**Saarland University**

**Bachelor's Thesis**


# Using Reinforcement Learning to solve Quadratic Assignment Problems

Submitted by:

Tim Göttlicher

Submitted on:

June 28, 2022

Reviewers:

Univ.-Prof. Dr. Verena Wolf

Dr. Andreas Karrenbauer

## Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

## Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

## Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

## Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, June 28, 2022

_____

Your Name

# Abstract

We design a deep reinforcement learning system to learn a heuristic method for the quadratic assignment problem (QAP), which can solve unseen problems. We propose a sequential decision process for the QAP that reduces the size of the problem in every step, allowing the system to learn incrementally from small problems and generalize to larger problems.

The policy of the system is guided by a message passing graph neural network. We show that learning progress of such a neural network on dense graphs, as they occur in the QAP, can be slow because the network cannot distinguish some nodes. We compare separation layers to remedy this issue.

Finally, we train different reinforcement learning algorithms on small randomly generated QAPs, and evaluate them on larger problems and the collection of problems in QAPLIB. While the results of our system cannot compete with solutions found by state-of-the-art learning-free QAP solvers at similar runtime, we show that it achieves considerable generalization to problems even when training only on small problems and is comparable to or better than a different learning-based solver from the literature.

# Acknowledgments

I would like to thank Dr. Andreas Karrenbauer and Prof. Dr. Verena Wolf for supervising and reviewing this thesis and letting me learn a lot about both reinforcement learning and optimization.

Furthermore, I am very grateful to my supervisor Joschka Groß for the valuable discussions and feedback, and his continuous support.

I am also thankful to my brother Max for his useful comments. Finally, I want to thank Yiting for her support while I was working on this thesis.

# Contents

# 1. Introduction

## 1.1. Challenges in deep learning for the QAP

In the quadratic assignment problem (QAP), the nodes of one graph have to be assigned to the nodes of a second graph, minimizing an objective function that consists of a cost term for each edge. The problem was introduced by Koopmans and Beckmann [12] as a model for location theory in economics. In that context, the edges in one graph represent transport volume between facilities, while the other graph is defined by the transport cost per unit between two locations. The total transportation cost of an assignment of facilities to locations should be minimized. Another example of a QAP is keyboard layout optimization [10], where the first graph represent letters with frequencies of letter bigrams as edge weights, and the second graph represents keys and the time it takes a user to press one key after another. In this case, the expected time to type a text should be minimized.

Finding an optimal solution to a QAP is known to be NP-hard, and even approximation within a constant bound is not possible in polynomial time unless P = NP [17]. Even some moderately sized problems found in QAPLIB [2] have not been solved[1], despite having been known for multiple decades. Therefore, it is interesting to consider approximation algorithms, which are usually not able to find an optimal solution, but instead try to find a reasonably good solution within a short runtime.

Our goal is to explore deep learning-based systems to train heuristic algorithms for the QAP. Such a system involves a neural network that should learn to encode statistical patterns of the problem to guide the algorithm to better results, reducing the reliance on manual analysis and hand-crafted heuristics.

However, successfully applying deep learning to the QAP is not straightforward because of the discrete and dense structure of the problem. While neural networks are usually trained based on gradients of a loss function, the QAP is a discrete optimization problem, so its objective function is not directly differentiable. This means that different methods to compute gradients and update the parameters of the neural network have to be used.

[1] See `https://www.miguelanjos.com/qaplib` for the current list of unsolved QAPLIB problems.

## 1.2. Related work

There exists some recent work to apply deep learning and deep reinforcement learning with graph neural networks to solve general quadratic assignment problems. Wang et al. [29] use graph neural networks to find solutions by either supervised or unsupervised training on quadratic assignment problems. They use sinkhorn normalization to generate a soft permutation given by a double-stochastic matrix. For unsupervised training they use a differentiable continuous relaxation of the more general, but also computationally more expensive QAP formulation by Lawler[14]. Liu et al. [15] use reinforcement learning for Lawler's QAP with a decision process in which the agent can add new pairs to an assignment, but also override previously assigned pairs.

A closely related problem is graph matching, where the nodes of two unweighted graphs have to be matched to get an optimal alignment of the edges. It can be viewed as a special case of the QAP where the edge weights are binary. Liu [16] developed a reinforcement learning approach using DQN with a graph neural network policy for graph matching, adding edges between the two graphs for pairs of assigned nodes. Nowak et al. [21]; Wang et al. [28]; Fey et al. [4] use graph neural networks and sinkhorn normalization for supervised learning on graph matching problems. Fey et al. [4] additionally use a second learned stage based on a per-node alignment measure to improve the matching.

Besides these recent attempts in using machine learning methods for the QAP and graph matching, a large variety of other heuristic solution algorithms exist for the QAP [17]. Our method falls into the class of constructive methods that incrementally build an assignment from individual pairs, that are selected heuristically [17]. An early example of such an algorithm was developed by Gilmore [6]. There are also many successful applications of metaheuristics, such as simulated annealing [30], genetic algorithms [25] or tabu search [18]. These methods are similar in spirit to our goal of adapting a general computational framework to the QAP. As a benchmark for our system representative of non-learning heuristics algorithms, we use the FAQ (Fast Approximate Quadratic assignment) algorithm [27], which computes a local optimum to a relaxation of the Koopmans-Beckmann QAP.

## 1.3. Methodology overview: A learning system to enable generalization on the QAP

In this work we want to build a reinforcement learning system for general Koopmans-Beckmann QAPs, which is able to generalize well to unknown instances. The main contributions of our method over previous work are using a decision process formulation which reduces assigned QAPs to QAPs of smaller size, and improving training convergence speed of a message passing graph neural network architecture on dense uniform

graphs with a separation layer.

We use reinforcement learning to train the neural network based on rewards equivalent to the objective value of the generated solutions. The problem is modeled as a sequential decision process, adding one pair of nodes to the assignment per step. This allows us to apply a reduction rule for assigned nodes described in section 3.1, simplifying the state space to improve training and generalization performance.

As a policy network, we use a graph neural network (GNN). GNNs are permutation equivariant and work with graphs of arbitrary size, which is necessary to generalize from the training data. To obtain good results on dense weighted graphs, it is important to incorporate the edge weights and increase the ability of the network to distinguish between nodes, which are problems we will discuss in chapter 4.

Finally, we show that training our system on small random instances is sufficient to achieve reasonable generalization performance on problems larger than it was trained on. We compare our results on QAPLIB to a relaxation-based approximate solver and a learning-based solver. These experiments are presented and discussed in chapter 5.

# 2. Background

## 2.1. The quadratic assignment problem

The objective function of the Koopmans-Beckmann formulation of the QAP is defined in terms of the edge weights of the two input graphs with node sets $\mathcal{A}$ and $\mathcal{B}$, as well as edge weights $a_{i,j}$ with $i, j \in \mathcal{A}$ and $b_{u,v}$ with $u, v \in \mathcal{B}$. Given an assignment $f : \mathcal{A} \to \mathcal{B}$ of nodes in $\mathcal{A}$ to nodes in $\mathcal{B}$, the total cost is defined by the cost of all assigned edges,

$$V = \sum_{i,j \in \mathcal{A}} a_{i,j} b_{f(i),f(j)}. \tag{2.1}$$

The objective is to minimize the total cost $V$ by finding an appropriate assignment $f$. Figure 2.1 is an illustration for the computation of the cost term of one edge, where A is assigned to 1 and D is assigned to 4. The cost of this partial assignment can be computed as $a_{A,D} \cdot b_{1,4} + a_{D,A} \cdot b_{4,1}$ (since the graph in the example is undirected, the second term for the reverse of the edge can be dropped without changing the optimal assignment).

It is possible to get a slightly more general form by adding a linear[1]assignment term with the cost $c_{u,v}$ for $u \in A$ and $v \in B$ (which can be interpreted as the cost of placing a facility at a certain location without transportation costs to other facilities):

$$V = \left( \sum_{i,j \in \mathcal{A}} a_{i,j} b_{f(i),f(j)} \right) + \sum_{i} c_{i,f(i)}. \tag{2.2}$$

In the most general form of the QAP introduced by Lawler [13] the objective value is

[1]Here, the names quadratic and linear stem from the integer programming formulation, which uses a binary variable $x_{i,j}$ for each possible assignment pair. It is a coincidence that in the Koopmans-Beckmann form the quadratic term is also quadratic in the weights $a$ and $b$.

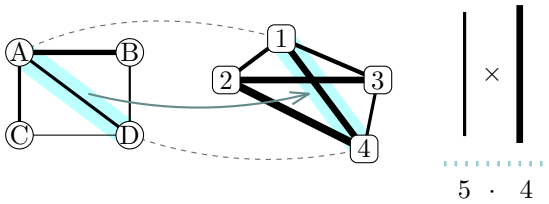



Figure 2.1.: Example of computing cost of a partial assignment

defined as

$$V = \sum_{i,j \in \mathcal{A}} c_{i,j,f(i),f(j)}. \tag{2.3}$$

This form includes the Koopmans-Beckmann formulations as special cases. However, it requires storing $n^4$ coefficients, compared to $2n^2$ and $3n^2$ coefficients in the forms 2.1 and 2.2 respectively.

We will focus on the Koopmans-Beckmann form with linear term to avoid time and memory overhead of Lawler's QAP. The linear term is useful because it comes up while constructing a decision process in 3.1, even if the initial problem is not in this form. Any mentions of the QAP in the remaining text will assume a problem in the form of equation 2.2 unless otherwise mentioned. We will identify the set of QAPs by tuples $(\mathbf{A}, \mathbf{B}, \mathbf{C}) \in \mathrm{QAP}_n = (\mathbb{R}^{n \times n})^3$, that correspond to the coefficient matrices.

## 2.2. Deep reinforcement learning

Reinforcement learning studies algorithms that learn policies to maximize rewards by interacting with a sequential environment. In this section we first give a brief introduction into the framework of reinforcement learning on Markov decision processes. Then, we introduce the key ideas of the reinforcement learning algorithms DQN, REINFORCE and A2C to help put our findings in context. The descriptions in this entire section are mostly based on [24].

### 2.2.1. Fundamental concepts

**Markov decision process** A *Markov decision process* (MDP) is defined as a tuple $(S, A, T)$, where $S$ is the set of states, $A(s)$ is the set of actions in state $s$ and $T$ is the transition function. In general, $T$ is defined as a conditional probability distribution $T(s', r \mid s, a)$, assigning a probability to the next state $s' \in S$ and the immediate reward $r \in \mathbb{R}$, given the current state $s \in S$ and an action $a \in A(s)$. For a deterministic MDP, we will write the transition function as $T(s, a) = (s', r)$ for simplicity.

Additionally, an initial state distribution $h(s_0)$ specifies for each state a probability with which it will be chosen as the starting state. States $s$ with no transitions to other states, i.e., $\forall a \in A(s) : T(s, 0 \mid s, a) = 1$ are terminal.

**Return** Given a starting state $s_0$ and a sequence of actions $a_1, \ldots, a_n$ that produces a sequence of states $s_1, \ldots, s_n$ ending in a terminal state $s_n$ and rewards $r_1, \ldots, r_n$ on an MDP (i.e., a sequence that constitutes an *episode*), the discounted sum of

future rewards at step $t$

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{k+t+1} \tag{2.4}$$

is called the *return*, with a discount factor $\gamma \in [0, 1]$. Later, we will only consider the undiscounted case with $\gamma = 1$, because in our application the sum of rewards has to be preserved.

**Policy** A *policy* $\pi(a \mid s)$ maps a state $s \in S$ to a probability distribution over the applicable actions $a \in A(s)$. The main goal of reinforcement learning is to find an optimal policy $\pi^*$ that generates actions in each state such that the expected return $\mathbb{E}_{\pi^*}(G_0)$ when following $\pi^*$ is maximized when evaluated with a given initial state distribution.

**Value functions** The *state-value function*, defined as

$$v_\pi(s) = \mathbb{E}_\pi(G_t \mid s_t = s), \tag{2.5}$$

gives the expected return when following a policy $\pi$ from a state $s$. To compare actions without having to look at possible transitions, one can also define the *action-value function*

$$q_\pi(s, a) = \mathbb{E}_\pi(G_t \mid s_t = s, a_{t+1} = a), \tag{2.6}$$

which is the expected return when following policy $\pi$ after choosing action $a$ in state $s$.

Deep reinforcement learning algorithms are reinforcement learning methods that involve deep neural networks for function approximation. In contrast to tabular reinforcement learning algorithms, which compute explicit values for each state, deep reinforcement learning is also applicable to very large or continuous state spaces. There are two classes of such algorithms that we consider, value-based methods and policy gradient methods, and two algorithms for each class, that differ (loosely speaking) in the way they estimate a value function, either by Monte Carlo estimation or by temporal difference learning. This means that we use four algorithms that can roughly be classified as in table 2.1. Next, we describe these algorithms in detail.

| | Value-based | Policy gradient |
|---|---|---|
| Monte Carlo | MCQ | REINFORCE |
| Temporal difference | DQN | A2C |

Table 2.1.: Classification of the used deep reinforcement learning algorithms

### 2.2.2. Value-based algorithms

*Value-based* algorithms estimate the action-value function of the current policy with a neural network. Given such an estimate, the current policy can be improved by greedily following the action with the highest action value (in tabular reinforcement learning this is guaranteed by the policy improvement theorem, but theoretical guarantees for improvement are lost with function approximation).

The value function estimate can be learned with different methods. *Monte Carlo* methods directly estimate the expectation in the definition of the action-value function $q_\pi(s, a)$ (or the state-value function $v(s)$) by $k$ samples $g_i$ of the returns generated by following the policy $\pi$ as

$$q_\pi(s, a) \approx \frac{1}{k} \sum_{i=1}^{k} g_i. \tag{2.7}$$

For a deterministic MDP and deterministic policy, this is even an equality already for $k = 1$. An estimator $\hat{q}_\theta$ parameterized by $\theta$ can be learned from $n$ sampled tuples of state, action and return $(s_i, a_i, g_i)$ with the mean squared error loss function

$$L(\theta) = \frac{1}{n} \sum_{i=0}^{n-1} \left(\hat{q}_\theta(s_i, a_i) - g_i\right)^2, \tag{2.8}$$

by minimizing it with, for example, a simple stochastic gradient descent update rule with learning rate $\alpha$

$$\theta' = \theta + \alpha \nabla L(\theta). \tag{2.9}$$

This can be directly used to create an algorithm we call Monte-Carlo Q learning, or MCQ (lacking a better name).

**Algorithm 1** MCQ pseudocode

**Require:** MDP defined by $(S, A, T)$ and initial state distribution $h$; estimator $\hat{v}_\theta$, initial parameters $\theta$

```
for N episodes do
    Sample s_0 ~ h
    n ← 0
    while s_n is not terminal do                    ▷ Run episode
        a_n ← arg max_a(q̂_θ(s_n, a))
        s_{n+1}, r_{n+1} ← T(s_n, a_n)
        n ← n + 1
    for i ∈ {0, ..., n − 1} do                      ▷ Compute returns
        g_i ← Σ_{k∈{i+1,...n}} r_k
    Compute L(θ) by eq. (2.8)
    Update parameters θ based on ∇_θ L
```

*Temporal difference* learning is an alternative that allows training without waiting for

the end of an episode (which is necessary to get the return) and off-policy training, i.e., improving one policy from the sampled transitions of another policy, such as a separate policy that is forced to explore the state space. This works by *bootstrapping* the value estimate from the next step, based on the Bellman equation

$$q_\pi(s,a) = \mathbb{E}_{r,s'\sim\pi}[r + v_\pi(s')]]. \tag{2.10}$$

The DQN algorithm [19] approximates the action-value function $q_{\pi^*}$ of the optimal policy $\pi^*$ with an estimator $\hat{q}_\theta(s,a)$ parameterized by $\theta$ by using temporal difference learning. The state-value function $v_{\pi^*}$ is estimated as

$$\hat{v}(s) = \arg\max_a \hat{q}_\theta(s,a). \tag{2.11}$$

For each transition encountered it stores the tuple $(s,a,s',r)$ in a data set (known as replay buffer). The network is continuously trained on mini-batches of $k$ transitions $(s_i,a_i,s_i',r_i)$ sampled from the replay buffer by minimizing the loss function

$$L(\theta) = \frac{1}{k}\sum_{i=0}^{k-1}[((r_i + \max_{a'}\hat{q}_{\theta_t}(s_i',a')) - \hat{q}_\theta(s_i,a_i))^2], \tag{2.12}$$

using stochastic gradient descent, aligning the predicted action-value in the current step with the best action-value in following next step for each known transition. If $s'$ is terminal, the target $\max_{a'}\hat{q}_{\theta_t}(s_i',a')$ is replaced with 0. The target action-value uses parameters $\theta_t$ that are held fixed while the parameters $\theta$ are learned, and are updated as $\theta_t \leftarrow \theta$ at fixed intervals. Separating the target parameters $\theta_t$ and the policy parameters $\theta$, together with breaking correlation between experienced transitions by sampling them from a replay buffer makes training the agent with a neural network more stable.

Often in reinforcement learning, it is important that the agent sufficiently explores unknown states, as they might lead to better results than the states the agent has visited while following its current policy. To ensure exploration, a DQN agent can use an exploratory policy during training, usually the $\epsilon$-greedy policy that chooses

$$\arg\max_a(\hat{q}_\theta(s,a))$$

with probability $1-\epsilon$ and a random action with probability $\epsilon$. While this could also be done in MCQ, it would estimate the value of the noisy $\epsilon$-greedy policy, while DQN only uses the seen transitions but not the value estimate of the exploration policy.

**Algorithm 2** DQN pseudocode

**Require:** MDP defined by $(S, A, T)$ and initial state distribution $h$; estimator $\hat{v}_\theta$, initial parameters $\theta$

**for** N episodes **do**
- Sample $s \sim h$
- **while** $s$ is not terminal **do** ▷ *Run episode*
  - **if** chance with probability $\epsilon$ **then**
    - $a \leftarrow$ random action
  - **else**
    - $a \leftarrow \arg\max_a(\hat{q}_\theta(s, a))$
  - $s', r_{n+1} \leftarrow T(s, a)$
  - Store $(s, a, s', r)$ in replay buffer
  - $s \leftarrow s'$
- Sample $k$ tuples from replay buffer
- Compute $L(\theta)$ by eq. (2.12)
- Update parameters $\theta$ based on $\nabla_\theta L$
- Every X steps: $\theta_t \leftarrow \theta$.

### 2.2.3. Policy gradient algorithms

The other class are *policy-gradient* algorithms, which directly learn a parameterized stochastic policy $\pi_\theta(s, a)$ by updating parameters based on an estimation of the gradient $\nabla_\theta \mathbb{E}_{\pi_\theta}(G_t)$ over episodes sampled with $\pi_\theta$. While it is often not possible to compute $\nabla \mathbb{E}_{\pi_\theta}(G)$ directly, it is possible to derive the equivalence

$$\nabla_\theta \mathbb{E}_{\pi_\theta}(G_t) = \mathbb{E}_{\pi_\theta}\left[G_t \nabla \log(\pi_\theta(s_t, a_t))\right] \tag{2.13}$$

using the policy gradient theorem. This means that only sampling log-probabilities and returns is sufficient to estimate the gradient to improve the policy. Directly making use of this idea, the REINFORCE algorithm executes a policy and updates its parameters after each episode according to the gradient computed from the sampled values.

This directly uses the Monte Carlo returns. It is also possible to include baseline function $b(s)$ such that the policy gradient becomes

$$\nabla_\theta \mathbb{E}_{\pi_\theta}(G_t) = \mathbb{E}_{\pi_\theta}\left[(G_t - b(s_t)) \nabla \log(\pi_\theta(s_t, a_t))\right]. \tag{2.14}$$

Doing so can reduce the variance of the estimation with an appropriate baseline. We only use the mean of all encountered returns as a baseline for simplicity.

**Algorithm 3** REINFORCE pseudocode

**Require:** MDP defined by $(S, A, T)$ and initial state distribution $h$; estimator $\pi_\theta$, initial parameters $\theta$

```
for N episodes do
    Sample s_0 ~ h
    n ← 0
    while s_n is not terminal do                ▷ Run episode
        Sample a_n ~ π_θ(s_n, ·)
        s_{n+1}, r_{n+1} ← T(s_n, a_n)
        n ← n + 1
    for i ∈ {0, ..., n − 1} do                  ▷ Compute returns
        g_i ← Σ_{k∈{i+1,...n}} r_k
    Update parameters θ with ∇_θ E_{π_θ}(G_t) from eq. (2.13) or eq. (2.14).
```

The advantage actor-critic algorithm (A2C) is based on temporal difference learning instead of the Monte Carlo returns in REINFORCE. It uses the baseline $\hat{v}_\xi(s) \approxeq v(s)$ and replaces the return in equation 2.14 by the bootstrapped estimate $r_{t+1} + \hat{v}_\xi(s_{t+1})$. This makes the estimation of the gradient,

$$\nabla_\theta \mathbb{E}_{\pi_\theta}(G_t) \approx \mathbb{E}\left[(r_{t+1} + \hat{v}_\xi(s_{t+1}) - \hat{v}_\xi(s_t)) \nabla \log(\pi_\theta(s_t, a_t))\right], \tag{2.15}$$

biased (i.e., the equality does not hold anymore in general), but accelerates learning in some situations. The term $\delta = r_{t+1} + \hat{v}_\xi(s_{t+1}) - \hat{v}_\xi(s_t)$ is an estimation of the advantage of the action $a_t$ and serves as a critic of the action taken by the policy, which is where

the name of the algorithm stems from. The value estimator $\hat{v}_\xi$ is trained to minimize the error of the value function, i.e.,

$$L(\xi) = (r_{t+1} + \hat{v}_\xi(s_{t+1}) - \hat{v}_\xi(s_t))^2, \tag{2.16}$$

where the temporal difference target $r_{t+1} + \hat{v}_\xi(s_{t+1})$ is treated as a constant for differentiation. Thus, the gradient is

$$2\delta\nabla_\xi \hat{v}_\xi(s). \tag{2.17}$$

In contrast to REINFORCE, the updates of both $\pi_\theta$ and $\hat{v}_\xi$ can be done in every step because they do not depend on the return.

**Algorithm 4** A2C pseudocode

**Require:** MDP defined by $(S, A, T)$ and initial state distribution $h$; estimators $\pi_\theta$ and $\hat{v}_\xi$, initial parameters $\theta$ and $\xi$

```
for N episodes do
    Sample s ~ h
    n ← 0
    while s is not terminal do                    ▷ Run episode
        Sample a ~ π_θ(s, ·)
        s', r ← T(s, a)
        Compute δ = r + v̂_ξ(s') − v̂_ξ(s)
        Update parameters θ based on the policy gradient eq. (2.15)
        Update parameters ξ based on the critic gradient eq. (2.17).
        s ← s'
```

# 3. The QAP as a reinforcement learning problem

## 3.1. Markov Decision Process model

To bring the QAP into the standard form of reinforcement learning, one needs to define a suitable Markov Decision Process (MDP). A valid MDP formulation should satisfy the property that any action sequence that yields an expected return $G$ can be transformed into an assignment with objective value $V = G$, so that a reinforcement learning agent trained on it can also be used as a solver for the QAP.

Unlike an MDP, the QAP is not inherently sequential. A simple way to make the solution process sequential is by incrementally building the assignment from individual pairs of nodes. In every step the agent can choose two nodes, one from each graph, that should be assigned.

For successfully designing a reinforcement learning system, a crucial point is the state representation of the MDP, as it forms the input from which the agent has to derive its decisions. The state space of the MDP should be structured in a way that the function approximation of the reinforcement learning algorithm can generalize across states (by interpolation and to some extent extrapolation for out-of-distribution problems). This notion is difficult to formalize, but we test it experimentally.

Since a QAP can be represented by two weighted graphs $\mathcal{A}$ and $\mathcal{B}$ as introduced in section 2.1, we will use graph representations together with graph-based policies in the agent. A simple suitable state representation is a binary variable for each pair of nodes $u \in \mathcal{A}$ and $v \in \mathcal{B}$ that indicates whether $u$ has been assigned to $v$ (which is briefly evaluated in section 5.6).

Interestingly, it is also possible to create a state space that consists only of QAPs without additional information, i.e., $S = \bigcup_{i=1}^{n} \mathrm{QAP}_i$. This is possible because the Koopmans-Beckmann QAP with linear term can be reduced to a problem of size $n - 1$ if one assignment pair is fixed: Let a QAP in the form of eq. (2.2) be given, defined by the objective value for an assignment $f$

$$V = \left( \sum_{i,j \in \mathcal{A}} a_{i,j} b_{f(i),f(j)} \right) + \left( \sum_{i \in \mathcal{A}} c_{i,f(i)} \right).$$

After assigning $f(u) := v$ the quadratic terms that contain $u$ are now of the form $a_{u,j} b_{v,f(j)}$ and now only depend on the assignment of the node $j$, so they reduce to

linear cost terms and can be added to $c_{j,f(j)}$. The same applies to the inverse edge pairs $a_{i,u}b_{f(i),v}$. The linear term for the assigned nodes $c_{u,v}$ is now constant. The remaining assignment objective can again be written as a QAP without $u$ and $v$,

$$V = \left( \sum_{i,j \in \mathcal{A} \setminus \{u\}} a_{i,j} b_{f(i),f(j)} \right) + \left( \sum_{i \in \mathcal{A} \setminus \{u\}} c'_{i,f(i)} \right) + d, \tag{3.1}$$

with the linear cost and constant terms

$$c'_{i,j} = a_{i,u}b_{j,v} + a_{u,i}b_{v,j} + c_{i,j} \qquad \forall i \in \mathcal{A} \setminus \{u\}, \forall j \in \mathcal{B} \setminus \{v\}, \tag{3.2}$$

$$d = c_{u,v}. \tag{3.3}$$

This is again a QAP of the same form as the original problem, and the objective value of the full assignment $f$ remains unchanged (see appendix A for a more formal derivation). This property also holds for the more general Lawler's QAP, as proven by Lawler [14]. Note that after assigning the last pair of nodes $V = d$. This means the constant term added in each step can be used as negative reward, since it adds up to the assignment value. Figure 3.1 illustrates the reduction with an example. It also gives an intuition of the reduction as a transformation of the QAP graph.

Using this reduction step as a transition function which is closed under the set of QAPs, we can now define a complete MDP:

**State space** Set of QAPs with linear term,

$$S = \bigcup_{i=0}^{\infty} \mathrm{QAP}_i \tag{3.4}$$

**Action space** Set of possible assignment pairs of nodes,

$$A(s) = \{1, \ldots, n\}^2, \tag{3.5}$$

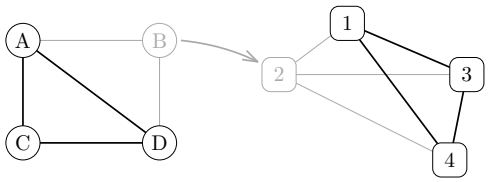
where $n$ is the size of the QAP.

**Transitions** Reduction of a QAP with a fixed assignment pair to a smaller QAP and a reward $-d$, with a transition function of the type

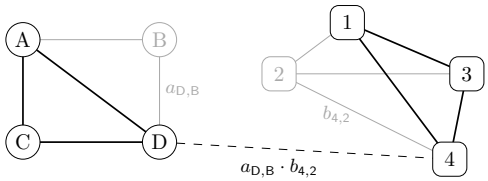
$$T : (\mathrm{QAP}_n, \{1, \ldots, n\}^2) \to (\mathrm{QAP}_{n-1}, \mathbb{R}), \tag{3.6}$$

Appendix B contains a formal definition for the transition function $T$.

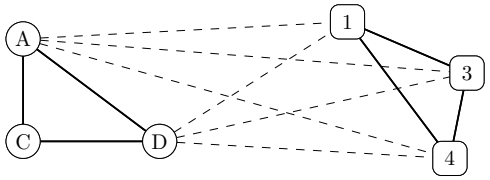
The initial state distribution determines the training data. It is possible to train the agent on a single fixed QAP, a fixed set of QAPs or a distribution of randomly generated QAPs.




(a) Node B is assigned to node 2. The edges around the now fixed nodes would become uninformative when more nodes are assigned. In the next steps, the remaining information is distilled into linear cost terms.




(b) The linear assignment cost for every possible pair of neighbors of B and 2 is computed. In the example, assigning D to 4 would directly incur the cost $a_{\mathsf{D},\mathsf{B}}b_{4,2}$ (in addition to the inverse $a_{\mathsf{B},\mathsf{D}}b_{2,4}$).




(c) After computing all linear cost terms, the fixed nodes are redundant and can be removed from the graphs. The linear terms can be viewed as weighted edges between the graphs.

Figure 3.1.: An illustration of the reduction step starting from a QAP without linear cost. While the reduction process adds many new weights in the first step, the subsequent steps only increase the value of existing linear weights while removing more nodes.

The advantage of this construction of the MDP is that it does not introduce artificial states. Instead, every intermediate state is part of the problem set that we are trying to solve, with less redundant information. Since the problem size is decreasing in every step, this also enables incremental learning, where the ability to solve an easy small problem helps to solve a harder larger problem. At the same time, training on problems of different sizes helps to achieve generalization at least to smaller sizes, but possibly also to larger problems to some extent.

## 3.2. Deep RL allows stochastic policies

In this section, we explain why a deep reinforcement learning system that splits the problem by sampling an individual pair at each step (such as our system or [15]) is in theory able to express arbitrary probability distributions over assignments, whereas methods based on a continuous relaxation of a permutation matrix (such as [29; 28; 4]) can only express a restricted set of distributions. Learning probability distributions is useful because it encodes uncertainty in the model during training, and allows sampling of multiple possible solutions at inference time. This motivates the use of deep reinforcement learning for the QAP and other permutation problems (in the QAP, a permutation of the nodes in one graph is also a valid assignment and vice versa, as long as both graphs have the same number of nodes).

In the following a permutation $p$ is considered as a set of pairs

$$p = \{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\} = \{p_1, p_2, \ldots, p_n\}. \tag{3.7}$$

In general, a probability distribution over permutations has the form

$$P(p) = P(p_1 \wedge p_2 \wedge \ldots \wedge p_n). \tag{3.8}$$

Explicitly representing such a distribution by assigning a probability to each possible input is practically impossible because there are $n!$ possible permutations.

Any $n \times n$ matrix contains only $n^2$ values. In particular, this means that a double stochastic matrix, as in a continuous QAP relaxation, cannot represent a general stochastic distribution over policies, but only distributions where the probabilities of all pairs are independent, i.e.,

$$P(p) = P(p_1) \cdot P(p_2) \cdot \ldots \cdot P(p_n). \tag{3.9}$$

However, a deep reinforcement system computes a sequence of conditional probabilities,

$$P(p) = P(p_1) \cdot P(p_2 \mid p_1) \cdot \ldots \cdot P(p_n \mid p_1 \wedge p_2 \wedge \ldots \wedge p_{n-1}), \tag{3.10}$$

which is a valid decomposition of joint distributions as in eq. (3.8). With a suitable policy network, an arbitrary permutation distribution can be approximated.

# 4. GNN network architecture

## 4.1. Basic architecture

Deep reinforcement learning algorithms learn the parameters of a differentiable policy function that receives the current state as input and outputs an action or assigns probabilities to all possible actions. In our setting, the state is represented as a graph, so we use a policy based on a graph neural network to encode the structure of the graph into node embedding vectors. Message passing graph neural networks create node embeddings by aggregating a function of their neighbor's embeddings. Applying this repeatedly passes information about the graph through the network.

By design, graph neural networks are *permutation equivariant*, which means that the computed embedding of each node is the same under any ordering of the nodes. Arguably more importantly, they work with graphs of arbitrary size, which is necessary since the decision process we formulated for our reinforcement learning pipeline generates problems of decreasing sizes as states, and we want to evaluate the system on larger graphs than it was trained on.

Our basic graph neural network layer has the form of a general message passing layer[5; 1]

$$\mathrm{GNN}(\mathbf{x}, \mathbf{e})_i = \psi\left(\mathbf{x}_i, \sum_{j \in \mathcal{N}(i)} \phi(\mathbf{x}_j, \mathbf{e}_{ji})\right) \tag{4.1}$$

where $\mathbf{x}_i$ is the initial feature vector for node $i$, $\mathbf{e}_{ij}$ is the feature vector of edge $(i, j)$ and $\mathcal{N}(i)$ are the neighbors of node $i$. We use the notation $\mathbf{M}_i$ to refer to a row of matrix $\mathbf{M}$, so the output of $\mathrm{GNN}(\mathbf{x}, \mathbf{e})$ is a matrix of node embeddings.

For the transformations $\psi$ and $\phi$ we will always use fully connected neural networks. Often, simpler functions for the inner transformation $\phi$ are used in graph neural networks for computational efficiency [11; 26], as it has to be evaluated $n^2$ times. However, to capture the structure of the problem without node features, we have to use a formulation that incorporates an expressive transformation of the edge weights with a neural network. This is slower and more memory intensive, especially when training on larger graphs. Training this architecture on small graphs is still viable, and we found that it is sufficient for our purposes. The forward propagation runtime of one message passing layer in terms of the number of nodes $n$ is in $\mathcal{O}(n^2)$, with a constant factor depending on the hyperparameters of the neural networks and the size of the feature vectors.

With the neural network layer defined for one graph, we can extend it to a QAP

consisting of two graphs and linear costs. We apply graph neural networks with the same structure but separate parameters to both graphs. To incorporate the linear cost term into the network, we use a heterogeneous architecture. The function computed by one layer of the heterogeneous network is the following ($\mathbf{x}$ and $\mathbf{y}$ denote the initial node embedding matrices of the two graphs, respectively; $\mathbf{A} = [a_{ij}]$, $\mathbf{B} = [b_{ij}]$ and $\mathbf{C} = [c_{ij}]$ are matrices of the QAP weights as in equation (2.2); $\mathrm{GNN_i}$ are graph neural network layers as described above):

$$\begin{aligned} \mathbf{x}' &= \mathbf{x} + \mathrm{GNN}_1(\mathbf{x}, \mathbf{A}) + \mathrm{GNN}_2(\mathbf{y}, \mathbf{C}) \\ \mathbf{y}' &= \mathbf{y} + \mathrm{GNN}_3(\mathbf{y}, \mathbf{B}) + \mathrm{GNN}_4(\mathbf{x}, \mathbf{C}^T) \end{aligned} \tag{4.2}$$

This operation can be applied repeatedly. Note that the heterogeneous architecture transfers information between the two graphs. A similar operation is the CrossConv used in the graph matching network of [28].

After the GNN layer, we use the node embeddings $x$ and $y$ to compute a value for every pair of nodes between the graphs by creating the Cartesian product

$$\forall i, j \in \{1, \ldots, n\} : \mathbf{k}_{i,j} = \mathrm{concat}(\mathbf{x}_i, \mathbf{y}_j), \tag{4.3}$$

where $\mathrm{concat}(a, b)$ represents concatenation of $a$ and $b$ along the feature dimension.

Depending on the algorithm, we use a final neural network output head $f$ in different ways. In the value-based algorithms DQN and MCQ, it estimates the state-action-value

$$\hat{q}(s, (i, j)) = f(\mathbf{k}_{i,j}) \tag{4.4}$$

for the input QAP state $s$. In the policy-gradient algorithms REINFORCE and A2C the output of $f$ is used as the log action probability

$$\log \hat{\pi}(s, (i, j)) = \log f(\mathbf{k}_{i,j}) - \alpha, \tag{4.5}$$

with a normalization offset $\alpha$ such that the probabilities $\hat{\pi}(s, \cdot)$ sum to 1. The composed architecture of an action-value or policy network is illustrated in figure 4.1.

Additionally, in A2C the state-value $v(s)$ has to be estimated with a critic network. It uses a second GNN, separate from the policy network, which outputs node embeddings $\mathbf{x}_i^v$ and $\mathbf{y}_i^v$. It does not concatenate pairs, but instead performs a global sum pooling over the embeddings of the two graphs and computes the state value estimate with a learned neural network transformation $f^v$:

$$\hat{v}(s) = f^v \left( \sum_{i=1}^{n} \mathbf{x}_i^v + \sum_{i=1}^{n} \mathbf{y}_i^v \right). \tag{4.6}$$

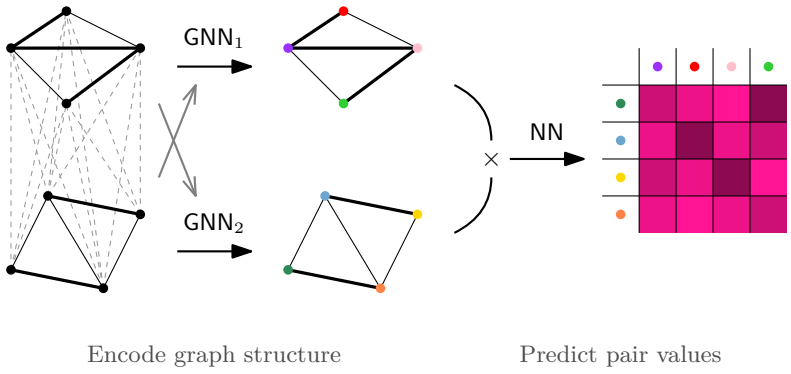



Figure 4.1.: Overview of the GNN-based policy architecture

## 4.2. Separating node embeddings

After training REINFORCE with the architecture described above initially failed to produce results distinguishable from randomness, we found that node embeddings might be very similar when a uniform random adjacency matrix is given as input to a randomly initialized network (see section 4.2.1 for a simple experiment demonstrating the issue in isolation and section 5.5 for results of REINFORCE). With multiple message passing layers the embeddings becomes even more similar, an effect sometimes called *oversmoothing* [32].

During training this causes undesired symmetries where the network cannot distinguish all nodes. Learning to separate these nodes is very slow, likely because this symmetry represents a saddle point in the learning landscape, where different directions of decreasing loss cancel out.

A symptom of this effect was that running REINFORCE on a fixed single problem converged to a stochastic policy that uniformly generated multiple solutions with different objective values, whereas we expected the probability of the better solution to increase. Since the network could not distinguish all nodes, it could also not assign different probabilities to solutions where such nodes were exchanged.

To work around such problems, we try to add a *separation layer* to allow the message passing layer to break this symmetry. In our implementation, we add such a separation layer $f$ into the GNN layer from eq. (4.1) as

$$A(\mathbf{x}, \mathbf{e})_i = \sum_{j \in \mathcal{N}(i)} \phi(\mathbf{x}_j, \mathbf{e}_{ji}), \tag{4.7}$$

$$A'(\mathbf{x}, \mathbf{e}) = f(A(\mathbf{x}, \mathbf{e})), \tag{4.8}$$

$$\mathrm{GNN}_f(\mathbf{x}, \mathbf{e})_i = \psi\left(\mathbf{x}_i, A'(\mathbf{x}, \mathbf{e})_i\right). \tag{4.9}$$

Unlike the learned transformations, the separation layer can perform operations on all

nodes rather than only on individual nodes. We will describe and compare multiple different functions that could serve as separation layer in the following.

Zhao and Akoglu [32] enforce fixed mean distance of the node embeddings by centering and rescaling the embeddings. For an embedding matrix $\mathbf{x} \in \mathbb{R}^{n \times d}$ they define a PairNorm layer as

$$\mathsf{PairNorm}(\mathbf{x})_i = s \cdot \frac{\mathbf{x}_i - \bar{\mathbf{x}}}{\sqrt{\frac{1}{n}\sum_{j=1}^{n} \|\mathbf{x}_j - \bar{\mathbf{x}}\|^2}}, \tag{4.10}$$

where $s$ is a constant. The vector $\bar{\mathbf{x}} \in \mathbb{R}^d$ is the mean of $\mathbf{x}$ over all rows (individual node embeddings),

$$\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i. \tag{4.11}$$

While it does improve separation, we observed that PairNorm downgrades performance when applied in DQN (see section 5.5). Subtracting the mean of all node embeddings might remove absolute information that is necessary to learn a value function. Instead, only rescaling the differences and not centering the node embeddings yields the same distances between node embeddings:

$$\mathsf{KeepMeanNorm}(\mathbf{x})_i = s \cdot \frac{\mathbf{x}_i - \bar{\mathbf{x}}}{\sqrt{\frac{1}{n}\sum_{j=1}^{n} \|\mathbf{x}_j - \bar{\mathbf{x}}\|^2}} + \bar{\mathbf{x}}. \tag{4.12}$$

Furthermore, we can generalize PairNorm and KeepMeanNorm to use the mean as global information by adding a learnable transformation to the mean term, leading to

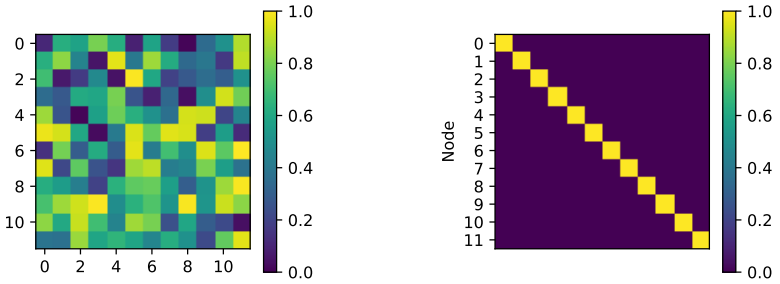
$$\mathsf{TransformedMeanNorm}(\mathbf{x})_i = s \cdot \frac{\mathbf{x}_i - \bar{\mathbf{x}}}{\sqrt{\frac{1}{n}\sum_{j=1}^{n} \|\mathbf{x}_j - \bar{\mathbf{x}}\|^2}} + \mathrm{ReLU}(\mathbf{W}\bar{\mathbf{x}} + \mathbf{b}), \tag{4.13}$$

with the rectifier activation function $\mathrm{ReLU}(x) = \max(x, 0)$, as well as the learnable parameters $\mathbf{W} \in \mathbb{R}^{d \times d}$ and $\mathbf{b} \in \mathbb{R}^d$. Finally, we tried to replace the rescaling term with a learned transformation as well,

$$\mathsf{MeanSeparation}(\mathbf{x})_i = s \cdot \mathrm{ReLU}(\mathbf{V}\mathbf{x}_i + \mathbf{c}) + \mathrm{ReLU}(\mathbf{W}\bar{\mathbf{x}} + \mathbf{b}), \tag{4.14}$$

with the additional learnable parameters $\mathbf{V} \in \mathbb{R}^{d \times d}$ and $\mathbf{c} \in \mathbb{R}^d$. Due to the ReLU activation function, it is possible to completely separate components of the mean and of the residual in the output, but MeanSeparation does not perform explicit normalization anymore.

The scaling factor $s$ can be used to tune the initial scale of the residual compared to the mean. It likely has a similar effect in all layers described above, even though we only used it in MeanSeparation. Unless otherwise specified, it is set to 1.




(a) Input edge matrix (b) Target node embedding matrix

Figure 4.2.: Visualization of the experiment setup in section 4.2.1.

### 4.2.1. Experimental analysis: Identifying nodes

These separation layers define a spectrum of different ideas, between maintaining the distance of the node embeddings to separating the mean from the residual, which allows us to analyze different potential sources of the problems in separating node embeddings with our graph neural network.

To demonstrate the issue and the effect of our proposed separation layers in a simple, reproducible way, we ran the following experiment: We generate a $12 \times 12$ matrix with uniform random entries in the range $[0, 1]$ as edge weight matrix. Then, we train a single message passing layer to identify each node on this input. More precisely, we use an MSE loss with a $12 \times 12$ diagonal matrix with ones on the diagonal as a target node embedding matrix, i.e., a one-hot encoding of the node index. Since a message passing network performs the same operations on all nodes, this shows us how well it can distinguish the nodes.

The network is trained with different separation layers integrated as described in eq. (4.7). For all configurations, the parameters of the common linear layers are initialized to the same values. We use the Adam optimizer with learning rate 5e-3. We want to compare how long the network takes to learn to identify nodes on a fixed input.

Interestingly, the results shown in figs. 4.3 and 4.4 indicate that centering the node embeddings is an important factor to learning separation, perhaps more important than distance between node embeddings. This can be seen most notably by the discrepancy between PairNorm and KeepMeanNorm.

Judging from these results, the effect might be related to the ratio between mean and standard deviation after the aggregation step. We illustrate this with a simple model. Assume a random adjacency matrix defined by independent random edge weights $E_{i,j}$ that are identically distributed with $\mathbb{E}(E_{i,j}) = \mu$ and $\mathrm{Var}(E_{i,j}) = \sigma^2$. Simplifying the transformations $\phi$ and $\psi$ in the message passing layer, we can model the aggregation of the neighboring edge weights of one node over a random matrix as a sum of random

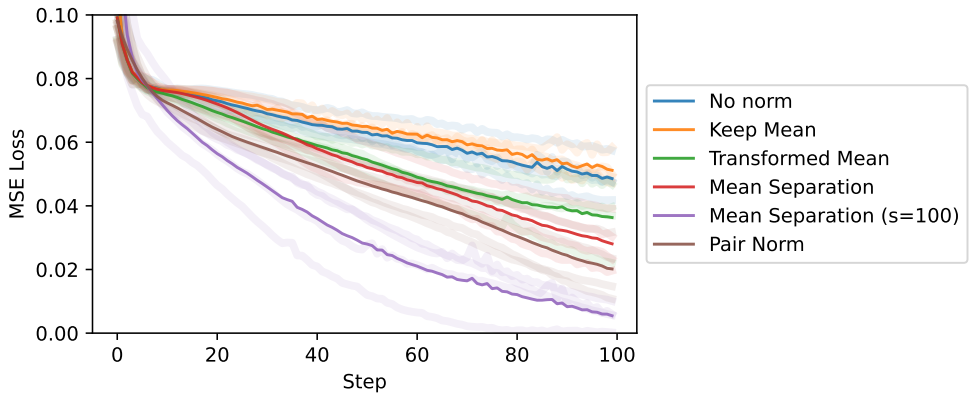



Figure 4.3.: Training loss for different separation layers in the synthetic experiment described in section 4.2.1. Depicted is the mean over 4 initialization seeds, blurred lines indicate runs on individual seeds. Note that for all runs, the loss decreases rapidly from a high initial value (beyond the range of the plot) until the bias matches the mean of the target. After this point, the loss decreases only very slowly without separation layers.

variables

$$Y_i = \sum_{j=1}^{n} E_{i,j},$$

that has expectation

$$\mathbb{E}(Y_i) = \sum_{j=1}^{n} \mathbb{E}(E_{i,j}) = \mu n$$

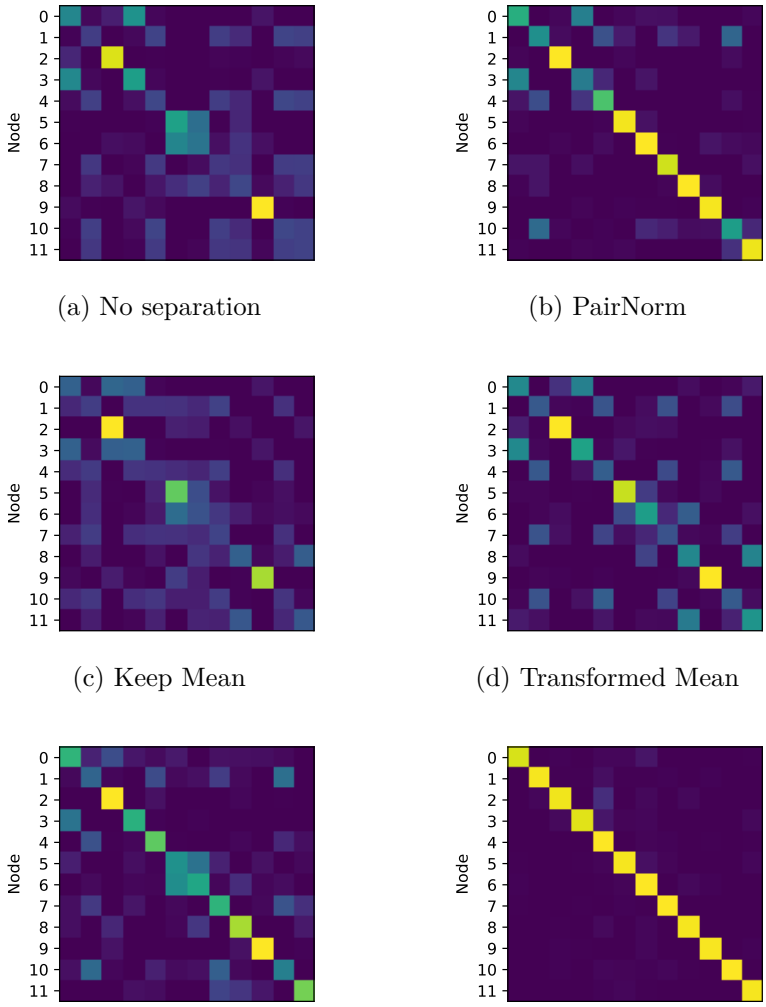
and a standard deviation

$$\sqrt{\mathrm{Var}(Y_i)} = \sqrt{\sum_{j=1}^{n} \mathrm{Var}(E_{i,j})} = \sigma\sqrt{n}.$$

Since the expectation grows with the factor $n$ and the standard deviation only with the factor $\sqrt{n}$, it will dominate the output for larger $n$. Thereby the output becomes more similar relative to its mean. Of course, this does not yet prove the impact on the gradient. An in-depth of this effect could yield useful insights in the future. For now, we will rely on our experimental results.

## 4.3. Expressiveness limitations

Another potential issue is that our graph neural network is limited in its expressiveness. Any message passing rule in the framework of eq. (4.1) described above cannot distinguish some structures [31; 22]. As an example, consider two graphs, a cycle of 3, and a cycle




(a) No separation

(b) PairNorm

(c) Keep Mean

(d) Transformed Mean

(e) Mean Separation

(f) Mean Separation ($s = 100$)

Figure 4.4.: Matrices of node embeddings in the experiment in section 4.2.1 after training for 100 steps with and without separation layer. Indistinguishable node embeddings are the main reason why the network did not match the target correctly: For almost every incorrect node embedding there is another very similar embedding (e.g., see node 0 and 3 in the examples shown). Adding a separation layer makes more nodes distinguishable and improves learning performance.

of 4 nodes. In both graphs, let all edge weights be fixed to 1 and the initial node embeddings be 0. Since all nodes have 2 neighbors, a sum aggregation will return the same value for all nodes. The final node embeddings on both graphs are identical, despite them being structurally different (non-isomorphic).

However, to solve QAPs in general, the agent would have to be able to solve the isomorphism problem [17]. It is possible to construct an equivalent QAP for every isomorphism problem on unweighted graphs: Let $a_{ij}, b_{kl} \in \{0, 1\}$ be the adjacency matrices of two graphs. Then the QAP

$$V = \sum_{i,j} a_{ij}(1 - b_{f(i),f(j)}) \tag{4.15}$$

has the optimal value $V = 0$ iff the graphs are isomorphic.

Overcoming the issue of expressiveness is an active area of research. Several options have been proposed, such as higher order GNNs ([20]) or random node features [22]. A different approach is to use local search, such as MCTS [3; 23], in the reinforcement learning algorithm, since it will be able to distinguish actions by the future feedback from the environment, as well as improving the quality of the output.

Note that these expressiveness issues are not necessarily a problem when we are using QAPs with randomly generated weights, as almost all generated graphs will have unique edge weights adjacent to each node. For this reason, we believe that oversmoothing due to the GNN aggregation as described in section 4.2 is more important, although methods with increased theoretical expressiveness might also remedy this issue. Since it is beyond the scope of this thesis, we leave the study of such methods to future work.

# 5. Experiments

## 5.1. Training and evaluation problem generation

We train and evaluate our agent on random artificial QAPs. The QAPs are generated as two undirected $n$-node graphs without self-loops (symmetric $n \times n$ matrices with 0 diagonal). The edge weights are chosen uniformly at random between 0 and 1, the initial linear cost is set to 0.

While such synthetic random problems are not very realistic, this is a good way to check generalization to different problem sizes, since problems of arbitrary size from the same distribution can be generated easily. In fact, the only way an agent can get good results even on the training task is by learning general rules (as opposed to overfitting to specific inputs), since no problem is ever seen twice. Furthermore, some instances in QAPLIB (which we use for evaluation in section 5.7) have been generated from such a uniform random distribution, including the currently smallest unsolved instance `tai35a`.

We trained our MCQ, DQN, REINFORCE and A2C algorithms on randomly generated QAPs of size 8. Initially we used only 8-node QAPs for training for performance reasons, but later training on 16-node QAPs did not significantly improve the results (see fig. 5.2b).

All agents use the same neural network architecture, except for the separation layer. The network consists of 2 heterogeneous message passing layers (eq. (4.2)), that each use an edge transformation network ($\psi$ in terms of eq. (4.1)) with 2 hidden layers, a single node transformation layer ($\phi$) without hidden layers. The final output transformation ($f$ in eqs. (4.4) and (4.5)) uses 3 hidden layers. The size of all embeddings and hidden vectors is 32. The only hyperparameters that differ between the algorithms are learning rate and the type of separation layer. For each algorithm, the seed and hyperparameter combination with the lowest objective value on the training set is used for evaluation. The hyperparameters chosen for the evaluations in this chapter (unless otherwise specified) are shown in table table 5.1.

## 5.2. Benchmark solvers

The minimum requirement for an agent is that it finds better assignments than a uniformly random policy does. Additionally, we use a simple baseline "Last-8-Opt" to prove generalization ability of our agents. It does random assignment steps until the

remaining QAP has size at most 8. Then, it uses a naive algorithm that solves the remaining subproblem to optimality by enumerating all possible solutions. This serves to verify whether an RL agent trained on $\text{QAP}_8$ and evaluated on problems with size $n > 8$ actually learned useful decisions rules on larger instances. Otherwise, the agent could make arbitrary choices and only solve the subproblem of size 8, while still beating randomness.

To get good reference values for timing and objective value, we used the FAQ (Fast Approximate Quadratic assignment) algorithm [27] implemented in the python library `scipy`[1] (version 1.8.0), which computes a local optimum to a relaxation of the Koopmans-Beckmann QAP. It has a runtime in $\mathcal{O}(n^3)$, the same asymptotic complexity as our system.

We also tried the proprietary optimizer Gurobi [7] (via `gurobipy` version 9.5.1) with a quadratic integer programming formulation with binary variables $x_{i,u}$[12]:

$$\begin{aligned} \text{minimize } & \sum_{i,j \in \mathcal{A}} \sum_{u,v \in \mathcal{B}} a_{i,j} b_{u,v} x_{i,u} x_{j,v}, \\ \text{subject to } & \forall i \in \mathcal{A} : \forall u \in \mathcal{B} : x_{i,u} \in \{0,1\}, \\ & \forall i \in \mathcal{A} : \sum_{j \in \mathcal{B}} x_i j = 1, \\ & \forall j \in \mathcal{B} : \sum_{i \in \mathcal{A}} x_i j = 1. \end{aligned} \tag{5.1}$$

Constructing the model with the Python interface was too slow for problems of size $n \geq 32$ with this formulation, since it contains a sum of $n^4$ terms. Even on problems of size 16 and 24 where the model could still be constructed within a few minutes, with a solving time limit of 5 seconds it performed worse than FAQ (whereas FAQ ran only for a few milliseconds). Thus, we only use FAQ as a baseline.

All experiments were performed on a virtual machine with 8 assigned virtual cores running on an AMD EPYC 7551 processor and no GPU acceleration.

[1] `https://docs.scipy.org/doc/scipy/reference/optimize.qap-faq.html`

| Agent | Learning rate | Separation layer |
|---|---|---|
| MCQ | 3e-4 | Mean Separation ($s = 100$) |
| DQN | 5e-4 | Mean Separation ($s = 1$) |
| REINFORCE | 1e-4 | Mean Separation ($s = 100$) |
| A2C (Actor) | 4e-5 | Mean Separation ($s = 100$) |
| A2C (Critic) | 4e-5 | None |

Table 5.1.: Hyperparameters used for evaluation

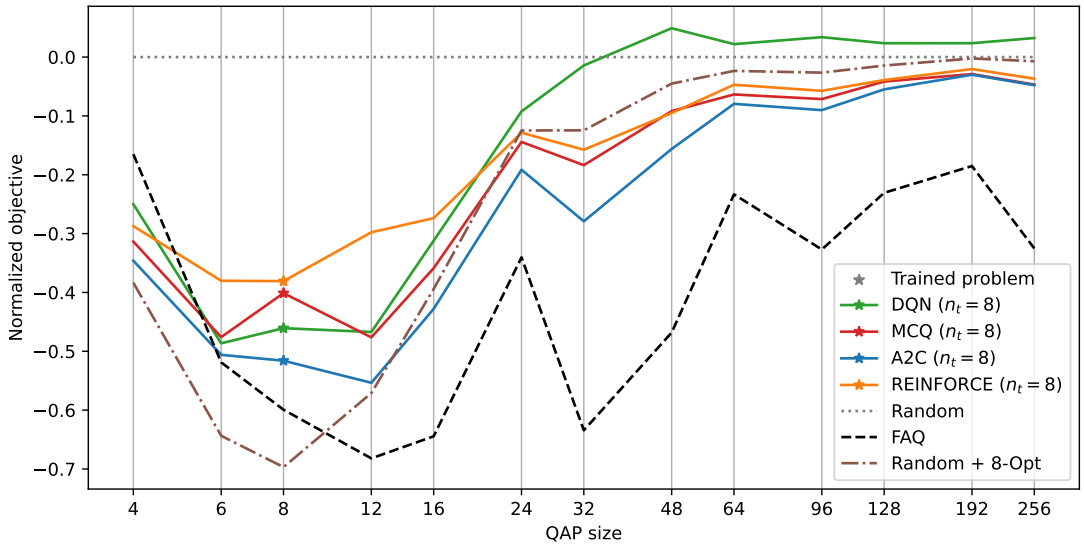



Figure 5.1.: Comparison of our RL agents trained on random 8 node QAPs with FAQ solver and Last-8-Opt baseline (referred to as Random+8-Opt in the legend), normalized according to eq. (5.2), lower is better. The agents were evaluated on 20 randomly generated problems for each size. While our agents cannot match the results of FAQ for larger graphs, the comparison with Last 8 shows that they exhibit generalization beyond the problems they were trained on.

## 5.3. Results on QAPs of different sizes

We test the trained agents on randomly generated QAPs of size up to 256. For each size, we compute the objective value $v$ on 20 problems and normalize it to the random policy as

$$\text{normalized objective} = \frac{\text{mean}(v) - \text{mean}(v_{\text{random}})}{\text{variance}(v_{\text{random}})}, \tag{5.2}$$

where mean and variance are computed over the 20 problems. The normalization makes the quadratically growing values easier to compare and plot across different sizes.

Figure 5.1 shows the results. MCQ, A2C and REINFORCE can generalize to some degree to larger problems than they were trained on. A2C consistently performs better than all the other RL algorithms.

DQN performs worse than the other algorithms at generalization on QAPs with size $n > 24$, notably MCQ (a value-based algorithm, like DQN) and A2C (a bootstrapping algorithm, like DQN). Note that a key aspect in which our DQN implementation differs from the other algorithms is that it does off-policy training with a replay buffer, which does not seem to be beneficial in our setup. With randomly generated problems, the

advantage of using a separate exploration strategy is small (section 5.4) and the episodes are not strongly correlated, which diminishes the advantages of off-policy training.

It is not clear whether learning a value function by bootstrapping is useful in our case of small problems from these results. Our REINFORCE implementation does not use a learned value function as baseline, so unfortunately we cannot conclude whether the bootstrapping done in the A2C algorithm is a major advantage over REINFORCE, or whether it obtains better results due to the better baseline.

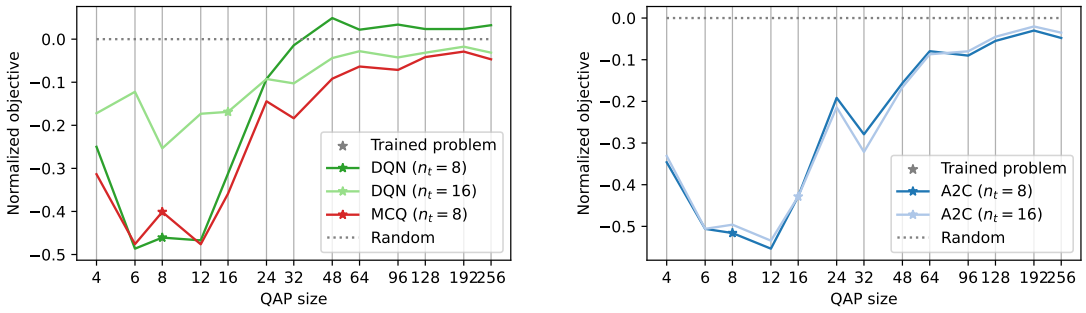
Given the size of the gap in the solution objective value between the reinforcement learning algorithms and the FAQ solver on large problems, it is clear that our system cannot currently compete with state-of-the-art solvers for the QAP, especially since FAQ also has a lower runtime (fig. 5.3). Our graph neural network architecture is likely unsuitable for graphs with high degree, as the sum aggregation over many neighbors makes it difficult to preserve information about individual connections. This could also explain that training our best algorithm, A2C, on 16 node QAPs compared to training on only 8 node QAPs did not significantly improve the results as shown in fig. 5.2b.

## 5.4. Exploration on random graphs

In many reinforcement learning tasks, it is important to explore sufficiently to learn about the environment. However, with random training graphs, the agent is forced to see different states. so using an explicit exploration strategy is not necessary. We evaluated this on the DQN agent by comparing a purely greedy policy to an $\epsilon$-greedy policy with $\epsilon$ decaying from 0.5 to 0.005. The results in fig. 5.4 show that $\epsilon$-greedy does not significantly improve performance of DQN, and it is still worse than MCQ, the simpler value-based algorithm, with a greedy policy (we did not try running MCQ with $\epsilon$-greedy policy).

## 5.5. Effect of separation layers on the combined system

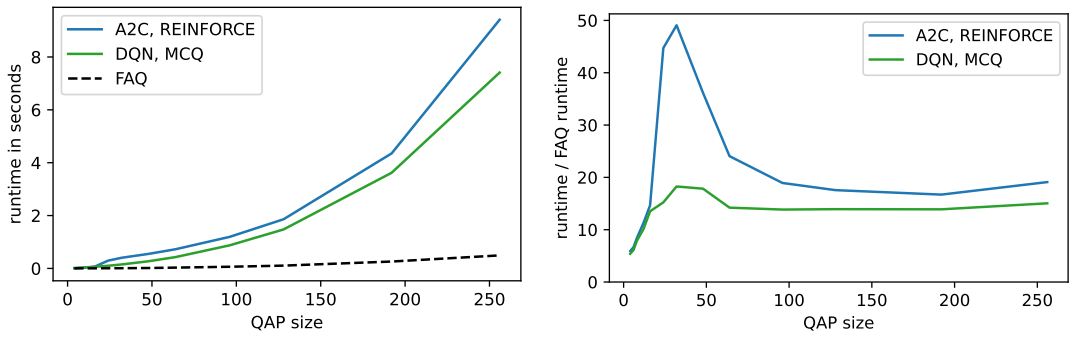
In section 4.2.1 we tested different separation layers on a simple node labeling task. We want to demonstrate stronger separability also improves the performance when integrated in the GNN policy of a reinforcement learning algorithm. For this purpose, we train the value-based DQN algorithm and the policy-gradient algorithms REINFORCE and A2C on random QAPs of size $n = 8$ with different separation layers. Figure 5.5 shows curves of the training progress. While the same could be done with MCQ, we did not study the effect of separation layers on it at a similar scale (although it is likely to benefit from the separation due to the high-variance Monte Carlo gradients, similar to REINFORCE).




(a) Monte-Carlo Q-Learning outperforms DQN even if DQN was trained on $\text{QAP}_{16}$.

(b) Training on $\text{QAP}_{16}$ does not improve the results of A2C, even in $\text{QAP}_{16}$ itself.

Figure 5.2.: Results when training on random QAPs of size 16. The plots use the same data as fig. 5.1 (lower is better). In the legends, $n_t$ indicates the size of problems during training.




(a) Absolute runtime

(b) Factor over FAQ runtime

Figure 5.3.: Comparison of evaluation runtime between FAQ and our RL agents (lower is better). The overhead of the policy-gradient algorithms is the sampling step.

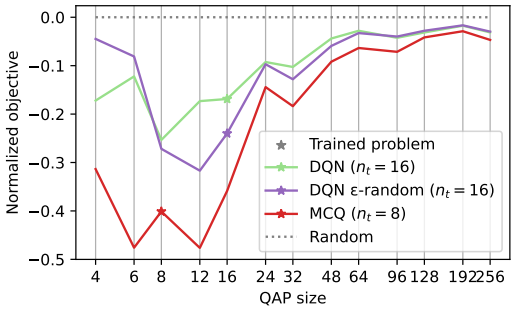
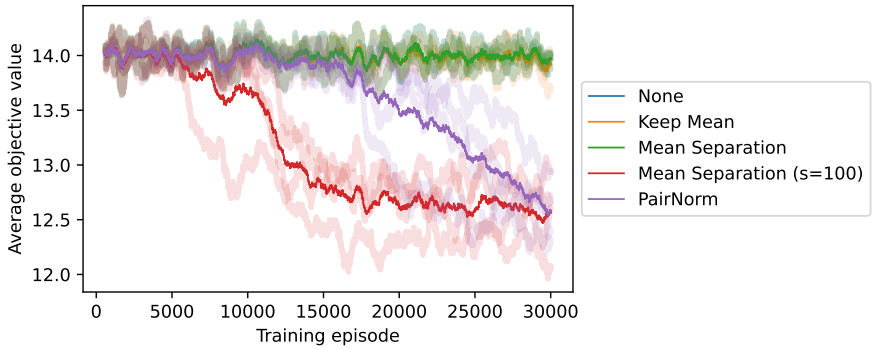
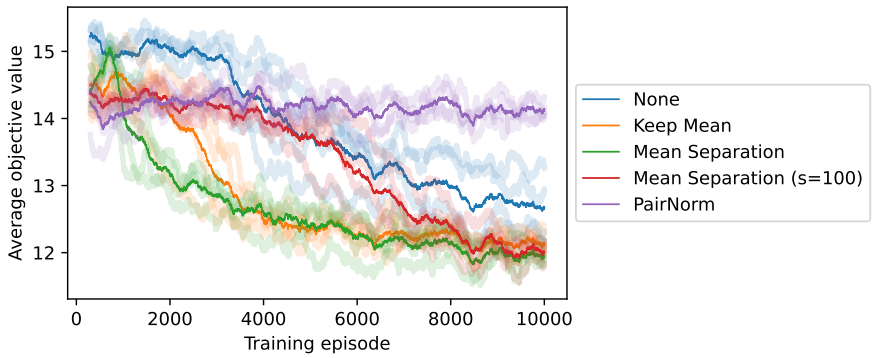


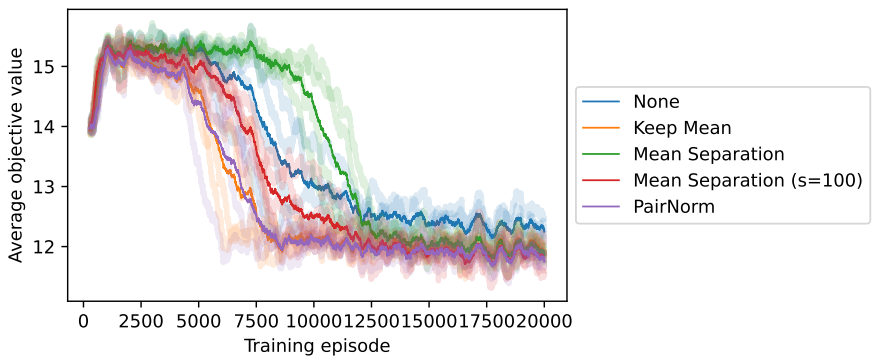

Figure 5.4.: Using an $\epsilon$-greedy policy in DQN changes the results only slightly compared to a greedy policy.




(a) Training objective value of REINFORCE, moving average over 600 steps. Here, *None*, *Keep Mean* and *Mean Separation* all stay at 14, the average value of random assignments.




(b) Training objective value of DQN, moving average over 300 steps.




(c) Training objective value of A2C, moving average over 300 steps. The separation layer is only used in the actor (policy), and not in the critic (state-value) network.

Figure 5.5.: Objective value during training of different RL algorithms on random 8 node QAPs with different separation layers (lower is better). Mean over 4 random seeds, individual runs shown as blurred lines.

REINFORCE is very susceptible to not being able to distinguish node embeddings, possibly due to the high variance gradients it generates. In fact, without enforced separation the policy even remains uniformly random. Only with PairNorm or scaled Mean Separation it can learn different probabilities per pair. These separation layers also performed best in section 4.2.1, showing that the simplified setup for identifying nodes is a useful model to study in isolation.

While DQN and A2C can be trained without a separation layer, they converge faster and to better values with some norm layers. The exception is DQN with PairNorm, which might remove too much information to learn a value function (as discussed in section 4.2). Compared to REINFORCE and the experiment in section 4.2.1, the effect on the result is different. For example, KeepMeanNorm performs better than no separation layer. It is possible that here the normalization and separation contribute to stabilizing the internal distribution between neural network layers, as observed with batch normalization [9], instead of only distinguishing node embeddings.

We also tested the performance at the generalization task of different separation layers in DQN and the actor network of A2C from section 5.3 (since REINFORCE requires a norm to learn anything meaningful, we do not test it in this experiment). The plots are found in appendix D. The separation layers seem to improve generalization on small to medium-sized (12-32) problems to some extent. However, the main contribution and intended purpose of the separation layers is improving training convergence, especially in REINFORCE.

## 5.6. Ablation study: State representations

Since the reduction performed as part of the closed transition function in our decision process for the QAP is an important piece of our system, we tested whether we could achieve good results with other decision process models.

As a comparison that does not use reduction, we defined a different MDP for the QAP (without linear term) where the graphs remain unmodified after an assignment, and instead only an edge between two assigned nodes is added in each step. The links can still be written with a matrix with binary entries, so we can again use the heterogeneous GNN architecture from eq. (4.2), trained with DQN.

The results in fig. 5.6 show that the DQN agent does not learn well in the link-adding MDP with random graphs. This is likely due to the fact that the actions cause only weak feedback by an added edge, which does not influence the graph neural network much. In contrast, the reduction MDP creates successively smaller problems, which helps with bootstrapping the value function (since the value function will be easier to estimate for smaller problems) in DQN. In general, the reduction MDP can be viewed as a way to let the RL agent learn incrementally by transferring knowledge from easier problems to

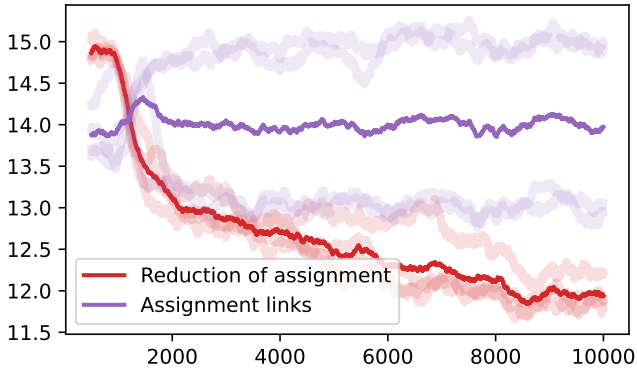



Figure 5.6.: Training objective value of DQN trained with the reduction MDP described in section 3.1 compared to a simpler MDP that adds links between assigned nodes (lower is better). Moving average over 500 episodes.

more difficult problems. This should help with generalization, which is important both for good results on the training task and out-of-distribution problems. Furthermore, the linear cost also explicitly gives the agent information about the immediate reward of an action, so the action value should be easier to learn for the agent, at least for one step ahead.

## 5.7. Evaluation on QAPLIB

Beyond just uniform random QAPs, we evaluate our system on QAPLIB[2] [2], a collection of Koopmans-Beckmann QAP instances (no linear term) from various sources. It includes synthetic problems from a variety of generation methods, but also problems from applications such as hospital layout planning and keyboard layout optimization. This should give some insights into the performance on problems with special structure beyond the uniform random QAPs we considered before.

Again, we use the RL agents trained on random 8-node QAPs for evaluation (the same as in fig. 5.1). To bring the QAP weights of the problems to a similar scale as the training data, we perform a normalization by shifting and scaling the input. Given the weights $a_{i,j}$ and $b_{i,j}$, the normalized weights are computed as

$$\begin{aligned} a'_{i,j} &= \frac{a_{i,j} - \min a_{i,j}}{\max a_{i,j} - \min a_{i,j}}, \\ b'_{i,j} &= \frac{b_{i,j} - \min b_{i,j}}{\max b_{i,j} - \min b_{i,j}}. \end{aligned} \tag{5.3}$$

[2]We use the data and best known solutions from `http://www.mgi.polymtl.ca/anjos/qaplib/inst.html`.

This does not change the ordering of solutions (the short proof can be found in appendix C).

In most problems of QAPLIB, the structure of the two input graphs can be very different, but the neural network in our system is not symmetric in the order of these two graphs. We improve the robustness of our system for the QAPLIB evaluation by running them both on the original and a version with swapped inputs, and taking the better result. This doubles the runtime, but could easily be parallelized. A comparison between the results of the original and swapped version over different classes can be found in appendix E.2. It would be interesting to see whether a "siamese" architecture where both graphs share the same weights yields better results or whether it is less flexible, but we did not try it and remain with the asymmetric variant.

In addition to FAQ, we also compare our agents to the reported results of a different deep learning QAP solver, NGM [29]. We only use the values of their best model NGM-G5K, which samples 5000 assignments from the neural network output. Unlike our agents, it was trained directly on the instances of each class in QAPLIB. Furthermore, we compute the minimal value of 10000 random assignments as a baseline to beat.

We measure the performance in terms of the gap of an assignment value $v$ to the best known value $v_{\text{known}}$,

$$\text{gap} = \frac{v}{v_{\text{known}}} - 1. \tag{5.4}$$

The mean gap of instances up to size 150 per class is shown in table 5.2 (NGM does not have results on larger problems), results per instance are listed in appendix E.1. On most classes FAQ is the best among all solvers, but on the classes `els` (which is in fact only a single problem), `esc` and `scr` one of the RL algorithms performs better than FAQ. A2C is pairwise better than any of the other learning-based algorithms on the majority of categories, including NGM-G5K, which appears to rely heavily on its sampling step, judging by the correlation of its results to the random sampling baseline.

Finally, note that QAPLIB was designed as a challenge for combinatorial solvers, rather than a test set for approximate machine learning algorithms. In many classes, weight matrices (or submatrices thereof) are reused in different combinations in multiple problems. This correlation can be problematic for an accurate evaluation, since, for example, errors in the encoding of one graph in the graph neural network can accumulate (taking the minimum with the swapped problem might make it slightly more robust). Nevertheless, the categories of QAPLIB are more diverse than our random data, and as the arguably most popular benchmark for the QAP, evaluation on QAPLIB gives us comparable results among solvers for QAP.

| | FAQ | MCQ | DQN | RF | A2C | NGM | 10k rand |
|---|---|---|---|---|---|---|---|
| bur 26-26 | **0.2** | 3.0 | 14.4 | 4.0 | 3.8 | 3.4 | 3.1 |
| chr 12-25 | **54.9** | 125.3 | 70.4 | 87.9 | 74.5 | 121.3 | 130.5 |
| els 19-19 | 23.7 | **9.7** | 132.2 | 132.2 | 98.9 | 57.0 | 56.5 |
| esc 16-128 | 32.0 | 22.1 | **12.6** | 19.8 | 17.8 | 32.0 | 28.8 |
| had 12-20 | **0.8** | 8.0 | 9.4 | 4.0 | 2.9 | 4.4 | 4.5 |
| kra 30-32 | **5.7** | 20.5 | 28.9 | 21.7 | 22.0 | 31.4 | 31.4 |
| lipa 20-90 | **2.5** | 14.4 | 14.2 | 13.4 | 12.0 | 14.1 | 14.1 |
| nug 12-30 | **3.0** | 18.9 | 20.6 | 12.3 | 12.2 | 16.3 | 15.4 |
| rou 12-20 | **3.8** | 18.7 | 14.0 | 14.3 | 14.2 | 13.1 | 11.2 |
| scr 12-20 | 17.4 | 36.9 | 21.1 | 28.1 | **16.9** | 30.2 | 30.2 |
| sko 42-100 | **1.3** | 15.2 | 21.5 | 14.0 | 11.8 | 18.1 | 17.3 |
| ste 36-36 | **7.0** | 61.0 | 141.1 | 51.8 | 66.6 | 102.2 | 108.0 |
| tai 12-150 | **7.0** | 18.8 | 63.4 | 45.5 | 40.9 | 22.2 | 22.9 |
| tho 30-150 | **2.4** | 24.7 | 32.8 | 22.8 | 19.0 | 25.3 | 26.0 |
| wil 50-100 | **0.8** | 10.0 | 9.9 | 6.6 | 6.9 | 9.4 | 8.8 |
| mean | **10.8** | 27.2 | 40.4 | 31.9 | 28.0 | 33.4 | 33.9 |

Table 5.2.: Mean gap to best known value (in %) on different QAPLIB classes, colored by gap normalized per category. The numbers under the classes indicates the range of the size of tested problems. Bold is the best solver, underlined is the best among all except the benchmark solver FAQ. Here, MCQ, DQN, REINFORCE and A2C are run both on the original and the QAP with both graphs swapped, the minimum is taken for each instance. "10k rand" is the minimum over 10000 randomly sampled assignments. NGM results are taken from `https://thinklab.sjtu.edu.cn/project/NGM/index.html`.

# 6. Conclusion and future prospects

In this thesis, we considered several aspects of a reinforcement learning solver for the quadratic assignment problem that help to get good results both on training data and on problems of size and structure unseen in the training data.

First, we construct an MDP that performs a simple transformation of a QAP with a fixed partial assignment to a smaller QAP with free assignment, by reducing redundant information of previously assigned nodes. This has the advantage that it does not add artificial states to the state space, but instead all states are problems that could also serve as initial input to the system, and the states in each episode become smaller and easier to solve in each step. This is an important part of what allows our system to achieve generalization and could also be a useful principle to consider when designing sequential decision processes that incrementally construct solutions in other problems.

While training our system with a graph neural network policy, we discovered that very similar node embeddings cause slow training, which we discussed in section 4.2. We improve training performance with special separation layers. This issue deserves more study in isolation, a theoretical analysis could yield insights to improve GNNs in general.

Although there is still a considerable gap to good learning-free QAP solvers in terms of the objective value with comparable runtime on most problems, we demonstrate generalization ability of our system both to larger random problems and problems in QAPLIB, while training only on small random problems with 8 nodes. Our best algorithm A2C outperforms the recent deep learning-based solver NGM. We found that when training on randomly generated inputs, exploration is not an issue for the reinforcement learning algorithm and conceptually simpler algorithms such as MCQ and REINFORCE perform comparably well.

Our system has the potential for further improvements. It is possible that for large dense graphs our GNN architecture is less suited, especially because of the sum aggregation, which dilutes information about individual connections with an increasing number of neighbors. As alternatives, different aggregation schemes or attention mechanisms could be explored. Another limiting factor of our GNN is the relatively high computational demand compared to other architectures, because of the requirement to evaluate a neural network on each edge. An architecture that makes a compromise between runtime and expressiveness has to be investigated. For very large graphs with sufficient statistical uniformity, sampling smaller sets of neighbors to be used for message passing is also an option [8].

Note that we only used little domain knowledge about the QAP, except for simple

structural transformations which are easy to derive. On the one hand, this remains close to the goal of automatically learning a heuristic solver to reduce the need for manual analysis, and keeps our approach general enough to transfer ideas to other problems. On the other hand, it seems unlikely that this is sufficient to train a system that achieves good results on arbitrary problems at a runtime that is competitive with established fast approximation solvers. As a middle ground, combination of deep reinforcement learning with traditional methods is also possible, since the sequential decision-making is very similar to tree search methods, such as branch-and-bound algorithms.

To solve specific classes of QAPs, the system could be fine-tuned with a given set of problems after pretraining on random QAPs. Real world problems are much more structured than QAPs with uniform random weights, so adapting the learned heuristic to this structure could greatly improve the results on a specific problem class. Even using a different random training distribution that is more similar to the target distribution in terms of its statistical properties could already be an improvement. The effectiveness of different training data and fine-tuning needs to be evaluated.

Another interesting extension of our work would be to adapt it to unweighted graph matching. This requires a modified graph neural network architecture and more consideration of the expressiveness limitations mentioned in section 4.3, since one can no longer rely on random noise in the edge weights to distinguish nodes and edges.

# Bibliography


[1] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

[2] Rainer E Burkard, Stefan E Karisch, and Franz Rendl. Qaplib–a quadratic assignment problem library. *Journal of Global optimization*, 10(4):391–403, 1997.

[3] Guillaume Maurice Jean-Bernard Chaslot. *Monte-carlo tree search*. PhD thesis, Maastricht University, 2010.

[4] Matthias Fey, Jan E Lenssen, Christopher Morris, Jonathan Masci, and Nils M Kriege. Deep graph matching consensus. *arXiv preprint arXiv:2001.09621*, 2020.

[5] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[6] Paul C Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *Journal of the society for industrial and applied mathematics*, 10(2): 305–313, 1962.

[7] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL `https://www.gurobi.com`.

[8] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[10] Andreas Karrenbauer and Antti Oulasvirta. Improvements to keyboard optimization with integer programming. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 621–626, 2014.

[11] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[12] Tjalling C. Koopmans and Martin Beckmann. Assignment problems and the

location of economic activities. *Econometrica*, 25(1):53–76, 1957. ISSN 00129682, 14680262. URL http://www.jstor.org/stable/1907742.

[13] Eugene L. Lawler. The quadratic assignment problem. *Management Science*, 9: 586–599, 1963.

[14] Eugene L Lawler. The quadratic assignment problem. *Management science*, 9(4): 586–599, 1963.

[15] Chang Liu, Runzhong Wang, Zetian Jiang, Junchi Yan, Lingxiao Huang, and Pinyan Lu. Revocable deep reinforcement learning with affinity regularization for outlier-robust graph matching, 2021.

[16] Xiyang Liu. Graph matching by graph neural network. Master's thesis, University of Illinois at Urbana-Champaign, 2018.

[17] Eliane Loiola, Nair Abreu, Paulo Boaventura-Netto, Peter Hahn, and Tania Querido. A survey of the quadratic assignment problem. *European Journal of Operational Research*, 176:657–690, 01 2007. doi: 10.1016/j.ejor.2005.09.032.

[18] Alfonsas Misevicius. A tabu search algorithm for the quadratic assignment problem. *Computational Optimization and Applications*, 30(1):95–111, 2005.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[20] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 4602–4609, 2019. doi: 10.1609/aaai.v33i01.33014602. URL https://ojs.aaai.org/index.php/AAAI/article/view/4384.

[21] Alex Nowak, Soledad Villar, Afonso S. Bandeira, and Joan Bruna. Revised note on learning algorithms for quadratic assignment with graph neural networks, 2018.

[22] R. Sato. A survey on the expressive power of graph neural networks. *ArXiv*, abs/2003.04078, 2020.

[23] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL `http://incompleteideas.net/book/the-book-2nd.html`.

[25] David M Tate and Alice E Smith. A genetic approach to the quadratic assignment problem. *Computers & Operations Research*, 22(1):73–83, 1995.

[26] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[27] Joshua T Vogelstein, John M Conroy, Vince Lyzinski, Louis J Podrazik, Steven G Kratzer, Eric T Harley, Donniell E Fishkind, R Jacob Vogelstein, and Carey E Priebe. Fast approximate quadratic programming for graph matching. *PLOS one*, 10(4):e0121002, 2015.

[28] Runzhong Wang, Junchi Yan, and Xiaokang Yang. Learning combinatorial embedding networks for deep graph matching. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 3056–3065, 2019.

[29] Runzhong Wang, Junchi Yan, and Xiaokang Yang. Neural graph matching network: Learning lawler's quadratic assignment problem with extension to hypergraph and multiple-graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2021. doi: 10.1109/TPAMI.2021.3078053.

[30] Mickey R Wilhelm and Thomas L Ward. Solving quadratic assignment problems by 'simulated annealing'. *IIE transactions*, 19(1):107–119, 1987.

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

[32] Lingxiao Zhao and Leman Akoglu. Pairnorm: Tackling oversmoothing in gnns. *arXiv preprint arXiv:1909.12223*, 2019.

# Appendices

# A. Reduction of partial assignments

We consider a QAP where we want to match nodes from the set $\mathcal{A}$ to nodes from $\mathcal{B}$. Given the edge coefficients $a_{i,j}$ and $b_{p,q}$ for all $i, j \in \mathcal{A}$ and $p, q \in \mathcal{B}$, as well as a bijective assignment $f : \mathcal{A} \to \mathcal{B}$, the QAP objective $V$ can be written as:

$$V = \sum_{i,j \in \mathcal{A}} a_{i,j} b_{f(i),f(j)}$$

Now, we define a partial assignment $g \subseteq f$, such that $g : \mathcal{S} \to f(\mathcal{S})$ with $\mathcal{S} \subseteq \mathcal{A}$. We also define the complement $\overline{\mathcal{S}} = \mathcal{A} \setminus \mathcal{S}$. In our application, we will treat $\mathcal{S}$ as the set of free (unassigned) nodes and $\overline{\mathcal{S}}$ as the set of fixed (assigned) nodes. Then,

$$\begin{aligned} V &= \sum_{i,j \in \overline{\mathcal{S}}} a_{i,j} b_{f(i),f(j)} + \sum_{i,j \in \mathcal{S}} a_{i,j} b_{g(i),g(j)} + \sum_{i \in \overline{\mathcal{S}}, j \in \mathcal{S}} a_{i,j} b_{f(i),g(j)} + \sum_{i \in \mathcal{S}, j \in \overline{\mathcal{S}}} a_{i,j} b_{g(i),f(j)} \\ &= \sum_{i,j \in \overline{\mathcal{S}}} a_{i,j} b_{f(i),f(j)} + \sum_{i,j \in \mathcal{S}} a_{i,j} b_{g(i),g(j)} + \sum_{i \in \overline{\mathcal{S}}, j \in \mathcal{S}} \left( a_{i,j} b_{f(i),g(j)} + a_{j,i} b_{g(j),f(i)} \right) \end{aligned}$$

Now, if the assignment $\overline{g} = f \setminus g$ of the nodes in $\overline{\mathcal{S}}$ is fixed, we can introduce new values

$$c_{j,q} = \sum_{i \in \overline{\mathcal{S}}} \left( a_{i,j} b_{f(i),q} + a_{j,i} b_{q,f(i)} \right)$$

that are constant w.r.t. $g$, simplifying the last term to

$$\begin{aligned} &\sum_{i \in \overline{\mathcal{S}}, j \in \mathcal{S}} \left( a_{i,j} b_{f(i),g(j)} + a_{j,i} b_{g(j),f(i)} \right) \\ =& \sum_{j \in \mathcal{S}} \sum_{i \in \overline{\mathcal{S}}} \left( a_{i,j} b_{f(i),g(j)} + a_{j,i} b_{g(j),f(i)} \right) \\ =& \sum_{j \in \mathcal{S}} c_{j,g(j)}. \end{aligned}$$

The first term is constant:

$$D = \sum_{i,j \in \overline{\mathcal{S}}} a_{i,j} b_{f(i),f(j)}.$$

This means that the objective value of $f$ in the original QAP is equivalent to the objective of $g$ in a QAP with linear and constant terms that does not require explicit information about the fixed part of the assignment $\overline{g}$:

$$V = \left( \sum_{i,j} a_{i,j} b_{g(i),g(j)} \right) + \left( \sum_{i} c_{i,g(i)} \right) + D.$$

Note that this can easily be extended to a reduction that is closed under QAPs with linear term and constant term, by applying the procedure shown before to the quadratic term, and adding the linear term of fixed assigned pairs to the constant term.

# B. Transition function for the reduction MDP

The reduction from section 3.1 can also be written in matrix form, which we will use more formally as a transition function

$$T : (\mathrm{QAP}_n, \{1, \ldots, n\}^2) \to (\mathrm{QAP}_{n-1}, \mathbb{R})$$

that is closed under the space of QAPs:

$$T((\mathbf{A}, \mathbf{B}, \mathbf{C}), (u, v)) = ((\mathbf{A}_{\bar{U},\bar{U}}, \mathbf{B}_{\bar{V},\bar{V}}, \mathbf{C}_{\bar{U},\bar{V}} + \mathbf{C}'_{\mathrm{in}} + \mathbf{C}'_{\mathrm{out}}), r) \tag{B.1}$$

where

$$\begin{aligned}
\bar{U} &= \{1, \ldots, n\} \setminus \{u\}, \\
\bar{V} &= \{1, \ldots, n\} \setminus \{v\}, \\
\mathbf{C}'_{\mathrm{in}} &= \mathbf{A}_{\bar{U},\{u\}} \mathbf{B}^T_{\bar{V},\{v\}}, \\
\mathbf{C}'_{\mathrm{out}} &= \mathbf{A}^T_{\{u\},\bar{U}} \mathbf{B}_{\{v\},\bar{V}}, \\
r &= -C_{u,v}.
\end{aligned}$$

Here, the notation $\mathbf{M}_{I,J}$, where $I$ and $J$ are sets, stands for the submatrix of $\mathbf{M}$ containing only the rows $I$ and the columns $J$.

Note that with this transition function the empty problem $\mathrm{QAP}_0$ is the terminal state because it has no actions. Additionally, we skip problems of size 1 ($\mathrm{QAP}_1$) by directly assigning the single last remaining pair, since there is no choice to be made for the agent.

# C. Shift and scale invariance of the QAP

In general, shifting the weights $a_{i,j}$ and $b_{i,j}$ with offsets $\delta_A$ and $\delta_B$ and scaling with positive factors $\gamma_A$ and $\gamma_B$, only shifts and scales the objective value of any permutation $f$. This can be easily proven by substituting shifted and scaled weights in the form of Koopmans-Beckmann QAP in eq. (2.1),

$$V' = \sum_{i,j \in \mathcal{A}} (\gamma_A a_{i,j} - \delta_A)(\gamma_B b_{f(i),f(j)} - \delta_B) \tag{C.1}$$

$$= \gamma_A \gamma_B \left( \sum_{i,j \in \mathcal{A}} a_{i,j} b_{f(i),f(j)} \right) - \sum_{i,j \in \mathcal{A}} \gamma_A a_{i,j} \delta_B - \sum_{i,j \in \mathcal{A}} \gamma_B b_{i,j} \delta_B \tag{C.2}$$

$$= \gamma_A \gamma_B \left( \sum_{i,j \in \mathcal{A}} a_{i,j} b_{f(i),f(j)} \right) - \sum_{i,j \in \mathcal{A}} \gamma_A a_{i,j} \delta_B - \sum_{i,j \in \mathcal{B}} \gamma_B b_{i,j} \delta_B. \tag{C.3}$$

The last two sum terms are constant with respect to the permutation $f$. This means that the ordering of solutions is preserved (as long as the scaling factors $\gamma$ have the same sign). In particular, the optimal solution for the transformed problem is also an optimal solution for the original problem.

By substituting $\min a_{i,j}$ and $\frac{1}{\max a_{i,j} - \min a_{i,j}}$ for $\delta_A$ and $\gamma_A$, and the equivalent for $\delta_B$ and $\gamma_B$, we can normalize the weights to the range $[0, 1]$.

# D. Generalization of different separation layers

We tested A2C and DQN (trained on 8 node QAPs) with different separation layers. The setup is similar to section 5.3. The agents were evaluated on 20 randomly generated problems for each size. The results are normalized according to eq. (5.2).

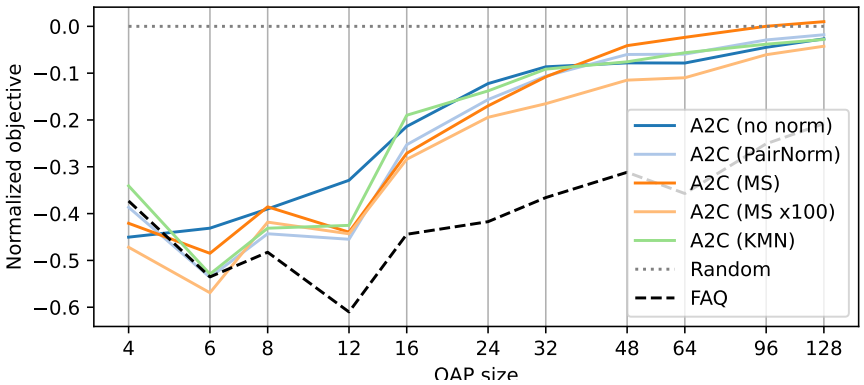



Figure D.1.: A2C agents with different separation layers, lower is better.

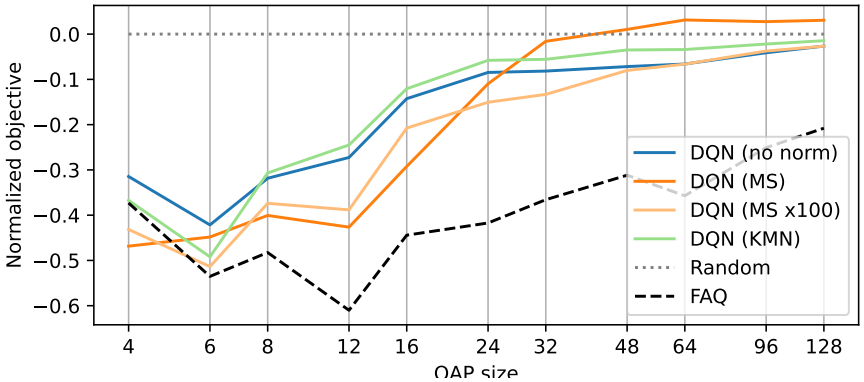



Figure D.2.: DQN agents with different separation layers, lower is better.

# E. Additional results on QAPLIB

## E.1. Results on individual instances

The table below shows the individual results per instance underlying table 5.2. Again, the best solver is in bold, the underlined solver is the best of all except the benchmark solver FAQ. This table also shows the results of the Last-8-Opt baseline from section 5.3.

| | Known | FAQ | MCQ | DQN | RF | A2C | NGM | 10k rand | last 8 |
|---|---|---|---|---|---|---|---|---|---|
| bur26a | 5426670 | **5434914** | 5504953 | 6061955 | 5566700 | 5570459 | 5621774 | 5564520 | 5793218 |
| bur26b | 3817852 | **3830246** | 3896180 | 4312957 | 3964817 | 3956253 | 3927943 | 3943916 | 4136113 |
| bur26c | 5426795 | **5434782** | 5623103 | 6131072 | 5564032 | 5580009 | 5608065 | 5612786 | 5730730 |
| bur26d | 3821225 | **3826287** | 3964691 | 4415621 | 4003244 | 3995071 | 3962317 | 3971837 | 4058964 |
| bur26e | 5386879 | **5399558** | 5507301 | 6239394 | 5571350 | 5631989 | 5536142 | 5560674 | 5641258 |
| bur26f | 3782044 | **3784562** | 3870206 | 4441846 | 3925759 | 3925945 | 3949711 | 3874815 | 4151842 |
| bur26g | 10117172 | **10145072** | 10482870 | 11509722 | 10456302 | 10454199 | 10433439 | 10409944 | 10570517 |
| bur26h | 7098658 | **7119460** | 7433670 | 8134616 | 7633835 | 7444114 | 7348866 | 7313327 | 7815424 |
| chr12a | 9552 | 33082 | 21228 | 15056 | 15056 | 15056 | **14940** | 15776 | 16936 |
| chr12b | 9742 | **10468** | 23614 | 15534 | 21764 | 17148 | 14984 | 16752 | 22958 |
| chr12c | 11156 | 13088 | **11798** | 12832 | 12832 | 12832 | 16346 | 18730 | 18014 |
| chr15a | 9896 | **19852** | 27164 | 23594 | 23594 | 23594 | 20442 | 20674 | 22648 |
| chr15b | 7990 | **9112** | 21478 | 17890 | 21202 | 13420 | 22048 | 19966 | 52700 |
| chr15c | 9504 | 16884 | 24854 | **15654** | **15654** | **15654** | 24190 | 26632 | 24366 |
| chr18a | 11098 | **15440** | 33470 | 21094 | 21094 | 21094 | 33124 | 37224 | 40312 |
| chr18b | 1534 | 1792 | 2284 | **1534** | 1542 | 1716 | 2504 | 2466 | 3440 |
| chr20a | 2192 | **3172** | 3756 | 3874 | 5510 | 4134 | 5178 | 5700 | 6722 |
| chr20b | 2298 | 3206 | 3258 | 2900 | 3082 | **2628** | 5766 | 5580 | 8600 |
| chr20c | 14142 | **19836** | 57672 | 29942 | 38560 | 48776 | 49770 | 56684 | 71644 |
| chr22a | 6156 | 8430 | 8168 | 7838 | **7572** | 7734 | 9348 | 8810 | 9748 |
| chr22b | 6194 | 8714 | 7900 | **7636** | **7636** | **7636** | 9006 | 8794 | 13198 |
| chr25a | 3796 | **5580** | 13200 | 10268 | 10268 | 8538 | 11648 | 11294 | 15680 |
| els19 | 17212548 | 21297242 | **18885784** | 39960516 | 39960516 | 34239276 | 27029748 | 26935464 | 29507806 |
| esc128 | 64 | 72 | 84 | 96 | 96 | **66** | 242 | 240 | 328 |
| esc16a | 68 | **70** | 72 | 76 | 76 | 74 | 78 | 78 | 90 |
| esc16b | 292 | 320 | 296 | 296 | 296 | 294 | **292** | **292** | 306 |
| esc16c | 160 | **168** | 170 | 176 | 174 | 184 | 174 | 172 | 220 |
| esc16d | 16 | 62 | 22 | 20 | 22 | **18** | 20 | 22 | 30 |
| esc16e | 28 | **30** | 32 | 32 | 34 | **30** | 32 | 32 | 40 |
| esc16g | 26 | **30** | 34 | 32 | 34 | 34 | 32 | 32 | 36 |
| esc16h | 996 | 1518 | 1050 | **996** | **996** | 1222 | 1004 | 1004 | 1258 |
| esc16i | 14 | **14** | **14** | **14** | **14** | **14** | 18 | 16 | 18 |
| esc16j | 8 | **8** | 12 | **8** | 10 | 12 | **8** | 10 | 20 |
| esc32a | 130 | **160** | 206 | 194 | 220 | 188 | 298 | 292 | 322 |
| esc32b | 168 | **196** | 320 | 248 | 312 | 304 | 368 | 364 | 456 |
| esc32c | 642 | **650** | **650** | 652 | 656 | 652 | 754 | 760 | 890 |
| esc32d | 200 | 226 | 224 | 228 | 238 | **216** | 284 | 274 | 340 |
| esc32e | 2 | **2** | **2** | **2** | **2** | **2** | **2** | **2** | 44 |
| esc32g | 6 | 10 | 8 | **6** | **6** | **6** | 10 | **6** | 38 |
| esc32h | 438 | 486 | 468 | 456 | 458 | **446** | 534 | 554 | 618 |
| esc64a | 116 | **118** | 126 | 120 | 140 | **118** | 200 | 198 | 264 |

Continued on next page

| | Known | FAQ | MCQ | DQN | RF | A2C | NGM | 10k rand | last 8 |
|---|---|---|---|---|---|---|---|---|---|
| had12 | 1652 | **1666** | 1804 | 1812 | 1760 | 1686 | 1700 | 1708 | 1728 |
| had14 | 2724 | **2726** | 2976 | 2986 | 2796 | 2784 | 2866 | 2842 | 3020 |
| had16 | 3720 | **3736** | 3956 | 4022 | 3844 | 3816 | 3902 | 3918 | 4066 |
| had18 | 5358 | **5442** | 5694 | 5930 | 5482 | 5532 | 5558 | 5600 | 5594 |
| had20 | 6922 | **6988** | 7546 | 7540 | 7292 | 7220 | 7300 | 7270 | 7888 |
| kra30a | 88900 | **96620** | 105330 | 122580 | 108850 | 107960 | 114410 | 119170 | 123040 |
| kra30b | 91420 | **94810** | 113760 | 117960 | 108000 | 113130 | 118130 | 116490 | 126230 |
| kra32 | 88700 | **92930** | 105180 | 106300 | 110360 | 107160 | 120930 | 117810 | 129910 |
| lipa20a | 3683 | **3772** | 3880 | 3857 | 3861 | 3854 | 3853 | 3853 | 3867 |
| lipa20b | 27076 | 31118 | 33721 | 33983 | 32780 | **29223** | 33125 | 33240 | 34097 |
| lipa30a | 13178 | **13516** | 13745 | 13645 | 13612 | 13682 | 13631 | 13655 | 13673 |
| lipa30b | 151426 | **151426** | 188352 | 186160 | 185424 | 185758 | 187607 | 186859 | 190378 |
| lipa40a | 31538 | **32149** | 32454 | 32486 | 32455 | 32471 | 32454 | 32395 | 32448 |
| lipa40b | 476581 | **476581** | 599911 | 600565 | 597368 | 588493 | 601848 | 600922 | 614959 |
| lipa50a | 62093 | **63056** | 63850 | 63544 | 63589 | 63713 | 63671 | 63679 | 63893 |
| lipa50b | 1210244 | **1210244** | 1513968 | 1522247 | 1495535 | 1512689 | 1523856 | 1525231 | 1534842 |
| lipa60a | 107218 | **108568** | 109433 | 109654 | 109523 | 109349 | 109595 | 109645 | 109787 |
| lipa60b | 2520135 | **2520135** | 3201268 | 3170610 | 3170610 | 3141530 | 3208501 | 3207887 | 3237756 |
| lipa70a | 169755 | **171682** | 173001 | 173316 | 173022 | 173025 | 173220 | 173175 | 173428 |
| lipa70b | 4603200 | **4603200** | 5880290 | 5811162 | 5811162 | 5813932 | 5890161 | 5880879 | 5957295 |
| lipa80a | 253195 | 258434 | 257521 | 257424 | 257701 | **257312** | 257663 | 257566 | 258336 |
| lipa80b | 7763962 | **7763962** | 9998673 | 9908551 | 9908551 | 9937875 | 9983040 | 9992745 | 10060982 |
| lipa90a | 360630 | 367146 | 366354 | 366213 | 366503 | **366041** | 366508 | 366360 | 367225 |
| lipa90b | 12490441 | **12490441** | 15960857 | 15993362 | 15979647 | 15932518 | 16076956 | 16071615 | 16213425 |
| nug12 | 578 | **598** | 678 | 670 | 636 | 642 | 634 | 622 | 654 |
| nug14 | 1014 | **1074** | 1206 | 1148 | 1118 | 1084 | 1156 | 1140 | 1168 |
| nug15 | 1150 | **1238** | 1388 | 1374 | 1254 | 1338 | 1318 | **1238** | 1328 |
| nug16a | 1610 | **1660** | 1828 | 2042 | 1784 | 1740 | 1836 | 1806 | 1962 |
| nug16b | 1240 | **1282** | 1466 | 1510 | 1468 | 1430 | 1396 | 1422 | 1540 |
| nug17 | 1732 | **1744** | 2048 | 2114 | 1884 | 1858 | 1980 | 1988 | 1960 |
| nug18 | 1930 | **1972** | 2262 | 2250 | 2088 | 2172 | 2242 | 2214 | 2320 |
| nug20 | 2570 | **2624** | 3110 | 3054 | 2842 | 2866 | 2936 | 2992 | 3164 |
| nug21 | 2438 | **2474** | 3008 | 3098 | 2816 | 2708 | 2916 | 2930 | 3242 |
| nug22 | 3596 | **3712** | 4168 | 4698 | 3994 | 4138 | 4298 | 4114 | 5056 |
| nug24 | 3488 | **3558** | 4338 | 4324 | 3968 | 4022 | 4234 | 4160 | 4338 |
| nug25 | 3744 | **3796** | 4558 | 4428 | 4312 | 4176 | 4420 | 4478 | 4908 |
| nug27 | 5234 | **5406** | 6172 | 5974 | 5974 | 5866 | 6208 | 6128 | 6812 |
| nug28 | 5166 | **5290** | 6082 | 5850 | 5850 | 5886 | 6128 | 6200 | 6386 |
| nug30 | 6124 | **6290** | 7214 | 7780 | 7038 | 7014 | 7294 | 7360 | 7570 |
| rou12 | 235528 | **245168** | 287124 | 264660 | 264660 | 265478 | 264898 | 257132 | 251312 |
| rou15 | 354210 | **371458** | 418058 | 411656 | 414478 | 413792 | 403872 | 395068 | 397112 |
| rou20 | 725522 | **743884** | 843902 | 823696 | 823696 | 820018 | 817776 | 819272 | 846126 |
| scr12 | 31410 | 42420 | 34962 | 35912 | 38540 | **33372** | 36292 | 35248 | 49370 |
| scr15 | 51140 | **55206** | 77786 | 62464 | 68980 | 57842 | 68768 | 65840 | 87154 |
| scr20 | 110030 | **120160** | 162216 | 139582 | 139582 | 144614 | 154636 | 164666 | 201366 |
| sko100a | 152002 | **153918** | 168634 | 174088 | 166416 | 166268 | 172810 | 173476 | 176286 |
| sko100b | 153890 | **155928** | 171442 | 179842 | 168704 | 167914 | 175588 | 175138 | 179206 |
| sko100c | 147862 | **149524** | 167150 | 170646 | 165072 | 161426 | 169806 | 169414 | 171778 |
| sko100d | 149576 | **151724** | 167388 | 171038 | 165084 | 161400 | 170816 | 170132 | 174500 |
| sko100e | 149150 | **150848** | 165108 | 173512 | 166128 | 162888 | 170958 | 171032 | 173750 |
| sko100f | 149036 | **151328** | 164524 | 172486 | 163898 | 161352 | 169986 | 169306 | 172038 |
| sko42 | 15812 | **16060** | 18412 | 19756 | 18078 | 17652 | 18716 | 18722 | 19460 |
| sko49 | 23386 | **23670** | 26782 | 28372 | 26648 | 26030 | 27554 | 27158 | 27856 |
| sko56 | 34458 | **34894** | 39460 | 40750 | 39138 | 38738 | 40684 | 40462 | 41676 |
| sko64 | 48498 | **49102** | 55560 | 56600 | 54508 | 54190 | 56222 | 56464 | 58176 |
| sko72 | 66256 | **66340** | 74734 | 77638 | 73438 | 73160 | 76870 | 76280 | 78662 |
| sko81 | 90998 | **91946** | 101762 | 105116 | 101678 | 100542 | 104710 | 104112 | 106082 |
| sko90 | 115534 | **117122** | 129214 | 133882 | 128268 | 126810 | 132942 | 132862 | 135760 |

Continued on next page

| | Known | FAQ | MCQ | DQN | RF | A2C | NGM | 10k rand | last 8 |
|---|---|---|---|---|---|---|---|---|---|
| ste36a | 9526 | **10240** | 13796 | 18316 | 13148 | 13658 | 16768 | 16636 | 26044 |
| ste36b | 15852 | **17110** | 33280 | 56402 | 29540 | 34312 | 43248 | 45408 | 66058 |
| ste36c | 8239110 | **8707898** | 10553054 | 14435908 | 10803778 | 11537998 | 12988352 | 13429044 | 17987966 |
| tai100a | 21044752 | **21517680** | 23590874 | 23550132 | 23512864 | 23470348 | 23644528 | 23637912 | 23763708 |
| tai100b | 1185996137 | **1249810688** | 1404335360 | 1734195072 | 1370186624 | 1467823360 | 1612020992 | 1618039296 | 1733170688 |
| tai12a | 224416 | **244672** | 255738 | 263516 | 266298 | 249696 | 255158 | 247756 | 249622 |
| tai12b | 39464925 | 49891524 | 50411536 | 67953696 | 50980424 | 49486132 | 47252044 | 48173912 | **43475228** |
| tai150b | 498896643 | **513680608** | 558494720 | 643951040 | 557571904 | 553060736 | 628349568 | 620252288 | 650813824 |
| tai15a | 388214 | **397376** | 459388 | 445840 | 428824 | 405682 | 436968 | 431382 | 435146 |
| tai15b | 51765268 | **52133496** | 52830876 | 233960816 | 233960816 | 233427984 | 52871608 | 52736756 | 52329208 |
| tai17a | 491812 | **520696** | 594652 | 548986 | 567328 | 566840 | 544754 | 562530 | 559782 |
| tai20a | 703482 | **736140** | 833112 | 834026 | 827138 | 813276 | 806382 | 806772 | 825994 |
| tai20b | 122455319 | **139297728** | 157104800 | 332220000 | 291811008 | 230026016 | 140704160 | 139619296 | 224385280 |
| tai25a | 1167256 | **1219484** | 1371982 | 1365222 | 1308850 | 1322812 | 1352912 | 1344170 | 1376482 |
| tai25b | 344355646 | **368570592** | 405573376 | 769520192 | 561361536 | 558362560 | 518647040 | 528149760 | 655664512 |
| tai30a | 1818146 | **1894404** | 2077222 | 2080590 | 2080590 | 2083158 | 2065706 | 2056508 | 2042980 |
| tai30b | 637117113 | **721752448** | 753673920 | 1187664128 | 891391872 | 895894208 | 896379008 | 906371584 | 1422203392 |
| tai35a | 2422002 | **2514002** | 2850472 | 2825392 | 2825392 | 2730778 | 2786748 | 2756206 | 2791432 |
| tai35b | 283315445 | **306237120** | 355326432 | 460506048 | 344314432 | 324173536 | 377687744 | 391818720 | 425610880 |
| tai40a | 3139370 | **3225948** | 3649812 | 3705580 | 3598250 | 3564348 | 3610604 | 3614546 | 3661876 |
| tai40b | 637250948 | **702232320** | 797046592 | 1028942464 | 777437760 | 772687680 | 917498816 | 918116480 | 1036146496 |
| tai50a | 4938796 | **5123102** | 5685040 | 5781798 | 5635668 | 5491122 | 5677282 | 5674880 | 5778250 |
| tai50b | 458821517 | **476748704** | 543388672 | 708391232 | 552917760 | 553325568 | 614638528 | 633366720 | 720331584 |
| tai60a | 7205962 | **7440930** | 8340788 | 8273732 | 8273732 | 8144674 | 8281996 | 8259128 | 8358918 |
| tai60b | 608215054 | **646196480** | 768286464 | 974227968 | 784349824 | 779974272 | 862969152 | 875504000 | 981004672 |
| tai64c | 1855928 | 5893540 | **1887500** | 1941100 | 1941100 | 1981314 | 2133738 | 2097848 | 2706866 |
| tai80a | 13499184 | **13765708** | 15069342 | 15391236 | 15356734 | 15071662 | 15283138 | 15259256 | 15424076 |
| tai80b | 818415043 | **850424256** | 963430976 | 1258663936 | 945602560 | 929808832 | 1120577408 | 1117269376 | 1235937280 |
| tho150 | 8133398 | **8217176** | 9191296 | 9617000 | 9140046 | 9055532 | 9557766 | 9549556 | 9751330 |
| tho30 | 149936 | **155256** | 187066 | 203632 | 180334 | 177318 | 185622 | 187974 | 198764 |
| tho40 | 240516 | **243690** | 299726 | 312352 | 301564 | 288160 | 304878 | 304702 | 305034 |
| wil100 | 273038 | **274696** | 291716 | 293482 | 290668 | 287210 | 294172 | 295110 | 298582 |
| wil50 | 48816 | **49220** | 53700 | 53628 | 52024 | 52160 | 53418 | 53134 | 53948 |

## E.2. Comparison between results on original and swapped QAPs

Table E.2 shows the results of the RL agents both on QAPLIB and on swapped QAPLIB, where in every QAP the weight matrices $A$ and $B$ are exchanged. Note that often, a good result on the original problem does not imply a good result on the swapped problem, since the two graphs have different structure and represent different roles for many problems in QAPLIB, and the heterogeneous graph neural network is not symmetric in the order of the input graphs.

In our experiments with sets of randomly generated QAPs, this is not as important since every graph can occur equally likely in any position. While swapping the graphs could sometimes improve the result on individual problems by sampling a second assignment, it would not change the overall trend on random distributions.

| | MCQ | | DQN | | REINFORCE | | A2C | |
|---|---|---|---|---|---|---|---|---|
| | A | B | A | B | A | B | A | B |
| bur 26-26 | **3.0** | 6.4 | 21.6 | 14.4 | 4.0 | 14.3 | 5.0 | 3.9 |
| chr 12-25 | 254.2 | 125.3 | 96.7 | 89.0 | 276.6 | 87.9 | **74.5** | 198.8 |
| els 19-19 | **9.7** | 258.2 | 209.8 | 132.2 | 164.5 | 132.2 | 98.9 | 179.7 |
| esc 16-128 | 151.8 | 22.5 | 26.1 | 21.2 | 256.4 | 20.3 | **20.0** | 136.0 |
| had 12-20 | 10.9 | 8.1 | 12.2 | 9.4 | 4.0 | 8.9 | 4.1 | **3.9** |
| kra 30-32 | 27.4 | 32.2 | 35.5 | 30.4 | 27.0 | 30.4 | **25.7** | 26.6 |
| lipa 20-90 | 15.2 | 15.0 | 17.1 | 14.2 | 13.4 | 14.3 | **12.2** | 14.0 |
| nug 12-30 | 21.7 | 22.3 | 30.8 | 21.7 | 14.0 | 21.8 | 15.8 | **12.7** |
| rou 12-20 | 18.7 | 21.8 | 19.3 | 14.3 | 20.2 | 14.3 | 17.0 | **14.2** |
| scr 12-20 | 56.5 | 36.9 | 21.5 | 28.1 | 60.7 | 28.1 | **16.9** | 49.6 |
| sko 42-100 | 15.2 | 19.6 | 28.4 | 21.5 | 14.0 | 21.5 | 16.4 | **11.8** |
| ste 36-36 | 61.0 | 141.6 | 171.2 | 141.1 | **51.8** | 141.1 | 136.7 | 66.6 |
| tai 12-150 | **19.1** | 55.2 | 105.7 | 66.1 | 72.8 | 66.1 | 51.2 | 70.5 |
| tho 30-150 | 24.7 | 28.9 | 50.6 | 32.8 | 22.8 | 32.8 | 25.8 | **19.0** |
| wil 50-100 | 10.7 | 10.0 | 15.4 | 9.9 | **6.6** | 9.9 | 7.8 | 6.9 |
| mean | 46.7 | 53.6 | 57.5 | 43.1 | 67.3 | 42.9 | **35.2** | 54.3 |

Table E.2.: Results of RL agents with swapped inputs. A is the original, B the swapped problem.