CONFIGURING MIXED-INTEGER LINEAR PROGRAMMING SOLVERS WITH DEEP METRIC LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

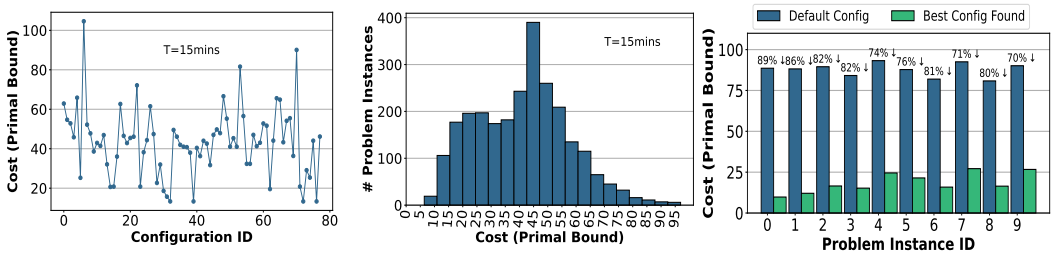
Mixed Integer Linear Programming (MILP) solvers expose a large number of configuration parameters for their internal algorithms. Solutions, and their associated costs or runtimes, are significantly affected by the choice of the configuration parameters, even when problem instances are coming from the same distribution. On one hand, using the default solver configuration leads to poor suboptimal solutions. On the other hand, searching and evaluating an exponential number of configurations for every problem instance is time-consuming and in some cases infeasible. In this work, we propose MILPTune – a machine learning-based approach to predict an instance-aware parameters configuration for MILP solvers. It enables avoiding the expensive search of configuration parameters for each new problem instance, while tuning the solver's behavior for the given instance. Our method trains a metric learning model based on a graph neural network to project problem instances to a space where instances with similar costs are closer to each other. At inference time, and given a new problem instance, we first embed the instance to the learned metric space, and then predict a parameters configuration using nearest neighbor data. Empirical results on real-world problem benchmarks show that our method predicts configuration parameters that improve solutions' costs by 10-67% compared to the baselines and previous approaches.

1 INTRODUCTION

Mixed Integer Linear Programs (MILP) are a class of NP-hard problems where the goal is to minimize a linear objective function subject to linear constraints, with some or all variables restricted to integer or binary values (Karp, 1972). This formulation has applications in numerous fields, such as transportation, retail, manufacturing and management (Paschos, 2014). For example, last-mile delivery companies repeatedly solve the vehicle routing problem as daily delivery tasks (stops and routes) change, with the goal of minimizing total delivery costs (Louati et al., 2021). Similarly, crew scheduling problems have to be solved daily or weekly in the aviation industry, where the MILP formulation is the most practical notation for expressing such problems (Deveci & Demirel, 2018). Over the years, powerful solvers have been well-researched and practically-engineered to address these problems, such as SCIP (Gamrath et al., 2020), CPLEX (Manual, 1987), and Gurobi (Bixby, 2007). These solvers mostly use branch-and-bound methods combined with heuristics to direct the search process for solving a MILP (Achterberg, 2007). In order to tune their behavior, they expose a large number of configuration parameters that control the search trajectory. For example, SCIP contains more than 2,500 parameters with integer, continuous, or categorical configuration spaces.

Figure 1 shows how branch-and-bound configuration parameters significantly affect the solution quality. In Figure 1(a), different configuration parameters directly impact the solution's cost of the same problem instance. In addition, using a single configuration for all problem instances does not yield the same solution's cost as shown in Figure 1(b). However, as observed in Figure 1(c), a significant cost reduction can be obtained by searching for a parameters configuration that makes the branch-and-bound algorithm more efficient for a given problem instance. Unfortunately, this search is time-consuming and cannot be performed online for every new problem instance. Therefore, there is a need for methods to configure solvers on-the-fly while maintaining the per-instance performance.

Recently, machine learning (ML) has shown promising results for solving MILP problems (Bengio et al., 2021; Cappart et al., 2021). The motivation behind applying machine learning is to capture



(a) Same Problem Instance (b) Same Configuration (c) Cost Reduction using SMAC

Figure 1: Effect of configuration parameters on the solution cost using SCIP (Maher et al., 2016) ($T = 15mins$). (a) changing the parameters of the branch-and-bound algorithm on the same instance. (b) using a single configuration on different instances. (c) searching the configuration space of every problem instance independently using SMAC (Lindauer et al., 2022). All problem instances have 195 variables and 1083 constraints.

redundant patterns and characteristics in problems that are being repeatedly solved. Researchers have been able to achieve promising results by either integrating models within the solver's branch-and-bound loop (Gasse et al., 2019; Li et al., 2018; Wang et al., 2021; Khalil et al., 2017b) or replacing the solver with an end-to-end solution (Khalil et al., 2017a; Vinyals et al., 2015; Bello et al., 2016; Kool et al., 2018). Learning to configure solvers has been explored early in (Kadioglu et al., 2010), where features from problem instances are hand-engineered and configurations are selected using clustering methods. More recently, configuring solvers using ML has seen growing interest (Kruber et al., 2017; Bonami et al., 2018). Valentin et al. (2022) have proposed a supervised learning approach to predict a configuration for a specific problem instance amongst a finite set of configurations. However, this approach is limited to the set of hand-crafted configurations chosen a priori for training (< 60) and restricts the ability to explore the broader configuration space.

In this work, we propose to perform an offline search of the configuration space only on a small subset of problem instances, and use their configurations for unseen instances that are similar. The main questions would be: (1) how can we measure similarity between two MILP instances? And (2) does the similarity of MILP instances correlate with the solver's performance (i.e. solution's cost)? We address these questions in a novel way through two contributions. First, we learn an embedding space for MILP instances using deep metric learning where instances with similar costs are closer to each other. Unlike existing instance-aware approaches, instances' features are not hand-engineered, but learned based on a graph convolutional network. Second, we predict a parameters configuration for new problem instances using nearest neighbor search on the learned metric space, which does not limit the number of configurations to predict from. We show that our predictions correlate with the final solution's cost. In other words, finding a closer instance in the learned metric space and using its well-performing configuration parameters would ultimately improve the solver's performance on the given instance. We evaluate our approach on real-world datasets from the ML4CO competition (ML4CO, 2021) using SCIP solver (Gamrath et al., 2020), and compare against both using an incumbent configuration from SMAC (Lindauer et al., 2022), and predicted configurations from baseline instance-aware methods. Our method solves more instances than the baselines and achieves a 10-67% improvement in the cost.

2 PRELIMINARIES

MILP Formulation. In this work, we consider MILP instances formulated as:

$$\arg\min_{\mathbf{x}} \quad \mathbf{c}^\top \mathbf{x}, \qquad \text{subject to} \quad \mathbf{A}^\top \mathbf{x} \geq \mathbf{b}, \qquad \text{and} \quad \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \tag{1}$$

where $\mathbf{c} \in \mathbb{R}^n$ denotes the coefficients of the linear objective, $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$ denote the coefficients and upper bounds of the linear constraints respectively. n is the total number of variables, $p \leq n$ is the number of integer-constrained variables, and m is the number of linear constraints. The goal is to find feasible assignments for $\mathbf{x}$ that minimize the objective $\mathbf{c}^\top \mathbf{x}$. A MILP solver constructs a search tree to find feasible solutions with minimum costs. The cost of the solution found by the solver by the end of its search, or if the time limit is reached, is called the primal bound. It serves as

an upper bound to the set of feasible solutions. While there are other methods to measure the solver's performance, (e.g. dual bound, primal-dual gap, primal-dual integral (Achterberg, 2007)), we adopt the primal bound at the end of the time limit for the purpose of training the metric learning model.

Metric Learning. Deep learning models require a vast amount of data in order to make reliable predictions. In a supervised learning setting, the goal is to map inputs to labels as in a standard classification or regression problem. When the number of classes is huge, supervised learning fails to address real-world applications. For example, face verification systems have a large number of classes but the number of examples per class is small or non-existent (Schroff et al., 2015). In this case, the goal is to develop a model that learns object categories from a few training examples. But deep learning models do not work well with a small number of data points. In order to address this issue, we learn a similarity function between data points, which helps us to predict object categories given small data for training. This paradigm is known as metric learning (Kulis et al., 2013). In this paradigm, a model is trained to learn a distance function (or similarity function) over the inputs themselves. Here, similarity is subjective, so the distance may have a different meaning depending on the data. In other words, the model learns relationships in the training data regardless of what it actually means in its domain application. Metric learning has seen growing adoption in real-world applications such as face verification (Schroff et al., 2015; Wang et al., 2018; Deng et al., 2019), video understanding (Lee et al., 2018) and text analysis (Davis & Dhillon, 2008).

Measuring distances is a critical aspect of metric learning. Given two instances of some object representation, $\mathcal{I}_i$ and $\mathcal{I}_j$, a distance function, d, measures how far the two instances are from each other. The Euclidean distance is less meaningful in higher dimensions even if the data is perfectly isotropic and features are independent from each other. Therefore, the goal is to define new distance metrics in higher dimensional spaces that are based on the properties of the data itself. These are non-isotropic distances reflecting some intrinsic structures of the data. A parametric model is trained to project instances to the new metric space through either a linear transformation of the data such as the Mahalanobis distance (De Maesschalck et al., 2000), or a non-linear transformation of the data using deep learning (Kaya & Bilge, 2019).

In this setup, instead of requiring labels for training, the model requires weak supervision at the instance level, where triplets of (anchor a, positive p, negative n) are fed into the model. The model is trained to learn a distance metric that puts positive instances close to the anchor and negative instances far from the anchor. This is achieved by a Triplet loss function (Schroff et al., 2015):

$$L = \sum_{i}^{N} [||f(\mathcal{I}_i^a) - f(\mathcal{I}_i^p)||^2 - ||f(\mathcal{I}_i^a) - f(\mathcal{I}_i^n)||^2 + \alpha]_+ \tag{2}$$

where N is the number of triplets sampled during training. $\mathcal{I}^a$, $\mathcal{I}^p$ and $\mathcal{I}^n$ represent the anchor instance, similar instance and dissimilar instance respectively. f is a parametric model that projects instances to a learned metric space. The loss increases when the first squared distance (anchor-positive) is larger than the second squared distance (anchor-negative). So, f is trained to decrease this loss. In other words, it tries to make the first squared distance smaller, and the second square distance larger. Here, the loss, L, will be equal to zero if the first squared distance is α-less than the second squared distance. In Section 4, we present a number of modifications during training in order to avoid having a zero-loss early during training as it ends the learning process prematurely.

3 Related Work

Algorithm Configuration. Methods for algorithm configuration try to find a single robust configuration [1] across a set of problem instances i from a finite set $\mathcal{I}$. Random search (Bergstra & Bengio, 2012), evolutionary algorithms (Olson et al., 2016), Bandit methods (Li et al., 2017), and Bayesian-based optimization (Shahriari et al., 2015) are among the top performing methods. The open-source SMAC tool (Lindauer et al., 2022) implements state-of-the-art Bayesian optimization methods described in (Hutter et al., 2011). It empirically searches the configuration space by running sequential rounds of evaluation where it eliminates poorly-performing candidate configurations from subsequent rounds and collects more evidence using other problem instances on potential candidate configurations. With a massive search space (both number of parameters and their ranges), and

[1] Also called incumbent configuration in the context of parameters configuration search; not to be confused with the incumbent solution of the solver itself, which is the $\mathbf{x}$'s assignment with minimum cost.

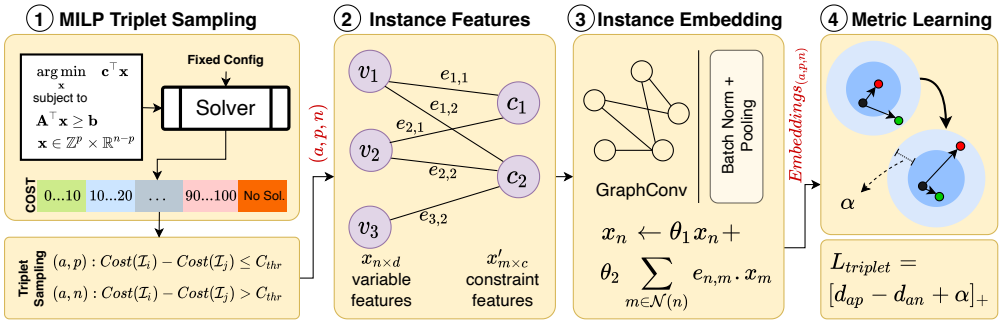


Figure 2: MILPTune Metric Learning Method. Triplet samples are first collected on a few instances using a fixed configuration. Instance features are extracted as a bipartite graph (Gasse et al., 2019), then embedded using a graph convolutional network. A triplet loss (Schroff et al., 2015) function is used to train the model end-to-end. C_{thr}: cost threshold for similarity, a: anchor instance, p: positive/similar instance, n: negative/dissimilar instance, θ: learnable parameters of the GNN, d: distance between embeddings, as defined in Equation 2.

a large number of problem instances to evaluate on, an exhaustive search is infeasible, ultimately resulting in a configuration that performs poorly on average. Although narrowing the search space may be achieved by using expert-crafted features (Zhang, 2022), it is restricted and cannot be generally applied. Instance-aware configuration methods have been explored early in ISAC (Kadioglu et al., 2010). The method extracts features from problem instances and performs clustering to select parameter configurations for new instances. However, features are hand-engineered (i.e. shallow embedding) and need to be adapted for each problem as in (Ansótegui et al., 2016). Hydra-MIP Xu et al. (2011) enhanced this approach by including features from short solver runs before selecting a configuration for a complete solver run. Our approach is different since problem features are *learned* during training (i.e. deep embedding), and correlates similarity to the costs of final solutions. Moreover, previous approaches assign a single parameters configuration for each cluster, which limits the portfolio of configurations available at inference time. Section 5 compares our method to shallow embedding, and Appendix C presents a thourough discussion and a detailed comparison.

Machine Learning for Combinatorial Optimization. Learning-based optimization methods have seen growing interest lately (Bengio et al., 2021; Cappart et al., 2021). Broadly speaking, they can be divided into methods inside the solvers (Khalil et al., 2017b; Gasse et al., 2019; Li et al., 2018; Wang et al., 2021), methods outside the solvers (Kruber et al., 2017; Bonami et al., 2018), and methods that replace the solvers (Khalil et al., 2017a; Vinyals et al., 2015; Bello et al., 2016; Kool et al., 2018). Our work is amongst methods outside the solver, which aims at improving the solver's performance by instantly predicting instance-aware parameters configuration. This is orthogonal to existing work and can be benefit from existing hyper-parameter search methods when performed offline.

4 Methodology

The fundamental motivation of our work is to define similarity among MILP instances based on their final solutions' costs after running the solver on the same environment (i.e. host machine, software environment, configuration parameters, time limit, and random seed), and under the assumption that all MILP instances are coming from the same problem distribution. Same distribution instances are problem instances that share similar number of variables and constraints, and define a problem that is being solved repeatedly. Towards that goal, our approach is to use deep metric learning to learn the instance embeddings and predict instance-aware parameters configuration. Figure 2 shows an overview of the learning methodology. In contrast to supervised learning where a large amount of data needs to be collected in order to train the model, MILPTune collects training data on a small subset of the problem instances available. The data is fed into our model in triplets of (anchor, positive, negative) instances. The model learns an embedding space where positive instances are close to their anchor than negative instances. Below, we elaborate on each step of our method.

1. MILP Triplet Sampling. In Step 1 of Figure 2, and given two MILP instances $\mathcal{I}_i$ and $\mathcal{I}_j$, we need to define whether they are similar or not. As discussed in Section 2, similarity is subjective and

Algorithm 1: MILP Triplet Sampling	**Algorithm 2: Predicting a Configuration**
Input : MILP Instances ($\mathcal{I}_i$), Costs (C_i)s	**Input :** Unseen MILP Instance (I)
Input : Cost Threshold (C_{thr})	**Input :** Model (M), k, $n\_configs$
Output: Triplets (a, p, n)	**Output:** Configuration parameters
1. Select anchor a from solved training subset	1. Embed instance using M
2. Find p with $\lvert C(\mathcal{I}_a) - C(\mathcal{I}_p)\rvert < \mathrm{C}_{thr}$	2. Perform KNN
3. Find n with $\lvert C(\mathcal{I}_a) - C(\mathcal{I}_n)\rvert \gg \mathrm{C}_{thr}$	3. Retrieve k nearest neighbors stored configs
4. Train model for e_1 epochs	4. Select best-performing $n\_configs$
5. Find n with $\lvert C(\mathcal{I}_a) - C(\mathcal{I}_n)\rvert > \mathrm{C}_{thr}$	from each neighbor
6. Train model for e_2 epochs	5. Sort configurations by cost
Using Loss $\mathrm{L}_{triplet} = [\mathrm{d}_{ap} - d_{an} + \alpha]_+$	6. Use best configuration as input to solver

depends on the domain. In our case, there is no natural way to find out whether two instances are similar or not just from their given problem formulation (Equation 1). Even though one could map it to a graph isomorphism problem, small perturbations of $\mathbf{A}$ can lead to different solutions from the solver. For example, a slight change in a constraint's coefficients could make the constraint trivial, or make the MILP instance infeasible. In our method, if the difference between the solution cost of instance $\mathcal{I}_i$ and $\mathcal{I}_j$ is below a certain threshold C_{thr}, then $\mathcal{I}_i$ and $\mathcal{I}_j$ are considered similar for the purpose of training the model. If the cost difference is above C_{thr}, the instances are considered dissimilar. In the triplet sampling step, the goal is to look up for similar and dissimilar instances in the training dataset. Algorithm 1 shows the steps for the mining and training procedures. We introduce a new sampling schedule for our training procedure. The goal is to avoid crunching the loss (Equation 2) to zero prematurely. Lines 1-4 in Algorithm 1 starts with hard negative sampling by looking for instances that have larger cost difference. The idea is that when starting with these negative pairs (a, n), the model gets a chance to be able to push their embeddings further away from each other. Then, in lines 5-6, this restriction is relaxed and the training loop starts seeing negative instances that have slightly larger cost difference than positive instances.

2. Feature Extraction. The MILP formulation represented in Section 2 does not restrict the order of the variables in the objective, nor the number and order of the constraints. Therefore, a feature extractor needs to be invariant to their order to handle instances of varying sizes. In Step 2 of Figure 2, we represent a MILP instance using the bi-partite graph representation from (Gasse et al., 2019). Each variable is represented as a node, and each constraint is also represented as a node. An undirected edge between a variable, v_i, and a constraint, a_j, exists if v_i appears in a_j, that is if $\mathbf{A}_{ij} \neq 0$. Variable nodes have features represented as the variable type (binary, integer or continuous) in addition to its lower and upper bounds. They are represented as $X \in \mathbb{R}^{n \times d}$, where n is the number of nodes and d is the features dimension. Constraint nodes have features represented in their (in)equality symbol ($<$, $>$, $=$). They are represented as $X' \in \mathbb{R}^{m \times a}$, where m is the number of constraints and a is the features dimension. Edge features represent the coefficients of a variable appearing in a constraint, $E \in \mathbb{R}^{n \times m \times e}$, where e is the number of edges. These features are extracted once before the solver starts the branch-and-bound procedure, namely at the root node. Therefore, each problem instance has a single graph structure representation before any cuts happen at the root node (part of the heuristics-based algorithms). While the original representation in (Gasse et al., 2019) has additional features, we only extract the features of the problem instance, and not the solver's state.

3. Instance Embedding. In Step 3 of Figure 2, we parameterize our distance metric model using a graph convolutional neural network (Kipf & Welling, 2016). The network structure has four convolutional layers, and the convolutional operator is implemented as defined in (Morris et al., 2019). The network parameters, θ_1 and θ_2, are updated within the end-to-end training procedure where features of the variables are updated as: $x_n \leftarrow \theta_1 x_n + \theta_2 \sum_{m \in \mathcal{N}(n)} e_{n,m}.x'_m$. Similarly, the features of the constraints are updated as $x'_m \leftarrow \theta_1 x'_m + \theta_2 \sum_{n \in \mathcal{N}(m)} e_{m,n}.x_n$. Graph embeddings are then passed through batch normalization, max-pooling and attention pooling layers to produce a latent vector which is used for the downstream metric learning loss.

4. Training. In Step 4 of Figure 2, the model is trained end-to-end using the loss function defined in Equation 2. The distance function used is the Euclidean distance on the learned metric space. The training proceeds for a number of predefined epochs, while ensuring that the loss does not fall to zero by adopting the proposed triplet sampling schedule. The larger the value of α, the further positive

instances are pushed away from negative ones. However, choosing a large value of α will make the model set the value of the distance function d as zero. Thus, α should be chosen carefully for training.

Predicting Parameters Configuration. A trained model is a model capable of measuring a distance metric between MILP instances. The final embeddings of the instances are saved to be used in the prediction step. Algorithm 2 gives the steps performed for predicting a parameters configuration for a new unseen MILP instance. In Step 1, the problem instance is first embedded using the trained model. In Step 2, we perform a nearest neighbor search on the learned metric space. We introduce two tuning parameters for the prediction: (1) k, representing the number of nearest neighbors we want to find, and (2) $n\_configs$, representing the number of configurations (sorted by their solution cost) for each neighbor that we want to consider. In Steps 4-6, we predict a parameters configuration as the one with the minimum cost. If $k = 1$ and $n\_configs = 1$, then the algorithm predicts the lowest cost configuration parameters of the nearest neighbor. In multi-core environments (e.g. cloud), a practitioner may choose to run the solver in parallel using different configuration parameters and gather an ensemble of solutions for the new problem instance. In this case, k and $n\_configs$ can be exposed as hyper-parameters for the prediction model. In Section 5, we will show that similar instances correlate with solution quality.

Configuration Space Exploration. The goal of our method is to eliminate the need for collecting exponential number of data points to train a supervised prediction model for parameters configuration. However, during inference, the model may have access to embedded MILP instances that were never seen during training. The idea is to find similarity in the embedded space, and retrieve a parameters configuration that proved to perform well on the similar instance (from a previous solver run). We refer to this as the exploration problem, where a parameters search method, such as SMAC (Lindauer et al., 2022), could be used offline to search for the best parameters configuration only on a small subset of the data. New instances will benefit from the already-explored configuration space.

5 EXPERIMENTS

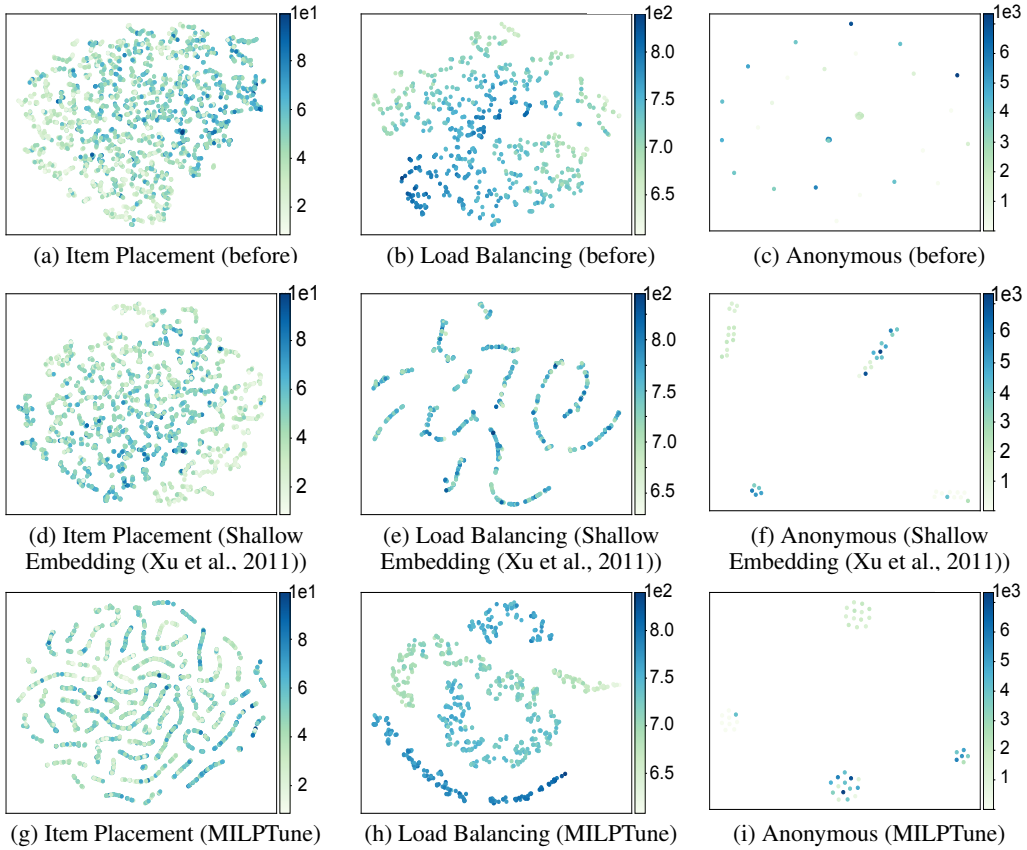
Dataset. We used the publicly available dataset from the ML4CO competition (ML4CO, 2021). The dataset consists of three problem benchmarks. The first two problem benchmarks (item placement and load balancing) are extracted from applications of large-scale systems at Google, while the third benchmark is an anonymous problem extracted from a large-scale industrial application. The item placement and load balancing benchmarks contain 10,000 MILP instances for training (9,900) and testing (100), while the anonymous problem contains only 118 instances (98 and 20 for training and testing respectively). Detailed descriptions of the dataset is available in (ML4CO, 2021).

Setup. The experimental results are obtained using a machine with Intel Xeon E5-2680 2x14cores@2.4 GHz, 128GB RAM, and a Tesla P40 GPU. The model was developed using PyTorch (v1.11.0+cu113) (Paszke et al., 2019), Pytorch Geometric (v2.0.4) (Fey & Lenssen, 2019), and PyTorch Metric Learning (v1.3.0) (Musgrave et al., 2020). We used Ecole (v0.7.3) (Prouvost et al., 2020) for graph feature extraction, convolution operators modified and adopted from (Valentin et al., 2022), PySCIPOpt (v3.5.0) (Maher et al., 2016) as the MILP solver, and SMAC3 (v1.2) (Lindauer et al., 2022) for the offline search.

MILP Triplet Sampling. Given the training dataset, we run the MILP solver on all instances using the default parameters configuration of the solver with a time limit of 15 minutes. The total number of solved instances by the end of the time limit were 2599, 1727 and 38 for the item placement, load balancing and anonymous datasets respectively. This represents 26%, 17% and 38% of the training datasets respectively. We implemented the triplet sampling schedule as discussed in Section 4, where hard negative sampling was used for the first 50 epochs, and the training continues for 100 epochs in total. We used a batch size of 256 for the item placement, 64 for load balancing, and the full 98 instances for the anonymous dataset.

Model Training. The model consists of a graph neural network of four layers with 64 as the dimension of the hidden layers. The output from the convolutional layers is passed into a batch normalization layer, followed by a max pooling layer and an attention pooling layer. The output embedding size is chosen to be 256. We set $\alpha = 0.1$ in the loss function.

Instance Embedding. We visualize the instance embeddings of the GNN before and after model training and compare it to using shallow embeddings in Figure 3. The color bar represents the cost



(a) Item Placement (before) (b) Load Balancing (before) (c) Anonymous (before)

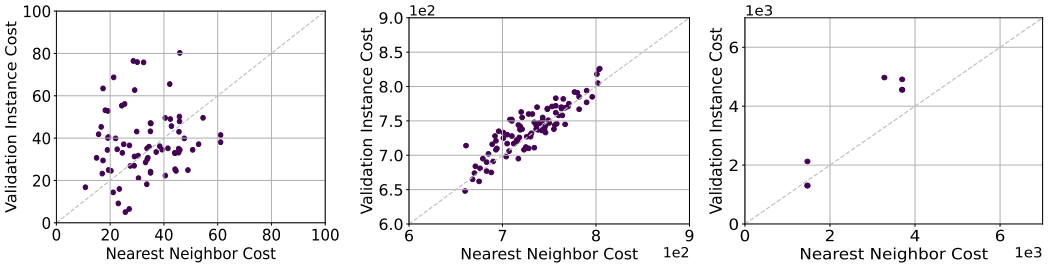
(d) Item Placement (Shallow Embedding (Xu et al., 2011)) (e) Load Balancing (Shallow Embedding (Xu et al., 2011)) (f) Anonymous (Shallow Embedding (Xu et al., 2011))

(g) Item Placement (MILPTune) (h) Load Balancing (MILPTune) (i) Anonymous (MILPTune)

Figure 3: (a-c) Before embedding, (d-f) shallow embedding using (Xu et al., 2011) and (g-i) MILP-Tune deep embedding (learned). Colors represent the solutions' cost. In the item placement dataset, shallow embedding does not offer any discriminative capability. In the load balancing dataset, it could cluster problem instances, but clusters are not correlated with the final solver's costs. Shallow embedding uniquely embeds the anonymous dataset and the embedding is correlated to the final costs. The discriminative power of MILPTune's deep embedding is evident in the three datasets. Visualization using t-SNE (Van der Maaten & Hinton, 2008). Dataset characteristics in Appendix A.

of the solution using the default configuration parameters as discussed in Section 4. The shallow embedding vector encodes presolving statistics as in (Xu et al., 2011), which include the problem size, the minimum, maximum, average and standard deviation of the objective coefficients ($\mathbf{c}$) and the constraints coefficients ($\mathbf{A}$, $\mathbf{b}$). While Hydra-MIP's shallow embedding includes more features such as the cutting planes usage and the branch-and-bound tree information, such information is not available before running the solver[2]. From Figure 3, we observe that in the item placement dataset, shallow embeddings do not offer any discriminative power to the problem instances. In the load balancing dataset, shallow embeddings could indeed cluster problem instances, but clusters are not correlated with the final solver's costs. In the anonymous dataset, instances with similar costs were clustered close to each other, which gives shallow embedding a discriminative power in this case. Analyzing this result in light of the datasets statistics (please see Table 2 in Appendix A), we see that the anonymous dataset has a high variance in the number of variables and constraints. Therefore, a feature vector that includes aggregated values could distinguish the problem instances. On the other hand, item placement has the same number of variables and constraints. A shallow feature vector cannot capture the graph connectivity properties, nor the coefficients values. Between these two cases, the load balancing dataset has the same number of variables, while the number of constraints do not have a high variance (64,081 to 64,504 constraints). Shallow embedding was able to cluster

[2]The implementation of shallow embedding is provided in the supplementary material. There is no publicly available implementation of Hydra-MIP.



(a) Item Placement (MAE=18.07) (b) Load Balancing (MAE=14.46) (c) Anonymous (MAE=801.36)

Figure 4: Cost (primal bound) of the predicted configuration from the nearest neighbor in the learned metric space (x-axis) as compared to the actual cost after using it for the validation instance (y-axis).

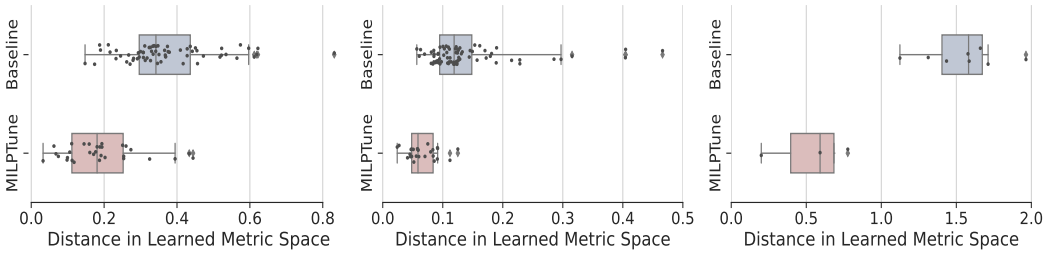
Table 1: MILPTune vs. Baselines. There are 100, 100 and 20 test instances for the item placement, load balancing and anonymous datasets respectively. Imprv. represents the solution's cost improvement over the lowest cost from other methods' configurations. Shallow embedding uses the same embedding vector as (Xu et al., 2011). MILPTune is evaluated at $k = 1$ and $n\_configs = 1$.

	Item Placement		Load Balancing		Anonymous	
Configuration	Wins	Imprv. ↓	Wins	Imprv. ↓	Wins	Imprv. ↓
No Solution Found	2	0	0	0	11	0
Default SCIP Config	0	0	0	0	0	0
SMAC (Lindauer et al., 2022)	15	0.26±0.09	20	0.08±0.07	1	0.01±0.00
Hydra-MIP (Xu et al., 2011)	22	0.32±0.12	27	0.05±0.02	0	0
Shallow Embedding + KNN	30	0.35±0.10	17	0.07±0.05	4	0.20±0.05
MILPTune Deep Embedding	**31**	**0.66±0.12**	**36**	**0.10±0.05**	**4**	**0.67±0.03**

problem instances, but its clusters were not correlated to the final solver's costs. MILPTune's learned embedding is discriminative in the three datasets.

MILPTune Prediction Accuracy. A key question in our approach is whether the nearest neighbor in the embedding space would exhibit a similar solver behavior when using its parameter configuration. Here, we embed the validation instances using our trained model, and then obtain a parameters configuration from the nearest neighbor. Then, we run the solver using the predicted parameters configuration on the validation instances (T=15mins). Figure 4 plots the solution's cost of the predicted parameters configuration from the nearest neighbor (x-axis) vs. its solution's cost on the validation instance (y-axis). It shows that there is indeed a correlation between the final cost of the solution using the predicted parameters configuration, and the stored nearest neighbor cost using that configuration. The mean absolute errors (MAE) were 18.07, 14.46, and 801.36 for item placement, load balancing and anonymous respectively. This correlation proves that in reality similar MILP instances based on the learned metric space expose similar solver behaviors yielding similar solution costs. In other words, finding a good parameters configuration for one problem instance can be used for similar instances without repeating an exhaustive search at deployment time.

Comparing to Baselines. We compare MILPTune against baselines in Table 1. The first baseline is using SCIP's default configuration, which is usually used by most practitioners. In addition, we obtain an incumbent configuration by performing a configuration space search on the training instances using SMAC (Lindauer et al., 2022). We perform this search for each dataset separately. Although the number of unique configurations explored was 51012 over a period of over 12000 core-hours, this represents a small subset of the configuration space. Moreover, we implement Hydra-MIP (Xu et al., 2011) which uses a statistics-based vector for instance embedding and pair-wise weighted random forests for configuration selection. In Hydra-MIP, the pairwise weighted random forests (RFs) method is used to select amongst m algorithms for solving the instance, by building $m.(m-1)/2$ RFs and taking a weighted vote. In our processed dataset, the number of *unique* configurations explored offline using SMAC are 22580, 27971 and 461 for the item placement, load balancing and anonymous training datasets respectively. Among those, the number of *unique* configurations that worked *best* on their respective instances (excluding unsolved instances) are 4325, 3987 and 53. As a result for the Hydra-MIP approach, building the portfolio by performing algorithm selection using pairwise RFs is



(a) Item Placement (b) Load Balancing (c) Anonymous

Figure 5: Similarity in the learned embedding space. The x-axis represents the distance between the validation instance and its nearest neighbor in the learned metric space. The y-axis represents the method that gives a better solution. The closer the nearest neighbor to the validation instance, the better the predicted configuration by MILPTune. Baseline represents the method with the lowest cost.

computationally infeasible (memory and compute). For example, in the item placement dataset, a total of $4325 \times 4324/2 = 9350650$ RFs are needed. To obtain results for Hydra-MIP, we selected a subset of the top 100 performing configurations in the item placement and the load balancing datasets, and used all 53 best configurations of the anonymous dataset. Lastly, we compare against using the shallow embedding from Hydra-MIP with KNN, which avoids the scalability limitation of RFs. Table 1 reports the number of instances solved with the lowest cost in each method, along with the cost improvement over the second-lowest cost from other methods. We see that MILPTune predicts configurations that solve more instances, with a 10-67% improvement in the cost of the objective function (confidence level of 95%).

Similarity in the Learned Embedding Space. We investigate how MILPTune brings instances with similar final costs close to each other by plotting the winning MILPTune predictions against their distance from their neighbors in the learned embedding space. In Figure 5, the x-axis represents the distance between the validation instance and its nearest neighbor, while the y-axis represents which method offers a better parameters configuration. We observe that the smaller the distance between the validation instance and its nearest neighbor in the learned embedding space, the more probable the neighbor's parameters configuration to yield a better solution than other baselines. In other words, MILPTune correlates the similarity of the learned embedding to the final solution costs.

Limitations. Our adoption of metric learning in configuring MILP solvers relies on data coming from the same distribution. This means that in order to learn the internal characteristics of the instances, the MILP formulation needs to represent a problem being solved repeatedly, which is materialized in the number of variables or constraints in the problem. That is why our learned model separated instances of the Anonymous dataset (high variance in the number of variables and constraints) by a large distance as seen in Figure 5(c). We conclude that similarity learning works best when the underlying problem instances have common patterns in their representation.

Reproducibility. In Section 5, we refer the reader to the original dataset to download. A link to the processed dataset (learned embeddings) is available in the supplementary material. In addition, we describe our setup for training and the pipeline architecture. The source code for the package along with the training recipe is available in the supplementary material.

6 Conclusions and Future Work

We address the challenge of selecting configuration parameters for Mixed-Integer Linear Programming (MILP) solvers. We propose MILPTune – an instance-aware method that predicts a parameters configuration for new problem instances using deep metric learning. We show that it is possible to *learn* a reliable similarity metric between MILP instances that correlates with the solver's behavior (i.e. final solution's cost) when using the same parameters configuration. As compared to existing approaches, MILPTune offers more discriminative power to instances' features, and predicts parameters configurations that leads to better solutions by 10-67%. In the future, we will investigate the potential of utilizing the learned similarity metric to generate new parameters configurations that were not seen during the offline search. This paves the way for new configuration space search algorithms based on the learned similarity of problem instances.

REFERENCES

Tobias Achterberg. Constraint integer programming. 2007. 1, 3, 13

Carlos Ansótegui, Joel Gabas, Yuri Malitsky, and Meinolf Sellmann. Maxsat by improved instance-specific algorithm configuration. *Artificial Intelligence*, 235:26–39, 2016. 4, 13, 14

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016. 2, 4

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. 1, 4

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012. 3

Bob Bixby. The gurobi optimizer. *Transp. Re-search Part B*, 41(2):159–178, 2007. 1

Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 595–604. Springer, 2018. 2, 4

Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021. 1, 4

Jason V Davis and Inderjit S Dhillon. Structured metric learning for high dimensional problems. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 195–203, 2008. 3

Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. The mahalanobis distance. *Chemometrics and intelligent laboratory systems*, 50(1):1–18, 2000. 3

Jiankang Deng, Jia Guo, Niannan Xue, and Stefanos Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 4690–4699, 2019. 3

Muhammet Deveci and Nihan Çetin Demirel. A survey of the literature on airline crew scheduling. *Engineering Applications of Artificial Intelligence*, 74:54–69, 2018. 1

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 6

Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, et al. The scip optimization suite 7.0. 2020. 1, 2

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. 2, 4, 5, 14

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pp. 507–523. Springer, 2011. 3

Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac–instance-specific algorithm configuration. In *ECAI 2010*, pp. 751–756. IOS Press, 2010. 2, 4, 13, 14

Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pp. 85–103. Springer, 1972. 1

Mahmut Kaya and Hasan Şakir Bilge. Deep metric learning: A survey. *Symmetry*, 11(9):1066, 2019. 3

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017a. 2, 4

Elias B Khalil, Bistra Dilkina, George L Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Ijcai*, pp. 659–666, 2017b. 2, 4

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 5

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018. 2, 4

Markus Kruber, Marco E Lübbecke, and Axel Parmentier. Learning when to use a decomposition. In *International conference on AI and OR techniques in constraint programming for combinatorial optimization problems*, pp. 202–210. Springer, 2017. 2, 4, 14

Brian Kulis et al. Metric learning: A survey. *Foundations and Trends® in Machine Learning*, 5(4): 287–364, 2013. 3

Joonseok Lee, Sami Abu-El-Haija, Balakrishnan Varadarajan, and Apostol Natsev. Collaborative deep metric learning for video understanding. In *Proceedings of the 24th ACM SIGKDD International conference on knowledge discovery & data mining*, pp. 481–490, 2018. 3

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017. 3

Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in neural information processing systems*, 31, 2018. 2, 4

Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022. 2, 3, 6, 8, 15

Ali Louati, Rahma Lahyani, Abdulaziz Aldaej, Racem Mellouli, and Muneer Nusir. Mixed integer linear programming models to solve a real-life vehicle routing problem with pickup and delivery. *Applied Sciences*, 11(20):9551, 2021. 1

Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, pp. 301–307. Springer International Publishing, 2016. doi: 10.1007/978-3-319-42432-3_37. 2, 6

CPLEX User's Manual. Ibm ilog cplex optimization studio. *Version*, 12:1987–2018, 1987. 1

ML4CO. Machine learning for combinatorial optimization - neurips 2021 competition. ml4co competition. `https://www.ecole.ai/2021/ml4co-competition/`, 2021. Accessed: 2022-05-16. 2, 6

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pp. 4602–4609, 2019. 5

Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. Pytorch metric learning, 2020. 6

Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the genetic and evolutionary computation conference 2016*, pp. 485–492, 2016. 3

Vangelis Th Paschos. *Applications of combinatorial optimization*, volume 3. John Wiley & Sons, 2014. 1

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. 6

Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020. URL https://openreview.net/forum?id=IVc9hqgibyB. 6, 14

Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823, 2015. 3, 4

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1): 148–175, 2015. 3

Romeo Valentin, Claudio Ferrari, Jérémy Scheurer, Andisheh Amrollahi, Chris Wendler, and Max B. Paulus. Instance-wise algorithm configuration with graph neural networks, 2022. URL https://arxiv.org/abs/2202.04910. 2, 6, 14

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008. 7

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015. 2, 4

Hao Wang, Yitong Wang, Zheng Zhou, Xing Ji, Dihong Gong, Jingchao Zhou, Zhifeng Li, and Wei Liu. Cosface: Large margin cosine loss for deep face recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5265–5274, 2018. 3

Runzhong Wang, Zhigang Hua, Gan Liu, Jiayi Zhang, Junchi Yan, Feng Qi, Shuang Yang, Jun Zhou, and Xiaokang Yang. A bi-level framework for learning to solve combinatorial optimization on graphs. *Advances in Neural Information Processing Systems*, 34, 2021. 2, 4

Junyuan Xie, Ross Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, pp. 478–487. PMLR, 2016. 14

Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Hydra-mip: Automated algorithm configuration and selection for mixed integer programming. In *RCRA workshop on experimental evaluation of algorithms for solving problems with combinatorial explosion at the international joint conference on artificial intelligence (IJCAI)*, pp. 16–30, 2011. 4, 7, 8, 13, 14

Mengyuan Zhang. Mdo's method for neurips 2021 ml4co's configuration task. 2022. 4

APPENDICES

A DATASET DETAILS

The dataset is available to download from the ML4CO competition website [3] with a full description on the problems formulation and their sources. In this section, we show some statistics on the dataset and reflect on how they affect our approach of metric learning.

Table 2 shows the number of variables and constraints in each dataset. In Section 4, we built our approach on the basis that MILP similarity learning works on problem instances coming from the same distribution. From the perspective of a solver, high variance in the number of variables or constraints (e.g. Anonymous dataset) has a direct impact on its solution. Hence, it also affects the embedding of these instances. While the high variance gives more discriminative power to the learned

[3] Link: https://github.com/ds4dm/ml4co-competition

Table 2: Dataset Statistics

	# Variables			# Constraints		
Dataset	Count	Avg.	Median	Count	Avg.	Median
Item Placement	195	195	195	1,083	1,083	1,083
Load Balancing	61,000	61,000	61,000	64,081–64,504	64,307	64,308
Anonymous	1,613–92,261	33,998	4,399	1,080–12,6621	43,373	2,599

model of similarity, it does not directly serve the purpose of finding a parameters configuration for new unseen instances from the nearest neighbor. The reason is that the nearest neighbor might indeed not be close in distance in the learned metric space because of the high discriminative power of the model (see Figure 5), and the predicted parameters configuration would not be directly correlated to the solver's solution. As revealed later, the anonymous dataset was extracted from MIRPLIB – a library of maritime inventory routing problems [4]. Although instances were compiled from a dataset from one business domain, it did not mean they are similar. Therefore, it is critical that the definition of "same distribution" instances include the number of variables and constraints for the purpose of finding a parameters configuration using metric learning.

B On Using The Primal Bound for Similarity Measurement

There are a number of metrics used to measure a solution's quality and a solver's performance (Achterberg, 2007). The primal bound represents the value of the solution found (to be minimized), which also serves as an upper bound to the feasible solutions. The dual bound gives a lower bound on the feasible solutions and is usually used within the branch-and-bound algorithm to trim out solutions. It is also used to measure the optimality of the solution (i.e. how far it is from the optimal solution). The primal-dual gap (also called duality gap) is the difference between the two values. A primal-dual gap integral is the area under the primal bound and dual bounds over the solving time.

As discussed in Section 2, similarity is subjective and mainly depends on the problem domain. However, in a production deployment where MILP instances are being solved repeatedly, there is a time limit, T, for obtaining a solution. Therefore, in this work, we adopt the primal bound at the end of the solver's run as the metric of similarity. The reason we did not use the primal-dual integral as used in the competition is that it requires repetitive querying of the solver's state. As a consequence, the primal-dual integral metric is sensitive to the solver's states at the fixed time intervals where the state is read. This has yielded difficulties in finding a threshold cost that should determine whether two instances are similar or not. Using the primal bound at the end of the time limit eliminates inconsistencies the primal-dual integral calculations. It proved to be a more robust metric for learning similarity among MILP instances.

C MILP Shallow Embedding Methods

As previously discussed in Section 3, instance-specific configuration methods have been proposed early in (Kadioglu et al., 2010; Ansótegui et al., 2016; Xu et al., 2011). MILPTune is similar to these methods in the approach of building an offline portfolio of configuration parameters and predicting a parameters configuration from them at run time using unsupervised learning methods. For example, ISAC uses g-means clustering (Kadioglu et al., 2010), Hydra uses decision forests (Xu et al., 2011), and MILPTune uses k-nearest neighbors. The main difference between MILPTune and previous techniques is the embedding method of the problem instance (i.e. feature extraction). In previous work, a problem instance is characterized by a "shallow embedding" of the given problem definition. In particular, ISAC hand-crafts a feature vector for the Set Covering problem that includes a normalized cost vector $\mathbf{c}$, bag densities, item costs and coverings, in addition to other density functions. These values are aggregated (using minimum, maximum, average and standard deviation) to construct the final feature embedding of the problem instance. Similarly, MaxSAT (Ansótegui et al., 2016) uses ISAC's (Kadioglu et al., 2010) method and focuses on the maximum satisfiability

[4] link `https://mirplib.scl.gatech.edu/instances`

Table 3: Summary of instance-aware algorithm configuration methods.

	MaxSAT/ISAC (Kadioglu et al., 2010) (Ansótegui et al., 2016)	**Hydra-MIP** (Xu et al., 2011)	**MILPTune**
Features	Hand-crafted	Hand-crafted	Learned
Embedding	Shallow	Shallow	Deep
Injectivity	Non-injective	Non-injective	Injective
Offline Search	Genetic Algorithm	Regression/Iterative	Bayesian Search
Inference	G-means Clustering	Decision Forests	KNN
# Configs Predicted	1	k	k

problem, with hand-engineered features that include problem size, balance features and local search probe features. Hydra (Xu et al., 2011) extracts more features by executing short runs of the solver (CPLEX) using a default configuration on each new instance. These features include pre-solving statistics, cutting planes usage, and the branch-and-bound tree information.

In MILPTune, a problem instance embedding is *learned* during training. In other words, MILPTune uses "deep embedding" that is characterized by the learnable parameters (θ) of the GNN. As shown in the literature, shallow embedding functions are not injective (Xie et al., 2016). This could lead in two completely different problem instances to have the same embedding. Moreover, and by design, shallow embeddings are not necessarily correlated with the final costs of the solver's solutions.

We conclude that problems that are being solved repeatedly, where the size of the problem remains relatively similar, but the coefficients ($\mathbf{c}$, $\mathbf{A}$, $\mathbf{b}$) vary, deep embedding has a larger discriminative power over shallow embedding. In problems where the problem size varies widely, shallow embedding could be used. In designing a system that is invariant to the problem size, deep embedding addresses the need without hand-engineering features for each problem separately.

D Generalizing to Other Solvers

A solution's cost depends primarily on: (1) the problem instance, (2) the solver used (including the specific solver version), (3) the time limit, (4) the hardware resources given to the solver (cores and memory), in addition to (5) the configuration parameters. For the purpose of learning similarity between MILP instances, the solver's costs are used as a subjective measure of the similarity between two instances that use the same solver version, time limit, hardware resource, and configuration parameters. Replacing the solver with another solver is possible for the sake of getting costs that could be used to measure the similarity between different MILP instances. However, it is critical to fix all parameters of the solving environment (hardware, solver tool and its version, time limit, configuration parameters) in order for the cost to be representative of the similarity.

Once a similarity measurement is established, two similar instances under solver's 1 environment could potentially be used to determine that these two instances will have similar costs under solver's 2 environment. We have not investigated this path in this work, but we believe this direction has a reasonable potential as a future work. Note that different solvers expose different configurations for their internal algorithms. For example, while SCIP exposes over 2500 parameters[5], CPLEX exposes 182 parameters[6] and GUROBI exposes 100 parameters[7]. Due to the different algorithm implementations, only a small subset of parameters have an exact match across all solvers.

SCIP has been used in this work for a number of reasons: (1) it is a stable open-source solver and its algorithms are comprehensively documented, while commercial tools hide their implementation details (2) it exposes a large number of configuration parameters to tune, and (3) it has been used in previous related works (e.g. (Gasse et al., 2019; Kruber et al., 2017; Prouvost et al., 2020; Valentin et al., 2022).

[5] https://www.scipopt.org/doc/html/PARAMETERS.php

[6] https://www.ibm.com/docs/en/icos/12.8.0.0?topic=cplex-list-parameters

[7] https://www.gurobi.com/documentation/9.0/refman/parameters.html

E DATA MANAGEMENT

In order to offer a seamless integration of MILPTune in existing environments, a data store is required to save the results from the offline configuration space search. In this work, we use MongoDB [8] for that purpose. For each dataset, we create a collection that contains records for each problem instance in that dataset. Listing 1 shows the schema used for each instance. It keeps track of configurations explored for that instance along with their costs. In addition, it records the embedding vector of the instance in order to be searched later with the nearest neighbor algorithm.

The parameters presented in the listing are the ones that were used for the configuration space exploration using SMAC Lindauer et al. (2022). A detailed description of the definition of these parameters can be found in their official documentation [9]. As discussed in Section 4, the metric learning approach does not limit the number of configuration parameters explored offline. It also does not limit which parameters are explored since it focuses on learning an embedding space where similarity between instances can be quantified reliably. Thus, it is possible to learn a model for similarity once and keep expanding the offline configuration space search without requiring to re-train the model. In other words, the more comprehensive the offline configuration space exploration, the better, and closer correlated, the predictions of MILPTune would be.

```
instance_record = {
    "configs": [
        {
            "seed": 0,
            "cost": 0,
            "time": 0,
            "params": {
                "branching/scorefunc": "s",
                "branching/scorefac": 0.167,
                "branching/preferbinary": False,
                "branching/clamp": 0.2,
                "branching/midpull": 0.75,
                "branching/midpullreldomtrig": 0.5,
                "branching/lpgainnormalize": "s",
                "lp/pricing": "l",
                "lp/colagelimit": 10,
                "lp/rowagelimit": 10,
                "nodeselection/childsel": "h",
                "separating/minortho": 0.9,
                "separating/minorthoroot": 0.9,
                "separating/maxcuts": 100,
                "separating/maxcutsroot": 2000,
                "separating/cutagelimit": 80,
                "separating/poolfreq": 10
            }
        },
        :
        :   # all configurations explored offline
    ],
    "bipartite": {
        "vars_features": [...],
        "cons_features": [...],
        "edge_features": [...]
    },
    "embedding": [...]
}
```

Listing 1: Problem Instance Record

[8] Link: https://www.mongodb.com/

[9] Link: https://www.scipopt.org/doc/html/PARAMETERS.php