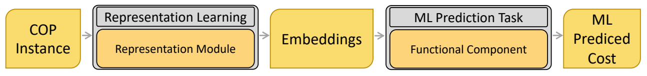
Graph Representation Learning with Graph Transformers in Neural Combinatorial Optimization

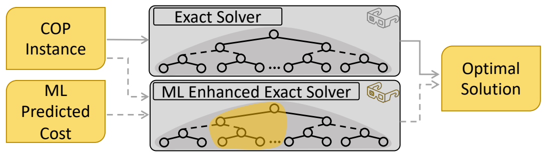
Tianze Wang, Amir H. Payberah, and Vladimir Vlassov
Department of Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden
Emial: {tianzew, payberah, vladv}@kth.se

***Abstract*—Neural combinatorial optimization aims to use neural networks to speed up the solving process of combinatorial optimization problems, i.e., finding the optimal solution of a problem instance from a finite set of feasible solutions that minimize a given objective function. Recently, researchers have applied convolutional neural networks to predict the optimal solution's cost (defined by the objective function) to give as extra input to an exact solver to speed up the solving process. In this paper, we investigate whether graph representations that explicitly model the inherent constraints in combinatorial optimization problems would improve the performance of predicting the optimal solution's cost. Specifically, we use graph neural networks with neighborhood aggregation and graph Transformer models to capture and embed the knowledge in the graph representations of combinatorial optimization problems. We also propose a benchmark dataset containing the Traveling Salesman Problem (TSP) and Job-Shop Scheduling Problem (JSSP), and through the empirical evaluation, we show that graph Transformer models achieve an average loss decrease of 61.05% on TSP and 66.53% on JSSP compared to the baseline convolutional neural networks.**

***Index Terms*—Graph Representation Learning, Combinatorial Optimization, Graph Transformer, Job-Shop Scheduling Problem, Traveling Salesman Problem**



(a) ML prediction pipeline takes as input a COP instance, e.g., matrix or graph representation, and uses a two-step pipeline to predict the cost of the optimal solution of the input COP instance.



(b) Using ML prediction. Top: an exact solver finds the optimal solution by searching the dark grey solution space. Bottom: an ML-enhanced solver prioritizes the search in the yellow-highlighted area predicted by ML while maintaining optimality (same search space).

Fig. 1: ML prediction pipeline and how to use the prediction in an ML enhanced exact solver.

I. Introduction

Combinatorial optimization [1], [2] aims to find the optimal configurations from a finite set of objects minimizing a cost metric. It is widely used in many industries, such as transportation, supply chain, and scheduling [3]–[5]. However, due to its NP-hard or NP-complete nature [6], [7], *exact solvers* that enumerate all feasible solutions in the solution space are often impractical for larger instances due to the vast number of configurations [8]. Traditional research in combinatorial optimization focuses on approximation methods [9] and heuristic/meta-heuristics [10] to find solutions close to the optimal within a reasonable computational budget.

Developing an improved and efficient solver for a Combinatorial Optimization Problem (COP) requires expert knowledge, domain experience [11], and significant trial and error [12]. Researchers recently explored using Neural Networks (NNs) to accelerate finding solutions for COP instances [4], [11]. For example, deep reinforcement learning (RL) that combines NN with RL allows for approximating COP solutions [8], [12]–[16], whereas combining Convolutional Neural Networks (CNN) with exact COP solvers allows for accelerating finding the optimal COP solutions [17]. An exact COP solver can be enhanced with a Machine Learning (ML)-based predictor to reduce the search space of the solver looking for the exact solution. For example, one possible implementation of such an end-to-end neural optimizer is a pipeline where an ML predictor provides an exact solver extra constraints prioritizing searching for a solution in the area of the predicted optimal cost [17]. The more accurate the optimal solution cost prediction is, the faster the exact solver finds the optimal solution. In this work, we focus on improving the accuracy of predicting the optimal solution. Figure 1 shows the overview of our prediction tasks. We investigate whether graph representations that explicitly model the inherent constraints in COPs would improve the accuracy of the optimal solution cost prediction.

To use a NN on COP, we need to represent a COP instance in a required input format of the chosen NN, e.g., as a matrix [17] or as a graph [4]. This paper focuses on the graph-based approaches and explores Graph Neural Network (GNN) to represent COP instances. Moreover, we apply graph Transformer models in COP and investigate their performance, as many recent studies on graph Transformer models show further advantages over GNN models [18]–[21]. In this paper, we work on two well-known and representative COPs: Traveling Salesman Problem (TSP) and Job-Shop Scheduling Problem (JSSP). We evaluate CNN, GNN, and graph Transformer models on TSP and JSSP. Our empirical evaluation suggests

The authors would like to acknowledge funding from Vinnova for the Digital Cellulose Competence Center (DCC), Diary number 2016-05193.

that graph representations in COP have better downstream regression task loss value than the matrix representations used in CNN models, and graph Transformer models further bring down the loss value compared to the baseline GNNs. Through empirical evaluation, we show that graph Transformer models achieve an average loss decrease of 61.05% on TSP and 66.53% on JSSP compared to the baseline CNN.

Specifically, our contributions are as follows:

1) Demonstrate the advantage of graph representations in predicting the cost of optimal solutions for COP instances, e.g., predicting the optimal tour length in TSP and optimal makespan in JSSP. We show that compared to our previous work [17] using CNNs on matrix representations, GNNs with graph representations have better performance in the prediction task.
2) Explore graph Transformer models in COP and demonstrate their advantages in improving the quality of predictions. We empirically show that graph Transformer models have better performance on COP cost prediction over classical GNN and CNN models.
3) Propose and open source four COP benchmark datasets with TSP and JSSP instances. The proposed benchmark datasets can be used to evaluate and compare the performance of ML models in future studies in the field of using Machine Learning in Combinatorial Optimization.

II. Task Formulations and Preliminaries

In this section, we formulate a Machine Learning (ML) task in a COP and introduce the preliminaries of our work.

A. Task Formulation

We define the task of predicting optimal COP costs, explain how to use the ML prediction with an ML-enhanced solver, and present ways to represent COP instances for ML models.

1) ML Prediction Task: Assuming a lower cost means a better solution, our goal in solving a COP instance is to find feasible solutions with the lowest cost, i.e., the *optimal solution*. However, it is challenging as most COPs are NP-hard or NP-complete, meaning a brute-force enumeration of all potential solutions soon becomes intractable due to combinatorial explosion as the problem size increases. In real-world settings, people often use approximation and heuristic approaches to find solutions that are usually very close to the optimal solutions on COP instances of reasonable size. In this work, we approach the challenging task of directly predicting the cost of the optimal solution.

We formulate a regression task on COP (see Figure 1a), where an ML model $f : X \to \mathbb{R}$ takes a COP instance X as input and predicts the cost of the optimal solution of X (see Section II-B). The ML model is trained in a supervised fashion on the training dataset of COP instances where the cost of the optimal solutions is known (see Section IV-A1). The intuition is that the ML model would learn from the training dataset the underlying data distribution of COP instances and would be able to generalize its prediction with a decent on unseen test COP instances drawn from the same underlying data distribution as those from the training dataset.

2) Integrading ML prediction into COP Solvers: Once the ML model achieves a good generalization performance through training, we can use it to predict unseen COP instances during inferencing. One can develop an ML-enhanced COP exact solver that prioritizes its search for the optimal solutions in the search space where the cost is close to the predicted cost (see Figure 1b). The result of the ML-enhanced exact solver would still be correct since it does not abandon any of the regions in the original search space. Instead, the enhanced exact solver could find the optimal solution faster than the original exact solver by focusing on the region highlighted in the search space and pruning the search space much faster than the original exact solver without such prior. Integrating the prediction into an exact solver applies to many types of COPs. For example, in a JSSP problem, we can design a search schema in the exact solver to prioritize the region of the predicted cost of the optimal solution. In a TSP, the exact solver can focus on the search space where the optimal tour length is close to the prediction values.

3) Representing COP for ML Models: In our previous work [17], we demonstrate the feasibility of using CNN models for predicting the cost of the optimal solution of COP and how the predictions can be used in an ML-enhanced exact solver to speed up the process of finding the optimal solution. While it is intuitive to represent a COP problem as a matrix, there are some potential challenges. First, the matrix input to a CNN model is usually of a fixed, predetermined size that limits the input data size. Second, the matrix representation might make it hard for CNN models to pick up related elements far from each other in the input matrix. In this work, we focus on improving the performance of the ML model in predicting the cost of the optimal makespan. Specifically, we investigate how the graph representations, GNN, and graph Transformers can further improve the performance of the mentioned earlier regression tasks compared to the CNN model used in [17].

B. Combinatorial Optimization Problems

In this subsection, we introduce COP and provide definitions of the two COPs we study in this paper, i.e., JSSP and TSP.

1) General Definition: COP [1], [2] can be formulated as a constraint optimization problem with a set of elements (variables) and a set of constraints. For each COP instance i, a set of elements represents variables, and a set of constraints represents natural or imposed restrictions on the elements within the problem [11], e.g., the same resource can not be used by different elements at the same time or one operation must be executed before another operation.

Let S be the set of all feasible solutions of i where all the constraints are satisfied. For each feasible solution $s \in S$, an objective function $f_c : S \to \mathbb{R}$ is a mapping from a feasible solution to a scalar value representing the cost of the solution s. Most often, our goal in solving a COP instance is to find a feasible solution with the lowest cost. In this work, we focus on exact COP solvers where the solver finds and proves the optimality of a solution, i.e., a solution with the lowest cost.

2) Traveling Salesman Problem (TSP): TSP [22] is one of the canonical examples of COP. We are interested in finding the shortest path for a salesman to traverse a set of n cities, starting from one city, visiting all the other cities once and exactly once, and returning to the start city. $\mathbf{X}_{tsp} \in \mathbb{R}^{n\times n}$ represent the pair-wise distance between two cities where the value x_{ij} on the i^{th} row and j^{th} column represents the cost of traveling from city i to city j. Figure 2a shows the matrix representation of a TSP instance with seven cities.

Our goal in solving a TSP instance is to find the shortest tour to traverse all the cities. The cost of a solution to a TSP instance is the length of the tour, i.e., the summation of the cost to travel between cities to complete the tour. The optimal tour length is the cost of the optimal tour.

3) Job-Shop Scheduling Problem (JSSP): JSSP [23] is one of the hardest COPs with many variants known as NP-hard or NP-complete. This work focuses on the following version of JSSP. A JSSP instance contains a set of m machines $M = \{M_1, M_2, \cdots, M_m\}$ and n jobs $J = \{J_1, J_2, \cdots, J_n\}$ where each job J_i has m operations $O_i = \{O_{i_1}, O_{i_2}, \cdots, O_{i_m}\}$. Operations in O_i needs to be excuted sequentially in the order from O_{i_1} to O_{i_m} and each operation O_{i_k} would require machine M_{i_k} for d_{i_k} units of time.

A JSSP instance can be represented as a 2D matrix $\mathbf{X}_{jssp} \in \mathbb{R}^{n\times 2m}$ where the i^{th} row has $2m$ elements representing the requirements of $J_i = \{M_{i_1}, d_{i_1}, M_{i_2}, d_{i_2}, \cdots M_{i_m}, d_{i_m}, \}$. A JSSP instance can also be represented as a graph $\mathbf{G}_{jssp}(V, E)$, where V represents the set of all operations and E represents the two sets of constraints: (1) precedence constraints: the i^{th} operation in each job needs to be executed before $\{i+1\}^{th}$ operation; and (2) machine constraints: the set of operations that require the same machine cannot be executed simultaneously.

Our goal in solving a JSSP instance is to find the optimal solution, i.e., a feasible solution (schedule) with the lowest cost. The cost of a solution in JSSP could be the *makespan* of a solution, i.e., the total elapsed time if a JSSP instance is to be executed according to the solution. The optimal makespan is the makespan of the optimal solution. The left of Figure 2b shows the matrix representation of a JSSP instance with three jobs with three operations requiring three machines.

C. Graph Representation Learning

This section discusses the ML task around COP, focusing on graph representation learning and graph Transformer models.

1) Graph Neural Networks: Let $G(V, E)$ be a graph where $v_i \in V$ is a node and $e_{ij} = (v_i, v_j) \in E$ is an edge from node v_i to v_j. Graph Representation Learning (GRL) aims to create vectorized representations, i.e., embeddings, to represent nodes, edges, or the entire graph. For example, we may want to create a $d \in N$ dimensional embedding $\mathbf{x}_i \in \mathbb{R}^d$ to represent a node v_i in the graph where the structural and semantic information of the node is well preserved. A single embedding that summarizes the entire graph can later be derived by aggregating all the nodes' embeddings.

State-of-the-art Graph Representation Learning (GRL) methods use GNN [24] to create an embedding for a target node by collecting and aggregating its neighbor embeddings. GCN [25], GIN [26], GAT [27], and GraphSAGE [28] are well-known networks of the GNN models in GRL. Without losing generality, a GNN model can be represented as follows.

$$X_G^{(l+1)}, E_G^{(l+1)} = GNN_e^{(l)}(X_G^{(l)}, E_G^{(l)}, A) \qquad (1)$$

where $A \in \mathbb{R}^{N\times N}$ is the adjacency matrix of a graph $G(V, E)$ with N nodes. $X_G^{(l)}$ and $E_G^{(l)}$ are the input node and edge embeddings, correspondingly, at the l^{th} layer of GNN and $X_G^{(l+1)}$ and $E_G^{(l+1)}$ represent the output node and edge feature of the l^{th} layer of GNN. Note that a GNN does not need to have both node and edge representations. The GNN layer in Eq. 1 can be stacked with each other to form a multilayer GNN model that creates fixed-size vectorized node, edge, or graph representations to be used in downstream applications. We refer readers to the work of Hamilton [24] for a detailed introduction to GRL and GNNs.

2) Graph Transformer Models: Transformer [29] models have seen great success in many fields of ML, e.g., natural language processing [30] and computer vision [31], [32]. Recently, lots of work has been emerging in adapting Transformer models in GRL and has demonstrated great performance [18]–[21]. It can potentially solve known problems in GNN, e.g., over-smoothing [33] and over-squashing [34].

The key to successfully applying Transformer models on graphs, in many research papers [18], is to encode node positional and structural information. For example, Graphormer [21] uses centrality, spatial, and edge encodings to add to a graph and achieves significant performance at the time on Open Graph Benchmark Large-Scale Challenge (OGB-LSC) [35]. More recently, Rampášek et al. [19] propose GraphGPS as a way to build a general, powerful, and scalable graph Transformer that achieves SOTA performance on many benchmarks. GraphGPS categorizes positional (position of a node in a graph) and structural encodings (structure of the graph) into three categories, i.e., local, global, and relative, and explores a modular system to add them as soft bias in the Transformer model. The key contributing factor to GraphGPS is the GPS layer can be described as follows:

$$\begin{aligned} X_G^{(l+1)}, E_G^{(l+1)} &= GPS^{(l)}(X_G^{(l)}, E_G^{(l)}, A) \\ &= MLP^{(l)}(GNN_e^{l}(X_G^{(l)}, E_G^{(l)}, A) \\ &\quad + Transformer^{(l)}(X_G^{(l)})) \end{aligned} \qquad (2)$$

where GNN_e^l, $Transformer$, and MLP represent GNN, Transformer, and multilayer perceptron layer, and the rest of the notations follow the same convention as in Eq. 1. The GPS layer presented in Eq. 2 can be viewed as a hybrid layer aggregating the output of GNN and Transformer. In this way, the GPS layer can: (1) utilize GNN knowledge while alleviating known GNN limitations like over-smoothing, over-squashing, and limited expressiveness by WL test, (2) explore the self-attention mechanism in Transformer. In this study, we use GraphGPS as a SOTA graph Transformer example and compare it to GNN models in GRL for COP tasks.

III. Optimal Cost Prediction with Graph-based ML

This section introduces Graph Representation Learning for Combinatorial Optimization Problems (COPs), focusing on GNN and graph Transformer models.

A. Graph Representation of a COP

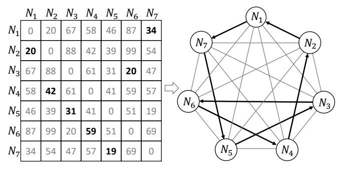
Here, we describe graph representations of the two representative Combinatorial Optimization Problems that we work with in this paper, namely, the Traveling Salesman Problem (TSP) and Job-Shop Scheduling Problem (JSSP). A straightforward way to represent a TSP instance is with a square matrix $\mathbf{X}_{tsp}$, e.g., Figure 2a on the left, with $x_{i,j}$ represent the cost of traveling from city i to city j. The graph representation of a TSP instance is also straightforward, with nodes representing cities and edge features representing the cost between the two cities. Figure 2a on the right shows an example of a graph representation of TSP, and one of the optimal tours is highlighted with directed edges in boldface.

Figure 2b shows the matrix and graph representation of a JSSP instance with 3 machines and 3 jobs (J_1 to J_3). Each job has 3 sequential operations, each of which needs to be executed on machine m_i for d_i units of time ($i \in \{1, 2, 3\}$). We construct a JSSP graph with nodes representing operations and edges representing the constraints. We use directed edges to encode precedence constraints between operations and undirected edges to represent machine constraints. If a set of operations needs the same machine to execute, then there is an undirected edge between each pair of the nodes in the set. We encode the execution time of operations, i.e., d_i, as node features and add start and end operations s and e to connect the graph. There are directed edges from the start operation s to the first operation in each job and from the end operation e in each job to the end operation in each job.

Compared to matrix representations of COP, a graph can represent COP instances of different sizes and easily encode prior knowledge of nodes in the graph. For example, the two operations in a JSSP instance require the same instances can be connected with an edge that represents their conflict with each other. An edge in a TSP instance graph can represent the distance between the two adjacent cities. Graph representations of COP instances have advantages over the matrix representation counterpart as they can directly model the constraints and knowledge needed for the prediction tasks.

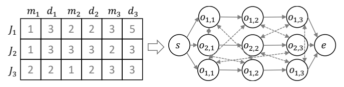
B. GNN and Graph Transformer

In this subsection, we discuss the benefits of using GNN and graph Transformer models for learning relevant features in the COP prediction task. GNN can deal with graphs of varying sizes. This can be of interest in the domain of COP, where acquiring ground truth about larger instances can grow exponentially more expensive as the sizes of COP instances grow. Furthermore, the neighborhood aggregation of GNN models over edges simplifies encoding the inherent constraints in COP instances. Different types of COP problems might require the ML model to acquire different knowledge to make good predictions on the cost of the optimal value. For example, given a TSP instance, we would like an ML model to identify the shortest path to traverse the graph. Once the ML model learns the potential shortest Hamiltonian cycle in a given TSP, then the cost prediction can summarize the tour length.



	N_1	N_2	N_3	N_4	N_5	N_6	N_7
N_1	0	20	67	58	46	87	**34**
N_2	**20**	0	88	42	39	99	54
N_3	67	88	0	61	31	**20**	47
N_4	58	**42**	61	0	41	59	57
N_5	46	39	**31**	41	0	51	19
N_6	87	99	20	**59**	51	0	69
N_7	34	54	47	57	**19**	69	0

(a) A TSP instance with seven cities (N_1 to N_7). Left: TSP instance matrix showing travel costs between cities. The optimal tour length is the sum of boldface values. Right: TSP instance graph, where edge features represent travel costs (not shown in visualization). One of the optimal tours is highlighted with boldface directed edges.



	m_1	d_1	m_2	d_2	m_3	d_3
J_1	1	3	2	2	3	5
J_2	1	3	3	3	2	3
J_3	2	2	1	2	3	3

(b) A JSSP instance with three machines and three jobs (J_1 to J_3). Every job (J_1 to J_3) consists of three sequential operations, each requiring a machine m_i for d_i units of time ($i \in 1, 2, 3$). Left: JSSP instance matrix. Right: JSSP instance graph with nodes representing operations, directed edges showing sequential execution order, and undirected edges indicating operations that require the same machine. We add a start operation s and an end operation to connect the graph.

Fig. 2: Benefits of graph representation for COP.

In a JSSP instance, one way to predict the optimal makespan is to learn the critical path, i.e., the longest path in a potential schedule. The prediction of the optimal makespan can be calculated as the amount of time accumulated on the critical path. Note that the optimal makespan, instead of a simple summation of operation time along the critical path, when the last operation finishes its execution, as the operations on the critical path might be blocked due to machine unavailability. While the prediction of optimal cost in different COP might require different knowledge, this knowledge is usually encoded in the graph representation of the COP.

Apart from the success of graph Transformer models in recent years, we are motivated by the finding that GNNs with attention mechanisms tend to outperform other types of GNNs. This could indicate that attention mechanisms are an excellent addition to GNNs for extracting knowledge needed to create representations of COP problems. For example, the attention mechanism could pick up signals on which nodes are on the critical path in a JSSP instance or whether an edge between two nodes is in the optimal tour of a TSP instance.

The attention mechanism also allows each node to learn other nodes' importance and update its representation accordingly. Furthermore, the attention mechanism could also be used to discover and synthesize knowledge in the COP graph that domain experts cannot easily discover and describe in human or computer language. As we shall see in the empirical evaluation section, graph transformer models perform better than even GNNs with attention mechanisms.

C. ML Prediction Pipeline

Figure 1a shows the proposed two-step ML pipeline for predicting the cost of the optimal solution of COP. In the first step, a GRL module uses a COP instance as input and produces embeddings that summarize the input COP instance. In the second step, a functional component, e.g., NN, takes the generated embeddings from the previous step and predicts the target, i.e., the cost of the optimal solution.

We can use either CNN, GNN, or graph Transformer models in the representation learning step. For CNN models, a CNN takes the matrix representation of COP instances as input and produces as output a single embedding for each input instance. For GNN models, a GNN takes graph representation of COP instance as input and uses multiple GNN layers (Eq. 1) with neighborhood aggregation to produce a fixed-size embedding for each node in the input graph. Then, a graph pooling method, e.g., summation, takes all the node embedding as input and generates a single graph embedding for each of the input graphs. For graph Transformer models, the representation learning module is similar to that of GNN, except that it uses Eq. 2 instead of Eq. 1.

In the ML prediction step, we use functional components, e.g., a Multilayer Perceptron (MLP), to predict the cost of the optimal solution. The functional component takes a single embedding generated for each COP instance as input and directly predicts the target values, i.e., optimal tour length in TSP and optimal makespan in JSSP. The next session reveals more details about the end-to-end training of the ML pipeline.

IV. Empirical Evaluation

In this section, first, we present the experiment setup, including a general COP benchmark for our empirical evaluation, the baseline models, and evaluation procedures; next, we present and discuss the results of our evaluation experiments.

A. Experiment Setup

1) Towards a General COP Benchmark: To our best knowledge, existing benchmarks[1][2], primarily created for exact and approximate, and RL-based solvers, do not have enough COP samples with known optimal costs for training ML models. Therefore, we create four COP benchmark datasets to compare CNN, GNN, and graph Transformer models empirically. `JSSP-9×9` dataset contains $10,000$ JSSP instances, each with nine jobs. Each job has nine operations, requiring one of the nine machines to run a certain amount of time. `JSSP-10×10` dataset is generated similarly as `JSSP-9×9` except that there are $100,000$ JSSP instances, each with ten jobs, ten machines, and ten operations. In both `JSSP-9×9` and `JSSP-10×10`, operation duration is an integer sampled uniformly from 1 to 99 (boundary included). `TSP-30` and `TSP-40` dataset each contains $10,000$ TSP instances with 30 and 40 cities. The distance between two cities is an integer sampled uniformly between 1 and 99 (inclusive boundaries). The distance from a city to itself is denoted as zero, representing infinity. We run a constraint programming solver written with GECODE[3] to acquire the ground truth of the cost of the optimal solution, i.e., optimal tour length for `TSP-30` and `TSP-40` and optimal makespan for `JSSP-9×9` and `JSSP-10×10`. The datasets are accessible in our code repository and could support future ML research for COP.

2) Training and Evaluation Procedures: In the supervised training setting, we split each dataset into the train (50%), validation (30%), and test (20%) datasets. For each combination of ML model and dataset, we run the experiment ten times with different random seeds and report the average test Mean Square Error (MSE) with standard deviation when the validation MSE is the reach minimal across 3000 epoch of training with AdamW optimizer [36]. The details of datasets, code, and hyperparameters are available [4].

3) Baseline Models: For the CNN model, we use the CNN model proposed in [17] initially designed for directly predicting the makespan (cost of optimal solution) for JSSP. The CNN model contains CNN and MLP layers, and we slightly tweak the parameters of the models to adapt to the different input sizes of COP instances. For the GNN model, we composite a suite of classical and SOTA GNN models, e.g., GCN [25], GIN [26], GINE [37], GraphSAGE [28], GAT [27], and GATv2 [38]. We use GraphGPS [19] as our graph Transformer model and use Laplacian positional encoding (LapPE) [19] and random-walk structural encoding (RWSE) [19] to encode positional and structural information into GraphGPS. In Table I and II, we use the following naming conventions to denote GraphGPS models with different combinations of GNNs and encodings. GraphGPS-GCN represents a GraphGPS model with GCN model as GNN in Eq. 2 without any positional and structural encodings. GraphGPS-GINE+LapPE+RWSE represents a GraphGPS model with GINE as its GNN part, and both LapPE and RWSE are added to the inputs of the GraphGPS model. We treat the number of attention heads in GAT, GATv2, and GraphGPS as a hyperparameter and use a default value of four in our experiments unless otherwise stated.

We use a combination of PyTorch[5], PyTroch Geometric[6], and the original implementation of GraphGPS [19] for our experiment. The experiments are conducted in a remote GPU cluster with NVIDIA Tesla T4 or at a local benchmark machine with NVIDIA GeForce RTX 2070 Super.

B. Experiment Results

Table I shows the results on the `TSP-30` and `TSP-40` datasets. GCN, GAT, and GATv2 models do not outperform the baseline CNN models. This could be explained by the fact that the graph structure of the TSP variant in this work is a fully connected graph with edge features indicating the cost of traveling from one node (city) to another node (city). In

[1] https://www.math.uwaterloo.ca/tsp/concorde/benchmarks/bench99.html
[2] https://github.com/tamy0612/JSPLIB

[3] https://www.gecode.org/
[4] Our experiment code is available at https://github.com/bwhub/GRLCOP
[5] https://pytorch.org/
[6] https://pytorch-geometric.readthedocs.io/

this case, the GCN model cannot differentiate between TSP instances of the same size, as the graph structure of those instances will be the same with the same node features. The useful features for predicting the optimal cost are encoded as edge features that are oblivious to the GCN model. GINE is the best performer among the GNN models as it considers the edge explicitly features when building graph representations. GraphGPS models with a GNN model that can learn from edge features, e.g., GINE, can outperform that baseline CNN model and achieve performance comparable to GINE.

Table II shows the experimental results on the `JSSP-9×9` dataset. The simplest GNN model outperforms the CNN model by a large margin, with an average test MSE of 1215.01 of GCN compared to the 2979.13 that of CNN. This indicates that the graph representations are beneficial for predicting the optimal cost of the COP model. Furthermore, more expressive GNN models, e.g., GIN (1184.26) and GraphSAGE (1095.78), also lead to better performance than GCN. We also see this in the comparison of GNN with attention mechanism, where GATv2 [38] (1110.29), a more expressive dynamic attention model compared to GAT [27] (1174.58), has a better performance than the GAT model.

While GNN models perform better than CNN models, graph Transformer models perform even better. For example, GraphGPS with both LapPE and RWSE encodings achieves the best performance on the `JSSP-9×9` dataset with an average test MSE of 1041.93. We tested other variants of the GraphGPS models with different combinations of encodings. As shown in Table II, while all the variants achieve good performance, overall, we get the best performance boost when RWSE encoding is included. One explanation could be that structural information provided critical information to create good representations of the model.

Table II also shows the experiments on the `JSSP-10×10` dataset. The results are similar to the results for the `JSSP-9×9` dataset: GNN models outperform the CNN model due to the graph representation that explicitly models the complex relations and constraints in COP instances. Notably, GATv2 outperforms all the other GNN models, and overall, GraphGPS models are the best performers. Even the vanilla version of GraphGPS, without positional and structural encodings, can outperform all GNN models thanks to the more expressive power enabled by the attention mechanism. Note that for the GraphGPS model reported in Table II, we use the GCN model as the GNN model in Eq. 2. The performance of GraphGPS models can be further improved by replacing the GCN model in GraphGPS with GNN models with more expressive power, e.g., GAT, GATv2, and GINE.

Figure 3 is the violin plot that shows the performance difference of GAT (Figure 3a), GATv2(Figure 3b), and GPS-GCN without any positional or structural encodings ((Figure 3c)) with different numbers of attention heads. GAT and GATv2 with only one attention head have worse average MSE than GCN in Table II and exhibit large variance. The average performance of both GAT and GATv2 improves with an increasing number of attention heads. The performance

TABLE I: Results on TSP dataset. Both GNN and GraphGPS that consider the edge feature outperform the CNN baseline.

Model	Dataset	
	`TSP-30`	`TSP-40`
	Test MSE (mean±std)	
CNN	473.85±16.21	356.24±16.29
GCN	521.10±0.00	389.70±0.00
GAT	521.10±0.00	389.70±0.00
GATv2	521.10±0.00	389.70±0.00
GINE	**178.98±10.35**	**144.63±12.02**
GraphGPS-GINE	185.51±8.03	146.89±8.12
GraphGPS-GINE+LapPE	181.29±3.53	**142.51±4.64**
GraphGPS-GINE+RWSE	189.45±7.38	148.69±7.51
GraphGPS-GINE+LapPE+RWSE	**179.41±3.99**	166.52±73.72

TABLE II: Results on JSSP dataset. GNN outperforms CNN, and GraphGPS further improves performance.

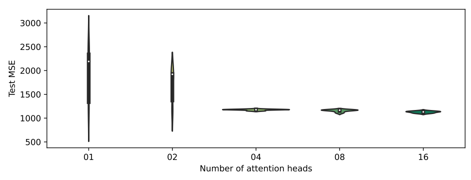
Model	Dataset	
	`JSSP-9×9`	`JSSP-10×10`
	Test MSE (mean±std)	
CNN	2979.13±108.57	2944.77±25.42
GCN	1215.01±12.72	1193.82±10.85
GIN	1184.26±25.38	1188.67±14.33
GraphSAGE	**1095.78±30.89**	1121.95±30.21
GAT	1174.58±12.69	1196.87±31.26
GATv2	1110.29±169.36	**1073.38±46.71**
GraphGPS-GCN	1195.77±33.78	969.95±4.87
GraphGPS-GCN+LapPE	1140.91±50.73	970.23±6.91
GraphGPS-GCN+RWSE	1051.21±42.13	943.99±25.51
GraphGPS-GCN+LapPE+RWSE	**1041.93±25.11**	**939.70±23.85**

difference concerning the number of attention heads could be that with only one attention head, the model might focus on the sub-optimal sub-features or areas in the embeddings space for calculating the attention score, which leads the GNN model to be potentially biased on only a sub-optimal subset of the features. With the added attention heads, the problem is significantly eased as the model calculates different attention scores that focus on different aspects of the COP instance. This can also be partially supported by the result that GAT and GATv2 with only one attention head exhibit a much more significant variance on test MSE over the ten repeated runs than their four attention heads counterparts. In the GraphGPS model, without any positional or structural encodings, we see that the performance is more stable than GAT and GATv2. This indicates that GraphGPS provides a more stable attention mechanism than the ones used in GAT and GATv2.

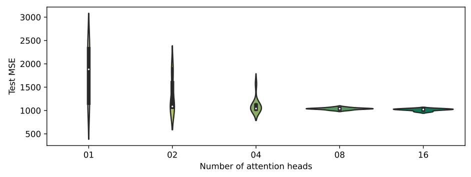
C. Discussion

Overall, GNN outperforms the CNN baseline, and the graph Transformer models can improve the performance further. This is consistent with the findings not only in graph Transformers but also in other domains, e.g., computer vision [31], [32] and natural language processing [30], where self-attention mechanism plays a critical role.

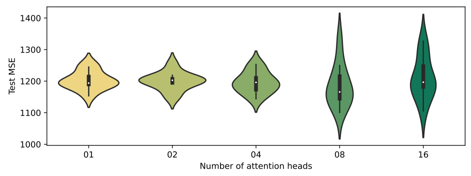
The modular design of GraphGPS means that each part in Eq. 2 can be easily switched to more powerful and expressive models to improve the performance or switched to more efficient models to achieve a desired balance of computation cost and performance. This design choice can be crucial in



(a) Violin plot on the performance of GAT.



(b) Violin plot on the performance of GATv2.



(c) Violin plot on the performance of GraphGPS-GCN.

Fig. 3: Effects of the number of attention heads on model performance. The performance of GAT and GATv2 improves with an increased number of attention heads. GraphGPS-GCN achieves good performance with only a single attention head and is less sensitive to the number of attention heads.

application domains like combinatorial optimization, where successfully applying ML for decision-making in an exact solver depends on the absolute predictive performance and the relative inference cost for an ML model to generate a prediction. For example, if an exact solver needs to constantly query an ML model to make decisions, the overall inference cost for generating all the predictions can quickly build up through the solving process. In this case, the solver might have a slow solving process due to the ML inference cost, even though it makes good decisions each time it is called. On the other hand, if an exact solver only needs a single prediction on the cost of the optimal solution, it is less sensitive to the computation cost of the ML model. In this case, it might even favor an ML model with a relatively high computation cost but offers high-quality predictions, as better predictions might lead the solver to prune its search space much faster, thus fastening the speed for finding the optimal solution.

Adapting GRL to our version of the TSP problem is challenging due to the highly similar, if not identical, graph structures of different TSP instances with various node-to-node costs. This poses challenges as the critical information (cost of traveling between two nodes) is not easily learnable for a wide variety of GNN models that cannot be easily adapted to learn from edge features. Graph Transformer models, on the other hand, can utilize their attention mechanism to learn useful features. We expect graph Transformer models, explicitly designed to include edge features, could outperform the GraphGPS model used in our experiment.

V. Related Work

Combinatorial Optimization Problems (COPs) are challenging to solve due to their discrete and non-convex nature [39]. Applying ML in COP is not a new topic [40], [41], and in recent years, we have seen a surge in using ML, especially GNN [4] and Reinforcement Learning [13] for COP problems.

Many papers focus on using a Neural Network (NN) to directly construct a solution to a COP instance. Bello et al. [12] propose a Neural Combinatorial Optimization framework that explores policy gradient method to train a Recurrent Neural Network, i.e., pointer network [42] to iteratively generate a solution to the given COP instance. However, this method does not fully utilize the graph structure, and the policy gradient method is not very sample-efficient [8]. Dai et al. [8] propose to automate the process of designing approximation and heuristic methods using a meta-algorithm that explores Q-learning to incrementally construct a solution with a combination of graph embeddings and RL. Nazari et al. [14] explore neural networks and RL to learn to generate a sequence of decisions for the Vehicle Routing Problem. Kool et al. [43] using an encoder and decoder network with attention layers and getting solutions that are close to optimal.

Research papers also use NN to learn heuristics. Chen and Tian [44] propose NeuRewriter that uses a neural network to learn to pick heuristics and rewrite part of current solutions to improve the current solution iteratively. Gasse et al. [45] propose to enhance exact methods with graph convolutional neural network model to learn branch-and-bound variable selection policies. The authors train the model with imitation learning to learn directly from an expert branching rule that is otherwise computationally too expensive for an exact solver. Zhang et al. [15] use GNN and RL to learn priority dispatching rules that are otherwise costly and time-consuming to design, even for experts with domain knowledge. Cappart et al. [16] propose using RL to learn branching strategies to guide a Constraint Programming solving process.

Our work differs from the existing work in that we train an ML model to directly predict the cost of the optimal solution of a COP instance. In one view, the prediction performance can be viewed as the ability of the underlying model to encode the given COP instance and extract information that is usefully predicting the cost of the optimal solution. In another view, our study is an orthogonal direction to existing work, which can be easily combined with existing work for potential further improvements, e.g., switching the GRL module with graph Transformer models in existing work for learning to construct approximate solutions or learning heuristics.

VI. Conclusion and Future Work

In this work, we study graph representations in Combinatorial Optimization Problems (COPs) on the task of predicting the cost of the optimal solution of a COP instance by considering two representative COPs: traveling salesman and job-shop scheduling. Our empirical evaluation shows that graph representations and Graph Neural Networks (GNNs) have a better prediction performance than matrix representations and

CNNs. Furthermore, we show that graph Transformer models can further improve the performance of the prediction task compared to GNNs. Implementing end-to-end neural optimizers that combine ML-based optimal solution cost preditors with exact COP solvers is a subject of our future work. A natural extension of our current work is the unsupervised or semi-supervised learning settings of GNNs and graph Transformers in COP to alleviate the need for getting ground truth labels to train the prediction task. Another exciting direction is to study the performance of graph Transformer models in settings where graph representation learning and reinforcement learning are combined to get approximate solutions to COP instances. Furthermore, extending the COP benchmark proposed in this work with larger datasets and more types of COP instances would benefit future research in this area.

Acknowledgment

The computations were partially enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at the PDC Center for High Performance Computing, KTH Royal Institute of Technology, and Chalmers Centre for Computational Science and Engineering (C3SE), partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

References

[1] B. Korte et al., *Combinatorial optimization*. Springer, 2011, vol. 1.

[2] L. Wolsey and G. Nemhauser, *Integer and combinatorial optimization*. John Wiley & Sons, 1999, vol. 55.

[3] V. Paschos, *Applications of combinatorial optimization*. John Wiley & Sons, 2014, vol. 3.

[4] Y. Peng et al., "Graph learning for combinatorial optimization: a survey of state-of-the-art," *Data Science and Engineering*, vol. 6, no. 2, pp. 119–141, 2021.

[5] Q. Cappart et al., "Combinatorial optimization and reasoning with graph neural networks," *arXiv preprint arXiv:2102.09544*, 2021.

[6] J. Lenstra and A. Kan, "Computational complexity of discrete optimization problems," in *Annals of discrete mathematics*. Elsevier, 1979, vol. 4, pp. 121–140.

[7] J. Lenstra et al., "Complexity of machine scheduling problems," in *Annals of discrete mathematics*. Elsevier, 1977, vol. 1, pp. 343–362.

[8] E. Khalil et al., "Learning combinatorial optimization algorithms over graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[9] V. Vazirani, *Approximation algorithms*. Springer, 2001, vol. 1.

[10] I. Boussaïd et al., "A survey on optimization metaheuristics," *Information sciences*, vol. 237, pp. 82–117, 2013.

[11] Y. Bengio et al., "Machine learning for combinatorial optimization: a methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[12] I. Bello et al., "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[13] N. Mazyavkina, et al., "Reinforcement learning for combinatorial optimization: A survey," *Computers & Operations Research*, vol. 134, p. 105400, 2021.

[14] M. Nazari et al., "Reinforcement learning for solving the vehicle routing problem," *Advances in neural information processing systems*, vol. 31, 2018.

[15] C. Zhang et al., "Learning to dispatch for job shop scheduling via deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1621–1632, 2020.

[16] Q. Cappart et al., "Combining reinforcement learning and constraint programming for combinatorial optimization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 3677–3687.

[17] T. Wang et al., "Convjssp: Convolutional learning for job-shop scheduling problems," in *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2020, pp. 1483–1490.

[18] E. Min et al., "Transformer for graphs: An overview from architecture perspective," *arXiv preprint arXiv:2202.08455*, 2022.

[19] L. Rampášek et al., "Recipe for a general, powerful, scalable graph transformer," *Advances in Neural Information Processing Systems*, vol. 35, pp. 14 501–14 515, 2022.

[20] S. Yun et al., "Graph transformer networks," *Advances in neural information processing systems*, vol. 32, 2019.

[21] C. Ying et al., "Do transformers really perform badly for graph representation?" *Advances in Neural Information Processing Systems*, vol. 34, pp. 28 877–28 888, 2021.

[22] K. Hoffman et al., "Traveling salesman problem," *Encyclopedia of operations research and management science*, vol. 1, pp. 1573–1578, 2013.

[23] J. Błażewicz et al., "The job shop scheduling problem: Conventional and new solution techniques," *European journal of operational research*, vol. 93, no. 1, pp. 1–33, 1996.

[24] W. Hamilton, *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[25] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[26] K. Xu et al., "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[27] P. Veličković et al., "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[28] W. Hamilton et al., "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[29] A. Vaswani et al., "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[30] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.

[31] K. Han et al., "A survey on vision transformer," *IEEE transactions on pattern analysis and machine intelligence*, vol. 45, no. 1, pp. 87–110, 2022.

[32] S. Khan et al., "Transformers in vision: A survey," *ACM computing surveys (CSUR)*, vol. 54, no. 10s, pp. 1–41, 2022.

[33] Q. Li et al., "Deeper insights into graph convolutional networks for semi-supervised learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[34] Q. Sun et al., "Position-aware structure learning for graph topology-imbalance by relieving under-reaching and over-squashing," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 1848–1857.

[35] W. Hu, "Ogb-lsc: A large-scale challenge for machine learning on graphs," *arXiv preprint arXiv:2103.09430*, 2021.

[36] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[37] W. Hu et al., "Strategies for pre-training graph neural networks," *arXiv preprint arXiv:1905.12265*, 2019.

[38] S. Brody et al., "How attentive are graph attention networks?" *arXiv preprint arXiv:2105.14491*, 2021.

[39] R. Karp, *Reducibility among combinatorial problems*. Springer, 2010.

[40] J. Hopfield and D. Tank, ""neural" computation of decisions in optimization problems," *Biological cybernetics*, vol. 52, no. 3, pp. 141–152, 1985.

[41] K. Smith, "Neural networks for combinatorial optimization: a review of more than a decade of research," *Informs journal on Computing*, vol. 11, no. 1, pp. 15–34, 1999.

[42] O. Vinyals et al., "Pointer networks," *Advances in neural information processing systems*, vol. 28, 2015.

[43] W. Kool et al., "Attention, learn to solve routing problems!" *arXiv preprint arXiv:1803.08475*, 2018.

[44] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[45] M. Gasse et al., "Exact combinatorial optimization with graph convolutional neural networks," *Advances in neural information processing systems*, vol. 32, 2019.