Multi-View Graph Contrastive Learning for Solving Vehicle Routing Problems (Supplementary Material)

A MULTI-HOP RANDOM WALK

Regarding the subgraph samples used in our graph contrastive learning (as mentioned in the subsection of **Node-level representation learning** in the main paper), we first generate a n_q-neighbourhood-subgraph (i.e., n_q = 20 in our setting) around an anchor node, and then apply multi-hop random walk (MHRW) [Zhang et al., 2013] to generate subgraphs as the augmented samples (i.e. g^q and g^k), which are fed into the GNN encoders. We elaborate MHRW as follows.

In general, a VRP graph is a weighted undirected graph G with n nodes, and the cost c_{ij} ($c_{ij} > 0$) denotes the length of the edge $e_{i,j}$ between node v_i and node v_j, where we define c_{ii} = 0. The random walk on the graph G acts like that we roll dice at a node to decide which edge will be traversed next and lead to a new node. In order to sample a walk, we start from the anchor node, and then iteratively move from the current node to another node v_j within the n_q-neighbourhood-subgraph of the anchor node. In our implementation, the walk sampling depends on parameters that specify probabilities of walking to each node at the current step, i.e., the transit probability matrix $T = \{t_{ij}\}$. As stated in the main paper, we intuitively encourage more aggregation of the structural information from the vicinity than that from non-vicinity. Therefore, in our walk sampling, we specify the transit probability t_{ij} from node v_i to node v_j as $1/c^{\alpha}_{i,j}$, where α is the hyperparameter controlling the importance of edge cost during the walk (i.e., α = 1 in our setting). With the defined T, we iteratively visit the neighbourhood of the current node, until $\frac{3}{4}n_q$ nodes are collected for constructing a subgraph. Meanwhile, we hope the random walk concentrates more around the anchor node, and thus introduce a positive probability r (i.e., r = 0.8 in our setting) at each step for leading the walk back to the anchor node as did in [Tong et al., 2006].

During MHRW sampling, we collect the visited nodes to build the node set of the subgraph, and keep all edges between these nodes from the original graph. The generated subgraphs are then used as augmented samples to be processed by the GNN encoders for contrastive learning.

B DETAILS OF POMO AND ACTIVE SEARCH

B.1 POMO

In training, we adopt the state-of-the-art neural heuristic, i.e., Policy Optimization with Multiple Optima (POMO) [Kwon et al., 2020], to learn the solution construction step by step. Since POMO is developed on top of another popular neural heuristic, i.e., Attention Model[1] (AM) [Kool et al., 2019], we first introduce AM before POMO.

AM is a specialized encoder-decoder neural architecture for VRPs. The encoder mainly consists of multiple self-attention layers, which encode each node with its relationship to other nodes into a vector (i.e. embedding). Then, the decoder creates a route sequence (the solution) in a step-by-step manner, by utilizing the node embeddings and context embedding (from the encoder) to compute the query, key and value vectors for the dot-product attention mechanism.

Given node embeddings $\{x_i\}_{i=1}^{n}$ for each node v_i (note: in our work, they are the learned node representations from

[1] https://github.com/wouterkool/attention-learn-to-route

the pre-trained GNN), the encoder of AM firstly computes d_h-dim embeddings $h_i^{(0)}$ through two linear projections, i.e., $h_i^{(0)} = W^2(W^1 x_i + b^1) + b^2$. Then the node embeddings are processed by L self-attention layers. The embeddings produced by layer ℓ ($\ell \in \{1, \ldots, L\}$) is denoted as $h_i^{(\ell)}$. Specifically, each self-attention layer consists of two sub-layers, i.e., a multi-head attention (MHA) layer that executes message passing between the nodes, and a node-wise fully connected feed-forward (FF) layer, as follows,

$$\begin{aligned} \hat{h}_i &= \mathrm{BN}^{\ell}\left(h_i^{(\ell-1)} + \mathrm{MHA}_i^{\ell}\left(h_1^{(\ell-1)}, \ldots, h_n^{(\ell-1)}\right)\right), \\ h_i^{(\ell)} &= \mathrm{BN}^{\ell}\left(\hat{h}_i + \mathrm{FF}^{\ell}\left(\hat{h}_i\right)\right), \end{aligned} \tag{1}$$

where each sub-layer is also equipped with the skip-connection and batch normalization (BN) for stabilizing the training. Afterwards, node embeddings $h_i^{(L)}$ from the last layer of the encoder are fed into the decoder. The decoder sequentially calculates the probabilities of visiting each unvisited node with the attention layer followed by a Softmax function, which is used to construct the solution in a node-by-node manner.

POMO is essentially developed on top of AM. During policy optimization, POMO samples a set of solution trajectories $\left\{\tau^1, \tau^2, \ldots, \tau^N\right\}$ that start from each of all nodes, and gather each return $R\left(\tau^i\right)$ (i.e. the negative of tour length). During the training, it maximizes the expected return J by REINFORCE algorithm Williams [1992], with gradients computed as below,

$$\nabla_\theta J(\theta) \approx \frac{1}{n} \sum_{i=1}^{n} \left(R\left(\boldsymbol{\tau}^i\right) - b^i(s)\right) \nabla_\theta \log p_\theta\left(\boldsymbol{\tau}^i \mid s\right), \tag{2}$$

where $p_\theta\left(\boldsymbol{\tau}^i \mid s\right)$ means the probability produced by AM for the solution τ^i, given the instance s. Additionally, POMO uses a shared baseline $b^i(s)$ for the above gradients to reduce the variance as below,

$$b^i(s) = \frac{1}{n} \sum_{j=1}^{n} R\left(\boldsymbol{\tau}^j\right), \quad \text{for all } i. \tag{3}$$

During the inference, POMO produces multiple greedy trajectories by rotating each input instance and starting the trajectory from each of all nodes. The final solution is specified as the best one among all the sampled trajectories.

B.2 ACTIVE SEARCH

The active search, known as an inference boosting mechanism, is originally proposed in Bello et al. [2017], which actively updates the parameters of a model while it is used to infer the solution to an instance. Specifically, the inference starts with a trained model and iteratively optimizes its parameters with inferred solutions to an individual testing instance, while keeping track of the best one generated during the search (inference). This approach is verified to be competitive given a long runtime since it focuses on updating parameter for each individual instance. However, updating all parameters of the model as did in [Bello et al., 2017] is expensive and impractical. For example, it may cost several days to infer 10000 TSP100 instances. To tackle this issue, Efficient Active Search (EAS) is proposed in [Hottung et al., 2022] to only adjust a subset of parameters during inference, while keeping all other parameters fixed.

We adopt a similar technique as EAS, which adds instance-specific residual layers before the output layer of the attention decoder (within POMO) and only updates the parameters of these layers. In our MVGCL, the residual layers accept both node embeddings from the last attention layer and the graph embedding x_g from the pre-trained encoder f_q, as formulated in Eq. (3) in the main paper. The instance-specific layers are updated with the aforementioned REINFORCE algorithm in POMO, but we also use the imitation loss J_{IL} to increase the log-likelihood of generating the incumbent solutions such that,

$$\nabla_{\theta'} J_{IL}(\theta) = \log p_{\nabla_{\theta'}}\left(\overline{\boldsymbol{\tau}} \mid s\right), \tag{4}$$

where θ' represents parameters of the instance-specific layers and $\overline{\tau}$ is the incumbent solution so far, i.e., the best solution till the current search iteration.

C IMPLEMENTATION DETAILS

All the experiments are conducted with a single NVIDIA GTX 2080Ti GPU and i9-10940X CPU with 14 cores, including (pre-)training and inference. The implementation details of baselines and our method are described as below.

Problem		TSP50							TSP100						
Distribution	**Metric**	Concorde	AM	POMO	LCP	HAC	DROP	MVGCL	Concorde	AM	POMO	LCP	HAC	DROP	MVGCL
Uniform	Len.	5.72	5.91	5.86	5.86	5.87	5.87	**5.82**	7.78	8.10	7.93	7.94	8.08	7.97	**7.92**
	Gap	0.00%	3.32%	2.45%	2.45%	2.62%	2.62%	**1.75%**	0.00%	4.11%	1.93%	2.06%	3.86%	2.44%	**1.80%**
Avg. Inf. Time (s)		0.08	0.07	0.01	0.53	0.08	0.01	0.87	0.50	0.22	0.02	1.50	0.23	0.03	3.70
Problem		CVRP50							CVRP100						
Distribution	**Metric**	**HGS**	AM	POMO	LCP	**LKH**	DROP	MVGCL	**HGS**	AM	POMO	LCP	**LKH**	DROP	MVGCL
Uniform	Len.	10.55	10.82	10.71	10.76	10.56	10.74	**10.56**	15.68	16.26	16.10	16.20	15.78	16.18	**15.72**
	Gap	0.00%	2.56%	1.52%	1.99%	0.09%	1.80%	**0.09%**	0.00%	3.70%	2.68%	3.32%	0.64%	3.19%	**0.26%**
Avg. Inf. Time (s)		30	0.22	0.01	2.83	18.2	0.01	1.07	30	0.29	0.03	5.85	33.6	0.05	4.43

Table 1: Results of tour lengths and gaps for TSP and CVRP on the Uniform distribution, where LKH3 is also included

C.1 BASELINES

We calculate the optimal solution for TSP with Concorde solver.[2] Regarding CVRP, the best solution is calculated with a Python wrapper[3] for Hybrid Genetic Search algorithm (HGS). Regarding neural heuristic baselines, we adopt their open-sourced code on Github by keeping most of their original hyperparameters unchanged, except that we reduce the batch size in POMO [Kwon et al., 2020] to 56 for CVRP100 due to the memory limit. The original HAC was tested on TSP50 [Zhang et al., 2022], and we only change the node number to train a TSP100 model for its evaluation. In addition, we also modify the data loading process in each baseline so that they can be compatible with the data format in our experiments.

C.2 MVGCL

We adopt a 5-layer Graph Isomorphism Network[4] (GIN) [Xu et al., 2018] with 64 units per layer as the graph encoders in our implementation. In MHRW, we sample subgraphs with $\alpha = 1$, $n_q = 20$ and $r = 0.8$. Regarding MoCo, we set the Momentum $m = 0.999$ and InfoNCE temperature to 0.7.

Our MVGCL is developed on top of POMO[5] and active search[6]. For a fair comparison, we use the same neural architecture in POMO as the one in our MVGCL, where the dimension for node embeddings is 128 and the dimension for hidden units in the feed-forward layer is 512. The multi-head attention uses 8 heads and the dimension for the key in attention layers is 16. Besides, we set the logit clipping value to 10 and the weight decay factor to 1e-6 for the policy network. For the distribution-preserved augmentation, we set $(p1, p2, p3)$ to $(0.7, 0.2, 0.1)$. During training, we apply *early stopping* when the gap reduction is not significant. Regarding active search, the iteration number for each instance is fixed to 200. We add only one instance-specific residual layer before the output layer of the decoder for faster inference, though we find more layers for active search slightly improve the performance at the expense of significantly longer runtime. **The implementation code of our MVGCL will be made publicly available.**

D EXPERIMENTS ON UNIFORM DISTRIBUTION

We randomly generate 2000 instances of uniform distribution for the two VRP variants, i.e., TSP50, TSP100, CVRP50 and CVRP100. We test the same trained models as used in subsections of **Generalization on TSP** and **Generalization on CVRP** in the main paper. We also include LKH3 [Helsgaun, 2017] with 3000 trails as a baseline on CVRP50 and CVRP100. Results are gathered in Table 1, and it shows that our MVGCL can also deliver superior performance on uniform distribution, where the other neural heuristic baselines cannot generalize well. For example, our MVGCL achieves gaps of 1.75% and 1.80% on TSP50 and TSP100, respectively, while other baselines report gaps around 2%-4%. Besides, the gaps of our MVGCL on CVRP50 and CVRP100 are also fairly close to HGS at 0.09% and 0.26%, respectively, which outperform other

[2] https://www.math.uwaterloo.ca/tsp/concorde.html

[3] https://github.com/chkwon/PyHygese

[4] https://github.com/weihua916/powerful-gnns

[5] https://github.com/ydkwon/POMO/tree/master/OLD_ipynb_ver

[6] https://github.com/ahottung/EAS/blob/main/source/eas_lay.py

neural heuristic baselines and even LKH3 on CVRP100.

E DETAILED RESULTS ON TSPLIB

Here we display the detailed results of the 25 instances from TSPLib [Reinelt, 1991]. These instances are gathered from various sources, with different node distributions and problem sizes, which are desirable to be used for assessing the generalization performance of neural heuristics. Specifically, we solve the instances with 51-299 nodes by the MVGCL model trained on TSP100 with the mixed distributions. As shown in Table 2 (in the next page), our method significantly outperforms the neural heuristic baselines and most gaps to the optimal solutions are within 2.5%. It suggests our MVGCL is effective in tackling realistic instances from TSPLib, which are completely unseen during training.

References

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *Proc. of ICLR*, 2017.

Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 2017.

André Hottung, Yeong-Dae Kwon, and Kevin Tierney. Efficient active search for combinatorial optimization problems. In *Proc. of ICLR*, 2022.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *Proc. of ICLR*, 2019.

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. In *Proc. of NeurIPS*, 2020.

Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 1991.

Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast random walk with restart and its applications. In *Proc. of ICDM*, 2006.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

Zeyang Zhang, Ziwei Zhang, Xin Wang, and Wenwu Zhu. Learning to solve travelling salesman problem with hardness-adaptive curriculum. In *Proc. of AAAI*, 2022.

Zhongzhi Zhang, Tong Shan, and Guanrong Chen. Random walks on weighted networks. *Physical Review E*, 2013.

Instance	Metric	Opt.	AM	POMO	LCP	HAC	DROP	MVGCL
a280	Len.	2579	3132	3024	3031	3390	3038	**2765**
	Gap	0.00%	21.44%	17.27%	17.52%	31.43%	17.80%	**7.20%**
berlin52	Len.	7542	7978	7681	7593	7749	7575	**7573**
	Gap	0.00%	5.78%	1.84%	0.68%	2.75%	0.44%	**0.41%**
bier127	Len.	118282	126705	120501	121492	122108	123110	**119429**
	Gap	0.00%	7.12%	1.88%	2.71%	3.23%	4.08%	**0.97%**
ch130	Len.	6110	6360	6156	**6140**	6258	6262	6157
	Gap	0.00%	4.10%	0.75%	**0.50%**	2.42%	2.49%	0.76%
ch150	Len.	6528	6780	6600	6626	6986	6670	**6572**
	Gap	0.00%	3.87%	1.11%	1.51%	7.01%	2.17%	**0.68%**
d198	Len.	15780	18037	16660	16494	21455	17185	**16203**
	Gap	0.00%	14.30%	5.58%	4.53%	35.96%	8.90%	**2.68%**
eil101	Len.	629	659	648	**643**	665	644	644
	Gap	0.00%	4.73%	2.98%	**2.26%**	5.69%	2.42%	2.43%
eil51	Len.	426	438	435	440	437	435	**431**
	Gap	0.00%	2.88%	2.08%	3.24%	2.53%	2.06%	**1.10%**
eil76	Len.	538	567	557	554	554	565	**552**
	Gap	0.00%	5.33%	3.50%	2.97%	3.05%	5.08%	**2.55%**
gil262	Len.	2378	4254	3226	3232	5117	3236	**2649**
	Gap	0.00%	78.91%	35.65%	35.91%	115.18%	36.08%	**11.38%**
kroA150	Len.	26524	28634	27126	26973	29045	26971	**26842**
	Gap	0.00%	7.96%	2.27%	1.69%	9.50%	1.69%	**1.20%**
kroB150	Len.	26130	26898	26632	26391	28337	26642	**26302**
	Gap	0.00%	2.94%	1.92%	1.00%	8.44%	1.96%	**0.66%**
lin105	Len.	14379	15272	14604	14718	15632	**14513**	14536
	Gap	0.00%	6.21%	1.56%	2.36%	8.72%	**0.93%**	1.09%
pr107	Len.	44303	45681	44933	46595	49153	45865	**44416**
	Gap	0.00%	3.11%	1.42%	5.17%	10.95%	3.53%	**0.25%**
pr152	Len.	73682	79293	**74902**	76145	79990	77069	75061
	Gap	0.00%	7.61%	**1.66%**	3.34%	8.56%	4.60%	1.87%
pr226	Len.	80369	87801	83754	85406	92527	85437	**81896**
	Gap	0.00%	9.25%	4.21%	6.27%	15.13%	6.31%	**1.90%**
pr264	Len.	49135	57716	54589	54735	71243	54445	**50713**
	Gap	0.00%	17.46%	11.10%	11.40%	44.99%	10.81%	**3.21%**
pr299	Len.	48191	59850	53926	54920	67690	53479	**51354**
	Gap	0.00%	24.19%	11.90%	13.96%	40.46%	10.97%	**6.56%**
pr76	Len.	108159	108582	**108404**	111196	109787	108572	109826
	Gap	0.00%	0.39%	**0.23%**	2.81%	1.50%	0.38%	1.54%
rat195	Len.	2323	2620	2490	2490	2693	2490	**2374**
	Gap	0.00%	12.78%	7.18%	7.19%	15.93%	7.17%	**2.18%**
rat99	Len.	1211	1263	1237	1256	1297	1255	**1232**
	Gap	0.00%	4.33%	2.18%	3.74%	7.13%	3.62%	**1.76%**
rd100	Len.	7910	8030	8004	8132	8140	7996	**7922**
	Gap	0.00%	1.52%	1.19%	2.80%	2.91%	1.08%	**0.16%**
st70	Len.	675	693	685	691	693	683	**678**
	Gap	0.00%	2.67%	1.51%	2.39%	2.65%	1.14%	**0.43%**
tsp225	Len.	3916	4352	4159	4209	4588	4215	**4024**
	Gap	0.00%	11.15%	6.21%	7.47%	17.17%	7.63%	**2.75%**
u159	Len.	42080	44908	42888	44062	47500	42751	**42623**
	Gap	0.00%	6.72%	1.92%	4.71%	12.88%	1.59%	**1.29%**
Avg. Gap		0.00%	10.53%	5.16%	5.92%	16.75%	5.79%	**1.58%**
Avg. Inf. Time (s)		-	0.48	0.47	69.26	0.48	0.35	48.11

Table 2: Results on TSPLib