Learning Lagrangian Multipliers for the Travelling Salesman Problem

Augustin Parjadis
Polytechnique Montréal, Canada

Quentin Cappart
Polytechnique Montréal, Canada

Bistra Dilkina
Center for Artificial Intelligence in Society, University of Southern California, Los Angeles, CA, USA

Aaron Ferber
Center for Artificial Intelligence in Society, University of Southern California, Los Angeles, CA, USA

Louis-Martin Rousseau
Polytechnique Montréal, Canada

Abstract

Lagrangian relaxation is a versatile mathematical technique employed to relax constraints in an optimization problem, enabling the generation of dual bounds to prove the optimality of feasible solutions and the design of efficient propagators in constraint programming (such as the weighted circuit constraint). However, the conventional process of deriving Lagrangian multipliers (e.g., using subgradient methods) is often computationally intensive, limiting its practicality for large-scale or time-sensitive problems. To address this challenge, we propose an innovative unsupervised learning approach that harnesses the capabilities of graph neural networks to exploit the problem structure, aiming to generate accurate Lagrangian multipliers efficiently. We apply this technique to the well-known Held-Karp Lagrangian relaxation for the traveling salesman problem. The core idea is to predict accurate Lagrangian multipliers and to employ them as a warm start for generating Held-Karp relaxation bounds. These bounds are subsequently utilized to enhance the filtering process carried out by branch-and-bound algorithms. In contrast to much of the existing literature, which primarily focuses on finding feasible solutions, our approach operates on the dual side, demonstrating that learning can also accelerate the proof of optimality. We conduct experiments across various distributions of the metric traveling salesman problem, considering instances with up to 200 cities. The results illustrate that our approach can improve the filtering level of the weighted circuit global constraint, reduce the optimality gap by a factor two for unsolved instances up to a timeout, and reduce the execution time for solved instances by 10%.

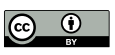
2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Theory of computation → Constraint and logic programming; Computing methodologies → Machine learning

Keywords and phrases Lagrangian relaxation, unsupervised learning, graph neural network

Digital Object Identifier 10.4230/LIPIcs.CP.2024.22

Supplementary Material
Software (Code source): `https://github.com/corail-research/learning-hk-bound`

Acknowledgements We sincerely thank the anonymous reviewers for their constructive feedback. Their comments helped us better position our contribution within the field. Furthermore, their insights have provided guidance for our future research directions.



1 Introduction

The *travelling salesman problem* (TSP) has been the subject of extensive research and has broad practical applications. Due to its NP-hard nature, numerous approaches have been proposed to solve it efficiently, ranging from exact to heuristic methods. Exact solvers not only need to identify the optimal solution but also to prove that it is optimal, often via a dual bound. Held and Karp (1970) [25] proposed a relaxation that provides strong dual bounds in practice. For instance, these bounds are used in Concorde, the state-of-the-art TSP solver [3] or in the design of global constraints in constraint programming [6, 5]. An associated branch-and-bound algorithm using this relaxation was subsequently proposed by Held and Karp (1971) [26], which enabled the optimality proof for several open benchmark instances at the time of its publication. Briefly, this algorithm leverages a combinatorial structure, referred to as *minimum 1-tree*, that can serve as a valid relaxation for the TSP and obtain dual bounds. However, this algorithm is based on a few heuristic design choices which have an important impact on the tightness of the relaxation. One is the procedure to generate the bounds from Lagrangian multipliers (explained in the next section), which can be assimilated as a hill-climbing algorithm. Starting from initial bounds, the algorithm refines the bound iteratively with local perturbations until convergence. There are two drawbacks to this process. First, it requires several potentially costly iterations to get accurate bounds, and second, it only converges to local minima. Our research hypothesis is that this procedure can be improved thanks to a learning-based approach. The idea is to train a model in an unsupervised fashion with similar TSP instances and to use it to predict Lagrangian multipliers that can be used to obtain a valid dual bound instead of computing it iteratively. In the field of constraint programming, Lagrangian decomposition has been also considered to provide dual bounds [8, 23], but without resorting to a learning-based component. We also note that other algorithms have been considered to improve the bounds for arborescence problems in constraint programming [28].

Machine learning has helped guide heuristic components in branch-and-bound [31, 38, 22, 57], constraint programming [13], SAT solving [49], local search [56], and non linear optimization [20]. We refer to the survey of Bengio et al. (2021) [7] for an extended literature review on this topic. Most of such works operate on the branching decisions (e.g., selecting the next variable to branch on) or on the primal side. However, learning to improve the quality of relaxations by means of better dual bounds has been much less considered in the literature. To our knowledge, this has only been addressed for the restricted use case of solvers based on decision diagrams [11], for combinatorial optimization over factor graphs [15] (e.g., see the max-sum labeling problem [54] and soft arc consistency [14] for relevant applications) and for learning relaxations of integer linear programs [1]. This last work is contemporaneous to ours.

Additionally, recent work in *decision-focused learning* (DFL) [40] has approached settings where the problem formulation is not fully specified at the time of decision-making. Thus, these approaches train gradient-based deep learning models to predict the missing components, with a key component being to determine how to train the deep learning model to improve the downstream decision quality. As training for deep networks is done using gradient descent, the difficulty lies in deriving methods for differentiating the output of the optimization model with respect to its predicted inputs. Our proposed approach seeks to predict the parameters of the Held-Karp relaxation such that the resulting relaxed solution provides a dual bound as tight as possible. This is achieved by deriving gradients for the relaxation to learn parameters that directly optimize the related bound. Differentiation has been

successfully deployed for quadratic programs [2], linear programs [55, 18, 39], mixed integer linear programs [19], MAXSAT [52], and blackbox discrete optimization [44, 42], among others discussed in these surveys [46, 35]. However, this approach is the first to consider using differentiable optimization together with learning to improve the filtering of a global constraint.

Coming back to the TSP, the design of learning-based solving approaches has also sparked a great interest in the research community [16, 34, 37]. In an industrial context, this methodology is relevant for practitioners who are solving similar problem instances every day and want to leverage historical decisions, e.g. in last-mile package delivery [41]. *Graph neural networks* (GNN) is a neural architecture [47, 33] widely considered for the TSP [29]. More generally, GNNs also play a crucial role in the success of applying deep learning to combinatorial optimization [30, 12]. They allow for the extraction of rich hidden representations by successively aggregating the weights of neighboring nodes in a graph, on which many combinatorial problems are defined.

Based on this context, the contribution of the paper is an approach based on unsupervised learning and graph neural networks to generate appropriate Lagrangian multipliers for the TSP, which are then used to improve the Held-Karp relaxation. We highlight that, compared to most of the related work, we do not learn a primal heuristic but a learning-based strategy that derives valid and tight dual bounds. Additionally, we integrate this mechanism inside a branch-and-bound algorithm with domain filtering and constraint propagation [5] to improve exact TSP solving. Experiments are carried out on three distributions of metric TSPs and the results show that our approach can improve the filtering level of the weighted circuit global constraint, reduce the optimality gap by a factor of two for unsolved instances up to a timeout, and reduce the execution time for solved instances by 10%.

The following section briefly overviews the Held-Karp relaxation principle for the TSP. Building upon this, we next describe the proposed learning approach for generating bounds through unsupervised learning on the Lagrangian multipliers of the Held-Karp relaxation. Finally, we discuss the training and integration of dual-bound generation within a branch-and-bound algorithm to evaluate their impact.

2 Held-Karp Lagrangian Relaxation

Finding optimal solutions for large TSP instances requires sophisticated approaches due to the combinatorial explosion of the solution space. With branch-and-bound, optimization bounds are employed to prune the search tree and accelerate the search, allowing solvers to prove optimality without exploring the entire tree. To achieve this, the *Held-Karp relaxation* [25] offers a robust dual bound based on a variant of minimum spanning trees.

Let $G = (V, E)$ be a complete graph with a cost attached to each edge. A *minimum 1-tree* is a minimum spanning tree of $G \backslash \{1\}$ to which we add the node 1 along with the two cheapest edges connecting it to the tree. We note that the choice of node 1 is arbitrary, depending on the labeling of V. A minimum 1-tree can be obtained by solving the integer program presented in Equations (1) to (5). Constraints (2) and (3) define the 1-tree structure and Constraint (4) enforces the elimination of sub-tours. This problem involves finding a minimum spanning tree that can be solved in $\mathcal{O}(E \log V)$ by Kruskal's algorithm. Here, $\delta(v)$ denotes the edges containing node $v \in V$ and μ_e, ν_e denotes the two nodes linked by an edge $e \in E$. We use $c_e \in \mathbb{R}$ to represent the cost of an edge $e \in E$, and $x_e \in \{0, 1\}$ is the decision variable indicating whether edge e is included in the 1-tree.

$$\min \sum_{e \in E} c_e x_e \tag{1}$$

$$\text{s. t.} \sum_{e \in \delta(1)} x_e = 2 \tag{2}$$

$$\sum_{e \in E} x_e = |V| \tag{3}$$

$$\sum_{\substack{\mu_e \in S \\ \nu_e \in S \\ \mu_e < \nu_e}} x_e \leq |S| - 1 \qquad \forall S \subset V \backslash \{1\} \wedge |S| \geq 3 \tag{4}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{5}$$

Let us note that every tour in G is a 1-tree, and if a minimum 1-tree is a tour, it is an optimal solution to the TSP. Therefore any minimum 1-tree is a valid relaxation for the TSP, which is an interesting property to leverage. However, a solution of this integer program is not ensured to be a tour. To do so, a new set of constraints must be enforced.

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall v \in V \backslash \{1\} \tag{6}$$

These constraints force each node to have only two edges, an incoming and an outgoing one, and turn the problem in finding a minimum-cost Hamiltonian cycle, which is NP-hard. To obtain a valid 1-tree relaxation efficiently, one can then move these constraints (one for each node) into the Objective (1) and penalize their violations with associated Lagrangian multipliers $\theta_v \in \mathbb{R}$ for each $v \in V \backslash \{1\}$. The updated objective function is as follows.

$$\min \sum_{e \in E} c_e x_e - \sum_{v \in V \backslash \{1\}} \theta_v \left(2 - \sum_{e \in \delta(v)} x_e \right) \tag{7}$$

Intuitively, each node having a degree other than two will be penalized. An optimal 1-tree relaxation can be found by optimizing over the θ_v variables. To do so, an iterative approach has been proposed by Held and Karp [25, 26]. The idea is to adjust the Lagrangian multipliers θ step-by-step to build a sequence of 1-trees which provides increasingly better bounds. An overview of the process is proposed in Figure 1.

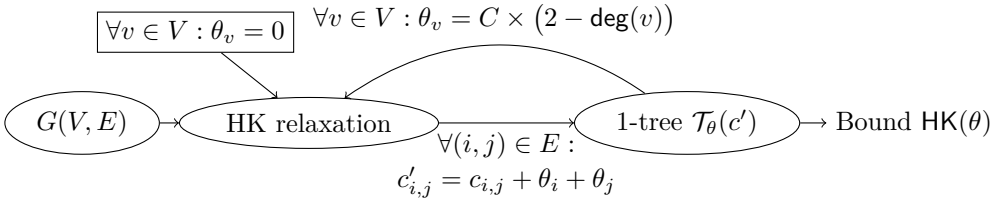


Figure 1 Approach of Held and Karp [25, 26] - Iterative process for improving θ multipliers.

First, an initial minimum 1-tree is computed by finding a minimum spanning tree on $G \backslash \{1\}$ and adding the two cheapest edges incident to node 1. If the optimal 1-tree is a tour, it corresponds to the optimal TSP solution. Otherwise, some constraints are penalized as at least one node has a degree greater than 2. The main idea of Held and Karp [25, 26] is to penalize such nodes by modifying the cost $c_{i,j}$ of edges $(i, j) \in E$, based on the values of θ_i and θ_j (i.e., the multipliers of adjacent nodes). Let $c'_{i,j} \in \mathbb{R}$ be the modified costs. They are computed as follows.

$$c'_{i,j} = c_{i,j} + \theta_i + \theta_j \qquad \forall (i, j) \in E \tag{8}$$

A property proved by Held and Karp [25, 26] is that the optimal TSP tour is invariant under this perturbation, whereas the optimal 1-tree is not. This gives room to improve the solution by finding better multipliers. Equation (9) proposes a standard choice to compute the multipliers, where $C \in \mathbb{R}$ is an arbitrary constant and $\mathsf{deg}(v)$ denotes the degree of node $v \in V$ in the current 1-tree.

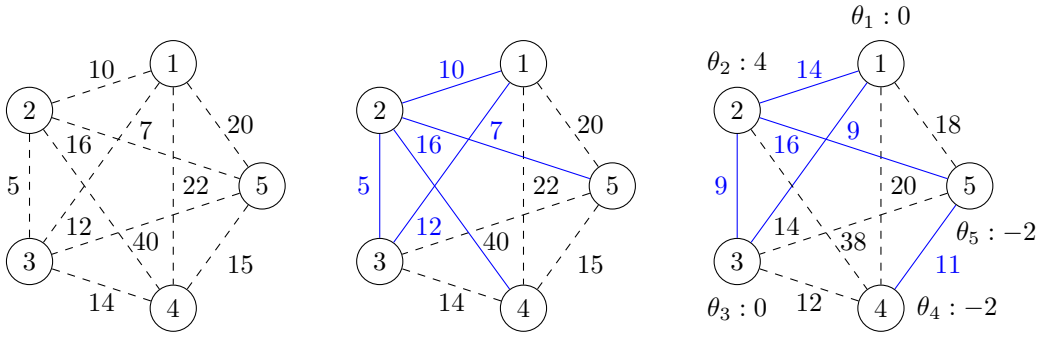
$$\theta_v = C \times \big(2 - \mathsf{deg}(v)\big) \qquad \forall v \in V \tag{9}$$

Finally, a new minimum 1-tree is computed from the graph with the updated costs $c'_{i,j}$. We note this 1-tree as $\mathcal{T}_\theta(c')$ where $c' = \{c_1, \ldots, c_{|E|}\}$ is the set of all modified costs, and $\theta = \{\theta_1, \ldots, \theta_{|V|}\}$ is the set of all multipliers. We also use the notation $\mathsf{cost}\big(\mathcal{T}_\theta(c')\big)$ to refer to the total cost of the 1-tree. This process is reiterated, and a new 1-tree $\mathcal{T}_\theta(c')$ is obtained until no improvement is obtained (i.e., when a local minimum is reached). The cost of the optimal 1-tree gives a lower bound on the objective value as follows.

$$\mathsf{HK}(\theta) = \mathsf{cost}\big(\mathcal{T}_\theta(c')\big) - 2\sum_{i=1}^{|V|} \theta_i \tag{10}$$

This bound, $\mathsf{HK}(\theta)$, is commonly referred to in the literature as the *Held-Karp bound*. This is a valid lower bound as if the related solution is not a tour, the optimal TSP tour will have a higher value. Otherwise, the optimal tour would have been obtained as tours are 1-trees. This approach is typically incorporated into a branch-and-bound algorithm, using this bound to prune the search. While computing a 1-tree is generally computationally efficient, the iterative adjustment of the θ multipliers can be computationally expensive. Our contribution is dedicated to mitigating this issue thanks to an unsupervised learning process.

Example. Figure 2 illustrates the Held-Karp relaxation for a graph with an optimal TSP tour value of 62 (a). A 1-tree is computed on the original graph without Lagrangian multipliers, which yields a bound of 50 (b). Considering Equation (9) with $C = 2$, we obtain the following multipliers: $\{\theta_1 : 0, \theta_2 : 4, \theta_3 : 0, \theta_4 : -2, \theta_5 : -2\}$. The corresponding penalized 1-tree with Lagrangian multipliers modifying the edge costs provides a bound of 59, which is tighter (c).



(a) Initial TSP instance. The optimal cost is 62.

(b) Minimum 1-tree under the initial costs. Lower bound obtained is 50 (gap of 20%).

(c) Minimum 1-tree with modified costs. Lower bound obtained is 59 (gap of 5%).

Figure 2 Illustration of a single iteration of Held-Karp relaxation for an arbitrary TSP instance.

In constraint programming, the *weighted circuit* constraint [4] ensures that a set of variables Y forms a Hamiltonian circuit on a graph, while also satisfying a specified condition on the total cost z of the circuit. This can be intuitively understood as a conjunction of

a *circuit* constraint [36] and a second constraint enforcing the circuit to be lower than a threshold. Briefly, the standard filtering algorithms for this constraint typically involve: (1) identifying edges that must be included in any feasible solution, (2) eliminating edges that cannot be part of any solution, and (3) determining the minimum possible value of the cost threshold z. This information is then used for narrowing the domains of the variables Y. Given that establishing bounds consistency for this constraint is NP-hard, Benchimol et al. (2012) [5] proposed a filtering algorithm that uses relaxations of the weighted circuit constraint, specifically leveraging the Held-Karp relaxation. A stronger relaxation, such as a 1-tree that provides a tighter lower bound, allows for more extensive filtering. This in turn improves the efficacy of the weighted circuit propagator, which is the focus of this paper. More broadly, this propagator is part of the *cost-based filtering* family, which utilizes valid bounds for effective filtering. Generally, a tighter bound leads to more efficient filtering within this framework.

3 Learning Held-Karp Multipliers

The Held-Karp bound $\mathsf{HK}(\theta)$ has two interesting properties: (1) it can be parameterized thanks to the θ Lagrangian multipliers, and (2) it is always valid, meaning it will never exceed optimal TSP cost. Both properties open the opportunity to use a learning-based approach to compute the bound. To do so, we propose to build a model $\Phi_w : G(V, E) \to \mathbb{R}^{|V|}$ able to directly predict all the θ multipliers for a TSP instance given as input (i.e., a graph). The model is parameterized with p parameters $w = \{w_1, \ldots, w_p\}$. There are two benefits to this. First, it eliminates parts of the iterative process of Held and Karp [25] depicted in Figure 2 and saves execution time. Second, it allows us to obtain tighter bounds. The process is illustrated in Figure 3.

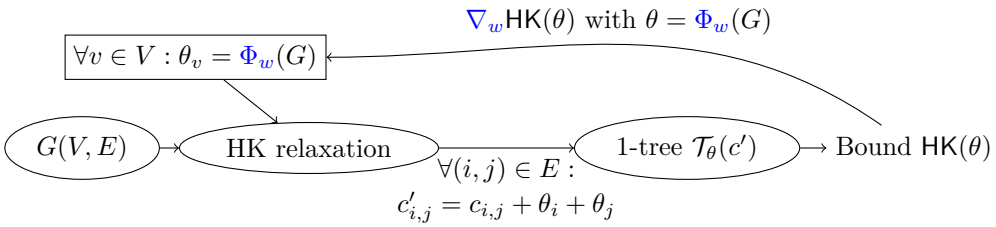


Figure 3 Our contribution - Unsupervised learning approach to obtain θ multipliers through backpropagation.

The goal is to find model parameters w yielding the highest possible bound. This corresponds to a maximization problem that can be solved by gradient-based optimization. The obtained bound is provably valid, regardless of the trained model's accuracy thanks to the second property. We consider this a major strength of our contribution, as obtaining guarantees with machine learning for combinatorial optimization is known to be a challenge [35]. We formulate the bound maximization problem and its gradient below.

$$\max_w \mathsf{HK}\big(\Phi_w(G)\big) \longmapsto \nabla_w \mathsf{HK}(\Phi_w(G)) \tag{11}$$

However, computing the gradient of this expression is not trivial, as the bound is obtained by means of the 1-tree combinatorial structure $\mathcal{T}_\theta(c')$ (see Equation (10)). As the tree is parameterized by θ, the chain rule can be applied to clarify the dependencies between model parameters w and Lagrangian multipliers θ.

$$\nabla_w \mathsf{HK}\big(\Phi_w(G)\big) = \frac{\partial \mathsf{HK}(\Phi_w(G))}{\partial \theta} \times \frac{\partial \theta}{\partial w} \tag{12}$$

The right term corresponds to the differentiation of the predictive neural network model and is easily obtained by backpropagation. However, the left term requires to differentiate the expression depicted in Equation (10) for all θ_i with $i \in V$.

$$\frac{\partial \mathsf{HK}(.)}{\partial \theta} = \frac{\partial \mathsf{cost}\big(\mathcal{T}_\theta(c')\big)}{\partial \theta} - 2\frac{\partial \sum_{i=1}^{|V|} \theta_i}{\partial \theta} \tag{13}$$

The cost of the 1-tree (i.e., $\mathsf{cost}\big(\mathcal{T}_\theta(c')\big)$ corresponds to the weighted sum of the selected edges (i.e., variables $x_{i,j}$ for each $(i,j) \in E$). The cost $c'_{i,j}$ defines the weights.

$$\frac{\partial \mathsf{HK}(.)}{\partial \theta} = \frac{\partial \Big(\sum_{(i,j)}^{\in E} c'_{i,j} x_{i,j}\Big)}{\partial \theta} - 2\frac{\partial \sum_{i=1}^{|V|} \theta_i}{\partial \theta} \tag{14}$$

Let us consider a specific multiplier θ_i associated to node $i \in V$ and let us unroll the cost as $c'_{i,j} = c_{i,j} + \theta_i + \theta_j$ (see Equation (8)). The partial derivative of θ_i is non-zero only for the node itself and its adjacent edges, i.e. $(i,j) \in \delta(i)$.

$$\frac{\partial \mathsf{HK}(.)}{\partial \theta_i} = \frac{\partial \sum_{(i,j)}^{\in \delta(i)} (c_{i,j} + \theta_i + \theta_j) x_{i,j}}{\partial \theta_i} - 2\frac{\partial \theta_i}{\partial \theta_i} \tag{15}$$

$$= \frac{\partial \sum_{(i,j)}^{\in \delta(i)} \theta_i x_{i,j}}{\partial \theta_i} - 2\frac{\partial \theta_i}{\partial \theta_i} \tag{16}$$

$$= \sum_{(i,j) \in \delta(i)} x_{i,j} - 2 \tag{17}$$

This gives the partial derivative for each θ_i and allows us to maximize a bound obtained by a neural network directly with gradient ascent. Interestingly, this signal is non-zero when the degree of the node is different than 2 in the 1-tree. This is aligned with the intuition that we want to adjust the multipliers of conflicting nodes.

We note that the derivative obtained is correct only *locally* and not *globally*. Indeed, the cost function in Equation (13) is an optimization problem consisting in finding a minimum 1-tree (i.e., setting variables x) from the current costs c' and that the values of x will depend on θ. We experimented with a globally valid derivative by computing all insertion and replacement costs to integrate them in the derivative of x, but this resulted latter in an unstable training, likely because of the non-convexity of the optimization landscape. For such a reason, we carried out a *subgradient ascent* on a locally valid derivative instead. Variations of variables x are then taken into account in the subsequent gradient ascent step. We have empirically observed increased stability as a result of this procedure.

The training procedure is formalized in Algorithm 1. It gives as output the parameters w of the trained neural network Φ_w. We note that this training loop can be easily improved with standard techniques in deep learning, such as mini-batches or using another gradient-based optimizer, such as Adam [32]. Unlike gradient descent, we aim to maximize the bound, explaining the + term at Line 10. We highlight that the training is *unsupervised* as it does not require ground truth on known tight bounds for training the model, nor the corresponding Lagrangian multipliers. In each iteration of the algorithm, the values of x will change as the multipliers (θ) are updated. This explains how the variations of x are implicitly considered during each subgradient ascent step. Finally, two aspects of the methodology require clarification: the architecture of the network Φ_w and how the training set $\mathcal{D}$ is built. Both are discussed in the following sections.

Algorithm 1 Training phase from an input graph $G(V, E)$.

▷ **Pre:** D is the set of instances used for training.
▷ **Pre:** Φ_w is the differentiable model to train.
▷ **Pre:** w are randomly initialized parameters.
▷ **Pre:** K is the number of training epochs.

for k **from** 1 **to** K **do**
  $G := \mathsf{SampleFromTrainingSet}(\mathcal{D})$
  $\theta := \Phi_w(G)$
  $\mathcal{T}_\theta(c') := \mathsf{HeldKarpRelaxation}(G, \theta)$
  $\mathsf{HK}(\theta) := \mathsf{cost}(\mathcal{T}_\theta(c')) - 2\sum_{i=1}^{|V|} \theta_i$
  $w := w + \nabla_w \mathsf{HK}(\Phi_w(G))$
end for
return w

3.1 Training Set Construction

The training is carried out from a dataset $\mathcal{D}$ consisting of a set of graphs $G(V, E)$ serving as TSP instances. The graphs can either be obtained from historical problem instances (e.g., previous routing networks and costs for a delivery company) or randomly generated. Each graph has six features f_i for each node $i \in V$ and three features $k_{i,j}$ for each edge $(i, j) \in E$. The features we used are presented in Table 1 and in Table 2. We have incorporated the features we believe are important for this task, but we have not analyzed the individual impact of each feature. One strength of deep neural networks is their ability to learn to disregard features that are not beneficial for the task. Although most of the features are relatively standard, $k^2_{i,j}$ and $k^3_{i,j}$ introduce the notions of *mandatory* and *forbidden* edges. In the context of a branch-and-bound algorithm, some decision variables are fixed after branching operations. An edge is mandatory if it must be part of the TSP solution (i.e., $x_{i,j} = 1$) and it is forbidden if it cannot be in the solution (i.e., $x_{i,j} = 0$). This information is crucial as we plan to compute bounds several times during a branch-and-bound execution, with the intention of leveraging partial solutions to get better bounds.

Table 1 Summary of the features on nodes i for each $i \in V$ used in an input graph $G(V, E)$.

Symbol	Formalization	Description
$f^1_i, f^2_i \in \mathbb{R}^2$	$\mathsf{xPos}(i), \mathsf{yPos}(i)$	2-dimensional coordinate of the node.
$f^3_i \in \mathbb{R}$	$\frac{1}{\lvert V\rvert}\sum_{j=1}^{\lvert V\rvert} \lVert\mathsf{coord}(i) - \mathsf{coord}(j)\rVert_2$	Average euclidean distance with the other nodes.
$f^4_i \in \mathbb{R}$	$\min_{j\neq i}\ (f^3_1, \ldots, f^3_j, \ldots, f^3_{\lvert V\rvert})$	Distance to the nearest node in the graph.
$f^5_i \in \mathbb{N}^+$	$\mathsf{deg}(i)$	Degree in terms of incoming and outgoing edges.
$f^6_i \in \{0, 1\}$	1 iff $i = 1, 0$ otherwise	Indication if it is the excluded node in $G\backslash\{1\}$.

A direct observation is that there are no fixed edges at the root node of a branch-and-bound tree, and consequently, for none of the instances in the training set. This causes a distributional shift between instances used for the training (only at the root node) and the ones occurring at the testing phase (also inside the branch-and-bound tree). To address this limitation, we propose to enrich the training set with partially solved TSP instances extracted from explored branch-and-bound nodes. In practice, it is done by fixing a threshold $k \in \mathbb{N}^+$ on the number of nodes to consider in the training set. This makes the computation tractable as it avoids considering all the nodes of an exponentially sized tree search.

Table 2 Summary of the features on edges (i,j) for each $(i,j) \in E$ used in an input graph $G(V,E)$.

Symbol	Formalization	Description
$k^1_{i,j} \in \mathbb{R}$	$c_{i,j}$	The cost of the edge.
$k^2_{i,j} \in \{0,1\}$	1 iff (i,j) is forbidden, 0 otherwise	Binary value indicating if the edge is *forbidden*.
$k^3_{i,j} \in \{0,1\}$	1 iff (i,j) is mandarory, 0 otherwise	Binary value indicating if the edge is *mandatory*.

3.2 Graph Neural Network Architecture

A TSP instance exhibits a natural graph structure. For this reason, we built the model Φ_w with a *graph neural network* [47, 33] (GNN). This architecture has been widely used in related works for the TSP, thanks to their ability to handle instances of different size, to leverage node and edge features, etc. In its standard version, GNNs are dedicated to computing a vector representation of each node of the graph. Such a representation is commonly referred to as an *embedding*. The embedding of a specific node is computed by iteratively transforming and aggregating information from the neighboring nodes. Each aggregation operation is referred to as a layer of the GNN and involves weights that must be learned. This operation can be performed in many ways, and there exist in the literature different variants of GNNs. An analysis on the performances of various architectures is proposed by Dwivedi et al. (2023) [17]. Our model is based on the *edge-featured graph attention network* [53] which is a variant of the well-known *graph attention network* [50] dedicated to handle features on the edges. The whole architecture is differentiable and is trained with backpropagation.

Let $G(V,E)$ be the input graph, $f_i \in \mathbb{R}^6$ be a vector concatenating the 6 features of a node $i \in V$, and $k_{i,j} \in \mathbb{R}^3$ be a vector concatenating the three features of an edge $(i,j) \in E$. The GNN architecture is composed of L layers. Let $h^l_i \in \mathbb{R}^d$ be a d-dimensional vector representation of a node $i \in V$ at layer $l \in \{1,\dots,L\}$, and let $h^{l+1}_i \in \mathbb{R}^{d'}$ a d'-dimensional vector representation of i at the next layer. The inference process consists in computing the next representation (h^{l+1}_i) from the previous one h^l_i for each node i. The first representation is set with the initial features of the node, i.e. $h^1_i = f_i$ for each $i \in V$. The computation is formalized in Equations (18) to (20), where w^l_1 and w^l_2 are two weight tensors that need to be trained for each layer.

$$h^{l+1}_i = \mathsf{ReLU}\Big(\sum_{j \in \mathcal{N}(i)} \alpha^l_{(i,j)} w^l_1 h^l_j\Big) \qquad \forall i \in V \ \wedge \ \forall l \in \{1,\dots,L-1\} \quad (18)$$

$$\alpha^l_{(i,j)} = \mathsf{Softmax}\Big(\mathsf{LeakyReLU}\Big(w^l_2 \times \big(h^l_i \| k_{i,j} \| h^l_j\big)\Big)\Big) \quad (19)$$

$$\theta_v = \mathsf{FCNN}\big(h^{|L|}_i\big) \qquad \forall i \in V \quad (20)$$

Equation (18) shows the message passing operation in a layer. Each node i aggregates information of all its neighbors $\mathcal{N}(i)$. The aggregation is subject to parameterized weights w^l_1 and a *self-attention score* $\alpha^l_{i,j}$. This score allows the model to put different weights on the incoming messages from neighboring nodes. We note that the attention integrates information about the node itself (h^l_i), its neighbor (h^l_j), and the features attached to the adjacent edge $(k_{i,j})$. Such information is concatenated $(.\|.)$ into a single vector. Non-linearities are added after each aggregation and the final node embeddings $h^{|L|}_i$ are given as input to a fully-connected neural network (FCNN) outputing the corresponding θ_i multiplier for each $i \in V$. The GNN has 3 graph attention layers with a hidden size of 32 and the fully-connected neural network has 2 layers with 32 neurons.

4 Experimental Evaluation

The goal of the experiments is to evaluate the efficiency of the approach to speed-up a TSP solver based on branch-and-bound and constraint programming [5]. To do so, the learned bounds are integrated into the Held-Karp relaxation used by the *weighted circuit* constraint [4] and are used to filter unpromising edges. The model is used only for the 10 first nodes expanded in the branch-and-bound tree (parameter k). We refer to HK for the standard solver of Benchimol et al. (2012) [5] and to HK+GNN for the one we introduce. We also considered a version using the learned multipliers but without the Held-Karp refinement (i.e., GNN without HK) but the results showed that the bounds obtained only with the learned multipliers alone were far from the optimal bound and are not included in the next experiments. Combining the learned bounds with the Held-Karp refinement is thus required.

4.1 Experimental Protocol

This section outlines the experimental protocol employed to evaluate the efficacy and reliability of our approach. It details the specific datasets, hardware configurations, software tools, and performance metrics used across various testing scenarios.

4.1.1 Datasets

Five datasets of different complexity are considered. They correspond to variants of the metric TSP (i.e., the graphs are complete and the distances are euclidean) on which the cities are localized with different patterns.

1. Random100 (and 200): the cities (100 or 200) are uniformly generated in the $[0, 1]^2$ plan.
2. Clustered100 (and 200): inspired by Fischetti and Toth (1989) [21], five clusters are uniformly generated in the $[0, 1]^2$ plan. Then, the cities (100 or 200) are uniformly generated inside the 0.1-radius circles for each cluster.
3. Hard: introduced by Hougardy and Zhong (2021) [27], these 50 instances ranging from 52 to 199 cities have been generated to have a large integrality gap and are provably hard to solve for branch-and-bound methods.

A test set of 50 instances is built for each configuration and is used for evaluation. For the last dataset, as it is relatively small, only 6 instances are taken for evaluation.

4.1.2 Implementation

The graph neural network has been implemented with *deep graph library* [51] and *Pytorch* [43]. During the training, the minimum spanning trees have been computed with *NetworkX* [24]. We used the *C++* implementation of Benchimol et al. (2012) [5] for the branch-and-bound solver. The interface between the *python* and the *C++* code has been done with native functions from both languages. The solver used to compute the optimal solution is *Concorde* [3]. The implementation and the datasets used are released on Github[1].

4.1.3 Training

The training phase corresponds to the execution of Algorithm 1. A specific model is trained for the five configurations. The training sets for Random and Clustered have 100 instances sampled from the given generating scheme. For the Hard dataset, 40 instances uniformly

[1] `https://github.com/corail-research/learning-hk-bound`

sampled from the 50 available instances are taken. For each instance, 10 subgraphs are generated (parameter k) and are added to the training set. They correspond to partially solved instances that could be found inside the branch-and-bound tree. We highlight that we do not need to label the training instances with known multipliers as the learning is unsupervised. Training time is limited to 4 hours on a AMD Rome 7502 2.50GHz processor with 64GB RAM. No GPU has been used. Models are trained with a single run and we observed the convergence of the loss function on a validation set of 20 instances. The Adam optimizer with a learning rate of 10^{-3} has been used.

4.1.4 Hyperparameters

The branch-and-bound has been configured with the standard settings and most of the hyperparameters follow the recommended values. No hyperparameter tuning has been carried out due to our limited resources. We think that our results can be improved with a more thorough calibration (e.g., slightly changing the number of layers), but as our goal was not to build the most efficient neural network but rather to show the promise of using them to get valid multipliers, we have not carried out this operation. A notable exception is the threshold k on the maximum number of nodes which has an important impact on the performances. We tested values in $\{1, 5, 10, 20, 50\}$ and selected $k = 10$ as it provided the best results and trade-off between accuracy and computation time.

4.1.5 Evaluation Metrics

Five metrics are reported in the results: (1) the execution time to prove the optimality of a solution, (2) the number of instances solved to optimality, (3) the percentage of edges filtered before branching through the propagation, (4) the optimality gap for unsolved instances, and (5) the primal-dual integral. This last metric measures the convergence of the dual bound and the primal bound over the whole solving time [10]. At each iteration, we record the best primal and dual bounds during the whole solving process. Then, the primal-dual integral value consists in taking the area between the two resulting curve obtained. Intuitively, the smaller the better. Each instance is solved only once per experiment as no randomness is involved in the execution. We clarify that our objective is not to compete with Concorde, which is capable of optimally solving all the instances we consider. Instead, our goal is to enhance the filtering of the weighted circuit constraint. This constraint can then be used to more complex variants of the Traveling Salesman Problem, such as the TSP with time windows.

4.2 Empirical Results

This section presents the results obtained from our experimental analysis. These findings are compiled to provide a quantitative evaluation of the performance and effectiveness of our approach under different scenarios. Each experiment is designed to test a specific hypothesis. Finally, a discussion about current limitations are proposed.

4.2.1 Main Results: Quality of the Learned Bounds

Table 3 summarizes the results for HK and HK+GNN on the five datasets. Values are averaged for each configuration. First, we can observe that our approach gives consistently better results on all the metrics compared to the baseline. As expected, it provides better filtering on the edges. This is reflected by a higher number of solved instances, a reduced execution time,

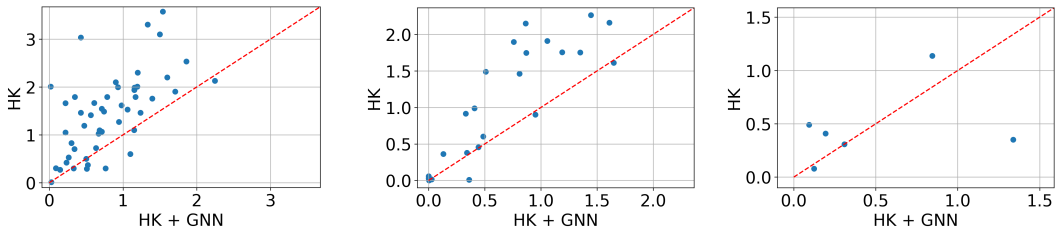
and a reduced optimality gap for unsolved instances. The primal-dual integral confirms that tighter dual bounds are obtained during the search. Second, we notice that the improvements on the Hard dataset are more modest. This can be explained by the fact that they are designed to be challenging. It is consequently more difficult to get improvements on these ones. We note that the time savings obtained with our methods could not provide to a potential faster calculation of the multipliers. The reason is that the cost of computing the multipliers is higher with our method as we combine both the prediction and the iterative HK process (HK+GNN). The speed-up is consequently due to an improved filtering.

Table 3 Comparison of our approach (HK + GNN) with the standard branch-and-bound of Benchimol et al. (HK). The primal bound is 2% above the optimal solution cost computed with Concorde. The statistics considered are: the execution time up to a timeout of 1,800 seconds (*Time*), the number of instances solved to optimality with proof (*# solved*), the primal-dual integral (*PDI*), the percentage of edges filtered (*Filt.*) and the optimality gap for unsolved instances (*Gap*). The relative improvement compared to the baseline is also depicted.

Configuration	Branch-and-bound with standard Held-Karp (HK)					Branch-and-bound with our approach (HK+GNN)				
	Time (sec.)	# solved	PDI	Filt. (%)	Gap (%)	Time (sec.)	# solved	PDI	Filt. (%)	Gap (%)
Random100	559	41/50	1127k	75.9	0.88	497 (- 11%)	46/50 (+ 5)	965k (- 14%)	77.7 (+ 2%)	0.48 (- 45%)
Random200	1800	0/50	4.71m	67.8	1.82	1800	0/50 (+ 0)	4.26m (- 10%)	70.6 (+ 4%)	0.59 (- 68%)
Clustered100	643	38/50	497k	17.7	0.19	590 (- 8%)	40/50 (+ 2)	470k (- 5%)	20.3 (+ 15%)	0.08 (- 58%)
Clustered200	1800	0/50	922k	9.9	0.68	1800	0/50 (+ 0)	690k (- 25%)	12.6 (+27%)	0.38 (- 44%)
Hard	1800	0/6	9.59M	6.4	0.32	1800	0/6 (+ 0)	9.36M (- 2%)	6.5 (+1%)	0.31 (- 3%)

4.2.2 Analysis: Focus on Individual Instances

Figure 4 provides an analysis of the optimality gap for the three hardest configurations by means of scatter plots. Each dot corresponds to a specific instance. When a dot is upper than the diagonal, it means that our approach provided better results than the baseline. Unlike the previous experiments, it provides insights about the robustness of the method. For the majority of the instances, our approach gave better or similar results, except for one instance in the Hard dataset.



(a) Results on Random200. **(b)** Results on Clustered200. **(c)** Results on Hard.

Figure 4 Scatter plots comparing the optimality gap (%) for HK and HK + GNN on the three hardest configurations.

4.2.3 Analysis: Addressing the Optimality Proof

This next experiment evaluates the ability of proving the optimality of a solution *only once this solution has been found*. Concretely, instead of taking a reasonable upper bound of 2%, we assume that the optimal solution has been found and we use it as a perfect upper

bound. The idea is to mimic results achieved by a good primal heuristic and to close the search by proving the optimality of the solution provided with this information. Results are summarized in Table 4. In such a situation, the improvements with the baseline are still positive, especially for the largest and hardest configurations. This shows the potential of learning dual bounds to accelerate optimality proofs and the synergies with approaches operating in the primal side.

Table 4 Comparison of our approach (HK + GNN) with the standard branch-and-bound of Benchimol et al. (HK) for optimality proof. Compared to Table 3, the primal bounds are now the cost of the optimal solution.

Configuration	Branch-and-bound with standard Held-Karp (HK)					Branch-and-bound with our approach (HK+GNN)				
	Time (sec.)	# solved	PDI	Filt. (%)	Gap (%)	Time (sec.)	# solved	PDI	Filt. (%)	Gap (%)
Random100	209	48/50	591k	92.1	0.05	203 (- 3%)	48/50 (+ 0)	544k (- 7%)	92.2 (+ 0%)	0.05 (- 0%)
Random200	1800	0/50	4.44m	85.6	1.16	1800	0/50 (+ 0)	4.11m (- 7%)	89.1 (+ 4%)	0.43 (- 62%)
Clustered100	112	48/50	103k	25.4	0	110 (- 1%)	49/50 (+ 1)	100k (- 3%)	25.5 (+ 0%)	0 (- 0%)
Clustered200	1800	0/50	713k	14.5	0.43	1800	0/50 (+ 0)	644k (-9%)	16.8 (+15%)	0.30 (- 30%)
Hard	1800	0/6	7.59M	17.8	0.26	1800	0/6 (+ 0)	7.02M (- 7%)	18.0 (+ 1%)	0.19 (- 24%)

4.2.4 Analysis: Generalization Ability

This last experiment analyzes how the models are able to generalize to new configurations, either for a higher number of cities or with another generation scheme. Concretely, we considered four configurations (Random100, Random200, Clustered100 and AllDataset, the latter being trained on the instance of all datasets) and evaluated them on Clustered200. The idea of this experiment is to compare the generalization ability of models trained on specific datasets (Random100, Random200 and Cluster100). Results on AllDataset provide an idea of the best performances that the specific models can achieve. Results are presented in Table 5. Although the performance of the specific model is not reached, we observe that the models trained on the other distributions are still able to outperform the standard model. Training a model on all datasets (AllDataset) allows to improve upon out-of-distribution models but does not achieve the performance of the specialized model. This confirms the intuitive benefit to know beforehand the distribution of the instances to solve.

Table 5 Analysis of the generalization. The different models are used to solve Clustered200 graphs.

Model	Branch-and-bound with HK+GNN		
	PDI	Filt. (%)	Gap (%)
Clustered200	690k	12.6	0.38
HK without GNN	922k	9.9	0.68
Clustered100	817k	11.1	0.54
Random100	845k	10.4	0.61
Random200	784k	11.3	0.49
AllDataset	722k	12.0	0.45

4.2.5 Discussion: Application to Non-Euclidean Instances

Previous experiments have shown promising results for various configurations of metric TSPs. However, it is important to note that the performance on other types of TSP instances, such as asymmetric or non-Euclidean TSPs, which are known to be more challenging, has not been evaluated in this paper. Consequently, it remains uncertain whether the observed performance will extend to these more complex instances. Exploring this further constitutes an interesting avenue for future research. Such an analysis can be facilitated using the instances available in TSPLib [45], providing a robust framework for testing under more diverse conditions.

4.2.6 Discussion: Considering the Training Time

As is common practice with machine learning tools, our framework assumes that we can train a model offline before its deployment to solve new instances, ideally following a similar distribution. In such cases, the training time can be disregarded, as it will be amortized over a large number of future instances. However, this assumption does not hold in all realistic scenarios where training time cannot be separated from solving time. In such situations, while our approach remains applicable, the time required to train the model must be considered and can be prohibitive. To address this, an initial analysis could involve monitoring how the learned bounds improve with training time. This would provide a more detailed understanding and help identify when training can be stopped to proceed directly to the solving phase. Intuitively, this approach may offer slightly less effective filtering compared to HK+GNN, but it is aimed to result in a reduced total execution time.

5 Conclusion

Learning-based methods have been extensively considered to provide approximate solutions to combinatorial optimization problems, such as the travelling salesman. However, learning to obtain dual bounds has been less considered in the literature and is much more challenging as there is no trivial way to ensure that the obtained bounds are valid. This paper introduces an unsupervised learning approach, based on graph neural networks and the Held-Karp Lagrangian relaxation, to tackle this challenge. The core idea is to predict accurate Lagrangian multipliers and employ them as a warm start for generating Held-Karp relaxation bounds. These bounds are subsequently used to enhance the filtering level of the weighted circuit global constraint and improve the performances of a branch-and-bound algorithm. The empirical results on different configurations of the TSP showed that the learning component can significantly improve the algorithm. We believe that the methodology can be integrated into existing CP solvers. To do so, one will need to refactor the weighted circuit global constraint implemented in the related solver to add the learned bounds. Although centered on the TSP, we note that weighted circuit global constraint could be used for other, and more challenging, TSP variants including time windows or time-dependent costs. Analyzing how these variants can be handled efficiently is part of our future work. Also, we believe that the methodology can be easily reused for other propagators using Lagrangian relaxation, such as for the *AtMostNValue* [9] or for the general framework of CP-based Lagrangian relaxation [48].

References

1 Ahmed Abbas and Paul Swoboda. DOGE-train: Discrete optimization on GPU with end-to-end training. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(18):20623–20631, 2024.

2 Brandon Amos and J. Zico Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 136–145. JMLR.org, 2017.

3 David L. Applegate, Robert E. Bixby, Vašek Chvatál, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006. URL: `http://www.jstor.org/stable/j.ctt7s8xg`.

4 Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and computer Modelling*, 20(12):97–123, 1994.

5 Pascal Benchimol, Willem-Jan van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17:205–233, 2012.

6 Pascal Benchimol, Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher, and Willem-Jan Van Hoeve. Improving the Held and Karp approach with constraint programming. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 40–44. Springer, 2010.

7 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

8 David Bergman, Andre A Cire, and Willem-Jan van Hoeve. Improved constraint propagation via lagrangian decomposition. In *International Conference on Principles and Practice of Constraint Programming*, pages 30–38. Springer, 2015.

9 Frédéric Berthiaume and Claude-Guy Quimper. Local alterations of the lagrange multipliers for enhancing the filtering of the atmostnvalue constraint. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 68–83. Springer, 2024.

10 Timo Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.

11 Quentin Cappart, David Bergman, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Augustin Parjadis. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS Journal on Computing*, 34(5):2552–2570, 2022.

12 Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Velickovic. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023. URL: `http://jmlr.org/papers/v24/21-0449.html`.

13 Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3677–3687, 2021.

14 Martin C Cooper, Simon De Givry, Martı Sánchez, Thomas Schiex, Matthias Zytnicki, and Tomas Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.

15 Yanchen Deng, Shufeng Kong, Caihua Liu, and Bo An. Deep attentive belief propagation: Integrating reasoning and learning for solving constraint optimization problems. *Advances in Neural Information Processing Systems*, 35:25436–25449, 2022.

16 Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the TSP by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181. Springer, 2018.

17 Vijay Prakash Dwivedi, Chaitanya K Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *Journal of Machine Learning Research*, 24(43):1–48, 2023.

18 Adam N Elmachtoub and Paul Grigas. Smart "predict, then optimize". *Management Science*, 68(1):9–26, 2022.

19 Aaron Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. MIPaaL: Mixed integer program as a layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1504–1511, 2020.

20 Aaron M Ferber, Taoan Huang, Daochen Zha, Martin Schubert, Benoit Steiner, Bistra Dilkina, and Yuandong Tian. SurCo: Learning linear surrogates for combinatorial nonlinear optimization problems. In *International Conference on Machine Learning*, pages 10034–10052. PMLR, 2023.

21 Matteo Fischetti and Paolo Toth. An additive bounding procedure for combinatorial optimization problems. *Operations Research*, 37(2):319–328, 1989.

22 Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

23 Minh Hoàng Hà, Claude-Guy Quimper, and Louis-Martin Rousseau. General bounding mechanism for constraint programs. In *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 158–172. Springer, 2015.

24 Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab. Los Alamos, NM (United States), 2008.

25 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970. URL: `http://www.jstor.org/stable/169411`.

26 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 18(1):6–25, 1971. `doi:10.1007/BF01584070`.

27 Stefan Hougardy and Xianghui Zhong. Hard to solve instances of the euclidean traveling salesman problem. *Mathematical Programming Computation*, 13:51–74, 2021.

28 Vinasetan Ratheil Houndji, Pierre Schaus, Mahouton Norbert Hounkonnou, and Laurence Wolsey. The weighted arborescence constraint. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 185–201. Springer, 2017.

29 Chaitanya K Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent. Learning the travelling salesperson problem requires rethinking generalization. *Constraints*, 27(1-2):70–98, 2022.

30 Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6351–6361, 2017.

31 Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), February 2016. `doi:10.1609/aaai.v30i1.10080`.

32 Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

33 Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL: `https://openreview.net/forum?id=SJU4ayYgl`.

34 Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

35 James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning: A survey. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4475–4482. International Joint Conferences on Artificial Intelligence Organization, August 2021. Survey Track. doi:10.24963/ijcai.2021/610.

36 Jena-Lonis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, 10(1):29–127, 1978.

37 Yang Li, Jinpei Guo, Runzhong Wang, and Junchi Yan. From distribution learning in training to gradient search in testing for combinatorial optimization. *Advances in Neural Information Processing Systems*, 36, 2024.

38 Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, 2017.

39 Jayanta Mandi, Emir Demirovic, Peter J Stuckey, and Tias Guns. Smart predict-and-optimize for hard combinatorial optimization problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1603–1610, 2020.

40 Jayanta Mandi, James Kotary, Senne Berden, Maxime Mulamba, Victor Bucarey, Tias Guns, and Ferdinando Fioretto. Decision-focused learning: Foundations, state of the art, benchmark and future opportunities. *arXiv preprint arXiv:2307.13565*, 2023.

41 Daniel Merchán, Jatin Arora, Julian Pachon, Karthik Konduri, Matthias Winkenbach, Steven Parks, and Joseph Noszek. 2021 Amazon last mile routing research challenge: Data set. *Transportation Science*, 2022.

42 Mathias Niepert, Pasquale Minervini, and Luca Franceschi. Implicit MLE: backpropagating through discrete exponential family distributions. *Advances in Neural Information Processing Systems*, 34:14567–14579, 2021.

43 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

44 Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2019.

45 Gerhard Reinelt. TSPLIB — A traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

46 Utsav Sadana, Abhilash Chenreddy, Erick Delage, Alexandre Forel, Emma Frejinger, and Thibaut Vidal. A survey of contextual optimization methods for decision-making under uncertainty. *European Journal of Operational Research*, 2024.

47 Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

48 Meinolf Sellmann. Theoretical foundations of CP-based lagrangian relaxation. In *International Conference on Principles and Practice of Constraint Programming*, pages 634–647. Springer, 2004.

49 Daniel Selsam, Matthew Lamm, B Benedikt, Percy Liang, Leonardo de Moura, and David L Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2018.

50 Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL: https://openreview.net/forum?id=rJXMpikCZ.

51 Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

52 Po-Wei Wang, Priya Donti, Bryan Wilder, and Zico Kolter. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, pages 6545–6554. PMLR, 2019.

53 Ziming Wang, Jun Chen, and Haopeng Chen. EGAT: Edge-featured graph attention network. In Igor Farkaš, Paolo Masulli, Sebastian Otte, and Stefan Wermter, editors, *Artificial Neural Networks and Machine Learning – ICANN 2021*, pages 253–264, Cham, 2021. Springer International Publishing.

54 Tomas Werner. A linear programming approach to max-sum problem: A review. *IEEE transactions on pattern analysis and machine intelligence*, 29(7):1165–1179, 2007.

55 Bryan Wilder. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.

56 Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. NeuroLKH: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. *Advances in Neural Information Processing Systems*, 34:7472–7483, 2021.

57 Kaan Yilmaz and Neil Yorke-Smith. A study of learning search approximation in mixed integer branch and bound: Node selection in SCIP. *AI*, 2(2):150–178, April 2021. doi:10.3390/ai2020010.