UCLouvain

École polytechnique de Louvain

Evaluating decision trees in a Predict-and-Optimize Setting

Authors : **Sixto Castro**
Supervisors : **Siegfried Nijssen**
Readers : **Marco Saerens**, **Harold Kiossou**
Academic year 2023–2024
Master [120] in Computer Science

Acknowledgments

Coming from Ecuador to Belgium has been a big challenge in my life, where I have learned many things that have helped me grow professionally and personally. I am happy for this new chapter in my life with enriching experiences and good friends.

I dedicate this project first to God for allowing me to reach this special moment in my life, and for giving me wisdom and strength daily to successfully complete this stage of studies.

To my parents for providing me with unconditional support in both good and bad times, and for guiding me throughout this journey until reaching the goal.

A big thanks to Professor Siegfried Nijssen for his continuous great help and guidance during the elaboration of my thesis and master's courses.

Abstract

This thesis studies the performance of decision trees in a Predict-and-Optimize setting, where a decision tree predicts the unknown input parameters for an optimization problem, and then decisions are made using these parameters to address the optimization problem. For this study, two loss functions are used for training decision trees: MSE loss and the Smart-Predict-then-Optimize (SPO) loss, and each loss function is considered in two algorithms: greedy decision trees and optimal decision trees (DL8.5). Therefore, the models under study are: MSE trees and SPO trees (SPOTs), along with DL8.5 algorithms—DL8.5 with MSE and DL8.5 with SPO—aimed at building optimal decision trees. Then, these models are employed to address two optimization problems: the Shortest path problem and the Traveling salesman problem (TSP). In the context of the TSP problem, the greedy models and the DL8.5 decision trees are assessed with different TSP algorithms (exact and heuristic models) to tackle this NP-hard optimization problem of finding the shortest tour. We conduct several experiments on synthetic workloads with different degree and noise parameters, involving the prediction of the travel times for these optimization problems. Based on the results, we underscore the significance of employing SPO metric for building decision trees because it provides better quality decisions and lower model complexity than trees designed to minimize the prediction error (MSE). Furthermore, employing DL8.5 with SPO consistently outperforms the other methods when analyzing decision error.

Contents

Chapter 1

Introduction

Real-world decision making often reflects optimization problems in which the primary goal is to attain the most favorable outcome among all feasible solutions [3]. This involves maximizing benefits, minimizing costs, or achieving an optimal balance across multiple factors. These scenarios encompass allocating resources, optimizing strategies, and making choices within constraints [21]. For instance, this might involve a company optimizing its production processes to reduce costs while meeting demand, or an individual choosing the best commute route considering time, distance, and traffic conditions.

Optimization problems often feature uncertainty and interconnected variables, demanding robust or adaptable solutions. Estimation of these variables is commonly executed using historical data, observations, or expert insights. Once estimated, optimization techniques can be employed to find the optimal solution under the given uncertainty [3]. Incorporating data-driven estimations of uncertain parameters into optimization problems is a practical and effective approach across many fields, such as finance, operations research, engineering, and supply chain management, among others.

One of the most common examples of optimization problems is the Delivery Routing Problem**.** A particular case is the Traveling salesman problem (TSP), where a delivery vehicle aims to minimize travel time for deliveries by finding the shortest route visiting cities once and returning to the start. It considers uncertain factors impacting travel times like traffic, road, gas, and weather conditions. Historical delivery data helps estimate these uncertainties. A decision tree predicts expected costs for routes based on this data. Once the predicted costs for all paths are available, they can now be used as input for the optimization TSP problem, allowing a TSP algorithm to find an efficient route despite potential travel time variations due to uncertain conditions.

A well-known approach for addressing problems involving uncertain input parameters is the predict-then-optimize framework. It consists of two steps: (i) predicting the uncertain parameters using a machine learning model trained on historical data, and (ii) generating decisions by solving the corresponding optimization problem using the predicted parameters. Typically, machine learning models in this framework are trained using loss functions that measure prediction error, without considering the impact of the predictions on the downstream optimization problem. However, in many cases, we are more interested in obtaining near-optimal decisions based on the estimated input parameters rather than minimizing prediction error. This

work introduces a methodology for training decision trees within the predict-then-optimize framework by Elmachtoub, Liang and McNellis [3], with a focus on minimizing decision error instead of prediction error.

This thesis will focus on the evaluation of decision trees using a predict-then-optimize framework in order to tackle two different optimization problems: the Shortest path problem and the Traveling salesman problem (TSP), mainly performed on a 4x4 grid. The datasets generated are synthetic with different configuration parameters, therefore the analysis is more in a theatrical way. The greedy algorithms we study are fitted under SPO and MSE functions (SPO and MSE trees) for solving the shortest path problem. While to address the Traveling salesman problem, these decision trees are adapted to different TSP algorithms, such as: exact (dynamic programming) and heuristic models (local search, simulated annealing, and routing). Finally, the DL8.5 algorithm was also included in both optimization problems to assess the performance of optimal decision trees under the mentioned loss functions, and since DL8.5 can be combined with TSP algorithms, we also evaluate the DL8.5 decision tree with the SPO loss function to address the TSP problem.

The document is organized as follows: Section II provides an overview of decision trees, CART algorithm, and DL8.5 trees. Section III details the explanation of the Predict-then-Optimize framework alongside the adaptation of DL8.5 to incorporate the loss functions under study. Following this, Section IV introduces the Shortest path problem, while Section V delves into the Traveling salesman problem. The subsequent section discusses various experiments conducted for both optimization problems. Finally, Section VII presents the concluding remarks.

Chapter 2

Decision trees

In this chapter, we cover the basic concepts of decision trees, as well as an illustrative example. Then, the CART algorithm is explained spanning the classification and regression approaches. Lastly, the DL8.5 decision tree is introduced as it will be the object under study in the next sections against greedy decision trees.

2.1 Background

In recent times, there has been an increasing attention to decision trees due to their interpretability, as they can be perceived as rules that are understandable by domain experts [1] in decision-making across various fields like operations, finance, and project management [4]. Their popularity in machine learning is driven by their simple interpretative nature and efficient algorithms capable of learning trees of good quality [2]. Moreover, decision trees find extensive application in both classification and regression tasks within supervised learning, highlighting their adaptability and utility across diverse problem types. For that reason, this thesis will focus on evaluation of decision trees.

A decision tree is a decision-making tool that employs a tree-like structure to represent decisions and their potential outcomes. It is represented as a hierarchical tree structure consisting of internal nodes, branches, and leaf nodes. Each internal node represents a decision point or a test on a specific attribute, branches illustrate the outcomes of these decisions or feature tests, and each terminal node, known as leaf node, represents a class label or a predicted value [5].

The generation of a decision tree is done through a recursive process of dividing the training data into subsets, guided by attribute values. This division continues until a stopping condition is fulfilled, which could be determined by factors like reaching the maximum depth of the tree or requiring a minimum number of samples to split a node. Lastly, predictions are made by traversing the tree from its root to a leaf node, based on the values of the input variables.

Decision trees are known for their interpretability, capability to handle diverse features and missing data, and robustness against outliers. Nevertheless, they have a propensity to overfit data. To mitigate this, strategies such as pruning and ensemble methods are employed to enhance

generalization. Additionally, as the tree grows deeper, its complexity increases, rendering interpretation more challenging [6].

2.2 Decision tree representation

A common illustration of a decision tree is demonstrated in Figure 2.1 representing the Titanic case study.

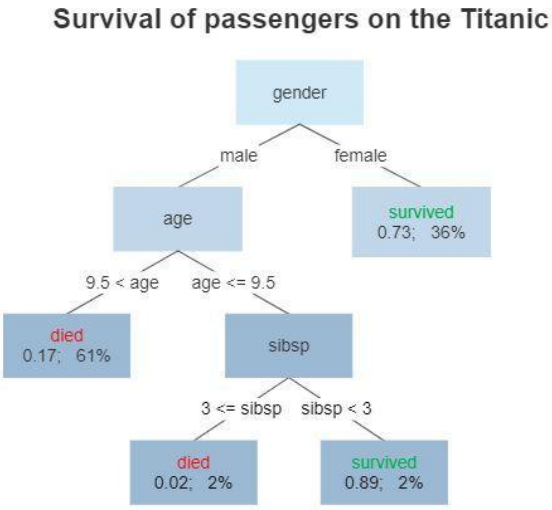


Figure 2.1: Decision tree example from Wikipedia [7].

In the context of the Titanic case, a decision tree is an algorithm used to predict the survival outcome of passengers based on various features or attributes. The goal is to build a model that can classify passengers into two categories: "Survived" or "Not Survived."

The decision tree algorithm starts with the entire dataset of Titanic passengers and recursively splits it into subsets based on different features. At each node (split), the algorithm asks a question related to a specific feature, and the data is partitioned into two branches based on the possible answers to the question. For example, one of the first splits we can see in Figure 2.1 is based on the gender of the passengers: "Is the passenger male or female?" The data is then divided into two groups: one for male passengers and another for female passengers.

Then, it continues splitting the data further based on different features, such as age, ticket class, number of siblings/spouses aboard, number of parents/children aboard, etc. Each split is determined based on a criterion that maximizes the separation of survival outcomes, such as Gini impurity or information gain for classification tasks.

Eventually, the algorithm creates a tree structure where the leaf nodes represent the final predictions: "Survived" or "Not Survived". In the end, we realize that the leaf nodes corresponding to the chances of survival can be either: the passenger was a female, or a male younger than 9.5 years with at most 2 siblings.

In summary, the decision tree algorithm for the Titanic case builds a tree structure based on passenger features to predict the survival outcome, making it a valuable tool for understanding patterns and factors that influenced survival during the tragic event.

When considering various types of decision trees, notable examples include C4.5 and CART (Classification and Regression Trees). Here, we'll delve into the CART algorithm. Supervised learning algorithms effectively tackle classification and regression problems, with decision trees being a key tool for addressing these tasks. Let's explore the concepts of classification and regression trees further.

2.2.1 Classification trees

Classification trees are used for categorical outcome variables within specific classes or categories, such as fraud detection, sentiment analysis, and customer segmentation [6]. Their main objective is to partition the feature space into regions aligning with different class labels, essentially creating a model capable of predicting data instance class labels (*C*) based on their feature values (*F*).

Given:

- *X* is the input data instance with feature values $X = \{x_1, x_2, ..., x_n\}$.
- *F* is the set of features, $F = \{f_1, f_2,, ..., f_n\}$.
- *C* is the set of class labels, $C = \{C_1, C_2,, ..., C_k\}$.

The classification process [8] can be mathematically represented using the argmax function as follows:

$$C = argmax\, P(C_i \mid X)\ for\ all\ C_i\ in\ C \tag{2.1}$$

where C is the predicted class label for the input data instance *X*. P(C_i | *X*) is the conditional probability of the data instance *X* belonging to class C_i given its feature values. It represents the confidence or probability that the instance falls into class C_i based on the learned patterns from the training data and the decision tree model *T*.

Finally, the argmax function selects the class label with the highest conditional probability as the predicted class label for the input instance, indicating the class that the decision tree model *T* believes is the most likely class for the given data instance *X*.

2.2.2 Regression trees

Regression trees are employed when the target variable is continuous, representing numerical values. Their objective is to segment the feature space into regions and predict a continuous value for each of these regions. Similar to classification trees, regression trees also operate by posing a sequence of binary questions regarding the features.

Constructing a regression tree consists of splitting the predictor space into J distinct regions. Within each region, the predicted response value for an observation is the mean of the response values present in that specific segment. The splitting of regions aims to minimize the Residual Sum of Squares (RSS) using a top-down greedy approach known as recursive binary splitting, meaning that all observations initially belong to a single region, then the algorithm proceeds to recursively split the regions, dividing them further based on the predictor variables.

The approach is considered "greedy" because, at each step of the process, the algorithm selects the best split available without considering future splits. In other words, it chooses the locally optimal split at each node, aiming for the lowest RSS at that specific step, without looking ahead to check if a different split in the future would result in a better overall prediction.

The regions given by the split (j,s) are represented as follows:

$$R_1(j,s) = \{X \mid X_j < s\} \; and \; R_2(j,s) = \{X \mid X_j \geq s\} \tag{2.2}$$

And the goal is to find the best split (j,s) that minimizes the RSS error, calculated as the sum of squared errors for both resulting regions. This is represented by the formula 2.3:

$$min_{j,s} \sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2 \tag{2.3}$$

Here, $R_1(j,s)$ and $R_2(j,s)$ denote the resulting regions after the split, while $\hat{y}_{R_1}$ and $\hat{y}_{R_2}$ represent the predicted response values for these regions, typically calculated as the mean of the observed response values within each region. The goal is to minimize the total sum of squared differences between the predicted and true values in both resulting regions after the split [9].

2.3 CART algorithm

One of the most popular algorithms to build decision trees is the CART (Classification and Regression Trees), which is widely used in machine learning and data mining. CART is a recursive partitioning algorithm that constructs binary decision trees by recursively splitting the dataset based on the values of input features.

The CART algorithm starts with the entire dataset and selects the best feature to split the data based on certain criteria, such as Gini impurity. The dataset is then divided into two subsets based on the chosen feature's values. This splitting process is repeated recursively on each subset until a stopping criterion is met, such as reaching a maximum tree depth, a minimum number of observations in the leaf nodes, or other criteria. Once the decision tree is constructed, it can be used to make predictions on new, unseen data by traversing the tree based on the values of the input features until reaching a leaf node, which provides the predicted class or value.

In the case of classification problems, CART creates decision trees that classify instances into different classes or categories. For regression problems, CART constructs decision trees that predict a continuous target variable. Overall, the CART algorithm is known for its simplicity, interpretability, and ability to handle both classification and regression tasks.

2.3.1 Greedy Splitting

- **Classification**

In the context of decision trees for Classification, CART effectively determines accurate and efficient classifications by posing appropriate questions at each node. Its goal is to recursively split the training data into smaller subsets in a way that minimizes the impurity within each subset.

For classification tasks, CART uses the Gini Impurity measure (Gini index), which indicates the probability to misclassify an observation within a subset that's labeled according to the majority class, meaning that a subset is more pure when the Gini impurity is lower.

The Gini index is calculated as follows:

$$Gini = 1 - (p_1^2 + p_2^2 + p_3^2 + \ldots + p_k^2) \quad (2.4)$$

The Gini index function is used to assess the "purity" of the leaf nodes, indicating how mixed the training data assigned to each node is. In the formula, this index considers all classes, denoted by

p_k representing the portion of training instances labeled as class "k" in a specific node. A node with complete class purity (all instances belonging to the same class) achieves Gini = 0, while a node with the highest mixing of classes achieves a higher Gini index value [10].

For binary classification problems, the Gini index can be re-written as follows:

$$Gini = 1 - (p_1^2 + p_2^2) \tag{2.5}$$

- **Regression**

However, when dealing with continuous variables for prediction, the conventional approach becomes unsuitable. Instead, we require a different measure that quantifies the deviation of our predictions from the actual target values. That measure is known as the mean square error (MSE) and is represented as follows.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 \tag{2.6}$$

For the calculation of the MSE metric, we have actual target values (Y) and predicted values ($\hat{Y}$), and our main concern is the variability of the predictions from the targets, irrespective of the direction [10]. To quantify this variability, we calculate the squared differences between the predicted and actual values and then take the average of these squared differences over all records.

Regarding the predicted value in a terminal node, it can be determined as the average of the target value of the observations present at that leaf node as shown below.

$$\bar{y}_m = \frac{1}{N_m} \sum_{\left(x^{(i)}, y^{(i)}\right) \in S_m} y^{(i)} \tag{2.7}$$

Similar to Classification trees, in Regression trees, we aim to partition the data into subsets at each node. However, the goal is to minimize the Mean Square Error rather than Gini or entropy. The splitting process involves selecting the feature and the corresponding threshold that minimizes the MSE for the child nodes, thus ensuring that the predictions within each subset are more accurate and closer to the actual target values.

2.3.2 Stopping criteria

The recursive binary splitting process described above requires a stopping criterion to determine when to halt the splitting procedure as it progresses down the tree using the training data.

The most used stopping criterion involves setting a minimum threshold on the number of training instances assigned to each leaf node. If the count of instances falls below this minimum threshold, the split is not accepted, and the current node is considered a final leaf node.

The count of training instances is adjusted based on the specific dataset, typically set to values like 5 or 10. This parameter influences how closely the tree fits the training data. If the count is too small (e.g., set to 1), the tree may overfit the training data, resulting in poor performance on the test set. Hence, the choice of this count helps strike a balance between generalization and fitting the training data closely.

Commonly used stopping criteria [11] in decision tree algorithms include:

1. **Minimum Leaf Size:** Stops tree construction when the number of instances in a leaf node falls below a specified threshold, preventing overfitting.
2. **Maximum Tree Depth:** Limits tree growth to maintain interpretability and prevent capturing noise or outliers.
3. **Minimum Improvement in Impurity:** Halts tree construction when the impurity improvement from a split falls below a predefined threshold.
4. **Minimum Number of Samples for Splitting:** Avoids splitting nodes with insufficient samples, ensuring meaningful splits.
5. **Maximum Number of Leaf Nodes:** Controls tree size and complexity by stopping tree building when a limit on leaf nodes is reached.

2.3.3 Pruning the tree

Pruning the tree is a technique used to reduce the complexity and size of a decision tree by removing branches or nodes that may lead to overfitting. Overfitting occurs when a decision tree captures noise or random fluctuations in the training data, resulting in poor generalization to new, unseen data. Pruning helps to create a simpler and more generalized tree that performs better on test data.

Different approaches exist for pruning decision trees, all focusing on removing branches or nodes that contribute less to the overall predictive power of the tree. The goal is to strike a balance between capturing useful patterns and avoiding overfitting.

Two common pruning methods [12] are:

1. **Post-Pruning (Backward Pruning):** Initially growing the tree to maximum depth using training data, then iteratively removing nodes while evaluating the pruned tree's performance on validation or test data. Accepting pruning if performance remains similar or improves.
2. **Cost Complexity Pruning (Weakest Link Pruning):** Introducing a complexity parameter (alpha) to balance tree size and accuracy. By varying alpha and evaluating different pruned trees, the optimal alpha value is determined using techniques like cross-validation.

Pruning helps in preventing overfitting, enhancing the tree's generalization to new data, and improving predictions' robustness. It's a crucial step in decision tree learning, ensuring the creation of a well-performing and interpretable model.

2.4 DL8.5 decision tree

The fact that conventional algorithms like CART use a greedy approach (heuristics), they tend to reach sub-optimal solutions when the depth of the tree is limited. CART faces a trade-off between ensuring adequate accuracy and maintaining a reasonable depth. To address this, DL8.5 decision tree is introduced by the authors [2], whose goal is to infer Optimal Decision Trees (ODTs) under depth constraints.

DL8.5 conceptualizes decision tree paths as itemsets, representing features as positive or negative items, as illustrated below in Figure 2.2. Itemset mining techniques are used to efficiently look through the space of all possible paths, and it employs dynamic programming over itemsets to find an optimal decision tree. DL8.5 also highlights the importance of a minimum support constraint to reduce the size of the search space.

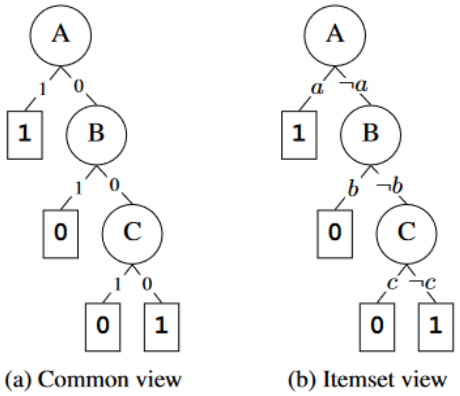


Figure 2.2: Itemset representation from the authors.

Figure 2.2 shows the difference between the common view of a tree representation in a binary way when branching the tree, while in the Itemset view, the feature variables are represented as positive or negative features, forming a sequence (itemset) when reaching a leaf. For instance, the leftmost leaf is the itemset ‘a’, and the rightmost leaf contains the itemset ‘¬a, ¬b, ¬c’.

2.4.1 Methodology

DL8.5 utilizes a search algorithm that combines Dynamic Programming with Branch-and-Bound search techniques. It systematically explores all feasible splits and selects the split having the lowest score. During the search process, it's common for different sequences of splits to select the same subset of data instances, leading to potential redundancies in computations. DL8.5 addresses this issue by employing a caching system, allowing for the reuse of results and optimizing the search process.

Additionally, it uses a bounding system to prune the search space. Once a subtree is discovered, its score serves as an upper-bound, constraining the quality of subsequent subtrees. Leveraging the additive nature of the scoring function, DL8.5 prunes a right-hand subtree if the left-hand subtree yields an error equal to or greater than the upper-bound. This approach optimizes the search by eliminating unnecessary exploration of suboptimal branches.

In summary, DL8.5 implements several strategies to enhance efficiency, such as branch-and-bound search to prune substantial parts of the search space, a novel caching method storing itemset information, and the consideration of different branching heuristics to speed up the search for optimal trees. Moreover, the algorithm is designed as an "any-time" algorithm, allowing halting at any point to report the best tree identified so far.

2.4.2 Learning ODTs

The learning of optimal decision trees (ODTs) can be performed by DL8.5 if the scoring function follows an additive criterion, for instance, an additive function over the leaves. The objective of DL8.5 is to determine the decision tree that minimizes the scoring function f(T), given by arg $min_{T \in T}$ f (T), adhering to certain constraints:

- T denotes the set of all decision trees within a depth of ≤ maxdepth, where the leaves must cover a minimum number of instances from the training data (minimum support).

• The function f(T) is an additive function across the leaves of the tree T, expressed as

f(T) = $\sum_{l \in leafs(T)} g(l)$, where g(l) ≥ 0 and individually assesses the quality of each leaf.

This function $g(l)$ evaluates the misclassification error of instances covered by a leaf that do not align with the majority class of that leaf. Despite its original application, DL8.5 is adaptable to other additive scoring functions such as MSE, and the class-based support score function shown below.

Error leaf calculation: Class-based support

Now, let's introduce a case of a score function based on class-based supports representing the error of an itemset, which follows an additive criterion.

The error in a leaf can be calculated as follows:

$$leaf\_error(I) = |cover(I)| - max_{c \in C}\{Sup(I,c)\} \quad (2.8)$$

where *cover (I)* indicates the set of transactions that belong to itemset *I*, and *Sup(I,c)* corresponds to the class support for an itemset indicating the number of transactions for a given class *c*, and *C* corresponds to the set of classes.

In the next chapter, we will cover the use of this leaf error function to address optimal decision trees with the loss functions under study (MSE, SPO).

Chapter 3

Previous work

This chapter introduces the primary concept that inspired this study, originating from the research conducted by Elmachtoub, Liang, and McNellis [3]. They explored the utilization of "Decision Trees for Decision-Making under the Predict-then-Optimize Framework", which sparked the initial idea for this thesis. Then, we introduce the adaptation of DL8.5 algorithm with SPO and MSE loss functions.

3.1 The predict and optimize framework

For the predict-then-optimize framework, a more general class of decision-making of optimization problems is proposed for training decision tree models using the SPO loss. The authors characterized these problems as optimization tasks with known constraints and an unknown linear objective function at the time of solving. The objective function can be estimated or predicted using feature data.

Let's define S $\subseteq$ R^d as the feasible region for decisions, where *d* corresponds to the Decision space dimension. The optimization decision problem can be stated as follows:

- $z^*(c) = min_{w \epsilon S}\ c^T w$; where c is a cost vector of the problem, and w is a binary vector representing decision variables.
- $W^*(c)$ = arg $min_{w \epsilon S}\ c^T w$ is the set of optimal decisions related to $z^*(c)$, and w^* corresponds to any member of the W set.

For this framework, true costs are not known in advance when solving $w^*(.)$ for optimal decisions, so a predicted cost vector ĉ is determined instead. The predictions relies on a ML model trained with an input dataset $\{(x_1, c_1), (x_2, c_2),\ldots,(x_n, c_n)\}$, where *x* denotes a vector composed of *p* features used for predicting the cost vector ĉ.

Now, let's consider $l(.,.)$: $R_d \times R_d \rightarrow R_+$ as the loss function used for training the model, measuring the loss incurred between the predicted cost ĉ and the true cost vector c. The ML models are trained by solving this empirical minimization problem.

$$f^* = arg\,min_{f \epsilon H} \frac{1}{n} \sum_{i=1}^{n} l(f(x_i), c_i) \tag{3.1}$$

where H represents the hypothesis class of ML model candidates, and f is a model within H used for predicting the cost vectors given the input feature vectors. So, the predicted cost vector ĉ corresponds to $f(x_i)$ and is evaluated against the real cost vector c_i under the loss function l. Finally, f^* refers to the optimal trained model within the Hypothesis class H.

In the context of loss functions, the Mean Squared Error (MSE) serves as a prevalent method to assess prediction error. However, the SPO loss focuses on decision quality rather than prediction error. SPO loss denoted as $c^T w^*(\hat{c}) - w^*(c)$, quantifies the additional cost or excess incurred when a potentially suboptimal decision $w^*(\hat{c})$ is made based on the predicted cost ĉ, considering c as the true cost.

As we know $W^*(c)$ can include more than one optimal solution associated to the predicted cost, so the authors proposed defining the SPO loss based on the worst-case decision among all the optimal solutions associated with the predicted cost vector ĉ.

$$l_{SPO}(\hat{c}, c) := max_{w \epsilon W^*(\hat{c})} \{c^T w\} - z^*(c) \tag{3.2}$$

It's important to note that the SPO loss function is discontinuous, non-convex, and therefore not differentiable given a predicted cost vector ĉ. Thus, some authors propose modifying the objective function of the optimization problem to create a differentiable, surrogate loss function, as pointed out by Wilder et al. [13]. Unlike previous approaches, several techniques can be applied to train decision trees directly using the SPO loss function, and it can be simplified by considering decision trees and exploiting their structural properties.

3.1.1 Methodology

Training decision trees using SPO loss function are known as SPO Trees (SPOTs). The goal of decision trees is to partition the observations into L leaves (R_1,R_2,.., R_l) to ensure that the predictions made by these leaves, when considered together, minimize a specified loss function.

$$min_{R_{1:l} \in T} \frac{1}{n} \sum_{l=1}^{L} (min_{\hat{c}_l} \sum_{i \in R_l} l_{SPO}(\hat{c}_l, c_i)) \tag{3.3}$$

The constraint $R_{1:l} \in$ T implies that the assignment of observations to leaves should adhere to a tree structure, which means that it is determined through successive splits on features.

Due to the non-convexity or discontinuity of SPO predicted costs, a simple way is considering the average cost vectors corresponding to a leaf which minimizes the SPO loss inside itself.

$$\bar{c}_l := \frac{1}{|R_l|} \sum i \epsilon R_l \tag{3.4}$$

Let $\bar{c}_l$ be the average cost of all observations of leave l as shown in 3.4. If $\bar{c}_l$ has a unique minimizer, then $\bar{c}_l$ minimizes within the leaf $\bar{c}_l = \arg min_{\hat{c}_l} \sum_{i \in R_l} l_{SPO}(\hat{c}_l, c_i)$. This theorem states that cost vector minimizing within the SPO leaf, can be simplified as the average of cost vectors corresponding to that leaf. It is helpful for simplifying the optimization problem as follows:

$$min_{R_{1:L} \in T} \frac{1}{n} \sum_{l=1}^{L} \left(min_{\hat{c}_l} \sum_{i \in R_l} l_{SPO}(\hat{c}_l, c_i) \right) = min_{R_{1:L} \in T} \frac{1}{n} \sum_{l=1}^{L} \sum_{i \in R_l} (c_i^T w^*(\bar{c}_l) - z^*(c_i)) \tag{3.5}$$

Recursive partitioning

For a reliable solution to the optimization problem, they propose the use of recursive partitioning for training SPO trees, as CART does for the minimization of the decision error.

Let's consider a tree split as (*j*,*s*), where j is the splitting feature and s is the splitting point. The splitting point *s* partitions the observations into 2 regions: $R_1(j,s)$= {i ∈ [n] | $x_{i,j} \leq s$}, and R_2 (*j*,*s*)= {i ∈ [n] | $x_{i,j} > s$}, where n is a set of observations {1,2,..,n}. The first split of the tree is made by choosing the splitting pair (*j*,*s*) that minimizes the optimization problem.

$$min_{j,s} \frac{1}{n} \left(\sum_{i \in R_1(j,s)} (c_i^T w^*(\bar{c}_l) - z^*(c_i)) + \sum_{i \in R_2(j,s)} (c_i^T w^*(\bar{c}_l) - z^*(c_i)) \right) \tag{3.6}$$

The training procedure follows a greedy approach by selecting a single split that yields the best Smart Predict-then-Optimize (SPO) loss on the training set. After a first split is chosen, this greedy approach is then recursively applied in the resulting leaves until one of potentially several

stopping criteria is met. Common stopping criteria can include a maximum depth size for the tree and/or a minimum number of training observations per leaf.

3.2 DL8.5 decision trees under MSE and SPO loss functions

Now, we will address the adaptation of the DL8.5 algorithm to be trained using the MSE function, and subsequently the SPO loss function. In the upcoming Experiments Chapter, we will explore and analyze the effects of employing optimal decision trees based on these specific error functions. This section makes reference to the work performed by Delphine Leclercq in her master's thesis [14].

Thanks to the implementation of DL8.5 python wrapper library (PyDL8.5), and its compatibility with scikit-learn library, it is possible to adapt or modify the error function for training the model with one of the loss functions mentioned.

3.2.1 MSE loss

Let's first describe the adaptation of the MSE error function for training a DL8.5 decision tree.

$$leaf\_error(I) = \sum_{\forall i: x_i \in cover(I)} \sum_{\forall j} (c_{i,j} - \hat{c}(I)_j)^2 \tag{3.7}$$

In the formula 3.7, the leaf_error function refers to the error_function to be adapted in the DL8.5 Predictor. Here, $c_{i,j}$ represents the j-th element of the cost vector c_i, while $\hat{c}(I)_j$ refers to the j-th element of the predicted cost vector associated with all observations located in the leaf node that denotes itemset *I*. Meaning that all feature vectors within the leaf related to itemset *I* will receive the same prediction value.

$$leaf\_value\ (I) = \sum_{\forall i: x_i \in\ cover(I)} c_i\ *\ (\frac{1}{cover(I)}) \tag{3.8}$$

The $leaf\_value$ equation (3.8) represents the predicted cost vector $\hat{c}(I)$ to be adjusted in the leaf_value_function of the DL8.5 Predictor. It essentially stands for an average cost vector computed from the cost vectors of the observations within the leaf node characterizing itemset *I*.

3.2.2 SPO loss

Next, we present the adaptation of the SPO loss function for training an optimal decision tree. We will consider the formulas of the Predict-then-Optimize framework for its implementation.

For the SPO loss, the formula 3.3 is adapted for only considering one leaf at a time to enable the application of the recursive partitioning approach within the algorithm, as follows:

$$min_R \frac{1}{n}\left(min_{\hat{c}_R} \sum_{i:x_i \in R} l_{SPO}(\hat{c}_l, c_i)\right) \tag{3.9}$$

where x_i represents the features vectors that respect the conditions to reach the leaf *R*. Then, the formula is simplified thanks to the theorem described in equation 3.5.

$$min_R \frac{1}{n} \sum_{i:x_i \in R} (c_i^T w^*(\bar{c}_l) - z^*(c_i)) \tag{3.10}$$

As mentioned before, $\bar{c}_l$ represents the average cost of the cost vectors of the observations within the leaf *R*. Following the formula, $c_i^T w^*(\bar{c}_l)$ represents the predicted shortest path, and $z^*(c_i)$ stands for the optimal shortest path (travel time).

Then, regarding the *leaf_error* function of the DL8.5 algorithm, we will adapt this formula to calculate the SPO loss error within one leaf.

$$leaf\_error(R) = \frac{1}{n} min_R \sum_{i:x_i \in R} (c_i^T w^*(\bar{c}_l) - z^*(c_i)) \tag{3.11}$$

Hence, in the context of the shortest path problem to be explained in the next chapter, the leaf error function calculates the shortest path for every observation within the leaf *R*. As observed, it calculates the difference between the predicted shortest path ($c_i^T w^*(\bar{c}_l)$) and the optimal shortest path represented by $z^*(c_i)$, and then it is divided by the number of training observations.

For this SPO scenario, the *leaf_value* function is the same as the one used for the DL8.5 with MSE loss function, which calculates an average cost of the observations within the leaf.

3.2.3 Adaptations

SPO loss

Regarding the SPO loss function, instead of using the Dijkstra algorithm for finding the shortest path given the cost vector of any observation, the *leaf_error* function of the DL8.5 Predictor was adapted to use the Gurobipy shortest path model [22]. As well, the binarization or discretization of input features was applied on both sizes on the SPO and MSE greedy trees, and the optimal DL8.5 trees for comparing both tree approaches in the same conditions.

```
def get_SPO_loss(tids):
    tids = list(tids)
    C_subset = train_cost[tids]
    SUM = 0

    A = find_opt_decision(C_subset)['objective']

    weights=np.array([1]*len(C_subset))
    mean_cost = (np.matmul(weights,C_subset)/sum(weights)).reshape(-1)
    pred_decision = find_opt_decision(mean_cost.reshape(1,-1))['weights'].reshape(-1)

    SUM=np.matmul(C_subset,pred_decision).reshape(-1) - A
    avg=np.dot(SUM,weights)/len(train_cost);

    return avg
```

Figure 3.1: Code of the error function of the DL8.5 SPO tree for the shortest path problem.

In Figure 3.1, the function loss is implemented for the leaf error of the DL8.5 Predictor for the shortest path problem.

- It receives a *tids* parameter representing the transactions (observations) within the leaf.
- Then, *A* denotes a list containing the shortest distance for all the *C_subset* (true cost of observations in that leaf), obtained by the gurobipy shortest path model.
- Next, mean cost is calculated by averaging the costs of *C_subset*.
- The prediction decision vector (*pred_decision*) is determined by the shortest path model on the predicted costs (mean cost).
- Finally, the SPO loss is calculated based on equation 3.11: where *C_subset* is multiplied by *pred_decision* to get the predicted travel time, and it is subtracted by the optimal travel time *A*, for then dividing this difference by the training size.

The adaptations of the SPO loss function for addressing the Traveling Salesman Problem (TSP) will be thoroughly discussed in Section 5.2. The approach follows this logic, where instead of using a shortest path model on the predicted cost vector, a TSP algorithm is instead applied to find the shortest tour.

Chapter 4

Shortest path problem

Now, let's consider the case of using decision trees for solving the shortest path problem. It is a common optimization problem which consists of finding the shortest path between two vertices in a weighted graph such that the total sum of the weights of its edges is minimized.

For our analysis, a 4x4 grid is considered as in the authors' paper [3]. This graph consists of edges (roads) which only follow south and east directions as shown in Figure 4.1. The driver begins at the northwest corner, whose goal is to travel and arrive at the southeast corner following the shortest path. The costs of the 24 edges of the network are unknown, but they can be predicted by a decision tree model given the input feature variables.

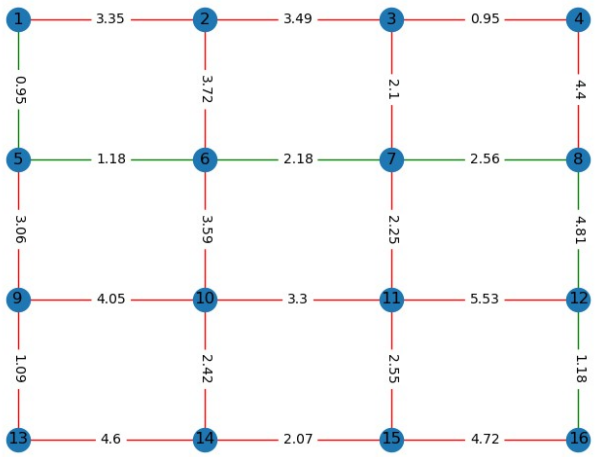


Figure 4.1: Grid graph example representing the shortest path problem.

4.1 Dataset generation

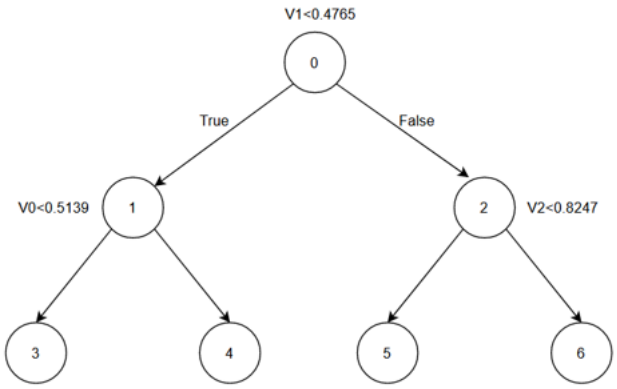
Synthetic datasets of size n {200,1000} cost-pairs are generated by sampling n features $\{x_1,\ldots, x_n\}$, each one following a Uniform distribution $U(0,1)^p$ where p is the number of features. Then, sampling a matrix $B \in \{0, 1\}\ d{\times}p$, following a Bernoulli distribution (1, 0.5), where d corresponds to the dimension of the decision space, and d=24 edges for the 4x4 grid. Each feature vector x_i is associated with a cost vector c_i according to 4.1.

$$c_{ik} = (\frac{1}{\sqrt{p}} \left(Bx_i\right)_k + 1)^{deg} . \varepsilon_i^k \quad (4.1)$$

where $(Bx_i)_k$ corresponds to the k_{th} element of Bx_i, and *deg* refers to degree which is a positive number controlling the non-linearity in the mapping features-cost vectors, and ε_i^k represents the noise terms which are multiplicative i.i.d. following a uniform distribution U([1-ε,1+ε), where ε>=0.

4.2 Illustration of a SPO tree for the shortest path problem

Let's see in detail a case for the shortest path problem using a SPO tree model, and then evaluate the performance of this model by using the normalized extra travel time metric.



Node 3

ĉ: [3.05, 3.49, 1.58, 1.62, 2.65, 1.86, 2.86, 2.17, 2.94, 2.11, 1.81, 2.98, 0.99, 2.25, 1.87, 3.82, 1.84, 3.41, 1.88, 3.34, 1.01, 2.54, 1.75, 1.51]

w*(ĉ): [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0.]

Node 4

ĉ: [3.97, 3.67, 1.69, 1.66, 2.64, 1.92, 4.05, 2.95, 3.77, 2.91, 1.91, 3.67, 1.02, 2.90, 1.81, 4.72, 2.45, 3.65, 1.87, 4.61, 1.00, 2.72, 2.63, 1.49]

w*(ĉ): [0. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1.]

Node 5

ĉ: [3.94, 3.82, 1.41, 1.42, 3.09, 2.43, 3.82, 3.22, 4.06, 3.20, 2.48, 3.93, 0.99, 3.24, 2.47, 4.95, 2.18, 3.89, 2.37, 3.38, 0.987, 2.99, 2.06, 1.52]

w*(ĉ): [0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1.]

Node 6

ĉ: [4.75, 4.60, 2.03, 2.00, 4.01, 2.47, 4.28, 3.17, 4.04, 3.21, 2.53, 4.93, 1.00, 4.01, 2.50, 5.80, 1.84, 4.78, 3.14, 4.21, 1.00, 3.92, 2.05, 1.53]

w*(ĉ): [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0.]

Figure 4.2: SPO tree representation.

In Figure 4.2, we present the structure of a decision tree which was fitted using the SPO loss function. This SPO tree was trained with a dataset consisting of 200 observations and was limited to a maximum depth of 2 for better and simpler analysis. Concerning the input vector, which comprises 5 features, 3 of them were selected as the best ones for the generation of this tree. At the root of the tree, V1 was chosen as the splitting point. Then, V0 and V2 were chosen as splitting points at depth 1. At the end, we observe the 4 leaves of the tree containing the predicted costs (24-edge vector).

Below the tree representation, the predicted cost vector $\hat{c}$ and the decision vector $w^*(\hat{c})$ for each leaf of the tree can be appreciated. As mentioned before, ĉ is the predicted cost representing the average cost of the observations within the leaf; while $w^*(\hat{c})$ is the binary decision vector resulting from applying the shortest path model over the ĉ vector. In practice, the shortest path is calculated over the predicted costs (average cost vector) using the shortest path model from the Gurobipy python package [22].

The 10 input observations for this analysis are as follows:

1. [0.80880135 0.67302529 0.22581728 0.46024956 0.70989696]
2. [0.00206493 0.37917248 0.51759482 0.91301433 0.09065132]
3. [0.5603494 0.1544516 0.38058657 0.90677262 0.30499968]
4. [0.01345053 0.4541592 0.48430903 0.85851887 0.30090809]
5. [0.80695748 0.3067844 0.89506413 0.08319855 0.17923856]
6. [0.16325439 0.99489785 0.91370149 0.82214478 0.19529747]
7. [0.48998723 0.40694317 0.69145744 0.94018254 0.78994814]
8. [0.25372992 0.1050244 0.92986928 0.73106338 0.05549343]
9. [0.89483322 0.69205976 0.69860098 0.41950782 0.43658883]
10. [0.86532842 0.2446527 0.96432677 0.93375804 0.11932977]

Next, we present the predictions for these 10 observations taken from the training set. These predictions are generated by employing the SPO tree in combination with the shortest path model.

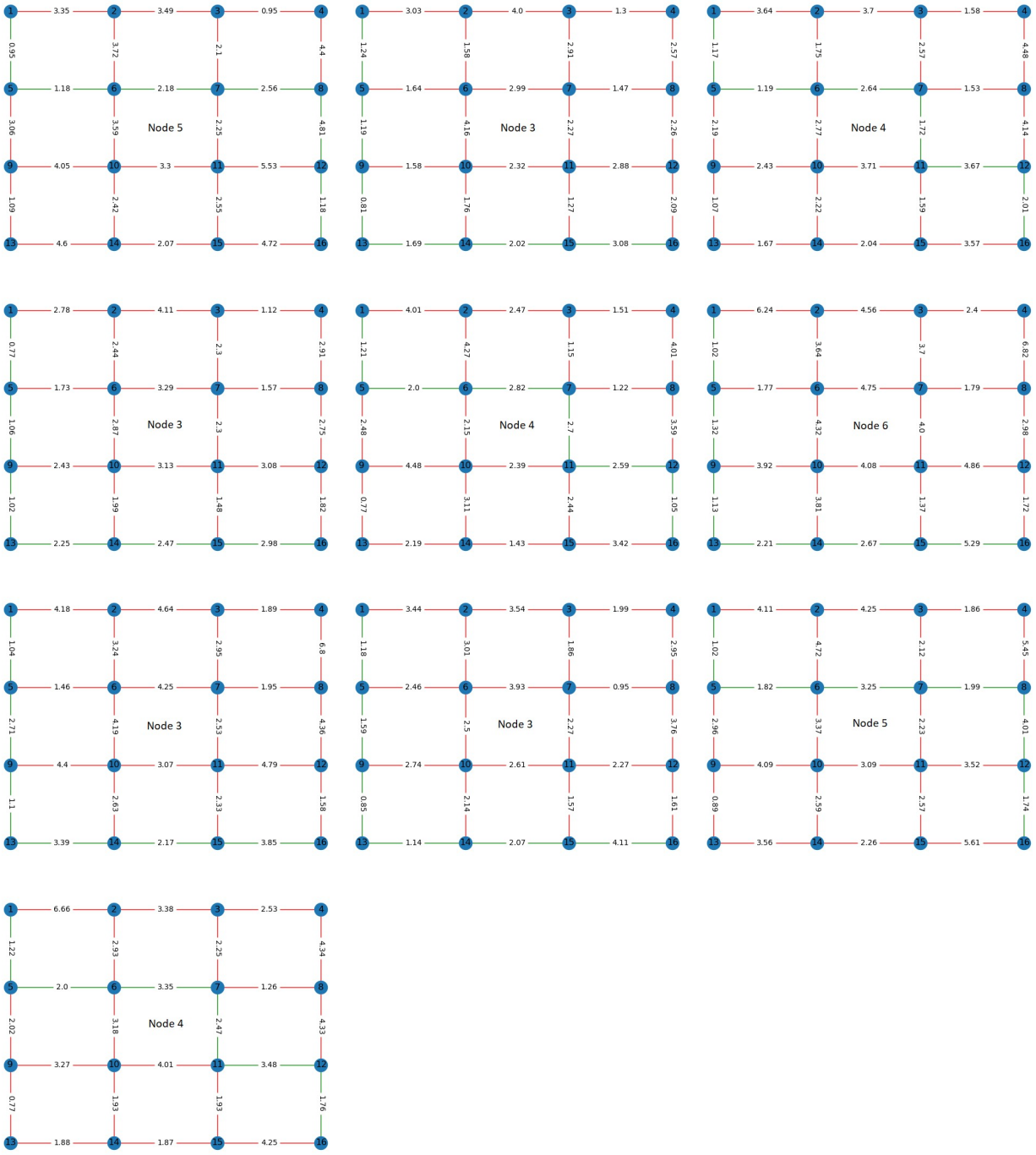


Figure 4.3: Shortest path found for the 10 observations using SPO loss (observations are sorted from top left to right).

There are 10 graphs in Figure 4.3, where each one illustrates the shortest path found for each observation. This is achieved by first using the SPO tree model to predict the 24 cost edges and then the decision vector is derived by applying the shortest path model over the predicted costs. At each graph, all the edges represent the real true costs, and the selected path in green is

determined by the decision vector. Moreover, it is indicated which leaf node the observation belongs to, and we can also notice the number of observations for each leaf: node 3 (#4), node 4 (#3), node 5 (#2), node 6 (#1).

4.2.1 Calculation of the shortest path

By following the formula $c_i^T w^*(\bar{c}_l)$, the real cost vector c_i^T is multiplied by the decision vector $w^*(\bar{c}_l)$. The decision vector essentially selects the costs (edges) that minimize the total cost path or travel time, as illustrated in Figure 4.3. This process entails summing the costs linked to the selected edges, as indicated by this decision vector.

That is, after predicting the cost vectors, a shortest path is computed based on each prediction. The travel time is the time taken to travel along this path determined by summing the costs associated with all the edges chosen in the graph for the shortest path.

Now, let's calculate the predicted shortest path:

- Predicted shortest path/travel time for the first observation:
  - 0.95+1.18+2.18+2.56++4.81+1.18 = 12.85
- Predicted shortest path for all observations:
  - [**12.85**, 10.03, 12.41, 10.56, 12.37, 13.64, 14.25, 10.94, 13.82, 14.29]

Then, the optimal shortest path is given by $z^*(c) = min_{w \in S} c^T w$, where c denotes the real costs and w is the decision vector for the optimal case found by the shortest path model applied over the real test costs.

- Optimal shortest path for the 10 observations:
  - [12.85, 10.03, 11.71, 10.56, 11.38, 13.64, 14.25, 10.94, 13.57, 12.00]

4.2.2 Extra travel time

Finally, let's calculate the Extra travel time for these observations:

The decisions are scored on a held out set of data using the metric of "normalized extra travel time", defined as the cumulative SPO loss normalized by the cumulative optimal decision costs.

$$\sum_{i=1}^{n} l_{SPO}\left(\hat{c}_i, c_i\right) / \sum_{i=1}^{n} z^*(c_i) \tag{4.2}$$

This is a metric that measures the difference between the travel time of the predicted costs $\hat{c}_i$ and the true travel time. And then, it is normalized by dividing by the optimal travel time.

Let's see another case, the fifth observation in terms of its predicted and optimal travel times, and the extra travel time calculation.

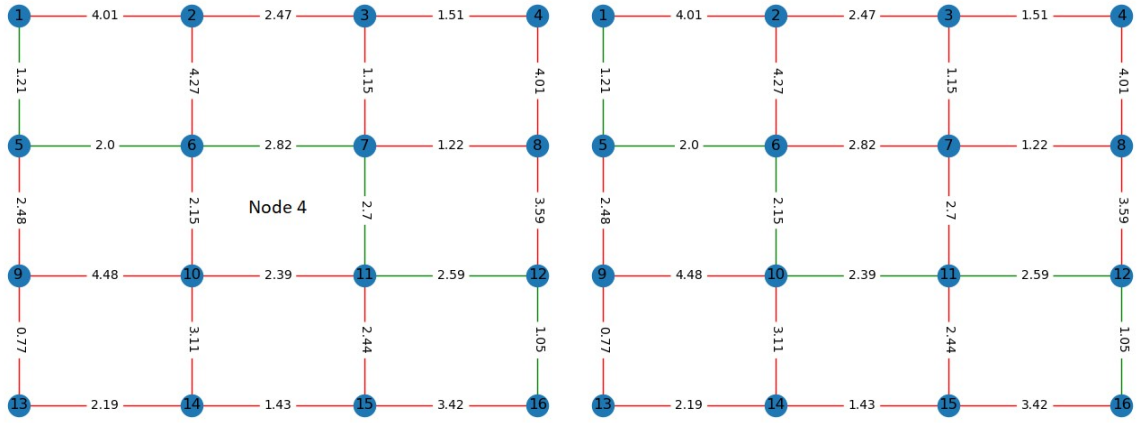


Figure 4.4: Predicted shortest path (left) vs Optimal shortest path (right) for the fifth observation.

For Figure 4.4, the normalized extra travel time for the fifth observation is calculated as follows:

- SPO Extra travel time (predicted shortest path):
  - 1.21+2.00+2.82+2.7+2.59+1.05=12.37
- Optimal shortest path:
  - 1.21+2.00+2.15+2.39+2.59+1.05=11.38
- Normalized Extra travel time: (Predicted – Optimal)/Optimal
  - (12.37 - 1.38)/ 1.38 = 0.09

The same calculations are performed for the 10 observations:

- Normalized Extra travel time:
  - [0.00, 0.00, 0.06, 0.00, **0.09**, 0.00, 0.00, 0.00, 0.02, 0.19]
- Average extra travel time: 0.04

We can determine that this SPO tree model was able to solve the shortest path problem, showing superior performance over these 10 observations by achieving the best-case scenario (decision error or extra travel time of 0) in 6 out of 10 instances. This showcases a notably strong performance over these training observations with an average extra travel time of 0.04.

Chapter 5

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a well-known combinatorial optimization problem where the goal is to find the shortest possible route that allows a salesman to visit a set of given cities exactly once and return to the starting city. The problem is often described as finding the optimal Hamiltonian cycle in a complete graph, where the cities are represented as nodes and the distances between them as edges [24], as shown below.

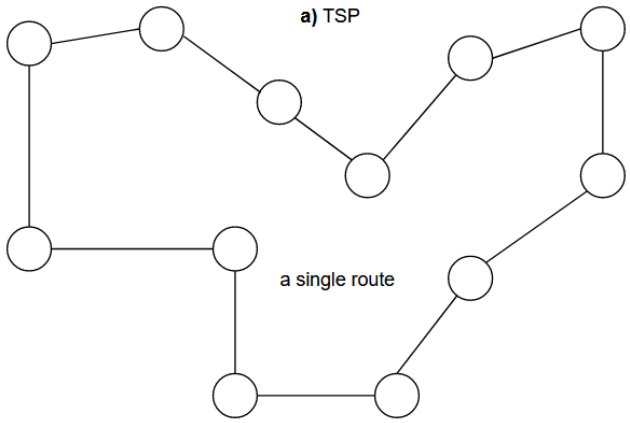


Figure 5.1: TSP representation.

The TSP is NP-hard, meaning that there is no known polynomial-time algorithm that can solve it for all instances, and it is considered one of the most challenging combinatorial optimization problems.

Formally, the TSP can be defined as follows.

Given:

- A set of cities $\{C_1, C_2, ..., C_n\}$, where each city has a coordinate in a 2D plane.
- A distance or cost function $d(C_i, C_j)$ that represents the distance or cost between two cities C_i and C_j.

The goal is to find a permutation of cities (a tour) that minimizes the total distance or cost while visiting each city exactly once and returning to the starting city, forming a circuit or closed tour.

There are several algorithms and approaches to tackle the TSP, and we are going to focus on four algorithms: TSP dynamic programming as an exact algorithm, and heuristic algorithms such as: TSP local search, TSP simulated annealing, and vehicle routing model.

5.1 TSP Algorithms

In this subsection, the four TSP algorithms will be described, specifically those used in the part of the experiments that involves the use and evaluation of exact and heuristic TSP models with decision trees.

5.1.1 TSP Dynamic programming

Dynamic programming

Dynamic programming is a method for solving optimization problems by breaking them down into smaller, overlapping subproblems and efficiently solving each subproblem only once. The approach involves formulating the problem recursively and using memoization or tabulation to store the solutions of subproblems for reuse, leading to improved efficiency [15].

This algorithm is employed when seeking minimum or maximum solutions to a problem. It ensures finding the best possible solution if such a solution exists.

It allows us to efficiently solve complex optimization problems that would otherwise be computationally infeasible using other techniques, such as brute force or exhaustive search. It is a powerful tool for solving a wide range of problems in various fields, including computer science, operations research, and algorithm design.

TSP

Dynamic programming can be applied to solve the Traveling Salesman Problem (TSP), and its approach for solving TSP is known as the Held-Karp algorithm.

Here's the formulation of the TSP problem using dynamic programming:

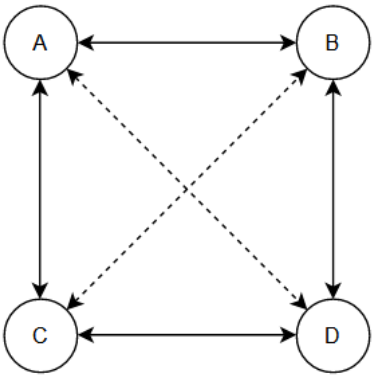
$$g(i, V) = \{ g(j, V - \{i\}) + d(j, i)\} \quad (5.1)$$

Let's define *d(i, j)* as the distance between cities i and j (base case). The function *g(j, V – { i })* is the cost of the path starting from city j, and it is the recurrence formula. *V* corresponds to the list of vertices (cities) in the given graph. The goal is to minimize the cost function, where $j \in V$ and $i \notin V$.

Assuming the graph contains n vertices V1, V2, ..., Vn, the TSP aims to find a path that covers all vertices exactly once while minimizing the total traveling distance.

This formula represents the recursive relationship of the cost function *g(i, V)*, which is defined as the minimum distance from city i to the remaining set of cities *V*. It iterates over all possible cities *j* in *V* and calculates the cost of traveling from city i to city j, (*d[i, j]*), plus the cost of the path starting from city *j* and excluding *i* from the set of vertices, (*g(j, V – {i})*). The minimum value among all such combinations gives the shortest path for traveling from city *i* to all other cities in *V*. The process is repeated until the optimal solution is achieved [16].

Let's see an illustrative example:



	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0

Figure 5.2: Vertices Graph with its respective distance matrix.

Figure 5.2 shows a connected graph of four cities, for which the recursive formula will be applied to find the shortest tour starting from city A.

$$g(A,\{B,C,D\}) = \min \{\ d(A,B) + g(B,\{C,D\}) = 16 + 22 = 38$$
$$d(A,C) + g(C,\{B,D\}) = 11+28 = 39$$
$$d(A,D) + g(D,\{B,C\}) = 6+17 = \mathbf{23} \text{ -> The minimum path}$$
$$\}$$

Below, we only focus on d(A,D) + g(D,{B,C}) to directly see the final calculations of the shortest path for this graph.

$$g(D,\{B,C\}) = \min \{\ d(D,B) + g(B,\{C\})\ \} = 12+17 = 29$$
$$d(D,C) + g(C,\{B\}) = 2+15 = \mathbf{17} \text{ -> Minimum}$$
$$\}$$

$$g(B,\{C\}) = \min\{\ d(B,C) + g(C, \Phi)\ \} = 13 + g(C,A) = 13+4 = 17$$

$$g(C,\{B\}) = \min \{\ d(C,B) + g(B, \Phi)\} = 7 + g(B,A) = 7+8 = 15$$

From d(A,D) + g(D,{B,C}) as the shortest path, we can obtain the final sequence:

$$A\text{->}D\text{->}C\text{->}B\text{->}A = 23$$

5.1.2 TSP Local Search

TSP local search is an optimization algorithm that starts with an initial solution and iteratively improves it by making small changes to the current solution. It explores the neighborhood of the current solution to find better solutions, aiming to reach a local optimum. Local search algorithms may use different neighborhood structures, such as 2-opt (swapping two edges) or 3-opt (swapping three edges), to explore different solutions. The algorithm keeps searching in the neighborhood until no further improvements can be made, at which point it terminates and finds good solutions quickly, but it may not guarantee the global optimal solution.

Let's introduce two TSP local search algorithms.

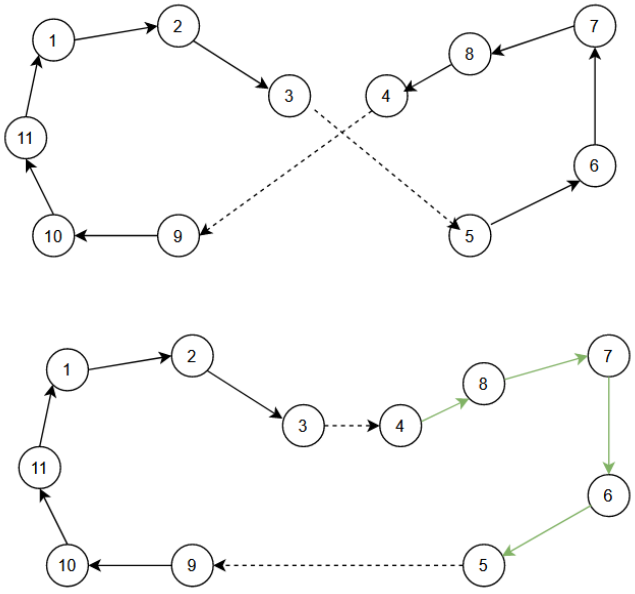


Figure 5.3: Local Search 2-opt approach.

TSP Local Search with 2-opt

It is an optimization technique used to improve solutions in the Traveling Salesman Problem (TSP). The local search algorithm aims to enhance the initial solution (e.g., obtained from a heuristic or other methods) by iteratively exploring the neighborhood of the current solution. In the case of 2-opt, it focuses on swapping two edges of the tour to create a new potential solution.

The 2-opt algorithm for TSP local search [17] can be represented mathematically as follows:

Given a set of cities $\{C_1, C_2, ..., C_n\}$ and a distance matrix d(i, j) representing the distance between city i and city j:

1. Start with an initial tour T0 that visits all cities in a specific order (T0 obtained from a nearest neighbor or other heuristics).
2. Compute the total distance D(T0) of the initial tour T0 by summing the distances between consecutive cities in the tour:

$$\text{best\_distance} = D(T0) = d(C_1, C_2) + d(C_2, C_3) + \ldots + d(C_{n-1}, C_n) + d(C_n, C_1)$$

3. Repeat the following steps until no improvements can be made:
   a) For each pair of edges (C_i, C_j) and (C_k, C_l) in the current tour T, where i, j, k, l are distinct indices, compute the new tour T' by swapping the edges:

$$T' = (C_1, C_2, \ldots, C_i, C_k, \ldots, C_j, C_l, \ldots, C_n).$$

   b) Compute the total distance D(T') of the new tour T' by summing the distances between consecutive cities in the tour:

$$\text{new\_distance} = D(T') = d(C_1, C_2) + d(C_2, C_3) + \ldots + d(C_i, C_k) + d(C_k, C_j) + d(C_j, C_l) + \ldots + d(C_{n-1}, C_n) + d(C_n, C_1).$$

   c) If new_distance is less than best_distance:
      - Update the current tour T to T'.
      - Update best_distance to new_distance
4. Return the final tour T when no further improvements can be made.

The 2-opt local search is called "2-opt" because it deals with pairs of edges. It is a simple yet effective optimization method that can improve the quality of a TSP solution. By iteratively performing these edge swaps, the algorithm explores different solutions within the neighborhood of the current tour and aims to find a better overall tour with shorter total distance. However, like other local search algorithms, it does not guarantee finding the global optimum, but it can significantly improve the initial solution and provide good results for many TSP instances.

TSP Local Search with Perturbation Scheme (PS6)

The 2-opt and 3-opt are the common techniques used for improving solutions in local search algorithms. But during the experiments, we realized that working with the PS6 scheme was providing better results for the Traveling Salesman Problem (TSP) in the context of two specific TSP algorithms: Local search and Simulated annealing.

S6: A subsequence is reversed and/or moved

1 2 **3 4 5 6** 7 8 9 10
1 2 7 8 **6 5 4 3** 9 10

Figure 5.4: Perturbation scheme from solvers of this repository [18].

As we can see on the sequence above, it operates directly within the permutation space. It consists of taking a subsequence and altering its order by reversing and moving it. It relies on a variation of neighborhood search or perturbation scheme such as the example below, where the edges (2,3), (8,9), (6,7) were altered and now correspond to (2,7), (8,6), (3,9); and the direction or sense of these edges was reversed: (3,4), (4,5), (5,6)

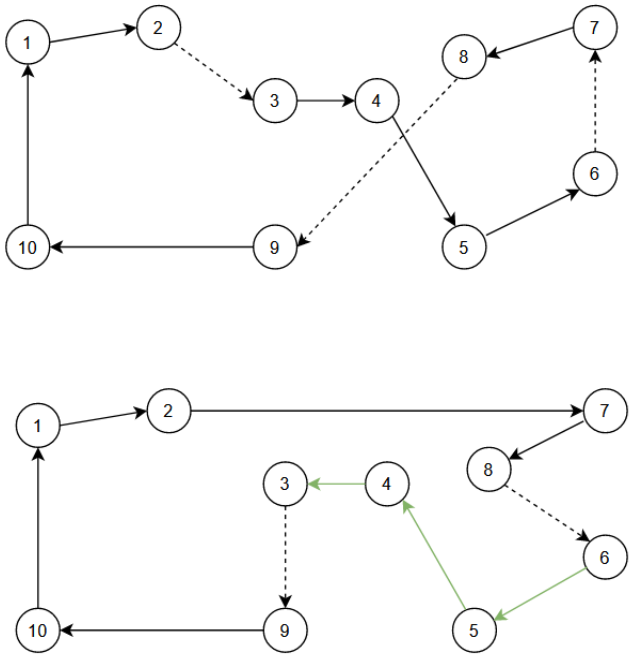


Figure 5.5: Local search with PS6 scheme.

5.1.3 TSP Simulated Annealing

Simulated Annealing (SA) is a heuristic optimization method applied to solve combinatorial optimization problems like the Traveling Salesman Problem (TSP). The algorithm starts with an initial solution, often a randomly generated tour, and iteratively explores neighboring solutions by making changes (swapping edges in the tour, for instance).

The core idea of SA lies in accepting new solutions even if they are worse than the current solution to avoid getting stuck in local optima. It does this by allowing "bad moves" initially but gradually reducing the probability of accepting worse solutions as it progresses.

The formula for acceptance probability, often based on this criterion [19], is:

$$P(e, e', T) = \begin{cases} 1 & if\ e' < e \\ e^{-\Delta E/T} & if\ e' \geq e \end{cases} \quad (5.2)$$

Where:

- e is the energy (or cost) of the current solution.
- e′ is the energy (or cost) of the proposed new solution.
- T is the current temperature.
- ΔE is the change in energy between the current and proposed solutions.

The temperature T decreases over time according to an annealing schedule, controlling the probability of accepting worse solutions. Initially, the algorithm explores widely (high temperatures), gradually reducing exploration (lower temperatures), allowing it to converge towards an optimal or near-optimal solution.

Regarding the search heuristics, this algorithm can also use different neighborhood structures, such as 2-opt, 3-opt, PS (perturbation schemas), to explore different solutions like the local search TSP algorithm.

5.1.4 Google OR-Tools: Routing model

Vehicle routing is one of most common optimization problems, where the goal is to find the best path or route (least total distance) for a group of vehicles that need to visit a set of cities.

The most well-known routing problem is Traveling salesperson (TSP), which consists of finding the shortest path for a single car (salesperson) who needs to visit a set of customers at different locations and come back to the starting point. While a more general version is when there are multiple vehicles, this problem transforms to a vehicle routing problem (VRP) where there are other constraints to consider such as the maximum volume or weight of packages that a vehicle can carry, and specific time windows that drivers can visit a location that usually customers request [20]. Below, there is an illustration showing the difference between TSP and VRP.

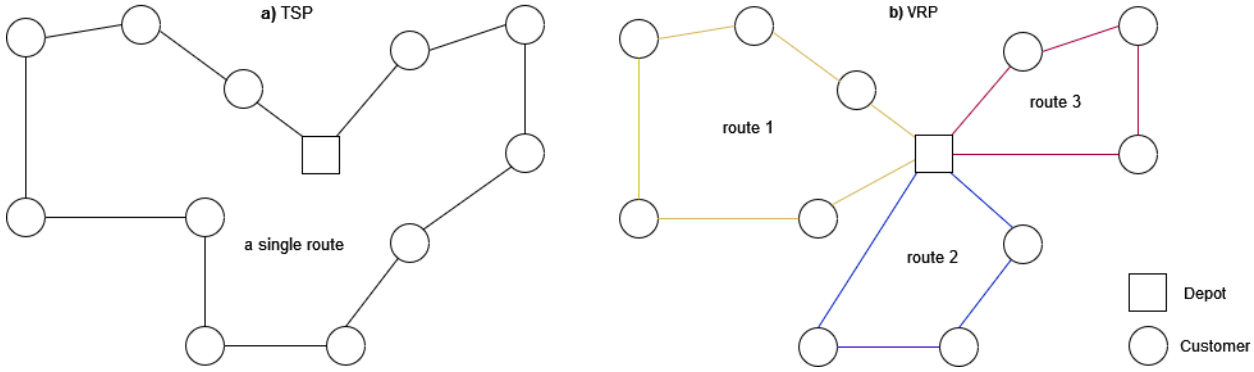


Figure 5.6: Representation of TSP and VRP problems.

For this study, we are going to focus on solving the TSP problem using the routing vehicle approach in order to find the shortest possible route (single route) for one vehicle that will visit a set of cities and return to the starting point. Hence, the VRP will become a TSP problem, and it will be called the routing model in the next sections.

Regarding the Experiments section, the routing model under study comes from the Google OR-tools python package [20]. The default parameters used in this model are: Path Cheapest Arc as the first solution strategy, and automatic local search metaheuristics for the search parameters, thus it will use a greedy approach for finding the shortest tour.

5.2 Decision trees with TSP algorithms

For the Traveling Salesman Problem TSP, we extended the utilization of decision trees beyond the shortest path problem to encompass the TSP as well. Hence, the SPO and MSE tree models were adapted to tackle the TSP, aiming to determine the shortest possible route or the minimum tour among a set of cities or nodes.

5.2.1 SPO loss

SPO tree

To carry out this process using the SPO loss function, there are some adaptations to highlight in the implementation of the SPO greedy model, such as:

- When fitting the model, as mentioned before, the leaf value contains the average cost of the observations within that leaf. There is also a decision vector based on the average cost vector (24 edges), for which it was before solved by a shortest path model, and now a TSP model (dynamic programming, local search, simulated annealing, or routing) is applied instead to find the shortest tour.
- Regarding the SPO loss function, '*SPOloss =* np.matmul*(Y,self.decision) - A'*, it calculates the difference between the predicted shortest tour and the optimal one. Here, *Y* states for the real cost vector, and is multiplied by the decision vector obtained from a TSP model which is applied on the average cost vector, and *A* represents the optimal shortest tour calculated by the TSP dynamic programming algorithm.

In the DL8.5 SPO tree, more details will be expanded because the logic is the same for both greedy and optimal tree approaches for the leaf error function.

DL8.5 SPO tree

For solving the TSP problem using DL8.5 algorithm, the equation 3.11 was also adapted to use the SPO function in the leaf_error function of the DL8.5 Predictor.

```
total_edges=16

def get_SPO_loss(tids):
    tids = list(tids)
    C_subset = train_cost[tids]
    SUM = 0
    A_subset = A[tids]

    weights=np.array([1]*len(C_subset))
    mean_cost = (np.matmul(weights,C_subset)/sum(weights)).reshape(-1)
    pred_decision = calculateMinTourAlgo(mean_cost, algoPredict)[0]
    total_decision=np.sum(pred_decision)

    if(total_decision < total_edges):
        diff= total_edges - total_decision
        total_dist = (np.matmul(C_subset,pred_decision).reshape(-1)) + (diff*25000)
    else:
        total_dist= np.matmul(C_subset,pred_decision).reshape(-1)

    SUM=total_dist - A_subset
    avg=np.dot(SUM,weights)/len(train_cost);

    return avg
```

Figure 5.7: Code of the error function of the DL8.5 SPO tree for the TSP problem.

Figure 5.7 shows the implementation of the SPO loss in the DL8.5 Predictor leaf error function for the TSP problem. It is basically the same logic as the SPO greedy loss function including the penalization variable. Below the details:

- It receives a *tids* parameter representing the transactions (observations) within the leaf.
- Then, *A* denotes a list containing the shortest tour for all training observations which was pre-computed due to the time execution when using the exact TSP algorithm (dynamic programming) on the true costs, thus it is then filtered based on the *tids* transactions to obtain a subset (*A_subset*).
- Next, *mean_cost* is calculated by averaging the costs of *C_subset*.
- The prediction decision vector (*pred_decision*) is determined by applying a TSP model (exact or heuristic algorithm) on the predicted costs (*mean_cost*).
- We also consider a variable total_edges for determining if the heuristic algorithm was able to find the complete shortest tour, that is, the 16 edges for the 4x4 grid because sometimes local search and simulated annealing models cannot find all edges; therefore a penalisation of 25000 multiplied by every missed edge is added to the predicted travel time in the IF condition. It will be further explained in Section 6.2.
- Finally, the SPO loss is calculated based on equation 3.11: where *C_subset* is multiplied by *pred_decision* to get the predicted travel time plus the penalization if it exists. Then, it is subtracted by the optimal travel time *A_subset*, for then dividing this difference by the training size.

5.2.2 MSE loss

For the MSE model, it is the same logic used in the shortest path problem regarding its loss function, which calculates the average squared difference between the predicted costs and the real costs. This function keeps the same because it doesn't evaluate decision errors.

Once the predicted cost vector is found by the MSE tree, the calculation of the shortest tour can be determined by applying a TSP algorithm (exact or heuristic model) over the predicted costs to find the decision vector. Then, we proceed to sum the cost of edges selected (travel time), and finally the extra travel time of the model is calculated.

This approach was only implemented for the MSE greedy trees for addressing the TSP optimization problem.

Observation

We can notice that the MSE loss function is based on a Predict-Then-Optimize approach because MSE trees are trained for minimizing the prediction cost error in their loss function, so they will first predict the unknown parameters for then addressing the optimization problem. Conversely, SPO loss function is rather based in a Predict-And-Optimize approach because SPOTs apply a combination of both predicting and optimizing phases during the training phase, meaning that iteratively they predict the cost vector to then use it in the optimization function with the goal of minimizing the decision error. Once the tree is fitted, the process is similar to MSE (predicts the costs and then resolves the optimization task). Thus, the difference lies in the training phase.

5.2.3 An illustrative example: TSP extra travel time calculation

Let's see an example when training a SPO tree with the Routing model, and then used to find the minimum cost tour. For this scenario, we will evaluate the extra travel time of this model for the TSP problem.

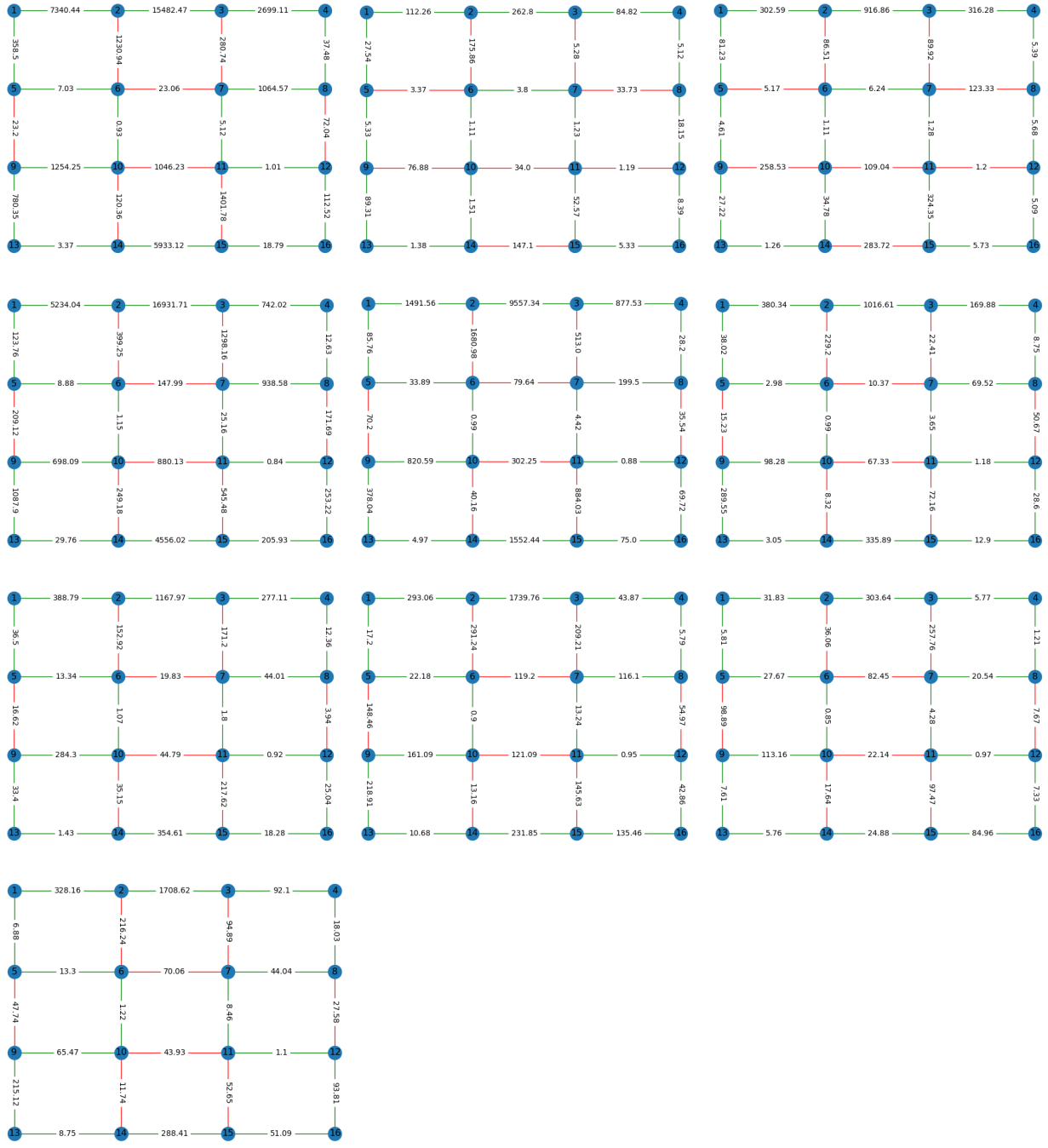


Figure 5.8: TSP Routing model: 10 test observations using dataset 0 (deg=10, eps=0.25, dep=3) Observation: The samples are ordered from top left to right.

In Figure 5.8, we present 10 test observations where each one is represented by a graph of 24 real cost edges, and contains the 16 edges chosen in green representing the minimum cost tour. For this purpose, the TSP routing was adapted to train the SPO tree model. Once fitted, this tree model predicts the costs of the 24 edges for each observation. Finally, the TSP routing algorithm is applied on this cost edge vector to find the decision vector for the minimum tour.

This decision binary vector contains the 16 chosen edges and is applied over the real test costs. In terms of formula, this involves the expression $c_i^T w^*(\bar{c}_l)$, where c_i^T is the real test cost vector, and $w^*(\bar{c}_l)$ is the decision vector obtained after applying the TSP routing model over the predicted cost vector $\bar{c}_l$, which is an average cost vector representing a leaf of the tree model.

When applying the formula, the result for each observation is obtained by multiplying the decision vector over the real costs, and it is represented at each graph with its respective minimum selected tour in green color.

Below, the corresponding travel time for each observation:

- Routing model: [35099.05, 680.64, 2039.71, 30849.67, 15180.86, 2460.19, 2660.91, 3053.89, 646.29, 2944.56]

The first element of the array 35099.05 can be calculated by following the selected path of the first graph on the Figure above:

- Minimum tour found: 1-2-4-8-7–11-12-16-15-14-13-9-10–6-5-1
- Result (summing the costs of the edges of minimum tour): 7340.44+15482.47+2699.11+37.48+1064.57++5.12++1.01+112.52+18.79+5933.12+3.37 +780.35+1254.25+0.93+7.03+358.5 = 35099.05

Then, the optimal minimum tour is given by: $z^*(c) = min_{w \in S}\, c^T w$, where c represents the real costs and w is the decision vector for the optimal case for which the TSP dynamic programming was applied over these costs.

- Optimal result: [15548.91, 630.64, 1326.67, 11590.26, 6616.58, 1406.76, 1454.97, 1838.79, 602.82, 1378.77]

Finally, the extra travel time for each instance is calculated by: $(c_i^T w^*(\bar{c}_l) - z^*(c_i))/z^*(c_i)$; that is: (Routing model result - Optimal result) divided by Optimal result

- Extra travel time: [1.26, 0.08, 0.54, 1.67, 1.29, 0.75, 0.83, 0.66, 0.07, 1.14]
- Average: 0.83

Chapter 6

Experiments

Now that we've outlined the shortest path and TSP problems, it's time to evaluate how the SPO and MSE loss functions impact the performance of both greedy and optimal decision trees. The assessment will primarily take place on a 4x4 grid for both scenarios, expanding to a 16x16 grid solely for the shortest path problem.

This chapter encompasses various experiments conducted across different dataset configurations, specifically degrees $\in$ 2, 10, $\epsilon \in$ 0, 0.25, n_train=200, n_test=1000 (always). These synthetic datasets were generated as detailed in Section 4.1 (based on the authors' methodology). The focus of the analysis will be on these datasets to address both optimization problems.

Decision tree configuration

For the experiments, the greedy decision trees (SPO and MSE trees) and the optimal trees (DL8.5_SPO and DL8.5_MSE) are configured as follows:

- n_train=200 (default, set by authors) or n_train=10000 (larger datasets, greedy models only).
- minimum leaf size (min_sup) = 10, ensuring sufficient accuracy for cost vector predictions.
- maximum depth=3 (default), otherwise specified in the experiment, such as: 1, 2, 3, unlimited.
- time_limit=14400, applied for the DL8.5 algorithm.

Regarding the greedy algorithm, the authors' implementation was used [23], where they refer to SPO trees as SPOTs and MSE as CARTs. But for our analysis, since they did not consider alternative algorithms in their study, for instance: DL8.5 can be used with MSE loss, and the greedy algorithm is also used with SPO loss. It would seem better to refer to their approaches as MSE trees (greedy) and SPO trees (greedy).

As for the extra travel time metric assessed in the experiments, it will always refer to the "normalized extra travel time" for both optimization problems.

Next, let's first cover the evaluation of decision trees in the shortest path problem.

6.1 Shortest path problem

In the upcoming experiments, we will first evaluate the performance of the SPO and MSE greedy trees over the shortest path problem based on the extra travel time metric. Subsequently, we will analyze the performance impact of employing larger training sets and assess the complexity of these greedy trees. Following this, DL8.5 is introduced to conduct comparisons between optimal and greedy trees using both loss functions, and then analyze the time performance of these algorithms. Lastly, we expand the scope of our analysis to a 16x16 grid to further evaluate the performance of DL8.5 against greedy decision trees.

6.1.1 Comparing MSE with SPO trees

Extra travel time vs Depth

This experiment evaluates the extra travel time performance of the MSE and SPO trees across different depths: 1, 2, 3, and unlimited depth. It was executed on 10 different datasets for each dataset configuration, so each point on the graphs represents an average extra travel time for the training and test sets.

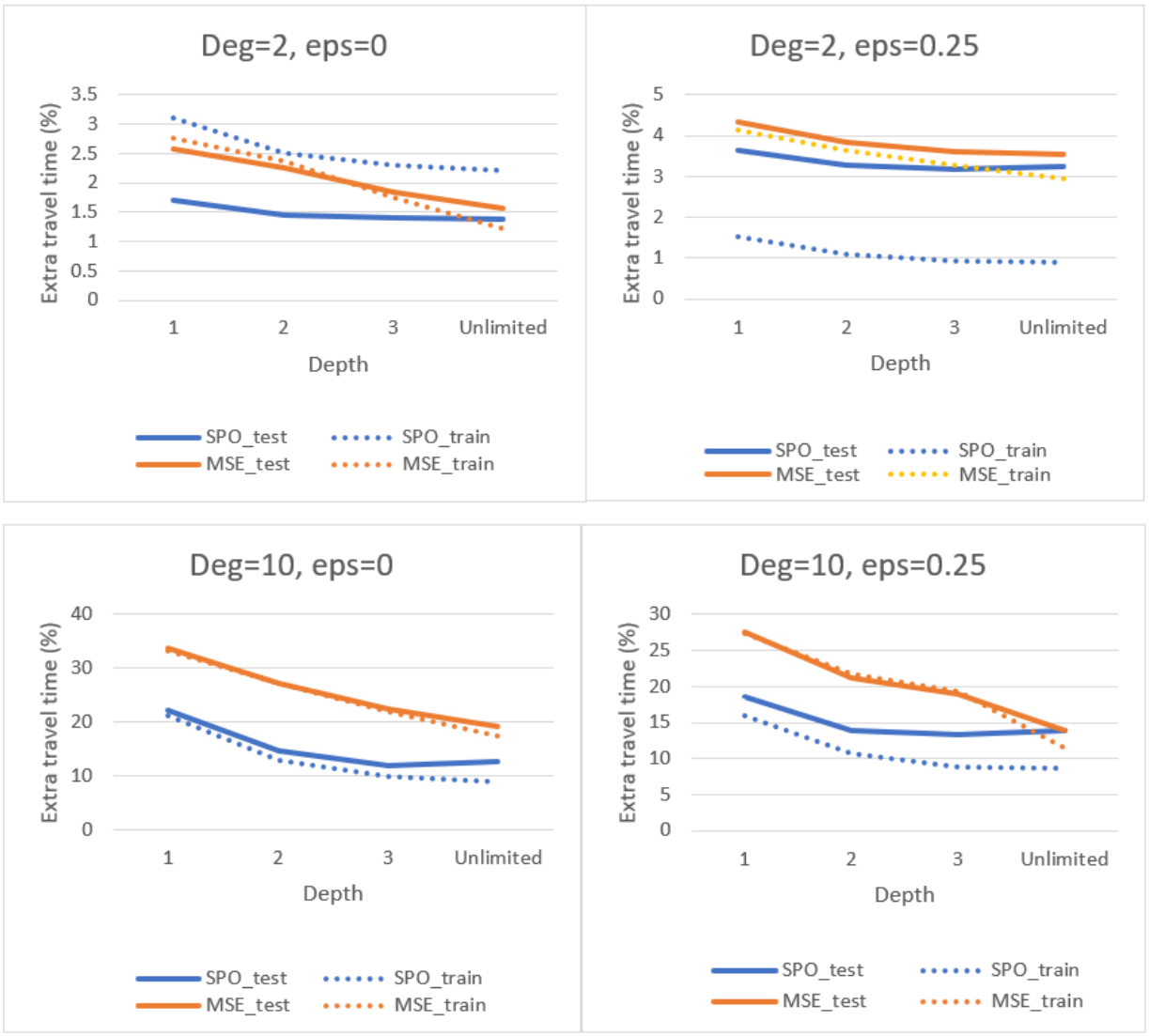


Figure 6.1: Extra travel time vs Depth (n_train=200).

In Figure 6.1, it is observed that for both datasets of degree 2, SPO is the best performer when varying the depth size, but as the depth approaches its maximum value, MSE shows a tendency to perform very similarly to SPO, specially for the test predictions. Regarding the training predictions, we can notice that SPO tries to fit the data more than MSE for the noisy cases, however, it continues being the best model.

When working with datasets of degree 10, the extra travel time between the two models is more pronounced compared to the previous scenario (degree 2), due to the non-linearity of the data, and we observe SPO surpasses MSE with lower decision error at every small depth. However, when both trees are at their maximum depth (unlimited), they tend to display similar results, particularly when dealing with noisy datasets (eps=0.25), and for the unlimited depth case (right), MSE slightly outperform SPO (13.85 vs 13.91).

The failure of the MSE model is given by the fact it optimizes over the error prediction rather than the decision error, and it is also given by the limited amount of data. But it starts to have very close behaviors when working with trees of larger depths.

Another aspect to remark is the performance on the training set of these decision trees for the case of noisy data (both degrees). The extra travel time tends to be smaller when fitting a tree with SPO loss, but this improved performance does not necessarily translate to similar results for the test dataset. However, it is still the best performer. Conversely, the MSE trees typically exhibit comparable performance between the training and test sets.

Computational time

Depth	SPO (sec)	MSE (sec)
1	0.29	0.32
2	0.53	0.55
3	0.73	0.74
Unlimited	1.10	0.94

Table 6.1: Computational time vs Depth for larger datasets (n_train=200).

Regarding the time performance for fitting SPO and MSE trees, both models present very similar results. Their time increases when scaling the depth of the trees, but it is minimal. As we can see, for the unlimited depth case, the time consumption of these algorithms tend to be 1 second.

In the next experiments, we will realize how the complexity of these trees is affected when working with larger datasets or using a larger grid scenario.

6.1.2 Comparing MSE with SPO trees: Larger datasets

For this analysis, we assess the decision performance of both greedy trees using larger training sets of 10000 observations across different depths, and a test set of 1000 observations. Each point on the graphs represents an average across 10 datasets. Only for the number of leaves vs depth experiment, we focus on 1 dataset.

Extra travel time vs Depth

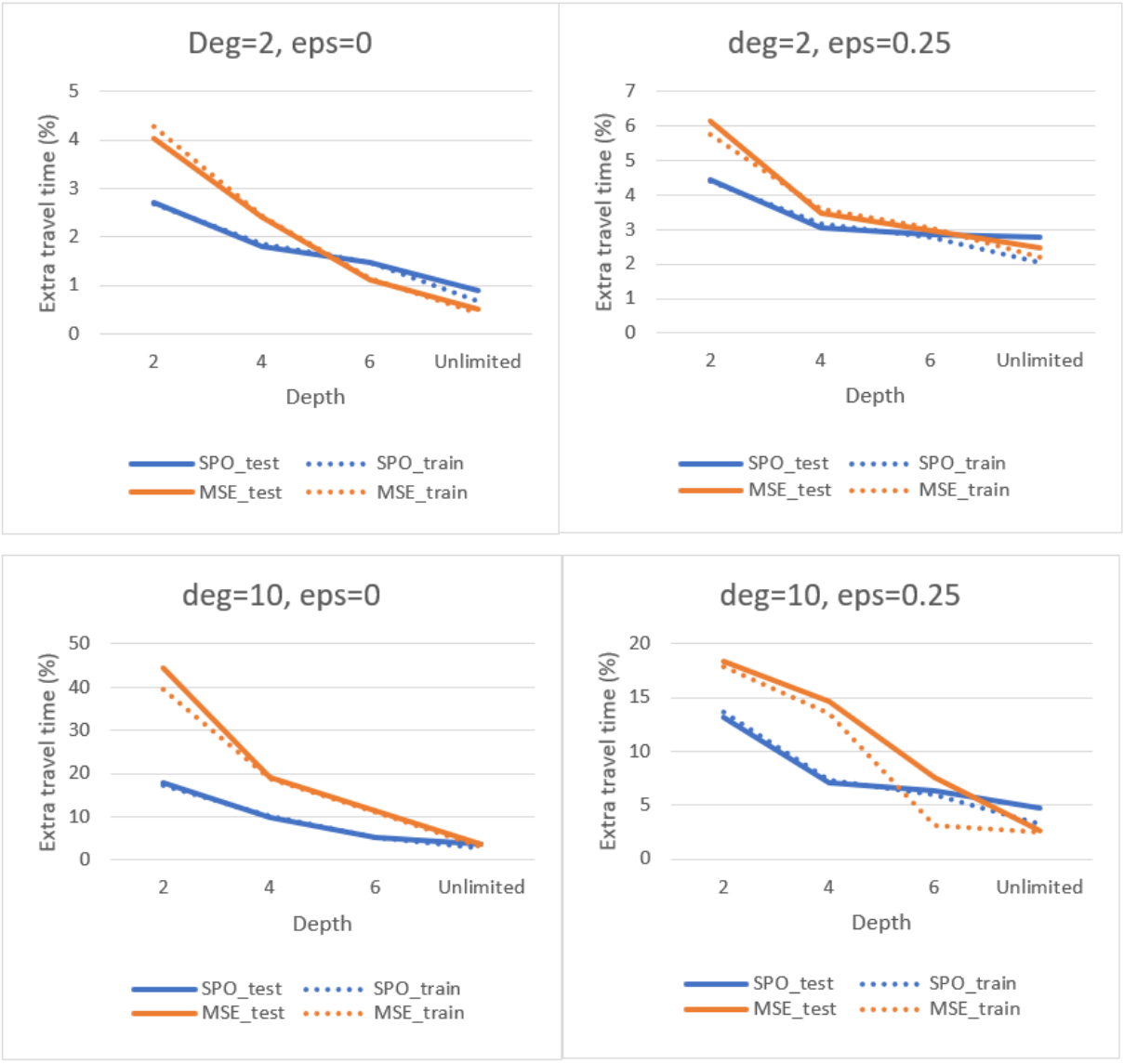


Figure 6.2: Extra travel time vs Depth (n_train=10000).

In Figure 6.2, we can determine that the SPO model performs better than MSE, presenting lower extra travel time for small depths (2, 4). But, as the depth increases beyond 6, MSE surpasses

SPO, indicating that with higher depths, MSE progressively achieves significantly accurate predictions, leading to nearly optimal decisions. However, this enhanced predictive capability comes at the cost of interpretability, as the trees grow to a size that becomes challenging to visualize and interpret easily.

Also, we realize that both greedy trees fit and generalize better the data, when working with larger training sets. Here, on the observations, we can find that the difference between their training and test predictions is small and it is lower when comparing trees fitted with 200 observations in the previous experiments.

Number of leaves vs Depth

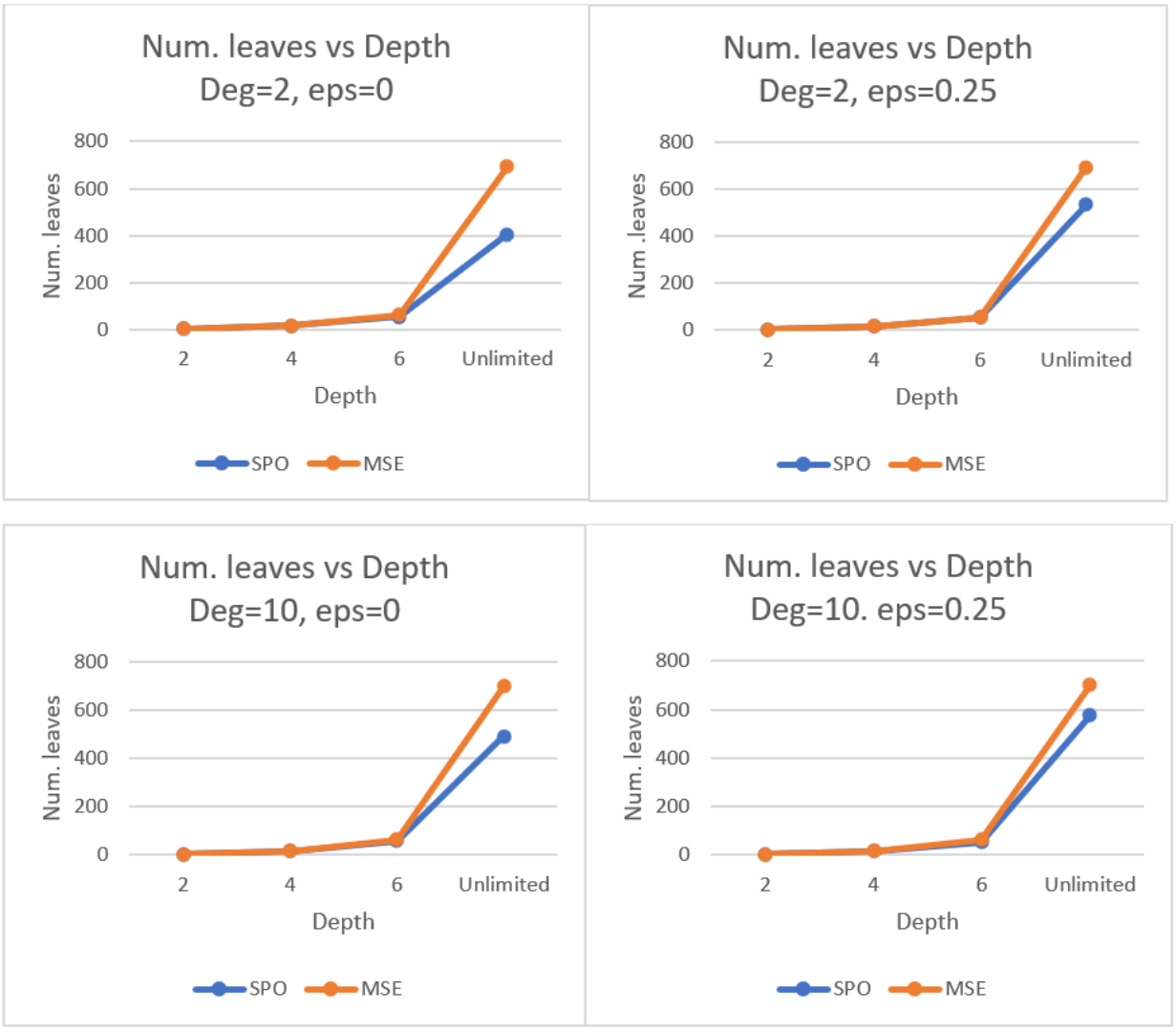


Figure 6.3: Number of leaves vs Depth for 1 dataset (n_train=10000, n_test=1000).

In Figure 6.3, we analyze the number of leaves of both models when fitted with SPO and MSE loss functions for the same larger datasets. As seen before, MSE needs more complexity in its tree structure to perform good decision errors. In this context, MSE requires more leaves, especially from depth 6 onward, to achieve comparable or even superior results compared to SPO trees. We note that for the case of unlimited depth, specially for noisy datasets, SPOTs require more leaves than a tree fitted with non-noisy data; however MSE trees continue producing a greater number of leaves, thus achieving more complexity in terms of interpretability.

Computational time vs Depth

Depth	SPO (sec)	MSE (sec)
1	340.59	376.70
2	503.37	594.31
3	620.96	690.59
Unlimited	1289.19	2616.87

Table 6.2: Computational time vs Depth for larger datasets (n_train=10000).

Table 6.2 shows the computational time during the model fitting process for the two tree models when using larger datasets. This experiment was performed over 10 datasets for each configuration to ensure a reliable average metric when evaluating the computational time. These tests were conducted specifically in scenarios with eps=0 and deg=10, and similar computational times were observed across other dataset configurations.

The results indicate comparable computational performance between the two tree models when employing small depths. However, as the depth increases, a divergence in computational time emerges. MSE starts to exhibit increased computational time, and then, when setting no depth constraints, it consumes roughly double the time compared to SPO. Although MSE models with unlimited depths showcase competitive or slightly improved decision error rates compared to SPOTs, this advantage comes at the expense of structural complexity. Furthermore, these deeper MSE trees tend to become less easy to interpret.

6.1.3 Evaluating DL8.5 against greedy trees

In this subsection, we introduce the performance analysis of DL8.5 against greedy trees, and the depth variations will range from 1 to 4 due to the long time it takes to work with depth 4 when using DL8.5 with SPO. The analysis is performed on datasets of: n_train=200, n_test=1000.

Extra travel time vs Depth

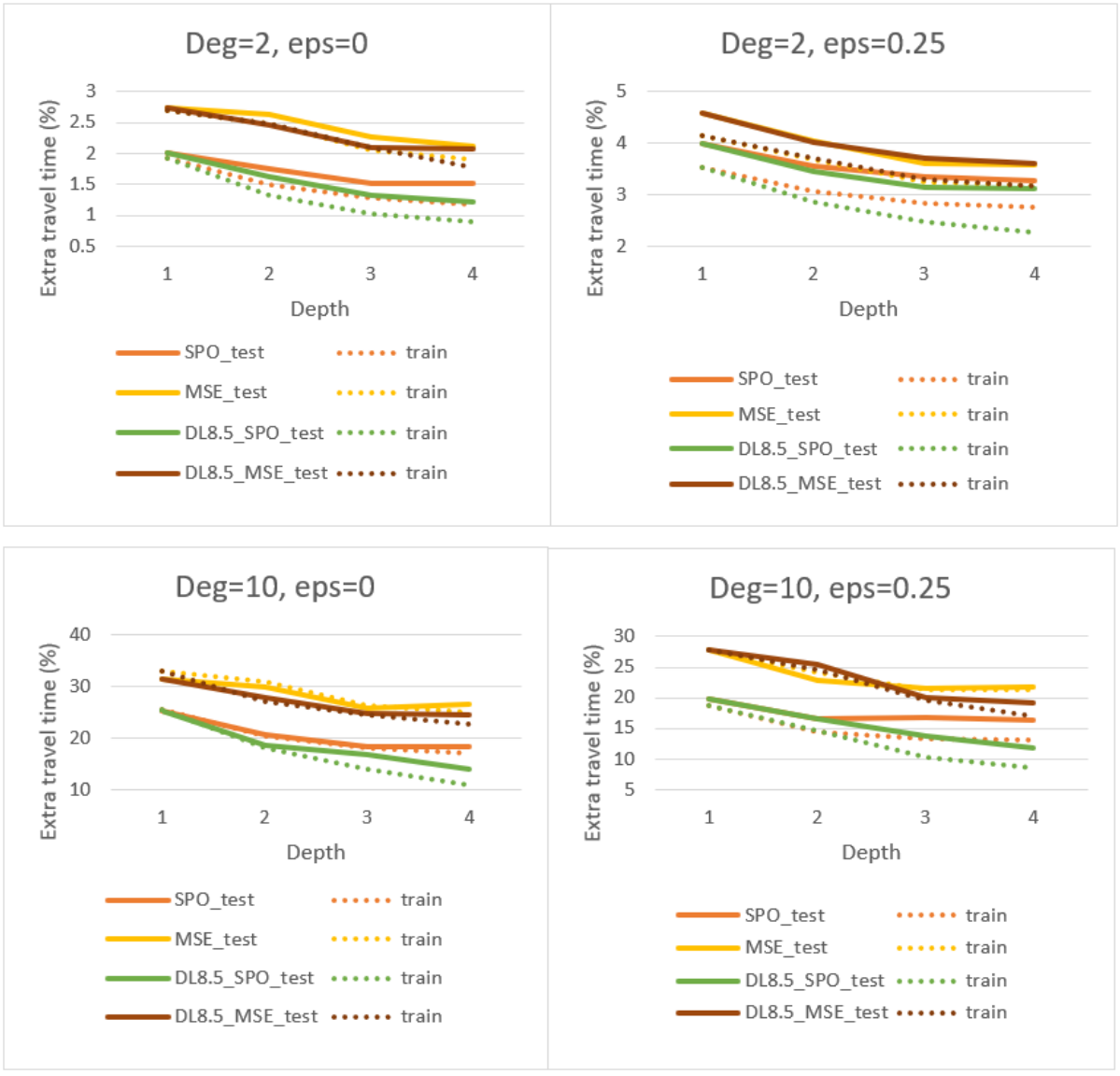


Figure 6.4: Extra travel time vs Depth for 1 dataset (n_train=200).

Figure 6.4 shows that DL8.5_SPO is performing as the best model in terms of its extra travel time for both cases: training and test sets. This model is able to find an optimal tree considering the specified depth and loss function. Meanwhile, SPO (the greedy tree) ranks as the

second-best, demonstrating superior performance compared to both MSE models. Additionally, DL8.5 with MSE exhibits slightly improved extra travel time compared to MSE (greedy) in the majority of cases.

Computational time

Depth	SPO	MSE	DL8.5_SPO	DL8.5_MSE
1	0.29	0.32	0.49	0.07
2	0.53	0.55	6.26	0.53
3	0.73	0.74	46.59	4.93
4	1.10	0.94	248.06	34.42

Table 6.3: Comparing computational time (seconds) between greedy and DL8.5 models (n_train=200).

This table compares the computational time required for fitting these algorithms under SPO and MSE loss functions. As the depth increases, SPO and MSE trees demonstrate relatively similar and efficient computational times. However, in the case of DL8.5 models, as the depth increases, the computational time becomes more pronounced. The difference in computational time between consecutive depths notably increases, especially apparent for DL8.5 SPO, with a significant jump observed at depth 4. Furthermore, DL8.5_MSE displays very good performance for smaller depths (1,2), but it starts to consume more time compared to greedy trees for higher depths.

The DL8.5 approach requires considerable time as the depth augments because it explores the complete decision space, and every feasible combination of features (given depth and min_sup parameters). Moreover, when employing the SPO loss, it must calculate the shortest path for each of these combinations, resulting in a considerable slowdown in its execution. This scenario reflects a relatively small scale, but it foreshadows larger differences in subsequent experiments when dealing with higher complexity.

6.1.4 Evaluating DL8.5 against greedy trees: 16x16 grid

Once the DL8.5 models have been introduced, let's evaluate them against the greedy trees on a higher complexity 16x16 grid.

Extra travel time vs Depth

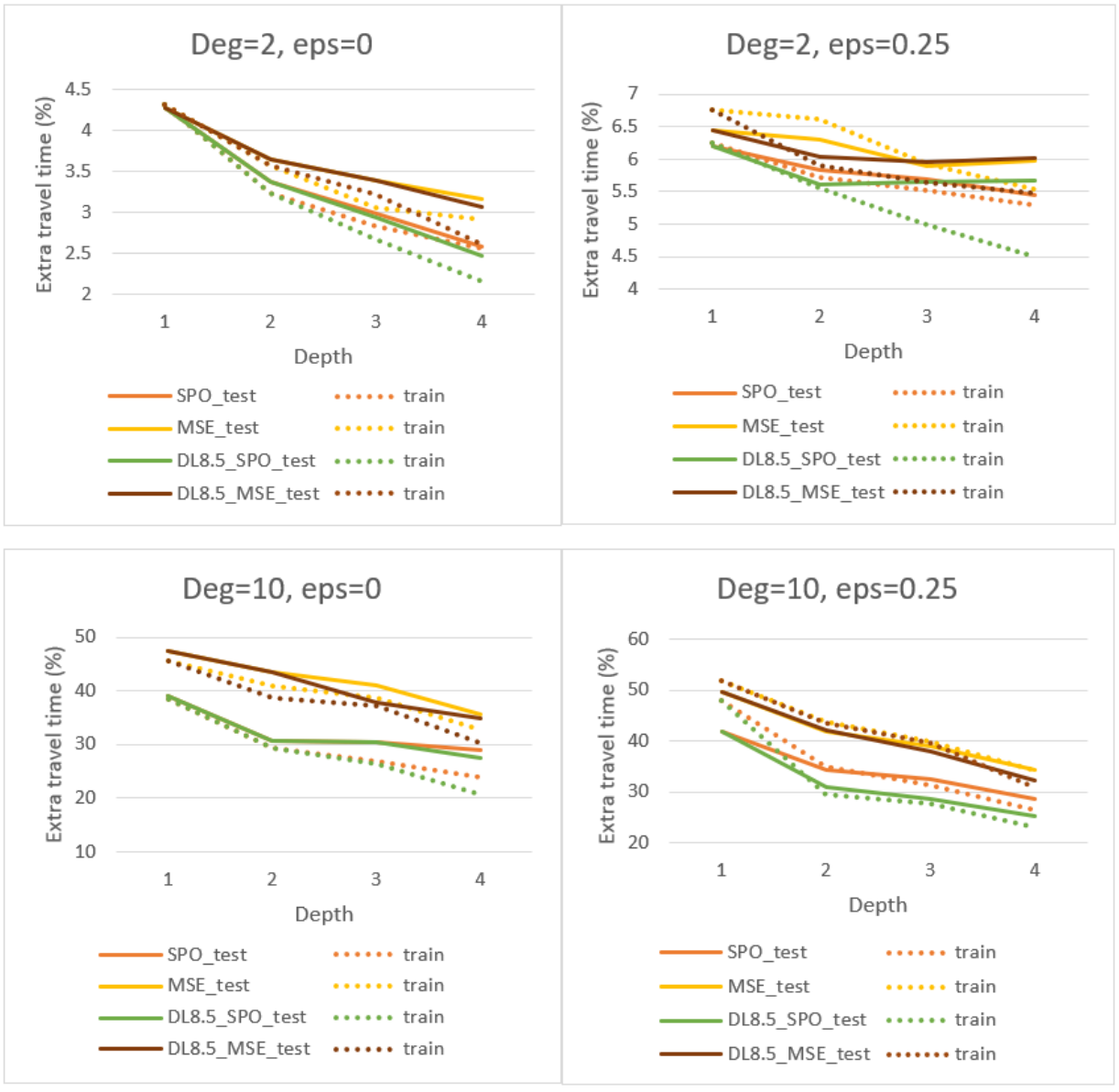


Figure 6.5: Extra travel time vs Depth (DL8.5 vs greedy algorithms) on a 16x16 grid.

Figure 6.5 displays a comparison of these models on a 16x16 grid, with respect to their extra travel time while varying the tree depth. Observations indicate that DL8.5_SPO outperforms the SPO tree with better decision errors, and similarly, DL8.5_MSE tends to surpass the MSE trees

for both training and test sets, especially for the training set due to the optimal decision trees found by DL8.5. Notably, when dealing with noisy data for both degrees, the difference in error becomes more pronounced and is particularly evident on the training set when contrasting a greedy algorithm against its DL8.5 version, where the latter exhibits a better extra travel time. However, when working with non-noisy data, the difference in error is minimal. Furthermore, it's worth noting that overall, the extra travel time tends to be higher, when operating on a 16x16 grid compared to a 4x4 scenario because of the incremented complexity.

Computational time

Depth	SPO	MSE	DL8.5_SPO	DL8.5_MSE
1	2.25	2.20	5.99	0.97
2	4.01	4.18	91.16	12.26
3	5.26	5.28	757.63	126.05
4	6.16	6.02	3936.02	741.05

Table 6.4: Computational time (seconds) vs Depth on a 16x16 grid (n_train=200).

Table 6.4 showcases the computational time for fitting greedy and optimal decision trees using SPO and MSE loss functions when varying the tree depth from 1 to 4. Consistently, DL8.5 decision trees exhibit notably higher time consumption during the model fitting process, particularly evident in DL8.5_SPO, showcasing significant spikes in computational time as the depth increases. Following this trend, DL8.5_MSE is the second model consuming more time, and it seems to perform well for smaller depths: depth 1 (the best case) and also for depth 2. Finally, SPO and MSE greedy trees emerge as the top performers, presenting very similar results and the best time performance on a 16x16 grid.

6.2 Traveling Salesman Problem

For the TSP experiments, the analyses will only focus on a 4x4 grid scenario. Concerning the datasets, only two datasets were employed due to computational time and resource limitations associated with exact algorithms like dynamic programming-based models. Each different dataset configuration presented (variations in epsilon and degree) has its own corresponding dataset 1 and dataset 2, meaning that two versions exist for each configuration, and they are the same datasets used in the shortest path problem.

TSP algorithms

Next, the TSP algorithms under study are listed below, along with their respective labels used for the experiments.

- Regarding the SPO trees, the TSP methods adapted for calculating the shortest tour include:
    - Dynamic Programming (DP).
    - Local Search (LS).
    - Simulated Annealing (SA).
    - Vehicle Routing model (Routing).
    - TSP variants (one algorithm during training and another during predictions):
        - LS DP, denoting local search for the training phase and dynamic programming for finding the shortest tour based on the predicted costs.
        - SA DP, similar in nature to LS DP.
- Concerning MSE trees, the TSP methods were exclusively utilized during the prediction phase. This means that these trees were trained using MSE loss, and subsequently, a TSP algorithm was used to identify the shortest tour once the unknown costs were predicted. The variants used for these experiments are: MSE DP, MSE LS, and MSE SA.

Note: Those models that do not have composite nomenclatures mean that the same TSP method is applied for both training phase and predictions. This notation only applies to the SPO approach. Finally, regarding DL8.5 decision trees, only the SPO loss will be considered, and details will be explained further in its respective section.

In this section, we outline the experiments conducted, which involve assessing different TSP policies applied to both greedy and optimal models. Initially, we'll examine the extra travel time of the greedy models at depth 3. Subsequently, an analysis of these models will be conducted, varying the tree depth while evaluating them on the same metric. Finally, the DL8.5 models (SPO) will be evaluated against the greedy algorithms to address the TSP problem in terms of extra travel time and computational time.

Before delving into this analysis, a preliminary analysis is performed on the decision performance of 2-opt and PS6 schemes concerning LS and SA algorithms.

6.2.1 Comparing 2-opt and PS6 perturbation schemas

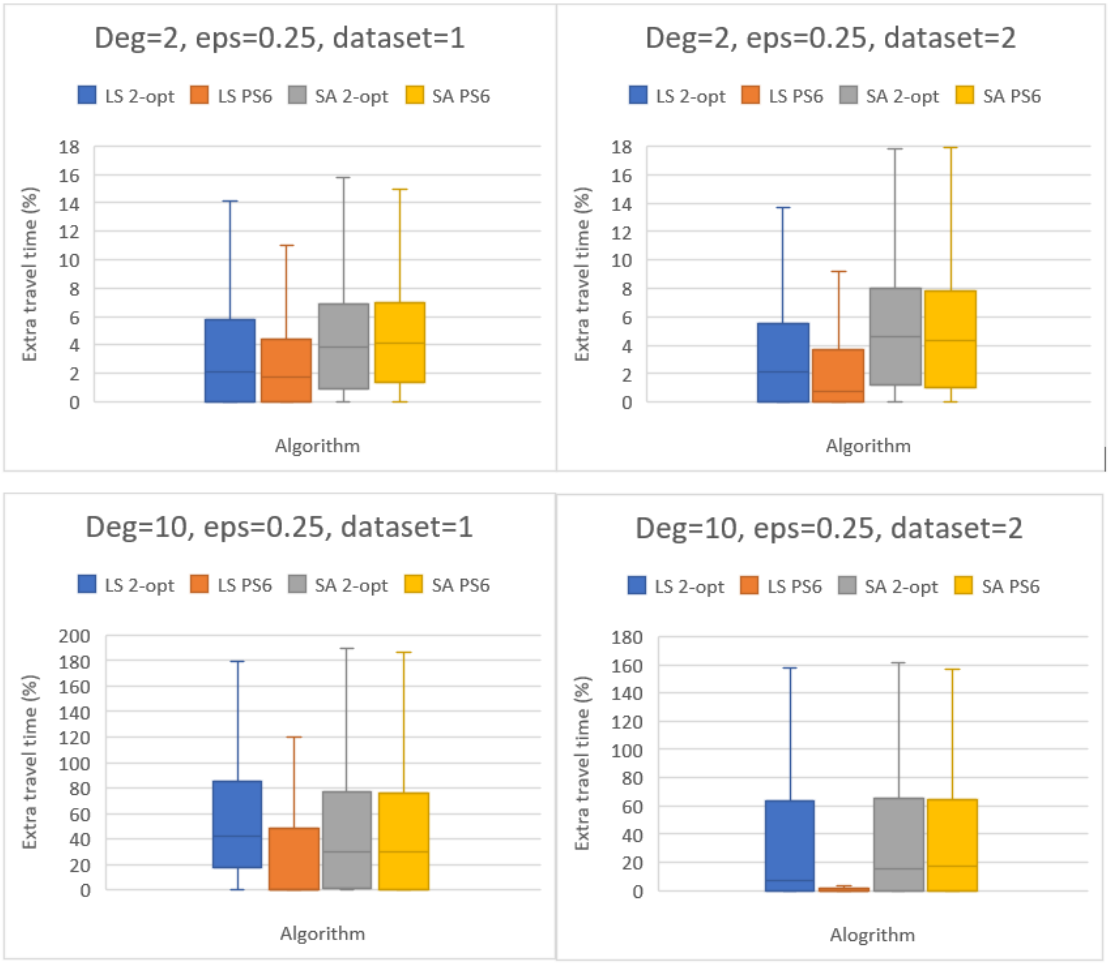


Figure 6.6: Local search and Simulated Annealing with 2-opt and PS6 schemas.

In Figure 6.6, we examine the impact of employing two different neighborhood schemas, where 2-opt is the default schema for the local search and simulated annealing algorithms, and PS6 is the other neighborhood schema explained in the previous chapter. Our observations indicate that employing the PS6 schema in the LS model notably enhances its performance, particularly notable in noisy datasets. This improvement is more pronounced in datasets of degree 10, and there's also a visible impact on datasets of degree 2. While for the Simulated annealing model, PS6 schema behaves very similarly to that of 2-opt for all datasets.

Observations

Something to highlight is that when employing local search or greedy search algorithms using various Python libraries (e.g., python_tsp, tsp_solver), issues arise when dealing with graphs that are not entirely connected, while for the other algorithms like DP (exact algorithm) and Routing, their implementations are always capable of finding a complete solution (decision vector). In our scenario, where there are no connections between 2 nodes, this was denoted by a considerable distance of 25k in the distance matrix, along with testing other larger distances. Even implementing that, the 2-opt algorithms managed to identify 13-15 edges for several cases instead of the expected 16, whereas PS6 yielded improved results, such as identifying 15-16 edges on the 4x4 grid.

Concerning these missing edges, a penalization of 25k multiplied by the number of missing edges was added on the total tour cost for the training phase as well as when predicting the shortest tour for new observations. Also, a maximum of 10 iterations were executed if these policies failed to discover the 16 edges, and if a complete decision vector was found in one of these iterations, that solution was immediately chosen; otherwise the penalisation was applied.

Hence, one of the compelling reasons to incorporate the perturbation schema (PS6) in our upcoming experiments is its promising performance based on the obtained results. Consequently, we will utilize and adopt the **PS6** schema for the next sections.

6.2.2 Extra travel time vs Algorithm

Below, the assessment of the TSP algorithms with respect to their extra travel time metric on the test set is presented, using two datasets for each configuration.

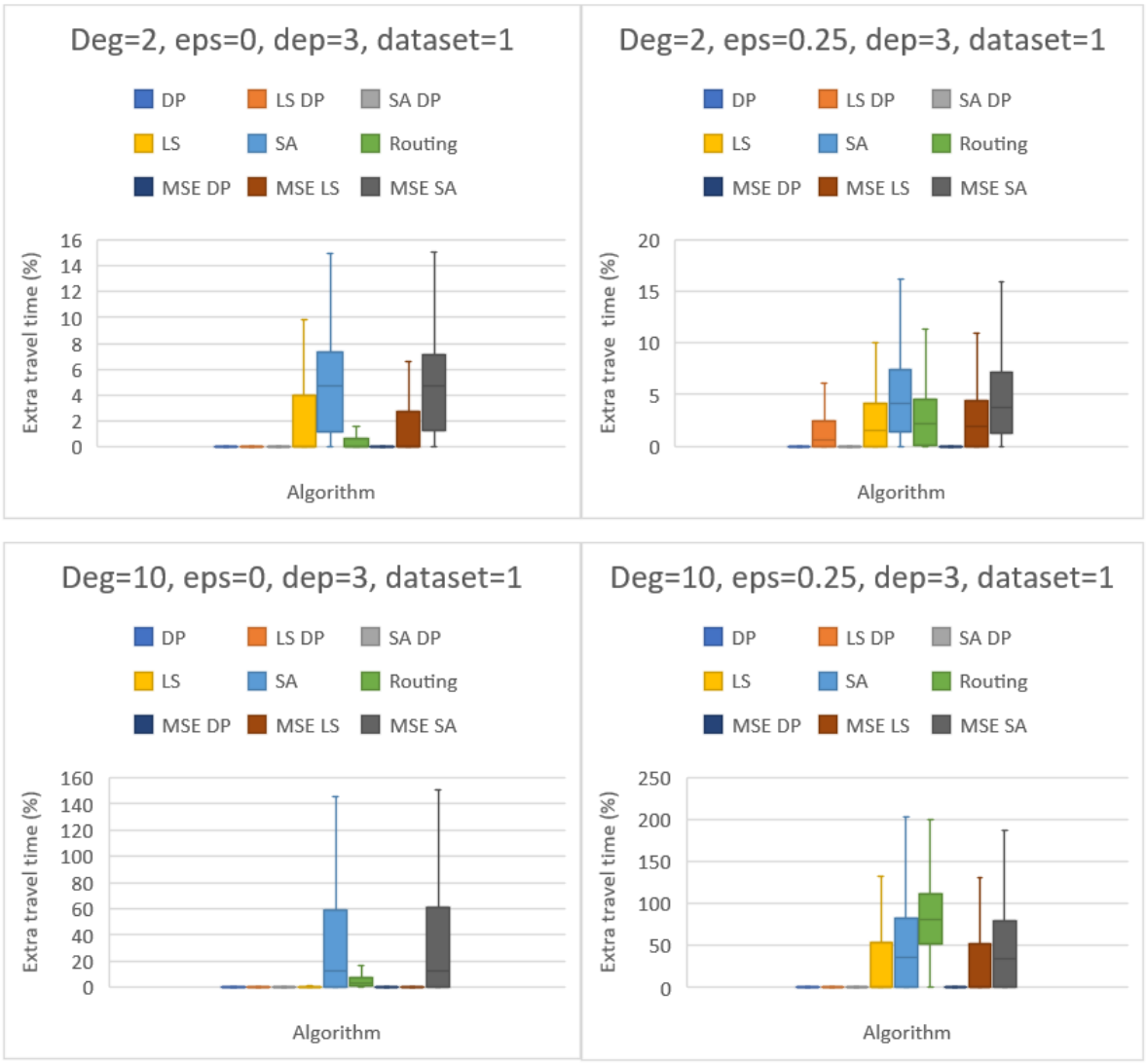


Figure 6.7: Extra travel time vs Algorithm (dataset 1).

In Figure 6.7, we evaluate the extra travel time across the different TSP variants when using SPO and MSE trees with a maximum depth of 3 for one dataset. We note that when working with datasets of degree 2 and 10 with eps=0, all models in general exhibit a lower decision error compared to datasets with eps=0.25 (noisy).

Regarding the non-noisy datasets, dynamic programming-based models (DP and DP variants) are the best performers, followed by the Routing model which presents a lower extra travel time against the LS-based policies (LS, MSE LS), especially for degree 2; while for higher degree, LS models are a bit better. Finally, SA and MSE SA are the ones showcasing the highest error.

However, when dealing with noisy datasets (eps=0.25), the most effective algorithms are DP-based models, and then LS-based competitors. Surprisingly, the Routing model, which previously performed well, now displays the lowest result for the dataset of degree 10, while still

showing acceptable results for the 2-degree dataset. Furthermore, SA-based models (SA, MSE SA) continue to behave as one of the worst competitors.

Something to highlight is that the Routing algorithm yields very good results and the time consumption is the lowest one among the SPO models.

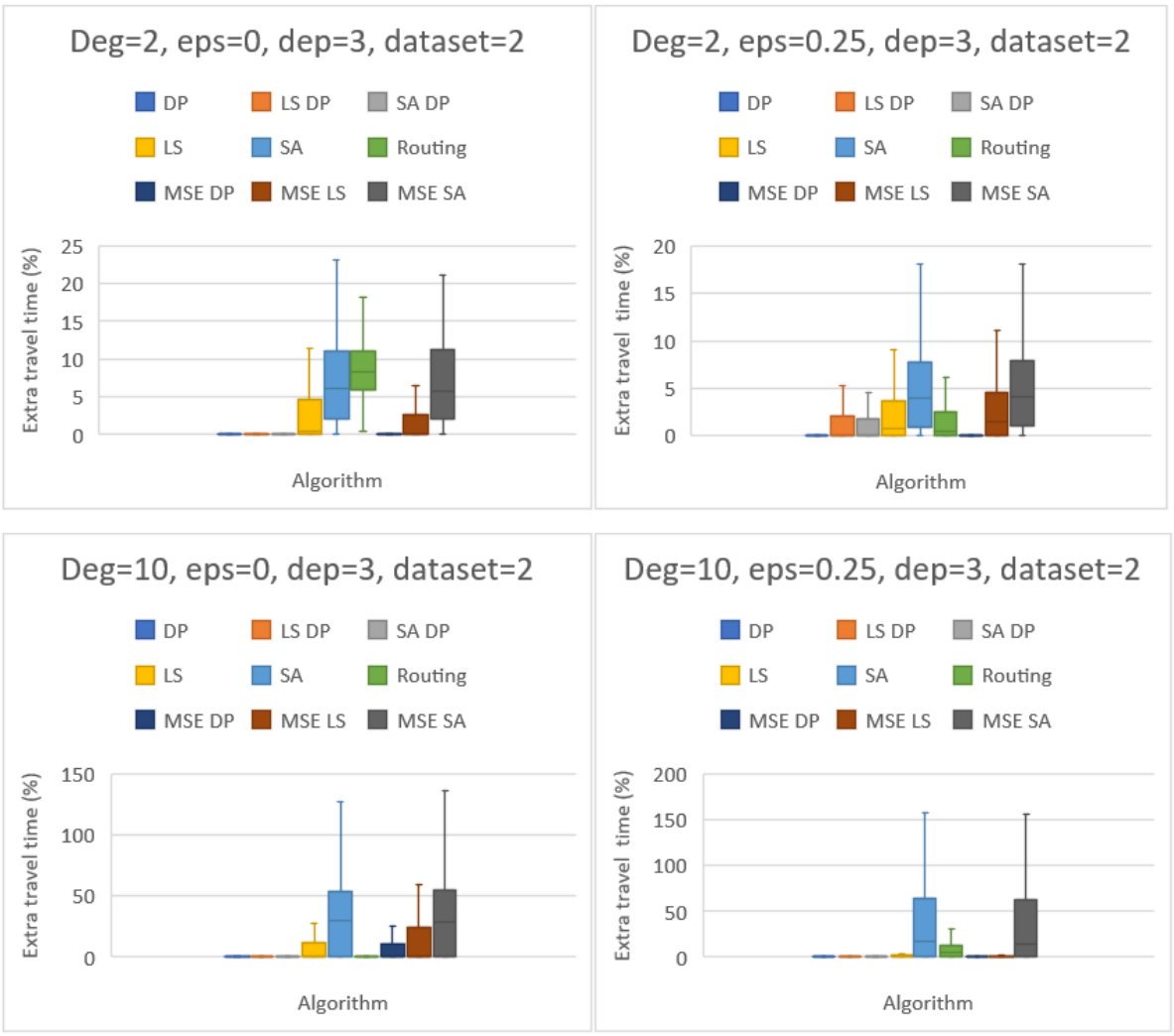


Figure 6.8: Extra travel time vs Algorithm (dataset 2)

In Figure 6.8, the extra travel time is now performed over a new dataset. First, DP-based policies continue to be the top-performing competitors, even though DP LS and DP SA augmented their decision error for the case [deg=2, eps=0.25]. Then, the Routing model is presenting a very close decision error when compared with the DP variants in most cases, even for the case [deg=10, eps=0.25], where in the previous dataset for this configuration, it was the worst performer. Additionally, LS-based competitors exhibit superior performance compared to both SA algorithms, which consistently rank as the worst-performing TSP variants.

6.2.3 Extra travel time vs Depth

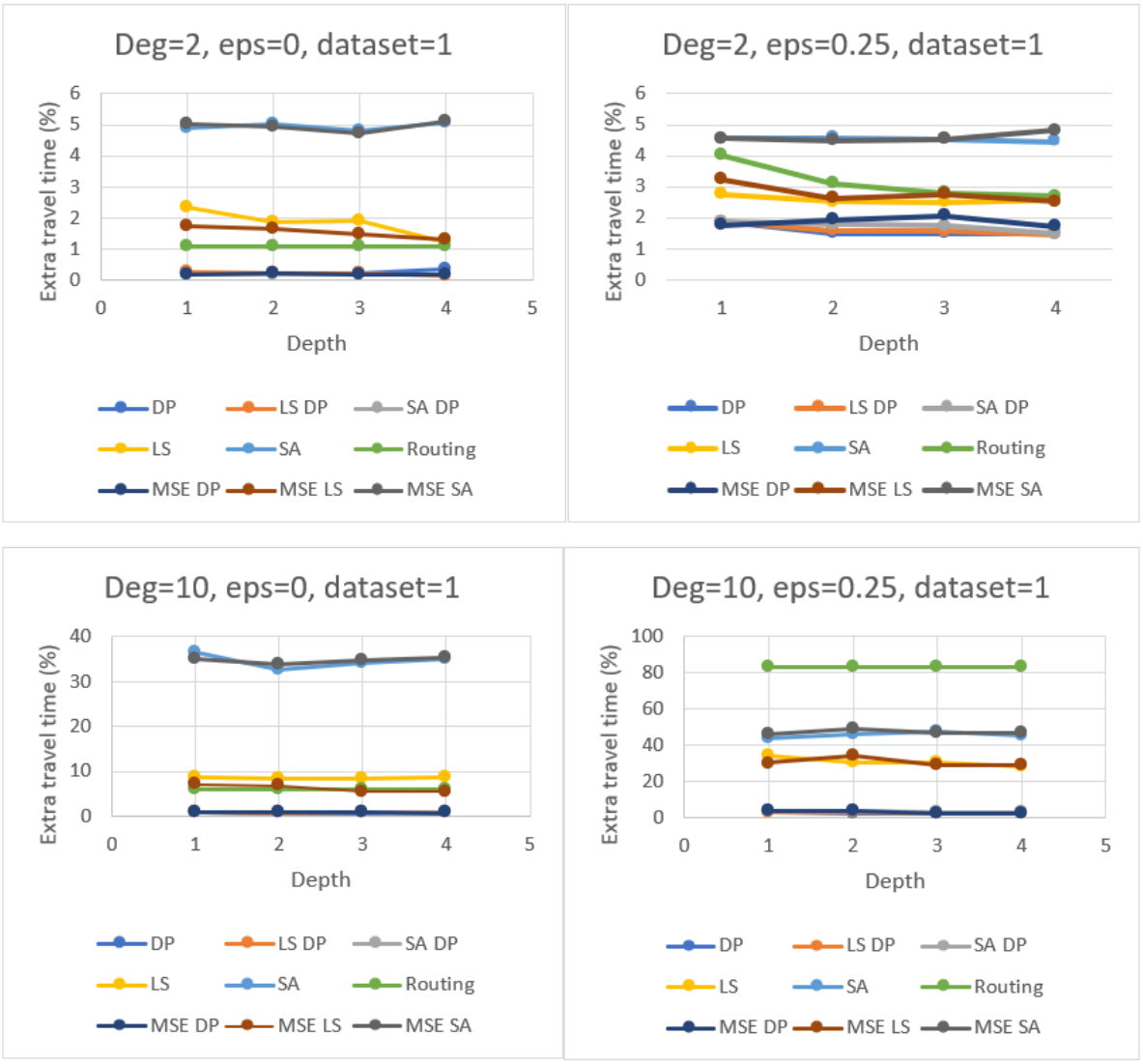


Figure 6.9: Extra travel time vs Depth (dataset 1)

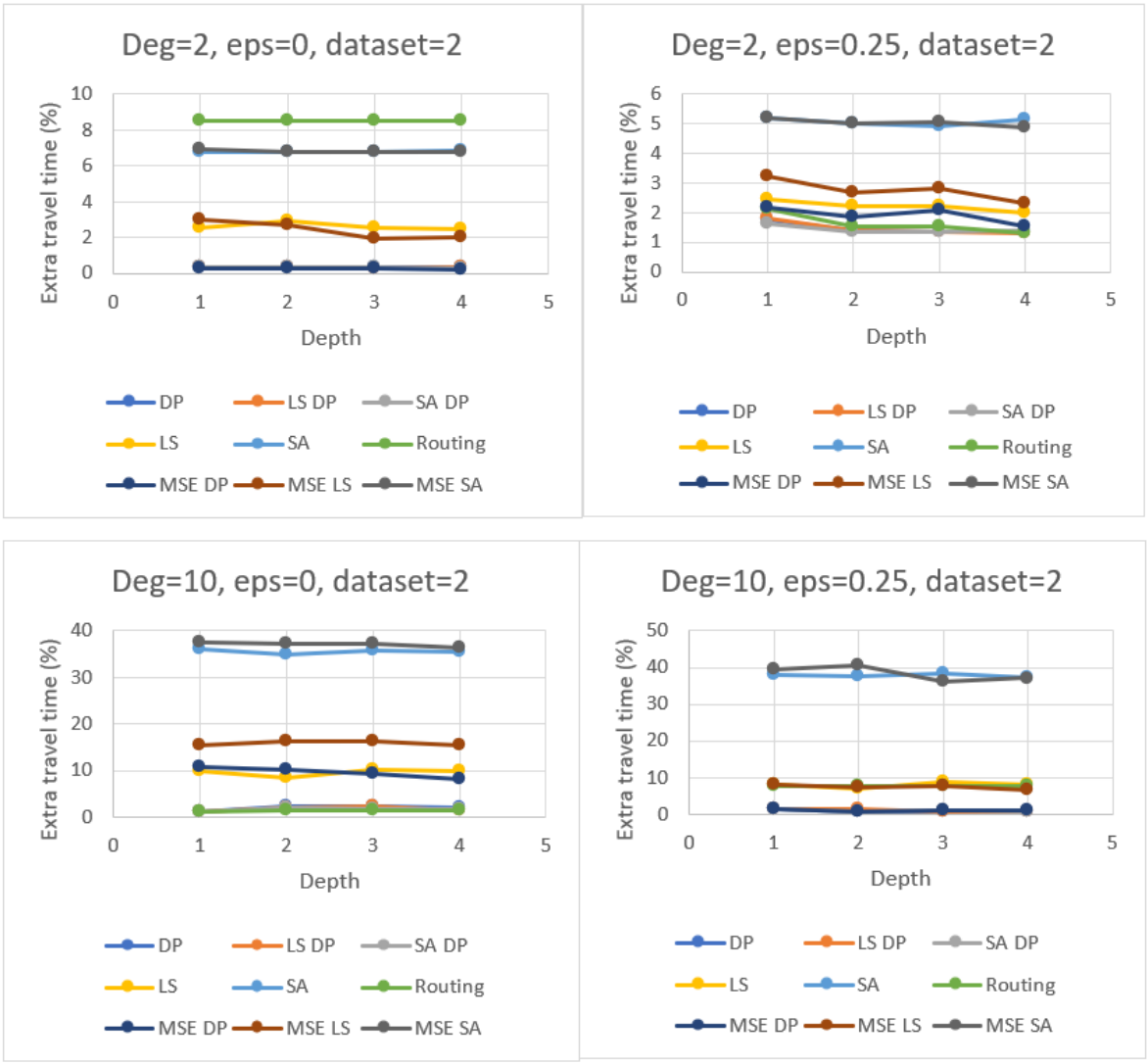


Figure 6.10: Extra travel time vs Depth (dataset 2).

In Figures 6.9 and 6.10, the extra travel time performance is evaluated when varying the depth of the tree from 1 to 4, where each point represents the mean extra travel time over the test set. We can determine that DP-based algorithms are the best competitors, then LS-based models and routing model, both present competitive results, and in most scenarios both SA models present the worst extra travel time.

Regarding the Routing algorithm, it demonstrates a good extra travel time when compared to DP algorithms in datasets 1 (both degrees, eps=0). For datasets 2, it generally performs slightly better than LS-based policies and exhibits better performance for the noisy dataset as well. The most optimal scenarios, as observed in Figure 6.10, are [deg=2, eps=0.25] and [deg=10, eps=0], where it showcases lower extra travel time, closely resembling the performance of DP policies. However, there is an instance in both Figures where its results tend to be the worst. Despite this, overall, it remains consistently effective across various depth variations.

When comparing LS and MSE LS, the latter generally outperforms the LS algorithm in non-noisy datasets. However, for datasets with eps=0.25, LS performs better. It's noteworthy that both competitors tend to improve their extra travel time as their depth increases, and in some instances, they rank well just after the DP-based models.

6.2.4 Training set analysis for DP-based algorithms

Now, the comparison on the training set for the DP-based models is presented for the different dataset configurations. Tables are presented to better visualize the extra travel time of these algorithms, given their notable similarity when employing the DP algorithm.

Degree	Epsilon	DP	LS DP	SA DP	MSE DP
2	0	0.36560932	0.36560932	0.41530383	0.67084135
2	0.25	2.36846397	2.3798241	2.82200228	1.88286162
10	0	0.15720309	0.15854409	0.19658799	0.19658799
10	0.25	1.08054722	1.13129967	1.24697607	1.7195385

Table 6.5: Comparing the mean extra travel time of DP-based models (dataset 1).

Degree	Epsilon	DP	LS DP	SA DP	MSE DP
2	0	0.1970039	0.16097638	0.19920804	0.24238607
2	0.25	1.14349801	1.17757612	1.22582859	1.76752538
10	0	1.8320009	1.964064292	1.7429404	1.83131089
10	0.25	0.40579745	0.40579745	0.82714299	0.89463492

Table 6.6: Comparing the mean extra travel time of DP-based models (dataset 2).

It is determined that when a SPO greedy tree is trained with an exact TSP algorithm, its performance tends to be the best in most scenarios as shown in Tables 6.5, 6.6, where each value represents the mean extra travel time for the different dataset configurations. Notably, when the greedy trees are trained with a heuristic algorithm (LS or SA), the results tend to be the closest to DP model only if an exact algorithm is applied on these predicted cost vectors. Meaning that a heuristic algorithm can be useful for the training phase by speeding up the learning process, but the model can still entail considerable time consumption when using an exact algorithm for handling larger instances during the prediction phase.

Complete results on the training sets for the other TSP algorithms can be found in the Appendix B section.

6.2.5 DL8.5 with SPO loss function

Now, we are going to focus the analysis on the performance of the SPO and DL8.5 SPO decision trees for solving the TSP problem. The maximum depth of the trees is set to 2 due to the considerable time consumption when working with larger depths, especially for DL8.5 SPO trees, even more for exact TSP algorithm (DP). Regarding the TSP algorithms, we will center on four of them: dynamic programming (DP), local search (LS), simulated annealing (SA), and routing model.

Extra travel time vs Algorithm: Training set

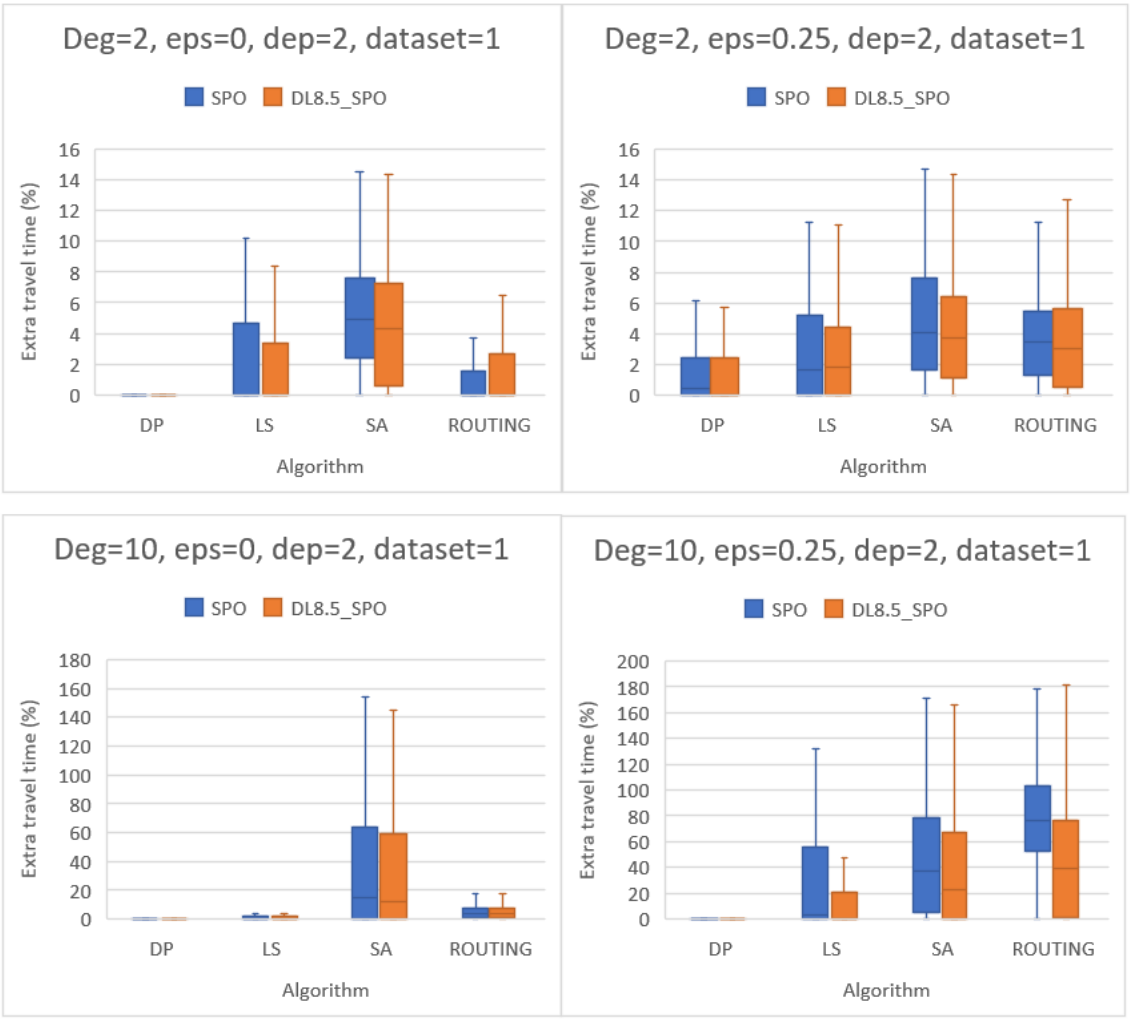


Figure 6.11: Extra travel time vs Algorithm (SPO vs DL8.5 SPO) on training set for dataset 1.

Degree	Epsilon	DP	DP_DL8.5
2	0	0.19658799	0.15349896
2	0.25	1.51295979	1.50252945
10	0	0.67069077	0.41269887
10	0.25	2.87249732	2.62246421

Table 6.7: Mean extra travel time for DP models on training set for dataset 1.

In Figure 6.11, the evaluation of the extra travel time for both SPO and DL8.5 decision trees is conducted on the training set across the different TSP variants. It's noticeable that the predictions on the training data consisting of 200 observations exhibit slightly improved results for DL8.5_SPO trees, underscoring the efficacy of optimal decision trees (DL8.5) in addressing the TSP problem. When comparing the exact algorithms (DP), both the greedy and optimal models generally perform comparably, but it can be contrasted in Table 6.7, where a little enhancement is observed in the DL8.5 version. Then, the DL8.5 versions of the LS and SA models demonstrate little improvements over their respective SPO greedy approaches. Finally, it is not the same behavior for the DL8.5 Routing model, but at least its performance is similar and acceptable compared to its greedy SPO version.

Due to the fact that the same datasets are used for this experiment, the same tendency (as in Figure 6.7) can be appreciated between the TSP competitors: DP models are at the forefront, followed by LS and Routing models that display competitive results, and lastly, the both SA competitors. This observed effect also persists when employing the DL8.5 approach.

The reason why the SPO greedy version can sometimes surpass the DL8.5_SPO tree on training data, is because of the heuristic nature of the TSP algorithms. But when considering an exact TSP method, DL8.5_SPO should always be the best performer due to the fact that it always finds the optimal tree, and that is the case. Table 6.7 shows the comparison between the two TSP exact algorithms (DP) for the mean extra travel time, where DL8.5 tends to be slightly better than its greedy version. Thus, DL8.5_SPO works well for this TSP problem.

Extra travel time vs Algorithm: Test set

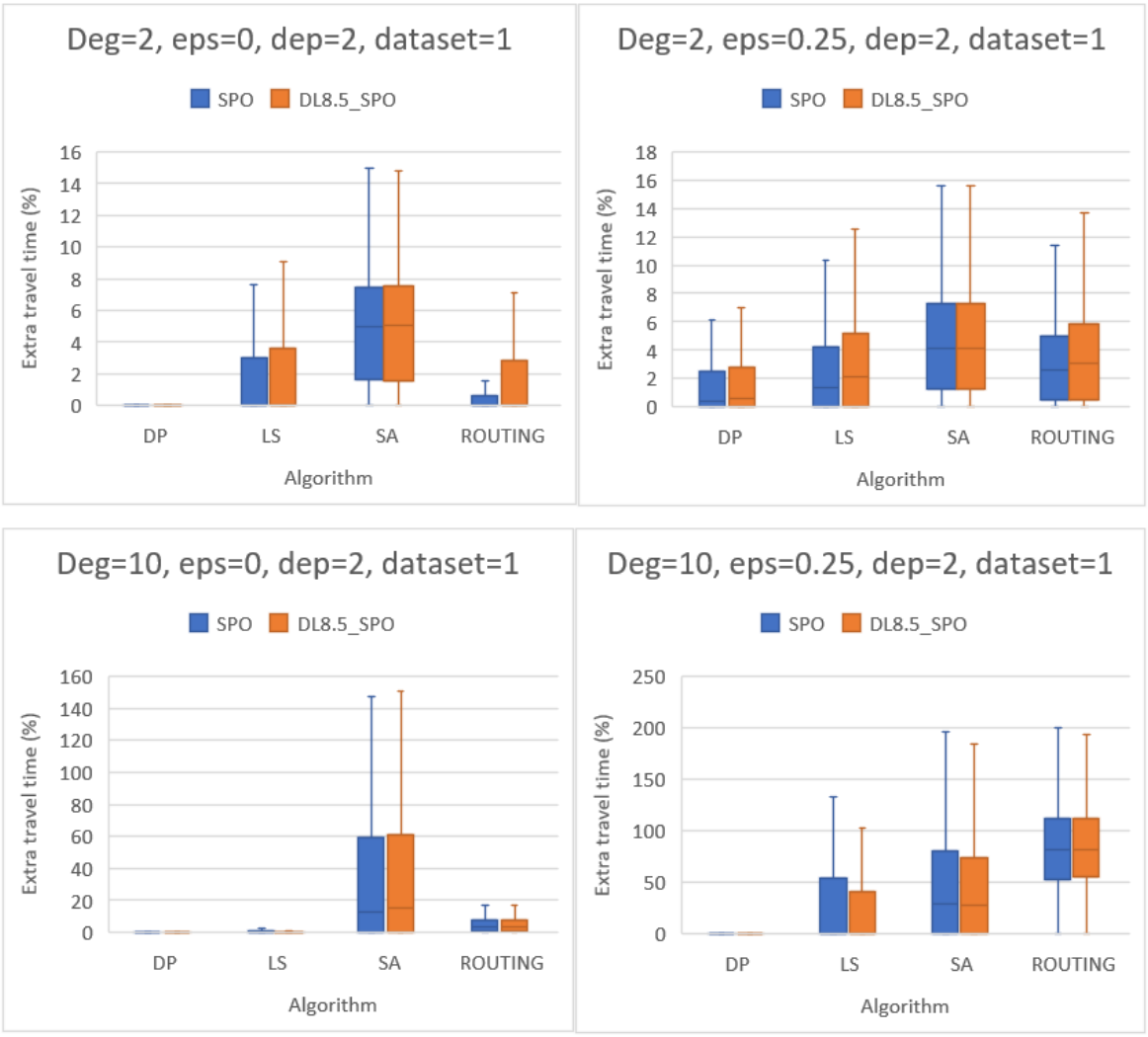


Figure 6.12: Extra travel time vs Algorithm (SPO vs DL8.5 SPO) on test for dataset 1.

Degree	Epsilon	DP	DP_DL8.5
2	0	0.20439926	0.17851543
10	0	0.73626548	0.78597859
10	0.25	2.54519192	2.07802948

Table 6.8: Mean extra travel time for DP models on test set for dataset 1.

After evaluating the extra travel time on the training set, let's now see the performance over the test set. In Figure 6.12, we note that the TSP algorithms behave very similarly on both greedy

and optimal decision trees across the four datasets. But, there is a particular case, the noisy dataset of degree 10, where DL8.5 policies outperform the greedy trees, even for the DP model where we can observe in Table 6.8 that the average extra travel time of DP_DL8.5 is a bit lower than its greedy version (2.078 vs 2.545). Furthermore, the extra travel time of SA is better than the Routing model on both tree approaches in the same particular dataset (the only one), as seen before in previous charts for the greedy approach.

Additionally, a table containing the mean error is attached to better appreciate the extra travel time difference between the DP algorithms (SPO and DL8.5_SPO), for the cases where the boxplots of this exact algorithm turn around 0. In Table 6.8, we appreciate that DP_DL8.5 performs a bit better than the DP greedy approach in most cases, even if the difference is minimal.

Computational time

Depth	Model	DP (sec)	LS (sec)	SA (sec)	Routing (sec)
1	SPO	209.89	3.52	5.86	0.95
	DL8.5_SPO	191.23	2.01	3.65	0.60
2	SPO	476.51	7.26	12.77	1.42
	DL8.5_SPO	4599.26	46.41	84.41	11.19

Table 6.9: Comparison of computational time of SPOTs and DL8.5 decision trees.

In the provided Table 6.9, we're examining the performance of SPO trees against the optimal SPO trees (DL8.5_SPO) across the four TSP competitors. This analysis focuses on depths 1 and 2 due to long time execution related to TSP exact algorithms when moving to depth 3, especially with DL8.5 trees. At depth 1, DL8.5_SPO models are slightly better than their SPO greedy version. However, when the depth is increased to 2, the computational time significantly rises for DL8.5 models. This increase is particularly pronounced, especially for DP DL8.5, where the difference is around 10 times the SPO greedy tree. Additionally, the Routing model is the least computationally time-consuming competitor for SPO and DL8.5_SPO, and the second best is local search for both tree approaches.

Therefore, it is demonstrated that DL8.5 can also work well with a SPO loss function for addressing the TSP problem, specially when working with DP exact algorithm, even though this implicates long time execution. Furthermore, the SPO loss was possible to be adapted to DL8.5 because it is an additive function when considered over the leaves.

Conclusion

Within the Predict-And-Optimize setting, different methodologies for training decision trees were addressed to solve the Shortest path problem and the Traveling Salesman problem (TSP). In the context of MSE loss function, decision trees are trained to minimize the prediction error. Whereas, the SPO loss intends to combine the predict and optimize stages during the training phase with the goal of minimizing decision error, meaning that the importance lies in identifying the optimal shortest path rather than the significance of the predicted costs. This research demonstrates that SPO decision trees proficiently generate higher-quality decisions while maintaining lower model complexity compared to conventional machine learning methods that minimizes prediction error.

For the Shortest path problem, we can state that SPO showcases better-quality decisions across various dataset configurations when tree depth varies, due to the fact that SPO uses the decision error in its learning process. When working with larger datasets, SPO trees present lower decision error than MSE trees, which represents an advantage for SPOTs in terms of interpretability and performance. However, when setting MSE trees to unlimited depth, they attain comparable or even superior performance to SPO. Nonetheless, achieving this level of performance requires deeper levels and a larger number of leaves, resulting in increased complexity, reduced interpretability, and higher computational requirements.

Then, the DL8.5 algorithm was introduced to tackle the shortest path problem, and it was possible to improve the extra travel time when using DL8.5 with the MSE and SPO loss functions, due to the fact that DL8.5 explores all feasible splits and selects the split having the lowest score (optimal tree). In our findings, when combining DL8.5 with MSE loss, it presents comparable results against MSE (greedy), but it tends to be a bit better for the different dataset scenarios. Then, DL8.5 with SPO loss emerged as the best performer across all test scenarios, including the scaled 16x16 grid where it was able to build an optimal decision tree for the given depth and min support in a leaf. Thus, we can determine that the DL8.5 algorithm can be combined with these loss functions due to its additive nature on the leaves, especially with SPO (best competitor).

Regarding the Traveling salesman problem (TSP), SPO and MSE trees were combined with exact and heuristic TSP algorithms for finding the shortest tour across various datasets and limited to a 4x4 grid scenario because of the considerable computational time when using an exact algorithm. Overall, DP-based algorithms generally performed better than the other heuristic models. The combination of heuristic algorithms for faster training and an exact algorithm for predictions showed potential, although with time limitations in larger cases. Comparatively, Routing and LS-based models exhibited similar performance, with Routing

presenting closer decision errors when compared to DP algorithm in some scenarios. Additionally, introducing the DL8.5 algorithm with SPO loss showcased similar test predictions to its greedy version, and a slight improvement in training predictions in most cases, and for the exact (DP) algorithm, it was determined that the DL8.5 version was able to build an optimal tree. Therefore, we can conclude that DL8.5 with SPO can also work well together for this test scenario.

Finally, this study offers promising avenues for future exploration and innovation. The adaptability of the SPO methodology beyond decision trees presents a good opportunity to extend its utility to diverse supervised machine learning tools. Moreover, extensive testing of DL8.5 in other constraint-based decision tree learning scenarios could unveil its strengths and limitations, which would help fine-tune its performance in machine learning and optimization. These directions promise new discoveries and advances in the field.

Bibliography

[1] Aglin, G., Nijssen, S., & Schaus, P. (2021, January). Pydl8. 5: a library for learning optimal decision trees. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence* (pp. 5222-5224).

[2] Aglin, G., Nijssen, S., & Schaus, P. (2020, April). Learning optimal decision trees using caching branch-and-bound search. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 34, No. 04, pp. 3146-3153).

[3] Elmachtoub, A. N., Liang, J. C. N., & McNellis, R. (2020, November). Decision trees for decision-making under the predict-then-optimize framework. In *International Conference on Machine Learning* (pp. 2858-2867). PMLR.

[4] Kevin D.Davis. (2023, October 31). Decision Tree Analysis in Project Management with Examples. URL: https://www.knowledgehut.com/blog/project-management/decision-tree-analysis-pmp. Last access: November 28, 2023.

[5] Garg, A. (2020, July 24). *Decision trees in machine learning*. Medium. URL: https://medium.com/data-science-community-srm/decision-trees-in-machine-learning-7fdaf3d09a0. Last access: November 30, 2023.

[6] Hrvoje Smolic. (2023, December 8). A Comprehensive Guide to Decision Trees: Everything You Need to Know. URL: https://graphite-note.com/a-comprehensive-guide-to-decision-trees-everything-you-need-to-know. Last access: December 2, 2023.

[7] Titanic Survival Decision Tree. (2017, September). URL: https://commons.wikimedia.org/wik i/File:Decision_Tree.jpg. Last access: December 2, 2023.

[8] Siva Balakrishnan. (2019, March 5). Mostly Classification: Review 1. URL: https://www.stat.cmu.edu/~siva/teaching/462/lec14.pdf. Last access: January 3. 2024.

[9] Marco Peixeiro. (2019, July 25). The Complete Guide to Decision Trees. Towards Data Science. https://towardsdatascience.com/the-complete-guide-to-decision-trees-17a874301448. Last access: December 15, 2023.

[10] Classification And Regression Trees for Machine Learning. (2020, August 15). Machine Learning Algorithms. URL: https://machinelearningmastery.com/classification-and-regression-trees-for-machine-learning/. Last access: December 15, 2023.

[11] CART Classification (R). URL: https://www.harshaash.com/R/CART-Classification/. Last access: December 20, 2023.

[12] Akhil Anand. (2020, December 10). Post-Pruning and Pre-Pruning in Decision Tree. Analytics Vidhya. URL: https://medium.com/analytics-vidhya/post-pruning-and-pre-pruning-in-decision-tree-561f3df73e65. Last access: December 27, 2023.

[13] Wilder, B., Dilkina, B., and Tambe, M. Melding the data decisions pipeline: Decision-focused learning for combinatorial optimization. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, pp. 1658– 1665, 2019a.

[14] Leclercq, Delphine. *DL8.5 decision tree used in a Predict-then-Optimize framework : decision tree learning using SPO loss error function.* Ecole polytechnique de Louvain, Université catholique de Louvain, 2023. Prom. : Nijssen, Siegfried.

[15] Chiradeep BasuMallick. (2022, October 19). What is Dynamic Programming? Working, Algorithms, and Examples. URL: https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/. Last access: December 12, 2023.

[16] tutorialspoint. Traveling Salesman Problem using Dynamic Programming. URL: https://www.tutorialspoint.com/design_and_analysis_of_algorithms/travelling_salesman_problem_dynamic_programming.htm. Last access: December 13, 2023.

[17] 2-opt. URL: https://en.wikipedia.org/wiki/2-opt. Last access: December 20, 2023.

[18] Fillipe Goulart. Github. URL: https://github.com/fillipe-gsm/python-tsp/blob/master/docs/solvers.rst. Last access: January 3, 2024.

[19] Francis Allanah. (2022, January 31). URL: https://medium.com/@francis.allanah/travelling-salesman-problem-using-simulated-annealing-f547a71ab3c6. Last access: December 20, 2023.

[20] Google OR-Tools. Traveling Salesperson Problem. URL: https://developers.google.com/optimization/routing/tsp. Last access: January 4, 2024.

[21] Djillali Boutouili. (2023, October 18). Optimization Model Types: Full Guide From Linear to Non-Linear. URL: https://theblogsail.com/technology/optimization/optimization-model-types/. Last access: January 4, 2024.

[22] Austin Buchanan. Shortest Path Problem LP formulation. URL: https://github.com/AustinLBuchanan/Combinatorial-Optimization-in-Gurobi/blob/main/shortest-path.ipynb. Last access: November 30, 2023.

[23] SPOTree. https://github.com/rtm2130/SPOTree/tree/master/Applications. Last access: October 5, 2023.

[24] Travelling salesman problem. URL: https://en.wikipedia.org/wiki/Travelling_salesman_problem. Last access: November 25, 2023.

Appendices

A. The shortest path problem

A.1 Extra travel time vs Training Size

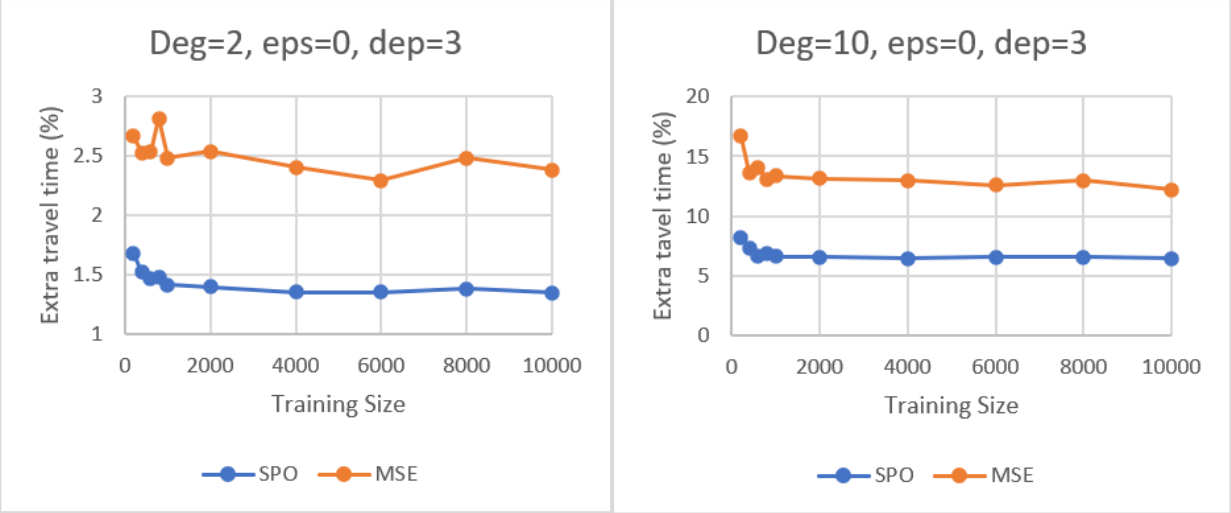


Figure A.1: Extra travel time vs Training Size: degree 2 (left), degree 10 (right)

In this experiment, the extra travel time for both the SPO and MSE models is assessed by varying the training set size from 200 to 1k, and subsequently from 1k to 10k. Each point on the charts represents an average error of 10 datasets with the respective configuration, and the test set size is 1000 observations. In Figure A.1 (left), featuring a dataset configuration of degree 2, epsilon 0, and depth 3, SPO consistently outperforms MSE with better quality decisions across all training set sizes. Notably, its error decreases as the training size increases (for small depths); however, the improvement plateaus beyond a size of 1-2k. Conversely, in the right chart (10-degree datasets), SPO has a better performance across all the different training set sizes with a difference between 5-8% in the extra travel time metric against MSE.

B Traveling salesman problem (TSP)

B.1 Extra travel time vs Algorithm: Training set

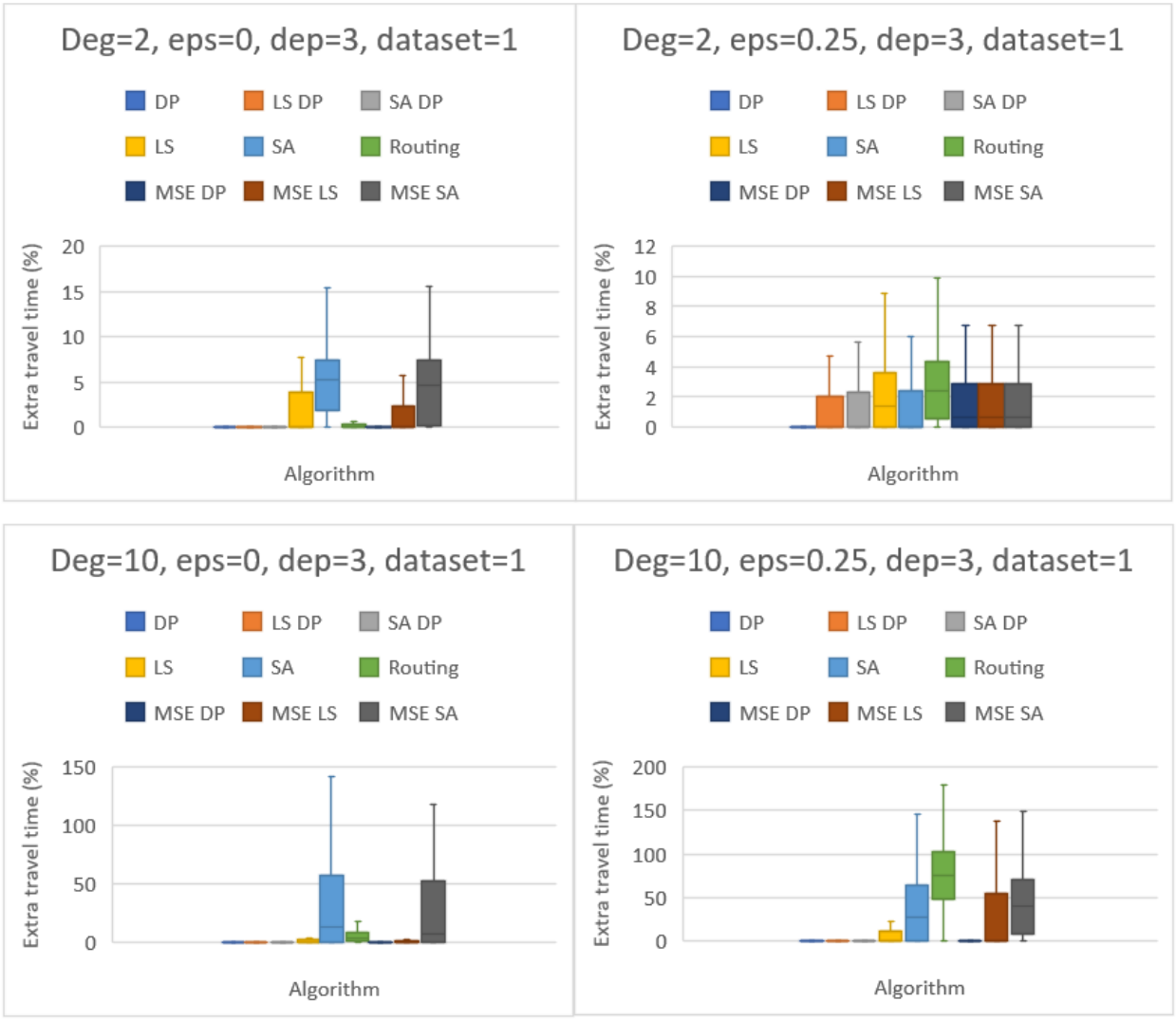


Figure B.1: Extra travel time vs Algorithm for training set (dataset 1).

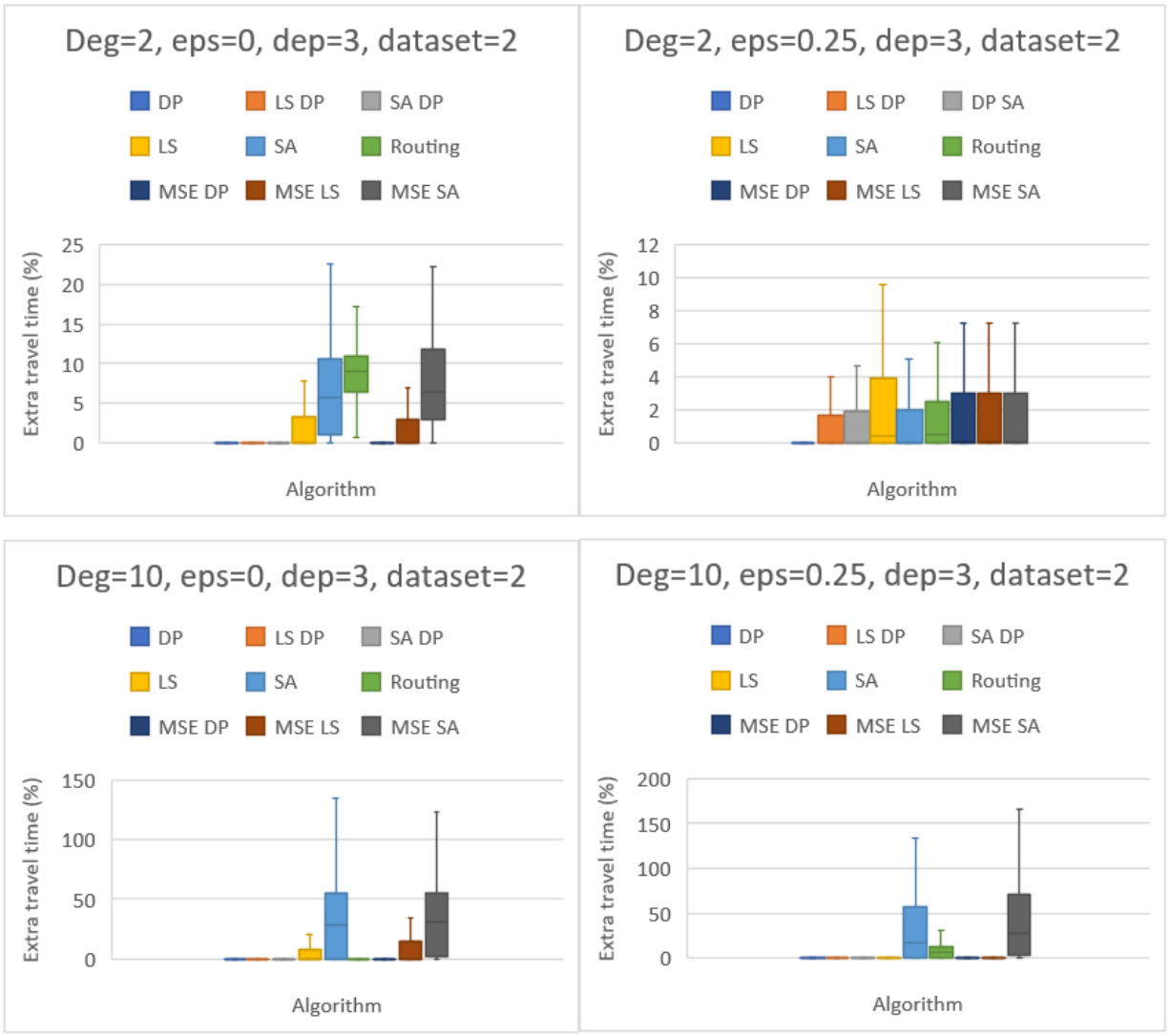


Figure B.2: Extra travel time vs Algorithm for training set (dataset 2).

As shown in Figures B.1 and B.2, the extra travel time metric of the TSP algorithms trained under the SPO and MSE loss, is in general better when making predictions over the training data in comparison to the test data (Experiments chapter). This trend in performance persists: DP-based models consistently emerge as the top-performing methods, and the routing model generally outperforms local search-based models. However, when dealing with noisy data, the routing model's performance becomes comparable or slightly inferior to that of LS-based algorithms.

Concerning the SA algorithm, it displays the poorest performance when operating with non-noisy data. However, surprisingly, when dealing with noisy data (epsilon 0.25), this algorithm showcases competitive results against local search models and slightly outperforms the routing algorithm for datasets of degree 2. This contrast in performance wasn't observed when working with the test data.

B.2 DL8.5 with SPO loss function

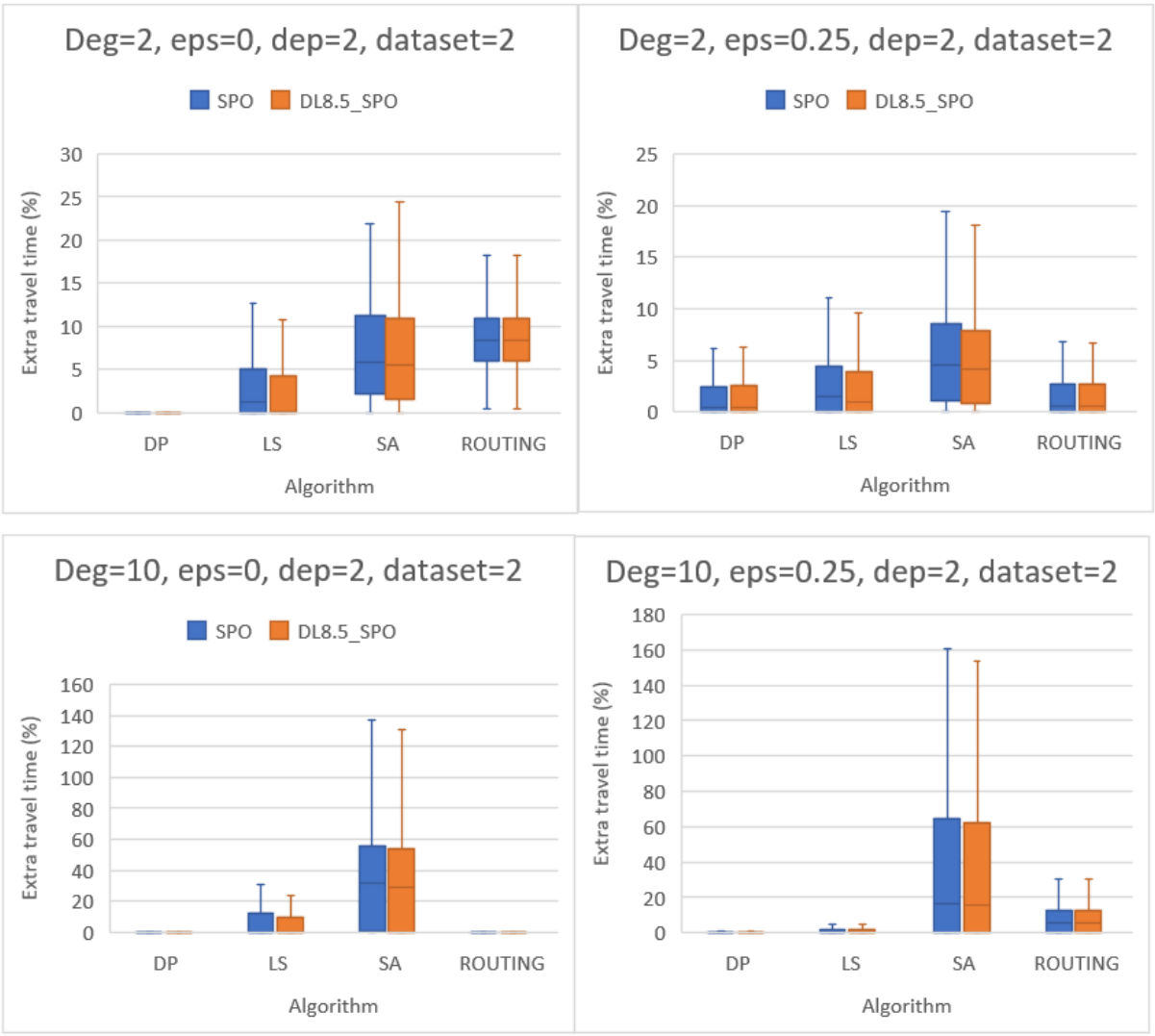


Figure B.3: Extra travel time vs Algorithm (SPO vs DL8.5 SPO) on test set for dataset 2.

Degree	Epsilon	LS	LS_DL8.5	SA	SA_DL8.5
2	0	2.84253629	2.39195433	6.85025656	6.5897455
2	0.25	2.72429567	2.40926513	5.38867422	4.93687869
10	0	11.0880716	9.22450992	37.0527975	35.040561
10	0.25	7.00523656	7.92699432	37.5029452	37.1093104

Table B.1: Mean extra travel time for LS and SA models for dataset 2.

Figure B.3 illustrates the extra travel time evaluated on the test set on a new dataset (2). It is observed that both decision tree approaches (greedy and DL8.5) when fitting them with SPO, tend to perform almost equally for a depth of 2. However, for the LS and SA models, there is a minimum improvement on the extra travel time in the DL8.5 approach against the greedy one, and it could be more appreciated when observing their mean value in most cases for these algorithms in Table B.1. Regarding the DP and Routing models, both are presenting lower decision errors than the other algorithms, and when comparing their SPO tree approach with the DL8.5_SPO one, both exhibit nearly identical performance. DP remains the top performer due to its reliance on an exact algorithm rather than a heuristic nature, but it's more time-consuming.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**