On the importance of edge weights and features in the Travelling Salesman Problem

Project for the GNNs course in BISS 2024

Davide Zago[1] and **Lorenzo Sciandra**[2]

[1]University of Turin, Dept. of Computer Science, via Pessinetto 12, 10149 Torino
[2]University of Turin, Dept. of Mathematics, Via Carlo Alberto, 10, 10123 Torino

September 2024

1 Introduction

The Travelling Salesman Problem (TSP) is a classic one in Combinatorial Optimization (CO), and for decades an enormous amount of research has been devoted to its study. The symmetric TSP is defined on an undirected graph $G = (V, E)$ where $V = \{1, ..., n\}$ are vertexes and $E = \{(i, j) \mid i, j \in V\}$ are edges, each one with its value c_{ij} that defines the cost of moving from city i to j. The problem requires determining a Hamiltonian cycle H^* (i.e. a closed path that passes through all vertices of G exactly once), in this context called *tour*, so that it minimizes the total edge traversing cost $c(H^*) = \sum_{(i,j)\in H^*} c_{ij}$. The TSP has been proven to be NP-complete.

This project aims to test how the use of edge weights and edge features impacts the performance of state-of-the-art Graph Neural Networks (GNNs) on the TSP. This work has been developed as an outcome of the GNN course held at the Bertinoro International Spring School 2024. The rest of the article is structured as follows. Section 2 describes in detail the tasks that have been tackled with our investigation. Then, the deep learning and graph-based neural models will be described in section 3. Some implementation details are highlighted in section 4 as well as the code framework used for training and testing. Results will be presented in section 6 with the description of the experimental setting in section 5, and in section 7 some final considerations will be said along with some possible future developments. Our code can be consulted at the Github repository: TSP tasks with GNNs.

2 Tasks

In this work, we consider the Euclidean version of the TSP, in which the cost between cities is the Euclidean metric. In particular, we consider the 2D Euclidean version, which has been extensively studied in both CO and Machine Learning (ML) literature.

As usually done in literature, we can represent a 2D Euclidean TSP instance with a matrix $\mathbf{X} \in \mathbb{R}^{|V| \times 2}$, in which each row $\mathbf{x}_i$ represents the 2D coordinates of a point in the cartesian plane. Every 2D Euclidean instance can be rescaled to fit in the $[0, 1] \times [0, 1]$ without deforming the set of optimal tours. Therefore, we can generate random 2D Euclidean TSP instances sampling coordinates in this space uniformly without loss of generality.

The cost c_{ij} between two nodes i and j is computed as the euclidean distance from their coordinates $\|\mathbf{x}_i - \mathbf{x}_j\|_2$. The distance can be used in different ways, e.g. as a scalar edge feature for edges, or as edge weight for models that support weighted adjacency matrices (see section 3 for details).

In this work, we tackled three tasks related to the TSP:

1. **Cost regression**: predict the cost of the optimal tour given a TSP instance;
2. **Tour prediction**: approximate a symmetric matrix $\mathbf{D}$ that contains in each entry the probability p_{ij} that the corresponding edge (i, j) is in one optimal tour;
3. **Heuristic solution generation**: construct the best possible feasible heuristic solution sequentially via iterated sampling of cities.

Tasks 1 (section 2.1) and 2 (section 2.2) are solved with supervision using optimal solutions obtained through Concorde solver [App+06]. Task 3 (section 2.3) is solved with Reinforcement Learning (RL) with negative tour cost as a reward signal, therefore without the need for supervision.

2.1 Cost regression

Predicting the optimal tour cost of a TSP instance is a nontrivial task that requires an appropriate model able to create a meaningful latent representation of the underlying graph. We use a deep learning model f^e_θ (that we call *encoder*) as a function to map a 2D Euclidean TSP instance X to $\mathbf{H} \in \mathbb{R}^{|V| \times d_h}$, where d_h is the dimension of the latent vectorial representation of the nodes. These representations of nodes are also called *embeddings*.

The optimal tour cost is a piece of information that requires global knowledge over the instance. For this reason, another function f^a needs to be defined to aggregate the information from all the embeddings $\{\mathbf{h}_i\}_{i=1,\dots,n}$ into a single global graph embedding $\mathbf{h}_G$. This function can be the sum, the average, or other more sophisticated aggregation techniques, or it can be itself another neural

model. Finally, a classifier f_ψ^c is used to map the global embedding into the predicted optimal tour cost.

Weight updates are done by computing the gradient of a loss function. We define our cost regression loss L^{CR} as the Mean Squared Error between the predicted cost and the actual optimal tour cost of all instances in the dataset D, as follows.

$$L^{CR}(\theta, \psi) = \frac{1}{|D|} \sum_{X \epsilon D} (c^*(X) - f_{\theta,\psi}(X)))^2$$

where $c^*(X)$ is the cost of the optimal tour $\pi^*(X)$ of the instance X and $f_{\theta,\psi} = f_\psi^c \circ f^a \circ f_\theta^e$ is the composition of the encoder, aggregation and classifier functions.

2.2 Tour prediction

Using the previous notation, the tour prediction task can be defined as a function $f_\theta = f^p \circ f_\theta^e$, where the encoder component f_θ^e may be the same as above. In addition to this, the prediction function $f^p : \mathbb{R}^{d_h} \times \mathbb{R}^{d_h} \to [0, 1]$ is used to forecast the likelihood of an edge between each pair of nodes using their embeddings $\mathbf{H}$. One of the simplest ways to define f^p is as a similarity function operating on the final embeddings, such as the cosine similarity. Subsequently, a rescaled function, like a sigmoid, can be employed to ensure that each value falls within the range of $[0, 1]$ if desired.

In this task, the loss function used is the Mean Frobenius norm of the difference between the ground truth and the predicted matrices over all instances:

$$L^{TP}(\theta) = \frac{1}{|D|} \sum_{X \epsilon D} \|\mathbf{D}_X^* - f_\theta(X)\|_F$$

where $\mathbf{D}_X^*$ is the gold matrix a priori defined with Concorde for each instance X.

2.3 Tour generation

[Bel+16] has been the first work to propose a Reinforcement Learning (RL) technique to solve CO problems of variable sizes. This paper opened the way for a multitude of works that using similar techniques can solve CO problems with increasing levels of precision. Inspired by the original work and the following ones [Bel+16; KVW18; Kwo+20; HKT21], we experiment with the effectiveness of neural models with a typical heuristic tour generation technique with RL.

Heuristic tour generation is an unsupervised task in which the model learns to improve the generated solution solely on solution cost (the reward, in RL terms). The overall neural architecture is composed of a first model f_θ^e that builds embeddings of nodes and a subsequent model f_ψ^p that predicts the next city.

As frequently done in construction heuristics with RL, the REINFORCE [Wil92] loss functions is used:

$$L^G(\theta, \psi) = \frac{1}{|D|} \sum_{X \in D} \log P_{\theta,\psi}(\pi|X) \cdot c(\pi)$$

where π is the generated tour with probability $P_{\theta,\psi}(\pi|X) = \prod_{i=1}^{|V|} f_\psi^p(\pi_i|\mathbf{H}, \pi_{<i})$ and $c(\pi)$ is its cost.

3 Models

3.1 Graph neural networks background

In the context of GNNs, it is usual to denote the set of all node features with a matrix $\mathbf{X}$ and the topology of the network with an adjacency matrix $\mathbf{A}$ (where $a_{i,j} = 1$ iff there is an edge $(i, j) \in E$, 0 otherwise). The adjacency matrix can also be weighted, therefore containing any real value.

In our setting, we experiment with GNNs models constructed to preserve graph isomorphism and so by dealing with the symmetry group Σ_n of permutations over the n nodes [Bro+21]. More precisely, depending on the considered task, we want to achieve equivariance or invariance to the group action $\mathfrak{g}.G$, $\forall \mathfrak{g} \in \Sigma_n$ for all the graphs G with n nodes. Since we are interested in linear group actions, called group representations, we can associate to every group action $\mathfrak{g}$ the corresponding invertible matrix $\rho(\mathfrak{g})$, that, in this scenario leads to a permutation matrix $\mathbf{P}$.

For example, when we are dealing with the cost regression task we want the function f learned by the neural network to be invariant to every permutation of the nodes:

$$\forall \mathbf{P} \in \Sigma_n. \quad f(\mathbf{PX}, \mathbf{PAP}^T) = f(\mathbf{X}, \mathbf{A})$$

and this will be achieved through a series of equivariant GNNs layers followed by an invariant aggregation that constructs, at the end, the embedding of the overall graph. The regression is made through a single linear projection of the learned latent representation. Sometimes instead we want the output of the GNN to be equivariant to the permutation:

$$\forall \mathbf{P} \in \Sigma_n. \quad f(\mathbf{PX}, \mathbf{PAP}^T) = \mathbf{P} f(\mathbf{X}, \mathbf{A})$$

This is the case of the prediction of the tour matrix. We would like that, no matter the permutation of the nodes of the graph, the probability p_{ij} associated with an edge $(i, j) \in E$ to be in an optimal tour would be the same after the relabelling of the cities i and j. To do that only equivariant layers are used in the model.

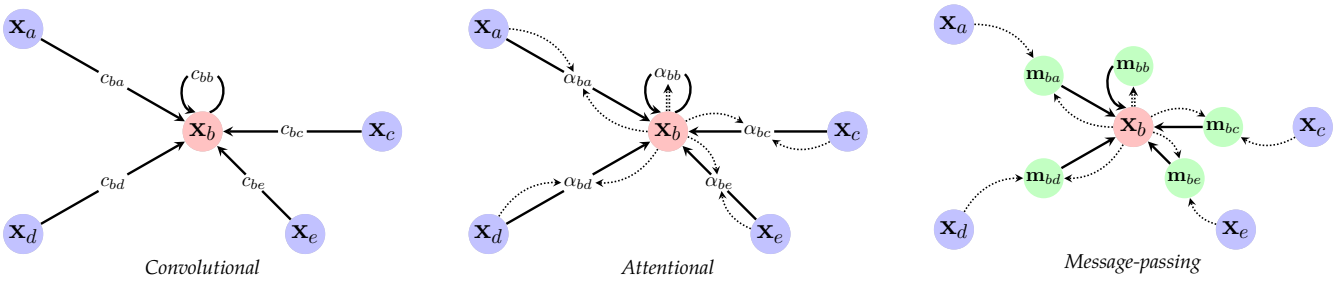

Figure 1: Credit [Bro+21]

Every GNN layer can be defined through the message-passing scheme. This approach is defined as local, i.e. the update, and so the information extraction is done and then propagated at node level. The general update formula for the embedding $\mathbf{x}_i$ at layer depth l, is:

$$\mathbf{x}_i^{(l)} = \gamma^{(l)}\left(\mathbf{x}_i^{(l-1)}, \bigoplus_{j \in \mathcal{N}(i)} \phi^{(l)}\left(\mathbf{x}_i^{(l-1)}, \mathbf{x}_j^{(l-1)}, \mathbf{e}_{ij}^{(l-1)}\right)\right) \qquad \forall i \in V$$

where $\mathcal{N}(i)$ is the set of all neighbors of the node i, $\gamma^{(l)}$ and $\phi^{(l)}$ are differentiable functions, $\bigoplus$ a permutation invariant aggregation (e.g. sum), and $\mathbf{e}_{ij}$ are edge level features.

Depending on the definition of the propagation function ϕ, we can identify three different "flavors" of GNNs as depicted in fig. 1:

1. **Convolutional**: the aggregation of the neighborhood nodes is done with fixed weights w_{ij} that specifies the importance of node j to node i's representation:

   $$\phi\left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{ij}\right) = w_{ij} \cdot \xi(\mathbf{x}_j)$$

   where $\xi(\cdot)$ is generally a non-linear function of the embedding. Note that w_{ij} are a priori defined for all pairs of nodes and don't change across layers;

2. **Attentional**: there is a self-attentional mechanism that dynamically computes the importance (per-level) weights α_{ij}:

   $$\phi\left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{ij}\right) = a(\mathbf{x}_i, \mathbf{x}_j) \cdot \xi(\mathbf{x}_j) = \alpha_{ij} \cdot \xi(\mathbf{x}_j)$$

   that defines a distribution over the neighborhood;

3. **Message-passing**: layers in which the ϕ function is not simplified to a re-scaling factor (as in the previous cases) fall into this category. The propagation here is a learnable function that computes arbitrary "messages" (essentially vectors) across edges, generally using the corresponding features $\mathbf{e}_{ij}$.

3.2 Edge weights and edge features

The objective of this work is to investigate the importance of edge weights and features when solving TSP-related problems. Given a certain GNN model, we identify three non-overlapping settings depending on which graph information is used: 1) node feature-only; 2) presence of edge weights; and 3) presence of edge features. A set of relevant deep neural models is considered: Graph Convolutional Network [KW16] (GCN), Graph Isomorphism Network [Xu+18] (GIN), Graph Attention Network [Vel+17; BAY21] (GAT), and lastly we choose to compare the Transformer [Vas17] as baseline. Notably, the Transformer, as shown in [Jos20], can be classified as a GNN in the attentional category. Indeed, it assumes a complete graph for the input feature vectors, and a dynamic attention weight is computed to define the importance of one vector to another.

Each model natively uses some specific information about the graph, and we extend its standard formulation including edge weights and edge embedding updates when missing. Note that the extension with the edge weights does not alter the model classification as convolutional, attentional, or message-passing.

By introducing edge weights in GAT layers, we modify the node embedding update formula as follows:

$$\mathbf{x}'_i = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{ij} \Theta_x \mathbf{x}_j \quad \Longrightarrow \quad \mathbf{x}'_i = \sum_{j \in \mathcal{N}(i) \cup \{i\}} w_{ij} \alpha_{ij} \Theta_x \mathbf{x}_j$$

where the attention weights are computed with the formula

$$\alpha_{ij} = \frac{\exp\left(\mathbf{a}^T \sigma(\Theta_x \mathbf{x}_i + \Theta_x \mathbf{x}_j)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathbf{a}^T \sigma(\Theta_x \mathbf{x}_i + \Theta_x \mathbf{x}_k)\right)}$$

and the weight w_{ij} is computed through Kipf normalization of the edge weights (see [KW16] for details). Similarly, the standard GIN update is modified:

$$\mathbf{x}'_i = h_\Theta\left((1+\epsilon) \cdot \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j\right) \quad \Longrightarrow \quad \mathbf{x}'_i = h_\Theta\left(w_{ii}(1+\epsilon) \cdot \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} w_{ij} \mathbf{x}_j\right)$$

Apart from GCNs, all the graph neural networks we consider support edge features. We choose to execute the edge embedding update before every GNN layer as follows:

$$\mathbf{e}'_{ij} = \mathrm{MLP}_\theta([\mathbf{e}_{ij}, \mathbf{x}_i, \mathbf{x}_j])$$

where MLP is a Multilayer Perceptron and $[\cdot]$ is the concatenation. The obtained edge embeddings are used in node embedding updates. The set of models used with their modifications is summarized in table 1.

Type	Model	Convolutional	Attentional	Message-passing
Features-only	Transformer	✓	✓	✗
	GCN	✓	✗	✗
	GIN	✓	✗	✗
	GAT	✓	✓	✗
Edge weights	GCN	✓	✗	✗
	GIN*	✓	✗	✗
	GAT*	✓	✓	✗
Edge features	GIN†	✓	✗	✓
	GAT†	✓	✓	✓

Table 1: Summary of the characteristics of each graph neural network used. "$*$" indicates models in which we implemented the Kipf normalization [KW16]. "†" indicates modes (with built-in edge features utilization) in which we added an edge feature layer (see 3.2). Note: GIN† can be actually seen as GINE [Hu+19].

4 Implementation Details

We develop a modular and customizable framework for our experiments with PyTorch [Pas+17] following an agent-based architecture using the `salina` library [Den+21]. This tool extends PyTorch and enables to implementation of many Deep Learning tasks (especially sequence problems) by composing simple building blocks, promoting readability and modularity. We design our graph models using the popular `PyTorch Geometric` library [FL19] (PyG).

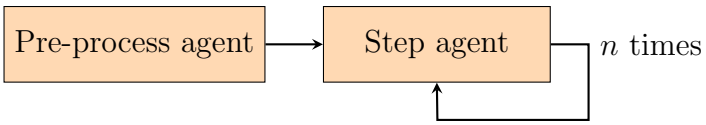

Figure 2: Diagram of our agent-based framework.

In fig. 2 is displayed the general architecture we used to define all of our tasks. The *pre-process agent* is a composition of agents (essentially functions) that are executed once and before everything else; the *step agent* is a composition of agents that are executed n times until a stop condition is reached.

We can define all of our tasks with this simple agent-based abstraction (table 2). **Cost prediction** pre-processing sequence consists of instance generation; acquisition of the optimal tour with Concorde; and prediction of the optimal tour with the model. The step agent consists solely

Task	Pre-process	Step
Cost prediction	(1) Instance generation, (2) Concorde, (3) Cost prediction†	(1) Cost computation
Tour prediction	(1) Instance generation, (2) Concorde, (3) Tour prediction†	-
Tour generation	(1) Instance generation, (2) Encoding†	(1) Prob. dist. computation†, (2) Action, (3) Masking, (4) Cost computation

Table 2: Summary of the components of each task. The components marked with "†" are neural networks.

of computing the optimal tour cost. **Tour prediction** can be defined analogously but no step action is needed. **Tour generation** pre-processing includes instance generation and computation of city embeddings. Every step the following actions are executed: calculation of the probability distribution over the next city with a PointerNetwork [VFJ15]; sampling or greedy-choice of the next city; masking of the chosen city for future choices; update of the cost of the partially generated tour.

5 Experimental Settings

All the experiments have been run on a computing node with a 16-core Intel Core i9-9900KF CPU 3.60GHz, 32 GB of RAM, and a Nvidia TITAN RTX 24GB GPU.

5.1 TSP instances

As mentioned above we used 2D Euclidean instances with node coordinates sampled in the $[0, 1] \times [0, 1]$ square. The size of the analyzed graphs is set at $|V| = 100$ since it strikes a good trade-off, tackling fairly complex problems without necessitating excessively large models that would require powerful dedicated hardware for training.

5.2 Architectural and training hyperparameters

Many hyperparameters of the GNNs are set equal across all models that share them:

- **Normalization** and **residual connections**: to train deep models without incurring the over-smoothing or suspended animation problem [Pri23] a normalization rule coupled with residual connections are used in the definition of every GNN layer. The normalization rule employed is the GraphNorm, which normalizes only across embeddings within the same graph, which we found to be more effective than the standard BatchNorm. The normalization step is always applied after the residual connection to control the mean and variance of the activations, preventing them from becoming excessively large;
- **Embeddings**: the dimension of the node embeddings is fixed to $d_h = 128$ that is achieved with a first linear mapping $\mathbb{R}^2 \to \mathbb{R}^{d_h}$ applied to the input features of every node. Not all models use edge embeddings, but when they do, the same dimensionality is applied to allow us to compose edge and node embeddings. More precisely, every edge embedding is initially created using a linear map $\mathbb{R} \to \mathbb{R}^{d_h}$ from the corresponding weight. At each step, they are updated with a residual connection and a non-linear transformation (with an MLP) of the embeddings of the nodes they connect. Finally, a GraphNorm is applied;
- **Activation function**: we experimented with various activation functions, including sigmoid, tanh, and those from the ReLU family. Ultimately, we found that no single activation function consistently outperformed the others across all tasks and GNN layers. Therefore, when non-linearity is required, we opted to use the classical tanh;
- **Number of layers**: we tested different depths, ranging from 2 to 8, and observed that usually beyond the 3rd layer, no significant improvement was achieved. As a result, we consistently set the number of layers for both the MLPs and GNNs to 3;
- **Optimizer**: for the training procedure, we used the classic Adam optimizer with gradient clipping and learning rate 1e−3 for generation and 1e−4 for supervised tasks. Hyperparameter tuning is considered a future development.

6 Results

The results will be displayed here in tables for readability purposes. For a more detailed overview, please refer to the following experiment links: cost regression, tour prediction and tour generation.

6.1 Cost regression

In addition to the classical training and test loss, we also include in the cost regression table 3 the `Err%` metric, which represents the percentage of the mean gap between the model's output and the true optimal value. Note that every standard model improves in performance when enhanced with edge information, and indeed, the best results are consistently achieved by the models using edge weights. However, there is no correlation between increasing the amount of additional edge information and performance improvement. It is important to note that the Transformer performs poorly compared to GNN models, predicting a minimum that, on average, costs more than one-third of the actual optimal solution.

Table 3: Cost regression results. The best value in each comparison is highlighted in bold.

		Train		Test	
Type	**Model**	**Loss**	**Err%**	**Loss**	**Err%**
Features-only	Transformer	0.142	4.419%	7.913	35.015%
	GCN	0.137	4.601%	0.022	1.421%
	GIN	0.179	4.707 %	0.039	1.879%
	GAT	0.129	3.900%	0.021	1.367%
Edge weights	GCN	0.095	3.353 %	0.029	1.676%
	GIN	**0.071**	**2.923%**	0.050	2.263%
	GAT	0.142	4.127%	**0.015**	**1.229%**
Edge features	GINE	0.127	4.030%	0.031	1.547%
	GAT	0.158	4.305%	0.018	1.438%

6.2 Tour prediction

For the tour prediction results in table 4, an actual metric is the `#Err`, which measures the average number of errors as the differences between the edges of the true optimal tour and those predicted by the model. Similar to the cost regression task, we found that the enhanced models—whether using only edge weights or edge features outperformed the classical ones. The best results were again achieved by directly incorporating only the edge weights. However, we observe that the Transformer performs significantly better than both the GCN and GAT, even in their extended versions. The only GNN model that consistently achieves better results is GIN, across all three of its variants.

Table 4: Tour prediction results. The best value in each comparison is highlighted in bold.

		Train		Test	
Type	**Model**	**Loss**	**#Err**	**Loss**	**#Err**
Features-only	Transformer	16.925	3.75	17.230	3.688
	GCN	60.436	33.25	59.911	33.125
	GAT	65.813	35.625	65.341	35.406
	GIN	17.030	**3.25**	**17.157**	3.469
Edge weights (ours)	GCNW	30.852	8	31.616	9.649
	GATW	60.674	34	59.902	33.156
	GINW	**16.813**	**3.25**	17.830	**3.344**
Edge features (ours)	GATE	56.923	30.25	58.959	32.594
	GINE	17.226	3.125	17.877	3.625

6.3 Tour generation

In the tour generation task 5, the average cost of the generated solutions is used as a quality metric for the model. GIN shows peak performance in the feature-only setting, directly followed by GAT with edge weights. GAT with edge features surprisingly shows high performance in training but loses its potential in generalization. Overall, every GNN model outperforms the Transformer. It's important to note that the algorithm used for tour generation (and reinforcement learning in general) are very sensitive to changes and show high variance. Therefore, in many cases the expressivity of the model is not the major factor that influences the final performance.

7 Conclusion

Since the best results across all the tasks are often achieved using enhanced models with edge weights or embeddings, we can conclude that, as expected, the direct use of cost information c_{ij} for each edge $(i, j) \in E$ is relevant for TSP-related tasks, as they inherently involve minimizing the overall tour. However, it does not appear necessary to increase the number of parameters and computations with edge embedding; relying solely on edge weights is almost always sufficient. Although, further investigation on larger-scale problems is needed to better evaluate the importance of model components.

In conclusion, the optimal choice of the enhanced model version and GNN architecture appears to be task-dependent, although the differences in performance are not highly significant.

Table 5: Tour generation results. The best value for the quality metric is highlighted in bold (the lower the better).

Type	Model	Train Mean Cost	Validation Mean Cost
Features-only	Transformer	9.760	9.843
	GCN	8.282	8.647
	GAT	8.312	8.629
	GIN	8.178	**8.541**
Edge weights (ours)	GCN	8.270	8.773
	GAT	8.338	8.595
	GIN	8.336	8.821
Edge features (ours)	GAT	**8.139**	8.745
	GIN	8.630	8.777

References

[Pri23] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023. URL: http://udlbook.com.

[BAY21] Shaked Brody, Uri Alon, and Eran Yahav. "How attentive are graph attention networks?" In: *arXiv preprint arXiv:2105.14491* (2021).

[Bro+21] Michael M. Bronstein et al. "Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges". In: *arXiv preprint arXiv:2104.13478* (2021).

[Den+21] Ludovic Denoyer et al. *SaLinA: Sequential Learning of Agents*. https://gitHub.com/facebookresearch/salina. 2021.

[HKT21] André Hottung, Yeong-Dae Kwon, and Kevin Tierney. "Efficient active search for combinatorial optimization problems". In: *arXiv preprint arXiv:2106.05126* (2021).

[Jos20] Chaitanya Joshi. "Transformers are Graph Neural Networks". In: *The Gradient* (2020).

[Kwo+20] Yeong-Dae Kwon et al. "Pomo: Policy optimization with multiple optima for reinforcement learning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 21188–21198.

[FL19] Matthias Fey and Jan E. Lenssen.
"Fast Graph Representation Learning with PyTorch Geometric".
In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[Hu+19] Weihua Hu et al. "Strategies for pre-training graph neural networks".
In: *arXiv preprint arXiv:1905.12265* (2019).

[KVW18] Wouter Kool, Herke Van Hoof, and Max Welling.
"Attention, learn to solve routing problems!"
In: *arXiv preprint arXiv:1803.08475* (2018).

[Xu+18] Keyulu Xu et al. "How powerful are graph neural networks?"
In: *arXiv preprint arXiv:1810.00826* (2018).

[Pas+17] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[Vas17] A Vaswani. "Attention is all you need".
In: *Advances in Neural Information Processing Systems* (2017).

[Vel+17] Petar Veličković et al. "Graph attention networks".
In: *arXiv preprint arXiv:1710.10903* (2017).

[Bel+16] Irwan Bello et al. "Neural combinatorial optimization with reinforcement learning".
In: *arXiv preprint arXiv:1611.09940* (2016).

[KW16] Thomas N Kipf and Max Welling.
"Semi-supervised classification with graph convolutional networks".
In: *arXiv preprint arXiv:1609.02907* (2016).

[VFJ15] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer networks".
In: *Advances in neural information processing systems* 28 (2015).

[App+06] David L. Applegate et al. *The Traveling Salesman Problem: A Computational Study*.
Princeton University Press, 2006. ISBN: 9780691129938.

[Wil92] Ronald J Williams. "Simple statistical gradient-following algorithms for
connectionist reinforcement learning". In: *Machine learning* 8 (1992), pp. 229–256.