notable observation is the significant performance gap between the two CART variants. We can see that CART struggles with achieving high test accuracy when restricting the maximum size of the tree. In contrast, both DDLS and IAI, despite being subject to the same size constraints, maintain a strong performance under limited model capacity.

5.4.3. Model Complexity and Interpretability Analysis

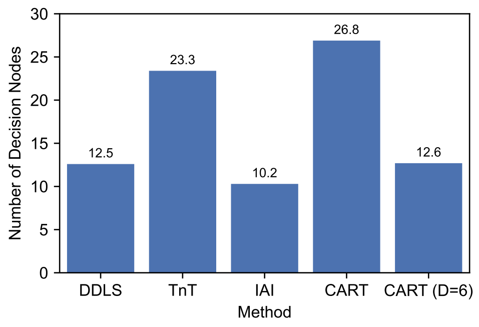
Beyond test accuracy, we are also interested in evaluating the interpretability and structural properties of the trees and diagrams generated by the different methods. Our goal is to gain deeper insight into the types of structures produced by our approach, how they compare to those created by other decision tree and decision diagram methods, and how these structural differences influence both interpretability and performance. To this end, we consider several key metrics, which are explained below:

Number of decision nodes The first metric is the number of decision nodes in the tree or diagram. Previous research has demonstrated that the size of decision trees plays a significant role in determining how interpretable it is [24], and may also indicate better generalization by reducing fragmentation. We focus exclusively on decision nodes rather than leaf nodes, because decision diagrams can merge leaf nodes to match the number of classes, which would make direct comparisons with trees unfair.

Average support per node Second, we analyze the degree of fragmentation in the structure. To quantify this, we use the statistical support per decision node, which is defined as the percentage of training samples that reach a given decision node. By averaging this value across all decision nodes, we obtain an overall measure of fragmentation, where a higher average support suggests lower fragmentation. This metric allows us to assess how fragmented the models produced by each method are and whether decision diagrams actually reduce fragmentation compared to decision trees.



(a) Nemenyi critical distance ranking test.



(b) Average number of decision nodes of each method.

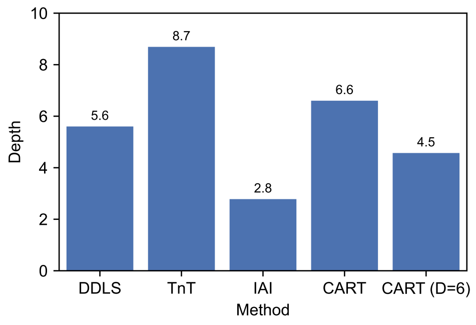
Figure 5.5: Comparison of the number of decision nodes for different methods for the 57 UCI datasets.

Depth and question length Finally, previous studies have demonstrated that the depth of a decision model is a key factor in its interpretability, with shallower models generally being more interpretable [24]. For this reason, we examine the depth and the question length of the models. The depth corresponds to the longest path from the root to a leaf node, while the question length is defined as the average number of decision nodes visited by a sample before reaching a leaf. Unlike depth, question length incorporates the frequency of sample paths and provides a more representative view of the average complexity of the classification process.

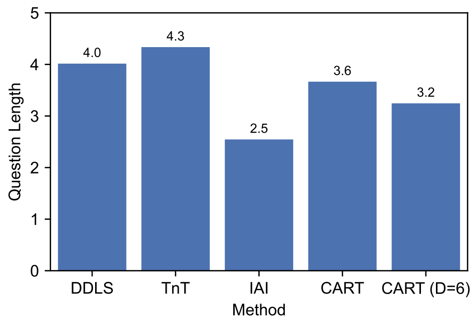
Figure 5.5 presents both the Nemenyi critical distance ranking test and the average values for the number of decision nodes for each method. We see that IAI significantly outperforms the other methods in terms of size, achieving both the best rank and the lowest average number of decision nodes. Following IAI are CART (D=6), and DDLS. We also notice that, since neither TnT nor CART imposes constraints on model size, their average number of decision nodes is considerably higher compared to the methods that are limited in the number of decision nodes.

The Nemenyi critical distance ranking test shows that the models produced by DDLS are not significantly smaller than those generated by the other methods. In fact, DDLS exhibits comparable performance to CART (D=6) in terms of both average rank and the average number of decision nodes. This goes against the expectation that the diagrams produced by DDLS would be smaller than the decision trees created by other methods. A similar pattern is observed for TnT, which not only produces relatively large models but also performs worse than most other methods.

From Figure 5.6, we observe the average depth and average question length for all methods. These results align more closely with expectations. Both decision diagram models, DDLS and TnT, exhibit relatively high depth and question length. This behavior is expected, as decision diagrams do not need to grow as wide as decision trees and can instead go deeper. This is especially the case for our method where the width of the possible structures is fairly limited. To preserve expressiveness under such limitations, the models must grow deeper, resulting in increased depth and question length.

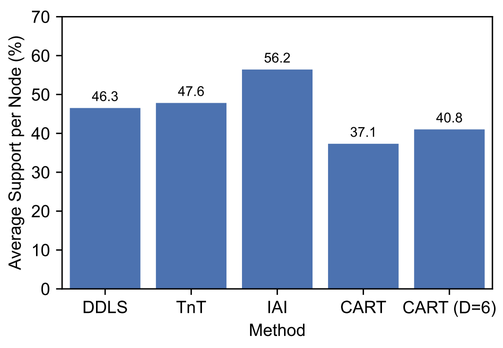


(a) Average depth for each method.

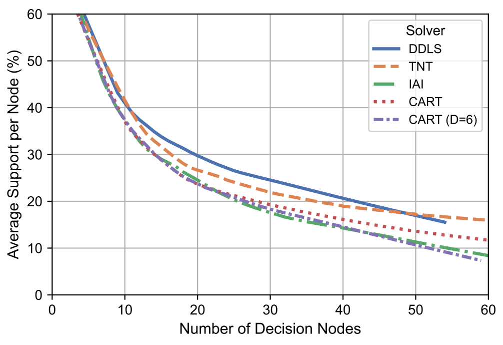


(b) Average question length of each method.

Figure 5.6: Comparison of the depth and question length for different methods for the 57 UCI datasets.



(a) Average fragmentation.



(b) Fragmentation for different sizes.

Figure 5.7: Fragmentation analysis of the models produced by each method. (a) Shows the average statistical support per decision node, where higher values indicate less fragmentation. (b) Displays the relationship between average support per node and model size.

CART also exhibits a relatively high average depth, which may be attributed to outlier cases where it generates particularly large trees. In contrast, IAI demonstrates the most compact structure, achieving the lowest values in both average depth and question length, suggesting more interpretable models.

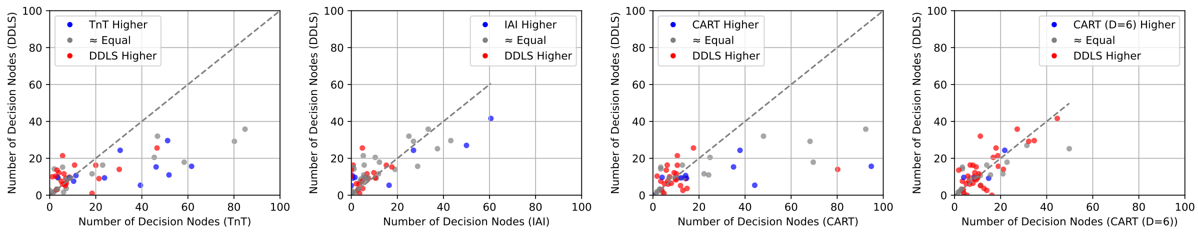
Finally, we analyze the degree of fragmentation across the different methods. The results are presented in Figure 5.7. The average statistical support per decision node is shown in (a), where higher values indicate lower fragmentation. Both DDLS and TnT show relatively high average support, suggesting they experience less fragmentation than other methods. This is particularly notable given that these methods also tend to produce larger models than CART (D=6). Despite the increased size, the decision diagrams generated by DDLS and TnT maintain a higher average support per node, likely because they grow deeper rather than wider. By leveraging the structural flexibility of decision diagrams to combine rather than exclusively split data, they can expand in size without introducing excessive fragmentation. We also observe that IAI achieves the highest average support overall, indicating the lowest fragmentation. This aligns with previous observations that IAI produces the smallest and shallowest trees.

Since fragmentation is inherently related to model size, (b) presents a more size-independent view by plotting average support per decision node against the number of decision nodes. For each dataset and run, we scatter the average support per node over the number of decision nodes in the resulting model. We then apply a locally weighted scatterplot smoothing (LOWESS) curve to reveal trends in fragmentation relative to model size [2].

From this plot, it becomes clear that both DDLS and TnT consistently have higher average support across varying model sizes, indicating that they produce less fragmented models even as their size increases. For example, a diagram generated by DDLS with 40 decision nodes can exhibit similar average support to a decision tree from other methods with only 25 nodes. This highlights the ability of decision diagrams to scale in size while limiting the amount of fragmentation. In contrast, the decision tree methods exhibit comparable levels of fragmentation for a given model size, suggesting that the tree structure makes it more difficult to manage fragmentation as the number of nodes increases.

5.4.4. Trade-Offs Between Model Size and Predictive Performance

Lastly, we analyze how the size of the models produced by each method compares to that of DDLS, and how this affects the test accuracy. For this, we calculate the average number of decision nodes and the average test accuracy per dataset for each method, and plot them against the corresponding values of DDLS. In these scatter plots, shown in Figure 5.8, each point represents a dataset, with the y-axis showing the average size of DDLS and the x-axis the average size of the comparison method. The color of each point indicates which method achieved higher average test accuracy on that dataset, or shows gray when both methods performed similarly. This visualization allows us to examine the relationship between model size and predictive performance across different datasets.



(a) DDLS and TnT **(b)** DDLS and IAI **(c)** DDLS and CART **(d)** DDLS and CART (D=6)

Figure 5.8: Comparison of the average number of decision nodes between DDLS and each competing method across the 57 UCI datasets. Each point represents a dataset, with its position determined by the average model size (number of decision nodes) of DDLS and the compared method. Points are colored according to which method achieved higher average test accuracy on that dataset and are colored gray when the difference in test accuracy is smaller than 1%.

From the plots, we observe that in all cases, except for CART (D=6), when the comparison method produces larger models, DDLS yields more compact diagrams. However, it appears that as the size of the competing models increases, DDLS struggles to match their accuracy. One reason for this is the structural limitation in the current implementation of DDLS. For instance, with structure IV and a maximum depth of ten, our method can produce at most 20 distinct leaf nodes. This becomes a constraint in datasets with many classes, such as the *connectionist-vowel* dataset, which contains 11 classes. In such cases, DDLS may not be able to produce models with the width necessary to achieve high accuracy. Additionally, as the diagram grows, the optimization process may become more challenging due to the interconnected nature of the diagram structure, making it harder to find effective improvements.

Another observation is that, for datasets where IAI produces small models (fewer than ten decision nodes), DDLS tends to generate comparatively larger diagrams. Despite this, DDLS often achieves higher accuracy in these small-model scenarios, suggesting that it makes effective use of the additional decision nodes. A similar, but less noticeable pattern can be seen for the other methods.

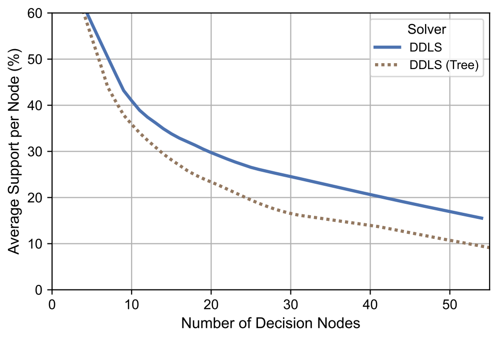
5.5. DDLS With Tree Structure

To better understand the specific advantages offered by decision diagrams, we compare the standard DDLS approach with a restricted variant that is constrained to operate exclusively on decision tree structures, called DDLS (Tree). In this restricted version, a tree skeleton is used as a template, and the move that redirects edges to arbitrary nodes in the subsequent layer is disabled. As a result, the model remains a proper decision tree, although it is still optimized using the same local search framework. This comparison allows us to assess whether the added flexibility of decision diagrams yields actual benefits, or whether comparable performance can be achieved using standard trees. If the restricted tree-based version performs similarly to the full DDLS, it may suggest that the search procedure itself is the main driver of performance.

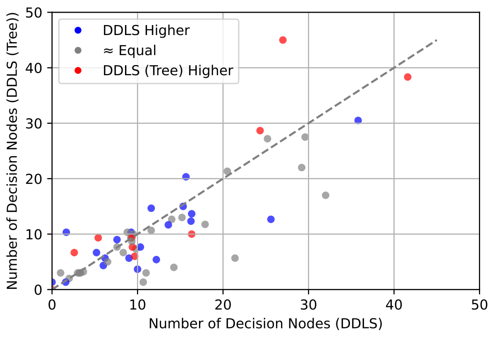
We follow the same experimental setup as previously described, using the 57 UCI datasets. Since this comparison is based on tree structures rather than decision diagrams, we no longer need to select from the structural templates, and this parameter is therefore excluded from hyperparameter tuning. The maximum depth is limited between one and six to match the maximum number of decision nodes for both methods. All other hyperparameters remain unchanged.

Method	Test accuracy (%)	Number of decision nodes	Average support per decision node (%)	Depth	Question length
DDLS (Tree)	82.9	**10.9**	44.6	**4.1**	**3.0**
DDLS	**83.7**	12.5	**46.3**	5.6	4.0

Table 5.4: Comparison of DDLS with its tree-based variant DDLS (Tree) in terms of average test accuracy, model size, fragmentation, depth, and question length. Averages are computed across the 57 UCI datasets. Bold values indicate the better result for each metric.



(a) Fragmentation for different sizes.



(b) DDLS (Tree) and DDLS

Figure 5.9: Comparison of fragmentation and model size between DDLS and DDLS (Tree). (a) Average support per decision node as a function of model size, showing fragmentation. (b) Comparing model sizes, with colors representing which method attains higher test accuracy, or equal when the difference in test accuracy is smaller than 1%

5.5.1. Comparing DDLS (Tree) and DDLS

We begin by examining how the test accuracy of our method changes when using decision trees instead of decision diagrams. The results are presented in Table 5.4. On average, the test accuracy of DDLS is slightly higher than that of DDLS (Tree). A Wilcoxon signed-rank test confirms that this difference is statistically significant ($p = 0.029$), although with small effect size ($\gamma = -0.120$). These results indicate that decision diagrams provide a modest but consistent performance advantage over decision trees, suggesting that the improvement stems not only from the effectiveness of our local search strategy, but also from the added expressiveness and flexibility offered by decision diagrams.

In terms of model size, measured by the number of decision nodes, we observe that tree-based models are significantly smaller than their diagram-based counterparts. As shown in Table 5.4, DDLS (Tree) achieves a lower average number of decision nodes. This difference is statistically significant according to the Wilcoxon signed-rank test ($p = 0.006$), though the effect size remains small ($\gamma = -0.114$). These findings are consistent with earlier results, such as those observed for IAI, which also produces more compact models than DDLS.

The depth and question length similarly align with the characteristics of tree-based models. Both the depth and question length are lower for DDLS (Tree), reflecting the shallower nature of decision trees.

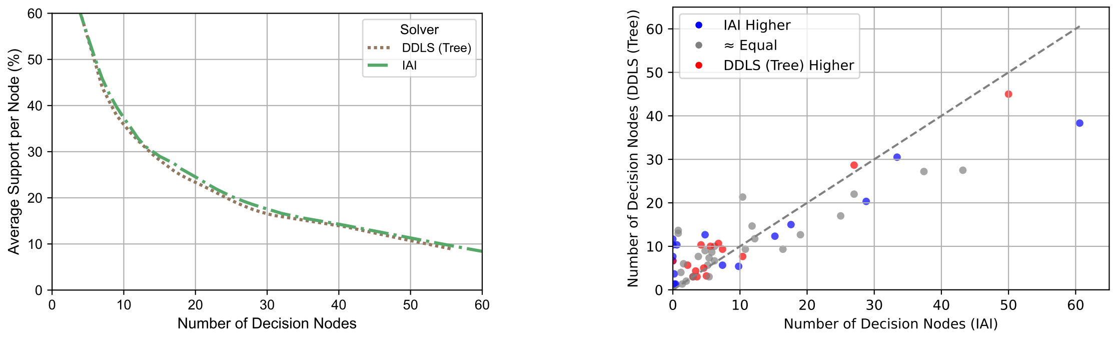
We further examine fragmentation and test accuracy in relation to model size in Figure 5.9. The results in (a) show that DDLS exhibits less fragmentation compared to DDLS (Tree), consistent with the trends observed in earlier comparisons with other decision tree methods. Unlike previous comparisons, however, DDLS does not appear to produce smaller models when DDLS (Tree) generates larger trees, as can be seen in (b). Instead, DDLS consistently results in larger models than DDLS (Tree). Despite this, the increased size seems to contribute positively to performance, as DDLS generally achieves higher test accuracy across most cases, suggesting that the additional complexity leads to more expressive and effective models.

5.5.2. Comparing DDLS (Tree) and IAI

Our algorithm was inspired by IAI and shares several key characteristics. Both methods employ a local search approach and include a similar move that finds the optimal threshold for a given decision node. Furthermore, out of the various decision tree models considered in this work, IAI has demonstrated the strongest performance across datasets. To assess how closely our DDLS (Tree) approach aligns with

Method	Test accuracy (%)	Number of decision nodes	Average support per decision node (%)	Depth	Question length
DDLS (Tree)	82.9	10.9	44.6	4.1	3.0
IAI	**83.3**	**10.2**	**56.2**	**2.8**	**2.5**

Table 5.5: Comparison of DDLS (Tree) and IAI in terms of average test accuracy, model size, fragmentation, depth, and question length. Averages are computed across the 57 UCI datasets. Bold values indicate the better result for each metric.



(a) Fragmentation for different sizes.

(b) DDLS (Tree) and IAI

Figure 5.10: Comparison of fragmentation and model size between DDLS (Tree) and IAI. (a) Average support per decision node as a function of model size, showing fragmentation. (b) Comparing model sizes, with colors representing which method attains higher test accuracy, or equal when the difference in test accuracy is smaller than 1%

IAI's performance, we conduct a direct comparison between the two methods.

Table 5.5 presents several key metrics for both methods. The results show that DDLS (Tree) and IAI achieve comparable test accuracy and a similar number of decision nodes, with IAI exhibiting a slight advantage in both. This is supported by the Wilcoxon signed-rank test, which finds no statistically significant differences between the two methods for either the test accuracy or the number of decision nodes.

However, when examining model depth and question length, IAI produces significantly shallower trees with shorter question lengths than DDLS (Tree). This difference may be due to IAI's specific hyperparameter tuning strategy and its use of cost-complexity pruning.

We further analyze fragmentation relative to model size in Figure 5.9. Plot (a) shows that fragmentation levels are very similar between the two methods as the number of decision nodes increases, consistent with other tree-based methods.

In plot (b), which compares model sizes and test accuracy, a pattern similar to the comparison between IAI and DDLS can be seen. When IAI produces smaller models (fewer than ten decision nodes), DDLS (Tree) tends to produce slightly larger models that achieve higher test accuracy. Conversely, as the number of nodes increases for IAI, DDLS (Tree) generates smaller models but often loses in terms of test accuracy.

5.6. Synthetic Data Experiments

In addition to the 57 UCI datasets, we evaluated our approach on synthetic datasets generated following the procedure described in Section 5.1. By default, these datasets contain five features, two classes, 1,000 samples, and no noise. We systematically vary one parameter at a time, namely the number of features, classes, samples, or the level of class and feature noise, and observe the resulting changes in accuracy, model size, and fragmentation. These experiments provide insight into our method's robustness and performance across different data characteristics. The results are summarized in Figure 5.11.

5.6.1. Effect of Data Characteristics on Accuracy

Since the synthetic datasets are generated using decision diagrams templates as ground truth, we expected our DDLS method, capable of representing such diagrams exactly, to perform well. Surprisingly, DDLS often underperforms compared to other methods, including tree-based approaches like CART and IAI, and even compared to our own DDLS (Tree) variant using tree structures instead of diagrams. In particular, IAI and TnT consistently achieve the highest test accuracy across most synthetic configurations. This result is counterintuitive, given that DDLS is theoretically capable of exactly recovering the ground truth model used to generate the data.

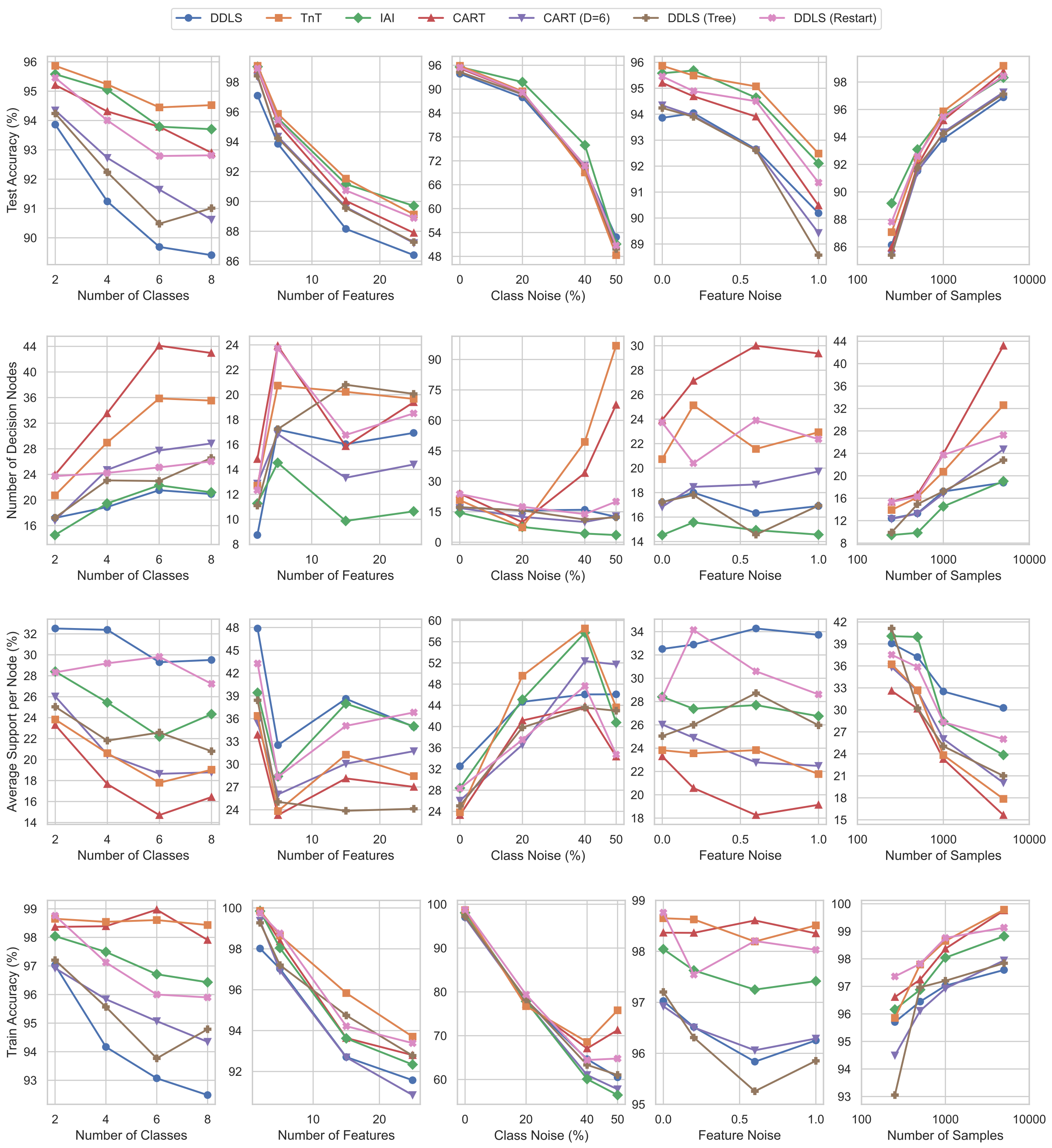


Figure 5.11: Performance of different methods on synthetic datasets across three metrics: test accuracy (top row), number of decision nodes (second row), fragmentation measured by average support per node (third row), and training accuracy (bottom row). Each column varies one of five dataset characteristics: number of features, number of classes, number of samples, feature noise, and class noise, while keeping the others fixed. This setup isolates the impact of each factor on model behavior. Confidence intervals are omitted for clarity and visibility.

Some decline in performance was expected for DDLS in settings with a large number of classes, since DDLS is constrained in the number of leaf nodes it can produce. As the number of classes increases, the model struggles to separate all outputs within its limited width. However, we can also see that when increasing the number of features or the number of samples, DDLS consistently performs the worst out of all the methods. The only exceptions are when we have increased class or feature noise, for which DDLS seems to perform relatively better as the noise increases.

Furthermore, the training accuracy reveals a similar pattern, both DDLS and DDLS (Tree) consistently exhibit the lowest performance, with DDLS (Tree) performing slightly better. This suggests that both methods are likely underfitting to the training data. A plausible explanation is that the datasets are too complex for these models to learn consistently. Many of the original diagrams used to generate the datasets are large and contain complex dependencies, which may be difficult to recover by DDLS and DDLS (Tree). We have already observed that performance tends to decline as the size of diagram increases. Additionally, because trees have independent branches and are generally easier to learn than decision diagrams, this may explain the slight performance advantage of DDLS (Tree) over DDLS on both training and test accuracy.

5.6.2. Effect of Data Characteristics on Size and Fragmentation

When examining model size, we again observe that IAI consistently produces the most compact models. This is followed by DDLS, DDLS (Tree), and CART (D=6), all of which yield models of comparable size. In contrast, CART and TnT tend to produce significantly larger models.

In terms of fragmentation, DDLS consistently achieves a high average support per decision node, indicating a low degree of fragmentation. This aligns with our earlier findings that DDLS produced models with low fragmentation. However, in this synthetic setting, this reduced fragmentation does not appear to translate into higher test accuracy.

5.6.3. Improving DDLS with Random Restarts

As previously noted, the training accuracy of DDLS is also worse than all methods in most cases, indicating that it is likely underfitting to the training data. Two possible explanations for this behavior are: 1) the regularization parameter α may be too large, discouraging model complexity; or 2) due to the complexity of the dataset, the algorithm may struggle to consistently find high-quality solutions due to the randomness in its search process.

To address these issues, we introduce two modifications. First, we expand the range of the regularization parameter α to $[10^{-12}, 0.1]$, allowing for weaker regularization and greater flexibility in model complexity. Second, we incorporate random restarts to improve solution quality. Specifically, after tuning hyperparameters using the same procedure as before (now with the extended α range), we initialize DDLS (Restart) once using information gain and perform 50 independent local search runs. In each run, we set the maximum number of idle iterations at 1,000, resulting in 50 candidate solutions. We then select the decision diagram with the lowest cost among them. This approach reduces the chance of selecting a diagram where the algorithm converged to a poor local optimum and increases the likelihood of discovering a high-quality solution.

The results, shown in Figure 5.11, demonstrate that DDLS (Restart) substantially outperforms the standard version of DDLS in both training and test accuracy. In many cases, it approaches or even matches the performance of IAI, though this comes at the cost of slightly larger models and increased fragmentation. These findings suggest that the complexity of the synthetic datasets makes it difficult for a single run of local search to consistently find strong solutions. By performing multiple independent runs and selecting the best result, random restarts reduce the likelihood of settling on a solution that became trapped in a poor local optimum early in the search. In contrast, such restarts might have been less important on real-world datasets, due to the lower complexity. For synthetic diagrams with intricate dependencies, however, random restarts clearly provide a substantial benefit.

5.7. Scalability

Since one of our primary goals is to develop an algorithm that scales effectively to larger datasets, we also evaluate the runtime performance of each method. We exclude the MILP-based approach

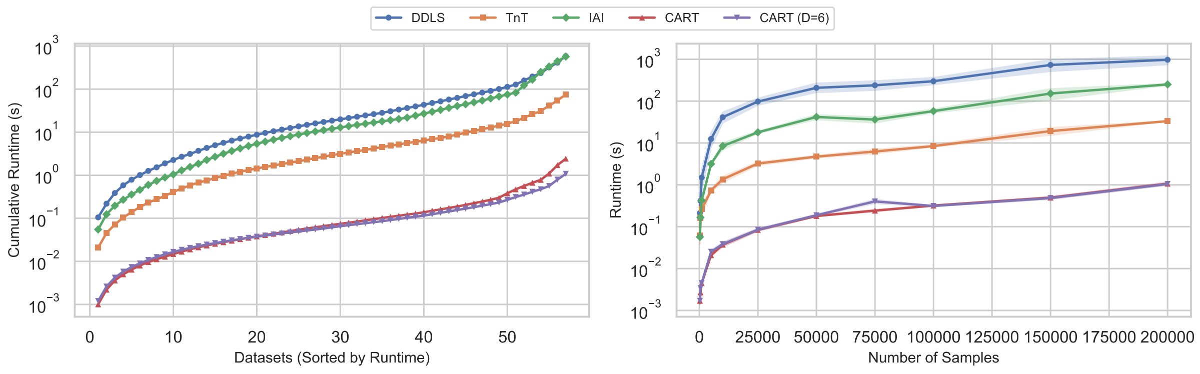


Figure 5.12: Runtime comparison of our method against baseline approaches on (a) real-world UCI datasets using cumulative runtime and (b) synthetic datasets with varying sample sizes. Error bars represent 95% confidence intervals

from this comparison, as it struggles to solve problems with more than 200 samples within a 20-minute timeout, making it unsuitable for larger-scale experiments.

We assessed runtime in two ways: by cumulating the average runtime of each dataset across 57 UCI datasets and by observing runtime behavior on synthetic datasets of varying sizes. The results are presented in Figure 5.12.

In Figure 5.12 (a), which shows results on the UCI datasets, both versions of CART are noticeably the fastest. DDLS and IAI exhibit the slowest runtimes, while TnT's runtime falls between these two groups. As expected, methods based on local search tend to be slower due to their more complex optimization procedures.

Figure 5.12 (b) displays average runtimes on synthetic datasets as the number of samples increases. For this experiment, we use the largest available structure for each method and disable regularization. While DDLS is the slowest among the compared methods, it still scales effectively, maintaining tractable runtimes even on datasets with up to 200,000 samples. This demonstrates that despite its higher computational cost, DDLS remains far more scalable than exact methods and is viable for large-scale applications.

6 Conclusion

In this thesis, we introduced Decision Diagram Local Search (DDLS), a novel local search algorithm for learning decision diagrams. DDLS aims to bridge the gap between fast, greedy methods for constructing decision diagrams, which often yield suboptimal solutions, and exact optimization techniques that are computationally expensive and scale poorly. Our approach uses a fixed diagram structure and a local search to efficiently explore the space of interpretable models. Through extensive experimentation, we demonstrate that DDLS is able to find near-optimal solutions for small datasets, while significantly improving the runtime compared to exact methods. When benchmarked against state-of-the-art decision diagram and decision tree models on 57 real-world UCI datasets, DDLS outperforms all methods in terms of accuracy while maintaining a compact model size. The results showed that while our diagrams are typically deeper, they suffer less from fragmentation, even as their size increases.

A limitation of DDLS becomes apparent in experiments on synthetic datasets generated from decision diagrams, where it underperforms relative to other methods. We observe that DDLS tends to underfit the training data, likely due to getting stuck in local optima caused by the complexity of these datasets. Preliminary experiments incorporating random restarts, where we run DDLS multiple times and select the candidate with the lowest cost, have already demonstrated significant improvements in both training and test accuracy. Future research could further investigate this strategy, improving it and exploring its potential in combination with other metaheuristics, such as simulated annealing. Additionally, subsequent work might consider moving beyond fixed structural templates by initializing DDLS with a decision tree structure, from which every possible diagram can be created.

These initial results are promising and suggest that DDLS, and decision diagrams in general, have considerable untapped potential. With continued refinement, it may become an even more powerful and flexible tool for learning interpretable models, capable of performing well on both real-world and complex synthetic settings.

References

[1] Milton Friedman and. “The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance”. In: *Journal of the American Statistical Association* (1937), pp. 675–701.

[2] William S. Cleveland and. “Robust Locally Weighted Regression and Smoothing Scatterplots”. In: *Journal of the American Statistical Association* (1979), pp. 829–836.

[3] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Chapman and Hall/CRC., 1984.

[4] Gianpiero Cabodi, Paolo E. Camurati, Alexey Ignatiev, Joao Marques-Silva, Marco Palena, and Paolo Pasini. “Optimizing Binary Decision Diagrams for Interpretable Machine Learning Classification”. In: *2021 Design, Automation and Test in Europe Conference and Exhibition*. 2021, pp. 1122–1125.

[5] Migueí Carreira-Perpiñán and Pooya Tavallali. “Alternating Optimization of Decision Trees, with Application to Learning Sparse Oblique Trees”. In: 2018, pp. 1217–1227.

[6] Sara Ceschia, Francesca Da Ros, Luca Di Gaspero, and Andrea Schaerf. “EasyLocal++ a 25-year Perspective on Local Search Frameworks: The Evolution of a Tool for the Design of Local Search Algorithm”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2024, pp. 1658–1667.

[7] Janez Demsar. “Statistical Comparisons of Classifiers over Multiple Data Sets”. In: *Journal of Machine Learning Research* (2006), pp. 1–30.

[8] Li Deng. “The MNIST database of handwritten digit images for machine learning research”. In: *IEEE Signal Processing Magazine* (2012), pp. 141–142.

[9] Luca Di Gaspero and Andrea Schaerf. “EasyLocal++: an object-oriented framework for the flexible design of local search algorithms and metaheuristics.” In: 2001, pp. 733–765.

[10] Jack Dunn. “Optimal trees for prediction and prescription”. PhD thesis. Massachusetts Institute of Technology, 2018.

[11] Alexandre M. Florio, Pedro Martins, Maximilian Schiffer, Thiago Serra, and Thibaut Vidal. “Optimal Decision Diagrams for Classification”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* (2023), pp. 7577–7585.

[12] Qiong Gao, Ming Li, and Paul Vitányi. “Applying MDL to Learning Best Model Granularity”. In: *Artificial Intelligence* (2000), pp. 1–29.

[13] Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. “Revisiting deep learning models for tabular data”. In: *Proceedings of the 35th International Conference on Neural Information Processing Systems*. NIPS ’21. 2021, pp. 18932–18943.

[14] Hao Hu, Marie-José Huguet, and Mohamed Siala. *Optimizing Binary Decision Diagrams with MaxSAT for classification*. 2022.

[15] Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. *The UCI Machine Learning Repository*. `https://archive.ics.uci.edu`. 2021.

[16] Ron Kohavi. “Bottom-up induction of oblivious read-once decision graphs”. In: *Machine Learning: ECML-94*. 1994, pp. 154–169.

[17] Min Lin, Qiang Chen, and Shuicheng Yan. *Network In Network*. 2014. arXiv: `1312.4400 [cs.NE]`.

[18] Jacobus G. M. van der Linden, Daniël Vos, Mathijs M. de Weerdt, Sicco Verwer, and Emir Demirović. *Optimal or Greedy Decision Trees? Revisiting their Objectives, Tuning, and Performance*. 2025. eprint: `2409.12788`.

[19] Shingo Mabu, Masanao Obayashi, and Takashi Kuremoto. “An Evolutionary Algorithm for Making Decision Graphs for Classification Problems”. In: *Journal of Robotics, Networking and Artificial Life* (2016), pp. 45–49.

[20] J. J Mahoney and Raymond J Mooney. *Initializing ID5R with a Domain Theory: Some Negative Results*. Tech. rep. USA, 1991.

[21] P. B. Nemenyi. “Distribution-Free Multiple Comparisons”. PhD thesis. Princeton University, 1963.

[22] Arlindo L. Oliveira and Alberto Sangiovanni-Vincentelli. “Using the minimum description length principle to infer reduced ordered decision graphs”. In: *Machine Learning* (1996), pp. 23–50.

[23] Jonathan J. Oliver. “Decision Graphs - An Extension of Decision Trees”. In: *Proceedings of the 4th international workshop on artificial intelligence and statistics (AISTATS)*. 1993, pp. 343–350.

[24] Rok Piltaver, Mitja Luštrek, Matjaž Gams, and Sanda Martinčić-Ipšić. “What makes classification trees comprehensible?” In: *Expert Systems with Applications* (2016), pp. 333–346.

[25] John Platt, Nello Cristianini, and John Shawe-Taylor. “Large Margin DAGs for Multiclass Classification”. In: *Advances in Neural Information Processing Systems* (2000), pp. 547–553.

[26] J. R. Quinlan. “Induction of Decision Trees”. In: *Machine Learning* (1986), pp. 81–106.

[27] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993. ISBN: 1558602402.

[28] J. Rissanen. “Modeling by shortest data description”. In: *Automatica* (1978), pp. 465–471.

[29] Pouya Shati, Eldan Cohen, and Sheila McIlraith. “SAT-Based Learning of Compact Binary Decision Diagrams for Classification”. In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. 2023, 33:1–33:19.

[30] Ravid Shwartz-Ziv and Amitai Armon. “Tabular data: Deep learning is not all you need”. In: *Information Fusion* 81 (2022), pp. 84–90.

[31] Ricardo Vilalta, Gunnar Blix, and Larry Rendell. “Global data analysis and the fragmentation problem in decision tree induction”. In: *Machine Learning: ECML-97*. 1997, pp. 312–326.

[32] Frank Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Biometrics Bulletin* (1945), pp. 80–83.

[33] Bingzhao Zhu and Mahsa Shoaran. *Tree in Tree: from Decision Trees to Decision Graphs*. 2021. arXiv: `2110.00392 [cs.LG]`.

A

Detailed Accuracy Results

Dataset	DDLS	DDLS (Tree)	TnT	CART	CART (D=6)	IAI
acute-inflammations-nephritis	100.0	100.0	99.2	100.0	100.0	100.0
acute-inflammations-urinary	100.0	100.0	100.0	100.0	100.0	100.0
balance-scale	79.9	79.0	78.4	77.0	75.0	79.2
bank-marketing	90.0	90.1	90.3	90.2	89.9	90.2
banknote-authentication	98.7	97.5	98.9	98.7	98.5	98.3
blood-transfusion-service	77.5	78.1	79.0	75.9	75.9	78.2
breast-cancer-diagnostic	93.8	94.4	91.9	93.0	93.0	94.4
breast-cancer-prognostic	73.5	67.3	72.7	73.2	73.2	75.3
breast-cancer-wisconsin	96.2	93.9	94.0	93.4	93.7	95.0
car-evaluation	85.8	84.1	97.5	93.5	84.7	85.9
chess-kr-vs-kp	94.3	96.2	99.6	99.4	93.6	96.7
climate-simulation-crashes	90.6	88.9	90.4	90.4	90.7	91.5
congressional-voting	96.0	96.3	95.2	94.0	94.0	94.7
connectionist-mines-vs-rocks	80.0	72.7	70.2	71.6	71.1	77.4
connectionist-vowel	50.6	65.5	76.2	76.4	51.4	58.5
contraceptive-method-choice	54.9	54.7	53.0	56.2	56.4	55.6
credit-approval	86.8	87.2	84.8	85.0	84.4	85.1
cylinder-bands	74.6	65.7	67.5	66.4	63.5	69.7
dermatology	96.2	96.0	95.6	95.4	94.8	95.1
dry-bean-dataset	90.7	89.7	91.3	91.1	89.6	90.8
echocardiogram	97.5	95.8	96.8	96.8	96.8	96.8
fertility-diagnosis	84.8	78.7	83.0	81.0	80.0	88.0
habermans-survival	73.5	69.3	71.6	70.3	71.3	72.9
hayes-roth	79.0	80.8	81.2	83.8	84.4	76.9
heart-disease-cleveland	57.9	52.0	49.2	51.9	49.8	55.9
hepatitis	89.0	83.3	78.8	77.5	77.5	81.2
image-segmentation	95.6	94.7	95.9	94.9	90.9	95.4
indian-liver-patient	69.2	69.7	70.5	70.6	70.0	71.8
ionosphere	92.0	92.8	89.7	88.0	88.9	90.6
iris	96.8	95.6	94.7	95.3	95.3	94.0
magic-gamma-telescope	84.0	84.1	85.4	84.9	83.7	84.8
mammographic-mass	83.1	83.8	83.0	81.2	81.2	82.7
monks1	50.8	52.0	51.1	51.1	44.8	51.1
monks2	54.3	54.7	63.2	54.4	54.7	53.9
monks3	48.7	42.7	48.6	50.4	45.3	48.7
optical-recognition	81.9	85.1	90.8	90.1	77.1	86.3
ozone-eighthr	94.3	92.7	92.9	92.7	92.7	93.1
ozone-onehr	96.2	95.5	96.5	96.7	96.7	96.4
parkinsons	93.5	89.8	85.1	86.2	86.2	89.7
pima-indians-diabetes	76.9	76.2	73.9	73.4	72.4	75.9
planning-relax	71.3	72.5	71.4	67.6	66.0	71.4
qsar-biodegradation	83.0	82.2	82.9	82.0	81.7	82.9
seeds	91.2	96.9	92.4	91.9	91.9	91.0
seismic-bumps	93.3	93.4	93.4	93.0	93.0	93.3
soybean-small	100.0	100.0	97.8	95.6	95.6	97.8
spambase	92.3	91.7	91.9	91.6	90.8	91.7
spect-heart	68.7	74.6	73.0	70.4	70.4	73.8
spectf-heart	80.6	77.6	79.0	78.3	79.4	79.4
statlog-german-credit	71.7	75.5	71.6	72.6	72.3	71.5
statlog-landsat-satellite	85.5	84.5	87.5	86.2	83.9	85.5
teaching-assistant-evaluation	51.8	45.6	58.2	58.9	41.1	47.6
thoracic-surgery	86.2	81.9	84.3	83.2	82.3	85.1
thyroid-ann	99.9	99.8	99.8	99.7	99.7	99.7
thyroid-new	95.1	95.1	94.9	92.6	92.6	94.4
tic-tac-toe	91.4	93.9	94.1	95.0	93.1	92.6
wall-following-robot-2	100.0	100.0	100.0	100.0	100.0	100.0
wine	95.0	95.6	87.7	89.3	89.3	94.4

Table A.1: Complete overview of the test accuracy (%) for different methods on all 57 UCI datasets.