UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

UNIVERSITY OF MODENA AND REGGIO EMILIA

"Enzo Ferrari" Department of Engineering

Master’s Degree in Artificial Intelligence Engineering (LM-32)

Graph Attention–Based Neural Embedding for Atom Placement Optimization in Neutral-Atom Quantum Computing

Supervisor:
Prof. Simone Calderara
Co-supervisor:
Dr. Anita Camillini

Candidate:
Aldo Flotta
Student ID 193242

ACADEMIC YEAR 2024/2025

Abstract

Graph Attention–Based Neural Embedding for Atom Placement Optimization in Neutral-Atom Quantum Computing

Neutral-atom quantum computing has emerged as a promising platform for programmable quantum systems due to its flexible geometry and controllable interactions between qubits. In these architectures, the spatial arrangement of atoms plays a crucial role, as it directly determines the interaction graph that can be realized on hardware. This thesis investigates the fundamentals of neutral-atom quantum computing and explores the use of machine learning to optimize atom placement in quantum registers.

Starting from the method proposed in the paper "Neural-powered unit disk graph embedding qubits connectivity for some QUBO problems", which employs a neural network to generate spatial embeddings compatible with hardware constraints, this work develops an enhanced architecture aimed at improving the quality and generalization of the learned embeddings. In particular, the original approach is extended by introducing a Graph Neural Network capable of explicitly modeling the structure of the interaction graph. The proposed model leverages Graph Attention layers to learn adaptive importance weights between nodes, allowing the network to better capture relational dependencies and structural patterns in the graph during the embedding process.

The improved architecture is evaluated on a simplified formulation of a quantum molecular docking problem, where molecular interaction constraints must be translated into feasible atom placements on a neutral-atom quantum device. The resulting toy model demonstrates how graph-based neural architectures, especially attention-based mechanisms, can improve the optimization of atom positions by iteratively adjusting graph node coordinates to satisfy the physical constraints of neutral-atom quantum registers.

Keywords: AI, Machine Learning, Graph Neural Network, Graph Attention, Quantum Computing

Contents

Abstract **1**

1 Introduction **1**

1.1 CINECA 1

1.1.1 EuroHPC and Leonardo 1

1.1.2 Quantum Computing Lab 2

1.2 The traineeship 3

2 Graph Machine Learning **5**

2.1 Neural Networks 5

2.1.1 Model Training and Loss Function 6

2.1.2 Backpropagation and Optimization 7

2.1.3 Hyperparameters and Regularization 7

2.1.4 Autoencoder as an example of neural network 8

2.2 Graph Neural Networks 9

2.2.1 Graph Convolutional Networks 11

2.2.2 Graph Attention Networks 13

3 Neutral Atom Quantum Computing **16**

3.1 Quantum Computing Models 17

3.2 Physical implementations of quantum computers 18

3.3 Applications of quantum computers 19

3.4 The NISQ era 20

3.5 Quantum computing with neutral atoms 22

3.5.1 Qubits 23

3.5.2 Quantum algorithm workflow 24

3.5.3 Quantum register 25

3.5.4 Digital and analog modes 26
3.5.5 Operating an atomic qubit register 29
3.5.6 Interacting atoms 32
3.5.7 Applications of a neutral atom QPU 35

4 GEAN: Graph Embedding Autoencoder Network **41**
4.1 Motivation 41
4.2 Unit disk graph embedding under hardware constraints 42
4.3 GEAN Methodology 42
4.3.1 Architecture 42
4.3.2 Loss Function and Optimization 45
4.4 Experimental Results 45
4.4.1 Problem Families 45
4.4.2 Comparison with Classical Solver 45
4.4.3 3D Extension 46
4.5 Replication 47
4.5.1 Utils 47
4.5.2 The neural network 57
4.5.3 Testing on the antennas problem 62
4.5.4 Testing on the protein problem 64

5 Improvements and testing of GEAN **69**
5.1 The improved neural network 69
5.1.1 Testing on proteins 74
5.2 Quantum Molecular Docking 74
5.2.1 Molecule to graph 76
5.2.2 Binding interaction graph 78
5.2.3 From WMC to MWIS problem 80
5.3 Implementation of QMD 82
5.3.1 Data Preparation 82
5.3.2 Pharmacophore Feature Extraction 82

5.3.3 Binding Interaction Graph construction 85
5.3.4 Complementary Graph and Neutral Atom Embedding 86
5.3.5 Quantum Register and adiabatic evolution 88
5.4 Conclusions 89
5.5 Future work 90

References **92**

List of Code

4.1 CSV to coordinate converter . . . 47
4.2 Coordinate-to-graph converter . . . 48
4.3 Graph-to-coordinate converter . . . 48
4.4 Connected components graph extractor . . . 48
4.5 Graph to adjacency matrix converter . . . 49
4.6 Visualization and comparison of graphs . . . 49
4.7 Side by side comparison of graphs . . . 50
4.8 Compact graph checksums . . . 52
4.9 Violated constraint checker . . . 52
4.10 Saving a graph . . . 53
4.11 Loading a graph . . . 53
4.12 Statistics of the results . . . 54
4.13 Statistics to CSV . . . 56
4.14 Random graph generator . . . 57
4.15 Autoencoder . . . 57
4.16 Difference and distance layers . . . 58
4.17 Full Model . . . 59
4.18 Tanh activation function . . . 60
4.19 Custom loss function . . . 60
4.20 Model training function . . . 61
4.21 Example of Slurm Script . . . 65
5.1 GAT-based Node Encoder . . . 69
5.2 Full improved model . . . 71
5.3 Improved training function . . . 72

1. Introduction

In the following sections, I will explain the work I have done during the 6 month trainsheep at CINECA that ended up in this master thesis.

1.1 CINECA

Cineca stands as one of Italy's largest computing centers and is globally recognized for its leadership in High Performance Computing (HPC). At the same time, Cineca serves as a crucial provider of solutions and services for universities, research centers, the Ministry of Education, the Ministry of University and Research, and other institutions. Cineca drives and champions the digital transition through innovation, developing cutting-edge technological solutions, and creating integrated platforms tailored for and in collaboration with its consortium members. Its primary focus is on supporting institutions and administrations. In the realm of supercomputing, Cineca manages infrastructure, pioneers frontier applications, collaborates on co-designed technological solutions, and leads research and innovation projects. Established in 1969 in Casalecchio Di Reno (Bologna), Cineca operates as a consortium of publicly held entities on a not-for-profit basis, dedicated to advancing the common good and serving the interests of its consortium members and the national system. Its membership comprises 122 entities, including 2 ministries (Ministry of Education, the Ministry of University and Research), 71 Italian universities, 49 national public institutions (such as research institutions, university hospital companies-IRCCS, AFAM institutions, agencies, and institutions)[1].

1.1.1 EuroHPC and Leonardo

The European High-Performance Computing Joint Undertaking (EuroHPC JU) is a public-private partnership that allows the European Union, its participating countries and private partners to

[1] `https://www.cineca.it/en/about-us/cineca-today`

coordinate their efforts and pool their resources to make Europe a world leader in supercomputing[2]. The strong partnership with the EuroHPC initiative has led to the realization of the Leonardo project, a significant step forward in raising European research in the field of computational sciences. The ultimate goal is to strengthen the European presence in high-performance computing, a strategic asset to promote the technological growth of the member states of the Union [3]. Leonardo is one of the three pre-exascale systems announced by EuroHPC JU, it debuted on the TOP500 in November 2022 ranking fourth in the world and now it is still in the the 10th place of the ranking. EuroHPC is also deploying a family of quantum computers hosted in Europe and tightly coupled to EuroHPC supercomputers. These systems will allow users to explore hybrid classical–quantum workflows, where quantum processors are used alongside powerful HPC systems for specific algorithmic steps. This gives European researchers, industry and public administrations early access to a range of quantum platforms within a trusted European infrastructure [4].

1.1.2 Quantum Computing Lab

The Quantum Computing Lab is a new initiative by Cineca with the aim to investigate and develop Quantum Computing (QC) tools for information processing and computation. The emergence of QC is leading to a radical paradigm shift in the approach to computation exploiting the principles of quantum mechanics. It has the potential to address and solve several classes of problems that cannot be treated with classical techniques with the current supercomputers[5]. At the end of 2022, CINECA achieved a significant milestone by being selected as the hosting institution for one of the six quantum computers under the EuroHPC JU[6]. Within the framework of EuroQCS-Italy, a quantum simulator developed by Pasqal (Orion Beta model) has been installed. The system is based on neutral-atom technology and features 140 qubits operating in analog mode, with a planned upgrade by 2027 toward a hybrid analog/digital architecture. In parallel, a 54-qubit digital quantum computer based on full-stack superconducting qubit technology has been deployed by IQM

[2] https://www.eurohpc-ju.europa.eu/about/discover-eurohpc-ju_en
[3] https://leonardo-supercomputer.cineca.eu/
[4] https://www.eurohpc-ju.europa.eu/quantum-technologies/quantum-computing_en
[5] https://www.quantumcomputinglab.cineca.it/en/the-project/
[6] https://www.hpc.cineca.it/our-activities/technologies/quantum-computing/

(Radiance 54 model).[7].

1.2 The traineeship

During this six-month period, I worked at the Quantum Computing Lab, where I investigated the integration of machine learning techniques with emerging quantum hardware. In particular, I focused on optimizing qubit positioning within the register of a Pasqal neutral-atom quantum computer, addressing the problem as a constrained graph-embedding task. This work was motivated by a concrete and challenging real-world application, namely Quantum Molecular Docking, where the efficient mapping of interaction graphs onto physical quantum devices plays a crucial role in enabling scalable simulations. The core idea underlying this thesis is that hardware-aware representations are essential when bridging classical and quantum computation. In near-term quantum devices, and especially in neutral-atom platforms, physical constraints such as limited connectivity, geometric arrangement, and control restrictions significantly impact the feasibility of running quantum algorithms. For this reason, I explored the use of graph machine learning, specifically graph neural networks, as a tool to learn embeddings that respect such constraints while preserving the structural properties of the original problem. The thesis is structured to progressively introduce both the theoretical background and the practical contributions of this work. In Chapter 2, I present the fundamentals of machine learning, starting from basic neural network models and moving toward graph neural networks, emphasizing their suitability for handling relational and structured data. Particular attention is given to how node features, edge information, and local neighborhoods can be leveraged to encode complex constraints in a learnable way. In Chapter 3, I introduce the principles of quantum computing that are relevant to this work, with a focus on the Noisy Intermediate-Scale Quantum (NISQ) era and its limitations. The discussion then narrows to neutral-atom quantum computing, detailing its operational mechanisms, advantages, and hardware constraints that directly influence algorithm design and problem encoding. The final chapters present the original contributions of this thesis. I first introduce a graph embedding autoencoder framework designed to generate hardware-compatible qubit placements, and I evaluate its performance on benchmark scenarios as well as on molecular docking instances. Building on the

[7] https://www.cineca.it/en/infrastructure/evolution-infrastructure

insights and limitations observed, I then propose an improved architecture that incorporates more expressive message-passing mechanisms, enabling a better integration of adjacency information. Finally, I demonstrate a complete end-to-end pipeline for Quantum Molecular Docking, showing how classical graph representations can be transformed into quantum-ready encodings and executed on neutral-atom platforms. Overall, this work aims to provide a concrete and reproducible bridge between graph-based machine learning methods and quantum computing applications, highlighting how learning-based approaches can play a central role not only in preprocessing, but in enabling the practical use of near-term quantum devices for complex optimization problems.

2. Graph Machine Learning

In recent years, Artificial Intelligence (AI) has grown rapidly. This trend is also reflected in the 2024 Nobel Prizes: the Chemistry prize was awarded one-half to David Baker for "computational protein design" and one-half to Demis Hassabis and John M. Jumper for "protein structure prediction," while the Physics prize was awarded to John J. Hopfield and Geoffrey E. Hinton for "foundational discoveries and inventions that enable machine learning with artificial neural networks" [11]. In this chapter, I provide basic definitions of AI. AI is defined as the capability of computational systems to perform tasks typically associated with human intelligence, such as learning, reasoning, problem-solving, perception, and decision-making [12]. In particular, Machine Learning (ML) has become increasingly important. ML studies statistical algorithms that learn from data without being explicitly programmed (Arthur Samuel, 1959), and generalize to unseen data for a specific task.

2.1 Neural Networks

One of these algorithms is the Neural Network (NN), a computational model inspired by the brain. NNs are organized in layers: one input layer, one output layer, and one or more hidden layers in between. Each layer contains nodes (artificial neurons) connected by edges. Each neuron receives signals from connected neurons, processes them, and sends an output after applying a non-linearity called an activation function. The strength of each connection is determined by a weight, which is updated during training. An NN with at least two hidden layers is called a Deep Neural Network (DNN). DNNs are the foundation of Deep Learning (DL), a subfield of ML[1]. DL is a set of ML algorithms that model high-level abstractions in data using architectures composed of multiple non-linear transformations (LeCun, 2015).

[1] https://en.wikipedia.org/wiki/Neural_network_(machine_learning)

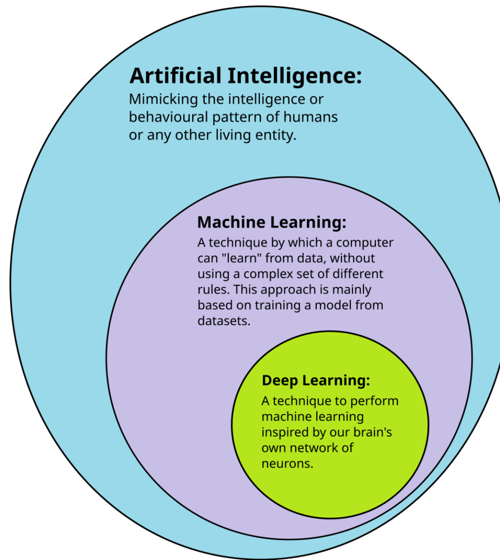

Figure 2.1: AI, ML and DL - from `https://en.wikipedia.org/wiki/Deep_learning`

Before introducing Graph Neural Networks (GNNs), it is necessary to formalize the fundamental principles underlying the training of machine learning models. These concepts will be extensively used in Chapter 4 and 5 and provide the theoretical foundation for understanding how GNNs learn from graph-structured data. Throughout this section inputs will be denoted by $\mathbf{x}$, targets by $\mathbf{y}$, model parameters by $\boldsymbol{\theta}$, and datasets by $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i), i = 1, \ldots, N\}$.

2.1.1 Model Training and Loss Function

A machine learning model is formalized as a parametric function

$$f_{\boldsymbol{\theta}} : \mathcal{X} \rightarrow \mathcal{Y}, \tag{2.1}$$

where $\mathcal{X}$ denotes the input space, $\mathcal{Y}$ the output space, and $\boldsymbol{\theta}$ the set of learnable parameters. In deep neural networks, $\boldsymbol{\theta}$ consists of weights and biases across multiple layers.

Ideally, the training objective is to estimate optimal parameters $\boldsymbol{\theta}^*$ that minimize the expected risk over the true data-generating distribution p_{data}:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{data}}[\ell(f_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y})]. \tag{2.2}$$

Since the true distribution is generally unknown, we rely on the Empirical Risk Minimization (ERM) framework. This framework explicitly distinguishes between the point-wise loss $\ell(\cdot,\cdot)$, which computes the error of the model's prediction on a single data point, and the global empirical loss $\mathcal{L}(\boldsymbol{\theta})$. The latter is defined over the available dataset $\mathcal{D}$ as the average of the individual point-wise losses:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \ell\big(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i\big). \tag{2.3}$$

The choice of ℓ depends on the specific task (e.g., mean squared error for regression, cross-entropy for classification). The practical optimization problem thus becomes minimizing the global empirical loss:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}). \tag{2.4}$$

2.1.2 Backpropagation and Optimization

To minimize the loss function, we iteratively update the parameters using gradient-based optimization. Backpropagation efficiently computes the gradient of the loss with respect to the parameters, $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$, by applying the chain rule through the computational graph of the model [5].

Stochastic Gradient Descent (SGD) and its variants perform these updates using mini-batches $\mathcal{B} \subset \mathcal{D}$:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}_t), \tag{2.5}$$

where η is the learning rate. An epoch corresponds to one full pass over the dataset $\mathcal{D}$. Training typically requires multiple epochs, allowing the model to iteratively refine its parameters until convergence. In practice, adaptive optimization algorithms such as Adam [7] are frequently employed to accelerate this process.

2.1.3 Hyperparameters and Regularization

Hyperparameters define the configuration of the learning algorithm and are not learned from the data. They include architectural choices (e.g., number of layers, hidden dimensions), optimization settings (e.g., learning rate, batch size), and regularization strategies.

The learning rate η is particularly critical, as it controls the step size in the parameter space. Scheduling strategies (e.g., decay, warm-up) are commonly used to improve training stability. Furthermore, to mitigate overfitting and prevent the co-adaptation of neurons, regularization techniques such as dropout are widely applied. Formally, dropout introduces a stochastic masking operator $\mathbf{m} \sim \text{Bernoulli}(p)$ to the hidden representations $\mathbf{h}$ of a given layer:

$$\tilde{\mathbf{h}} = \mathbf{m} \odot \mathbf{h}, \tag{2.6}$$

where $\odot$ denotes element-wise multiplication and p is the retention probability. In practice, this operation effectively deactivates (or "drops") a random subset of nodes along with their corresponding connections during each forward pass of the training phase. By preventing the network from relying on any specific subset of neurons, dropout forces the model to learn more robust and distributed feature representations. During the inference phase, the stochastic behavior is disabled, and the full capacity of the network is utilized, with the activations typically scaled by p to preserve the expected value of the node outputs.

2.1.4 Autoencoder as an example of neural network

An autoencoder is a feed-forward neural network trained to reconstruct its input at the output. The network can be viewed as two parts: an encoder $h = f(x)$ and a decoder that produces a reconstruction $r = g(h)$. The goal is for h to capture useful and salient features of the training data.

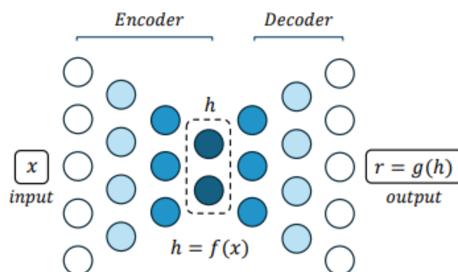

Figure 2.2: Autoencoder - from [10]

The objective is to minimize the error between the input x and the reconstruction r.

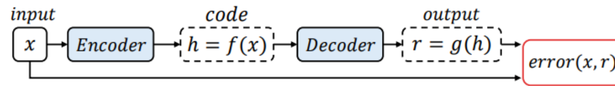

Figure 2.3: General structure of an Autoencoder - from [10]

It is not useful if an autoencoder simply learns the identity mapping, i.e., $g(f(x)) = x$ everywhere. This can happen when the encoder and decoder have too much capacity, for example when the hidden code has a larger dimension than the input. A first regularization strategy is to constrain h to have a smaller dimension than x (undercomplete representation), forcing the model to capture the most salient factors of variation. Training minimizes $\mathcal{L}(x, g(f(x)))$, where $\mathcal{L}$ penalizes dissimilarity between x and $g(f(x))$ (e.g., MSE). However, even undercomplete autoencoders may still learn a near-copying function if model capacity is too high. To avoid learning the identity function, autoencoders are regularized so they can reconstruct only approximately:

- Regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output.
- Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.
- This trade-off between reconstruction and regularization helps capture the structure of the data-generating distribution.

There are many variants of regularized autoencoders; in this thesis, I consider a simple case based on Dropout.

2.2 Graph Neural Networks

Although the core ideas of DL have existed for decades, performance has improved dramatically only in recent years. This progress is due to complementary factors, especially increased computational power, GPU training (including distributed settings), and large datasets [10]. Data has been a key driver of the DL revolution. Depending on input structure, different NN architectures were

developed: Convolutional Neural Networks (CNNs) for images, Recurrent Neural Networks (RNNs) for sequential data such as text, speech, and time series, and Transformers, which power large language models such as ChatGPT. The data types discussed so far mostly have regular structure and can be viewed in Euclidean domains. However, many real-world domains are non-Euclidean, such as road networks, citation networks, body-pose skeletons, and molecular graphs. For these data types, specialized models called Graph Neural Networks (GNNs) are used [9].

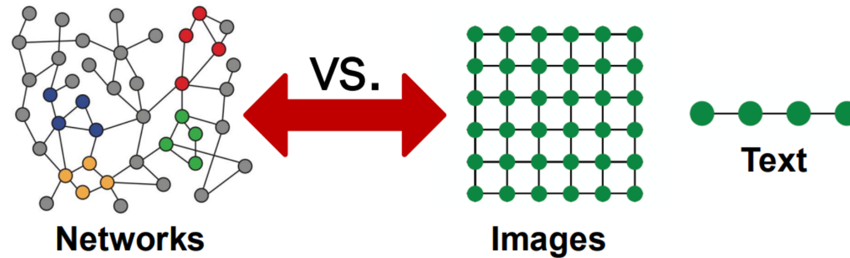

Figure 2.4: Euclidean vs Non Euclidean domains - from [9]

Graphs are a general language for describing entities, relations, and interactions. Complex domains often have rich relational structure that can be represented naturally as graphs. A graph $G = (V, E)$ consists of a set of nodes (vertices) V and a set of edges $E \subseteq V \times V$ that represent relationships between vertices. Graphs can be directed or undirected, and weighted or unweighted. To learn from graph structure, we need a formal representation of connectivity. Listing edges explicitly quickly becomes unwieldy, so matrix-based representations are preferred. One of the most common is the adjacency matrix, whose entries contain the edge weight (or 1 in unweighted graphs) when two nodes are connected, and 0 otherwise. Graph-structured data are complex: they can have arbitrary size, irregular topology, no fixed node ordering or reference point, dynamic connectivity, and multimodal features. For this reason, GNNs must address several challenges:

- Irregular structure: nodes may have different numbers of neighbors (no fixed-size receptive field).
- No spatial locality: unlike images, there is no inherent notion of distance or order between nodes.
- Permutation invariance: the output should not depend on the order in which nodes are

processed.

- No global coordinate system: we cannot rely on a fixed layout or orientation.
- Dynamic topology: edge connectivity may evolve or depend on node features.

To learn from a graph, we map nodes (or entire graphs) to embeddings so that distances in embedding space reflect similarity in the original network. These embeddings are useful for downstream tasks such as node classification, link prediction, graph classification, anomaly detection, and clustering. Classical DL models are generally not suitable as graph encoders because they do not naturally handle these challenges, especially permutation invariance. A graph embedding function should produce the same result regardless of node ordering. In summary, GNNs apply permutation-invariant message-passing and aggregation functions over node neighborhoods. Typical aggregation operators include max, sum, and mean. The aggregated information is then passed through a neural layer to generate an updated node embedding that captures both node features and local context [10].

2.2.1 Graph Convolutional Networks

One of the earliest ideas was to generalize convolutions beyond regular lattices, accounting for the fact that graphs have no fixed sliding window and no canonical node ordering. In 2017, Kipf & Welling [8] introduced the Graph Convolutional Network (GCN) layer, where each neural network layer can be written as a non-linear function

$$H^{(l+1)} = f(H^{(l)}, A) \tag{2.7}$$

where $H^{(l)}$ are node embeddings at layer l, with $H^{(0)} = X$ (the node feature matrix), and A is the adjacency matrix.

The adjacency matrix A is a key component of GNNs:

- Acts as a form of structural prior about the geometry of the domain.

- Enables the model to reflect spatial or structural constraints of the data.
- Helps reduce overfitting by enforcing locality, limiting message passing to meaningful connections and injecting inductive bias into the model, enabling better generalization.

The specific model differs only in how $f(\cdot)$ is chosen and parametrized. As an example we can consider the following very simple form of a layer-wise propagation rule

$$f(H^{(l)}, A) = \sigma(AH^{(l)}W^{(l)}) \tag{2.8}$$

where $W^{(l)}$ is the learnable weight matrix for the l-th neural network layer and $\sigma(\cdot)$ is a non-linear activation function like the ReLU. This rule has two limitations: multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself (unless there are self-loops in the graph). We can "fix" this by adding the identity matrix to A. The second major limitation is that A is typically not normalized and therefore the multiplication with A will completely change the scale of the feature vectors. Normalizing A such that all rows sum to one, i.e. $D^{-1}A$ where D is the diagonal node degree matrix, gets rid of this problem. Multiplying with $D^{-1}A$ now corresponds to taking the average of neighboring node features. In practice, dynamics get more interesting when we use a symmetric normalization, i.e. $D^{-1/2}AD^{-1/2}$ (as this no longer amounts to mere averaging of neighboring nodes). Combining these two tricks, we essentially arrive at the propagation rule introduced in Kipf & Welling (ICLR 2017) [8]:

$$f(H^{(l)}, A) = \sigma(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^{(l)}W^{(l)}) \tag{2.9}$$

where $\hat{A} = A + I$ and $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$[2]. In summary, a GCN layer aggregates information from the first-order neighborhood (1-hop neighbors). Stacking multiple layers enables the model to capture information from higher-order neighborhoods. However, GCNs also have limitations:

[2] `https://tkipf.github.io/graph-convolutional-networks/`

- Over-smoothing: node representations become indistinguishable as the number of layers increases.
- Aggregation: all neighbors contribute equally without attention or adaptive weighting.
- Limited expressivity: GCNs heavily depend on the relations encoded in the input adjacency matrix.

These limitations motivated Graph Attention Networks (GATs) [10].

2.2.2 Graph Attention Networks

Attention [14] is an operator that complements convolutions, fully connected layers, and pooling by allowing models to focus on the most relevant parts of the input. Attention mechanisms have become a de facto standard in sequence-based tasks. They handle variable-size inputs and emphasize the most relevant elements for each decision. When attention is computed within a single sequence, it is called self-attention. In many tasks, not all input elements are equally relevant for a given output. Attention addresses this by assigning dynamic weights to input elements based on context. Attention is computed as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.10}$$

Here, Q, K, and V denote queries, keys, and values: three linear projections of the input embedding. Intuitively, the query represents what a token is looking for, the key represents what it offers, and the value carries the information to pass on. In self-attention, queries, keys, and values all originate from the same input. This concept can be extended to graphs. The idea is to compute each node representation by attending over its neighbors, following a self-attention strategy [15]. As in GCNs, the graph attention layer first computes a linear transformation of node features. For attention, it uses the node's transformed feature as a query and its neighbors' transformed features as keys and values [3]. We now derive, step by step, the embedding produced by a GAT layer:

[3] https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial7/GNN_overview.html#Graph-Attention

1. Apply a shared linear transformation to all node features

$$h_i^W = Wh_i \ \forall \text{ node } i \tag{2.11}$$

2. Compute the unnormalized attention score between node i and its neighbor j:

$$e_{ij} = \text{LeakyReLU}(\mathbf{a}^T [h_i^W \parallel h_j^W]) \tag{2.12}$$

   where $\parallel$ denotes concatenation and **a** is a learnable weight vector

3. Compute normalized attention scores across all neighbors of i. The score e_{ij} indicates the importance of node j's features to node i.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(\rangle)} \exp(e_{ik})} \tag{2.13}$$

4. Compute the final embedding as a weighted sum

$$h_i' = \sigma\Big(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} h_j^W \Big) \tag{2.14}$$

To increase expressiveness, Velickovic et al. proposed extending GAT to multiple heads, similarly to Multi-Head Attention in Transformers. This applies K independent attention layers in parallel and concatenates their outputs:

$$h_i' = \Vert_{k=1}^{K} \sigma\Big(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(k)} W^{(k)} h_j \Big) \tag{2.15}$$

Brody et al. [2] later showed that standard GAT does not fully compute dynamic attention. In common formulations, the neighbor-scoring is static: for any query, ranking is monotonic with respect to per-node scores. As a result, GAT cannot express some simple alignment problems. To address this limitation, they proposed GATv2. By modifying the order of operations, GATv2 defines a strictly more expressive attention function. The core issue in standard GAT scoring is that the learned layers W and **a** are applied consecutively and can collapse into a single linear layer. In GATv2, this is avoided by applying **a** after the nonlinearity (LeakyReLU), with W applied after concatenation.

$$\text{GAT:} \quad e_{ij} = \text{LeakyReLU}(\mathbf{a}^T [Wh_i \parallel Wh_j]) \tag{2.16}$$

$$\text{GATv2:} \quad e_{ij} = \mathbf{a}^T \text{LeakyReLU}(W \cdot [h_i \parallel h_j]) \tag{2.17}$$

In the next sections, I describe the integration of GATv2 layers into the proposed GNN architecture. Their role will be further detailed in Chapter 4 and extended in Chapter 5, where they are used to enhance adjacency-aware message passing and improve embedding performance.

3. Neutral Atom Quantum Computing

According to Moore's Law, the number of transistors on a chip roughly doubles every two years. As a result, transistor size shrinks while transistor count increases at a regular pace, improving integrated-circuit functionality and performance while decreasing costs. Recently transistors are approaching a size such that the laws of quantum mechanics impact their functioning. Instead of treating quantum effects as a limitation, quantum computing seeks to exploit them. Indeed, there are extremely complex problems where traditional computers are limited, such as simulating molecules and chemical compounds for drug discovery or optimizing systems with millions of variables, as in logistics. To solve these problems, traditional computers may require months or years to reach an exact solution, so we are often forced to rely on approximate methods and accept trade-offs between accuracy and computation time. With the advent of quantum computers, however, this scenario could radically change[1].

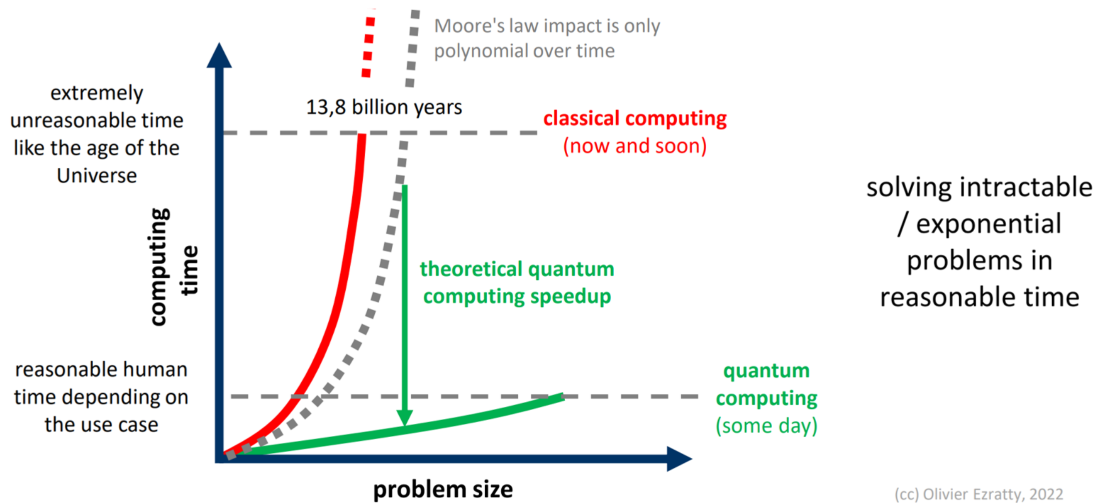

Figure 3.1: Classical and quantum scaling of computational tractability - from [3]

The goal is not only to solve currently tractable problems faster, but also to address problems that are currently intractable or would take impractically long times to solve with classical approaches.

[1] https://www.osservatori.net/blog/quantum-computing/quantum-computing-significato/

3.1 Quantum Computing Models

In QC, there is a wide range of approaches for turning different quantum systems into usable computers.

- Circuit (or gate) model: it operates in discrete steps, similarly to classical computing. Instead of classical logic gates acting on bits, quantum circuits use quantum gates acting on qubits. By arranging these gates in a specific sequence, one can implement a quantum algorithm and measure the qubits at the end to obtain the desired output.

- Adiabatic model: it takes advantage of a fundamental behavior in physics, namely that systems tend to evolve toward minimum-energy states. Adiabatic quantum computing frames a problem so that the system's lowest-energy state represents the solution.

- Annealing: not a universal quantum computing scheme but works on the same principle as adiabatic quantum computing, with the system finding the minimum energy state of an energy landscape that you give it. Quantum annealing can be viewed as a practical implementation or subset of the adiabatic approach. Adiabatic quantum computing is a universal model of quantum computation (that has been proven polynomially equivalent in power to the standard gate-based model). Quantum annealing, on the other hand, usually refers to methods and devices (like D-Wave's quantum processors) that perform this slow-evolution approach specifically for optimization problems[2].

- Measurement-based model: it uses intermediate measurements to drive the computation, rather than only extracting the final answer.

- Topological: information is stored and manipulated in topological invariants of two dimensional materials. The driving forces behind this form of quantum computation are anyons, particle-like defects in two dimensional materials which behave non-trivially when braided around one-another. In this approach, qubits are built from an entity in physics called a Majorana zero-mode quasi-particle, which is a type of non-abelian anyon. These quasi-particles

[2] https://postquantum.com/quantum-modalities/annealing-adiabatic/

are predicted to be more stable due to their physical separation from each other.

- Discrete Time Quantum Walk: quantum variation on a random walk, in which there is a 'walker' (or multiple 'walkers') which takes small steps in a graph (e.g. a chain of nodes, or a rectangular grid). The difference is that where a random walker takes a step in a randomly determined direction, a quantum walker takes a step in a direction determined by a quantum "coin" register, which at each step is "flipped" by a unitary transformation rather than changed by re-sampling a random variable.

3.2 Physical implementations of quantum computers

There are many physical implementations of quantum computers because many quantum systems can, in principle, be used to build them. Here are some of the most widely used approaches:

- Superconducting: superconducting qubits are currently the most popular approach. These qubits are made from superconducting wires with a break in the superconductor called a Josephson junction. The most popular type of superconducting qubit is called a transmon.
- Quantum Dot: Qubits are made from electrons or groups of electrons and the two-level system is encoded into the spin or charge of the electrons. These qubits are built from semiconductors like silicon.
- Linear Optical: Optical quantum computers use photons of light as the qubits and operate on these qubits using optical elements like mirrors, waveplates, and interferometers.
- Trapped Ion: Charged atoms are used as qubits, and these atoms are ionized, having a missing electron. The two-level state that encodes the qubit is the specific energy levels of the atom, which can be manipulated or measured with microwaves or laser beams.
- Color Center or Nitrogen Vacancy: These qubits are made from atoms embedded in materials like nitrogen in diamond or silicon carbide. They are created using the nuclear spins of the embedded atoms and are entangled together with electrons.
- Neutral Atoms in Optical Lattices: This approach captures neutral atoms into an optical

lattice, which is a crisscrossed arrangement of laser beams. The two-level system for the qubits can be the hyperfine energy level of the atom or excited states.

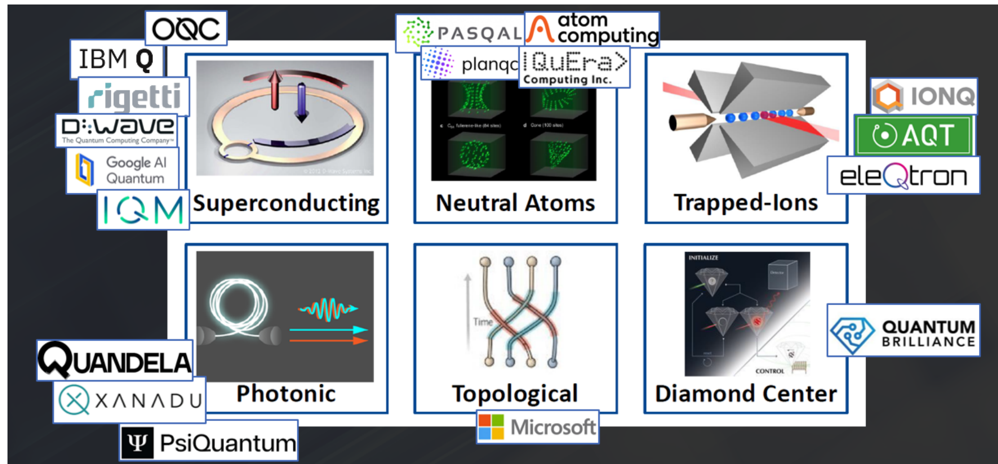

Figure 3.2: Physical implementations of quantum computers and principal companies. From "Anita Cammillini - Introduction to Quantum Computing - MHPC 2025"

These are some of the key approaches to building quantum computers, each with its own unique characteristics and challenges. Quantum computing is changing quickly, and picking the right approach is vital for future success.

3.3 Applications of quantum computers

Quantum computers have many potential applications. Examples include:

- Quantum simulation: quantum computers have the potential to simulate complex quantum systems, making it possible to study chemical reactions, electron behavior in materials, and other physical properties. Applications include better solar panels, batteries, drug development, and aerospace materials.
- Quantum algorithms: algorithms like Shor's Algorithm and Grover's Algorithm can solve problems that are thought to be intractable for classical computers. These have applications in cryptography, optimizing complex systems, machine learning, and AI.

- Cybersecurity: quantum computers pose a threat to classical encryption systems. However, they can also contribute to cybersecurity through the development of quantum-resistant encryption schemes. Quantum cryptography, a field related to quantum computing, can enhance security.
- Optimization problems: quantum computers can tackle complex optimization problems more efficiently than classical computers, with applications ranging from supply-chain management to financial modeling.
- Weather forecasting and climate change: quantum computers could potentially improve weather forecasting models and help address climate change-related challenges. This is an area that may benefit from quantum computing in the future.
- Quantum cryptography: it can strengthen data security by using quantum principles for secure communication.

Now we need to be a little careful about the potential of hype here, as a lot of the claims of what quantum computers will be good for come from people who are actively raising money to build them. Ideally, any physical device could implement every quantum-computing model and run every possible application, but this is not yet feasible. We are currently in the so-called NISQ era.

3.4 The NISQ era

NISQ, which stands for Noisy Intermediate-Scale Quantum, refers to the current generation of quantum computers. These devices have several tens to a few hundred qubits and are characterized by their ability to perform quantum operations, but with significant noise and errors that limit their capabilities. The term was coined by John Preskill in 2018 to describe the near-term quantum computing landscape [3]. The scientific community believes that NISQ quantum computers could outperform traditional classical computers for specific applications. In this era, quantum computing includes many categories of devices.

[3] http://quandela.com/resources/quantum-computing-glossary/nisq-noisy-intermediate-scale-quantum-computing/

- Classical computers: we include classical computers because they can implement:
  - Quantum-inspired computing: it uses classical algorithms on classical hardware that are inspired by quantum methods and can provide new efficiencies. It is not about emulating quantum code on a classical computer. Typical quantum-inspired algorithms use tensor-network libraries.
  - Quantum emulators: these execute quantum algorithms on traditional computers, from laptops to supercomputers depending on the number of qubits to emulate. They simulate quantum gates and qubits using large vectors and matrices, enabling algorithm testing without quantum hardware.
- Digital quantum computers: these are universal, gate-based quantum computers.
- Analog quantum computers: they implement a form of quantum computing that uses continuous variables to represent and manipulate quantum information. Unlike digital quantum computing, which relies on discrete qubits and gates, analog quantum computing operates on continuous quantum states and employs continuous transformations. It is a versatile approach with applications in quantum simulation, optimization, and more. There are two types of analog quantum computers:
  - Quantum annealing computers: they use slow, controlled evolution of qubits linked according to a chosen topology, following the adiabatic principle. The process is initialized in the ground state of an initial Hamiltonian, and the evolution drives the system toward a low-energy state, ideally the ground state of the target Hamiltonian. This technique is used to search for minima in complex optimization landscapes.
  - Analog quantum simulators: these simulate quantum phenomena without gate-based qubit systems. They operate in an analog way, so the parameters coupling qubits are continuous. At present, they are mainly laboratory tools. The most commonly used technique is based on neutral atoms cooled and controlled by lasers.

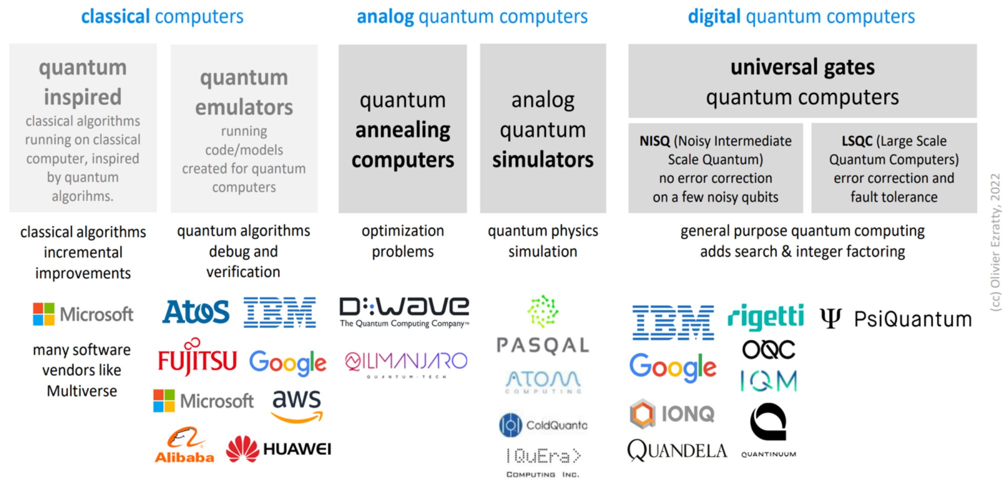

Figure 3.3: The different computing paradigms with quantum systems, hybrid systems and classical systems. From [3]

For this thesis, I focused on neutral-atom technologies, in particular the platform developed by Pasqal [4].

3.5 Quantum computing with neutral atoms

As discussed above, building a viable quantum processor requires investigating a broad range of physical platforms. Among them, arrays of single neutral atoms manipulated by light beams appear to be a powerful and scalable technology, enabling quantum registers with up to a few thousand qubits.

[4] https://community.pasqal.com/learning

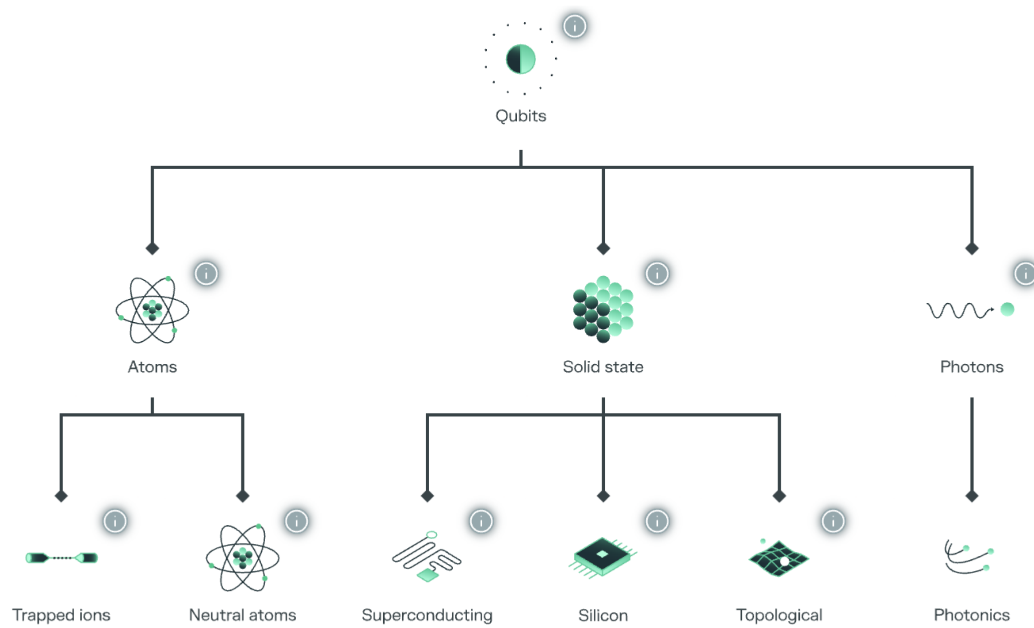

Figure 3.4: Qubit family tree - from `https://community.pasqal.com/learning/level-1/hardware-platforms/the-qubit-family-tree`

3.5.1 Qubits

A quantum bit, or qubit, is the basic unit of quantum information. A qubit can be prepared in a superposition of two basis states, usually labeled 0 and 1.

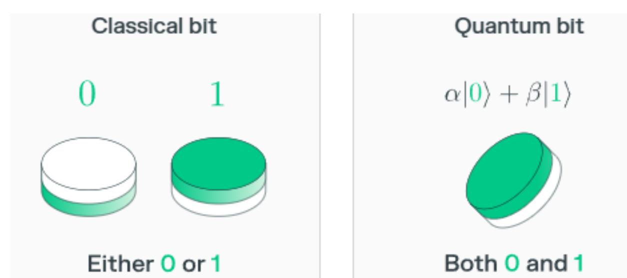

Figure 3.5: Bit vs. qubit - from `https://community.pasqal.com/learning/level-1/fundamental-principles/qubits-as-quantum-information-carriers`

Each basis state has an associated weight, or amplitude. In the infographic above, these are labeled alpha and beta. In general, amplitudes are complex numbers. When a qubit is measured, the

superposition is lost and the qubit collapses to either state 0 or state 1, i.e., to a classical state. Probabilities play a central role in quantum physics. The probability of obtaining 0 (or 1) upon measurement is given by the squared modulus of the amplitude associated with basis state 0 (or 1, respectively). For more than one qubit, the superposition principle extends naturally. For example, for $N = 3$ qubits, the register can be in a superposition of eight basis states (000, 001, 010, ..., 111), each with its own amplitude. This richer state space underpins the potential power of quantum computation. As before, the probability of obtaining a specific basis state in a joint measurement is given by the squared modulus of its corresponding amplitude.

Figure 3.6: N bits vs. N qubits - from https://community.pasqal.com/learning/level-1/fundamental-principles/qubits-as-quantum-information-carriers

3.5.2 Quantum algorithm workflow

Quantum algorithms usually follow a three-step process:

1. Initialization: each candidate solution is mapped to a basis state. For example, in the traveling salesman problem, one basis state can represent each possible route. In most cases, initialization prepares a superposition, often not uniform, over all basis states, or a relevant subset. At this stage, a measurement typically has only a small probability of returning the optimal solution.

2. Processing: a sequence of quantum operations is applied to modify the amplitudes of the basis states. Ideally, the amplitude associated with the optimal solution is amplified while the others are suppressed.

3. Readout: the qubits are measured to extract a solution, ideally with high probability for the optimal one.

3.5.3 Quantum register

Neutral atom quantum processors are based on configurable arrays of single neutral atoms. An array can be seen as a register, where each single atom plays the role of a qubit. The array is assembled using laser cooling and trapping techniques. A quantum register is the arrangement of qubits, initialized in the ground state, on which quantum processing is performed.

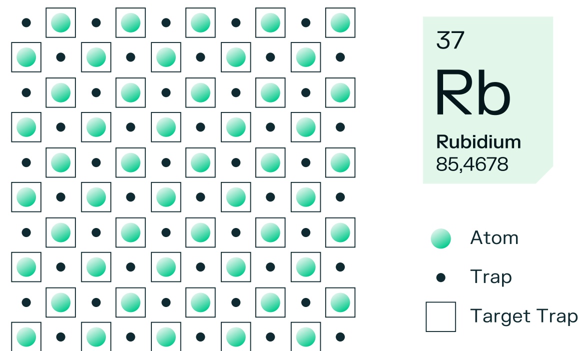

Figure 3.7: Quantum register - from `https://community.pasqal.com/learning/level-3/neutral-atom-arrays/overview`

One common choice here is rubidium, an alkali atom with one electron in the valency layer. Rubidium is a prevalent species in atomic physics with well-established technological solutions, especially for lasers. For instance, commercial rubidium atomic clocks enabled by grade lasers for rubidium transitions are designed to meet the latest commercial, military and aerospace requirements where time stability is a critical feature. In neutral atom based quantum computers, two electronic levels of the rubidium atoms are selected to serve as the two qubit states, denoted $|0\rangle$ and $|1\rangle$.

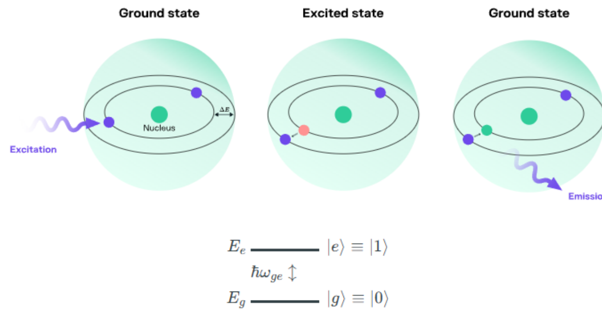

Figure 3.8: Two-level transition - from `https://community.pasqal.com/learning/level-3/neutral-atom-arrays/overview`

Each task requires a laser with a specific wavelength. Additionally, electronic controls are needed to tune the light properties, apply instructions from the quantum software stack, and extract information through atomic detection. Light is the main tool to control both the position and the quantum state of the atoms. It is used to:

- assemble and read out registers made of hundreds of qubits
- perform fully programmable quantum processing

3.5.4 Digital and analog modes

In theory, neutral-atom quantum processors (QPUs) can operate in either analog or digital mode. However, large-scale digital quantum computing remains a long-term goal because it requires further technological advances, including quantum error correction. In digital mode, qubit time evolution is described by quantum gates. In practice, strongly focused laser beams implement these gates by addressing individual atoms with resonant pulses.

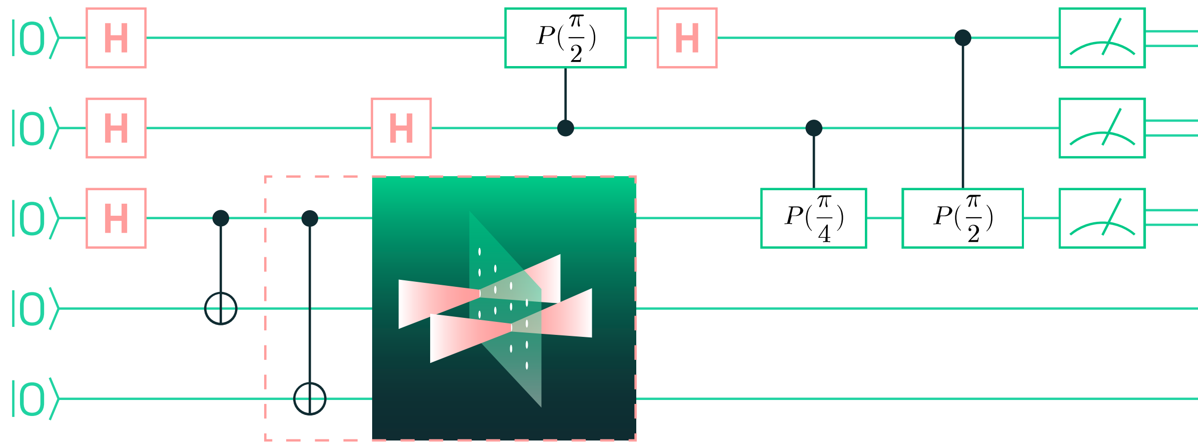

Figure 3.9: A succession of gates is applied to the qubits to implement a quantum algorithm. Each gate is performed by addressing the qubits individually with laser beams. From `https://community.pasqal.com/learning/level-3/neutral-atom-arrays/quantum-processing-with-atomic-qubits-a-preview`

In analog mode, the user directly manipulates the Hamiltonian describing the evolution of the atomic ensemble. Quantum processing is carried out by illuminating all atoms simultaneously with a single laser beam, allowing the system Hamiltonian to be used directly as a computational resource.

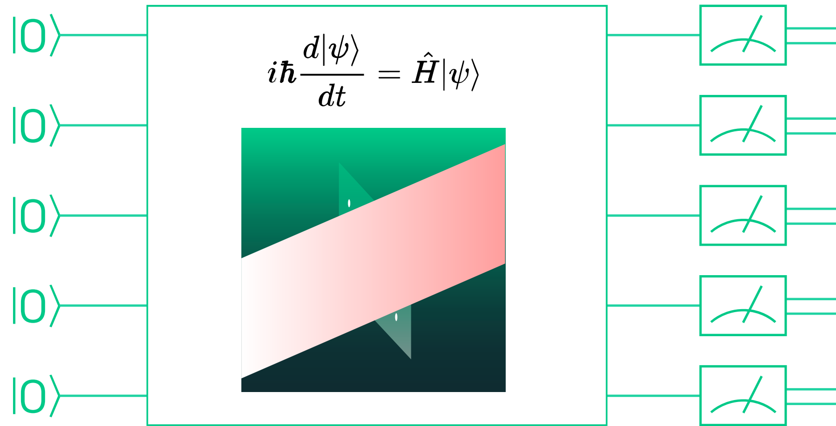

Figure 3.10: The qubits evolve under a tailored Hamiltonian H, for instance by illuminating the whole register with a laser beam. The wavefunction $|\psi\rangle$ of the system follows the Schrödinger equation. From `https://community.pasqal.com/learning/level-3/neutral-atom-arrays/quantum-processing-with-atomic-qubits-a-preview`

In classical physics, the Hamiltonian is a real scalar quantity corresponding to the total energy of a system. In quantum physics, it is a self-adjoint (Hermitian) operator whose eigenvalues correspond to the system's energy levels.

$$H = \begin{bmatrix} E_0 & & & \\ & E_1 & & \\ & & \ddots & \\ & & & E_N \end{bmatrix} = H^\dagger \tag{3.1}$$

$$E_0 \leq E_1 \leq \cdots \leq E_N$$

In quantum mechanics, knowing the system Hamiltonian $H(t)$ is essential because it determines the Schrödinger equation governing the system dynamics. The Schrödinger equation describes how the system will evolve or change in time.

3.5.5 Operating an atomic qubit register

Unlike in solid-state quantum processors (e.g., superconducting, silicon, or NV platforms), the register in an atomic QPU is not permanent and is reconstructed after each run. Hence, a typical computation cycle consists of three phases:

- Register preparation
- Quantum processing
- Register readout

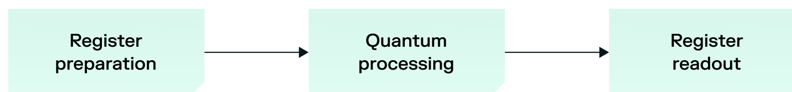

Figure 3.11: Computation cycle - from `https://community.pasqal.com/learning/level-3/neutral-atom-arrays/operating-an-atomic-qubit-register`

Register preparation

To prepare a register made of neutral atoms, one can use arrays of optical tweezers. Optical tweezers are highly focused laser beams at a specific wavelength used to generate dipole traps in which atoms can be loaded:

1. Firstly, a dilute atomic vapor is formed inside an ultra-high vacuum system operated at room temperature. Using an initial laser system, a cold ensemble of about one million atoms within a 1 mm^3 volume is prepared inside a 3D magneto-optical trap, leveraging various laser cooling and trapping techniques.

2. Then, a second trapping laser system isolates individual atoms. Using high-numerical-aperture lenses, the trapping beam is focused into multiple spots of about 1 μm in diameter, known as optical tweezers. Within a trapping volume of a few μm^3, each tweezer contains at most one atom at a time.

Before passing through the lens, the trapping beam is reflected onto a spatial light modulator (SLM), which imprints an adjustable phase pattern on the light.

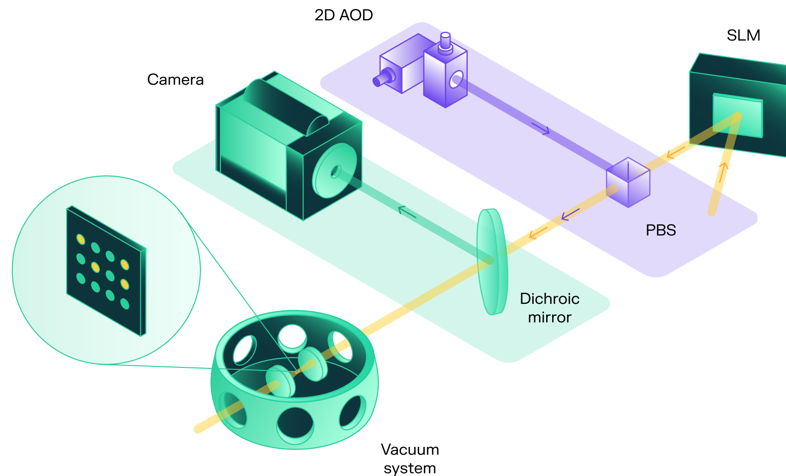

Figure 3.12: Overview of the main hardware components constituting a quantum processor: the trapping laser light (yellow) is shaped by the spatial light modulator (SLM) to produce multiple microtraps at the focal plane of the lens (see inset). The moving tweezers (purple), dedicated to rearranging the atoms in the register, are controlled by a 2D acousto-optic laser beam deflector (AOD) and superimposed on the main trapping beam with a polarizing beam-splitter (PBS). The fluorescence light (green) emitted by the atoms is separated from the trapping laser light by a dichroic mirror and collected onto a camera. From `https://community.pasqal.com/learning/level-3/neutral-atom-arrays/operating-an-atomic-qubit-register`

Sub-register rearranging

Each tweezer can host at most one atom, but this occurs in only about 50% of the cases; the rest of the time, the tweezer is empty. To detect which tweezers are filled, the atoms' fluorescence is collected and imaged onto a sensitive camera. The atoms are then moved from site to site in order to generate a pre-defined sub-register with unit filling. This operation is done using programmable moving optical tweezers, with a success rate above 99%. From analysis of the initial image, an algorithm computes on the fly the set of moves needed to rearrange the initial configuration into the desired fully assembled sub-register. To implement this active feedback within a few tens of

milliseconds, data are transferred through an FPGA and the algorithm runs on an external GPU. The moves are then communicated to the drivers of 2D acousto-optic deflectors (AODs), which control the pointing and intensity of the moving tweezers. After rearrangement, a second image confirms the new atomic positions in the sub-register.

Quantum processing and register readout

Once the register is fully assembled, quantum processing can start. Processing itself is extremely fast (less than 100 μs), while the overall sequence, including loading and readout, lasts approximately 200 μs. Once processing is done, the atomic register is read out by taking a final fluorescence image. It is performed such that each atom in qubit state $|0\rangle$ will appear as bright, whereas atoms in qubit state $|1\rangle$ remain dark. All images are acquired with an electron-multiplying charge-coupled device (EMCCD) camera, which converts fluorescence photons into electrons to produce a detectable signal with high sensitivity. Given the probabilistic nature of measurement outcomes in quantum mechanics, computation cycles are repeated many times to reconstruct the relevant statistical properties of the final quantum state produced by the algorithm.

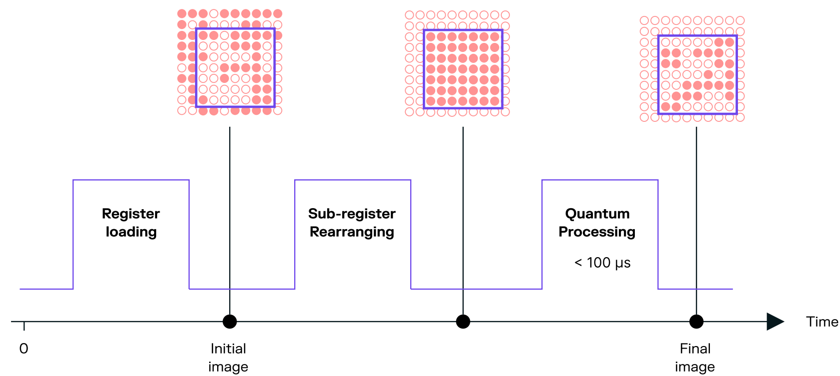

Figure 3.13: Temporal sequence of one computation cycle. The loading of the register being random, atoms are first rearranged to realize a defect-free sub-register, on which the quantum processing is performed. From `https://community.pasqal.com/learning/level-3/neutral-atom-arrays/overview`

3.5.6 Interacting atoms

In neutral-atom devices, atoms are driven to Rydberg states as a way to make them interact over large distances. A Rydberg state is a state of an atom or molecule in which one of the electrons has been excited to a high principal quantum number orbital. Classically, such a state corresponds to putting one electron into an orbit whose dimensions are very large compared to the size of the leftover ion core. Among the novel properties of these states are extreme sensitivity to external influences such as fields and collisions, extreme reactivity, and huge probabilities for interacting with microwave radiation. Think of an atom in a Rydberg state as a huge electric dipole, where the positive pole is the nucleus, and the negative pole is the electron in a Rydberg level.

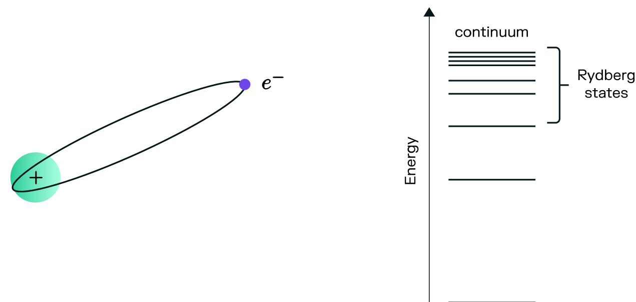

Figure 3.14: A dipole can be engineered by exciting an atom to a Rydberg level - from `https://community.pasqal.com/learning/level-3/interacting-atoms/rydberg-states`

The energy levels of two Rydberg atoms strongly interact when their distance is smaller than a characteristic threshold called the blockade radius.

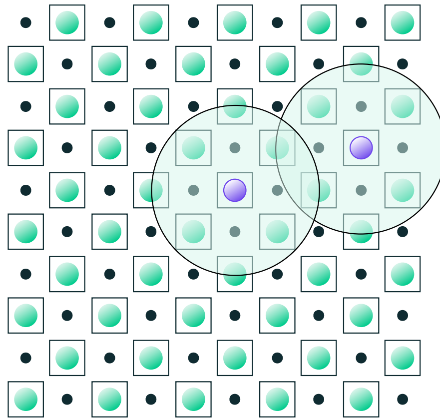

Figure 3.15: Illustration of the dipole-dipole interaction in between two atoms in a neutral-atom register. The circles illustrate the range of the interaction. Two atoms strongly interact when their circles intersect. From `https://community.pasqal.com/learning/level-3/interacting-atoms/rydberg-states`

Rydberg blockade

The interaction between two atoms at distance R and at the same Rydberg level is described by the van der Waals force, which scales as R^{-6}. In physics and chemistry, the van der Waals force is a distance-dependent interaction between atoms or molecules. This force quickly vanishes at longer distances between interacting systems. The size of the blockade radius R_b, among other things, depends on the energy of the Rydberg state (through the C_6 constant) and the intensity of the exciting lasers ($\hbar\Omega$):

$$R_b = \left(\frac{C_6}{\hbar\Omega}\right)^{1/6} \tag{3.2}$$

In digital implementations, the atom-atom interaction can be exploited to create fast and robust two-qubit gates, using the so-called Rydberg Blockade effect between them.

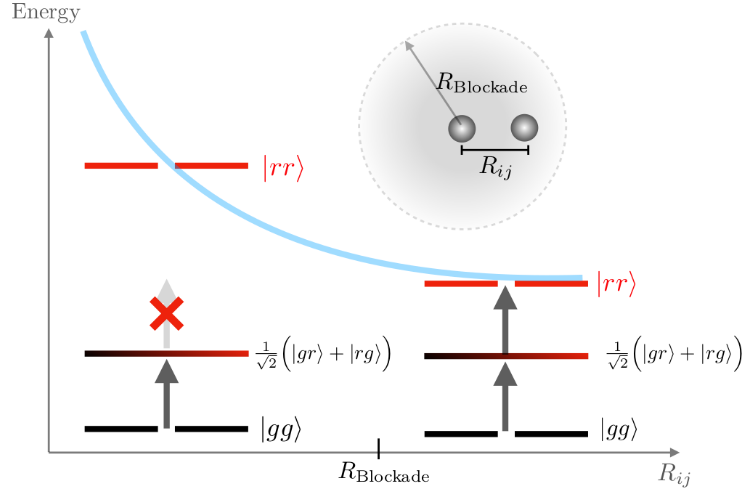

Figure 3.16: The Rydberg blockade: The energy of the $|rr\rangle$ state (blue line) increases signficantly for atoms less than a blockade radius ($R_{Blockade}$ in this picture) away. As such, the transition to $|rr\rangle$ is suppressed when $R_{ij} << R_{Blockade}$. Source: Quantum 6, 629 (2022)

Ising Hamiltonians

Because of the Rydberg blockade, an atom cannot be excited to a Rydberg level if a nearby atom is already excited. The interaction adds a penalty term to the drive Hamiltonian describing the transition between the ground state $|g\rangle$ and the Rydberg state $|r\rangle$:

$$U_{ij} n_i n_j$$

where $n = (I + Z)/2$ is the projector on the Rydberg state, U_{ij} is proportional to R_{ij}^{-6} where R_{ij} is the distance between the atoms i and j. The proportionality constant is specific to the chosen Rydberg level. A pulse-driven transition between two energy levels can be mapped to a qubit system i through the drive Hamiltonian:

$$\frac{H_i^D(t)}{\hbar} = \frac{\Omega_i(t)}{2} cos(\phi_i) X_i - \frac{\Omega_i(t)}{2} sin(\phi_i) Y_i - \frac{\delta_i(t)}{2} Z_i \tag{3.3}$$

where $\Omega(t)$ is the Rabi frequency (or amplitude), $\delta(t)$ the detuning and ϕ a fixed phase. An entire

array of interacting atoms, acted on by the same pulse with zero phase, can be represented as an Ising-like Hamiltonian:

$$H(t) = \sum_i \left(H_i^D(t) + \sum_{j<i} U_{ij} n_i n_j \right) \tag{3.4}$$

$$H(t) = \sum_i \left(\frac{\hbar\Omega_i(t)}{2} X_i - \frac{\hbar\delta_i(t)}{2} Z_i + \sum_{i<j} U_{ij} n_i n_j \right) \tag{3.5}$$

Ising Hamiltonians are very common Hamiltonians in physics. Beyond neutral atoms, they are used to describe spin lattices in statistical physics. Through the continuous manipulation of $\Omega(t)$ and $\delta(t)$ one has a very high degree of control over the system's dynamics and properties. This way, the analog approach enables the quantum simulation of many-body quantum systems, but also provides novel ways of solving combinatorial problems that can be mapped onto the Hamiltonian above.

3.5.7 Applications of a neutral atom QPU

The MIS problem

A graph $G(V, E)$ is a structure consisting of a set of objects where some pairs of the objects are in some sense "related". The objects are represented by abstractions called vertices V and each of the related pairs of vertices is called an edge E. Typically, a graph is depicted in diagrammatic form as a set of dots or circles for the vertices, joined by lines or curves for the edges. A key advantage of neutral-atom platforms is the ability to reconfigure qubit-register geometry from shot to shot. This feature enables native hardware-level embedding of graph-structured problems, with important implications for complex optimization and machine-learning tasks. In graph theory, an adjacency matrix is a square matrix used to represent a finite graph $G(V, E)$. The elements of the adjacency matrix are either 1 or 0, indicating whether pairs of vertices are connected by an edge or not. When atoms are promoted to Rydberg states, they exhibit two-body interactions derived from van der Waals forces. By tuning pairwise distances, one can program interaction strengths and engineer an Ising Hamiltonian of the form:

$$H_{\mathrm{Ising}}(t)/\hbar = \frac{\Omega_i(t)}{2}\sum_i X_i - \delta(t)\sum_i n_i + \sum_{i<j}\frac{C_6/\hbar}{R_{ij}^6} n_i n_j \tag{3.6}$$

that embeds the adjacency matrix of a graph $G(V,E)$. This is illustrated in Fig. 3.17, where the Ising Hamiltonian of a specially designed neutral-atom register closely reproduces the adjacency matrix of a given graph $G(V,E)$. The interaction term (the last sum) is symmetric, just like the adjacency matrix of an undirected graph.

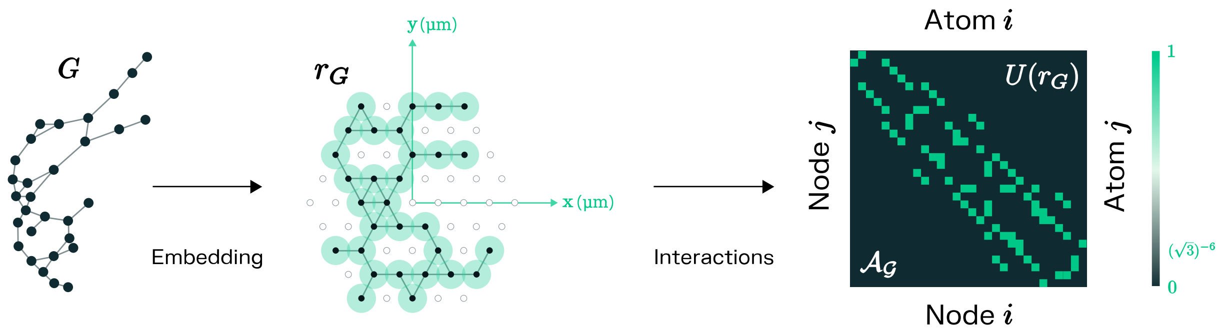

Figure 3.17: In an adequate spatial layout of the atoms, the interactions can reproduce the adjacency matrix A_G of an input graph $G(V,E)$. The sharp decay of the interaction allows neglecting terms between second (and further) nearest neighbours. Source: Eur. Phys. J. A 60, 177 (2024). https://doi.org/10.1140/epja/s10050-024-01385-5

This encoding works only when the graph is a unit-disk (UD) graph, i.e., when vertex positions exist such that two vertices are connected if and only if their distance is smaller than a given threshold, here taken as the blockade radius.

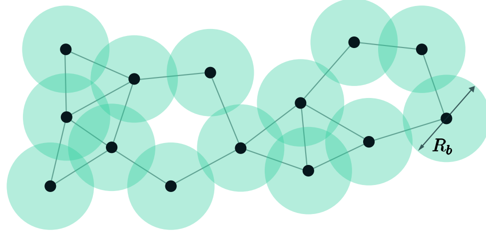

Figure 3.18: The disks in the unit disk graph have diameter R_b (the blockade radius) and there is an edge between two vertices if and only if their corresponding disks intersect. From `https://community.pasqal.com/learning/level-3/analog-quantum-computing-with-neutral-atoms/analog-quantum-computing-with-neutral-atoms`

The native Ising Hamiltonian obtained in neutral atom platforms can be harnessed for tackling computational problems on graphs. It relies on the continuous evolution of the quantum state from an initial state $|\psi_0\rangle$ to:

$$|\psi(t)\rangle = \mathcal{T}\exp\left(-\frac{1}{\hbar}\int_{s=0}^{t} H_{\text{Ising}}(s)d(s)\right)|\psi_0\rangle \tag{3.7}$$

where $\mathcal{T}$ denotes the time-ordering operator, required for time-dependent Hamiltonians. One can thus encode the graph properties and use the evolution to produce graph-dependent quantum states, subsequently employed as resources for a variety of computing tasks. A few years ago, researchers showed that the Rydberg blockade effect can be leveraged to tackle the well-known NP-hard Maximum Independent Set (MIS) problem on UD graphs. Given a graph, the MIS problem consists of finding the largest subset of vertices that do not share an edge. Using binary variables (0/1), it can be formulated as follows:

$$\text{Minimize} \qquad C(\vec{x}) = -\sum_{i\in V} x_i + U\sum_{(i,j)\in E} x_i x_j \tag{3.8}$$

$$\text{such that} \qquad x_i = \begin{cases} 1 & \text{if vertex } i \text{ belongs to the candidate set} \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

where the linear term rewards adding vertices to the candidate set (expected to become the MIS after optimization), while the quadratic term penalizes adding adjacent vertices, which would violate independence. The penalty weight is controlled by the scalar $U > 0$. The UD-MIS problem is an example of a Quadratic Unconstrained Binary Optimization (QUBO) problem. The native use of interacting neutral atoms for UD-MIS stems from the fact that the Rydberg blockade restricts qubit dynamics to coherent superpositions of independent sets. Setting $\Omega = 0$ and choosing $\delta > 0$ enables mapping the ground state of the neutral-atom Ising Hamiltonian to the UD-MIS solution:

$$H_{\text{Ising}}(t)/\hbar = \frac{\Omega_i(t)}{2}\sum_i X_i \boxed{-\delta(t)\sum_i n_i + \sum_{i<j}\frac{C_6/\hbar}{R_{ij}^6} n_i n_j}$$

$$\Updownarrow$$

$$C(\vec{x}) = -\sum_{i \in V} x_i + U \sum_{(i,j)\in E} x_i x_j$$

where $n = (I + Z)/2$ is the occupancy operator for the Rydberg level. This operator can be viewed as the quantum analogue of a binary variable: for a qubit in the Rydberg state, its eigenvalue is 1, and 0 otherwise. One can then use a system of driven, interacting neutral atoms to approximately solve UD-MIS using quantum adiabatic annealing (QAA) or an analog version of the quantum approximate optimization algorithm (QAOA).

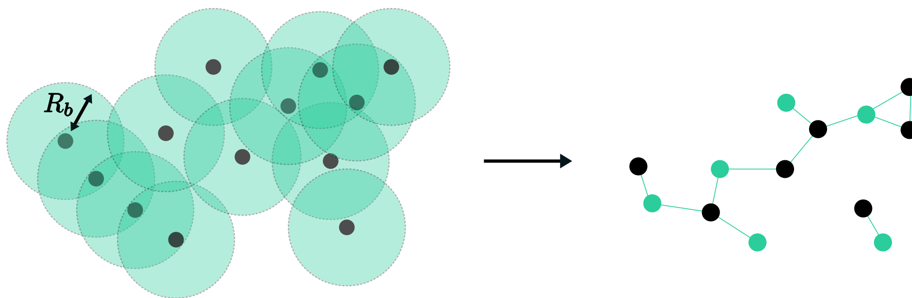

Figure 3.19: From a spatial configuration of Rydberg atoms to a unit-disk MIS graph problem. Source: Quantum 4, 327 (2020). `https://doi.org/10.22331/q-2020-09-21-327`

Figure 3.19 shows an example atomic register in which atom positions are chosen to match the

target graph directly. As noted above, two Rydberg atoms strongly interact when they are closer than the blockade radius, preventing both from being simultaneously excited to the Rydberg state. This naturally enforces the independent-set constraint in the graph defined by the atoms, where edges connect atoms separated by less than the blockade radius. The original graph is shown on the right, with black vertices forming one independent set. The ability to solve the MIS problem has broad implications, extending to various other graph-related problems, given their connection in graph theory.

Beyond MIS

The UD-MIS problem can be generalized to the UD Maximum Weight Independent Set (UD-MWIS), where nodes carry weights. These weights add complexity and can make instances harder to solve. Site-dependent detuning, for example via a DMM channel, facilitates encoding node-specific weights into the Hamiltonian. This provides a mechanism to encode UD-MWIS instances. More generally, advances in local controls and the use of ancillary atomic chains have expanded the native UD-MIS capabilities of neutral atoms to tackle optimization problems in higher complexity classes. These results open the door to encoding any unweighted QUBO problem into a UD-MWIS problem. Examples include the following problems and applications:

- Maximum Clique (Financial portfolio optimization, Social networks)
- Minimum Vertex Cover (Network security, Financial networks, Economic analysis)
- Minimum Dominating Set (Telecom networks, Document summarization)
- Graph Coloring (Task scheduling, Map labelling, Wireless channel allocation)

In this chapter, we introduced the fundamental concepts of quantum computing, with particular emphasis on neutral-atom platforms and the constraints that characterize NISQ devices. These constraints naturally lead to the formulation of computational tasks as combinatorial optimization problems defined on graphs. Among these, the Maximum Independent Set (MIS) problem emerges as a central primitive, both because of its relevance in encoding a wide range of applications and because of its natural compatibility with the native interactions of neutral-atom systems. In

this framework, the problem Hamiltonian can be directly mapped onto the physical interactions between atoms, making MIS a particularly suitable target for implementation on such devices. This perspective will be central in the remainder of this thesis. In the following chapters, we will see how the construction and embedding of problem graphs, specifically in the context of Quantum Molecular Docking, can be reduced to instances of MIS, and how learning-based approaches can be used to generate hardware-compatible embeddings that enable their execution on neutral-atom quantum processors.

4. GEAN: Graph Embedding Autoencoder Network

After studying the theory behind machine learning and neutral-atom quantum computing, I moved on to the analysis of an existing neural network [16] that solves a class of optimization problems and could be a good starting point for my work. Chiara Vercellino et al. presented their paper at the 2022 IEEE International Conference on Quantum Computing and Engineering (QCE), addressing the embedding of interaction graphs under the physical connectivity constraints of neutral-atom quantum simulators. Instead of relying only on combinatorial solvers, they proposed a neural approach, the Graph Embedding Autoencoder Network (GEAN), which learns feasible qubit placements in a bounded 2D (and later 3D) domain while enforcing minimum-distance and adjacency/non-adjacency constraints. They discussed the physical motivation, formalized the constrained unit-disk graph embedding problem, described the GEAN architecture and loss design, and reported results on antenna, protein-folding, and graph-coloring instances. Their experimental evidence showed that GEAN could quickly identify feasible embeddings in cases where classical optimization might fail within a fixed time budget.

4.1 Motivation

The NISQ era is characterized by a limited number of reliable qubits, finite coherence times, and non-negligible gate errors. These limitations make hardware-aware modeling essential: a solution for an optimization problem may be theoretically valid, yet difficult to execute if its interaction graph cannot be realized by the target architecture. For this reason, hybrid approaches and specialized platforms such as quantum annealers and neutral-atom simulators have attracted increasing interest. The focus is on neutral-atom devices, where qubits are represented by atoms positioned in space and coupled through a blockade mechanism. In this context, solving an optimization problem is not only a matter of objective quality, but also of geometric realizability: graph nodes must be mapped to coordinates so that distances reflect desired edges and non-edges. The practical challenge is therefore an embedding problem under hard physical constraints. GEAN was designed precisely to

automate this mapping.

4.2 Unit disk graph embedding under hardware constraints

They considered a register where qubits are placed in a circular domain of radius 50 μm, implying pairwise distances bounded by 100 μm. Let $\mathcal{V}$ be the vertex set and $\mathcal{E}$ the edge set of the target graph. Let d_{ij} denote the Euclidean distance between nodes i and j. Feasibility requires:

$$d_{ij} \leq 100\ \mu\text{m} \quad \forall i, j \in \mathcal{V}, i < j \qquad \text{(Domain limit)}$$

$$d_{ij} \geq 4\ \mu\text{m} \quad \forall i, j \in \mathcal{V}, i < j \qquad \text{(Minimum distance)}$$

$$d_{ij} \leq \tilde{r}_b \quad \forall (i, j) \in \mathcal{E} \qquad \text{(Adjacent pairs)}$$

$$d_{hk} > \tilde{r}_b \quad \forall (h, k) \in \bar{\mathcal{E}} \qquad \text{(Non-adjacent pairs)}$$

Here $\tilde{r}_b = 10.26\mu m$ is the maximum blockade radius, which depends on the coherence-time limit $\tilde{t}$ of the machine (Pasqal's R&D prototype, Chadoq2). This formulation naturally captures the unit disk graph condition: adjacent vertices must lie within an interaction radius, while non-adjacent vertices must remain beyond it. The difficulty lies in satisfying all constraints simultaneously on dense or irregular graphs.

4.3 GEAN Methodology

4.3.1 Architecture

GEAN combines an autoencoder and deterministic distance-computation layers.

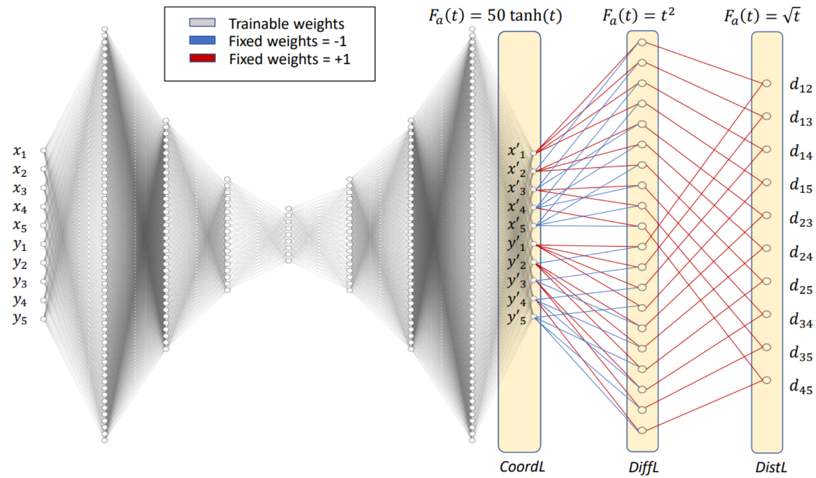

Figure 4.1: GEAN - from [16]

In the reference configuration, the autoencoder has shape $64 \rightarrow 36 \rightarrow 18 \rightarrow 9 \rightarrow 18 \rightarrow 36 \rightarrow 64$ and transforms initial coordinates (x_i, y_i) into refined coordinates (x'_i, y'_i) through ReLU activations with dropout 0.5. The coordinate layer uses a bounded activation, $F_a(t) = 50\tanh(t)$, to keep positions in a physically meaningful range.

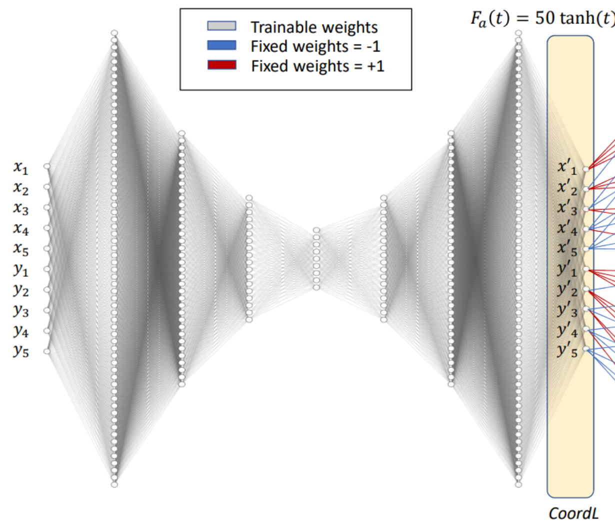

Figure 4.2: GEAN Autoencoder - from [16]

After this geometric refinement, GEAN computes pairwise differences and distances through non-trainable layers.

A difference layer (with fixed weights ±1) produces coordinate offsets using $F_a(t) = t^2$:

- $x'_i - x'_j \quad i, j \in \mathcal{V} \quad i < j$
- $y'_i - y'_j \quad i, j \in \mathcal{V} \quad i < j$

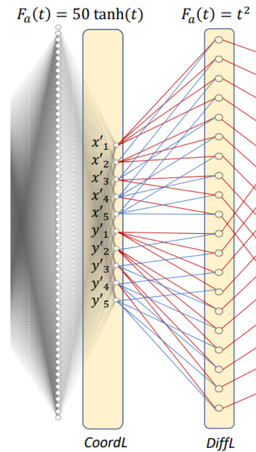

Figure 4.3: GEAN difference layer layer - from [16]

Then a distance layer (fixed weights 1 and activation $F_a(t) = \sqrt{t}$) returns $d_{ij} = \sqrt{(x'_i - x'_j)^2 + (y'_i - y'_j)^2}$.

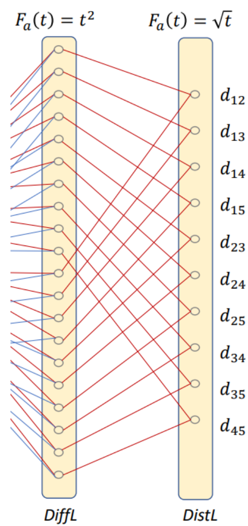

Figure 4.4: GEAN - distance layer - from [16]

This design separates representation learning from exact geometric evaluation.

4.3.2 Loss Function and Optimization

Training is guided by four terms, each matching one class of physical constraints:

$$\text{loss}_1(d) = \operatorname*{avg}_{i<j\in\mathcal{V}} |\max(100, d_{ij}) - 100| \qquad \text{(Domain constraint)}$$

$$\text{loss}_2(d) = \operatorname*{avg}_{i<j\in\mathcal{V}} |\min(4, d_{ij}) - 4| \qquad \text{(Minimum distance)}$$

$$\text{loss}_3(d) = \operatorname*{avg}_{i<j\in\mathcal{V}} A_{ij}|\max(\tilde{r}_b, d_{ij}) - \tilde{r}_b| \qquad \text{(Adjacent pairs)}$$

$$\text{loss}_4(d) = \operatorname*{avg}_{i<j\in\mathcal{V}} (1 - A_{ij})|\min(\tilde{r}_b + \epsilon, d_{ij}) - (\tilde{r}_b + \epsilon)| \qquad \text{(Non-adjacent)}$$

with total objective

$$\text{loss}(d) = \text{loss}_1 + \text{loss}_2 + \text{loss}_3 + \text{loss}_4. \tag{4.1}$$

Optimization uses AdamW with learning rate 10^{-3} for up to 5000 epochs.

4.4 Experimental Results

4.4.1 Problem Families

They evaluated GEAN on three graph classes. The first is a maximum independent set setting on real antenna layouts in Turin, where conflict distance induces graph adjacency. The second is protein-folding instances, initialized through Fruchterman–Reingold placement. The third is graph coloring instances, also using force-directed initialization.

4.4.2 Comparison with Classical Solver

Table 4.1 summarizes representative outcomes against Gurobi with a 2-minute limit. The key trend is that GEAN remained effective on larger or harder instances where Gurobi might not return a feasible embedding in time.

Instance	$\|V\|$	$d_{\max}$	$\|K_{\max}\|$	GEAN epochs	Gurobi (2 min)
G_a (Antennas)	13	8	5	733	Yes
G_c (Antennas)	6	4	4	384	Yes
$G_{12,6}$ (Protein)	5	4	3	187	Yes
$G_{17,7}$ (Protein)	10	9	4	492	No
$G_{22,8}$ (Protein)	9	7	4	404	Yes
3D Antennas	87	14	10	2120	No
3D Graph Coloring	21	6	3	767	No

Table 4.1: Representative feasibility comparison between GEAN and Gurobi.

A detailed example is protein instance $G_{17,7}$.

- $r_{min} = 11.601 \rightarrow 4.056\ \mu$m
- $r_{max} = 64.463 \rightarrow 20.237\ \mu$m
- $r_{adj} = 32.719 \rightarrow 10.255\ \mu$m
- $r_{\overline{adj}} = 22.277 \rightarrow 10.606\ \mu$m

Before GEAN, non-adjacent and adjacent distance statistics overlap in ways that violate unit-disk separability. After optimization, minimum and maximum distances are physically valid, and the separation between edge and non-edge distances is restored. In this case, the largest loss contributions come from adjacency-related terms (loss_3 and loss_4), highlighting that edge-consistency is the most demanding part of training.

4.4.3 3D Extension

To overcome geometric limits in 2D (e.g., clique and degree bottlenecks), GEAN was extended to 3D by changing input/output layers from $2n$ to $3n$. This extension enabled embeddings that were not achievable in planar layouts. For instance, an 87-qubit antenna problem with conflict distance $D_c = 250\,m$ became tractable, converging in 2120 epochs with substantial improvement in

minimum-distance feasibility.

4.5 Replication

In this section, I describe the first practical part of my thesis: implementing GEAN in Python, specifically with the PyTorch library[1].

4.5.1 Utils

To implement GEAN, I first created a utility file containing methods used throughout the project.

CSV to coordinate converter

This utility loads node positions from a CSV file in which each row represents one (x, y) coordinate pair. Input is the CSV path; output is a tensor of shape (#nodes, 2).

Code 4.1: CSV to coordinate converter

```
def csv_to_pos(path: str) -> torch.Tensor:
    pos = []
    with open(path, 'r') as f:
        reader = csv.reader(f)
        for row in reader:
            if len(row) == 2:
                x = float(row[0])
                y = float(row[1])
                pos.append([x, y])
    return torch.tensor(pos, dtype=torch.float32)
```

Coordinate-to-graph converter and vice versa

This utility creates a conflict graph from node positions and a conflict distance. Two antennas are connected if their distance is below the conflict threshold. Inputs are a tensor of shape (#nodes, 2) and the conflict distance; output is a NetworkX [2] graph with edges between conflicting nodes.

[1] https://pytorch.org/
[2] https://networkx.org/

Code 4.2: Coordinate-to-graph converter

```
def pos_to_graph(pos: torch.Tensor, cdist: float) -> nx.Graph:
    n_nodes = len(pos)
    G = nx.Graph()
    for i in range(n_nodes):
        G.add_node(i, pos=pos[i])
    for i in range(n_nodes):
        for j in range(i + 1, n_nodes):
            dist=torch.sqrt((pos[i,0]-pos[j,0])**2+(pos[i,1]-pos[j,1])**2)
            if dist < cdist:
                G.add_edge(i, j)
    return G
```

The inverse method extracts node positions from a NetworkX graph and returns a tensor of shape (#nodes, 2).

Code 4.3: Graph-to-coordinate converter

```
def graph_to_pos(G: nx.Graph) -> torch.Tensor:
    nodes = sorted(G.nodes())
    pos = []
        for node in nodes:
            pos.append(G.nodes[node]['pos'])
    return torch.stack(pos)
```

Connected components graph extractor

This utility extracts conflict subgraphs. Inputs are a graph and the minimum number of nodes required for a component to be considered; output is a list of NetworkX subgraphs.

Code 4.4: Connected components graph extractor

```
def subgraphs(G: nx.Graph, min_nodes: int = 3) -> list[nx.Graph]:
    connected_components = list(nx.connected_components(G))
    subgraphs = []
    for component in connected_components:
        nodes_list = sorted(list(component))
```

```
        n_nodes = len(nodes_list)
        if n_nodes >= min_nodes:
            subgraph = G.subgraph(nodes_list).copy()
            subgraphs.append(subgraph)
    return subgraphs
```

Graph to adjacency matrix converter

It converts a NetworkX graph into an adjacency mask for ordered pairs.

Code 4.5: Graph to adjacency matrix converter

```
def graph_to_adj(G: nx.Graph) -> torch.Tensor:
    nodes = sorted(G.nodes())
    n_points = len(nodes)
    m = n_points * (n_points - 1) // 2
    adj = torch.zeros(m)
    idx = 0
    for i in range(n_points):
        for j in range(i + 1, n_points):
            if G.has_edge(nodes[i], nodes[j]):
                adj[idx] = 1
            idx += 1
    return adj
```

Visualization and comparison of graphs

It visualizes a 2D graph with nodes and edges from the adjacency vector. Inputs are a tensor of shape (#nodes, 2) and the adjacency vector for ordered pairs. The implementation uses Matplotlib[3].

Code 4.6: Visualization and comparison of graphs

```
def visualize(pos: torch.Tensor, adj: torch.Tensor) -> None:
    pos = pos.detach().cpu().numpy()
    adj = adj.detach().cpu().numpy()
    n_nodes = len(pos)
```

[3] https://matplotlib.org/

```
5       plt.figure(figsize=(10, 10))
6       idx = 0
7       for i in range(n_nodes):
8           for j in range(i + 1, n_nodes):
9               if adj[idx] == 1:
10                  plt.plot([pos[i, 0], pos[j, 0]],
11                          [pos[i, 1], pos[j, 1]],
12                          'b-', alpha=0.5, linewidth=1)
13              idx += 1
14      plt.scatter(pos[:, 0], pos[:, 1], c='red', s=100, zorder=5)
15      for i in range(n_nodes):
16          plt.text(pos[i, 0], pos[i, 1], str(i),
17                  fontsize=8, ha='center', va='center')
18      plt.xlabel('X')
19      plt.ylabel('Y')
20      plt.title('Graph Visualization')
21      plt.grid(True, alpha=0.3)
22      plt.axis('equal')
23      plt.show()
```

A companion function visualizes two graphs side by side for comparison. It takes two tensors of shape (#nodes, 2) and a shared adjacency vector for ordered pairs.

Code 4.7: Side by side comparison of graphs

```
1   def compare(pos1: torch.Tensor, pos2: torch.Tensor, adj: torch.Tensor) -> None
        :
2       pos1 = pos1.detach().cpu().numpy()
3       pos2 = pos2.detach().cpu().numpy()
4       adj = adj.detach().cpu().numpy()
5       n_nodes = len(pos1)
6       fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
7       idx = 0
8       for i in range(n_nodes):
9           for j in range(i + 1, n_nodes):
10              if adj[idx] == 1:
11                  ax1.plot([pos1[i, 0], pos1[j, 0]],
```

```
                        [pos1[i, 1], pos1[j, 1]],
                        'b-', alpha=0.5, linewidth=1)
            idx += 1
    ax1.scatter(pos1[:, 0], pos1[:, 1], c='red', s=100, zorder=5)
    for i in range(n_nodes):
        ax1.text(pos1[i, 0], pos1[i, 1], str(i),
                fontsize=8, ha='center', va='center')
    ax1.set_xlabel('X')
    ax1.set_ylabel('Y')
    ax1.set_title('Original Graph')
    ax1.grid(True, alpha=0.3)
    ax1.axis('equal')
    idx = 0
    for i in range(n_nodes):
        for j in range(i + 1, n_nodes):
            if adj[idx] == 1:
                ax2.plot([pos2[i, 0], pos2[j, 0]],
                        [pos2[i, 1], pos2[j, 1]],
                        'b-', alpha=0.5, linewidth=1)
            idx += 1
    ax2.scatter(pos2[:, 0], pos2[:, 1], c='green', s=100, zorder=5)
    for i in range(n_nodes):
        ax2.text(pos2[i, 0], pos2[i, 1], str(i),
                fontsize=8, ha='center', va='center')
    ax2.set_xlabel('X')
    ax2.set_ylabel('Y')
    ax2.set_title('Optimized Graph')
    ax2.grid(True, alpha=0.3)
    ax2.axis('equal')
    plt.tight_layout()
    plt.show()
```

Finally, a utility function prints compact graph checksums for quick comparison across machines. It takes a tensor of shape (#nodes, 2), the adjacency vector for ordered pairs, and an optional graph name.

Code 4.8: Compact graph checksums

```
def print_graph(pos: torch.Tensor, adj: torch.Tensor, name: str = "Graph") ->
    None:
     print(f"{name}: nodes={len(pos)}, edges={int(adj.sum())}, pos_sum={pos.sum
        ().item():.6f}, adj_sum={adj.sum().item():.0f}")
```

Violated constraint checker

It checks whether a graph satisfies all constraints and returns the total number of violations. Inputs are a tensor of shape $(\#\text{nodes}, 2)$, the adjacency vector for ordered pairs, the Rydberg radius, an epsilon margin for the non-adjacent constraint, and a domain radius (domain $[-r, r]$).

Code 4.9: Violated constraint checker

```
def check_constraints(pos: torch.Tensor, adj: torch.Tensor, rb: float = 10.26,
     eps: float = 0.1, r: float = 50.0) -> int:
        n_nodes = len(pos)
    violations = 0
    domain_violations = torch.sum((pos < -r) | (pos > r))
    violations += domain_violations.item()
    idx = 0
    for i in range(n_nodes):
        for j in range(i + 1, n_nodes):
            dist = torch.sqrt((pos[i, 0] - pos[j, 0])**2 + (pos[i, 1] - pos[j,
                1])**2)
            if dist < 4:
                violations += 1
            if adj[idx] == 1:
                if dist > rb:
                    violations += 1
                else:
                    if dist < rb + eps:
                        violations += 1
                idx += 1
        return violations
```

Saving and loading graphs

It saves graph data (positions and adjacency) to a `.pt` file. Inputs are a tensor of shape (#nodes, 2), the adjacency vector for ordered pairs, and the output path.

Code 4.10: Saving a graph

```
def save_graph(pos: torch.Tensor, adj: torch.Tensor, filepath: str) -> None:
    graph_data = {
        'pos': pos,
        'adj': adj
    }
    torch.save(graph_data, filepath)
    print(f"Graph saved to {filepath}")
```

The complementary function loads graph data from a `.pt` file. It takes the file path and returns the position tensor and adjacency vector.

Code 4.11: Loading a graph

```
def load_graph(filepath: str) -> tuple[torch.Tensor, torch.Tensor]:
    graph_data = torch.load(filepath)
    pos = graph_data['pos']
    adj = graph_data['adj']
    print(f"Graph loaded from {filepath}")
    return pos, adj
```

Statistics of the results

It computes comprehensive statistics for a 2D instance by comparing initial and final positions. Inputs are the initial and final tensors (both of shape $(\#\text{nodes}, 2)$) and the adjacency vector for ordered pairs; output is a dictionary with the following statistics:

- V: cardinality of the graph (number of vertices)
- d_max: maximal degree of the graph
- K_max: size of estimated maximum clique

- r_min_i: minimum distance among all qubits (initial)
- r_min_f: minimum distance among all qubits (final)
- r_max_i: maximum distance among all qubits (initial)
- r_max_f: maximum distance among all qubits (final)
- r_adj_max_i: maximum distance among adjacent qubits (initial)
- r_adj_max_f: maximum distance among adjacent qubits (final)
- r_non_adj_min_i: minimum distance among non-adjacent qubits (initial)
- r_non_adj_min_f: minimum distance among non-adjacent qubits (final)
- n_constraints_i: number of violated constraints (initial)
- n_constraints_f: number of violated constraints (final)

Code 4.12: Statistics of the results

```
1  def stats(pos_i: torch.Tensor, pos_f: torch.Tensor, adj: torch.Tensor) -> dict
       :
2      n_nodes = len(pos_i)
3      all_distances_i = []
4      adjacent_distances_i = []
5      non_adjacent_distances_i = []
6      all_distances_f = []
7      adjacent_distances_f = []
8      non_adjacent_distances_f = []
9      degree = torch.zeros(n_nodes)
10     G = nx.Graph()
11     G.add_nodes_from(range(n_nodes))
12     idx = 0
13     for i in range(n_nodes):
14         for j in range(i + 1, n_nodes):
15             dist_i = torch.sqrt((pos_i[i, 0] - pos_i[j, 0])**2 + (pos_i[i, 1]
                   - pos_i[j, 1])**2)
```

```
16              all_distances_i.append(dist_i)
17              dist_f = torch.sqrt((pos_f[i, 0] - pos_f[j, 0])**2 + (pos_f[i, 1]
                    - pos_f[j, 1])**2)
18              all_distances_f.append(dist_f)
19              is_adjacent = adj[idx] > 0
20              if is_adjacent:
21                  adjacent_distances_i.append(dist_i)
22                  adjacent_distances_f.append(dist_f)
23                  degree[i] += 1
24                  degree[j] += 1
25                  G.add_edge(i, j)
26              else:
27                  non_adjacent_distances_i.append(dist_i)
28                  non_adjacent_distances_f.append(dist_f)
29              idx += 1
30      all_distances_i = torch.stack(all_distances_i)
31      all_distances_f = torch.stack(all_distances_f)
32      adjacent_distances_i = torch.stack(adjacent_distances_i) if
            adjacent_distances_i else torch.tensor([])
33      adjacent_distances_f = torch.stack(adjacent_distances_f) if
            adjacent_distances_f else torch.tensor([])
34      non_adjacent_distances_i = torch.stack(non_adjacent_distances_i) if
            non_adjacent_distances_i else torch.tensor([])
35      non_adjacent_distances_f = torch.stack(non_adjacent_distances_f) if
            non_adjacent_distances_f else torch.tensor([])
36      n_constraints_i = check_constraints(pos_i,adj)
37      n_constraints_f = check_constraints(pos_f,adj)
38      stats = {
39          'V': n_nodes,
40          'd_max': int(degree.max().item()),
41          'K_max': len(nx.approximation.max_clique(G)),
42          'r_{min}_i': all_distances_i.min().item() if len(all_distances_i) > 0
                else 0.0,
43          'r_{min}_f': all_distances_f.min().item() if len(all_distances_f) > 0
                else 0.0,
44          'r_{max}_i': all_distances_i.max().item() if len(all_distances_i) > 0
```

```
        else 0.0,
45      'r_{max}_f': all_distances_f.max().item() if len(all_distances_f) > 0
        else 0.0,
46      'r_adj_max_i': adjacent_distances_i.max().item() if len(
        adjacent_distances_i) > 0 else 0.0,
47      'r_adj_max_f': adjacent_distances_f.max().item() if len(
        adjacent_distances_f) > 0 else 0.0,
48      'r_non_adj_min_i': non_adjacent_distances_i.min().item() if len(
        non_adjacent_distances_i) > 0 else float('inf'),
49      'r_non_adj_min_f': non_adjacent_distances_f.min().item() if len(
        non_adjacent_distances_f) > 0 else float('inf'),
50      'n_constraints_i': n_constraints_i,
51      'n_constraints_f':n_constraints_f
52  }
53  return stats
```

It appends statistics as a new row in a CSV file. Inputs are the dictionary returned by `stats` and the CSV path.

Code 4.13: Statistics to CSV

```
1  def stats_to_csv(stats: dict, csv_path: str) -> None:
2      ordered_keys = list(stats.keys())
3      file_exists = os.path.isfile(csv_path)
4      with open(csv_path, 'a', newline='') as f:
5          writer = csv.writer(f)
6          if not file_exists:
7              writer.writerow(ordered_keys)
8          row = []
9          for key in ordered_keys:
10             value = stats[key]
11             # Round to 3 decimal places if it's a float, otherwise keep as is
12             if isinstance(value, float):
13                 row.append(round(value, 3))
14             else:
15                 row.append(value)
16         writer.writerow(row)
```

```
17      print(f"Statistics appended to {csv_path}")
```

Random graph generator

It generates a random graph with random positions and random adjacency. Inputs are the number of nodes and a connection percentage (0–100); outputs are a tensor of shape $(n_{\text{nodes}}, 2)$ with random coordinates and the adjacency vector for ordered pairs.

Code 4.14: Random graph generator

```
1  def random_graph(n_nodes: int, conn: int) -> tuple[torch.Tensor, torch.Tensor
       ]:
2      pos = torch.randn(n_nodes, 2)
3      m = n_nodes * (n_nodes - 1) // 2
4      n_edges = int(m * conn / 100)
5      adj = torch.zeros(m)
6      if n_edges > 0:
7          indices = torch.randperm(m)[:n_edges]
8          adj[indices] = 1
9      return pos, adj
```

4.5.2 The neural network

After implementing the utility file, I built the neural network, which is composed of an autoencoder and the so-called distance-computation layers.

Autoencoder

The constructor of the class autoencoder can take in input many parameters in order to construct the neural network with different hyperparameter choices. In particular, it takes as input the input dimension (2 * n_points), a list of hidden layer dimensions for the encoder (Decoder will be symmetric), the dropout probability, and a Scale factor for output activation.

Code 4.15: Autoencoder

```
1  class PointAutoencoder(nn.Module):
2      def __init__(self, input_dim, hidden_dims, dropout_prob, scale):
```

```
        super(PointAutoencoder, self).__init__()
        encoder_layers = []
        prev_dim = input_dim
        for dim in hidden_dims:
            encoder_layers.extend([
                nn.Linear(prev_dim, dim),
                nn.ReLU(),
                nn.Dropout(dropout_prob),
            ])
            prev_dim = dim
        self.encoder = nn.Sequential(*encoder_layers)
        decoder_layers = []
        reversed_dims = list(reversed(hidden_dims))
        for i in range(len(reversed_dims) - 1):
            decoder_layers.extend([
                nn.Linear(reversed_dims[i], reversed_dims[i + 1]),
                nn.ReLU(),
                nn.Dropout(dropout_prob),
            ])
        decoder_layers.extend([
            nn.Linear(reversed_dims[-1], input_dim),
            ScaledTanh(scale)
        ])
        self.decoder = nn.Sequential(*decoder_layers)
    def forward(self, x):
        z = self.encoder(x)
        out = self.decoder(z)
        return out, z
```

Then there are the classes of Fixed-weight layer DiffL for pairwise coordinate differences and of Fixed-weight layer DistL for Euclidean distance.

Code 4.16: Difference and distance layers

```
class PairwiseDifferenceLayer(nn.Module):
    def __init__(self, n_points):
        super().__init__()
```

```
4         self.n_points = n_points
5         self.pairs = [(i, j) for i, j in itertools.combinations(range(n_points
              ), 2)]
6     def forward(self, transformed_coordinates):
7         n = self.n_points
8         x_coords = transformed_coordinates[:n]
9         y_coords = transformed_coordinates[n:]
10        x_diffs = []
11        y_diffs = []
12        for (i, j) in self.pairs:
13            x_diffs.append(x_coords[i] - x_coords[j])
14            y_diffs.append(y_coords[i] - y_coords[j])
15        diffs = torch.cat([torch.stack(x_diffs), torch.stack(y_diffs)])
16        return diffs**2
17 class EuclidianDifferenceLayer(nn.Module):
18        def __init__(self, n_points):
19            super().__init__()
20            self.n_points = n_points
21        def forward(self, squared_diffs):
22            n = self.n_points
23            m = n * (n - 1) // 2
24            x_diffs = squared_diffs[:m]
25            y_diffs = squared_diffs[m:]
26            diffs = torch.stack([x_diffs[i] + y_diffs[i] for i in range(m)])
27            return torch.sqrt(torch.clamp(diffs, min=1e-8))
```

Here is the code of the full model combining the Autoencoder and Pairwise Difference layer.

Code 4.17: Full Model

```
1 class PointModel(nn.Module):
2  def __init__(self, n_points, hidden_dims, drop_prob, scale):
3      super().__init__()
4      self.autoencoder = PointAutoencoder(
5          input_dim=2*n_points,
6          hidden_dims=hidden_dims,
7          dropout_prob=drop_prob,
```

```
        scale=scale
    )
    self.pairwise_diff = PairwiseDifferenceLayer(n_points=n_points)
    self.euclidian_diff = EuclidianDifferenceLayer(n_points=n_points)
def forward(self, input):
    coordLOutput, latent = self.autoencoder(input)
    diffLOutput = self.pairwise_diff(coordLOutput)
    distLOutput = self.euclidian_diff(diffLOutput)
    return coordLOutput, latent, diffLOutput, distLOutput
```

Custom output activation

This is the class for the activation function of the autoencoder.

Code 4.18: Tanh activation function

```
class ScaledTanh(nn.Module):
def __init__(self, scale):
    super(ScaledTanh, self).__init__()
    self.scale = scale
def forward(self, t):
    return self.scale * torch.tanh(t)
```

Custom loss function

This is the code of the loss function that takes as input a tensor of [n_points*(n_points-1)/2], which contains $\sqrt{dx^2 + dy^2}$, the adjacency vector, the Rydberg radius and an Epsilon.

Code 4.19: Custom loss function

```
def loss_function(d: torch.Tensor, adj: torch.Tensor, rb: float, eps: float)
   -> torch.Tensor:
    loss1 = torch.mean(torch.abs(torch.clamp(d, min=100) - 100))
    loss2 = torch.mean(torch.abs(torch.clamp(d, max=4) - 4))
    loss3 = torch.mean(adj * torch.abs(torch.clamp(d, min=rb) - rb))
    loss4 = torch.mean((1 - adj) * torch.abs(torch.clamp(d, max=rb + eps) - (
        rb + eps)))
    return loss1 + loss2 + loss3 + loss4
```

Model training function

The following training function is used to train the model on a graph instance from positions and an adjacency matrix. Inputs are node positions, adjacency matrix, compute device (CPU or GPU), maximum training epochs, early-stopping loss tolerance, Rydberg-radius constraint, epsilon for non-adjacent-pair constraints, learning rate, dropout probability, encoder hidden-layer dimensions, and output-activation scale factor. Output is the final position tensor of shape $(n_{\text{points}}, 2)$.

Code 4.20: Model training function

```
def train(pos: torch.Tensor,
        adj: torch.Tensor,
        dev: str = "cpu",
        epochs: int = 10000,
        tolerance: float = 1e-6,
        rb: float = 10.26,
        eps: float = 0.1,
        lr: float = 1e-3,
        dropout: float = 0.5,
        hidden_dims: list = [64, 36, 18, 9],
        scale: float = 50.0) -> torch.Tensor:
    device = torch.device(dev)
    print(f"Using device: {device}")
    n_points = len(pos)
    pos = pos.to(device)
    adj = adj.to(device)
    input_data = torch.cat([pos[:, 0], pos[:, 1]])
    model = PointModel(n_points, hidden_dims=hidden_dims, drop_prob=dropout,
        scale=scale).to(device)
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
    start_time = time.time()
    for step in range(epochs):
        optimizer.zero_grad()
        coordLOutput, latent, diffLOutput, distLOutput = model(input_data)
        loss = loss_function(distLOutput, adj, rb, eps)
        if torch.isnan(loss):
            print(f"NaN detected at step {step}")
```

```
27                break
28            loss.backward()
29            optimizer.step()
30            if step % 100 == 0:
31                print(f"Step {step:5d}, loss = {loss.item():.6f}")
32            if loss.item() <= tolerance:
33                print(f"Feasible solution found at step {step}, loss = {loss.item
                     ():.6f}")
34                break
35        training_time = time.time() - start_time
36        print("Autoencoder output shape:", coordLOutput.shape)
37        print("Latent representation shape:", latent.shape)
38        print("DiffL output shape:", diffLOutput.shape)
39        print("DistL output shape:", distLOutput.shape)
40        print("Training time:", training_time)
41        return torch.stack([coordLOutput[:n_points], coordLOutput[n_points:]], dim
              =1).cpu()
```

4.5.3 Testing on the antennas problem

The first dataset on which I tested the GEAN implementation was the original CSV file containing the Turin antennas data, kindly provided by Chiara Vercellino who recovered this data from OpenCelliD[4], the world's largest Open Database of Cell Towers. The results were in line with those reported in the paper, as shown in the following table.

[4] https://www.opencellid.org/

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c
7	5	4	47.43	4.576	241.45	22.283	120.568	10.239	136.519	11.557	27	0
4	3	3	21.958	4.856	143.273	11.226	127.687	8.244	143.273	11.226	13	0
6	4	4	0.000	4.926	269.394	24.773	126.746	10.179	160.301	12.633	20	0
6	5	3	36.036	4.258	217.793	18.862	129.967	10.079	134.680	11.577	22	0
13	8	5	8.310	4.084	357.550	24.610	129.759	9.442	132.025	10.700	60	0
7	4	4	48.549	4.381	293.257	19.137	126.290	9.661	133.603	10.669	25	0
5	4	4	15.882	4.335	133.308	11.990	122.476	9.305	133.308	11.990	19	0
3	2	2	85.193	5.543	178.094	15.110	93.796	9.576	178.094	15.110	8	0
3	2	2	55.281	7.699	135.233	11.741	97.607	9.287	135.233	11.741	8	0
3	2	2	76.982	7.047	183.296	16.247	108.899	9.714	183.296	16.247	8	0

Table 4.2: Turin antennas replication results.

To describe the characteristics of the embeddings, the following notation is introduced:

- V: the cardinality of the graph
- d_m: the maximum degree
- K_m: the size of the estimated maximum clique, where clique stands for complete subgraph.
- r^i_{min}: minimum distance among qubits in the initial configuration
- r^f_{min}: minimum distance among qubits in the final configuration
- r^i_{max}: maximum distance among qubits in the initial configuration
- r^f_{max}: maximum distance among qubits in the final configuration
- r^i_{adj}: maximum distance among adjacent qubits in the initial configuration
- r^f_{adj}: maximum distance among adjacent qubits in the final configuration
- $r^i_{\overline{adj}}$: minimum distance among non-adjacent qubits in the initial configuration
- $r^f_{\overline{adj}}$: minimum distance among non-adjacent qubits in the final configuration
- n^i_c: number of violated constraints in the initial configuration
- n^f_c: number of violated constraints in the final configuration

- Conn: percentage of connection of the graph

4.5.4 Testing on the protein problem

Given the satisfactory results on the antenna dataset, I proceeded to test the network on the more complicated protein problem, since it was close in difficulty to the final problem I had to tackle (quantum molecular docking), which I discuss in the next chapter. Since graphs in the quantum molecular docking problem do not follow a specific pattern, I generated the protein dataset using random graphs grouped by number of nodes and connection percentage. I iteratively tested the network from 10 nodes and 10% connectivity up to 100 nodes and 100% connectivity. For each node count, the test started at 10% connectivity and increased connectivity by 10% until unresolved constraints appeared. At that point, the number of nodes was increased and the process restarted from 10% connectivity. As soon as the network failed even at the minimum connectivity, the script stopped. In this case, the results were not as satisfactory as those for the antennas problem, as shown in the table below.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	5.351	2.334	47.806	1.586	9.773	0.347	10.556	86	0	10%
10	3	2	0.581	4.489	4.603	36.794	3.911	9.438	0.581	10.376	78	0	20%
10	5	3	0.152	3.967	4.581	24.114	4.581	12.745	0.152	6.649	76	5	30%
20	4	2	0.115	3.993	4.321	42.454	4.321	17.973	0.139	10.333	360	10	10%

Table 4.3: Protein results

Since the baseline configuration did not yield satisfactory results, several architectural and hyper-parameter variations were explored by leveraging the computational resources of the Leonardo supercomputer. In particular, multiple jobs were executed in parallel using Slurm. Slurm (Simple Linux Utility for Resource Management) is an open-source cluster manager and job scheduler widely used in high-performance computing environments. It handles resource allocation for a specified duration, provides a framework for job execution and monitoring, and manages resource contention through a queueing system[5]. In this work, a separate job was submitted for each neural network configuration. Submitting a job involves requesting a specific set of computational resources for a

[5] `https://slurm.schedmd.com/overview.html`

defined time interval. This is achieved through a Bash script composed of two main sections. The first section contains Slurm directives, which specify parameters such as memory, time limits, job name, partition, number of tasks, CPUs per task, and potential dependencies on other jobs. The second section includes standard Bash commands, such as loading required modules, activating Python environments, and executing the training script with the desired parameters. An example is reported below.

Code 4.21: Example of Slurm Script

```
1  #!/bin/bash
2  #SBATCH -A PHD_flotta
3  #SBATCH -p boost_usr_prod
4  #SBATCH --time 00:01:00
5  #SBATCH -N 1
6  #SBATCH --gres=gpu:1
7  #SBATCH --ntasks-per-node=1
8  #SBATCH --cpus-per-task=1
9  #SBATCH --mem=32000
10 #SBATCH --job-name=test
11 #SBATCH --output=/leonardo_work/PHD_flotta/embeddingnn/out/%x_%j.log
12 #SBATCH --error=/leonardo_work/PHD_flotta/embeddingnn/out/%x_%j.log
13 export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
14 cd /leonardo_work/PHD_flotta/embeddingnn
15 source .venv/bin/activate
16 python proteins.py --model gean --device cuda --hidden_dims 64 32 16
```

The general pattern is stable: GEAN frequently reduces initial constraint violations to near-feasible or feasible regimes, though convergence quality depends on connectivity and graph size. Experiments on proteins with smaller architectures (e.g., $32 \rightarrow 16 \rightarrow 8$ or $64 \rightarrow 32$) show competitive behavior on lightly connected instances but degrade on denser cases. Deeper configurations (e.g., $64 \rightarrow 36 \rightarrow 18 \rightarrow 9$) improve representational flexibility but do not uniformly solve high-connectivity settings. Dropout sweeps (0, 0.25, 0.5, 0.75) suggest a trade-off between robustness and precision: moderate regularization can help, while aggressive settings may hinder convergence on difficult

graphs. Learning-rate sweeps (10^{-4} to 10^{-2}) confirm that too-small values slow adaptation and too-large values can destabilize sensitive constraints, especially non-adjacent separation. All the results are reported in the tables below.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	6.389	2.334	49.711	1.586	8.561	0.347	11.069	86	0	10%
10	3	2	0.581	4.504	4.603	39.446	3.911	9.412	0.581	10.952	78	0	20%
10	5	3	0.152	4.014	4.581	29.024	4.581	13.801	0.152	10.345	76	3	30%
20	4	2	0.115	4.133	4.321	42.414	4.321	17.985	0.139	10.334	360	4	10%

Table 4.4: Protein results with architecture 32-16-8.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	5.465	2.334	39.287	1.586	9.061	0.347	10.779	86	0	10%
10	3	2	0.581	4.760	4.603	35.441	3.911	9.035	0.581	10.882	78	0	20%
10	5	3	0.152	3.982	4.581	29.317	4.581	13.685	0.152	10.350	76	4	30%
20	4	2	0.115	3.995	4.321	47.660	4.321	18.473	0.139	9.492	360	7	10%

Table 4.5: Protein results with architecture 64-32.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	5.205	2.334	44.070	1.586	9.783	0.347	10.437	86	0	10%
10	3	2	0.581	4.385	4.603	38.903	3.911	10.224	0.581	10.460	78	0	20%
10	5	3	0.152	3.864	4.581	28.961	4.581	13.555	0.152	10.303	76	8	30%
20	4	2	0.115	3.994	4.321	48.602	4.321	22.358	0.139	10.259	360	10	10%

Table 4.6: Protein results with Dropout 0.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	6.002	2.334	42.126	1.586	9.559	0.347	10.471	86	0	10%
10	3	2	0.581	6.186	4.603	33.794	3.911	9.634	0.581	10.675	78	0	20%
10	5	3	0.152	3.991	4.581	28.635	4.581	13.065	0.152	10.345	76	5	30%
20	4	2	0.115	5.726	4.321	64.006	4.321	10.193	0.139	10.691	360	0	10%
20	8	3	0.069	3.923	3.401	38.378	2.749	20.702	0.069	5.771	342	27	20%
30	5	3	0.086	3.933	4.798	58.941	3.928	29.542	0.086	6.404	822	40	10%

Table 4.7: Protein results with Dropout 0.25.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	7.187	2.334	44.569	1.586	9.299	0.347	10.578	86	0	10%
10	3	2	0.581	3.996	4.603	33.192	3.911	13.675	0.581	10.331	78	4	20%
20	4	2	0.115	4.379	4.321	43.787	4.321	17.972	0.139	10.337	360	5	10%

Table 4.8: Protein results with Dropout 0.5.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	4.468	2.334	45.126	1.586	9.027	0.347	11.477	86	0	10%
10	3	2	0.581	6.401	4.603	42.242	3.911	10.155	0.581	10.565	78	0	20%
10	5	3	0.152	3.993	4.581	30.479	4.581	13.655	0.152	10.332	76	5	30%
20	4	2	0.115	4.010	4.321	49.873	4.321	13.647	0.139	10.309	360	3	10%

Table 4.9: Protein results with Dropout 0.75.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	4.930	2.334	51.161	1.586	10.131	0.347	12.410	86	0	10%
10	3	2	0.581	3.842	4.603	36.604	3.911	12.218	0.581	8.641	78	3	20%
20	4	2	0.115	5.345	4.321	57.775	4.321	10.208	0.139	10.441	360	0	10%
20	8	3	0.069	2.842	3.401	32.379	2.749	20.383	0.069	4.483	342	30	20%
30	5	3	0.086	3.304	4.798	54.053	3.928	18.409	0.086	5.716	822	42	10%

Table 4.10: Protein results with learning rate 0.0001.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	7.316	2.334	47.797	1.586	9.549	0.347	12.421	86	0	10%
10	3	2	0.581	3.969	4.603	24.050	3.911	13.667	0.581	10.359	78	3	20%
20	4	2	0.115	6.798	4.321	61.931	4.321	10.049	0.139	10.490	360	0	10%
20	8	3	0.069	3.926	3.401	36.332	2.749	26.802	0.069	3.997	342	35	20%
30	5	3	0.086	0.114	4.798	54.999	3.928	27.179	0.086	6.457	822	45	10%

Table 4.11: Protein results with learning rate 0.0005.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	8.556	2.334	51.836	1.586	9.716	0.347	12.818	86	0	10%
10	3	2	0.581	5.047	4.603	35.886	3.911	9.812	0.581	12.455	78	0	20%
10	5	3	0.152	3.980	4.581	26.069	4.581	11.326	0.152	9.921	76	9	30%
20	4	2	0.115	4.018	4.321	61.097	4.321	17.091	0.139	10.207	360	4	10%

Table 4.12: Protein results with learning rate 0.005.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	3	2	0.347	8.809	2.334	59.071	1.586	9.847	0.347	13.109	86	0	10%
10	3	2	0.581	5.569	4.603	42.030	3.911	9.416	0.581	10.843	78	0	20%
10	5	3	0.152	3.875	4.581	24.421	4.581	13.594	0.152	9.323	76	6	30%
20	4	2	0.115	3.906	4.321	65.220	4.321	18.035	0.139	10.116	360	8	10%

Table 4.13: Protein results with learning rate 0.01.

Overall, these results are not fully satisfactory, which motivated further, more in-depth network modifications described in the next chapter. This study reframes hardware-aware optimization-problem deployment as a constrained geometric learning problem and shows that a neural architecture can directly optimize physical realizability. GEAN integrates learned coordinate refinement with deterministic distance computation and constraint-aligned losses, providing a practical alternative to purely combinatorial embedding approaches. The empirical evidence across antenna, protein, and graph-coloring instances indicates that GEAN can find feasible embeddings quickly and at scales where classical solvers may fail under strict time budgets. The 3D extension further broadens applicability by relaxing planar limitations. Future directions include stronger physics-informed regularization, adaptive curricula for dense graphs, and tighter integration with downstream quantum execution pipelines. Despite these results, the experiments also highlight a key limitation of the current architecture: the implicit treatment of adjacency information during the embedding process. In particular, while GEAN successfully enforces geometric constraints, it does not fully exploit the relational structure of the input graph during representation learning. This observation motivates the introduction of a more expressive message-passing mechanism. In the following chapter, I address this limitation by incorporating Graph Attention (GATv2) layers, allowing the model to explicitly weight neighborhood interactions and better encode adjacency patterns before the geometric embedding stage. This modification leads to a more informed latent representation and ultimately improves the quality and robustness of the resulting embeddings.

5. Improvements and testing of GEAN

In the previous chapter, we described how GEAN works and how I implemented and tested it on the antenna and protein problems. GEAN handled the first problem well, while some network limitations emerged in the second one. In this chapter, I describe the changes we introduced to improve network performance.

5.1 The improved neural network

Since we work with graphs where adjacency information is crucial (in fact, the largest contributions to the loss updates came from adjacency-related terms), we added a GNN block before the autoencoder. In particular, we use GATv2 layers that take the adjacency structure into account, not only node coordinates. Here is the code of the new encoder based on graph attention layers, which creates an embedding for every node through message passing on the adjacency matrix. The encoder constructor takes as input the node feature dimension, a list of hidden dimensions for intermediate GAT layers, the output embedding dimension, the dropout probability, and the number of attention heads per layer (the length should be `len(hidden_dims) + 1`). If a single integer is provided, it is used for all layers except the last.

GAT-based Node Encoder

Code 5.1: GAT-based Node Encoder

```
1  class GATEncoder(nn.Module):
2      def __init__(self, in_features=2, hidden_dims=[16], out_features=8, heads
           =[4], dropout=0.2):
3          super(GATEncoder, self).__init__()
4          if isinstance(heads, int):
5              heads = [heads] * len(hidden_dims) + [1]
6          elif len(heads) == len(hidden_dims):
7              heads = list(heads) + [1]  # Last layer always has 1 head
```

```
8        assert len(heads) == len(hidden_dims) + 1, \
9            f"heads length ({len(heads)}) must be hidden_dims length + 1 ({len
               (hidden_dims) + 1})"
10       self.gat_layers = nn.ModuleList()
11       self.dropout = nn.Dropout(dropout)
12       self.elu = nn.ELU()
13       all_dims = [in_features] + list(hidden_dims) + [out_features]
14       for i in range(len(all_dims) - 1):
15           in_dim = all_dims[i]
16           out_dim = all_dims[i + 1]
17           n_heads = heads[i]
18           is_last = (i == len(all_dims) - 2)
19           if i > 0:
20               in_dim = all_dims[i] * heads[i - 1]
21           self.gat_layers.append(
22               GATv2Conv(
23                   in_channels=in_dim,
24                   out_channels=out_dim,
25                   heads=n_heads,
26                   dropout=dropout,
27                   concat=not is_last  # Don't concat on last layer
28               )
29           )
30       self.n_layers = len(self.gat_layers)
31   def forward(self, x, edge_index):
32       for i, gat_layer in enumerate(self.gat_layers):
33           x = gat_layer(x, edge_index)
34           if i < self.n_layers - 1:
35               x = self.elu(x)
36               x = self.dropout(x)
37       return x
```

Full model: GAT + Autoencoder + Distance layers

The full model takes as input the number of nodes, the autoencoder hidden dimensions, dropout probability, output-activation scale, GAT hidden dimensions, node embedding size, and the list of

attention heads for each GAT layer.

Code 5.2: Full improved model

```
class PointModelGAT(nn.Module):
    def __init__(self, n_points, hidden_dims, drop_prob, scale,
                     gat_hidden_dims=[16], gat_out=8, gat_heads=[4]):
    super().__init__()
    self.n_points = n_points
    self.gat_encoder = GATEncoder(
        in_features=2,  # (x, y) per nodo
        hidden_dims=gat_hidden_dims,
        out_features=gat_out,
        heads=gat_heads,
        dropout=drop_prob
        )
    gat_output_dim = n_points * gat_out
    self.projection = nn.Sequential(
        nn.Linear(gat_output_dim, 2 * n_points),
        nn.ReLU()
        )
    self.autoencoder = PointAutoencoder(
        input_dim=2*n_points,
        hidden_dims=hidden_dims,
        dropout_prob=drop_prob,
        scale=scale
    )
    self.pairwise_diff = PairwiseDifferenceLayer(n_points=n_points)
    self.euclidian_diff = EuclidianDifferenceLayer(n_points=n_points)
    def forward(self, pos, edge_index):
        node_embeddings = self.gat_encoder(pos, edge_index)
        flat_embeddings = node_embeddings.flatten()
        projected = self.projection(flat_embeddings)
        coordLOutput, latent = self.autoencoder(projected)
        diffLOutput = self.pairwise_diff(coordLOutput)
        distLOutput = self.euclidian_diff(diffLOutput)
        return coordLOutput, latent, diffLOutput, distLOutput, node_embeddings
```

The following training function trains the model on a graph instance defined by node positions and an adjacency matrix. Its inputs are node positions, adjacency matrix, training device (CPU or GPU), maximum number of epochs, early-stopping tolerance, Rydberg radius constraint, epsilon for non-adjacent pairs, learning rate, dropout probability, autoencoder hidden dimensions, output scale, GAT hidden dimensions, node embedding size, and attention heads. It returns the final position tensor with shape `(n_points, 2)`.

Code 5.3: Improved training function

```
def train_gat(pos: torch.Tensor,
          adj: torch.Tensor,
          dev: str = "cpu",
          epochs: int = 10000,
          tolerance: float = 1e-6,
          rb: float = 10.26,
          eps: float = 0.1,
          lr: float = 1e-3,
          dropout: float = 0.5,
          hidden_dims: list = [64, 36, 18, 9],
          scale: float = 50.0,
          gat_hidden_dims: list = [16],
          gat_out: int = 8,
          gat_heads: list = [4]) -> torch.Tensor:
    device = torch.device(dev)
    print(f"Using device: {device}")
    n_points = len(pos)
    pos = pos.to(device)
    adj = adj.to(device)
    adj_matrix = torch.zeros(n_points, n_points, device=device)
    idx = 0
    for i in range(n_points):
        for j in range(i + 1, n_points):
            adj_matrix[i, j] = adj[idx]
            adj_matrix[j, i] = adj[idx]
            idx += 1
    edge_index, _ = dense_to_sparse(adj_matrix)
```

```
28     edge_index = edge_index.to(device)
29     adj_pairs = adj.clone()
30     model = PointModelGAT(
31         n_points,
32         hidden_dims=hidden_dims,
33         drop_prob=dropout,
34         scale=scale,
35         gat_hidden_dims=gat_hidden_dims,
36         gat_out=gat_out,
37         gat_heads=gat_heads
38     ).to(device)
39     optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
40     start_time = time.time()
41     for step in range(epochs):
42         optimizer.zero_grad()
43         coordLOutput, latent, diffLOutput, distLOutput, node_emb = model(pos,
               edge_index)
44         loss = loss_function(distLOutput, adj_pairs, rb, eps)
45         if torch.isnan(loss):
46             print(f"NaN detected at step {step}")
47             break
48         loss.backward()
49         optimizer.step()
50         if step % 100 == 0:
51             print(f"Step {step:5d}, loss = {loss.item():.6f}")
52         if loss.item() <= tolerance:
53             print(f"Feasible solution found at step {step}, loss = {loss.item
                   ():.6f}")
54             break
55     training_time = time.time() - start_time
56     print("Node embeddings (GAT) shape:", node_emb.shape)
57     print("Autoencoder output shape:", coordLOutput.shape)
58     print("Latent representation shape:", latent.shape)
59     print("DiffL output shape:", diffLOutput.shape)
60     print("DistL output shape:", distLOutput.shape)
61     print("Training time:", training_time)
```

```
62      return torch.stack([coordLOutput[:n_points], coordLOutput[n_points:]], dim
            =1).cpu()
```

5.1.1 Testing on proteins

A Graph Attention Network front-end (GAT+GEAN) improves some low-to-medium connectivity protein instances, but it still struggles on the hardest dense cases. This indicates that message passing helps initialization and representation, but does not remove the geometric hardness of the problem, as shown in the table below.

V	d_m	K_m	r^i_{min}	r^f_{min}	r^i_{max}	r^f_{max}	r^i_{adj}	r^f_{adj}	$r^i_{\overline{adj}}$	$r^f_{\overline{adj}}$	n^i_c	n^f_c	Conn
10	2	2	0.105	5.103	3.822	49.349	2.016	10.174	0.105	10.928	86	0	10%
10	4	2	0.422	5.911	4.215	31.624	4.215	9.996	0.500	11.218	78	0	20%
10	6	3	0.243	4.122	4.426	28.761	3.506	9.427	0.243	10.755	74	0	30%
10	6	4	0.266	4.737	4.145	27.045	4.145	10.100	0.266	10.703	71	0	40%
10	7	5	0.291	3.998	3.442	20.660	3.121	13.666	0.587	7.784	68	7	50%
20	4	3	0.103	4.398	3.873	57.755	3.873	9.939	0.103	10.450	361	0	10%
20	7	3	0.186	0.124	4.838	34.177	3.456	23.530	0.209	4.328	340	43	20%
30	9	2	0.195	3.950	4.712	49.604	3.944	26.728	0.195	3.993	816	48	10%

Table 5.1: Protein results with GAT+GEAN.

Due to time constraints, we decided to keep the architecture as is and proceed to test it on a real-world problem called Quantum Molecular Docking.

5.2 Quantum Molecular Docking

Virtual screening (VS) is a computational technique that uses molecular modeling and optimization algorithms to identify and prioritize potential drug candidates from large chemical libraries. Molecular docking (MD), a core VS task, predicts the preferred orientation of a ligand when bound to a target receptor to form a stable complex. Docking explores the configuration space of the two molecules and scores each candidate using an energy function built from physical or empirical parameters, typically calibrated on experimental data. To accurately rank candidate poses, both precise scoring functions and efficient search strategies are required. MD is computationally demanding because the problem has many degrees of freedom. Quantum computing approaches for molecular

docking have already been explored in the literature: for example, quantum annealing has been used for molecular unfolding, and photonic platforms such as Gaussian Boson Samplers have been investigated for docking prediction. Molecular docking is a problem that can be naturally addressed on neutral-atom devices. The authors of [4] provide a proof of concept showing this feasibility. In their work, Pulser is used to simulate neutral-atom hardware. Pulser is an open-source Python framework from Pasqal for designing and simulating pulse sequences on programmable atom arrays. The first key step is mapping molecules to graphs, as described in [1]. With this mapping, docking is reduced to a maximum clique problem, which is equivalent to a maximum independent set (MIS) problem on the complementary graph (Fig. 5.1). An independent set is a subset of vertices with no edges between them; the largest one is called a maximum independent set.

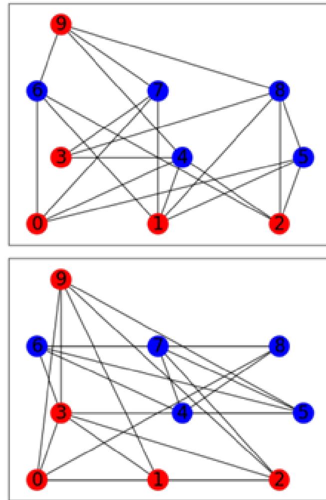

Figure 5.1: Upper figure: the maximum independent set solution, lower figure: the maximum clique solution in the complementary graph. From [4].

One advantage of neutral-atom devices is that the Hamiltonian ground state naturally encodes MIS solutions on unit-disk (UD) graphs. A UD graph can be embedded in the 2D Euclidean plane so that two vertices are connected if their Euclidean distance is below a threshold. Each physical atom represents a graph vertex, and low-energy states penalize pairs of nearby atoms simultaneously excited to the Rydberg state. This blockade effect naturally enforces independent-set constraints in the ground state.

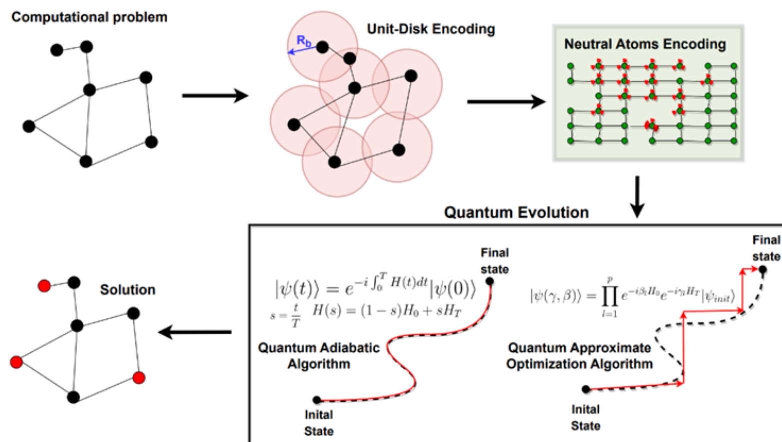

Figure 5.2: Solving process of a MIS problem on a neutral-atoms device - from [4]

Finally, the authors in [4] provide the result of the proposed proof-of-concept conducted with two small molecules capable of interacting with a molecular docking site, namely acetic acid and ethylene glycol, showing promising results.

5.2.1 Molecule to graph

As noted above, solving MD on a neutral-atom device requires mapping molecules to graphs. The first step is to inspect the molecular structure and identify pharmacophore points.

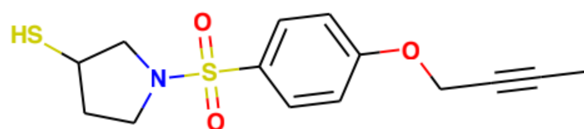

Figure 5.3: Planar structure of the ligand molecule - from [1]

A pharmacophore is a set of points that have a large influence on the molecule's pharmacological and biological interactions. These points may define a common subset of features, such as charged chemical groups or hydrophobic regions, that may be shared across a larger group of active compounds. Examples of pharmacophore points could be: negative/positive charge, hydrogen-bond donor/acceptor, hydrophobe, and aromatic ring.

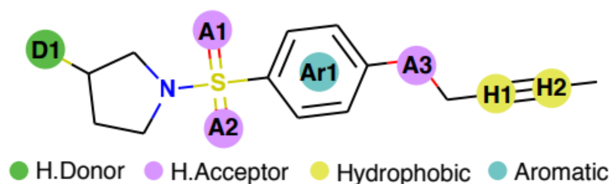

Figure 5.4: Identification of pharmacophore points - from [1]

With the 3D structure of the molecule, we can determine the pairwise distance between the pharmacophore points.

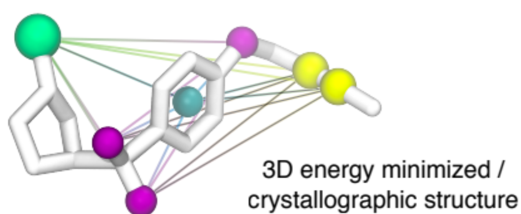

Figure 5.5: Measure of their pairwise distance by using the known 3D structure - from [1]

This information represents a molecule as a graph with weighted edges. Hence, we refer to this molecular graph representation as a labeled distance graph. A labeled distance graph is constructed as follows for both the ligand and receptor:

1. Heuristically identify pharmacophore points likely to be involved in the binding interaction. These form the vertices of the graph.

2. Add an edge between every pair of vertices and set its weight to the Euclidean distance between the pharmacophore points they represent.

3. Assign a label to every vertex according to the respective type of pharmacophore point it represents.

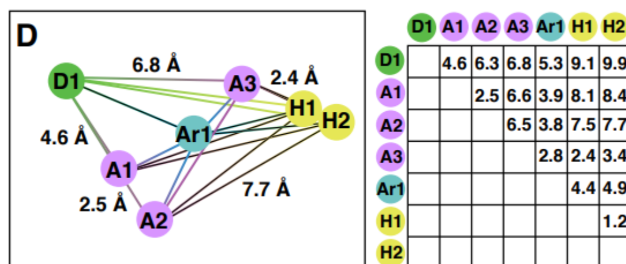

Figure 5.6: Combine the knowledge about pharmacophore points and pairwise distances to create a first version of the labelled distance graph where vertices represent the pharmacophore points and edge weights their respective pairwise distance (the complete weight matrix is on the right). From [1]

The labeled distance graphs capture the geometric three-dimensional shapes and the molecular features of both the protein binding site and the ligand that interacts with it. Now we combine these two graphs into a single binding interaction graph. Subsequently, we reduce the molecular docking problem to the problem of finding the maximum weighted clique.

5.2.2 Binding interaction graph

To model possible binding poses, the protein and ligand are represented as graphs. Different pharmacophore feature types can exhibit specific attractive interactions. When a ligand feature and a receptor feature are mutually attractive, they define a potential contact. These contacts become the nodes of the binding interaction graph.

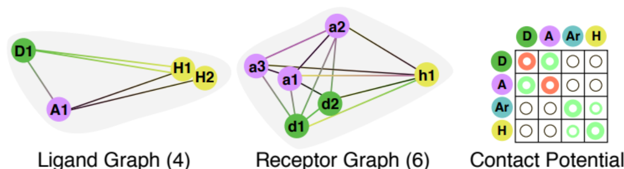

Figure 5.7: Two labelled graphs (one for the ligand and one for the receptor) and the corresponding contact potential that captures the interaction strength between different types of vertex labels. We denote the vertices on the ligand and receptor with upper and lower case letters respectively. From [1]

A binding pose can be defined by a set of at least three non-colinear contacts. We model contacts as interacting vertex pairs from the labelled distance graphs of the ligand and the binding site. Let G_L and G_B be these graphs, with vertex sets V_L and V_B. A contact is represented by a vertex $c_i \in V_L \times V_B$. Therefore, the vertices of the binding interaction graph are all possible contacts. In principle, any ligand pharmacophore point may interact with any binding-site pharmacophore point, so all possible pairs must be considered. The binding interaction graph then contains nm vertices, where $n = |V_L|$ and $m = |V_B|$.

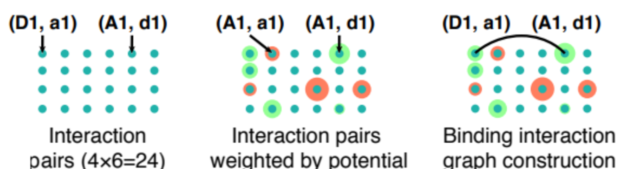

Figure 5.8: The binding interaction graph is constructed by creating a vertex for each possible contact between ligand and the receptor, weighted by the contact potential. From [1]

However, not every combination of contacts is physically realizable. Two contacts are incompatible if realizing them together would violate ligand or binding-site geometry. To capture this, two contact vertices are connected by an edge if and only if they are compatible. Consequently, any pairwise compatible set of contacts forms a complete subgraph (clique). Compatibility is modeled through the notion of τ-flexibility (Fig.5.9). Although both ligand and binding site may be flexible, the geometric distances induced by two contacts should remain approximately consistent on both molecules. Two contacts (v_{l_1}, v_{b_1}) and (v_{l_2}, v_{b_2}) form a τ-flexible pair if the ligand-side distance and the binding-site distance differ by at most $\tau + 2\epsilon$, where τ is the flexibility constant and ϵ is the interaction-distance tolerance.

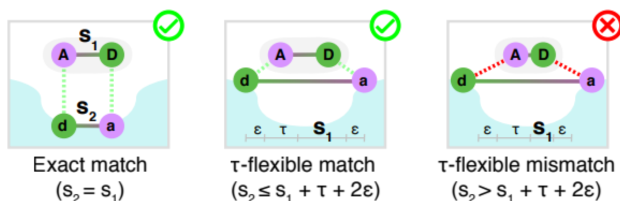

Figure 5.9: Pairs of vertices that represent compatible contacts are connected by an edge - from [1]

To model different interaction strengths among pharmacophore types, we assign a weight to each

vertex of the binding interaction graph. These weights are derived from the pharmacophore labels in the two labelled distance graphs. Given a label set $\mathbb{L}$, a potential function $\kappa : \mathbb{L} \times \mathbb{L} \to \mathbb{R}$ assigns an interaction strength to each label pair. This weighting biases the search toward stronger intermolecular interactions. Potential functions may be derived from data-driven approaches (e.g., statistical or knowledge-based potentials) or from quantum-mechanical models. Under this formulation, the most likely binding poses correspond to the heaviest cliques in the binding interaction graph. Therefore, the task is a maximum weighted clique problem, i.e., a weighted generalization of the maximum clique problem.

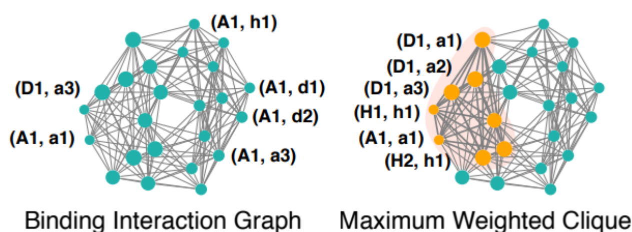

Figure 5.10: The resulting graph is then used to search for potential binding poses. These are represented as cliques of the graph as they form a set of pairwise compatible contacts. The heaviest vertex-weighted cliques represent the most likely binding poses (depicted in orange). From [1]

When G has n vertices, the number of possible subgraphs is $O(2^n)$, so brute-force search quickly becomes infeasible as n grows. The max-clique decision problem is NP-hard; therefore, unless P=NP, exact algorithms may require superpolynomial time in the worst case.

5.2.3 From WMC to MWIS problem

The maximum clique problem seeks the largest complete subgraph of a graph. In this context, a clique in the binding interaction graph represents a set of mutually compatible pharmacophore contacts. For docking, clique size alone is not sufficient: interaction energies must also be considered. We therefore formulate a weighted maximum clique problem (WMCP), where each vertex $i \in V$ has a positive weight w_i, collected in a vector $w \in \mathbb{R}^n$. The WMCP identifies conformations that satisfy geometric constraints while maximizing total interaction strength, and can be written as follows:

$$max \sum_{i=1}^{n} w_i x_i \tag{5.1}$$

$$s.t. \quad x_i + x_j \leq 1, \forall (i,j) \in \bar{E} \tag{5.2}$$

$$x_i \in \{0,1\}, i = 1, \ldots, n \tag{5.3}$$

where $\bar{E} = \{(i,j) | i,j \in V, i \neq j, (i,j) \notin E\}$ are the edges of the complement graph $\bar{G} = (V, \bar{E})$. Because of the physics of neutral-atom devices, they are not naturally suited to solve the maximum clique problem directly. However, they are well suited to solve the maximum weighted independent set problem, which is obtained from WMCP on the complementary graph. We can translate WMCP into a maximum weighted independent set (MWIS) problem on the complement of the BIG. An independent set is a set of nodes with no edges between them. Although MWIS is also NP-hard, it is the formulation compatible with neutral-atom hardware; therefore, from this point onward we work on the Complementary Binding Interaction Graph (CBIG). As discussed in Chapter 3, MWIS can be formulated as an energy-minimization problem. In particular, the ground state of the neutral-atom register Hamiltonian can encode the MWIS solution.

$$H = \frac{\hbar}{2} \sum_i \Omega(t) \sigma_i^x \boxed{- \frac{\hbar}{2} \sum_i \delta(t) \sigma_i^z + \sum_{i<j} U_{ij} n_i n_j}$$

$$\Updownarrow$$

$$C_{\text{MWIS}}(z_1, \ldots, z_N) = - \sum_{i}^{N} w_i z_i + \sum_{(i,j) \in E} u_{ij} z_i z_j$$

The next challenge is embedding the CBIG into the neutral-atom register. In fact, the high number of CBIG edges makes mapping difficult even for very small molecules. Here, deep learning can help; in particular, we can exploit the neural network described in the previous section. We then feed the CBIG node coordinates and adjacency matrix to the network. It returns adjusted coordinates that preserve adjacency while satisfying neutral-atom register constraints. After encoding the CBIG on a neutral-atom register through this deep neural network, we apply a Quantum Adiabatic Algorithm (QAA) to solve the problem on a Pasqal neutral-atom device. Although this workflow

could in principle run on real quantum hardware, the preliminary tests were performed on classical neutral-atom emulators, as described in the next section.

5.3 Implementation of QMD

We study the binding interaction between tumor necrosis factor-α converting enzyme (TACE) and a thiol-containing aryl sulfonamide ligand (AS). TACE is selected because of the planar geometry of its active-site cleft and its pharmaceutical relevance. Since TACE regulates the release of membrane-anchored cytokines such as tumor necrosis factor-α, it is a promising target for treatments of several cancers, Crohn's disease, and rheumatoid arthritis. The ligand considered here belongs to a series of thiol-containing aryl sulfonamides known to potently inhibit TACE.

5.3.1 Data Preparation

The pipeline begins by importing the necessary libraries for cheminformatics (RDKit), molecular dynamics (MDAnalysis), and 3D visualization (py3Dmol). A crucial data preparation step is performed to merge structural and chemical data. While pdb files contain the global 3D placement of the protein-ligand complex, sdf files are more reliable for extracting accurate chemical and pharmacophore information for the ligand. The code extracts the 3D atomic coordinates from the pdb file (specifically mapping atom by atom) and updates the sdf file to generate an aligned ligand sdf file. This ensures that subsequent chemical feature analyses are performed on a perfectly positioned 3D molecule.

5.3.2 Pharmacophore Feature Extraction

The RDKit ChemicalFeatures factory is used to parse the aligned ligand and the receptor (protein) and extract 3D pharmacophore points (e.g., hydrogen-bond donors/acceptors, hydrophobes, aromatic rings). A manual refinement step can remove spurious or overly rigid features when needed. Ligand and protein pharmacophore features are then mapped in 3D space, and py3Dmol is used to render them as labeled spheres (for example, SingleAtomDonor in blue and SingleAtomAcceptor in red).

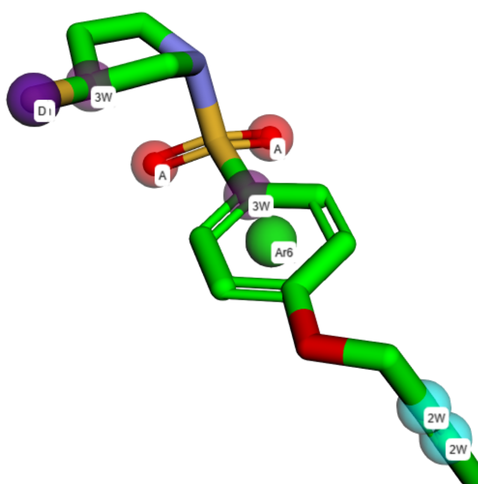

Figure 5.11: 3D molecule with pharmacophore points

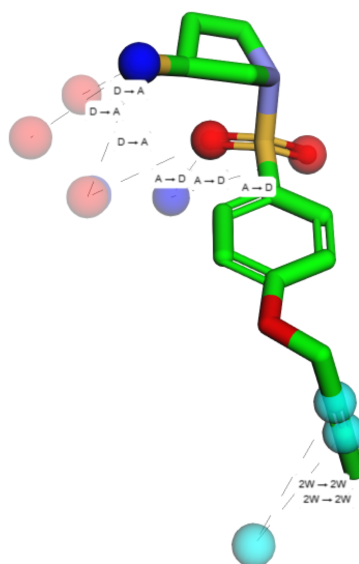

Figure 5.12: 3D Visualization of ligand and matched protein features

Interaction Matching and Contact Potentials

To form valid binding interactions, the algorithm iterates over all possible combinations of ligand and protein pharmacophore points. It filters these pairs based on two constraints:

1. Chemical Compatibility: Only valid interaction pairs are kept (e.g., Donor matched with

Acceptor, Hydrophobe with Hydrophobe, etc.).

2. Spatial Proximity: A distance threshold (4.0 Å) is applied to ensure that only features within physical interaction range are considered.

Each valid matched pair is assigned an "interaction potential" (weight) based on predefined parameters sourced from existing literature (e.g., Gaussian Boson Sampling models). A contact matrix is calculated and plotted as a heatmap to represent the strengths of different feature-feature interactions mathematically. The physical distances between these features within their separate groups (Ligand-Ligand and Protein-Protein) are also calculated for later constraint verification.

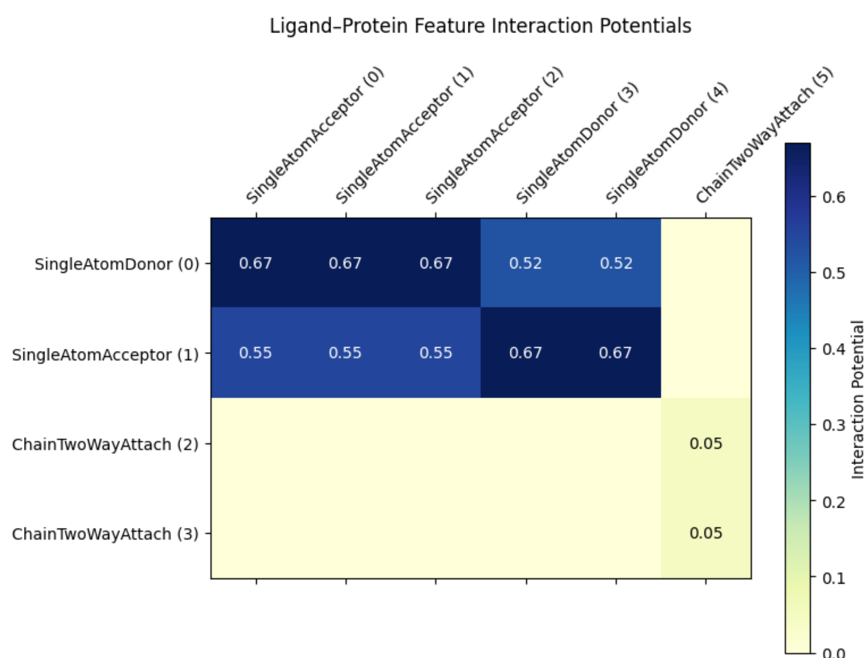

Figure 5.13: Ligand–Protein Feature Interaction Potentials

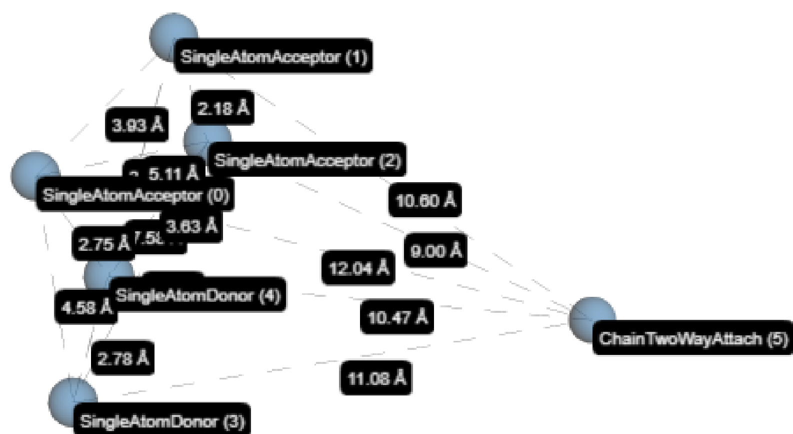

Figure 5.14: Protein feature distances

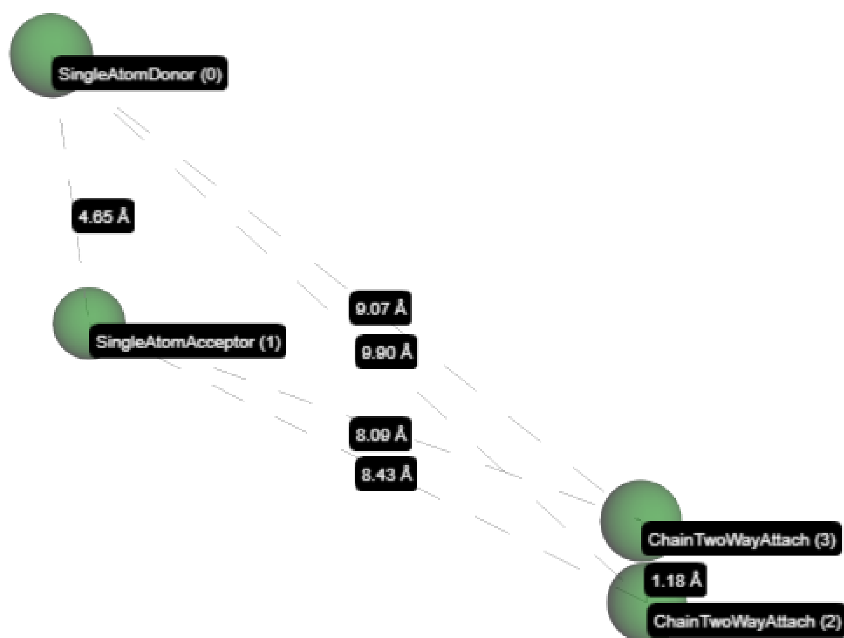

Figure 5.15: Ligand feature distances

5.3.3 Binding Interaction Graph construction

The core representation of the binding problem is formulated as a Binding Interaction Graph (BIG) using NetworkX.

- Vertices (Nodes): Each node represents a valid, matched pharmacophore pair (ligand point + protein point) and is weighted according to its interaction potential. Zero-weight nodes are discarded.

- Edges between nodes denote structural compatibility. Two contact pairs are connected if they are mutually compatible, i.e., if they form a τ-flexible contact pair. This constraint requires that the distance between the two ligand points and the corresponding distance between the two protein points differ by at most $\tau + 2\epsilon$. The optimal binding configuration is therefore the Maximum Weight Clique of this graph.

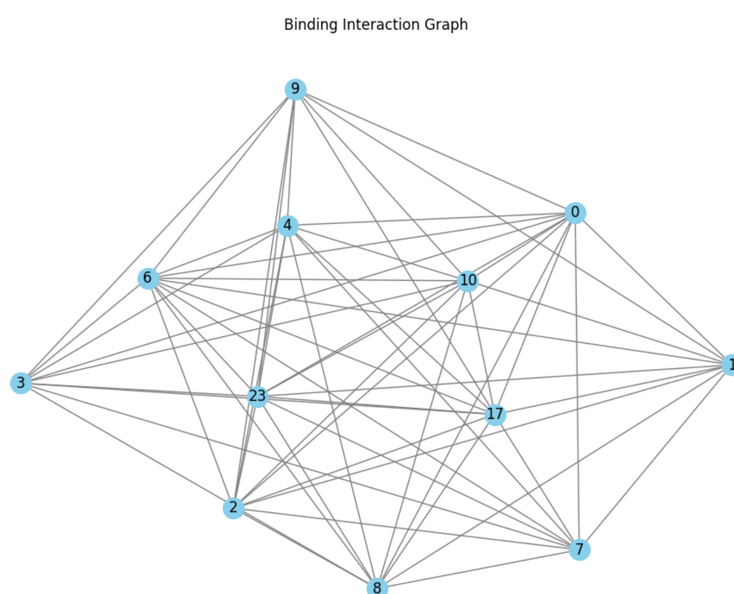

Figure 5.16: Binding Interaction Graph

5.3.4 Complementary Graph and Neutral Atom Embedding

To solve the problem with Quantum Adiabatic Algorithms (QAA) on neutral-atom architectures, the Maximum Weight Clique formulation is converted into Maximum Weight Independent Set (MWIS) by constructing the Complementary Binding Interaction Graph (CBIG).

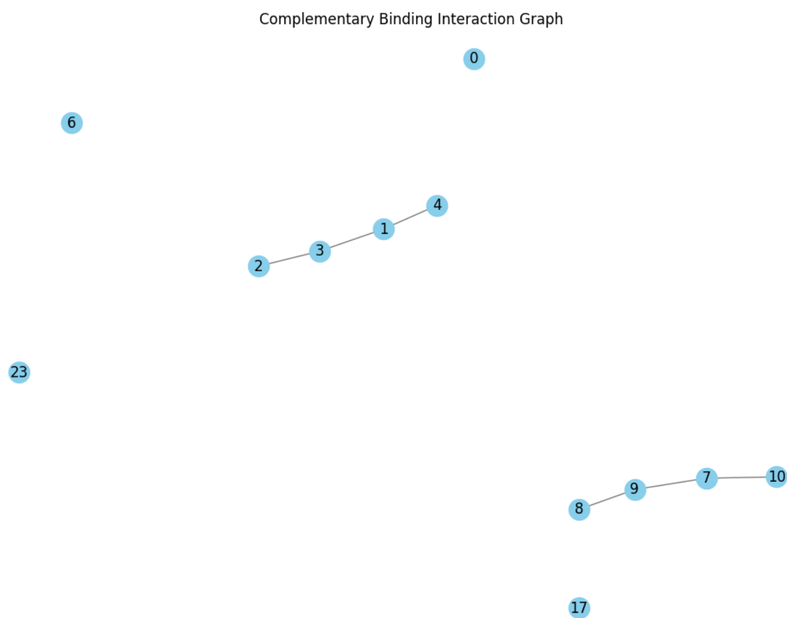

Figure 5.17: Complementary Binding Interaction Graph

Neutral-atom hardware (including Pulser simulations) requires nodes (qubits) to be embedded as 2D coordinates, with connectivity constrained by the Rydberg blockade radius. To satisfy this requirement, we use the Graph Attention Network that learns a 2D layout preserving target adjacencies while meeting hardware constraints, and outputs physical coordinates for the atomic register.

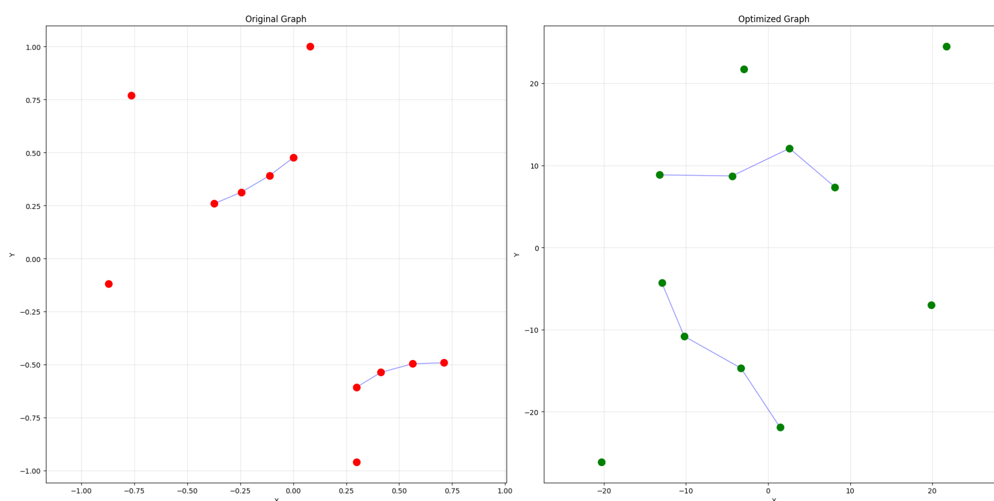

Figure 5.18: Initial and optimized graphs

5.3.5 Quantum Register and adiabatic evolution

The 2D coordinates predicted by the GAT are fed into a Pulser quantum register. A MockDevice is used to simulate the neutral-atom array. The layout is visualized together with the device Rydberg blockade radius to verify that connected nodes lie within blockade distance.

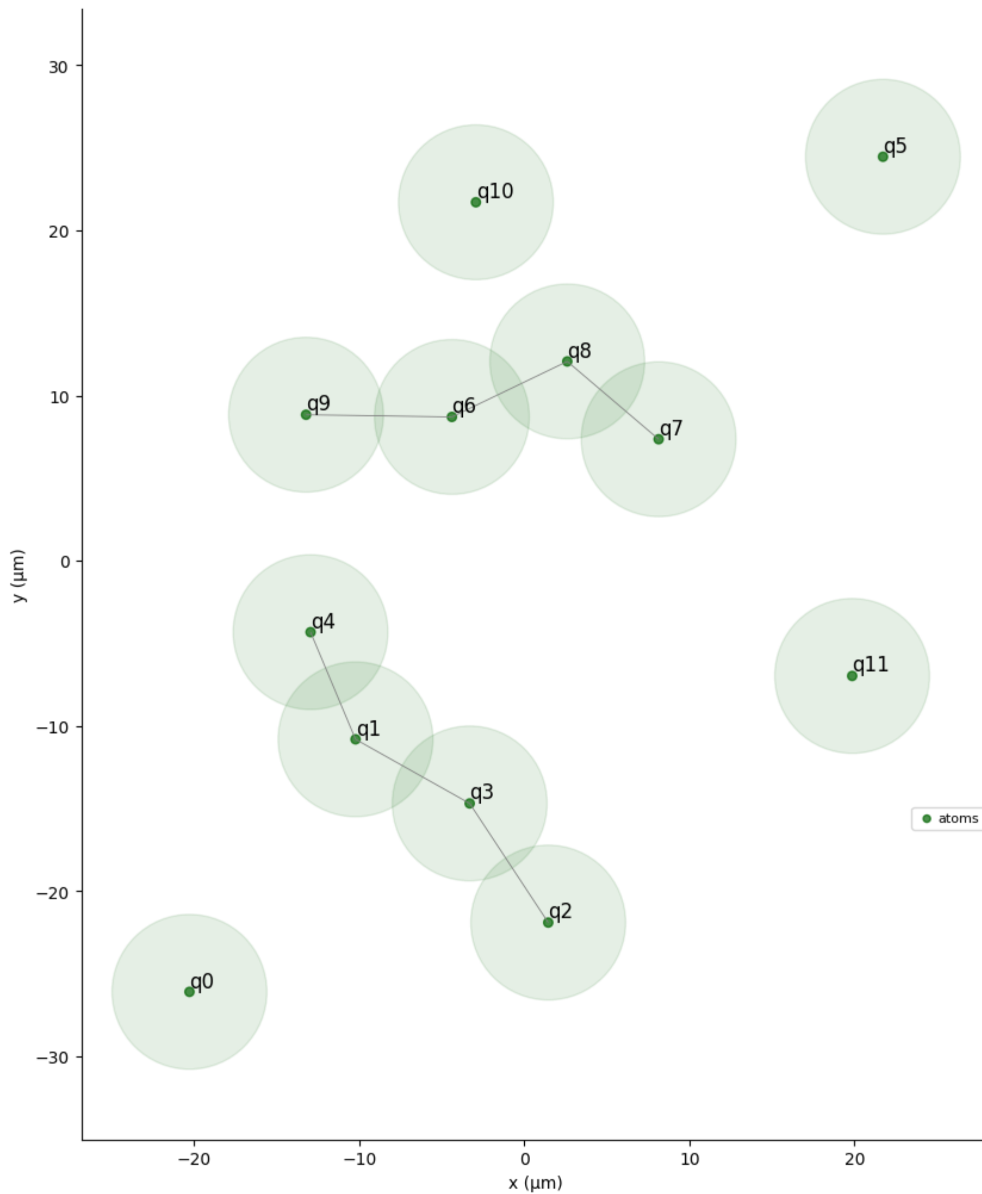

Figure 5.19: Pulser Quantum Register

Then, the adiabatic evolution drives the system toward a final state that encodes the optimal solution

to the problem and finally, the simulation produces a distribution of outcomes from which the most probable configuration, corresponding to the desired solution, is identified.

5.4 Conclusions

This thesis explored how graph machine learning and neutral-atom quantum computing can be combined to tackle hardware-constrained optimization problems, with a focus on quantum molecular docking. Chapter 2 introduced the machine learning foundations used in this work, from neural networks to graph neural networks, highlighting why graph-structured representations are essential when edge information and neighborhood relations carry most of the problem constraints. Chapter 3 presented the quantum-computing background required for the application domain, with particular attention to neutral-atom devices, their operating principles, and the practical constraints that shape algorithm design in the NISQ era. Chapter 4 developed the GEAN framework as a graph embedding autoencoder network for hardware-aware placement. The results showed that GEAN can effectively produce feasible embeddings in relevant benchmark settings and can be extended from 2D to 3D scenarios. At the same time, the protein-related experiments exposed limitations that motivated architectural refinements. Chapter 5 addressed those limitations by introducing an improved architecture with GATv2-based message passing before the autoencoder stage, so that adjacency information is explicitly integrated in the learned representation. This chapter also demonstrated the complete end-to-end pipeline for Quantum Molecular Docking, from pharmacophore extraction and binding-interaction graph construction to complementary graph embedding, neutral-atom register preparation, and adiabatic simulation. Overall, the main contribution of this thesis is a practical and reproducible workflow that connects classical graph learning to quantum-compatible problem encoding. The work shows that learning-based embeddings can be used not only as a preprocessing utility, but as a key enabling component for running graph optimization tasks on neutral-atom platforms.

5.5 Future work

Several limitations and open challenges remain, which suggest promising directions for future research. First, a significant limitation of the present work is the lack of domain-specific expertise in computational chemistry, which prevents a rigorous assessment of the adopted contact potentials. A deeper understanding of these potentials could improve both the physical reliability of the model and the effectiveness of the graph construction process. From a hardware perspective, the current analog nature of the Pasqal device imposes important constraints. In particular, the absence of partial qubit addressability limits the level of control over individual atoms, as only global Hamiltonian engineering is currently available. The transition to a fully digital architecture, expected in future hardware generations, would enable the application of local quantum gates and significantly enhance the flexibility of the approach. On the algorithmic side, although this thesis demonstrates the feasibility of finding optimal embeddings for the CBIG, the high edge density of the resulting graphs makes their mapping onto the quantum register extremely challenging, even for very small molecular instances. It remains unclear whether more refined contact potentials could be exploited to prune the graph more aggressively, or whether alternative compatibility criteria could lead to sparser and more structured binding interaction graphs. A particularly promising research direction is the development of preprocessing and graph reduction techniques tailored to this setting. In this context, the recent qReduMIS [13] framework provides a compelling reference point. This hybrid quantum-classical algorithm combines classical kernelization methods with information extracted from a quantum device to iteratively simplify the problem. In particular, it leverages quantum sampling to identify "frozen" vertices—nodes that are highly likely (or unlikely) to belong to a maximum independent set—and removes them to further reduce the graph while preserving solution quality. This strategy has been shown to significantly improve performance compared to purely quantum or purely classical approaches, especially on hard instances. Inspired by qReduMIS, future work could investigate reduction pipelines capable of simplifying CBIG graphs before embedding, or even dynamically during the optimization process. Such approaches may help bridge the gap between the size and density of real-world molecular graphs and the limited connectivity and qubit count of near-term quantum devices. In particular, integrating reduction techniques with embedding strategies for mapping graphs onto the two-dimensional atomic array

could prove essential for scaling the method.

Bibliography

[1] Leonardo Banchi et al. “Molecular docking with Gaussian Boson Sampling”. In: *Science Advances* 6.23 (June 2020). ISSN: 2375-2548. DOI: 10.1126/sciadv.aax1950. URL: http://dx.doi.org/10.1126/sciadv.aax1950.

[2] Shaked Brody, Uri Alon, and Eran Yahav. *How Attentive are Graph Attention Networks?* 2022. arXiv: 2105.14491 [cs.LG]. URL: https://arxiv.org/abs/2105.14491.

[3] Olivier Ezratty. *Understanding Quantum Technologies 2025*. 2025. arXiv: 2111.15352 [quant-ph]. URL: https://arxiv.org/abs/2111.15352.

[4] Mathieu Garrigues, Victor Onofre, and Noé Bosc-Haddad. *Towards molecular docking with neutral atoms*. 2024. arXiv: 2402.06770 [quant-ph]. URL: https://arxiv.org/abs/2402.06770.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Book in preparation for MIT Press. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[6] Loïc Henriet et al. “Quantum computing with neutral atoms”. In: *Quantum* 4 (Sept. 2020), p. 327. ISSN: 2521-327X. DOI: 10.22331/q-2020-09-21-327. URL: http://dx.doi.org/10.22331/q-2020-09-21-327.

[7] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[8] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG]. URL: https://arxiv.org/abs/1609.02907.

[9] Jure Leskovec. *Stanford CS224W, Machine Learning with Graphs*. https://cs224w.stanford.edu/.

[10] Simone Calderara. *Unimore, Machine Learning and Deep Learning*.

[11] Pak Ching Li and Stephen Gilbert. “Artificial Intelligence awarded two Nobel Prizes for innovations that will shape the future of medicine”. In: *NPJ Digital Medicine* 7 (2024). URL: https://api.semanticscholar.org/CorpusID:274280003.

[12] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN: 9781292401133. URL: http://aima.cs.berkeley.edu/.

[13] Martin J. A. Schuetz et al. *qReduMIS: A Quantum-Informed Reduction Algorithm for the Maximum Independent Set Problem*. 2025. arXiv: 2503.12551 [quant-ph]. URL: https://arxiv.org/abs/2503.12551.

[14] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[15] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML]. URL: https://arxiv.org/abs/1710.10903.

[16] Chiara Vercellino et al. "Neural-powered unit disk graph embedding: qubits connectivity for some QUBO problems". In: *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2022, pp. 186–196. DOI: 10.1109/QCE53715.2022.00038.