Decision Diagram Focused Learning

by

Jop Schaap

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday January 29, 2025 at 12:15 PM.

Project duration:	April 16, 2024 – January 29, 2025	
Thesis committee:	Prof. Dr. M. M. de Weerdt,	TU Delft, supervisor
	Ir. J. G. M. van der Linden,	TU Delft, daily supervisor
	MSc. K. Sidorov,	TU Delft, daily supervisor
	Dr. T. Abeel	TU Delft

An electronic version of this thesis is available at
http://repository.tudelft.nl/.

TU Delft
Delft University of Technology

Preface

For the last nine months, I have been working on this thesis. This period brought me many insights and an excellent opportunity to deepen my knowledge in numerous subjects, mainly Decision Diagrams. Even though I had some stressful periods, during which I worked far into the night, the goal I had in mind and the enthusiasm of my supervisors made this journey worthwhile. This thesis marks the end of the extraordinary journey of my Master of Computer Science at the Delft University of Technology.

First, I am grateful for all the support and feedback from my supervisor, Mathijs de Weerdt. I am thankful for the biweekly meetings in which he gave me valuable input for new and interesting research directions. I want to thank him for all the opportunities he has given me.

Secondly, I want to thank my daily supervisors, Koos van der Linden and Konstantin Sidorov. They have given me their advice and guidance over these past months. I am incredibly grateful for their insightful ideas during our weekly meetings. In these weekly meetings, we had many interesting discussions regarding Decision Diagrams and Decision-Focused Learning. Above all, they showed me how to effectively set up my research.

During my thesis, I was also able to join the weekly Monday Outstanding Orations sessions organized by the helpful people of the Algorithmics research group at the TUDelft. Throughout these sessions, I got to acquaint myself with all kinds of interesting lines of research. In one of these meetings, I also had the opportunity to present a paper myself, which was a valuable learning experience. Additionally, I am grateful for the opportunity to present my thesis topic to Willem-Jan van Hoeve, an expert in the field of Decision Diagrams, who provided helpful insights.

Finally, I want to thank my family and girlfriend for their endless support during these past months. I am incredibly thankful for their feedback on my writing and for always believing in me.

Jop Schaap
Delft, January 2025

Contents

Preface iii

Abstract vii

Nomenclature ix

1 Introduction 1
- 1.1 Research Questions . . . 3
- 1.2 Contributions . . . 4

2 Preliminaries 7
- 2.1 Problem Setting . . . 7
- 2.2 Regression and Gradient Descent . . . 8
- 2.3 Decision-Focused Learning . . . 9
- 2.4 Decision Diagrams for Optimization . . . 11
  - 2.4.1 Relaxed DDs . . . 11

3 Background 13
- 3.1 Smart Predict-and-Optimize Loss . . . 13
- 3.2 Noise Contrastive Estimation . . . 14
- 3.3 Construction of DDs . . . 16
  - 3.3.1 Top-Down Approach . . . 16
  - 3.3.2 Iterative Refinement Approach. . . . 18

4 Method 21
- 4.1 Learning the Predictor . . . 21
  - 4.1.1 Node Wise Decision Loss . . . 21
  - 4.1.2 Path Wise Decision Loss . . . 22
  - 4.1.3 Probabilistic Path-Finding . . . 23
  - 4.1.4 Maximum Likelihood Loss . . . 24
- 4.2 Constructing the Relaxed Decision Diagram . . . 25
- 4.3 Additional Performance Improvements . . . 28
  - 4.3.1 Vectorized Calculation . . . 28
  - 4.3.2 Log-Sum-Exp Trick . . . 28
  - 4.3.3 Partial Derivatives of Loss Functions . . . 29

5 Experiments and Results 31
- 5.1 Experimental Setup. . . . 31
  - 5.1.1 The Two-Stage Baseline . . . 32
  - 5.1.2 Knapsack Problem . . . 32
  - 5.1.3 Traveling Salesperson Problem. . . . 33

5.2 Runtime Experiments 34
5.2.1 Knapsack Problem 34
5.2.2 Traveling Salesperson Problem 35
5.2.3 Discussion 37
5.3 Loss Performance in Exact Decision Diagrams 37
5.3.1 Knapsack Problem 38
5.3.2 Traveling Salesperson Problem 38
5.3.3 Discussion 39
5.4 Loss Performance in Approximate Decision Diagrams 40
5.4.1 Knapsack Problem 41
5.4.2 Traveling Salesperson Problem 42
5.4.3 Discussion 44

6 Conclusion **47**
6.1 Future Work 48

A Proof for Equivalence Equation (4.13) and Equation (4.10) **55**

B Derivation of Gradients for Probabilistic Path Finding Loss **59**
B.1 Basic Definitions 59
B.2 Gradient of Loss and Expectation 59
B.3 Gradient of $\mathrm{P}(a \in \mathcal{P} \mid \mathrm{from}(a) \in \mathcal{P})$ 60
B.4 Gradient of $\mathbb{F}[n|\vec{c}]$ 60
B.5 Conclusion 61

C Derivation of Gradients for Maximum Likelihood Loss **63**
C.1 Basic Definitions 63
C.2 Gradient of the Normalization Value Z 63
C.3 Conclusion 64

Abstract

Decision-Focused Learning (DFL) focuses on a setting where a system gets as input some features and needs to predict coefficients to a downstream optimization problem. Classically, one would apply a two-stage solution, which trains the predictor as a regression task and only uses the optimizer during evaluation. However, the two-stage solution fails to optimize the downstream optimization problem. As such, one might use DFL techniques to train the predictor. Nonetheless, these fail to take the entire solution space into account and only optimize toward the optimal true solution, and as such, they might fail to optimize the total downstream value. Furthermore, these techniques are computationally expensive.

We tackle these two problems using Decision Diagrams (DDs) for DFL. DDs for DFL have three main benefits. First, the DD can be cached between runs, speeding up the training loop. Second, we present four novel loss functions that use DDs to reason efficiently over entire solution spaces. Furthermore, we introduce a novel method to relax the DDs to reduce solve-time during training.

We experimentally show that the DDs speed up the training loop substantially. We further show that the DFL losses perform on par with other state-of-the-art DFL losses. Finally, we experimentally show when and which losses work with relaxed DDs.

Nomenclature

Abbreviations

Abbreviation	Definition
DAG	Directed Acyclic Graph
DD	Decision Diagram
DFL	Decision-Focused Learning
DFJ	Dantzig-Fulkerson-Johnson
KP	Knapsack Problem
MAP	Maximum A Posteriori
ML	Maximum Likelihood
MLE	Maximum Likelihood Estimation
NCE	Noise Contrastive Estimation
PPF	Probabilistic Path Finding
SPO	Smart Predict-and-Optimize
TSP	Traveling Salesperson Problem

Symbols

Symbol	Definition	Type
N	The number of data points in the dataset $\mathcal{D}$	$\mathbb{Z}$
p	The number of features in the input to the predictor	$\mathbb{Z}$
d	The number of coefficients in the input to the optimizer	$\mathbb{Z}$
$\mathcal{D}$	The dataset containing both the features and coefficients	(X, C)
X	Matrix containing all features of all data points in the dataset	$\mathbb{R}^{N \times p}$
$\vec{x}$	Data point corresponding all features of a single instance of X	$\mathbb{R}^p$
C	All data points' true coefficients (or labels)	$\mathbb{R}^{N \times d}$
$\vec{c}$	The true coefficients for the current data point	$\mathbb{R}^p$
$\vec{\hat{c}}$	The predicted coefficients for the current data point, as produced by $t_\theta(\vec{x})$	$\mathbb{R}^p$
$t_\theta(\vec{x})$	The predictor parameterized by θ	$\mathbb{R}^p \rightarrow \mathbb{R}^d$
$\mathcal{V}$	The set containing all solutions	$\subseteq \mathbb{Z}^d$

Symbol	Definition	Type
v	A single feasible solutions	$\mathcal{V}$
$v^*(\vec{c})$	The optimizer, which produces the optimal solution corresponding to the values $\vec{c}$	$\mathbb{R}^d \to \mathbb{Z}^d$
$\mathcal{L}(\vec{c}, \hat{\vec{c}})$	The loss function	$\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$
$\mathcal{N}$	The set of all nodes in the DD	n/a
n	A single node in the DD	$\mathcal{N}$
r	The root node in the DD	$\mathcal{N}$
t	The sink node in the DD	$\mathcal{N}$
$\mathcal{A}$	The set of all arcs in the DD	n/a
a	A single arc in the DD	$\mathcal{A}$
$\mathrm{from}(a)$	The function to get the node the arc a is originating from	$\mathcal{A} \to \mathcal{N}$
$\mathrm{to}(a)$	The function to get the node the arc a is pointing to	$\mathcal{A} \to \mathcal{N}$
c_a	The coefficients belonging to $\vec{c}$ which corresponds to arc a	$\mathbb{R}$

1 Introduction

Many optimization problems in real life require optimizing a future decision. The parameters required for such an optimization problem are frequently predicted from data. For instance, one might wish to optimize machine usage when electricity prices are unknown upfront or find a traffic route when unsure how much traffic there is that day. Here, the parameters they optimize over have some uncertainty. Generally, this stems from a machine learning algorithm producing these parameters. In this work, we are interested in the paradigm of training such machine learning algorithms specifically for such optimization tasks. Our focus is on problems where we predict coefficients of the objective function, and thus consider the constraints fixed.

These learning problems, are structured as follows: first, a machine learning algorithm produces predicted coefficients for the objective function based upon some features, which we call the predictor. The resulting predicted coefficients are hereafter fed into an optimizer, which makes decisions. Finally, the actual value of these combined decisions is determined, which should be as high as possible.

The naive method separates the task of training and optimizing. This means that the predictor is first trained in a regression task. The resulting predictor is then used to generate the parameters the optimizer optimizes over. We call this method the two-stage method since it works in the two stages of first learning and hereafter optimizing over these parameters. Thus, the general structure of such problems is the following: first, the predictor is trained on the values of the coefficients of the optimization function [1]. Hereafter, when evaluating new data points, the predictor first predicts the optimization parameters, after which the optimization algorithm is run to find the optimal assignment.

However, the prediction objectives these methods optimize, such as mean squared error, cannot measure the quality of the ensuing decisions appropriately [1–4]. The two-stage method's underperforming in these situations stems from the training objective not including that the predictor and optimizer workings are highly interconnected. Since the absolute accuracy of the predictor is less important than the relative value between parameters for optimization problems. For instance, if we have a problem with a linear objective function. Then, we could multiply all values by some positive constant without

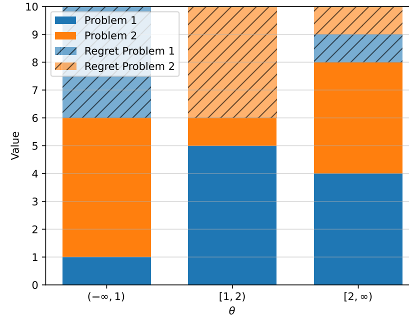


Figure 1.1: Example for a maximization problem where the predictor parameter (θ) is a single scalar value. Here, we have two problems that are optimal in a different location to where the global optimal lies. Namely, problem one would be optimal for $\theta \in [1, 2)$ range, and problem two for $\theta \in (-\infty, 1)$. However, the global optimal lies in the $[2, \infty)$ range.

affecting the assignment of the optimizer. Still, the prediction error will be higher. Similarly, often by increasing the prediction error, we can improve the system output. In other words, the quality of a predictor is measured in the decisions it causes and not in the prediction error [5].

As an alternative to the two-stage method, one could, instead of optimizing the predictor independently, minimize the difference between the optimizer value based upon predicted and the true value; this is the so-called regret [5, 6]. However, directly optimizing this regret is hard. To remedy this, one can use Decision-Focused Learning (DFL) techniques. One such method is to optimize a substitute loss instead of the regret [1, 7]. These losses, nonetheless, have two downsides. First is the computational cost of using a solver for every training problem every epoch. Second, these losses are defined on individual optimal solutions and thus might fail to find global optima.

The computational complexity of solving all instances in every epoch severely impacts runtime performance. Since it requires solving many problems, which quickly can become a bottleneck [6]. Therefore, it is beneficial to determine the set of feasible solutions upfront so that only the value has to be calculated after the prediction step.

The second downside for DFL losses is that they are defined such that the resulting predictor causes the optimal solution to be picked [1, 7]. However, the goal of DFL is not to optimize the best solution as often as possible but to obtain the maximum total value for the parameters of the predictor (θ).

Example 1.1. For instance, we have two training data points for a maximization problem, which both have three possible solutions. Let the first data point have solutions with values $1, 5, 4$ and the second solutions with values $5, 1, 4$, for three settings of the predictor parameters $(-\infty, 1), [1, 2), [1, \infty)$, as shown in Figure 1.1. Here, DFL losses defined as the optimal solution for individual problems will move to the $(-\infty, 2]$ range. However, Figure 1.1 shows that the global optimal value for θ lies in the range $[2, \infty)$. Thus, these losses will fail to find the globally optimal point for the parameters.

We combat both previously mentioned downsides of DFL by using a Decision Dia-

gram (DD) during learning. Since, using DDs allows us to combat both: the computationally expensive step of optimizing during loss calculation and that previous methods did reason over multiple solutions.

DDs help with the computational complex step of solving the optimization problems since they efficiently encode all feasible solutions for a given problem [8]. Thus, we could search for the optimal solution in a DD. Consequently, we would only have to construct the DD once upfront and hereafter reuse it instead of using a solver. This is especially beneficial since searching through a DD for the optimal solution takes linear time with respect to the number of arcs in the DD [8]. Therefore, by using DDs for DFL, we should substantially reduce the number of operations performed to find the optimal solution since we do not have to recreate the DD for every problem instance.

To efficiently reason over multiple solutions, we can again use DDs and reason over the solutions represented in their structure. This reasoning is especially beneficial since the way DDs encode solutions means that if two solutions in a DD share a sub-graph, they also share part of their solution assignment. As a result of DDs sharing a subgraph, we can define losses that reason over the entire solution space while requiring only linear time with respect to the number of arcs in the DD. Thus, using DDs in loss functions should produce more effective loss functions that can better optimize regret since they allow reasoning over entire solution spaces.

Finally, DDs can be used for problems that are too complex to solve in a reasonable time, which can be achieved by constructing an approximate DD and using this to find solutions for the relaxed problem [9]. Relaxed DDs are similar to the linear relaxations for Mixed Integer Programs (MIP) problems as discussed by Tang and Khalil [3]. For DDs, however, we are not required to use a fixed relaxation; we could instead also permute the relaxed DD so that certain parts of the solution space become more relaxed and others are more exact. Thus, DDs have the additional benefit of allowing dynamic relaxation, where their relaxation of certain parts of the solution space can become more or less.

In this work, we focus on defining loss functions for DFL that enable us to reason over solution spaces with the help of DDs. Furthermore, we look at how we can relax DDs such that they benefit from training in the DFL training loop:

1.1. Research Questions

The existing methods for DFL have the drawbacks that they require running a solver many times for the same problem, and that they fail to optimize the total regret instead minimizing the regret of single solutions. Therefore, we opt to use the novel method of employing DDs for DFL. DDs have the benefit of efficiently storing the entire solution space and thus allow for quick solving and reasoning over the solution space. Since we conjecture that there should exist a substantial benefit to use DDs in DFL, thus we formulated the following research question:

> ***RQ:*** *How can DDs be (re)used in the DFL setting to compute update steps during learning that minimize regret?*

Runtime To answer the research question, we came up with several novel methods for using DDs to calculate losses, that aim to minimize the regret. These methods,

furthermore, need to run the update steps to the predictor, in a reasonable time window. Thus, the first sub-question we would like the answer is:

> ***SQ 1:*** *What is the effect of using DDs in place of traditional solvers in terms of runtime performance?*

By answering this sub-question, we aim to give insight into whether the caching of the DD helps reduce the time spent solving for DFL-based methods. It would give insight into what problems the methods could apply to since they should be able to finish in reasonable time.

We hypothesize that using DDs for solving the problems results in a substantial speedup because the optimal solution of the problem can be found in linear time with respect to the number of arcs in the DD. This, in contrast to solver-based techniques which explicitly have to solve the entire problem for every new prediction.

Obtained Regret The second sub-question relates to what the effect is of the proposed learning methods on the total regret obtained. By answering this we would gain insight into how useful the learning methods are for obtaining a low regret.

> ***SQ 2:*** *How effective are the proposed losses at minimizing the obtained regret?*

We expect that the proposed losses will result in faster and better performance in terms of regret because the proposed methods are able to reason over the entire solution space instead of only on the found solution and optimal solution.

Approximate DDs The final sub-question regards to whether the learning methods still work if a dynamic approximate DD is used during training instead of an exact one. It associates with the main question in that the answer should clarify whether these methods would still be beneficial if the original problem is relaxed.

> ***SQ 3:*** *How effective are approximate DDs for learning the predictor coefficients compared to using a linear relaxed mixed integer formulated model?*

We conjecture the following hypothesis to this sub-question. We expect that the approximate DDs will outperform the linear relaxation method when the approximate size is larger, and similarly that it performs worse once the size gets smaller. Where, we anticipate that at the smallest sizes it fails to learn entirely and might even get worse during training.

1.2. Contributions

The contributions of this work are threefold. The first is that we obtained a substantial speedup compared to previous methods regarding the optimization step of the learning algorithms. The second contribution is the introduction of three novel loss functions that highly interconnect with the DDs to reason over the whole or parts of the solution space as defined by the DD. We also show that dynamic DDs achieve training the

predictor and show when they do not. We will make all the code for the experiments publicly available on GitHub[1].

[1] https://github.com/JopSchaap/Decision-Diagram-Focused-Learning

2
Preliminaries

This section explains the notation and terms used in the rest of this work. We start by explaining the basic notation used for the DFL setting. Hereafter, we discuss the workings of regression and gradient descent. Finally, we discuss the use of DDs for solving optimization problems. This section also introduces many of the notation used throughout this work, this notation is based upon the notation of [3] for the DFL part and [8] for DDs.

2.1. Problem Setting

The DFL problem consists of two main parts: the predictor and the optimizer. However, the dataset and its definition are essential to understand how the DFL setting works. This section explains what data the DFL settings assume during training and while using the system.

The predictor gets as input some features for each problem, and we represent these features as a vector of real numbers $\vec{x} \in \mathbb{R}^p$, where p is the number of features. The entire dataset contains N of these vectors, and we represent this as the matrix X of size $N \times p$, where each row is associated with the features of one data point $\vec{x}$.

Similarly, the coefficients corresponding to an individual data point is $\vec{c} \in \mathbb{R}^d$, with the d being the number of coefficients for an individual problem. Similarly, C is the matrix of size $N \times d$ corresponding to all coefficients for all data points in the dataset, with each row being the coefficients for an individual problem $\vec{c}$. When using the system for new data, these values would be unknown and thus have to be predicted by the predictor. Furthermore, when the predictor predicts these values, we denote them with a hat: $\hat{\vec{c}}$ and $\hat{C}$.

Finally, we define the solution set of the optimization problem as $\mathcal{V} \subseteq \mathbb{R}^d$, these are all the feasible solutions to the optimization problem. We denote a single solution to the problem as $\vec{v} \in \mathbb{R}^d$. As such, we denote the value of a solution as the dot product of the solution and the true coefficients $\vec{c}^T \vec{v}$.

One such optimization problem is the Knapsack Problem (KP). The intuitive goal

of the KP is select the maximal value of items while remaining with some secondary budget. We state the definition of the KP in Definition 2.1 to provide a basis for later examples.

Definition 2.1. (**KP**). In the KP the goal is selecting a subset of items that maximize the total selected value while ensuring that the resulting sum of weights of the selected items is less than the capacity of the knapsack W [10]. Thus, the optimization can be defined as the following set of equations:

$$\begin{aligned} \text{maximize}_{\vec{v} \in V} \quad & \vec{v}^T \vec{c} \\ \text{s.t.} \quad & \vec{v}^T \vec{m} \leq W \end{aligned} \tag{2.1}$$

Where m is a vector representing the weight of the different items, $\vec{v}$ is a binary vector indicating which items the current solution would put in the knapsack.

2.2. Regression and Gradient Descent

Regression problems generally take the form of a minimization problem. Here, we want to minimize the distance between some predicted values and the true values. In other words we have some function $t_\theta(\vec{x}) : \mathbb{R}^n \to \mathbb{R}$, which serves as a predictor producing $\hat{c}$. Similarly, we have the true values denoted as c. Finally, we have some distance metric called the loss, which we denote as: $\mathcal{L}(c, \hat{c})$. Thus, we formulate the entire problem as in Equation (2.2) [11, p. 80]:

$$\underset{\theta}{\text{argmin}} \frac{1}{N} \sum_{i=0}^{N} \mathcal{L}\left(C_i, t_\theta(X_i)\right) \tag{2.2}$$

Cauchy [12] devised a way to optimize equation like (2.2) by a technique they called gradient descent. The gradient descent method tries to minimize functions by using gradients to take a small step in the function's input such that the function's value becomes less, as seen in Equation (2.3). These small steps are iteratively repeated until a stopping criterion is reached.

$$\theta_{new} = \theta - \epsilon \nabla_\theta \sum_{i=0}^{N} \mathcal{L}\left(C_i, t_\theta(X_i)\right) \tag{2.3}$$

The gradient descent technique works since the gradient gives the slope at a certain point with respect to the function's input. Thus, taking a small enough step in the opposite direction of the gradient should reduce the value of the function.

A fact that helps compute the gradients of Equation (2.3) is that we can use the chain rules to calculate partial derivatives. As long as we can define gradients for every function concerning their input, we can compute gradients for the entire equation. For more information on how this works and how it is implemented in software, see [11, pp. 200-220].

A special case for gradient descent is when the gradient is the all-zero vector. Since, in such a case, the gradient gives no information as to which direction to move to reduce

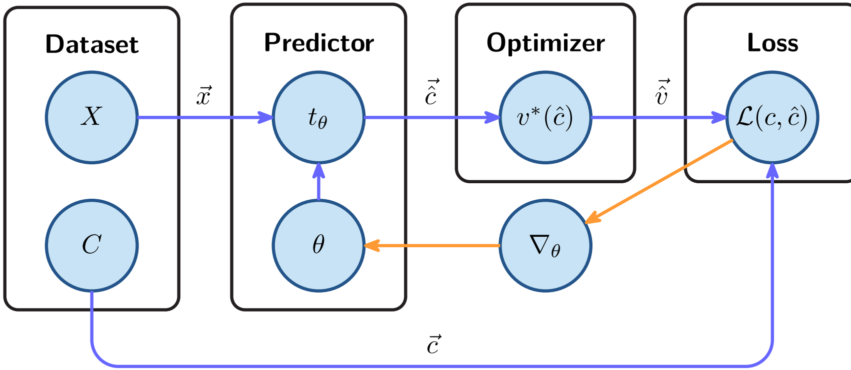


Figure 2.1: Illustration of the DFL setting. In the forward pass (blue arrows), the features $\vec{x}$ flow through the predictor, which generates the coefficients for the optimizer $\vec{\hat{c}}$, which produces a solution. Then, in the backward step (orange arrows), the system calculates the gradients with respect to the parameters of the predictor, which are hereafter updated.

the function. These points are called critical points [11, 13]. These points can happen when we find a (local) minima, (local) maxima, saddle points or plateaus.

Another point where gradient descent is unhelpful is for points where the gradient is undefined. This generally happens for discontinuous functions. In these functions, there are points were taking an infinitesimally small step in a direction would change the value of the function in a non-infinitesimal amount. In other words, the function takes a jump, and thus, the function has not a single tangent at this point.

Thus, gradient descent works when the gradient is defined and the shape of the function has gradients that point towards the desired solution. It is especially helpful when there is no analytical solution for the minimum, and obtaining a value that is close to optimal is good enough, such as when training for complex problems.

2.3. Decision-Focused Learning

The regression tasks, as described in Section 2.2, can be directly used to fit the predictor on the true coefficients, known as the two-stage solution. However, the two-stage solution ignores the downstream optimization step, as noted in Section 1, and thus is less fit for the DFL setting.

Definition 2.2. (DFL). The *DFL* setting consists of four parts, the first being the *dataset* $\mathcal{D} = (X, C)$, the second the *predictor* $t_\theta : \mathbb{R}^p \to \mathbb{R}^d$, the downstream *optimization problem* $v^* : \mathbb{R}^d \to \mathbb{Z}^d$ and the *loss function* $\mathcal{L} : (\mathbb{R}^d \times \mathbb{R}^d) \to \mathbb{R}$.

Figure 2.1 shows a schematic representation of the DFL setting, where the predictor is trained through gradient descent. The figure highlights the building blocks of the DFL setting.

The first part of the DFL setting is the dataset $\mathcal{D}$. This consists of labelled entries, where each entry consists of some features $\vec{x} \in \mathbb{R}^d$ and true optimization values $\vec{c} \in \mathbb{R}^d$, where the label is the actual value. Both the dataset and the true optimization variables are assumed to be given in the DFL setting.

The second part is the predictor, which functions as a regression model, as discussed in Section 2.2. The predictor task is to generate values $\vec{\hat{c}}$ that cause the downstream to make high-value decisions. We denote the predictor as a linear model, even though DFL is not limited to such predictors. Other possible predictors include but are not limited to deep neural networks or random forests [14, 15]. However, this work will exclusively focus on linear models as predictors. We will denote the predictor as the function $t_\theta(\vec{x})$, where x is the feature vector and θ are the predictor's parameters, which we wish to learn.

The predictor function outputs a cost vector $\vec{\hat{c}}$ that forms the coefficients for an optimization problem with a linear objective function. These coefficients can hereafter be passed to the third part, being the optimization solver $v^*(\vec{\hat{c}})$. This solver calculates the best assignment of the optimization variables $\vec{v}$, which satisfies the problem constraints.

In the DFL setting, the goal is to train a predictor $t_\theta(x)$ that generates values $\vec{\hat{c}}$, such that the actual optimal decisions are taken. The decision step here is denoted by $v^*(\vec{c}) \rightarrow \mathbb{R}^d$, which returns the optimal decisions given the weights $\vec{c}$. Furthermore, the value of the optimal decision is denoted as $\vec{c}^T v^*(\vec{c})$.

In other words, DFL's main problem is learning a model that minimizes the missed reward because of imprecise prediction [16]. This difference is the so-called regret and is formally defined for maximization problems in Equation (2.4). Similarly, for minimization problems, the regret is defined by taking the negative value of the maximization regret, as seen in Equation (2.5) [16]. Since the decisions made using the predicted values can never be better than the true values, we know regret is nonnegative.

$$\text{Regret}_{\text{maximize}}(\vec{\hat{c}}, \vec{c}) = \vec{c}^T v^*(\vec{c}) - \vec{c}^T v^*(\vec{\hat{c}}) \tag{2.4}$$

$$\text{Regret}_{\text{minimize}}(\vec{\hat{c}}, \vec{c}) = -\,\text{Regret}_{\text{maximize}}(\vec{\hat{c}}, \vec{c}) \tag{2.5}$$

Equations (2.4) and (2.5) give the regret for a single data point. However, we are typically interested in the entire dataset's regret. We might naturally take the sum over the whole dataset. Nevertheless, this would make the regret dependent on the dataset's size and the coefficients' size in the optimization functions. Thus, to reduce this, we use the normalized regret, as seen in Equation (2.6) [1].

$$\text{NormalizedRegret}(\hat{C}, C) = \frac{\sum_{i=0}^{n} \text{Regret}(\hat{C}_i, C_i)}{\sum_{i=0}^{n} C_i^T v^*(C_i)} \tag{2.6}$$

Regret is a clear goal to use for training since it is a direct metric of how much value the system obtains. However, using regret as a loss function with gradient descent does not work since the regret only contains stationary points or points where the gradient is undefined [1]. This is the case since for almost all values for the predicted points $\vec{\hat{c}}$ adding on infinitesimal small values, no change in regret will happen. Furthermore, points on decision boundaries between solutions have an undefined gradient. This gradient is undefined in these locations since an infinitesimally small step exists for the optimization coefficients that change the solution, such that the true value changes by a non-infinitesimal amount. In other words, regret is a noncontinuous function, and noncontinuous functions have undefined gradients at the point of discontinuity.

One common way to train models in the DFL paradigm is to use a specialized loss function and then use gradient descent to train [1, 7, 17]. These losses are created such that we can calculate useful gradients over it that we can use to perform gradient descent on the predictor parameters θ.

2.4. Decision Diagrams for Optimization

As discussed in the introduction, DDs are a promising candidate to apply for DFL. This section discusses how DDs work in the optimization setting. We further discuss the workings of relaxed DDs.

DDs have recently seen use in optimization where they are used for solution space encoding for combinatorial problems [8]. When using DDs for optimization, the DD is defined as a Directed Acyclic Graph (DAG) encoding feasible solutions to an optimization problem. More formally we use the definition from Castro et al. [8], as seen in Definition 2.3.

Definition 2.3. (DD). A DD is a layered DAG defined by a *set of nodes* $\mathcal{N}$ and a *set of directed arcs* $\mathcal{A}$, where each node $n \in \mathcal{N}$ is associated with exactly one layer. The first and last layers contain exactly one element, the root node $\mathrm{r} \in \mathcal{N}$ and the sink node $\mathrm{t} \in \mathcal{N}$. All arcs in $\mathcal{A}$ have a source $\mathrm{from}(a)$ and target node $\mathrm{to}(a)$, where the $\mathrm{to}(a)$ node lies in the consecutive layer from $\mathrm{from}(a)$ [8].

The paths through a DD represent the solution set $\mathcal{V}$ to an underlying problem. Here, the mapping between a path $\mathcal{P}$ through the DD and the solution $\vec{v}$ is as follows. We first define the path as a series of arcs taken $\mathcal{P} = (a_1, a_2, .., a_d)$. We define the decision variables as $\vec{z} \in \mathbb{Z}^d$. Finally, each arc $a \in \mathcal{A}$ corresponds to exactly one decision variable z_a. Now, every arc in the path adds 1 to the decision variable z_a. To obtain the value of a solution, we take the length of the path through the DD. Arc lengths are determined by the coefficients of that respective decision variable c_a.

The benefit of DDs in optimization is that they provide an explicit and potentially compact representation of the solution space. This structure allows searching for the optimal solution in linear time concerning the number of nodes.

The rest of this section focuses on techniques for decreasing the size of the DD by approximating the solution space. We explain how DDs can approximate the solution space, with so-called relaxed DDs.

2.4.1. Relaxed DDs

For DDs, we wish that the set of all $\mathrm{r} - \mathrm{t}$ paths represent the same solution set as all feasible solutions, since finding the optimal solution corresponds to finding the optimal path through the DD. We call DDs that exactly represent the set $\mathcal{V}$ exact DDs. Thus, we have an exact DD iff the following equation holds:

$$\mathcal{V} = \mathrm{Sol}(DD) \tag{2.7}$$

However, exact DDs can grow too large, making them impractical for bigger problems. To deal with this potential exponential growth of the DD size, the DD is often relaxed. Andersen et al. [18] first proposed the method for creating relaxed DDs. The

main idea in approximating DDs is to limit the size of the DD and merge similar subgraphs whenever the size of the DD during construction exceeds the maximum size. This merging needs to happen so that no paths are removed, but it allows for additional paths to be created. The resulting relaxed DD has the following property:

$$\mathcal{V} \subseteq \mathrm{Sol}(DD) \tag{2.8}$$

3

Background

We based our work on several lines of research in both the DFL and approximate DD construction domains. In this section, we discuss some of these works. We start by discussing several DFL losses that train the predictor in a DFL setting: SPO+ [1] and the contrastive losses [7]. Hereafter, we discuss two construction methods for relaxed DDs: the top-down construction method [19] and the iterative refinement method [20].

3.1. Smart Predict-and-Optimize Loss

Elmachtoub and Grigas [1] discuss a loss for use in the DFL setting, called the Smart Predict-and-Optimize (SPO) loss. The base SPO loss is the same definition as the regret function with the exception that when there exist several feasible solutions that are both considered optimal concerning $\vec{\hat{c}}$. For our regret definition, when more than one optimal solution exists, the optimizer can decide which version to take. On the other hand, SPO requires that whenever there exists more than one optimal solution, the optimizer picks the worst one for $\vec{c}$ [1]. This is done to ensure that there is no coupling to the particular optimization oracle.

The base SPO loss exhibits the same problems as optimizing the regret directly. Namely, the gradient of the SPO loss is always either zero or undefined [1]. To counteract this, Elmachtoub and Grigas [1] propose the surrogate loss SPO+ as defined in Equation (3.1).

$$\mathcal{L}_{\text{SPO+}}(\vec{c}, \vec{\hat{c}}) = (2\vec{\hat{c}} - \vec{c})^T v^*(2\vec{\hat{c}} - \vec{c}) - 2\vec{\hat{c}}^T v^*(\vec{c}) + \vec{c}^T v^*(\vec{c}) \tag{3.1}$$

Elmachtoub and Grigas [1] explain that the SPO+ loss is the sum of the distances between the incorrectly chosen decision variables. To see this, note that we can rewrite the SPO+ loss as Equation (3.3). This equation contains the term $(v^*(2\vec{\hat{c}} - \vec{c}) - v^*(\vec{c}))$, which is the output of the optimization problem based upon the predicted value and on the right the ground truth solution. The resulting vector of this term exposes whether the predicted values $\vec{\hat{c}}$ were too low, too high or correct. Namely, a value smaller than

zero means that the predicted value was too low, and a greater than zero value indicates that it was too high. Meanwhile, the other term indicates the distance of that distance variable to the true solution. Thus, the SPO+ loss only sums up the distances of predicted coefficients of incorrectly chosen decision variables.

$$\mathcal{L}_{\text{SPO+}}(\vec{c}, \vec{\hat{c}}) = (2\vec{\hat{c}} - \vec{c})^T v^*(2\vec{\hat{c}} - \vec{c}) - (2\vec{\hat{c}} - \vec{c})^T v^*(\vec{c}) \tag{3.2}$$

$$\mathcal{L}_{\text{SPO+}}(\vec{c}, \vec{\hat{c}}) = (2\vec{\hat{c}} - \vec{c})^T (v^*(2\vec{\hat{c}} - \vec{c}) - v^*(\vec{c})) \tag{3.3}$$

Example 3.1. Take, for example, a unit KP with true values $\vec{c} = [1, 2, 3, 4]^T$ and a capacity of 2. Furthermore, let the predicted values be $\vec{\hat{c}} = [0.5, 4, 3, 2]^T$. Now, the optimizer will select incorrect values for the KP, namely items two and three instead of three and four. Thus, the resulting predicted solution is: $\vec{\hat{v}} = [0, 1, 1, 0]^T$, and the true solution is: $\vec{v} = [0, 0, 1, 1]^T$. The SPO+ loss now is:

$$\mathcal{L}_{\text{SPO+}}\left(\begin{pmatrix}1\\2\\3\\4\end{pmatrix}, \begin{pmatrix}0.5\\4\\3\\2\end{pmatrix}\right) = \begin{pmatrix}0\\6\\3\\0\end{pmatrix}^T v^*\left(\begin{pmatrix}0\\6\\3\\0\end{pmatrix}\right) - 2\begin{pmatrix}0.5\\4\\3\\2\end{pmatrix}^T v^*\left(\begin{pmatrix}1\\2\\3\\4\end{pmatrix}\right) + \begin{pmatrix}1\\2\\3\\4\end{pmatrix}^T v^*\left(\begin{pmatrix}1\\2\\3\\4\end{pmatrix}\right) \tag{3.4}$$

$$= \begin{pmatrix}0\\6\\3\\0\end{pmatrix}^T \begin{pmatrix}0\\1\\1\\0\end{pmatrix} - 2\begin{pmatrix}0.5\\4\\3\\2\end{pmatrix}^T \begin{pmatrix}0\\0\\1\\1\end{pmatrix} + \begin{pmatrix}1\\2\\3\\4\end{pmatrix}^T \begin{pmatrix}0\\0\\1\\1\end{pmatrix} \tag{3.5}$$

$$= -(6 + 3) + 2 \cdot (3 + 2) - (3 + 4) = 9 - 10 + 7 = 6 \tag{3.6}$$

Here, we note that the incorrectly predicted first value does not impact the loss since the optimizer's decision for this variable was the same for both the true and predicted optimal assignments. Furthermore, the correctly predicted value of 3 cancels out in the second half of the equation. Thus, the only predicted values that impact the loss are 4 and 2, 4 from the left-hand side and 2 from the right-hand side.

Finally, the SPO+ loss is a convex upper bound on the SPO loss [21]. Thus, SPO+ is also a convex upper bound of the regret since the SPO loss is itself an upper bound to the regret. However, a downside of the SPO+ loss is that every time we generate a new value for $\vec{\hat{c}}$, we have to recompute the value of $(\vec{c} - 2\vec{\hat{c}})^T v^*(\vec{c} - 2\vec{\hat{c}})$, which is computationally expensive when optimizing over an NP-complete problem [21], such as for the KP.

3.2. Noise Contrastive Estimation

Next to SPO+, contrastive losses also fall under DFL loss functions [7]. These losses are based upon Maximum Likelihood Estimation (MLE), which associates a probability to each solution. Then, during the training phase, the probability of the true optimal solution is maximized through gradient descent. This section will explain these losses.

Mulamba et al. [7] use MLE to define a loss function for the DFL paradigm. To do this, they define the entire DFL system as a probabilistic generator of solutions, where

the probability of generating the solution is related to the value associated with that solution. MLE aims to find the predictor parameters θ that maximize the likelihood of seeing the true optimal solutions $v^*(\vec{c})$.

In the context of DFL, instead of taking the data as is, we would first optimize the problem and want to find the model that maximizes taking the same decisions as the solution found using the true values [7]. Thus, the task becomes finding the value for θ that causes the predictor to produce values $\vec{\hat{c}}$, such that the true optimal solution $v^*(\vec{c})$ is as likely as possible. That is, we wish to find θ that maximizes the posterior of the probabilistic optimizer $\tilde{v}(\vec{\hat{c}})$.

$$\underset{\theta}{\operatorname{argmax}} \mathrm{P}(\tilde{v}(\vec{\hat{c}}) = v^*(\vec{c})|\theta, \vec{x}) \tag{3.7}$$

Mulamba et al. [7] construct such a posterior for output values given some model. They use the exponential distribution to model the probability of producing the solution $\vec{v}$ as the optimal solution given the model.

$$P(\vec{v}|\theta, \vec{x}) = \frac{1}{Z} \exp(\vec{\hat{c}}^T \vec{v}) \tag{3.8}$$

Here, v is a feasible assignment to the problem. Furthermore, Z is a normalization constant to ensure that the posterior over all feasible assignments sums to one [7]. Thus, the value for Z is defined as:

$$Z = \sum_{\vec{v} \in \mathcal{V}} \exp(\vec{c}^T \vec{v}) \tag{3.9}$$

In this equation, V is the set of all feasible solutions to the problem. Mulamba et al. [7] state that Z is hard to compute since it requires enumerating all solutions in the solution space, which is intractable for most problems. They argued this since the solution spaces for these problems often are exponential.

To remedy this Mulamba et al. [7] use Noise Contrastive Estimation (NCE). NCE, as described by Gutmann and Hyvärinen [22], is an estimation method that prevents having to evaluate Z exactly. Instead of evaluating Z directly, NCE maximizes the fraction of the probability of the optimal solution and the solutions of the set $\mathcal{S} \subset \mathcal{V} \setminus v^*(\vec{c})$. NCE maximizes the product of the fraction, as seen in Equation (3.10) [7].

$$\underset{\theta}{\operatorname{argmax}} \prod_{i}^{N} \prod_{v \in \mathcal{S}} \frac{\mathrm{P}(\tilde{v}(\vec{\hat{c}}) = v^*(\vec{c})|\theta, \vec{x})}{\mathrm{P}(\tilde{v}(\vec{\hat{c}} = v)|\theta, \vec{x})} \tag{3.10}$$

To make Equation (3.10) more numerically stable, Mulamba et al. [7], take the logarithm of it. Then, when one fills in the equation for the probability (see Equation (3.8)), the logarithm transforms the product into a sum and removes the exponent inside the probability function, as seen below:

$$\underset{\theta}{\text{argmax}} \log \prod_{i}^{N} \prod_{v \in \mathcal{S}} \frac{\mathrm{P}(\tilde{v}(\vec{\hat{c}}) = v^*(\vec{c})|\theta, \vec{x})}{\mathrm{P}(\tilde{v}(\vec{\hat{c}} = v)|\theta, \vec{x})} \tag{3.11}$$

$$= \underset{\theta}{\text{argmax}} \sum_{i}^{N} \sum_{v \in \mathcal{S}} \log \exp(\vec{\hat{c}}^T v^*(\vec{c})) - \log \exp(\vec{\hat{c}}^T v) \tag{3.12}$$

$$= \underset{\theta}{\text{argmax}} \sum_{i}^{N} \sum_{v \in \mathcal{S}} \vec{\hat{c}}^T v^*(\vec{c}) - \vec{\hat{c}}^T v \tag{3.13}$$

$$= \underset{\theta}{\text{argmax}} \sum_{i}^{N} \sum_{v \in \mathcal{S}} \vec{\hat{c}}^T \left(v^*(\vec{c}) - v\right) \tag{3.14}$$

One potential problem with this equation is that for linear optimization functions, the degenerate solution of all the zero vector $\vec{0}$ minimizes it [7]. To remedy this Mulamba et al. [7] use $\vec{\hat{c}} - \vec{c}$ instead of using the value of $\vec{\hat{c}}$ as input to Equation (3.10)'. Finally, Mulamba et al. [7] covert this equation to a loss by inverting the sign, as shown in Equation (3.15).

$$\mathcal{L}_{\text{NCE}}(\vec{c}, \vec{\hat{c}}) = \sum_{v \in \mathcal{S}} (\vec{\hat{c}} - \vec{c})^T \left(v^*(\vec{c}) - \vec{v}\right) \tag{3.15}$$

As an alternative to NCE, Mulamba et al. [7] also discuss contrastive Maximum A Posteriori (MAP) estimation as a special case for NCE. Namely, it uses the singleton set $\mathcal{S} = \left\{v^*(\vec{\hat{c}})\right\}$. That is, it uses the predicted optimal solution. Applying the same transformations as for NCE, they obtain the following loss for MAP:

$$\mathcal{L}_{\text{MAP}}(\vec{c}, \vec{\hat{c}}) = (\vec{\hat{c}} - \vec{c})^T \left(v^*(\vec{c}) - v^*(\vec{\hat{c}})\right) \tag{3.16}$$

3.3. Construction of DDs

We now discuss two ways of constructing DDs in literature. There exist two main ways in the literature to construct DDs efficiently. The top-down approach [19] and the iterative refinement [20]. Here, the top-down applies both relaxed and exact DDs, and the iterative refinement is mainly used for relaxed DDs.

3.3.1. Top-Down Approach

The top-down construction algorithm uses a recursive formulation for the problem, which hereafter is encoded in the DD [19]. The top-down algorithm starts with a DD consisting of a root node with some initial state. Hereafter, the algorithm takes the root node as the current layer, for which the algorithm creates a new layer. For every node in the current layer, the algorithm makes all the possible outgoing arcs along with the new nodes, which have the state as determined by the state of the starting node combined with the taken arc. If the newly created node has an infeasible state, the new arc and new node are removed from the DD instead, and the algorithm resumes. Here, we note

that if a node has no valid outgoing arcs, it also represents an infeasible state; thus, the algorithm can remove it and all incoming arcs. Furthermore, if the state of the newly created node already exists in the next layer, then the new arc is redirected to the existing node.

Algorithm 1 Exact or Relaxed Top-Down DD Compilation [19]

1: **procedure** CONSTRUCT-DD(s_0, $D : \mathcal{N} \to$ decisions, W)
2: $\quad$ r $\leftarrow (s_0)$ ▷ Initialize the root node with the initial state
3: $\quad L_0 \leftarrow \{\mathrm{r}\}$
4: $\quad$ **for** $i \in [0..n)$ **do** ▷ For every layer in the DD
5: $\quad\quad$ reduce-width(L_i, W) ▷ Reduce width if layer is too wide
6: $\quad\quad L_{i+1} \leftarrow \emptyset$
7: $\quad\quad$ **for** $n \in L_i$ **do** ▷ For every node in the current layer
8: $\quad\quad\quad$ **for** $d \in D(n.state)$ **do** ▷ For every decision possible from the current node
9: $\quad\quad\quad\quad n' \leftarrow$ transition(d, n) ▷ Get the output state of taking the transition d
10: $\quad\quad\quad\quad$ **if** $n' \notin L_{i+1}$ **then**
11: $\quad\quad\quad\quad\quad L_{i+1} \leftarrow L_{i+1} \cup \{n'\}$ ▷ If this is a new state create new node
12: $\quad\quad\quad\quad$ **end if**
13: $\quad\quad\quad\quad a \leftarrow (n, n')$ ▷ Create new arc a and add it to the layer
14: $\quad\quad\quad\quad \mathcal{A} \leftarrow \mathcal{A} \cup \{a\}$
15: $\quad\quad\quad$ **end for**
16: $\quad\quad$ **end for**
17: $\quad$ **end for**
18: **end procedure**

1: **procedure** REDUCE-WIDTH(L_i, W, $\mathcal{A}$)
2: $\quad$ **while** $|L_i| > \mathrm{W}$ **do** ▷ While layer is too wide
3: $\quad\quad n_a, n_b \leftarrow$ node-select(L_i)
4: $\quad\quad n_{new} \leftarrow n_a \oplus n_b$ ▷ Create n_{new} by using the merging operator $\oplus$
5: $\quad\quad L_i \leftarrow L_i \setminus \{n_a, n_b\}$ ▷ Remove old nodes
6: $\quad\quad L_i \leftarrow L_i \cup \{n_{new}\}$ ▷ Add new node
7: $\quad\quad$ **for** $\{a \in \mathcal{A} | \operatorname{to}(a) = n_a \vee \operatorname{to}(a) = n_b\}$ **do** ▷ For all arcs going to n_a, n_b
8: $\quad\quad\quad \operatorname{to}(a) \leftarrow n_{new}$ ▷ Redirect arc to the new node
9: $\quad\quad$ **end for**
10: $\quad$ **end while**
11: **end procedure**

Example 3.1. For example, take the KP, where we have to fit items into a weight-limited knapsack such that the total value of all taken items is maximized. For the DD formulation, we define the state of a node as the weight of the knapsack thus far. Additionally, each layer corresponds to an item, and each node has one or two arcs possible; the first would be not taking the item, thus adding zero to our current weight; the other arc would indicate taking the item and adding the weight of the current item

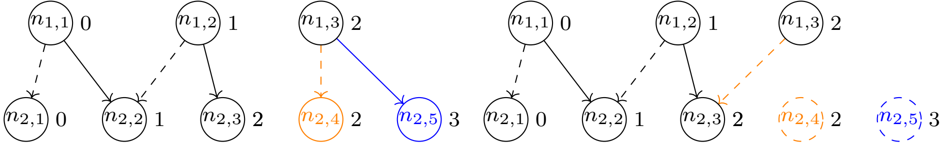


Figure 3.1: Example for the top-down approach generating a new layer, with the capacity of the knapsack equal to 2 and the weight of the current item is 1. Here, node $n_{2,4}$ gets removed since node $n_{2,3}$ has the same weight, and node $n_{2,5}$ gets removed since its weight is higher than the knapsack capacity.

to the state, where the algorithm can only create the second arc if it would not exceed the capacity of the knapsack.

Figure 3.1 shows how creating a new layer would work. The figure shows the case for a KP instance where the capacity is 2, and the weight of the current item is 1. In the figure, the dashed arcs indicate not taking the item, and the solid arcs indicate taking the item. In the left diagram of Figure 3.1, we see that for node $n_{1,1}$ and $n_{1,2}$, the next layer is already created, which resulted in the three nodes $n_{2,1}, n_{2,2}$ and $n_{2,3}$. Furthermore, we see two candidate nodes $n_{2,4}$ and $n_{2,5}$ with state 2 and 3 respectively. When generating these two candidate nodes, we first see that the state of node $n_{2,4}$ already exists in the next layer, namely node $n_{2,3}$ thus on the right of Figure 3.1 we redirect the arc to node $n_{2,3}$ and delete node $n_{2,4}$. Finally, node $n_{2,5}$ has an infeasible state since the knapsack would exceed capacity. Thus, we also delete this node and arc.

3.3.2. Iterative Refinement Approach

The second DD construction algorithm is the iterative refinement method, defined by Hadzic et al. [20]. This method starts with a fully relaxed DD and slowly unmerging (splitting) nodes. This method does not require the merging operator as for the top-down, but instead, we require a fully relaxed state for the DD $\mathcal{N}_0$ and a splitting operator $\text{split} : \mathcal{N} \rightarrow (\mathcal{N} \times \mathcal{N})$.

Algorithm 2 Iterative Refinement DD Compilation [20]

1: **procedure** ITERATIVE-REFINEMENT($\mathcal{N}_0$, $\mathcal{A}_0$, S)
2: $\quad$ $\mathcal{N} \leftarrow \mathcal{N}_0$
3: $\quad$ $\mathcal{A} \leftarrow \mathcal{A}_0$
4: $\quad$ **while** $|\mathcal{N}| < \text{S}$ **do**
5: $\quad\quad$ $n_{old} \leftarrow \text{find-node}(\mathcal{N})$
6: $\quad\quad$ $n_a, n_b \leftarrow \text{split}(n_{old})$
7: $\quad\quad$ $\mathcal{N} \leftarrow \mathcal{N} \cup \{n_a, n_b\}$
8: $\quad\quad$ Check all outgoing arcs and remove infeasible ones
9: $\quad\quad$ Update state information for all other nodes removing impossible nodes
10: $\quad$ **end while**
11: **end procedure**

The pseudocode for the iterative refinement method is described in Algorithm 2.

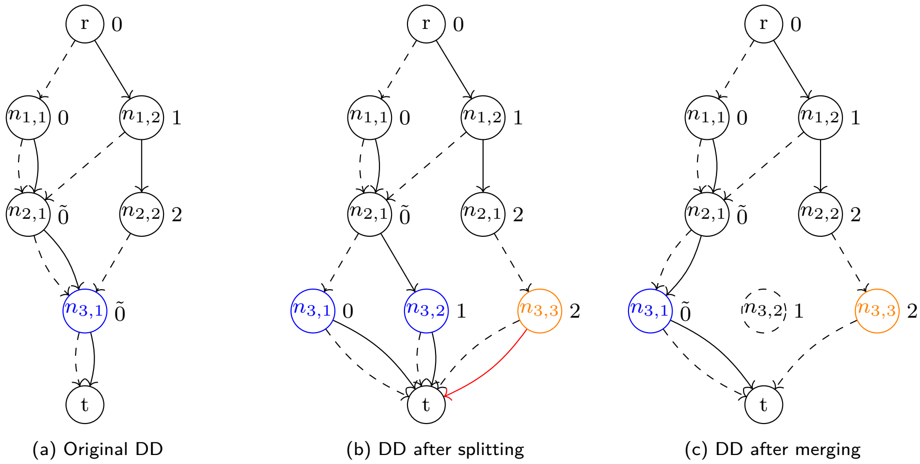


Figure 3.2: Example of splitting by merging in the unit knapsack problem. Here, we wish to split the blue node $n_{3,1}$ into two other nodes. We see that first, all incoming arcs of node $n_{3,1}$ are expanded to their exact values. Hereafter, the procedure merges the two blue nodes ($n_{3,1}$ and $n_{3,2}$). Finally, it removes the infeasible arcs.

The algorithm takes as input a fully relaxed DD defined by its nodes $\mathcal{N}_0$ and arcs $\mathcal{A}_0$. It additionally receives the maximum size S. If one has defined a merge operator, it can then be used to define both the initial DD ($\mathcal{N}_0$ and $\mathcal{A}_0$) and the splitting operator.

Castro et al. [8] illustrate that the merge operator allows the construction of a fully relaxed DD. They explain that the top-down construction method as seen in Algorithm 1, combined with a $\mathrm{W} = 1$ constructs such a fully relaxed DD.

The merge operator can be used to construct a splitting operator. The splitting operator works by temporarily creating exact nodes for all incoming arcs, and hereafter, these nodes are divided into two groups. These two groups hereafter merge repeatedly until one node in each group remains.

Example 3.2. For example, take the same unit cost KP as before. Figure 3.2a shows the DD after two splits. The algorithm chooses node $n_{3,1}$ to split. The algorithm first takes all incoming arcs to node $n_{3,1}$, this being $\{a \in \mathcal{A} | \, \mathrm{to}(a) = n_{3,1}\}$. These arcs then expand to their exact output state, as seen in Figure 3.2b. We additionally copy all outgoing arcs of $n_{3,1}$ to the new nodes. From these newly copied arcs, we remove those infeasible, indicated by red in Figure 3.2b. Finally, we split the new nodes into two groups: blue ($n_{3,1}$ and $n_{3,2}$) and orange ($n_{3,3}$). Each of these groups repeatedly merges until exactly one node exists in each group. The resulting DD is shown in Figure 3.2c, where $n_{3,2}$ is removed.

4

Method

As mentioned in Section 3, the goal of DFL is to find a predictor that, instead of minimizing the discrepancies of the predictions, tries to predict values that result in the optimal decisions by the optimizer.

In this work, we introduce two components to the DFL framework that have not previously been explored. Firstly, we construct dynamic relaxed DDs in which we regularly expand and shrink nodes based on their usage in optimal solutions. Second, we present three novel loss functions to train the predictor by gradient descent. The loss functions presented use the structure of the DD to calculate a differentiable loss.

In Section 4.1, we start by explaining the different loss functions; hereafter, in Section 4.2, we will discuss the different loss functions; finally, in Section 4.3, we will discuss some implementation-specific optimizations.

4.1. Learning the Predictor

For the predictor, we propose three gradient descent methods losses. The first method defines the loss at every node in such a way that the gradient descent causes the values of the optimal path to increase while the values of incorrect paths decrease. The other gradient descent method constructs a likelihood of selecting the optimal path. Hereafter, we explain how we derive a loss using this likelihood, which we use for gradient descent.

4.1.1. Node Wise Decision Loss

The Node Wise Decision loss is defined such that it pushes the predictor to make predictions that cause the optimizer to make optimal decisions. For the sake of simplicity, we abbreviate the Node Wise Decision loss as the Node loss. To do this, we look at all nodes in the DD where the optimizer makes the wrong decision. In other words, every node where the optimal path takes a different path than the optimized path.

Thus, we defined the first loss in terms of the predicted value obtained and the predicted value obtained when the optimizer uses the true values. We define this and all other losses for the maximization problem here. However, to use them for a minimization

problem, one must take the negative of the predicted and true values to make the loss work.

We define the corresponding loss in terms of the predictor's value for the actual optimal arc and the predicted value that the optimizer picks. Here, we denote the arc leaving node n, chosen by the optimizer based upon $\vec{\hat{c}}$ as $v_n^{(1)}(\vec{\hat{c}})$. Furthermore, we denote the predicted value for arc $a \in \mathcal{A}$ as $\hat{c}_a$. Thus, we define the value of the chosen arc as $\hat{c}_{v_n^{(1)}(\vec{\hat{c}})}$, and the predicted value for the actual optimal arc as $\hat{c}_{v_n^{(1)}(\vec{c})}$. Hence, we define the loss as:

$$\mathcal{L}(\vec{c}, \vec{\hat{c}}) = \sum_{n \in \mathcal{N}} \left(\hat{c}_{v_n^{(1)}(\vec{\hat{c}})} - \hat{c}_{v_n^{(1)}(\vec{c})} \right) \tag{4.1}$$

Looking at the loss of this function at a single node n, we see that the loss is defined in terms of the taken arc leaving node n based upon the predicted and true values. Namely, we see that the taken arc based on true values is positive and based upon the predicted values is negative. We note that if the predicted arc and the true arc are equivalent, then the two terms in Equation (4.1) cancel out, and the loss for that node is zero.

This loss causes the gradient to reduce the predictor output for incorrect sub-paths but increases the output for the true optimal sub-paths. Therefore, we would get positive gradients on true optimal arcs and negative gradients on the path with predicted weights.

The fact that the definition of this loss is a sum over all nodes where the optimizer makes an incorrect decision means that it also produces gradients for decisions that are not currently in the optimal path. This is beneficial since predictions should be such that decisions in all sub-paths would be optimal. The aforementioned fact also serves to regularize the predictor, preventing overfitting. However, when a sub-path is unlikely to be in the optimal path, it still generates a loss, which might cause the predictor to underfit.

Finally, the loss is scaled by how far the true optimal sub-path is from the found sub-path, starting in the current node. Thus, we would get an additional term in the loss function, and the resulting loss function is:

$$\mathcal{L}_{\text{node}}(\vec{c}, \vec{\hat{c}}) = \sum_{n \in \mathcal{N}} \left(\vec{c}^T v_n^*(\vec{\hat{c}}) - \vec{c}^T v_n^*(\vec{c}) \right) \cdot \left(\hat{c}_{v_n^{(1)}(\vec{\hat{c}})_1} - \hat{c}_{v_n^{(1)}(\vec{c})_1} \right) \tag{4.2}$$

4.1.2. Path Wise Decision Loss

We also introduce the Path Wise Decision loss, abbreviated as Path loss, as an alternative to the loss defined in Equation (4.2). This loss is the same as the Node loss, except that we do not sum over all nodes. Instead, we only sum over nodes on the true optimal path. Therefore, if we let all nodes along the true path be $\mathcal{P}$, we get the following loss function:

$$\mathcal{L}_{\text{path}}(\vec{c}, \vec{\hat{c}}) = \sum_{n \in P} \left(\vec{c}^T v_n^*(\vec{\hat{c}}) - \vec{c}^T v_n^*(\vec{c}) \right) \cdot \left(\hat{c}_{v_n^{(1)}(\vec{\hat{c}})_1} - \hat{c}_{v_n^{(1)}(\vec{c})_1} \right) \tag{4.3}$$

The loss defined in Equation (4.3) has the benefit over the previous loss that disused sub-paths would not constitute generating any gradients.

Example 4.1. For example, take the KP case where we have many items with a small weight and one item with the weight of the capacity of the knapsack. If, for certain inputs, the large item's true value is significantly more, then we would only want to capture this fact in our loss. However, the old loss would also try to make correct decisions for the sub-paths corresponding to not taking the large item. Thus, the model might spend too much predictive power on sub-problems that do not matter.

4.1.3. Probabilistic Path-Finding

For this technique, we assume the DD optimizer will probabilistically choose paths through the DD. The optimizer works top-down, meaning that it starts at the root node and calculates the probability of selecting an outgoing arc as part of the path based on how much the predictor values that arc plus the expected value we would get traveling to the node from which the edge originates.

We can thus define the expected value for this probabilistic pathfinder, as seen in Equation (4.4). Additionally, we define the probability of taking an edge as in Equation (4.5).

$$\mathbb{E}[n|\vec{\hat{c}}] = \begin{cases} 0 & \text{if } n = n_{end} \\ \sum\limits_{a \in A|\,\mathrm{from}(a)=n} P(a \in P | n \in P, \vec{\hat{c}}) \cdot (c_a + \mathbb{E}[\mathrm{to}(a)|\vec{\hat{c}}]) & \text{else} \end{cases} \tag{4.4}$$

$$P(a \in P|\,\mathrm{from}(a) \in P, \vec{\hat{c}}) = \frac{\sigma(\hat{c}_a + f[\mathrm{to}(a)|\vec{\hat{c}}])}{\sum_{a' \in A|\,\mathrm{from}(a')=\mathrm{from}(a)} \sigma(\hat{c}_{a'} + f[\mathrm{to}(a')|\vec{\hat{c}}])} \tag{4.5}$$

Here, $f[n|\vec{\hat{c}}]$ represents the approximated expectation. That is the expectation given that the value c_a, as seen in Equation (4.4), is approximated as $\hat{c}_a$. σ is an activation function we can choose to change the distribution type. Here, we choose the exponential function, which results in an exponential distribution. The exponential function would make Equation (4.5) equivalent to the softmax function.

Additionally, we note that the probability in Equation (4.5) is efficiently computable using the DD since Equation (4.5) is a recursive formulation over the DD. Thus, to calculate the solution, we can apply dynamic programming, which runs in linear time with respect to the number of arcs in the DD.

Now, we want to predict $\hat{c}_a$ in such a way that it maximizes the expected value of the root node r. Hence, we define the loss function as seen in Equation (4.6). We call this the Probabilistic Path-Finding (PPF) loss since it optimizes the expected reward of the probabilistic path-finding algorithm.

$$\mathcal{L}_{\mathrm{PPF}}(\vec{c}, \vec{\hat{c}}) = -\mathbb{E}[\mathrm{r}\,|\vec{\hat{c}}] \tag{4.6}$$

To conclude, the PPF loss can optimize the obtained total expected value of the probabilistic predictor. Getting useful gradients over the expected total value is substantially

different from other losses since these generally only calculate gradients corresponding to solving a single training problem to optimality. Meanwhile, the sum of the PPF for all training problems produces gradients that result in a higher expected total reward when using the probabilistic optimizer.

4.1.4. Maximum Likelihood Loss

In Section 3.2, we discussed the NCE loss based on maximum likelihood estimation. The idea for such a maximum likelihood loss relies on maximizing the probability of the true solution being picked based on a probabilistic optimizer. Mulamba et al. [7] use the probability function in Equation (4.7). However, Mulamba et al. [7] mention that to exactly calculate the probability function one would have to calculate the normalizing constant Z, which we restate in Equation (4.8). However, they mention that calculating Z would be too computationally expensive since it would require iterating over all solutions for the problem. Thus, they employ NCE to estimate these probabilities.

$$P(\vec{v}|\theta, \vec{x}) = \frac{1}{Z} \exp(\vec{\hat{c}}^T \vec{v}) \tag{4.7}$$

$$Z = \sum_{\vec{v} \in \mathcal{V}} \exp(\vec{c}^T \vec{v}) \tag{4.8}$$

However, by rewriting the definition of Z, we can transform it into a recursive function, which we can calculate using the respective DD. This is possible since we can replace the dot product inside the exponent with a sum of products, as seen in Equation (4.9). Finally, we rewrite the exponent of a sum to the product of exponents, as seen in Equation (4.10).

$$Z = \sum_{\vec{v} \in \mathcal{V}} \exp\left(\sum_{i=1}^{d} v_i \cdot c_i\right) \tag{4.9}$$

$$= \sum_{\vec{v} \in \mathcal{V}} \prod_{i=1}^{d} \exp(v_i \cdot c_i) \tag{4.10}$$

From Equation 4.10, we note that if two feasible solutions share part of their assignment, we can reuse part of their calculation. We can do this by factoring out the equivalent expression in the product.

Example 4.2. We will illustrate the previous point in the following example to better see why and how to reuse partially calculated values. Take the case where there exist two unique solutions: $\mathcal{V} = \{[0, 1, 1]^T, [1, 0, 1]^T\}$, with $\vec{c} = [1, 2, 3]^T$. Now the calculation for Z would be:

$$Z = (\exp(1 \cdot 2) \cdot \mathbf{exp(1 \cdot 3)}) + (\exp(1 \cdot 1) \cdot \mathbf{exp(1 \cdot 3)}) \tag{4.11}$$

The equation above shows that the calculations for both solutions in $\mathcal{V}$ share a common factor: $\exp(-1 \cdot 3)$. Hence, we can factor out the common factor to obtain the following equation:

$$Z = (\exp(1 \cdot 2) + \exp(1 \cdot 1)) \cdot \exp(1 \cdot 3) \tag{4.12}$$

By using DDs, we can efficiently factor this. Namely, when two paths in a DD share part of their path, they share factors in the calculation for Z corresponding to the respective arcs taken in the path.

Formally, we define the recursive definition of Z, as seen in Equation 4.13. Where, n is the current node, and a is the arc from node $\mathrm{from}(a)$ to $\mathrm{to}(a)$. Now, we recursively define Z as follows:

$$Z_n = \begin{cases} 1 & \text{if } n = \mathrm{t} \\ \sum_{a \in A | \,\mathrm{from}(a) = n} Z_{\mathrm{to}(a)} \cdot \exp\left(c_a\right) & \text{else} \end{cases} \tag{4.13}$$

The definition of Z_r (r being the root node) in Equation (4.13) is equivalent to the definition of Z, as seen in Equation (4.9) from [7]. We prove this in Appendix A.

Mulamba et al. [7] mention that if $t_\theta(\vec{x})$ produces the maximum value for the optimal assignment $v^*(\vec{c})$, then the deterministic optimizer will also pick the optimal solution. Thus, maximizing the likelihood function concerning θ corresponds to learning a θ that makes the intended true solution $v^*(\vec{c})$ be the best scoring solution [7].

Finally, we take as the loss the inverse of the product of all posteriors for all data points and call it the Maximum Likelihood (ML) loss, as seen in (4.14). We additionally take the logarithm of this product to turn the product into a sum and reduce numerical instability. This also allows us to define the loss in terms of a single data point, as seen in Equation (4.17).

$$\mathcal{L}(C, \vec{C}) = \log\left(\prod_{i}^{N} \frac{1}{P(v^*(C_i)|\theta, X_i)}\right) \tag{4.14}$$

$$= -\sum_{i}^{N} \log\left(P(v^*(C_i)|\theta, X_i)\right) \tag{4.15}$$

$$= -\sum_{i}^{N} -\log(Z_{\mathrm{r}}) + \left(v^*(C_i)^T \hat{C}_i\right) \tag{4.16}$$

$$\mathcal{L}_{\mathrm{ML}}(\vec{c}, \vec{\hat{c}}) = -\log(Z_{\mathrm{r}}) + \left(v^*(\vec{c})^T \vec{\hat{c}}\right) \tag{4.17}$$

4.2. Constructing the Relaxed Decision Diagram

In the following section, we discuss the method for constructing the approximate DDs for the DFL setting where the objective function coefficients might change over time. Firstly, we discuss how we decide which nodes to split and merge. Hereafter, we explain how the merging and splitting of nodes work.

Algorithm 3 depicts the construction method for a relaxed DD, based upon the iterative refinement method as discussed in Section 2.4. The algorithm refines the DD after every epoch, and here, the algorithm restructures the DD to better represent the optimal solution spaces of the problem. This refinement process works by splitting often used approximate nodes in two. Furthermore, the algorithm also merges nodes used infrequently in optimal paths to prevent the DD from becoming too big.

As mentioned before, the starting DD consists of a fully relaxed DD. In the case of the KP, the relaxed state would be zero since every node has one incoming weight zero arc from a node with weight zero.

The algorithm keeps track of the times it includes each node in an optimal path. The algorithm uses this value to determine which nodes should split or merge by splitting nodes used frequently and merging nodes used infrequently.

We perform a split every epoch for as long as any optimization step in the epoch produces an infeasible solution found for the dataset. The algorithm first picks an approximated node. The algorithm picks the approximate node with the highest count, e.g., the most included node in optimal paths. The algorithm then splits this chosen node in two, as seen in Algorithm 3.

The splitting procedure works the same as for the iterative refinement DD construction method, as explained in Section 3.3.2. It takes the chosen node n and splits it into two nodes n_a, n_b. The incoming arcs are split into two groups and are redirected to either n_a or n_b. Finally, we set the count of the two new nodes to half the count of the original node.

The grouping of these arcs is generally performed based on the designer's decisions, often based upon some heuristic [8]. For the KP, we group the arcs by their minimum weight plus the weight of the origin of the arc. In other words, we group the arcs based on the minimum weight of a new node, given that the current arc would be the only arc to that node. We conjecture that this is an effective approximation since bigger weights are more likely to prune infeasible arcs.

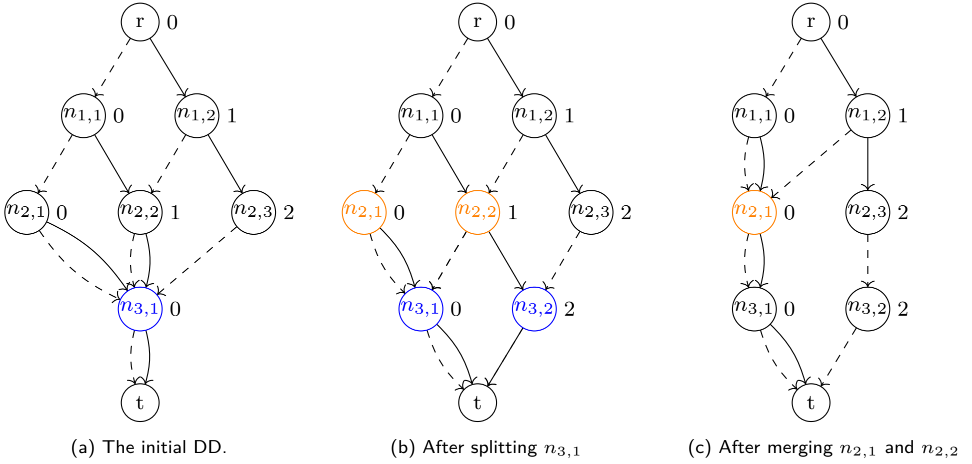
The merging operations take two nodes n_a and n_b as input and merge these in the following way. The operator redirects the incoming arcs of both nodes to point to the newly created merged node n. The new state of the node is calculated based on its parents. Hereafter, the outgoing arcs are updated. The algorithm here takes the arcs from the more relaxed node. For the KP, this would be the lower bound with the lowest minimum weight. Finally, we calculate the visitation count for the new node n by summing the count for nodes n_a, n_b.

Example 4.1. For example, Figure 4.1a shows a relaxed DD for a KP with unit weight and a capacity of two. After creating the new node $n_{3,2}$, the algorithm then takes the incoming arcs and calculates the possible output weight for each arc. This results in the following weights (for the arcs left to right) $[0, 1, 1, 2, 2]$. Now, the algorithm splits the group in two, creating a new node with weight 2, thus splitting into the following two groups: $[0, 1, 1] \rightarrow 0$ and $[2, 2] \rightarrow 2$. Finally, we generate the leaving arcs; here, the solid arc from node $n_{3,2}$ is pruned since the weight of $n_{3,2}$ plus the arc weight (1) would surpass the maximum weight.

Figure 4.1c, shows the resulting DD when merging nodes $n_{n2,1}$ and $n_{2,2}$. The algorithm first takes the node with minimum weight, node $n_{2,1}$. Hereafter, the arcs

Algorithm 3 The Dynamic DD Construction Algorithm

1: **procedure** DYNAMIC-COMPILE($\mathcal{N}_0$, $\mathcal{A}_0$, S)
2: $\theta \leftarrow initLearner()$
3: $\mathcal{N} \leftarrow \mathcal{N}_0$
4: $\mathcal{A} \leftarrow \mathcal{A}_0$
5: **while** training **do**
6: **for** $\vec{x} \in X$ **do**
7: $\vec{\hat{c}} \leftarrow t_\theta(\vec{x})$
8: $\mathcal{P} \leftarrow$ shortestPath$(DD, \vec{\hat{c}})$ ▷ Determine the optimal path
9: **for** $n \in \mathcal{P}$ **do**
10: $n.count \leftarrow n.count + 1$
11: **end for**
12: **end for**
13: **for** numSplitsToPerform **do** ▷ Hyper-parameter indicating the number of splits to perform each epoch
14: $n_{best} \leftarrow \mathrm{argmax}_{n_i \in \mathcal{P}}(Score(n_i))$
15: split(n_{best})
16: **if** $|\mathcal{N}| > \mathrm{S}$ **then**
17: $n_{bad\,1}, n_{bad\,2} \leftarrow$ findWorstPair$(\mathcal{N})$
18: $n_new \leftarrow n_{bad,1} \oplus n_{bad,2}$
19: $n_{old} \leftarrow \text{find-node}(\mathcal{N})$
20: $n_a, n_b \leftarrow \text{split}(n_{old})$
21: $\mathcal{N} \leftarrow \mathcal{N} \cup \{n_a, n_b\}$
22: Check all outgoing arcs and remove infeasible ones
23: Update state information for all other nodes removing impossible nodes
24: **end if**
25: $merge(DD, n_{bad\,1}, n_{bad\,2})$
26: **end for**
27: **end while**
28: **end procedure**



(a) The initial DD. (b) After splitting $n_{3,1}$ (c) After merging $n_{2,1}$ and $n_{2,2}$

Figure 4.1: An example DD before splitting (left), after splitting (center), and after merging (right). Here, the DD encodes a KP with a capacity equal to 2 and every item having weight 1. Dashed arcs relate to not taking the item. Meanwhile, full arcs indicate taking an item and thus adding its value to the objective and its weight to the total weight.

entering node $n_{2,2}$ are redirected to node $n_{2,1}$. Since node $n_{2,1}$ has a lower minimum weight than node $n_{2,2}$, there is no impact on the minimum weight of the resulting node. Finally, the arcs leaving node $n_{2,2}$ are deleted.

4.3. Additional Performance Improvements

Finally, we discuss additional methods that improve the performance of the loss calculations and the gradient calculation step. First, we explain how to improve the algorithm's speed by vectorizing. Hereafter, we discuss the log-sum-exp trick and how we apply it. Finally, we discuss derivations of the derivates of some functions to speed up the program.

4.3.1. Vectorized Calculation

The first is the vectorization of solution calculation, which works by doing all operations as a vector for the entire batch. This optimization is possible since calculating the value for a node always requires the same number of operations. The resulting algorithm will execute faster since it can use efficient vectorized instructions for calculating values. Furthermore, several libraries exist that perform further optimizations of these vectorized calculations.

4.3.2. Log-Sum-Exp Trick

In Section 4.1.4, we discussed the ML loss, as seen in Equation (4.17). After adding the logarithm to make this function more numerically stable, we note that the calculation of Z is still numerically unstable. Since the definition of Z still requires calculating the sum of possibly large numbers, which causes overflow errors. These errors reduce the

numerical precision of the calculation and thus make the definition more unstable [23].

$$\log(Z_n) = \begin{cases} 0 & \text{if } n = \text{t} \\ \log\left(\sum_{a \in A \mid \text{from}(a)=n} \exp\left(\log\left(Z_{\text{to}(a)}\right) + c_a\right)\right) & \text{else} \end{cases} \tag{4.18}$$

To remedy this, we start by placing the logarithm inside the recursive formulation of Z, as seen in Equation 4.18. Hereafter, we use the log-sum-exp trick, which reduces the numerical instability [23].

We do this by factoring out the largest argument and subtracting it from the rest. Since this would make Equation (4.18) unwieldy, we show it with the following example:

Example 4.1. Take the case where we apply the log-sum-exp on the set $\{10, 11, 20\}$ as seen in Equation (4.19). Now, instead of taking the logarithm of the sum of exponents, can we take the maximum (20) factor it out and subtract it from the rest as seen in Equation (4.20). This equation is more numerically stable since the exponents are now solely taken with small negative numbers.

$$\log\left(\sum_{a \in \{10,11,20\}} \exp(a)\right) \tag{4.19}$$

$$= 20 + \log(\exp(10 - 20) + \exp(11 - 20)) \tag{4.20}$$

4.3.3. Partial Derivatives of Loss Functions

The gradients necessary for the update step in gradient descent might be inefficient to calculate by an autograd engine because of the many steps required in iterating over the DD. To improve the implementation, we derive the partial derivatives for the parts that require iterating over the DD. Below, we discuss these partial derivatives for the recursive normalization constant Z and for the PPF loss $\mathcal{L}_{\text{PPF}}$.

For the ML loss, we only describe here the gradients for the normalizing constant Z, as seen in Equation (4.18), since the gradients for probability do not require iterating over the DD. The gradient for this equation is also a recursive definition over the DD. We state the resulting partial derivatives in Equation (4.21). To see the complete derivation of this equation, see Appendix C.

$$\nabla_{\vec{\hat{c}}} Z_n = \sum_{a \in \mathcal{A} \mid n = \text{from}(a)} \exp(\hat{c}^T e_a + \nabla_{\vec{\hat{c}}} \log(Z_{\text{to}(a)})) + \exp(\log(Z_{\text{to}(a)}) + \vec{\hat{c}}) \odot \vec{e_a} \tag{4.21}$$

For PPF, we split the partial derivatives over multiple functions since three recursive functions are required to find the partial derivatives for the loss, as shown in Equation (4.22). Here, the three recursive functions are for the expectation, as seen in Equation (4.23), the probability of choosing an arc (Equation (4.24)), and the predicted expectation (Equation (4.25)). We discuss the complete derivation of these partial derivatives in Appendix B.

$$\nabla_{\vec{\hat{c}}} \mathcal{L}_{\text{PPF}}(\vec{c}, \vec{\hat{c}}) = -\nabla_{\vec{\hat{c}}} \mathbb{E}[n_r \mid \vec{\hat{c}}] \tag{4.22}$$

$$\nabla_{\vec{\hat{c}}} \mathbb{E}[n|\vec{\hat{c}}] = \sum_{a \in \mathcal{A} | n = \text{from}(a)} (c_a + \mathbb{E}[\text{to}(a)|\vec{\hat{c}}]) \odot (\nabla_{\vec{\hat{c}}} \text{P}(a \in \mathcal{P} \,|\, \text{from}(a) \in \mathcal{P})) \tag{4.23}$$

$$\begin{aligned} \nabla_{\vec{\hat{c}}} \text{P}(a \in \mathcal{P} \,|\, \text{from}(a) \in \mathcal{P}) = & \frac{(\vec{e_a} + \nabla_{\vec{\hat{c}}} \mathbb{F}[\text{to}(a)|\vec{\hat{c}}]) \odot \exp(\hat{c}_a + \mathbb{F}[\text{to}(a)|\vec{\hat{c}}])}{\sum_{a' \in \mathcal{A} | n = \text{from}(a')} \exp(c_a + \mathbb{F}[\text{to}(a)|\vec{\hat{c}}])} \\ & - \frac{\exp(c_a + \mathbb{F}[\text{to}(a)|\vec{\hat{c}}]) \odot (\vec{e_a} + \nabla_{\vec{\hat{c}}} \mathbb{F}[\text{to}(a)|\vec{\hat{c}}]) \odot \exp(\hat{c}_a + \mathbb{F}[\text{to}(a)|\vec{\hat{c}}])}{\left(\sum_{a' \in \mathcal{A} | n = \text{from}(a')} \exp(\hat{c}_a + \mathbb{F}[\text{to}(a)|\vec{\hat{c}}])\right)^2} \end{aligned} \tag{4.24}$$

$$\begin{aligned} \nabla_{\vec{\hat{c}}} \mathbb{F}[n|\vec{\hat{c}}] = \sum_{a \in \mathcal{A} | n = \text{from}(a)} & (\hat{c}_a + \mathbb{F}[\text{to}(a)|\vec{\hat{c}}]) \odot \nabla_{\vec{\hat{c}}} \text{P}(a \in \mathcal{P} \,|\, \text{from}(a) \in \mathcal{P}) \\ & + \text{P}(a \in \mathcal{P} \,|\, \text{from}(a) \in \mathcal{P}) \odot (\vec{e_a} + \nabla_{\vec{\hat{c}}} \mathbb{F}[\text{to}(a)|\vec{\hat{c}}]) \end{aligned} \tag{4.25}$$

5

Experiments and Results

In this section, we provide the experimental setup used for empirical evaluation of the methods described in Section 4. We use the results of the experiments to verify how well the previously presented loss functions train the predictor and how using approximate DDs impacts this process. Hence, this section is split into the following two parts: first, we discuss the effectiveness of the different learning algorithms used with exact diagrams; then, we show how well these algorithms perform when combined with relaxed DDs.

5.1. Experimental Setup

In this section, we discuss the general setup of the experiments. We start by explaining the general specific settings of the datasets. Hereafter, we discuss the implementation of the algorithms as discussed in Section 4. Finally, we define how the datasets were generated.

All predictors consist of d linear regression models, with each model having p inputs and predicting a single value of our $\vec{c}$ vector. Every linear regressor gets every feature as input and produces a single value. Additionally, we also trained the intercept model parameter, also called the bias parameter. We implemented these regressor models as a fully connected linear layer in PyTorch[1]. Thus, these models have $(p)\cdot(d+1)$ parameters to train.

For all experiments, we ran all the losses as described in Section 4.1. These are the Node, Path, MLE, and PPF loss. We furthermore used the SPO+ loss from Elmachtoub and Grigas [1], as described in Section 3.1. Furthermore, for the small regret experiments, we also examined NCE and contrastive MAP from Mulamba et al. [7], as described in Section 3.2. We did not add these in other experiments since preliminary tests showed they underperformed in terms of regret compared to the other methods, and their implementation was not optimized to work with DD, so we did not look at them for runtime. Finally, we compared all methods with the baseline two-stage method,

[1] https://github.com/JopSchaap/Decision-Diagram-Focused-Learning

where for the regret experiments, we used the least squares solution, and for comparing the runtime, we used the Squared Error loss.

The experiments use the following hyperparameters during training unless specified otherwise for the specific experiment. First, we use a batch size that consists of the entire train set, thus equal to 700. Furthermore, we use the Adam optimizer with a learning rate of 0.01 and the default values for *betas* ($b_1 = 0.9, b_2 = 0.999$). Finally, we trained the models for 250 epochs.

We ran all the experiments on the DelftBlue high-performance cluster [24]. Here, every experiment consisted of training a single predictor. Each instance runs on a single CPU core with 5 GB of memory. The experiments are implemented in Python (version `3.10.12`) and will be made available on GitHub[2]. As a comparison to our DD method, we use the Gurobi solver, which uses an equivalent Mixed Integer Programming (MIP) model.

We experiment with synthetic datasets, which consist of 1000 instances. For the experiments, we split these datasets into a train and test set for evaluation. We perform this split such that the train set contains 70% (or 700 instances) of the data, and the test set the other 30% (or 300 instances). Finally, we generated a small and large instance of the dataset for the exact and approximate experiments. Next, we discuss the way we obtained the datasets for the experiments.

5.1.1. The Two-Stage Baseline

To speed up training, we initialize each model using the two-stage solution. Additionally, we compare both to the analytical solution of the two-stage method and against the runtime of the Squared Error loss. As such, in this section we define both the Squared Error loss and the analytical solution of it.

The squared error loss takes the squared distance between the predicted value $\vec{\hat{c}}$ and the actual value $\vec{c}$, as seen in Equation (5.1). This means that data points where the predictor makes a large error greatly impact the loss, while data points that are already close are relatively insignificant.

$$\mathcal{L}_{SE}\left(\vec{c}, \vec{\hat{c}}\right) = \left(\vec{c} - \vec{\hat{c}}\right)^2 \tag{5.1}$$

The resulting regressor model is called the least squares solution. It has the property that the model parameters are such that it obtains the lowest squared error loss possible on the training data.

5.1.2. Knapsack Problem

For the first experiment, we use the KP, where we artificially generated the values and features similarly as by Tang and Khalil [3]. Each KP in this dataset consists of a KP with $d = 48$ items and thus the same number of optimization coefficients $\vec{c}$. 16 of the items have a weight of $3, 5,$ and 7. The features for each problem consist of a vector of Gaussian distributed values $\vec{x} \sim \mathcal{N}_d(0, 1)$. Hereafter, we generate the item values as

[2] `https://github.com/JopSchaap/Decision-Diagram-Focused-Learning`

follows:

$$\vec{c} = \frac{1}{3.5^{\text{deg}}}\left((\frac{1}{\sqrt{d}}(\beta\vec{x}))^{\text{deg}} + 1 \right) \cdot \epsilon \cdot \vec{w} \tag{5.2}$$

Where β is a hidden matrix of shape $p \times d$, $d = 48$ is the number of items, ϵ is additional random noise, and $\vec{w}$ is the item weight vector. Here, we set the deg variable to 4 to introduce non-linearity in the value definition. Furthermore, we take $\epsilon\ \mathcal{U}(1-\bar{\epsilon}, 1+\bar{\epsilon})$, where we set the $\bar{\epsilon}$ variable to equal 0.1. Finally, we let β be a matrix where each element was drawn from the absolute value of a standard normal distribution $\beta \sim |\mathcal{N}_{d\times p}(0,1)|$.

Large Instances To increase the problem instances for the KP, we increased the number of items to choose from and the capacity of the knapsack. Namely, we took $d = 72$ possible items for each problem instead of $d = 48$, and we increased the capacity of the knapsack to 120.

5.1.3. Traveling Salesperson Problem

The second problem we use for running experiments is the Traveling Salesperson Problem (TSP). The goal for the TSP is to find the shortest tour through a graph, visiting every location in a graph exactly once. For this experiment, the predictor has to calculate the weight of taking an edge, after which the optimizer finds the shortest tour that passes through all vertices exactly once.

For this experiment, we generate TSP instances as in [3]. We start by generating eight points on a two-dimensional plane, which we do by taking two values from a Gaussian distribution $\mathcal{N}(0,1)$ summed with a uniform distribution $\mathcal{U}(-2,2)$. We then set the weight of each edge as the Euclidean distance between these points. These distances remain fixed for all data points in this dataset. Then, for every edge, we add an additional value, which we calculated using the following equation:

$$\frac{1}{3^{deg-1}}\left(\frac{1}{\sqrt{p}}(\mathcal{B}x_i)_j + 3\right)^{deg} \cdot \epsilon_{ij} \tag{5.3}$$

In Equation (5.3), we set the hyperparameter for the degree (deg) to four since preliminary testing showed two to be too easy. Furthermore, we have the multiplicative noise variable $\epsilon_{ij}\ \mathcal{U}(1-\bar{\epsilon}, 1+\bar{\epsilon})$, where we set the $\bar{\epsilon}$ variable to equal 0.1.

Large Instances For TSP, we increased the problem size by increasing the number of vertices in the TSP graph. Namely, we increased the graph size to eleven vertices instead of the original eight.

Decision Diagram Model We base the DD model for the TSP on the DD model for the single-machine scheduling problem, as proposed by de Weerdt et al. [25]. Here, we simplify the model, removing release times, deadlines, setup times, and the rejection. The resulting model is as follows. The state of each node is a tuple consisting of the vertices not yet visited in the current tour and the last vertex visited. Outgoing edges are the edges that go to nodes still in this set. After taking an edge, we remove the

associated vertex for this edge from the set and update the last visited node accordingly. If we let v be the current vertex, v' be the next vertex, and T be the vertices left to visit, then we would get the following recursive equation:

$$v^*(T, \mathrm{v}) = \begin{cases} 0 & \text{if } |T| = 0 \\ \min_{\mathrm{v}' \in T} v^*(T \setminus \{\mathrm{v}'\}, \mathrm{v}') + c_{\mathrm{v},\mathrm{v}'} & \text{else} \end{cases} \tag{5.4}$$

We note that for the TSP, one can choose any vertex as the start and end vertex in the tour since the tour is circular. This choice for the starting vertex affects the state of nodes in the DD and, as such, which arcs are possible in which nodes throughout the diagram. However, this decision does not matter for the size of the exact DD since the graph we are solving TSP over is defined as being fully connected.

MIP Model The choice of what model to use for the MIP representation has a severe impact, contrary to the KP case. Namely, we use the Dantzig-Fulkerson-Johnson (DFJ) [26] formulation since Tang and Khalil [3] state that it is faster to solve than alternative formulations for TSP.

5.2. Runtime Experiments

The first research sub-question relates to the runtime performance of the algorithms, as stated below. To answer this sub-question, we keep track of the algorithm's runtime during training.

> ***SQ 1:*** *What is the effect of using DDs in place of traditional solvers in terms of runtime performance?*

To achieve this, we alter the basic experimental setup as described in Section 5.1. The first change is that we only ran each experiment once since averaging over the number of epochs gives a good enough average. We also set a timeout of 30 minutes. Furthermore, we experimented with different batch sizes to determine the effect on runtime. Finally, for TSP, we experimented with graphs of different sizes.

5.2.1. Knapsack Problem

In this section, we look at the KP runtime experiments. We used these experiments to determine how fast all the proposed losses work and, in combination, how fast the problem-solving is using DDs. We start by looking at the runtime of the experiments in the default setting for the small experiments, as described in Section 5.1.2. After this, we look at the effect of the batch size hyperparameter on the runtime.

Table 5.1 shows the average runtime of the separate parts of the learning methods. This table shows that the loss calculation requires finding the optimal solution for all but the MSE loss. The first thing we observe from the table is that the DD has a substantially lower runtime than Gurobi when using SPO+. We also note that the runtime is dominated by calculating the loss for most losses. However, the exceptions to this are ML and PPF, which also spend a substantial part of their runtime in the backward step. Additionally, we remark that SPO+ has the lowest runtime of all DFL

Loss	Model	Loss (ms) mean	std	Backward (ms) mean	std
MLE	DD	1057.1	±109.8	12300.7	±171.8
Node	DD	2627.8	±35.3	5.9	±0.8
PPF	DD	2085.0	±33.6	8119.7	±230.2
Path	DD	1433.0	±18.1	6.2	±0.7
SPO+[1]	DD	737.2	±12.8	6.4	±0.8
SPO+[1]	Gurobi	3319.4	±46.3	7.9	±0.9

Table 5.1: The average runtime per epoch of the various parts of the different learning algorithms on the **KP**. Here, there exist 48 items with the capacity of the knapsack set to 60.

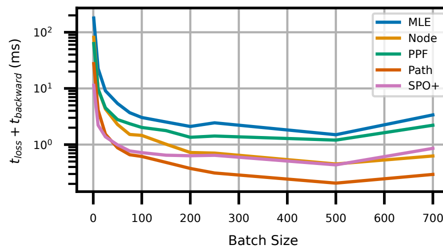


Figure 5.1: Runtime for the various methods with differing batch sizes on the **KP**. The times are the average time spent calculating the loss and performing the backward step during one epoch. There are 48 items with the knapsack capacity set to 60.

losses. Thus, the DD algorithm speeds up the training loop substantially, being especially efficient in combination with SPO+.

To illustrate the effect of the batch size on the runtime of the loss functions, Figure 5.1 shows the runtimes of the different losses over varying batch sizes. This figure shows that the DD-based methods ran substantially faster when the batch size increased. Additionally, all methods have a higher runtime with a small batch size than Gurobi with SPO+. However, with a larger batch size, the other techniques perform similarly to Gurobi. For the SPO+ and the Node loss, DD methods outperform Gurobi. To conclude, Figure 5.1 using a large batch size for DDs substantially reduces the training time.

5.2.2. Traveling Salesperson Problem

The second problem type is TSP, which we look at in this section to determine the runtimes in a different setting. We start by examining the runtime in the regular setup with the small instances as described in Section 5.1.3. We start by discussing the runtimes of the algorithms in different parts of the algorithm. Hereafter, we look at the effect of different batch sizes on the runtime. Finally, we look at how our methods scale to larger TSP graphs.

		Loss (ms)		Backward (ms)	
		mean	std	mean	std
Loss	Model				
MLE	DD	2022.0	±60.7	17492.6	±165.3
Node	DD	5151.7	±76.8	27.4	±1.1
PPF	DD	4016.2	±69.7	13919.6	±208.4
Path	DD	3268.5	±27.7	26.4	±1.3
SPO+[1]	DD	1682.3	±20.6	27.5	±1.5
SPO+[1]	Gurobi	50123.5	±3307.3	25.3	±3.0

Table 5.2: The average runtime per epoch of the various parts of the different learning algorithms on the **TSP**. Here, eight vertices exist in a fully connected graph, with each arc being symmetric.

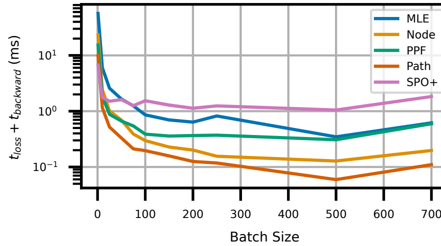


Figure 5.2: Runtime for the various methods with differing batch sizes, on the **TSP**. Here, the times are the average time spent calculating the loss during one epoch.

The resulting runtime for two parts of the training algorithm is stated in Table 5.2. This table shows similar results as Table 5.1, in that all DD-based methods have a substantially lower loss calculation time than Gurobi. However, the effect is significantly more pronounced than that of KP. We also again see that in Table 5.2, the MLE and PPF loss spend a long time in the backward step.

Figure 5.2 shows the time needed for calculating the loss and performing the backward pass for the different losses. The figure shows that for a batch size of 1, all losses that use a DD perform worse than those of Gurobi. However, with a larger batch size, we see that the DD-based methods perform better than Gurobi. We furthermore see that SPO+ with the DD solver runs faster than all other experiments. Finally, we see that PPF runs the slowest of all DD methods.

Similarly, Figure 5.3 shows the time needed for calculating the loss for different problem sizes. In this figure, the x-axis indicates the number of vertices in the TSP graph, and the y-axis is the time in seconds used to calculate the loss and perform the backward step. The results indicate that the DD-based solutions solve the smallest TSP instances faster than Gurobi, but the DD scales less well when the problem grows. We also note that the PPF loss scales the worst, being that the runtime increases the most over time.

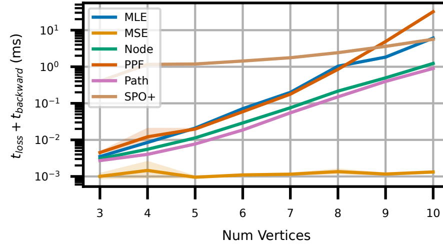


Figure 5.3: Runtime of the various methods with differing problem sizes for the **TSP**.

5.2.3. Discussion

To conclude, in both KP and TSP, we saw that the DDs outperformed Gurobi in terms of runtime. We also saw that the PPF and the MLE loss spent the most time in their respective backward pass. We conjecture this is the case since the backward pass requires propagating gradients for all input values. Therefore, we have to propagate a vector of length d, which thus would require extra computations for the length of the vector. However, we also conjecture that the implementation of this backward step could be substantially sped up since we focussed on correctness over speed while implementing.

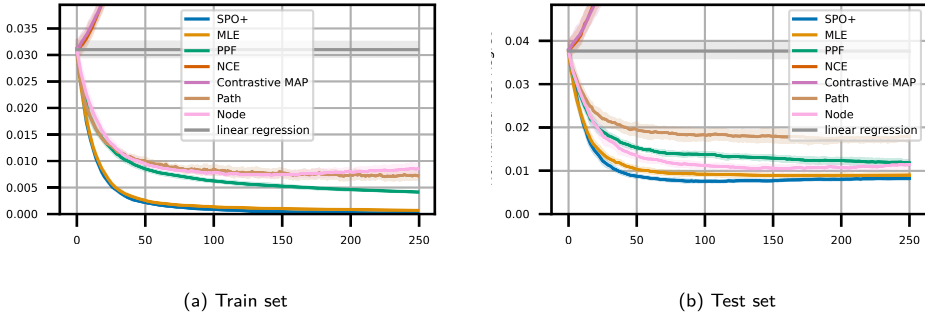
Furthermore, we saw that setting a large batch size had a positive effect on the runtime of the DD-based methods. Here, we conjecture that this effect has two main reasons, the first being that for smaller batch sizes, more time is spent running in a Python interpreter, which is notoriously slow compared to compiled languages [27–29]. Furthermore, we argue that a larger batch size allows for more efficient calculation since specialized vectorized CPU instructions can be used.

Finally, we looked at the effect of larger problem sizes. Here, we saw that the DD-based methods solved small instances extremely fast compared to Gurobi. However, we saw that as the problem increased in size, Gurobi scaled better. We conjecture that this is mainly caused by the decades of research that have gone into making Gurobi more efficient and scale better. Similarly, the fact our method is better for small problems is likely caused by the simplicity of our method combined with the reduced work caused by caching the DD structure between solver calls.

To conclude, the results show that for small problems, the DD-based methods outperform solver-based methods. We further show that the Node and Path losses have practically no impact on the loss compared to SPO+. Furthermore, we saw that the MLE and PPF loss spent most of the compute time in the backward step.

5.3. Loss Performance in Exact Decision Diagrams

The main challenge in DFL is to train the predictor so that its predictions result in good solutions to the optimized problem. In this section, we experiment with how well the proposed methods perform in terms of regret. We do this by comparing our methods with the SPO+. Here, we exclusively use exact DDs and look at both the regret on the train set and the regret on the test set. Thus, the experiments in this section look at the following sub-question:



(a) Train set

(b) Test set

Figure 5.4: The regret of the models during training with different losses on the **KP**. Here, there exist 48 items, with the capacity of the knapsack set to 60.

SQ 2: *How effective are the proposed losses at minimizing the obtained regret?*

This section is structured as follows: We start by examining the regret on the KP as discussed in Section 5.1.2; hereafter, we examine an alternative KP dataset. Finally, we look at the regret performance on TSP, with the dataset as explained in Section 5.1.3.

5.3.1. Knapsack Problem

Figure 5.4 shows the results of the learning methods as discussed in Sections 3 and 4.1. Specifically, Figure 5.4a shows the train regret obtained, and Figure 5.4b shows the test regret obtained for the different learning methodologies.

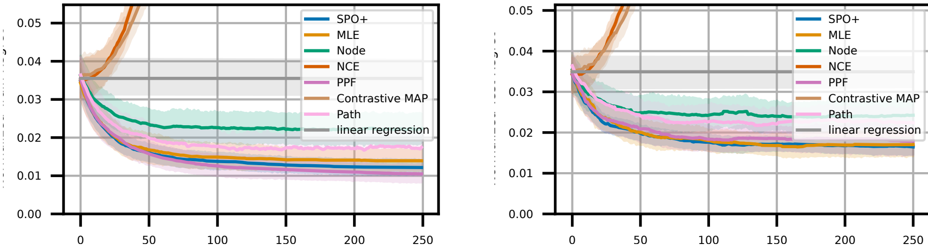
The resulting regret shows that all learning methods can achieve better train and test regret in comparison to the two-stage method, except for both contrastive losses. We also note that both SPO+ and ML can obtain a close to zero train regret, and thus, they outperformed the other DFL losses. Furthermore, Figure 5.4a shows that Node and Path losses perform very similarly in terms of train regret, but the Node loss performed substantially better in terms of test regret, as shown in Figure 5.4b. Conversely, the PPF loss improved the train regret but learned slowly after a while. To conclude, the SPO+ and ML losses performed the best of all losses.

5.3.2. Traveling Salesperson Problem

For the TSP, we again record the train and test regret during training. In this section, we show the results of the TSP experiments. We start by discussing the train regret. Hereafter, we show the test regret, and finally, we discuss these results.

Figure 5.5 shows both the train and test regret during training. The x-axis shows the epoch of the training cycle, and the y-axis depicts the regret obtained. The error bars indicate the 95% confidence interval.

Figure 5.5a shows that similarly to the KP all losses improve the regret over the two-stage method, except for both contrastive losses. We see in Figure 5.5a that PPF obtains the lowest regret for these other losses. However, we also note that the confidence intervals overlap for most losses. Finally, Figure 5.5b shows SPO+, ML and PPF



(a) The regret of the models, on the **train** set, during training with different losses.

(b) The regret of the models, on the **test** set, during training with the different losses.

Figure 5.5: The regret of the models during training with different losses on the **TSP**. Here, the problem is a fully connected graph with 8 vertices.

perform similarly on the test set. Thus, SPO+, ML, and PPF perform well on the TSP instances.

Finally, in comparison with the KP experiments, we note that the confidence interval for the TSP is substantially wider. This indicates that the effectiveness depends highly on the generated dataset instance. Thus, to reduce this, we could either increase the number of experiments or increase the size of the dataset.

5.3.3. Discussion

With all the results considered, we will make some final remarks concerning the exact experiments. First, we note that both contrastive losses failed to improve both the train and test regret from the two-stage solution; other than the contrastive losses, all other DFL methods outperform the two-stage approach for both problems. We also conclude that in both experiments, Node and Path losses perform substantially worse than the other methods. We also note that MLE and SPO+ perform similarly for both problems. Finally, we conclude that the PPF loss is theoretically able to perform better since it can obtain a better train regret in the TSP, but it fails in both experimental setups to obtain an actual better test regret.

First, we see from Figure 5.4 and Figure 5.5 that the DFL methods are able to improve the regret from the two-stage solution. We argue that this happens since the metric that the two-stage solution optimizes cannot properly capture the ensuing decisions properly.

From the KP experiments, we see that the train regret approaches zero for some losses. We conjecture that these losses reaching zero train regret show that separating the train set is possible for this particular problem. This is a case where SPO+ is especially effective since SPO+ computes gradients that push the predicted solution towards the true solution. This means that if there exists a configuration where all solutions that cause all predicted train solutions to be equal to their optimal solution, SPO+ is likely to find it.

Similarly, MLE also focuses on the true optimal solution and optimizes the probability that the probabilistic model would select it as the predicted solution. This, in turn, also

means that in problems where the optimal solution can be exactly predicted from the input features using the predictor, then MLE is likely to succeed.

Furthermore, we see that the Node and Path losses give the worst results of the DFL losses considered. We hypothesize that this is likely caused by these losses focusing too much on suboptimal solutions. This would cause the different models to spend too much of their representative power on solutions that are impossible to obtain anyway.

The experiments further show that PPF loss on TSP seems to produce the best results, at least on the training set. Meanwhile, on the test set, we see almost no difference between PPF, MLE and SPO+. We argue that this indicates that there could exist datasets on which PPF outperforms SPO+ since if the train set had been the exact distribution of data for these instances, then PPF would also have gotten a higher test regret.

Finally, we note that both PPF and MLE show promise for use as a DFL loss. We also conclude that SPO+ seems to comparably be very effective as a loss for instances for which it is possible to obtain a predictor that always predicts the optimal solution.

5.4. Loss Performance in Approximate Decision Diagrams

The problems where DFL techniques typically are used are NP-hard. Thus, there is no known solution to solve them efficiently. Relaxing techniques are beneficial in these cases due to their substantial speedup with the trade-off of allowing for small violations of the constraints in the solution found. Our solution presented in Section 4.2 is such an approach and the focus of this section. In this section, we look at the third sub-question:

> ***SQ 3:*** *How effective are approximate DDs for learning the predictor coefficients compared to using a linear relaxed mixed integer formulated model?*

For this research sub-question, we used the experimental setup as discussed in Section 5.1. However, we adapted the setup to use the approximate DD construction algorithm, as described in Section 4.2. We used the Gurobi model as before for the other relaxed solving methods, but we linearly relaxed the problem. For the datasets, we used the large datasets as described in Sections 5.1.2 and 5.1.3.

We measured the effectiveness of the construction algorithm by calculating the regret over the train and test set. To do this, we let the predictor produce the optimization coefficients $\vec{c}$, which get fed into the exact Gurobi model, which produces the solution for regret calculation. We do this since we want to measure the effectiveness of our algorithm concerning how well the predictor trains. However, using Gurobi on every epoch would cause the learning loop to run for a long time. Therefore, we only calculated the regret every ten epochs.

Additionally, we compare the losses we have used before with SPO+ loss using the linearly relaxed Gurobi model. However, for TSP, the DFJ model was not available as a relaxed version since it requires column generation [3]. Therefore, we use the Miller-Tucker-Zemlin [30] formulation implemented by Tang and Khalil [3].

The rest of this section is structured as follows: We start by giving some KP-specific experiment details for the approximate experiments. Hereafter, we discuss the results for the approximate KP. Then, we do the same for TSP. Finally, we give some concluding remarks and discuss the approximate results.

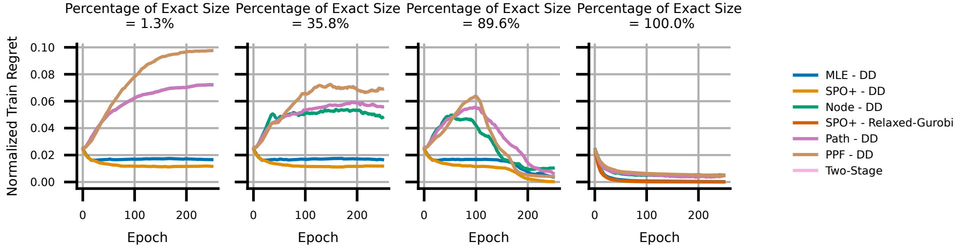


Figure 5.6: The **train** regret of the models during training with different losses on the **KP**. The KP consists of 72 items with a capacity of 120. Here, the percentage is the size of the relaxed diagram as a fraction of the exact size.

5.4.1. Knapsack Problem

For the KP, we relax the problem by allowing the solution to contain more weight than the capacity of the knapsack. Thus, for DDs, relaxing the state happens by removing some nodes and pointing all arcs towards that node to the node with the next highest weight. Thus, the relaxed state along a path through the relaxed DD will always be a smaller weight or equal weight to the true solution.

For the experiments, we initialize the DD fully relaxed with a minimal number of nodes. Thus, for the relaxed KP DD, the state of all nodes is zero, and all layers contain precisely one node. Furthermore, every non-terminal node had two outgoing arcs pointing to the node in the next layer. This works since there always exists a path through a layer that has weight, which is the path of taking no item.

To make the DD more exact, we have to define the splitting operator for the KP. This operator takes in a relaxed node and makes it less relaxed by splitting and pointing some incoming arcs to a newly created node. Specifically, the KP splitting operator calculates all possible states of the node based on its parents and their arc to the child. Hereafter, we sort the nodes based on their state, and we point the bottom half to the old node and the top half edges we point to the new node.

Similarly, the merging operator takes in one node and deletes this, making sure that all incoming arcs are pointed to another node, resulting in that node becoming more relaxed. The merging operator for the KP works in the following way: it starts by finding another node in the same layer which has the highest weight while being strictly less than the weight of the to-merge node. The merging operator then points all incoming arcs from the initial node to the found node. For instance, if two nodes with a state of 2 and 3 would get merged, the newly merged node would have the state of 2. Thus, any path that would previously go through the deleted node now has state 2, meaning that it might fit another item later on.

The resulting regret for the approximate KP experiments is shown in Figure 5.6 and Figure 5.6. Each figure contains multiple plots, each representing a different level of relaxation. Here, we denote the relaxation as a percentage of the maximum DD size and the size the exact DD would have. We also included the SPO$+$ loss with a linearly relaxed version of the Gurobi model, shown in the rightmost plot.

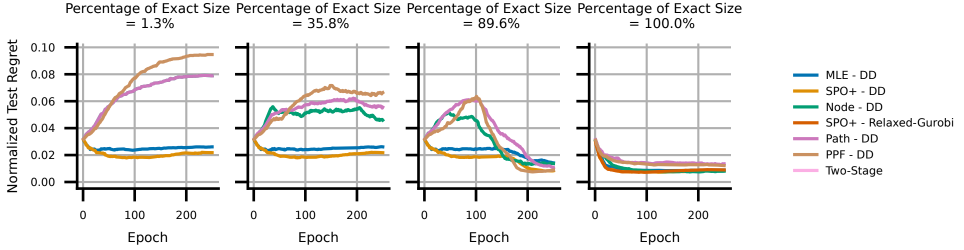


Figure 5.7: The **test** regret of the models during training with different losses on the **KP**. The KP consists of 72 items with a capacity of 120. Here, the percentage is the size of the relaxed diagram as a fraction of the exact size.

From the approximate results for the KP, we first note that there do not seem to be substantial differences between the train and test regrets. The only difference between the two figures is that the train regret is slightly lower than the test regret. We also note that the 100% figure looks substantially different from the second to last. We argue that this is caused by the fact that the 100% figure starts at the exact solution, while the second to last plot starts fully relaxed.

Furthermore, we see from Figure 5.6 and Figure 5.7 that MLE and SPO$+$ are always able to improve the regret, no matter how relaxed the DDs are. Here, it should be noted that the leftmost plot is the plot for the fully relaxed DD, which, in this case, means that the optimizer selects all items for which a non-negative weight was selected.

Secondly, we observe that for most relaxed DDs, the behavior of the regret is relatively similar. However, for the plots, the figures look relatively similar except for the second to last plot, in which partway through training, we suddenly see a drop in regret, after which all methods suddenly are able to reduce the regret substantially.

Finally, we note that we do not see this drop at the 100%. The difference between these two experiments is that for the second to last plot, the DD started completely relaxed and only grew over time, while the 100% plot was exact from the start.

5.4.2. Traveling Salesperson Problem

For the TSP instances, the relaxation would still enforce the length of the tour and the fact that the tour starts and ends in the same vertex. However, the tour could visit the same vertices twice and some vertices not at all. Thus, the resulting solution would be a closed walk instead of a tour.

The relaxed state of a TSP node allows it to have nodes in its T set that some paths through that node already might have visited. This also means that at certain layers, the size of this set is not determined by the layer, and the set might not be empty in the final layer.

The initialization of the TSP DD works in the following way: Every layer is initialized with the same number of nodes as the number of vertices in the TSP graph. The state of every node was that of being at their respective vertex and having to visit all other vertices except themselves. Furthermore, every node had an outgoing arc respective to

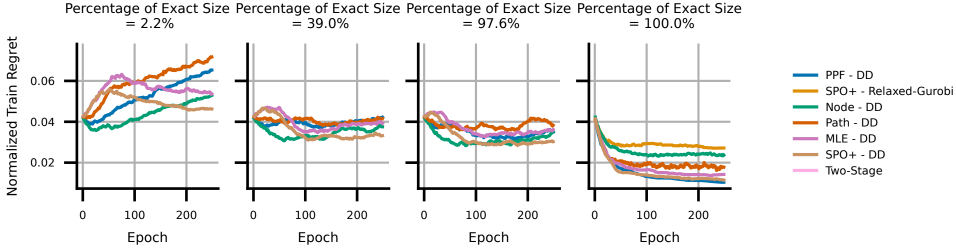


Figure 5.8: The **train** regret of the models during training with different losses on the **TSP**. The TSP consists of a fully connected graph of 11 vertices. Here, the fraction is the size of the diagram as a fraction of the exact size.

the edges leaving the vertex. Thus, for the initial DD, every arc had nine outgoing edges and nine incoming edges, and every layer had ten nodes, except the root and sink layers, which are equivalent to the exact case.

The TSP splitting operator first calculates all possible output states. Next, it chooses a random vertex that is in some but not both of the sets. Finally, this vertex was split by pointing all edges that would have contained this vertex in their set to one node and all others to the other. After doing this, we removed infeasible arcs and updated the nodes' state. Thus, the splitting operation removes infeasible paths by making the number of nodes left to visit more exact.

The merging operation for TSP inverts the process of splitting and creates a single node from two nodes. The merging operator takes two nodes with equivalent current vertex and then unions the sets of the to-visit vertices (T). For TSP, we select the second node to merge into based on the counts, so we pick the two nodes that can be merged and have the lowest count associated with them.

The resulting regret for the approximate experiments is shown in Figure 5.8 and Figure 5.9. Each figure contains multiple plots, each representing a different level of relaxation. Here, the relaxation is denoted as the percentage of the maximum DD size and the size the exact DD would have. We also included the SPO+ loss with a linearly relaxed version of the Gurobi model, shown in the rightmost plot.

We note from Figure 5.8 and Figure 5.9 that the train and test set behave similarly. Both figures show that for the most relaxed DDs, almost no method was able to learn anything. We also note that when the maximum DD size grows, the regret seems to decrease; thus, the methods train better. However, the exact plot shows a substantial difference in terms of regret in comparison to the second-to-last figure, which is almost exact.

Additionally, the figures show that the model with linearly relaxed Gurobi performs better or on par with the other relaxed models, except for the Node loss, which outperformed Gurobi in terms of test regret. Finally, we note that the Node loss seemed to obtain relatively good results with more relaxed DDs compared to the other methods, even though this does not seem to be the case for the exact case.

To conclude, the results for the approximate TSP experiments indicate that when

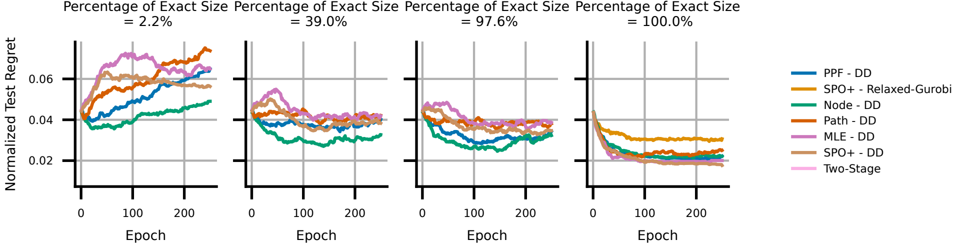


Figure 5.9: The **test** regret of the models during training with different losses on the **TSP**. The TSP consists of a fully connected graph of 11 vertices. Here, the fraction is the size of the diagram as a fraction of the exact size.

the size of the approximate DDs increases, the ability to learn also increases for all methods.

5.4.3. Discussion

To conclude, we see from the experiments that most losses fail to reduce the regret below the two-stage solution. However, on the KP, SPO+ and MLE were able to improve the regret even for very relaxed DD sizes. Furthermore, we also note that for the KP, at a certain point, all methods started to learn very well, coming close to the regret obtained with the exact DD. In this section, we explain why we think these things happen.

First, we see that for the KP, even for the fully relaxed variant, SPO+ and MLE are able to reduce the train regret. Here, it should be noted that for the KP, the relaxed predicted solution picks all items that are not negative. This means that if all items have a predicted weight greater than zero, they are picked. Here, we conjecture that these two methods still learn in these situations because of two reasons. Firstly, the true solution is used as the target to train towards, and this is obtained using an exact solver. Secondly, even for a fully relaxed DD, it is possible to set the predicted values so that the true optimal solution is picked.

We conjecture that being able to optimize to the exact true solution helps since even if the predicted solution would be the solution of taking all items, an update step can still be made that would improve the prediction. To give an intuition of why we conjecture this happens, we examine what happens to the SPO+ loss when the predicted solution is picking all the items, in other words, that $\hat{\vec{v}} = \vec{1}$. Filling some predicted vector $\hat{\vec{c}}$ in the SPO+ loss gives the following:

$$\mathcal{L}_{SPO+}(\hat{\vec{c}}, \vec{c}) = \left(2\hat{\vec{c}} - \vec{c}\right)^T \tilde{v}^* \left(2\hat{\vec{c}} - \vec{c}\right) - \left(2\hat{\vec{c}} - \vec{c}\right)^T v^*(\vec{c}) \tag{5.5}$$

$$\mathcal{L}_{SPO+}(\hat{\vec{c}}, \vec{c}) = \left(2\hat{\vec{c}} - \vec{c}\right)^T (\vec{1} - v^*(\vec{c})) \tag{5.6}$$

$$\nabla_{\hat{\vec{c}}} \mathcal{L}_{SPO+}(\hat{\vec{c}}, \vec{c}) = 2(\vec{1} - v^*(\vec{c})) \tag{5.7}$$

Thus, we see that every item that gets picked in $v^*(\vec{c})$ produces zero gradient, but items that do not get picked get a gradient of 2. Thus, we see that using gradient

descent, items that should not be in the knapsack get reduced in value as long as their value is at least half of the true value. When their value drops below this point, it will no longer be picked, even by the fully relaxed DD.

A similar reasoning holds for the MLE loss; namely, we could always increase the probability of an optimal solution by making the coefficients that adhere to that solution bigger and reducing the size of the other values.

Additionally, for the KP, one can always pick coefficients such that any solution is the optimal solution. Namely, we can set the value coefficients for the items we wish to add as a positive value and the items we do not wish as negative.

Meanwhile, this is not true for the TSP instances since we examine symmetric TSP. This breaks the previous statement since, for the fully relaxed case, when the optimizer chooses a second edge, it could, for its third choice, always decide to take the reverse edge. The DD could, for instance, always choose to do one half of the path twice, with it choosing the half that has a lower cost.

Furthermore, we reason that for the relaxed DDs, the other losses give bad performance since they are not trying to select the exact true solution but instead are trying to optimize the solution that would give the best result given the relaxed optimizer. However, it seems that this is a bad assumption when you want to find the true exact solution. In other words, while MLE and SPO+ try to let the predictor predict $\vec{c}^T v^*(\vec{c})$, do the other losses optimize towards making $\vec{c}^T \tilde{v}^*(\vec{c})$.

Finally, we note that at the start, all relaxed DDs have about the same learning curve shape. Only after about 50 to 100 epochs do we see the larger DDs outperform the smaller DDs. We argue this is the case since all approximate DDs are initialized fully relaxed. Thus, it would take a while before the sizes of the approximate DDs are actually different.

6

Conclusion

Decision-Focused Learning (DFL) techniques aim to train a predictor such that the downstream optimization problem obtains a high total value. Contrasting the DFL techniques, two-stage techniques train on a regression task that matches the input values as much as possible. However, the objective the two-stage solution maximizes does not reflect the downstream optimization problem. As such, the two-stage method underperforms in comparison to DFL methods.

DFL techniques for combinatorial problems usually use black box solvers to obtain the current predicted optimal solution and the true optimal solution. However, such methods ignore large parts of the solution space, which might still contain reasonable solutions. Decision Diagrams (DD) provide a fundamental solution to this problem since they encode all feasible solutions within their structure. Thus, the research question we answered in this work is:

> ***RQ:*** *How can DDs be (re)used in the DFL setting to compute update steps during learning that minimize regret?*

We make four main contributions to answer our research question. First, We show that reusing DDs gives a substantial computational speedup over using a solver. Additionally, we present multiple novel losses that use the characteristics of DDs as an integral step during training. Finally, we propose a novel DD relaxation method for the DFL setting, to further enhance the run speed.

Our first contribution is in using DDs to obtain a substantial computational speedup. We empirically showed that DD-based methods allow for caching large parts of the optimization step. Specifically, for the regular instances, we saw an increase in speed for both the Knapsack Problem (KP) and the Traveling Salesperson Problem (TSP). Here, the instances of the TSP consisted of a fully connected graph of 8 vertices. For these TSP instances, the DD solution ran about 29.9 times as fast, taking 1708.8 ms instead of 50148.8 ms Gurobi used. The DD techniques were similarly able to increase the speed for the KP. The KP instances consisted of filling a knapsack where the optimizers

had to pick the best from 42 items while remaining in the capacity. The DD technique ran about 4.5 faster than Gurobi, taking 737.2 ms instead of 3319.4 ms per epoch. Furthermore, these results did not require any compromise on quality for the produced solutions.

The three presented novel losses, node and path wise decision, MLE and PPF, use DDs as an integral part, which allows reasoning over entire solution spaces. We showed that, since they reason over the entire solution space they theoretically improve over previous methods, and we demonstrate that they achieve comparable results to state-of-the-art methods.

Lastly, we proposed a novel method to use relaxed DDs for the DFL paradigm. We validated that this relaxation method allows for training beyond the regret of the two-stage solution, even when using relaxed DDs. However, we found that whether losses outperform the two-stage solution when training with an approximate DD depends on the specific problem being trained. For instance, for the KP, the SPO+ and MLE loss outperformed the two-stage method even for very relaxed DDs. We conjecture that this is the case since for this specific problem the SPO+ and MLE loss make the ground truth solution more likely even when the predicted solution is relaxed. Meanwhile, for TSP we did not see this happening and instead the all losses only perform slightly better for than the two-stage method, and then only for DD sizes close to the exact size. We conjecture that the difference here depends on whether the DFL loss trains to particular targets, and whether using it in combination with the relaxed problem gives useful gradients.

Ultimately, DDs help in the DFL paradigm by efficiently encoding solutions in their structure, which we exploited. This allows for encompassing valuable information about multiple solutions at the same time in the loss. Thus, DDs encompass an exciting line of research that produces a novel look at how to reason over solutions spaces.

6.1. Future Work

In this work, we have shown several methods for using DDs for the DFL task. DDs are especially beneficial for DFL since they efficiently describe the entire solution space. As such, one can still expand the method presented. This section provides some research directions we believe are promising to explore further for Decision Diagram Focused Learning.

During this work, we focussed on two NP-hard problems, namely the KP and the TSP. The losses, as previously described, work on these problems. However, they are not limited to them. For a start, they are compatible on other NP-hard problems, with the only requirements being that the problem can be solved using dynamic programming. The losses also apply to problems for which polynomial algorithms are known. For instance, the techniques described here apply to shortest-path problems. However, special care has to be taken into consideration regarding how the graph changes based on the predicted values since the dynamic programming computational graph depends on the objective values.

As an alternative to the use of relaxed DDs, restricted DDs might be used. Restricted DDs work similarly to relaxed DDs except that while relaxed DDs introduce extra infeasible solutions, the restricted DD remove feasible solutions. In other words, where the relaxed DD is a superset of the solution space, the restricted DD is a subset

of the solution space.

These restricted DDs could be constructed from the known true solutions. This could, for instance, be done by using some external solver to find all true optimal solutions for the problem. All these solutions would form paths through the DD. These paths combined form a restricted DD. Finally, one should merge identical subdiagrams to reduce the DD's size and discover new paths that share parts of both parents. Then, the losses, as discussed in this work, could be applied to these restricted DDs. This alternative DD method has the theoretical benefit that the true optimal solution is the same during training and testing. Additionally, it will not try to learn infeasible solutions. Finally, solutions which are always suboptimal are also ignored during training. Thus, using restricted DDs would both speed up the algorithm, and also focuses more on the solutions already seen as optimal.

We also conjecture that with such constructed restricted DDs, one could add relatively close paths, such as all solutions where one decision variable is different, which would make the DDs represent a bit more solutions, theoretically improving the generalization ability of the system.

Finally, we also note that for many problems, the size of exact DDs is determined by the variable ordering used. Finding the optimal ordering, in general, is NP-complete [31]. However, we only need to create the DD once and can reuse it hereafter. Thus, it would theoretically be beneficial to spend some time upfront to discover the best variable ordering, or at least better than random. Alternatively, with the relaxed DDs, one could have an ensemble of DDs and choose a random DD from each iteration of this ensemble.

These are just some directions that can be explored with Decision Diagram Focussed Learning. We believe DDs prove to be an interesting new line of research for DFL.

Bibliography

[1] A. N. Elmachtoub and P. Grigas, "Smart "Predict, then Optimize"," *Management Science*, vol. 68, no. 1, pp. 9–26, 2022. DOI: 10.1287/mnsc.2020.3922.

[2] Y. Bengio, "Using a Financial Training Criterion Rather than a Prediction Criterion," *International Journal of Neural Systems*, vol. 8, no. 4, pp. 433–443, 1997. DOI: 10.1142/S0129065797000422.

[3] B. Tang and E. B. Khalil, *PyEPO: A PyTorch-based End-to-End Predict-then-Optimize Library for Linear and Integer Programming*, 2023. DOI: 10.48550/arXiv.2206.14234. arXiv: 2206.14234.

[4] E. Demirović et al., "An Investigation into Prediction + Optimisation for the Knapsack Problem," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, L.-M. Rousseau and K. Stergiou, Eds., 2019, pp. 241–257. DOI: 10.1007/978-3-030-19212-9_16.

[5] Ö. Elçi and J. Hooker, "Stochastic Planning and Scheduling with Logic-Based Benders Decomposition," *INFORMS Journal on Computing*, vol. 34, no. 5, pp. 2428–2442, 2022. DOI: 10.1287/ijoc.2022.1184.

[6] J. Mandi and T. Guns, "Interior Point Solving for LP-based prediction+optimisation," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 7272–7282.

[7] M. Mulamba, J. Mandi, M. Diligenti, M. Lombardi, V. Bucarey, and T. Guns, "Contrastive Losses and Solution Caching for Predict-and-Optimize," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, 2021, pp. 2833–2840. DOI: 10.24963/ijcai.2021/390.

[8] M. P. Castro, A. A. Cire, and J. C. Beck, "Decision Diagrams for Discrete Optimization: A Survey of Recent Advances," *INFORMS Journal on Computing*, vol. 34, no. 4, pp. 2271–2295, 2022. DOI: 10.1287/ijoc.2022.1170.

[9] A. A. Cire and W.-J. Van Hoeve, "Multivalued Decision Diagrams for Sequencing Problems," *Operations Research*, vol. 61, no. 6, pp. 1411–1428, 2013. DOI: 10.1287/opre.2013.1221.

[10] H. M. Salkin and C. A. De Kluyver, "The knapsack problem: A survey," *Naval Research Logistics Quarterly*, vol. 22, no. 1, pp. 127–144, 1975. DOI: 10.1002/nav.3800220110.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. 2016.

[12] A. Cauchy, "Méthode générale pour la résolution des systemes d'équations simultanées," *Comptes Rendus Hebd. Séances Acad. Sci*, vol. 25, pp. 536–538, 1847. Translated by R. J. Pulskamp, Xavier University, 2010.

[13] I. Panageas and G. Piliouras, *Gradient Descent Only Converges to Minimizers: Non-Isolated Critical Points and Invariant Regions*, 2016. DOI: 10.48550/arXiv.1605.00405. arXiv: 1605.00405.

[14] M. V. Pogančić, A. Paulus, V. Musil, G. Martius, and M. Rolinek, "Differentiation of blackbox combinatorial solvers," in *International Conference on Learning Representations*, 2020.

[15] A. N. Elmachtoub, J. C. N. Liang, and R. Mcnellis, "Decision Trees for Decision-Making under the Predict-then-Optimize Framework," in *Proceedings of the 37th International Conference on Machine Learning*, 2020, pp. 2858–2867.

[16] E. Demirović et al., "Predict+Optimise with Ranking Objectives: Exhaustively Learning Linear Functions," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 2019, pp. 1078–1085. DOI: 10.24963/ijcai.2019/151.

[17] B. Tang and E. B. Khalil, "CaVE: A Cone-Aligned Approach for Fast Predict-then-optimize with Binary Linear Programs," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, B. Dilkina, Ed., vol. 14743, 2024, pp. 193–210.

[18] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann, "A Constraint Store Based on Multivalued Decision Diagrams," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed., 2007, pp. 118–132. DOI: 10.1007/978-3-540-74970-7_11.

[19] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. N. Hooker, "Discrete Optimization with Decision Diagrams," *INFORMS Journal on Computing*, vol. 28, no. 1, pp. 47–66, 2016. DOI: 10.1287/ijoc.2015.0648.

[20] T. Hadzic, J. N. Hooker, B. O'Sullivan, and P. Tiedemann, "Approximate Compilation of Constraints into Multivalued Decision Diagrams," in *Principles and Practice of Constraint Programming*, P. J. Stuckey, Ed., 2008, pp. 448–462. DOI: 10.1007/978-3-540-85958-1_30.

[21] U. Sadana, A. Chenreddy, E. Delage, A. Forel, E. Frejinger, and T. Vidal, "A survey of contextual optimization methods for decision-making under uncertainty," *European Journal of Operational Research*, 2024. DOI: 10.1016/j.ejor.2024.03.020.

[22] M. Gutmann and A. Hyvärinen, "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, pp. 297–304.

[23] P. Blanchard, D. J. Higham, and N. J. Higham, "Accurately computing the log-sum-exp and softmax functions," *IMA Journal of Numerical Analysis*, vol. 41, no. 4, pp. 2311–2330, 2021. DOI: 10.1093/imanum/draa038.

[24] D. H. P. C. C. (DHPC), *DelftBlue Supercomputer (Phase 2)*, Manual, 2024.

[25] M. de Weerdt, R. Baart, and L. He, "Single-machine scheduling with release times, deadlines, setup times, and rejection," *European Journal of Operational Research*, vol. 291, no. 2, pp. 629–639, 2021. DOI: 10.1016/j.ejor.2020.09.042.

[26] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a Large-Scale Traveling-Salesman Problem," *Journal of the Operations Research Society of America*, vol. 2, no. 4, pp. 393–410, 1954. DOI: 10.1287/opre.2.4.393.

[27] R. Pereira et al., "Energy efficiency across programming languages: How do energy, time, and memory relate?" In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017, 2017, pp. 256–267. DOI: 10.1145/3136014.3136031.

[28] Z. Alomari, O. E. Halimi, K. Sivaprasad, and C. Pandit, *Comparative Studies of Six Programming Languages*, 2015. arXiv: 1504.00693v1.

[29] M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics," *BMC Bioinformatics*, vol. 9, no. 1, pp. 1–9, 2008. DOI: 10.1186/1471-2105-9-82.

[30] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer Programming Formulation of Traveling Salesman Problems," *J. ACM*, vol. 7, no. 4, pp. 326–329, 1960. DOI: 10.1145/321043.321046.

[31] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 993–1002, 1996. DOI: 10.1109/12.537122.

A

Proof for Equivalence Equation (4.13) and Equation (4.10)

In this section, we proof the equivalence of Equation (4.13) and Equation (4.10). We restate these equations again in order to ease reading, see Equation (A.1) and Equation (A.2) This is necessary, since, Equation (4.13) can efficiently be computed using DDs allowing us to.

Proof. To prove: $Z_\mathrm{r} = \tilde{Z}_\mathrm{r}$, where:

$$Z_n = \begin{cases} 1 & \text{if } n = \mathrm{t} \\ \sum_{a \in A \mid \mathrm{from}(a) = n} Z_{\mathrm{to}(a)} \cdot \exp\left(c_a\right) & \text{else} \end{cases} \tag{A.1}$$

$$\tilde{Z} = \sum_{\vec{v} \in \mathcal{V}} \prod_{i=1}^{|\vec{v}|} \exp(-v_i \cdot c_i) \tag{A.2}$$

Step 1. Defining a Multivalued-DD Path For multivalued DDs, the relation between the original arcs and the solution vector $\vec{v}$ are determined by the coefficients index of the corresponding arc (and not on the layer, which is the case for regular DDs). Thus, to get the solution vector $\vec{v}$ from a path $\mathcal{P}$, we have to calculate the following:

$$\vec{v} = \sum_{a \in \mathcal{P}} \vec{e_a} \tag{A.3}$$

Where $\vec{e_a}$ is the standard basis vector, which is zero everywhere except for index a where it is one. Thus, we get:

$$\tilde{Z} = \sum_{\vec{v} \in V} \prod_{a \in P(\vec{v})} \exp(-\vec{e_a}^T \vec{c}) = \sum_{\vec{v} \in V} \prod_{a \in P(\vec{v})} \exp(c_a) \tag{A.4}$$

Finally, we use the fact that DDs encode all feasible solutions to a problem as a path to the graph. This allows us to define an equation for $\tilde{Z}$ which is valid for every node in the graph. We do this by letting $\rho(n)$ be all the paths from node n to the sink node t. By the definition of DDs equivalent to the solution set for the sub-problem, defined by the sub-diagram of with the root node being n.

$$\tilde{Z}_n = \sum_{\vec{p} \in \rho(n)} \prod_{i=0}^{i<|\vec{p}|} \exp(-c_{p_i}) \tag{A.5}$$

$$\tilde{Z}_{\mathrm{r}} = \tilde{Z} \tag{A.6}$$

Step 2. Proof by Induction We will now use mathematical induction to prove that the recursive formulation for any node n is equal to the non-recursive definition $\tilde{Z}_{\mathrm{r}}$.

$$Z_{\mathrm{r}} = \sum_{a \in \mathcal{A} |\, \mathrm{r} = \mathrm{from}(a)} Z_{\mathrm{to}(a)} \cdot \exp(c_a) = \sum_{\vec{p} \in \rho(\mathrm{r})} \prod_{i=0}^{i<|P|} \exp(-c_{p_i}) = \tilde{Z}_{\mathrm{r}} \tag{A.7}$$

Base Case We take as a base case Z_{t}, note here that the base case is for the sink node, as such there exists exactly one path to the sink node t, which has length zero. Thus, $\rho(\mathrm{t})$ consists only of the zero length path []. Therefore, we get $A_{\mathrm{t}} = \emptyset$ By definition of Equation (A.1):

$$Z_{\mathrm{t}} = 1 \tag{A.8}$$

For the value of $\tilde{Z}_{\mathrm{t}}$ we use the fact that the path corresponding to the terminal node is the path of length zero []. By the definition of $\tilde{Z}$, we get:

$$\begin{aligned} \tilde{Z}_{\mathrm{t}} &= \sum_{\vec{p} \in \rho(\mathrm{t})} \prod_{i=0}^{i<|\vec{p}|} \exp(-c_{p_i}) \\ &= \prod_{i=0}^{i<|[]|} \exp(-c_{p_i}) \\ &= 1 \end{aligned} \tag{A.9}$$

Induction Step For the induction step, we wish to proof that for node n in layer k, that $Z_n = \tilde{Z}_n$, given that $Z_m = \tilde{Z}_m$, for any node m in layer $k+1$.

Here, we start with the formulation for $\tilde{Z}_n$ and proof that this is equivalent to Z_n. To perform this step, we note that $n \neq t$, since this is handled in the base case, and thus that $|p| \geq 1$.

Firstly, we define the set consisting of all first arcs in any of the path:

$$\phi(n) = \{a \in \mathcal{A} |\, \mathrm{from}(a) = p_0, \mathrm{to}(a) = p_1, \vec{p} \in \rho(n)\} \tag{A.10}$$

We now use the definition of $\phi(n)$ to factor out the first arc in the path calculation.

$$\tilde{Z}_n = \sum_{\vec{p}\in\rho(n)} \prod_{i=0}^{i<|\vec{p}|} \exp(-c_{p_i}) \tag{A.11}$$

$$= \sum_{\vec{p}\in\rho(n)} \exp(-c_{p_0}) \prod_{i=1}^{i<|\vec{p}|} \exp(-c_{p_i}) \tag{A.12}$$

$$= \sum_{a\in\phi(n)} \exp(-c_a) \sum_{\vec{p}\in\rho(\mathrm{to}(a))} \prod_{i=0}^{i<|\vec{p}|} \exp(-c_{p_i}) \tag{A.13}$$

$$= \sum_{a\in\phi(n)} \exp(-c_a)\tilde{Z}_{\mathrm{to}(a)} \tag{A.14}$$

Finally, we just need to proof that:

$$\phi(n) = \{a \in \mathcal{A} | n = \mathrm{from}(a)\} \tag{A.15}$$

From the definition of ϕ, we note that it must be a subset of A_n, since any path to t starting in node n should start with an arc leaving node n, and thus should be in the set of arcs leaving n, or $\{a \in \mathcal{A}|n = \mathrm{from}(a)\}$.

Furthermore, by definition every node in a DD should have a path to node t, otherwise the node should be removed from the DD. Thus, every outgoing arc from node n should point to a node that has no path to node t. Which means that there exists a path from n that uses that arc.

$$\nexists a \in \{a \in \mathcal{A}|n = \mathrm{from}(a)\} \rightarrow a \notin \phi(n) \tag{A.16}$$

Therefore, we conclude that:

$$\phi(n) = \{a \in \mathcal{A}|n = \mathrm{from}(a)\} \tag{A.17}$$

Which thus means that, by the principles of induction that $Z_{\mathrm{r}} = \tilde{Z}_{\mathrm{r}}$. □

B

Derivation of Gradients for Probabilistic Path Finding Loss

In this appendix, we will derive the formula for the gradient of the PPF loss as defined in Section 4.1.3. We do this since the experiments showed that the autograd engine slowed down when calculating these gradients. By defining the gradients explicitly, we were able to substantially speed up these calculations.

B.1. Basic Definitions

For sake of clarity and ease of reading, we restate the equations of the PPF loss here again:

$$\mathcal{L}_{\mathrm{PPF}}(\vec{c}, \vec{\hat{c}}) = -\mathbb{E}[n_r|\vec{\hat{c}}] \tag{B.1}$$

$$\mathbb{E}[n_r|\hat{c}] = \begin{cases} 0 & \text{if } n = \mathrm{t} \\ \sum_{a\in\mathcal{A}|n=\mathrm{from}(a)} \mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) \cdot (c_a + \mathbb{E}[\mathrm{to}(a)|\vec{\hat{c}}]) & \text{else} \end{cases} \tag{B.2}$$

$$\mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) = \frac{\exp(\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}])}{\sum_{a'\in\mathcal{A}|n=\mathrm{from}(a')} \exp(\hat{c}_{a'} + \mathbb{F}[\mathrm{to}(a')|\vec{\hat{c}}])} \tag{B.3}$$

$$\mathbb{F}[n|\vec{\hat{c}}] = \begin{cases} 0 & \text{if } n = \mathrm{t} \\ \sum_{a\in\mathcal{A}|n=\mathrm{from}(a)} \mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) \cdot (\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}]) & \text{else} \end{cases} \tag{B.4}$$

B.2. Gradient of Loss and Expectation

The formula for the gradients of the PPF loss is equivalent to the negative of the gradients of the expectation, as seen below:

$$\nabla_{\vec{\hat{c}}} \mathcal{L}(\vec{c}, \vec{\hat{c}}) = -\nabla_{\vec{\hat{c}}} \mathbb{E}[n_r|\vec{\hat{c}}] \tag{B.5}$$

The first step in calculating the gradient over this expectation is:

$$\nabla_{\vec{c}}\mathbb{E}[n|\vec{c}] = \begin{cases} \vec{0} & \text{if } n = \mathrm{t} \\ \sum_{a\in\mathcal{A}|n=\mathrm{from}(a)} \nabla_{\vec{c}}\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P}) \cdot (c_a + \mathbb{E}[\mathrm{to}(a)|\vec{c}]) & \text{else} \end{cases} \tag{B.6}$$

From this equation, we see that the conditional statement is unnecessary, since the set $\{a \in \mathcal{A}|n = \mathrm{from}(a)\}$ is empty for $n = \mathrm{t}$, and the sum over an empty set is zero.

$$\nabla_{\vec{c}}\mathbb{E}[n|\vec{c}] = \sum_{a\in\mathcal{A}|n=\mathrm{from}(a)} \nabla_{\vec{c}}\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P}) \cdot (c_a + \mathbb{E}[\mathrm{to}(a)|\vec{c}]) \tag{B.7}$$

$$= \sum_{a\in\mathcal{A}|n=\mathrm{from}(a)} (c_a + \mathbb{E}[\mathrm{to}(a)|\vec{c}]) \cdot (\nabla_{\vec{c}}\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P})) + \mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P}) \cdot (\vec{e_a} + (\nabla_{\vec{c}}\mathbb{E}[\mathrm{to}(a)|\vec{c}])) \tag{B.8}$$

This equation shows that we only are left with having to define the gradient of $\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P})$.

B.3. Gradient of $\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P})$

From Equation (B.3), we see that $\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P})$ is defined as the softmax of the output of $t(a, \vec{\hat{c}})$, where the sum is applied over all outgoing arcs of the node. Thus, the gradient of $\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P})$ is the gradient of $t(a, \vec{\hat{c}})$ multiplied with the softmax gradient, as shown below:

$$\nabla_{\vec{\hat{c}}}\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P}) = \nabla_{\vec{\hat{c}}}\frac{t(a, \vec{\hat{c}})}{\sum_{a\in\mathcal{A}|n=\mathrm{from}(a)} t(a, \vec{\hat{c}})} \tag{B.9}$$

$$= \frac{\nabla_{\vec{\hat{c}}}t(a, \vec{\hat{c}})}{\sum_{a'\in\mathcal{A}|n=\mathrm{from}(a')} t(a', \vec{\hat{c}})} - \frac{t(a, \vec{\hat{c}})\nabla_{\vec{\hat{c}}}t(a, \vec{\hat{c}})}{\left(\sum_{a'\in\mathcal{A}|n=\mathrm{from}(a')} t(a', \vec{\hat{c}})\right)^2} \tag{B.10}$$

B.4. Gradient of $\mathbb{F}[n|\vec{\hat{c}}]$

Finally, we have to define the gradients of the predicted expectation as defined in Equation (B.4). Here, the first step is to note that the gradient of the zero vector is also the zero vector and that we can move the gradient operator inside the sum.

$$\nabla_{\vec{\hat{c}}}\mathbb{F}[n|\vec{\hat{c}}] = \begin{cases} \vec{0} & \text{if } n = \mathrm{t} \\ \sum_{a\in\mathcal{A}|n=\mathrm{from}(a)} \nabla_{\vec{\hat{c}}}\mathrm{P}(a \in \mathcal{P}\,|\,\mathrm{from}(a) \in \mathcal{P}) \cdot (\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}]) & \text{else} \end{cases} \tag{B.11}$$

Here, again we note that the conditional statement is not necessary since the set $a \in \mathcal{A}|n = \mathrm{from}(a)$ is empty for node t.

$$
\begin{aligned}
\nabla_{\vec{\hat{c}}}\mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) \cdot (\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}]) \\
&= (\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}]) \cdot \nabla_{\vec{\hat{c}}}\mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) \\
&\quad + \mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) \cdot (\vec{e_a} + \nabla_{\vec{\hat{c}}}\mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}])
\end{aligned} \tag{B.12}
$$

B.5. Conclusion

$$
\nabla_{\vec{\hat{c}}}\mathcal{L}_{\mathrm{PPF}}(\vec{c}, \vec{\hat{c}}) = -\nabla_{\vec{\hat{c}}}\mathbb{E}[n_r|\vec{\hat{c}}] \tag{B.13}
$$

$$
\nabla_{\vec{\hat{c}}}\mathbb{E}[n|\vec{\hat{c}}] = \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} (c_a + \mathbb{E}[\mathrm{to}(a)|\vec{\hat{c}}]) \odot (\nabla_{\vec{\hat{c}}}\mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P})) \tag{B.14}
$$

$$
\begin{aligned}
\nabla_{\vec{\hat{c}}}\mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) &= \frac{(\vec{e_a} + \nabla_{\vec{\hat{c}}}\mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}]) \odot \exp(\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}])}{\sum_{a' \in \mathcal{A} | n = \mathrm{from}(a')} \exp(c_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}])} \\
&- \frac{\exp(c_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}]) \odot (\vec{e_a} + \nabla_{\vec{\hat{c}}}\mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}]) \odot \exp(\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}])}{\left(\sum_{a' \in \mathcal{A} | n = \mathrm{from}(a')} \exp(\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}])\right)^2}
\end{aligned} \tag{B.15}
$$

$$
\begin{aligned}
\nabla_{\vec{\hat{c}}}\mathbb{F}[n|\vec{\hat{c}}] = \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} & (\hat{c}_a + \mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}]) \odot \nabla_{\vec{\hat{c}}}\mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) \\
&+ \mathrm{P}(a \in \mathcal{P} \,|\, \mathrm{from}(a) \in \mathcal{P}) \odot (\vec{e_a} + \nabla_{\vec{\hat{c}}}\mathbb{F}[\mathrm{to}(a)|\vec{\hat{c}}])
\end{aligned} \tag{B.16}
$$

C

Derivation of Gradients for Maximum Likelihood Loss

In this appendix, we will derive the formula for the gradient of the ML loss as defined in Section 4.1.4. We do this since the experiments showed that the autograd engine slowed down when calculating these gradients. By defining the gradients explicitly, we were able to substantially speed up these calculations.

One thing to note here is that we define the gradients of the entire ML loss, but we only implemented the gradients of Z, for our experiments, this was done since this was the problematic part, where most time calculating was spent.

C.1. Basic Definitions

For sake of clarity and ease of reading, we restate the equations of the ML loss here again.

$$\mathcal{L}_{\text{ML}}(\vec{c}, \vec{\hat{c}}) = \log(Z) - (v(\vec{c})^T \vec{\hat{c}}) \tag{C.1}$$

$$\log\left(Z_{n_i}\right) = \begin{cases} 0 & \text{if } n_i = \text{t} \\ \log\left(\sum_{a \in A |\, \text{from}(a) = n_i} \exp\left(\log\left(Z_{\text{to}(a)}\right) + c_a\right)\right) & \text{else} \end{cases} \tag{C.2}$$

Now, we want to find the gradient of Equation (C.1) with respect to $\vec{\hat{c}}$, thus we get:

$$\nabla_{\vec{\hat{c}}} \mathcal{L}(\vec{c}, \vec{\hat{c}}) = \nabla_{\vec{\hat{c}}} \log(Z) - v(\vec{c})^T \vec{\hat{c}} \tag{C.3}$$

$$= (\nabla_{\vec{\hat{c}}} \log(Z)) - v(\vec{c}) \tag{C.4}$$

C.2. Gradient of the Normalization Value Z

Then, we just have to find define the gradients of $\log(Z)$ with respect to $\vec{\hat{c}}$.

$$\nabla_{\vec{\hat{c}}} \log(Z) = (Z)^{-1} \cdot (\nabla_{\vec{\hat{c}}} Z_r) \tag{C.5}$$

$$\nabla_{\vec{\hat{c}}} Z_n = \begin{cases} \vec{0} & \text{if } n = \mathrm{t} \\ \nabla_{\vec{\hat{c}}} \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} Z_{\mathrm{to}(a)} \exp(\vec{\hat{c}}^T \vec{e_a}) & \text{else} \end{cases} \tag{C.6}$$

Now, we only take the second term, and we get the following gradient:

$$\nabla_{\vec{\hat{c}}} \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} Z_{\mathrm{to}(a)} \cdot \exp(\vec{\hat{c}}^T \vec{e_a}) \tag{C.7}$$

$$= \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} \nabla_{\vec{\hat{c}}} Z_{\mathrm{to}(a)} \cdot \exp(\vec{\hat{c}}^T \vec{e_a}) \tag{C.8}$$

$$= \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} (\nabla_{\vec{\hat{c}}} Z_{\mathrm{to}(a)}) \exp(\vec{\hat{c}}^T \vec{e_a}) + Z_{\mathrm{to}(a)} (\nabla_{\vec{\hat{c}}} \exp(\vec{\hat{c}}^T \vec{e_a})) \tag{C.9}$$

$$= \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} \exp(\vec{\hat{c}}^T \vec{e_a}) (\nabla_{\vec{\hat{c}}} Z_{\mathrm{to}(a)}) + Z_{\mathrm{to}(a)} (\exp(\vec{\hat{c}})^T \vec{e_a}) \tag{C.10}$$

Finally, we can use Equation (C.10) in (C.5) as $\nabla_{\vec{\hat{c}}} Z_r$.

C.3. Conclusion

Thus, we get the following gradient definitions for the ML loss.

$$\nabla_{\vec{\hat{c}}} \mathcal{L}(\vec{c}, \vec{\hat{c}}) = Z^{-1} \cdot (\nabla_{\vec{\hat{c}}} Z_r) - v(\vec{c}) \tag{C.11}$$

Where $\nabla_{\vec{\hat{c}}} Z_r$ is defined as:

$$\nabla_{\vec{\hat{c}}} Z_n = \begin{cases} \vec{0} & \text{if } n = \mathrm{t} \\ \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} \exp(\vec{\hat{c}} \odot \vec{e_a}) (\nabla_{\vec{\hat{c}}} Z_{\mathrm{to}(a)}) + Z_{\mathrm{to}(a)} (\exp(\vec{\hat{c}}) \odot \vec{e_a}) & \text{else} \end{cases} \tag{C.12}$$

Finally, we note that we can remove the conditional statement since, the sink node has no outgoing edges, and a sum of zero elements is zero. Furthermore, we slightly rewrite Equation (C.12), this is done to make the implementation more numerical stable, since most calculation could be done in the log domain.

$$\nabla_{\vec{\hat{c}}} Z_n = \sum_{a \in \mathcal{A} | n = \mathrm{from}(a)} \exp(\hat{c}^T \vec{e_a} + \nabla_{\vec{\hat{c}}} \log(Z_{\mathrm{to}(a)})) + \exp(\log(Z_{\mathrm{to}(a)}) + \vec{\hat{c}}) \odot \vec{e_a} \tag{C.13}$$