POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Graph Neural Networks and Reinforcement Learning for Minor Embedding in Quantum Annealing

Tesi di Laurea Magistrale in
High Performance Computing Engineering

Giorgio Miani, 10685401

Advisor:
Prof. Maurizio Ferrari Dacrema
Co-advisors:
Riccardo Nembrini

Academic year:
2023-2024

Abstract: Quantum Annealers (QA) are a paradigm of Quantum Computers designed to solve combinatorial optimization problems by leveraging quantum mechanical laws, such as quantum tunnelling and entanglement. Quantum Annealers are a promising technology with the potential of solving NP-hard combinatorial optimization problems in a reasonable time, offering a relevant speed-up w.r.t. classical computers. One of the main challenges when solving a problem on a QA, is the process of mapping the problem written in the QUBO formulation to the topology of the QA. Due to the variable structure of the problem to be solved and the rigid and sparse structure of the Quantum Annealers, finding a good mapping can become computationally expensive and it can reduce the advantage obtained by using a Quantum Computer. In this thesis we develop a PPO-based Reinforcement Learning model capable of finding Minor Embeddings for QUBO problems. The model is shaped to work as a Curriculum Learning process and it is integrated with Graph Neural Networks to help the agent better understand the structure of the problem and the topology of the QA. The model is trained on a dataset composed of connected graphs with 3 to 11 nodes representing generic QUBO problems, ordered by increasing size and equally distributed according to their isomorphisms and their number of edges. The embeddings are performed on D-Wave Chimera and Zephyr topologies with up to 1152 and 2176 qubits, respectively. The goodness of the results is compared with state-of-the-art heuristic MinorMiner, since for the problems faced this algorithm's solutions are near-optimal. We show that our model is always able to find valid embeddings using on average a maximum of 21% more qubits than MinorMiner. Overall the quality of the results is good and the model developed represents an innovative and promising approach, it has the potential to serve as a starting point for using Reinforcement Learning models to find Minor Embedding for Quantum Annealers.

Key-words: Quantum Annealers, Minor Embedding, Reinforcement Learning, Graph Neural Networks

1. Introduction

Quantum Annealers [13] are a paradigm of Quantum Computers able to find in a reasonable time optimal solutions for combinatorial optimization problems that are computationally expensive for classical computers. Unlike gate-based quantum computers, Quantum Annealers leverage quantum tunnelling and superposition to

explore the solution space and find near-optimal solutions to complex problems. This approach has significant implications for various industries, including logistics [21], cryptography [22], finance [1] and machine learning [4], where optimization plays a crucial role.
Quantum Annealers, such as those developed by D-Wave, are designed to solve problems expressed in the form of a Quadratic Unconstrained Binary Optimization (QUBO) model. These problems must be mapped onto hardware using a process called **Minor Embedding**, where logical qubits are assigned to physical qubits on the quantum annealer. Due to the sparse and rigid structure of Quantum Annealers, this mapping is challenging, often leading to inefficient qubit use and inconsistent solutions, which negatively affect performance.
Finding minor embedding efficiently is essential to fully exploit Quantum Annealers' potential, especially for large-scale problems. Traditional embedding methods often can be suboptimal or computationally expensive. This thesis explores the use of **Deep Reinforcement Learning** (DRL) to optimize minor embedding, aiming to reduce chain lengths, improve solution quality, and make quantum annealing more efficient for practical applications. By representing both the problem faced and the topology of the QA as graphs, we exploit **Graph Neural Networks** (GNNs) by integrating them into the DRL framework and help the model to learn how to map the embedding based on the structures of the problem and the topology chosen. The model developed is called GReM, which stands for Graph-based Reinforcement learning for Minor embedding.
The thesis begins with an overview of quantum computing, focusing on quantum annealers and their unique architecture. It then focuses into the minor embedding problem, exploring how logical qubits are mapped to physical qubits and the associated challenges, especially in the context of QUBO problems. Following this, the thesis introduces Reinforcement Learning and explains how DRL can be applied to address the embedding problem. The role of Graph Neural Networks (GNNs) is discussed as a way to capture complex relationships in the qubit structure, further improving the embedding process.
Additionally, the structure and design of the implemented model are presented, detailing how the DRL and GNN components interact to optimize minor embedding. The thesis concludes with an analysis of the experimental results, highlighting the improvements achieved and discussing the potential implications for future research in quantum annealing.

2. Quantum Computing

In this Section we cover the basics of Quantum Annealers. We start with their introduction, followed by the definition of QUBO problems. Then we move to D-Wave devices, explaining their structures and mechanisms. We conclude with the description of the Minor Embedding process and how it can be solved.

2.1. Quantum Annealing

In metallurgy, thermal annealing is a process in which a metal is heated to a high temperature and then allowed to cool slowly. If the temperature is sufficiently high and the cooling process is gradual, the atoms in the metal rearrange themselves into a more ordered state, reaching the system's lowest energy configuration.
This concept is mirrored in simulated annealing, an algorithm used to solve combinatorial optimization problems. In simulated annealing, the system is allowed to explore different configurations by mimicking the heating and slow cooling process. As the system simulates the cooling procedure, it gradually settles into a configuration that represents a near-optimal solution, escaping local optimal by making thermal jumps over energy barriers.
Quantum annealing builds upon this idea but replaces thermal jumps with quantum tunnelling. Instead of relying on thermal energy to overcome high energy barriers, quantum annealers use tunnelling to pass through these barriers, allowing the system to reach the global minimum more efficiently. This enables quantum annealers to explore the solution space in a way that is fundamentally different from simulated annealing, potentially finding optimal solutions faster and more effectively for certain classes of problems.
The Adiabatic Quantum Computers (AQC) are a paradigm of Quantum Annealers considered to be Universal Quantum Computers. They leverage quantum tunnelling the same way as QA while guaranteeing the adiabatic condition of slow evolution. The core idea behind Adiabatic Quantum Computers is to represent the problem to be solved as the energy of a quantum system. We start by configuring the quantum annealer in an initial energy state, allowing it to evolve slowly toward its lowest energy configuration, or ground state.
Initially, the system is prepared in a superposition of all possible states, which means that the system has equal probability to be measured in any of these states. At this point, the system's ground state is easy to find. As the annealing process continues, we slowly alter the system, gradually shaping its energy landscape to match the problem we want to solve. This is done by reducing the influence (or weight) of the initial configuration until it eventually vanishes. By the end of the annealing process, the system's energy is entirely dependent on the problem at hand. If the process is slow and controlled enough, the system will remain in its ground state, which corresponds to the optimal solution of the problem.

The time dependent Hamiltonian operator describes the energy of a quantum system. For this purpose, it can be considered as formed by two terms:

$$\hat{H}(t) = A(t)\hat{H}_A + B(t)\hat{H}_B \tag{1}$$

$\hat{H}_A$ is the Hamiltonian for the initial state, $\hat{H}_B$ is the Hamiltonian for the problem to be solved and $A(t)$ and $B(t)$ are the weights. For the initial Hamiltonian it is frequently used the following formulation:

$$\hat{H}_A = -\sum_{i}^{n} \hat{\sigma}_x^{(i)} \tag{2}$$

where each term $\hat{\sigma}_x^{(i)}$ is the n-fold tensor product of $n-1$ identity operators I and the Pauli NOT operator for the i-th qubit. The identity operator, when applied to a quantum state, leaves the state unchanged. On the other hand, the Pauli NOT operator flips the value of the quantum bit, similarly to the NOT gate for classical bits. With such Hamiltonian the ground state can easily be found.
On the other hand, the final system is usually described by the Ising Hamiltonian, which is shaped in the following way:

$$\hat{H}_B = \sum_{i} h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j} \hat{\sigma}_z^{(i)} \hat{\sigma}_z^{(j)} \tag{3}$$

in which:

- h_i represents a bias on qubit i.
- $J_{i,j}$ describes the coupling strength or entanglement between qubits i and j. Two qubits are entangled when they are linked in such a way that the state of one qubit directly affects the state of the other one, regardless of the distance between them.
- $\hat{\sigma}_z^{(i)}$ is the n-fold tensor product of $n-1$ identity operators I and the Pauli (Z) operator for the i-th qubit.

This particular Hamiltonian is frequently used since it allows to describe QUBO problems

2.2. QUBO

QUBO stands for quadratic unconstrained binary optimization problem. It can be mathematically expressed as [9]:

$$\min_{x \in \{0,1\}^n} x^T Q x \tag{4}$$

where:

- x is a vector of the binary variables of the problem, $x_i \in \{0, 1\}$ for $i = 0, 1, ..., n$
- Q is a $n \times n$ matrix containing the coefficients of the objective function

The goal is to find the values for a set of binary variables that minimizes a quadratic objective function. The QUBO can be rewritten in a formulation equivalent to an Ising formulation:

$$\arg\min_{x \in \{0,1\}^n} x^T Q x = \arg\min_{x \in \{0,1\}^n} \left(\sum_i a_i x_i + \sum_{i \geq j} Q_{i,j} x_i x_j \right) \tag{5}$$

In such notation it is clear that a_i and $Q_{i,j}$ respectively represent the qubit bias h_i and the coupling strength $J_{i,j}$, while x_i corresponds to the n-fold tensor product $\hat{\sigma}_z^{(i)}$.
QUBO allows us to represent a wide number of problems. Let's look at one simple example. Consider a 2-SAT problem with five variables $x_1, ..., x_5$ that we want to solve on a Quantum Annealer. We have a set of clauses with only two variables, possibly negated and the goal is to find an assignment to the variables which satisfies all clauses. The clauses are formulated as follows:

$$\left\{ \begin{array}{l} C1 : x_1 \vee \overline{x_4} \\ C2 : \overline{x_2} \vee \overline{x_3} \\ C3 : x_3 \vee x_1 \\ C5 : \overline{x_3} \vee x_5 \end{array} \right\} \tag{6}$$

Since the variables are already binary, we only have to transform the clauses into penalty terms for the QUBO formulation. Each clause should be transformed in such a way it produces 0 when it is satisfied, and 1 otherwise. Based on the negation on the clause there are three different penalty terms:

$$\begin{array}{l} x_i \vee x_j \rightarrow 1 - x_i - x_j + x_i x_j \\ x_i \vee \overline{x_j} \rightarrow x_j - x_i x_j \\ \overline{x_i} \vee \overline{x_j} \rightarrow x_i x_j \end{array} \tag{7}$$

Following these rules, the problem faced becomes:

$$\left\{ \begin{array}{l} C1 : x_4 - x_1 x_4 \\ C2 : x_2 x_3 \\ C3 : 1 - x_1 - x_3 + x_1 x_3 \\ C5 : x_3 - x_3 x_5 \end{array} \right\} \tag{8}$$

Finally we can merge the penalty terms and obtain the following QUBO formulation for the problem considered:

$$\arg \min_{x \in \{0,1\}^5} 1 - x_1 + x_4 + x_1 x_3 - x_1 x_4 + x_2 x_3 - x_3 x_5 \tag{9}$$

And we can write the Q matrix as:

$$Q = \begin{pmatrix} -1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{10}$$

2.3. D-Wave Quantum Annelears

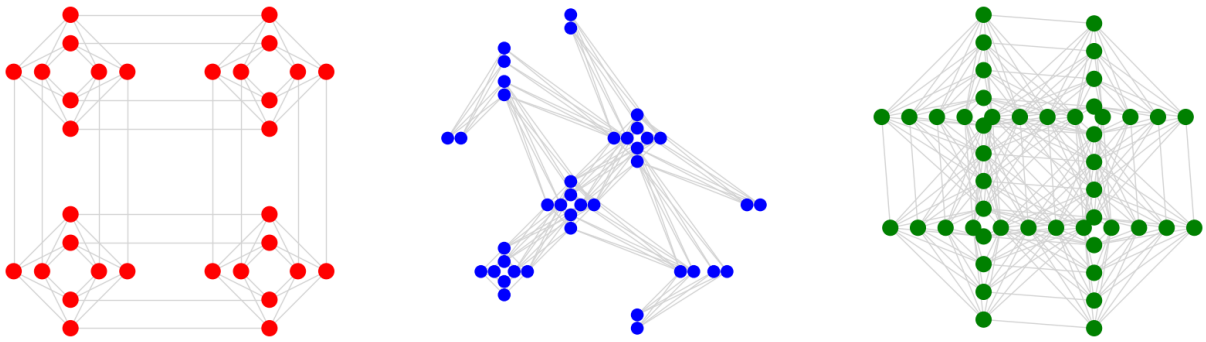
One of the leading companies in the Quantum Computing sector is D-Wave. D-Wave quantum devices are heuristic devices that leverage adiabatic evolution to minimize Ising functions by modelling them as physical systems. Since they relax the assumption of the adiabatic theorem they cannot be considered as perfectly Adiabatic QCs, instead they are referred to as Quantum Annelears.
These devices replace the classical bit with their quantum counterpart called quantum bit, or qubit. A qubit can be either in state 0, state 1 or, since it is a quantum object, it can be in a superposition of both states. In D-Wave Quantum Annelears, qubits are implemented through Superconducting Quantum-Interference Devices (SQUIDs).
SQUIDs are superconducting electronics. They are composed by a loop of superconducting material with a nano-meter long insulating layer, called Josephson junction. Electrical currents can flow through the loop with no resistance thanks to low temperatures. The direction of the flow determines the state of the qubit and magnetic fields are exploit in order to favour a certain direction with respect to the other.
Those devices can be arranged on in such a way that the magnetic field produced by one device affects its surroundings, allowing an interaction between qubits. When two qubits interact with each other they are considered entangled, so they can be identified as a single object, producing four states. The energy of each state depends on the qubits and on their interaction. It is evident how qubits can interact with each other only if they are close. Due to this physical limit, qubits can interconnect only with few qubits, corresponding to its surrounding. This leads Quantum Annealers to have a rigid structure with sparse connections between qubits. The structure depends on the physical implementation of the computer and the number of links can change.

2.3.1 Topologies

D-Wave offers multiple Quantum Annealers with three different topologies listed below:

- **Chimera**: each qubit is connected to 6 different qubits at most. Its relative simplicity is helpful to understand more complex Quantum Processing Units (QPU).
- **Pegasus**: currently commercialized topology with a maximum qubit degree of 15 [5];
- **Zephyr**: characterizes the next generation of QPUs where each qubit is connected to not more than 20 other qubits.

The structure of these Quantum Annelears can be represented through a graph, where each node is a qubit and each edge is a connection. An example is shown in Figure 1



(a) Chimera 2×2 topology. (b) Pegasus topology. (c) Zephyr topology.

Figure 1: Graph representations of examples of D-Wave Quantum Annealer topologies. Figure 1a depicts a Chimera topology with a total of 32 qubits. Figure 1b shows an example of Pegasus topology, with 40 qubits. Figure 1c illustrates a Zephyr structure with a total of 48 qubits.

When deploying a problem written in QUBO formulation, the structure of the problem has to fit in the topology of the Quantum Annealer currently in use. This process is not necessary straight forward since as mentioned above, the structure of the annealer is rigid and sparsely connected and it may not reflect well the problem. The process of mapping the QUBO formulation on the Quantum Annealer is called Minor Embedding, and it is a fundamental phase when using such devices.

2.4. Minor Embedding

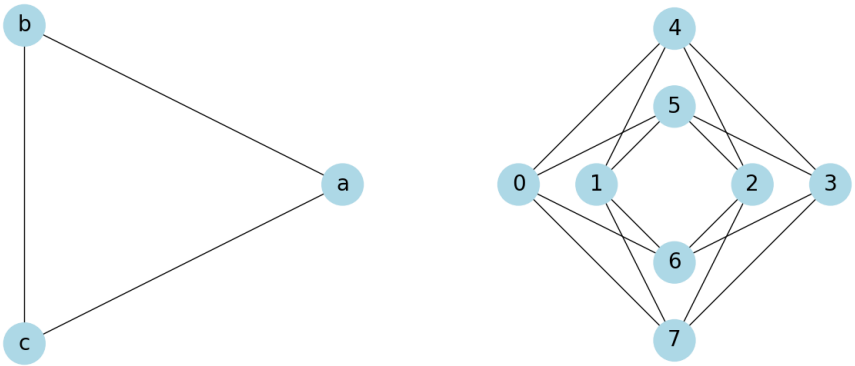
Formally, a Minor Embedding of a graph $\mathbf{G}$ with size $|\mathbf{G}|$ (representing the problem) on a graph $\mathbf{H}$ with size $|\mathbf{H}|$ (reflecting the topology of the Quantum Annealer) can be defined by a function $\phi(x) : |\mathbf{G}| \rightarrow 2^{|\mathbf{H}|}$ [6], where x is a node of $\mathbf{G}$, such that:

- for each vertex of $\mathbf{G}$, the sub-graph induced by $\phi(x)$ is connected;
- $\phi(x)$ and $\phi(y)$ are disjoint for all $x \neq y$ in $\mathbf{G}$;
- if x and y are adjacent in $\mathbf{G}$, then it exist an edge between $\phi(x)$ and $\phi(y)$ in $\mathbf{H}$.

In other words the Minor Embedding represents a mapping between the graph problem and the graph topology, but computing one is not always straightforward. In fact Minor Embedding is itself a NP-Hard problem. Let's look at an example, consider the following QUBO problem:

$$y = w_{a,b}x_a x_b + w_{a,c}x_a x_c + w_{b,c}x_b x_c \tag{11}$$

We can represent it with a triangular graph like the one shown in Figure 2a. Suppose we want to embed it on a Chimera topology with 8 qubits such as the one shown in Figure 2b.



(a) QUBO problem. (b) Square shaped topology.

Figure 2: Figure 2a shows a graph presentation of the QUBO formulation described by equation 11. Figure 2b illustrates a graph representation of a Chimera topology with 8 qubits.

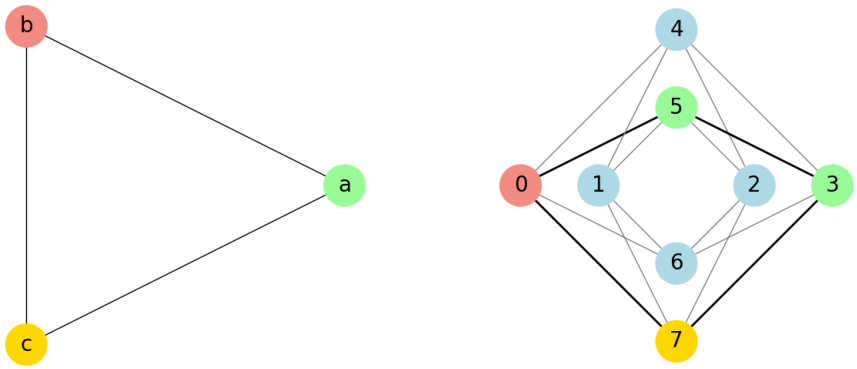
If we map x_b with qubit q_0 and x_c with qubit q_7, we have that x_b and x_c are coupled. But in order to create a connection between variables x_a and x_b and variables x_a and x_c, we have to duplicate x_a, for example, on both qubits q_3 and q_5. This creates a chain where qubits belonging to the same chain need to behave in the exact same way. When the components of a chain assume different values it breaks, leading to an inconsistent solution. The final value of a broken chain can be decided using majority voting between all the qubits of the chain, but this may still result in a sub-optimal solution.
We produce the embedding QUBO formulation:

$$y = w_{a,b}q_3x_b + w_{a,c}q_5x_c + w_{b,c}x_bx_c + \gamma_c \tag{12}$$

with γ_c being a equality constraint equal to:

$$\gamma_c = -1 + q_3 + q_5 - 2q_3q_5 \tag{13}$$

Figure 3 illustrates the Minor Embedding obtained.



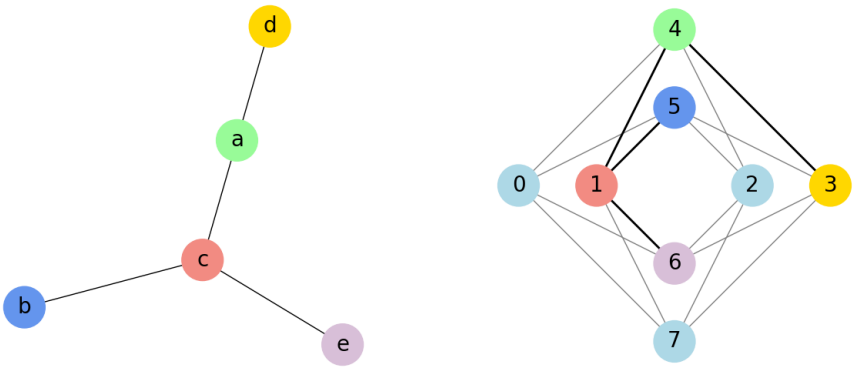
(a) Problem Graph. (b) Topology Graph.

Figure 3: Visual representation of the embedding between the triangular graph problem and the squared topology. The figure shows a connection between variable x_a and qubits q_3 and q_5, variable x_b and qubit q_0 and variable x_c and qubit q_7.

Using chains has some drawbacks:

- First of all chains embed multiple qubits for the same variable, hence they increase the number of physical qubits necessary to represent a problem. This reduces the number of logical variable that can be mapped to the Quantum Annealer and it limits the size of the problem that can be embedded to it. In the example above we are using four qubits to compute a three variable problem, hence one extra qubits is being occupied.
- Secondly longer chains have more chances to break [10]. Having more qubits behaving all the same way means there is an higher probability that a qubit assumes a different state, causing a suboptimal or even an inconsistent solution.
- Finally longer chains also have higher probability to suffer from freeze-out. When a qubits freezes it tends to not change his state anymore and high coupling strengths tend to move the freeze-out point earlier in time. This leads to qubits in a Quantum Annealer having different freeze-out points.

Hence a good embedding for a problem should have less chains as possible with equal lengths.
If we take into consideration the 2-SAT problem defined in Section 2.2 and a Chimera Quantum Annealer with 8 qubits, a possible embedding of the problem can be seen in Figure 4.



(a) Problem Graph. (b) Chimera Topology Graph.

Figure 4: Figure 4a shows a graph representation of the 2-SAT problem formulated in Section 2.2, Figure 4b is the representation of a possible embedding of the problem on a single cell of a Chimera topology. We can see that variable x_a is embedded to qubit q_4, x_b to q_5, x_c to q_1, x_d to q_3 and finally x_e to q_6.

Multiple algorithms have been developed in order to face this process, D-Wave Systems are currently using an heuristic tool called MinorMiner.

2.4.1 MinorMiner

As mentioned above, Minor Embedding process is a NP-Hard problem and exact algorithms take an exponential time to solve it. Because of this, multiple heuristics have been proposed in literature to compute the embedding in a reasonable time. They can be classified based on the level of the generalization that can be achieved, although we can distinguish three macro-categories:

- **Generic**. These algorithms are able to perform the embedding for various types of input graphs and topologies. Their main characteristic is being capable to work for a general problem, although they may not scale well for big and incomplete structures.
- **Specific**. To this group belong algorithms that work well for only a small and limited set of cases, such as specific problems or structures. Although their efficiency increases, they are restricted to the specific problems that they can cover.
- **Other algorithms**. To this final class belong all other algorithms that use different techniques and methods not necessarily heuristic.

MinorMiner is a tool developed by D-Wave Systems in 2014 and it belongs to the Generic heuristic macro-category [6]. Consider the mathematical formulation of Minor Embedding explained in subsection 2.4, we are going to call $\phi(x)$ the *vertex model* of a vertex x belonging to a graph $\mathbf{G}$. The heuristic works by iteratively constructing a vertex model for each vertex of the graph problem $\mathbf{G}$, based on the locations of its neighbours vertex-models. It starts with an initial phase where it chooses a vertex order at random and it tries to find a vertex model for each vertex. If a vertex x does not have neighbours with a vertex model, the algorithm chooses a random vertex of $\mathbf{H}$ to represent $\phi(x)$. In the opposite case, the approach attempts to find a $\phi(x)$ close to the vertex models of its neighbours in $\mathbf{H}$. Suppose that x is adjacent to nodes $y_1, ..., y_k$ which already have vertex models $\phi(y_1), ..., \phi(y_k)$. The heuristic calculates the shortest path from $\phi(y_j)$ to every node h in the sub-graph of $\mathbf{H}$ of unused vertices. It stores this information as a cost variable $c(h, j)$ and it chooses the h^* with the smallest sum of the shortest paths:

$$argmin_h = \sum_{j=1}^{k} c(h, j) \tag{14}$$

This vertex h^* is considered the root of the vertex model of x and $\phi(x)$ is constructed by the union of the shortest paths from h^* to each $\phi(y_j)$. If such vertex in the sub-graph of unused nodes in $\mathbf{H}$ does not exist, the heuristic allows vertex models to overlap. Once the initial stage is concluded, the procedure refines the vertex models found such that no vertex of $\mathbf{H}$ represents more than one vertex of $\mathbf{G}$. The heuristic accomplishes this by iteratively removing a vertex model and inserting a better one, while trying to reduce the overlapping.
This heuristic was proven to be effective when the graph of the QUBO problem and the topology graph are sparse with a maximum of few hundreds nodes [6]. The execution time mostly depends on the calculation of the shortest paths, in the worst case it is $O(|G||H|E_G(E_H + |H|log|H|))$, where $|G|, |H|$ are the sizes and E_G, E_H

are the number of edges of respectively **G** and **H** . However, in practice the number of iterations of the heuristic usually are less than $|G||H|$.

3. Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a paradigm of Machine Learning which combines Reinforcement Learning with deep neural networks to enable agents to make decisions in complex environments. In DRL, an agent learns to take actions that maximize a reward by interacting with the environment and using deep learning to approximate optimal policies.
In this Section we first explore the key components of DRL and we explain their purposes, then we focus on the main existing approaches and we finally describe a particular paradigm of Reinforcement Learning called Curriculum Learning.

3.1. Main components

The agent is an entity that receives information from the environment through what is known as observations. When the agent receives an observation, it uses this information to choose an action according to its policy, which is a function that maps observations to actions.
As the agent performs an action, it interacts with the environment, resulting in modifications to that environment. This interaction triggers a change that leads to a new observation. The entire process of observing the environment and executing an action is referred to as a step. Specifically, given the observation of the state S_t at step t, the agent selects an action A_t and receives a reward R_t from the environment. The reward serves as feedback, guiding the agent in refining its future decisions. Figure 5 shows a simple representation of the process described.

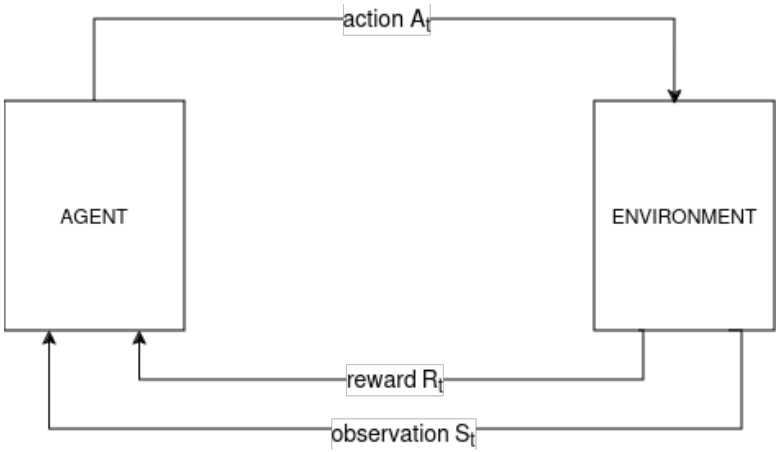


Figure 5: Schematic representation of the DRL process. The Agent receives from the environment a reward R_t and an observation of the current state S_t and computes these values. Then it performs an action A_t on the environment modifying its state.

The cycle of observation, action, and feedback occurs continuously, enabling the agent to learn and adapt its behaviour over time. An episode consists of a sequence of these steps, beginning from an initial state of the environment and continuing until specific stopping conditions are met.
As the agent navigates through episodes, it gathers experiences that inform its policy adjustments.

3.1.1 Policy

The policy is a key part of the DRL model. It can be defined as a mapping between the current state of the environment and the actions to be taken. It encapsulates the way that the agent behaves in a given step and it is the core of the agent since it alone defines how it behaves. Policies can be deterministic, like a look-up table, or stochastic, like computing the probability of selecting an action given a certain state.

3.1.2 Reward signal

The reward defines the goal of a Reinforcement Learning problem. After each action the environment sends to the agent a value called reward and the objective of the model is to maximize the total reward. It can be both positive when it is needed to highlight good events, or negative when there is the necessity to penalize bad actions. The reward signal serves as the principal mechanism for modifying the policy, since a low reward following a certain action can change the policy leading to a different choice in the future.

3.1.3 Value function

While the reward defines the goodness of an action in the immediate time, the value function specifies what is good in the long run. The value of a certain state is the sum of the rewards that we expect to receive starting from that state. In other words it is a prediction of the future rewards, and it is a key part when making and evaluating a decision. Action choices are made based on the value judgement, and the agent purpose is to choose an action that leads to a state with the highest value. This is due to the fact that a state may have a low reward, but it can lead to more promising states, resulting in a higher value. Nevertheless value functions are secondary to rewards, since they are calculated starting from it and the only goal of estimating values is to receive higher rewards. Contrary to rewards, values are not given by the environment, but they must be estimated from the observations encountered. A value function can be associated both to states or state-action pairs.
A state-value function $V^\pi(s)$ starting from a state s and given a policy π at a certain time-step t can be defined as:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s \right] \tag{15}$$

where:

- R_t is the reward at time-step t.
- $\gamma \in [0, 1]$ is the discount factor that determines how future rewards are weighted.

On the other hand in an action-value function $Q^\pi(s, a)$ we also take into account the action a taken by the agent at time-step t:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a \right] \tag{16}$$

3.1.4 Model

The model of the environment is a structure used to make inference on how the environment may behave. They are used to plan the next action by considering possible future situations. Methods that use models are called model-based methods, opposite to model-free methods [19].

3.2. Methods

Different Reinforcement Learning algorithms exist based on how the agent learns. In the following sections we define the different macro-categories in which they can be divided.

3.2.1 Value-based

Value-based methods focus on learning the value functions associated with states or state-action pairs. This process aims to find a policy indirectly by learning value functions, which are used by the agent to make decisions. These methods use Bellman equations to iteratively update value estimates. For simplicity we consider only state-value functions, in which the equation is defined as follows:

$$V^\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s \right] \tag{17}$$

3.2.2 Policy-based

Opposed to value-based methods there are policy-based ones, where the policy is optimized without the use of a value function. In other words these methods try to find the optimal policy by maximizing the cumulative reward, instead of estimating the value function. The key component is a parametrized policy, defined as:

$$\pi_\theta(a \mid s) \tag{18}$$

where a is an action that can be performed on state s. The goal is to maximize the expected cumulative reward J with respect to a parameter θ:

$$J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \right] \tag{19}$$

where γ is the discount factor. The policy gradient theorem is used to calculate the gradient of J:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(a \mid s) Q^\pi(s, a) \right] \tag{20}$$

and exploit it to update the parameters θ in the direction that maximizes J.

3.2.3 Actor-Critic

Actor-critic methods combine both value-based and policy-based methods. They are composed by an actor and a critic, each one with different roles:

- The actor represents the policy. It decides the next action based on the current state and its goal is to update the policy using a policy-based approach.
- The critic estimates the state or action-state pair value. His goal is to calculate the value function to help the actor.

Another fundamental component is the advantage function $A(s, a)$ defined as:

$$A(s, a) = Q(s, a) - V(s) \tag{21}$$

which explains the goodness of an action a in a state s compared to the average action in that state. Different actor-critic methods exist, but we will focus only on the Proximal Policy Optimization (PPO) algorithm.

3.2.4 PPO

PPO is an advanced Reinforcement Learning algorithm based on actor-critic methods [18]. The first step in this algorithm is to collect sequences of action-state-reward pairs, called trajectories. Once it has collected a batch of trajectories, it calculates the advantage function $A(s, a)$ defined above. To estimate it PPO uses the Generalized Advantage Estimation or GAE:

$$A(s_t, a_t) \approx r + \gamma V(s_{t+1}) - V(s_t) \tag{22}$$

Then the algorithm updates the policy using the clipped surrogate objective function. First of all it calculates the probability ratio between the old policy and the new policy

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\text{old}}(a_t \mid s_t)} \tag{23}$$

Then it updates the policy by maximizing the clipped surrogate function $L_{CLIP}(\theta)$ defined as:

$$L_{\text{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta) A(s_t, a_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A(s_t, a_t) \right) \right] \tag{24}$$

where $r_t(\theta)$ is the ratio defined in equation 23 and $A(s_t, a_t)$ is the advantage function described in equation 22. The function $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$ restricts the ratio to keep it in the range $[1-\epsilon, 1+\epsilon]$. The function uses the minimum between the unclipped term $r_t(\theta)A(s_t, a_t)$ and the clipped term, such that if $r_t(\theta)$ falls out of the range $[1-\epsilon, 1+\epsilon]$, the clipped term is used. This design penalizes the updates of the policy that make large changes, preventing large policy shifts. The objective aims to increase the probability of choosing an action that produces a positive advantage, with respect to actions resulting in a negative one. It is an upgrade with respect to the surrogate function since it introduces a clip term depending on a small hyper-parameter ϵ with the purpose of controlling the maximum change permitted in the policy. So if the ratio $r_t(\theta)$ is close to one the update proceeds normally, while if the policy changes too much, the clipped function on the ratio limits how much it can vary. This procedure is the core of PPO since it guarantees stability by controlling how much a policy can differ from the previous one in a single update. Finally there is the critic update phase, where the value function is updated using classic value-based methods. Typically the algorithm minimizes the mean squared error between the predicted value $V(s_t)$ and the actual return:

$$L_{\text{critic}}(\theta_V) = \mathbb{E}\left[(V(s_t) - R_t)^2 \right] \tag{25}$$

The whole process is repeated iteratively until convergence is reached.

PPO brings multiple benefits with respect to other Actor-Critic methods, including:

- stability, thanks to the clipped function mentioned before.
- data efficiency, due to the reuse of the same batches of trajectories.
- simplicity, relatively simple to implement.

3.3. Curriculum Learning

Curriculum learning (CL) is a particular paradigm of RL where the problem is presented to the model with increasing difficulty, starting from the simplest case and increasing the complexity progressively. Curriculum Learning is inspired by how humans and animals learn, starting from an easy example and gradually introducing more complexity. The goal of CL is to reach convergence faster and in a more generalized way. It has been proven that CL improves those aspects when facing shape recognition and language modelling problems [3].

4. Graph Neural Networks

Graph Neural Networks (GNNs) are a particular class of Artificial Neural Networks capable of handling data structured as graphs [17]. They can be considered as a transformation on all the attributes of a graph that does not change the graph symmetry and structure [7, 16]. In this Section we will start with a brief introduction to graphs, followed by the description of GNN mechanisms.

4.1. Graphs

Graphs are mathematical structures able to represent various systems thanks to their flexibility. They are composed by nodes (or vertices) and edges (or links) that connect the nodes. The nodes represent the objects or entities of the graph, while the edges are the relationships between the entities. As explained in Section 2, we can both represent the QUBO formulation and the topology of a Quantum Annealer as graphs. In the former case nodes are the variables of the problem and the quadratic terms are the relationships, in the latter case nodes represent qubits and an edge is a possible interaction between those qubits. More formally, a graph A can be defined as a pair (V, E) where:

- V is a non-empty set of nodes.
- E is a set of ordered or unordered pairs of nodes, called edges.

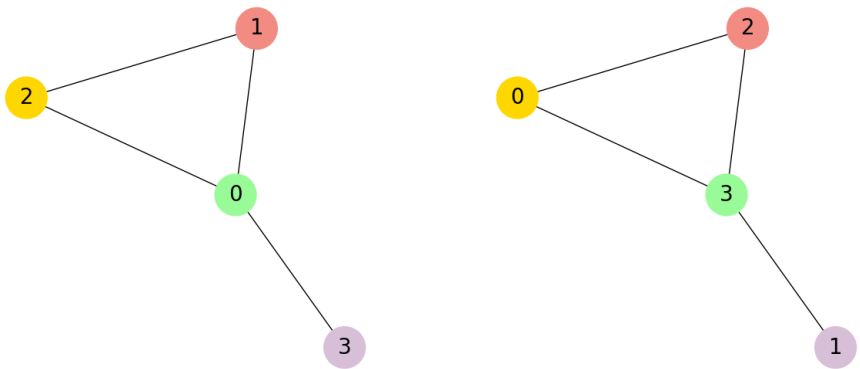
$|A|$ represents the sizes. We will focus on connected and undirected graphs, where there is always a path between two nodes and if an edge $(u, v) \in E$, then also $(v, u) \in E$. We will also deal with complete graphs, which means that for each node $u, v \in V$ it exists an undirected edge $(u, v) \in E$. Finally, it is important to mention a property of graphs called degree of a node, which is defined as the number of edges incident in that node.

4.1.1 Isomorphism

An isomorphism between two graphs A and C is a bijective function between the vertex set of A and C:

$$f : |A| \to |C| \tag{26}$$

such that any two vertices u and v are adjacent in A if and only if $f(u)$ and $f(v)$ are adjacent in C. If an isomorphism exists between two graphs, they are called isomorphic and a set of isomorphic graphs is called isomorphism class [2]. In Figure 6 is shown an example of isomorphic graphs.



(a) Graph. (b) Isomorphic graph.

Figure 6: Example of isomorphic graphs: the coloured nodes reflect the same structure, even though they have different labels.

The computational problem of determining if two graphs are isomorphic is called Graph Isomorphism problem. The problem is not known to be solvable in polynomial time nor to be NP-complete, and therefore it may be in a computational complexity class between these two [20].

4.2. Graph Tasks

Information can be stored on graphs in multiple ways: we can have node level attributes, edge level attributes or even global information. How the attributes are stored also depends on the task being performed on the graph [16]. There are:

- Graph-level tasks, the goal is to predict a property of the entire graph or to assign a label to it. As an example we may want to classify whether a molecule will bind to a receptor implicated in a disease or not.
- Node-level tasks, the aim is to predict the label of each node of a graph. For instance we can have two groups and try to predict each node to which one it belongs.
- Edge-level tasks, the scope is to predict relationships between nodes of a graph. For example we may have nodes representing entities in an image and we may want to classify each edge based on the relationship between the entities in the frame.

4.3. Message Passing

Message passing is the core of Graph Neural Networks and it consist in exchanging information between neighbour nodes and edges to influence each other's updated embeddings. The message passing process can be formally divided in three phases [8, 16].
Consider a specific time-step t, first the message $m_{uv}^{(t)}$ from node u to node v is computed using a message function M_t. The message is computed based on the hidden states $h_v^{(t)}$ and $h_u^{(t)}$ containing the features of nodes u and v and if possible the edge features e_{vw} associated to the link connecting u and v.

$$m_{uv}^{(t)} = M_t(h_u^{(t)}, h_v^{(t)}, e_{uv}) \tag{27}$$

Then each node aggregates the messages computed by its neighbours $\mathcal{N}(v)$ using an aggregation function A_t:

$$m_v(t) = A_t\left(\{m_{uv}(t) \mid u \in \mathcal{N}(v)\}\right) \tag{28}$$

Common choices for the aggregation are sum, max and mean functions.
Finally after receiving and aggregating the messages, each node updates its own hidden state h_v by applying an update function U_t:

$$h_v(t+1) = U_t\left(h_v(t), m_v(t)\right) \tag{29}$$

By stacking multiple message passing layers a node can gather information coming from across the entire graph.

4.3.1 GCN

Various message passing layers exist depending on the message, aggregation and update function chosen. Our main focus will be on the Graph Convolution Networks (GCN) [14]. Consider $X^{(l)}$ the matrix of node feature vectors $X_i^{(l)}$, the GCN layer can be written in matrix form in the following way:

$$X^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}X^{(l)}W^{(l)}\right) \tag{30}$$

where:

- $\tilde{A} = A + I$ s the adjacency matrix of the computed graph with added self-connections.
- $\tilde{D}$ denotes the degree matrix, defined as a diagonal matrix with $D_{ii} = \sum_j A_{i,j}$.
- $W^{(l)}$ is a layer-specific trainable weight matrix.
- $\sigma()$ represents an activation function.

4.4. Pooling

In some cases it is necessary to collect information coming from all nodes or edges. In this process it is very useful to have a pooling layer. When pooling we are completing two steps:

- First for each item to be pooled we gather and collect each of their information;

- Then we perform an aggregating operation.

Depending on how the first step is performed we can have three types of pooling:

- **Global pooling**: this is the simplest approach where all information are gathered in a single vector.
- **Hierarchical Pooling**: this involves progressively reducing the graph size while preserving important structural information.
- **Attention-based Pooling**: Uses attention mechanisms to decide which nodes are more important and should be aggregated. The idea is to weight node contributions based on their significance in the task.

We will focus only on global pooling. Common operation for this layer include: mean, sum and max pooling

5. GReM Design

In this Section we describe the structure and the functioning of GReM, our *"Graph-based Reinforcement learning for Minor embedding"* model, focusing also on the developed training and testing processes and the dataset construction. The agent is trained using the Proximal Policy Optimization (PPO) algorithm, as detailed in Section 3.2.4, with two variations: one that incorporates the Curriculum Learning process, mentioned in Section 3.3 and one without it. The main components of the model are the performable actions, the state of the environment, the reward and the agent. For training and testing phases we used a dataset composed of a set of graphs with different sizes, designed to help the model in the learning process.

5.1. Action

The agent interacts with the environment by observing its state and choosing an action accordingly. At each step a node of **G** is selected in a cyclic order using a round robin scheduling. An action taken by the model consists of mapping the selected node of **G** to an available node in **H**. Therefore, the set of possible actions corresponds to the probability for each node of **H** to be selected for the embedding. Invalid Action Masking (IAM) [12] prevents the agent from choosing qubits which are already mapped to a variable by setting to zero the probability of choosing them.

5.2. State

The state of the environment is composed of a combination of two graphs: the problem graph, denoted as **G**, and the topology graph, denoted as **H**. When performing training, **G** is sampled from a dataset containing multiple graphs with varying sizes, ranging from 3 to a maximum size defined by an hyper-parameter, while **H** is fixed, with both its size and topology determined by hyper-parameters. Nodes in both graphs contain specific features. Each node in **G** has three features:

- **Degree**: Represents the degree of the node.
- **Chain**: Defined as a vector with a length equal to the size **H**; each element is **1** if the node in **G** is mapped to the corresponding node in **H**, and **0** otherwise.
- **Round robin**: A scalar value set to **0** if the corresponding node of **G** is the selected one to be mapped in the current step, **1** otherwise.

Each node in **H**, on the other hand, has a single feature corresponding to its degree.

5.3. Agent

The agent is composed of an actor and a critic, both with similar structures. Each consists of a Graph Neural Network divided into two towers: one processing **G** and the other **H**. The **G** tower contains a single **GCN** layer followed by a **global mean pooling** layer. The **H** tower includes three **GCN** layers, whose outputs are concatenated and passed through a **global mean pooling** layer [15]. The outputs of both towers are then concatenated and passed through the deep learning network, which consists of three dense layers with two *tanh* activation functions.

The critic outputs a single value corresponding to the value of the state, as discussed in Section 3.2.3, while the actor produces a number of values equal to the number of qubits, representing the probabilities for each qubit to be selected. IAM sets to zero the probability for qubits already mapped to a variable.

Figure 7 shows a simple visualization of the structure of the agent.

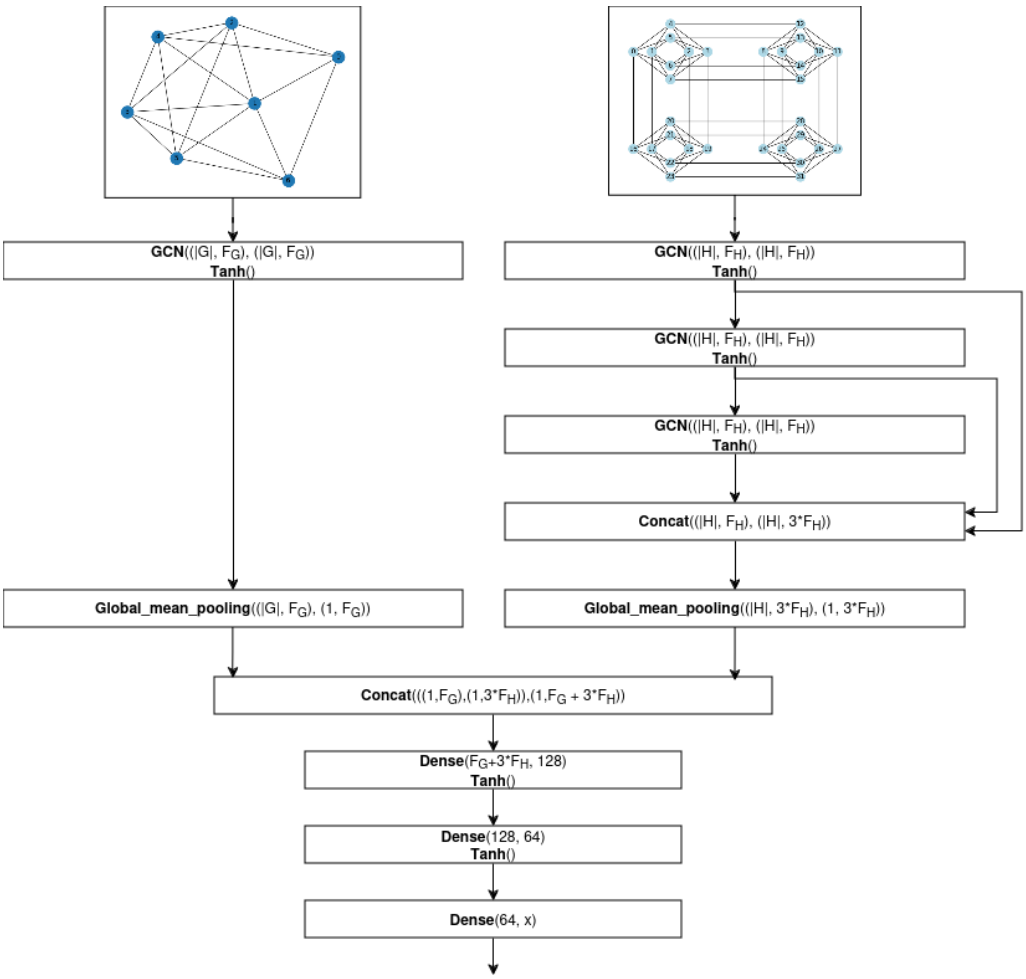


Figure 7: Schematic representation of the Agent adopted. In this illustration, $|G|$ and $|H|$ represent the sizes of **G** and **H**, while F_G and F_H are the length of the node features. The output dimension of the last dense layer depends on whether the agent is an actor or a critic. The former computes a vector of length equal to the number of actions, while the latter outputs a single value. For simplicity, a single step has been considered, while the agent accepts batches of steps.

5.4. Reward

The reward function currently in use assigns a **negative reward** of -0.1 for each action taken. If all qubits in **H** are unavailable, but the embedding found is not valid, the reward is set to **-1**. This penalization encourages the model to minimize the number of actions required to create the embedding, effectively learning to find an embedding while using as few qubits as possible.

5.5. Dataset

The dataset is built considering the following properties:

- **connected**: all graphs of the dataset are connected.
- **ordered**: the graphs are grouped by their size in ascending order, starting from 3 nodes to a maximum size.
- **balanced**: all sizes have almost the same number of graphs.
- **distributed**: the graphs are generated in order to equally distribute the number of edges of the graphs and the isomorphism classes.
- **split**: the dataset is divided into training and testing based on the testing percentage. The properties mentioned above apply to each group separately.

The graph construction process varies depending on the size, as larger graphs require faster and more approximate algorithms.

For graphs with 3 to 5 nodes, the dataset includes all possible combinations of connected graphs along with all

their isomorphisms. First, all possible graphs are created and grouped based on edge count and isomorphism. Since the number of graphs is relatively small, the dataset is augmented by duplicating graphs within each group until the target number is reached.
For graphs with 6 to 8 nodes, a different process is applied, as creating all possible graphs and isomorphisms is infeasible. Instead, we generate all possible isomorphism classes and randomly select graphs from each class until reaching the desired count. For each node count, the process begins with the complete graph. An edge is removed from it, the number of edges is recorded, and we check if a graph isomorphic to the one obtained already exists. If not, the graph is saved. This process is repeated for each edge of the complete graph and subsequently for the newly created graphs, resulting in the collection of all possible isomorphism classes sorted by edge count. If the number of isomorphism classes obtained is lower than the target number of graphs, one or multiple isomorphisms are generated from each class until the total number is reached. Instead, if the classes exceeds the number of graphs desired, then a class is selected for each number of edges and a single graph is generated from it until the process is completed.
For graphs with more than 8 nodes, the above procedures are not feasible due to the high computational cost of checking graph isomorphisms. Instead, we use a different technique, where isomorphism classes are not individually created. A connected graph with n nodes can have between $n-1$ and $\frac{n(n-1)}{2}$ edges. For each edge count, a graph is created, it is checked if an isomorphism of it is already stored, and if not it is saved. This process continues until the required number of graphs is obtained.
For each size, the graphs collected are split based on the testing percentage to create the testing and training datasets.
Figure 9 illustrates the distribution of the isomorphism classes for their number of edges grouped by their size.

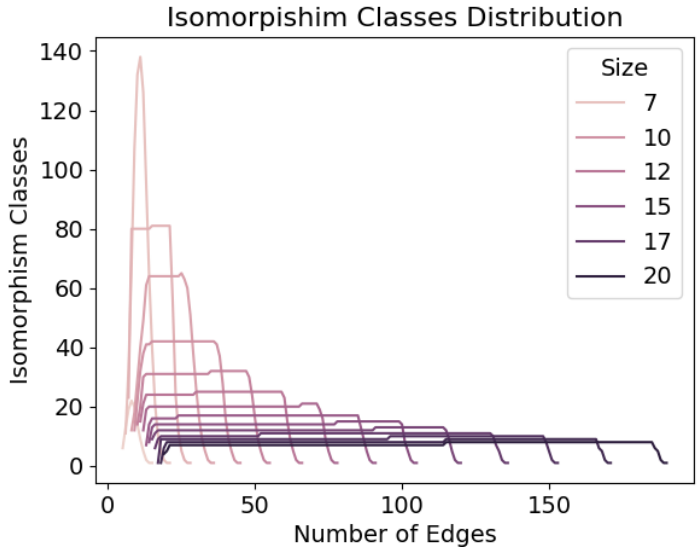


Figure 8: Distribution of the number of isomorphism classes for the number of edges grouped by the graph size.

5.6. Training

To train our model, we utilize the PPO algorithm implemented by *CleanRL* [11]. We developed two versions of it, one modified to function as a Curriculum Learning process and it other one without CL. Most of the hyper-parameters used are set using the default ones of *CleanRL*, but additional parameters have been introduced for CL.
For a fixed number of steps the model repeats the following procedure: the agent receives an observation containing the features of $\mathbf{G}$ and $\mathbf{H}$, processes it, and produces a probability distribution used to sample the next action. The model performs an action by mapping the current node of $\mathbf{G}$ to a node of $\mathbf{H}$ and receives a reward and a new observation, which is processed in the next iteration.
After collecting a batch of steps, the model enters the roll-out phase where the agent's weights are updated. This phase depends on the batch size of the data used to update the policy and the number of epochs used during the update.
Once the roll-out phase is completed, the model repeats the whole process. While training, the model faces an increasing level of difficulty of the problems. It start by embedding basic graphs with 3 nodes and the graphs

size increases until it reaches the maximum size of the problem graphs. The condition that has to be satisfied in order to switch to the next size depends on the version implemented.
In the Curriculum Learning version the process continues to embed graphs with a fixed size until a convergence criterion is satisfied. Upon convergence, training proceeds to the next size and performs the computation again. Training stops when the model reaches convergence for graphs with the maximum size.
Convergence is based on the quota of **G** graphs embedded and a threshold for this quota is used to determine the model's convergence. This quota is the number of problem graphs embedded by the model for a **G** size. During training, the learning rate (lr) decreases linearly and it becomes constant when it reaches a fixed lower bound. At each step performed by the agent, the learning rate is updated as shown in equation 31

$$\text{new lr} = \text{initial lr} * \text{fraction} \tag{31}$$

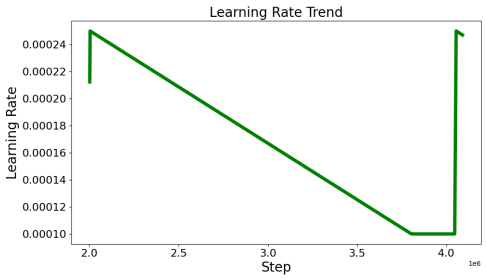
where fraction is defined in equation 32

$$\text{fraction} = 1 - \frac{\text{local step}}{\text{total steps}} \tag{32}$$

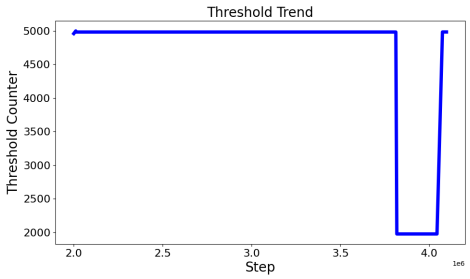
with local step defined as the number of steps performed for the specific **G** size, and total steps defined as the total number of steps set for each size as a parameter at the beginning of training. Additionally, training remembers the reward for the smallest embedding found for each **G** size, allowing the model to reset its counter when an embedding with an higher reward is discovered. When this happens the threshold is decreased linearly using the fraction defined for the learning rate in equation 32:

$$\text{new threshold} = \text{initial threshold} * \text{fraction} \tag{33}$$

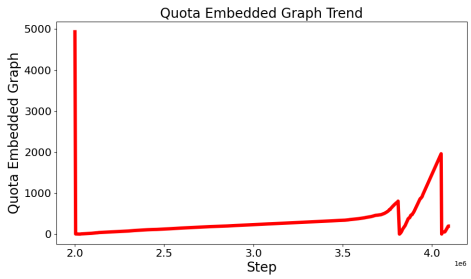
Finally, when the environment reaches the defined threshold, the model advances to the next graph size and it resets the learning rate and the thresholds to their initial values, while also decreasing the counter to zero.
Figures 9 show an example for the trend of the learning rate, the threshold and the quota of embedded graphs during training of the model. The Figures depicts few steps over the total number, when the model is embedding graphs with 9 nodes.



(a) Learning Rate Trend.



(b) Threshold Trend.



(c) Quota Trend.

Figure 9: Figures show an example of trends of the learning rate, threshold and the quota of embedded graphs obtained during training of the model. The lr starts from a default value of 0.00024, decreases linearly and remains constant at 0.0001. At around step 3.7M the quota resets and the threshold decreases from the initial value of 5000 to 2000 due to a better embedding found by the model. At around step 4M the counter reaches the threshold and all the variables are set to their initial values.

Algorithm 1 shows a synthetic description of training process using Curriculum Learning.

Algorithm 1 PPO Algorithm with Curriculum Learning

```
Initialize Agent
Initialize Environments
while done do
  if Quota of embedded graphs > threshold then
    if Graph size is max then
      Terminate training
    else
      Move to next graph size
      Reset lr, threshold, counter, quota and maximum reward
    end if
  end if
  for t = 1, ..., total steps do
    Agent reads observations S_t
    Agent performs action A_t on each environment by embedding a qubit
    Environments return new observation S_{t+1}
    if Graph is embedded then
      if Reward embedding > maximum reward then
        Reset threshold and counter
      else
        Increase quota
      end if
      Move to next graph
    end if
  end for
  for epoch = 1, ..., total epochs do
    Update agent weights
  end for
end while
```

Moreover, we noticed that for each graph size, reintroducing a small percentage of previously trained graphs helps to maintain the model's proficiency on smaller graphs even as it advances to larger ones. The percentage of smaller graphs reintroduced is fixed at the beginning of training.
Instead, in the basic implementation without CL, a fixed number of graphs for each size is selected and when the agent has computed an embedding for all graphs with a specific size, it moves to the next one. When the model has processed the whole dataset, the process repeats from the minimum size, with no reset of training phase. This continues until the model has exhausted a given number of training steps. No condition is used to determine if the model has reached convergence.

5.7. Testing

During the testing phase, the trained model is loaded to compute embeddings for the testing graphs in the dataset. The testing dataset is fixed for all experiments. The testing graphs have the same sizes of the ones in the training set, but are never seen by the model during training. Moreover, the graphs are equally distributed according to their number of edges and their connectivity, similarly to the training set. For each graph the model produces multiple embeddings, controls if they are valid and calculates the minimum number of qubits used between the embeddings and the time spent to find each embedding. Then it compares the number of qubits and the time used for each instance with the results produced by the state-of-the-art heuristic, MinorMiner.

6. Results

In this Section we are going to show and analyse the results obtained from the tests discussed in Section 5.7. First we are going to look at the results for the complete model described in Section 5. Then we are going to discuss the key components of the model together with the motivation and the results that lead to using such components. All results shown, except for Section 6.3.1, are obtained by collecting 10 experiments using

different seeds. Moreover, the model and MinorMiner are set to find 10 embeddings for each problem graph in the testing dataset. The results of MinorMiner shown are obtained considering the smallest embedding found for each problem graph, while for our model it depends on the metric considered.

6.1. Metrics

The evaluation of the quality of the results for the different versions of the model is based on three main criteria:

Success Rate (SR) The success rate is determined by the number of problem graphs in the testing dataset successfully embedded by the model. A $\mathbf{G}$ graph is successfully embedded if all of the 10 embeddings produced for that graph are valid.

Qubit Utilization Ratio (QUR) Provided the success rate is sufficiently high, we compare the number of qubits used by our model with those used by MinorMiner, employing the following ratio:

$$\text{Qubit Utilization Ratio} = \frac{|E_{\text{RL}}| - |E_{\text{MM}}|}{|E_{\text{RL}}|} \tag{34}$$

where $|E_{\text{RL}}|$ and $|E_{\text{MM}}|$ represent the number of qubits in the smallest embedding produced for a single problem graph in the test set by our model and MinorMiner, respectively. Considering the size of the problems used to test our model, MinorMiner is highly efficient at finding good embeddings using a minimum number of qubits. This makes it an almost optimal reference point for comparing our results. An efficient embedding minimizes the use of qubits. Therefore, the closer $|E_{\text{RL}}|$ is to $|E_{\text{MM}}|$, the better the embedding quality, moving the ratio closer to zero. Conversely, excessive qubit usage by the model will increase $|E_{\text{RL}}|$ w.r.t. $|E_{\text{MM}}|$, pushing the ratio towards 1.

Execution Time Ratio (ETR) Lastly, we examine the time required by the model to compute an embedding during testing. While this is not the primary focus, it is essential that the embedding process remains efficient enough to preserve the computational advantages of the Quantum Annealer. We compare the results with MinorMiner performance by computing the Execution Time Ratio as:

$$\text{Execution Time Ratio} = \frac{T_{RL}}{T_{MM}} \tag{35}$$

where T_{RL} is the time taken by the model to find the smallest embedding for a problem graph in the test set, except for the first graph of the dataset where the minimum time was taken . On the other hand, T_{MM} is the minimum time taken by Minor Miner across all 10 embeddings found for the same problem graph. After detailing these metrics, we proceed to evaluate the quality of the results obtained by the model, as described in Section 5.

6.2. GReM

The primary objective is to develop a model capable of finding valid embeddings for a general problem on a specific Quantum Annealer. In addition to ensuring embedding validity, we aim to minimize the number of qubits required and optimize the model's computational efficiency.
For the following experiments we have used default values for the *CleanRL* parameters, except for the number of epochs used to update the weights set to 10. The number of steps considered for each $\mathbf{G}$ size is $3M$, the threshold is set to $5K$ graphs and the learning rate lower bound is $1.0e^{-4}$. For the dataset we have chosen to use approximatively 1250 graphs per size. For each size around 80% of the graphs are used for training only, which corresponds to 1000 graphs, and the remaining one for testing. Notice that the number of graphs for each size used during testing is not necessarily 250, since the graphs duplicated to balance the dataset are not considered during testing. For these experiments, for each $\mathbf{G}$ size, we chose to reintroduce during training 500 smaller.
All results were collected from 10 experiments conducted on graphs with node counts ranging from 3 to 11, embedded within a Zephyr topology of 2176 qubits. For each of these experiments we train the same model using different seeds.
Figure 10 shows that the model achieved a Success Rate of **100%** across all experiments. Figure 11 depicts the Qubit Utilization Ratio obtained by the model.

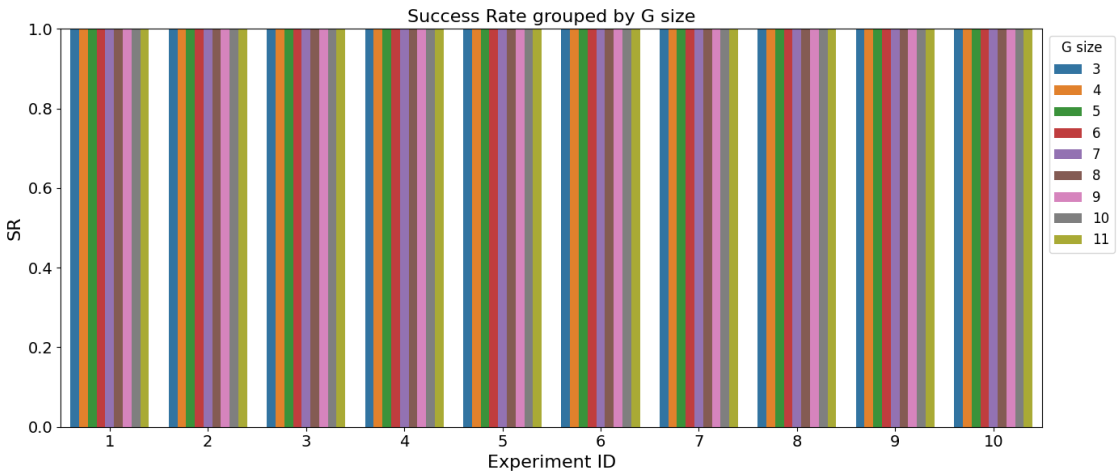


Figure 10: Success Rate as defined in Section 6.1 across 10 experiments. The X-axis represents the experiment ID, and the Y-axis shows the ratio. Colours indicate the size of the embedded G graphs.

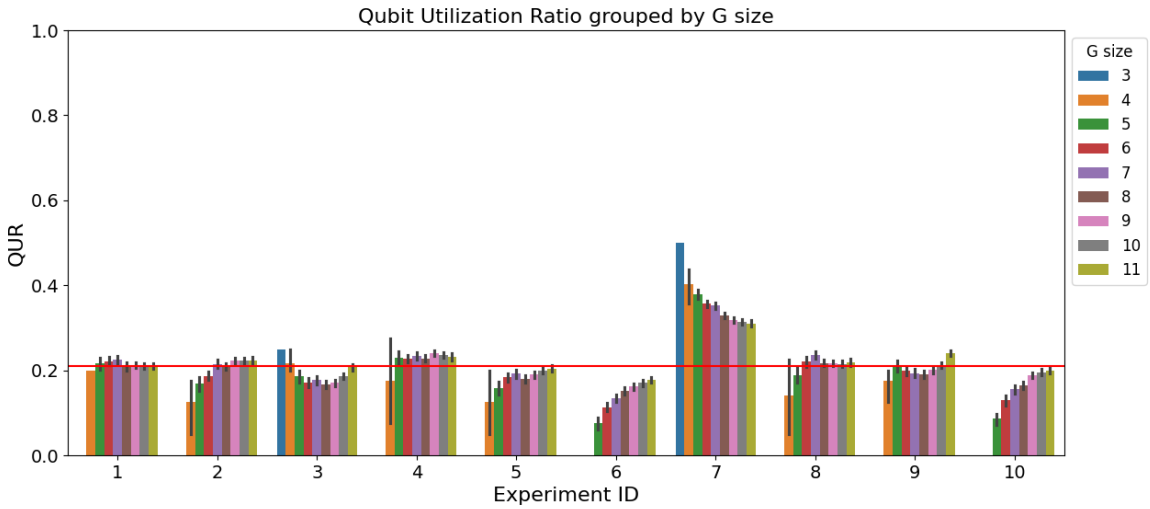


Figure 11: Qubit Utilization Ratio as defined in Section 6.1 across 10 experiments. The X-axis represents the experiment ID, and the Y-axis shows the ratio. Colours indicate the size of the embedded G graphs.

The results demonstrate that GReM achieves embeddings comparable to those generated by MinorMiner. The average QUR across all experiments is **20.89%**, indicating that MinorMiner uses, on average, **20.89%** fewer qubits than the model. Additionally, GReM's performance appears consistent across different experiments. Notably, it performs better on smaller **G** graphs and tends to find better embeddings for larger graphs when it performs well on smaller ones, as observed in experiments 6 and 10 where it obtained a QUR of **14.37%** and **16.36%**, respectively.
It is interesting to comment on the difference between the size of the embeddings found by the model and the embeddings found by MinorMiner in the same experiments. Figure 12 shows this metric grouped by **G** size across all experiments.

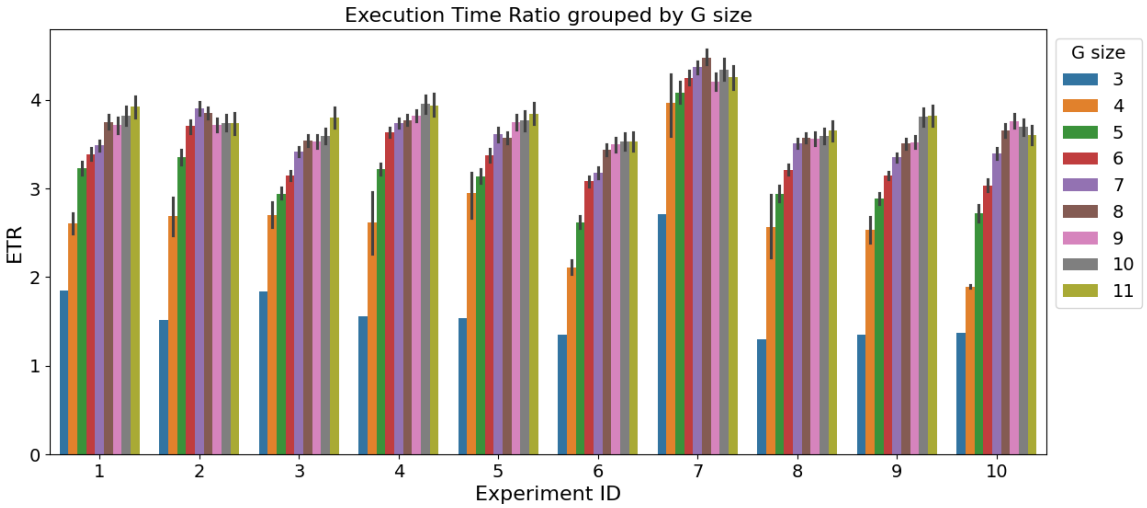


Figure 13: Execution Time Ratio as defined in Section 6.1 across 10 experiments. The X-axis represents the experiment ID, and the Y-axis shows the ratio between the time taken by the model and the time taken by MinorMiner to find the best embedding. Colours indicate the size of the embedded G graphs.

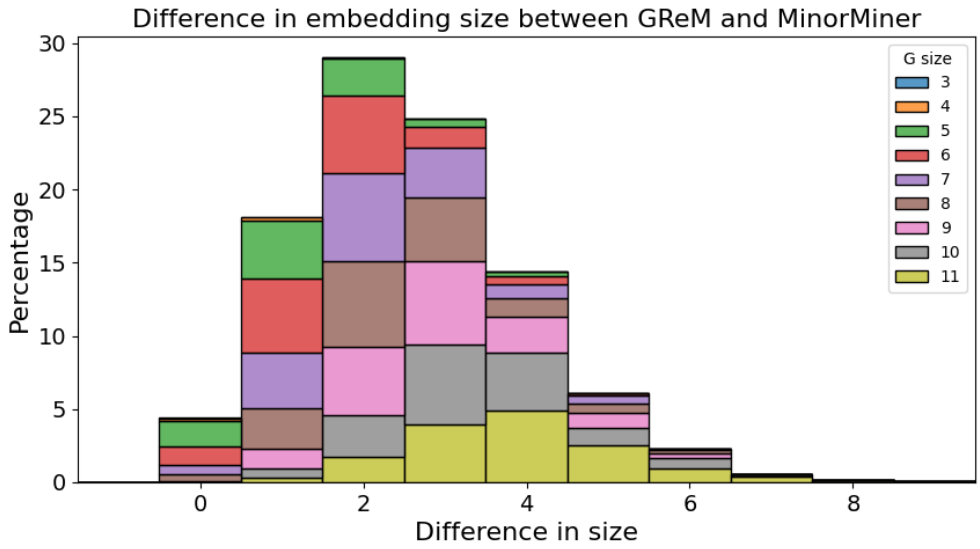


Figure 12: Histogram of the difference in size between the embeddings found by the model and the embeddings found by MinorMiner. The X-axis represent the difference in qubits used for each embedding and the Y-axis is the percentage of embeddings found. The colours depict the size of the **G** graph.

We can see that for almost 30% of the test graphs GReM finds embeddings using only two more qubits than MinorMiner. This difference increases for larger problems, as if we focus on graphs with 11 nodes, we can see that the model, on average, tends to find embeddings using 4 more qubits than the state-of-the-art heuristic. Lastly, we evaluate the Execution Time Ratio obtained by the model. Figure 13 illustrates the average ETR per experiment, based on the size. We can see that GReM is 2 to 4 times slower than MinorMiner with an average of 3.6 times. This means that, on average, the model requires approximately 3.6 times more time to find an embedding compared to MinorMiner. Interesting to notice is that, as the size of **G** increases, the ETR increases at a slower pace. For some experiments the ratio even decreases for bigger graphs, suggesting that GReM may increase its performance w.r.t. to MinorMiner for larger problems.

6.3. Ablation Studies

In this Section, we outline the key components of the model and the rationale behind its design by showing the results of some ablation studies on such components. First, we examine the agent, focusing on how the GNNs

and their structure assist the model in finding effective embeddings. Next, we review the dataset and discuss the process involved in its construction. Finally, we discuss the Curriculum Learning training and analyse how it increases the model's performance across various graph types.

6.3.1 Agent

As described in Section 5.3, the agent is composed by two separate GNNs, one processing the graph problem **G** and the other one the topology graph **H**. These GNNs play a crucial role in the model. Their purpose is to enable the agent to comprehend the structural characteristics of both graphs, increasing the model's generalization capabilities and its flexibility to operate effectively on graphs of varying sizes.
For the **H** graph we decided to combine three GCN layers and to use skip connections to concatenate the results of each layer. Each GCN layer aggregates information from a node's neighbours. By stacking multiple layers, the model can capture information from a wider neighbourhood, allowing the agent to understand more complex patterns in the graph. For **G**, due to the small sizes of the problems, we found one GCN layer to be sufficient. One of the early analyses showed that using Graph Neural Networks instead of Feed-Forward Neural Networks increased the ability of the model of mapping problems using less qubits. In particular, stacking three GCN layers for the topology graph helps the model learn how to perform Minor Embedding on larger topologies.
Table 1 shows these results. The models for the experiments were trained on fully-connected graphs with a fixed size mapped to a Chimera topology having fixed number of qubits. The values shown on Table 1 correspond to the size of the best embeddings found by each model during training. The models analysed are the following ones: the baseline is a simple model without Graph Neural Networks, the GCN-1 model has an agent composed by one GCN layer for **G** and one for **H** and finally the last model, GCN-3, implements the three GCNs layers concatenated for **H**. To have an idea of how good the embedding was, the results of MinorMiner are shown too.

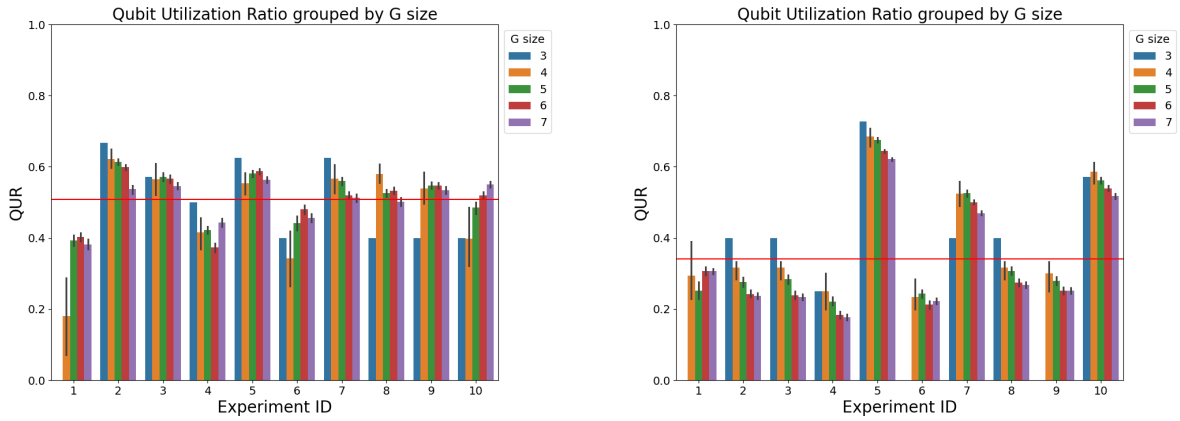
|G|	|H|	Baseline	GCN-1	GCN-3	MinorMiner
3	**512**	10	6	6	3
5	**72**	8	7	8	8
8	**32**	22	24	24	22
8	**128**	38	29	27	22

Table 1: The table compares the embeddings found by three different models for four different input sizes. **|G|** is the size of the problem graph, while **|H|** is the number of qubits of a Chimera topology. Baseline is a simple model without GNNs, GCN-1 contains just one convolution for graph and GCN-3 concatenates three convolutions for **H**. The values shown are the size of the best embedding found for each instance by the models during training. The last column shows the results obtained by MinorMiner for comparison.

We can notice a significant improvement in the embeddings concerning large topologies. In particular, the experiments for problem graphs with 8 nodes and Chimera topologies with 128 qubits highlight that concatenating 3 convolutions helps the model to understand the structure of **H** when we increase its size, and allows to find smaller embeddings even for bigger instances of **G**. Information in the agent is aggregated using average pooling layers. Explorative experiments were made also on max and sum pooling, but no significant change in the results was found.

6.3.2 Dataset

Baseline In the early experiments where a dataset was introduced, it was initially composed by connected graphs with a size ranging form 3 to 7, collected without a precise order and with an unbalanced number of graphs per size. For each **G** size the graphs were equally distributed according to the number of edges and the isomorphism classes. CL was not implemented yet and the model would simply train on the dataset until a fixed number of steps was reached. The model was tested on both Chimera and Zephyr topologies, the former with 1152 qubits, the latter with 2176 qubits. For both experiments we have collected the Success Rate and the Qubit Utilization Ratio. The Success Rate is not show since for both experiments it was **100%**. Instead, the qubits ratio is illustrated in Figure 14. The average qubit ratios for all experiment was **50.81%** for Chimera



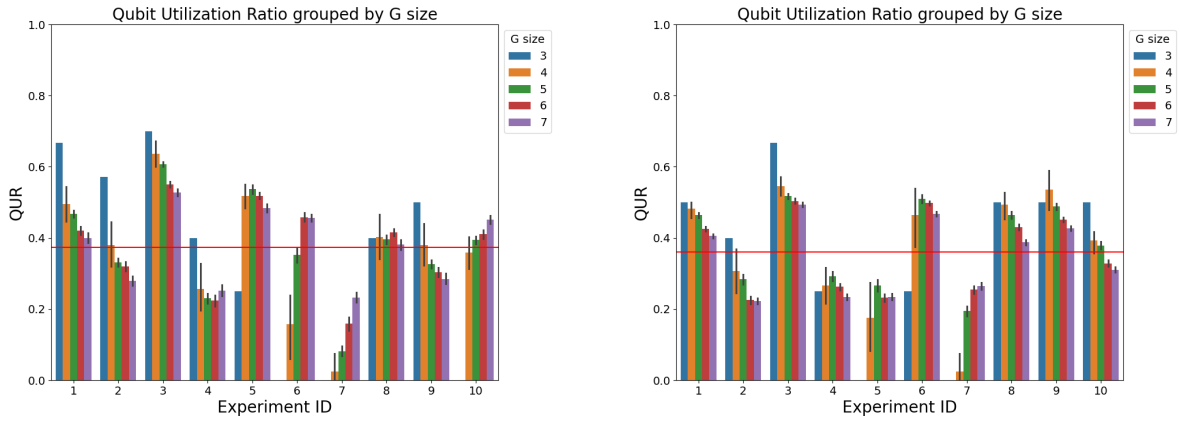
(a) Chimera topology with 1152 qubits.

(b) Zephyr topology with 2176 qubits.

Figure 14: Figure shows the Qubit Utilization Ratio (described in paragraph 6.1) obtained by the model for 10 experiments. The X-axis are the experiment IDs, while the Y-axis is the ratio. The different colours depict the size of the embedded G graphs. Figure 14a shows the results obtained on the Chimera topology, while Figure 14b the ones on Zephyr.

and **34.13%** for Zephyr. This means that using this initial implementation of the dataset, MinorMiner was able to find an embedding using half the qubits of the model. Notice that the performance are better on Zephyr thanks to the increased connectivity of the topology compared to Chimera, as shown in paragraph 2.3.1.

Ordered Once we had a baseline for the Dataset experiments, we tried to improve it with the objective of minimizing the size of the embeddings found by the model on the test set. The first characteristic we introduced to the dataset was to order the graphs based on the increasing size, in such a way that the model would face all graphs with 3 nodes first, then it would move to the set of graphs with 4 nodes, and so on. Every time the model has seen the entire dataset, it would be fed again starting from graphs with 3 nodes and increasing, until all training steps were completed. The goal of this component was to help the model learn better embeddings for larger graphs, starting from simpler cases and progressively increasing the difficulty of the problem during training. The dataset was again tested on Chimera and Zephyr topologies with the same size as in the previous case. Figure 15 shows the results of the experiments.



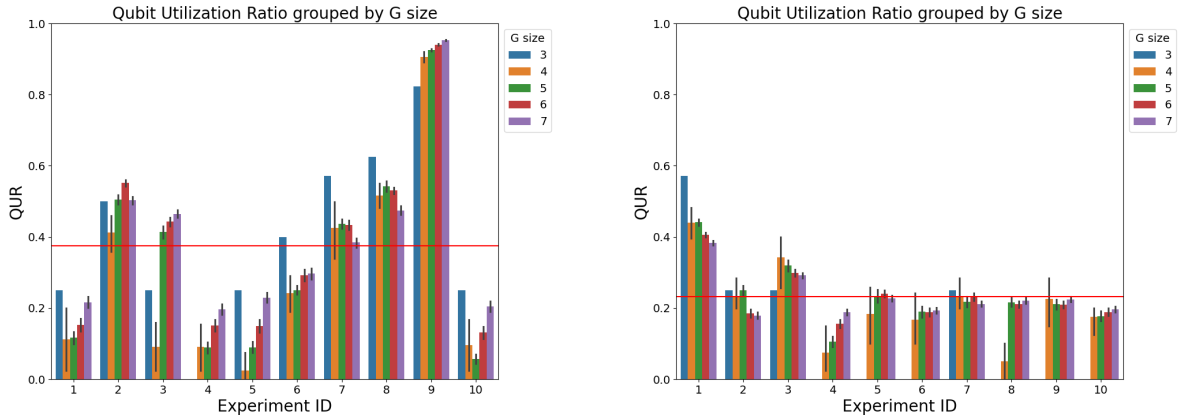
(a) Chimera topology with 1152 qubits.

(b) Zephyr topology with 2176 qubits.

Figure 15: Figure shows the Qubits Utilization Ratio described in paragraph 6.1 obtained by the model for 10 experiments. The X-axis are the experiment IDs, while the Y-axis is the rate. The different colours depict the size of the embedded G graphs. Figure 15a are the results obtained on the Chimera topology, while Figure 15b on the Zephyr.

The average ratios for Chimera and Zephyr are respectively **38.42%** and **35.99%**. The results show a significant improvement in the embeddings for Chimera, while the performance on Zephyr remained almost constant, probably due to the higher connectivity of the topology that already helps the model in finding a good embedding for small graphs.
It is particularly interesting to study how good the model behaves based on the distribution of the Qubit Utilization Ratio for each **G** size in each experiment. We can see that the model tends to find overall good embeddings when it learns better how to map smaller graphs. In fact if we look at experiment 7 in Figures 15a and 15b, we can notice that the model is able to find good embeddings for graphs with 3, 4 and 5 nodes, and therefore it maps relatively well also graphs with 6 and 7 nodes. On the other hand for experiments 1 and 3 in the same Figures, the model fails to build good embeddings for small graphs, hence it is not able to map well larger graphs, too.

Ordered and Balanced After ordering the dataset, we acted on the number of graphs per size. We opted for a balanced dataset, where each G size has roughly the same number of graphs. For smaller sizes, where the total number of graphs is very limited, the graphs are repeated until a set number is reached. The number of graphs per size and the percentage of graphs used for training is the same as the one defined for the complete model in Section 6.2. The objective is to allow the model to see enough graphs for each size to better learn how to perform the embedding on smaller graphs, by giving it more training time on them. Training of the model was not altered from the previous versions. It was tested again on the same Chimera and Zephyr topologies as before and the results are shown on Figure 16.

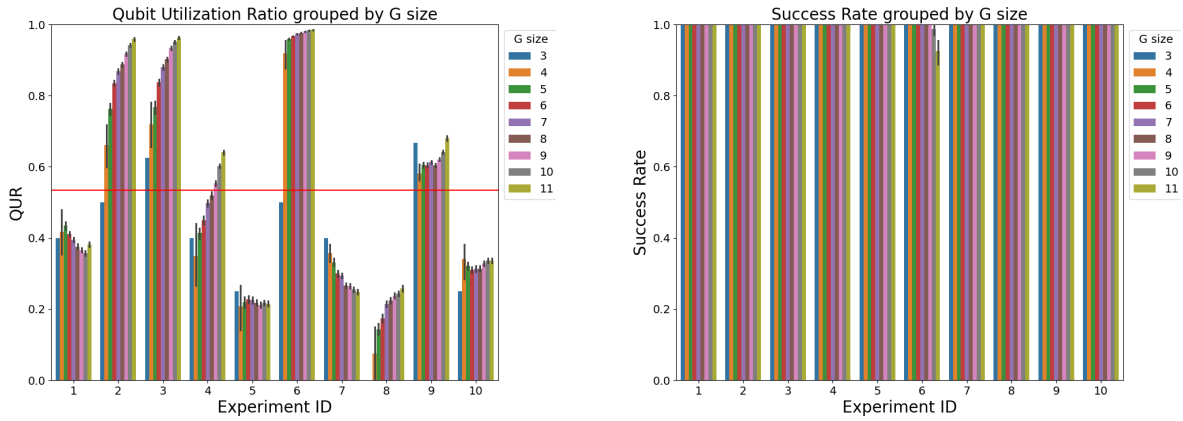


(a) Chimera topology with 1152 qubits.

(b) Zephyr topology with 2176 qubits.

Figure 16: Figure shows the Qubits Utilization Ratio described in paragraph 6.1 obtained by the model for 10 experiments. The X-axis are the experiment IDs, while the Y-axis is the rate. The different colours depict the size of the embedded G graphs. Figure 16a are the results obtained on the Chimera topology, while Figure 16b on the Zephyr.

The average ratios obtained for Chimera and Zephyr are **37.48%** and **23.22%**. Results show that balancing the number of graphs per size improves the performance of the model on the Zephyr topology, while maintaining almost constant the performance on Chimera.
Despite the overall performance on Chimera does not significantly improve, we can notice that the model is able to find better embeddings for multiple experiments compared to the results of the previous model shown in Figure 15a, like experiments 1,4,5 and 10 in Figure 16a. The average Qubits Utilization Ratio does not improve probably due to the results of experiment 9.
Until this point the model was tested on datasets composed by graphs with maximum 7 nodes, so we tried to push the size of the graphs in order to evaluate the performance on more difficult problems. Figure 17 shows the results of the experiments conducted on graphs with up to 11 nodes, on a Zephyr topology with 2176.



(a) Qubit Utilization Ratio.

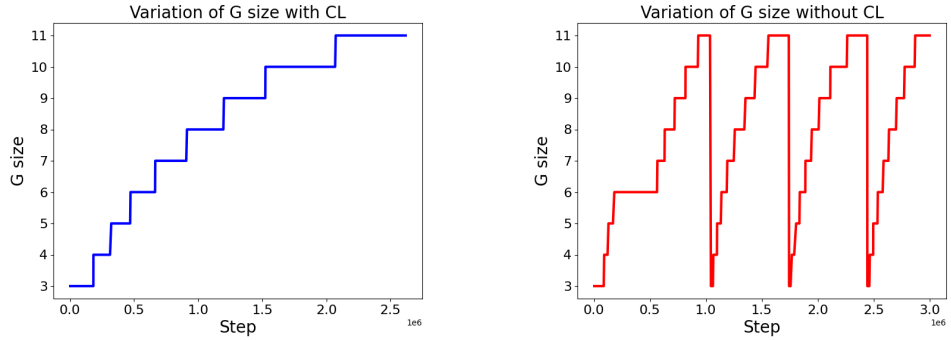
(b) Success Rate.

Figure 17: Figure shows the Qubit Utilization Ratio (Figure 17a) and the Success Rate (Figure 17b) described in paragraph 6.1 obtained by the model for 10 experiments. The X-axis are the experiment IDs, while the Y-axis are the rates. The different colours depict the size of the embedded G graphs. The results were obtained oh a Zephyr topology with 2176 qubits

The average Qubit Utilization Ratio obtained by the model is **50.40%**, while the average success rate is **99.87%**. Results show a significant decrease of the performance on the number of qubits used for the embeddings, and an average success rate slightly worse than the one obtained by the other models, probably due to experiment 6. The model implemented so far is not able to capture well the complexity of larger problems, leading to a decrease in the performance. Working on the dataset was not enough, so we modified training to help the model better learn how to perform minor embedding for each size. To do so we took inspiration from the Curriculum Learning process described in Section 3.3

6.3.3 Curriculum Learning

Baseline CL Up to this point, the model would simply iterate over the dataset multiple times until reaching a fixed number of steps. Instead, as discussed in Section 5.6, by using the CL process, we allow the model to train over graphs with a certain size until some convergence criteria are satisfied. This is repeated for each graph size, in a increasing way, until the maximum size is completed. Figures 18 depicts how the **G** size variates during training with and without CL. The figure clearly shows that when using CL the **G** size progressively increases until the model has reached convergence for the maximum size. Instead, without CL training increases the size and resets it when the model has seen all the graphs in the dataset, repeating the process until a fixed number of steps (set to 3M in our experiments) is reached.



(a) G size with CL.

(b) G size without CL.

Figure 18: Figure illustrates the variation of the **G** size during training with CL (Figure 18a) and without CL (Figure 18b). Without CL, **G** size progressively increases until the maximum size is reached and starts again from the minimum until a fixed number of steps is reached. Instead, with CL the model trains for each size and stops when training reaches convergence for the maximum size.

This method allows the model to better learn the embeddings for each size, and having the graphs ordered helps it to use the knowledge learned for smaller graphs to more easily understand how to map complex problems. Figure 19 shows the results of the initial implementation of CL, conducted on problem graphs with up to 11 nodes mapped on a Zephyr topology with 2176 qubits.

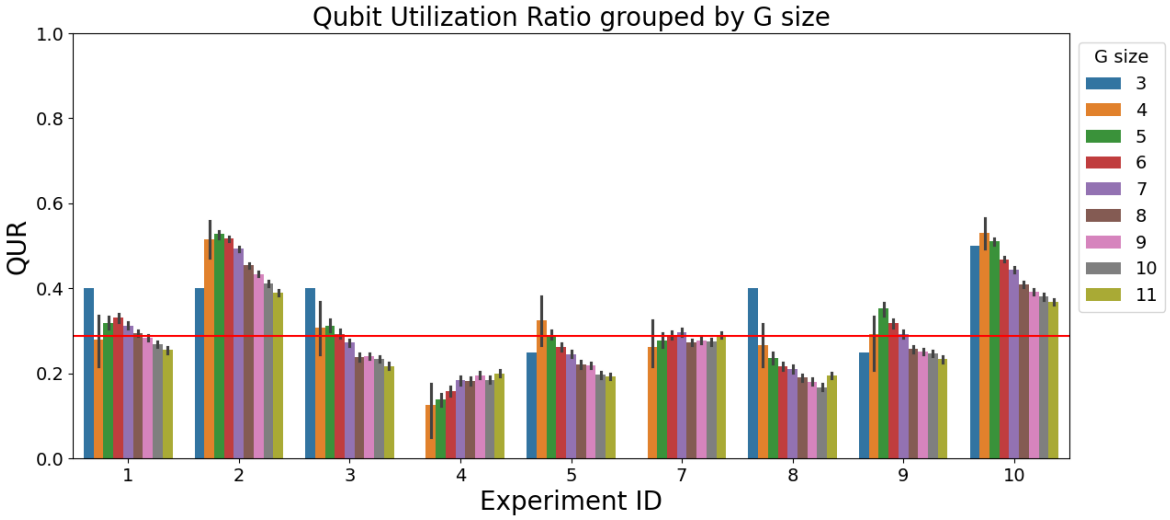


Figure 19: Figure shows the Qubit Utilization Ratio described in paragraph 6.1 obtained by the model for 10 experiments. The X-axis are the experiment IDs, while the Y-axis is the rate. The different colours depict the size of the embedded G graphs. Results were obtained on a Zephyr topology with 2176 qubits. Note that experiment 6 is missing since the model failed to reach convergence

The model obtained an average rate for the qubits of **28.72%**, but it failed to reach convergence for experiment 6. Results show a significant improvement in the performance of the model on graphs with up to 11 nodes using Curriculum Learning. However, the model continues to have some limitations. First of all it failed to reach convergence and produce embeddings for an entire experiment. Furthermore, we noticed that when trained on graphs with larger size, the algorithm would tend to forget how to perform embeddings on smaller graphs, decreasing its performance. If we analyse the QUR for graphs with up to 7 nodes, the model obtained an average QUR of **31.52%**, higher than the QUR obtained by the model without CL when training on graphs with maximum 7 nodes (QUR of **23.22%**) . This demonstrates that, even though the model performance improve on larger problem graphs, its performance on smaller graphs decreased.

Graph re-usage To overcome this problem, we added to training process the re-use of smaller graphs. In other words, when training on graphs with $n > 3$ nodes the model would randomly face graphs with $n-1, n-2, ..., 3$ nodes, in such a way it continues to train on small instances while learning complex problems. The percentage of old graphs introduced over the total number of graphs per size can be tuned. For these experiments we chose to reintroduce 500 smaller graphs for each **G** size, as mentioned in Section 6.2. The model obtained is the complete implementation of the thesis and the results collected are shown in Section 6.2. They show an improvement in the model to a final Qubit Utilization Rate of **20.89%**.

6.4. Further Experiments

Up to this point, we have tested the model on large topologies, with a number of qubits available much bigger than the number of qubits needed to find valid embeddings. So we tried to train GReM on a smaller instance of a Zephyr topologies to analyse how it would behave when the agent has less qubits to choose from. Figure 20 shows the Qubits Utilization Ration obtained for a Zephyr topology with 160 qubits.

Figure 20: Figure shows the Qubit Utilization Ratio described in paragraph 6.1 obtained by the model for 10 experiments. The X-axis are the experiment IDs, while the Y-axis is the rate. The different colours depict the size of the embedded G graphs. Results were obtained on a Zephyr topology with 160 qubits.

It is clear that the quality of the embeddings found by the model increases, with a QUR of **16.64%** when performing embeddings on smaller instances of Zephyr. The Success Rate obtained by GReM is **100%**. Hence reducing the size of the topology helps the model to find better mappings using less qubits.
Figure 21 shows the Execution Time Ratio of the model for these experiments.

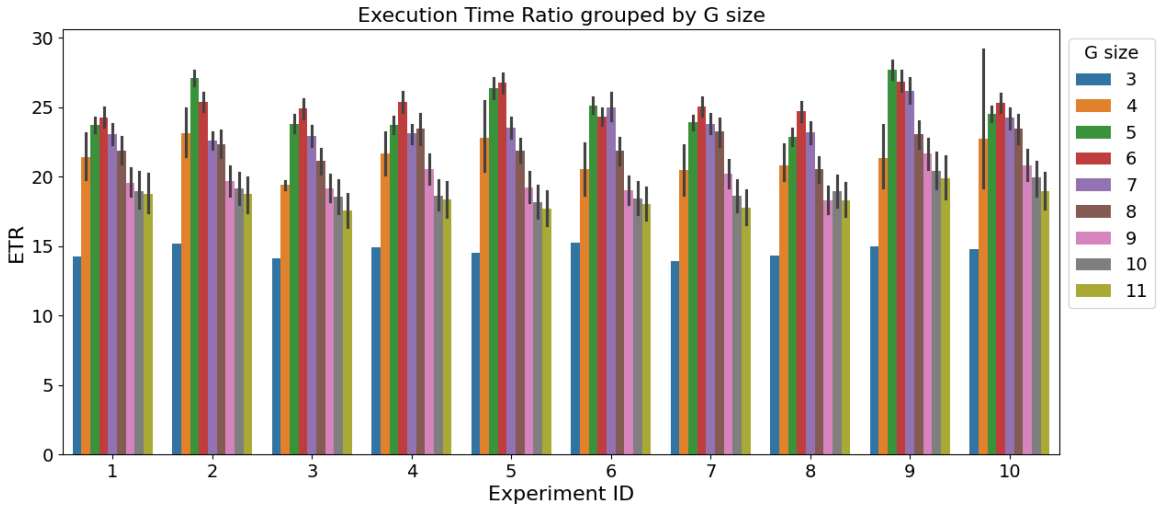
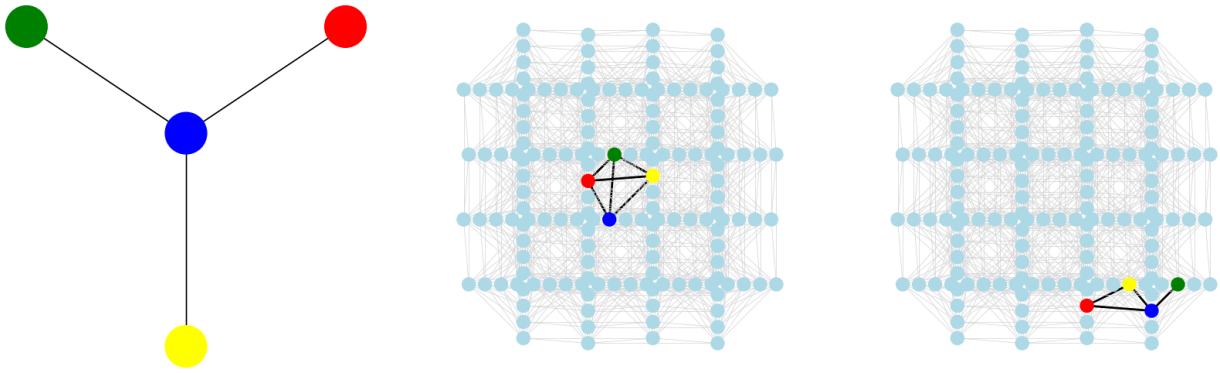


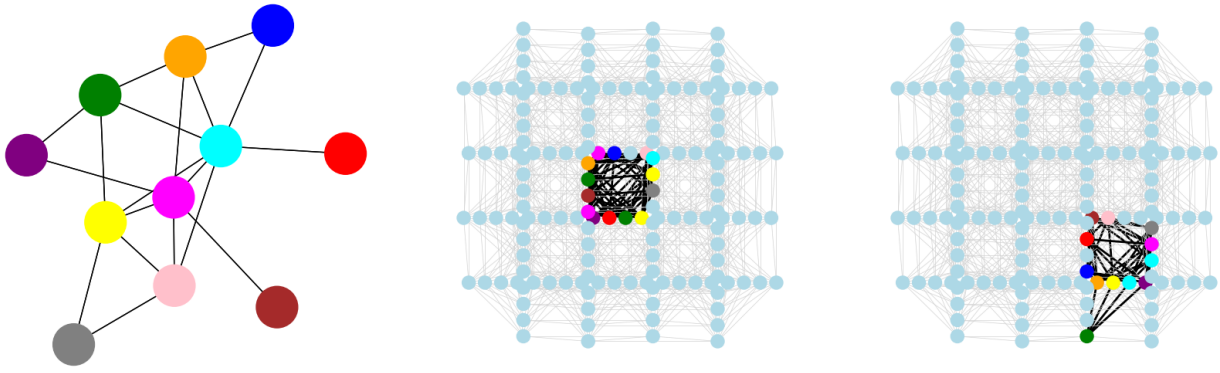
Figure 21: Figure shows the Execution Time Ratio described in paragraph 6.1 obtained by the model for 10 experiments. The X-axis are the experiment IDs, while the Y-axis is the rate. The different colours depict the size of the embedded G graphs. Results were obtained on a Zephyr topology with 2176 qubits.

The model's performance decreases w.r.t. to MinorMiner when performing embeddings on smaller topologies becoming around 15 to 27 times slower than the algorithm, probably due to MinorMiner being able to find embeddings faster on small topologies. Despite this, the results highlight the ability of the model to reduce the time ratio when mapping larger **G** graphs. This trend was already discussed in Section 6.2, but it is more evident in these experiments.
Finally Figures 22 and 23 illustrate an example of embeddings found by GReM and MinorMiner during the testing phase of one of these experiments.



(a) Problem Graph. (b) Model Embedding. (c) MinorMiner Embedding.

Figure 22: Figure 22a shows a **G** graph with 4 nodes in the test dataset of the model. Figure 22b shows an embedding found by the model for the graph shown in Figure 22a on a Zephyr topology with 160 qubits during the testing of the experiment 8. Figure 22c illustrates an embedding found by MinorMiner for the same **G**.



(a) Problem Graph. (b) Model Embedding. (c) MinorMiner Embedding.

Figure 23: Figure 23a shows a **G** graph with 11 nodes in the test dataset of the model. Figure 23b shows an embedding found by the model for the graph shown in Figure 23a on a Zephyr topology with 160 qubits during the testing of the experiment 8. Figure 23c illustrates an embedding found by MinorMiner for the same **G**.

Figures 22b and 22c show examples of embeddings found by the model and MinorMiner on the test graph depicted in Figure 22a with 4 nodes. We can see that for this problem the model is able to find an embedding using the same amount of qubits as MinorMiner. Instead, Figures 23b and 23c show examples of embeddings found on the test graph depicted in Figure 23a with 11 nodes. In this case, the model is not able to find a perfect embedding, but it still manages to use only 14 qubits for the mapping, two more than the state-of-the-art algorithm.

7. Conclusions

To conclude, we developed a PPO-based Reinforcement Learning model integrated with Graph Neural Networks capable of finding Minor Embeddings for generic QUBO problems on D-Wave Quantum Annealers. Results show that using GNNs to work with graphs helps the agent recognize complex patterns within both the problem structures and the topology graphs. Moreover, the Curriculum Learning procedure helps the model better learn how to perform embeddings on graphs with different sizes, increasing the level of generalization obtained. In particular, we demonstrated that it is able to achieve results comparable to the state-of-the-art heuristic MinorMiner for problems with up to 11 nodes, mapped on next generation Quantum Annealers with up to 2176 qubits. The model is able to find valid embeddings for all graphs on the test dataset, producing mappings that occupy on average around **21%** more qubits than MinorMiner embeddings across all experiments. Results also

showed that when searching for embeddings on a Zephyr topology with only 160 qubits, the model is able to reduce the number of qubits used, occupying, on average, only around **17%** more qubits than MinorMiner. When embedding on a Zephyr topology with 2176 qubits, the model is 2 to 4 times slower than MinorMiner depending on the size of the problems faced, but the speed up obtained by MinorMiner grows less when increasing the size of the problems, suggesting that the model may increase its performance w.r.t. to the state-of-the-art heuristic when facing larger problems. This shows that Reinforcement Learning and Graph Neural Networks represent an innovative and promising approach for finding Minor Embeddings for generic problems on Quantum Annealers. In future work, GReM could be trained on problem and topology graphs with larger size, progressively increasing the complexity of the problems, until MinorMiner is no longer able to produce efficient embeddings. Moreover, other aggregation and pooling layers can be tested for the Graph Neural Networks, especially when testing on more complex problems. Finally, new features can be added to the graphs to enrich the information encapsulated in the state of the environment, helping the agent understand more complex structures.

References

[1] Esteban Aguilera, Jins Jong, Frank Phillipson, Skander Taamallah, and Mischa Vos. Multi-objective portfolio optimization using a quantum annealer. *Mathematics*, 12:1291, 04 2024.

[2] Steve Awodey. *Category Theory*. Oxford University Press, Inc., USA, 2nd edition, 2010.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 41–48. ACM, 2009.

[4] Jacob D. Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nat.*, 549(7671):195–202, 2017.

[5] Kelly Boothby, Paul Bunyk, Jack Raymond, and Aidan Roy. Next-generation topology of d-wave quantum processors, 2020.

[6] Jun Cai, William G. Macready, and Aidan Roy. A practical heuristic for finding graph minors. *CoRR*, abs/1406.2741, 2014.

[7] Ameya Daigavane, Balaraman Ravindran, and Gaurav Aggarwal. Understanding convolutions on graphs. *Distill*, 2021. https://distill.pub/2021/understanding-gnns.

[8] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017.

[9] Fred W. Glover and Gary A. Kochenberger. A tutorial on formulating QUBO models, 2018.

[10] Erica Grant and Travis S Humble. Benchmarking embedded chain breaking in quantum annealing*. *Quantum Science and Technology*, 7(2):025029, mar 2022.

[11] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[12] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *CoRR*, abs/2006.14171, 2020.

[13] Tadashi Kadowaki and Hidetoshi Nishimori. Quantum annealing in the transverse ising model. *Phys. Rev. E*, 58:5355–5363, Nov 1998.

[14] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[15] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. *CoRR*, abs/1904.08082, 2019.

[16] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. https://distill.pub/2021/gnn-intro.

[17] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009.

[18] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

[20] Jacobo Torán. On the hardness of graph isomorphism. *SIAM Journal on Computing*, 33(5):1093–1108, 2004.

[21] Sheir Yarkoni, Elena Raponi, Thomas Bäck, and Sebastian Schmitt. Quantum annealing for industry applications: introduction and review. *Reports on Progress in Physics*, 85(10):104001, September 2022.

[22] Anpeng Zhang and Xiutao Feng. Applications of quantum annealing in cryptography. *CoRR*, abs/2211.10076, 2022.

Abstract in lingua italiana

I Quantum Annealers (QA) sono un paradigma di computer quantistici progettati per risolvere problemi di ottimizzazione combinatoria sfruttando leggi della meccanica quantistica. I Quantum Annealers rappresentano una tecnologia promettente, con il potenziale di risolvere in modo efficiente problemi di ottimizzazione combinatoria NP-hard, offrendo un rilevante vantaggio in termini di velocità rispetto ai computer classici. Una delle principali sfide nella risoluzione di un problema su un QA è il processo di mappatura del problema, scritto nella formulazione QUBO, sulla topologia del QA. A causa della struttura variabile del problema da risolvere e della struttura rigida e sparsa dei Quantum Annealers, trovare una buona mappatura può diventare computazionalmente costoso, riducendo così il vantaggio ottenuto con l'uso di un computer quantistico. In questa tesi sviluppiamo un modello di Reinforcement Learning basato su PPO, capace di trovare Minor Embeddings per problemi QUBO. Il modello è strutturato per funzionare come un processo di Curriculum Learning ed è integrato con Graph Neural Networks per aiutare l'agente a comprendere meglio la struttura del problema e la topologia del QA. Il modello è stato addestrato su un dataset composto da grafi connessi con 3 a 11 nodi, che rappresentano problemi QUBO generici, ordinati per dimensione crescente e distribuiti equamente in base ai loro isomorfismi e al numero di archi. Gli embeddings sono stati eseguiti sulle topologie Chimera e Zephyr di D-Wave, con un massimo di 1152 e 2176 qubits rispettivamente. La qualità dei risultati è confrontata con l'algoritmo MinorMiner, considerato di riferimento, poiché le soluzioni di questo algoritmo sono quasi ottimali per i problemi affrontati. Dimostriamo che il nostro modello riesce sempre a trovare embeddings validi, utilizzando in media un massimo del 21% di qubit in più rispetto a MinorMiner. Complessivamente, la qualità dei risultati è buona e il modello sviluppato rappresenta un approccio innovativo e promettente, con il potenziale per costituire un punto di partenza nell'uso di modelli di Reinforcement Learning per trovare Minor Embeddings per Quantum Annealers.

Parole chiave: Quantum Annealers; Minor Embedding; Apprendimento per Rinforzo; Reti Neurali su Grafi

Acknowledgements

We acknowledge the financial support from ICSC - "National Research Centre in High Performance Computing, Big Data and Quantum Computing", funded by European Union – NextGenerationEU. We acknowledge the CINECA award under the ISCRA initiative, for the availability of high-performance computing resources and support.