UNIVERSITY OF CALIFORNIA,
IRVINE

Quantum Computing for Software Engineering: Code Clone Detection and Beyond

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Samyak Jhaveri

2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

There are many people whom I am deeply grateful to for their unfailing support and encouragement over the years. In particular, I would like to thank my advisor, the advancement committee, the faculties at USC ISI, my friends, and my family.

My Advisor. I am deeply grateful to my Ph.D. advisor, Professor Cristina Lopes, to have introduced me to graduate-level research from when I was in my undergraduate program, and encouraged me to pursue a research-focused graduate program. She has always been available and open to brainstorming new ideas and deliberating atypical solutions. I am eager to continue learning from her throughout this academic program.

Faculty at USC ISI. I also want to thank Professor Srivatsan Ravi and Professor Federico Spedalieri at the University of Southern California's Information Sciences Institute in Marina del Ray, for their mentoring and granting us access to the state-of-the-art D-Wave Leap Quantum Computing Cloud Service. Access to the latest iteration of the D-Wave Quantum Annealer processor and hybrid solvers made it possible to develop and test the algorithms presented in this paper.

The Dissertation Committee members. Besides my advisor and the faculty at USC ISI, I am grateful to the committee members, Dr. Alberto Krone-Martins, Professor Sandy Irani, and Assistant Joshua Garcia for their critical remarks and constructive feedback on the thesis. I want to thank Dr. Alberto Krone-Martins, the astrophysicist who helped us start out on this quantum path, and provided me the initial insight into building intuition for quantum computing research. He has also been a guiding light on the emphasizing the importance of attention to detail as well as on strategies on how to turn a high-level intuition into a tangible research plan.

My friends and family. My family has been the greatest source of strength in the most testing times and also the anchor that has been instrumental in keeping me centered and focused on what matters. I am profoundly thankful to my mother for all the different ways she expresses her love for me, even from thousands of miles away, be it with her encouraging words, the vivid narration of stories of our memories, or the care packages she sends with delicacies that remind me of home. I am grateful to have my father be my most rigorous teacher, the strictest critic, and yet the most compassionate and caring best friend whom I can rely on and confide in. Living by his wisdom and life lessons has made it possible for me to have a fulfilling life spiritually, emotionally, socially and professionally. I also want to thank my sister, who has helped raise me into the person I am. I am inspired by her never-ending enthusiasm for her craft, and the passion with which she grows and nurtures the bonds she shares with the people she loves. She means the world to me. Lastly, I would like to acknowledge and thank the incredible friends I have made along the way, spending time with whom, almost miraculously, makes everything feel right. The trips and excursions we have been on have always added a dash of adventure and kept the spirits high.

ABSTRACT OF THE THESIS

Quantum Computing for Software Engineering: Code Clone Detection and Beyond

By

Samyak Jhaveri

Master of Science in Software Engineering

University of California, Irvine, 2023

Professor Cristina Lopes, Chair

Quantum computers are becoming a reality. They have the potential to accelerate many computationally complex processes, and also to find better results in complex solution landscapes. However, the kinds of problems which these computers are currently a good fit for, and the ways to express those problems, are substantially different from the kinds of problems and expressions used in classical computing. Quantum annealers, in particular, are an interesting kind of quantum computers to considering how they are promising in solving specific types of problems efficiently in the near term. However, they are also the most foreign compared to classical programs, as they require a different kind of computational thinking.

In my work, I have created a novel formulation of the well known software engineering problem of code clone detection by expressing it as an optimization problem in the framework of quantum annealing, a type of quantum computing. It also serves as an example of how software engineering problems can be formulated to be solved using quantum annealing. This thesis elaborates on how I rendered the code clone detection problem as a subgraph isomorphism problem and formulated it as a quadratic optimization. The formulation compares the Abstract Syntax Tree (AST) representations of two given code fragments and reports an energy value indicative of their similarity. It is then implemented on a quantum annealer.

The motivation behind this research goes well beyond code duplicate detection: this novel

approach to thinking about software engineering problems as optimization problems paves the way into expressing them as problems that an be solved using optimization architectures like quantum annealing.

Chapter 1

Introduction

The material in this thesis is based on our work in the following paper, and is included here with the permission from ACM.

- S. Jhaveri, A. Krone-Martins, C. V. Lopes, "Cloning and Beyond: A Quantum Solution to Duplicate Code", Onward! 2023: Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software in Cascais, Portugal, 32 - 49, Oct 2023. https://doi.org/10.1145/3622758.3622889.

The presence and detection of code clones in software systems has been widely studied and is acknowledged as a significant problem in software engineering [3, 57, 59, 98]. Over the years, a myriad of code clone detection techniques and tools have been proposed and implemented, differing from each other in its goals and granularity of detection. This paper explores one familiar software engineering problem in the context of quantum computers, specifically - quantum annealers [39, 91, 94]. The possibility of using quantum mechanics to either significantly speed-up calculations or to enable otherwise classically intractable problems to be solved has been the main motivation for the development of quantum computers.

Present-day quantum computers are becoming sufficiently large to undertake computational problems that may be infeasible using classical computing [22, 43, 79]. Currently, two major families of machines are being developed that exploit quantum mechanic principles to solve computational problems: quantum gate-based computers and quantum annealers [75, 76]. Both use state of superposition and entanglement of quantum bits (qubits) to represent data and operations and, albeit not yet proven, they both hold the potential to solve problems that are hard to solve otherwise, including in software engineering [97, 111]. As quantum computers become more powerful, it is important to start getting acquainted with their different ways of solving computational problems. By expressing a well-known software engineering problem as an optimization problem that can be solved using a quantum annealer, this research claims to help build bridges between what we are used to and what comes next.

1.1 Foundations

To help understand how a quantum computer is used to solve the code clone detection problem as an optimization problem, it is useful to know some terms related to the research area of code clone detection and quantum computing technology. In this section, I elaborate the following well-accepted terms and definitions that are essential in my dissertation.

1.1.1 Quantum Computing Terms and Definitions

Quantum Information Science: The study of how information is or can be encoded in a quantum system, including the associated statistics, limitations, and unique properties of quantum mechanics.

Qubit: A qubit (short for quantum bit) is the basic unit of information in quantum computing and the quantum counterpart to the bit (binary digit) in classical computing. Its value can be either 0 or 1 or a quantum superposition of 0 and 1.

Superposition: The quantum mechanical phenomenon exhibited by quantum particles, like qubits, where it exists in a combination of multiple states at the same time.

Entanglement: Quantum entanglement is a special phenomenon whereby a group of quantum particles become intrinsically interconnected, enabling them to interact and influence the quantum states of one another, even when separated by vast distances. Einstein called this "spooky action at a distance". This phenomenon is pivotal to quantum computers as entangled qubits are imbued with information to perform complex calculations. The state of such entangled qubits becomes correlated, the fundamental idea that underpins the remarkable parallelism of quantum computers, distinguishing them from classical computers.

Measurement: The act of observing the quantum state of a qubit. This measurement could be any desired interaction with the qubit, usually to detect the resultant solution. When a qubit's quantum state is measured, it's superposition collapses into a classical state of either 0 or 1 following the probability defined by its quantum state.

Collapse: The phenomenon that occurs upon measurement of a quantum system where the system reverts to a single observable state. In other words, after a qubit is put into a superposition, upon measurement it collapses to either 1 or 0.

Noise: Unwanted fluctuations in a physical system which impacts a qubit and leads to error and unwanted results. Noise can be electromagnetic charges, gravity or temperature fluctuations, mechanical vibrations, voltage changes, scattered photons, etc. As noise accumulates in a quantum system, it increases the likelihood of errors, corrupts information stored in and between physical qubits, and disrupts the quantum state in which qubits must exist to run calculations. This phenomenon is called decoherence. Because of the precise nature of qubits, they must be isolated from the environment around them. Since noise is nearly impossible to prevent, it requires substantial error-correction (to correct for the noise) in order to allow the qubits to perform desired calculations. With too much noise, a quantum computer is no longer useful.

Coherence/Decoherence: Coherence is the ability of a qubit to maintain its state over time. Decoherence generally occurs when the quantum system is interrupted by noise. Coherence time is the length of time a quantum superposition state can survive. Longer coherence times generally enable more computations and therefore more computational power.

NISQ: Noisy Intermediate-Scale Quantum, coined by John Preskill in 2018 [85], refers to the current phase of quantum computers that are not error-corrected but are stable enough to effectively carry-out a computation before the system loses coherence. A NISQ computer can be digital or analog. All commercial quantum computers operating today are considered NISQ-era machines.

Quantum Error Correction: Quantum error correction is the process of correcting errors that arise in quantum computers while running circuits. Many schemes have been proposed to do this, usually referred to as "quantum correction codes". Most quantum correction codes assemble multiple physical qubits into a single "logical qubit". Without quantum error correction, small decoherence errors can snowball over the course of a long circuit, resulting in random results.

Quantum Supremacy: Demonstrating that a programmable quantum device can solve any problem that no classical computing device can solve in a feasible amount of time, irrespective of the usefulness of the problem.

Quantum Advantage: Refers to the demonstrated and measured success in processing a real-world problem faster on a quantum computer than on a classical computer.

Quantum Speedup: The improvement in speed for a problem solved by a quantum algorithm compared to running the same problem through a conventional algorithm on conventional hardware.

Quantum Algorithm: A quantum algorithm is a set of calculations that follows the laws of quantum mechanics, including the properties of superposition, entanglement, and interference.

Spin: Spin is a quantum property that can take on discrete values, often denoted as either "spin up" ($\uparrow$) or "spin down" ($\downarrow$) to indicate the quantum state of the qubits. A qubit, when in the state of superposition, can be both "spin up" and "spin down".

Hamiltonian: A mathematical representation of the energy landscape of a physical system. In quantum mechanics, a Hamiltonian takes the form of a linear algebraic operator. The ground state (the lowest energy state) can encode the solution arrangement of variables to a computational optimization problem.

Ising model: The Ising model is a mathematical equation that represents the energy of a collection of molecules in a magnetic material, each of which has a spin which can align or anti-align with an applied magnetic field, and interact with each other through a pairwise term. The variables either representing "spin up" ($\uparrow$) or "spin down" ($\downarrow$) states that can correspond to +1 and -1 values [19]. The Ising model provides a convenient way of expressing the Hamiltonians of a quantum system as it makes it simple and mathematically tractable to describe the interaction between the qubits. The objective function of a computational optimization is expressed in a quantum variation of the Ising model known as the Quadratic Unconstrained Binary Optimization (QUBO), which I shall elaborate further in this thesis.

Adiabatic Quantum Computing: Adiabatic quantum computers harness the natural evolution of quantum states of the qubits. They are realized by quantum annealers to explore a large solution spaces by using quantum properties like superposition, entanglement and quantum tunneling. Annealing is used to harden iron, where the temperature is raised so the molecular speed increases and strong bonds are formed. The iron is then slowly cooled which reinforces these new bonds, a process called "annealing" in metallurgy. Quantum annealing works in a similar way, where the temperature is replaced by energy and the lowest energy state, the global minimum of a system, is found via annealing. Quantum annealing is a quantum computing method used to find the optimal solution of problems involving many solutions, by taking advantage of properties intrinsic to quantum physics. Since there are no "gates", the mechanics of annealing are less daunting than gate-based quantum computing. It uses the quantum adiabatic theorem to solve optimization problems [4]. Quantum annealers realize adiabatic quantum computing by harnessing the properties of the adiabatic quantum theorem of evolving the natural quantum states of the qubits to find a good solution in the solution space. The system of qubits is initialized to a state of quantum superposition, represented by an initial Hamiltonian, and then slowly "annealed", to its final low-energy state that represents the solution of the optimization, represented by the final Hamiltonian. This evolution from the initial Hamiltonian to the ground state of the final

Hamiltonian is done adiabatically, which means it occurs slowly enough that the system remains in its ground state throughout the process. This evolution cannot be faster than an inverse polynomial in the spectral gap of the current Hamiltonian. Spectral gap is the difference in energy between the lowest energy state and the next highest energy state, making it a critical factor in determining how slowly the quantum annealing can be executed. During the annealing process, the system avoids being stuck in local minima by using the property of quantum tunneling, and “tunneling” through the potential energy barrier - closer to the global minimum energy state which represents the best or a good-enough solution.

1.1.2 Code Clone Detection Terms

Code Fragment or Block: A continuous segment of source code.

Clone Pair: A pair of code fragments that are similar.

Abstract Syntax Tree (AST): A graph representation of the tokens of a code fragment in the form of a tree data-structure, arranged such that the nodes and edges represent the semantic structure of the code fragment for the compiler to understand.

1.1.3 Code Clone Types

Code clones are classified into two prevalent categories – syntactic and semantic code clones. Syntactic clones are comparable based on their textual structure [2, 92]. Conversely, semantic clones are functionally similar [42], but syntactically different. The code clone detection literature identifies four types of code clones [14], wherein the first three types pertain to syntactic clones and the fourth type pertains to semantic clones:

(i) **Exact Clones (Type 1)**: These are identical code fragments, except for variations in comments, layouts, and whitespaces.

(ii) **Renamed Clones (Type 2)**: These are code fragments that are syntactically or structurally similar, except for comments, identifier names, and literal names.

(iii) **Near Miss Clones (Type 3)**: These are code fragments that are essentially Type 2 clones and, additionally, have undergone modifications such as the addition or removal or reorder of statements. Despite these modifications, their outcomes are similar, if not the same.

(iv) **Semantic Clones (Type 4)**: These are code segments that are functionally similar but implemented using different syntax. There may be more than one code segment that fits this description [3].

1.2 Motivation

The detection and removal of code clones have been widely studied and is acknowledged as a significant software engineering research problem, as it can be detrimental to the quality, maintainability, and evolution of software systems [3, 57, 91, 93, 95, 98]. Type 3 clones are of particular interest in situations such as code theft, where the source code remains essentially preserved but subjected to the aforementioned transformations. Detecting Type 3 clones, however, is computationally very intensive. Consequently, several approaches and heuristics have been proposed in the literature to mitigate its computational costs [50, 69, 100]. Abstract Syntax Tree (AST) representations of source code has been used for subtree comparisons [11, 105, 109]. In AST-based techniques, code clones are detected by measuring the similarity between corresponding ASTs or their subtrees. In previous research on comparing the ASTs of two code fragments to detect if they are code clones, a distance value that quantifies the similarity between the two code fragments is generated and presented [52, 55]. Code clone detection has also been solved using Program Dependency Graphs (PDGs) [49]. Both AST- and PDG-based approaches are equivalent to the subgraph isomorphism problem. While using ASTs or PDGs as a basis for code comparison is beneficial in terms of precision/recall, those approaches result in clone detection tools that are too slow, requiring aggressive heuristics to make them feasible in practice [50, 101].

One way of speeding up clone detection is to forego graphs altogether and use tokens as the basis of comparison, but those approaches typically suffer from false positive and false negative errors.

This is where the idea of leveraging quantum computing for solving a formulation of code clone detection comes in. The ability of quantum computers to harness the principles of quantum mechanics, such as superposition and quantum entanglement, in novel and interesting ways, forms the basis of an immensely powerful form of computation, provided a system is built that can be easily manipulated and measured. While Richard Feynman [40] is often credited with the conception of using quantum computers for simulating nature, there were several researchers who anticipated the idea. This possibility has been meticulously studied since the first early works and proposals by eminent physicists and computer science pioneers such as Richard Feynman [40], Paul Benioff [15], Yuri Manin [71], David Deutsch [34], among others.

A quantum annealer can solve a computational problem if it can be formulated as an optimization problem and mapped into a physical adiabatic quantum system of qubits [96]. In the realms of software engineering and computer science, several problems can be mathematically cast as some form of optimization suitable to map onto quantum annealers.

Quantum annealing changes the solution space for Type 3 clone detection. Rather than foregoing graphs or using aggressive heuristics to make them practical, we can formulate the problem as an optimization problem, and then use a quantum annealer to find the optimal solution. Since several demonstrations have established the efficacy of quantum annealers in solving hard problems related to large graphs like graph community detection [90], graph isomorphism [112], minimum vertex cover problems [84], I formulated the code clone detection problem as an optimization problem that can be solved using a quantum annealer. Additionally, my approach can be applied to many more graph-based software problems as mentioned earlier in the thesis, not just clone detection.

The earlier findings I mentioned about the promising potential of quantum annealers to solve hard graph problems, combined with the intuition I cultivated through an in-depth exploration of the intricacies involved in implementing quantum annealing algorithms, and the voracity to substantiate the utility of quantum annealing as a means of addressing software problems, especially in the context of code clone detection, make it desirable to pursue this research.

1.3 Thesis

In my work, I have created a novel formulation to solve the code clone detection problem by expressing it as an optimization problem in the framework of quantum annealing. This thesis elaborates on how the formulation has been crafted and the empirical evaluation of its effectiveness by implementing it using a quantum annealer. I formulated the code clone detection problem within the Quadratic Unconstrained Discrete Optimization (QUDO) framework as a graph/subgraph isomorphism problem and solved it on actual quantum annealer hardware.

I state my thesis as follows:
It is possible to formulate the software code clone detection problem as an optimization problem, that can be solved on quantum annealing computers. Furthermore, this approach is computationally scalable with the size of the input code fragments being compared.

This thesis statement is claimed for problem instances that can fit onto the Quantum Processing Unit (QPU).

Additionally, I believe that my approach can be applied to many software engineering problems. In software engineering literature, several heuristic optimization formulations such as resource allocation, feature selection, and optimal product feature selection problems [27, 48, 51, 53] are, in many cases, formulated as search-based problems solved via optimization [28, 46]. For instance, code module optimization is another important area of software engineering research: by representing each module or any part of the code as a binary variable, one can optimize a code base by choosing the minimal combination of modules that satisfy a certain set of requirements and constraints. Similarly, one can also represent alternative formulations of a code and/or choice of software optimization and compilation techniques as binary variables and cast an objective function that attains a minimum value for maximum overall code performance while minimizing resource consumption.

Beyond the aforementioned optimization and constraint satisfaction, there are several other examples of software engineering problems that can be mapped to knapsack [12, 65, 67], set partitioning [82], max clique [26], max cut [72], graph coloring [104], SAT [86] problems, which can all be easily mapped and solved on quantum annealers.

1.4 Contributions

The contributions of my work are as follows:

1. Building on a prior formulation of graph isomorphism as a Quadratic Unconstrained Binary Optimization (QUBO) by Zick et. al. [112] and QUDO, I show how to formulate the *subgraph* isomorphism problem as a Quadratic Unconstrained Discrete Optimization (QUDO) such that it is amenable to an adiabatic Quantum Processing Unit (QPU).
2. I introduced a method to incorporate additional node type-based constraints into the problem formulation, a critical element of most software graphs.
3. Empirically validated the scalability of using quantum annealing for code clone detection for small code fragments.
4. I show how optimization algorithms formulated for quantum annealing can be improved for graph-based problems by adding bespoke constraints tailored to the problem at hand to guide the annealing process towards desired results at a scale suitable for solving practically big enough problems in the future.

Chapter 2

Background and Related Work

This thesis centers on the formulation of software code clone detection as an optimization problem and then solving it using a quantum annealer. It assumes the reader has a foundational understanding of code clones. This includes their genesis, the issues arising from unintentional clones in systems, the utility of clone detection techniques and tools, and the metrics used for evaluating these techniques. If not, much information can be found in [3, 13, 14, 78, 88, 91]

2.1 An Overview of Quantum Annealing Computers

While there are debates among researchers about the advantages offered by currently available Noisy Intermediate-Scale Quantum (NISQ) computers, the prospect of harnessing quantum technology to solve real-world problems by developing quantum algorithms is an area of vigorous research [85].

The family of quantum computation devices commonly called gate-based quantum computers relies on quantum mechanics for the implementation of quantum logic gates [10, 80]. These gates are the quantum analogue of Boolean logic gates used in classical computing. The distinguishing factor for quantum logic gates, from their classical counterparts, is the fact that they employ quantum properties such as superposition and entanglement for conducting computation. Quantum gates can be arranged in a quantum circuit to manipulate qubits to solve complex computations similarly to how one would program a classical computer by combining Boolean logic gates [37].

Currently, machines such as the IBM Q [54] are only available with a small number of qubits, and they are still heavily affected by noise [85]. Quantum computers are fundamentally noisy in nature and incur errors with each operation. To remedy noise and make the computer run as desired, one option could be quantum error correcting codes. In spite of significant advances in quantum technologies over the last decade, this type of error correction is currently out of reach. As such, huge efforts have been devoted to finding out whether current noisy quantum technologies could provide a practical advantage over classical computers without error correction [103]. The emphasis here is on practicality, since expectation values in shallow quantum circuits can formally be computed in polynomial time on a classical computer [21, 108]. This limits their current adoption to small toy problems, experiments, and algorithmic prototyping [1].

Adiabatic quantum computing is another type of quantum computing realized by a *quantum annealer*. Quantum annealing was proposed in [41] and [58] as a method to optimize discrete energy functions. It harnesses the natural evolution of quantum states. This is unlike gate-based quantum computers as they are designed to manipulate the quantum state to arrive at a solution, which could be more delicate to work with, but can be used to solve a broader range of problems. Quantum annealers employ qubits to explore and optimize complex solution spaces by exploiting their quantum properties like superposition, entanglement, and other quantum phenomena such as quantum tunneling and the quantum adiabatic theorem of quantum mechanics [75] [see Figure 2.2]. Quantum superposition of the qubits allows quantum annealers to explore multiple solutions at once, while quantum tunneling allows qubits to escape from local minima (suboptimal solutions) and reach global minima (optimal solutions). These quantum machines solve optimization problems [4] by evolving a physical system towards its ground state or minimum energy state, finding the low-energy states of a problem and therefore the optimal or near-optimal combination of elements [107, 7, 23, 36, 38, 81]. Optimization challenges encompass mathematical problems that involve finding the best solution from a set of possible solutions. Note that, like gate-based quantum computers, quantum annealers are also affected by noise and require error correction to provide reliable solutions. Nevertheless, despite the presence of noise in quantum annealing computers, and the fact that many combinatorial optimization problems are typically NP-hard, or close, some quantum algorithms developed for quantum annealing computers have demonstrated promise for some combinatorial optimization problems, like graph isomorphism [9, 47, 77, 89] or job shop scheduling [25].

BQP is a complexity class of decision problems that can be efficiently solved by a quantum computer using a polynomial amount of computational resources and with a bounded probability of error [8, 36, 61] [see Figure 2.1]. The exact power of BQP is still an open question in quantum complexity theory. However, BQP is widely believed to capture the power of quantum computing, meaning that any problem that can be efficiently solved by a quantum

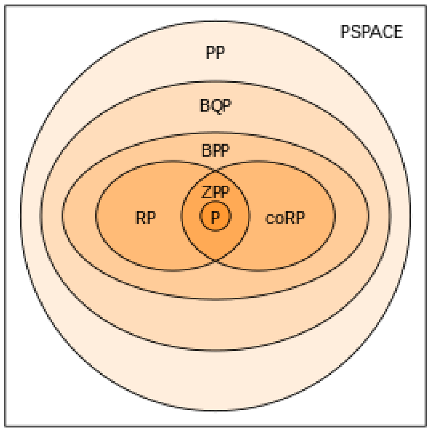

Figure 2.1: BQP Complexity Class in P Space.

computer can also be solved by a quantum computer in BQP [16].

Programming quantum annealers, however, requires casting the required computation as a specific form of mathematical optimization problem, which are typically binary or discrete combinatorial problems [17, 68, 87].

D-Wave Systems, Inc. has been at the forefront of commercial quantum annealing computers, creating several iterations of quantum annealers that realize this specialized variant of the Quantum Adiabatic Algorithm (QAA) [20], proposed by Farhi et al. [38].

Quantum annealing commences with a system of a group of qubits in a superposition state, meaning they are in a combination of all possible states simultaneously. The system undergoes a gradual and slow process of “annealing”, ultimately evolving into a low-energy state that represents the optimal solution to the problem. The annealing process is controlled by a parameter known as the annealing time, which determines how slowly the system evolves from an initial state to a final state.

Typically, optimization problems are encapsulated mathematically by an objective function that needs to be minimized or maximized. When expressed as a quadratic mathematical model of the energy of the system, the optimization problem can be embedded onto the

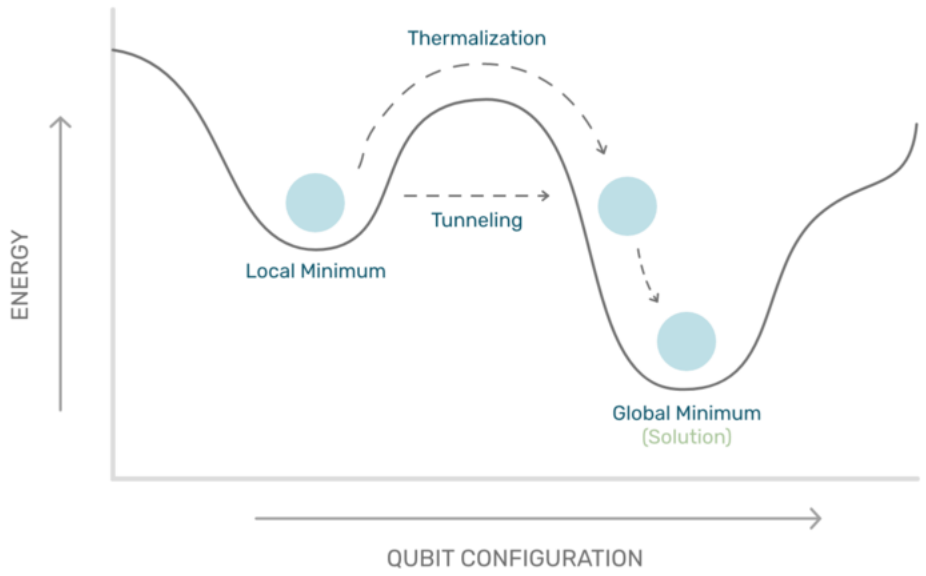

Figure 2.2: Quantum Annealing, Visualized.

qubits of the Quantum Adiabatic Annealer. These qubits, encoded with the model are then subjected to the annealing process where they are evolved from an initial state of superposition to a final state that is apossible optimum configuration of the states of the qubits that represents the best or good enough solution to the problem. This result, ideally, is found at the global minimum energy state of the system of qubits.

2.2 Conceptual Framework for Quantum Annealing

To effectively understand the approach of formulating optimization problems for quantum annealers, one must consider a conceptual framework that encompasses three essential components governing the practical implementation of quantum annealing - the fundamental quantum physics at the qubit level that underpins quantum annealing, the mathematical models that articulate the objective function of the optimization problem, and the software and hardware environment of the annealer that orchestrates computation, submitted as problems, on its quantum and classical computing components across different levels of the software stack. Developing a sense of these interconnected components helps build intuition for effectively leveraging the capabilities of quantum annealers in solving complex optimization problems.

2.2.1 Underlying Quantum Physics of Quantum Annealing

This subsection sheds light on the quantum physics principles that governs the quantum annealing process in a D-Wave quantum annealer at the qubit level.

Qubits in a Quantum Processing Unit (QPU) are synthesized by supercooling a thin metal wire to make it superconducting loops. These superconducting wires have a circulating current and a corresponding magnetic field which stimulates it to exhibit quantum properties. Unlike classical bits that are distinctly either 0 or 1, a qubit can be in a superposition of 0 state and the 1 state at the same time. At the end of the quantum annealing process, each qubit collapses from superposition state into either a 0 or 1 (classical) state.

Visualizing the quantum annealing process is facilitated by an energy landscape that evolves dynamically over time. This is depicted in (a), (b), and (c) in Figure 2.3 where the stages represent the different phases of the annealing process.

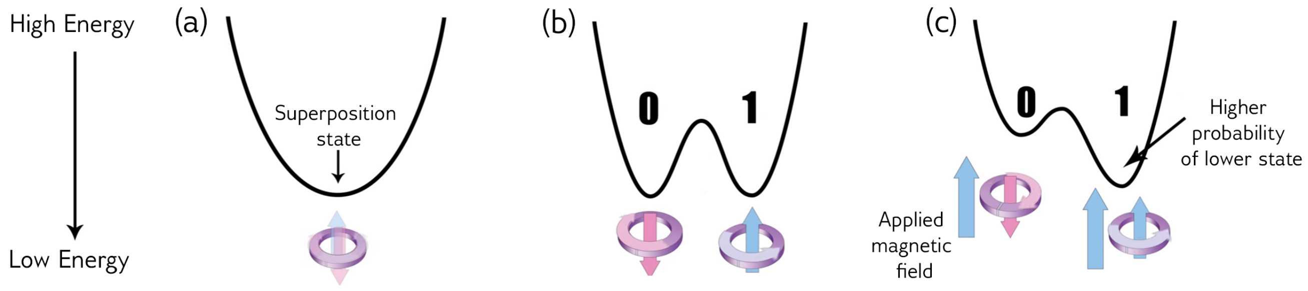

Figure 2.3: Energy diagram changes over time as the quantum annealing process runs and a bias is applied.

Initially, the energy landscape presents a single valley (a), indicative of a single minimum energy state. As the quantum annealing progresses, the energy barrier is raised, thus transforming the landscape into a *double-well potential* (b). Here, the low point of the left valley corresponds to the 0 state, and the low point of the right valley corresponds to the 1 state. In this situation, the probability of the qubit falling into the 0 or the 1 state is equal (50%). The probability of the qubit falling into either states at the end of the anneal is a tunable parameter controlled by the application of an external magnetic field (c). This field tilts the double-well potential either towards 0 or 1, increasing the probability of the qubit ending up in the lower well. The programmable quantity that controls the external magnetic field is called the *bias*.

However, the bias term alone is not useful. The real power of qubits as a computational unit is exercised when they are entangled so they can influence each other. This is done with a device called a *coupler* which can make two qubits tend to align either in identical states or opposite states. Like a qubit bias, the coupling strength is also programmable, allowing for customized correlation weights between coupled qubits.

Considering a pair of entangled qubits, we can represent them as a combined entity with four possible states - each corresponding to a different combination of the qubits: (0, 0), (0, 1), (1, 0), (1, 1) [see Figure 2.4]. The relative energy of each state are tailored by the biases of qubits and the coupling between them. In Figure 2.4, during the anneal, the qubit states

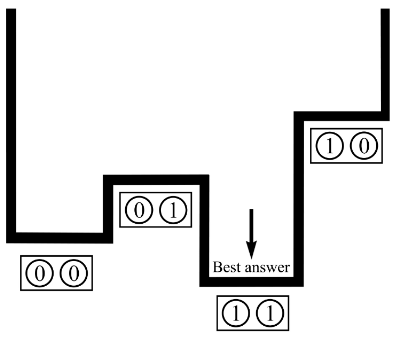

Figure 2.4: Energy Diagram showing the best answer a 2-qubit entangled system.

are potentially delocalized in this landscape before finally settling into (1,1) at the end of the anneal. The programmable biases and couplings define an energy landscape, and the quantum annealer is guided to settle into the lowest energy configuration at the minimum energy of that landscape.

2.2.2 Mathematical Models of Quantum Annealing

The Hamiltonian

A Hamiltonian is a mathematical description of the total energies of a physical system, encapsulating the state of the energy of a system whether comprised of a singular element or a composition of many elements, and yields the energy of the system for any given state. For most non-convex Hamiltonians, finding the minimum energy state is an NP-hard problem.

For a quantum system, a Hamiltonian is a transformative function that maps certain energy states, called *eigenstates*, to discrete energies. Only when the system is in an eigenstate of the Hamiltonian is its energy well-defined and called *eigenenergy*. When the system is in any other state, its energy is uncertain. The collection of eigenstates with defined eigenenergies make up the *eigenspectrum* [35].

The Hamiltonian characterizing a quantum annealing process is given as:

$$\mathcal{H}_{ising} = \underbrace{-\frac{A(s)}{2}\left(\sum_i \hat{\sigma}_x^{(i)}\right)}_{\text{Initial Hamiltonian}} + \underbrace{\frac{B(s)}{2}\left(\sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j}\hat{\sigma}_z^{(i)}\hat{\sigma}_z^{(j)}\right)}_{\text{Final Hamiltonian}} \tag{2.1}$$

In this equation, $\hat{\sigma}_{x,z}^{(i)}$ represent the Pauli matrices operating on the state of the qubit q_i, and h_i and $J_{i,j}$ represent the qubit biases and coupling strengths, respectively.

The Hamiltonian is the sum of two distinct terms:

- Initial Hamiltonian (first term), or the *tunneling* Hamiltonian, establishes the ground state where all the qubits exist in a superposition state of 0 and 1.
- Final Hamiltonian (second term), known as the *problem* Hamiltonian defines the ground state which is the solution to the optimization problem at hand. The final state is a classical state, and includes the qubit biases and the couplings between qubits.

During quantum annealing, the system starts in the lowest-energy ground eigenstate of the initial Hamiltonian. It then undergoes an evolution where the problem Hamiltonian, which contains the biases and coupling strengths, is gradually introduced into the system. Slowly enough, the problem Hamiltonian supersedes the initial Hamiltonian and reduces its influence. This transiting, when performed without external perturbations and at a suitably slow pace, is called an adiabatic process, giving rise to the term "adiabatic quantum computation". Upon completion of the annealing cycle, the system is expected to be in an eigenstate of the problem Hamiltonian, ideally, residing in the minimum energy state throughout the quantum annealing process thereby presenting a solution to the optimization problems. By the end of the anneal, each qubit is a classical object. Because no real-world computation can run in perfect isolation, quantum annealing may be thought of as the real-world counterpart to adiabatic quantum computing, a theoretical ideal.

Each optimization problem necessitates a unique Hamiltonian construction, leading to a distinctive eigenspectrum to be explored for solutions.

The Ising Model

The Ising model is a mathematical model of ferromagnetism that represents the energy of a collection of elements, usually molecules, using discrete variables to denote magnetic dipole moments of spin states of the elements. These elements can either align or anti-align with an externally applied magnetic field, engaging in interactions with each other that are characterized by a pairwise term. The variables either represent "spin up ($\uparrow$) or "spin down" ($\downarrow$) states denoted by +1 and -1 values [19], respectively. The Ising model provides a convenient way of expressing the Hamiltonians of a quantum system as it makes it simple and mathematically tractable to describe the interaction between the qubits.

For solving optimization problems using quantum annealing computers, the objective function must be formulated as a Hamiltonian of the Ising model. The objective function expressed as an Ising model is as follows [110], with h_i representing the linear coefficients corresponding to qubit biases, and $J_{i,j}$ representing the quadratic coefficients corresponding to coupling strengths. Here, the binary variables s_i and s_j may take values from $\{-1, +1\}$.:

$$H_{ising}(s) = \sum_{i=1}^{N} h_i s_i + \sum_{i=1}^{N} \sum_{j=i+1}^{N} J_{i,j} s_i s_j \tag{2.2}$$

QUBO

Quadratic Unconstrained Binary Optimization (QUBO) is a mathematical framework for solving combinatorial optimization problems that involve binary variables (variables that can take on only two possible values, typically 0 or 1) [19]. QUBO models can embrace an exceptional variety of important combinatorial optimization problems found in industry, science and government, as documented in studies such as [5] and [63]. In QUBO, the optimization problem is formulated as a quadratic equation, where the variables represent the binary decision variables and the coefficients of the quadratic terms represent the costs or penalties associated with particular combinations of variable values.

Quantum annealers are designed to solve Ising and QUBO models. Through special reformulation techniques that are easy to apply, quantum annealers can be used to efficiently solve many important problems once they are put into the QUBO framework [44]. The goal is to find the configuration of qubits that minimizes the energy of these models.

The following describes the QUBO Model [32, 63, 64], where the binary variables x_i and x_j may take values from $\{0, 1\}$.

$$f = \min_x \left(\sum_{i=1}^{n} p_i x_i + \sum_i \sum_{j>i} q_{i,j} x_i x_j + c \right)$$

The coefficients p_i and $q_{i,j}$ are constant real numbers that define our problem, as is the constant c. The binary variables x_i and x_j are the values that we are looking for to solve our problem. The best solutions for these variables is the value for each x_i and x_j that produces the smallest value for the overall expression. The QUBO equation has three main components:

1. Linear Term - The first summation, $\sum_{i=1}^{n} p_i x_i$, with each component having just one variable.

2. Quadratic Term - The second summation, $\sum_i \sum_{j>i} q_{i,j} x_i x_j$, with each component in the summation containing a product of two variables.

3. Constant c - This constant makes no difference in the results.

The Ising and QUBO models are mathematically equivalent – one can be translated into the other with a simple substitution. To transform an Ising model to QUBO, the following substitution is used:

$$s_i \mapsto 2x_i - 1 \tag{2.3}$$

And to transform a QUBO model to an Ising model, the following substitution is used:

$$x_i \mapsto \frac{s_i + 1}{2} \tag{2.4}$$

QUBO is *binary*, because the variables take only two possible values; it is *quadratic* in the sense that it represents an optimization problem that involves *pairwise interactions* between the variables of the problem.

The goal of a QUBO model is to find the assignment of variable values $(x_1, x_2, x_3, ..., x_n)$ that minimizes the overall cost or penalty of the quadratic equation. Constraints are the rules that determine which solutions are acceptable. The cost function or penalty function is typically a sum of the quadratic terms, where the coefficients (p_i and $q_{i,j}$) of the terms represent the costs or penalties associated with particular combinations of variable values.

In graph problems, for example, the binary variables can represent the existence (1) or absence (0) of edges, or the existence/absence of a node-to-node correspondence between two graphs, or a number of other binary properties.

An optimization problem expressed as a QUBO, or QUDO, can be solved on a quantum annealer if the variables are *binary* (0 or 1) or discrete, the optimization objective function is quadratic, and/or the constraints can be effectively expressed as linear or quadratic terms in the objective function [107]. Solutions to this objective function are found by sampling it on the QC hardware, which can either be purely quantum, purely classical, or a hybrid of both classical and quantum working in tandem.

QUDO

QUBO formulations are encoded in to the quantum annealer as a Binary Quadratic Model (BQM). BQMs are too low-level for many problems as they can only yield results as binary variables. However, in some cases it may be more practical to use a higher-level formulation where the results can be discrete variables. A Quadratic Unconstrained Discrete Optimization (QUDO) (note: not Binary, but Discrete Optimization) can be formulated that can be defined over *discrete* values such as Paris, Rome, Madrid, Athens} or {1, 2, 3, 4}. A QUDO can be encoded in a quantum annealer as a Discrete Quadratic Model (DQM). The possible values that these discrete variables can take are called the variable's cases. DQMs build on top of BQMs by substituting each discrete variable with a vector of binary variables using one-hot encoding. [1]

QUDO models are a well-suited for graph problems. For instance, an edge in a graph can be depicted as a connection between two nodes, x_i and x_j, with the coefficient $q_{i,j}$ representing the interaction strength of the coupling between the qubits that are encoded as the member nodes of that edge.

Researchers have successfully reformulated various problems as QUBO or QUDO equations

[1] `https://docs.ocean.dwavesys.com/en/stable/concepts/dqm.html#discrete-quadratic-models`

for quantum annealers to solve for both research and commercial applications [18, 56, 83]. The most significant contribution of this research thesis is the novel approach to solving the code clone detection problem by translating it into an optimization problem and formulating a discrete quadratic model that is solved on an actual D-Wave quantum annealer.

For this reason, this work uses QUDO models, which are then automatically translated into QUBOs [74].

2.2.3 Submitting Programs to a Quantum Annealer

The D-Wave Advantage_system4.1 system employed in this study comprises a QPU with over 5,000 qubits and at least 35,000 couplers. This configuration results in a high coupler-to-qubit ratio of 15, as reported in [30]. Interaction with the D-Wave QPU and hybrid solvers is streamlined through the D-Wave Leap Cloud Quantum Computing service and the accompanying open-source Ocean software development kit [30, 31].

The D-Wave processor minimizes a set of input coefficients in the formulated QUBO or Ising model and returns a set of qubit values representing a potential optimal solution, which is referred to as a "sample." Typical queries generate a 100 samples, although some cases may demand up to a 1,000 samples for a single problem submitted. The Ocean SDK translates Ising and QUBO models into matrices and vectors for embedding onto the QPU chip. It also provides the annealing results and relays metadata about the annealing process for that particular formulation back to the client. Within this framework, a "solver" is a resource that executes a problem, whereas "samplers" are processes that run multiple iterations of a problem to obtain a collection of samples, each of which is a potential solution to the problem. For convenience, this thesis refers to both solvers and samplers under the umbrella term "Ocean's solvers" [31].

The increase in the number of code lines in input fragments proportionately increases the size of their abstract syntax trees (ASTs), resulting in larger input graphs. This increase in the number of nodes and edges of the input graphs necessitates additional comparisons between the two code fragment graphs, in turn escalating the number of variables in the corresponding Ising or QUBO Hamiltonian. To address this issue of scaling, D-Wave Leap offers a Hybrid Solver Service (HSS) that combines the power of CPUs, GPUs, and QPUs, that operate in tandem in portfolios depending on the type of input they get and the size of the problem they need to solve. This service dynamically adapts the distribution of tasks to obtain a solution to a problem amongst the three processing units. It harnesses the QPU's exceptional ability to quickly identify solutions, extending this feature to a wider range of inputs than what is typically feasible with just the QPU. Users can submit problems as Binary Quadratic Model (BQM), Discrete Quadratic Model (DQM) or a Constrained Quadratic Model (CQM), depending on the kind of inputs the problem requires. In this implementation, I found the Discrete Quadratic Model to be most suitable for the application of graph isomorphism as it facilitates the formulation and execution of QUDOs in the D-Wave Hybrid Solvers. The *hybrid_discrete_quadratic_model_version1* solver of the Ocean SDK's DQM suite adeptly handles discrete problems, allowing for QUDO variables to have any integer value instead of being restricted to binary values (0 or 1) as is the case for QUBOs using the BQM.

The HSS is particularly valuable for tackling combinatorial optimization problems that are too large to fit onto current-generation QPUs. From a performance perspective, using hybrid workflows combining the best features of classical and quantum computation can often produce better results than either approach used alone. There are also fast and effective post-processing utilities in place in these QPU systems, such as Majority Voting (MV) among other heuristic methods, to boost the quality of raw QPU results by repairing broken chains in the output solutions. The MV post-processing, which is the simplest one and is thus invoked by default in some QPU-based Ocean solvers, assigns a value to each logical node based on a "vote" of the qubits in its corresponding chain (breaking ties at random). Thus, the MV post-processor converts all broken solutions into intact solutions, typically in just a few milliseconds of additional computation time [30].

Additionally, the HSS includes software that manages the low-level operational parameters of the annealing process, eliminating the need for specialized knowledge of QPU control fine-tuning. The cloud-based platform allows simultaneous operation of multiple portfolio solvers, ensuring that the optimal solution is selected from a collective pool of results [74].

2.3 Graph and Subgraph Isomorphism

2.3.1 Graph Isomorphism

In simple terms, Graph Isomorphism (GI) is a computational problem that involves determining whether two given graphs, denoted as G_1 and G_2, are structurally identical, such that the adjacency matrices can be identical with a relabeling of the vertices. The solution to be expected upon solving an instance of graph isomorphism for a pair of graphs, is a mapping that matches the vertices and edges of one graph with the other. Although this task might appear straightforward, the computational complexity escalates with the graphs' size, i.e. the number of vertices in the graphs, making it far from trivial. The GI problem shares a computational complexity profile with the integer factoring problem, the first and most famous problem for which a quantum algorithm has demonstrated an exponential speed-up over the best classical algorithm [99]. Notably, while the integer factoring problem can be solved in polynomial time using Shor's algorithm [99], a similar feat is rarely seen for the GI problem. Like factoring, the common belief is that the GI problem is unlikely to be NP-complete [47].

Graph isomorphism by itself an interesting problem to solve. It is of practical interest as it is encountered repeatedly in many applications in like verifying circuit equivalence in very large-scale integrated circuit design [66], in biology, bio-informatics and drug discovery where a graph representing a molecular compound is compared with a entire database of molecule graphs for search [77], and also in mathematics, supply chain, computer vision and pattern matching, among many others [112]. It is repeatedly studied because of its broad range of applications [62].

Now, delving into the details of the graph isomorphism problem, it entails determining whether two given graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, each having n vertices, are

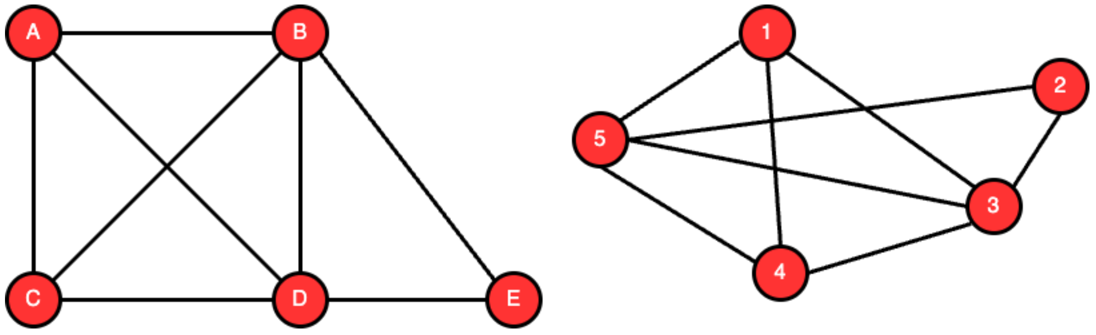

Figure 2.5: Graph Isomorphism Example, G_1 (left) and G_2 (right).

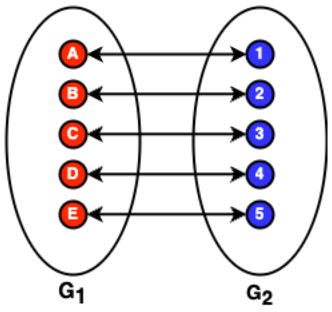

Figure 2.6: Bijective Mapping between G_1 (left) and G_2 (right).

isomorphic or not. The set of edges E consists of unordered pairs of the finite non-empty set of vertices V such that $u, v \in V$. The adjacency matrix A of a graph G with n vertices is an $N \times N$ matrix based on vertex labels, where $A_{ij} = 1$ if an edge connects vertices i and j, and zero otherwise. The vertices are labelled as $V = \{v_i | 0 \leq i < n\}$, and edges are denoted by $e = uv$ or $e = \{u, v\}$, with graph's size denoted by the number of edges, m.

A minor embedding of a graph $G_1 = (V_1, E_1)$ onto a graph $G_2 = (V_2, E_2)$ is a function $f : V_1 \rightarrow V_2$ that satisfies the following three conditions:

1. The sets of vertices $f(u)$ and $f(v)$ are disjoint whenever $u \neq v$.
2. For all vertex $v \in V_1$, there is a subset of edges $E' \subseteq E_2$ such that $G' = (f(v), E')$ is connected.
3. If $\{u, v\} \in E_1$, then there exist $u', v' \in V_2$ such that $u' \in f(u)$, $v' \in f(v)$ and $\{u', v'\}$ is an edge in E_2.

For the minor embedding f defined above, G_1 is referred to as the *guest* graph while G_2 is called the *host* graph [24].

For two given graphs G_1 and G_2 to be isomorphic, they would have to be of the same order and the same size, i.e., they would have to have same number of vertices and edges, respectively. If the graphs are isomorphic, the output is a bijective edge-invariant vertex mapping $f : V_1 \rightarrow V_2$ (which is calculated); edge-invariant means that for every pair of vertices $\{u, v\}$, we have $uv \in E_1$ if and only if $f(u)f(v) \in E_2$.

Formally, the graph isomorphism problem can be described as follows:
Instance: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $|V_1| = |V_2|$ and $|E_1| = |E_2|$.
Question: Determine whether there exists a bijective edge-invariant vertex mapping (isomorphism) $f : V_1 \rightarrow V_2$.

The required mapping f is a permutation of vertices in V_1. To represent any of the $n!$ permutations as a ranked index we need $min\{k|2^k \geq n!\} = \lceil lg(n!) \rceil$ bits, which is about $n\ lg\ n$ bits. Formulating a QUBO with this theoretical lower bound remains a significant challenge [24].

2.3.2 Subgraph Isomorphism

The Subgraph Isomorphism problem (SGI) essentially involves determining whether the smaller graph, from a pair of given graphs G_1 and G_2 of different sizes, is a subgraph of the larger graph, such that it can be mapped onto the topology of the larger graph. The solution to be expected upon solving an instance of subgraph isomorphism for a pair of graphs is also a mapping, like for the GI problem. However, this mapping is from the vertices and edges of the smaller graph to those of the larger graph; and this mapping must include all the vertices and edges of the smaller graph while it may or may not include all the vertices and

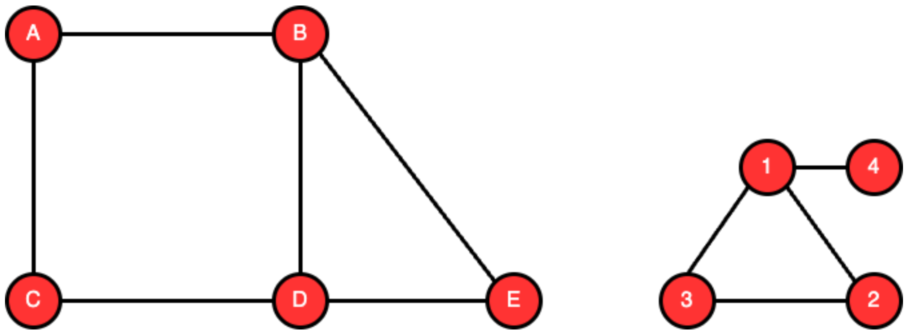

Figure 2.7: Subgraph Isomorphism Example, G_1 (left) and G_2 (right).

edges of the larger graph.

The SGI problem is an important problem to solve since it has many applications in chemistry, drug-discovery, biology and many other. However, most cases of subgraph isomorphism are considered to be NP-Hard [24] which makes it difficult to find an exact solution for large instances of the problem. Consequently, researchers have to rely on heuristics and approximation algorithms to find good solutions. Furthermore, the difficulty of finding good solutions with heuristics increase with increase in the size of the problem space which corresponds to the increase in the number of nodes in the graphs being compared. Quantum annealing has the potential to provide better solutions since it can explore large solution spaces parallelly and avoid getting stuck in local minima. Additionally, quantum annealing can take advantage of quantum tunneling to overcome energy barriers and find lower-energy solutions that may be missed by classical heuristics. However, the effectiveness of quantum annealing for solving subgraph isomorphism problems depends on the specific problem instance and the quality of the QUBO formulation used.

Briefly, the SGI problem involves finding out the mapping of the vertices and edges of a smaller graph, here in this thesis referred to as $G_2 = (V_2, E_2)$, to the vertices and edges of a larger graph, $G_1 = (V_1, E_1)$. If a mapping is found, then the smaller graph is said to be a subgraph isomorph of the larger graph.

Instance: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $n_1 = |V_1| \leq |V_2| = n_2$ and

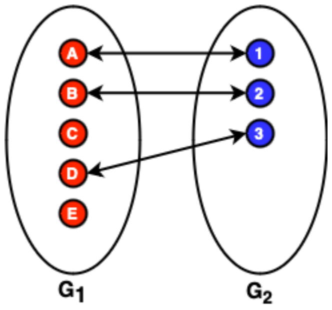

Figure 2.8: Injective Mapping between G_1 (left) and G_2 (right).

$|E_1| \leq |E_2|$.

Question: Find an edge-preserving injective function $f : V_1 \rightarrow V_2$.

Note that for this problem the function f is not necessarily edge-invariant: it has only to be "edge-preserving", that is, for every $uv \in E_1$ we have $f(u)f(v) \in E_2$ [24].

The non-induced subgraph isomorphism problem seeks to establish an injective mapping from the vertices of a smaller given graph, G_2, to the vertices of a designated target graph, G_1, which preserves adjacency. In contrast, the induced subgraph isomorphism variant demands that the mapping upholds non-adjacency, thereby precluding the existence of "extra edges" in the pattern replica. Subgraph isomorphism finds practical applications in a diversity of fields such as computer vision [33], biochemistry [6], and pattern recognition [29, 73].

More specifically, a non-induced subgraph isomorphism entails an injective mapping i from $V(G_2)$ to $V(G_1)$ that preserves adjacency. For all adjacent vertices v and w in $V(G_2)$, the vertices $i(v)$ and $i(w)$ must be adjacent in G_2. Furthermore, an induced subgraph isomorphism retains non-adjacency, meaning that if v and w are not adjacent in G_2, then $i(v)$ and $i(w)$ cannot be adjacent in G_1. Non-induced subgraph isomorphism is denoted as $i : G_2 \longmapsto G_1$, whereas induced subgraph isomorphism is denoted as $i : G_2 \hookrightarrow G_1$ [73].

Chapter 3

Software Code Clone Detection as an Optimization Problem

In this thesis, I conceptualize code clone detection as an optimization problem. At a very high-level, the objective is to map one program onto another while minimizing the energy of that mapping. The lower the energy, the greater the similarity between the two programs. The challenge is, then, in specifying mapping and establishing the penalties to be applied to the problem Hamiltonian for incorrect mappings.

There can potentially be many ways of formulating this problem. In essence, this involves solving the subgraph isomorphism problem and formulating it as a QUDO model. The main idea for solving subgraph isomorphism with a QUDO model is penalizing the node-to-node mappings from G_2 to G_1 that have non-equivalent edges, which will be explained next.

But this alone is not enough: the nodes of these graphs are of different types – conditionals, assignments, function calls, etc. If we ignore node types, we may end up with topologically perfect subgraph mappings that do not represent the syntactic structure of the programs. As such, a critical second step is to include penalties for mappings between nodes of different types.

In the following subsections, I delve into a detailed description of how I developed and implemented these ideas. I start by explaining the basic approach of expressing graph isomorphism as a QUBO model, followed by its adaptation into a QUDO model. Subsequently, I explain how I express AST similarity within the framework of a QUDO model.

3.1 Graph Isomorphism as QUBO

Zick et. al. [112] were the first to convert Lucas' [70] Graph Isomorphism Ising Hamiltonian into a QUBO (note: not QUDO) formulation and experimentally evaluate it on a D-Wave device. This QUBO formulation detects whether two given input graphs are isomorphic or not. The applied penalties are designed such that the ground state energy becomes zero if and only if the input graphs G_1 and G_2, are isomorphic.

In this QUBO formulation, there are $N \times N$ binary variables $x_{u,i} \in \{0, 1\}$, one for every possible mapping of a node u in G_2 to a node i in G_1; in a solution, the variable is 1 if u is mapped to i, and 0 otherwise.

Their QUBO formulation penalizes certain node-to-node and edge-to-edge mappings between two graphs, target graph G_1 and query graph G_2, by increasing the energy of the QUBO by a fixed value each time a mapping violates the rules of graph isomorphism. Two types of penalties are applied to the variables and interactions between variables. The first penalty, denoted as C_1, penalizes non-bijective node mappings. For example, as seen in Figure 2.5, if both node 1 and node 2 of G_2 are mapped to node A of G_1, this mapping would violate the bijectivity constraint, resulting in a fixed penalty being added to the QUBO, increasing the QUBO model's energy. Note that if at least one of the two variables resolves to 0, the energy of the coupling is zero. The second penalty, denoted as C_2, applies to edge inconsistencies. An edge inconsistency occurs when nodes u and v of G_2 are mapped to nodes i and j of G_1, respectively, and an edge uv exists in G_2, but there is no corresponding edge ij in G_1. For example, in Figure 2.5, this situation occurs if node 1 of G_2 is mapped to node A of G_1 and node 2 of G_2 is mapped to node E of G_1.

The penalties have a fixed value, and every time a node-to-node mapping violates a constraint and activates a penalty, the energy of the QUBO is increased by that fixed penalty value. Thus, the energy of the solution to the QUBO is directly proportional to the number of incorrect node-to-node mappings. Additional details on the QUBO formulation for graph isomorphism can be found in Lucas et. al. [70].

When the QUBO is solved on a D-Wave Quantum Annealer, a dictionary with G_2 nodes as keys and the corresponding G_1 nodes as values is returned, accompanied by the respective minimum ground state energy of the QUBO solution. This resulting global minimum energy can be used to quantify the similarity between the two graphs; a value of 0.0 indicates that the two graphs are identical, while a non-zero positive value indicates that the two graphs are not identical. The number of unmatched or mismatched nodes between the two graphs is directly proportional to the energy of the QUBO solved with those two graphs.

3.2 Graph Isomorphism as QUDO

At the time of Zick's paper [112], using binary variables was the only way to formulate a QUBO and one could only use either the classical simulated annealing solver or the smaller 2000 qubit QPU, separately. D-Wave's open source GitHub repository of examples [see `https://github.com/dwave-examples/circuit-equivalence`] includes an alternative implementation of Zick's graph isomorphism. This alternative implementation solves the Circuit Equivalence problem by developing a Discrete Quadratic Model (DQM) formulation of graph isomorphism. A DQM is instantiated and solved on D-Wave Leap's hybrid solvers.

In this formulation, instead of using $N \times N$ binary variables like in the QUBO formulation, N discrete variables are used, taking advantage of the discrete quadratic model, where N denotes the number of nodes in one of the graphs (must be the same as the other). Each discrete variable has N cases that represent the nodes on the other graph. Lucas' Ising formulation [70] for graph isomorphism, the objective function on which this discrete quadratic model is based, has two components - H_A and H_B. The first component, H_A, is used to enforce the constraint that each node in each of the two graphs is selected exactly once. The second component, H_B, uses interaction terms to penalize mappings that select an edge in the first graph that is not present in the second graph, or vice versa.

Below is the Python code from the circuit-equivalence repository that establishes the penalties for node-to-node mappings [see top code listing in Figure 3.1]. It starts by initializing the DQM object. The first *for* loop adds all the nodes of G_1 as discrete variables that has n (the number of nodes of G_2) as cases, the second *for* loop establishes the first component, H_A, as set of penalties.

The interaction coefficients associated with H_B are defined in the code using two double-loops [see bottom code listing in Figure 3.1]. Each double-loop includes an outer loop over the edges of one of the two graphs, along with an inner loop over all possible node combinations,

```python
# n is the number of nodes in the graphs
dqm = dimod.DiscreteQuadraticModel()
for node in G1.nodes:
  # Discrete variable for node i in graph G1,
  # with cases representing the nodes in G2
  dqm.add_variable(n, node)

# Set up the coefficients associated with the
# constraints such that each node in G2 is
# chosen once. Penalize by 2.0 otherwise. This
# represents the H_A component of the energy function.
for node in G1.nodes:
    dqm.set_linear(node, np.repeat(-1.0, n))
for itarget in range(n):
  for ivar,node1 in enumerate(G1_nodes):
    for node2 in G1_nodes[ivar+1:]:
      dqm.set_quadratic_case(node1, itarget, node2, itarget, 2.0)
```

```python
# The penalty coefficient B
# controls the weight of H_B relative to H_A
# in the energy function. (same here)
B = 2.0

# For all edges in G1, penalizes mappings
# to edges not in G2
for e1 in G1.edges:
  for e2_indices in itertools.combinations(range(n), 2):
    e2 = (G2_nodes[e2_indices[0]], G2_nodes[e2_indices[1]])
    if e2 in G2.edges:
      continue
    # In the DQM, the discrete variables
    # represent nodes in the first graph and are
    # named according to the node names. The
    # cases for each discrete variable represent
    # nodes in the second graph and are indices from 0..n-1
    dqm.set_quadratic_case(e1[0], e2_indices[0], e1[1], e2_indices[1], B)
    dqm.set_quadratic_case(e1[0], e2_indices[1], e1[1], e2_indices[0], B)

# For all edges in G2, penalizes mappings to edges
# not in G1
for e2 in G2.edges:
  e2_indices = (G2_nodes.index(e2[0]), G2_nodes.index(e2[1]))
  for e1 in itertools.combinations(G1.nodes, 2):
    if e1 in G1.edges:
      continue
    dqm.set_quadratic_case(e1[0], e2_indices[0], e1[1], e2_indices[1], B)
    dqm.set_quadratic_case(e1[0], e2_indices[1], e1[1], e2_indices[0], B)
```

Figure 3.1: Code listing for implementing Node-Mapping Penalty (top), and for implementing Edge-Mapping Penalty (bottom)

so that penalties can be added for all invalid combinations:

In this code above, *set_quadratic_case()* is a built-in function of the Discrete Quadratic Model package of D-Wave's Ocean SDK that sets the penalties associated with the interaction between the two cases of nodes. Recalling Figure 2.5 from earlier as context for understanding how the *set_quadratic_case()* function operates: An instance of this function that penalizes non-bijective mappings of nodes 1 and 2 of Graph G_2 to node A of Graph G_1 would look like *set_quadratic_case(node1, nodeA, node2, nodeA, p)*, where nodes 1 and 2 of G_2 are the discrete variables and node A of G_1 is the case, and p is the fixed penalty value. Simply put, this function penalizes mappings of nodes from Graph G_2 to Graph G_1 when the mapping violates the bijectivity rule of graph isomorphism which states that one node of one graph can only be mapped to one node of another graph.

The discrete quadratic model is then solved using the *LeapHybridDQMSampler()*. If the two graphs are isomorphic, then the ground state energy is zero.

3.3 Subgraph Isomorphism for Code Clone Detection

Graph isomorphism assumes that the graphs being compared have an identical number of nodes and edges. Clearly, that does not work for code clone detection as the graphs representing the different programs may vary in size, or one program could be a a snippet of a larger program, resulting in graphs with different number of nodes and edges. This necessitates the use of subgraph isomorphism.

My design of the subgraph isomorphism algorithm builds upon Zick's [112] implementation of the QUBO for Graph Isomorphism, but, more directly, on D-Wave's example of circuit equivalence using a DQM solver. The fundamental idea of my formulation is a QUDO model, when encoded into the quantum annealer with the help the DQM, that converges into the global minimum energy state when a smaller graph G_2 is mapped perfectly onto a larger graph G_1. Such a perfect mapping indicates that the smaller graph G_2 is a subgraph of of the larger graph G_1. My formulation also has the two types of penalties:

The Node-mapping Penalty - Given that the Target Graph G_1 has n_1 nodes and the Query Graph G_2 has n_2 nodes, with $n_1 > n_2$, the Node-mapping Penalty ensures that every node of G_2 must be mapped to only one node of G_1. All the possibilities where two different nodes of G_2 are mapped to the same node of G_1 are penalized. However, since not every node of G_1 has to correspond to a node of G_2 this penalty only penalizes the mappings from the set of nodes of G_2 to the set of nodes of G_1. This penalty is crucial because it ensures injectivity, instead of bijectivity which was the case for graph isomorphism.

The Edge-mapping Penalty - The Edge-mapping Penalty constraint ensures that the edges present between a pair of nodes in G_2 are only mapped to edges present between a pair of nodes in G_1. Any other mapping is penalized. The mapping from G_2 to G_1 is therefore edge-preserving.

3.4 Node Types

The initial design of my subgraph isomorphism algorithm focused primarily on the topological mapping of one graph on top another, without considering the types of the nodes in the AST graphs. This approach, while effective for certain scenarios, proved insufficient for the specific requirements of code clone detection.

To address this issue, I add an additional component that incorporates node type comparisons. In this enhanced model, when a node from the smaller graph G_2 is mapped onto a node in the larger graph G_1, their node types are compared. If the node types are the same, no penalty is applied to the mapping and the energy of the QUDO model is not increased. But if the node types don't match, a fixed penalty is applied to the mapping. This further ensures that the mappings generated are accurate and produce reliable results.

This type of conditional penalties can be applied for special cases as well. For instance, a smaller fixed value (e.g. 0.2, instead of the larger 2.0) can be applied to the DQM if the program driver detects that the two given code fragments have two different types of loops executing the same functionality. An example of this can be seen in *s6* and *s6_t3_v1* [see code fragments in Table A.3 in the Appendix] where the former has a *for* loop and the latter has a *while* loop. This can be used to detect Type 3 code clones that are functionally similar but have a different type of code component executing that functionality.

3.4.1 Complete Algorithm

Finally, the complete algorithm used to generate the results for this thesis can be seen in Algorithm 1. The algorithm starts by initializing the DQM object and setting its *offset* value to n_2. The *offset* property of the DQM object is a constant value that is added to the energy of the lowest-energy state of the model. This adjustment is essential as it effectively shifts

Algorithm 1: Subgraph Isomorphism Algorithm, with node-type constraint added

```
Input        : Target Graph $G_1$, Query Graph $G_2$
Precondition: $G_1$ is larger than $G_2$
Output       : DQM with updated Quadratic Penalties, and added Node-Type comparison
               Linear Penalty
1  Set DQM Offset as $n_2$
2  for each $node_2 \in G_2$ do
3      Add $node_2$ as a Discrete Variable to the Discrete Quadratic Model with $n_1$ as the Number
         of Cases $node_2$ can be mapped to
4  end
5  for each $node_2 \in G_2$ do
6      Set Linear Coefficient of $node_2$ as $-1.0$ in the QUBO
7  end
8  for $itarget \leftarrow 0$ to $n_1$ do
9      for $ivar \leftarrow 0$ to $n_2$ and each $node_1 \in G_2\_nodes$ do
10         for each $node_2 \in G_2\_nodes$ at index position from $ivar + 1$ to $n_2$ do
11             Penalize mapping of $node_1 \rightarrow G_1$ node at index $itarget$, $node_2 \rightarrow G_1$ node at index
                 $itarget$
12         end
13     end
14 end
15 for $edge_2 \in G_2$ do
16     for $e_1 \in \{$all pair-wise combinations of $G_1$ nodes$\}$ do
17         $node_1\_indices \leftarrow$ indices of a pair $e_1 \in G_1\_nodes$
18         if $e_1 \in G_1.edges$ then
19             continue
20         end
21         Penalize mapping of first node of $edge_2 \rightarrow$ first node of $node_1\_indices$, second node of
             $edge_2 \rightarrow$ second node of $node_1\_indices$
22         Penalize mapping of first node of $edge_2 \rightarrow$ second node of $node_1\_indices$, second node of
             $edge_2 \rightarrow$ first node of $node_1\_indices$
23     end
24 end
25 for each $node_2 \in G_2.nodes$ do
26     for each $node_1 \in G_1.nodes$ do
27         if $node_1\_type == node_2\_type$ then
28             continue
29         end
30         Penalize mapping $node_2 \rightarrow node_1$
31     end
32 end
```

the energy levels of all the states. The *offset* is usually set to a value that is larger than the minimum energy of the model, to ensure that the minimum energy state always has a non-negative energy value. This is because the quantum annealing optimization algorithm typically aims to minimize the energy of a model, so setting the *offset* to a value that is larger than the minimum energy ensures that the algorithm does not converge to an non-physical state, with negative energy.

In my algorithm, the *offset* value is set to n_2, which is the number of nodes in the Query Graph G_2, to ensure that the minimum energy state of the model has an energy value of at least n_2. This setting is particularly useful in scenarios where a code clone is a snippet of the original code fragment with one or more lines removed from it, meaning that the AST of the code clone would be a perfect subgraph of the AST of the original code fragment. The code clone's graph G_2 would map perfectly onto the nodes and edges of the original code fragment's graph G_1. The resulting global minimum energy of comparing these two graphs using the subgraph isomorphism algorithm should be 0.0.

In lines 2–4, the nodes of G_2 are added as discrete variables to the DQM with n_1 nodes as the cases. This implies that the nodes of G_2 act as variables of the QUDO that can be assigned any discrete value, called a case, which is the index of a list of nodes of G_1. Therefore, the QUDO contains n_2 discrete variables, each with n_1 cases.

Lines 5–14 of Algorithm 1 implement the constraint of mapping each node in G_2 to exactly one node in G_2. The lines 5–8 set up the linear coefficients with the constraint that each node in G_2 is chosen once. More specifically, this constraint is expressed in terms of penalties that are added to the model when a node in G_2 maps to more than one node in G_1.

The second component (lines 15–24) uses interaction terms to penalize settings that select an edge in the G_2 graph that is not in the G_1 graph. The interaction coefficients associated with the second component are defined in the code using a double-loop. The double-loop includes

Table 3.1: Original Code Fragments.

s1.py

```python
import cmath
num = 1+2j
num_sqrt = cmath.sqrt(num)
print('The square root of {0} is {1:0.3f}+{2:0.3f}j'.format(num,num_sqrt.real,num_sqrt.imag))
```

s2.py

```python
s = "quantum"
reverse = s[::-1]
print("Reverse of", s, "is", reverse)
```

s3.py

```python
import cmath
a = 1
b = 5
c = 6
d = (b**2) - (4*a*c)
sol1 = (-b-cmath.sqrt(d))/(2*a)
sol2 = (-b+cmath.sqrt(d))/(2*a)
print('The solution are {0} and {1}'.format(sol1, sol2))
```

s4.py

```python
lower = 100
upper = 2000
for num in range(lower, upper + 1):
    order = len(str(num))
    s = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        s += digit ** order
        temp //= 10
    if num == s:
        print(num)
```

s5.py

```python
def sequence(start, stop):
   builder = []
   i = start
   while (i < stop):
      if (i > start):
         builder.append(',')
      builder.append(i)
      i += 1
   return ''.join(builder)
```

s6.py

```python
num = 7
factorial = 1
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1,num + 1):
       factorial = factorial*i
   print("The factorial of",num,"is",factorial)
```

an outer loop over the edges of G_2, along with an inner loop over all possible node-node pair combinations of G_1 nodes, so that penalties can be added for all invalid combinations.

The final component (lines 25–32) introduces the penalties for mapping nodes of different types. Although not explicitly detailed here for the sake of simplicity, the algorithm also includes a special case handling for different loop types (for/while) in the code implementation.

3.5 Experiment Design

In order to empirically validate my proposed formulation, I created a dataset of six small Python programs and their corresponding clones, resulting in a total of 25 distinct programs. The clones were carefully created to represent Type 2 and Type 3 code clones, adhering to the specific transformation criteria associated with each type. I created this custom dataset because with this I was able to be in complete control of the size range of the programs being input and also the particular transformations necessary to demarcate boundaries between the clone types. This allowed us to have a reliable ground truth about which code fragments were what type of code clones of the original ones, so we could verify the results from the quantum annealer. Table A.1 lists the original code fragments and Tables A.2, A.3, and A.4 (in Appendix) show the corresponding Type 2 and Type 3 clones.

The experiment in this study compared each of these 25 code fragments with every other code fragment to create a symmetric result matrix for which I only fill the lower triangular part. There are a total of 325 unique comparisons, for which the ground truth is established and known.

I conducted two sets of experiments,

1. DQM for Simple Subgraph Isomorphism: This experiment did not include node type constraints.

2. DQM for Subgraph Isomorphism with Node Type Constraints: This variant incorporated additional constraints based on node types.

To ensure reliable results, each pairwise graph comparison was performed 10 times, for a total of 6,500 measurements (325 pairs without node types, 325 pairs with node types, each repeated 10 times). Note that a single pairwise graph comparison is one quantum machine query performed in the QPU that, by default, has a 100 reads/annealing cycles, which means every quantum machine query in the QPU executes the DQM model for that instance of the two input graphs a 100 times. At the end of the quantum machine query, the results of these 100 annealing cycles are put through a post-processing schemes, such as Majority Voting (MV) as discussed earlier in this thesis, to mitigate the effects of errors during the annealing processes There have been many advancements to post-processing methods in the D-Wave Advantage machines, some even using heuristics to mitigate the effects of errors and noise on the solutions. I repeated each pairwise comparison 10 times as a redundancy measure to reinforce the validity of the solutions I obtained, even when the D-Wave solver I used took care of mitigating errors with its post-processing schemes [106].

To reiterate, the D-Wave Advantage Quantum System used for this study has over 5,000 qubits, with over 35,000 couplers, based on the Pegasus Graph Topology of arranging qubits on the chip.

For this implementation, I used the *hybrid_discrete_quadratic_model_version1* solver of the Hybrid Solver Service (HSS). The HSS provided by D-Wave is a collection of both classical and quantum solvers that can efficiently read and solve large optimization problems [74]. Each hybrid solver contains an implementation of a classical heuristic that explores the discrete solution space defined by the input, together with a quantum module (QM) that formulates some number of quantum queries that are sent to an Advantage QPU. The D-Wave Advantage Quantum System was accessed via the D-Wave Leap Quantum Cloud Computing Service.

Chapter 4

Results and Analysis

4.1 Energy: Similarity Detection

The normalized average energy values for both configurations are shown in Figure 4.1. Each value in these tables is an average of 10 measurements, normalized by the number of nodes in the smaller graph G_1, so that we can compare them. The values range from 0.00 (green, indicating exact clones) to 2.02 (white, indicating no similarity). The interpretation of these values is as follows.

The fixed penalty of 2.00 is applied for every incorrect node mapping from G_2 to G_1, increasing the overall energy of the model by 2.00 points. This number was selected by trial and error. The minimum energies normalized by the number of nodes in the smaller AST of the two code fragments indicates how many incorrect mappings were made for every node in G_2, on average. For instance, in the comparison between *s3_t3_v1* and *s5_t3_v1* as seen in Figure 4.1 (bottom), the value of 2.02 means that all the nodes of the ASTs of the two graphs are incorrectly mapped, which is expected since both of these code fragments are very different. Conversely, a value of 0.00 means that all the nodes and edges in the smaller

	s1	s1_t2_v1	s1_t3_v1	s1_t3_v2	s2	s2_t2_v1	s2_t3_v1	s2_t3_v2	s3	s3_t2_v1	s3_t2_v2	s3_t3_v1	s3_t3_v2	s4	s4_t2_v1	s4_t3_v1	s4_t3_v2	s5	s5_t2_v1	s5_t3_v1	s5_t3_v2	s6	s6_t2_v1	s6_t3_v1	s6_t3_v2
s1	0.00																							
s1_t2_v1	0.00	0.00																						
s1_t3_v1	0.00	0.07	0.00																					
s1_t3_v2	0.05	0.02	0.00	0.00																				
s2	0.29	0.29	0.29	0.24	0.00																			
s2_t2_v1	0.29	0.29	0.29	0.24	0.00	0.00																		
s2_t3_v1	0.14	0.14	0.14	0.29	0.03	0.01	0.00																	
s2_t3_v2	0.28	0.32	0.26	0.26	0.00	0.00	0.04	0.00																
s3	0.25	0.23	0.30	0.15	0.21	0.19	0.14	0.21	0.00															
s3_t2_v1	0.26	0.25	0.29	0.14	0.20	0.21	0.14	0.26	0.00	0.00														
s3_t2_v2	0.26	0.25	0.29	0.12	0.22	0.19	0.14	0.21	0.00	0.01	0.00													
s3_t3_v1	0.44	0.41	0.45	0.19	0.21	0.22	0.06	0.20	0.26	0.28	0.27	0.01												
s3_t3_v2	0.44	0.37	0.44	0.14	0.17	0.18	0.14	0.27	0.42	0.42	0.38	0.48	0.16											
s4	0.37	0.39	0.36	0.21	0.21	0.21	0.14	0.26	0.56	0.55	0.54	0.62	0.65	0.01										
s4_t2_v1	0.39	0.41	0.37	0.18	0.25	0.20	0.14	0.26	0.55	0.55	0.55	0.62	0.65	0.01	0.02									
s4_t3_v1	0.40	0.40	0.36	0.22	0.22	0.21	0.14	0.26	0.56	0.54	0.57	0.64	0.56	0.05	0.03	0.00								
s4_t3_v2	0.39	0.41	0.37	0.21	0.21	0.24	0.14	0.27	0.54	0.53	0.53	0.56	0.55	0.03	0.04	0.04	0.03							
s5	0.29	0.33	0.29	0.22	0.29	0.29	0.14	0.30	0.41	0.40	0.40	0.52	0.44	0.30	0.30	0.30	0.31	0.00						
s5_t2_v1	0.30	0.33	0.29	0.20	0.29	0.29	0.14	0.29	0.41	0.39	0.42	0.51	0.44	0.29	0.29	0.30	0.32	0.00	0.00					
s5_t3_v1	0.29	0.29	0.29	0.13	0.25	0.21	0.14	0.35	0.35	0.33	0.32	0.46	0.37	0.31	0.31	0.35	0.39	0.11	0.11	0.00				
s5_t3_v2	0.34	0.33	0.31	0.18	0.29	0.29	0.14	0.30	0.36	0.36	0.36	0.48	0.43	0.28	0.28	0.28	0.29	0.00	0.00	0.22	0.00			
s6	0.44	0.41	0.43	0.24	0.20	0.22	0.29	0.26	0.35	0.36	0.38	0.38	0.34	0.24	0.24	0.25	0.34	0.50	0.48	0.44	0.45	0.09		
s6_t2_v1	0.43	0.41	0.43	0.24	0.23	0.20	0.29	0.26	0.34	0.34	0.34	0.38	0.36	0.23	0.23	0.24	0.35	0.50	0.50	0.43	0.45	0.08	0.04	
s6_t3_v1	0.58	0.58	0.64	0.24	0.19	0.19	0.29	0.26	0.37	0.37	0.38	0.33	0.33	0.25	0.25	0.25	0.25	0.32	0.32	0.48	0.31	0.36	0.37	0.00
s6_t3_v2	0.57	0.49	0.58	0.24	0.19	0.19	0.16	0.17	0.31	0.30	0.30	0.28	0.32	0.21	0.21	0.21	0.21	0.35	0.36	0.35	0.37	0.22	0.23	0.21	0.00

	s1	s1_t2_v1	s1_t3_v1	s1_t3_v2	s2	s2_t2_v1	s2_t3_v1	s2_t3_v2	s3	s3_t2_v1	s3_t2_v2	s3_t3_v1	s3_t3_v2	s4	s4_t2_v1	s4_t3_v1	s4_t3_v2	s5	s5_t2_v1	s5_t3_v1	s5_t3_v2	s6	s6_t2_v1	s6_t3_v1	s6_t3_v2
s1	0.00																							
s1_t2_v1	0.07	0.00																						
s1_t3_v1	0.21	0.32	0.00																					
s1_t3_v2	0.00	0.12	0.35	0.00																				
s2	1.29	1.00	1.19	1.42	0.00																			
s2_t2_v1	1.29	1.00	1.19	1.41	0.00	0.00																		
s2_t3_v1	1.29	0.86	1.29	1.29	0.00	0.00	0.00																	
s2_t3_v2	1.17	0.91	1.09	1.24	0.10	0.10	0.14	0.00																
s3	0.50	0.47	0.50	0.47	0.95	0.95	0.86	1.04	0.00															
s3_t2_v1	0.50	0.47	0.50	0.47	0.95	0.95	0.86	1.04	0.00	0.00														
s3_t2_v2	0.50	0.47	0.50	0.47	0.95	0.95	0.86	1.04	0.00	0.00	0.00													
s3_t3_v1	1.50	1.47	1.61	1.12	0.95	0.95	0.86	1.09	0.63	0.63	0.63	0.00												
s3_t3_v2	1.04	1.03	1.14	0.82	1.05	1.05	1.00	1.04	0.61	0.61	0.61	0.64	0.00											
s4	1.36	1.47	1.43	1.00	1.14	1.14	1.14	1.22	1.28	1.27	1.28	1.38	1.66	0.00										
s4_t2_v1	1.36	1.47	1.43	1.00	1.14	1.14	1.14	1.22	1.27	1.28	1.28	1.38	1.65	0.00	0.00									
s4_t3_v1	1.36	1.47	1.43	1.00	1.14	1.14	1.14	1.22	1.41	1.41	1.41	1.57	1.54	0.03	0.03	0.00								
s4_t3_v2	1.46	1.57	1.54	1.00	1.19	1.19	1.14	1.26	1.14	1.13	1.12	1.22	1.30	0.00	0.00	0.00	0.00							
s5	1.07	1.20	1.07	1.06	1.48	1.48	1.50	1.43	1.57	1.57	1.57	2.01	1.76	1.31	1.31	1.36	1.54	0.00						
s5_t2_v1	1.07	1.20	1.07	1.06	1.48	1.48	1.50	1.43	1.57	1.57	1.57	2.01	1.76	1.31	1.31	1.36	1.54	0.00	0.00					
s5_t3_v1	1.21	1.43	1.07	1.41	1.62	1.62	1.71	1.52	1.45	1.45	1.45	2.02	1.72	1.33	1.33	1.39	1.58	0.26	0.26	0.00				
s5_t3_v2	1.29	1.43	1.32	1.06	1.52	1.54	1.50	1.48	1.59	1.59	1.59	1.95	1.78	1.22	1.22	1.27	1.46	0.00	0.00	0.62	0.00			
s6	1.43	1.47	1.36	1.00	0.86	0.86	1.14	0.87	1.25	1.26	1.25	1.29	1.28	0.80	0.80	0.80	1.08	1.48	1.48	1.44	1.41	0.00		
s6_t2_v1	1.43	1.47	1.36	1.00	0.86	0.86	1.14	0.87	1.26	1.25	1.26	1.29	1.29	0.80	0.80	0.80	1.08	1.48	1.48	1.44	1.41	0.00	0.00	
s6_t3_v1	1.57	1.67	1.61	1.18	0.76	0.76	1.14	1.00	1.03	1.03	1.03	1.03	1.22	0.75	0.75	0.75	0.78	1.19	1.19	1.80	1.25	0.88	0.88	0.00
s6_t3_v2	1.43	1.41	1.50	1.00	0.76	0.76	1.14	0.78	0.90	0.90	0.90	0.93	0.97	0.59	0.59	0.59	0.69	1.43	1.43	1.59	1.52	0.38	0.38	0.57	0.00

Figure 4.1: Normalized Average Minimum Energy, without node type constraints (top) and with node type constraints (bottom).

AST were mapped correctly to the nodes and edges in the larger AST.

The first observation to make is that the normalized energy values in the top table of Figure 4.1 are much smaller than the values in bottom table. This means that the first DQM (pure subgraph isomorphism, without node type constraints) is able to find mappings more often than the second one (with node type constraints). This follows my expectations: the first DQM has fewer constraints. However, this also means that there is a much larger number of false positive clones in the top table. For example, the comparison between *s6_t3_v2* and *s2_t3_v1* [see Tables A.3 and A.4 in the Appendix for code fragments] has a normalized energy of 0.16, which is a relatively small value. Topologically, the smaller of these two programs can be nicely mapped to the larger one, but the two programs are considerably different, and are not clones: *s6_t3_v2* includes a for loop and a print statement that have no equivalent in *s6_t3_v2*.

A second observation about the top table of Figure 4.1 is that there are a few false negatives, too (i.e. true clones that are not classified as such). Specifically, I expected the energy values in the diagonal to always be 0.00, but in some cases they are greater than 0.00, with one case being as high as 0.16. This may be because the solver converged to a local minimum instead of the global minimum, and moreover, we are calculating the mean of several samples which are likely distorted by some outliers resulting in slightly higher values.

The numbers in the bottom table of Figure 4.1 reflect the ground truth much more accurately than those in the top table. For starters, the diagonal is consistently 0.00. Then, the clusters of clones of each sample program can be seen as the blocks of darker green over the diagonal, and the energy values tend to be lower for Type 2 clones (more similar) than for Type 3 (less similar). There is a 3 times higher contrast in the resultant energy levels when using the formulation with the node-type constraints. This is evident upon observation of the energy levels and is most distinctly noticed when comparing the highest minimum energy levels obtained for both the formulations – 0.64 in the table for the formulation without

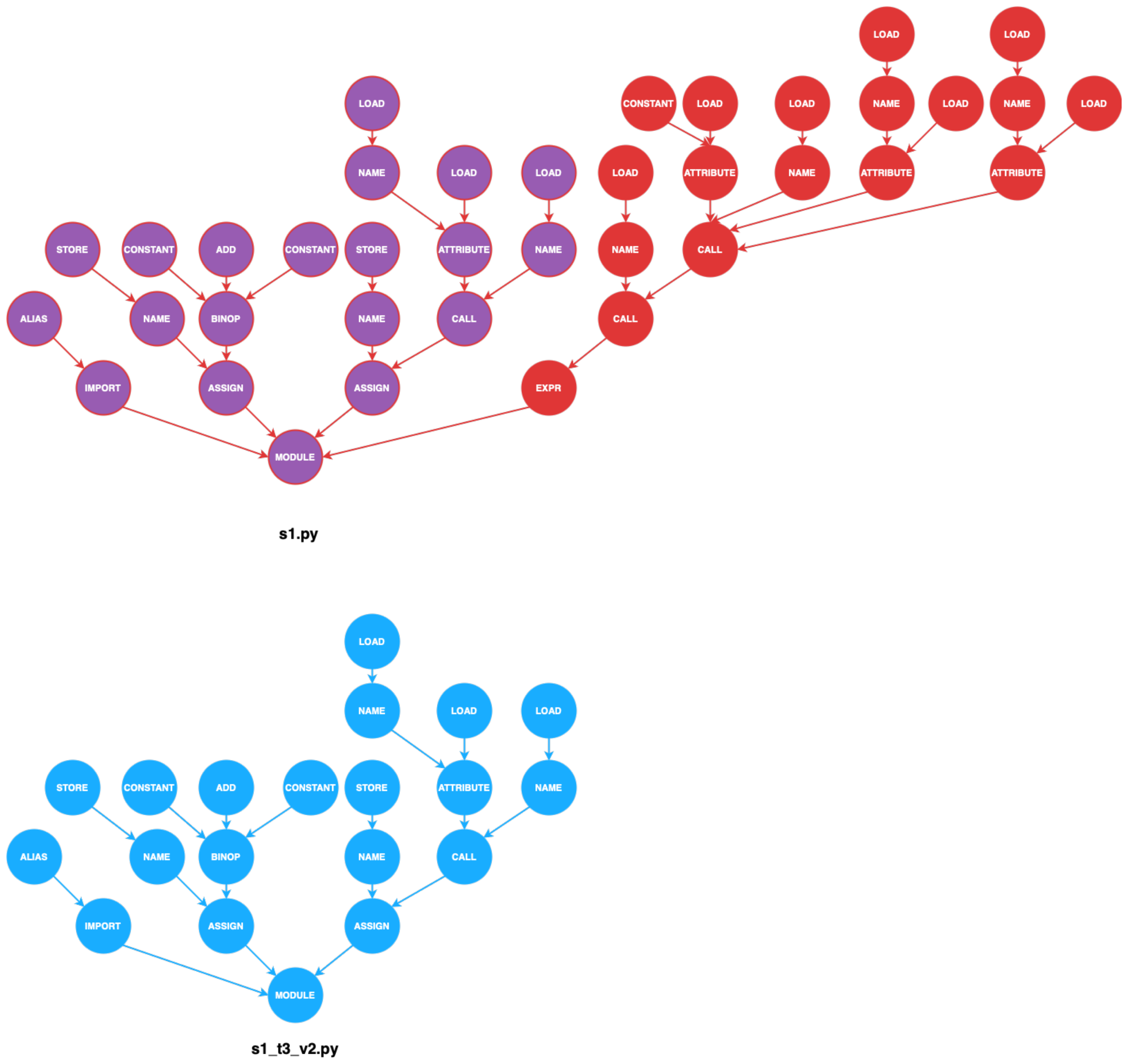

Figure 4.2: ASTs of s1 (entire tree) vs. s_1_t3_v2 (purple nodes only).

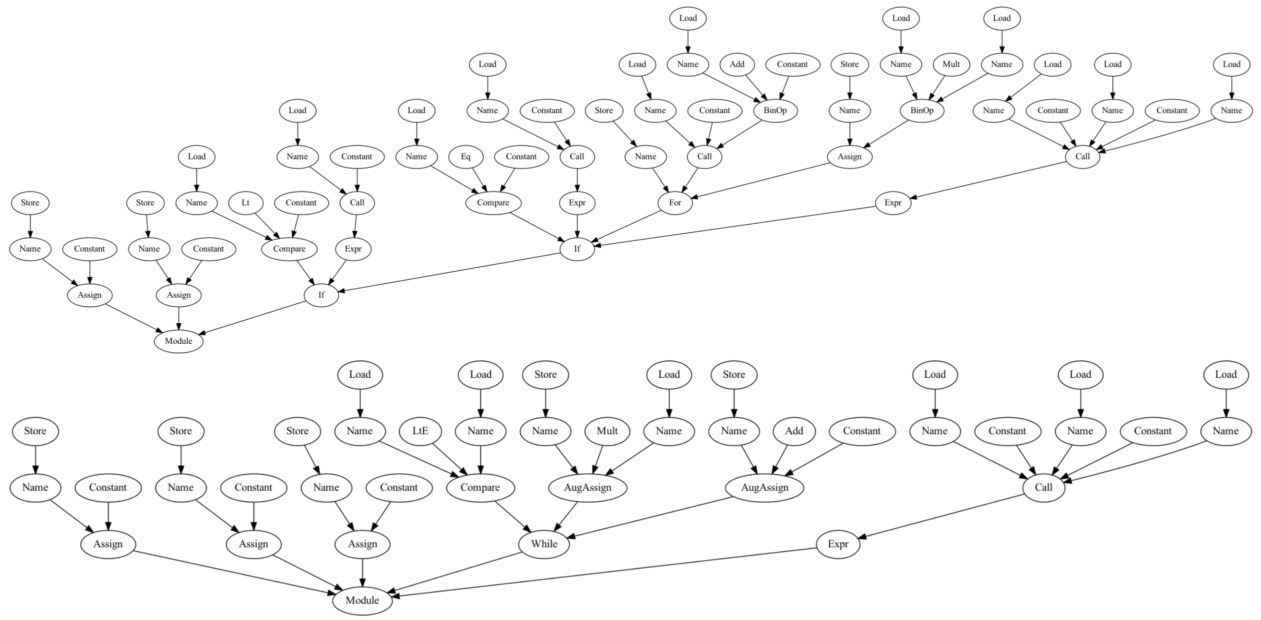

Figure 4.3: ASTs for s6 (top) vs. s6_t3_v1 (bottom).

node-type constraints, Figure 4.1 (top), and 2.02 in the table for the formulation with node-type constraints, Figure 4.1 (bottom). This indicates that the formulation with node-type constraints added better at distinguishing different types of code clones.

Moreover, pairs of programs that are very different end up having high energy values, typically above 1.00. With node type constraints, it is possible to establish a threshold, around 0.90, above which the similarities are very weak; that threshold is unclear in the simpler DQM.

For instance, code fragments *s2_t3_v1* and *s3* are distinctly different, and yet the structure of their ASTs is similar, making *s2_t3_v1* a very weak Type 3 code clone of *s3*. In the simple DQM [see top table of Figure 4.1], their energy is 0.14, a relatively small value, which would suggest strong similarities; in the DQM with node type constraints [see bottom table of Figure 4.1], their energy is 1.05, signaling weak similarities. In reality, they are topologically similar, but the node types are different. This demonstrates the accuracy of the second DQM.

Given that the second DQM is much more accurate than the first, I focus on a few pairs of code fragments and their corresponding energy values given by the second model:

- In the bottom table of Figure 4.1, it can be seen that comparing *s1* with *s1_t3_v2* yields a value of 0.00, even though the two programs are not exactly the same. This is because the nodes of the AST of *s1_t3_v2*, including their types, map perfectly onto those of the AST of *s1* [see Figure 4.2]. Indeed, *s1_t3_v2* is a syntactic subgraph of *s1*. In general, positions other than the diagonal containing 0.00 values indicate the normalized minimum energy from comparing code fragments where the AST of the query code fragment is a subgraph of the AST of the target subgraph, resulting in a perfect match. This is consistent with the actual pairs of code fragments, not just for *s1* vs. *s1_t3_v2*.

- Another interesting pair is *s3_t3_v1* and *s5_t3_v1*, which has the highest normalized energy value of 2.02. These code fragments are significantly different, as shown in Table A.3 (in Appendix). *s3_t3_v1* lacks a *function* call, a *for* loop, an *if* condition, and a *return* statement present in *s5_t3_v1*.

- As can be seen in Tables A.1 and A.3, *s6* has *if* conditions and a *for* loop while *s6_t3_v1* has a *while* loop. Typically, such code pairs can be classified as Type 3 code clones. Their ASTs can be see in the Figure 4.3. Comparing them yields an energy value of 0.88, which is below the threshold, and therefore can be classified as Type 3 clones.

4.2 Time: Performance and Scalability

The main advantage of quantum annealing is the use of superposition and entanglement of qubits that allows representing and searching very large state spaces. Whats more, one may recall that quantum annealing enables converging the problem Hamiltonian to the global minima faster by evading being stuck in the local minima. Even though the code fragments in my dataset are small, I wanted to experimentally verify whether there was any effect of the size of the graphs on the time the solver takes to converge to a solution. The computation times of the hybrid solver are shown in Figure 4.4, in milliseconds. The computation times observed in the results in Figure 4.4 indicate they are bound and relatively independent of the size and complexity of the input graphs under comparison.

Based on the values presented in Figure 4.4 and Figure 4.5, we can make the following observations:

- Mean: The mean timing values for comparisons range from 133 ms to about 155 ms. These values suggest that the algorithm's performance is relatively consistent across different code fragments.
- Standard Deviation: The variation in timing is relatively low for most code fragments, indicating consistent performance. However, there are some with higher standard deviations, such as *s3_t2_v1*, which might suggest that the comparison time varies more significantly for this code fragment, potentially due to its complexity or size [see the code fragment in Table A.2 in the Appendix].
- The time values vary between 109 ms and 190 ms. This corresponds to a ≈75% increase between the low end and the high end of the range.
- Even for very large differences in the size of the code fragments being compared, we see that the histogramin Figure 4.5 does not vary a lot and there is a well-defined peak

	# Nodes	s1	s1_t2_v1	s1_t3_v1	s1_t3_v2	s2	s2_t2_v1	s2_t3_v1	s2_t3_v2	s3	s3_t2_v1	s3_t2_v2	s3_t3_v1	s3_t3_v2	s4	s4_t2_v1	s4_t3_v1	s4_t3_v2	s5	s5_t2_v1	s5_t3_v1	s5_t3_v2	s6	s6_t2_v1	s6_t3_v1	s6_t3_v2
		28	30	28	17	21	21	14	23	65	65	65	67	61	61	61	68	52	42	42	36	37	49	49	32	29
s1	28	174																								
s1_t2_v1	30	150	179																							
s1_t3_v1	28	150	136	179																						
s1_t3_v2	17	174	171	172	165																					
s2	21	127	123	149	169	147																				
s2_t2_v1	21	137	135	145	167	164	157																			
s2_t3_v1	14	141	155	147	167	176	170	153																		
s2_t3_v2	23	131	141	139	179	169	154	162	163																	
s3	65	133	131	131	142	139	130	142	131	190																
s3_t2_v1	65	127	133	133	142	129	137	140	129	190	185															
s3_t2_v2	65	140	138	138	143	130	136	121	124	190	190	190														
s3_t3_v1	67	127	133	133	140	127	130	152	134	186	174	185	190													
s3_t3_v2	61	141	129	133	140	139	141	140	138	176	162	167	165	165												
s4	61	136	136	141	140	136	141	143	131	148	172	158	165	160	165											
s4_t2_v1	61	143	141	143	138	137	142	148	131	150	160	165	172	152	164	176										
s4_t3_v1	68	134	134	133	142	139	134	135	122	181	185	186	183	171	171	165	176									
s4_t3_v2	52	134	136	145	137	127	117	153	125	134	141	135	147	129	156	163	159	156								
s5	42	136	126	143	147	154	139	155	146	147	138	135	138	147	142	140	138	145	156							
s5_t2_v1	42	143	133	136	154	149	139	147	141	135	109	123	145	140	126	138	131	133	154	161						
s5_t3_v1	36	143	150	152	142	134	149	153	141	138	136	136	124	138	131	119	128	136	140	140	145					
s5_t3_v2	37	141	136	127	147	141	123	142	138	121	117	126	150	136	129	128	133	135	157	147	133	169				
s6	49	134	129	133	140	127	129	138	131	135	132	135	130	130	134	144	134	130	130	143	135	119	135			
s6_t2_v1	49	129	136	134	130	137	136	138	134	147	135	132	132	142	149	137	125	139	135	140	136	133	147	156		
s6_t3_v1	32	136	129	138	147	135	147	148	146	133	133	138	140	138	133	129	131	134	140	141	150	146	121	133	150	
s6_t3_v2	29	136	140	134	140	152	139	141	153	131	131	138	129	136	136	136	131	134	141	133	155	153	129	138	150	141

	# Nodes	s1	s1_t2_v1	s1_t3_v1	s1_t3_v2	s2	s2_t2_v1	s2_t3_v1	s2_t3_v2	s3	s3_t2_v1	s3_t2_v2	s3_t3_v1	s3_t3_v2	s4	s4_t2_v1	s4_t3_v1	s4_t3_v2	s5	s5_t2_v1	s5_t3_v1	s5_t3_v2	s6	s6_t2_v1	s6_t3_v1	s6_t3_v2
		28	30	28	17	21	21	14	23	65	65	65	67	61	61	61	68	52	42	42	36	37	49	49	32	29
s1	28	157																								
s1_t2_v1	30	172	171																							
s1_t3_v1	28	165	170	181																						
s1_t3_v2	17	181	176	184	176																					
s2	21	154	149	140	177	168																				
s2_t2_v1	21	151	147	152	169	161	152																			
s2_t3_v1	14	143	159	159	165	176	179	186																		
s2_t3_v2	23	160	156	163	164	158	159	179	160																	
s3	65	152	152	145	169	156	151	157	144	181																
s3_t2_v1	65	134	134	143	167	136	124	140	122	174	190															
s3_t2_v2	65	145	145	140	159	142	148	147	124	190	178	181														
s3_t3_v1	67	136	136	134	152	127	137	172	150	186	188	188	192													
s3_t3_v2	61	143	146	141	164	151	136	143	144	179	174	171	174	162												
s4	61	112	124	124	154	137	132	133	127	169	174	172	174	165	164											
s4_t2_v1	61	136	124	127	164	144	147	171	131	162	171	177	181	179	164	171										
s4_t3_v1	68	141	121	127	155	139	129	142	141	183	181	190	192	176	179	177	166									
s4_t3_v2	52	134	146	152	148	132	136	157	132	141	144	146	164	149	164	166	161	159								
s5	42	140	133	150	164	129	136	142	139	128	143	154	138	143	135	152	149	137	156							
s5_t2_v1	42	138	124	119	159	137	127	131	151	142	140	154	149	149	137	152	152	150	161	162						
s5_t3_v1	36	131	136	160	183	146	164	159	143	150	160	155	148	152	143	150	147	160	152	136	152					
s5_t3_v2	37	133	136	158	172	147	137	157	137	140	142	150	143	143	142	133	143	143	143	142	145	169				
s6	49	136	148	148	172	147	159	148	150	147	151	168	170	165	156	153	156	161	159	154	147	136	153			
s6_t2_v1	49	148	138	143	174	148	153	159	163	154	161	158	168	158	161	147	168	158	157	149	157	150	149	161		
s6_t3_v1	32	160	152	139	152	149	149	155	141	126	131	131	136	138	134	153	134	145	129	126	138	145	138	157	143	
s6_t3_v2	29	158	152	143	166	134	140	160	150	146	150	136	148	134	131	141	153	159	157	141	143	148	148	153	134	148

Figure 4.4: Average annealing times, in milliseconds, for the DQM with node-type comparisons (top), Average Annealing Times Matrix, in milliseconds, without node-type comparisons (bottom).

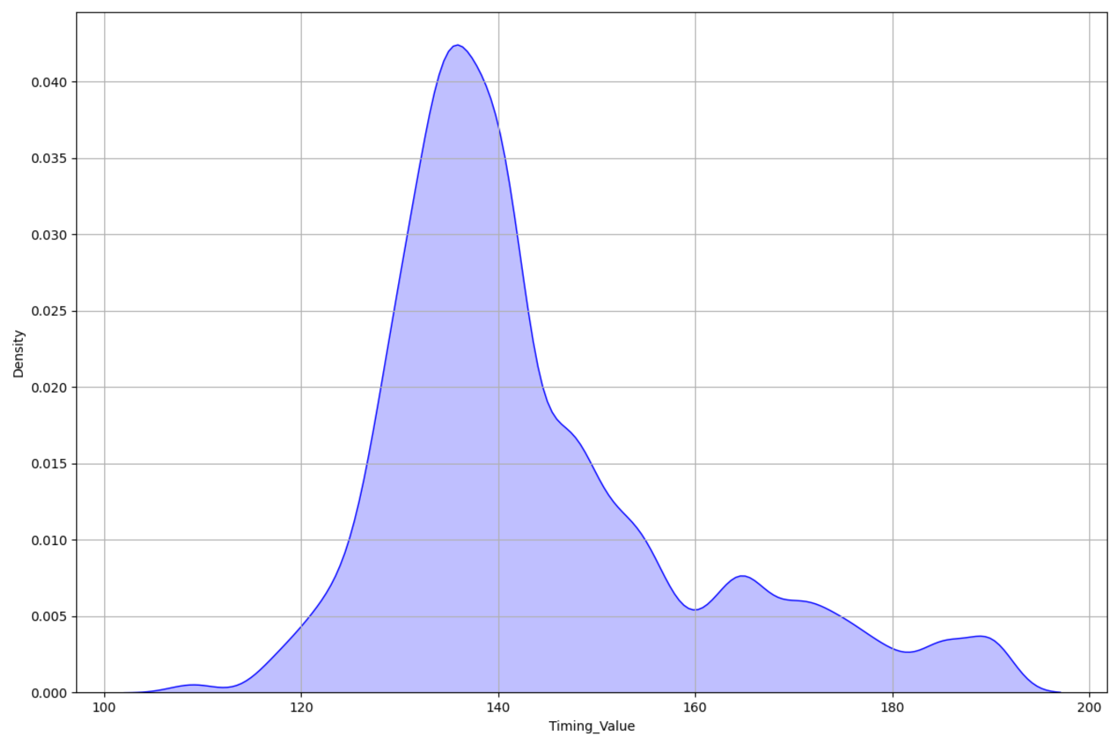

Figure 4.5: Histogram of average annealing times, in milliseconds, for the DQM with node-type comparisons.

at around 135 ms.

- The correlation coefficient between the timing values and the number of nodes of the first code fragment in each pair (# Nodes) is approximately 0.055. The correlation coefficient between the timing values and the number of nodes of the second code fragment in each pair (Comparison_Nodes) is approximately 0.150. These correlation coefficients are relatively low, suggesting that there is a weak positive relationship between the size of the code fragments (in terms of the number of nodes) and the time taken to compare them. In other words, while there is some indication that larger code fragments may take longer to compare, the number of nodes is not a strong predictor of the comparison time.

- The number of nodes in the code fragments varies between 14 and 68. This means that the state space of possible mappings in my experiments varies between $14 \times 14 = 196$

(*s2_t3_v1* compared to itself) and $68 \times 68 = 4,624$ (*s4_t3_v1* compared to itself). This corresponds to a ≈2,259% increase between the simplest and most complex problems in my dataset.

- Given the two growth rates mentioned above, it is clear that the time increase is much smaller than the complexity increase. In other words, if the two variables are correlated, the time to solve a given problem is not proportional to the complexity of the problem. This is likely due to a technical limitation of the quantum annealer, and is not a reflection of the computational or algorithmic complexity of the formulation.
- There is a noticeable block of high time values towards the middle of the table, corresponding to comparisons of code fragments with the highest number of nodes, between 61 and 68 nodes. However, there is also another block of high time values towards the top-left of the table where the code fragments are smaller. There doesn't seem to exist a correlation between the size of the problem and the time to solve it.
- In general, the highest times are found along, and close to, the diagonal, which is also where the most similar fragments are found (with lowest energies). This suggests some correlation between the time to solve the problem and the similarity of the fragments under comparison, where fragments that are more similar take longer to solve.

The D-Wave quantum annealer is sufficiently large for us to be able to run experiments with small, but real programs with a few lines of code. Real-world software problems would require computers with a much larger amount of available qubits, and my experiments will need to be replicated with larger examples before I can draw definite conclusions. Nevertheless, these empirical results are very encouraging: by treating clone detection as an optimization problem to be solved by quantum annealers, we may be able to compare and search large code bases in great detail without having to resort to heuristics. Moreover, my formulation is also amenable to alternative optimization architectures, such as Simulated Bifurcation

Machines [45, 60, 102] and other classical solvers. Using my formulation of the code clone problem, the energy values accurately reflect the level of code similarity, as shown in this section, and using quantum computing I am able to naturally address the scalability issue, as also shown here.

Chapter 5

Conclusion and Future Work

In this research, I present a novel approach to detecting software code clones using quantum annealers. It is the first time the software code clone detection problem has been formulated as an optimization of the subgraph isomorphism problem in the framework of a QUDO model, or any other variant of the QUBO model. It is also the first time the code clone detection problem has been successfully solved using quantum annealing. This method involves formulating the code clone detection problem as a Quadratic Unconstrained Discrete Optimization (QUDO) problem, and solving it on a D-Wave Quantum Computer. Building upon existing work in expressing graph isomorphism as QUBO and QUDO problems, my approach introduces and integrates never-done-before novel constraints that seem critical for code clone detection using subgraph isomorphism, as well as for other graph-based software engineering problems. Specifically, in software code clone detection we are not looking for graph isomorphism, but only subgraph isomorphism, which carry the core of the programs' syntax and semantics, and need to be expressed as optimization constraints. This is also an innovation in how QUDO or QUDO formulations for subgraph isomorphism can be tailored to specific problems by adding bespoke constraints to guide the annealing process into delivering desired results.

The Discrete Quadratic Model (DQM) developed for this implementation is discussed in detail, along with the experimental setup pipeline employed. The results of my experiments affirm the significance and credibility of node type constraints for the accuracy of clone detection. The model demonstrates proficiency in accurately classifying both Type 2 and Type 3 code clones based on the resultant energy of the annealing process. The time required to solve these optimizations seems to be bounded and largely independent of the complexity of the graphs under comparison.

While this study is exploratory in nature, its results are promising. Larger programs will need to be compared in order to draw stronger conclusions about accuracy and scalability. In the future, I plan on comparing the performance of my approach with pre-existing classical Python code clone detection tools to see where it stands relative to classical approaches. Nevertheless, the empirical results of this study are encouraging to explore a broader range of problems in software engineering that can be expressed as binary or discrete quadratic optimization problems. Quantum computers are already a reality, and rapid progress in their development is expected in the next decade. The 5,000 qubit D-Wave computer used in these experiments will soon be outdated by a larger model with more qubits. As quantum computers become more powerful, it is important to start getting acquainted with their different ways of solving computational problems. By expressing a well-known software engineering problem, clone detection, as a quantum annealing process, I hope to help build the bridges between what we are used to and what comes next.

5.1 Threats to Validity

- **Dataset and Annealer configuration:** The dataset used for this research is rather small and may even be limited in the variety of programs and the code clones it includes. We used the hybrid DQM Solver for this research. Since it abstracts away the low-level parameter controls at the qubit-level and tunes them automatically for depending on the DQM model it receives, it may be difficult to change the parameters like number of annealing cycles per quantum query or the annealing time to see how those parameters could affect the solutions. Biases in dataset selection or selecting inappropriate annealer parameters could skew the results.

- **Interpretation of results data of Similarity in Code Clones:** The normalized average resultant minimum energy threshold for identifying different clone types from on another is set heuristically. This same threshold value, for instance 0.9 for Type 3 code clones, may not be valid in other cases like for different datasets.

- **Consistency across trials:** Quantum computers are heavily affected by noise and errors. To mitigate the effects of noise and error, the experiments were conducted 10 times. This number was again selected heuristically, and was limited by the by the monthly quota for the number of quantum queries that could be made to the D-Wave quantum annealer through the Leap Cloud Quantum Computing Service.

Bibliography

[1] A. Adedoyin, J. Ambrosiano, P. Anisimov, W. Casper, G. Chennupati, C. Coffrin, H. Djidjev, D. Gunter, S. Karra, N. Lemons, et al. Quantum algorithm implementations for beginners. *arXiv preprint arXiv:1804.03719*, 2018.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-Wesley Reading, 2007.

[3] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. A systematic review on code clone detection. *IEEE Access*, 7:86121–86144, 2019.

[4] M. H. Amin. Consistency of the adiabatic theorem. *Physical Review Letters*, 102(22):220401, 2009.

[5] M. Anthony, E. Boros, Y. Crama, and A. Gruber. Quadratic reformulations of nonlinear binary optimization problems. *Mathematical Programming*, 162:115–144, 2017.

[6] A. Aparo, V. Bonnici, G. Micale, A. Ferro, D. Shasha, A. Pulvirenti, and R. Giugno. Fast subgraph matching strategies based on pattern-only heuristics. *Interdisciplinary Sciences: Computational Life Sciences*, 11:21–32, 2019.

[7] B. Apolloni, C. Carvalho, and D. De Falco. Quantum stochastic optimization. *Stochastic Processes and their Applications*, 33(2):233–244, 1989.

[8] R. Au-Yeung, N. Chancellor, and P. Halffmann. Np-hard but no longer hard to solve? using quantum computing to tackle optimization problems. *arXiv preprint arXiv:2212.10990*, 2022.

[9] L. Babai and E. M. Luks. Canonical labeling of graphs. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 171–183, 1983.

[10] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457, 1995.

[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[12] C. Bazgan, H. Hugot, and D. Vanderpooten. Solving efficiently the 0–1 multi-objective knapsack problem. *Computers & Operations Research*, 36(1):260–279, 2009.

[13] S. Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes [master's thesis]. *Institut fur Softwaretechnologie, Universitat Stuttgart, Stuttgart, Germany*, 2002.

[14] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[15] P. Benioff. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics*, 22(5):563–591, May 1980.

[16] E. Bernstein and U. Vazirani. Quantum complexity theory. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 11–20, 1993.

[17] Z. Bian, F. Chudak, R. Israel, B. Lackey, W. G. Macready, and A. Roy. Discrete optimization using quantum annealing on sparse Ising models. *Frontiers in Physics*, 2, 2014.

[18] Z. Bian, F. Chudak, W. Macready, A. Roy, R. Sebastiani, and S. Varotti. Solving sat (and maxsat) with a quantum annealer: Foundations, encodings, and preliminary results. *Information and Computation*, 275:104609, 2020.

[19] Z. Bian, F. Chudak, W. G. Macready, and G. Rose. The Ising model: teaching an old problem new tricks. *D-Wave Systems*, 2:1–32, 2010.

[20] S. Boixo, T. F. Rønnow, S. V. Isakov, Z. Wang, D. Wecker, D. A. Lidar, J. M. Martinis, and M. Troyer. Quantum annealing with more than one hundred qubits. *arXiv preprint arXiv:1304.4595*, 2013.

[21] S. Bravyi, D. Gosset, and R. Movassagh. Classical algorithms for quantum mean values. *Nature Physics*, 17(3):337–341, 2021.

[22] M. J. Bremner, A. Montanaro, and D. J. Shepherd. Achieving quantum supremacy with sparse and noisy commuting quantum computations. *Quantum*, 1:8, 2017.

[23] J. Brooke, D. Bitko, Rosenbaum, and G. Aeppli. Quantum annealing of a disordered magnet. *Science*, 284(5415):779–781, 1999.

[24] C. S. Calude, M. J. Dinneen, and R. Hua. QUBO formulations for the graph isomorphism problem and related problems. *Theoretical Computer Science*, 701:54–69, 2017.

[25] C. Carugno, M. Ferrari Dacrema, and P. Cremonesi. Evaluating the job shop scheduling problem on a D-Wave quantum annealer. *Scientific Reports*, 12(1):6539, 2022.

[26] G. Chapuis, H. Djidjev, G. Hahn, and G. Rizk. Finding maximum cliques on the D-Wave quantum annealer. *Journal of Signal Processing Systems*, 91:363–377, 2019.

[27] T. Chen and M. Li. Multi-objectivizing software configuration tuning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 453–465, 2021.

[28] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, et al. Reformulating software engineering as a search problem. *IEE Proceedings-Software*, 150(3):161–175, 2003.

[29] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[30] D-Wave Systems Inc. D-Wave White Paper: D-Wave Advantage Processor Overview. https://web.archive.org/web/20220703165806/https://www.dwavesys.com/media/s3qbjp3s/14-1049a-a_the_d-wave_advantage_system_an_overview.pdf.

[31] D-Wave Systems Inc. D-Wave White Paper: Ocean Programs for Beginners. https://web.archive.org/web/20220808194536/https://www.dwavesys.com/media/fmtj2fw3/20210920_ofbguide.pdf.

[32] D-Wave Systems Inc. D-Wave White Paper: Problem Formulation Guide. 2022. https://web.archive.org/web/20220703165755/https://www.dwavesys.com/media/bu0lh5ee/problem-formulation-guide-2022-01-10.pdf.

[33] C. De La Higuera, J.-C. Janodet, É. Samuel, G. Damiand, and C. Solnon. Polynomial algorithms for open plane graph and subgraph isomorphisms. *Theoretical Computer Science*, 498:76–99, 2013.

[34] D. Deutsch. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, 1985.

[35] N. G. Dickson, M. Johnson, M. Amin, R. Harris, F. Altomare, A. Berkley, P. Bunyk, J. Cai, E. Chapple, P. Chavez, et al. Thermally assisted quantum annealing of a 16-qubit problem. *Nature Communications*, 4(1):1903, 2013.

[36] H. N. Djidjev, G. Chapuis, G. Hahn, and G. Rizk. Efficient combinatorial optimization using quantum annealing. *arXiv preprint arXiv:1801.08653*, 2018.

[37] A. Ekert, P. Hayden, and H. Inamori. Basic concepts in quantum computation. In *Coherent atomic matter waves: 27 July–27 August 1999*, pages 661–701. Springer, 2001.

[38] E. Farhi, J. Goldstone, S. Gutmann, J. Lapan, A. Lundgren, and D. Preda. A quantum adiabatic evolution algorithm applied to random instances of an NP-Complete problem. *Science*, 292(5516):472–475, 2001.

[39] F. Farmahinifarahani, V. Saini, D. Yang, H. Sajnani, and C. V. Lopes. On precision of code clone detection tools. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 84–94. IEEE, 2019.

[40] R. P. Feynman et al. Simulating physics with computers. *International Journal of Theororetical Physics*, 21(6/7), 2018.

[41] A. B. Finnila, M. A. Gomez, C. Sebenik, C. Stenson, and J. D. Doll. Quantum annealing: A new method for minimizing multidimensional functions. *Chemical Physics Letters*, 219(5-6):343–348, 1994.

[42] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, pages 321–330, 2008.

[43] X. Gao, S.-T. Wang, and L.-M. Duan. Quantum supremacy for simulating a translation-invariant Ising spin model. *Physical Review Letters*, 118(4):040502, 2017.

[44] F. Glover, G. Kochenberger, R. Hennig, and Y. Du. Quantum bridge analytics I: a tutorial on formulating and using QUBO models. *Annals of Operations Research*, 314(1):141–183, 2022.

[45] H. Goto, K. Tatsumura, and A. R. Dixon. Combinatorial optimization by simulating adiabatic bifurcations in nonlinear Hamiltonian systems. *Science Advances*, 5(4):eaav2372, 2019.

[46] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61, 2012.

[47] I. Hen and A. Young. Solving the graph-isomorphism problem with a quantum annealer. *Physical Review A*, 86(4):042310, 2012.

[48] R. M. Hierons, M. Li, X. Liu, S. Segura, and W. Zheng. Sip: Optimal product selection from feature models using many-objective evolutionary optimization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(2):1–39, 2016.

[49] Y. Higo and S. Kusumoto. Enhancing quality of code clone detection with Program Dependency Graph. In *2009 16th Working Conference on Reverse Engineering*, pages 315–316. IEEE, 2009.

[50] Y. Higo and S. Kusumoto. Code clone detection on specialized PDGs with heuristics. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 75–84, 2011.

[51] Y. Hou, D. Ouyang, X. Tian, and L. Zhang. Evolutionary many-objective satisfiability solver for configuring software product lines. *Applied Intelligence*, pages 1–24, 2022.

[52] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability*, 70(1):304–318, 2020.

[53] A. Ibias, L. Llana, and M. Núñez. Using ant colony optimisation to select features having associated costs. In *IFIP International Conference on Testing Software and Systems*, pages 106–122. Springer, 2021.

[54] IBM. IBM Quantum, 2021. https://quantum-computing.ibm.com/.

[55] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[56] J. Job, A. Mott, J.-R. Vlimant, D. Lidar, and M. Spiropulu. Solving a Higgs detection optimization problem with quantum annealing for machine learning. In *APS March Meeting Abstracts*, volume 2018, pages S28–008, 2018.

[57] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.

[58] T. Kadowaki and H. Nishimori. Quantum annealing in the transverse Ising model. *Physical Review E*, 58(5):5355, 1998.

[59] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering*, 28(7):654–670, 2002.

[60] T. Kanao and H. Goto. Simulated bifurcation assisted by thermal fluctuation. *Communications Physics*, 5(1):153, 2022.

[61] K. Karimi, N. G. Dickson, F. Hamze, M. H. Amin, M. Drew-Brook, F. A. Chudak, P. I. Bunyk, W. G. Macready, and G. Rose. Investigating the performance of an adiabatic quantum optimization processor. *Quantum Information Processing*, 11:77–88, 2012.

[62] J. Kobler, U. Schöning, and J. Torán. *The graph isomorphism problem: its structural complexity*. Springer Science & Business Media, 2012.

[63] G. Kochenberger, J.-K. Hao, F. Glover, M. Lewis, Z. Lü, H. Wang, and Y. Wang. The unconstrained binary quadratic programming problem: a survey. *Journal of Combinatorial Optimization*, 28:58–81, 2014.

[64] G. A. Kochenberger, F. Glover, B. Alidaee, and C. Rego. A unified modeling and solution framework for combinatorial optimization problems. *OR Spectrum*, 26(2):237–250, 2004.

[65] R. Kumar and P. K. Singh. Assessing solution quality of biobjective 0-1 knapsack problem using evolutionary and heuristic algorithms. *Applied Soft Computing*, 10(3):711–718, 2010.

[66] Y. Kumar and P. Gupta. External memory layout vs. schematic. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(2):1–20, 2009.

[67] S. C. Leung, D. Zhang, C. Zhou, and T. Wu. A hybrid simulated annealing metaheuristic algorithm for the two-dimensional knapsack packing problem. *Computers & Operations Research*, 39(1):64–73, 2012.

[68] M. Lewis and F. Glover. Quadratic unconstrained binary optimization problem preprocessing: Theory and empirical analysis. *Networks*, 70(2):79–97, 2017.

[69] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *29th International Conference on Software Engineering (ICSE'07)*, pages 106–115, 2007.

[70] A. Lucas. Ising formulations of many NP problems. *Frontiers in Physics*, 2:5, 2014.

[71] Y. Manin. *Computable and Uncomputable*. Sovetskoye Radio, Moscow, 1980.

[72] F. McAndrew. Adiabatic quantum computing to solve the maxcut graph problem. *University of Melbourne School of Mathematics*, 2020.

[73] C. McCreesh, P. Prosser, C. Solnon, and J. Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *Journal of Artificial Intelligence Research*, 61:723–759, 2018.

[74] C. McGeoch and P. Farré. D-Wave White Paper: Hybrid solver for discrete quadratic models, 2020. https://web.archive.org/web/20221007195917/https://www.dwavesys.com/media/ssidd1x3/14-1050a-a_hybrid_solver_for_discrete_quadratic_models.pdf.

[75] C. C. McGeoch. Adiabatic quantum computation and quantum annealing: Theory and practice. *Synthesis Lectures on Quantum Computing*, 5(2):1–93, 2014.

[76] C. C. McGeoch, R. Harris, S. P. Reinhardt, and P. I. Bunyk. Practical annealing-based quantum computing. *Computer*, 52(6):38–46, 2019.

[77] B. D. McKay and A. Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014.

[78] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider. An empirical study of the impacts of clones in software maintenance. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 242–245. IEEE, 2011.

[79] T. Morimae, K. Fujii, and J. F. Fitzsimons. Hardness of classically simulating the one-clean-qubit model. *Physical Review Letters*, 112(13):130502, 2014.

[80] M. A. Nielsen and I. L. Chuang. Programmable quantum gate arrays. *Physical Review Letters*, 79(2):321, 1997.

[81] M. Ohzeki and H. Nishimori. Quantum annealing: An introduction and new developments. *Journal of Computational and Theoretical Nanoscience*, 8(6):963–971, 2011.

[82] K.-M. Osei-Bryson and A. Joseph. Applications of sequential set partitioning: a set of technical information systems problems. *Omega*, 34(5):492–500, 2006.

[83] D. O'Malley, V. V. Vesselinov, B. S. Alexandrov, and L. B. Alexandrov. Non-negative/binary matrix factorization with a D-Wave quantum annealer. *PloS one*, 13(12):e0206653, 2018.

[84] E. Pelofske, G. Hahn, and H. N. Djidjev. Solving large minimum vertex cover problems on a quantum annealer. *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019.

[85] J. Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.

[86] K. L. Pudenz, G. S. Tallant, T. R. Belote, and S. H. Adachi. Quantum annealing and the satisfiability problem. *New Frontiers in High Performance Computing and Big Data*, 30:253, 2017.

[87] A. P. Punnen. *The quadratic unconstrained binary optimization problem.* Springer, 2022.

[88] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[89] E. E. Reiter and C. M. Johnson. *Limits of computation: an introduction to the undecidable and the intractable.* CRC Press, 2012.

[90] H. Reittu, V. Kotovirta, L. Leskelä, H. Rummukainen, and T. D. Räty. Towards analyzing large graphs with quantum annealing. *2019 IEEE International Conference on Big Data (Big Data)*, pages 2457–2464, 2019.

[91] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[92] C. K. Roy and J. R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 157–166. IEEE, 2009.

[93] C. K. Roy and J. R. Cordy. Benchmarks for software clone detection: A ten-year retrospective. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 26–37. IEEE, 2018.

[94] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[95] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33. IEEE, 2014.

[96] G. E. Santoro and E. Tosatti. Optimization using quantum mechanics: quantum annealing through adiabatic evolution. *Journal of Physics A: Mathematical and General*, 39(36):R393, 2006.

[97] M. A. Serrano, R. Pérez-Castillo, and M. Piattini. *Quantum Software Engineering*. Springer International Publishing, 2022.

[98] A. Sheneamer and J. Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[99] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. Ieee, 1994.

[100] J. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 16–22, 2013.

[101] J. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 16–22. IEEE, 2013.

[102] K. Tatsumura, A. R. Dixon, and H. Goto. FPGA-based simulated bifurcation machine. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 59–66, 2019.

[103] J. Tindall, M. Fishman, M. Stoudenmire, and D. Sels. Efficient tensor network simulation of IBM's Eagle kicked Ising experiment. *arXiv preprint arXiv:2306.14887*, 2023.

[104] O. Titiloye and A. Crispin. Quantum annealing of the graph coloring problem. *Discrete Optimization*, 8(2):376–384, 2011.

[105] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, pages 128–135. IEEE, 2004.

[106] D. Wave Systems Inc. Error correction features. https://docs.dwavesys.com/docs/latest/c_qpu_error_correction.html?highlight=error

[107] D. Wave Systems Inc. Choosing good problems for quantum annealing, 2020. https://www.dwavesys.com/media/s10ohrq5/dwavedoc_annealing_guide.pdf.

[108] D. S. Wild and Á. M. Alhambra. Classical simulation of short-time quantum dynamics. *PRX Quantum*, 4(2):020340, 2023.

[109] W. Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755, 1991.

[110] S. Zbinden, A. Bärtschi, H. Djidjev, and S. Eidenbenz. Embedding algorithms for quantum annealers with Chimera and Pegasus connection topologies. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings*, pages 187–206. Springer, 2020.

[111] J. Zhao. Quantum software engineering: Landscapes and horizons. *arXiv preprint arXiv:2007.07047*, 2020.

[112] K. M. Zick, O. Shehab, and M. French. Experimental quantum annealing: case study involving the graph isomorphism problem. *Scientific Reports*, 5(1):1–11, 2015.

Appendix A

A.1 Complete Code Clone Dataset

Tables of the Original Code Clones and their Type 2 and Type 3 code clones.

Table A.1: Original Code Fragments.

s1.py

```python
import cmath
num = 1+2j
num_sqrt = cmath.sqrt(num)
print('The square root of {0} is {1:0.3f}+{2:0.3f}j'.format(num,num_sqrt.real,num_sqrt.imag))
```

s2.py

```python
s = "quantum"
reverse = s[::-1]
print("Reverse of", s, "is", reverse)
```

s3.py

```python
import cmath
a = 1
b = 5
c = 6
d = (b**2) - (4*a*c)
sol1 = (-b-cmath.sqrt(d))/(2*a)
sol2 = (-b+cmath.sqrt(d))/(2*a)
print('The solution are {0} and {1}'.format(sol1, sol2))
```

s4.py

```python
lower = 100
upper = 2000
for num in range(lower, upper + 1):
    order = len(str(num))
    s = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        s += digit ** order
        temp //= 10
    if num == s:
        print(num)
```

s5.py

```python
def sequence(start, stop):
   builder = []
   i = start
   while (i < stop):
      if (i > start):
         builder.append(',')
      builder.append(i)
      i += 1
   return ''.join(builder)
```

s6.py

```python
num = 7
factorial = 1
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1,num + 1):
       factorial = factorial*i
   print("The factorial of",num,"is",factorial)
```

Table A.2: Type 2 Code Clone Fragments.

s1_t2_v1.py

```python
import cmath
n = -4+5j
n_sqrt = cmath.sqrt(n)
print('The square root of {0} is {1:0.3f}+{2:0.3f}j'.format(n, n_sqrt.real, n_sqrt.imag))
```

s2_t2_v1.py

```python
text = "python"
rev = text[::-1]
print("Reverse of", text, "is", rev)
```

s3_t2_v1.py

```python
import cmath
a = 1
b = 4
c = 4
d = (b**2) - (4*a*c)
sol1 = (-b-cmath.sqrt(d))/(2*a)
sol2 = (-b+cmath.sqrt(d))/(2*a)
print('The solution are {0} and {1}'.format(sol1, sol2))
```

s4_t2_v1.py

```python
l = 100
u = 2000
for num in range(l, u + 1):
    order = len(str(num))
    s = 0
    temp = num
    while temp > 0:
        d = temp % 10
        s += d ** order
        temp //= 10
    if num == s:
        print(num)
```

s5_t2_v1.py

```python
def generate_sequence(start, stop):
   sequence_list = []
   i = start
   while (i < stop):
      if (i > start):
         sequence_list.append(',')
      sequence_list.append(i)
      i += 1
   return ''.join(sequence_list)
```

s6_t2_v1.py

```python
n = 7
factorial = 1
if n < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif n == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1, n + 1):
       factorial = factorial*i
   print("The factorial of",n,"is",factorial)
```

Table A.3: Type 3 Version 1 Code Clone Fragments.

s1_t3_v1.py

```python
import cmath
num = complex(3, 4)
num_sqrt = cmath.sqrt(num)
print('The square root of {0} is {1:0.3f}+{2:0.3f}j'.format(num,num_sqrt.real,num_sqrt.imag))
```

s2_t3_v1.py

```python
s = "quantum"
reverse = s[::-1]
```

s3_t3_v1.py

```python
a = 1
b = 5
c = 6
d = b**2 - 4*a*c
sol1 = (-b-d**(1/2))/(2*a)
sol2 = (-b+d**(1/2))/(2*a)
print(f"The solutions are {sol1} and {sol2}")
```

s4_t3_v1.py

```python
lower = 100
upper = 2000
for num in range(lower, upper + 1):
    order = len(str(num))
    s = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        s += digit ** order
        temp //= 10
    if num == s and num % 2 == 0:
        print(num)
```

s5_t3_v1.py

```python
def sequence(start, stop):
   builder = []
   for i in range(start, stop):
      if (i > start):
         builder.append(',')
      builder.append(i)
   return ''.join(builder)
```

s6_t3_v1.py

```python
num = 7
factorial = 1
i = 1
while i <= num:
    factorial *= i
    i += 1
print("The factorial of", num, "is", factorial)
```

Table A.4: Type 3 Version 2 Code Clone Fragments.

s1_t3_v2.py

```python
import cmath
num = 1+2j
num_sqrt = cmath.sqrt(num)
```

s2_t3_v2.py

```python
s = input("Enter a string: ")
reverse = s[::-1]
print("Reverse of", s, "is", reverse)
```

s3_t3_v2.py

```python
a, b, c = 1, 5, 6
d = b * b - 4 * a * c
sol1 = (-b - (d ** 0.5)) / (2 + a)
sol2 = (-b + (d ** 0.5)) / (2 + a)
print("The solutions are {0} and {1}".format(sol1, sol2))
```

s4_t3_v2.py

```python
lower = 100
upper = 2000
for num in range(lower, upper + 1):
    order = len(str(num))
    s = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        s += digit ** order
        temp //= 10
```

s5_t3_v2.py

```python
def sequence(start, stop):
   builder = []
   i = start
   while (i < stop):
      if (i > start):
         builder.append('|')
      builder.append(i)
      i += 1
```

s6_t3_v2.py

```python
num = 7
factorial = 1
for i in range(2, num + 1):
    factorial *= i
print("The factorial of", num, "is", factorial)
```