# Learning Shaped Rewards via Adversarial Imitation Learning for Combinatorial Optimisation

submitted by

Joe Rosenberg

for the degree of Master of Science

of the

University of Bath

Department of Computer Science

October 2020

Signature of Author 

Joe Rosenberg

## Summary

Designing specialised heuristics for combinatorial optimisation problems can be difficult, and often requires expert knowledge of the task at hand. By framing the construction of a solution as a sequential decision problem, it is possible to train a specialised RL agent to provide solutions without the need to design specialised heuristics.

Additionally, recent advances in deep learning have made it possible to create neural network models that generalise across inputs of different lengths. These models can also output sequences of varying length, such as a sequence of actions to perform. Such models have been successfully used as policy networks for agents that solve large combinatorial optimisation tasks.

However, deep reinforcement learning with sparse rewards can be very sample-inefficient, making it difficult to scale these methods to larger problems. One possible solution is to infer a better reward function from high-quality demonstrations of the task and then use this reward function to train an agent.

In this dissertation, we formulate an automated circuit routing problem as a Markov decision process and devise and implement a method for learning a reward function from demonstrations. The method learns a single function that generalises over all possible problem sizes, and we use it to learn a shaped reward function for accelerating the training of a deep RL agent with a sequence-to-sequence policy network. We also implement such an agent and train it using this reward function. The agent is able to supply good solutions for a large range of problem instances, and we identify methods for improving its performance on problem instances where it fails.

# Acknowledgements

I would like to thank my supervisor Dr Özgür Şimşek for her guidance and feedback for this project, as well as Dr Chithrupa Ramesh and Steffan Owen from Zuken for their illuminating comments on the circuit routing problem. I would also like to thank my family and friends for all of their support through these exceptional times.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The class of combinatorial optimisation problems is very general, containing both classical problems in mathematics and computer science and many important real-life problems. For many of these problems there is no known exact polynomial-time algorithm, making the exact solution of large problems infeasible. Due to this, much effort has gone into the design of efficient heuristics for approximately solving these problems. While no heuristic is better than another in general by the *no free lunch theorem* (Wolpert and Macready, 1997), it is possible to design specialised heuristics that have especially good performance on particular distributions of CO problems (Bengio et al., 2018).

Designing these specialised heuristics can be difficult, and often requires expert knowledge of the task at hand (Dai et al., 2017). By framing the construction of a solution as a sequential decision problem with the problem objective as feedback and supplying a simulation of the particular problem instance, we can train a specialised RL agent to provide near-optimal solutions without the need to design specialised heuristics.

Recent advances in deep learning have made it possible to create neural network models that generalise across sequence inputs of different lengths (Vinyals et al., 2015; Bahdanau et al., 2014) or graphs of different shapes (Gasse et al., 2019). These models can also output sequences of varying length, such as a sequence of actions to perform. Such models have been successfully used as policy networks for a reinforcement learning agent that solves relatively large combinatorial optimisation tasks (Bello et al., 2016; Dai et al., 2017; Gasse et al., 2019).

It is well-known that deep reinforcement learning with sparse rewards can be very sample-inefficient (Zheng et al., 2019), making it difficult to scale these methods to larger problems. One possible solution is to infer a better reward function from high-quality demonstrations of the task and then use this reward function to train an agent. Such an approach is called *inverse reinforcement learning* or *apprenticeship learning* (Ng and Russell, 2000).

In this project, we devise and implement a method for learning a reward function from optimal demonstrations of a combinatorial optimisation problem that arises in automated circuit routing. The desired properties of this method are that it learns a single function that generalises over all possible problem sizes, and that the learned reward function can be used to accelerate the training of a deep RL agent with a sequence-to-sequence policy network.

# Chapter 2

# Reinforcement learning

In this chapter, we will cover the fundamental concepts in the areas of reinforcement learning required for the project. Further details of how reinforcement learning can be applied to circuit routing (and combinatorial optimisation in general) are delayed until Chapter 4, as we will first explore the circuit routing problem in more detail in Chapter 3.

The field of reinforcement learning is concerned with the problem of determining how an agent should interact with an environment in order to maximise some notion of cumulative reward (Sutton and Barto, 2018). The theoretical framework of reinforcement learning is very general, and has been used to model dopamine-based learning in the brain (Dabney et al., 2020), 2, 3.

There have also been recent advances in the applications of reinforcement learning. AlphaGo is a system that uses reinforcement learning to learn a policy for the strategy game Go with no prior knowledge except for the rules of the game (Silver et al., 2017). An agent acting with the learned policy is capable of consistently beating the best Go players in the world. These results are notable due to the extremely large state space of Go (over $10^{170}$ board configurations) (Tromp and Farnebäck, 2007) and the ability to surpass human performance.

Other applications include learning how to control an RC helicopter to perform flight maneuvers (Kim et al., 2004) and learning to play video games from raw image data (Mnih et al., 2013).

## 2.1 Foundations

In reinforcement learning, we usually model the interaction of the agent with the environment, and the reward signal the agent receives, as a **Markov decision process** (Sutton and Barto, 2018).

A Markov decision process (MDP) is a type of discrete-time stochastic process. At time $t$ in an MDP, the state $S_t$ is known to the agent. The agent then selects an action $A_t$ depending on the current state. After taking the action, the agent observes a scalar reward $R_t$ and a new state $S_{t+1}$. The reward and new state are random variables whose distributions depend only on the current state $S_t$ and action $A_t$.

Conceptually, the reward signal represents some objective that the agent is aiming to maximise over time. For example, a cleaning robot could receive a reward of $+1$ upon depositing a piece of litter into a bin. Negative rewards can represent a cost that the agent is aiming to minimise. For example, an agent navigating a maze could receive a reward of $-1$ at each time step until it reaches the end of the maze.

We can define an MDP using four pieces of data:

- A set $\mathcal{S}$ of possible states.
- An initial state distribution $\mu : \mathcal{S} \to [0, 1]$, where $\mu(s) = \mathbb{P}(S_0 = s)$.
- For each state $s \in \mathcal{S}$, a set $\mathcal{A}(s)$ of actions that can be taken in that state.
- A dynamics function $p(s', r \mid s, a)$ that gives a probability distribution over $\mathcal{S} \times \mathbb{R}$ for each state-action pair $(s, a)$.

The dynamics function $p(s', r \mid s, a)$ gives the probability of observing reward $r$ and state $s'$ after taking action $a$ in state $s$ :

$$\mathbb{P}\left(S_{t+1} = s', R_t = r \mid S_t = s, A_t = a\right) = p(s', r \mid s, a).$$

We often do not require the full machinery of MDPs to model the agent-environment interaction. For example, the environment dynamics may be **deterministic**, meaning that there is only one next state and reward that can result from a choice of action in a state. In this case, a more amenable description of the environment

dynamics can be given in the form of a **next state function** $\eta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ and **reward function** $R : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$. The next state function $\eta(s, a)$ gives the state resulting from taking action $a$ in state $s$, and the reward function $R(s, s')$ gives the reward received when transitioning from state $s$ to state $s'$. These two functions are related to the dynamics function:

$$p(s', r \mid s, a) = \begin{cases} 1 & \text{if } \eta(s, a) = s' \text{ and } R(s, s') = r, \\ 0 & \text{otherwise.} \end{cases}$$

The agent acts according to a **policy** $\pi$ which gives a probability distribution over actions for each state. An agent with policy $\pi$ chooses action $a$ in state $s$ with probability $\pi(a \mid s)$.

We call $\bar{s} \in \mathcal{S}$ a **terminal state** if any action the agent takes in $\bar{s}$ gives a terminal state as the next state and a reward of 0 with probability 1. We consider the MDP to have ended once it reaches a terminal state, since no additional feedback in the form of meaningful state transitions or rewards is given. The time step $T$ at which the process first enters a terminal state is called the **episode length**, and the sequence of data $(S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T)$ is called a **trajectory** or **episode**.

We call an MDP **episodic** if any policy causes the MDP to eventually enter a terminal state with probability 1.

For an episodic MDP, we define the **return** at time $t$ as the sum of rewards starting from time $t$:

$$G_t = \sum_{k=0}^{\infty} R_{t+k}. \tag{2.1}$$

This sum converges with probability 1 in an episodic MDP, as all rewards after reaching a terminal state are zero. This may not be the case for non-episodic (a.k.a. **continuing**) MDPs. In this case, we introduce a **discount factor** $\gamma \in (0, 1)$ to define the **discounted return** from time $t$:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k}. \tag{2.2}$$

If the rewards $R_k$ are bounded, then the discounted sum converges. We can unify

the definition of the return for episodic MDPs with the definition for continuing MDPs by allowing the case where the discount factor $\gamma$ is equal to 1.

We call the return starting at time 0 the **total return** or **episodic return**.

The **state value function** $v_\pi(s)$ of a policy $\pi$ is the expected total return starting from state $s$ and following policy $\pi$:

$$v_\pi(s) = \mathbb{E}_\pi \left[ G_t \mid S_t = s \right]. \tag{2.3}$$

Similarly, we can define the **value of a policy** for an MDP with initial state distribution $\mu$ as the expected total return when following policy $\pi$:

$$v(\pi) = \mathbb{E}_\pi \left[ G_0 \mid S_0 \sim \mu \right]. \tag{2.4}$$

The **action value function** $q_\pi(s, a)$ of a policy $\pi$ is the expected total return starting from state $s$ and selecting action $a$ on the first time-step, but following $\pi$ afterwards:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[ G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[ R_t + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a \right]. \tag{2.5}$$

We say that $\pi$ is an **optimal policy** if it maximises the expected total return when starting from any state. Equivalently, we say that $\pi$ is optimal if $v_\pi(s) \geq v_{\pi'}(s)$ for any policy $\pi'$ and any state $s \in \mathcal{S}$.

If we only know the action value function $q_*(s, a)$ of an optimal policy, it is possible to recover an optimal policy from this function. In particular, the **greedy policy with respect to the optimal action value function** $q_*$ is optimal. This policy always selects the action $a$ that maximises $q_*(s, a)$ for each state $s$, with ties between actions broken arbitrarily.

Because of this relationship between the optimal action value function and the optimal policy, it is possible to learn an optimal policy by learning the optimal action value function. **Q-learning** is a method for learning $q_*$ purely from sampled transitions, without knowledge of the environment dynamics $p(s', r \mid s, a)$ (Watkins, 1989). It is applicable when we are able to tabulate the optimal action values $q_*(s, a)$ for all states $s$ and actions $a$. In Q-learning, we update the stored value of $q_*(s, a)$ whenever the agent takes action $a$ in state $s$. This update is

proportional to the **temporal difference error** $\delta$ and the **learning rate** $\alpha$.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \underbrace{\left( R_t + \gamma \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a) - Q(S_t, A_t) \right)}_{\text{Temporal difference error } \delta} \tag{2.6}$$

## 2.2 Approximate methods

In many applications, the MDPs that arise either have finite state spaces that are too large for an action value function to be tabulated using existing computer hardware, or have infinite state spaces that cause tabulation to be impossible. In these scenarios, we can instead learn an approximation $Q_\theta(s, a)$ to the optimal action value function. Common choices of function approximators are linear functions of state features and neural networks. The parameters $\theta$ of the function approximators are optimised to give good estimates of the optimal action value function $q_*$, with returns sampled from the environment used as targets (Sutton and Barto, 2018). Deep neural networks with parameters optimised using gradient descent have been used to approximate $q_*$ for an agent playing Atari games (Mnih et al., 2013).

An alternative approach is to learn an approximation $\pi_\theta(a \mid s)$ of the optimal policy $\pi_*$ directly. When combined with gradient-based optimization, these methods are known as **policy gradient methods**. They aim to directly maximise the average return $J(\theta)$ obtained by the policy (Sutton and Barto, 2018):

$$J(\theta) = \sum_{s \in \mathcal{S}} d_{\pi_\theta}(s) \sum_{a \in \mathcal{A}(s)} \pi_\theta(a \mid s) q_{\pi_\theta}(s, a). \tag{2.7}$$

where $d_\pi$ is the stationary distribution on states of the Markov chain induced by the policy $\pi$. By the **policy gradient theorem** (Sutton and Barto, 2018), we have a simple form for the gradient of this objective:

$$\nabla_\theta J(\theta) \propto \sum_{s \in \mathcal{S}} d_{\pi_\theta}(s) \sum_{a \in \mathcal{A}(s)} q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a \mid s) \tag{2.8}$$

$$= \mathbb{E}_{\pi_\theta} \left[ q_{\pi_\theta}(s, a) \nabla_\theta \log \pi_\theta(a \mid s) \right]. \tag{2.9}$$

Policy gradient methods mainly differ in how they estimate this gradient and how

they apply the gradient to update the policy parameters $\theta$. One of the simplest policy gradient methods, REINFORCE, estimates values of $q_{\pi_\theta}(s, a)$ using returns from a sampled episode where the agent is following the policy $\pi_\theta$. The algorithm then updates the parameters via stochastic gradient ascent (Williams, 1992). This gives the following update after each episode:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} G_t \nabla_\theta \log \pi_\theta(a_t \mid s_t). \tag{2.10}$$

Although REINFORCE is simple, it can still be effective. It has been used to train a recurrent neural network to solve the travelling salesman problem and the knapsack problem, two examples of combinatorial optimization problems (Bello et al., 2016).

**Actor-critic methods** combine policy gradient methods with methods for approximating the action-value function $q_\pi$. The approximation $Q_\pi$ is used to give a better estimate of $q_{\pi_\theta}$ in the policy gradient for $\pi_\theta$. Proximal policy optimization (PPO) and asynchronous-advantage actor-critic (A3C) are two examples of actor-critic methods (Schulman et al., 2017; Mnih et al., 2016).

We can take a single view of all of these types of policy gradient methods using the idea of *generalised advantage estimation* (Schulman et al., 2015). The idea is that various policy gradient methods use an estimate of the gradient $\nabla_\theta J$ obtained from an expression of the form

$$\nabla_\theta J \propto \mathbb{E}\left[\sum_{t=0}^{T} \Phi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right]. \tag{2.11}$$

For REINFORCE, we have $\Phi_t = G_t$. For actor-critic methods, we may have $\Phi_t = Q(s_t, a_t)$, or $\Phi_t = Q(s_t, a_t) - V(s_t)$. The latter function is known as the **advantage function**, as it measures the advantage of taking the action $a_t$ compared to following the policy as normal.

### 2.2.1 Proximal policy optimisation

Proximal policy optimisation, or PPO, is an actor-critic method that aims to avoid taking policy update steps so large that they would cause a major drop

in agent performance (Schulman et al., 2017). There are two variants of PPO (PPO-Clip and PPO-Penalty) that achieve this in different ways. We will focus on PPO-Clip as it is conceptually simpler.

PPO-Clip avoids taking overly large steps by clipping the objective function. At each iteration step, we aim to maximise an objective based on the action-probability normalised advantages of state-action pairs:

$$\theta_{k+1} = \arg\max_{\theta} \mathbb{E}_{\pi_{\theta_k}} \left[ L(s, a, \theta_k, \theta) \right],$$

where

$$L(s, a, \theta_k, \theta) = \min \left( \frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a),\ A^{\pi_{\theta_k}}(s, a) + \varepsilon \cdot |A^{\pi_{\theta_k}}(s, a)| \right).$$

We typically maximise this objective using stochastic gradient ascent with transitions sampled according to the agent's current policy.

Here, $\varepsilon$ serves as a hyperparameter that we can tweak to adjust how far away the new policy can be from the old one.

The value function estimates are updated after each policy update using the mean-squared error loss over a set of stored trajectories.

## 2.3 Reward shaping

Often, the natural reward signal for a task may be *sparse*, meaning that the agent only receives a meaningful reward signal at a few steps. The most extreme case of this occurs when rewards are zero for all time-steps except for the transition into the terminal state. This is the case in the typical formulations of games like Chess or Go, where a win gives a reward of $+1$, a loss gives a reward of $-1$ and a draw or transition into any non-terminal state gives a reward of 0. This is also the case in our circuit routing problem, as the only feedback is the total length of the paths in our solution once all pairs have been connected. Reinforcement learning algorithms can be very sample-inefficient when rewards are sparse (Zheng et al., 2019).

Because of this phenomenon, it is common to modify the reward in an MDP in order to provide extra feedback to an agent and guide it towards the desired goals.

This practice is known as **reward shaping**. However, naïve reward shaping can lead to undesirable behaviour by the agent, as the policy that optimises the shaped reward can be different to the one that optimises the original reward (Amodei and Clark, 2016). Ng et al. (1999) investigated the conditions under which modifying the reward preserves the optimal policy and found that besides positive linear transformations of the reward, the only transformations that preserve the optimal policy in general are *potential-based reward transformations*.

We say $F : S \times S \to \mathbb{R}$ is a potential-based shaped reward with potential $\Phi : S \to \mathbb{R}$ if it has the following form:

$$F(s, s') = \gamma\Phi(s') - \Phi(s).$$

The shaped reward is then used during training by replacing the observed reward $r_t$ in a transition $(s_t, a_t, r_t, s_{t+1})$ with the reward $r_t + \gamma\Phi(s_{t+1}) - \Phi(s_t)$.

# Chapter 3

# Circuit routing

## 3.1 The circuit routing problem

We are interested in solving the problem of finding the best way to draw paths between fixed pairs of points on a plane such that no two paths come too close to eachother. This problem arises when trying to find the most efficient way to wire together components on a circuit board under the constraint that wires must not touch.

We can cast this problem as a combinatorial optimisation problem using the concept of *orbits* (Steffan, 2020). First, we collapse all paths between pairs onto an imaginary line passing through the center of mass of the points, as in Figure 3-1. We then generate a solution by sequentially selecting pairs of points and a direction (left or right) to *orbit* the path for those points when it gets too close to an existing path. Figure 3-2 shows the left and right orbits for a path. To prevent the wires from touching, each path must satisfy a design rule of having at least 15 units of clearance around it.

This formulation reduces our problem to choosing the sequence of pairs and left/right orbits that minimises the total length of the resultant paths, while still satisfying the design rules. For a problem with $n$ pairs, there are $n$ decision stages with $2(n-i)$ actions available at the $i$th decision stage, since we must choose one of the remaining $n-i$ pairs to connect and a direction to orbit in. This gives a total of $2^n n!$ possible ways to connect the points, making brute force search computationally expensive for even moderate values of $n$.

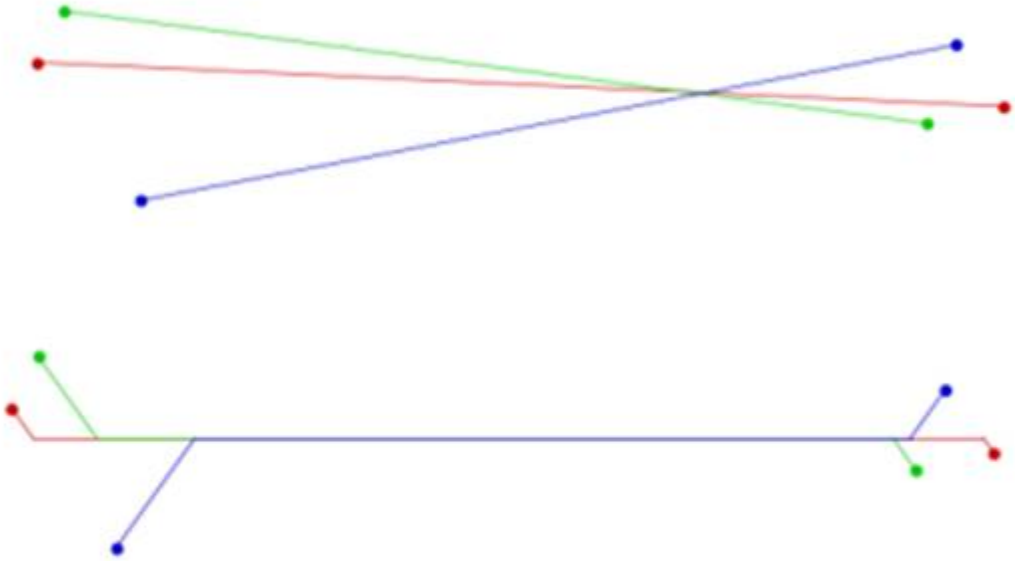

Figure 3-1: Collapsing paths onto an imaginary central line.

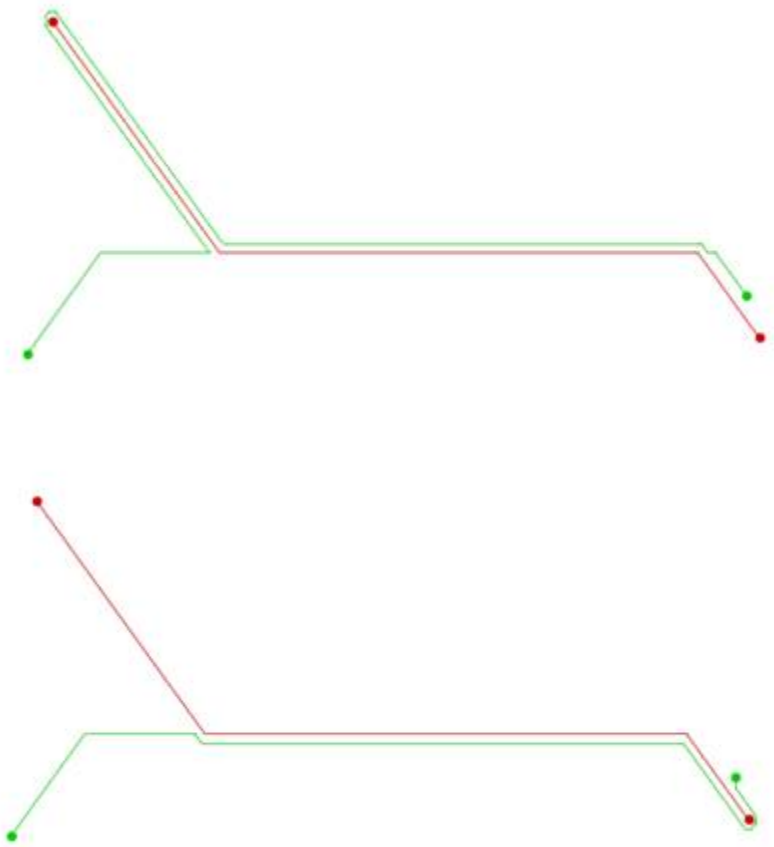

Figure 3-2: Top: Green path taking the left orbit around the existing red path. Bottom: Green path taking the right orbit around the existing red path.

## 3.2 The `copt` package

`copt` is a Python library supplied by Zuken for use in this project. It can generate random circuit routing problems, brute-force optimal solutions to given circuit routing problems, and evaluate the total path length of solutions given as sequences of base pairs and orbit directions (Steffan, 2020). These functionalities allow us to create a reinforcement learning environment for the circuit routing problem and generate optimal demonstrations for inverse reinforcement learning.

Problem instances are given as sequences of vectors in $\mathbb{R}^4$, where the first two coordinates of each vector correspond to the start point in a base pair, and the last two coordinates correspond to the end point. Solutions are given as sequences of 2-dimensional vectors, where the first coordinate denotes the base pair to connect (by its position in the problem statement sequence) and the second coordinate is a binary label denoting the direction of the orbit to use when connecting the pair.

## 3.3 Modelling circuit routing as a Markov decision process

Since the circuit routing problem is deterministic, the state at each time step is uniquely determined by the initial state and the sequence of actions chosen up to that point.

At the start of each episode, the initial state is chosen at random from a set of totally unconnected circuit routing problems. The number of base pairs in these problems can vary from 2 up to some predetermined $N$. The upper size limit $N$ can be increased to make the agent's task more challenging.

For a problem of size $n$, we can describe the initial state as an $n$-tuple of vectors in $\mathbb{R}^4$:

$$S_0 = (b_1, b_2, \ldots, b_n), \quad b_i \in \mathbb{R}^4.$$

We can describe any subsequent states $S_t$ with a tuple containing both

- the initial state $S_0$, in order to describe the problem instance;
- a tuple $(a_1, \ldots, a_t)$ containing the sequence of actions taken so far (i.e. the connected base pairs in order of connection):

$$S_t = (S_0, (a_1, \ldots, a_t)), \quad a_i \in \{1, 2, \ldots, n\}.$$

If we wish to specify the orbit directions for the base-pair connections, we can add an extra tuple of labels to the state description:

$$S_t = (S_0, (a_1, \ldots, a_t), (d_1, \ldots, d_t)), \quad d_i \in \{\texttt{L, R}\}.$$

A natural choice of sparse reward is given by the total path length of a complete solution. This would give a reward proportional to $-$(`total path length`) upon transitioning to the terminal state $S_n$. If we want our inverse reward model to learn an appropriate reward for transitioning to a terminal state $S_n$, then we must have distinct terminal states for all possible trajectories, rather than a single terminal state as is commonly used.

Some choices of actions may not be valid due to the paths being produced violating the design rules. We can deal with this in a 'hard' way by resampling actions from the policy whenever an action is invalid. Alternatively, we can deal with this in a 'soft' way by giving a large negative reward when actions that produce invalid paths are taken.

## 3.4 Issues with large circuit routing problems

In larger problems, we may run into difficulties in producing feasible solutions for two reasons:

1. The problem generator included in the `copt` package does not enforce a minimum distance between neighbouring start points for base pairs (similarly for end points). This may result in problems with no feasible solutions, as any order of connecting base pairs will violate the design rules. The probability of this occurring increases with the size of the problem.

2. The orbit heuristic used for generating routings may not scale well to larger problems, since collapsing all paths onto a central line and building outwards may not leave enough space for later connections between nodes in the middle of the routing area.

The first difficulty can be removed by simply resampling whenever two nodes are too close. The second difficulty is harder to fix, as it requires modifying the

heuristic used to generate solutions. One possible solution is to partition large routing problems into sub-problems and produce solutions for the sub-problems individually. The partitioning can be identified by iterating through the lists of start and end nodes of the base pairs (in vertical order) and finding the maximal matching subsequences.

## 3.5 Analysis of the circuit routing problem

We empirically analyse the circuit routing problem to better understand the difficulties a reinforcement learning agent may encounter while learning to solve it. For problems with six to nine base pairs, we brute-force every greedy-orbit solution for 100 instances of each size to find how many valid solutions there are, as well as the range of total path lengths for the valid solutions. We only consider problems where there are at least 30 units of distance between neighbouring nodes. This prevents us from counting problems where the design rules are fundamentally violated. Figure 3-3 shows the distribution of the number of valid solutions as a proportion of the overall search space, and Figure 3-4 shows how the range of total path lengths for valid solutions varies with the number of valid solutions for a problem.

For 71% of the problems of size 9, the valid solutions make up less than 0.14% of the search space, and all of the problems of size 9 have less than 1% of valid solutions as a proportion of the search space. Additionally, 17 of the problems of size 9 have no valid solution at all. This presents a challenge for a reinforcement learning agent learning from experience, as it is unlikely to produce valid solutions during the normal course of exploration.

Increasing the required clearance between neighbouring nodes from 30 to 45 does not significantly improve the situation for problems of size 9. 17 of the sampled problems still have no valid solution, and 78 of the problems have less than 0.28% of valid solutions as a proportion of the search space.

It may be the case that actual routing problems encountered during circuit design tend to have more valid solutions than the random routing problems generated by the copt module. Unfortunately, we do not have access to a data set of realistic circuit routing problems to analyse.

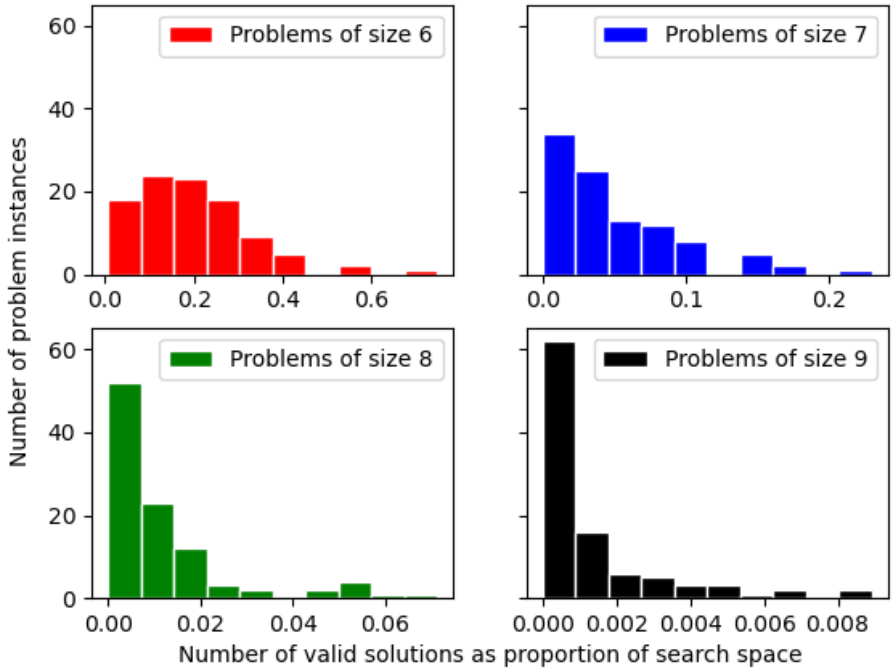


Figure 3-3: Distribution of the number of valid solutions (as proportion of the entire search space) for four different sizes of the circuit routing problem.

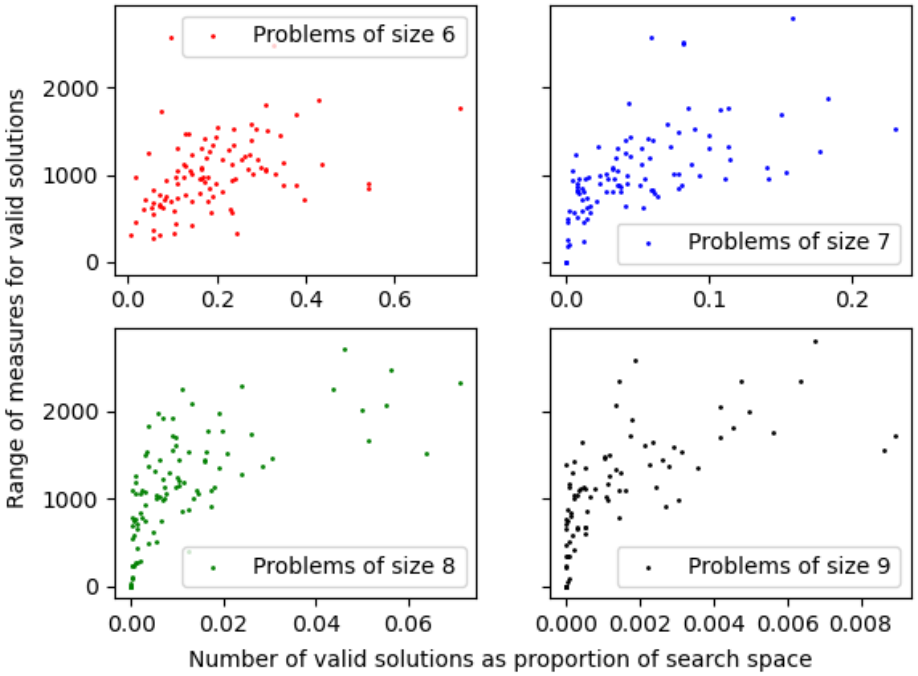


Figure 3-4: Range of total path lengths (maximum total path length − minimum total path length) against number of valid solutions for a problem.

# Chapter 4

# Reinforcement learning for combinatorial optimisation

The use of neural networks for combinatorial optimisation dates back to at least the 1980s (Smith-Miles, 1999), but early attempts did not have much success. They focused on supervised learning approaches, which are limited by the computational expense of obtaining optimal solutions as labels for a suitably large set of problems, and have poor ability to generalise (Bello et al., 2016).

Recent work (Bello et al., 2016) has shown success in training a recurrent neural network using reinforcement learning to solve moderately-sized combinatorial optimisation problems. The agent gave near-optimal solutions for traveling salesman problem instances with up to 100 nodes and gave optimal solutions for the knapsack problem with up to 200 items. They built on earlier work (Vinyals et al., 2015) that attempted to solve similar problems by training a recurrent neural network in a supervised manner. In the more recent paper, the recurrent neural net was trained as a policy network using an algorithm similar to A3C.

A major advantage of using a recurrent neural network to parametrise a policy is that it is able to generalise across problems of different sizes. For example, an agent trained to solve the travelling salesman problem on instances of up to 50 nodes could potentially solve larger problems of unseen size. If only feedforward neural networks could be used, we would need to train a completely separate network for each instance size, and any knowledge obtained from solving instances of one size could not be applied to solving instances of a different size.

## 4.1 Recurrent neural networks

A limitation of standard feedforward neural networks is that they can only accept inputs of a fixed size, where the size is determined by the network architecture. This is incompatible with natural choices of state representation for combinatorial optimisation problems, since the length of the representation will depend on the problem instance.

Recurrent neural networks (RNNs) can operate on sequences of variable size by utilizing the idea of *parameter sharing* (Goodfellow et al., 2016). The idea is to learn a single function $f_\theta$ which is then applied across all time steps.

Suppose we have a sequence of inputs $x^{(1)}, \ldots, x^{(T)}$. At each time step $t$, the function $f_\theta$ takes in the current input $x^{(t)}$ and the previous *hidden state* $h^{(t-1)}$ to produce a new hidden state $h^{(t)}$:

$$h^{(t)} = f_\theta(x^{(t)}, h^{(t-1)}). \tag{4.1}$$

A diagram of this form of RNN is shown in Figure 4-1.

We can think of the hidden state at time $t$ acts as a summary of all the inputs up to time $t$. The initial hidden state $h^{(0)}$ is usually treated as a trainable parameter of the RNN (Goodfellow et al., 2016).

The hidden states can be operated on in different ways to produce outputs of different shapes. For example, an RNN that maps a sequence to a sequence of the same size could apply the same function $g_\phi$ to each of the hidden states to get a sequence of outputs $g(h^{(1)}), \ldots, g(h^{(T)})$. An RNN that maps a sequence to a single value could apply a function to just the final hidden state, since it contains information about the whole sequence.

## 4.2 Pointer networks

Pointer networks (Vinyals et al., 2015) are an RNN architecture that can be used to parametrise a conditional probability distribution over a set whose size is equal to the input sequence. In terms of the circuit routing problem, this means that a pointer network can be used to evaluate the probability of a trajectory $\tau$, here

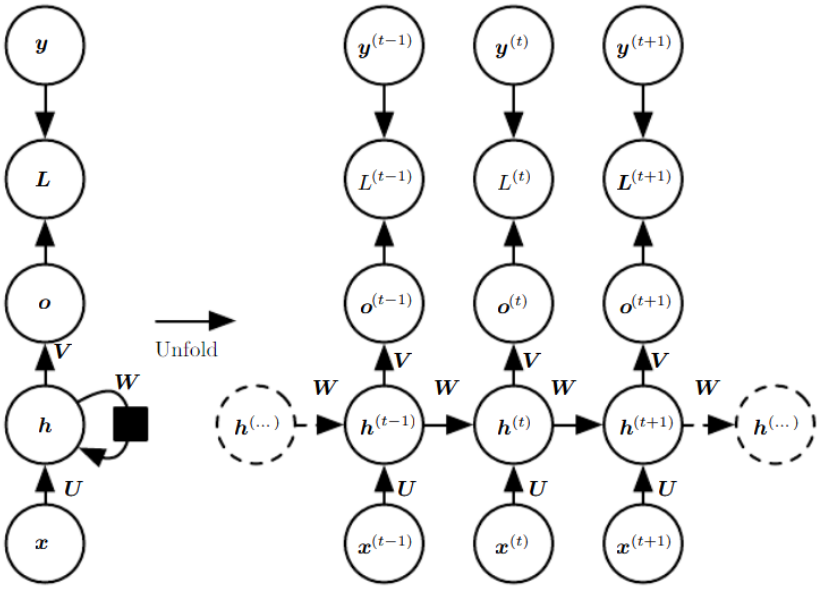


Figure 4-1: Diagram of how training loss is evaluated for a recurrent neural network, taken from (Goodfellow et al., 2016). Note how the same transformations are applied to each of the inputs $x^{(t)}$ and hidden states $h^{(t)}$ at each time step.

represented by the terminal state of the trajectory, $((b_1, \ldots, b_n), (a_1, \ldots, a_n))$:

$$q(\tau \mid b_1, \ldots, b_n; \theta) = \prod_{t=1}^{n} \pi_\theta(a_t \mid b_1, \ldots, b_n;\, a_1, \ldots, a_{t-1}). \tag{4.2}$$

To achieve this, the pointer network's encoder first produces a sequence of encoder hidden states $e_1, \ldots, e_n$ using the same method as in Equation 4.1, with the sequence of base pairs $b_1, \ldots, b_n$ as inputs.

The pointer network's decoder then steps through the sequence of actions $a_1, \ldots, a_n$, producing a sequence of decoder hidden states $d_1, \ldots, d_n$ and a conditional probability distribution $\pi_\theta(\cdot \mid b_1, \ldots, b_n;\, a_1, \ldots, a_{t-1})$ over all possible actions on the $t$th time step. The probability assigned to the $i$th action on the $t$th time step is given as follows (Bello et al., 2016):

$$u_{i,t} = \begin{cases} v^T \tanh(W_e e_i + W_d d_t) & \text{if } a_i \notin \{a_1, \ldots, a_{t-1}\} \\ -\infty & \text{otherwise.} \end{cases} \tag{4.3}$$

$$\pi_\theta(a_i \mid b_1, \ldots, b_n;\, a_1, \ldots, a_{t-1}) = \texttt{softmax}_i(u_{i,t}). \tag{4.4}$$

Since the energies $u_{i,t}$ (and hence the policy $\pi_\theta(\cdot \mid b_1, \ldots, b_n;\, a_1, \ldots, a_{t-1})$) only depends on the actions taken up to time $t-1$, we can use the pointer network for control by stepping through the decoder and sampling actions from the policy at each time step. Bello et al. (2016) use this method to sample trajectories and train the RNN by minimising expected solution cost with a policy gradient method.

When using the trained pointer network in deployment, we can use beam search to generate better solutions than those generated by simply sampling actions from the policy. Beam search is a heuristic search method for finding the optimal trajectory $q(\tau \mid b_1, \ldots, b_n; \theta)$ (Freitag and Al-Onaizan, 2017). At each time step while decoding, beam search selects the $n$ most probable actions as candidates for the next action (where $n$ is known as the *beam width*). It then branches on each of these candidates, and again selects the most probable $n$ actions for each of the branches. This process repeats until every trajectory terminates, and the trajectory with the lowest cost is selected.

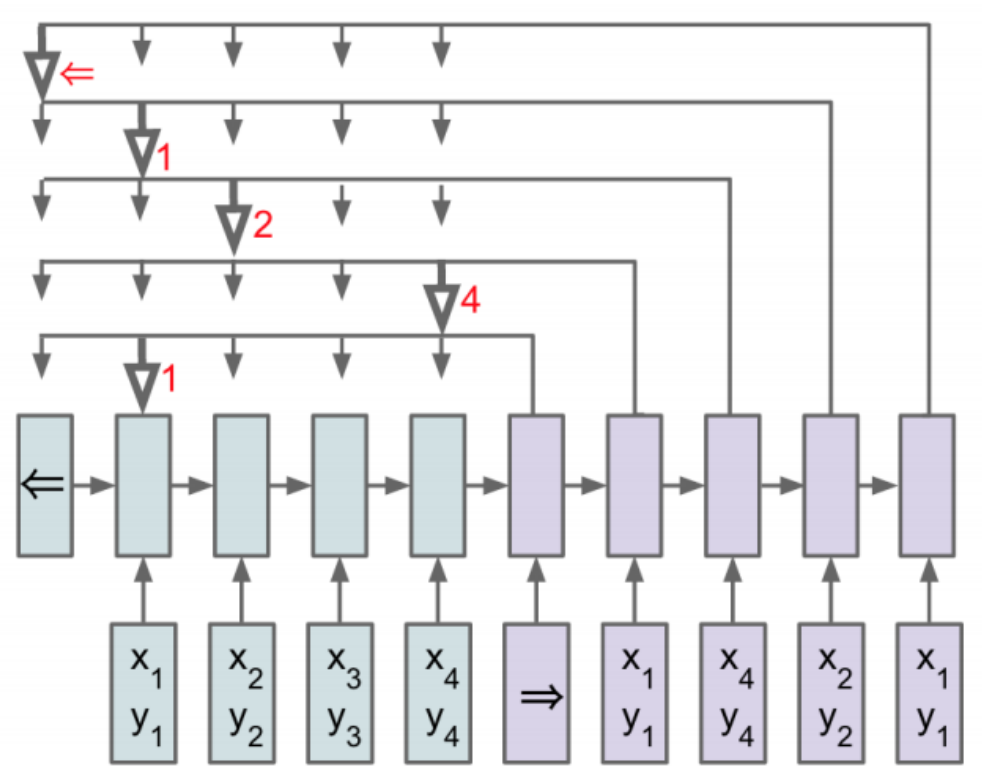


Figure 4-2: Diagram of pointer network taken from (Vinyals et al., 2015).

## 4.3 Transformers

Transformers are a type of sequence-to-sequence deep learning model originally created for use in large-scale natural language processing tasks (Vaswani et al., 2017). Like recurrent neural networks, a single transformer model is able to process input sequences and produce output sequences of arbitrary length. Unlike RNNs, they do not require recurrent computation of hidden states and instead rely solely on attention mechanisms, which can be computed in parallel for entire sequences. Due to the increased level of parallelisation, transformers can achieve better performance in a shorter amount of training time than recurrent nets for a variety of sequence modelling tasks (Vaswani et al., 2017; Karita et al., 2019).

The transformer model contains an encoder module and decoder module, with each module made up of a number of encoder and decoder layers. Typically, the encoder module processes the input sequence to create embeddings for each element of the input, which are then used by the decoder module to produce the outputs. At the heart of each encoder and decoder layer lies the *multi-head attention* mechanism. This is responsible for modelling the interaction between elements of the input and output sequence, as well as inter-input and inter-output interactions.

### 4.3.1 Multi-head attention

The input to each head of the multi-head attention module consists of a number of queries and key-value pairs. These queries, keys and values are vectors obtained from the input or output sequence through learned linear mappings for each attention head. Each head $h$ computes the compatibility score $u_{ij}^{(h)}$ between the $i$th query $q_i$ and the $j$th key $k_j$ via *scaled dot-product attention*:

$$u_{ij}^{(h)} = \frac{(W_h^Q q_i)^T (W_h^K k_j)}{\sqrt{d_k}}.$$

Here, $W_h^Q$ and $W_h^K$ are the learned linear maps for the queries and keys in head $h$, while $d_k$ is the dimension of the queries and keys.

The compatibility scores are then normalised to sum to 1 over keys using softmax, so that they can be used to compute weighted sums of values as outputs

for each head and query:

$$a_{ij}^{(h)} = \texttt{softmax}_j(u_{ij}^{(h)}),$$

$$o_i^{(h)} = \sum_j a_{ij}^{(h)} W_h^V v_j.$$

Finally, the output for query $i$ from the multi-head attention module is created by concatenating together all the outputs from different heads and applying a learned linear map $W^O$.

$$o_i = W^O[o_i^{(1)}, o_i^{(2)}, \ldots, o_i^{(N)}].$$

The query, key and value inputs change depending on the interactions the module is trying to capture. For the multi-head attention modules in the encoder layers, the queries, keys and values all correspond to the input sequence, as the encoder aims to produce embeddings of the inputs that capture information about the interactions between elements of the input sequence.

The first multi-head attention module in each decoder has queries, keys and values corresponding to elements of the output sequence so far in order to capture interactions between the output. However, the second module in each decoder layer has a query for each element of the output sequence and a key-value pair for each input embedding created by the encoder, allowing it to model the interactions between the input and output sequence.

To prevent elements from interacting with eachother, the compatibility scores $u_{ij}^{(h)}$ can be set to $-\infty$, which sets the corresponding weight $a_{ij}^{(h)}$ to zero. This is known as *attention masking* and is typically used to prevent output elements from querying future output elements.

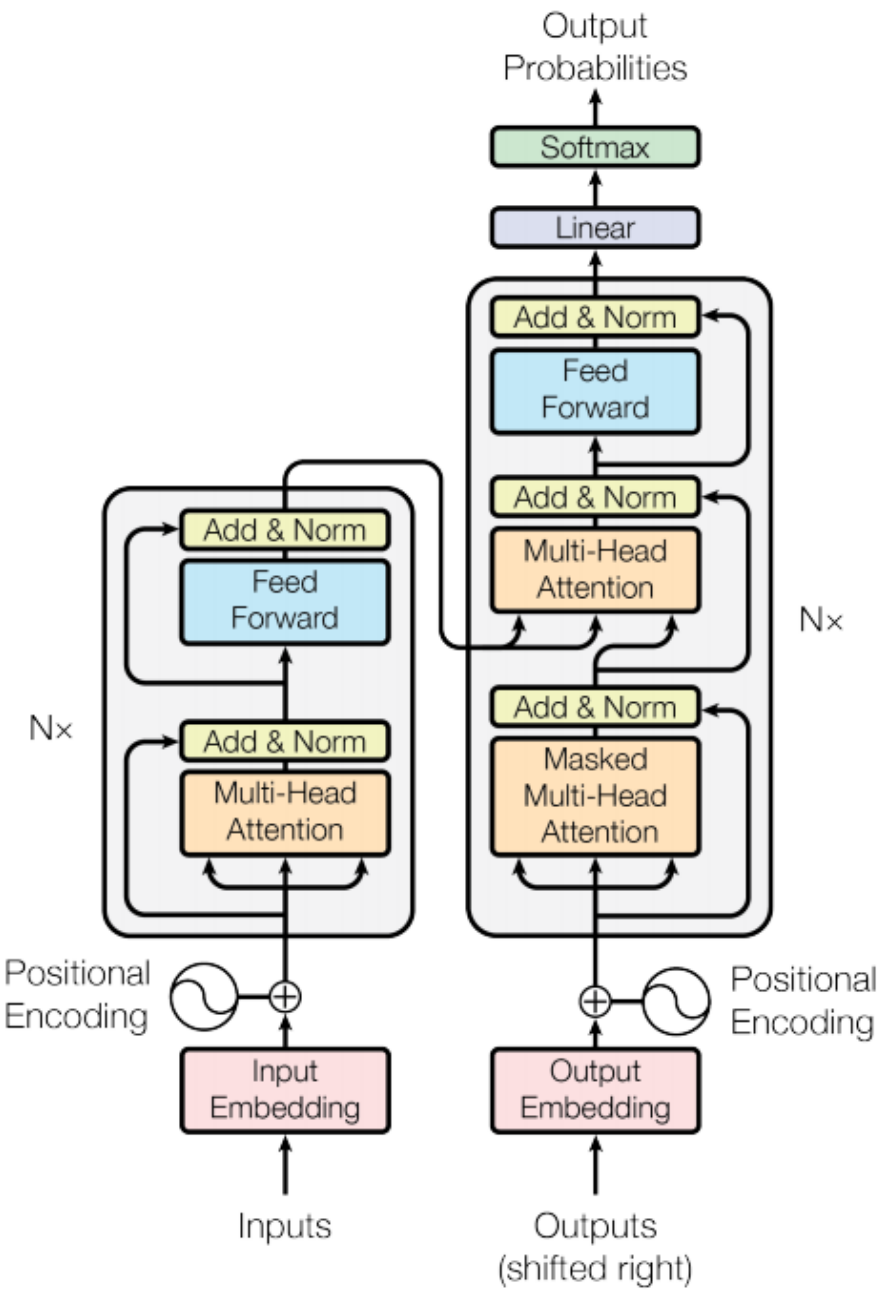


Figure 4-3: Diagram of the high-level architecture of the original transformer model (Vaswani et al., 2017).

# Chapter 5

# Inverse reinforcement learning

As an inverse problem, IRL is ill-posed in the sense that there is always more than one reward function that explains observed behaviour (Ziebart et al., 2008). In particular, any behaviour is optimal when the reward function gives a reward of zero for all transitions (Ng and Russell, 2000). Theoretical results on reward-shaping show that optimality of behaviour is preserved by the addition of a potential-based reward function to the current reward (Ng et al., 1999), as well as by affine transformations of the reward. Due to the non-uniqueness of solutions to this problem, any successful IRL method must therefore incorporate some form of **regularisation**, i.e. they must add some additional criteria to get a good reward function.

Different methods use different criteria to select the reward function. Many early methods seek reward functions that harshly penalise any deviation from the observed behaviour (Ng and Russell, 2000; Ratliff et al., 2006; Abbeel and Ng, 2004; Syed and Schapire, 2007). These are known as *maximum-margin methods*, as they aim to maximise the margin between the reward obtained by the policy underlying the observed behaviour and the reward obtained by any other policy.

Another class of methods seeks to maximise the likelihood of the demonstrated behaviour under the assumption that trajectories are sampled from a Boltzmann distribution parametrised by the reward function. The energy (un-normalised log-probability) of a trajectory is given by the sum of its rewards. These are known as *maximum-entropy methods* (Ziebart et al., 2008).

## 5.1 Classical inverse reinforcement learning methods

### 5.1.1 Maximum-margin methods

The first algorithm for inverse reinforcement learning from sampled behaviour was introduced in (Ng and Russell, 2000). The focus of the paper is on methods for extracting reward functions under which the expert's behaviour would be optimal. The most general algorithm they give learns a reward that is a linear function of some pre-defined features $\phi_i : \mathcal{S} \to \mathbb{R}$ of a continuous state space:

$$R(s) = \alpha_1\phi_1(s) + \alpha_2\phi_2(s) + \cdots + \alpha_d\phi_d(s).$$

This algorithm is heavily limited in terms of general applicability by the need to manually pick good features $\phi_1, \ldots, \phi_d$ of the states. Additionally, each iteration of the algorithm requires solving the MDP for the optimal policy, making it unsuitable for large problems.

Abbeel and Ng (2004) build on this method by introducing the concept of matching *feature expectations*. Let $\phi(s) = (\phi_1(s), \ldots, phi_d(s))^T$ be the feature vector. The feature expectations of a policy $\pi$ is then given by

$$\mu(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \phi(s_t) \mid \pi\right] \in \mathbb{R}^d. \tag{5.1}$$

Abbeel and Ng then show that the problem of finding weights $w$ that recover the expert's reward function $R = w^T\phi$ can be reduced to finding weights that minimise the difference between the feature expectations for the optimal policy w.r.t. those weights, $\mu(\pi_w)$, and an empirical estimate of the expert's feature expectations, $\mu_E$.

### 5.1.2 Maximum-entropy inverse reinforcement learning

Ziebart et al. (2008) introduced the maximum-entropy framework for inverse reinforcement learning. They observed that the method proposed by Abbeel and Ng (2004) for matching feature counts was ambiguous, as many policies could lead to the same feature counts. To resolve this ambiguity, they show that the Boltzmann distribution over trajectories with energies proportional to cumula-

tive reward is the maximum-entropy distribution over trajectories that matches expected feature counts.

The likelihood of a trajectory in an MDP with reward function $R(s)$ under the Boltzmann (a.k.a. maximum-entropy) distribution is

$$p(\tau) = \frac{1}{Z} \exp\left(\sum_{t=0}^{\infty} \gamma^t R(s_t)\right). \tag{5.2}$$

For a linear reward function in an MDP with no discounting, this becomes

$$p(\tau \mid w) = \frac{1}{Z(w)} \exp\left(\sum_{t=0}^{\infty} w^T \phi(s_t)\right). \tag{5.3}$$

Under this formulation, the problem of finding weights that explain expert behaviour is equivalent to finding the weights that maximise the likelihood of the expert trajectories in this model. They also show that the maximum likelihood weights match the feature expectations between the distribution and the expert.

Like other early IRL methods, the algorithm for inferring the weights $w$ requires knowledge of the environment dynamics, and is limited by the need to engineer good features.

## 5.2 Deep inverse reinforcement learning methods

In this section, we review some recent IRL and imitation learning techniques that are suitable for use on large problems where we do not have knowledge of the environment dynamics and finding an optimal policy in the inner loop is infeasible.

### 5.2.1 AggreVaTeD

AggreVaTeD (Sun et al., 2017) is an imitation learning method, meaning that it learns a policy directly from demonstrations instead of learning a reward function as an intermediate step. It operates similarly to typical policy gradient methods from reinforcement learning, but instead of using the agent's estimate of the action-value $Q_\pi(s, a)$ in the gradient, it uses the expert's estimate of the action-value under the expert's policy.

It can be used to learn a policy parametrised by a neural network (including recurrent networks), but requires access to an expert that can generate near-optimal solutions to any task during training time. This method could potentially be used for the circuit routing problem with brute-force search used to generate expert demonstrations during training.

This method is not ideal as the brute-force search does not scale to larger instances of the circuit routing problem. It also does not generalise to problems where we do not have access to an expert during training time. However, it could be used to initialise a policy by imitation-learning optimal solutions for small problems.

### 5.2.2 Truncated horizon policy search (THOR)

Truncated horizon policy search or THOR (Sun et al., 2018) also uses an expert to generate demonstrations during training time, but instead uses the generated demonstrations to learn a **potential-based shaped reward function**.

As we stated earlier, optimal policies are invariant under the addition of potential-based shaped rewards to the original reward (Ng et al., 1999). We say $F : S \times S \to \mathbb{R}$ is a potential-based shaped reward with potential $\Phi : S \to \mathbb{R}$ if it has the following form:

$$F(s, s') = \gamma\Phi(s') - \Phi(s).$$

The shaped reward is then used during training by replacing the observed reward $r_t$ in a transition $(s_t, a_t, r_t, s_{t+1})$ with the reward $r_t + \gamma\Phi(s_{t+1}) - \Phi(s_t)$.

The proposed potential function $\Phi(s)$ for the shaped reward is the expert's estimate of the reward-to-go from that state. For a combinatorial optimisation problem where the MDP models the construction of a solution, the potential would be the negative cost of the best solution with prefix $s$.

### 5.2.3 Guided cost learning (GCL)

Guided cost learning is a method for learning a reward function parametrised by a neural network from demonstrations (Finn et al., 2016b). It is a non-linear generalisation of the maximum-entropy IRL method (Ziebart et al., 2008) that

also has the considerable advantage of not requiring the optimal policy to be found in its inner loop. Instead, GCL updates the learned reward along with the policy.

The main contribution of the GCL algorithm is a sample-based method for approximating the partition function $Z$ in the maximum-entropy model of the expert's trajectories. It does this by performing importance sampling of trajectories from a learned policy $\pi$.

Let $R : S \to \mathbb{R}$ be a learned deterministic reward function parametrised by a neural network with weights $\theta$. Here, $R(s)$ gives the reward received when transitioning into a state $s$. Let $D_{demo}$ be the set of trajectories demonstrated by an expert and $D_{samp}$ the set of trajectories sampled by the agent from the environment. When $\tau$ is a trajectory, we use $R(\tau)$ to denote the sum of rewards for that trajectory, i.e.

$$R(\tau) = \sum_{t=1}^{T} R(s_t)$$

The parameters $\theta$ of the learned reward $R$ are adjusted using stochastic gradient ascent to maximise the following objective:

$$\mathcal{L}(\theta) = \frac{1}{|D_{demo}|} \sum_{\tau_i \in D_{demo}} R(\tau_i) \; - \log \left[ \frac{1}{|D_{samp}|} \sum_{\tau_j \in D_{samp}} w_j \exp(R(\tau_j)) \right].$$

This objective is an estimate of the log-likelihood of the demonstrated trajectories under the maximum-entropy model. The second term in the objective is an estimate of the $\log Z$ term.

Experiments in the paper show that GCL has higher sample efficiency and performance on continuous control tasks than a similar method, relative-entropy IRL (Boularias et al., 2011). However, the authors use continuous control-specific regularization methods to impose structure on the learned reward for these experiments. This may mean that the performance of GCL when applied to combinatorial optimization will be dependent on finding a suitable regularization method.

### 5.2.4 Adversarial inverse reinforcement learning

Adversarial inverse reinforcement learning (AIRL) was first introduced by Fu et al. (2017) as an extension of the generative adversarial imitation learning (GAIL) al-

gorithm proposed by Ho and Ermon (2016). In GAIL, a GAN (Goodfellow et al., 2014) is trained to both generate trajectories and discriminate between the generated trajectories and expert trajectories. By learning to produce trajectories that effectively fool the discriminator, the generator produces a policy that imitates the expert's behaviour. However, GAIL does not learn a reward function as a part of this process, which is problematic if we want to further refine the policy using reinforcement learning. AIRL extends GAIL to learn a reward as an intermediate step.

The discriminator in a GAN takes the form of a neural network $D$ that outputs the probability that a sample came from real data, as opposed to being produced by the generator. The discriminator in AIRL aims to classify individual transitions from demonstrated trajectories or trajectories produced by a learned policy, and has the following form:

$$D_{\theta,\phi}(s, a, s') = \frac{\exp[f_{\theta,\phi}(s, a, s')]}{\exp[f_{\theta,\phi}(s, a, s')] + \pi(a \mid s)}. \tag{5.4}$$

Here, $f_{\theta,\phi}$ has the form

$$f_{\theta,\phi}(s, a, s') = g_\theta(s, a) + \gamma h_\phi(s') - h_\phi(s) \tag{5.5}$$

where $g_\theta$ is a reward term and $h_\phi$ is a shaping term.

The policy $\pi$ is improved using the reward $r(s, a, s') = \log D(s, a, s') - \log(1 - D(s, a, s'))$. For the above form of the discriminator, this simplifies to

$$r(s, a, s') = f_{\theta,\phi}(s, a, s') - \log(\pi(a \mid s)).$$

The $-\log \pi(a \mid s)$ term is a result of the *entropy-regularized RL* setting used in this paper, which aims to produce policies that are as random as possible in order to encode "all of the ways of performing the task" (Haarnoja et al., 2017).

The authors also show that for a deterministic MDP, the optimal form of $g_\theta$ is equal to the ground-truth reward function and the optimal form of $h_\phi$ is equal to the optimal state-value function $V_*$ (up to additive constants). Since we know the ground-truth reward function for the circuit routing problem, we could fix $g_\theta$ to the ground-truth reward and then use AIRL to just learn a potential function

$h_\phi$.

Like Finn et al. (2016a), the experiments in the paper only test performance on continuous control tasks, with a particular focus on the suitability of the method for transfer learning (i.e. applying the learnt reward function to an environment with different dynamics). Tucker et al. (2018) test AIRL on Atari games, which have a discrete action space and a high-dimensional state space. AIRL performed poorly on the Atari games, and the authors were only able to obtain good performance on another simple game after using autoencoders to reduce the dimensionality of the input to the reward network for the discriminator. Although this is not encouraging when considering the applicability of AIRL to the circuit routing problem, it is possible that the extra structure given by using ground-truth rewards to replace $g_\theta$ in the circuit routing task could lead to good performance.

One further extension to AIRL is empowerment-regularised maximum-entropy IRL (Qureshi et al., 2019), which replaces the potential term $h_\phi$ with an *empowerment function* that measures the quantity of future states reachable from a given state. With an appropriately constructed MDP, empowerment could be used to quantify how likely it is that a given sequence of actions leads to a feasible solution for a combinatorial optimizaiton problem. The empowerment potential function could also be added to a regular potential function learnt using AIRL to give feedback about both feasibility and solution quality.

Another extension is given by Blondé and Kalousis (2018), who integrate AIRL with an actor-critic architecture in order to improve the sample efficiency of AIRL. This algorithm is intended for use in continuous control tasks and would require modification to be used in the circuit routing task, as it relies on a deterministic policy parametrised by a neural network that outputs a value in $\mathbb{R}^n$ to represent the action. Another paper by the same authors (Blondé et al., 2020) further improves this method by determining that local Lipschitz-continuity of the learned reward function is a necessary condition for the method to perform well, and gives a method for enforcing this by adding a gradient penalty during training.

Other extensions include combining behavioural cloning (learning a policy from demonstrations in a supervised manner) with GAIL to improve sample efficiency (Jena et al., 2020), learning options using AIRL (Venuto et al.) and using positive-unlabelled learning to decrease the sparsity of rewards learned using AIRL (Xu and Denil, 2019).

# Chapter 6

# Tackling the circuit routing problem

We aim to use reinforcement learning to learn a policy for the circuit routing problem that produces valid solutions with minimal total path length.

From our prior analysis of the circuit routing problem, we see two main difficulties:

1. It is only possible to evaluate the success of complete solutions. Even though we can evaluate partial solutions to see if all of the connections up to that point are successful, this doesn't give us useful feedback because there may be no way to extend a successful partial solution to a successful complete solution. Due to this, we can only construct a terminal reward from the evaluation, which does not tell us which actions were instrumental in making a good solution.

2. Large problems typically have a very small number of valid solutions, so nearly all trajectories sampled during early training will not provide very useful feedback on how to construct valid solutions.

Since smaller problems are easier to learn to solve, we ideally want to learn a policy in a way such that knowledge of how to solve smaller problems can be transferred to solving larger problems. For this reason, we use a single neural network to parametrise the policy for all problem sizes.

## 6.1 Overview

The policy $\pi$ is parametrised by a transformer model that has been modified to output probabilities over inputs in a manner similar to how Vinyals et al. (2015) modified recurrent neural networks to create pointer networks. We also use a separate transformer to compute the shaped reward terms $h$ , which are learnt by using them to classify transitions as either being produced by the current policy or coming from brute-forced solutions.

We choose to use a transformer model over a recurrent neural network model due to its ability to process large batches of sequences in parallel. The resultant performance gains will be most significant for computing embeddings of base pairs and when computing shaped reward terms for trajectories, as no sequential decoding is required for these tasks. This allows us to compute these in a single pass through the transformer model, rather than having to do a pass for each sequence item as in a recurrent net.

Additionally, recent work (Kool et al., 2019) has shown that a transformer model can produce significantly better solutions for a variety of TSP-related tasks compared to a pointer network model.

## 6.2 Reward function

As the MDP is deterministic, we specify the reward function in the form $R(s', s)$ which specifies the reward received when transitioning from state $s$ to state $s'$. The reward $R$ has the form

$$R(s', s) = T(s') + h(s') - h(s),$$

where $T$ is a terminal reward and $h$ is a learned shaping term.

The terminal reward $T$ is fixed, and measures the objective that we want the agent to achieve - namely, finding minimal valid solutions.

$$T(s) = \begin{cases} 2 - \frac{\text{total path length}}{1000 \times (\text{number of base pairs})} & \text{if the solution } s \text{ is valid;} \\ -5 & \text{if } s \text{ violates the design rules;} \\ 0 & \text{if } s \text{ is only a partial solution.} \end{cases}$$

This terminal reward lies approximately between 0 and 1 for valid solutions, with higher rewards corresponding to shorter solutions.

## 6.3 Policy network

We use a single transformer-based model to parametrise the policy for all problem sizes. The transformer encoder produces embeddings of the base pairs, and the transformer decoder uses these embeddings to compute the action probabilities.

### 6.3.1 Encoder

Each base pair in the input sequence is first mapped to a higher-dimensional representation with a learned affine map, $x \mapsto Wx + b$. We do not add any positional encodings as in Vaswani et al. (2017), since the coordinates of the base pair already encode their positions in the input sequence and this is preserved by the affine map. The higher-dimensional representations of the base pairs are then passed through three transformer encoder layers to produce representations for use in the decoder.

In each encoder layer, we allow all base pairs to attend to all other base pairs, since in general we cannot guarantee that the presence of one base pair will not impact the order we wish to connect all other base pairs. It is possible to restrict attention between base pairs when they are independent (e.g. if sets of base pairs do not cross), but we do not explore this avenue here.

### 6.3.2 Decoder

The decoder sequentially produces probabilities for the next base pair to connect. The first three layers of the decoder are identical to those of a regular transformer decoder, but the last layer is modified so that each output is a probability distribution over the inputs to the encoder rather than a fixed-dimensional vector.

At the first time step, the input to the decoder consists solely of a special "beginning-of-sequence" token, which is mapped to a learned vector. At further time steps $t$, the input sequence consists of the beginning-of-sequence token as well as the base pairs chosen at the previous time steps. Like the encoder, the base

pairs are mapped to a higher-dimensional representation using a learned affine function.

Each input in the decoder is allowed to attend to decoder inputs corresponding to previous time steps, as well as encoder outputs corresponding to base pairs that have not yet been connected.

In the final decoder layer, rather than using the attention weights from each head to compute a weighted sum of the values, we average the compatibility scores $u_{ij}^{(h)}$ over all heads before clipping to produce logits. The logits are then masked to prevent base pairs that have already been routed from being selected, before being normalised into action probabilities via softmax.

$$u_{ij} = \frac{1}{N} \sum_{h=1}^{N} u_{ij}^{(h)}$$

$$\bar{u}_{ij} = \begin{cases} 10 \cdot \tanh u_{ij} & \text{if base pair } j \text{ has not yet been routed at step } i \\ -\infty & \text{if base pair } j \text{ has already been selected.} \end{cases}$$

$$\pi(j \mid s_i) = \texttt{softmax}_j(\bar{u}_{ij}).$$

This method of calculating action probabilities from the final layer of a transformer decoder is similar to that given by Kool et al. (2019), except that we do not only use a single attention head.

| **Model hyperparameter** | **Value** |
|---|---|
| Number of encoder layers | 5 |
| Number of decoder layers | 4 |
| Input embedding dimension | 128 |
| Number of attention heads | 8 |
| Feedforward hidden units | 512 |
| Normalisation type | Layer normalisation |

Table 6.1: Model hyperparameters for the policy transformer.

## 6.4 Discriminator

To learn shaped rewards, we train a discriminator with the goal to classify state transitions as either being produced by the policy during training or as being sampled from the expert data. Inspired by adversarial imitation learning (Fu et al., 2017), our discriminator's predicted probability that a state transition $(s, s')$ belongs to the expert class is given by

$$D(s, s') = \frac{\exp(h(s') - h(s))}{\exp(h(s') - h(s)) + 1},$$

where $h$ is the shaping term in our reward function.

The shaping terms $h$ are predicted by a standard transformer model with the same architecture as the policy network, excluding the modified final decoder layer. To produce scalar values $h(s)$ from the decoder output, we add a feedforward network with two hidden layers of width 512 and ReLU activations to the end of the decoder.

The discriminator predicts that the transition comes from the expert data when $D(s, s') > 1/2$, which is equivalent to when $h(s') - h(s) > 0$, and predicts that the transition comes from the training policy when $h(s') - h(s) < 0$. With an appropriately trained discriminator, the shaping terms will give a positive reward to transitions that mimic the successful solutions given in the expert data and a negative reward to other transitions.

### 6.4.1 Expert data

To produce the expert data, we brute-forced solutions to a number of circuit routing problems of different sizes. Details of the expert data-set are given in Table 6.2. Since larger problems take a significant amount of time to brute-force, we saved a number of top solutions for larger problems rather than only the top solutions. Besides practical reasons, including sub-optimal trajectories within the expert data is useful for testing the general applicability of this method, since it is difficult to gather a large number of optimal solutions for combinatorial optimisation problems in general.

| **Problem size** | **Number of unique problems** | **Number of top solutions stored** | **Total number of solutions** |
|---|---|---|---|
| 3 | 25,000 | 1 | 25,000 |
| 4 | 25,000 | 1 | 25,000 |
| 5 | 12,500 | 2 | 25,000 |
| 6 | 5,000 | 5 | 25,000 |
| 7 | 5,000 | 5 | 25,000 |
| 8 | 100 | 10 | 1,000 |
| 9 | 100 | 20 | 2,000 |

Table 6.2: Details of the expert data set used for learning the shaped reward.

## 6.5 Training procedure

We train the policy network via gradient descent on a PPO loss (Schulman et al., 2017), and train the discriminator via gradient descent on cross-entropy loss. We use Adam (Kingma and Ba, 2015) as a gradient optimiser due to its suitability on a variety of problems. After a random search over training hyperparameters, we found that the hyperparameters in Table 6.3 gave the best balance between performance and stability.

At each training step, we select a random problem size between 6 and 9 and then sample 4 unique circuit routing problems of that size. Each sampled problem is checked to ensure that there are at least 30 units of distance between neighbouring start and end points of base pairs, since wires must have at least 15 units of clearance for the solution to be successful. We focus on training on larger problems as these are more difficult than smaller problems, and we found that the policy is able to successfully generalise to smaller problems.

For each unique problem, we then sample 128 trajectories according to the current policy, giving a total of 512 trajectories at each step. We only use 4 unique problem instances in each training to increase the probability that the agent sees a successful solution for a given instance.

After sampling trajectories from the policy, we sample 512 trajectories of the same length from the expert data and compute shaping terms $h(s)$ for each state in both the agent's trajectories and expert's trajectories. Class probability predictions $D(s, s')$ are then computed for all state transitions, and cross-entropy loss is computed using these predictions.

The shaping terms $h(s)$ are then used to compute returns for all agent trajectories, and PPO loss is computed using these returns and action probabilities. Finally, the policy and shaping net parameters are updated simultaneously using gradient descent on the PPO and cross-entropy losses.

| **Training hyperparameter** | **Value** |
|---|---|
| Learning rate | 1e-5 |
| Number of trajectories per policy batch | 512 |
| Number of unique problem instances per policy batch | 4 |
| Number of expert trajectories per discriminator batch | 512 |
| Minimum circuit routing problem size | 6 |
| Maximum circuit routing problem size | 9 |

Table 6.3: Training hyperparameters for the policy and reward-shaping transformers.

# Chapter 7

# Experiments

In the first experiment, we investigate how the use of learned shaped rewards, as well as different architectures for the shaped reward model, affects the learning efficiency of the agent. The second experiment evaluates how useful the learned policy is for constructing solutions via beam search in terms of success rate, total path length, and computation time. The third experiment investigates how useful the learned shaped rewards are for finding solutions for individual problems via an "active search" method (Bello et al., 2016).

## 7.1 Effects of shaped reward on training

We evaluate the learning progress of the agent in terms of success rate for solutions sampled during training, as well as the average terminal reward of sampled solutions. We test three different agents:

1. Agent with no shaped reward.
2. Agent with shaped reward learnt using a separate reward-shaping network.
3. Agent with shaped reward learnt using a reward-shaping network that shares an encoder with the policy network.

Each agent was trained for 4000 steps (approximately 4.5 hours).

We do not report policy loss as a measure of training progress. As the policy loss depends on the rewards during training, the different rewards given to each

agent make the losses incomparable. Additionally, increases or decreases in policy loss for the agents with learned shaped rewards may not even correspond to negative or positive changes in policy quality, as the rewards given for the same actions change during training.

Figures 7-1 and 7-2 show the training metrics at each training step. Since success rate depends heavily on the size of the problem, these graphs have been smoothed to aid comparison between methods.

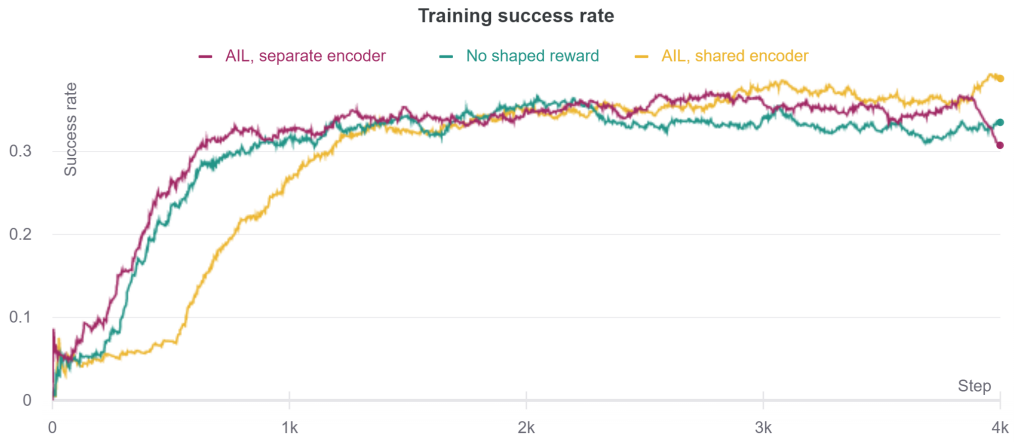


Figure 7-1: Comparison of success rate with respect to number of training steps for three agents with different shaped rewards.

Disappointingly, there does not seem to be much difference between the learning progress of the agent with the shaped reward given by the separate model and the agent with only terminal rewards. The first shaped-reward agent appears to learn marginally faster than the agent with no shaped reward, but this could be attributed to differences in parameter initialisation or training. Additionally, the agent with the shared policy-reward encoder learns much slower than both of the other agents. This may be due to the fact that the encoder weights are updated using gradients from both the policy loss and discriminator loss, which can conflict with each other.

There are several possible explanations for the lack of improvement. One is that the discriminator is only able to provide a useful reward later in training, at which point learning progress may be frustrated by another factor, such as the capacity of the policy model to express a good policy. From Figure 7-3, we can see

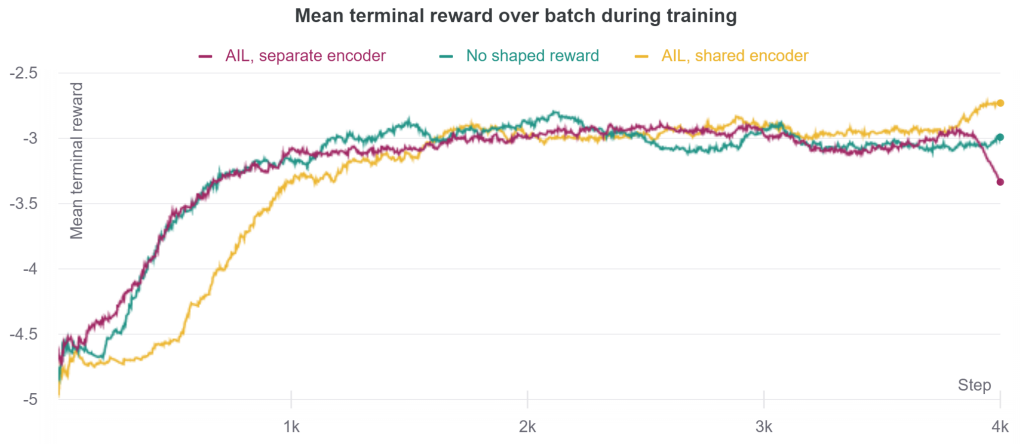


Figure 7-2: Comparison of terminal reward with respect to number of training steps for three agents with different shaped rewards.

that the false positive and false negative rates of the discriminator only reach a low level after approximately 2000 steps. In this context, a false positive corresponds to classifying a state transition caused by the agent as coming from the expert data set and assigning it a positive reward, while a false negative corresponds to classifying an expert transition as the agent's transition and assigning it a negative reward. Ideally, we would like false negatives to be as low as possible so that the discriminator rewards actions that mimic the expert, while false positives are a sign that the agent is fooling the discriminator by mimicking the expert. However, the false negative rate is high early on in training, suggesting that the discriminator can not supply appropriate rewards at that point.

Another related possibility is that the quality of the expert data is not sufficient to learn a good shaped reward. Since there are only a small number of unique problem instances in the data for problems of size 8 and 9, it is possible for the encoder in the discriminator to overfit to the supplied problem instances rather than learning representations that are useful for classifying transitions in general.

One last issue may lie in how the classification probabilities are calculated from the decoder outputs. Since the logits $h(s') - h(s)$ used to calculate classification probabilities are differences of decoder outputs $h(s)$, the decoder outputs must either monotonically increase or monotonically decrease along a trajectory in order to correctly classify it as an expert or agent trajectory. The $h(s)$ terms for later

states in trajectories should then generally have higher magnitudes than those of earlier states. However, since all parameters in a transformer are shared between time steps, the model may have difficulties fitting to the appropriate range of values for each time step.

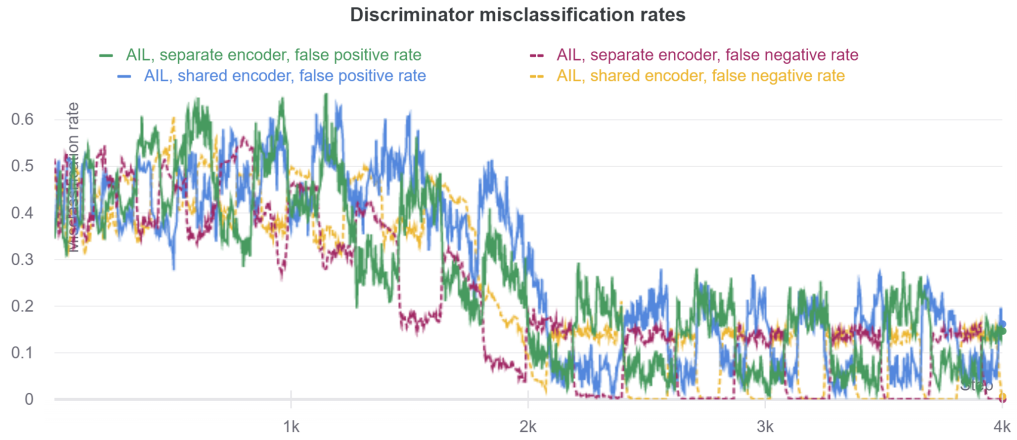


Figure 7-3: False positive and false negative misclassification rates for the discriminator in the shared and separate encoder architectures.

## 7.2 Performance of trained agents

In this experiment, we evaluate how effective the learned policies are at generating solutions through a beam search procedure. Beam search is a generalisation of greedy search, and is commonly used as an alternative to sampling for generating output sequences from sequence-to-sequence models (Wiseman and Rush, 2016). At each step of the beam search, we start with $k$ candidate trajectories (where $k$ is known as the *beam width*). Each candidate trajectory is expanded using all possible actions, and the $k$ expanded trajectories with the highest joint probability as given by the model are selected as candidates for the next step. This procedure continues until we have $k$ complete trajectories, at which point we select the one with the lowest cost. In the routing problem, this corresponds to selecting the successful solution with the lowest total cost (if there is one).

We report results for the trained model with shaped rewards and a separate reward encoder in Table 7.1, as this model had the overall best beam search

performance. For each problem size from 3 to 9, we sampled solutions via beam search for 1280 distinct problems in five batches of size 256. The beam width was equal to the problem size for each problem.

Our results include the proportion of successful solutions, the mean optimality gap of the successful solutions, and the average time taken to return a solution. We define the optimality gap as the percentage difference between the total path length of our solution and the total path length of the optimal solution. Additionally, we report the average time taken over 100 problems to find a single solution by brute-force.

| **Size** | **Success rate** | **Mean optimality gap** | **Average beam search time (s)** | **Average brute-force time (s)** |
|---|---|---|---|---|
| 3 | 100% | 0.33% | 0.0103 | 0.00176 |
| 4 | 99.69% | 1.19% | 0.0126 | 0.0170 |
| 5 | 97.03% | 2.06% | 0.0161 | 0.0391 |
| 6 | 86.09% | 2.56% | 0.0225 | 0.0727 |
| 7 | 71.88% | 3.06% | 0.0305 | 0.145 |
| 8 | 47.42% | 3.32% | 0.0380 | 0.337 |
| 9 | 31.48% | 3.51% | 0.0458 | 1.90 |

Table 7.1: Beam search performance of the trained model with a separate reward encoder on 1280 instances of each size.

From the results, we can see that the model is successful at finding solutions for smaller problems even though the model was not trained on problems of this size. The performance on larger problems is less strong, although it is promising considering the sparsity of valid solutions for problems of this size. Considering that in all of the problems of size 9 in our empirical analysis of the circuit routing problem the successful solutions made up less than 1% of the search space, the probability that at least one of any nine randomly selected solutions would be successful is less than 0.0865. Since our agent's success rate is higher than this, we can conclude that it has made some progress towards learning to generate successful solutions.

## 7.3 Active search performance of trained agents

In this experiment, we assess the performance of the three agents when using the "active search" method proposed by Bello et al. (2016). In active search, models are trained on a wide variety of problems before being fine-tuned on an individual problem instance to find solutions.

To implement this procedure, we take the trained models from the first experiment and freeze the reward-shaping networks. For each given problem instance, we take five policy update steps on batches of 128 trajectories for that instance, with early stopping if we find a successful solution at any point. The learned shaped rewards are used to compute the policy loss for these updates. Finally, if we have not found a solution during the five update steps, we perform a beam search using the fine-tuned network.

We tested this method on 500 problems of size 9, as this is the most challenging problem size. We report the success rate, mean optimality gap and average time taken to generate a solution using this procedure for all three models in Table 7.2.

| **Model type** | **Success rate** | **Mean optimality gap** | **Average solution time (s)** |
|---|---|---|---|
| No shaped reward | 48.2% | 3.52% | 9.23 |
| Learned shaped reward, separate encoder | 43.8% | 4.00% | 9.97 |
| Learned shaped reward, shared encoder | 52.4% | 3.39% | 8.72 |

Table 7.2: Active search performance of the three models on 500 problems of size 9.

This procedure outperforms pure beam search, but is several orders of magnitude slower. Interestingly, the method with shaped rewards and a shared encoder outperforms both of the other methods. This may suggest that the reward-shaping network with a shared encoder has learnt a better reward function than the method with a separate encoder, and that this shaped reward function is useful for accelerating learning progress on a single instance. However, it is difficult to perform a direct comparison as the results also depend on the policy learnt us-

ing each method. We can not simply apply the different learned shaped rewards to one policy, as the discriminator is trained using data generated by its partner policy and is therefore coupled to it.

The average time taken to generate a solution was lower for the shared-encoder model than the model with no shaped rewards, despite the shared-encoder model requiring more computation at each step. This can be attributed to the fact that this active search procedure stops when the first solution is found, so faster learning will result in fewer gradient steps taken and no need for a beam search to be performed.

We can potentially further leverage active search by choosing a training procedure with the aim of improving its performance. *Model-agnostic meta-learning* (Finn et al., 2017) is a general-purpose model training algorithm that serves this purpose. It aims to train a model on a wide variety of tasks in a way such that a small number of gradient steps on a single task rapidly increases performance.

## 7.4 Discussion

Contrary to our expectations, learning a shaped reward did not significantly improve the training performance of the agent. This is disappointing, but not entirely surprising. Previous reward-learning methods (Finn et al., 2016a; Ho and Ermon, 2016; Fu et al., 2017) were only tested on simpler problems with less complex, better-understood model architectures, such as two-layer feedforward nets.

The learned policy with beam search was able to produce good solutions for smaller problems, though the success rate for larger problems has a lot of room for improvement. However, the ability of the policy to produce solutions two orders of magnitude faster than brute-force can still lead to speed-ups, especially for large batches of problems. One potential solution would be to use beam search as a first pass, and then brute-force if it was not successful at finding a solution.

The potential of learning shaped rewards as a meta-learning method seems promising, and is a potential future research direction.

# Chapter 8

# Conclusions

In this project, we aimed to apply reinforcement learning to a circuit routing problem to find a policy for generating solutions, as well as investigate how effective learned shaped rewards are for improving training efficiency and policy quality.

To this end, we formulated the circuit routing problem as a Markov decision process and developed a sequence-to-sequence model architecture that allows a single policy model to be trained on all problem sizes. We also produced a method based on adversarial imitation learning in order to learn shaped rewards that encourage the agent to mimic expert demonstrations.

To evaluate the effectiveness of this method, we trained two agents with different reward model architectures and one agent without any shaped rewards. Although we found little difference between the training performance of the agents, we found that the learned shaped rewards allowed an agent to learn faster when trained on individual problem instances. Additionally, the performance of all agents was good on small problems, and good enough on larger problems to offer a potential speed-up when combined with another solution method.

## 8.1 Future Work

From our analysis of the experiments, we have identified a number of possible areas for improvement and further investigation:

- Investigate how larger expert data sets affect performance of agents with learned shaped rewards.

- Investigate how alternative training algorithms (such as model-agnostic meta-learning) can improve the performance of active search.
- Explore the potential of model compression methods, such as knowledge distillation (Hinton et al., 2015), for increasing solution generation speed from a trained model.
- Investigate alternative methods for producing discriminator class probabilities from learned shaped rewards.

# Bibliography

P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 1, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138385. doi: 10.1145/1015330.1015430. URL https://doi.org/10.1145/1015330.1015430.

D. Amodei and J. Clark. Faulty reward functions in the wild. Blog, Dec. 2016. URL https://openai.com/blog/faulty-reward-functions/.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2014.

I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. URL http://arxiv.org/abs/1611.09940.

Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *CoRR*, abs/1811.06128, 2018. URL http://arxiv.org/abs/1811.06128.

L. Blondé, P. Strasser, and A. Kalousis. Lipschitzness is all you need to tame off-policy generative adversarial imitation learning. *ArXiv*, abs/2006.16785, 2020.

L. Blondé and A. Kalousis. Sample-efficient imitation learning via generative adversarial nets, 2018.

A. Boularias, J. Kober, and J. Peters. Relative entropy inverse reinforcement learning. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the*

*Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 182–189, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL http://proceedings.mlr.press/v15/boularias11a.html.

W. Dabney, Z. Kurth-Nelson, N. Uchida, C. K. Starkweather, D. Hassabis, R. Munos, and M. Botvinick. A distributional code for value in dopamine-based reinforcement learning. *Nature*, 577(7792):671–675, Jan. 2020. ISSN 1476-4687. URL https://doi.org/10.1038/s41586-019-1924-6.

H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6351–6361, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

C. Finn, P. Christiano, P. Abbeel, and S. Levine. A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models, 2016a.

C. Finn, S. Levine, and P. Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. *CoRR*, abs/1603.00448, 2016b. URL http://arxiv.org/abs/1603.00448.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1126–1135. JMLR.org, 2017.

M. Freitag and Y. Al-Onaizan. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pages 56–60, Vancouver, Aug. 2017. Association for Computational Linguistics. doi: 10.18653/v1/W17-3207. URL https://www.aclweb.org/anthology/W17-3207.

J. Fu, K. Luo, and S. Levine. Learning robust rewards with adversarial inverse reinforcement learning. *CoRR*, abs/1710.11248, 2017. URL http://arxiv.org/abs/1710.11248.

M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks, 2019.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618.

I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies, 2017.

G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network, 2015.

J. Ho and S. Ermon. Generative adversarial imitation learning. *CoRR*, abs/1606.03476, 2016. URL `http://arxiv.org/abs/1606.03476`.

R. Jena, C. Liu, and K. Sycara. Augmenting gail with bc for sample efficient imitation learning, 2020.

S. Karita, N. Chen, T. Hayashi, T. Hori, H. Inaguma, Z. Jiang, M. Someki, N. E. Y. Soplin, R. Yamamoto, X. Wang, et al. A comparative study on transformer vs rnn in speech applications. In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 449–456. IEEE, 2019.

H. J. Kim, M. I. Jordan, S. Sastry, and A. Y. Ng. Autonomous helicopter flight via reinforcement learning. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 799–806. MIT Press, 2004. URL `http://papers.nips.cc/paper/2455-autonomous-helicopter-flight-via-reinforcement-learning.pdf`.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6980`.

W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL `http://arxiv.org/abs/1602.01783`.

A. Y. Ng and S. J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, page 663–670, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1558607072.

A. Y. Ng, D. Harada, and S. J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, page 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1558606122.

A. H. Qureshi, B. Boots, and M. C. Yip. Adversarial imitation via variational inverse reinforcement learning. In *International Conference on Learning Representations*, 2019.

N. D. Ratliff, J. A. Bagnell, and M. A. Zinkevich. Maximum margin planning. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 729–736, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933832. doi: 10.1145/1143844.1143936. URL `https://doi.org/10.1145/1143844.1143936`.

J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct. 2017. ISSN 1476-4687. URL `https://doi.org/10.1038/nature24270`.

K. Smith-Miles. Neural networks for combinatorial optimization: A review of more than a decade of research. *INFORMS Journal on Computing*, 11:15–34, 02 1999. doi: 10.1287/ijoc.11.1.15.

O. Steffan. Details of the copt package. Private communication, 2020.

W. Sun, A. Venkatraman, G. J. Gordon, B. Boots, and J. A. Bagnell. Deeply aggrevated: Differentiable imitation learning for sequential prediction. In *ICML*, 2017.

W. Sun, J. A. Bagnell, and B. Boots. Truncated horizon policy search: Combining reinforcement learning & imitation learning, 2018.

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

U. Syed and R. E. Schapire. A game-theoretic approach to apprenticeship learning. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, page 1449–1456, Red Hook, NY, USA, 2007. Curran Associates Inc. ISBN 9781605603520.

J. Tromp and G. Farnebäck. Combinatorics of go. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, editors, *Computers and Games*, pages 84–99, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-75538-8.

A. Tucker, A. Gleave, and S. Russell. Inverse reinforcement learning for video games, 2018.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2017.

D. Venuto, J. Chakravorty, L. Boussioux, J. Wang, G. McCracken, and D. Precup. oirl: Robust adversarial inverse reinforcement learning with temporally extended actions.

O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/5866-pointer-networks.pdf`.

C. J. C. H. Watkins. Learning from delayed rewards. 1989.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992. ISSN 1573-0565. URL `https://doi.org/10.1007/BF00992696`.

S. Wiseman and A. M. Rush. Sequence-to-sequence learning as beam-search optimization. In *EMNLP*, 2016.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

D. Xu and M. Denil. Positive-unlabeled reward learning, 2019.

Z. Zheng, J. Oh, M. Hessel, Z. Xu, M. Kroiss, H. van Hasselt, D. Silver, and S. Singh. What can learned intrinsic rewards capture?, 2019.

B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *Proc. AAAI*, pages 1433–1438, 2008.

# Appendix A

# Code structure overview

The code produced for this project has been structured as a Python module to facilitate reuse. The module includes code to reproduce all aspects of the project, including generating training data from brute-forced solutions, training models, logging training performance and hyperparamters, and running experiments. All code is commented and formatted according to the PEP8 standard, with docstrings included for all classes and methods. PyTorch was used to compose models and enable automatic differentiation of functions throughout the project. A more up-to-date version of the module may be found at `https://github.com/joerosenberg/copt-irl`.

We give a high-level overview of the parts of the module relevant to this project below.

## A.1 Model code

- `irlco.pointer_transformer`: Specifies the policy and reward-shaping models as PyTorch modules.
- `irlco.pointer_multihead`: Modified versions of the multi-head attention modules from the PyTorch library that return attention scores rather than attention weights. This is necessary for the final layer of the decoder in the policy model.
- `irlco.routing.reward`: Methods associated with the reward model. In-

cludes a method for computing shaping terms directly from environment states using the reward-shaping net, and a method for calculating rewards from shaping terms.

- `irlco.routing.policy`: Methods associated with the policy model. Includes methods for generating action sequences using greedy search, beam search and sampling.

## A.2 Environment

- `irlco.routing.env`: Implements the circuit routing MDP as an OpenAI Gym environment. Also contains a method for calculating terminal rewards from solution measures and successes, and a method for generating problem instances with at least 30 units of clearance between nodes.

## A.3 Training

- `irlco.routing.train`: Script that trains models for the circuit routing problem. Includes configurable parameters for changing model architecture, enabling learning shaped rewards, as well as other training hyperparameters. Also includes remote logging of model configuration and training performance through the Weights  Biases API, and a method for saving and loading expert data from pickle files to decrease startup time.

## A.4 Data

- `irlco.routing.data.generate_training_data`: Methods for generating and saving brute-forced solutions to circuit routing problems. The quantity and type of data to generate can be specified in a YAML config file, which then serves as a manifest for the data after generation. Two example config files are supplied.

- `irlco.routing.data`: Implements a class for loading generated data according to a YAML config file, and allows the loaded data to be accessed in batches. In practice, instances of these classes are pickled by the training

script after loading for the first time, as parsing the data from text files takes a long time.

## A.5 Experiments

- `irlco.routing.copt_analysis`: Used to find the number of solutions and ranges of measures in Chapter 3. The scripts used to create the plots are in the `irlco/routing/copt_analysis` folder.
- `irlco.routing.beam_search_experiment`: Used to evaluate beam search performance of the trained models in the second experiment.
- `irlco.routing.active_search_experiment`: Used to evaluate active search performance in the third experiment.

# Appendix B

# Ethics Checklist

**This form must be attached to the dissertation as an appendix.**

# Department of Computer Science

**12-Point Ethics Checklist for UG and MSc Projects**

**Student** Joe Rosenberg

**Academic Year** 2019/2020

**Supervisor** Dr Özgür Şimşek

*Does your project involve people for the collection of data other than you and your supervisor(s)?* YES / NO

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Have you prepared a briefing script for volunteers?* YES / NO

   Briefing means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?* YES / NO

   All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. *Is there any intentional deception of the participants?* YES / NO

   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. *Will participants be de-briefed?* YES / NO

   The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. This phase might wait until after the study is completed where this is necessary to protect the integrity of the study.

5. *Will participants voluntarily give informed consent?* YES / NO

Participants MUST consent before taking part in the study, informed by the briefing sheet. Participants should give their consent explicitly and in a form that is persistent –e.g. signing a form or sending an email. Signed consent forms should be kept by the supervisor after the study is complete.

6. *Will the participants be exposed to any risks greater than those encountered in their normal work life (e.g., through the use of non-standard equipment)?* YES / NO

Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life.

7. *Are you offering any incentive to the participants?* YES / NO

The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

8. *Are you in a position of authority or influence over any of your participants?* YES / NO

A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. *Are any of your participants under the age of 16?* YES / NO

Parental consent is required for participants under the age of 16.

10. *Do any of your participants have an impairment that will limit Their understanding or communication?* YES / NO

Additional consent is required for participants with impairments.

11. *Will the participants be informed of your contact details?* YES / NO

All participants must be able to contact the investigator after the investigation. They should be given the details of the Supervisor as part of the debriefing.

12. *Do you have a data management plan for all recorded data?* YES / NO

All participant data (hard copy and soft copy) should be stored securely, and in anonymous form, on university servers (not the cloud). If the study is part of a larger study, there should be a data management plan.