Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Reinforcement Learning of TSP Heuristics with Message Passing Neural Networks

Distributed Systems Lab Report

Loïc Holbein, Yannick Schmid

holbeinl@student.ethz.ch, schmidya@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory

Supervisors:
Joël Mathys, Flint Xiaofeng Fan
Prof. Dr. Roger Wattenhofer

August 4, 2023

Acknowledgements

We want to thank our supervisors Flint Xiaofeng Fan and Joël Mathys for guiding this thesis. They provided crucial advice on graph neural networks and reinforcement learning during our weekly consultations.

Furthermore, we would like to thank Prof. Dr. Roger Wattenhofer and the Distributed Computing Group of ETH Zürich for enabling this project by allowing us to use their computation cluster.

Abstract

The TSP is a hard combinatorial optimization problem, which requires heuristics to be solved in practice. We create two types of reinforcement learning environments that allow an agent to find feasible solutions for this problem, which correspond to two different approaches to design TSP heuristics. These environments represent the current state of the solution as graphs, so we employ graph neural networks to control our agents. These graphs are dense and contain multi-dimensional edge attributes. We find that existing graph attention layers struggle to extract information from these edge attributes, and implement an extension which overcomes these problems.

Contents

Chapter 1

Introduction

Combinatorial optimization problems have many important applications in logistics, network design, supply chain management, bioinformatics, or game theory. To solve these problems, one must find the element from a discrete set of feasible solutions that minimizes a given cost function. Unfortunately, feasibility regions are often exceedingly large and finding the optimum turns out to be computationally hard.

Heuristics offer a practical and efficient approach to tackle complex combinatorial optimization problems. They use domain specific knowledge to explore and navigate the problem space in order to arrive at close to optimal solutions in reasonable time. Good heuristics often appear surprisingly simple but their design requires in-depth expertise of the problem at hand. Using reinforcement learning (RL), machine learning models can be trained to acquire this knowledge. In the RL paradigm, the model is trained to maximize its reward when acting in some environment. This framework has allowed machine learning to be applied to a diverse set of problems, such as control tasks and games like chess and go.

In this project, we demonstrate the use of reinforcement learning to learn heuristics for the Travelling Salesperson Problem (TSP). The TSP is a canonical combinatorial optimization problem, where the goal is to find a tour (or Hamiltonian cycle) through a complete graph with minimal total edge weights. Because it is $\mathcal{NP}$-hard, finding a good heuristic is especially attractive for any practical application. Furthermore, one is often only interested in solutions for a specific type of graph, either of some fixed size or whose weight matrix observes some property: For instance, graphs with a weight matrix that obeys the triangle inequality, or weights that correspond to node distances in a plane embedding. Hence, it is desirable to create heuristics that are trained on some specific set of graphs which have these properties.

Like many other combinatorial optimization problems, the TSP is modeled on a graph, which complicates the use of standard neural networks. Instead, we propose to use graph neural networks (GNNs). GNNs are designed to be equivariant under graph isomorphisms, similar to image convolution networks being invariant under translations, and accommodate variable graph sizes, making them ideal for

a large number of graph problems. GNNs naturally produce one output per node. This fits the RL use case well, since agents must generate either a probability distribution or a value estimation over graph vertices, in order to continue the optimization task.

The TSP accommodates two types of heuristics: 1) a constructive approach, where the agent successively adds nodes until the tour is complete, or 2) a local search approach, where the agent starts with a complete tour and then iteratively modifies this tour by swapping out some edges. We implement both of these approaches and compare them. We test a number of RL methods and implementations. While we implement these algorithms ourselves, we also compare them against a set of implementations from a dedicated RL package. We compare a number of well-established graph convolution schemes, and also provide a custom convolution design, purpose-built to the given task.

Even though combining reinforcement learning and graph neural networks in order to solve combinatorial optimization problems is a challenging task, our results on the TSP show that it is possible to outperform other well-known heuristics on certain problem instances.

Chapter 2

Background

This work lies in the intersection of three research directions of mathematics and computer science: combinatorial optimization, reinforcement learning and graph neural networks. In this chapter, we will briefly introduce these three fields, and zoom-in on the topics which are most relevant for this thesis.

2.1 Combinatorial Optimization

In combinatorial optimization (CO), the goal is to find an optimal solution to a combinatorial problem. A problem is called combinatorial if its space of feasible solutions is discrete. As such, many problems with real life applications fall in this category.

2.1.1 The Traveling Salesperson Problem

The Traveling Salesperson Problem (TSP) is a canonical CO problem. Given a set of cities and travel costs between them, the goal is it to visit all cities exactly once in the cheapest way possible, and finally return back home to the starting city.

More formally, the cities can be represented as a (complete) graph $G = (V, E)$ with $|V| = n$. Additionally, an edge weight function $w : E \mapsto \mathbb{R}$ captures the travel costs between cities. The chosen route of visiting all cities induces an order $\mathcal{T} = v_0, v_1, \ldots, v_{n-1}, v_n$ over the nodes: $v_i \in V$ is the i'th city visited. Node v_0 is the starting city, also referred to as the *depot*. In order to fulfill the above constraints about the traveled route, $\mathcal{T}$ must encompass all vertices in V, must be n edge hops long, and its first and last element must be identical. In other words, $\mathcal{T}$ is a Hamiltonian Cycle. In the context of the TSP, such cycles are usually referred to as *tours*.

The ultimate goal is to find a tour $\mathcal{T}$ with the smallest possible associated travel cost. To that end, we introduce the cost function C which assigns travel

cost $C(\mathcal{T})$ to tours $\mathcal{T}$:

$$C(\mathcal{T}) := \sum_{i=0}^{n-1} w(v_i, v_{i+1})$$

Hence, the optimization problem becomes:

$$\begin{aligned} \min_{\mathcal{T}} \quad & C(\mathcal{T}) \\ \text{s.t.} \quad & V = \{v_0, v_1, \ldots, v_{n-1}\} \\ & v_0 = v_n \end{aligned} \tag{2.1}$$

This formulation appears deceptively simple but finding an optimum $\mathcal{T}^*$ for the TSP is in general $\mathcal{NP}$-hard [Mazyavkina et al., 2020]. Still there are a number of applications for TSP formulations, e.g. in logistics, genome sequencing, drilling problems, data clustering, etc. [Applegate et al., 2007].

2.1.2 Exact Algorithms

Even though the TSP turns out to be computationally difficult, there are a number of approaches which solve the problem exactly.

There are $n!$ permutations of V, each inducing a different ordering in which cities are visited, but some yield equivalent tours. Still, with increasing n, a naïve brute force approach becomes infeasible very quickly. The Held-Karp algorithm [Held and Karp, 1962] lessens the exploding runtime by utilizing dynamic programming (DP). It defines $g(S, v)$ for $S \subset V$ and $v \in V \setminus S$ as the shortest path from the depot to v going through exactly every vertex in S in some order. Observing that $g(S, v) = \min_i g(S_i, v_i) + w(v_i, v)$ with $S = S_i \cup \{v_i\}$ allows the exploitation of the problem's recursive nature. Finally, the minimum tour cost is equivalent to $\min_u g(V \setminus \{u\}, u)$ and its actual tour can be found via backtracking through the DP-table. All-in-all, the Held-Karp algorithm decreases the asymptotic runtime from the super-exponential term of a naïve brute-force approach to $\Theta(n^2 \cdot 2^n)$ but also uses $\Theta(n \cdot 2^n)$ space.

The TSP can also be formulated as a mixed integer linear program (MILP), and there even is an MILP solver specialized on the TSP, called Concorde [Mazyavkina et al., 2020]. But this approach comes with limitations. For example the famous MILP formulation due to Chvátal et al. [2010] imposes an exponential number of so called subtour elimination constraints. Another well-known formulation, the Miller-Tucker-Zemlin formulation [Miller et al., 1960], only calls for n^2 number of extra constraints. Instead, big-M constraints are utilized which often suffer from worse relaxations.

2.1.3 Heuristics

Heuristics produce solutions that are not optimal in general but are reasonable in practice, and usually offer a substantially better runtime than exact algorithms.

We can differentiate two kinds of heuristics: First, a *constructive* approach builds a solution step-by-step starting from scratch. In each step some parts of a potential solution is either included or excluded due to some (often locally optimal) criterion. For the example of the TSP, one might consider the following procedure: Start at the depot, travel to the city that is closest to the current location and has not been visited yet, repeat until every city has been visited, then return to the depot.

Second, the *local search* approach starts with some feasible solution, which might be randomly sampled or computed otherwise with some heuristic. This initial solution is assumed to be suboptimal, and the goal is to improve it step-by-step. The word *local* implies that each iteration only considers some small subset of other feasible solutions, which is usually called the neighborhood of the initial solution. The elements of this neighborhood can be computed efficiently by modifying the given solution in some specified way.

For the TSP, the most well-known local search heuristic is the so-called *two exchange*, or *2-opt* technique: In each step, the solution is modified by selecting two non-incident edges of the tour, which are then swapped out for two other edges. Since there is only one way to reconstruct another valid tour once these two edges have been removed, this technique is arguably the simplest local search technique for the TSP.

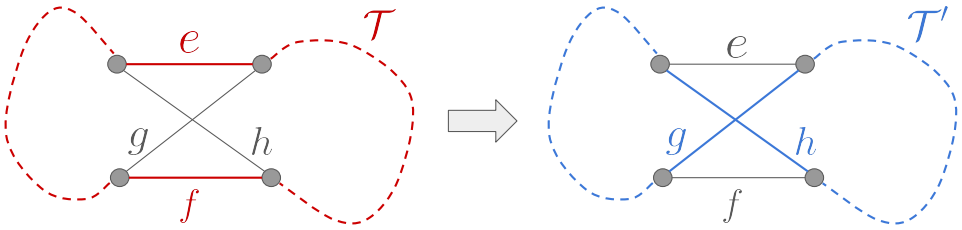


Figure 2.1: Schematic of a two exchange. Swapping edges e, f in tour $\mathcal{T}$ for edges g, h leads to new tour $\mathcal{T}'$. The cost difference is $C(\mathcal{T}') - C(\mathcal{T}) = w(g) + w(h) - w(e) - w(f)$.

2.1.4 Approximation Algorithms

Approximation algorithms are similar to heuristics in the sense that they generally produce suboptimal solutions but admit reasonable runtimes even for larger graphs. In contrast to heuristics, they come with some guarantees about how good the found solution is. Usually, this is quantified using the so-called approx-

imation factor α. An α-approximation algorithm outputs a solution that is at most α times worse than the optimum.

For the metric TSP, there exists a famous 2-approximation algorithm, that works as follows: First, construct a minimum spanning tree (MST) over the input graph. Then double each edge in the MST, resulting in a closed walk. Finally, take short-cuts by connecting repeated vertices of the closed walk directly. The cost of the MST is guaranteed to be smaller than the optimal TSP tour. Additionally, short-cutting will not increase the closed walk's length, if edge weights adhere to the triangle inequality, but will remove repeated vertices until a valid TSP tour is found. Hence, $\alpha = 2$.

The algorithm proposed by Christofides [1976] also employs an MST-based approach, but is more advanced. Still, it employs the short-cutting mechanism. Overall, it admits polynomial runtime coupled with an improved approximation factor of 3/2.

Ultimately, both of these algorithms can guarantee their approximation factors only for TSP instances with symmetric edge weights which adhere to the triangle inequality. Technically, their procedures can still be applied to more general graphs but found tours may be worse than α times the optimum.

2.2 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm, where an agent's goal is to maximize its cumulative reward when acting in some given environment. These RL environments are modelled using Markov Decision Processes (MDP) [Bellman, 1957].

A MDP is a tuple (S, A, T, R). S is the *state space* and A the *action space.* Taking action $a \in A$ in space $s \in S$ leads to a change in to a new state $s' \in S$. Such changes are modeled by the *transition function* T where $T(s'|s, a) = \Pr[s'|s, a]$, i.e. the probability of reaching s' through a when in s. In our case, T is deterministic. A (s, a, s') triple is called a transition. Finally, $R : S \times A \mapsto \mathbb{R}$ is a *reward function*, assigning a value $R(s, a)$ to taking an action a when being in state s.

The goal of reinforcement learning is to find a policy function $\pi : S \mapsto A$ such that the *cumulative reward*

$$\mathbb{E}\left[\sum_{t=0}^{H} \gamma^t R(s_t, a_t)\right] = \mathbb{E}\left[\sum_{t=0}^{H} \gamma^t R(s_t, \pi(s_t))\right]$$

is as large as possible. The *discount factor* $0 < \gamma \leq 1$ indicates how much the agent prioritizes short-term gains in the reward. The number of summands in the cumulative reward is bounded by *horizon* H.

2.2.1 On-policy Methods

In the *policy-based*, or *on-policy*, approach, the agent's policy is represented explicitly by $\pi_\theta(a|s)$ to determine the probabilities of taking action a in state s. In this project, this function will be represented by a neural network with weights θ.

The goal of policy-based reinforcement learning is to optimize θ to obtain as much cumulative reward as possible. This is done by gradient-ascent of the following *utility function*:

$$\begin{aligned} U(\theta) :&= \mathbb{E}\big[\sum_{t=0}^{H} \gamma^t R(s_t, a_t)|\pi_\theta\big] \\ &= \mathbb{E}\big[R(\tau)|\pi_\theta\big] \\ &= \sum_{\tau \in (S,A)^{\times(H+1)}} R(\tau) P(\tau|\pi_\theta) \end{aligned} \tag{2.2}$$

where we have used the letter τ to denote an entire *trajectory* or *rollout* of state-action pairs. And $R(\tau)$ denotes the cumulative discounted rewards, also called the *return*.

Current methods for on-policy RL can be grouped into two categories: 1) policy-gradient algorithms, and 2) trust-region methods.

Policy Gradient Methods Given the utility function $U(\theta)$ from Equation 2.2, policy gradient methods [Sutton et al., 1999] are based on the following gradient estimation:

$$\begin{aligned} \nabla_\theta U(\theta) &= \sum_{\tau \in (S,A)^{\times(H+1)}} R(\tau) \nabla P(\tau|\pi_\theta) \\ &= \sum_{\tau \in (S,A)^{\times(H+1)}} P(\tau|\pi_\theta) R(\tau) \frac{\nabla P(\tau|\pi_\theta)}{P(\tau|\pi_\theta)} \\ &= \mathbb{E}_{\tau \sim \pi_\theta}\big[\nabla_\theta \log P(\tau|\pi_\theta) R(\tau)\big] \end{aligned}$$

Note that this expectation can be approximated by simply sampling rollouts according to the current policy π_θ and averaging the bracketed expression. Due to the log, we can reformulate:

$$\nabla_\theta U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\big[\sum_{(s_t,a_t)\in\tau} \nabla_\theta \log \pi_\theta(a_t|s_t) \underbrace{R(\tau)}_{:=\hat{A}_t \text{ advantage est. of } (s_t,a_t)}\big]$$

Hence, the gradient can be computed w.r.t. the log-probability output of the policy network π_θ.

Note that most methods employ a modified version of this formula where the advantage estimate for a given state-action pair only includes the rewards of subsequent actions, as opposed to the whole trajectory. Another practical modification involves per-timestep normalization of the advantage estimates.

While this basic policy gradient method enables gradient ascent of the utility function U, more modern approaches have been developed in recent years. These methods involve even more sophisticated ways to estimate the advantage of a given state-action pair.

A popular choice are so-called *actor-critic* methods [Mnih et al., 2016]. They utilize an additional *value* or *critic* network, which is used to predict the returns that the policy will obtain starting from a given state. An even more general way to estimate the advantage has been introduced by Schulman et al. [2018], coined *generalized advantage estimation* (GAE). Since the critic network is used to determine the value of a given state, these estimates are less prone to high variance introduced by probabilistic environments. So these methods offer a bias-variance trade-off.

Trust Region Methods The methods introduced above suffer from a specific problem: Any data that is collected using the current parameters θ_t can only be used for one gradient ascent update since the sampling must be done w.r.t. π_{θ_t}. Trust region methods are based on a different derivation that avoids this sampling issue [Schulman et al., 2017a]:

$$\begin{aligned} U(\theta) &= \sum_{\tau \in (S,A)^{\times(H+1)}} R(\tau) P(\tau|\pi_\theta) \\ &= \sum_{\tau \in (S,A)^{\times(H+1)}} P(\tau|\pi_{\theta_{old}}) R(\tau) \frac{P(\tau|\pi_\theta)}{P(\tau|\pi_{\theta_{old}})} \\ &= \mathbb{E}_{\tau \sim \pi_{\theta_{old}}} \left[R(\tau) \frac{P(\tau|\pi_\theta)}{P(\tau|\pi_{\theta_{old}})} \right] \end{aligned}$$

This reformulation has two advantages:

1. The expectation can be approximated using old rollouts. Hence, we can reuse data for multiple updates.

2. Only the numerator is a function of θ, while the denominator is constant. This makes the expression suitable for backpropagation. Hence, this formulation of $U(\theta)$ can be used as a *surrogate loss*, which can be optimized using a plethora of optimizers.

In practice, there is one major drawback: Using rollouts collected by some policy

$\pi_{\theta_{old}}$ to update θ only makes sense in a small region around θ_{old}, called a *trust region*. There are various ways to address this challenge:

1. The trust region can be enforced using a set of *explicit constraints*. This approach is employed by the *Trust Region Policy Optimization* (TRPO) [Schulman et al., 2017a] algorithm, where it leads to a constrained optimization problem. This complicates the use of well-establish gradient descent based optimizers such as Adam [Kingma and Ba, 2017].

2. The trust region can be enforced implicitly using a *regularization* term. This approach is used by one variant of the *Proximal Policy Optimization* (PPO) [Schulman et al., 2017b] algorithm.

3. The trust region can be enforced implicitly by introducing *clipping* to the surrogate loss. This approach is used by a second variant of the PPO [Schulman et al., 2017b] algorithm.

Note that the latter two options enable the use of well-established optimizers to improve the policy network. Furthermore, most of the advantage estimation techniques that were presented in the previous paragraph can also be used for these methods.

Implementation When implementing any on-policy method, a number of choices must be made, e.g. the optimization algorithm or discount factors. The literature contains several works that provide an overview of these options and also attempt to find choices with decent performance in as many tasks as possible. The survey of Engstrom et al. [2020] provides such an overview for TRPO and PPO, while the work of Andrychowicz et al. [2021] tackles actor-critic methods

2.2.2 Off-policy Methods

While policy-based approaches try to optimize the parameters of a policy function, value-based approaches employ function approximation to predict the value of taking some action a in state s, denoted $Q(s, a)$. Formally, this so-called *state-action value* or *Q-value* for an actor acting according to policy π is defined:

$$Q^{\pi}(s, a) := \mathbb{E}\left[R(s, a) \mid s, a\right]$$

Additionally, for the optimal state-action value $Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$ the Bellman equation is satisfied:

$$Q^*(s, a) = \mathbb{E}_{s' \sim T(\cdot \mid s,a)}\left[R(s, a) + \gamma \max_{a' \in A} Q^*(s', a') \mid s, a\right]$$

Deep Q-Networks The Q-value can be approximated by a neural network Q_θ with weights θ, such a network is called a deep Q-network (DQN). The following loss L approximates the Bellman equation:

$$L(\theta) := \sum_{(s,a,s')\in\mathcal{D}} \frac{1}{2}\big(Q_\theta(s,a) - \underbrace{(R(s,a) + \gamma \cdot \max_{a'\in A} Q_{\theta'}(s',a'))}_{target}\big)^2 \tag{2.3}$$

where $Q_{\theta'}$ is an additional DQN, called the *target network*, with weights θ' to decouple computation of targets from estimated Q-values. $Q_{\theta'}$ is of the same shape as the *online network* Q_θ. Inference is performed on the online network by computing Q-value estimates $Q_\theta(s,a)$ for all actions a in a state s and then taking action $a^* = \max_a Q_\theta(s,a)$ to reach the next state.

The state-action-state triples are stored in some database $\mathcal{D}$, called the *replay memory*. This database is key to successful applications of Q-learning. The major challenges are as follows:

1. Given the current Q-value estimate, how can we collect interesting rollouts? While using the greedy policy to select $\text{argmax}_{a\in A}\, Q(s,a)$ when in state s yields the highest return (assuming the Q-value estimates are accurate), there might be a benefit to *exploration* by sampling a random action some times.

2. How do we sample from all the data we have collected? The formula above shows a uniform weighting over the full data set, but not all samples have the same importance. There may be a lot of actions with an average reward, but actions that result in significantly higher or lower reward are usually more relevant for training good policies.

Modern Off-Policy Methods Over the years a number of extensions to the classical DQN algorithm have been proposed. Because the literature is very extensive and rapidly evolving, we want to introduce the works that are relevant for our project.

The work of Hessel et al. [2017] surveys common extensions to the classical DQN algorithm and thoroughly tests them. While their experiments are conducted on the Atari 2600 benchmark suite [Bellemare et al., 2013], the findings are still of interest and help to discern which extension may or may not increase performance. Four of these extensions, each shown to improve results in most of the authors' experiments are the following:

1. ***Double Q-Learning:*** Vanilla Q-Learning suffers from potentially harmful overestimation bias. This problem is lessened by using the target network $Q_{\theta'}$ only to evaluate target Q-values. Selecting the maximizing action is

done by the online network Q_θ. Hence, the Double Q-Learning target is of the form:

$$R(s,a) + \gamma \cdot Q_{\theta'}(s', \underset{a' \in A}{\operatorname{argmax}}\, Q_\theta(s', a'))$$

2. ***Priority-based Replay:*** Conceptually, one may expect that Q-value estimates that are far off of the their corresponding target value are more relevant for training. E.g. significantly under or overestimating an action tells more in which direction the optimization should move rather than approximately correct estimates. Thus, instead of sampling uniformly random from the replay memory, a priority-based approach samples a transition (s, a, s') proportional to its *TD error*:

$$\left| Q_\theta(s,a) - (R(s,a) + \gamma \cdot \max_{a' \in A} Q_{\theta'}(s', a')) \right|^\alpha$$

The hyperparameter α gives additional control over the actual underlying sample distribution. Note that the TD error is already partially computed in the optimization step once a transition has been sampled, allowing for a more efficient implementation.

3. ***Dueling DQN:*** Based on the state-action function we can define the *value function* $V^\pi(s)$, which estimates the quality of a certain state s according to policy π, and the *advantage* $A^\pi(s,a)$, a measurement of the relative importance of an action a in state s:

$$V^\pi(s) := \mathbb{E}_{a \sim \pi(s)} \left[Q^\pi(s,a) \right]$$
$$A^\pi(s,a) := Q^\pi(s,a) - V^\pi(s)$$

A dueling DQN, instead of directly producing a state-action value estimate, splits its computation into two streams: one produces an estimate $V_\xi(s,a)$ for the value function, and the other streams models $A_\psi(s,a)$ as an advantage estimate. Finally these two streams are combined. A simple addition between the two estimate results in a problem of unidentifiability [Wang et al., 2016], hence the combination is slightly more involved:

$$Q_\theta(s,a) = V_\xi(s,a) + A_\psi(s,a) - \frac{\sum_{a' \in A} A_\psi(s,a')}{|A|}$$

4. ***n-step Learning:*** In the loss defined in equation 2.3 only a single step was considered when bootstrapping. Increasing the number of considered steps to a sliding window of n transitions may lead to faster learning [Hessel et al., 2017]. To that end define the truncated n-step reward R_n and an

alternate loss L':

$$R_n(s_t, a_t) := \sum_{i=1}^{n} \gamma^{i-1} R(s_{t+i}, a_{t+i})$$

$$L'(\theta) := \sum_{\tau_t^n \in \mathcal{D}} \frac{1}{2} \big(Q_\theta(s_t, a_t) - R_n(s_t, a_t) + \gamma^n \cdot \max_{a' \in A} Q_\theta(s_{t+n}, a')\big)^2$$

The database $\mathcal{D}$ now keeps track of n consecutive action-state pairs at each timestep t and replays them in the form of a partial trajectory $\tau_t^n = s_t, a_t, s_{t+1}, a_{t+1}, \ldots, s_{t+n}, a_{t+n}$.

2.2.3 Masked Policies

An issue that often arises in practice when applying RL methods are illegal actions: Some actions from the fixed action space A may not be legal during certain states of the environment. While it is in theory possible to force a policy or Q-network to learn these illegal actions by introducing some penalty or similar, it is often more efficient to mask the network outputs in a suitable manner. This simplifies the reinforcement learning task, since the distinction of legal and illegal actions does not have to be learned.

While the masking technique has been well-established in practice, recently Huang and Ontañò [2022] have provided a theoretical justification for it, at least for policy-based methods.

2.3 Graph Neural Networks

A Graph Neural Networks (GNN) is a type neural network that is well-suited for graph problem. GNNs are based on the concept of *graph convolutions*, or *message passing*: A graph convolution effectively consists of a communication round of a synchronous distributed system, where the messages are chosen to be vectors of some dimension. The computation performed on these messages is done by some form of neural network. In this design, two aspects must be highlighted:

1. A node aggregates the messages of its neighbors in a *permutation invariant* manner.

2. All nodes share the same parameters, hence the same copy of neural network is running on every node.

These two properties lead to a parameterized function that is equivariant under *graph isomorphism*, and can be optimized in much the same way as a classic neural network.

Another advantage of using GNNs as agent networks is that they are capable of handling variable sized graphs. This allows the usage of heterogeneous training sets, and more importantly, doing inference on graphs with a different number of vertices than train graphs. Additionally, various graph structures can be processed. Even though we restrict ourselves to undirected and symmetric TSP instances in this work, GNNs can also be applied to directed and asymmetric input graphs.

2.3.1 Mathematical Formulation

Each vertex v of a graph $G = (V, E)$ has an associated feature vector $x_v \in \mathbb{R}^d$. Analogously, edge feature $\mu_e \in \mathbb{R}^p$ encodes properties of edge e. A graph convolution consists of processing and aggregating the hidden vectors of all neighbors. There are several equivalent ways to express this, however we present here the conventions introduced by Paszke et al. [2019], for the PyTorch Geometric package:

$$x_v^{k+1} = f^k \left(x_v^k, \bigoplus_{u \in N(v)} g^k(x_v^k, x_u^k, \mu_{(v,u)}) \right)$$

Messages from adjacent vertices are aggregated with a permutation invariant function $\bigoplus$, e.g. a sum or an average. f^k and g^k might be neural networks, with parameters to optimize, or otherwise fixed differentiable mappings. Message passing is usually performed over multiple rounds of communication, hence terms corresponding to different iterations are indicated with a superscript k. The first hidden vector is usually chosen to be the node feature vector: $x_v^0 := x_v$.

Chapter 3

Related Work

In the literature, there are several works tackling the TSP or other CO problems using reinforcement learning. Some of these also employ GNNs. In this chapter we will highlight the ones that we have found most useful when attempting this project. Furthermore, we also introduce some relevant works in the rapidly evolving field of GNNs.

3.1 Reinforcement Learning for Combinatorial Optimization

Reinforcement learning has been successfully applied in order to solver various CO problems. The work of Mazyavkina et al. [2020] surveys a number of different publications about RL in combinatorial optimization. It highlights the diversity of methods that are employed: For TSP one can find approaches that use Pointer Networks [Bello et al., 2016], LSTMs [Chen and Tian, 2018], or GNNs [Cappart et al., 2020] in combinations with either policy-based or values-based methods.

3.1.1 Approaches Using GNNs

One of the earliest works by Dai et al. [2018] that claims to get results competitive with good heuristics utilizes a GNN architecture called *Structure2Vec* to parameterize Q-value functions. A n-step Q-learning algorithm (see Subsection 2.2.2) is used to train the neural network architecture. The authors employ a constructive approach: nodes correspond to actions and in each step the action with the largest Q-value gets added to the partial solution, which is initially empty. A particular way of inserting the chosen node v is used: instead of just appending v, it is inserted at the best possible position in the partial solution tour. This technique is called the *best insertion helper*.
We found it hard to find any success with Structure2Vec. The authors disclose very briefly that they flip the sign for their proposed reward, which to our understanding would result in maximization of the selected edge weight. Additionally,

they suggest a really small discount factor of $\gamma = 0.1$. Discount factors usually lie in the range $[0.9, 1.0]$ [Schulman et al., 2018]. A value of 0.1 means the actor highly priorities short-term gains. This is leads us to the suspicion that their approach trains an optimizer which is similar to a *greedy max insertion* heuristic: always choosing the edge which is furthest from the current tail of the partial solution, and inserting it at the position that minimizes the total cost of the tour. A closer inspection of the best insertion technique reveals that it likely does a lot of heavy lifting with respect to the reported performance. Indeed as we will later see in Section 5.2, another heuristic of just choosing a random node that is not yet in the partial tour and employing the best insertion helper when adding it, generally already performs really well.

A local search solver based on the two exchange technique was presented by de O. da Costa et al. [2020]. They employ a policy network that contains some graph convolutional layers. However, they rely on a basic message passing layer that only supports edge weights as opposed to multi-dimensional edge attributes. We will later see that this use of multi-dimensional edge attributes is key in our approach for representing which edges are part of the tour, as well as their orientation. Furthermore, it enables us to use a positional encoding of edge weights. Since the authors cannot rely on these mechanisms they also use a recurrent neural network and represent the tour as a sequence.

3.2 Actor Network Design

The performance of RL approaches can be limited by the design of the trained actor. Since we opt to employ a GNN based actor, we present here key advances in the area of graph convolution designs. Our approach relies on multi-dimensional edge attributes to represent the current status of the TSP tour for the actor. Hence, we will first discuss a selection of graph convolution layers that have support for such edge attributes, and also introduce a promising feature engineering technique to extend these edge attributes.

3.2.1 Graph Convolution Layers and Edge Attributes

The design space for graph convolutional layers is much larger than that of e.g. image convolution. As such, the literature contains a plethora of publications introducing specific layer designs that are suited to various different situations. We are interested in graph convolutional layers with support for multi-dimensional edge attributes, since these are a key part of our observation spaces. We want to highlight the following designs that fulfill this prerequisite:

- *Graph attention* (GAT) networks, introduced by Veličković et al. [2018], use any given edge attributes in the computation of the attention score that is

used when aggregating the messages of neighboring nodes. This design was extended by Brody et al. [2022] to be slightly more expressive by reducing parameter sharing. This version is known as GATv2.

- *Graph isomorphism networks* (GIN) were introduced by Xu et al. [2019]. They do not rely on attention, but instead simply aggregate all the messages of neighboring nodes before passing the combined vector into a standard MLP. This architecture was later extended by Hu et al. [2020] to also support edge attributes. These edge attributes are aggregated with the node messages after a linear transform, which equalizes the dimensionality.

- *Pathfinder discovery networks* (PDN), introduced by Rozemberczki et al. [2021], perform an element-wise multiplication of neighbor messages with the corresponding edge attributes. This is done after both have been transformed by two seperate MLPs (of equal output dimension), after which they are aggregated.

- Another layer with natural support for multi-dimensional edge attributes was introduced separately by Simonovsky and Komodakis [2017] as *edge conditioned convolution* on the one hand, and Gilmer et al. [2017] on the other. This layer type is best known as NN convolution. It is similar to the PDN convolutional layer, but instead of performing an element-wise multiplication, it uses the edge attributes to dynamically generate a matrix for each edge. This matrix is then used to linearly transform the incoming message of the neighbor before aggregation.

3.2.2 Fourier Features for Low-Dimensional Inputs

When attempting to solve the TSP, the key attribute of a graph is its edge weights. The agent must learn to recognize which selection of edges yields a tour of small cost. Deciding whether to include a certain edge in the tour over some other edge will likely involve some form of numerical comparison between the two edge weights. Such comparator functions have significant high frequency contents and are thus difficult to approximate well with neural networks.

To enable neural network to learn high frequency functions, Tancik et al. [2020] propose the use of Fourier features, consisting of sinusoids of increasing frequency. They successfully apply this technique to coordinate representations (e.g. to learn image functions) and other low dimensional inputs. Since the weight of a given edge is effectively a one-dimensional attribute, such Fourier features may be an effective representation.

Chapter 4

Our Approach

This chapter specifies how exactly our proposed RL agent interacts with a TSP environment. In general, this includes what the agent sees of the TSP, the *observation space*, how it can act upon the environment, the *action space*, and what its end goal is, the *reward*. Further, the proposed GNN architecture is outlined.

4.1 Environments for TSP

The input for a TSP problem instance is a graph $G = (V, E)$ and a weighting function $w : E \mapsto \mathbb{R}$. As previously mentioned in Section 2.3, the GNN input is given as node features $x_v \in \mathbb{R}^d$ and edge features $\mu_e \in \mathbb{R}^p$ for vertices v and edges e. The TSP usually considers complete graphs, i.e. all edges are present. Since the edge weights contain all the relevant information, it is key that we represent them in a way that allows the RL agent to learn a high-quality heuristic. We speculate that good performance on the TSP task requires learning high-frequency functions of these edge weights, e.g. comparator functions. Because of this, we opt to represent the weights as a vector of Fourier features [Tancik et al., 2020], consisting sines and cosines with increasing frequencies.

4.1.1 A Constructive Environment

A constructive environment includes a partial tour $\mathcal{T}$, either represented as an ordered set of vertices or as an (unordered) set of edges. Hence, taking an action corresponds to selecting a node or an edge respectively. We consider two types of initialization: *empty initialization* where $\mathcal{T} = \emptyset$, or *depot initialization*, i.e. we start with a predefined depot node v_0, $\mathcal{T} = \{v_0\}$. See Figure 4.1 for a demonstration of a trained agent acting in this environment.

Observations and Selection Strategy Consider a node selection approach. Based on the state we derive four observations encoded in the input node features, thus $x_v^0 \in \mathbb{R}^4$.

1. An indicator ($\in \{0, 1\}$) to tell whether v is currently in $\mathcal{T}$.
2. The normalized position of v in $\mathcal{T}$ if v is in $\mathcal{T}$, otherwise 0.
3. The normalized length of $\mathcal{T}$ as number of hops.
4. The length of $\mathcal{T}$ as sum of weights.

Note that observations 3 and 4 are identical for all $v \in V$. When employing the best insertion helper two additional observations, that would otherwise be redundant, are added.

5. The normalized step when v was added to $\mathcal{T}$ if v is in $\mathcal{T}$, otherwise 0.
6. The normalized position where v would be added to $\mathcal{T}$ if v was selected as the next action.

On the other hand, an edge selection approach would look significantly different. Conceptually, choosing actions to be the decision which edge should be added to $\mathcal{T}$ gives the actor more freedom.

GNNs as described in section 2.3 produce an output per node. One way of adapting to that is to work on the line graph $L(G)$ of G. Each edge in G is represented as a vertex in $L(G)$. There is an edge between two vertices in $L(G)$ if the corresponding edge share an endpoint in G.

The original edge weights are now nicely mapped to vertex features and the line graph can be considered to be unweighted with respect to its edges. However, in TSP we often consider G to be a complete graph K_n and its line graph $L(K_n)$ is significantly larger: it contains $m = \binom{n}{2} \in \mathcal{O}(n^2)$ vertices and $m(n-2) \in \mathcal{O}(n^3)$ edges.

There are other ways of designing an edge selection pipeline as we will see in Subsection 4.1.2. Yet, defining $\mathcal{T}$ to be an edge set comes with another drawback that is more fundamental. For efficient training, the state needs be updated relatively fast. For node selection, identifying the illegal actions was straightforward, here it is more elaborate. Any edge e not yet chosen should be considered an illegal action if e closes any cycle in $\mathcal{T}$. Hence, choosing $n-1$ valid edges implies that $\mathcal{T}$ is a path visiting all nodes. At this point, we can stop the procedure and add the edge which connects the path's endpoints to get a valid tour. Checking whether an edge closes a cycle can be done using an union-bound data-structure but the procedure still needs to iterate over every edge at least once, whereas for node selection setting a node to be an illegal action given its identifier only takes constant time. This results in a significant runtime difference when it comes to training.

Reward Both node and edge selection result in conceptually adding an edge to the partial solution. Hence, the reward is just the negative weight of that

edge. Negative because RL methods maximize reward which should translate to minimizing the cost of the found tour. Two special cases have to be considered: For the last added node or edge we also reward the negative weight of the edge that closes the cycle by connecting the first and last vertex. Secondly, when employing the best insert helper, a slight modification is needed: when a vertex u gets inserted between v_i and v_{i+1} already in the tour the reward is $-(w(v_i, u) + w(u, v_{i+1}) - w(v_i, v_{i+1}))$ instead. With that the final cumulative reward will correspond to the negative cost of the output tour.

4.1.2 A Local Search Environment

A local search environment keeps track of the current feasible solution, which is then modified by the agent in some number of steps. For the TSP, this feasible solution is a *complete tour* $\mathcal{T}$. The updates are the previously explained two exchange steps. See Figure 4.2 for a demonstration of a trained agent acting in this environment.

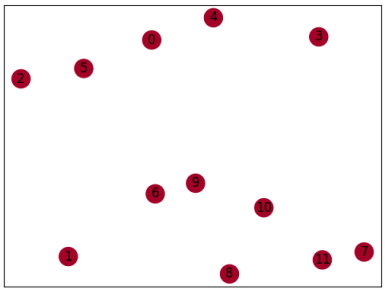
Reward Every time the agent performs a two exchange update, the reward is calculated to be the improvement of the tour length that this modification produces, i.e. the weight difference of the two removed edges and the two added edges. Note that this reward can be negative if the performed two exchange step increases the length of the tour.

Selection Strategy Since we employ the two exchange technique for the local search, the agent must select two non-incident tour edges to perform such a search step. There are different ways to design the action space A to enable the agent to perform this selection:

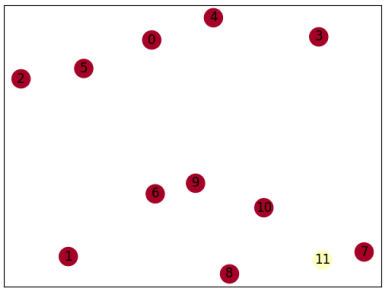
1. The agent selects a *node* in every step, hence $A = V$. With suitable masking, the agent will have selected four nodes (v_1, v_2, v_3, v_4) from V that define two non-incident tour edges after four steps. The environment then swaps out the edges $\{v_1, v_2\}, \{v_3, v_4\}$ according to the two exchange rule.

2. The agent selects a *tour edge* in every step, hence $A = E_{\mathcal{T}}$, the set of tour edges. Since GNNs naturally produce one output per node, we can employ a bijection $f_{\mathcal{T}} : V \to E_{\mathcal{T}}$, where $f_{\mathcal{T}}$ maps any node v to the outgoing tour edge at v. Hence, we can simplify this case to $A = V$, but we must (arbitrarily) fix the orientation of the tour edges and then represent this orientation in the observation.

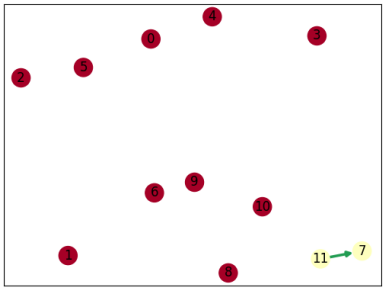
3. The agent selects a *pair of non-incident tour edges* in every step, hence $A \subset E_{\mathcal{T}} \times E_{\mathcal{T}}$. We can employ the same simplification as before using $f_{\mathcal{T}}$, so the agent must effectively select a pair of nodes in each step.



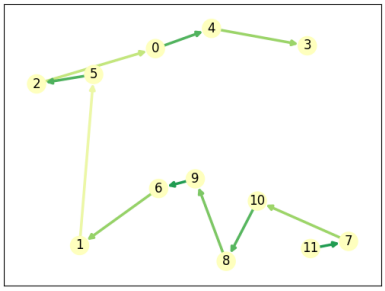
(a) Initial observation: The constructive environment starts with an empty tour by default.



(b) Step 1: The first node is added to the tour. The tour contains no edges yet, hence the reward of this action is 0.



(c) Step 2: A second node is added to the tour. The agent receives the negative edge weight as reward.



(d) Final state: The agent has added the last node. The final edge is missing in this visualization, but is considered in the reward calculation.

Figure 4.1: Demonstration of the constructive environment: A trained agent acts in the constructive environment, successively adding nodes to the tour. Note that only the tour edges are shown, while the graph topology for the GNN is always the same, namely a clique. Node states are represented with colors, the edge colors indicate the cost. These figures were rendered with the networkx package [Hagberg et al., 2008].

Note that each of these environment implementations has both benefits and drawbacks: From point 1 to 3, the observation and action spaces become successively *more complex*, however the reward becomes *less sparse* since fewer environment steps are necessary to perform a two exchange update.

Observations The environment must represent the current state of TSP in a way suitable for the policy or Q-value GNN. Naturally, this representation is a graph. It contains all the edges of the original graph (including the tour edges). Furthermore, there is a feature vector associated with each node and each edge.

The edge features are as follows:

- The tour orientation of the edge: Either an edge is not part of the tour at all, or it is either an incoming or outgoing edge. Note that the graph representation required by the GNN contains only *directed* edges. So every edge is represented by two directed edges. This also means that every tour edge has an incoming and an outgoing copy.
- The positional encoding of the edge cost.

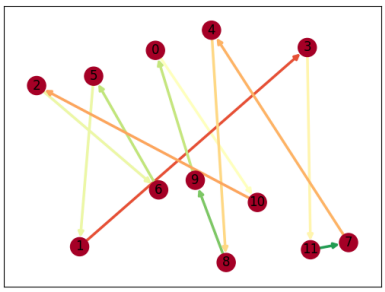
The node features are as follows:

- The status of the node: Depending on the selection strategy, there are between one and three node statuses. E.g. for the node selection strategy, the node status shows if the node has been selected to be part of the next two exchange step, and whether a neighbor has already been chosen or not.
- A scalar value in $[0, 1]$, indicating how many local search steps remain until the environment terminates.
- Optionally, the positional encoding of the incoming and outgoing tour edge length. By default this is turned off.

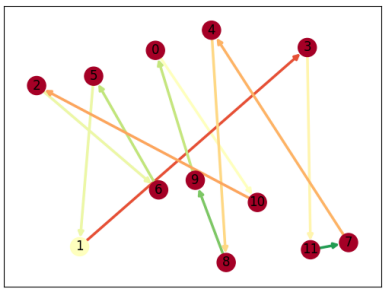
4.1.3 Masking Implementation

When performing local search or constructive environment steps, only a subset of the actions from the fixed action space A are actually valid. Consider for instance the constructive approach: Once some subset of nodes has already been added, these nodes cannot be selected again.

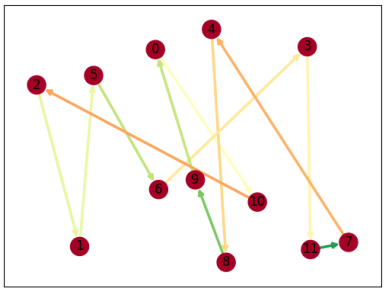
Another concrete scenario can be given for a two exchange environment when performing local search: Assume the policy selects a tour edge in every step, so a two exchange update occurs every two steps. Since the selected tour edges mustn't be incident (or identical) to each other, only a subset of the tour edges are valid actions in the second step.



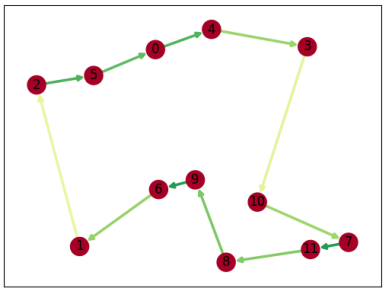
(a) Initial observation: In the local search environment, a tour is randomly sampled to serve as the initial feasible solution.



(b) Step 1: The agent has selected node 1, shown with the color change. The nodes act as representatives for their outgoing tour edge. Hence, the agent has selected edge $(1, 3)$ for the next two exchange. Now, the agent has to select a second node to perform a two exchange update. Since no two exchange update has taken place, the reward of this step is 0.



(c) Step 2: The agent has selected node 2, which represents the tour edge $(2, 6)$. The environment performs a two exchange update, removing edges $(1, 3)$ and $(2, 6)$, and adding edges $(2, 1)$ and $(6, 3)$. The next observation already shows the updated graph, with reset node states. The reward is calculated as the improvement in tour length.



(d) Final state: After n two exchange updates, the environment terminates. The resulting tour is much shorter than the initial tour, because we use a trained agent for this demonstration.

Figure 4.2: Demonstration of the local search environment: A trained agent acts in the local search environment. Note that only the tour edges are shown, while the graph topology for the GNN is always the same, namely a clique. Node states are represented with colors, the edge colors indicate the cost. Because this figure shows the *tour edge* selection option, there is a two exchange update every two steps. These figures were rendered with the networkx package [Hagberg et al., 2008].

To support the agent in making valid decisions, all environments supply a mask vector with each observation, encoding the valid actions from A during this step. Our masking strategy is different for policy and Q-networks:

- A policy network will produce a vector of unnormalized log probabilities. For any illegal actions, this value is replaced by a suitably large negative number, such as -10^6. Furthermore, we opt to introduce a lower bound for the log probabilities of any legal actions by way of the ELU [Clevert et al., 2016] activation function. In combination, these two measures ensure that the probabilities of invalid actions are negligible after the softmax activation.
- Since Q-values are not bounded either above or below, we simply replace them by the smallest Q-value estimate minus some safety margin. Another issue arises when collecting rollouts with a random strategy (e.g. ε-greedy exploration). Here too, the invalid actions must be avoided.

4.2 GNN Architecture and Attention Layer

There are a plethora of message passing layers implemented in the PyTorch Geometric [Fey and Lenssen, 2019] package. Furthermore, the package enables an easy implementation even of complex message passing schemes. Our goal is to compare a selection of existing layers to each other, and also introduce a custom message passing scheme which we designed with the TSP optimization task in mind.

4.2.1 Architecture for Complete Graphs with Edge Features

While these message passing layers can simply be chained together similar to image convolutional layers, it is common to combine them with dense layers, which perform some computation on the node messages in between rounds of message passing.

We do this in a generic architecture that is agnostic to the type of message passing layer used. Our architecture can be seen as a chain of communication rounds, using different parameters but identical structure. A single such round is summarized in Figure 4.3.

The following design choices were made with the TSP optimization task in mind:

1. Since our input graphs contain edge features which are not available for loop edges, the common practice of adding self loops to each node has been replaced by a skip layer.

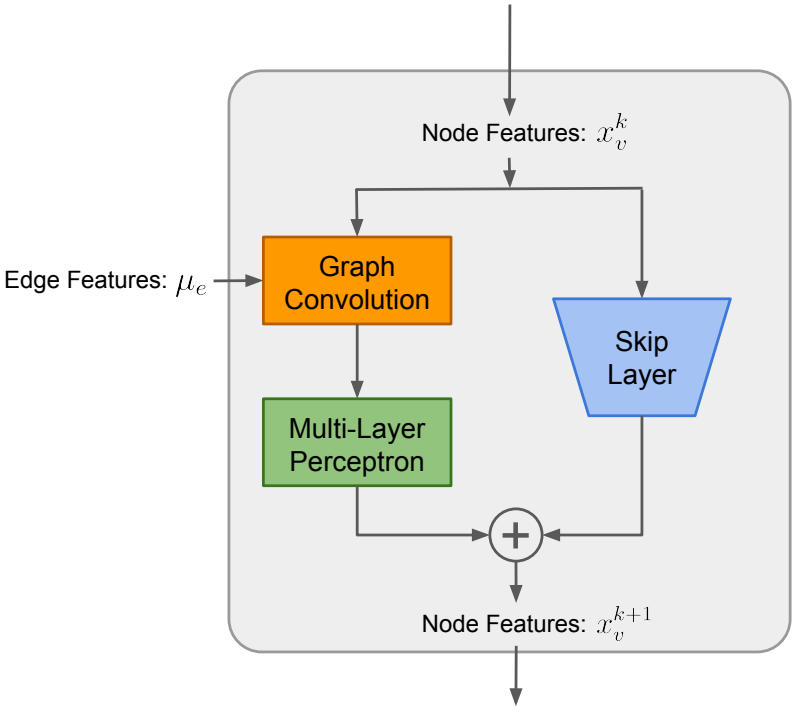


Figure 4.3: Schematic of the proposed generic architecture blueprint. Different types of layers can be plugged into the graph convolution layer box. Skip layers help to preserve local features between message passing rounds, similar to added graph loops, and multi-layer perceptrons add additional modeling power.

2. A considerable amount of computation is done in between the message passing layers. Since the TSP task involves complete graphs, information can propagate through the tolopology in few communication rounds. Our architecture contains a reasonable number of parameters, even when using few message passing layers.

4.2.2 TSPConv: an Attention-based Message Passing Layer

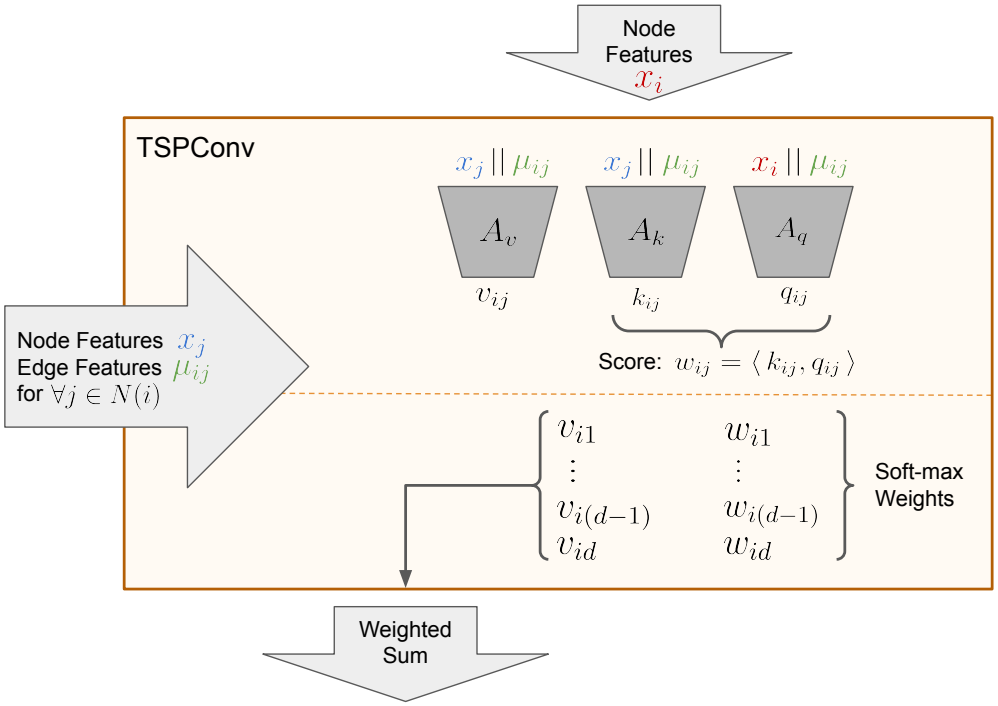


Figure 4.4: Attention mechanism behind TSPConv. The node features of a vertex i, the node features of its neighbours j and the features of the edges (i, j) connecting them produce values v, keys k and queries q. Which finally get aggregated and output as a weighted sum.

As described in section 2.3, a GNN layer can be seen as a message passing algorithm. Since the TSP is concerned with finding a certain subset of edges of a complete graph, we propose an attention-based GNN layer that takes edge features into account.

We introduce a simple dot-product attention layer using a softmax weighting strategy. Dot-product attention can be described in terms of keys, queries, and values, all of which are vectors obtained by a separate linear transform. Crucially, our layer uses not only the node features but also the edge features to compute all three of these vectors. We use the features of the sending node to compute the

keys and values, and the features of the receiving node for the queries. As usual, the attention mechanism is split into several heads. A single head is summarized in figure 4.4.

As mentioned above, we designed this layer with the TSP optimization task in mind: The goal was to enable a GNN to extract relevant edge features in an efficient manner. Since the TSP task involves dense graphs where most of the information is associated with edges rather than nodes, edges must be processed as efficiently as possible without ignoring their attributes completely. The proposed layer enables this by letting information from the edge attributes propagate into the node hidden vectors. There they can be processed more efficiently simply because there are much fewer nodes than edges.

Comparison to GAT Convolution Our design is directly inspired by the idea of graph attention networks [Veličković et al., 2018, Brody et al., 2022]. Notably, we include the edge attributes in the value computation, whereas the implementations of GAT and GATv2 that we are aware of only utilize the edge attributes in the computation of attention scores. Another smaller difference is the fact that GAT uses both sender and receiver hidden vectors in attention scores and values, though our design could certainly be extended in this way if the need were to arise.

Chapter 5

Results

In this chapter, we outline the experiments we performed and interpret the results we obtain. Due to the large design space of this project, we take a sequential approach: We tackle one or two design choices at a time while leaving all other settings at a reasonable default which we obtained during preliminary testing.

5.1 Experimental Setup

Before presenting the results of the different experiments, we introduce here the general approach.

5.1.1 Datasets

Even though there exists an often used TSP instance library called TSPLib [Reinelt, 1991], we heavily rely on randomized graphs for training and testing. TSPLib's main drawback is its irregular problem size distribution. The most graphs of the same size are six graphs of 100 nodes.

Using randomly generated graphs allows more flexibility when it comes to graph and dataset sizes. All graphs used are complete graphs, hence they can be specified by their weight (or distance) matrix. Two different procedures to generate this weight matrix $M_G \in \mathbb{R}^{n \times n}$ for a complete graph G are used:

1. *Euclidean*: For n nodes draw x and y coordinates from $\mathcal{U}(0, 1)$. Set $m_{i,j}$ to be the euclidean distance between node i and j. We will refer to these graphs also as *2d* graphs.

2. *Symmetric*: Draw elements $b_{i,j}$ of a temporary matrix $B \in \mathbb{R}^{n \times n}$ from $\mathcal{U}(0, 1)$, then let $M_G = 1/2 \cdot (B + B^T)$.

where $\mathcal{U}(0, 1)$ represents the uniform distribution over $[0, 1)$. As a last step the diagonal entries of M_G are set to zero, i.e. G should contain no loops.

Since our approach is completely agnostic to the embedding of the nodes into a geometric space, we use the symmetric non-euclidean graphs as a default. Unless otherwise specified, we use a train set of 64 graphs each with 32 nodes for training, and to evaluate a holdout set of the same dimensions. As such, experimental results are always a distribution of found tour lengths over the holdout set, shown as boxplots. Whiskers extend to the furthers sample in a range of 1.5 times the interquartile range below the first and above the third quartile respectively. Any datapoints outside the whisker and interquartile range are classified as outliers and shown individually as diamonds.

By now all TSPLib instances have been solved and their optimal tour costs are publicly available [Reinelt]. Hence, we will report optimality gaps for TSPLib graphs in subsection 5.2.6 to highlight generalization capabilities. But all models are exclusively trained on random graphs.

5.1.2 Baselines

We will show results from tests of up to 128 node graphs. Employing exact algorithms at this size becomes increasingly infeasible. Hence, results are compared against a number of baseline heuristics. These are:

1. **Two Approximation**: The well-known MST-based two-approximation algorithm, see Subsection 2.1.4.
2. **Greedy Minimum Hop**: Consider starting at an arbitrary node. From your current position take the minimal outgoing edge to an unvisited node. Repeat until all nodes have been visited. Finally, close the cycle by adding the edge from your last position to the starting node. This is the greedy minimum hop heuristic: locally minimal edges are greedily added until a tour was found.
3. **Minimum Insertion**: The minimum insertion heuristic also only considers minimum edges to unvisited nodes. The main difference is that the closest node from the current position is inserted at best possible position of the partial solution at the current timestep. The current position is always the last node in the sequence of visited nodes. Note that this corresponds to the previously introduces best insertion technique.
4. **Random Insertion**: To highlight the power of said best insertion helper we will also show found tour lengths of the random insertion heuristic. It's equivalent to minimum insertion except that a random node is chosen to be insertet instead of the closest one.
5. **Maximum Insertion**: Another variant of insertion heuristics. This variant always chooses the farthest node to be inserted at the best possible position.

In Section 5.2, experiments compare against all five baselines. For clarity some plots will only show the average tour length of the best performing baseline indicated with a dashed line.

5.1.3 Design Space Overview and Default Options

Our experiments attempt to cover the extensive design space consisting of TSP heuristic design (in the RL environment), the GNN based RL agent, and the actual RL method used. An overview of the options with the corresponding default is given below:

1. **RL method:** We test all methods from the Stable-Baselines3 [Raffin et al., 2021] package that support our action and observation spaces. These methods are: DQN [Bellemare et al., 2013], PPO [Schulman et al., 2017b], and A2C (a version of A3C [Mnih et al., 2016]). Furthermore, we test our own custom implementations of PPO, Dueling-DQN [Wang et al., 2016] trained with a 4-step double Q-learning algorithm with priority-based replay (implemented with a sum-tree data structure proposed by Schaul et al. [2016]), and an actor-critic policy gradient method based on generalized advantage estimation [Schulman et al., 2018]. The default is our custom PPO implementation, because it yielded consistent high-quality results. We use a default discount factor of 0.98, and train for a total of 500k timesteps.

2. **GNN architecture:** In Subsection 4.2.1 we introduced a generic GNN scheme that allows testing a number of different graph convolution layers. We tested the PyTorch Geometric [Fey and Lenssen, 2019] implementations *PDNConv* [Rozemberczki et al., 2021], *GATv2Conv* [Veličković et al., 2018, Brody et al., 2022], *GINEConv* [Hu et al., 2020], and *TSPConv* (ours, implemented as a custom layer type). We use our own custom layer as the default option. The default network architecture includes 3 communication rounds, and messages of dimension 64. Note that we also tested the *NNConv* [Gilmer et al., 2017, Simonovsky and Komodakis, 2017] layer type, but were unable to obtain reasonable results.

3. **TSP heuristic design:** We always test both the constructive and the local search environments. The constructive environment has the *best-insert* heuristic turned off by default. An additional option for the constructive approach is its starting set. By default it starts with a completely empty set. Alternatively, it can be initialized to start with an arbitrary starting node. The local search environment, based on two-exchange, uses the *tour edge* selection strategy by default and will reset to a random tour after n two-exchange steps. The default settings for the two environments are also shown in Figures 4.1 and 4.2, respectively.

Unless stated otherwise all testing parameters are set to their corresponding default value.

5.2 Experimental Results

In this section we show results from experiments that benchmark the previously mentioned design choices for reinforcement learning environments, but also different RL methods and different graph convolution layers. The results are structured in subsection and for the reader's convenience we provide a table of figures for this section:

5.2.1 Reinforcement Learning Algorithm

Choice of RL Method We have tested a large selection of RL methods for both tasks. While most of these yielded good results in some experiments, the main issue was stability and consistency. Most methods required several restarts to yield good results. We have shown the results of a typical comparison experiment in Figure 5.1.

Methods based on Stable-Baselines 3 are especially unstable in the aforementioned experiments. We identified the main mechanism to feed in multiple graphs as one major cause for variance, most likely due to normalization across the different input graphs. Indeed, limiting the training set to only one graph and reducing the number of total timesteps, paradoxically produces better results, see

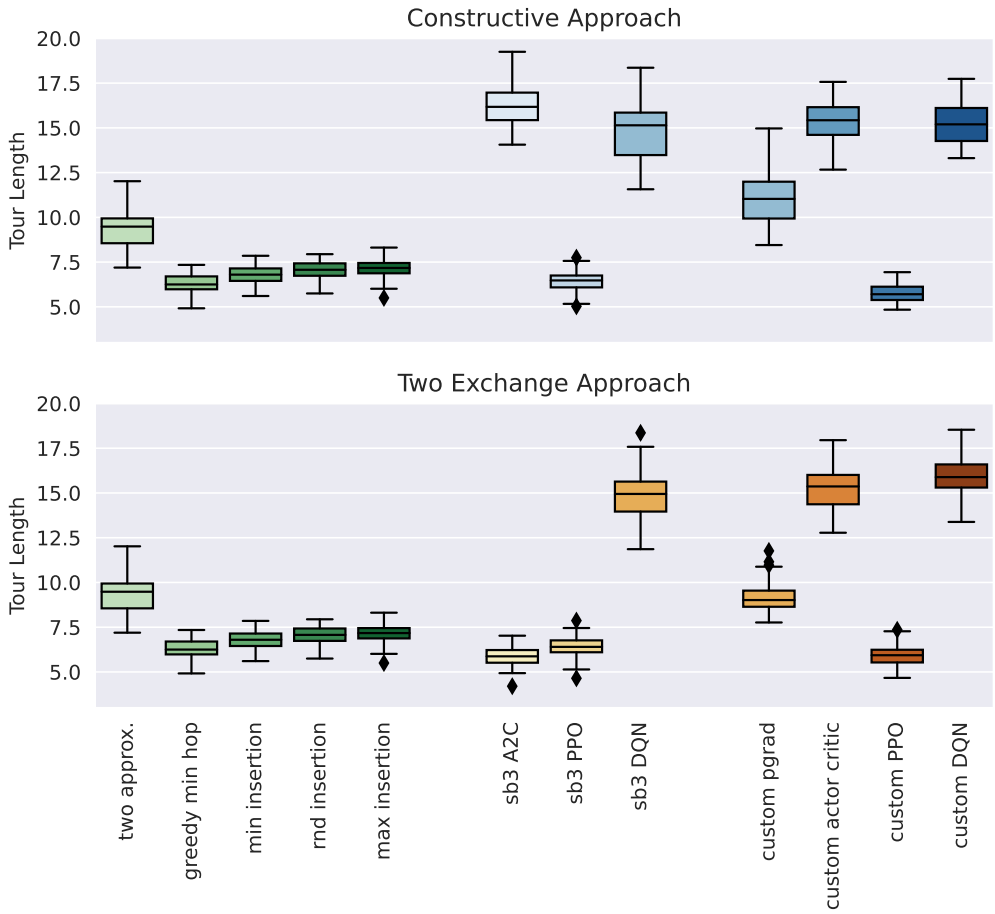
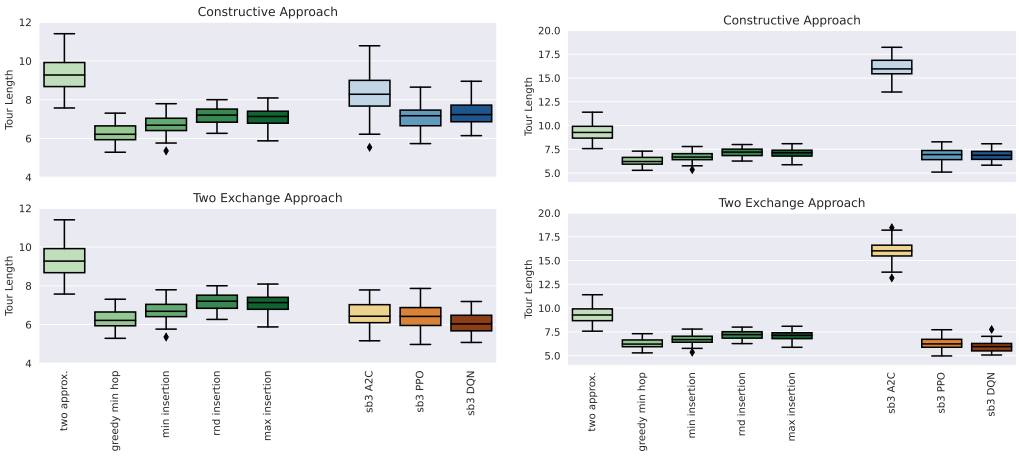


Figure 5.1: The comparison between reinforcement methods clearly filters out some implementation as non-competitive. The top plot shows the performance of constructive RL environments in a blue palette. In the bottom plot the two exchange RL approach's tour lengths in orange tones. Finally, the baselines in greens. The different solvers are indicated on the x-axis.



(a) 100k total timesteps **(b)** 250k total timesteps

Figure 5.2: RL methods based on the Stable-Baseline3 library perform better with less train set samples, here only one graph. The left plot shows the found tour lengths compared against our baselines after 100'000 total timesteps. On the right the performance after 250'000 total timesteps. Longer training leads A2C to diverge from previous better solutions.

Figure 5.2. But increasing the number of timesteps may already lead to some of the methods collapsing.

In general, the performance of each RL method seems to be independent of the choice of environment type. Two exceptions: Stable-Baseline3's A2C performs significantly better with two exchange, and custom policy gradient also slightly favors two exchange. A possible explanation for this is the fact that local search methods can succeed even with an inaccurate critic network: Given that the critic network is used to predict the returns beyond some partial rollout of k steps (where usually $k \approx 5$), even if this prediction is inaccurate, the reward from the k collected steps are enough to learn a greedy local search strategy.

Discount Factor Figure 5.3 shows a comparison between different discount factors used in the RL pipeline. As we can see, the two exchange method performs very similarly with a wide range of discount factors. In the local search setting it is possible to have local minima that require a long series of two exchange steps to find a better solution, however these long step sequences are very unlikely to be found during RL exploration. Hence, we might expect the actor to learn shorter, greedier, search sequences even with higher discount factors, which explains why the lower values do not affect performance.

For the construction approach, the discount factor plays a more important role: It performs better with higher discount factors, with the default choice of 0.98 being close to optimal. This might be explained by the fact that the

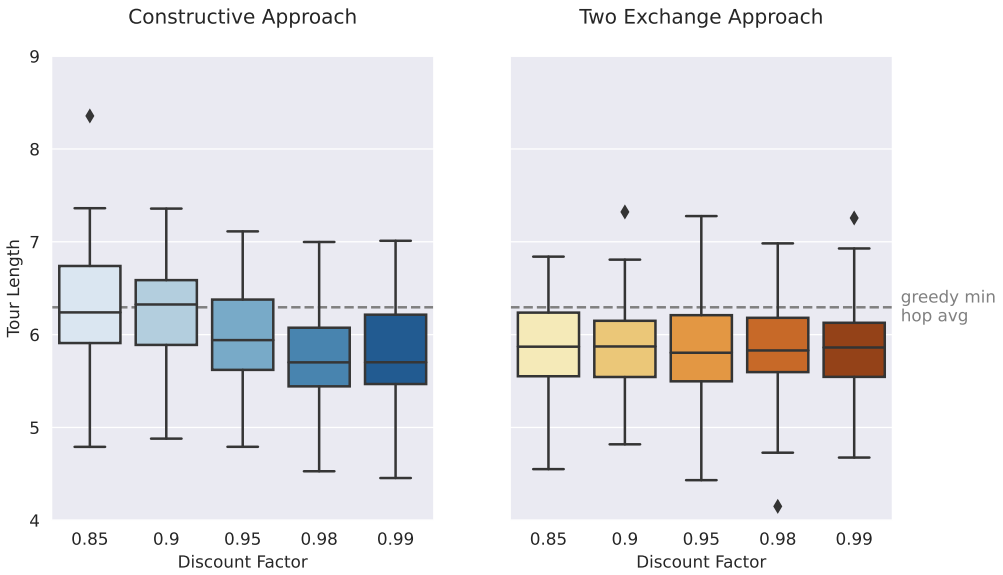


Figure 5.3: The choice of the discount factor has a larger impact on the constructive approach (left). Whereas, the two-exchange method's medians (right) outperform the best performing baseline's average for every tested discount factor.

construction approach will only yield decent results, if rewards from the later construction steps are also taken into account. Consider the later construction steps: The possible actions are severely constrained by the previous decisions. Hence, the only way to obtain good reward in these later steps is to choose suitable construction actions in the earlier parts of the construction.

Total Timesteps Figure 5.4 shows a comparison of different training times, in terms of total environment steps. This confirms our choice of 500k as a default, but also indicates that shorter training times are viable.

5.2.2 Actor Network

GNN Size We compare a number of different network sizes, varying the number of communication rounds and the size of the messages. The sizes are summarized below:

	S	M	L	XL
com. rounds	3	3	4	4
message size	32	64	128	256

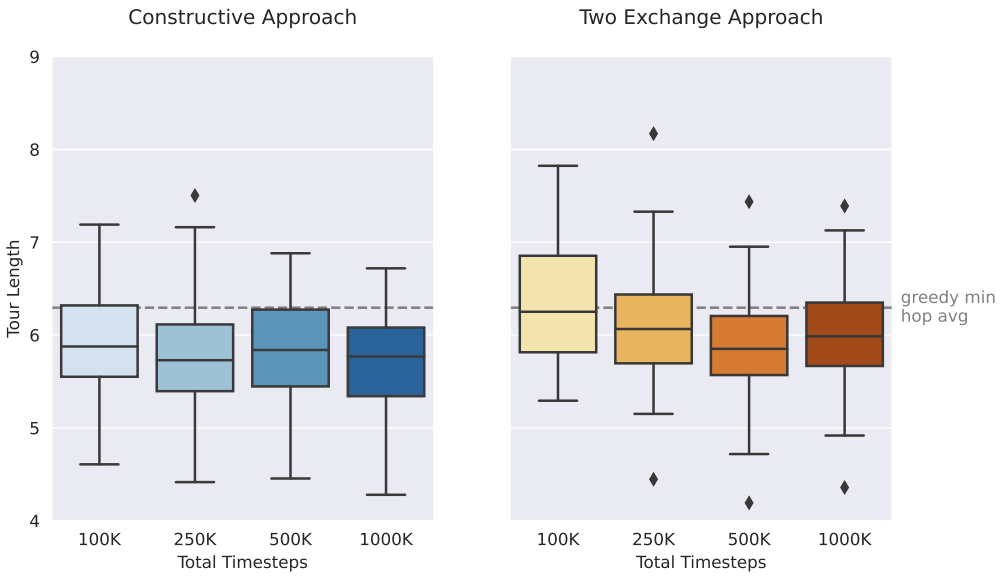


Figure 5.4: How long it takes a machine learning algorithm to train is a huge factor whether it can practically be employed. In our case 500k total environment steps seems to be the sweet spot where both the constructive (left) and the two-exchange approach (right) perform well. But the tour lengths of the modeled trained with less timesteps all produce competitive results.

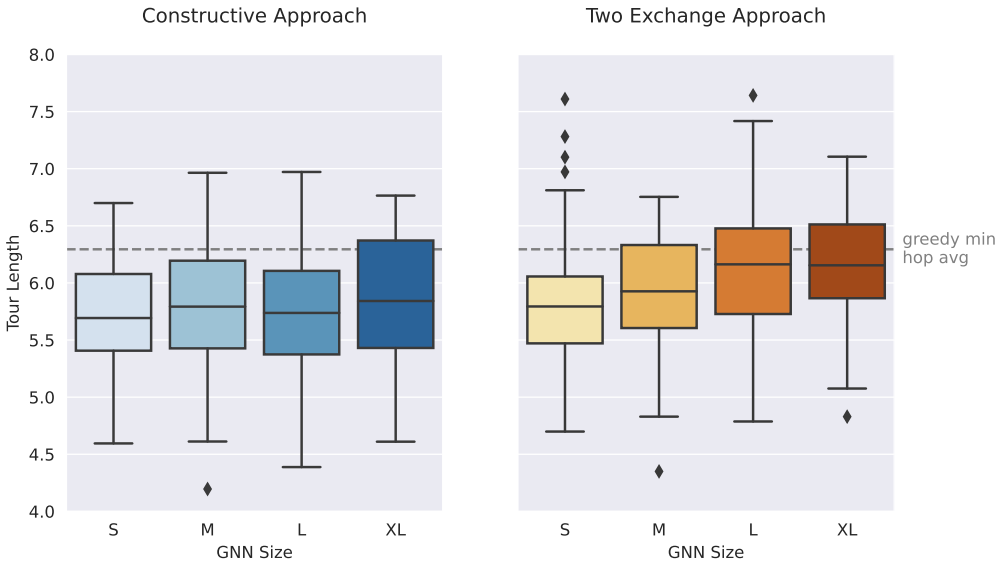


Figure 5.5: Surprisingly, networks on the smaller end perform generally better than larger ones. With a constructive approach (left) GNN sizes S,M,L perform better than XL. For the two-exchange environments (right) it is even more extreme: size S has clearly the lowest median tour length.

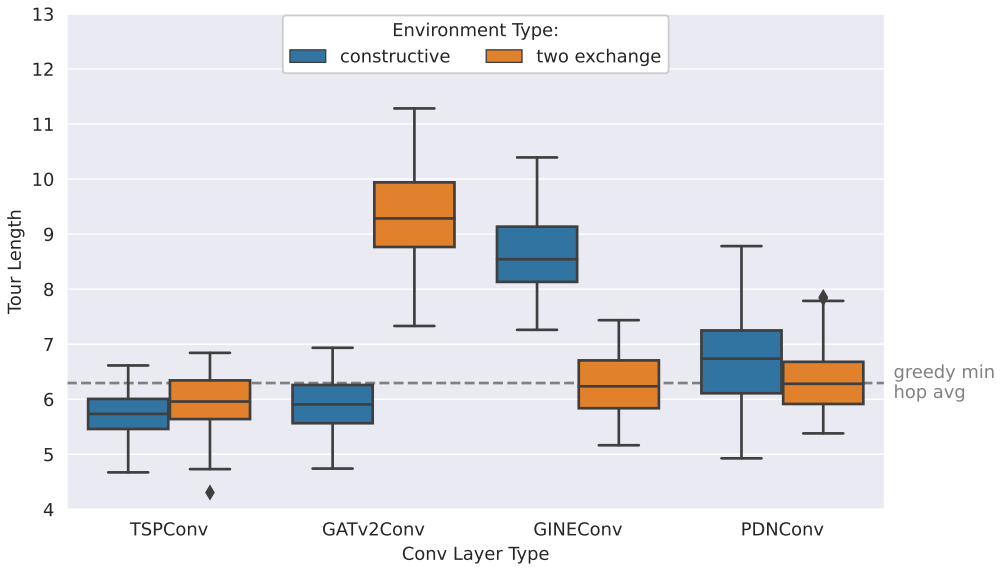


Figure 5.6: Choosing the right graph convolution layer seems to be key to good performance. Our proposed architecture performs best for both type of environments. GATv2 favors the constructive over two exchange approach where edge features are more important to encode the agent's state. For the GINE approach it is the other way around. For PDNConv, performance is more balanced.

We see that the smaller networks perform better. We can think of two possible explanations for this:

1. Since the graphs are complete, they have longest-shortest paths of one hop. So information is quickly distributed to all vertices, and additional communication rounds only increase the number of parameters.

2. Because we are in a RL setting, we lack direct supervision. So it is difficult to optimize a large number of parameters.

GNN Architecture The performance of the tested graph convolution layers is shown in Figure 5.6. We can see that our custom layer outperforms all others by a small margin, and it is the only layer that is able to outperform the baselines on both tasks. Note that both the GATv2 [Brody et al., 2022] and the GINE [Hu et al., 2020] layer only performed well on one of the two environments. This is discussed further in Section 5.2.5. While the PDN [Rozemberczki et al., 2021] layer did learn a reasonable heuristic for both tasks, it was unable to outperform the baselines.

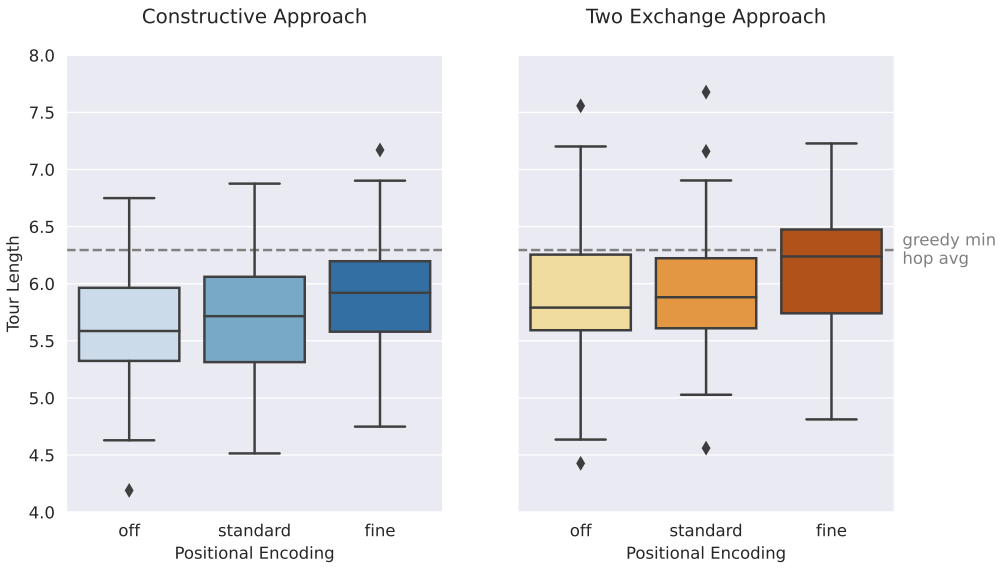


Figure 5.7: On the left the impact of positional encoding is shown for the constructive approach, and on the right the impact when using two-exchange environments. The resolution of the positional encoding corresponds to the smallest period of the sinusoids used. For the *standard* setting this is 10^{-2}, and 10^{-4} for the *fine* setting. Contrary to our previously stated speculation, positional encoding of graph edges does not increase performance.

Positional Encoding We have speculated that the positional encoding of edge weights may help the network learn to compare them. But our results show that these additional features do not lead to improved performance. This is summarized in Figure 5.7. The results suggests that the used graphs simply do not require high frequency comparator functions to be learned for good performance. Another issue that might impede the use of positional encoding is the fact that RL tasks lack direct supervision: The work of Tancik et al. [2020] applies this encoding to tasks where the network must learn a known mapping, e.g. from image coordinates to pixel color. In our approach, there simply might not be enough supervision to learn high frequency functions even if these were key to better performance.

5.2.3 Different Graph Types

We evaluate the performance of our agents on both the *2d* and *symmetric* graph types, presented in Section 5.1.1. We also compare the performance on graphs of different sizes, varying the number of nodes from 32 to 128. The number of nodes is the same for training and inference. The results are shown in Figures 5.8 and 5.9.

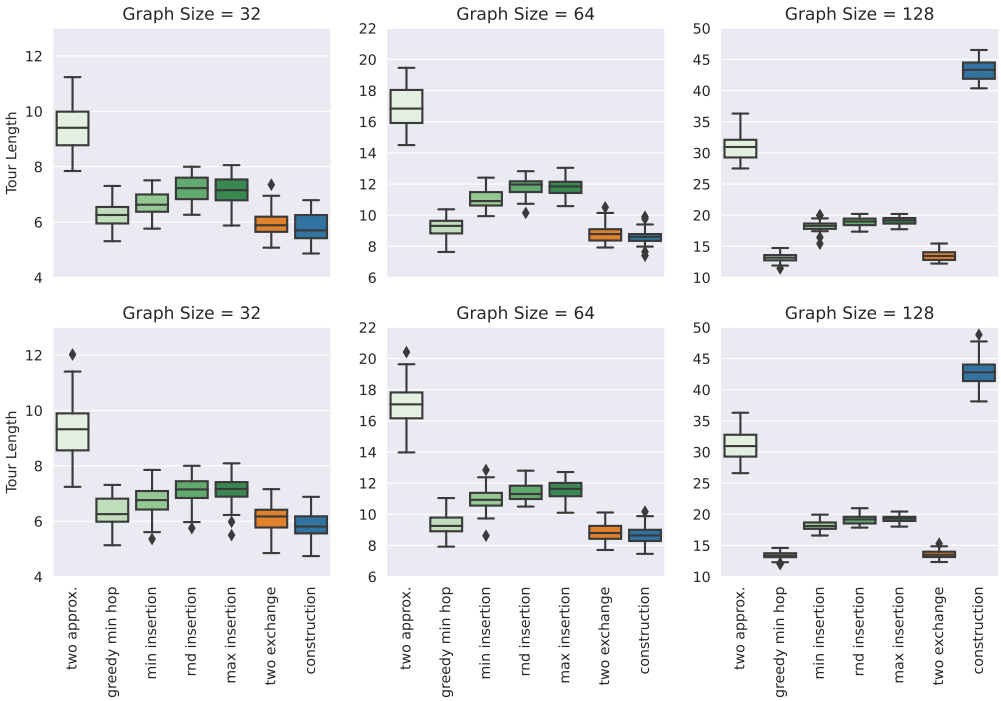


Figure 5.8: These plots show the performance of the baselines and the default RL solver under varying graph size. The top row of plots correspond to found tours of the train set, and the bottom row is the usual performance on the holdout set. The agents are trained and tested on a dataset of 64 symmetric graphs of the size marked at the top. Unsurprisingly, the performance on the train set is slightly better but it generally translates well to the holdout set. Up to 64 nodes the RL approach outperforms all baselines. For graphs of 128 nodes (right) the greedy minimum hop baselines is the winner. However, the two exchange based RL approach comes in as a close second. The constructive approach failed to learn any reasonable tours. This may indicate that training RL agents becomes increasingly harder with larger problem instances.

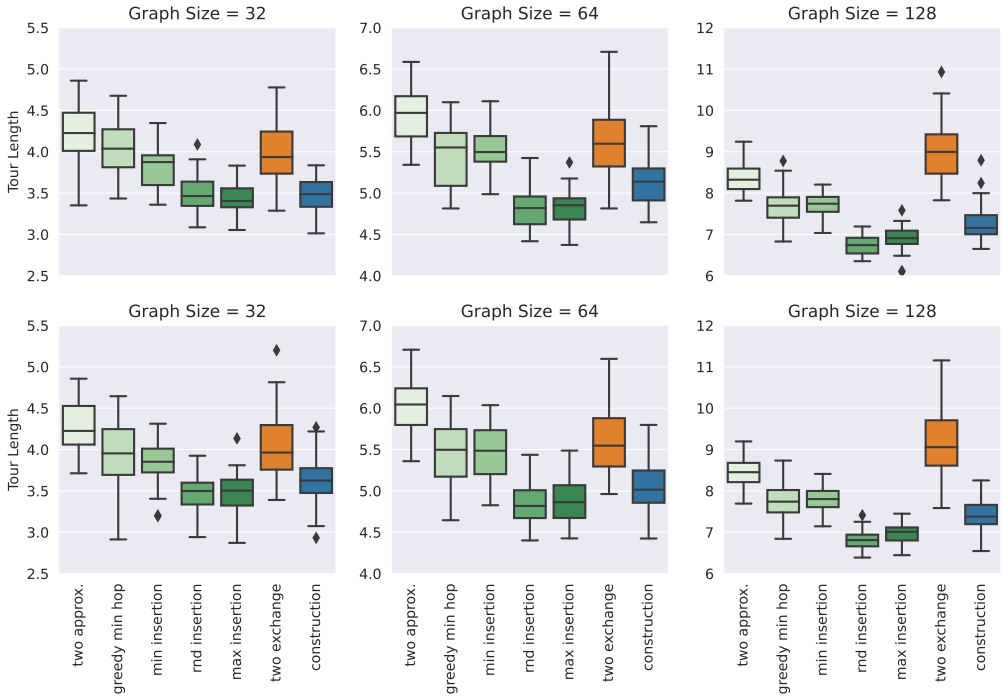
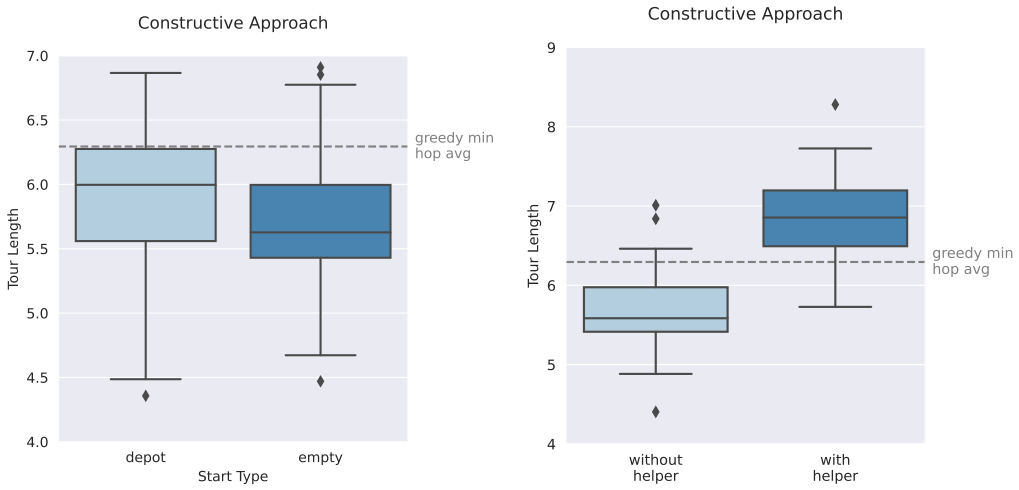


Figure 5.9: For these experiments 2d graphs were used. For a second time we show the performance on the train set graphs in the top row plots. The holdout set tours can be seen in the bottom row. Even on training graphs, our RL-based approaches get outperformed by some of the baselines. The relative performances translate almost one-to-one to the holdout set. While, both random and maximum insertion lag behind the greedy minimum hop baseline on symmetric graphs, they are at the top for 2d graphs.

We can see that our agents are capable of beating the baselines on the symmetric graphs, but not on the 2d graphs. This is likely due to the fact that our formulation is completely agnostic to the embedding of the nodes in the 2d-plane, and is only based on the $n \times n$ distance matrices. Additionally, the insertion-based baselines are much better on 2d graphs. There the best performing one is maximum insertion or random insertion depending on the graph size.

The performance comparison between the baselines and our learned heuristics is similar for all graph sizes, although on one occasion (symmetric graphs with $n = 128$), the agent struggled to learn a decent constructive heuristic. This might indicate that learning a constructive heuristic becomes harder on larger graphs, though we do not have enough empirical evidence to claim this, and instead suggest that future work investigate this further.



(a) Empty vs depot start **(b)** Best insertion helper

Figure 5.10: The two decisions that lead to different kinds of constructive RL heuristics: whether to start from scratch or from a fixed depot (left), and whether to employ the best insertion helper (right). The choice of start type seems to have less of an impact than whether one opts to choose the helper function, where using it on symmetric graphs has an negative impact on found tour lengths.

5.2.4 Construction Heuristic Options

In Subsection 4.1.1, some different choices that go into designing a constructive environment were already highlighted. Two of these decisions are tested here.

Empty vs. Depot Start When looking at non reinforcement learning heuristic, constructive approaches usually start with a fixed depot. However, due to the final tour's cyclic property, any vertex can theoretically act as a starting point. Moreover, the RL framework facilitates our actor to start with an empty vertex set and every node being a valid first action. In Figure 5.10a, we see the comparison of an RL actor that starts totally from scratch, with one that starts with an arbitrary depot.

We observe that the median of the empty start lies below the median of starting with a fixed depot, but differences are mostly negligible.

Best Insertion Helper At first glance, best insertion might seem strictly more powerful than just appending a chosen vertex to the partial solution, because it includes the possibility to insert at the tail. But this is generally not the case (see Appendix A.1) and we also see this in practice. In Figure 5.10b, an actor utilizing an environment without the best insertion helper clearly outperforms a

comparable actor using the helper. Generally, a properly trained policy should indeed be able to generate a sequence of actions that correspond to a decent tour without any added reshuffling.

In Subsection 5.2.3 we showed that the insertion-based baselines' relative performances vary with the used graph type. One might hope that a similar effect applies to the RL-based approach. We found this to not be the case: Even on 2d graphs opting to not use best insertion helper results in shorter tours. A possible explanation is that the RL-agent is not expressive enough to train the more complex behaviour of the best insertion over just appending a solution. Additionally, the fact that the maximum outperforms the minimum insertion baseline, may indicate that a more complex reward system is needed when using the helper.

5.2.5 Two Exchange Heuristic Options

We test a number of different options for the two exchange task. These options can be specified in the environment, to yield a different RL task which then enables the agent to learn better or worse heuristics.

Edge Attributes vs. Node Attributes As we have noted in Section 4.1.2, there is an option to copy the attributes of the incoming and outgoing tour edges into the attributes of the respective node. Since we effectively duplicate information that is already present in the edge attributes, this should not significantly impair performance. Moreover, we use this experiment to gauge how well a given graph convolutional layer type can extract information from edge attributes. If there is a large performance improvement when the tour edge attributes are added to the node attributes, then we may conclude that the graph convolution layer was unable to extract this information from the edge attributes by itself.

The results are shown in Figure 5.11. As we can see, our custom layer TSP-Conv outperforms the others by a small margin. More interesting is the comparison to the GATv2 convolution: While our layer is very similar to GATv2, it performs much better when the tour edge attributes are not present in the node attributes. We can therefore conclude that our attention layer is better at extracting information from edge attributes that is not present in the nodes. An explanation for this was already presented in Section 4.2.2: Commonly available implementations of graph attention do not allow for edge attributes to propagate into the node hidden vectors, because they only utilize the edge attributes for computing attention scores.

Note that trends of this subsection's data mirrors the observations about GNN types from Subsection 5.2.2, where we saw that GATv2 performs much better with a constructive approach which encodes all the agent's state in the node features and only keeps edge weights and positional encodings in edge attributes.

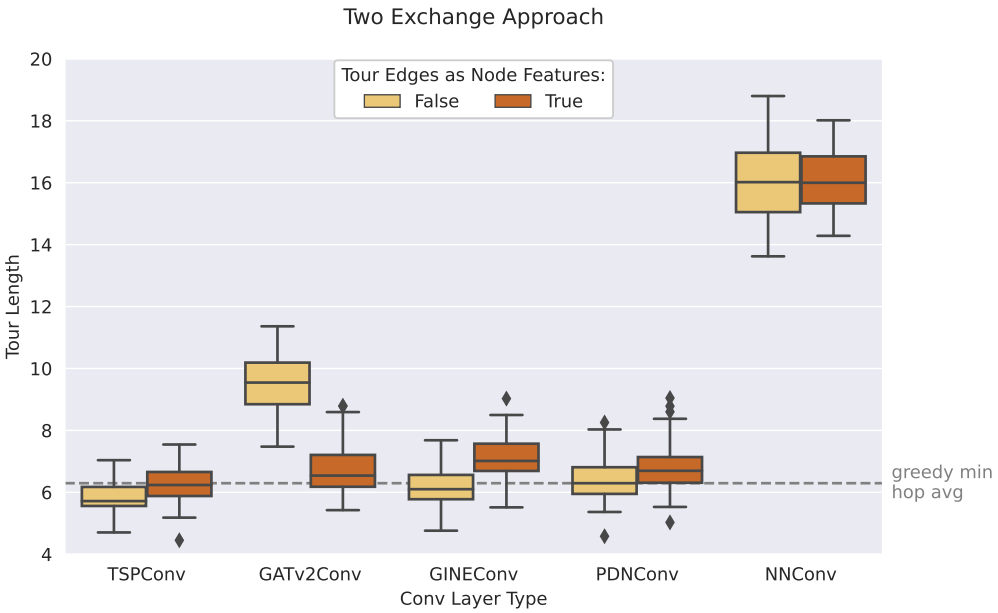
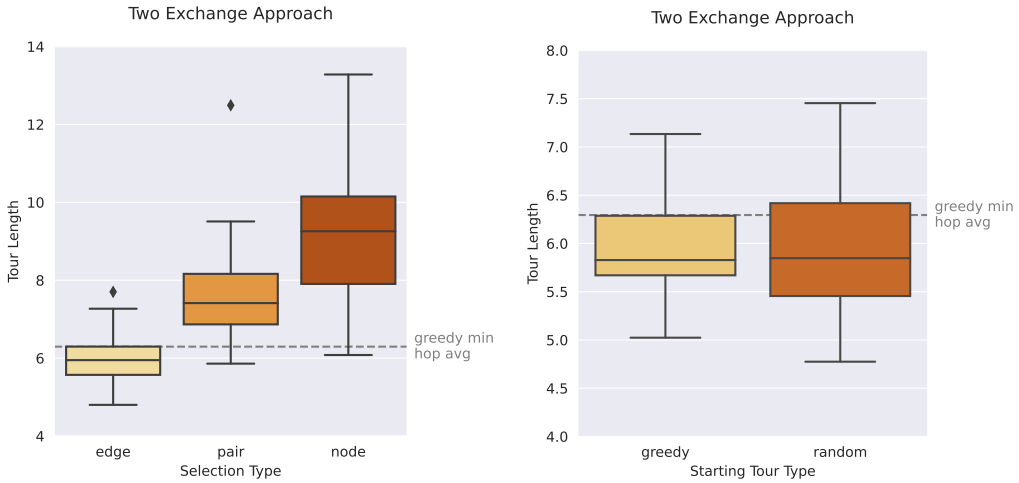


Figure 5.11: Results from running experiments with the two exchange approach that adds tour edge features redundantly to node features. Our proposed architecture, TSPConv, shows the shortest median tours in both cases. GATv2 performs better with more expressive node features, possibly because it fails to extract the relevant information from the edge features. Results from GINE, PDN and NNConv are more balanced than GATv2's but still worse than TSPConv's.



(a) Selection strategy **(b)** Starting from greedy solution

Figure 5.12: Three different selection strategies have been proposed for the two exchange approach. In the left plot we can see that the edge selection scheme produces the cheapest tours. It needs a less complicated masking scheme than the node, and a less complex action space than the edge pair selection strategy. On the right the impact of the starting tour to the local search approach: the median tour length is almost identical.

Selection Strategy As explained in Section 4.1.2, there are several ways to model the action space for two exchange. Either the agent selects nodes (which requires 4 environment steps per two exchange), or tour edges (requiring 2 steps per two exchange), or it selects a two exchange step directly in the form of a non-incident tour edge pair.

The comparison is shown in Figure 5.12a. We can see that the tour edge selection strategy outperforms the others by a small margin. We speculate that this approach beats the node selection strategy because the latter has a more complicated masking strategy (where every second step, the agent can only select a neighbor of the previously selected node). On the other hand, we conjecture that it beats the tour edge *pair* selection strategy because the latter has a more complicated action space: Since the agent has to select a pair of nodes in one step, the per-node outputs of the GNN are concatenated pairwise, and then processed again to yield an $n \times n$ matrix of scalar values, used either as log probabilities or Q-values. This more complicated architecture might inhibit performance to some degree.

Starting from a Greedy Solution The local search strategy has the benefit that the initial feasible solution can be constructed in a number of ways. While the default setting uses a random tour, we can also start from a tour that is

obtained by a greedy heuristic. However, as we can see in Figure 5.12b, this makes very little difference. Note that both approaches make n two exchange updates, hence the edges of the initial tour can be completely replaced anyways.

5.2.6 Evaluation on the TSPLib Dataset

Table 5.1 shows the performance of our RL agents trained with default settings on all graphs from TSP dataset with upto 100 vertices. Since the dataset mostly contains graphs that fulfill the triangle inequality, we also compare agents trained on the two different graph generation schemes, introduced in Section 5.1.1.

We can summarize the results as follows:

- In general, our RL agents generalize poorly to the TSPLib instances. In almost all cases, the agents cannot outperform the baseline heuristics.
- The constructive approach generalizes significantly better than the local search approach.
- The agents that are trained on the euclidean graphs perform very slightly better on average than those trained on graphs which do not have a plane embedding or satisfy metric properties such as the triangle inequality. The number of nodes seems to affect generalization as well: The training data consists of graphs with 32 nodes, hence the agents perform rather well on TSPLib graphs of sizes 22, 26, and 29, but rather poorly on graphs of size 100.

This suggests that the performance of RL methods on the TSP is dependent on the type of training data used.

5.3 Summary

We summarize the most important findings from our experiments:

- Both the constructive and the local search approach enable the GNN-based agent to learn TSP heuristics that are competitive with a number of baseline heuristics on randomly generated graphs. But this performance does not seem to generalize to other classes of graphs, such as the TSPLib dataset [Reinelt, 1991].
- Our custom attention layer outperforms the canonical implementation of graph attention layers when the task requires the extraction of information from the edge attributes. This is likely due to the fact that these other implementations only utilize the edge attributes for computing attention scores.

Used training data:		symmetric		2d	
graph name	best baseline	2-ex	construction	2-ex	construction
att48	1.055043	1.315017	1.170117	1.519477	1.269383
bayg29	1.03354	1.140373	1.144721	1.200621	1.054037
bays29	1.063862	1.09901	1.037129	1.155446	1.102475
berlin52	1.033545	1.453991	1.346195	1.291435	1.24516
brazil58	1.051703	1.885725	1.311282	1.627643	1.202481
burma14	1.057177	1.15498	1.183569	1.132711	1.244959
dantzig42	1.02289	1.284692	1.310443	1.254649	1.153076
eil51	1.049296	1.262911	1.119718	1.41784	1.117371
eil76	1.074349	1.8829	1.291822	1.947956	1.150558
fri26	1.01921	1.113127	1.258271	1.073639	1.140875
gr17	1.001439	1.076259	1.189448	1.282974	1.197122
gr21	1.0	1.162911	1.141485	1.157	1.096417
gr24	1.036164	1.193396	1.158019	1.091981	1.080189
gr48	1.020808	1.185295	1.137138	1.536861	1.087594
gr96	1.088084	3.052926	1.278306	2.95508	1.33884
hk48	1.041532	1.253468	1.158276	1.371521	1.201553
kroA100	1.083028	3.207547	1.32793	3.221925	1.185415
kroB100	1.065851	3.141005	1.23802	3.023757	1.193036
kroC100	1.098125	3.247241	1.208444	3.451347	1.209311
kroD100	1.07805	3.03245	1.231896	3.327651	1.196065
kroE100	1.038291	2.92369	1.325992	2.782082	1.250952
pr76	1.028338	2.401603	1.249318	2.080798	1.15038
rat99	1.113955	2.718415	1.1891	2.732453	1.251858
rd100	1.068521	3.507585	1.256258	3.594058	1.359671
st70	1.13037	2.031111	1.165926	1.921482	1.202963
swiss42	1.053417	1.269442	1.323645	1.195601	1.113119
ulysses16	1.005978	1.171745	1.226272	1.220732	1.151625
ulysses22	1.004848	1.177385	1.160702	1.156566	1.15015
average	1.050621	1.869507	1.219265	1.883045	1.182022

Table 5.1: Performance on TSPLib instances: The tour lengths are normalized to the known optimum. Hence, the table shows the approximation factor on these instances. TSPLib instances come with their unique names indicated on the left: the suffixed numbers indicate the number of nodes. Results are summarized in average at the bottom. The constructive approach generalizes better than two exchange but both approaches are beat by the best performing baseline.

- We have speculated that a positional encoding of the edge weights may enable the agents to learn to compare them with higher accuracy. Our experiments show that the use of the positional encoding does not improve performance.

Overall, our results prove the viability of RL in combination with GNN-based agents even for hard CO problems.

Chapter 6

Conclusion

We have applied reinforcement learning to the TSP, enabling GNN-based RL agents to learn high-quality heuristics. We have thoroughly analyzed how the design of the RL environment impacts the quality of the learned heuristic. Our demonstrations show that both approaches, constructive and two exchange, are viable options. They come with strengths and weaknesses, e.g. constructive approaches generalize better to unseen graph structures, whereas two exchange performs better with certain RL learning schemes such as A2C and vanilla policy gradient. Additionally, both types admit a number of different configurations of environments that yield decent results compared to non-parameterized heuristics. This suggests that the GNN-based agents have a high degree of flexibility.

We have compared a variety of different graph convolution layers. The results show that many of these layers perform well out-of-the-box under the condition that the RL environment encodes the information accordingly. This too proves the flexibility of GNN-based approaches. We have also proposed an extension to the popular attention based graph convolutions, which enables better performance in the presence of edge attributes.

Even though our agents perform well on test data drawn from the same distribution as the training data, generalization to other instances is relatively poor, whether they are trained on random symmetric or euclidean graphs. This suggests that using training data that matches the problem instances of the final application is key to real world performance. If this cannot be done, we suggest that the construction-based approach be preferred over the local search method, because it generalizes better to novel problem instances.

Future Work Having demonstrated the viability of combining RL with GNNs to solve CO problems, there are several promising research directions:

- Reformulating **other CO problems** as RL tasks may result in high-quality heuristics for previously inaccessible problems. Finding suitable formulations for CO problems that do not involve graphs but sets or other structures, might result in new neural network architectures.

- Improving the **generalization** of GNNs in general, may enable the agents to learn heuristics that perform well on new problem instances.
- We considered a general version of the TSP, only requiring that the edge weights be symmetric. A more **restricted version of the TSP** could be considered, where the nodes are known to be embedded in some k-dimensional space. The corresponding coordinates may be presented to the GNN as node features. Or alternatively, asymmetric or directed graphs may be of interest, as they arguably correspond to more realistic scenarios, e.g. sloped pathways that upon which travelling downhill is much faster than going up, or one-way-streets.

Bibliography

Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters for on-policy deep actor-critic methods? a large-scale study. In *International Conference on Learning Representations*, 2021.

David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. 2007.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957. ISSN 00959057, 19435274.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2022.

Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and André Augusto Ciré. Combining reinforcement learning and constraint programming for combinatorial optimization. *CoRR*, abs/2006.01610, 2020.

Xinyun Chen and Yuandong Tian. Learning to progressively plan. *CoRR*, abs/1810.00337, 2018.

Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Operations Research Forum*, 3, 1976.

Vašek Chvátal, William Cook, George Dantzig, Delbert Fulkerson, and Selmer Johnson. *Solution of a Large-Scale Traveling-Salesman Problem*, volume 2, pages 7–28. 11 2010. ISBN 978-3-540-68274-5. doi: 10.1007/978-3-540-68279-0_1.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2016.

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs, 2018.

Paulo R. de O. da Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning, 2020.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International Conference on Learning Representations*, 2020.

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

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry, 2017.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1):196–210, 1962.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks, 2020.

Shengyi Huang and Santiago Ontañò. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, may 2022. doi: 10.32473/flairs.v35i.130584.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey, 2020.

C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, oct 1960. ISSN 0004-5411. doi: 10.1145/321043.321046.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

Gerhard Reinelt. TSPLIB website. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Accessed: July 7, 2023.

Gerhard Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

Benedek Rozemberczki, Peter Englert, Amol Kapoor, Martin Blais, and Bryan Perozzi. Pathfinder discovery networks for neural message passing, 2021.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017a.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017b.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs, 2017.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains, 2020.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

Appendix A

Construction Heuristic Design

In this appendix, we list additional contents regarding the design of the construction heuristic that would otherwise hinder the flow of reading.

A.1 Best Insertion Helper

The experiments in Section 5.2 showed that the best insertion technique can improve performance under certain conditions. Especially on euclidean graphs, it pushes baseline implementation to extract shorter tours. For RL approaches, its application was less successful. As it is relatively easy to find an example where using the best insertion helper produces a better solution, one may conjecture that, for fixed action sequences, it will always find shorter tours than the more naïve approach of just appending a node to the partial solution. This is not the case, and it shown proofing the following lemma:

Lemma A.1. *Given an ordering $v_1, v_1, \ldots, v_{n-1}$ in which vertices are to be added to the solution, which is initialized with depot v_0, there are graphs $G = (V, E)$ with $|V| = n$ such that using the best insertion technique results in a longer tour.*

Proof. Consider the complete graph $G = K_5$ described in Figure A.1. Just appending nodes to constructive partial solution results in tour $\mathcal{T} = \{v_0, v_1, v_2, v_3, v_4, v_0\}$ with cost $C(\mathcal{T}) = 1.5$. Using best insertion gives rise to tour $\mathcal{T}' = \{v_0, v_1, v_3, v_2, v_4, v_0\}$ with worse cost $C(\mathcal{T}') = 2.8$. Problematic is the insertion of v_3 as the third city. It shoehorns the agent in to a costly insertion in the consecutive step for v_4.

□

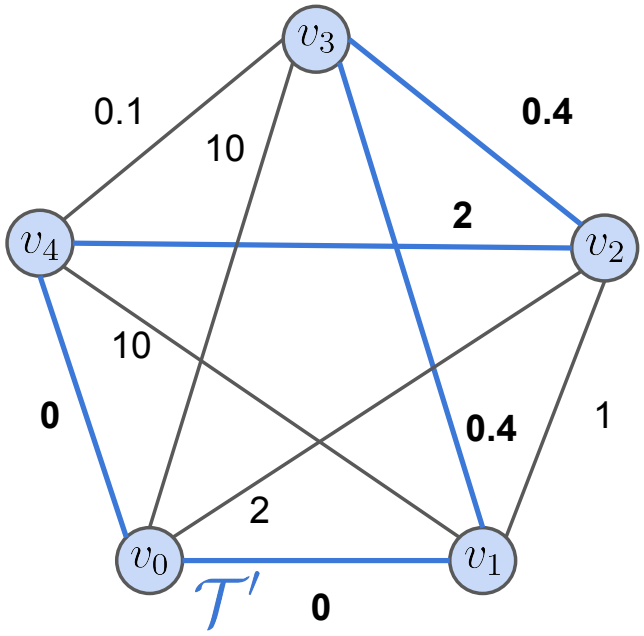


Figure A.1: Example for a graph where the best insertion technique results in a worse tour T' than just appending nodes.

Appendix B

Implementation Issues

The distributed systems laboratory is an opportunity for students to gain practical insights into active areas of research. Since this project combined multiple disciplines of mathematics and computer science, we have collected some of these insights, which might be of use for other students and researchers.

B.1 Practical Insights

The algorithms and data structures that we implemented for this lab project can be grouped by their respective disciplines:

1. Reinforcement learning algorithms
2. TSP heuristic design
3. Message passing neural networks

We will now describe some practical insights and pitfalls that we encountered when combining these three disciplines in our project.

B.1.1 Compatability Issues

Whereas for GNNs, Pytorch Geometric [Fey and Lenssen, 2019] has established itself as the major library used for implementation, for RL methods there is no such clear first choice. In this work we chose to both implement our own RL algorithms, as well as utilize the Stable-Baselines3 [Raffin et al., 2021] package. While it provides a number of well documented algorithms, it lacks built-in GNN support. Pytorch Geometric is most comfortably used with its custom data structures and its unique batching mechanism, differing from standard neural networks.

Like most RL packages, Stable-Baseline3 requires that custom environments comply with the OpenAI Gym [Brockman et al., 2016] environment interface.

Hence environments need to specify their action and observation spaces. While newer versions of the OpenAI Gym library support graph observations, this was not supported by the Stable-Baselines3 rollout buffers. Hence, we refactored our graphs into a dictionary of fixed sized tensors. This also required us to fix the topology of the graphs (at least during training). So we limited ourselves to complete graphs with a fixed number of nodes for training.

In our case major parts of the Stable-Baselines3 library had to be rewritten to accommodate GNNs. This in turn meant that we had to familiarize ourselves with all of the library's source code related to the needed algorithm. As is the case with many bigger libraries, the code is structured in a hierarchical way, with many sub and super classes making it hard to track the exact control-flow.

Implementing an RL algorithm oneself has both benefits and disadvantages. Well-established libraries are often more robust due to being used by many people. However, writing custom RL code for a limited set of tasks can greatly simplify the most complex data structures of RL algorithms, namely the rollout buffers. Given that all our environments used the same observation space and similar action spaces, we took advantage of this.

To summarize the key compatibility issues we encountered:

- The custom batching mechanism of Pytorch Geometric.
- The lack of support for graph observations in most RL libraries.

B.1.2 Throughput Issues

Another issue we encountered was the throughput of the TSP environments: Since these data structures must be called several times per gradient update, they can easily inhibit the learning speed of the actor. During earlier experiments, we tested environments that didn't simply return the original graph with suitable features, but instead transformed the topology of the graph based on the current state of the TSP solution.

For example, an alternative version of the constructive approach we introduced, returns a line graph where nodes correspond to edges of the original graph. Similarly, we considered a two-exchange environment that returned a graph whose nodes corresponded to possible two-exchange steps. These environments turned out to be much too slow to be of any use for learning a good heuristic using RL. In particular, these environments resulted in $\mathcal{O}(n^2)$ for steps calls, which was not acceptable.

B.1.3 Performance and Correctness

RL algorithms are sensitive to most of the standard normalization techniques that are used in neural networks, such as batch normalization [Ioffe and Szegedy, 2015]. We encountered such a problem, where a predefined multi-layer perceptron class from the PyTorch Geometric [Fey and Lenssen, 2019] package contained such layers by default. Only after disabling them, were we able to get good results.

Methods like PPO [Schulman et al., 2017b] (with clip loss) compute the probability ratio for a chosen action between the current policy and the policy used during the last rollout. Depending on the advantage of this action, the ratio (which should equal 1 right after collecting the rollout) is then marginally adjusted up or down using SGD. In our experience, batch normalization layers affect this ratio so much that it is no longer in a reasonable range (around 1) even at the start of training.

B.1.4 Unspecified Implementation Details in the Literature

Publications in this field of research are usually not very explicit on how they train models. Training set size, number of epochs or exact training procedures are not always mentioned. We found that once a method has shown success, these hyperparameters have not a huge impact on the performance (maybe hence they are sometimes omitted). For us, if an agent was unable to learn a good heuristic, it was usually due to some other issue, and time spent in trying to diagnose the problem in the training procedure was needlessly wasted.