Forschungszentrum Jülich
GmbH
Institute for Advanced Simulation
Jülich Supercomputing Centre

Fachhochschule Aachen
Fachbereich Elektrotechnik und
Informationstechnik

Computer Science, Bachelor of Science

Comparison of Factoring Algorithms on the D-Wave Quantum Annealer

---

Philipp Hanussek

23.02.2000, Leverkusen

Sommersemester 2024

Examined by

Dr. Dennis Willsch, Forschungszentrum Jülich, Jülich Supercomputing Centre
Prof. Dr. Dr.-Ing. Georg Hoever, Fachhochschule Aachen

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.
Stellen, die wortwörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.
Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.
Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, den

Abstract

The goal of this work is to implement and assess different approaches for solving the factoring problem on quantum annealers. We identify three promising approaches that use custom and heuristic embedding and experimentally test their performance on the Advantage quantum annealer by D-Wave Systems Inc.

To reduce terms of higher order than quadratic, we formulate an approach that takes into account the coefficient of the term to be reduced, and we show experimentally that it produces valid models for smaller problem sizes.

We evaluate the impact of using individual per-qubit offsets and find that this feature can significantly improve the success frequencies for some problem sizes. For others, applying offsets can lead to a decrease in success frequencies.

We find that all three examined factoring approaches exhibit a scaling with problem size that is qualitatively similar to random drawing. Generally, all methods fail to find solutions for larger problem sizes. On average, the success frequencies are only $10 - 100$ times higher than randomly drawing each bit of p and q. However, the approach with custom embedding is able to find ground states even for larger problem sizes, indicating a problem formulation that is well suited for the quantum annealer.

Contents

Acronyms

BQM binary quadratic model. 8, 14, 22, 28–30, 34, 39–41, *Glossary:* BQM

CFA Controlled Full-adder. 13, 21, 22, 24, 25, 27, 32, 34, 36, 37, *Glossary:* CFA

LSB least significant bit. 14, 16, 19, 21, 34

MSB most significant bit. 14, 19, 34

QA quantum annealing. 8, 12, 13, 21, 38

QPU quantum processing unit. 10, 11, 13, 22, 24, 25, 27, 30, 34, 43

QUBO quadratic unconstrained binary optimization. 8, 9, 39, *Glossary:* QUBO

SDK software development kit. 14

Glossary

BQM binary quadratic model, Ising or quadratic unconstrained binary optimization (QUBO) problem. 8

CFA controlled full-adder, full-adder with an additional input that activates or deactivates the first input variable . 13

ground state lowest energy state in a binary quadratic model, eigenstate corresponding to the lowest eigenvalue . 9, 10, 14, 18, 19, 26, 28, 34, 36, 42

Hamiltonian $2^N \times 2^N$ matrix for N qubits describing a system. 8–10, 12, 18, 25

Ising objective function of N variables $s = [s_i, \ldots, s_N]$ corresponding to physical Ising spins, where h_i are the biases and $J_{i,j}$ the couplings (interactions) between spins. 8–10, 12–15, 18, 22, 34, 36, 37

Lagrange penalty multiplier. 18, 27, 28

QUBO quadratic unconstrained binary optimization, objective function: $E(x) = \sum_{i \leq j} x_i Q_{i,j} x_j$ where $x_i \epsilon \{0, 1\}$. 8

spin floppiness flipping the spin leads to another classical state with the same energy. 25

Chapter 1

Introduction

The factoring problem has been a central mathematical problem from ancient Greek times to modern cryptography [1]. It is defined as finding the non-trivial divisors of a positive composite integer. In this work, we consider the factorization of bi-primes $N = pq$ as they are considered to be the hardest instances of the factoring problems. Our motivation is to assess current state-of-the-art methods for solving the factoring problem on quantum annealers. In particular, we will examine three different approaches to formulating the factoring problem. Quantum annealing (QA), or analog quantum computing, is one of two major quantum computing paradigms. Quantum annealers are used to solve optimization problems in the form of an Ising Hamiltonian. Digital, or gate-based quantum computing, on the other hand, does not focus on one particular class of problems and allows for the implementation of a wider range of algorithms. Gate-based quantum computers are also called universal quantum computers because in principle, every computer program can be implemented by means of a quantum gate circuit in polynomial time. Due to current limitations in quantum computing hardware, many problems still perform better on classical computers or cannot be implemented on quantum computers at all.

This thesis is structured as follows: In this chapter we introduce briefly the theoretical background of QA and the machine that we use for experiments, the D-Wave Advantage quantum annealer. Additionally, an introduction to the factoring problem will be provided. The second chapter describes the theoretical background of the three examined methods and how and to which extent we implemented them. In the following chapter, we present experimental results related to the performance of the three approaches that were obtained on the D-Wave quantum annealer. Finally, the findings of our study will be presented in the last chapter of conclusions.

1.1 Quantum Annealing

Quantum annealers are designed to solve a specific problem class, the so-called binary quadratic model (BQM). This can either be an Ising or quadratic unconstrained binary optimization (QUBO) model. Both models are equivalent; an Ising model can be converted to a QUBO model and vice versa. The

difference is in the range of the two-valued problem variables. An Ising model is a representation of the problem variables in terms of physical spins $s_i \in -1, +1$ of the form

$$E(s) = \sum_{i=1} h_i s_i + \sum_{i<j} J_{i,j} s_i s_j \tag{1.1}$$

whereas a QUBO model is defined as

$$E(x) = \sum_{i \leq j} x_i Q_{i,j} x_j \tag{1.2}$$

with $x_i \in \{0, 1\}$. Both problem formulations consist of weighted linear and quadratic terms. Although at first glance, there seem to be only quadratic terms in the QUBO notation, $Q_{i,i}$ is the weight of the linear terms, as $x_i^2 = x_i$ for binary variables.

The Ising problem is NP-complete, which means that all NP problems can be mapped to Ising models using a polynomial number of additional steps. Efficient procedures for finding the Ising formulation have been found for many problems, such as coloring and tree problems (e.g. Travelling Salesman) [2]. For any gate-based quantum circuit an equivalent formulation for quantum annealers can be found. It is, however, not possible yet to solve these formulations on the quantum annealer that we use, the D-Wave quantum annealer, as interactions of higher order than quadratic [3] or successive back-and-forth annealing are required [4].

The basic unit in quantum computation is a quantum bit (qubit), which is defined as superposition state of $|0\rangle$ and $|1\rangle$

$$|\Psi\rangle = \alpha \cdot |0\rangle + \beta \cdot |1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad \text{with} \quad \alpha, \beta \in \mathbb{C}, \quad |\alpha|^2 + |\beta|^2 = 1. \tag{1.3}$$

The first step in the annealing process is to initialize the system in a ground state of an initial Hamiltonian H_D that can be easily constructed, the so-called driver Hamiltonian. A common choice is

$$H_D = -\sum_{i=1}^{N} \sigma_x^{(i)} \tag{1.4}$$

for N qubits. σ_x is the Pauli-X Matrix

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{1.5}$$

with eigenvalues $+1$ and -1 and eigenvectors corresponding to

$$\begin{aligned} |+\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \\ |-\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}. \end{aligned} \tag{1.6}$$

Therefore, the ground state of $-\sigma_x$ is the $|+\rangle$ state.

The problem Hamiltonian is defined as

$$H_P = \sum_{i=1}^{N} h_i \sigma_z^{(i)} + \sum_{i=1}^{N} \sum_{j=1}^{i-1} J_{ij}^{(i)} \sigma_z^{(i)} \otimes \sigma_z^{(j)}, \tag{1.7}$$

where i and j are the qubit indices. σ_z is the Pauli-Z matrix,

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{1.8}$$

with eigenvectors corresponding to

$$\begin{aligned} |1\rangle &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \\ |0\rangle &= \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \end{aligned} \tag{1.9}$$

The ground state of H_P is defined to be the solution to the Ising problem.

During the annealing process, the system is slowly brought from H_D to H_P according to the so-called annealing schedules $A(s)$ and $B(s)$:

$$\begin{aligned} H(s) &= A(s)H_D + B(s)H_P \\ &= -A(s) \sum_{i=1}^{N} \sigma_x^{(i)} + B(s) \sum_{i=1}^{N} h_i \sigma_z^{(i)} + B(s) \sum_{i=1}^{N} \sum_{j=1}^{i-1} J_{ij}^{(i)} \sigma_z^{(i)} \otimes \sigma_z^{(j)}, \end{aligned} \tag{1.10}$$

where $s \in [0, 1]$ is the normalized time. $A(s)$ decreases the driving terms in H_D with s, and $B(s)$ activates the terms of the problem Hamiltonian H_P. The standard anneal schedules on the D-Wave quantum processing units (QPUs) are not linear, as can be seen in Figure 1.1. Each QPU has its own default anneal schedule, modified if necessary. In Equation (1.10) the driving is homogeneous with both schedules being controlled by a single timing signal. Another possibility, which will be discussed in chapter 2, is to make the timing signal qubit-dependent [5].

If the annealing process happens slowly enough, one expects to measure the solution of the problem at $s = 1$ with high probability according to the adiabatic theorem [7].

1.2 D-Wave Quantum Annealer

All results in this work have been obtained on quantum annealers produced by D-Wave Systems, a Canadian company that manufactures quantum annealers for commercial use. Its largest currently available device is the Advantage quantum annealer, which has more than 5600 superconducting qubits and around 40000 couplers. The exact number depends on the particular QPU and the fabrication process. On the D-Wave Advantage QPU not all qubits are connected to each other, but each qubit

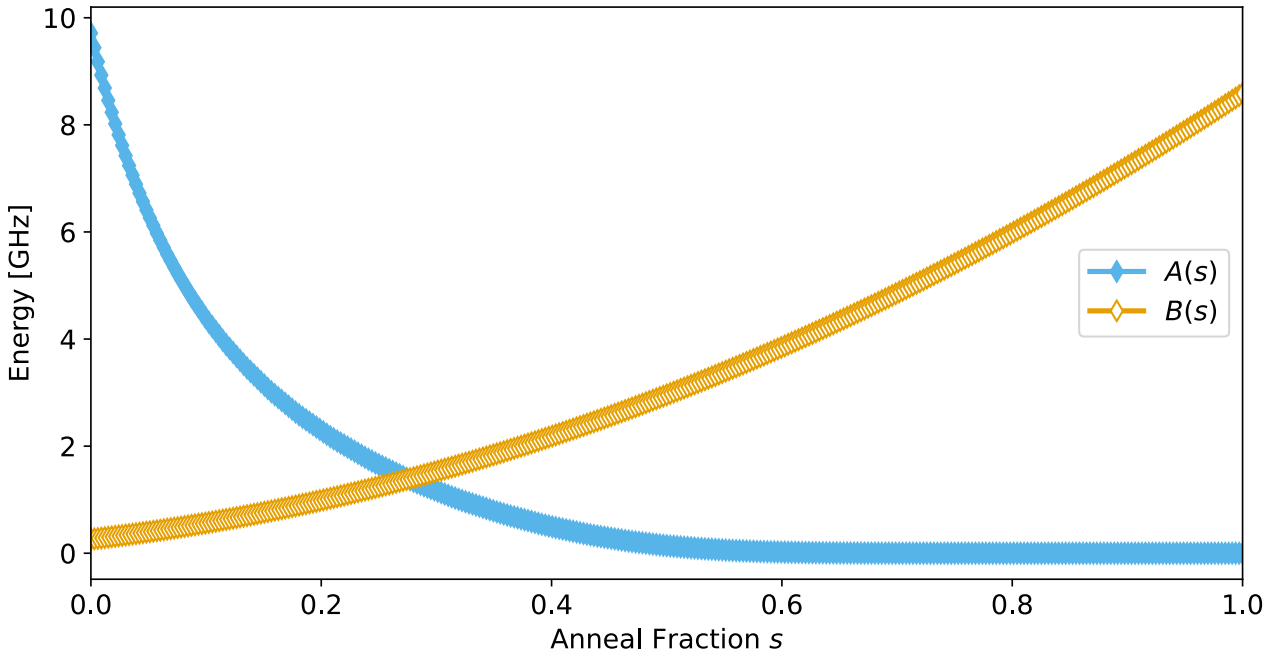


Figure 1.1: Illustration of default anneal schedules for the D-Wave Advantage 5.1 QPU. Solid markers represent $A(s)$, open markers represent $B(s)$. Courtesy of [6].

couples to 15 others on average (except for those on the outsides of the QPU). The QPU consists of a 15x15x3 grid of unit cells, with each cell containing 8 qubits (so-called Pegasus graph, Figure 1.2). The QPU can be represented as a graph with qubits as nodes and couplers as edges.

All calculations are performed either on the D-Wave Advantage 4.1 (located in Canada) or on the D-Wave Advantage 5.4, which is located in the Jülich Supercomputing Centre at Forschungszentrum Jülich. It is important to note that 4 and 5 in the QPUs' names are not versioning numbers, but identifiers. The versioning for each QPU is indicated after the dot (for example, the Advantage 5.4 identifies QPU number 5 in version 4). Both QPUs have received the latest major update (Advantage, performance update).

Not all qubits and couplers are functional, often because the fabrication process is very complex. On the Advantage 4.1 the yield of the working graph (the ratio of working qubits to an ideal QPU) is around 97.69% and on the Advantage 5.1 it is around 97.47%.

1.3 Factoring Problem

The factoring problem is defined by finding the non-trivial factors p and q of a composite integer N, so that $N = p \times q$. The most promising approach for solving the factoring problem in classical computing is the *general number field sieve* with sub-exponential time complexity. The current record on any computing device is held by Boudot et al., who succeeded in factoring a 250-decimal digits (829 bits) number using this method [8]. With Shor's algorithm for gate-based quantum computers,

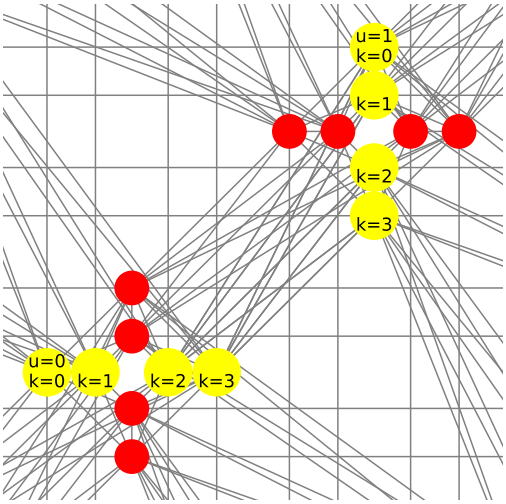


Figure 1.2: Visualization of two adjacent Pegasus unit cells. Each qubit in the cell is identified by indices u and k. u denotes the orientation (0=horizontal, 1=vertical) and k is the qubit's position in the row/column. Courtesy of [6].

it is in theory possible to factor integer numbers in polynomial time. In 2012, $15 = 5 \times 3$ was factored on a quantum computer using Shor's algorithm [9]. A large-scale simulation of Shor's algorithm on classical GPUs succeeded in factoring $549,755,813,701 = 712,321 \times 771,781$ [10].

One of the most promising methods for quantum annealers has been proposed by Ding et al. [11] in 2023, which succeeded in the factorization of $8,219,999 = 32,749 \times 251$. The method will be examined in detail in chapter 2.

To factor a composite integer N on a QA device, the factoring problem first needs to be encoded in an objective function (the Ising Hamiltonian). In the problem formulation (see Equation (1.7)), each bit of p, q and N is represented by one or multiple qubits $\sigma_z^{(i)}$. At the end of the annealing process, each qubit's state is measured, and the corresponding bit string can be interpreted as integers p, q and N.

Chapter 2

Theoretical Background and Implementation

We identified three promising approaches for formulating the corresponding objective function to the factoring problem, that can be solved on a QA device:

1. Multiplication circuit method
2. Modified multiplication table method (Shuxian Jiang et al. [12])
3. Controlled Full-adder (CFA) method (Jingwen Ding et al. [11])

All three approaches are based on a shift-and-add binary multiplication table (see Table 2.1). However, each approach derives the final objective function differently: The first method, the multiplication circuit, performs all required arithmetic operations by *half-adder*, *full-adder*, and *and* gates. Each gate has its own objective function, and the sum of these objective functions yields the final cost function.

The modified multiplication table method works differently: The necessary binary addition and multiplication operations are implemented using the linear and quadratic terms of the Ising model. Additionally, the multiplication table is split into blocks to reduce the number of carry bits. A particular configuration of the modified multiplication method is the *direct method* that we also examine.

The third method that we examine, the CFA method, is similar in theory to the first method (the multiplication circuit), but differs in the implementation: Like the multiplication circuit, the CFA method is based on the multiplication table, where arithmetic operations are performed by *full-adder* and *and* gates. The biggest difference between the CFA method and both previous methods is that the CFA method takes into account the QPU's strucutre and generates a final objective function that can be immediately processed on the QPU. The models obtained with the other two methods first need to be adapted to the Pegasus QPU in an *embedding* step, rendering them potentially more complex.

In this chapter, we present the theoretical background of these methods and their implementation.

	2^6	2^5	2^4	2^3	2^2	2^1	2^0
p					p_2	p_1	p_0
q				q_3	q_2	q_1	q_0
					p_2q_0	p_1q_0	p_0q_0
$+$				p_2q_1	p_1q_1	p_0q_1	
$+$			p_2q_2	p_1q_2	p_0q_2		
$+$		p_2q_3	p_1q_3	p_0q_3			
N	n_6	n_5	n_4	n_3	n_2	n_1	n_1

Table 2.1: 3×4-bit shift-and-add binary multiplication table.

2.1 Multiplication Circuit Method

In this section we introduce the multiplication circuit method, which is included in D-Wave's software development kit (SDK) [13].

2.1.1 Theory

One approach to constructing a BQM of the factoring problem is to generate a shift-and-add multiplication table with the necessary half- and full-adder gates. The cost functions for these gates are known [14] and can be added up to represent any fixed-size multiplication circuit.

A visualization of a 3×4-bit multiplication circuit is given in Figure 2.1. As we know both factors to be prime, the factors' least significant bit (LSB) and most significant bit (MSB) are set to 1, reducing the number of variables required in the circuit. Binary multiplication is performed by *and* gates that are not specifically displayed in the figure. Many helper variables (so-called auxiliary variables) are generated to transmit sum and carry bits in between gates. Each gate with its input and output variables has a separate cost function. Generally, objective functions are designed in a way that their global minimum (the ground state in the Ising model) is zero. Therefore, we can sum up all of the gates' objective functions to obtain the final cost function without altering the global minimum. To obtain an objective function for a specific semiprime N for the multiplication circuit in Figure 2.1, we fix the bits of N by replacing $n_{1...6}$ with their binary representation. Now the lowest energy state represents the solution to the factoring problem. In this circuit we need a total of 13 qubits (1 for p, 2 for q, 7 carries and 3 sums).

2.1.2 Implementation

D-Wave offers multiple methods for generating common models and optimization problems. One of them is the `dimod.generators.multiplication_circuit`. An explicit example of how to construct and solve a BQM for the factoring problem with $N = 91$ is given in Appendix B.

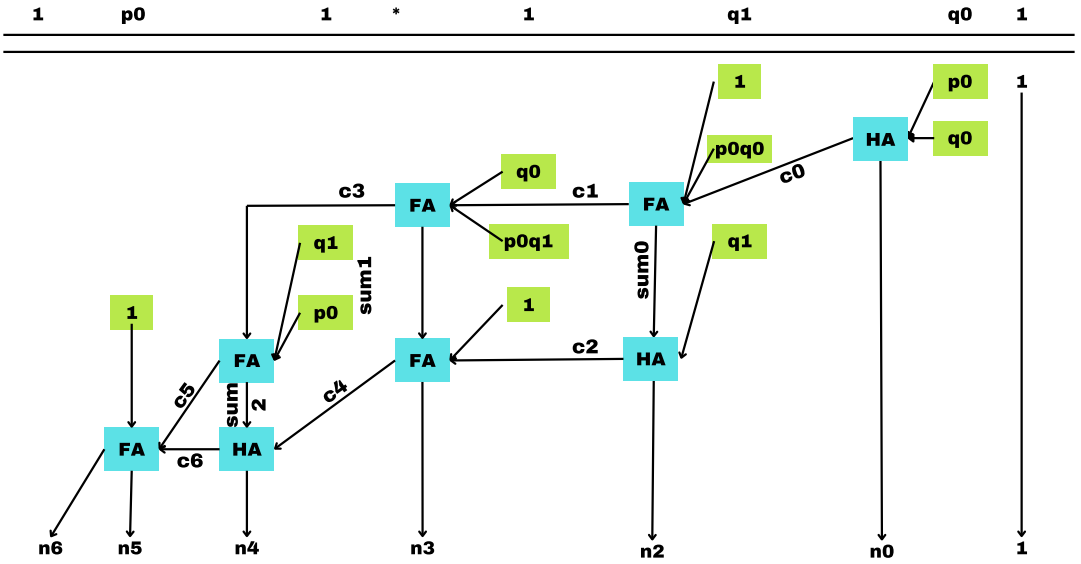


Figure 2.1: Multiplication circuit for a 3×4-bit multiplication of prime factors using *half-adder* (HA), *full-adder* (FA) and *and* gates (binary multiplication, not displayed). Arrow annotations are the names of the helper variables (auxiliary variables).

2.2 Modified Multiplication Table Method

Another possibility to implement the factoring problem on a quantum annealer is to obtain the cost function directly from the multiplication table, without generating adder gates. Arithmetic operations such as addition and multiplication of two binary variables can be implemented using the linear and quadratic terms of the Ising model. To multiply more than two binary variables together, higher order terms need to be reduced to at most quadratic terms. We examine different strategies for performing this reduction in chapter 3.

Jiang et al. [12] describe a strategy to reduce the number of linear terms, and therefore also the number of qubits in the model, by splitting the binary multiplication into blocks (*modified* multiplication table). Introducing blocks to the multiplication table reduces the number of carry variables. In the first subsection 2.2.1, we demonstrate the application of this method using the example of factoring $N = 91$. The second subsection 2.2.2 details its implementation.

2.2.1 Theory

In this subsection we give an example on how to obtain a cost function for the factoring of $N = 91$ with 3 and 4 bit factors using the modified multiplication table method. In the examples provided by Jiang et al., the proposed block size for multiplication tables of this size is 2. To reduce higher-order

	2^6	2^5	2^4	2^3	2^2	2^1	2^0
p					1	p_0	1
q				1	q_1	q_0	1
					1	p_0	1
+				q_0	p_0q_0	q_0	
+			q_1	p_0q_1	q_1		
+		1	p_0	1			
+					c_0		
+			c_2	c_1			
+		c_4	c_3				
+	c_6	c_5					
+	c_7						
N	1	0	1	1	0	1	1

(a) Minimum block size (block size 1).

	2^6	2^5	2^4	2^3	2^2	2^1	2^0
p					1	p_0	1
q				1	q_1	q_0	1
					1	p_0	1
+				q_0	p_0q_0	q_0	
+			q_1	p_0q_1	q_1		
+		1	p_0	1			
+			c_1	c_0			
+	c_3	c_2					
N	1	0	1	1	0	1	1

(b) Block size 2. Blocks are marked by vertical lines.

Table 2.2: 3×4-bit multiplication table for $N = (1011011)_2 = 91$. Comparison of multiplication tables a) with block size 1 and b) with block size 2.

terms to quadratic, we refine a strategy initially formulated by Jiang et al.

To obtain a cost function with the modified multiplication table method, the following strategy is used: The binary multiplication table is split into vertical blocks according to the previously defined block size. The column containing the LSB is not taken into consideration, as the LSB of semiprimes is always 1. Each row in a block is interpreted as an integer number. If a block's sum could potentially have more binary places than the block's length, the needed amount of carry bits is appended to the next block. For each block in the table, we add the bits of p and q and the carry bits together and set them equal to the corresponding bit of N.

The number of carries is determined by the maximum possible value for each block and can be easily calculated by substituting all variables with 1. Table 2.2 exemplifies how splitting the multiplication table into blocks reduces the number of carry bits, by comparing a 3×4-bit multiplication table with block size 1 and block size 2. A block size of 1 means that each column is considered to be a block. With block size 1, 8 carry bits are required. A block size of 2 already reduces that number to 4 required carry bits.

From Table 2.2b we obtain a set of 3 equations:

$$\begin{aligned} 2(1 + p_0q_0 + q_1 - (4c_1 + 2c_0)) + p_0 + q_0 &= (01)_2 = 1 \\ 2(q_1 + p_0 + c_1 - (4c_3 + 2c_2)) + q_0 + p_0q_1 + 1 + c_0 &= (11)_2 = 3 \\ 2c_3 + 1 + c_2 &= (10)_2 = 2 \end{aligned} \tag{2.1}$$

We can verify the equation set for this example by substituting all variables accordingly, as can be seen in Table 2.3.

	2^6	2^5	2^4	2^3	2^2	2^1	2^0
p					1	1	1
q				1	1	0	1
					1	1	1
+				0	$1*0$	0	
+			1	$1*1$	1		
+		1	1	1			
+			0	1			
+	0	1					
N	1	0	1	1	0	1	1

$$2(1+1*0+1-(4*0+2*1))+1+0=(01)_2=1$$
$$2(1+1+0-(4*0+2*1))+0+1*1+1+1$$
$$=(11)_2=3$$
$$2*0+1+1=(10)_2=2$$

Table 2.3: The equation set (2.1) derived from Table 2.2b for $N = 91$ with $p = (111)_2 = 7$ and $q = (1101)_2 = 13$ is correct, as can be seen on the right by substituting all variables.

The cost function has to satisfy the following conditions:

1. the cost function's global minimum is zero for the input bits of correct p, q, and c

2. the cost function only contains linear and quadratic terms

Jiang et al. [12] define the cost function by subtracting the right side of Equation (2.1) and squaring the result to fulfill condition 1:

$$(2(1+p_0q_0+q_1-(4c_1+2c_0))+p_0+q_0-1)^2=0$$
$$(2(q_1+p_0+c_1-(4c_3+2c_2))+q_0+p_0q_1+1+c_0-3)^2=0$$
$$(2c_3+1+c_2-2)^2=0$$

The next step is to add the terms together:

$$\begin{aligned} & f(p_0,q_0,q_1,c_0,c_1,c_2,c_3) \\ =&(2(1+p_0q_0+q_1-(4c_1+2c_0))+p_0+q_0-1)^2 \\ &+(2(q_1+p_0+c_1-(4c_3+2c_2))+q_0+p_0q_1+1+c_0-3)^2 \\ &+(2c_3+1+c_2-2)^2 \\ =&0 \end{aligned} \tag{2.2}$$

The expanded preliminary cost function looks as follows. Quadratic terms can be replaced by linear

ones according to the rule $x^2 = x$ for $x = 0, 1$:

$$\begin{aligned}
&f(p_0, q_0, q_1, c_0, c_1, c_2, c_3) \\
=&68c_0c_1 - 8c_0c_2 - 16c_0c_3 - 16c_0p_0q_0 + 2c_0p_0q_1 - 4c_0p_0 - 6c_0q_0 - 12c_0q_1 + 5c_0 - 16c_1c_2 \\
&- 32c_1c_3 - 32c_1p_0q_0 + 4c_1p_0q_1 - 8c_1p_0 - 12c_1q_0 - 24c_1q_1 + 44c_1 + 68c_2c_3 - 8c_2p_0q_1 - 16c_2p_0 \\
&- 8c_2q_0 - 16c_2q_1 + 31c_2 - 16c_3p_0q_1 - 32c_3p_0 - 16c_3q_0 - 32c_3q_1 + 96c_3 + 10p_0q_0q_1 + 22p_0q_0 \\
&+ 17p_0q_1 - p_0 + 8q_0q_1 + 4q_1 + 6
\end{aligned}$$

To fulfill condition 2 and reduce higher order terms to second order terms, Jiang et al. use the cost function for the `and` gate (see Appendix A), because $x_0x_1 = x_0 \wedge x_1$ for binary numbers:

$$\begin{cases} \alpha x_0x_1x_2 = \alpha x_2x_3 + \lambda(3x_3 + x_0x_1 - 2x_0x_3 - 2x_1x_3) & \text{if } x_3 = x_0x_1 \\ \alpha x_0x_1x_2 < \alpha x_2x_3 + \lambda(3x_3 + x_0x_1 - 2x_0x_3 - 2x_1x_3) & \text{if } x_3 \neq x_0x_1 \end{cases} \tag{2.3}$$

with $\lambda > 0$. The term $3x_3 + x_0x_1 - 2x_0x_3 - 2x_1x_3$ is zero if and only if $x_3 = x_0x_1$ and adds energy to the Hamiltonian otherwise. It is weighted by a penalty multiplier λ, also known as Lagrange parameter. Jiang et al. use a fixed $\lambda = 2$. We have modified Equation (2.3) as we found that a fixed λ can lead to Ising models where the ground state does not represent the solution. There are multiple approaches for choosing a suitable penalty multiplier, which will be discussed in detail in chapter 3. In the following, we set λ equal to the absolute coefficient α of the three-body term:

$$\alpha x_0x_1x_2 = \alpha x_2x_3 + |\alpha|\,(3x_3 + x_0x_1 - 2x_0x_3 - 2x_1x_3) \text{ if } x_3 = x_0x_1 \tag{2.4}$$

Applying rule 2.3 and substituting p_0q_0 with t_0 and p_0q_1 with t_1 yields the following cost function:

$$\begin{aligned}
&f(p_0, q_0, q_1, c_0, c_1, c_2, c_3, t_0, t_1) \\
=&68c_0c_1 - 8c_0c_2 - 16c_0c_3 - 4c_0p_0 - 6c_0q_0 - 12c_0q_1 - 16c_0t_0 + 2c_0t_1 + 5c_0 - 16c_1c_2 - 32c_1c_3 \\
&- 8c_1p_0 - 12c_1q_0 - 24c_1q_1 - 32c_1t_0 + 4c_1t_1 + 44c_1 + 68c_2c_3 - 16c_2p_0 - 8c_2q_0 - 16c_2q_1 - 8c_2t_1 \\
&+ 31c_2 - 32c_3p_0 - 16c_3q_0 - 32c_3q_1 - 16c_3t_1 + 96c_3 + 80p_0q_0 + 47p_0q_1 - 116p_0t_0 - 60p_0t_1 - p_0 + 8q_0q_1 \\
&- 116q_0t_0 + 10q_1t_0 - 60q_1t_1 + 4q_1 + 174t_0 + 90t_1 + 6
\end{aligned} \tag{2.5}$$

By enumerating all possible combinations of the values of p, q, c, and t we can verify that the final cost function represents the factoring problem in this example (Table 2.4).

2.2.2 Implementation

We have implemented this method from scratch, primarily relying on the Python library SymPy for symbolic mathematics. It offers features such as basic arithmetic, simplification and expansion of polynomials and substitution.

First, we demonstrate how we generated the multiplication table with the necessary carry bits. Then,

p0	q0	q1	c0	c1	c2	c3	t0	t1	p	q	energy
1	0	1	1	0	1	0	0	1	7	13	0
0	0	1	1	0	0	0	0	0	5	13	3
0	1	1	1	0	1	0	0	0	5	15	4
1	0	0	0	0	0	0	0	0	7	9	5

Table 2.4: Verification of cost function (2.5) by enumerating all possible states using the xubo solver [15]. Only the four lowest-energy states are shown. The ground state is the only correct solution with regard to all variables p, q, c, t. Equation (2.5) therefore correctly represents the 3x4-bit factoring of $N = 91$.

	Block 0			
	0		1	
idx i	2^1	2^2	2^3	2^4
	0	1	0	1
data	p_0	1	0	0
	q_0	p_0q_0	q_0	0
	0	q_1	p_0q_1	q_1
	0	0	1	p_0

(a) Visualization of `blocks` array. Each block contains a 2D array with symbols p, q, and c.

	Block 0		Next Block	
	2^1	2^2	2^3	2^4
idx i	0	1		
data	1	1		
	1	1		
	0	1		
	0	0		
sum	0	0	0	1
	LSB			MSB

(b) 2D array. All variables have been substituted with 1. The sum's length is two greater than the block size, so two carry bits are appended to the next block.

Table 2.5: 3×4-bit multiplication table (Table 2.2b) represented as an array of blocks. Note that the order is reversed, because the LSB is stored at array position 0 and the MSB at the last position.

we show how to derive the objective function from the generated multiplication table.

Generating the multiplication table

The multiplication table is represented as an array of blocks (see Table 2.5a). Each block contains a two-dimensional array. Each row is padded with zeros, so that it can be split into blocks of equal size. Initially, only the symbols for the binary variables of the factors p and q are generated. After generating the symbols for p and q, we split the multiplication table into blocks according to the previously defined `blocksize`. Then, we loop over each block, calculate the necessary amount of carry bits (see Table 2.5 for the example of a 3×4-bit table) and add them to the variable `carryLine`. `carryLine` is appended to the entire multiplication table because carries can span multiple blocks. It is important to subtract the carries that we have added to later blocks from the current one, so that the carries do not influence the block's sum (Algorithm 1).

Algorithm 1 Carry bit generation

```
blocks ← split(table, blocksize)                    ▷ split table into blocks of size blocksize
for i ← 1, len(blocks) do
    blocks ← split(table, blocksize)                ▷ split again to consider new carries
    block ← convertToOnes(blocks[i])
    overflowBits ← len(sum(block)) − blocksize
    if overflowBits > 0 then
        carryIndxStart ← (i + 1) ∗ blocksize        ▷ first column of next block
        carryIndxEnd ← carryIndxStart + overflowBits
        carries ← getCarrySymbols(overflowbits)
        carryLine ← [0] ∗ tableWidth
        carryLine[carryIndxStart : carryIndxEnd] = carries
                                                    ▷ place carries in the right position of the next blocks
        carrySum ← sum([carryBit ∗ 2^(j+1) for j, carryBit in enumerate(carries)])
                                                    ▷ sum up carries mulitplied by their place
        carryLine[(i + 1) ∗ blocksize − 1] ← −carrySum
                                                    ▷ subtract carries from the last column of the current block
        table.append(carryLine)
    end if
end for
```

	blocks					
blockIdx	0		1		2	
	2^1	2^2	2^3	2^4	2^5	2^6
idx i	0	1	0	1	0	1
data	p_0	1	0	0	0	0
	q_0	p_0q_0	q_0	0	0	0
	0	q_1	p_0q_1	q_1	0	0
	0	0	1	p_0	1	0
	0	$-(c_0 * 2 + c_1 * 4)$	c_0	c_1	0	0
	0	0	0	$-(c_2 * 2 + c_3 * 4)$	c_2	c_3
N	1	0	1	1	0	1

Table 2.6: Final `blocks` array after adding carries to the next block and subtracting them from the current one.

Deriving the cost equation

To derive the cost equation, for each block we sum up the columns, multiply them with their placement and subtract the respective part of N.

As a first step, we add all values in a column together. The block data is represented as a two-dimensional array in a row-oriented structure, therefore all values in a row are stored in an array. SymPy offers convenient and fast functions to add array values together, which we cannot use unless we first transpose the array so that the column values are stored in an array (and not the rows, as before). We multiply each column with 2^i, to consider its placement in the block.

Here is an extract of the Python code to perform the addition:

```
self.column_eq = [[sp.Add(*column) *2**i for i, column in enumerate(block.T)]
                  for block in self.blocks ]
```

Secondly, to obtain the full cost function for each block, we sum up the columns of each block:

```
self.block_eq = [sp.Add(*block_columns) for block_columns in self.column_eq]
```

Next, we subtract the respective part of N from each block. To do so, we first reverse the binary representation of N to fit it to the multiplication table structure, where the first element is the LSB. Also we discard the LSB as we know it to be 1.

```
N_bin_rev = self.N_bin[2:-1][::-1]
```

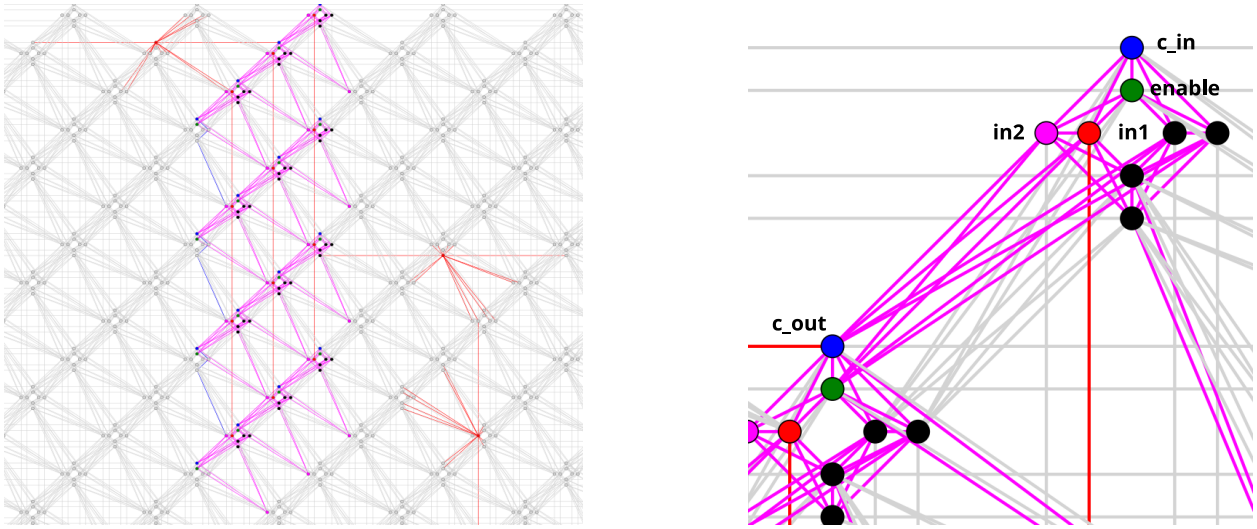
To subtract N from the block equations, we loop over each block, pick the respective part of N, reverse it back, convert it to integer and subtract it from the block equation:

```
self.cost_eq = [block_equation-int(N_bin_rev[block_inx*self.BLOCKSIZE:(block_inx+1)
                                                    * self.BLOCKSIZE][::-1],2)
                for block_inx, block_equation in enumerate(self.block_eq)]
```

The next steps are to square the equations and add them together, simplify and apply rule 2.3 to ensure that we only have linear and quadratic terms. The latter is done with SymPy's `subs` substitution function, where we can input a rule as a Python dictionary and perform the substitution (see `modifiedmultiplication/src/ProblemCreator.py` in [16]).

2.3 Controlled Full Adder Multiplier

To our knowledge, the factoring of the largest semiprime to date on a QA device was accomplished by Ding et al. using the CFA method, with the result $8,219,999 = 32,749 \times 251$. In this section, we introduce the theoretical background of this method, along with its implementation and potential tuning options.



(a) Encoding of a 3×4-bit multiplier on the QPU.

(b) CFA encoding into the Pegasus unit cell

Figure 2.2: Visualization of the multiplier encoding on the QPU. Generated with code provided in the git repository by Ding et al. [17].

2.3.1 Theory

BQMs generated with the previous methods cannot be solved directly with the quantum annealer, because they usually require logical connections between different qubits that do not physically exist on the device. Therefore, they need to be *embedded* to fit on the QPU, which means that single logical qubits are mapped to a chain of physical qubits on the device.

Both the multiplication circuit method and the modified multiplication method use *heuristic* embedding. This means that the Ising model's graph is mapped heuristically into the QPU's target graph. This can lead to increased complexity of the model, because one logical qubit might need to be represented by multiple physical qubits.

Ding et al. [11] propose to encode the multiplier directly into the Pegasus structure (see section 1.2), by taking advantage of the shift-and-add multiplier's regular structure (see Table 2.7). They introduce a new gate, the CFA gate. Like a regular full-adder it has three input lines $in1, in2, c\_in$ and two output lines out, c_out. Additionally, there is one *enable* line that enables or disables the first input line. The enable line allows for combining the multiplication and addition operations in a single gate, which can then be fit onto a Pegasus unit cell. To construct a multiplier for factors p and q of length l_p and l_q, the CFAs are placed on a $l_p \times l_q$ grid of unit cells. The CFA is an encoding of the equation $c\_out * 2 + out = (enable \wedge in\_1) + in\_2 + c\_in$. See Figure 2.2 for an encoding of the 3×4-bit multiplier into the Pegasus structure.

			p_2	p_1	p_0	$*q_0$	
		N_{03}	N_{02}	N_{01}	N_{00}		+
		p_2	p_1	p_0		$*q_1$	
	N_{14}	N_{13}	N_{12}	N_{11}			+
	p_2	p_1	p_0			$*q_2$	
N_{25}	N_{24}	N_{23}	N_{22}				+
p_2	p_1	p_0				$*q_3$	
N_{36}	N_{35}	N_{34}	N_{33}				

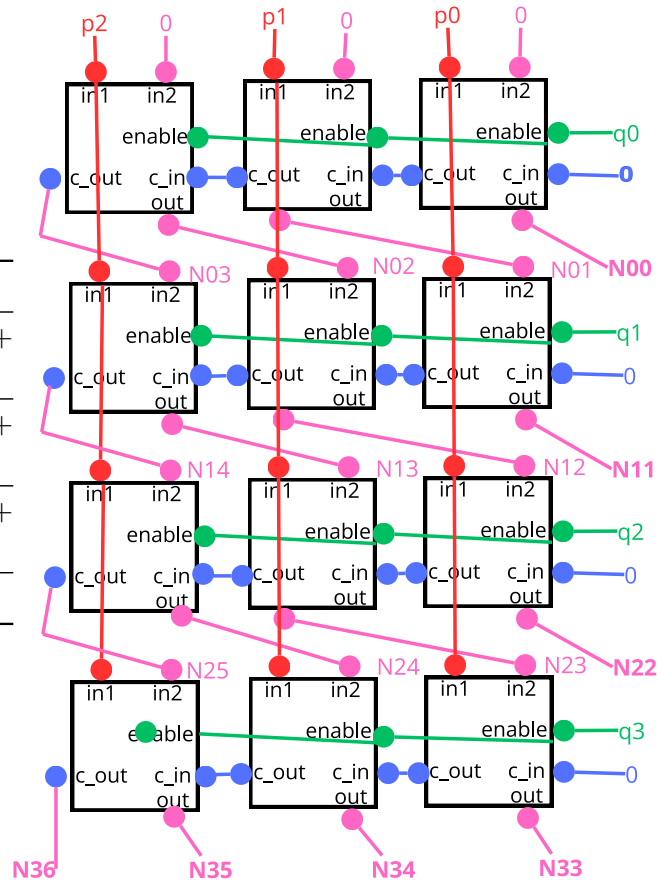
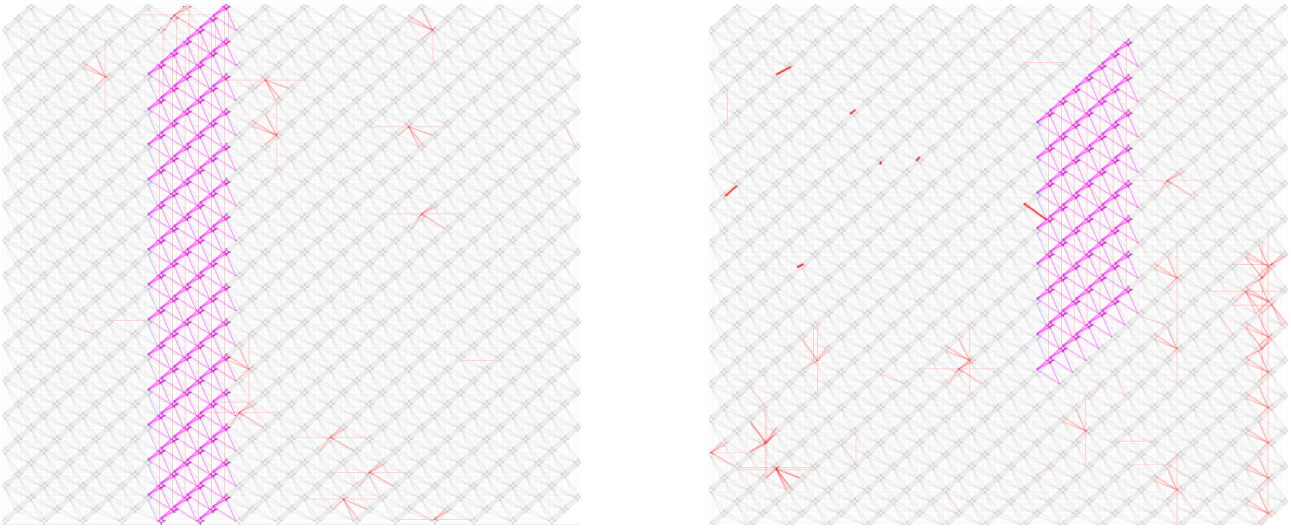


Table 2.7: Implementation of a 3×4-bit multiplier with CFAs [11, p. 5].



(a) Advantage 4.1 QPU: 17 × 6-bit multiplier

(b) Advantage 5.4 QPU: 7 × 8-bit multiplier.

Figure 2.3: Visualization of the largest multipliers implemented by us on the Advantage 4.1 QPU (left) and on the Advantage 5.4 QPU (right). Red lines outside the multiplier represent broken couplers and red dots broken qubits. Graphic generated with code provided in the git repository by Ding et al. [17]. Some broken couplers are drawn manually for better visibility.

2.3.2 Implementation

Along with the method description, Ding et al. made the corresponding git repository publicly available online [17]. It contains all the methods necessary to generate the CFA-multiplier. We made only minor changes to the format in which the results are saved to allow for more detailed analysis. Additionally, we made the following adjustments:

- assessing the feasibility of adapting the method to the Advantage 5.4 QPU
- implementing the anneal offset feature

Implementation on the Advantage System 5.4 QPU

The CFA method is adapted for the Advantage 4.1 QPU. Ideally, we would like to compare results that have been obtained on the same QPU. To implement the CFA method on the QPU with version 5.4 that is located in Forschungszentrum Jülich, we need to take into account the physical properties of that particular chip. As the multiplier is encoded directly into the QPU, all qubits and couplers that are part of the multiplier need to be working. Consequently, for a multiplier of factors with length l_p and l_q, a $l_p \times l_q$ grid of fully intact unit cells is required. To adapt the CFA method to the Advantage 5.1 QPU, we identified a region of the QPU that is free from broken qubits and couplers. We identified this region to be in the top right corner of the QPU (see Figure 2.3 for a visualization of multipliers on both QPUs).

As the 5.4 QPU has more broken qubits and more broken couplers compared to version 4.1 (see [18]),

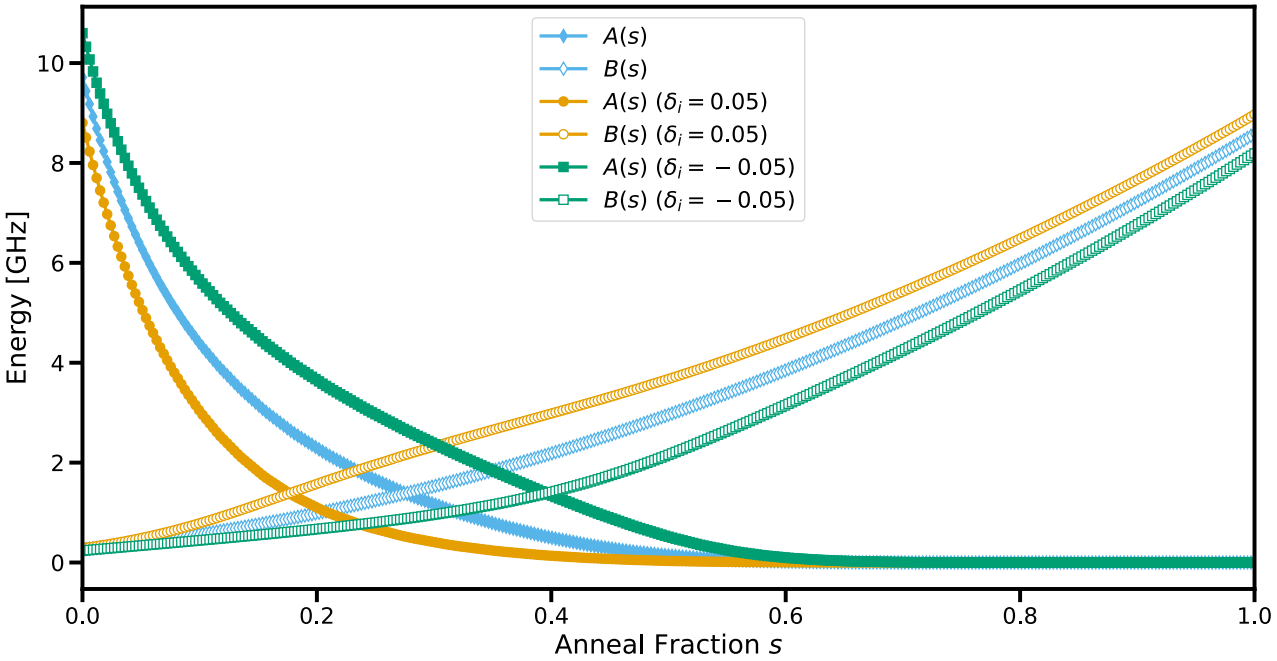


Figure 2.4: Illustration of anneal schedules when using anneal offsets. Shown are the baseline (blue diamonds), delayed (green squares), and advanced (yellow circles) schedules. Solid markers represent $A(s)$, open markers represent $B(s)$. Courtesy of [6].

the largest multiplier that we have been able to implement on version 5.4 is of dimensions 7×8 (compared to 17×6 on version 4.1). Therefore, all results for the CFA method are obtained on the Advantage System 4.1 QPU.

Tuning

The predictable problem's structure allows for the implementation of different tuning mechanisms, most notably the *anneal offset* feature. This feature delays or advances the anneal schedule (see section 1.1) individually for each qubit. With individual qubit offsets, the Hamiltonian defined in Equation (1.10) becomes

$$H_\delta(s) = -\sum_{i=1}^{N} A(s,\delta_i)\sigma_x^i + \sum_{i=1}^{N} B(s,\delta_i)h_i\sigma_z^i + \sum_{i=1}^{N}\sum_{j=1}^{i-1}\sqrt{B(s,\delta_i)B(s,\delta_j))}J_{ij}\sigma_z^i \otimes \sigma_z^i. \tag{2.6}$$

$\delta = (\delta_i, \ldots, \delta_N)$ is a vector of offsets for each individual qubit. Each qubit has individual time schedules $A(s,\delta_i)$ and $B(s,\delta_i)$ instead of general anneal schedules $A(s)$ and $B(s)$ for all qubits. The quadratic terms in H_P are annealed according to the geometrical mean of the qubits' anneal schedules $B(s,\delta_i)$ and $B(s,\delta_j)$. Figure 2.4 displays the baseline (no offsets) and modified anneal schedules. A negative offset δ_i delays the anneal process for qubit i, while a positive offset advances it. The anneal schedules are modified by the same δ_i for all qubits i.

There are multiple strategies for choosing the offset based on e.g. spin floppiness [19] or chain length

[20]. The chain length is defined as the largest number of physical qubits that represent the same logical qubit in the embedding. To calculate the per-qubit offset, we have opted for a broader approach that takes into account the general connectivity of a qubit (*effective field offsets*, Adame et al. [5]). The intuition behind this approach is to prevent early *freeze-out* of strongly linked qubits during the anneal process. *Freeze-out* means that some qubits freeze early on in the computation and do not change their values anymore, even though the system has not reached the final ground state yet. Adame et al. hypothesize that "qubits that are more strongly coupled to the rest of the system freeze out earlier than those that are only weakly coupled" [5, p. 3].

The absolute effective field of a qubit i is defined as the bias of i plus the sum of the couplers' weights J_{ij} to neighboring qubits j, where $s_{j1}, \ldots, s_{jN_i} \in [-1, +1]$ denotes some configuration of these spins:

$$F_i(s_{j_1}, \ldots, s_{j_{Ni}}) := \left| h_i + \sum_{j=j_1,\ldots,j_{N_i}} J_{ij} s_j \right| \tag{2.7}$$

The next step is to obtain the average effective field of all possible neighboring spin configurations:

$$\overline{F_i} := \frac{1}{2^{N_i}} \sum_{s_{j_1},\ldots,s_{j_{Ni}}} F_i(s_{j_1}, \ldots, s_{j_{Ni}}) \tag{2.8}$$

The last step is to normalize these values in the interval $[0, 1]$. Adame et al. denote the normalized average effective field of qubit i as r_i:

$$r_i := \frac{\overline{F_i}}{\max\limits_{k\in\{1,\ldots,N\}} \overline{F_k}} \tag{2.9}$$

The formal definition to calculate the actual offset value is

$$\delta_i(r_i) := \alpha(1 - 2r_i), \text{ with } \alpha > 0. \tag{2.10}$$

α is the offset magnitude that we are free to choose. See chapter 3 for an analysis on choosing a suitable offset magnitude. On the D-Wave quantum annealer, each qubit has a minimum offset $\delta_i^{\min}$ and a maximum offset $\delta_i^{\max}$ that can be applied to it. To ensure that the applied offset is within that range, the following strategy is used:

$$\delta_i(r_i) := \begin{cases} \max\{\alpha(1 - 2r_i), \delta_i^{\min}\} & \text{if } r_i \geq \frac{1}{2}, \\ \min\{\alpha(1 - 2r_i), \delta_i^{\max}\} & \text{if } r_i < \frac{1}{2} \end{cases} \tag{2.11}$$

In conclusion, the intention behind this strategy is not only to delay highly connected qubits, but to advance also qubits that have smaller effective fields. In chapter 3, we compare the success frequencies with and without applying individual offsets (see also Appendix B.2 for the implementation of this strategy).

Chapter 3

Results

In this chapter, we present results of solving the factoring problem on quantum annealers. Since quantum annealers are sampling devices, they produce a distribution of different samples for each problem. We measure success by the percentage of successful samples (success frequency).

All results for the multiplication circuit method and the modified multiplication table method have been obtained on the D-Wave Advantage System 5.4, which is located at Jülich Supercomputing Centre, Forschungszentrum Jülich. Results for the controlled full-adder method have been obtained on the D-Wave Advantage System 4.1 (located in Canada). The reason for this is that we have not been able to implement multipliers of sufficient sizes on the 5.4 chip due to the higher number of broken couplers and chains (see section 2.3.2). Comparability of results is still given because both QPUs are of the same model (Advantage, performance update). They only differ slightly in the physical properties of the calibrated QPU.

Before comparing the success frequencies of the three examined factoring methods, some preliminary observations are necessary. These include:

- assessing two different approaches for setting the Lagrange parameter to reduce higher order terms
- finding an optimal block size for the modified multiplication table method
- choosing a suitable offset magnitude for the CFA method

At the end of this chapter, we compare the scaling of success frequencies with problem size for the different methods. In all experiments that we perform we assume knowledge of the factors' lengths l_p and l_q and each approach is initialized accordingly.

3.1 Reducing Higher Order Terms

The quantum annealing formulation of the factoring problem introduced in section 2.2 requires terms of higher order than quadratic to be reduced to quadratic and linear terms (see Equation (2.3)).

To reduce higher order terms of the form $x_0x_1x_2$ to quadratic, Jiang et al. introduce an auxiliary variable x_3 and add a constraint to ensure $x_3 = x_0x_1$ (second addend in Equation (2.3)). If the constraint is not weighted appropriately in the final objective function, it can happen that it is ignored by the lowest-energy samples, as there are other terms with much higher coefficients that the model will try to satisfy first. This can lead to models where the ground state has a negative corresponding energy and does not represent the solution. Therefore, the constraint term needs to be multiplied with a penalty multiplier, also known as Lagrange parameter. We consider two approaches for choosing the Lagrange parameter:

1. Setting a fixed Lagrange parameter, so that all constraints are weighted equally:

$$\alpha x_0x_1x_2 = \alpha x_2x_3 + \lambda(3x_3 + x_0x_1 - 2x_0x_3 - 2x_1x_3) \text{ if } x_3 = x_0x_1, \text{ with fixed } \lambda \geq 1 \tag{3.1}$$

2. Multiplying the constraint with the term's coefficient for a dynamic Lagrange parameter

$$\alpha x_0x_1x_2 = \alpha x_2x_3 + |\alpha| (3x_3 + x_0x_1 - 2x_0x_3 - 2x_1x_3) \text{ if } x_3 = x_0x_1 \tag{3.2}$$

The first approach has the advantage of keeping the range of the linear terms of the BQM low by choosing a small penalty multiplier. However, optimizing the Lagrange parameters separately such that they are as small as possible is a separate problem and quickly becomes infeasible for larger models. Using a dynamic penalty multiplier, we can proportionally penalize constraint violations based on the coefficient of the corresponding term.

In Table 3.1, we present results obtained by numerically enumerating all possible input variable combinations of some example models with fixed and dynamic Lagrange multiplier. Contrary to quantum annealing, which is a sampling method and does not necessarily return the global minimum of the cost function, numerical enumeration on classical QPUs yields the corresponding energy to all possible input combinations. Because of memory and computing constraints the numerical enumeration is limited to small BQMs. In the examples tested, we observe that the dynamic approach immediately generates a BQM where the lowest possible energy is 0 and the corresponding bit string represents the solution to the factoring problem. The dynamic approach eliminates the need for heuristically determining the Lagrange parameter. Therefore, all results presented in the following sections are based on dynamic penalty multipliers.

One strategy to further optimize the reduction process is to combine both approaches by scaling down the coefficient of the three body term by a penalty multiplier $\lambda \in (0, 1)$:

$$\alpha x_0x_1x_2 = \alpha x_2x_3 + \lambda |\alpha| (3x_3 + x_0x_1 - 2x_0x_3 - 2x_1x_3) \text{ if } x_3 = x_0x_1, \text{ with fixed } \lambda \in (0, 1)$$

N	λ	min_energy	solution_valid
493	1	-199	False
493	2	-147	False
493	3	-103	False
493	4	-67	False
493	5	-51	False
493	6	-35	False
493	7	-19	False
493	8	-10	False
493	9	-2	False
493	10	0	True

(a) Fixed Lagrange parameter for $N = 493$. $\lambda \geq 10$ is the correct penalty multiplier.

N	min_energy	solution_valid
91	0	True
143	0	True
437	0	True
493	0	True

(b) Dynamic Lagrange parameter. The model is correct for all tested semiprimes without having to determine a suitable Lagrange parameter first.

Table 3.1: Comparison of fixed and dynamic Lagrange parameter. The column min_energy shows the minimum possible energy in each model generated with the modified multiplication method. The last column indicates whether the bit string of the minimal solution represents the factors p, q and N so that $N = p \times q$. We determine the values by numerically enumerating all possible variable combinations of the BQM with the xubo solver [15].

N	λ	min_energy	solution_valid
91	0.2	0	True
143	0.4	0	True
437	0.4	0	True
493	0.4	0	True

Table 3.2: Combination of fixed and dynamic penalty multipliers for four semiprimes N. The column λ displays the lowest examined penalty multiplier where the cost function's global minimum represents the solution to the factoring problem. Results obtained by numerically enumerating all possible variable combinations of the BQM with the xubo solver [15].

This has the effect of reducing the range of linear terms in the BQM. Preliminary experiments for four semiprimes N show that scaling down the coefficient α yields a cost function that fulfills the conditions defined in subsection 2.2.1. In Table 3.2, we numerically enumerate all models using the combined approach for four semiprimes with $\lambda \in [0.1, \ldots, 0.9]$. We observe that using the combined approach, the coefficient α can be scaled down by a factor λ of $0.2 - 0.4$. A possible hypothesis is that, as 3 is the largest coefficient in the cost function for the `and` gate (see Equation (2.3)), a penalty multiplier of ≈ 0.33 is a suitable choice. Further experiments are necessary to determine if this is also the case for larger semiprimes.

We proceed with reducing three-body terms by using dynamic penalty multipliers without applying an additional fixed λ (Equation (3.2)).

3.2 Modified Multiplication Table Method

Before solving factoring problems with the modified multiplication table method, a suitable block size needs to be established based on problem size. In the first subsection, we assess the impact of different block sizes on the BQM's parameters and on the success frequencies. The second subsection provides an analysis of the energy landscape of obtained samples. All calculations in this section have been performed on the D-Wave Advantage 5.1.

3.2.1 Optimal block size

The range of the coefficients of linear terms on the QPU used is between $[-4, 4]$. All BQMs where terms exceed that range will be rescaled accordingly. Jiang et al. argue that "reduc[ing] the range of Ising parameter values [reduces] the bits of precision required by control hardware" [12, p. 3]. In Figure 3.1, we can see that the BQM's highest absolute linear coefficient grows exponentially with block size before it saturates for the largest block sizes. The highest absolute coefficient is relevant because it determines the scaling factor for all coefficients. To obtain the cost function for each block, we multiply each block's column with a factor 2^{block_idx}, where $block\_idx$ is the index of the column in the block. Incrementing the block size, therefore, also increases the exponent of that term, leading to an exponential growth of the highest absolute linear coefficient with block size.

Increasing the block size to the maximum, which is the binary length of N, has the effect of removing all carry variables. This is because the cost function is obtained directly from the equation $(N - pq)^2 = 0$, after substituting the binary representations of prime numbers p and q (*direct method* [12, p. 2]). We observe the biggest decrease in the number of linear terms with smaller block sizes. This is because with larger block sizes the number of carry variables no longer changes significantly.

The number of linear coefficients, and therefore the overall complexity of the model, decreases most from block size 1 (no blocks) to block size 2. At the same time, the scaling factor for the linear terms is still comparably low at block size 2. Therefore, we would expect the success rates to be largest in the lower half, possibly around block size $2-3$ for the example models. To test this hypothesis, we compare the success rates of all block sizes for different semiprimes. We find that in almost all cases the success rates for the direct method are up to eight times higher compared to the best performing block size (Figure 3.2). We hypothesize that the negative impact of the rescaling is balanced out by the lower number of qubits needed for the direct method. In conclusion, we observe that the direct method consistently outperforms the block approach. Consequently, in subsequent sections, all models based on the modified multiplication table method are constructed with maximum block size (direct method).

3.2.2 Analyzing the energy landscape

A constraint violation can either be an incorrect binary place of p or q, or a violation of an equality constraint in the reduction of higher order terms to quadratic terms $t_{ij} = p_i q_j$ (see Equation (2.3)).

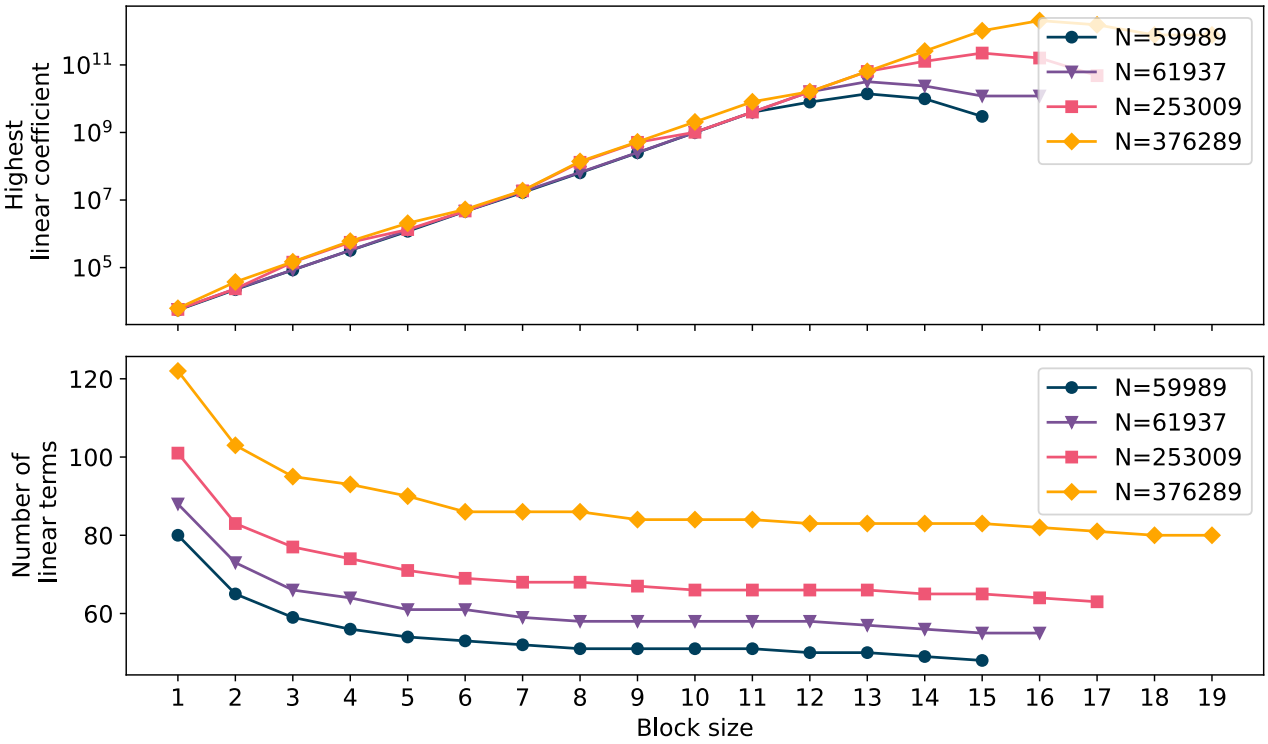


Figure 3.1: Visualization of the highest absolute linear coefficient and of the number of linear terms for different N with increasing block size.

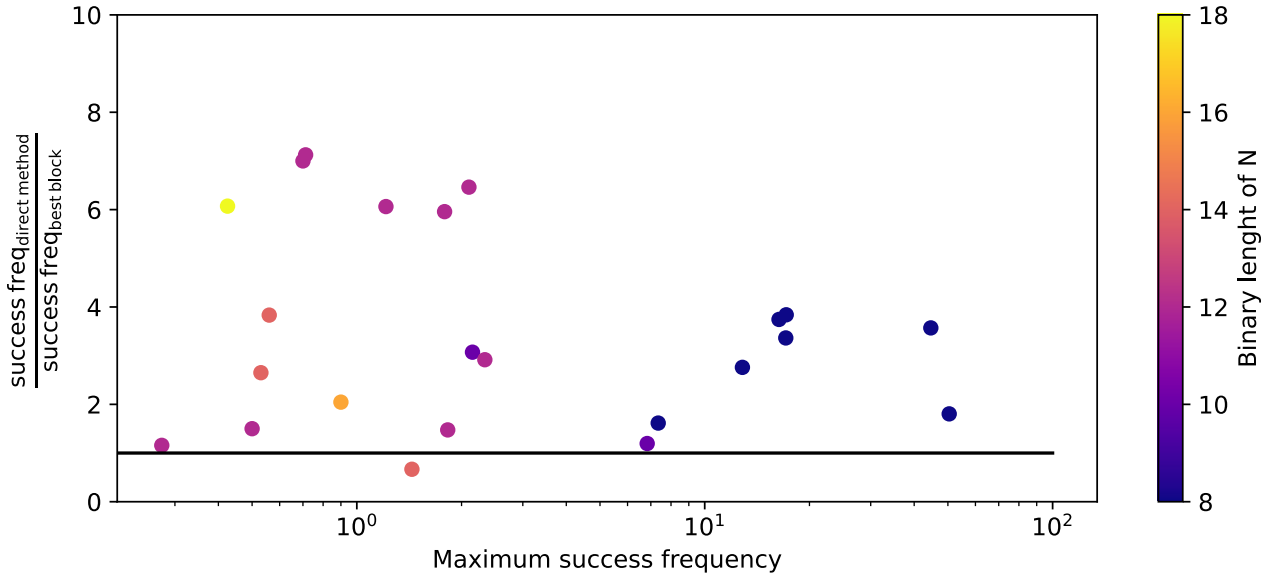


Figure 3.2: Comparison of the performance between the direct method and the modified multiplication table method. On the x-axis the highest obtained success rate for any block is displayed. The y-axis shows the ratio of the direct method's success rate to the success rate of the best-performing block.

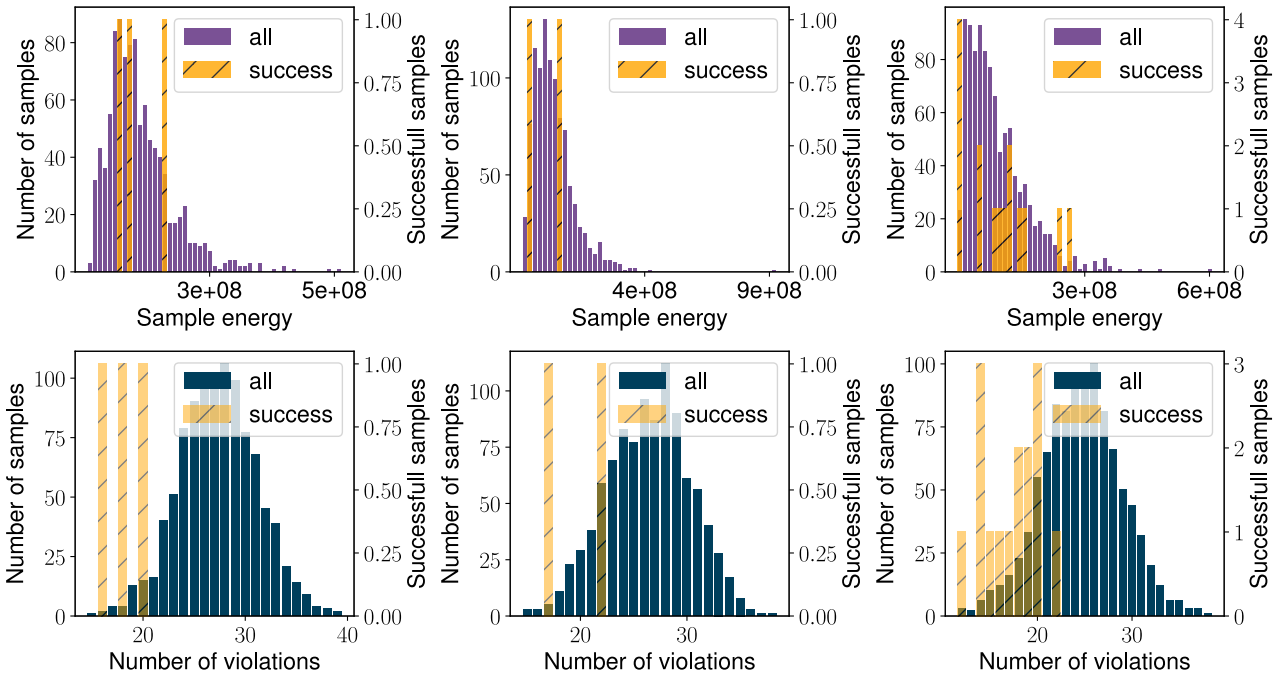


Figure 3.3: Comparison of histograms of three samples for $N = 503 * 503 = 253009$. Each histogram represents one sample with num_reads=*1000*. The first row contains histograms of the energy landscape for each sample. The second row is a histogram of the number of violations. The yellow bars represent the number of correct samples (right y-axis).

In Figure 3.5 we analyze the energy landscape of samples for $N = 503 \times 503 = 253009$. The figure's second row shows that successful samples tend to violate fewer constraints. While analyzing the energy landscapes further we observe that the correct samples have a comparably high energy and that they are distributed over the entire energy landscape. These violated constraints must, therefore, have a high penalty multiplier λ, leading to a high total energy. A hypothesis is that samples with a low number of violated constraints are sampled more frequently. Since these samples are more likely to represent the solution, we tend to sample the solution more often. However, further research is needed to validate this hypothesis.

3.3 Tuning with Individual Per-qubit Offsets

In section 2.3.2, we presented an approach by Adame et al. that aims to reduce early freeze-out of highly connected qubits by delaying their anneal schedule. In this section we describe an approach to determine a suitable offset magnitude for the anneal offset feature. All results in this section have been obtained by using the CFA method, as we have not implemented the anneal offset feature for the modified multiplication table and the multiplication circuit method.

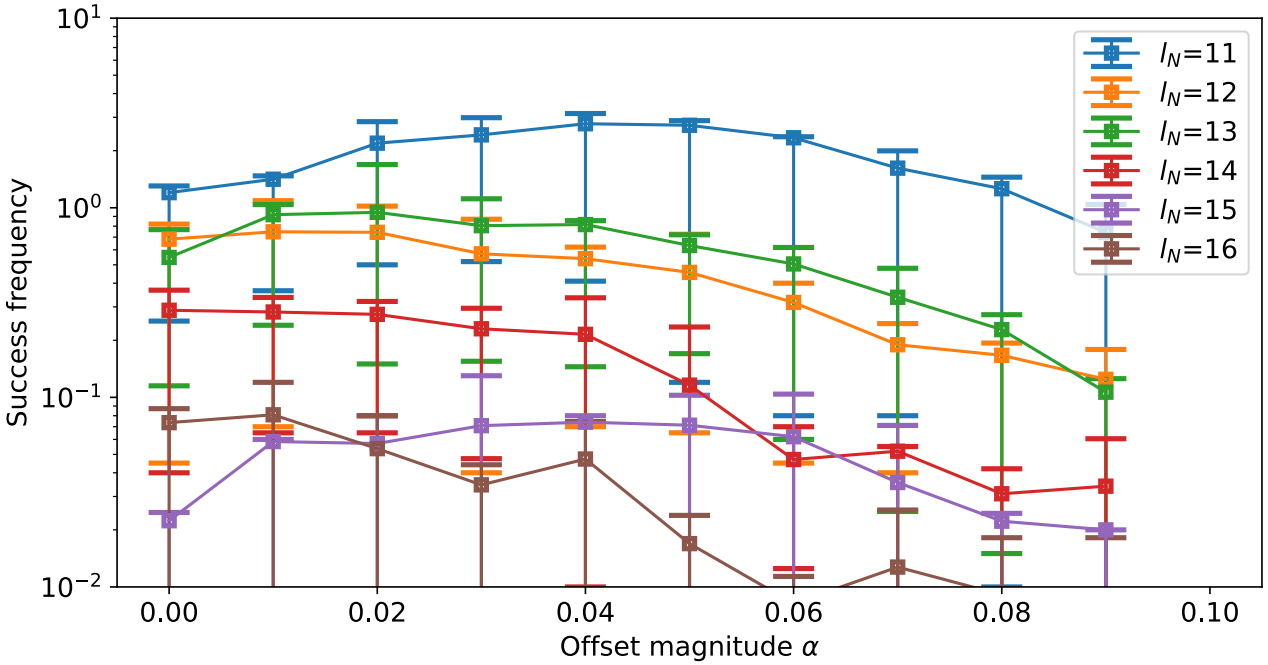


Figure 3.4: Success frequencies for semiprimes of different lengths l_N factored with the CFA method with ascending offset magnitude α. The error bars mark the 25% and 75% quantiles. For each l_N and each $\alpha \in [0, \ldots, 0.9]$, we sampled 5000 times across 10 semiprimes.

3.3.1 Choosing a suitable offset magnitude

The first step when using individual qubit offsets is to choose a suitable offset magnitude. The offset magnitude determines how much the size of the effective field delays or advances a qubit's anneal schedule. We applied offsets to problem instances of semiprimes with different lengths l_N (Figure 3.4) for offset magnitudes $\alpha \in [0, \ldots, 0.9]$. The curve's characteristic hill shape for $l_N = 11$ or $l_N = 15$ is similar to the observations made by Adame et al. for other problems, such as alternating sector chain problems [5, p. 9]. However, there are also problem sizes where applying offsets has a negative impact on success frequencies (for example $l_N = 14$). Due to the high fluctuations in success frequencies, we cannot draw any definite conclusions.

We determine a suitable offset magnitude α that can be applied to all problem classes. We weight each offset magnitude with its observed probability and calculate the average

$$\alpha_{l_N} = \frac{\sum_{\alpha_i=0,0.01,\ldots,0.09} p^{mean}_{success\ l_N}(\alpha_i) * \alpha_i}{\sum_{\alpha_i=0,0.01,\ldots,0.09} p^{mean}_{success\ l_N}(\alpha_i)}. \tag{3.3}$$

We then take the average of α_{l_N} over all l_N to calculate a general offset magnitude which is suitable for all problem classes. Another possibility is to not use a single average $\overline{\alpha}$ for all l_N, instead of assigning each problem class a separate α_{l_N}. However, with the big fluctuations in success frequencies that we observe, we cannot conclude that l_N influences α_{l_N}. We therefore choose to proceed with an average $\overline{\alpha} \approx 0.03$ over all α_{l_N}.

3.4 Comparison of Methods

We provide data in this section that compares all three factoring methods described in chapter 2. We assess the methods' success frequencies and the QPU's ability of finding the ground state using these methods.

Terminology: A run is defined as a collection of 1000 subsequent samples.

3.4.1 Comparability of methods

To ensure comparability of these three methods, we define the problem size as the number of unknown binary places in p and q: $l_p^* + l_q^*$. This allows for the comparison of the multiplication circuit method with the direct method and the CFA method, which have a different number of unknown variables. The direct method is designed in a way that the LSB and MSB of p and q are set to 1 in advance. Therefore, $l_p^* = l_p - 2$ and $l_q^* = l_q - 2$. For the other two methods $l_p^* = l_p$ and $l_q^* = l_q$.

The probability (in percentage) for randomly guessing all of the factor's bits of a semiprime N correctly is

$$P(N) = \frac{1}{2^{l_p^* + l_q^*}} * 100 \tag{3.4}$$

3.4.2 Conclusion

It is difficult to draw definite conclusions, as the high number of problems, to which no solution was found, increases the fluctuations in the data. Overall, the CFA method outperforms both the direct method and the multiplication circuit method. With this approach, factoring problem of size 21 are still possible, and solutions to factoring problems of this size can be found in around 10% of runs.

As can be seen in the second subfigure of Figure 3.5, samples obtained with the CFA method contain the ground state for bigger problem instances of size 18. Samples of BQMs generated with the multiplication circuit method and the direct method fail to find the ground state for small problem instances of sizes ≥ 11 and ≥ 12 respectively. In the analysis of energy landscapes in subsection 3.2.2, we have seen that solutions for the factoring problem obtained with the direct method are distributed over the entire energy landscape and that these solutions, therefore, do not correspond to the global minimum of the cost function. In contrast, the quantum annealer is able to *solve* Ising problems generated with the CFA method for larger problem instances. A possible hypothesis is that the regular structure of the CFA method's custom embedding positively influences the quantum annealer's ability of finding the ground state. On the one hand, the ability of finding the ground state indicates a problem formulation that is well suited for the QPU. On the other, it could be argued that the Ising problem does not need to be fully solved, as long as the resulting bit string at the end of the annealing process corresponds to the bits of the factors p and q. Both arguments are valid and further research is necessary to determine how these methods behave on D-Wave's next-generation QPUs based on the Zephyr topology. The Zephyr topology allows for a higher degree of qubit connectivity compared to

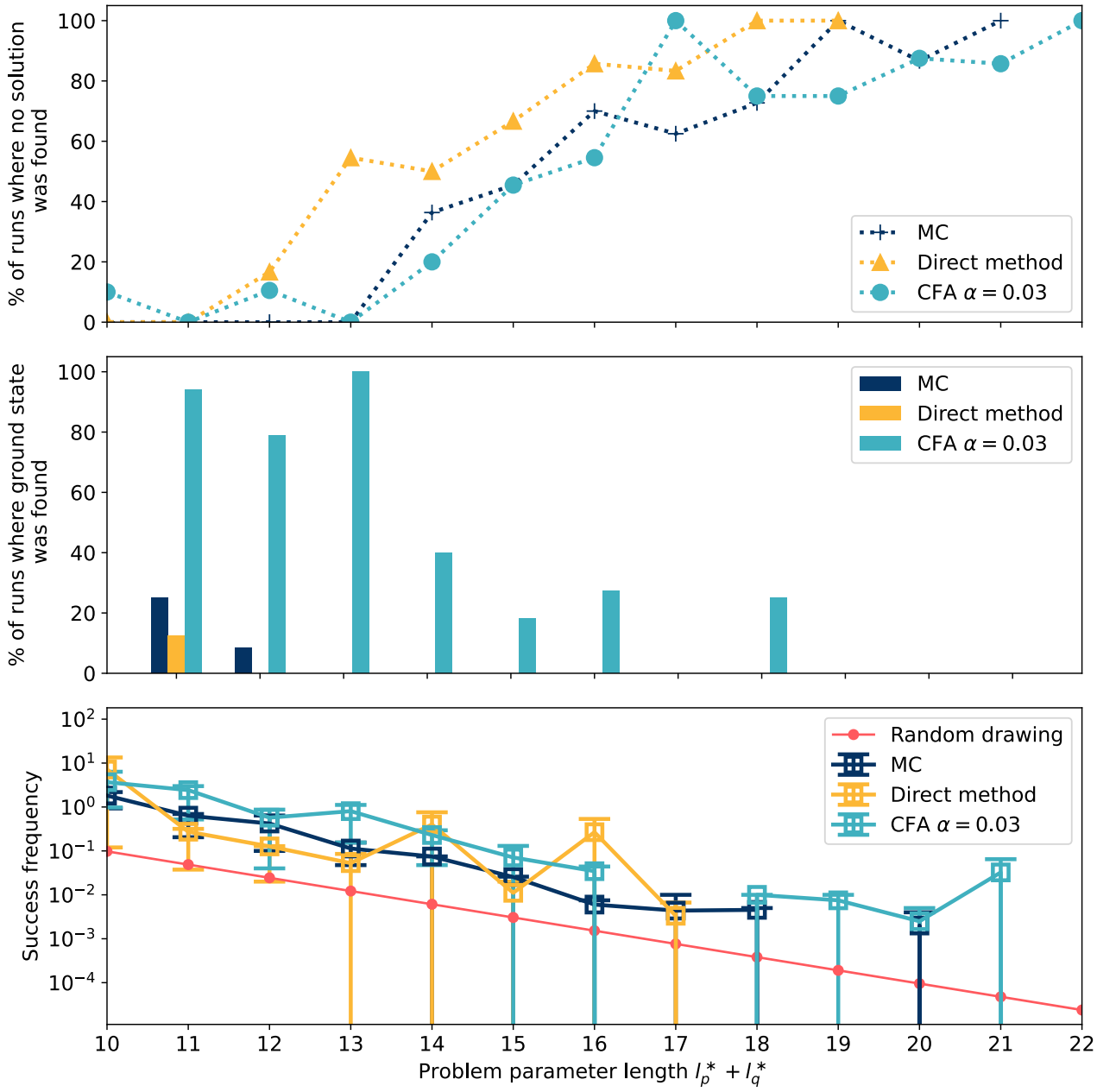


Figure 3.5: Overview over problems instances, to which no solution was found, ground states, and success frequencies for the three examined methods over problem sizes in range $[10, 22]$. The first subfigure shows the percentage of runs where no solution was found. The second subfigure displays the percentage of runs where at least one sample represents the ground state (minimum energy). The last subfigure show the success frequencies and the probability of randomly drawing the correct solution (red line). The error bars mark the 25% and 75% quantiles. For each problem size, we sampled 10 semiprimes linearly spaced out over the problem interval. The number of runs for each semiprime is 5 (or 5000 samples).

the current Pegasus topology (an average degree of connectivity of 20 compared to 15 currently).

It is difficult to draw conclusions related to scaling for problem sizes ≥ 13 because of the high fluctuations of the data. As quantum annealing is a sampling method the possibility exists that we are randomly sampling the factors p and q. Figure 3.5 shows an evaluation of the scaling of the success frequencies in comparison to random sampling (red dots). As we can see, the observed success frequencies for all methods are consistently higher than random drawing. Interestingly, even though the quantum annealer is not able to find the ground state of the Ising model in many cases, it still returns the corresponding bit strings of p and q with higher-than-random frequency.

Generally, the data currently suggests a scaling of the success frequencies with l_N similar to random drawing. The exponential decrease in success frequencies is observed for all methods. Moreover, all three methods perform only slightly better than randomly drawing the factor's bits. The CFA method still shows the best success frequencies which are about 100 times higher than those of random drawing.

Chapter 4

Conclusion

The goal of this work was to implement and assess state-of-the art factoring algorithms for quantum annealers, such as the multiplication circuit method, the modified multiplication method and the CFA method. We examined different approach for reducing higher order terms so that they can be mapped to the Ising model and implemented individual per-qubit offsets as a tuning strategy.
To reduce terms of higher order than quadratic, we formulated an approach that takes into account the coefficient of the term to be reduced, and we showed experimentally that it produces valid models for smaller problem sizes.
We also examined the growth of the linear coefficients with increasing block size for the modified multiplication table method. Our data indicates no benefit of using the block approach and we find that models constructed without blocks (direct method) consistently yield higher success frequencies.
As a possible tuning approach, we assessed individual per-qubit offsets by applying them to the CFA method. Our findings are inconclusive, and we cannot yet say that applying offsets improves the model's success frequencies. Some problem sizes perform better with anneal offsets, and for others, applying offsets impairs the performance.
We find that none of the three approaches examined is able to consistently solve factoring problems for larger semiprimes. The success frequencies exhibit a scaling that is qualitatively similar to random drawing, with success frequencies that are on average only $10 - 100$ times higher than if we were to randomly draw each bit of p and q. However, the CFA method's ability to produce ground states even for larger problem sizes is promising and suggests that approaches with custom embedding might be the preferred solution in the future.

4.1 Outlook

A major drawback of the CFA method compared to both other methods is that its custom embedding requires a fully intact grid of unit cells. D-Wave's next-generation topology, the Zephyr topology, implements greater qubit connectivity. It is an area of further research to determine if it is possible to adapt the CFA encoding in a way that broken couplers can be circumvented. This would allow for

the implementation of larger multipliers.

Another interesting aspect is to further improve the reduction method for higher order terms. We presented two approaches with fixed and dynamic multipliers. Further research is necessary to assess if a combination of both methods can consistently lead to valid models with lower coefficients compared to just using the dynamic approach.

Finally, the strategy that we employed already assumes some knowledge of the solution, by fixing the factor's length. Establishing a framework to solve problems without prior knowledge is necessary to factor cryptographically significant semiprimes on future QA devices.

Appendix A

QUBO Formulation for AND Gate

In this section we demonstrate how to find a QUBO formulation for an AND gate. The general QUBO formulation is $E(x) = \sum_{i \leq j} x_i Q_{i,j} x_j$ with $x_i \in \{0, 1\}$. For better readability, we denote the linear coefficients $Q_{i,i}$ as a_i and the quadratic coefficients $Q_{i,j}$ as b_{ij}. For three input variables, the QUBO model becomes

$$E(x_0, x_1, x_2) = a_0 x_0 + a_1 x_1 + a_2 x_2 + b_{01} x_0 x_1 + b_{02} x_0 x_2 + b_{12} x_1 x_2 \tag{A.1}$$

It is not necessary to add a constant offset c to the function to ensure that its global minimum is 0, as $E(0, 0, 0) = 0$. For an AND gate with $x_2 = x_0 x_1$ we obtain the following linear system of equations. We set the output of the function $E(x_0, x_1, x_2)$ to zero, if the condition $x_2 = x_0 x_1$ is fulfilled and to > 0 otherwise:

$$\begin{aligned}
&E(0,0,1) = a_2 \overset{!}{>} 0 \\
&E(0,1,0) = a_1 \overset{!}{=} 0 && \Rightarrow a_1 = 0 \\
&E(0,1,1) = a_1 + a_2 + b_{12} = a_2 + b_{12} \overset{!}{>} 0 \\
&E(1,0,0) = a_0 \overset{!}{=} 0 && \Rightarrow a_0 = 0 \\
&E(1,0,1) = a_0 + a_2 + b_{02} = a_2 + b_{02} \overset{!}{>} 0 \\
&E(1,1,0) = a_0 + a_1 + b_{01} = b_{01} \overset{!}{>} 0 \\
&E(1,1,1) = a_0 + a_1 + a_2 + b_{01} + b_{02} + b_{12} = a_2 + b_{01} + b_{02} + b_{12} \overset{!}{=} 0
\end{aligned}$$

A first approach to construct a BQM for the AND gate is to assign the same positive energy z to all cases where $x_2 \neq x_0 x_1$. However, it is not possible to penalize all four cases $x_2 \neq x_0 x_1$ equally with

x0	x1	x2	energy
0	0	0	0
1	0	0	0
0	1	0	0
1	1	1	0
1	1	0	1
0	1	1	1
1	0	1	1
0	0	1	3

Table A.1: Energy table generated for a BQM where $x_0 \wedge x_1 = x_2$.

$z > 0$:

$$
\begin{aligned}
E(0,0,1) &= a_2 \overset{!}{=} z && \Rightarrow a_2 = z \\
E(0,1,1) &= a_2 + b_{12} = z + b_{12} \overset{!}{=} z && \Rightarrow b_{12} = 0 \\
E(1,0,1) &= a_2 + b_{02} = z + b_{02} \overset{!}{=} z && \Rightarrow b_{02} = 0 \\
E(1,1,0) &= b_{01} \overset{!}{=} z && \Rightarrow b_{01} = z \\
E(1,1,1) &= a_2 + b_{01} + b_{02} + b_{12} = z + z = 2z \overset{!}{=} 0 && \lightning \text{ for } z > 0
\end{aligned}
$$

Therefore, a common choice is to set $E(0,0,1) = 3$ and the three other cases where $x_2 \neq x_0 x_1$ to one, as can be seen in the energy table for the D-Wave `dimod.generators.and_gate()` A.1. Solving the system of linear equations with the function values from Table A.1, gives us the following cost function:

$$E(x_0, x_1, x_2) = 3x_2 + x_0 x_1 - 2x_0 x_2 - 2x_1 x_2 \tag{A.2}$$

Appendix B

Code Examples

B.1 Solving $N = 91$ with D-Wave's `multiplication_circuit()`

The method `multiplication_circuit()` is part of the D-Wave SDK package. It is included in the `dimod.generators` package, which provides various methods for generating BQMs.

In the following example, we generate a 3×4-bit multiplication circuit to factor $N = 91$. This package uses the variable name `p` for the number to be factored and `a` and `b` for the factors.

```
from dimod.generators import multiplication_circuit
# here we generate a multiplication circuit of size 3,4
bqm = multiplication_circuit(3,4)

# now we need to fix the variables
N = bin(91)
fix_dict = {'p'+str(i): int(N_i) for i, N_i in enumerate(N[2:][::-1])}
bqm.fix_variables(fix_dict)

# initialize solver
sampler = DWaveSampler(region='na-west-1')
solver = EmbeddingComposite(sampler)

# solve the bqm on the D-Wave and retrieve lowest energy sample
result = solver.sample(bqm, num_reads =1000)
result.to_pandas_dataframe() \
    .sort_values('energy')[['a0','a1','a2','b0','b1','b2','b3']].head(1)
```

	a0	a1	a2	b0	b1	b2	b3	energy	num_occurrences
0	1	1	1	1	0	1	1	0.0	191

Table B.1: Lowest energy sample after solving the BQM for $N = 91$ on the D-Wave quantum annealer with 1000 reads. The ground state with the correct solution $a = 7$ and $b = 13$ was measured 191 times in this example.

B.2 Effective Field Calculation

One strategy to calculate individual per-qubit offsets is based on the qubit's connectivity, the so-called effective field (Adame et al. [5], see section 2.3.2). In this section, we show the implementation of this strategy: As a first step, we calculate the average effective field $\overline{F_i}$ (Equation (2.8)) for qubit i by looping over all possible neighbor spin configurations `s`. The input value `h` is the linear bias of qubit i and `adj` is a dictionary of its neighboring nodes with the quadratic biases (`J_terms`).

```
def calc_field_average(self, h, adj):
    result = 0
    Ni = len(adj)
    for s in range(0,2**Ni):
        result += self.calc_abs_effective_field(h,adj,s)
    return result / 2** Ni
```

The following function calculates the absolute effective field $F_i(s_{j_1}, \ldots, s_{j_{Ni}})$ (Equation (2.7)):

```
def calc_abs_effective_field(self, h, adj, s):
    result = 0
    b_format = f'0{len(adj)}b'
    Js = list(adj.values()) # connection strength to neighbors

    # convert s to binary
    config = list(f'{s:{b_format}}')
    # 0 values need to be substituted by -1
    config = list(map(lambda el: int(el) if el == '1' else -1,config))
    # multiply each spin with the corresponding J-value
    result += np.dot(config,Js)
    return abs(result +h)
```

Then, we assign the respective average effective field to each qubit `q`:

```
average_fields = {q: self.calc_field_average(h,self.bqm.adj[q])
                                for q, h in self.bqm.linear.items()}
```

The normalized effective field `rs` (Equation (2.9)) is calculated by determining `max(`$\overline{F_i}$`)` and dividing each qubit's average effective field by it:

```
max_Fi = max(average_fields.values())
rs = {q: Fi / max_Fi for q, Fi in average_fields.items()}
```

We cannot just assign an offset to the qubits in our model, but need to specify it for every qubit on the QPU. We also need to ensure that all our offsets are in the qubit's valid offset range (Equation (2.11)). `alpha` is the offset magnitude that we determine beforehand.

```
deltas = [0]*qpu.properties['num_qubits']
for q, ri in rs.items():
        delta_i_min, delta_i_max = qpu.properties['anneal_offset_ranges'][q]
        if ri >= 0.5:
                deltas[q] = max(self.alpha * (1-2*ri), delta_i_min)
        else:
                deltas[q] = min(self.alpha * (1-2*ri), delta_i_max)
params['anneal_offsets'] = deltas
```

Now the array `params['anneal_offsets']` contains an offset value for every qubit.

Bibliography

[1] D.M. Bressoud. *Factorization and Primality Testing*. Undergraduate Texts in Mathematics. Springer New York, 2012.

[2] Andrew Lucas. Ising formulations of many np problems. *Front. Phys.*, 2:5, 2014.

[3] Dorit Aharonov, Wim van Dam, Julia Kempe, Zeph Landau, Seth Lloyd, and Oded Regev. Adiabatic quantum computation is equivalent to standard quantum computation. *SIAM J. Comput.*, 37(1):166–194, 2007.

[4] Takashi Imoto, Yuki Susa, Ryoji Miyazaki, Tadashi Kadowaki, and Yuichiro Matsuzaki. Universal quantum computation using quantum annealing with the transverse-field ising hamiltonian. arXiv:2402.19114 `https://arxiv.org/abs/2402.19114`, 2024.

[5] Juan I Adame and Peter L McMahon. Inhomogeneous driving in quantum annealers can result in orders-of-magnitude improvements in performance. *Quantum Sci. Technol.*, 5(3):035011, jun 2020.

[6] Dennis Willsch. Solving qubo problems, juniq documentation. `https://jugit.fz-juelich.de/qip/juniq-platform/juniq-documentation/`, 2024. Last accessed 24 June 2024.

[7] Tameem Albash and Daniel A. Lidar. Adiabatic quantum computation. *Rev. Mod. Phys.*, 90:015002, Jan 2018.

[8] Boudot et al. Factorization of rsa-250. `https://sympa.inria.fr/sympa/arc/cado-nfs/2020-02/msg00001.html`, 2020. Last accessed 20 June 2024.

[9] Enrique Martín-López, Anthony Laing, Thomas Lawson, Roberto Alvarez, Xiao-Qi Zhou, and Jeremy L. O'Brien. Experimental realization of shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6(11):773–776, October 2012.

[10] Dennis Willsch, Madita Willsch, Fengping Jin, Hans De Raedt, and Kristel Michielsen. Large-scale simulation of shor's quantum factoring algorithm. *Mathematics*, 11(19):4222, October 2023.

[11] Jingwen Ding, Giuseppe Spallitta, and Roberto Sebastiani. Effective prime factorization via quantum annealing by modular locally-structured embedding. arXiv:2310.17574 `https://arxiv.org/abs/2310.17574`, 2023.

[12] Shuxian Jiang, Keith A. Britt, Alexander J. McCaskey, Travis S. Humble, and Sabre Kais. Quantum annealing for prime factorization. arXiv:1804.02733 https://arxiv.org/abs/1804.02733, 2018.

[13] D-Wave Systems. Documentation csp factory for multiplication circuit. https://docs.ocean.dwavesys.com/en/stable/docs_dimod/reference/generated/dimod.generators.multiplication_circuit.html, 2024. Last accessed 24 May 2024.

[14] D-Wave Systems. Documentation generators and application modeling. https://docs.ocean.dwavesys.com/en/stable/docs_dimod/reference/generators.html#constraints, 2024. Last accessed 15 July 2024.

[15] Dennis Willsch. Gpu solver for quadratic/polynomial/higher-order unconstrained binary optimization problems. https://jugit.fz-juelich.de/qip/xubo, 2024. Last accessed 12 June 2024.

[16] Philipp Hanussek. Code repository. https://jugit.fz-juelich.de/qip/jupsifactoring.

[17] Jingwen Ding, Giuseppe Spallitta, and Roberto Sebastiani. Git repository to: Effective prime factorization via quantum annealing by modular locally-structured embedding. https://gitlab.com/jingwen.ding/multiplier-encoder/. Last accessed 24 June 2024.

[18] D-Wave Systems. Qpu physical properties. https://docs.dwavesys.com/docs/latest/doc_physical_properties.html, 2024. Last accessed 3 July 2024.

[19] Ting-Jui Hsu, Fengping Jin, Christian Seidel, Florian Neukart, Hans De Raedt, and Kristel Michielsen. Quantum annealing with anneal path control: application to 2-sat problems with known energy landscapes. arXiv:1810.00194 https://arxiv.org/abs/1810.00194, 2018.

[20] Evgeny Andriyasha, Zhengbing Bian, Fabian Chudak, Marshall Drew-Brook, Andrew D. King, William G. Macready, and Aidan Roy. D-wave technical report: Boosting integer factoring performance via quantum annealing offsets. https://www.dwavesys.com/media/l0tjzis2/14-1002a_b_tr_boosting_integer_factorization_via_quantum_annealing_offsets.pdf/, 2016. Last accessed 24 June 2024.