

# ALOCAÇÃO DE QUBITS



MARCOS YUKIO SIRAICHI

## ALOCAÇÃO DE QUBITS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

COORIENTADOR: VINÍCIUS FERNANDES DOS SANTOS

Belo Horizonte

Fevereiro de 2019



MARCOS YUKIO SIRAICHI

## QUBIT ALLOCATION

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA  
CO-ADVISOR: VINÍCIUS FERNANDES DOS SANTOS

Belo Horizonte

February 2019

© 2019, Marcos Yukio Siraichi.  
Todos os direitos reservados.

**Ficha catalográfica elaborada pela Biblioteca do ICEx – UFMG**

Siraichi, Marcos Yukio

S619q Qubit Allocation / Marcos Yukio Siraichi. — Belo Horizonte, 2019  
xxii, 83 p. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação

Orientador: Fernando Magno Quintão Pereira  
Coorientador: Vinícius Fernandes dos Santos

1. Computação – Teses. 2. Alocação de Qubits.  
3. Computação Quântica. 4. Compiladores.  
5. Alocação de Registradores. I. Orientador.  
II. Coorientador. III. Título.

CDU 519.6\*14 (043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

## FOLHA DE APROVAÇÃO

Qubit Allocation

**MARCOS YUKIO SIRAICHI**

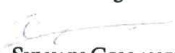
Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador  
Departamento de Ciência da Computação - UFMG

  
PROF. VINÍCIUS FERNANDES DOS SANTOS - Coorientador  
Departamento de Ciência da Computação - UFMG

  
PROF. SEBASTIÁN ALBERTO URRUTIA  
Departamento de Ciência da Computação - UFMG

  
PROFA. JULIANA KAIZER VIZZOTTO  
Centro de Tecnologia - UFSM

  
DR. SYLVAIN COLLANGE  
Centro de pesquisa Rennes - Bretagne Atlantique - INRIA

Belo Horizonte, 14 de fevereiro de 2019.





# Acknowledgments

This dissertation would certainly not be possible had Sylvain Collange, from Inria, not suggested the topic. It also would not be possible without the guidance, motivation, and opportunity provided by my adviser, Fernando Magno Quintão Pereira. I would also like to thank my co-adviser, Vinícius Fernandes dos Santos, who helped me, and came up with the base idea of part of this work.

I must also thank all of those who supported me on this two-year journey, and even before that. With no doubt, my family played a huge part in it. Without them, I would not have been able to enjoy and go through these two years. My fellow labmates also helped me whenever they could, and turned these past years into a memorable part of my life.

Finally, I would like to thank CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) for all the financial support given to me. Without it, not only this dissertation, but every experience would not have been possible.



*“ (...) sceptics won't accept that there's an invisible fire-breathing dragon in my garage because they're all atheistic materialists. ”*

*( Carl Sagan )*



# Abstract

In May of 2016, IBM Research has made a quantum processor available in the cloud to the general public. The possibility of programming an actual quantum device has elicited much enthusiasm. Yet, quantum programming still lacks the compiler support that modern programming languages enjoy today. To use universal quantum computers like IBM's, programmers must design low-level circuits. In particular, they must map logical qubits into physical qubits that need to obey connectivity constraints. This task resembles the early days of programming, in which software was built in machine languages. In this work, we shall formally introduce the qubit allocation problem and provide an exact solution to it. This optimal algorithm deals with the simple quantum machinery available today; however, it cannot scale up to the more complex architectures scheduled to appear. Thus, we will also provide heuristics to solve qubit allocation, one of which is faster, and another one that performs better than the current solutions already implemented to deal with this problem.

**Keywords:** Qubit Allocation, Quantum Computing, Compilers, Register Allocation.



# List of Figures

1.1	Circuit representation of $(a_1a_0 + b_1b_0)\%4$ . Each wire represents the state of a qubit as time passes by. The circuit is divided into three parts: (a) $a_1 \oplus b_1$ ; (b) $(a_0 \wedge b_0) \oplus b_1$ ; (c) $a_0 \oplus b_0$ . Output is stored in $b_1b_0$ . (d) shows the application of gates one in front of the other, and combination of them (parallel). . . . .	4
1.2	(a) The coupling graph of the IBM Yorktown computer. (b) Interactions between qubits of the circuit seen in Figure 1.1. (c) Dependences that have created these interactions. . . . .	7
1.3	(a) Reversal. (b) Bridge. (c) Swap. . . . .	8
1.4	(a) CNOT reversals, marked as grey boxes, invert the direction of $\text{CNOT}_{b_0b_1}$ . (b) The two black Hadamard gates can be simplified away, given the identity $HH = I$ . (c) Solution to qubit allocation showing embedding of the control graph onto the coupling graph. . . . .	9
1.5	Solution of qubit allocation for the circuit in Figure 1.1, using a CNOT swap. Grey lines represent physical qubits. We show the different mappings that we have at two points of the circuit. . . . .	9
1.6	(a) two undirected graphs $G$ and $H$ (left to right); (b) all possible subgraphs of $G$ that are isomorphic to $H$ . . . . .	10
1.7	(a) undirected graph of Figure 1.6 (a) with the initial state of the tokens ( $f_{src}$ ) inside the circle, the final state of the tokens ( $f_{tgt}$ ) outside the circle. Grey boxes indicate vertices whose tokens are already in its target place; (b) a sequence of swaps that solves the problem (highlighted edges). . . . .	11
1.8	Quantum circuit divided into a sequence of layers. Each column represents one layer. . . . .	12
1.9	Gate dependency graph of the quantum circuit in Figure 1.1. The vertices with no incoming edges (highlighted) may be executed. The vertices inside the dashed box do not have a correct order to be executed in. . . . .	13

2.1	Notation used in this work. . . . .	17
2.2	Eight states reachable from the initial mapping in the top-left side. In total, we have sixteen states. . . . .	19
2.3	Finding an initial mapping to qubit allocation. . . . .	21
2.4	(a) List of control dependences from the quantum circuit seen in Figure 1.1. (b) Weighted dependence graph. Grey boxes represent $w_p$ . (c-f) Step-by-step construction of the initial mapping. . . . .	22
2.5	Extending the initial mapping to satisfy $\Psi$ . . . . .	23
2.6	Steps of the <i>BMT</i> algorithm. (a) output of phase (i): partitioned program, and a set of mappings for each each of them; (b) output of phase (ii): one mapping for each program partition. . . . .	24
2.7	From left to right, we have the coupling graph $G_q^d$ , the list of control relations $\Psi$ , split into Maximal Isomorphic Sublists (solid boxes), and the graphs $\mathcal{G}_\Psi$ derived from sublists of $\Psi$ (dashed boxes). Next to each derived graph, we show if an isomorphism is possible ( $\checkmark$ ) or not ( $\times$ ). . . . .	25
2.8	Exhaustive mapping tree produced from the first instruction $(r_1, r_0)$ in our running example. The notation $p \cdot q$ indicates that pseudo qubit $p$ is mapped onto physical qubit $q$ . Mappings marked with $\times$ are dead-ends, i.e., we cannot continue the exhaustive construction of new mappings from them. We show the cost of each mapping next to it. . . . .	27
2.9	Tree of mappings for one partition. The $i^{th}$ level represents the possible mappings once we add the $i^{th}$ instruction (on the left) to the mappings already in place. Since the number of leaves grows exponentially, we prune by (a) limiting the number of children of each mapping; and (b) bounding the number of mappings for a partition. . . . .	28
2.10	Output from first phase <b>F</b> with some possible combinations represented by different paths. Each path in this figure represents one different solution. Highlighted, there is a path that represents the optimal solution. . . . .	29
2.11	Steps for transforming $f_{prev}$ into $f$ , assuming the coupling graph seen in Figure 2.7. The pseudo qubit assigned to a physical qubit is shown in brackets. Gray edges indicate qubits that shall be swapped. . . . .	30
2.12	Subproblem dependency for calculating <i>OPT</i> of the highlighted mapping. It shall be the minimum value of the sum of each dependency by its cost $\delta$ of transforming the previous one into the highlighted one. . . . .	33
2.13	Mappings $f$ and swapping sequences $\Delta$ (highlighted) gives us all the information that is necessary to transform a virtual quantum circuit into a physical quantum circuit. . . . .	34



2.14	(a) from left to right, we have the whole input program segmented into partitions (dashed box), and their respective mappings; (b) the translated input program segmented into partitions (dashed box), and the swap operations (highlighted) necessary to transform one mapping into another. . . . .	35
3.1	Coupling graph of the IBM Tokyo, a 20-qubit architecture [IBM, 2018a]. . . . .	40
3.2	Baseline is $M_p = 32$ and $M_c = 2560$ $\langle 32, 2560 \rangle$ . It shows the amount of partitions created by <i>BMT</i> in the first phase (Section 2.3.1) . . . . .	44
3.3	Baseline is the canonical version of the algorithm: <b>can</b> . In the Y-axis, we show how many partitions each allocator created in comparison to the baseline. In the X-axis, for clarity, we show the benchmarks that resulted in a difference of at least 5% among the allocators. . . . .	46
3.4	Baseline is the canonical version of the algorithm: <b>can</b> . In the Y-axis, we show the weighted cost of each allocator in comparison to the baseline. In the X-axis, for clarity, we show the benchmarks that resulted in a difference of at least 5% among the allocators. . . . .	47
3.5	Baseline is the canonical version of the algorithm: <b>can</b> . Ratios of the number of partitions created by each of these algorithms, in relation to the baseline. . . . .	47
3.6	Ratio of the weighted cost found by different allocators, in relation to the cost found by <b>bmtS</b> . The Y-axis shows the weighted cost in logarithmic scale. The X-axis shows benchmarks ordered in increasing order of the cost found by <b>bmtS</b> (shaded area). . . . .	48
3.7	Time spent by different allocators. The Y-axis shows time (seconds) in logarithmic scale. The X-axis shows benchmarks ordered in increasing order of the time spent by <b>bmtS</b> (shaded area). . . . .	50
3.8	Memory used by different allocators. The Y-axis shows the memory (bytes) in logarithmic scale. The X-axis shows benchmarks ordered in increasing order of the memory used by <b>bmtS</b> (shaded area). . . . .	51
3.9	Coupling graph of the IBM Yorktown, a 5-qubit architecture [IBM, 2018a]. . . . .	51
3.10	Template for the tables in this chapter. The major line for allocator <b>foo</b> indicates that $R_{comp}$ is $R_{foo}$ . In the caption we shall describe who is $R_{base}$ . Every column is the performance of $R_{foo}$ in comparison to $R_{base}$ . The highlighted lines indicate where $R_{foo}$ was better than $R_{base}$ . . . . .	53
3.11	Baseline is $M_p = 8$ and $M_c = 1280$ $\langle 8, 1280 \rangle$ . It shows the geometric mean ( $\mu_g$ ) of the metrics between all the combinations of parameters $M_c$ (lines) and $M_p$ (columns). The geometric standard deviation $\sigma_g$ (in parenthesis) shows the spread of the data. . . . .	54

3.12	Baseline is the canonical version of the algorithm: <b>can</b> . It shows the effects of the proposed optimizations relative to the canonical algorithm. Each allocator represents a new combination of improvements applied to the canonical version. . . . .	55
3.13	Baseline is the <i>BMT</i> ( <b>bmtS</b> ) allocator. Except for <b>Better Count</b> , the smaller the reported value, the better for the corresponding competing allocator. The rows (dimensions) where our <b>bmtS</b> loses are highlighted. . . .	56
3.14	Baseline is the optimal algorithm <b>dyn</b> allocator. Except for <b>Better Count</b> , the smaller the reported value, the better for the corresponding competing allocator. The rows (dimensions) where our <b>dyn</b> loses are highlighted. . . .	57
4.1	Quantum circuit indicating (in red) where the swaps would be ideally placed if allocated one layer at a time. . . . .	61
4.2	(a) creation of one layer of swaps $\{(a, c), (b, d)\}$ whose intersection is empty; (b) generation of one mapping for each edge, which represents the swap between its endpoints; (c) same as (b), but for all edges in $E(G_q^u) \setminus \{(q_0, *), (q_2, *)\}$ ; (d) creation of another layer of swaps, starting from the last mapping in the last layer of swaps; . . . . .	63
4.3	Illustration of two possible layer configurations, and the edges considered for creating the swap combinations. (a) shows gate $CNOT_{af}$ , where each $a$ and $f$ touch two edges each; yielding $2^4 = 16$ different swap combinations. (b) shows gates $CNOT_{ae}$ and $CNOT_{bf}$ , that touch every edge in the coupling graph; yielding $2^7 = 128$ different swap combinations. . . . .	64
4.4	Scheme of the whole algorithm proposed by Li et al. [2018]. The authors execute the allocation iteration five times, and get the best out of them. . .	66
4.5	(i)~(v) iterations of the execution of Miltzow et al. [2016] algorithm. On the left, there are the inputs to the algorithm: ( $T$ ) the token set $a, \dots, e$ ; ( $f_{src}$ ) the current configuration of tokens – inside the circles; ( $f_{tgt}$ ) the target configuration of tokens – outside the circles. On the right, there is the respective Swap Chain Graph to each iteration. Bold edges represent the swaps taken. . . . .	70
4.6	(a) initial token configuration on the left, and the bipartite graph $B$ on the right. Unused physical qubits and tokens $t$ that $ f_{tgt}(t)  > 1$ are represented as $\perp$ . (b) the modified initial token configuration on the left, where we replaced the two $\perp$ tokens for $\perp_1, \perp_2$ , so that they have only one target vertex. On the right, there is the resulting swap graph $S$ . . . . .	71

A.1 On the left, we have the same initial configuration as Figure 4.6.  $(T)$  is the token set  $\{a, c, e, \perp, \perp\}$ ;  $(f_{src})$  is the current configuration of tokens – inside the circles;  $(f_{tgt})$  is the target configuration of tokens – outside the circles. On the right, we have the corresponding swap graph  $S$  ignoring the  $\perp$  tokens. 82



# Contents

Acknowledgments	ix
Abstract	xiii
List of Figures	xv
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	3
1.1.1 Qubits and Quantum Gates. . . . .	3
1.1.2 Architectural Constraints. . . . .	5
1.1.3 Qubit Allocation – An Informal Overview. . . . .	6
1.1.4 Subgraph Isomorphism . . . . .	8
1.1.5 Token Swapping . . . . .	10
1.2 Quantum Programs Representation . . . . .	12
<b>2 The Qubit Allocation Problem</b>	<b>15</b>
2.1 Dynamic Programming . . . . .	17
2.2 Weighted Partial Mapping . . . . .	20
2.2.1 Finding the Initial Mapping . . . . .	20
2.2.2 Extending the Initial Mapping to handle $\Psi$ . . . . .	22
2.3 Bounded Mapping Tree . . . . .	23
2.3.1 Program Partitioning via Subgraph Isomorphisms . . . . .	24
2.3.2 Combining Mappings via Token Swapping . . . . .	28
2.3.3 Code Generation . . . . .	33
2.4 Improving <i>BMT</i> . . . . .	36
<b>3 Evaluation</b>	<b>39</b>
3.1 Evaluation Metrics . . . . .	40
3.2 Statistics Collected . . . . .	41

3.3	Research Questions . . . . .	42
3.3.1	RQ1: Parameters Effect . . . . .	43
3.3.2	RQ2: BMT Optimizations Effect . . . . .	44
3.3.3	RQ3: Quality of Allocation . . . . .	46
3.3.4	RQ4: Efficiency . . . . .	49
3.3.5	RQ5: Distance from Optimal . . . . .	50
<b>4</b>	<b>Literature Review</b>	<b>59</b>
4.1	IBM Qiskit Mapper . . . . .	61
4.2	A-star Search . . . . .	63
4.3	SABRE . . . . .	65
4.4	4-Approximative Token Swapping . . . . .	66
4.4.1	Colored Token Swapping . . . . .	68
<b>5</b>	<b>Conclusion</b>	<b>73</b>
	<b>Bibliography</b>	<b>75</b>
	<b>Appendix A Adapting the TWP Algorithm for BMT.</b>	<b>81</b>
A.1	Optimizing for Partial Mappings . . . . .	81

# Chapter 1

## Introduction

The recent introduction of cloud access to quantum computers has made experimental quantum computing (QC) available to a wide community [Devitt, 2016]. For instance, the IBM Quantum Experience program<sup>1</sup> lets users build experiments based on either a visual circuit representation or a gate-level language based on the Quantum Assembler (QASM) syntax [Cross et al., 2017; Svore et al., 2006]. However, developing quantum programs is challenging: the level of abstraction offered by current programming languages is low; circuits need to obey machine-specific restrictions [Häner et al., 2016]; and today’s quantum computers have tight resource constraints. As of today, IBM users have access to architectures with 5, 16, and 20 qubits, although a 50-qubit machine has been announced [Gil, 2017]. Nevertheless, the connectivity between qubits of these computers remains very restrictive. Consequently, manual mapping and tuning of quantum algorithms is difficult.

In addition, decoherence and noise effects severely constrain the execution time. Unlike classical digital gates that are inherently self-stabilizing, quantum gates accumulate noise. Although quantum error-correcting codes (QEC) hold the promise to address decoherence issues [Lidar and Brun, 2013], current hardware do not provide nearly enough resources to implement realistic QEC [Cross et al., 2017; Mohseni et al., 2017]. The longer a quantum program runs and the more operations it performs, the more it is susceptible to noise. Therefore, minimizing runtime and complexity is crucial, as it does not just affect the time-to-solution, but also the accuracy of the solution itself. For these reasons, compilation of quantum circuits demands extremely accurate compiler optimization.

Quantum circuits manipulate *qubits* – the quantum analogue of the classical bit. These qubits, which exist as abstractions within a quantum circuit, shall be called

---

<sup>1</sup><http://research.ibm.com/ibm-q/>

*pseudo* or *logical*. In this work, we are interested in mapping *pseudo qubits* into *physical qubits*, which denote the actual hardware units that store quantum bits. This problem henceforth shall be called *qubit allocation*. Just like registers in a classical computer architecture, quantum computers have a limited number of qubits. Furthermore, these units are not always fully connected, meaning that not every subset of physical qubits can participate as inputs and outputs to the same quantum gates. As we explain in Chapter 2, solving qubit allocation involves dealing with hard combinatorial problems.

This problem: the mapping of quantum circuits into arbitrary quantum machines has been referred as *quantum circuit placement* [Maslov et al., 2008]; *mapping problem* [Zulehner et al., 2018]; and *qubit allocation* [Siraichi et al., 2018]. Henceforth, we shall adopt the latter terminology. There exist different solutions to qubit allocation [Lin et al., 2015; Maslov et al., 2008; Pedram and Shafaei, 2016; Shafaei et al., 2014; Siraichi et al., 2018; Zulehner et al., 2018]; however, contrary to classic register allocation, a problem elegantly modelled via graph coloring [Chaitin et al., 1981], qubit allocation still lacks principled solutions. Subgraph isomorphism emerges as a candidate to provide a fundamental metaphor to it. Nevertheless, subgraph isomorphism can only model small instances of qubit allocation, which do not require transformations in the quantum circuit [Siraichi et al., 2018].

Summarizing, we formally describe the qubit allocation problem and introduce an exact solution to solve it in Chapter 2. The exact algorithm is an exponential-time solution. Although it works well for the small quantum systems available today, it cannot scale up to the more complex architectures that are likely to emerge in the future. Nevertheless, it sets a mark against which we can test different heuristics. To support this statement, we show how state-of-the-art implementations of qubit allocators fare against this exact baseline. This comparison has motivated us to go beyond these implementations; a task that we accomplish with two novel allocators of our own craft, which we also introduce in Chapter 2.

Chapter 3 provides a thorough evaluation of the different algorithms that exist today to perform qubit allocation. Not many classes of quantum algorithms are known; and even fewer accommodate the constraints of early quantum computers [Nielsen and Chuang, 2000]. Thus, we have assembled the collection of benchmarks most used by researchers in past work in the literature [Shafaei et al., 2014; Lin et al., 2015; Pedram and Shafaei, 2016; Lao et al., 2018; Lin et al., 2018; Zulehner et al., 2018; Li et al., 2018], and have implemented a generator of random programs used in Siraichi et al. [2018].



**Contributions.** An earlier version of this work was published in the International Symposium on Code Generation and Optimization (CGO'18) in 2018 [Siraichi et al., 2018], regarding the problem statement, as well as the simpler solution present in this paper: the exact dynamic programming solution (Section 2.1); and the weighted partial matching (Section 2.2). An extended version of this work was also submitted to a journal in the end of 2018, with the new heuristic, described in Section 2.3. Finally, as a by-product of this dissertation, the *Enfield* compiler<sup>2</sup> was built.

## 1.1 Background

This section introduces the base knowledge in order for one to understand the qubit allocation problem, as well as our described solutions. Qubit allocation involves modifying quantum circuits with specific combinations of quantum gates, which we call *transforms*. Although familiarity with qubits and quantum gates might be helpful to understand the problem, we shall try to keep our discussion on a level that suits the reader unversed with quantum computing. For a more thorough discussion, we refer the interested reader to Nielsen and Chuang [2000].

### 1.1.1 Qubits and Quantum Gates.

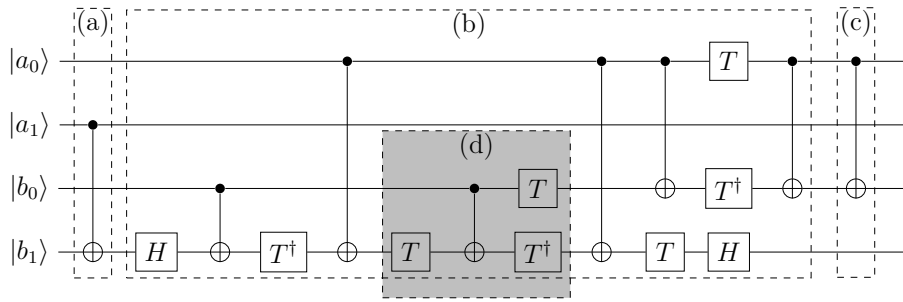
Quantum programs are made of qubits and reversible quantum gates, which receive qubits as inputs, and produce qubits as outputs. Figure 1.1 shows an example of a quantum circuit. This circuit has four qubits:  $a_0$ ,  $a_1$ ,  $b_0$  and  $b_1$ , which are represented as horizontal lines. It uses four different types of gates to operate on these qubits:  $H$ ,  $T$ ,  $T^\dagger$  and  $\text{CNOT}_{ab}$ , where  $\text{CNOT}_{ab}$  is depicted with a dot on qubit  $a$  and  $\oplus$  on qubit  $b$ . Gates change the *state* of qubits. The state of a single qubit is represented as a two dimensional complex vector:

$$\alpha |0\rangle + \beta |1\rangle = \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

In this case,  $|0\rangle$  and  $|1\rangle$  are the basis states of a 2D complex vector space, and  $\alpha$  and  $\beta$  are complex numbers. Under this terminology, quantum gates can be understood as unitary matrix operations applied on vectors that describe quantum states. Example 1 illustrates this view.

---

<sup>2</sup>OpenQASM source-to-source compiler, available at <https://github.com/ysiraichi/enfield>



**Figure 1.1.** Circuit representation of  $(a_1a_0 + b_1b_0)\%4$ . Each wire represents the state of a qubit as time passes by. The circuit is divided into three parts: (a)  $a_1 \oplus b_1$ ; (b)  $(a_0 \wedge b_0) \oplus b_1$ ; (c)  $a_0 \oplus b_0$ . Output is stored in  $b_1b_0$ . (d) shows the application of gates one in front of the other, and combination of them (parallel).

**Example 1.** The Hadamard-Walsh gate  $H$  maps the basis state  $|0\rangle$  to  $(|0\rangle + |1\rangle)/\sqrt{2}$ , and  $|1\rangle$  to  $(|0\rangle - |1\rangle)/\sqrt{2}$ . Thus, it is equivalent to multiplying the quantum state by the matrix:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Like the  $H$  and other single-qubit gates, the  $T$  gate is represented as a  $2 \times 2$  matrix that multiplies a quantum state. Gate  $T^\dagger$  is its inverse, meaning that  $TT^\dagger$  is the identity matrix. The CNOT (short for *Controlled Not*) gate is applied on two qubits.  $CNOT_{ab}$  indicates that  $a$  controls  $b$ . Informally, it negates  $b$ , the second qubit, when  $a$ , the first qubit, is  $|1\rangle$ . When  $a$  is  $|0\rangle$ , the gate leaves  $b$  unchanged. Below, we show the matrix for the CNOT operation:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

A  $CNOT$  gate is applied on pairs of qubits. Pairs of qubits are represented by 4-line vectors. These vectors come from the application of the tensor product  $\otimes$  to the 2D matrices that represents each individual qubit. Example 2 illustrates how pairs of qubits are combined.

**Example 2.** If  $|a\rangle$  and  $|b\rangle$  are the states of qubits  $a$  and  $b$ , then their combined state is given by the tensor product:

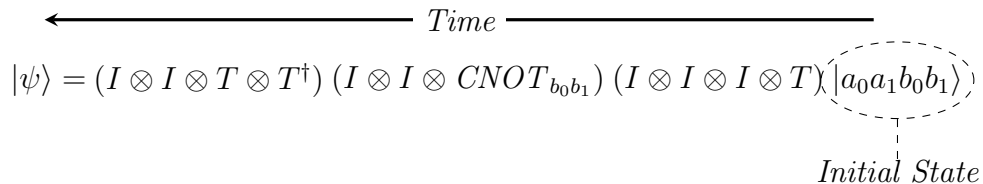
$$|a\rangle \otimes |b\rangle = |ab\rangle = \begin{bmatrix} \alpha_a \\ \beta_a \end{bmatrix} \otimes \begin{bmatrix} \alpha_b \\ \beta_b \end{bmatrix} = \begin{bmatrix} \alpha_a \alpha_b \\ \alpha_a \beta_b \\ \beta_a \alpha_b \\ \beta_a \beta_b \end{bmatrix}$$

The dimension of the combination is  $\dim(|a\rangle) \times \dim(|b\rangle)$ .

An informal summary of the semantics of a program such as the one seen on Figure 1.1 is the following: each individual gate, e.g.,  $H$ ,  $T^\dagger$  or  $T$ , represents a 2D-matrix; the application of such a gate into a qubit corresponds to matrix multiplication; and the combination of multiple gates applied to different qubits is given by the tensor product of those  $\otimes$ . Example 3 illustrates this view.

**Example 3.** Assuming that the initial state is given by  $|a_0 a_1 b_0 b_1\rangle = |a_0\rangle \otimes |a_1\rangle \otimes |b_0\rangle \otimes |b_1\rangle$ , and the final state is  $|\psi\rangle$ , the columns in Figure 1.1 (d) corresponds to the matrix product below:

$$|\psi\rangle = (I \otimes I \otimes T \otimes T^\dagger) (I \otimes I \otimes CNOT_{b_0 b_1}) (I \otimes I \otimes I \otimes T) |a_0 a_1 b_0 b_1\rangle$$



The empty wires over qubits  $a_0, a_1$  and  $b_0$  (in the first column) represents the identity matrix  $I$ , i.e. the absence of any gate.

### 1.1.2 Architectural Constraints.

The single-qubit gates plus the CNOT gate form a universal set of gates that can implement arbitrary circuits Barenco et al. [1995]. Nevertheless, their exact semantics is immaterial to our exposition. Important to us is whether they are single-qubit or two-qubit gates. Even though sequences of single-qubit gates may sometimes be simplified away, thus reducing the total cost of the output program, we shall focus only on CNOT gates in this work.

The placement of CNOT gates matters due to *architectural constraints*. Actual quantum computers might not allow CNOTs to be performed between arbitrary pairs

of qubits. In particular, quantum computers based on superconducting qubit technology are made of solid-state circuits that only allow local interactions between qubits that are physically connected [Devoret et al., 2004; Koch et al., 2007]. Technological reasons restrict the number of possible couplings and their organization [Gambetta et al., 2017]. As an example, Figure 1.2 (a) shows the *coupling graph* of the IBM Yorktown computer [Devitt, 2016]. The coupling graph determines which qubits can communicate, typically through CNOT gates. The coupling graph is defined in terms of CNOT gates as follows:

**Definition 1** (Coupling Graph). *Given a quantum architecture  $A$  with a set  $Q$  of qubits, its coupling graph is a directed graph  $G_q = (Q, E_q), E_q \subseteq Q \times Q$ . The edge  $(q_1, q_2) \in E_q$  if, and only if,  $CNOT_{q_1 q_2}$  is valid in  $A$ .*

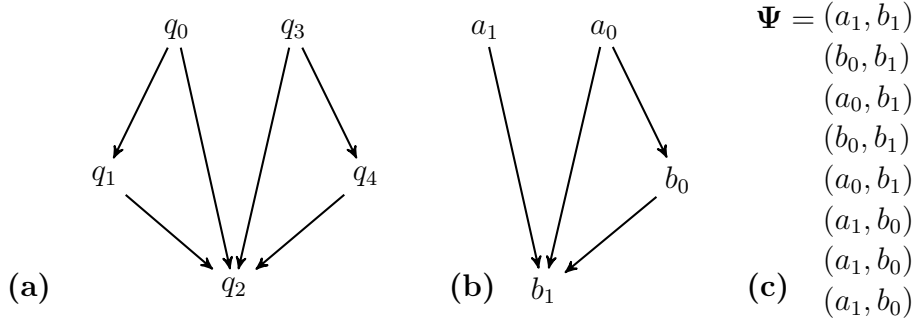
### 1.1.3 Qubit Allocation – An Informal Overview.

The connectivity relations in a quantum circuit need to be mapped to the coupling graph. For instance, in Figure 1.1, we have that the pseudo qubit  $a_0$  controls pseudos  $b_0$  and  $b_1$ . When allocating pseudo qubits onto the coupling graph, we would like to enable such control relations. Example 4 illustrates one valid allocation for the previously mentioned circuit. However, *perfect mappings* that enable all the control relations in a quantum circuit are not always possible.

**Example 4.** *It is possible to map the control circuit of Figure 1.1 directly onto the coupling graph of Figure 1.2 (a), with mapping  $f = \{a_0 \mapsto q_3, a_1 \mapsto q_0, b_0 \mapsto q_4, b_1 \mapsto q_2\}$ . The graph in Figure 1.2 (b) represents the control relations (dependencies) in that circuit, represented by Figure 1.2 (c). This graph (Figure 1.2 (b)) contains two nodes of in-degree two, which have no equivalent in Figure 1.2 (a).*

In its simplest version, the qubit allocation problem receives an ordered list of pairs, describing control relations in the quantum circuit, plus a coupling graph. This problem, which Definition 2 states, asks for a mapping between pseudos and physical qubits that respects the control relations. Because Definition 2 does not ask for ways to adapt a circuit to fit into a coupling graph, we call this version of qubit allocation the *Assignment Problem*. Perfect mappings might not exist. In this case, we must resort to *circuit transformations* to solve qubit allocation. This is a notion that we discuss in the rest of this section.

**Definition 2** (The Qubit Assignment Problem). **Input:** *a coupling graph  $G_q = (Q, E_q)$ , plus a list  $\Psi = (P \times P)^n, n \geq 1$  of  $n$  control relations between pseudo qubits.*



**Figure 1.2.** (a) The coupling graph of the IBM Yorktown computer. (b) Interactions between qubits of the circuit seen in Figure 1.1. (c) Dependences that have created these interactions.

**Output:** *yes, if there is a mapping between pseudo and physical qubits that respects the control relations in  $\Psi$ .*

**Circuit Transformations.** A transformation is a combination of gates that we can insert into a quantum circuit to emulate the semantics of non-existing CNOT relations or switch the state of physical qubits. We call the first category of transformations *virtual CNOTs*, and the latter *state changes*. Example 5 describes some of these transformations.

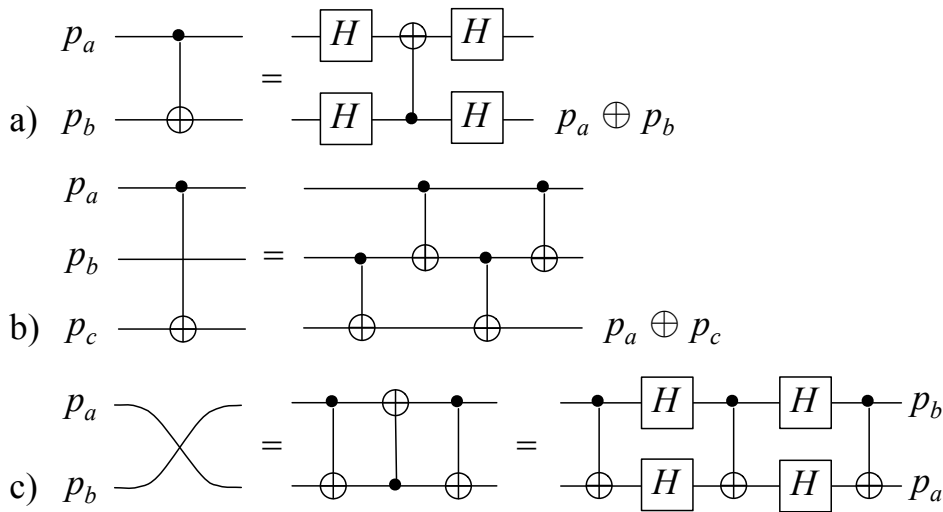
**Example 5.** *Below we list three kinds of transformations:*

**Reversal:** *Emulation of a virtual CNOT between  $p_a$  and  $p_b$  controlled by  $p_a$  using a CNOT from  $p_b$  to  $p_a$  (controlled by  $p_b$ ) and 2 extra levels of Hadamard gates, as shown in Figure 1.3 (a).*

**Bridge:** *Emulation of a virtual CNOT between  $p_a$  and  $p_c$  controlled by  $p_a$  using two CNOTs from  $p_a$  to  $p_b$  (controlled by  $p_a$ ), plus two CNOTs from  $p_b$  to  $p_c$  (controlled by  $p_b$ ), as shown in Figure 1.3 (b).*

**Swap:** *exchanges two pseudo qubits  $p_a$  and  $p_b$ , as shown in Figure 1.3 (c), at the expense of three CNOT and two levels of Hadamard gates.*

As Figure 1.3 shows, a CNOT reversal allows the mapping of “backward” edges on the coupling graph, at the cost of extra gates. A bridge uses four CNOTs to implement a virtual gate at distance 2 in the coupling graph. Finally, a CNOT swap allows the migration of pseudo qubits across physical qubits. Whereas reversals and bridges are gate transformations, swaps transform states. That is to say: a reversal



**Figure 1.3.** (a) Reversal. (b) Bridge. (c) Swap.

inverts the meaning of a CNOT gate, and a swap exchanges the position of two pseudo qubits. These transformations can be combined to map a quantum circuit onto a given architecture. Example 6 shows that.

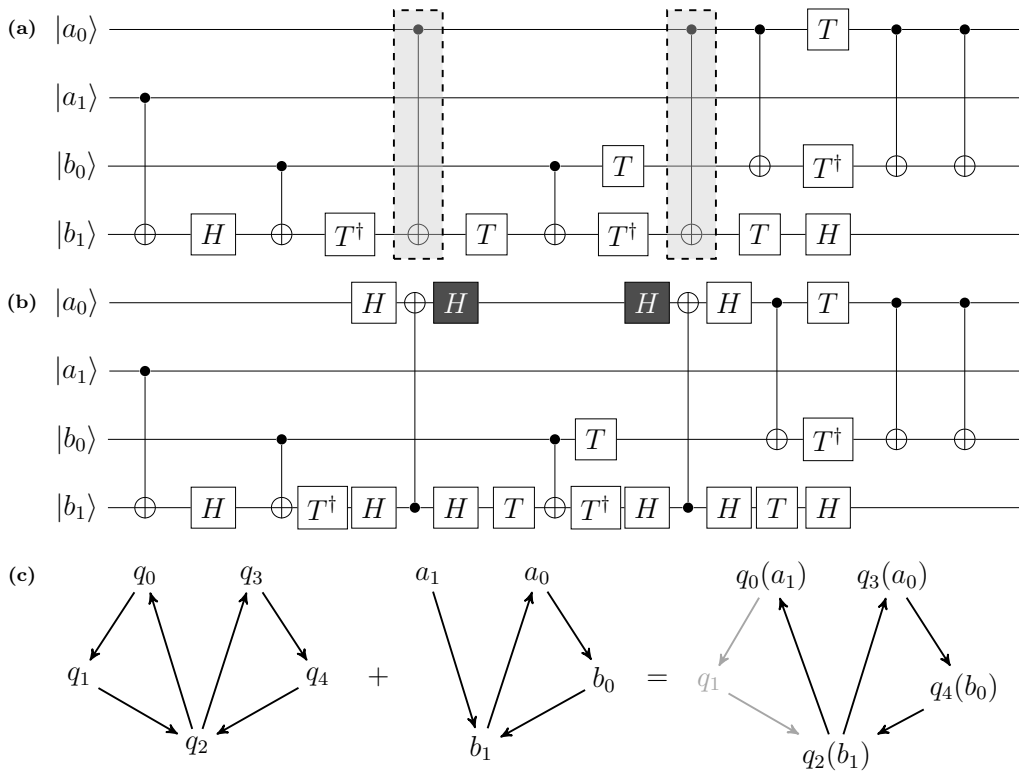
**Example 6.** *Figure 1.4 outlines a solution to qubit allocation for the program in Figure 1.1 using two CNOT reversals. Reversals add further complexity to the target circuit; however, some gates can be simplified away, given well-known quantum identities [Lomont, 2003].*

A particular instance of qubit allocation might have several different solutions. As an example, Figure 1.5 shows an allocation for our running example, this time using one CNOT swap, instead of two reversals. The quality of a solution is given by its *cost*, which we measure as the number of gates necessary to implement it. Thus, ideally we wish to minimize such cost.

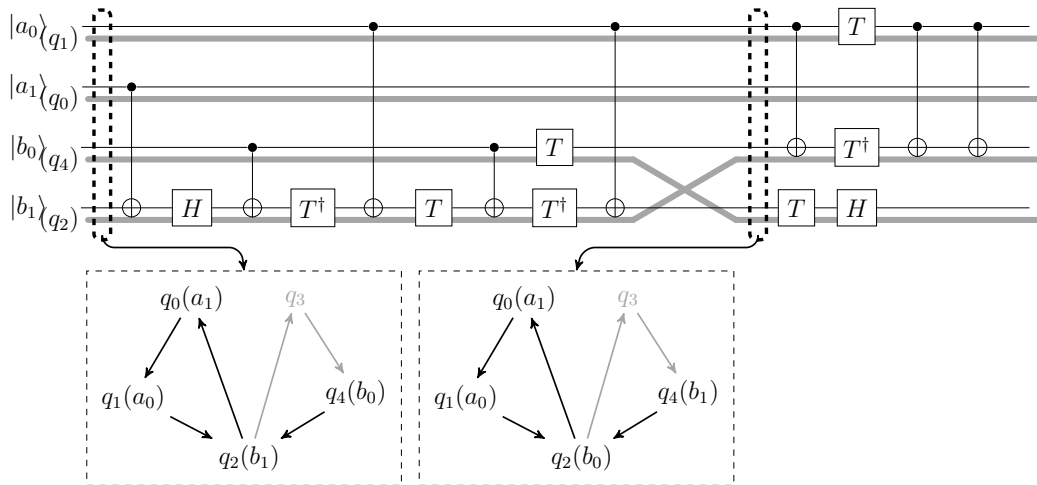
### 1.1.4 Subgraph Isomorphism

Subgraph isomorphism is one of the key components of our solution to qubit allocation. For the sake of completeness, we define this problem below, and illustrate it in Example 7.

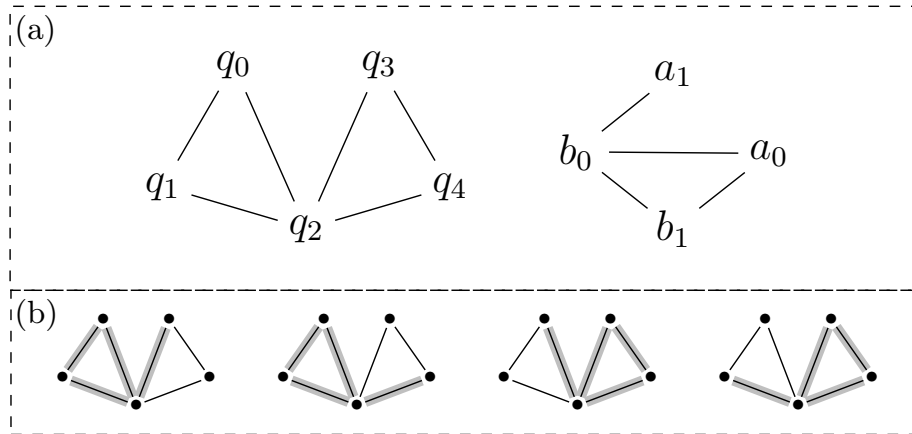
**Definition 3** (Subgraph Isomorphism – SIP). **Input:** undirected graphs  $G$  and  $H$ . **Output:** yes, if we can find a subgraph  $H'$  of  $H$ , plus a bijection  $f : V(G) \rightarrow V(H')$ , where for every edge  $(u, v) \in E(G)$ ,  $(f(u), f(v)) \in E(H')$ .



**Figure 1.4.** (a) CNOT reversals, marked as grey boxes, invert the direction of  $\text{CNOT}_{b_0 b_1}$ . (b) The two black Hadamard gates can be simplified away, given the identity  $HH = I$ . (c) Solution to qubit allocation showing embedding of the control graph onto the coupling graph.



**Figure 1.5.** Solution of qubit allocation for the circuit in Figure 1.1, using a CNOT swap. Grey lines represent physical qubits. We show the different mappings that we have at two points of the circuit.



**Figure 1.6.** (a) two undirected graphs  $G$  and  $H$  (left to right); (b) all possible subgraphs of  $G$  that are isomorphic to  $H$ .

**Example 7.** Figure 1.6 shows different solutions to an instance of subgraph isomorphism. Given the two undirected graphs in Figure 1.6 (a),  $G$  and  $H$  (left to right), Figure 1.6 (b) shows all possible isomorphic subgraphs. The bijections are:

- $\{a_0 \mapsto q_0, a_1 \mapsto q_3, b_0 \mapsto q_2, b_1 \mapsto q_1\}$ ;
- $\{a_0 \mapsto q_0, a_1 \mapsto q_4, b_0 \mapsto q_2, b_1 \mapsto q_1\}$ ;
- $\{a_0 \mapsto q_3, a_1 \mapsto q_0, b_0 \mapsto q_2, b_1 \mapsto q_4\}$ ;
- $\{a_0 \mapsto q_3, a_1 \mapsto q_1, b_0 \mapsto q_2, b_1 \mapsto q_4\}$ .

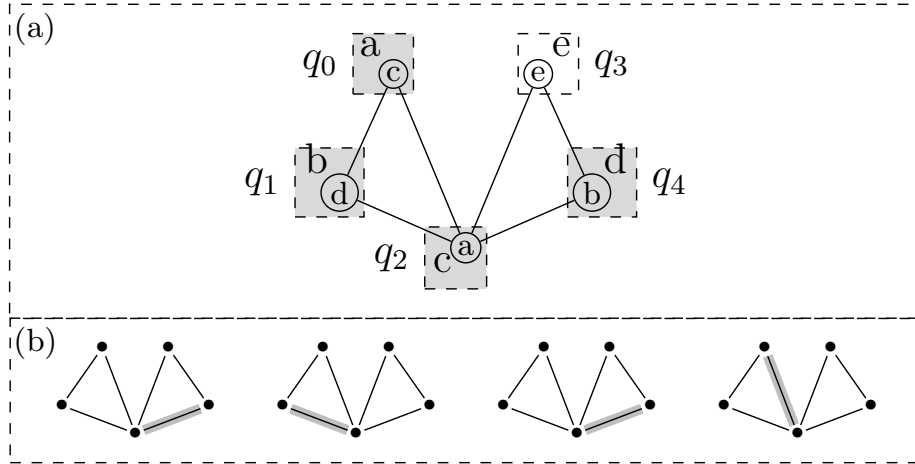
Notice that  $a_0$  and  $b_1$  are equivalent w.r.t. the rest of the graph; hence by switching  $a_0$  and  $b_1$  we double the number of bijections.

Problem 3 is NP-complete [Cook, 1971]. Different heuristics have been proposed to solve it [Cordella et al., 2004; He and Singh, 2008; Zhao and Han, 2010; Han et al., 2013]. As Section 2.3.1 will discuss, applying these heuristics would compromise the scalability of our algorithm. Therefore, we shall propose a parameterized algorithm to bound the number of instances of subgraph isomorphism to be solved, and shall rely on a greedy search to find solutions to individual instances of this problem. By bounding the search space we might not find an optimal solution to Problem 3. In other words, parameterization will let us exchange accuracy for time.

### 1.1.5 Token Swapping

Token Swapping is another central element to our solution to qubit allocation. This problem was introduced by Yamanaka et al. [2014], and proven to be NP-hard by





**Figure 1.7.** (a) undirected graph of Figure 1.6 (a) with the initial state of the tokens ( $f_{src}$ ) inside the circle, the final state of the tokens ( $f_{tgt}$ ) outside the circle. Grey boxes indicate vertices whose tokens are already in its target place; (b) a sequence of swaps that solves the problem (highlighted edges).

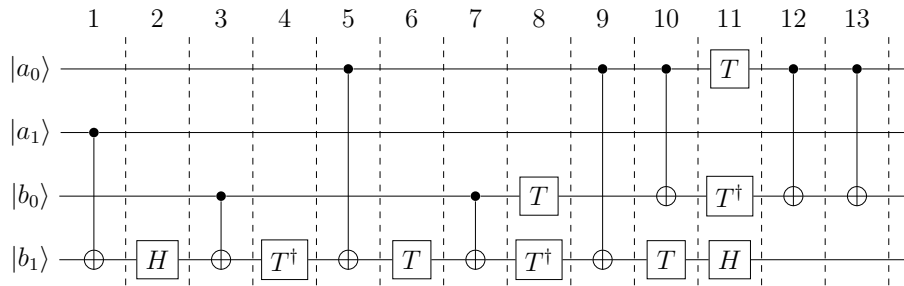
Miltzow et al. [2016]. For completeness, we restate the definition of this problem below:

**Definition 4** (Token Swapping - TWP). *Input:* a set of tokens  $T$ , an integer  $k$ , an undirected graph  $G = (V, E)$ , two bijective functions  $f_{src}, f_{tgt} : T \rightarrow V$  representing the initial state and the desired final state of the tokens, respectively. *Output:* yes, if we can transform  $f_{src}$  into  $f_{tgt}$  with up to  $k$  swap operations, where a swap is an operation that exchanges tokens in two adjacent vertices.

Research around Problem 4 is recent; hence, it has not been as deeply studied as Problem 3. Among current approaches to solve token swapping, we count two approximative algorithms [Miltzow et al., 2016; Yamanaka et al., 2017], and an exponential method [Surynek, 2018]. Example 8 illustrates Problem 4.

**Example 8.** Figure 1.7 (a) shows both  $f_{src} = \{a \mapsto q_2, b \mapsto q_4, c \mapsto q_0, d \mapsto q_1, e \mapsto q_3\}$  (bigger letters outside the circle), and  $f_{tgt} = \{a \mapsto q_0, b \mapsto q_1, c \mapsto q_2, d \mapsto q_4, e \mapsto q_3\}$  (smaller letters inside the circle). Thick edges in Figure 1.7 (b) represent swaps. The sequence of swaps  $(q_2, q_4), (q_1, q_2), (q_2, q_4), (q_0, q_2)$ , takes us from  $f_{src}$  to  $f_{tgt}$  in 4 steps.

As we shall see in Section 2.3.2, token swapping let us partition qubit allocation into smaller problems. We solve these smaller problems via subgraph isomorphism, and then use token swapping to stitch solutions together. The beauty of this approach is that the partitioning meets Bellman's *Principle of Optimality* [Bellman, 1958]. Hence,



**Figure 1.8.** Quantum circuit divided into a sequence of layers. Each column represents one layer.

we can find an optimal solution to qubit allocation by uniting solutions of the smaller problems via dynamic programming.

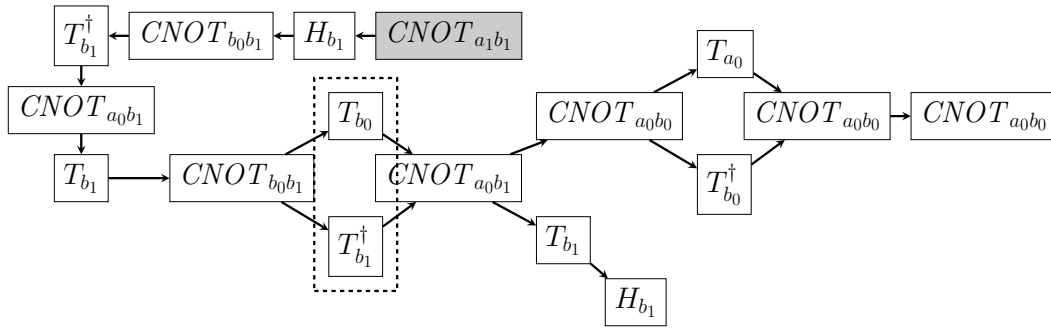
## 1.2 Quantum Programs Representation

In the previous section, we showed three representations for quantum programs: the quantum circuit; the graph of interactions between the qubits; and a sequence of dependencies (Figure 1.1, and Figure 1.2 (b) and (c), respectively). While the first one consists in a graphical representation, the second and the third one were used in previous work [Maslov et al., 2008; Shafaei et al., 2014; Shrivastwa et al., 2015; Lin et al., 2015; Pedram and Shafaei, 2016; Lin et al., 2018; Lao et al., 2018; Siraichi et al., 2018] for solving such problem. In this section, we shall present two more representations which can be derived from the sequence of dependencies (Figure 1.2 (c)): the layer sequence and the gate dependency graph.

**Layer Sequence.** A layer can be defined as a set of gates, such that there is no intersection on the input qubits of each of those gates. As such, it is possible to represent a quantum program as a sequence of layers. One way to easily visualize it, in the circuit, is by shifting all gates to the left as much as possible. Then, each column would correspond to one layer. Example 9 illustrates a sequence of layers.

**Example 9.** Figure 1.8 shows the circuit of Figure 1.1 segmented into layers. Each gate occupies up to one column. Each column corresponds to one layer. In this case, the circuit has a total of 13 layers.

**Gate Dependency Graph.** From a sequence of dependencies, we can build a graph  $G = (V, E)$ , such that  $V$  represents every gate in the program, and  $E$  shows the



**Figure 1.9.** Gate dependency graph of the quantum circuit in Figure 1.1. The vertices with no incoming edges (highlighted) may be executed. The vertices inside the dashed box do not have a correct order to be executed in.

chronological order between pairs of gates. i.e.  $(g_i, g_j) \in E$  iff  $g_i$  appears in the program before  $g_j$ , and the intersection between the qubits in  $g_i$  and  $g_j$  is not empty.

**Example 10.** Figure 1.9 shows the “Gate Dependency Graph” for the quantum program illustrated in Figure 1.1. There is one vertex for each gate, and there is only one edge from  $g_i$  to  $g_j$  in the graph iff gate  $g_i$  is to be executed strictly before  $g_j$ . The second vertex  $H_{b_1}$ , for example, can only be executed after the first vertex (highlighted), since they operate on the same qubit. Vertices such as  $T_{b_0}$  and  $T_{b_1}^\dagger$  (inside the dashed box) do not have any dependency among each other. Thus, they may be executed in any order.



## Chapter 2

# The Qubit Allocation Problem

Definition 2 states the Qubit Allocation Problem in its most basic form: given a quantum circuit and an architecture, we want to know if it is possible to map the pseudo qubits in the former to the physical qubits in the latter. Making an analogy with classic register allocation, the problem in Definition 2 is equivalent to knowing if we can map program variables (pseudo registers) onto the physical registers available in the target architecture. Like classic register allocation<sup>1</sup>, Qubit Assignment is NP-complete, as Theorem 1 states.

**Theorem 1.** *Qubit Assignment (Def. 2) is NP-complete.*

**Proof** (Sketch): We make a reduction from subgraph isomorphism, which is known to be NP-hard [Cook, 1971]. First, note that finding isomorphisms between *directed* graphs is also NP-hard, since replacing every edge by two directed edges doesn't change the answer of any input. Given an instance of subgraph isomorphism, where we wish to find a subgraph of  $G$  that is isomorphic to  $H$ , we can map the graph  $G$  to the coupling graph and the edges of the graph  $H$  to individual CNOT instructions in  $\Psi$ . Clearly, any solution of Qubit Assignment would find an embedding of  $H$  in  $G$  iff such embedding exists. Therefore, we can conclude that Qubit Assignment is NP-hard. To complete the proof it is enough to notice that checking in any solution if all pairs of  $\Psi$  are properly mapped can be done in polynomial time.  $\square$

Theorem 1 sets our expectations about having an exact solution to solve Qubit Allocation. However, from a practical standpoint, Qubit Assignment is not very useful: most of the instances of Qubit Allocation will require quantum transformations

---

<sup>1</sup>See the proof of Chaitin et al. [1981].

to be effectively solved. Going back to our analogy with register allocation, most instances of register allocation lead to spilling; hence, forcing the insertion of load and store instructions in the program: program changes equivalent to our transformations. Thus, in the rest of this section we extend Definition 2 to encompass more pragmatic descriptions of the Qubit Allocation problem. We start with the subproblem that asks for the minimization of swaps.

**Definition 5** (The Swap Minimization Problem). ***Input:** a coupling graph  $G_q = (Q, E_q)$ , a list  $\Psi = (P \times P)^n, n \geq 1$  of  $n$  control relations between pseudo qubits, and an integer  $K_s \geq 0$ . **Output:** yes, if we can use up to  $K_s$  swaps to produce a version of  $\Psi$  that complies with  $G_q$ .*

Swap Minimization is also NP-complete, because it involves solving a classic optimization problem known as the Token Swapping Problem [Yamanaka et al., 2014]. Quoting Kawahara et al. [2017], “For a given graph where each vertex has a unique token on it, token swapping requires to find a shortest way to modify a token placement into another by swapping tokens on adjacent vertices.” Token Swapping has been shown to be NP-Hard [Bonnet et al., 2016; Kawahara et al., 2017]. Swap Minimization is a special case of Qubit Allocation. In the most general problem, we can use quantum transformations other than swaps, and each one of them might have a different cost. We define this problem as follows:

**Definition 6** (The Qubit Allocation Problem). ***Input:** a coupling graph  $G_q = (Q, E_q)$ , a list  $\Psi = (P \times P)^n, n \geq 1$  of  $n$  control relations between pseudo qubits, an integer  $K_c \geq 0$ , a list of allowed quantum transformations  $\Theta$ , and a function  $C : \Theta \mapsto \mathbb{N}$  that gives the cost to implement each transformation. **Output:** yes, if we can produce a version of  $\Psi$  that complies with  $G_q$  with transformations whose total cost does not exceed  $K_c$ .*

Definition 6 subsumes the two simpler problems, stated in Definitions 2 and 5; therefore, it is unlikely that it can be solved exactly via a polynomial time algorithm. Definition 6 states the version of qubit allocation that we solve along the rest of this dissertation. In Section 2.1, we provide an optimal – exponential time – solution to that problem; in Section 2.2, we provide a heuristic solution to it. For the reader’s convenience, Figure 2.1 summarizes terms and notation adopted henceforth.

$P$	<b>Pseudo-qubits</b>	the set of qubits in $\Psi$ .
$Q$	<b>Physical Qubits</b>	the set of qubits present in the architecture. i.e. the vertices of the coupling graph.
$f$	<b>Mapping</b>	a function $f : P \rightarrow Q \cup \{\perp\}$ from pseudo-qubits to physical-qubits. The symbol $\perp$ denotes pseudo qubits that are not mapped to any physical qubit.
$\mathcal{F}$	<b>Set of All Mappings</b>	a set $\mathcal{F} = \mathcal{P}(P \rightarrow Q \cup \{\perp\})$ of all possible mappings.
$F$	<b>Set of Mappings</b>	a set $[F : \mathcal{F}]$ from all possible mappings.
$G_q^d$	<b>Directed Coupling Graph</b>	a graph, whose vertices ( $Q$ ) correspond to the physical qubits. An edge from $q_i$ to $q_j$ means that $q_i$ can control $q_j$ via a CNOT gate.
$G_q^u$	<b>Undirected Coupling Graph</b>	undirected version of the Directed Coupling Graph.
$\Theta$	<b>Quantum Transformations</b>	in our implementation, we consider $\Theta = \{\theta_s, \theta_c, \theta_r, \theta_b\}$ , representing swap, CNOTs, reversals and bridges, respectively.
$C$	<b>Cost Function</b>	the cost of transformations $C : \Theta \rightarrow \mathbb{N}$ .
$\Psi$	<b>Input Control Relations</b>	the sequence $\Psi : (P \times P)^n$ of $n$ control relations between pseudo qubits. These are the dependences that qubit allocation must satisfy.
$\mathcal{G}_\Psi$	<b>Derived Program Graph</b>	the unique graph determined by $\Psi$ . The graph has a vertex $v_p$ for each pseudo-qubit $p$ used in $\Psi$ , and an edge $(v_i, v_j)$ if $(i, j) \in \Psi$ .
$H \lesssim G$	<b>Subgraph Isomorphism Relation</b>	indicating that $H$ is isomorphic to some subgraph of $G$ .

Figure 2.1. Notation used in this work.

## 2.1 Dynamic Programming

We solve the Qubit Allocation problem, as given in Definition 6, using a dynamic programming algorithm. Our approach finds solutions gradually per index in the list of dependences  $\Psi$ . That is, given a collection of control dependences  $\Psi = (p_1, p_2), (p_3, p_4), \dots, (p_{2n-1}, p_{2n})$  between pseudo qubits that must be obeyed, we find

the optimal cost of allocating qubits up to dependence  $i$ . This algorithm is based on a function  $S(f, i)$ , which we define below.

**Definition 7** (Exact Solution). *Function  $S(f, i) : \mathcal{F} \times \mathbb{N} \mapsto \mathbb{N}$  is a solution to the qubit allocation problem if it gives the minimum cost of satisfying all the dependences in  $\Psi$ , up to index  $i$ , terminating with mapping  $f \in \mathcal{F}$ .*

We implement  $S(f, i)$  in terms of three auxiliary functions,  $\phi : \mathcal{F} \times \mathbb{N} \mapsto \mathbb{N}$  and  $\delta : \mathcal{F} \times \mathcal{F} \mapsto \mathbb{N}$ . Function  $\phi$  yields the minimum cost to satisfy a given dependence. Finally, function  $\delta$  gives the minimum cost of swaps necessary to transform a mapping  $f_1$  into another mapping  $f_2$ . We define  $\delta$  at the end of this section.

$$\phi(f, i) = \min_{\theta \in \Theta} \begin{cases} C(\theta) & \text{if mapping } f \text{ satisfies } i \text{ with } \theta \\ \infty & \text{else} \end{cases}$$

$$\delta(f_1, f_2) = \text{Transforms to convert } f_1 \text{ into } f_2$$

From  $\phi$  and  $\delta$ , we solve  $S(f, i)$  as follows:

$$S(f, i) = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } \phi(f, i) = \infty \\ \min_{f' \in \mathcal{F}} S(f', i - 1) + \delta(f', f) + \phi(f, i) & \text{otherwise} \end{cases}$$

**Theorem 2.** *The problem of computing  $S(f, i)$  has optimal substructure.*

**Proof:** We shall prove by induction:

- **Base Case 1:**  $i = 0$  (no dependencies are being considered). Since there are no dependencies, the cost is 0.
- **Base Case 2:**  $\phi(f, i) = \infty$  (can not satisfy this dependency). It is impossible to satisfy this dependency with the current mapping  $f$ .
- **Inductive Case:**  $i > 0$  and  $\phi(f, i) \neq \infty$  (the  $i$ -th dependency is satisfiable with mapping  $f$ ). Assume to have computed  $S(f', i - 1)$  independently, for each possible labeling  $f' \in \mathcal{F}$ . Let us prove by contradicting that  $S(f, i) < S_{min} = \min_{f' \in \mathcal{F}} S(f', i - 1) + \delta(f', f) + \phi(f, i)$ . We do know that we satisfied  $\Psi(i - 1)$  with some mapping  $f''$ . i.e. by definition,  $S(f, i)$  value comes from  $S(f'', i - 1)$ . However, we still have to transform  $f''$  into  $f$ , and account for the cost for satisfying  $\Psi(i)$ . i.e.  $S(f, i) = S(f'', i - 1) + \delta(f'', f) + \phi(f, i)$ .

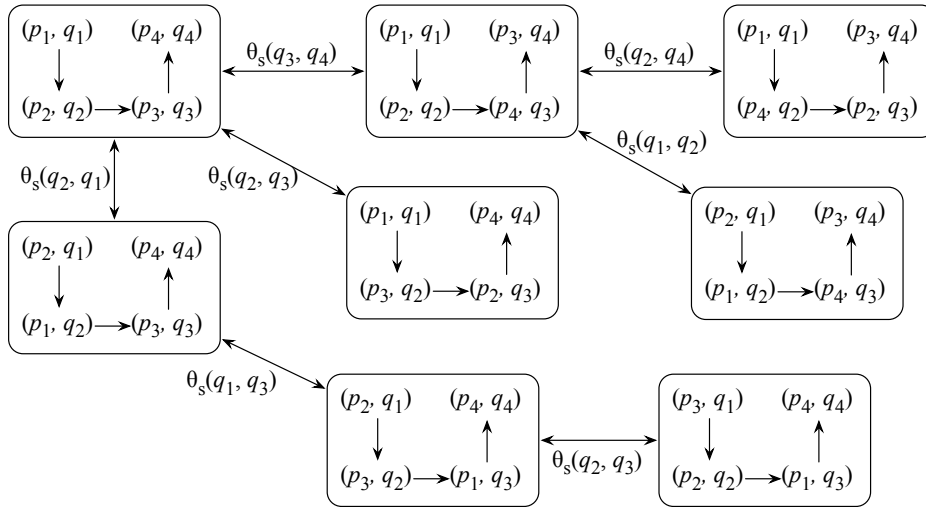


That said,  $S_{min}$  is already the minimum value over all  $f'' \in \mathcal{F}$ . Thus, it is a contradiction.

The implication of this fact is that the recurrence relation that produces  $S$  is a Bellman Equation [Bellman, 1958], a necessary enabler of a dynamic programming algorithm.  $\square$

**Memoizing the State Space** Memoization is an optimization technique that stores the results of function calls and returns the cached result when the same inputs occur again. In our case, memoization is useful to avoid searching repeatedly for optimal sequences of transformations that change a given labeling  $f$  onto another labeling  $f'$ . We memoize all these paths in a table  $\delta$ , already mentioned in the definition of  $S(f, i)$ . We compute  $\delta$  by brute-force, performing a breadth-first search on the space of possible mappings between pseudo and physical qubits. Example 11 illustrates this search.

**Example 11.** *Figure 2.2 shows a tree of different mappings reachable from the initial mapping  $f(p_1) = q_1, f(p_2) = q_2, f(p_3) = q_3$  and  $f(p_4) = q_4$ .*



**Figure 2.2.** Eight states reachable from the initial mapping in the top-left side. In total, we have sixteen states.

The exhaustive search of all the possible labeling gives us a graph  $G_{\mathcal{F}} = (\mathcal{F}, E_{\mathcal{F}})$ , whose vertices are elements  $f \in \mathcal{F}$ . We have an edge from  $f_1$  to  $f_2$  if it is possible to convert  $f_1$  into  $f_2$  with one swap transformation. The minimum sequence of swaps necessary to map a given labeling  $f$  onto another labeling  $f'$  is given by the shortest path between  $f$  and  $f'$  in this graph. The function  $\delta$  that produces the minimum

sequence of swaps transforming one state into another emerges naturally from this graph.  $\delta(f, f')$  is the shortest path between vertices  $f$  and  $f'$  in  $G_{\mathcal{F}}$ . As an artifact of implementation, whenever we compute  $\delta(f, f')$ , for any pair of labelings, we save this result, to avoid further computations, in case the same pair of labelings need to be connected posteriorly.

**On the Complexity of the Exact Solution.** The preprocessing described in the last section enables us to calculate  $\delta_{ff'}$  and  $\Delta(f, f')$  for every  $f, f' \in \mathcal{F}$  by preprocessing the coupling graph only one time. The time complexity of this part of the algorithm is  $O(|Q|! + |Q|! \cdot |E_q|)$ , since we will apply a BFS in  $|Q|!$  different permutations (labelings), each one with up to  $|E_q|$  edges. Given the set of edges  $E_q$ , the space complexity is  $O(|Q|! \cdot |E_q|)$ , since we will visit  $|Q|!$  vertices and for each vertex there are up to  $|E_q|$  edges.

After preprocessing, there is the dynamic programming algorithm. As we can see, it iterates all possible mappings for all dependences. Since we know that:  $O(|Q|!)$  is the number of possible mappings;  $|\Psi|$  is the number of dependences; and  $\delta$  takes linear time to execute, the time complexity of this algorithm is  $O(|Q|!^2 \cdot |Q| \cdot |\Psi|)$  and its space complexity is  $O(|Q|! \cdot |Q| \cdot |\Psi|)$ . Finally, merging the preprocessing with the main algorithm, the time complexity becomes  $O(|Q|!^2 \cdot |Q| \cdot |\Psi| + |Q|! + |Q|! \cdot |E_q|)$ . Thus,  $O(|Q|!^2 \cdot |Q| \cdot |\Psi|)$ .

## 2.2 Weighted Partial Mapping

The algorithm of Section 2.1 provides an exact solution to qubit allocation; however, its exponential runtime renders its application impossible in large coupling graphs. To circumvent this problem, in this section we discuss a heuristic solution to qubit allocation. Later, in Chapter 3 we will show that this faster algorithm leads to results that are close to those found by the exponential time implementation. Our heuristic consists of two stages. The goal of the first stage is to find an initial mapping  $f_0 \in \mathcal{F}$  that attempts to maximize the number of satisfied control dependences. In the ensuing stage, we build a solution that satisfies all the dependence relations in the list of constraints  $\Psi$ , starting from  $f_0$ .

### 2.2.1 Finding the Initial Mapping

Classic register allocation algorithms tend to keep in registers variables that are likely to be more used, such as those that appear in loops, or that appear in a larger number

Given a dependence graph  $G_p = (P, E_p, w_p, w_e)$ :

1. we sort the list of pseudos  $P$  in descending order given by  $w_p$ , thus producing a list  $P^s$  of sorted pseudo qubits;
2. for each element  $p \in P^s$  in order, if  $p$  has not been allocated already:
  - a) we allocate  $p$  to a physical qubit  $q$  that has the nearest out-degree;
  - b) for every  $(p, p') \in \Psi$ , if possible, we allocate  $p'$  to  $q'$ , such that  $(q, q') \in E_q$  and  $p'$  and  $q'$  have the closest out-degree. Then, we repeat for the children of  $p$  in the dependence graph.
3. if there are any unallocated pseudo qubits, we assign a free physical qubit to it.

**Figure 2.3.** Finding an initial mapping to qubit allocation.

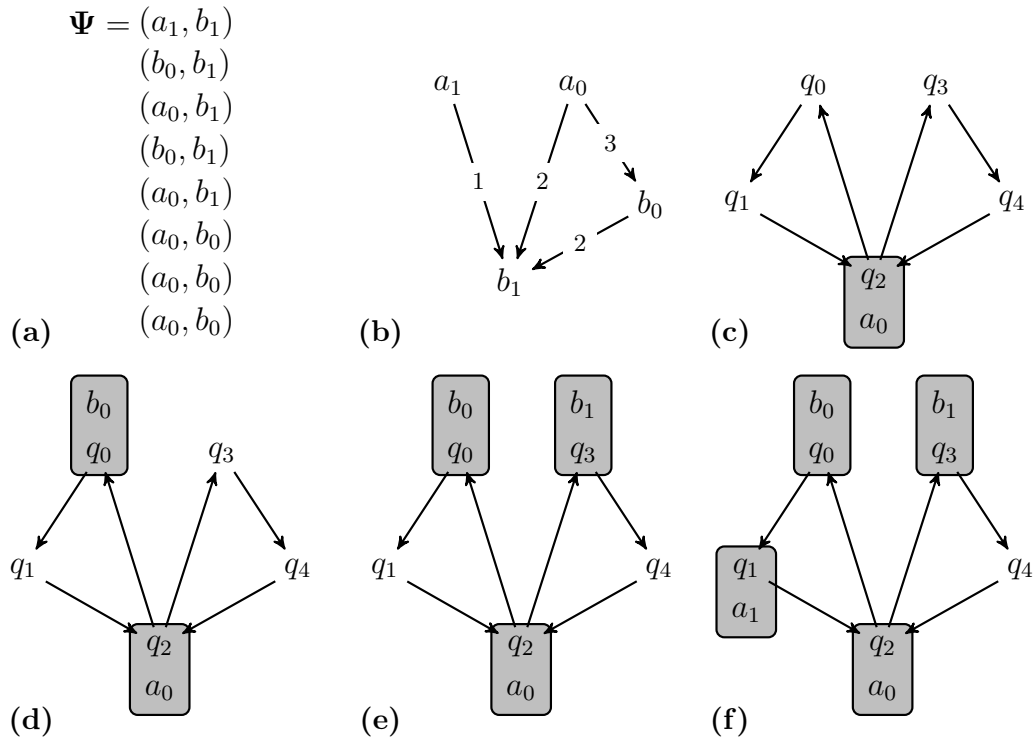
of instructions. Following this insight, in order to find an initial mapping  $f_0$  to some instance of the qubit allocation problem, we try to satisfy the dependences involving pseudo qubits that appear more times in the list of constraints  $\Psi$ .

**Weighted Dependence Graph.** From  $\Psi$ , we construct a weighted directed graph  $G_p = (P, E_p, w_p, w_e)$ , whose vertices are the pseudos that appear in  $\Psi$ . We have an edge  $(p_1, p_2) \in E_p$  whenever  $(p_1, p_2) \in \Psi$ . The weight function  $w_e : P \times P \mapsto \mathbb{N}$  counts the occurrences of dependences in  $\Psi$ . If  $w_e(p_1, p_2) = n$ , then the dependence  $(p_1, p_2)$  appears  $n$  times in  $\Psi$ . From  $w_e$  we define a function  $w_p : P \mapsto \mathbb{N}$  as follows:

$$w_p(a) = \sum w_e(a, b), \forall (a, b) \in E_d$$

**From Weighted Graphs to  $f_0$ .** To find the initial allocation  $f_0$ , we process  $G_p = (P, E_p, w_p, w_e)$  according to the algorithm in Figure 2.3. We use the out-degree criterion as a tie-breaker as a stimulus to allocate pseudos to physicals that will be able to satisfy dependences. If pseudo  $p$  has out-degree  $k$ , then there exist  $k$  other qubits that must, ideally, be allocated to physicals adjacent to the qubit that receives  $p$ . We settle for the closest out-degree to maximize the change that other pseudo qubits can still benefit from the physical qubits of large degree still available in the coupling graph. Example 12 illustrates these issues.

**Example 12.** *Figure 2.4 shows how we find the initial mapping for the circuit earlier seen in Figure 1.1. We shall allocate pseudos in the sequence  $a_0, b_0, a_1, b_1$ . The first pseudo,  $a_0$ , is mapped to  $q_0$ , as they have the same out-degree. In this case, the choice*



**Figure 2.4.** (a) List of control dependences from the quantum circuit seen in Figure 1.1. (b) Weighted dependence graph. Grey boxes represent  $w_p$ . (c-f) Step-by-step construction of the initial mapping.

between  $q_0$  and  $q_3$  is arbitrary. From  $a_0$ , we allocate, recursively,  $b_0$  and  $b_1$ , in a DFS-fashion.

## 2.2.2 Extending the Initial Mapping to handle $\Psi$

On the second stage of our heuristic, we extend  $f_0$ , found in the previous step, so that it satisfies all the dependences in  $\Psi$ . The sequence of steps that we perform to achieve this end is enumerated in Figure 2.5. That algorithm traverses the list  $\Psi$  of dependences that must be satisfied. For each one of them, it might insert transformations in the quantum circuits, if the dependence is not already fulfilled by the current mapping from pseudos to physical qubits.

To implement the dependence  $(p_0, p_1)$  with swaps, we try to move  $p_1$  to some qubit  $q$  that is the successor of  $f(p_0)$ . When performing this movement, we choose always the shortest path from  $f(p_1)$  to  $q$ . Sometimes, it is possible to avoid inserting a swap by changing  $f_0$ , the initial mapping built in Section 2.2.1. This happens when

Given a coupling graph  $G_q^d = (Q, E_q)$ , an initial mapping  $f_0$ , and the dependences  $\Psi$ , for each  $i$  in the domain of  $\Psi$ , let  $(p_0, p_1) = \Psi(i)$ . If  $(f_0(p_0), f_0(p_1)) \notin E_q$ , then:

1. if the edge  $(f_0(p_1), f_0(p_0)) \in E_q$ , then we use a reversal between  $f_0(p_1)$  and  $f_0(p_0)$ ;
2. else we find a path from  $p_0$  to  $p_1$ , and execute this chain of swaps. Thus, approaching both pseudo qubits, and enabling it to be translated directly.

**Figure 2.5.** Extending the initial mapping to satisfy  $\Psi$ .

this swap refers only to physical qubits that have not yet been visited by the loop in Figure 2.5. Example 13 clarifies this possibility.

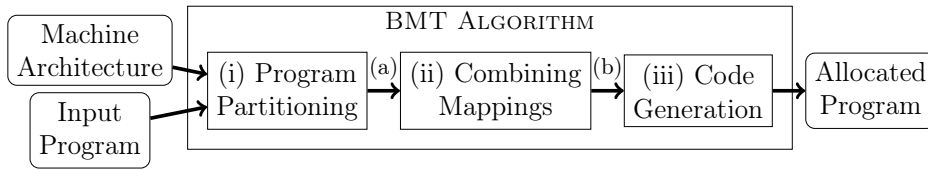
**Example 13.** *The first dependence in Fig. 2.4 that must be satisfied is  $(a_1, b_1)$ . There is no edge  $(f_0(a_1), f_0(b_1)) \in G_q^d$ . To handle this dependence, the algorithm in Fig. 2.5 would swap  $q_2$  and  $q_3$ . However,  $q_2$  and  $q_3$  have not been used as target or destination of any transformation thus far. Hence, we update  $f_0$ , so that  $f(b_1)$  becomes  $q_2$ , and  $f(a_0)$  becomes  $q_3$ .*

To support the optimization discussed in Example 13, we introduce the notion of *freezing*. A qubit is frozen the first time it is used in the loop of Figure 2.5. Frozen qubits cannot be modified in the original mapping. In contrast, qubits yet untouched are swapped “virtually” by changing their original allocation in  $f_0$ . When allocating for the circuit in Figure 1.1, our heuristic finds a solution to qubit allocation involving no swaps.

**Time Complexity.** We find an initial mapping (Section 2.2.1) in  $O(|Q| \cdot lg|Q| + |E_q| + |\Psi|)$ , since we have to order the vertices, and update precedences. The second phase of the heuristic (Section 2.2.2) is  $O(|\Psi| \cdot (|Q| + |E_q|))$ . The worst case scenario happens when we have to run a BFS for each dependence due to the need to implement swaps.

## 2.3 Bounded Mapping Tree

This section introduces *Bounded Mapping Tree*, or *BMT* for short. This algorithm solves qubit allocation by dividing it into two subproblems: Subgraph Isomorphism and Token Swapping. These problems are NP-complete; hence, BMT is *parameterized*. It searches a solution space bounded by two parameters: the *Maximum Number of Children* and the *Maximum Number of Partial Solutions*. We shall clarify in Sec-



**Figure 2.6.** Steps of the *BMT* algorithm. (a) output of phase (i): partitioned program, and a set of mappings for each each of them; (b) output of phase (ii): one mapping for each program partition.

tion 2.3.1 the meaning of these parameters. They let us control the size of the space of solutions that we search in order to solve qubit allocation.

*BMT* consists of three different phases, which Figure 2.6 highlights: (i) qubit allocation is partitioned into multiple instances of subgraph isomorphism, and each instance is independently solved; (ii) all combinations of isomorphisms are evaluated via a dynamic programming model; (iii) a final program is produced out of the best combination of isomorphisms, via token swapping. The following sections discuss details of the three steps enumerated in Figure 2.6.

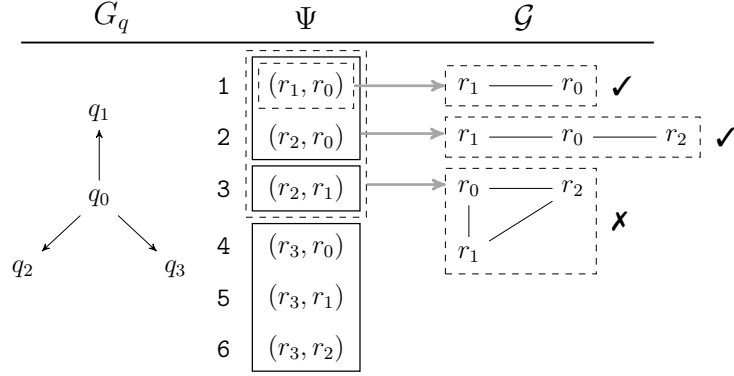
### 2.3.1 Program Partitioning via Subgraph Isomorphisms

The first phase of *BMT* splits the list of control relations into multiple partitions, and maps the pseudo qubits in each of these subsequences into physical qubits. This phase relies on the notion of Maximal Isomorphic Sublists, which Definition 8 introduces, and Example 14 illustrates.

**Definition 8.** [*Maximal Isomorphic Sublist - MIS*] Given a list of control relations  $\Psi$ , plus an (undirected) coupling graph  $G_q^u$ , we say that  $\Psi(i, j)$  is a Maximal Isomorphic Sublist if, and only if,  $\mathcal{G}_{\Psi(i, j)} \lesssim G_q^u$ , and  $\mathcal{G}_{\Psi(i, j+1)} \not\lesssim G_q^u$  or  $\Psi(j)$  is the last control relation. For simplicity, we shall refer to the sequence of Maximal Isomorphic Sublists of  $\Psi$  as  $\mathcal{S}(\Psi) = \{\Psi_1, \dots, \Psi_n\}$ .

**Example 14.** Figure 2.7 shows a coupling graph  $G_q^d$  and a list of control relations  $\Psi$ . The derived graphs  $\mathcal{G}_{\Psi(1,1)}$  and  $\mathcal{G}_{\Psi(1,2)}$  for  $\Psi(1,1)$  and  $\Psi(1,2)$  can be embedded into the undirected version of the coupling graph, i.e.,  $G_q^u$ . However,  $\mathcal{G}_{\Psi(1,3)} \not\lesssim G_q^u$ . Thus,  $\Psi(1,2)$  is a maximal isomorphic sublist.

The concept of maximal isomorphic sublist gives origin to the decision problem that we must solve in this phase of our algorithm. We state this problem below.



**Figure 2.7.** From left to right, we have the coupling graph  $G_q^d$ , the list of control relations  $\Psi$ , split into Maximal Isomorphic Sublists (solid boxes), and the graphs  $\mathcal{G}_\Psi$  derived from sublists of  $\Psi$  (dashed boxes). Next to each derived graph, we show if an isomorphism is possible ( $\checkmark$ ) or not ( $\times$ ).

**Definition 9** (Partitioning of Control Relations – PCR). **Input:** an (undirected) coupling graph  $G_q^u$ , a list  $\Psi$  of  $n$  control relations and an integer  $k$ ,  $k \leq n$ . **Output:** a sequence  $S$  of  $k$  partitions  $\Psi(1, i_1), \Psi(i_1 + 1, i_2), \dots, \Psi(i_{k-1} + 1, i_k)$ , such that for any  $\Psi(x, y) \in S$ , we have that  $\mathcal{G}_{\Psi(x, y)} \lesssim G_q^u$ .

**Solving PCR.** We solve PCR via an exhaustive recursive function  $S_{pcr}$ , which generates every possible sublists of  $\Psi$ . Given a list  $\Psi(1, n)$  of  $n$  control relations, let us assume that the sublist  $\Psi(1, i)$  has already been split into  $k'$  partitions,  $k' < k$ . Thus, we need to split  $\Psi(i + 1, n)$  into  $k - k'$  partitions. We shall find the largest prefix of  $\Psi(i + 1, n)$  that gives us a maximal isomorphic sublist.

To this end, we start with an empty mapping  $f_\emptyset = \{\}$ , i.e., the function that maps every pseudo qubit to an undefined physical qubit  $\perp$ . We then update  $f$  successively for each instruction in  $\Psi(i + 1, n)$ , until it is no longer possible. To implement  $S_{pcr}$ , we notice that, given a mapping  $f$ , which accounts for the  $x - 1$  instructions in the sequence  $\Psi(i, i + x - 1)$ , plus the next instruction  $\Psi(i + x)$ , only three actions are possible. To describe these three actions, we consider that  $\Psi(i + x) = (p_1, p_2)$ . A physical qubit that does not belong into the image of  $f$  is a *free qubit*. An edge in the coupling graph formed by two free qubits is a *free edge*:

- if  $f(p_1) = \perp$  and  $f(p_2) = \perp$ , then, for every free edge  $(q_1, q_2) \in E(G_q^u)$ , we create a new mapping  $f' = f \cup \{p_1 \mapsto q_1, p_2 \mapsto q_2\}$ , and call  $S_{pcr}$  recursively for every  $f'$  and  $\Psi(i + x + 1, n)$ .
- if  $f(p_1) = \perp$  and  $f(p_2) \neq \perp$  (or  $f(p_1) \neq \perp$  and  $f(p_2) = \perp$ ), then only one of the pseudo qubits needs to be mapped. Without loss of generality, let us assume

that  $f(p_1) \neq \perp$  and  $f(p_2) = \perp$ . For every  $(f(p_1), q_2) \in E(G_q^u)$ , such that  $q_2$  is free, we create a new mapping  $f' = f \cup \{p_2 \mapsto q_2\}$ , and continue recursively for every  $f'$  and the remaining list.

- if  $f(p_1) \neq \perp$ ,  $f(p_2) \neq \perp$  and  $(f(p_1), f(p_2)) \in E(G_q^u)$ , then no update is necessary. We continue recursively on  $f, \Psi(i + x + 1, n)$ .

If none of these three actions is possible, then  $\Psi(i, i + x - 1)$  defines another partition, being a maximal isomorphic sublist. In this case, we create a set of mappings  $F_{k'+1}$  containing all mappings that satisfy  $\mathcal{G}_{\Psi(i, i+x-1)} \lesssim G_q^u$ , and invoke  $S_{pcr}$  over the remaining list, this time with an empty mapping.

**The cost of a maximal isomorphic sublist.** Function  $S_{pcr}$  creates a sequence of sets of mappings  $F_1, F_2, \dots, F_m, m \leq k$ . These sets range over the undirected version of the coupling graph; however, the final product of our qubit allocation must be assigned to the actual, i.e., directed, version. Any mapping onto  $G_q^u$  can be adjusted onto  $G_q^d$ , because we can use *reversals*, a quantum transformation introduced by Example 5, to invert the semantics of an edge in the coupling graph. Yet, each reversal has a *constant cost*. Given a mapping  $f \in F_k$  ranging over the sublist  $\Psi(i, j)$ , we define its cost as the sum of the costs over each individual instruction  $\Psi(x) \in \Psi(i, j)$ . If  $\Psi(i) = (p_1, p_2)$ , then this individual cost is given by the necessity to apply an inversion to implement the edge  $(f(p_1), f(p_2))$ . Definition 10 formalizes this cost function:

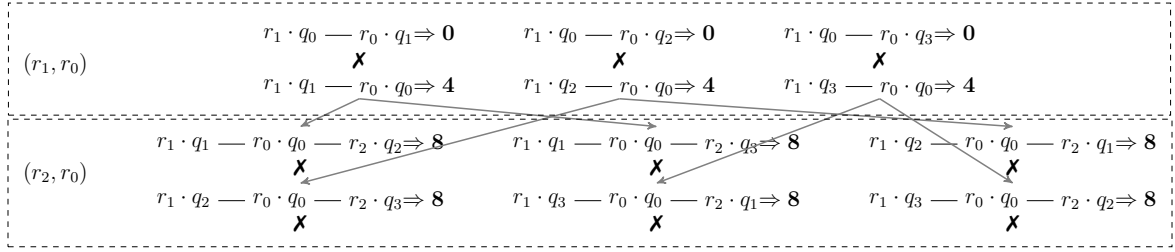
**Definition 10** (Cost of Mapping). *The cost function for each mapping  $C_\Psi : \mathcal{F} \rightarrow \mathbb{R}$  is defined below:*

$$C(f) = \sum_{(p_1, p_2) \in \Psi} \begin{cases} 0 & \text{if } (f(p_1), f(p_2)) \in E(G_q^d) \\ C_{rev} & \text{else} \end{cases}$$

**Example 15.** *Given the coupling graph  $G_q^u$  and the list  $\Psi$  shown in Figure 2.7, we computed the set of mappings for the first two instructions. Figure 2.8 illustrate these generated mappings for each one of the instructions that compose the first partition, as well as the cost for each one of them. Considering  $C_{rev} = 4$ , below, we describe the steps we used for each instruction:*

1. *Process  $(r_1, r_0)$ : both,  $r_1$  and  $r_0$ , are mapped to  $\perp$ , and all the edges in the coupling graph are free. Thus, we can map  $(r_1, r_0)$  onto any edge of that graph. There exists*





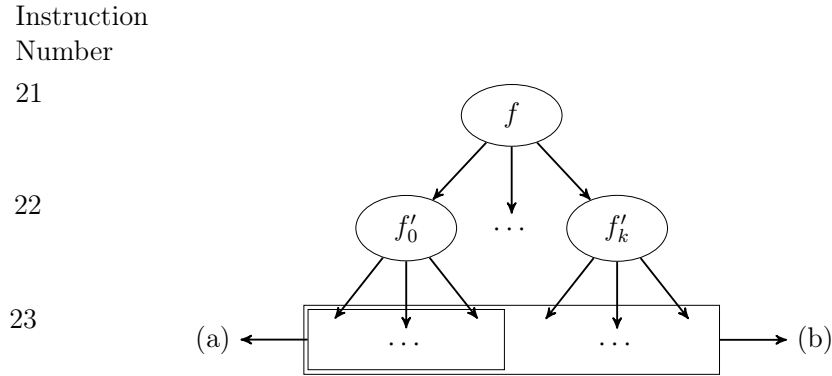
**Figure 2.8.** Exhaustive mapping tree produced from the first instruction  $(r_1, r_0)$  in our running example. The notation  $p \cdot q$  indicates that pseudo qubit  $p$  is mapped onto physical qubit  $q$ . Mappings marked with  $\mathbf{X}$  are dead-ends, i.e., we cannot continue the exhaustive construction of new mappings from them. We show the cost of each mapping next to it.

*six possible mappings. Cost:* since  $q_0$  has only outgoing edges, whenever  $r_1$  is not mapped to  $q_0$ , we have a cost of 4;

2. *Process  $(r_2, r_0)$ :*  $r_0$  was mapped in the previous step; hence, we need to allocate  $r_2$ . Each one of the six mappings of the previous step yields different possibilities for  $r_2$ . For example, the mapping  $\{r_1 \mapsto q_1, r_0 \mapsto q_0\}$  gives us two possible locations for  $r_2$  in the coupling graph:  $q_2$  or  $q_3$ . On the other hand, some mappings found in the previous step are dead-ends. For instance,  $\{r_1 \mapsto q_0, r_0 \mapsto q_1\}$  leaves no vertex for  $r_2$ , because the only neighbour of  $q_1$  in the coupling graph is exactly  $q_0$ , which was already taken by  $r_0$ . **Cost:** we sum the cost of the parent mapping with the cost of using the edge  $(f(r_2), f(r_0))$ ;
3. *Process  $(r_2, r_1)$ :* adding this instruction to the sequence  $[(r_1, r_0), (r_2, r_0)]$  makes it impossible to find a valid subgraph in  $G_q^u$ . Hence,  $\Psi(1, 2)$  is a maximal isomorphic subgraph, and to map  $(r_2, r_1)$  we must start afresh.

**Dealing with Combinatorial Explosion.** Function  $S_{per}$  is exponential, and becomes quickly unpractical as its input grows. To mitigate this problem, we bound the number of mappings via two parameters: (i) the *maximum number of children mappings* and (ii) the *maximum size of the set of current mappings*. The first parameter controls how many searches we are allowed to perform from each mapping. The other limits the number of mappings that we can consider. Figure 2.9 illustrates these two forms of pruning: 2.9(a) refers to (i); and 2.9(b) refers to (ii).

In the end of this first phase, we have the input program sliced into up to  $k$  Maximal Isomorphic Sublists  $\mathcal{S} = \Psi(1, i_1), \Psi(i_1 + 1, i_2), \dots, \Psi(i_{k-1} + 1, i_k)$ , plus the



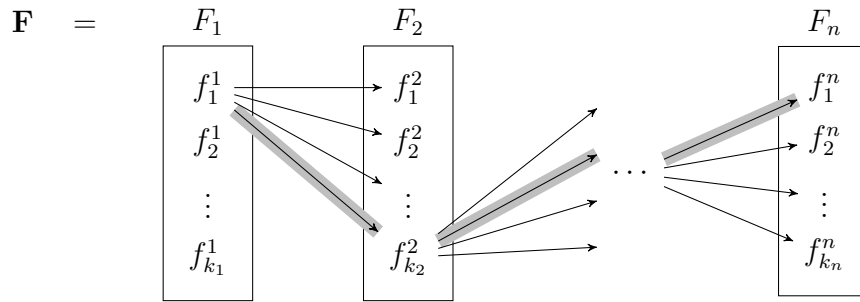
**Figure 2.9.** Tree of mappings for one partition. The  $i^{\text{th}}$  level represents the possible mappings once we add the  $i^{\text{th}}$  instruction (on the left) to the mappings already in place. Since the number of leaves grows exponentially, we prune by (a) limiting the number of children of each mapping; and (b) bounding the number of mappings for a partition.

corresponding sets of mappings  $\mathbf{F} = F_1, F_2, \dots, F_n$ . Each  $F_j$  contains multiple ways to map  $\Psi(i_{j-1} + 1, i_j)$  onto  $G_q^u$ . Each partition  $\Psi(x, y)$  gives origin to a derived graph  $\mathcal{G}_{\Psi(x,y)}$  isomorphic to some subgraph of  $G_q^u$ . A byproduct of this phase is the cost of each mapping, which is given by the function  $C_{\Psi}$  from Definition 10.

**Complexity Analysis of the First Phase.** We are generating exhaustively all the mappings that solve the subgraph isomorphism problem. To avoid the exponential complexity, we limit the generation process with two parameters: maximum number of children ( $M_c$ ) and maximum number of partial solutions ( $M_p$ ). Thus, for every instruction, we have to generate  $M_c$  mappings for each of the  $M_p$  partial solutions. Children mappings are created in  $O(|Q|)$ . Therefore, the time complexity of the first phase is  $O(M_c M_p |\Psi| |Q|)$ . Since we keep up to  $M_p$  mappings for each partition (which cannot be greater than  $|\Psi|$ ), and each mapping takes  $O(|Q|)$  space, the space complexity is  $O(M_p |\Psi| |Q|)$ .

### 2.3.2 Combining Mappings via Token Swapping

In Section 2.3.1 we have split the list of control relations into multiple partitions, and created a set of mappings for each one of them. Now, we need to connect these partitions, adapting, via swaps, one mapping into another. Figure 2.10 illustrates this idea: for each partition we have a collection of candidate mappings  $F$ . We must find a path from some mapping  $f_i \in F_1$  to some mapping  $f_n \in F_n$  which minimizes the cost of implementing the quantum program. Each path consists of  $n - 1$  hops, where



**Figure 2.10.** Output from first phase  $\mathbf{F}$  with some possible combinations represented by different paths. Each path in this figure represents one different solution. Highlighted, there is a path that represents the optimal solution.

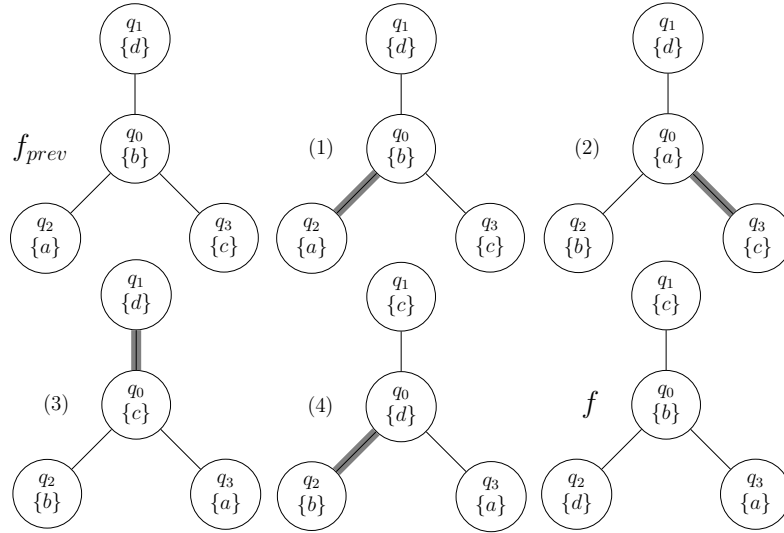
a hop is a way to transform  $f_i \in F_i$  into  $f_{i+1} \in F_{i+1}$ . This transformation is equivalent to Token Swapping, an NP-complete problem [Bonnet et al., 2016; Kawahara et al., 2017].

Between two successive sets of candidates, e.g.,  $F_i$  and  $F_{i+1}$ , there exist  $|F_i| \times |F_{i+1}|$  possible paths. Thus, there exists a potentially exponential number of paths between  $F_1$  and  $F_n$ . In other words, to find the optimal path illustrated in Figure 2.10, we must deal with an NP-complete problem, which would have to be solved an exponential number of times! Fortunately, there are ways to approximate Token-Swapping [Miltzow et al., 2016], as we discuss in Section 2.3.2.1; and we can handle the combinatorial explosion of paths via dynamic programming, as we explain in Section 2.3.2.2.

### 2.3.2.1 Solving the Token Swapping Problem

Recently, Miltzow et al. [2016] presented a 4-approximative solution to the Token-Swapping Problem (see Section 4.4). This algorithm is at least cubic on the size of the coupling graph, e.g.,  $O(|Q|^3)$ ; however, we must still run it for at least  $\min |F_i|^2 \times \#Partitions$  – a task that becomes impractical even for small settings. Fortunately, Miltzow et al. [2016] also gave us the necessary equipment to avoid this effort. Thus, instead of finding approximations for every instance of the Token-Swapping Problem, we only estimate the cost of each one of these problems (without providing an actual solution to it). We use the function  $\delta$  from Definition 11 to find such estimates. As noted in Miltzow et. al., the number of swaps – henceforth denoted by  $|\Delta(f_{prev}, f)|$  – is less or equal  $2 \times \delta(f_{freq}, f)$ . Thus, we use this upper bound as the estimation of the number of swap operations. Example 16 illustrates this estimate.

**Definition 11** (Cost of joining two successive mappings). *Let  $d : Q \times Q \rightarrow \mathbb{N}$  be a function that yields the minimum number of edges between two vertices in the coupling*



**Figure 2.11.** Steps for transforming  $f_{prev}$  into  $f$ , assuming the coupling graph seen in Figure 2.7. The pseudo qubit assigned to a physical qubit is shown in brackets. Gray edges indicate qubits that shall be swapped.

graph:  $f(a)$  and  $f_{prev}(a)$ . We define  $\delta$  as follows:

$$\delta(f_{prev}, f) = \sum_{p \in P, f_{prev}(p) \neq \perp} d(f_{prev}(p), f(p))$$

**Example 16.** Figure 2.11 shows the sequence of swap operations that transform  $f_{prev} = \{a \mapsto q_2, b \mapsto q_0, c \mapsto q_3, d \mapsto q_1\}$  into  $f = \{a \mapsto q_3, b \mapsto q_0, c \mapsto q_1, d \mapsto q_2\}$ , using the architecture from Figure 2.7. The  $\delta$  function gives us an estimate, not the best solution for the Token-Swapping Problem. In this example,  $\delta(f_{prev}, f) = d(q_2, q_3) + d(q_0, q_0) + d(q_3, q_1) + d(q_1, q_2) = 6$ , yet the optimal swap sequence between  $f_{prev}$  and  $f$  has 4 swaps.

**Ensuring that live qubits remain mapped.** If a pseudo qubit  $p$  appears for the first time in a control relation  $\Psi(i)$ , and for the last time in a control relation  $\Psi(j)$ , we say that  $p$  is *alive* at  $\Psi(i, j)$ . If a pseudo qubit is alive at  $\Psi(i, j)$ , then it must be allocated onto some physical qubit in every mapping  $f$  that refers to some partition that contains instructions from  $\Psi(i, j)$ . If that were not the case, we would produce an incorrect quantum circuit, which might “overwrite” qubits still in use. However, this hazard would naturally happen if some partition  $\Psi(i', j') \subset \Psi(i, j)$  does not contain any reference to  $p$ .

**Example 17** (Liveness). Pseudo qubit  $r_0$  is alive at the second partition seen in Figure 2.7; however, this partition only contains instruction  $\Psi(3) = (r_2, r_1)$ . A solution

to Problem 9 will set  $f(r_0) = \perp$  at partition 2. Yet,  $r_0$  shall be necessary in the third partition; hence, it must be propagated from the first mapping to the third.

To prevent this kind of situation, we ensure that qubits are allocated at every partition where they are alive. To explain how we perform it, let us assume that  $p$  is alive at  $\Psi(i', j')$ , but is not used within that partition. Let  $f'$  be a mapping for  $\Psi(i', j')$ , where  $f'(p) = \perp$ . We assume that  $f_{prev}$  is the mapping for the previous partition, and that  $f_{prev}(p) = q$ . We can safely assume that  $p$  is mapped by  $f_{prev}$  by an inductive argument:  $p$  is mapped at the first partition where it is used, and we shall propagate it along other mappings, until the last partition that uses it. To ensure that  $p$ 's image is defined at  $f'$ , we let  $f'(p) = q'$ , where  $q'$  is the unmapped qubit that is the closest to  $f_{prev}(p)$ . To see that  $q'$  exists, notice that there is more pseudo qubits than physical qubits, and a pseudo qubit is never mapped onto two different physical qubits in the same partition.

### 2.3.2.2 Dynamic Programming

The approximations discussed in Section 2.3.2.1 gives us the means to calculate the cost of transforming one mapping into another, thus bridging two successive partitions of  $\Psi$ . Yet, we must find one such path between every pair of successive partitions, as Figure 2.10 illustrates. As we have already discussed, the number of paths is exponentially in terms of the number of partitions. However, the problem of finding an optimal path admits an exact solution in polynomial time, via a dynamic programming algorithm. To explain how this algorithm works, we first introduce the problem that it solves:

**Definition 12** (Construction of a Complete Sequence of Transformations). *Input:* a sequence of  $n$  sets of mappings of pseudo to physical qubits  $F_1, F_2, \dots, F_n$ , the function  $C_{\Psi_i}$  from Definition 10 and the function  $\delta$  from Definition 11. *output:* a sequence  $f_1, f_2, \dots, f_n$ ,  $f_i \in F_i$ , which minimizes  $\sum \delta(f_{i-1}, f_i) + \sum C_{\Psi}(f_i)$ .

Problem 12 has *optimal substructure*. In other words, it can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to each sub-problem. Problems with such property admit exact solution via dynamic programming [Bellman, 2003]. Definition 13 describes the dynamic programming sub-problem, and Equation 2.1 shows its recurrence relation. In Theorem 3 we state and prove the correctness of our solution. Finally, Example 18 illustrates how we solve Problem 12.

**Definition 13.** (Subproblem) Our subproblem  $OPT(i, j)$  represents the optimal cost for allocating all control relations until the  $i$ -th partition, while using  $f_j^i$  (the  $j$ -th mapping that satisfies the subgraph isomorphism relation between  $\mathcal{G}_{\Psi_i}$  and  $G_q^u$ ) as the last mapping. i.e. it is the minimum cost using the  $j$ -th mapping generated for the  $i$ -th partition.

$$OPT(i, j) = \begin{cases} C_{\Psi_i}(f_j^i) & i = 1 \\ \min_{0 \leq k < |F_{i-1}|} (\delta(f_k^{i-1}, f_j^i) + OPT(i-1, k)) \\ \quad + C_{\Psi_i}(f_j^i) & i > 1 \end{cases} \quad (2.1)$$

**Theorem 3.** The recurrence relation given by Equation 2.1 yields an optimal solution to Problem 12.

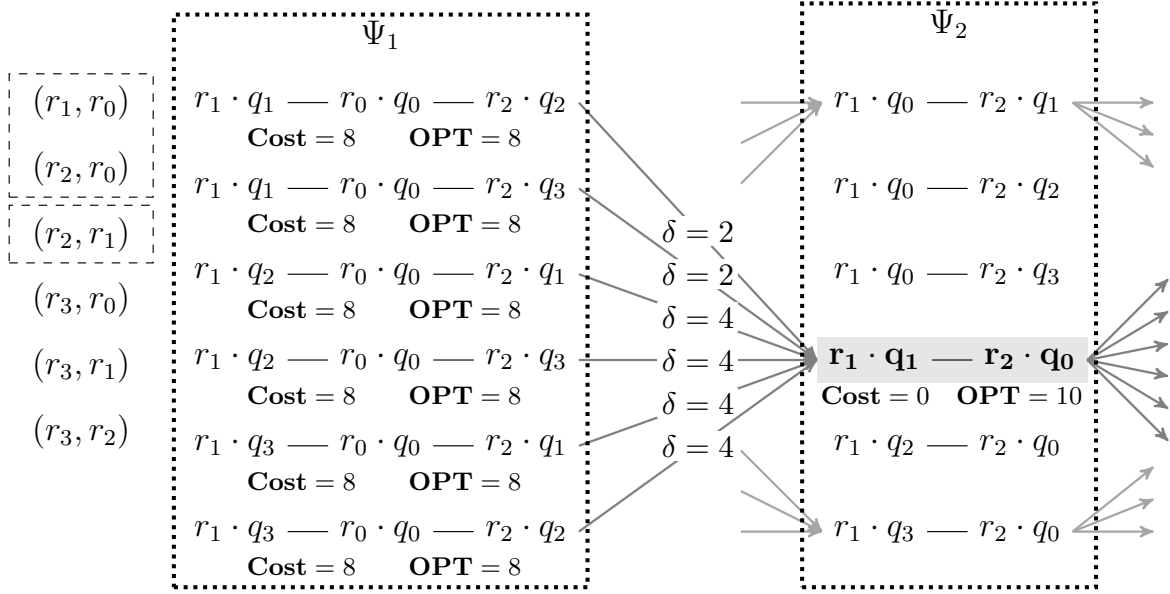
**Proof.** We shall prove by induction on the partitions:

1. **Base Case:**  $i = 1$  (there is only one partition). Since we have only one partition, and we are allocating it with  $f_j^i$  by definition, that is the optimal cost;
2. **Inductive Case:**  $i > 1$ . Suppose  $OPT(i, j)$  is not optimal. Since it is not optimal, there must exist another mapping  $f_{k'}^{i-1}$  from the previous partition  $i - 1$  such that we profit more when transforming  $f_{k'}^{i-1}$  into  $f_j^i$ . Thus,  $\delta(f_{k'}^{i-1}, f_j^i) + OPT(i-1, k') < OPT(i, j) = \min_k \delta(f_k^{i-1}, f_j^i) + OPT(i-1, k)$  must be true, a contradiction.

□

**Example 18.** Figure 2.12 shows how we test all combinations of mappings for the first two program partitions. Calculating  $OPT(1, j)$  for any  $1 \leq j \leq |F_1|$  is trivial (base case). For the other case, given the recurrence relation, we have to get the minimum value given by the sum of the previous subproblems' optimal solution, plus the estimated number of swap operations for transforming one mapping into another. In this case, every subproblem has an optimal cost of 8; hence, we pick the solution with the smallest estimated transformation cost  $\delta = 2$ . We repeat the process for every  $f \in F_2$ .

**Complexity Analysis of the Second Phase.** In the worst case, we have  $|\Psi|$  partitions, each formed by one instruction. Each partitions gives us up to  $M_p$  mappings. For each mapping, we have to find the minimum cost between all the  $M_p$  previous mappings, according to Equation 2.1. The estimation of the swap cost takes  $O(|Q|)$ ,



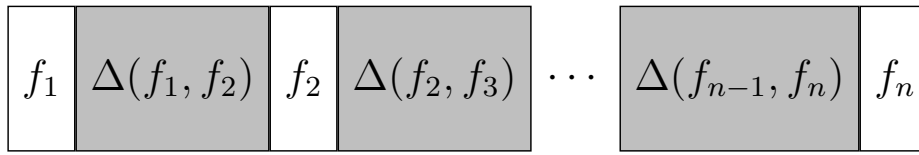
**Figure 2.12.** Subproblem dependency for calculating  $OPT$  of the highlighted mapping. It shall be the minimum value of the sum of each dependency by its cost  $\delta$  of transforming the previous one into the highlighted one.

while the cost for ensuring live qubits remain mapped is  $O(|Q|(|Q| + E(G_q^u)))$ , since we execute, in the worst case, one BFS for each qubit. Hence, the time complexity of this phase is  $O((M_p)^2|\Psi||Q|(|Q| + E(G_q^u)))$ . The space complexity is  $O(M_p|\Psi||Q|)$ : each subproblem is  $O(|Q|)$ , and we have  $O(M_p|\Psi|)$  of them.

### 2.3.3 Code Generation

The dynamic programming algorithm discussed in Section 2.3.2.2 gives us a sequence of mappings  $f_1, f_2, \dots, f_n$ , which shall guide us through the process of building a concrete program out of a virtual quantum circuit. Each mapping corresponds to a partition of  $\Psi$ . Let  $f_1$  be the mapping for  $\Psi(i, j)$ . Mapping  $f$  gives us the information necessary to allocate all the virtuals used between the first control instruction, e.g.,  $\Psi(i)$  and the last, e.g.,  $\Psi(j)$ . After  $\Psi(j)$ , a new partition,  $\Psi(j + i, k)$ , starts. Let us assume that the mapping that corresponds to this partition is  $f_2$ . We need to create sequences of swaps linking  $f_1$  and  $f_2$ . We shall use the term  $\Delta(f_1, f_2)$  to denote this sequence. Figure 2.13 shows the product of this phase.

**From estimates ( $\delta$ ) to concrete sequences ( $\Delta$ ).** In Section 2.3.2 we used a heuristic, the  $\delta$  function, to over-estimate the quantity of swapping operations necessary to link consecutive mappings. To generate code, we replace this function with the



**Figure 2.13.** Mappings  $f$  and swapping sequences  $\Delta$  (highlighted) gives us all the information that is necessary to transform a virtual quantum circuit into a physical quantum circuit.

4-approximative algorithm used by Miltzow et al. [2016] to solve the Colored Token Swapping Problem (Section 4.4.1). The  $\delta$  function is an over-approximation of Miltzow et al. [2016]’s algorithm. Thus, the actual cost of the sequence of swaps  $\Delta$  that links two successive mappings might be lower than the cost found through  $\delta$ .

**Dealing with partially defined mapping functions.** Miltzow et al. [2016]’s approximation receives two mappings,  $f_{prev}$  and  $f$ , and finds a sequence of swaps that transform  $f_{prev}$  into  $f$ . In the original formulation of their algorithm, Miltzow et al. [2016] assume that  $f_{prev}$  and  $f$  are permutations, i.e., functions with the same domain and image. However, in our case, these mappings do not necessarily enjoy this property, as the live ranges of virtual qubits are not always the same. In other words, we must account for any virtual qubit  $p$  such that  $f_{prev}(p) = \perp$  and  $f(p) \neq \perp$ . We recall that we do not need to consider the possibility of  $f_{prev}(p) \neq \perp$  and  $f(p) = \perp$ . This case will never happen, because, as discussed in Section 2.3.2.1, we ensure that live qubits remain always mapped. To make provision to partially defined mappings, we introduce a projection operator  $\nabla$ :

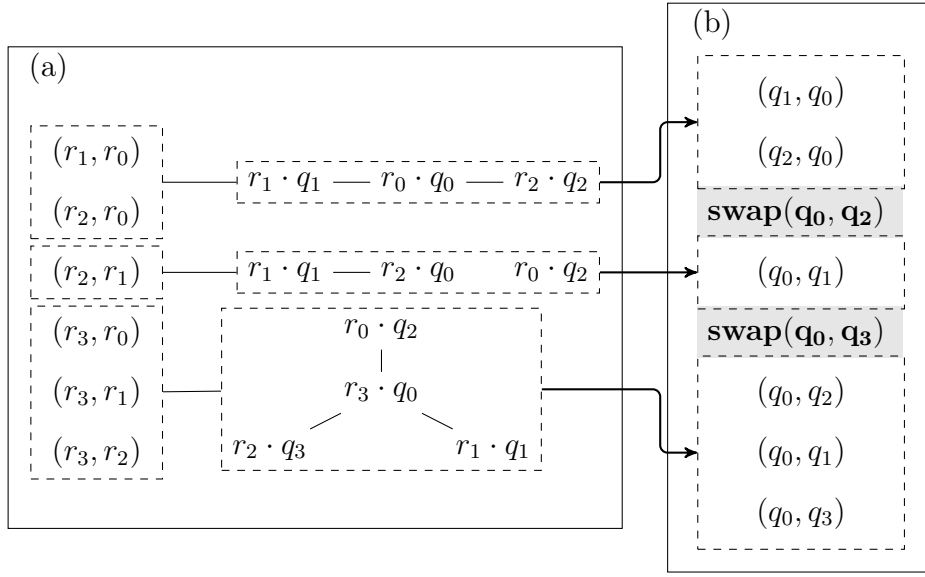
$$(f_{prev} \nabla f)(p) = \begin{cases} q & f(p) = q \text{ and } f_{prev}(p) \neq \perp \\ \perp & f(p) = q \text{ and } f_{prev}(p) = \perp \end{cases}$$

Instead of solving token swapping between  $f_{prev}$  and  $f$ , we solve it between  $f_{prev}$  and  $f_{prev} \nabla f$ . In other words, we solve the problem only for virtual qubits which are defined in both mappings. Lemma 1 shows that this approach is sound. We discuss later the approximation factor of this optimization in Chapter A. Example 19 illustrates this phase with the input program used in the previous sections.

**Lemma 1.** *Let  $undef(f) = \{p \mid f(p) = \perp, p \in P\}$ . Given the mapping  $f' = f_{prev} \nabla f$ , if the set  $undef(f_{prev}) \supseteq undef(f)$ , then  $\Delta(f_{prev}, f')$  is the minimum swap sequence we can get for  $\Delta(f_{prev}, f)$ .*

**Proof.** Since  $undef(f_{prev}) \supseteq undef(f)$ , all pseudo-qubits mapped to a





**Figure 2.14.** (a) from left to right, we have the whole input program segmented into partitions (dashed box), and their respective mappings; (b) the translated input program segmented into partitions (dashed box), and the swap operations (highlighted) necessary to transform one mapping into another.

physical-qubit  $q$ , given that  $q \neq \perp$ , in  $f_{prev}$  are also defined in  $f$ . Thus, we have to allocate correctly, at least, these pseudo-qubits that are not in  $undef(f_{prev})$ . That is exactly what  $f'$  is: a mapping of the pseudo-qubits mapped in  $f_{prev}$  to their respective physical-qubits mapped in  $f$ . Summarizing, after finding the swaps to transform  $f_{prev}$  into  $f'$ , the only thing that is left to be done is to map the qubits in  $undef(f_{prev}) \setminus undef(f)$  to their corresponding locations. It is straight-forward to observe that those locations are free, since otherwise, there would be two pseudo qubits mapped to the same physical qubit.

**Example 19.** Figure 2.14 (a) shows the mappings selected for each partition of the program. Notice that the second mapping  $r_1 \mapsto q_1, r_2 \mapsto q_0, r_0 \mapsto q_2$  is well-defined for  $r_0$ , although this pseudo qubit is not used in the second partition, for the reasons that we have discussed in Example 17. From these mappings, we are able to calculate the  $\Delta$  function for each pair of consecutive partition. Figure 2.14 (b) shows the output generated in the end. All the instructions (Input Control Relations) are translated into physical-qubits, and swapping operations are used to bridge differences between consecutive mappings.

**Complexity Analysis of the Third Phase.** The algorithm given by Miltzow et al. [2016] is time-wise expensive. That is because one of its steps is composed by the

Hungarian Algorithm for minimum matching [Kuhn, 1955] ( $O(|Q|^3)$ ). Besides that, the algorithm's main loop will execute  $O(|Q|)$  breadth first searches (BFS) for each swap. Thus, the time complexity of the approximative algorithm is  $O(|Q|^3 + |\Delta||Q|(|Q| + E(G_q^u)))$ .  $|\Delta|$  is bounded by the sum of the distance of the misplaced qubits. Miltzow's algorithm run once per partition ( $|\Psi| - 1$  times). Thus, the time complexity of this phase is  $O(|\Psi|(|Q|^3 + |\Delta||Q|(|Q| + E(G_q^u))))$ . The space complexity is the product of the number of mappings for each partition,  $O(|\Psi||Q|)$ , and the space-complexity of the approximative algorithm  $O(|Q|^2)$ .

## 2.4 Improving *BMT*

In its canonical form, *BMT* is full of simple heuristics that results in suboptimal solutions. There is a lot of room for improvement in it. Below we list some dimensions that we analyzed for improving our solution, indicating also the phase where it applies:

**(Phase - Section 2.3.1) Iterating the Quantum Program.** Instructions in a quantum circuit do not have to be processed in the order defined by the program (list of dependencies). Instead, *quantum gates over non-overlapping qubits may be executed in parallel*. It can be easily observed in the gate dependency graph (Section 1.2). A topological ordering of this graph determines the valid ways to iterate over the program. Whenever multiple orderings are possible, we rank those gates and select the next gate to be processed. To explain the predicates for ranking the gates, suppose that gates  $g_1, \dots, g_n$  can be processed, and that the pseudo-qubits used in a candidate gate  $g_i$  are  $(a, b)$ . We rank these gates in these four different categories (from higher to lower priority):

1. ***a* and *b* are mapped to adjacent physical-qubits;**
2. **one** of *a* or *b* is already mapped;
3. **none** of *a* and *b* are mapped;
4. **both** *a* and *b* are mapped.

**(Phase - Section 2.3.1) Pruning and Selecting Mappings.** There are many methods for selecting a few solutions among a population of them. In the canonical form of the algorithm, we used a simple selection that kept the first  $K$  out of the  $N$  possible solutions. As an improvement, we may use, for example, the weighted roulette

selection, the tournament selection, etc. As mentioned in the last section (Section 2.3), the pruning and selection happens in two dimensions (the two parameters of the algorithm): the maximum number of children ( $M_c$ ); and the maximum number of partial solutions ( $M_p$ ).

**(Phase II - Section 2.3.2) Estimating the Cost.** Our current cost estimation consists in the sum of the distances between the mapped qubits. i.e. given two mappings  $f, f'$ , and a coupling graph  $G$ :

$$\sum_a dist(f(a), f'(a)) \quad \text{for all } a \in P \quad \text{and} \quad f(a) \neq \perp$$

Increasing the accuracy of this estimation, could yield in better solutions. However, the optimal solution is shown to be NP-hard, and the time-complexity of the 4-approximative solution slows down the whole *BMT* algorithm significantly.



# Chapter 3

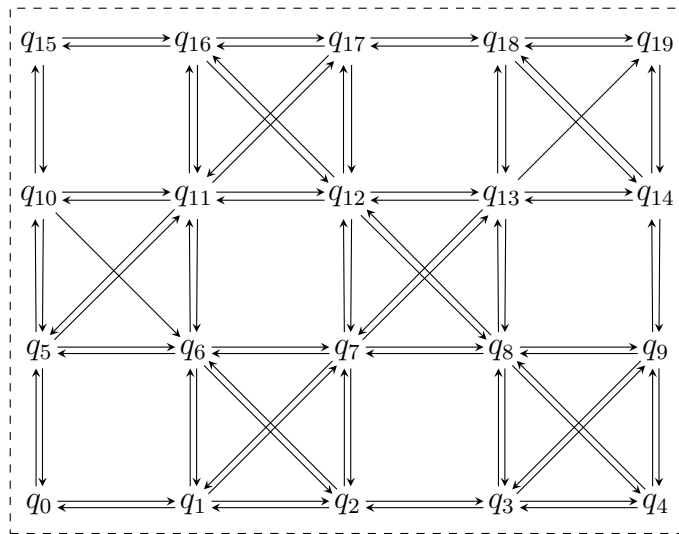
## Evaluation

In this chapter, we shall answer a number of research questions (more details in Section 3.3). To this end, we first describe the overall setup and runtime environment where our experiments were executed. Since there are a number of research questions, we devised different experiments to validate them. Therefore, assume the default (presented here), unless specified otherwise.

**The State-of-the-Art Algorithms:** We compare *BMT* against four other algorithms: IBM [2018b] Python SDK Terra (`ibm`); Zulehner et al. [2018] A\* (`jku`); Zulehner and Wille [2018] Winner of the IBM Developer Challenge (`chw`); Li et al. [2018] SWAP-based BidiREctional heuristic (`sbr`); and Siraichi et al. [2018] algorithms. Among them, there are: the exact dynamic programming algorithm (`dyn`); the weighted partial mapper (`wpm`); and two different configurations of the bounded mapping tree. One with smaller parameters, and fast (`bmtF`); and another one slow, but with a better performance (`bmtS`).

Towards a fairer comparison on the “efficiency” of the allocators, we implemented all of the competitors in our C++ OpenQASM open source compiler *Enfield*. Since all of them have their code publicly available, we based ourselves on those to complete our implementation. That said, we compared our results with the original sources, and we obtained equal results. Our implementations were faster and used less memory, enabling us to allocate all of our benchmarks with them.

**Runtime Environment.** All tests were executed on an Intel(R) Xeon(TM) E5-2660 CPU @ 2.20GHz, with up to 32G RAM, running Linux Debian Jessie 8.11.



**Figure 3.1.** Coupling graph of the IBM Tokyo, a 20-qubit architecture [IBM, 2018a].

**Benchmarks.** We used the same benchmarks evaluated by Zulehner et al. [2018]. These 158 programs were taken from the RevLib collection [Wille et al., 2008], Quipper [Green et al., 2013], and ScaffCC compiler [Javadi-Abhari et al., 2014]. These suites are staple in papers on Qubit Allocation [Shafaei et al., 2014; Lin et al., 2015; Pedram and Shafaei, 2016; Lao et al., 2018; Lin et al., 2018; Zulehner et al., 2018; Li et al., 2018].

**Quantum Architectures.** In this set of experiments, we used the biggest quantum architecture made available by *IBM*: Tokyo (not public, though). Figure 3.1 illustrates the coupling graph.

### 3.1 Evaluation Metrics

We shall evaluate the allocation algorithms in terms of efficiency and quality of the output program allocated. Most of them are metrics already used in the literature, but others we judge necessary for a fairer comparison. The next two paragraphs shall discuss them.

**Allocation Quality.** The quality of each solution will be evaluated along three dimensions:

- **Weighted Cost:** the combined cost of each gate used in the program. Following common methodology, we let the cost of each CNOT be 10, and of each single-qubit gate be 1. The rationale behind this cost assignment is the fact that CNOT gates have an error rate one order of magnitude larger than single-qubit gates [IBM, 2018a]. Defining only these two costs is enough for all practical purposes, because composite gates may be rewritten as a sequence of single-qubit gates and CNOTs.
- **Gates:** the total number of gates in the allocated program, without distinguishing CNOTs from single-qubit gates. This metric has been adopted in previous work [Shafaei et al., 2014; Shrivastwa et al., 2015; Pedram and Shafaei, 2016; Zulehner et al., 2018; Li et al., 2018].
- **Depth:** the depth is closely related to the time a program needs to terminate. This metric has also being used in previous work [Maslov et al., 2008; Amy et al., 2013; Zulehner et al., 2018].

**Allocation Efficiency.** The efficiency of each solution will be evaluated along two dimensions:

- **Memory Consumption:** although not a main concern, the memory required for search space exploration may grow exponentially, depending on the algorithm [Zulehner et al., 2018]; hence, rendering qubit allocation not scalable.
- **Allocation Time:** similar to memory consumptions, previous work still do not consider allocation time a main concern, because current architectures are small. However, increasing the number of qubits may yield some algorithms impractical.

## 3.2 Statistics Collected

We are always comparing the set of algorithms with some baseline. Assuming that  $R_{base}$  and  $R_{comp}$  correspond to the result of the baseline allocator `base` and the allocator to be compared `comp`, respectively, results are given by the ratio  $R_{comp}/R_{base}$ . Unless stated otherwise, our tables have a standard template illustrated by Figure 3.10. Each algorithm to be compared is represented by one major row. Sub-rows (cost, depth, gates, mem and time) represent dimensions of quality and efficiency. The meaning of each column is given below:

1. **Allocator:** `comp`, i.e. algorithm that we compare against `base`.

2. **Dim:** minor rows that represent the results on different dimensions of efficiency and quality.
3. **G. Mean:** geometric mean of ratios, taking **base** as baseline. The larger this number, the more **base** outperforms its competitor.
4. **G. Std. D.:** the geometric standard deviation of the ratios. The closer to 1, the smaller the spread of the data.
5. **Better (Worse) Count:** the number of benchmarks where **comp** was better (worse) than **base**. The higher (smaller), the better for **base**.
6. **Better (Worse) G. Mean:** the geometric mean of the ratios of the benchmarks where **comp** was better (worse) than **base**. This column answers the following question: “what would be the G. Mean for **comp** allocator, considering only the benchmarks where it performed better (worse) than **base**?”.

### 3.3 Research Questions

In this section, we shall provide answers to the following research questions:

- **[RQ1]:** how do the parameters affect the efficiency and quality of *BMT*?
- **[RQ2]:** how do the optimizations affect the efficiency and quality of *BMT*?
- **[RQ3]:** how does the quality of the programs allocated with our algorithm compare against the state-of-the-art approaches?
- **[RQ4]:** how efficient is our algorithm comparing against the state-of-the-art approaches?
- **[RQ5]:** how far from the exact algorithm are all the allocators?

The following sections shall discuss each of our research questions. All of the experiments shown here were executed five times each. For summarizing the results of these different executions, we used the arithmetic mean, which we shall represent by  $\mu$ . In order to obtain an overall comparison against another algorithm, we used the geometric mean and its corresponding (geometric) standard deviation, which we shall represent by  $\mu_g$ . The equations below illustrate the two means described above:



$$\begin{aligned}\mu(X) &= \sum_{x \in X} \frac{x}{|X|} \\ \mu_g(X) &= \sqrt[|X|]{\prod_{x \in X} x}\end{aligned}$$

### 3.3.1 RQ1: Parameters Effect

In this experiment, we varied the values of our two parameters (Section 2.3.1):  $M_c$  and  $M_p$ . Then, we compiled the whole benchmark set with each combination possible. We used the following value set:

$M_c$	1, 2, 4, 8, 16, 32
$M_p$	10, 20, 40, 80, 160, 320, 640, 1280, 2560

In total, there were six values for  $M_c$ , and nine values for  $M_p$ . Which means that we executed  $|M_c| * |M_p|$  different rounds. i.e. five times each of the 158 benchmarks, times the number of different combinations between  $M_c$  and  $M_p$ . In the end, we ran 42660 experiments.

Figure 3.11 summarizes the results obtained in these experiments. We choose to compare every combination with  $M_c = 8$  and  $M_p = 1280$ , because these parameters showed good trade-off between solution quality and efficiency. We shall refer to each of the combinations as a pair  $\langle M_c, M_p \rangle$  (e.g.  $\langle 8, 1280 \rangle$ ). Below we state the results we got:

- **Quality:**  $\langle 8, 1280 \rangle$  was off by, approximately, 1% of its neighbors, such as  $\langle 4, 640 \rangle$ ,  $\langle 4, 1280 \rangle$ ,  $\langle 8, 640 \rangle$ , and values higher than those. For smaller values of  $M_c$  and  $M_p$ , the weighted cost was at least at a distance of 2% of  $\langle 8, 1280 \rangle$  value. The depth and number of gates followed approximately the same behaviour;
- **Efficiency:** In terms of memory usage, every combination spent around the same amount. It varied from 94% to 114% of  $\langle 8, 1280 \rangle$ .  $\langle 8, 1280 \rangle$  was, on average, two times faster than the highest parameter values. However, the quality-wise competitive  $\langle 4, 1280 \rangle$  was, on average, 20% faster.

**Analysis of results.** As expected, the smaller  $M_c$  and  $M_p$ , the worse the results are. That is because we let the algorithm explore a smaller search space, ignoring partial solutions that may have been beneficial later. However, as we increase these same parameters, the quality stops improving, while the efficiency starts dropping. This fact is easily explainable by considering that there is most good quality mappings were

	10	20	40	80	160	320	640	1280	2560
1	1.82	1.53	1.34	1.22	1.21	1.21	1.21	1.21	1.21
2	1.38	1.19	1.1	1.05	1.02	1.01	1.01	1.01	1.01
4	1.32	1.16	1.08	1.03	1.01	1	1	1	1
8	1.3	1.15	1.08	1.03	1.01	1.01	1	1	1
16	1.3	1.15	1.08	1.03	1.01	1.04	1.01	1	1
32	1.31	1.16	1.09	1.04	1.02	1.04	1.04	1.03	1

**Figure 3.2.** Baseline is  $M_p = 32$  and  $M_c = 2560$   $\langle 32, 2560 \rangle$ . It shows the amount of partitions created by *BMT* in the first phase (Section 2.3.1)

probably already found. Since we are using the full-optimized *BMT*, we discard most high cost mappings that would possibly result in high cost solutions.

The number of partitions of the first phase in *BMT* also plays a great role in the final quality of the solution. That is because the greatest villain that worsens the quality of solutions is the swaps that joins two partitions together. As we decrease the search space with the parameters, we expected that the number of partitions would increase, since we end up not exploring mappings that would enable, for example, a real isomorphism between the program and the coupling graph. Figure 3.2 shows exactly this effect. Obviously, there we can not keep decreasing the number of partitions, since we have the subgraph isomorphism that limits us. That is the reason why we have many combinations (bottom right) that yield the same number of partitions.

There is, however, an interesting result. That is, if we increase  $M_c$ , while keeping  $M_p$  the same, we start generating more partitions. Figure 3.2 corroborates with Figure 3.11 in this observation for the weighted cost. It shows that having a bigger ratio of  $M_c/M_p$  is not so much beneficial, since  $M_p$  may not be able to sample correctly the solutions from  $M_c$ .

### 3.3.2 RQ2: BMT Optimizations Effect

In this experiment, we combined the optimizations described in Section 2.4 in order to check the effectiveness of each of them, related to the canonical algorithm described in the whole of Section 2.3. Then, we compiled the whole benchmark set with each combination possible. We implemented two of the three optimizations described (Section 2.4):

- (i) **Program Iteration:** instead of iterating the program sequentially (leaving the task of serializing to someone else), iterate via some kind of topological sort of the Gate Dependency Graph (Section 1.2). Theoretically, generating less partitions;

- (ii) **Pruning of Mappings:** instead of pruning all of the mappings generated, besides the first ones allowed, weight every mapping by its cost up till this moment. Then, apply an weighted roulette selection. It would generate variety, while biasing the ones with smaller cost.

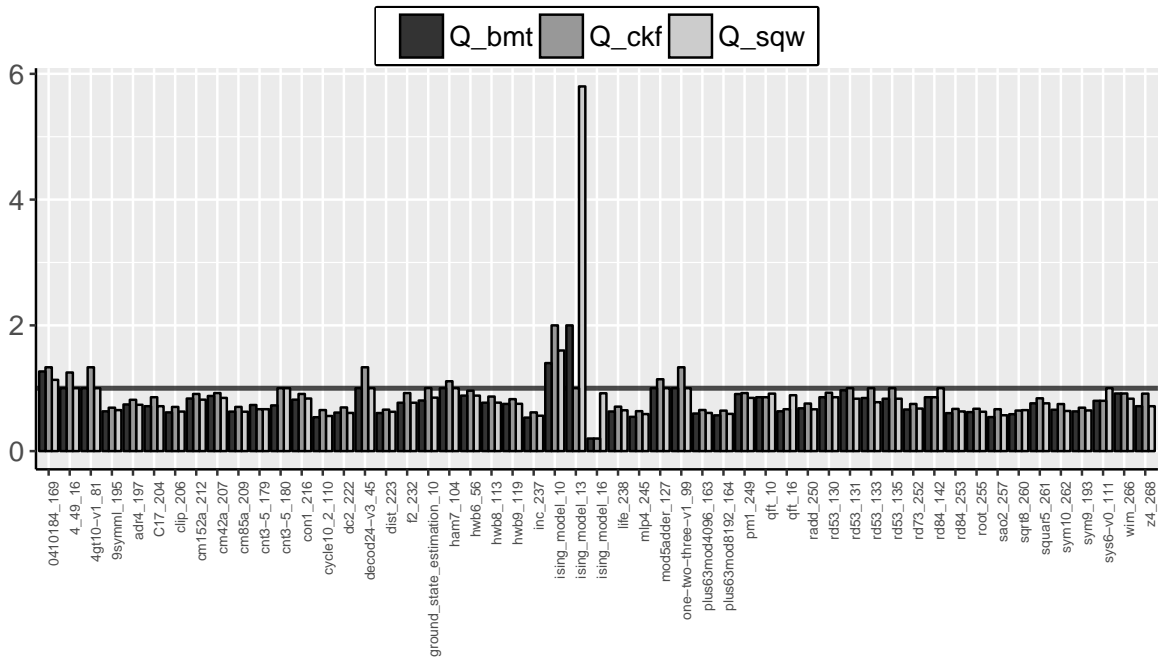
These two optimizations yield four different combinations, i.e. different allocators, as the following table illustrates. Each  $\times$  and  $\bigcirc$  indicates the absence and existence of such optimization, respectively, in the given allocator:

Allocator	(i)	(ii)
<b>can</b>	$\times$	$\times$
<b>ckf</b>	$\bigcirc$	$\times$
<b>sqw</b>	$\times$	$\bigcirc$
<b>bmt</b>	$\bigcirc$	$\bigcirc$

In total, there were four different allocators for this experiment. Which means that we executed four different rounds of experiments. i.e. five times each of the 158 benchmarks, times the number of different combinations of optimizations. In the end, we ran 3160 experiments. Figure 3.12 summarizes the results obtained in these experiments. Taking the last section into account, we set up *BMT* parameters for every allocator as  $M_c = 8$  and  $M_p = 1280 \langle 8, 1280 \rangle$ . Below we state the results we got:

- **Quality:** All of the allocators with optimizations enabled were close quality-wise. In increasing order of ratio of weighted cost between those allocators in relation to **can**, we have: **bmt** with 95.6%; **ckf** with 96.8%; **sqw** with 97.0% of the weighted cost found in the baseline. The ratios for the depth and gates were similar;
- **Efficiency:** Memory was uniform among almost all of the different allocators, except **sqw** which got 2% improvement, against 4% increase in the others. The allocation time taken by **bmt** and **ckf** was worsened by up to 80%. On the other hand, **sqw** improved in 13% of the time taken by the baseline.

**Analysis of results.** We expected that the allocators with the optimization (i) would benefit from it, creating less partitions and the allocators with optimizations (ii) would benefit from picking mappings that did not have used much reversals, since that means that its cost is low. Unexpectedly, Figure 3.3 shows that **sqw** (that has only optimization (ii)), was able to, in some cases, generate a smaller number of partitions than **ckf** (that uses only (i)), and even **bmt** (that uses both (i) and (ii)).

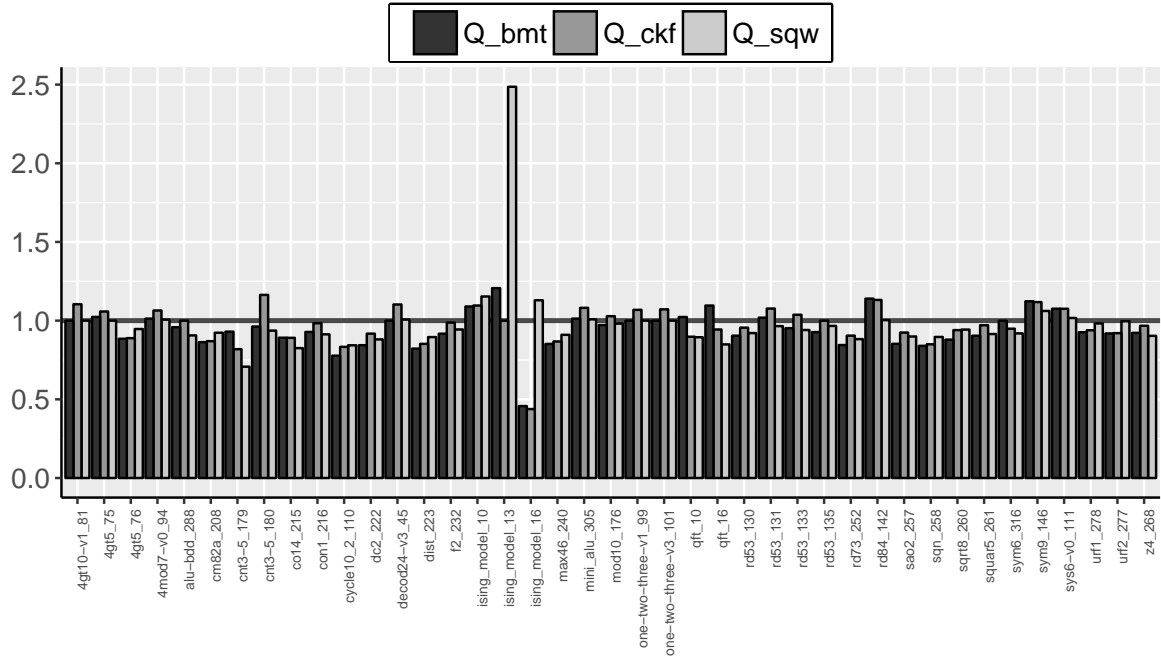


**Figure 3.3.** Baseline is the canonical version of the algorithm: `can`. In the Y-axis, we show how many partitions each allocator created in comparison to the baseline. In the X-axis, for clarity, we show the benchmarks that resulted in a difference of at least 5% among the allocators.

Figure 3.5 corroborates, showing that, indeed, `sqw` (pure *(ii)* optimization) is a bit better than `ckf` (pure *(i)* optimization) at reducing the number of partitions, as it manages to beat `can` in two more benchmarks than `ckf`. Unifying them both yields in a bit better partition splitter `bmt`. As expected, the number of partitions usually is a decisive factor when discovering better allocations. We found that in 87.97%, 86.08%, and 80.38% of the benchmarks tested, the allocators that had the smallest weighted cost, number of gates, and depth, respectively, were in the set of allocators that had the smallest number of partitions. Figure 3.4 let us have an overview of such phenomena.

### 3.3.3 RQ3: Quality of Allocation

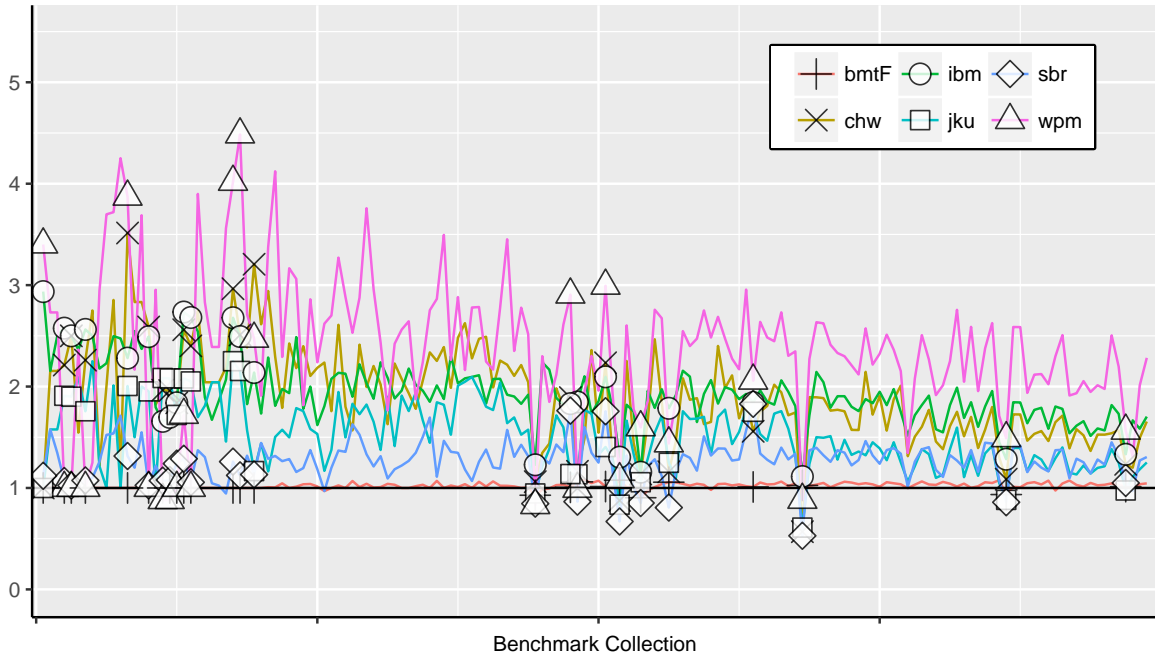
In this experiment, we compared the best combination of optimizations and two different configurations of parameters discussed in the previous sections, i.e.  $\langle 4, 320 \rangle$  (`bmtF`) and  $\langle 8, 1280 \rangle$  (`bmtS`) with the two optimizations enabled, with the state-of-the-art algorithms in the literature. In total, there were seven different allocators for this experiment. Which means that we executed seven different rounds of experiments. i.e. five times each of the 158 benchmarks, times the number of different combinations of



**Figure 3.4.** Baseline is the canonical version of the algorithm: `can`. In the Y-axis, we show the weighted cost of each allocator in comparison to the baseline. In the X-axis, for clarity, we show the benchmarks that resulted in a difference of at least 5% among the allocators.

Allocator	$\mu_g$	$\sigma_g$	Better Count	Better $\mu_g$	Worse Count	Worse $\mu_g$
bmt	0.8858	1.2593	66 (41.77%)	0.7338	3 (1.9%)	1.525
ckf	0.9198	1.232	61 (38.61%)	0.7754	9 (5.7%)	1.2924
sqw	0.9078	1.2618	63 (39.87%)	0.7558	3 (1.9%)	2.191

**Figure 3.5.** Baseline is the canonical version of the algorithm: `can`. Ratios of the number of partitions created by each of these algorithms, in relation to the baseline.



**Figure 3.6.** Ratio of the weighted cost found by different allocators, in relation to the cost found by `bmtS`. The Y-axis shows the weighted cost in logarithmic scale. The X-axis shows benchmarks ordered in increasing order of the cost found by `bmtS` (shaded area).

optimizations. In the end, we ran 3950 experiments.

Figure 3.6 and Figure 3.13 present the results of our experiments. Figure 3.6 shows an overview of the ratio of the weighted cost yielded by each of the allocators, in relation to the one yielded by `bmtS`. Figure 3.13 summarizes the results for all quality dimensions. In short, the results show:

- **Weighted Cost:** `bmtS` yielded the smallest cost. It outperformed `bmtF`, `sbr`, `jku`, `chw`, `ibm`, and `wpm` by 2%, 20%, 32%, 45%, 47%, and 55%, respectively;
- **Depth:** `bmtS` yielded the smallest depth. It outperformed `bmtF`, `sbr`, `jku`, `ibm`, `chw`, and `wpm` by 1%, 26%, 30%, 43%, 45%, and 53%, respectively;
- **Gates:** `bmtS` yielded the smallest number of gates. It outperformed `bmtF`, `sbr`, `chw`, `jku`, `ibm`, and `wpm` by 1%, 14%, 16%, 21%, 35%, and 43%, respectively.

**Analysis of results.** The ranking of the solutions is the same for all quality dimensions. `bmtS` delivers the best allocations, followed by `bmtF`, `sbr`, `jku`, `chw`, `ibm`, and

`wpm`, in this order. `bmtS`'s parameters allow users to, given enough time and space, get better solutions as they want. Notice that none of these algorithms finds a global optimal, because they all resort to some simplification of the problem. In our case, we segment the program greedily (which might not be the best way to do it), and also use an approximative algorithm to solve token swapping.

The fact that our algorithm outperforms all of its competitors over all dimensions indicates that we generate circuit with smaller error rates (given by the weighted costs), but also faster circuits (given by the depth). These results show that `bmtS` is able to yield good programs even while generating only 8 out of all 96 possible children (less than 10%), and allowing only 1280 mappings at each partition.

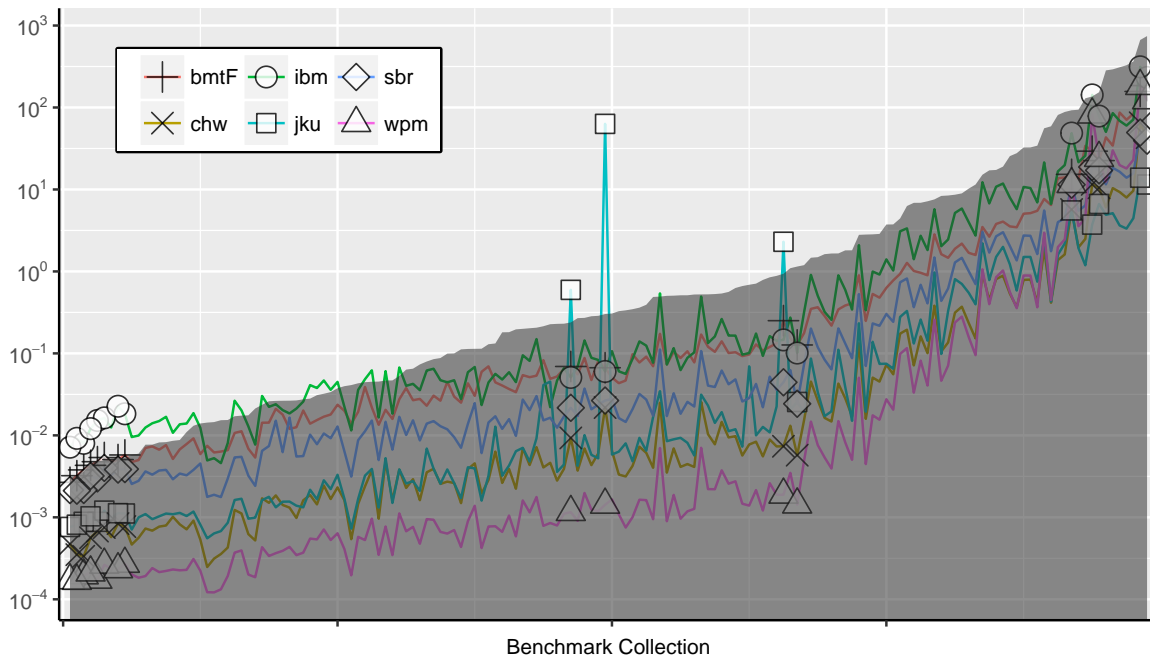
### 3.3.4 RQ4: Efficiency

For this research question, we used the results of the experiments executed for the last research question. Figure 3.7 and Figure 3.8 present the time and memory efficiency of the allocators, respectively. Figure 3.7 (3.8) shows an overview of the allocation time (memory usage) yielded by each of the allocators. Finally, Figure 3.13 summarizes the results for all efficiency dimensions. In short, the results show:

- **Allocation Time:** `bmtS` is the slowest algorithm. It was slower `ibm`, `bmtF`, `jku`, `sbr`, `chw`, and `wpm` by 115%, 244%, 740%, 2150%, 3384%, and 8830%, respectively;
- **Memory Consumption:** On average, there is not much variation memory-wise. Our algorithm better than `sbr` and `jku` by 5% and 2%, respectively. It was worse than `wpm`, `bmtF`, `chw`, `ibm` by 3%, 3%, 6%, and 8%, respectively.

**Analysis of results.** `BMT` is slower than the other allocators. On average (arithmetic mean), `bmtS` takes 26 seconds per benchmark, with a standard deviation of 100s. The other algorithms stay under 10 second per sample. The most time consuming part of `bmtS` is phase two's dynamic programming search. Dynamic programming accounts for, on average (geometric mean) 50% of the time, followed by the first phase with 37%. In spite of its higher computational complexity, `jku` runs faster than `bmtS`. Non-surprisingly, `wpm` is the fastest algorithm. This implementation uses a best-effort heuristic that is linear on the size of the program, and number of qubits; hence, it is expected to run fast.

`jku` uses an  $A^*$  tree to guide the search for a good solution to qubit allocation. Thus, it needs to store intermediate results of this quest, to make backtracking possible. In other words, by storing these intermediate nodes, `jku` trades time for space.



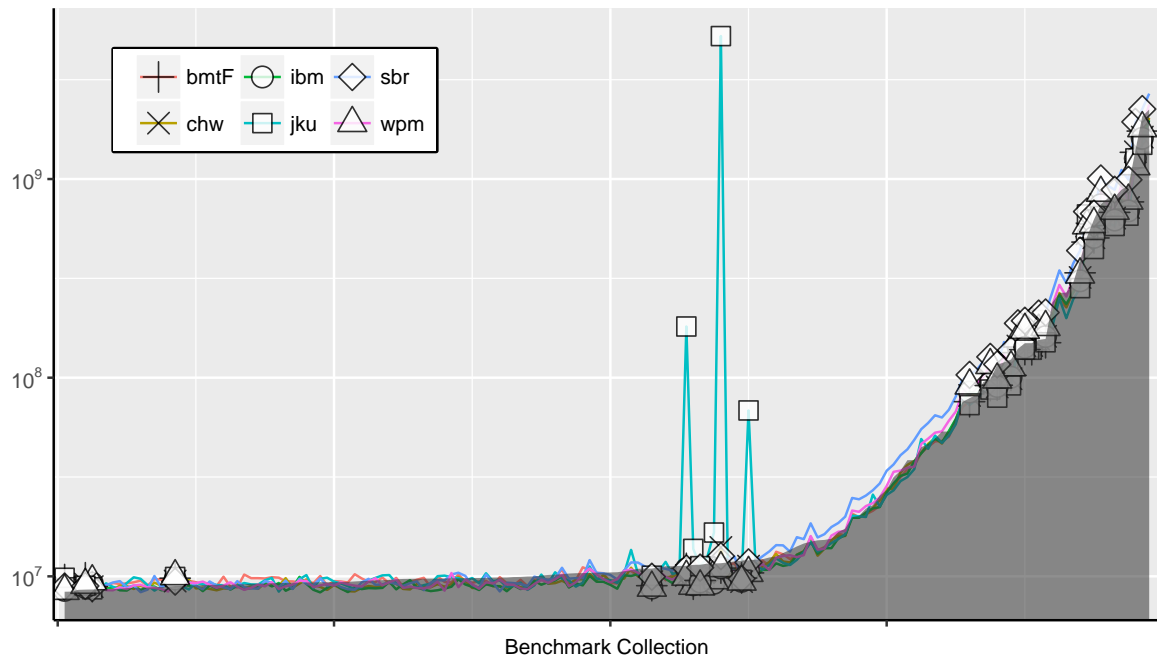
**Figure 3.7.** Time spent by different allocators. The Y-axis shows time (seconds) in logarithmic scale. The X-axis shows benchmarks ordered in increasing order of the time spent by `bmtS` (shaded area).

Figure 3.8 indicates that the memory usage of `wpm`, `sbr`, `ibm`, and `bmtS` grows in about the same rate for every program, while the same cannot be inferred for `jku`, which contains several outliers. More detail is covered in Section 4.2.

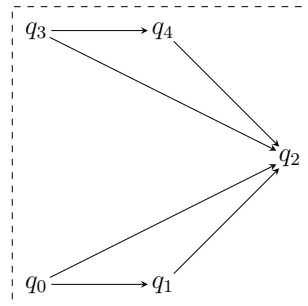
### 3.3.5 RQ5: Distance from Optimal

For this research questions, we asked how far were the results from the state-of-the-art algorithms, when compared to an exact algorithm presented earlier in this work (Section 2.1). Since the problem we solve here is proven to be NP-hard, the exact algorithm is exponential on the number of qubits, and thus, can be executed on a restricted number of qubits. It is also important to stress that `dyn` is optimal in terms of the weighted cost. So, we expect its weighted cost to be unmatched. We do evaluate the other metrics for completeness, though. As IBM [2018a] made available their 5-qubit quantum computer Yorktown (Figure 3.9), we decided to synthesize 5-qubit benchmarks, since the ones we had until now were benchmarks that used up to 16 qubits.





**Figure 3.8.** Memory used by different allocators. The Y-axis shows the memory (bytes) in logarithmic scale. The X-axis shows benchmarks ordered in increasing order of the memory used by `bmtS` (shaded area).



**Figure 3.9.** Coupling graph of the IBM Yorktown, a 5-qubit architecture [IBM, 2018a].

**Benchmarks Generation.** In order to generate benchmarks with number of gates  $Z$ , we simply used a random number generator to create  $Z$  *CNOT* gates, with random qubits. So, for each gate, we got one new random number that would represent each one of the two qubits of the *CNOT* gate. We generated 10 different benchmarks of 10, 20, 40,  $\dots$ , 640 *CNOT* gates each. Totalizing in 330 different benchmarks.

Following the pattern from the two last sections, we compiled each benchmark five times for each different allocator. Since there are now six allocators (`dyn` can now

be used), we ran in total 9900 benchmarks. Figure 3.14 summarizes the results shown below.

- **Weighted Cost:** `sbr` yielded the smallest ratio of the weighted cost. `dyn` was better than `sbr`, `bmtS`, `bmtF`, `jku`, `wpm`, `chw`, and `ibm` by 12%, 12%, 12%, 21%, 22%, 25%, and 38%, respectively;
- **Depth:** `bmtS` yielded the smallest ratio of the depth. `dyn` was better than `bmtS`, `bmtF`, `sbr`, `jku`, `wpm`, `chw`, and `ibm` by 15%, 15%, 18%, 17%, 24%, 29%, and 33%, respectively;
- **Gates:** `bmtS` yielded the smallest ratio of the number of gates. `dyn` was better than `bmtS`, `bmtF`, `sbr`, `jku`, `wpm`, `chw`, and `ibm` by 15%, 15%, 16%, 19%, 19%, 25%, and 47%, respectively;
- **Allocation Time:** `dyn` was the slowest allocator. `dyn` was slower than `sbr`, `ibm`, `bmtF`, `bmtS`, `jku`, `chw`, and `wpm` by 104%, 137%, 207%, 211%, 356%, 624%, and 1249%, respectively;
- **Memory Consumption:** On average, there is not much variation memory-wise. `dyn` consumed more memory than `sbr`, `chw`, `bmtS`, `bmtF`, `jku`, and `wpm` by 4%, 4%, 6%, 6%, 7%, and 7%, respectively. `ibm` consumed 2% more memory than `dyn`;

**Analysis of results.** As expected, `dyn` allocator was better than every other allocator. Non-surprising, it was also the slowest allocator. That aside, `bmtS`, `bmtF` and `sbr` got close results for the weighted cost. However, `bmtS` and `bmtF` was able to find solutions with significantly smaller depth and gates. As mentioned before (Section 3.1), the depth influences on the time it would take to execute the output program, and the number of gates influences on the overall error that the execution of the output program will generate.

It is important to emphasize, however, that the only algorithm that is bounded by the cost found by `dyn` is `wpm`. That is because the others change the sequence of *CNOT* relations, by interpreting the benchmarks in other non-sequential representations. That said, `dyn` was still the best one.

Allocator	Dim	G. Mean	G. Std. D.	Better Count	Better G. Mean	Worse Count	Worse G. Mean
foo	cost	x.xxxx	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx
	depth	x.xxxx	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx
	gates	x.xxxx	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx
	mem	x.xxxx	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx
	time	x.xxxx	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx	x ( <i>x.xx%</i> )	x.xxxx

**Figure 3.10.** Template for the tables in this chapter. The major line for allocator `foo` indicates that  $R_{comp}$  is  $R_{foo}$ . In the caption we shall describe who is  $R_{base}$ . Every column is the performance of  $R_{foo}$  in comparison to  $R_{base}$ . The highlighted lines indicate where  $R_{foo}$  was better than  $R_{base}$ .

$M_c - M_p$	Dim	10	20	40	80	160	320	640	1280	2560
1	cost	1.74 (1.16)	1.53 (1.14)	1.39 (1.12)	1.27 (1.1)	1.26 (1.1)	1.27 (1.1)	1.26 (1.1)	1.26 (1.1)	1.26 (1.1)
	depth	1.42 (1.1)	1.3 (1.08)	1.22 (1.07)	1.15 (1.06)	1.15 (1.06)	1.15 (1.06)	1.14 (1.06)	1.14 (1.06)	1.14 (1.06)
	gates	1.48 (1.11)	1.34 (1.09)	1.26 (1.08)	1.19 (1.07)	1.18 (1.06)	1.18 (1.06)	1.17 (1.07)	1.18 (1.07)	1.18 (1.07)
	mem	1.01 (1.05)	1 (1.04)	1.01 (1.04)	0.97 (1.04)	0.96 (1.03)	0.99 (1.04)	0.99 (1.04)	0.96 (1.03)	0.95 (1.03)
	time	0.07 (1.09)	0.08 (1.1)	0.09 (1.11)	0.12 (1.15)	0.12 (1.15)	0.12 (1.15)	0.12 (1.15)	0.12 (1.15)	0.13 (1.23)
2	cost	1.47 (1.1)	1.29 (1.07)	1.19 (1.05)	1.11 (1.04)	1.06 (1.03)	1.05 (1.02)	1.04 (1.02)	1.04 (1.02)	1.04 (1.03)
	depth	1.26 (1.06)	1.16 (1.04)	1.1 (1.03)	1.07 (1.03)	1.04 (1.02)	1.03 (1.02)	1.03 (1.02)	1.02 (1.02)	1.03 (1.02)
	gates	1.31 (1.07)	1.2 (1.05)	1.13 (1.03)	1.08 (1.03)	1.05 (1.02)	1.04 (1.02)	1.04 (1.02)	1.03 (1.02)	1.03 (1.02)
	mem	0.96 (1.03)	0.96 (1.03)	0.98 (1.04)	0.95 (1.03)	0.94 (1.03)	0.97 (1.04)	0.98 (1.04)	0.95 (1.03)	0.94 (1.03)
	time	0.07 (1.1)	0.08 (1.1)	0.1 (1.12)	0.13 (1.15)	0.19 (1.22)	0.22 (1.19)	0.21 (1.16)	0.22 (1.19)	0.23 (1.19)
4	cost	1.42 (1.09)	1.26 (1.06)	1.16 (1.05)	1.1 (1.03)	1.05 (1.02)	1.02 (1.01)	1.01 (1.01)	1 (1.01)	0.99 (1.01)
	depth	1.23 (1.05)	1.14 (1.04)	1.09 (1.03)	1.05 (1.02)	1.03 (1.02)	1.01 (1.02)	1 (1.01)	1 (1.01)	0.99 (1.02)
	gates	1.28 (1.06)	1.17 (1.04)	1.11 (1.03)	1.07 (1.02)	1.04 (1.02)	1.02 (1.01)	1.01 (1.01)	1 (1.01)	0.99 (1.01)
	mem	0.96 (1.03)	0.96 (1.03)	0.98 (1.04)	0.95 (1.03)	0.94 (1.03)	0.97 (1.04)	0.99 (1.03)	0.99 (1.02)	1.01 (1.02)
	time	0.07 (1.1)	0.08 (1.11)	0.1 (1.12)	0.15 (1.18)	0.21 (1.17)	0.34 (1.18)	0.53 (1.13)	0.83 (1.07)	1.31 (1.35)
8	cost	1.41 (1.09)	1.25 (1.06)	1.16 (1.05)	1.09 (1.03)	1.05 (1.02)	1.02 (1.01)	1.01 (1.01)	1 (1)	0.99 (1.01)
	depth	1.23 (1.05)	1.14 (1.04)	1.09 (1.03)	1.06 (1.02)	1.03 (1.02)	1.02 (1.01)	1 (1.01)	1 (1)	0.99 (1.01)
	gates	1.27 (1.06)	1.17 (1.04)	1.11 (1.03)	1.07 (1.02)	1.04 (1.02)	1.02 (1.01)	1.01 (1.01)	1 (1)	0.99 (1.01)
	mem	0.95 (1.03)	0.96 (1.03)	0.99 (1.04)	0.95 (1.03)	0.94 (1.03)	0.98 (1.04)	1 (1.03)	1 (1)	1.04 (1.03)
	time	0.07 (1.1)	0.08 (1.11)	0.1 (1.12)	0.15 (1.19)	0.21 (1.18)	0.36 (1.2)	0.59 (1.16)	1 (1)	1.59 (1.29)
16	cost	1.41 (1.09)	1.25 (1.06)	1.16 (1.04)	1.09 (1.03)	1.05 (1.02)	1.03 (1.02)	1.01 (1.01)	1 (1.01)	0.99 (1.01)
	depth	1.23 (1.05)	1.14 (1.04)	1.09 (1.03)	1.05 (1.02)	1.03 (1.02)	1.02 (1.02)	1.01 (1.02)	1 (1.01)	0.99 (1.01)
	gates	1.27 (1.06)	1.17 (1.04)	1.11 (1.03)	1.07 (1.02)	1.04 (1.02)	1.02 (1.01)	1.01 (1.01)	1 (1.01)	0.99 (1.01)
	mem	0.96 (1.03)	0.95 (1.03)	0.98 (1.04)	0.95 (1.03)	0.95 (1.03)	0.98 (1.04)	1.01 (1.03)	1.02 (1.02)	1.07 (1.05)
	time	0.08 (1.1)	0.08 (1.1)	0.11 (1.12)	0.16 (1.22)	0.22 (1.19)	0.39 (1.27)	0.63 (1.24)	1.1 (1.19)	1.78 (1.32)
32	cost	1.42 (1.09)	1.26 (1.06)	1.16 (1.05)	1.1 (1.03)	1.06 (1.02)	1.03 (1.02)	1.02 (1.02)	1.01 (1.01)	0.99 (1.01)
	depth	1.23 (1.06)	1.15 (1.04)	1.1 (1.03)	1.05 (1.02)	1.04 (1.02)	1.02 (1.02)	1.01 (1.02)	1 (1.01)	1 (1.01)
	gates	1.27 (1.06)	1.17 (1.04)	1.11 (1.03)	1.07 (1.02)	1.04 (1.02)	1.03 (1.01)	1.01 (1.01)	1 (1.01)	0.99 (1.01)
	mem	0.96 (1.03)	0.96 (1.03)	0.98 (1.04)	0.95 (1.03)	0.95 (1.03)	1 (1.04)	1.03 (1.04)	1.06 (1.05)	1.14 (1.11)
	time	0.08 (1.1)	0.09 (1.11)	0.11 (1.13)	0.16 (1.18)	0.24 (1.21)	0.41 (1.24)	0.69 (1.22)	1.21 (1.2)	2.06 (1.38)

**Figure 3.11.** Baseline is  $M_p = 8$  and  $M_c = 1280$   $\langle 8, 1280 \rangle$ . It shows the geometric mean ( $\mu_g$ ) of the metrics between all the combinations of parameters  $M_c$  (lines) and  $M_p$  (columns). The geometric standard deviation  $\sigma_g$  (in parenthesis) shows the spread of the data.

Allocator	Dim	G. Mean	G. Std. D.	Better Count	Better G. Mean	Worse Count	Worse G. Mean
bmt	cost	0.9568	1.0952	87 (55.06%)	0.9134	15 (9.49%)	1.0615
	depth	0.9792	1.0976	79 (50%)	0.9344	44 (27.85%)	1.0475
	gates	0.9739	1.0504	83 (52.53%)	0.9451	19 (12.03%)	1.0274
	mem	1.0474	1.0799	40 (25.32%)	0.9506	118 (74.68%)	1.0823
	time	1.7921	2.0671	19 (12.03%)	0.6789	139 (87.97%)	2.0464
ckf	cost	0.9689	1.0937	73 (46.2%)	0.9156	23 (14.56%)	1.0648
	depth	0.9804	1.1178	68 (43.04%)	0.9262	47 (29.75%)	1.0455
	gates	0.9806	1.0517	72 (45.57%)	0.945	22 (13.92%)	1.0457
	mem	1.0464	1.0919	48 (30.38%)	0.9411	110 (69.62%)	1.096
	time	1.7765	2.1566	23 (14.56%)	0.6863	135 (85.44%)	2.089
sqw	cost	0.9707	1.1007	83 (52.53%)	0.9283	21 (13.29%)	1.0725
	depth	0.9851	1.1054	79 (50%)	0.9482	31 (19.62%)	1.0609
	gates	0.983	1.0477	78 (49.37%)	0.9559	26 (16.46%)	1.032
	mem	0.9885	1.0569	97 (61.39%)	0.9549	61 (38.61%)	1.0443
	time	0.8767	1.6374	76 (48.1%)	0.6143	82 (51.9%)	1.2191

**Figure 3.12.** Baseline is the canonical version of the algorithm: `can`. It shows the effects of the proposed optimizations relative to the canonical algorithm. Each allocator represents a new combination of improvements applied to the canonical version.

Allocator	Dim	G. Mean	G. Std. D.	Better Count	Better G. Mean	Worse Count	Worse G. Mean
bmtF	cost	1.0207	1.0279	11 (6.96%)	0.9669	107 (67.72%)	1.0343
	depth	1.0179	1.0323	16 (10.13%)	0.9748	106 (67.09%)	1.0308
	gates	1.0188	1.0217	8 (5.06%)	0.9796	110 (69.62%)	1.0286
	time	0.2905	1.8705	154 (97.47%)	0.2809	4 (2.53%)	1.0534
	mem	0.9648	1.081	107 (67.72%)	0.9255	51 (32.28%)	1.0527
chw	cost	1.8398	1.3012	2 (1.27%)	0.7062	155 (98.1%)	1.87
	depth	1.9236	1.2447	2 (1.27%)	0.8133	155 (98.1%)	1.9533
	gates	1.4793	1.1923	2 (1.27%)	0.7954	155 (98.1%)	1.495
	time	0.0287	2.2563	158 (100%)	0.0287	0 (0%)	-
	mem	0.9415	1.0755	123 (77.85%)	0.9173	35 (22.15%)	1.0316
jku	cost	1.4729	1.254	5 (3.16%)	0.8386	149 (94.3%)	1.5167
	depth	1.4424	1.2204	5 (3.16%)	0.7801	149 (94.3%)	1.487
	gates	1.2789	1.1568	5 (3.16%)	0.8923	149 (94.3%)	1.303
	mem	1.0201	1.762	105 (66.46%)	0.9021	52 (32.91%)	1.3081
	time	0.0445	3.2652	155 (98.1%)	0.04	3 (1.9%)	10.8607
sbr	cost	1.2502	1.1817	10 (6.33%)	0.8189	148 (93.67%)	1.2865
	depth	1.3508	1.1652	7 (4.43%)	0.8873	151 (95.57%)	1.3773
	gates	1.1663	1.1186	10 (6.33%)	0.8927	148 (93.67%)	1.1876
	mem	1.0544	1.1246	65 (41.14%)	0.9483	93 (58.86%)	1.1356
	time	0.1193	2.3326	158 (100%)	0.1193	0 (0%)	-
wpm	cost	2.2586	1.3952	5 (3.16%)	0.8911	147 (93.04%)	2.4101
	depth	2.1698	1.3511	5 (3.16%)	0.8225	148 (93.67%)	2.3015
	gates	1.7553	1.2827	4 (2.53%)	0.9229	148 (93.67%)	1.8272
	mem	0.971	1.0903	101 (63.92%)	0.9233	56 (35.44%)	1.0628
	time	0.0112	3.106	158 (100%)	0.0112	0 (0%)	-
ibm	cost	1.9116	1.1742	0 (0%)	-	158 (100%)	1.9116
	depth	1.7664	1.1687	2 (1.27%)	0.849	156 (98.73%)	1.7831
	gates	1.5541	1.1347	0 (0%)	-	158 (100%)	1.5541
	mem	0.9258	1.0778	134 (84.81%)	0.9089	24 (15.19%)	1.026
	time	0.4634	2.3381	125 (79.11%)	0.3293	33 (20.89%)	1.6889

**Figure 3.13.** Baseline is the *BMT* (*bmtS*) allocator. Except for **Better Count**, the smaller the reported value, the better for the corresponding competing allocator. The rows (dimensions) where our *bmtS* loses are highlighted.

Allocator	Dim	G. Mean	G. Std. D.	Better Count	Better G. Mean	Worse Count	Worse G. Mean
bmtF	cost	1.145	1.0086	0 (0%)	-	33 (100%)	1.145
	depth	1.1846	1.0181	0 (0%)	-	33 (100%)	1.1846
	gates	1.1875	1.0234	0 (0%)	-	33 (100%)	1.1875
	mem	0.9397	1.0291	33 (100%)	0.9397	0 (0%)	-
	time	0.0048	1.3031	33 (100%)	0.0048	0 (0%)	-
bmtS	cost	1.1433	1.0083	0 (0%)	-	33 (100%)	1.1433
	depth	1.1826	1.0169	0 (0%)	-	33 (100%)	1.1826
	gates	1.1833	1.0227	0 (0%)	-	33 (100%)	1.1833
	mem	0.9404	1.0355	33 (100%)	0.9404	0 (0%)	-
	time	0.0047	1.2824	33 (100%)	0.0047	0 (0%)	-
chw	cost	1.2831	1.0423	0 (0%)	-	33 (100%)	1.2831
	depth	1.4128	1.0623	0 (0%)	-	33 (100%)	1.4128
	gates	1.3442	1.072	0 (0%)	-	33 (100%)	1.3442
	mem	0.9608	1.0216	32 (96.97%)	0.9596	1 (3.03%)	1.0006
	time	0.0016	1.0763	33 (100%)	0.0016	0 (0%)	-
ibm	cost	1.6196	1.0257	0 (0%)	-	33 (100%)	1.6196
	depth	1.4998	1.0474	0 (0%)	-	33 (100%)	1.4998
	gates	1.9025	1.0543	0 (0%)	-	33 (100%)	1.9025
	mem	1.0216	1.0285	7 (21.21%)	0.9842	26 (78.79%)	1.0319
	time	0.0072	1.0639	33 (100%)	0.0072	0 (0%)	-
jku	cost	1.269	1.0239	0 (0%)	-	33 (100%)	1.269
	depth	1.2173	1.0409	0 (0%)	-	33 (100%)	1.2173
	gates	1.244	1.0438	0 (0%)	-	33 (100%)	1.244
	mem	0.932	1.0381	32 (96.97%)	0.93	1 (3.03%)	1.0005
	time	0.0028	1.0777	33 (100%)	0.0028	0 (0%)	-
sbr	cost	1.1391	1.0174	0 (0%)	-	33 (100%)	1.1391
	depth	1.2298	1.0203	0 (0%)	-	33 (100%)	1.2298
	gates	1.2038	1.0246	0 (0%)	-	33 (100%)	1.2038
	mem	0.9557	1.0332	31 (93.94%)	0.9529	2 (6.06%)	1.001
	time	0.0095	1.1974	33 (100%)	0.0095	0 (0%)	-
wpm	cost	1.3404	1.0179	0 (0%)	-	33 (100%)	1.3404
	depth	1.3283	1.0184	0 (0%)	-	33 (100%)	1.3283
	gates	1.238	1.0216	0 (0%)	-	33 (100%)	1.238
	mem	0.9269	1.041	33 (100%)	0.9269	0 (0%)	-
	time	8e-04	1.0972	33 (100%)	8e-04	0 (0%)	-

**Figure 3.14.** Baseline is the optimal algorithm dyn allocator. Except for **Better Count**, the smaller the reported value, the better for the corresponding competing allocator. The rows (dimensions) where our dyn loses are highlighted.





# Chapter 4

## Literature Review

Quantum computing [Benioff, 1980], and the notion of universal quantum computers [Deutsch, 1985] date back to the eighties. In the late nineties we saw the first quantum algorithms with practical purpose, such as integer factorization [Shor, 1997] and database search [Grover, 1996]. Programming languages that let developers interact with quantum machines came later [Selinger, 2004; Gay, 2006; Balensiefer et al., 2005; Smith et al., 2017].

Because quantum computers are so recent, so is the interest on quantum compilers. Except for some early work [Tucci, 1999], compiler frameworks that translate high-level languages to quantum gates have only been proposed in recent years [Häner et al., 2016; Green et al., 2013; Wecker and Svore, 2014; Javadi-Abhari et al., 2014; Maslov, 2017]. Most of them involve solving the qubit allocation problem as part of the compilation flow when targeting partially-connected qubit machines like superconducting quantum computers. Therefore, the algorithms presented here are applicable to all these frameworks. We evaluated the qubit allocators of open-source projects in Chapter 3. Among classical architectures, clustered VLIW processors also have connectivity constraints between registers. However, the clustered VLIW register allocation problem is very different than the qubit allocation problem. The former is tightly linked with instruction scheduling [Codina et al., 2001], whereas this degree of freedom is not available in reversible circuits.

**On Prior Solutions to Qubit Allocation.** There has been previous attempts to solve qubit allocation [Maslov et al., 2008; Shafaei et al., 2014; Shrivastwa et al., 2015; Lin et al., 2015; Pedram and Shafaei, 2016; Zulehner et al., 2018; Lin et al., 2018; Lao et al., 2018; Li et al., 2018; Siraichi et al., 2018]. The main difference between them and our work is the fact that they focus on particular topologies of coupling graphs, and

use only swaps to implement the transitions between different logical-to-physical qubit mappings. In what follows, we shall discuss some earlier work, starting with Maslov et al. [2008]. In 2008, they have formalized an instance of the problem similar to our Definition 5, and have presented an exponential-time heuristic to solve it. Similar to `Q_ibm`, this heuristic partitions CNOT gates into sets that can be solved without swaps. Maslov et al. [2008] find these partial solutions via graph isomorphism (between the coupling graph and a subset of dependences). They use a heuristic to insert swaps connecting different partitions of the quantum circuit.

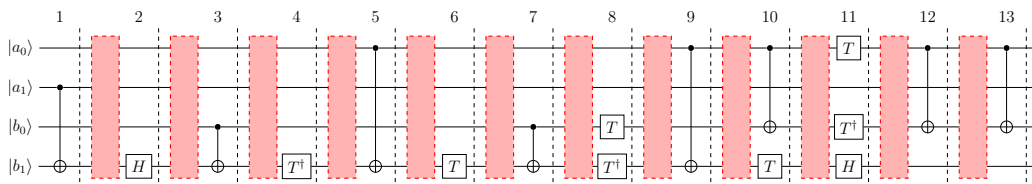
In 2014, Shafaei et al. [2014] have proposed a methodology to map logical into physical qubits based on Mixed Integer Programming (MIP) [Wolsey, 2008]. They focus on coupling graphs having a grid architecture, and rely on this assumption to provide a simple and elegant algorithm. In this research, we assume a general topology for the coupling graph. Furthermore, like Maslov et al. [2008], Shafaei et al. [2014] restrict the set of allowed transformations to swaps. Finally, whereas we use dynamic programming to find an exact solution to qubit allocation, they employ MIP, a different method. Both these exact solutions are exponential in their worst case. Along similar lines, Pedram and Shafaei [2016] use Minimum Linear Arrangement (MINLA)<sup>1</sup> to solve qubit allocation on 1D grid architectures, again, using only swaps to ensure the correct semantics of the implementation of the quantum circuit.

Like Shafaei et al. [2014], Lin et al. [2015] also present a solution to qubit allocation in 2D architectures. However, contrary to them, Lin et al. [2015] rely on heuristics to find said solution. Similar to the algorithm that we have discussed in Section 2.2, they split allocation into two phases, which they have called *placement* and *routing*. Placement fills a role similar to the algorithm in Figure 2.3. Routing, in turn, would have a purpose similar to the algorithm in 2.5. Nevertheless, our heuristics use different techniques, given that we deal with general coupling graphs, and resort to operations other than swaps, when transforming quantum circuits.

**Token Swapping and Subgraph Isomorphism.** Token swapping was formalized and proven NP-complete recently [Yamanaka et al., 2014]. Since then, this problem has been extensively studied [Yamanaka et al., 2014; Bonnet et al., 2016; Miltzow et al., 2016; Kawahara et al., 2017; Yamanaka et al., 2017]. We know two concrete implementations of algorithms that solves token swapping exactly [Miltzow et al., 2016; Siraichi et al., 2018]. These algorithms have exponential runtime. In this paper, to solve token swapping, we implement the 4-approximative algorithm from Miltzow et al.

---

<sup>1</sup>For further details on MINLA, we recommend the introduction written by Petit [2003]



**Figure 4.1.** Quantum circuit indicating (in red) where the swaps would be ideally placed if allocated one layer at a time.

[2016], which runs in polynomial time. In contrast to token swapping, subgraph isomorphism is an old problem, and there are several heuristics and exact algorithms to solve it [Cordella et al., 2004; He and Singh, 2008; Zhao and Han, 2010; Han et al., 2013]. We do not try to solve subgraph isomorphism exactly. We implement a best-effort approach: if Section 2.3.1’s search does not find a solution to subgraph isomorphism, this failure does not mean that such solution does not exist.

In the following sections, we shall describe in details some of the previous work done on qubit allocation. The next three sections shall present the state-of-the-art algorithms found in the literature [Zulehner et al., 2018; Li et al., 2018; IBM, 2018b], that we compared our results against in Chapter 3. Finally, we shall describe the implementation of the 4-approximative algorithm [Miltzow et al., 2016] for the token swapping problem in Section 4.4.

## 4.1 IBM Qiskit Mapper

Summarizing, for each layer  $l$ , the algorithm developed by IBM tries to find a sequence of swaps such that one is able to execute all gates in the layer  $l$  with the final mapping (the one after application of the swap sequence). The red areas highlighted in Figure 4.1 indicates where the sequence of swaps would be inserted (if any). Problem 14 formalizes the slight different qubit allocation problem that it solves.

**Definition 14** (Layer Allocation Problem). ***Input:** an initial mapping  $f$ , a coupling graph  $G_q = (Q, E_q)$ , plus a list  $\Psi = (P \times P)^n, n \geq 1$  of  $n$  control relations between pseudo qubits. **Output:** a swap sequence that transforms  $f$  into  $f'$ , such that  $f'$  respects the control relations in  $\Psi$ .*

**Finding the Swap Sequence.** The authors make use of a hill climbing-based algorithm. Assuming that the layer to be satisfied is  $l$ , the algorithm tries to find the first

mapping  $f$ , such that it minimizes Equation 4.1.

$$Cost(f) = \sum_{(a,b) \in l} dist(f(a), f(b))^2 * rand(0, 1/|Q|) \quad (4.1)$$

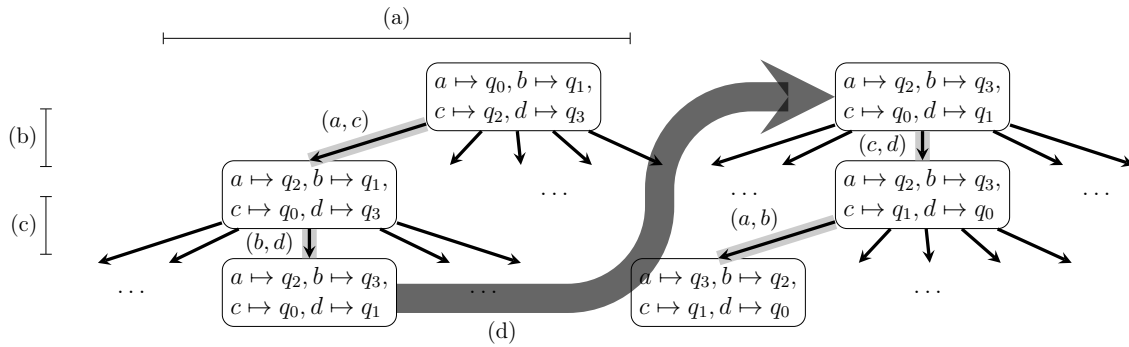
**Generating Adjacent Solutions.** At each mapping  $f$  to be processed by the algorithm, it creates one adjacent solution for each edge in the coupling graph, and chooses the one with the minimum cost. That said, the authors also try to minimize the number of layers, i.e. depth, by swapping disjoint sets of qubits. Example 20 illustrates the described algorithm.

**Example 20.** *Figure 4.2 (a) illustrates one layer of exploration of the mapping space. From each mapping, we generate  $|E(G_q^u) \setminus Used|$  new mappings (one for each edge), where  $Used$  is the set of physical qubits already used before in this tree. In the top, left-most mapping  $f_0 = \{a \mapsto q_0, b \mapsto q_1, c \mapsto q_2, d \mapsto q_3\}$ , we generate exactly  $|E(G_q^u)|$  adjacent mappings (Figure 4.2 (b)), since no qubit was used yet. One of them is  $f_1 = \{a \mapsto q_2, b \mapsto q_1, c \mapsto q_1, d \mapsto q_3\}$ , which consists in swapping  $(a, c)$  from  $f_0$ . Assuming  $f_1$  has the smallest cost among the other generated mappings, we repeat the process for it (Figure 4.2 (c)). However, this time, we generate  $|E(G_q^u) \setminus \{(q_0, *), (q_2, *)\}|$  mappings, since we have already used those qubits. Finally,  $|E(G_q^u) \setminus \{(q_0, *), (q_1, *), (q_2, *), (q_3, *)\}| = 0$ , i.e. we have used all physical qubits in the coupling graph. Thus, as Figure 4.2 (d) shows, we clear the  $Used$  set and start over from the last mapping, as long as the stop condition has not been met.*

**Stop Condition.** This algorithm explores the space of different mappings until it reaches one of two conditions: (i) a threshold on the number of layers of swap operations; or (ii) a state where the qubits of every two-qubit gate in  $l$  is mapped to an adjacent vertex in the coupling graph. i.e. it will stop if the set of swaps up until now surpasses a threshold, or if we reach a mapping  $f$  such that:

$$\sum_{(a,b) \in l} dist(f(a), f(b)) = |l|$$

**Trials.** There is a chance that the algorithm will not find a swap sequence that satisfies the stop condition as specified above. In those cases, it tries a specified number of times. If it still does not find a solution, it gives up and splits the current layer into one-gate layers (which are guaranteed to be solved).



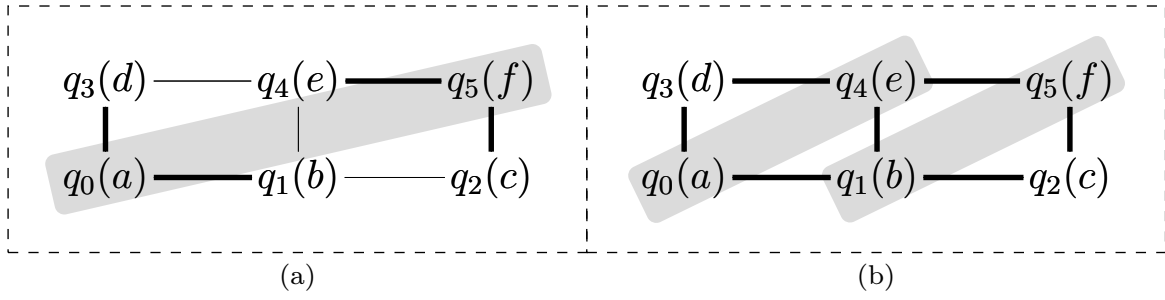
**Figure 4.2.** (a) creation of one layer of swaps  $\{(a, c), (b, d)\}$  whose intersection is empty; (b) generation of one mapping for each edge, which represents the swap between its endpoints; (c) same as (b), but for all edges in  $E(G_q^u) \setminus \{(q_0, *), (q_2, *)\}$ ; (d) creation of another layer of swaps, starting from the last mapping in the last layer of swaps;

## 4.2 A-star Search

Zulehner et al. [2018] used almost the same idea as IBM, and developed an A\*-search based algorithm for finding the swap sequence. The main idea remains the same to the one described in the last section (Section 4.1): for each layer  $l_i$ , find the smallest swap sequence that satisfies it, from the mapping used in the last layer  $l_{i-1}$  (illustrated in Figure 4.1). The authors modelled their A\*-search algorithm as follows:

- **State:** each state of this algorithm consists in a mapping  $f$ ;
- **Neighbor Function:** the neighbor function developed by the authors has exponential-time complexity. That is because for each mapping  $f$  being processed, they generate all possible combination of swaps, which totalizes in  $2^{|E(G_q^u)|}$  different sets;
- **Cost Functions:** their cost functions consists in a weighted sum that accounts for the minimum distance between the qubits used in each gate in the current layer, plus the same for the next layer. Assuming that  $l_i$  is the current layer, the heuristic is given by the following equation:

$$h(f) = \sum_{(a,b) \in l_i} \text{dist}(f(a), f(b)) + \sum_{(a,b) \in l_{i+1}} \text{dist}(f(a), f(b))$$



**Figure 4.3.** Illustration of two possible layer configurations, and the edges considered for creating the swap combinations. (a) shows gate  $CNOT_{af}$ , where each  $a$  and  $f$  touch two edges each; yielding  $2^4 = 16$  different swap combinations. (b) shows gates  $CNOT_{ae}$  and  $CNOT_{bf}$ , that touch every edge in the coupling graph; yielding  $2^7 = 128$  different swap combinations.

**Reducing the Number of Neighbors.** The authors also came up with an optimization for generating the neighbors, since it has exponential complexity (increasing rapidly the number of states). They observed that instead of considering all edges from the coupling graph, they could take into account only the ones being used in a  $CNOT$  in the current layer. That is because swapping two qubits not used in the current layer  $l$  would not change the fact of whether the mapping satisfies or not  $l$ . Even with this optimization, the asymptotic time complexity do not change. If there is a layer where all qubits are used in a  $CNOT$  gate in the current layer, it will be exponential on the number of edges. Example 21 illustrates the effects of this optimization, as well as a case where the worst case remains the same.

**Example 21.** *Figure 4.3 shows two possible layer configurations in a program. In Figure 4.3 (a) there is only one  $CNOT$  gate between pseudo-qubits  $a$  and  $f$ . Therefore, we consider only edges touching the physical qubits where the pseudo ones are mapped:  $q_0$  and  $q_5$ , respectively. There are two of such edges for each  $a$  and  $f$ :  $(q_0, q_3)$ ,  $(q_0, q_1)$ ,  $(q_5, q_4)$ ,  $(q_5, q_2)$ ; yielding  $2^4 = 16$  different combinations. Thus, 16 different states for the  $A^*$ -search. Figure 4.3 (b) illustrates the worst case scenario with two gates:  $CNOT_{ae}$  and  $CNOT_{bf}$ . Those qubits are enough to touch every edge in the coupling graph, yielding  $2^7 = 128$  different combinations.*

## 4.3 SABRE

Li et al. [2018] created a simple greedy algorithm. It is very brief and effective. For this algorithm, the authors use the gate dependency graph  $D$  (Section 1.2) representation. The idea of the algorithm consists in the following steps:

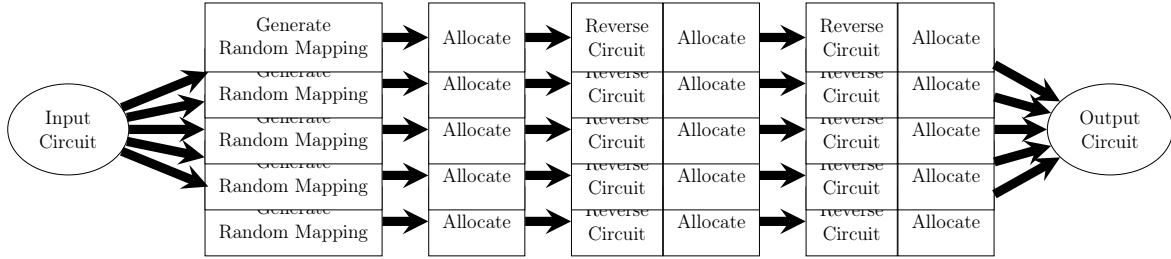
1. gather all the gates that have in-degree 0 in a set of gates  $W$  that may be already processed;
2. discard all gates from  $W$  (and remove from  $D$ ) that may be executed, i.e. respect the coupling graph constraints when translated, with the current mapping  $f$  (if this is the first iteration, generate one randomly). If there was any such gate, go to (1);
3. gather the next  $N$  gates in the remaining program, not already in  $W$ , in a set  $F$  of gates to be evaluated in the future;
4. given  $(a, b) \in W$ , for every edge  $(f(a), f(b)) \in E(G_q^u)$ , create a new mapping  $f' = f[a \mapsto f(b), b \mapsto f(a)]$ , and calculate a cost according to the following equation:

$$Cost(f') = \sum_{(a,b) \in W} dist(f'(a), f'(b)) + 0.5 \sum_{(a,b) \in F} dist(f'(a), f'(b))$$

5. reassign  $f$  to the  $f'$  that has the minimum cost. Go back to (1).

**Finding an Initial Mapping.** The algorithm described above assumes the presence of an initial mapping, not specifying a method for finding it. The authors notable insight is that: by reversing the order of the circuit, it is possible to finish the allocation with a good mapping for the beginning of the circuit. Thus, by reversing and allocating the circuit, with the final mapping of the previous iteration as its initial mapping, they are able to achieve a better allocation result. Example 22 illustrates this method.

**Example 22.** *Figure 4.4 illustrates the whole execution of the SABRE algorithm. It consists in five different executions of the algorithm described in this section. Each of these executions start from a different random mapping, and apply the three allocation method in order to find a good initial mapping. Along with the initial mapping, they are able to allocate the given input circuit into a circuit of smaller cost. In the end, SABRE yields the allocation that yielded the smaller cost among the five different executions.*



**Figure 4.4.** Scheme of the whole algorithm proposed by Li et al. [2018]. The authors execute the allocation iteration five times, and get the best out of them.

## 4.4 4-Approximative Token Swapping

Miltzow et al. [2016] created a 4-approximative algorithm for the token swapping problem (Section 1.1.5), and also proved that it is NP-hard. The algorithm consists in finding (un)happy swap chains (Definition 15) with a data structure, which we shall call Swap Chain Graph  $S$ . The main assumption is that each token has only one target vertex. i.e. the relation  $f_{tgt}$  is a function. There are cases where this assumption is broken, to which we will address later in Section 4.4.1.

**Building the Swap Chain Graph.** Given that the input to this problem are, as specified in Section 1.1.5, the undirected graph  $G = (V, E_G)$  and two mappings  $f_{src}, f_{tgt}$ , the authors build the directed chain graph  $S = (V, E_S)$  as follows. Given that token  $t$  is inside vertex  $v_i \in V$ , there is an edge  $(v_i, v_j) \in E_S$  iff the path  $v_j \rightsquigarrow f_{tgt}(t)$  is bigger than  $v_i \rightsquigarrow f_{tgt}(t)$ . i.e. the edge  $(v_i, v_j)$  exists iff by swapping the tokens in each of those vertices, the distance of the token inside  $v_i$  to its target vertex is reduced. Figure 4.5 shows different examples of swap chain graphs.

**Definition 15** ((Un)Happy Swap Chain). *A happy swap chain consists in a cycle in the Swap Chain Graph  $S$ . An unhappy swap is a swap in the edge  $(v_i, v_j) \in E_S$  such that  $v_j$  has out-degree 0. i.e. the token in  $v_j$  is in its target vertex.*

**Finding the Swap Sequence.** Given the *Swap Chain Graph  $S$* , and the definition of (un)happy swap chains (Definition 15), we are able to build the swap sequence following the steps below, until  $S$  has no more edges. Example 23 illustrate the steps shown below:

1. build the swap chain graph  $S$ ;



2. if there is a happy swap chain  $H$  in  $S$ :
  - a) insert  $H$  to the solution, and apply it to the current mapping. Go to (1).
3. else, get one arbitrary unhappy swap, insert it to the solution, and apply it to the current mapping. Go to (1).

**Example 23.** Figure 4.5 shows the sequence of iterations the algorithm takes to find the solution. In this example, the token set is  $T = \{a, b, c, d, e\}$ ,  $f_{src} = \{a \mapsto q_2, b \mapsto q_4, c \mapsto q_0, d \mapsto q_1, e \mapsto q_3\}$ , and  $f_{tgt} = \{a \mapsto q_0, b \mapsto q_1, c \mapsto q_2, d \mapsto q_4, e \mapsto q_3\}$ . At each step,  $f_{src}$  is updated, i.e. the swaps are applied to  $f_{src}$ , while being inserted in the solution. Since  $f_{src}$  is updated, the swap graph  $S$  on the right is also iteratively updated. In Figure 4.5 (i) it is possible to find a happy swap chain of length one:  $(q_0, q_2)$ . In the next step (ii), there are no happy swap chains. Therefore, we choose one of the unhappy swap available:  $(q_1, q_2)$  or  $(q_4, q_2)$ . Both steps (iii) and (iv) present happy swap chains of length one:  $(q_2, q_4)$  and  $(q_1, q_2)$ , respectively. Finally, we reach step (v), where  $f_{src} = f_{tgt}$ , which means that  $E_S = \emptyset$ .

**Proving the Approximation Factor.** Given that  $L$  is the sum of the distances between each token and its target vertex, we know that: (i) each happy swap chain of length  $l$  always decreases  $l + 1$  from  $L$ ; and (ii) each unhappy swap does not change  $L$ . Consider Lemmas 2 and 3. We use these two lemmas, and shows a sketch of the proof for the approximation factor of the algorithm created by Miltzow et al. [2016].

**Lemma 2** (Lower Bound - Miltzow). *The minimum number of swaps needed is:*

$$|swaps| \geq \frac{\sum_{t \in T} dist(f_{src}(t), f_{tgt}(t))}{2} = \frac{L}{2}$$

**Proof.** In the best case, we shall use only happy swap chains, where each swap will decrease 2 from  $L$ . □

**Lemma 3** (Kinds of Swap - Miltzow). *If one token  $t$  already in its target vertex has been through an unhappy swap, its next operation will be a happy swap.*

**Proof.** Let us show by contradiction. Since  $t$  was in its target vertex  $v_t = f_{tgt}(t)$ , we know that, after the unhappy swap,  $v = f_{src}(t)$  has only one outgoing edge  $(v, v_t) \in E_S$  (“Building the Swap Chain Graph”) to its target vertex  $v_t$ . Therefore, in order for another unhappy swap to be applied over  $v$  (vertex that currently contains  $t$ ), there are two possible cases:

- unhappy swap between  $(v_i, v)$ : it would mean that  $v$  has out-degree 0, which it is known that it is not true, since  $v$  is not  $t$ 's target vertex;
- unhappy swap between  $(v, v_i)$ : it would mean that  $v_i$  has out-degree 0. However, we know that  $v$  has only one edge  $(v, v_t)$ . So, if  $v_i = v_t$ , and  $v_t$  has out-degree 0, it means that  $v_t$  is the target vertex for a token other than  $t$ . Impossible by definition.

□

**Theorem 4** (4-Approximative Algorithm - Miltzow). *The solution described is a 4-approximative algorithm.*

**Proof.** Since each happy swap chain decreases  $l + 1$  from  $L$ , we know that:

$$|happy| < L$$

Lemma 3 shows that:

$$|happy| \geq |unhappy|$$

Thus, we obtain:

$$\begin{aligned} |happy| + |unhappy| &= |swaps| \\ |happy| + |happy| &\geq |swaps| \\ 2 * |happy| &\geq |swaps| \end{aligned}$$

Finally, Lemma 2 shows that the optimum solution has at least  $L/2$  swaps. Therefore:

$$2 * |happy| < 2L = 4 * \frac{L}{2}$$

□

#### 4.4.1 Colored Token Swapping

The colored token swapping consists in a broader version of the token swapping problem [Yamanaka et al., 2015]. The difference is that in the colored version, the number of target vertices for each token can be greater than 1. i.e.  $f_{tgt}$  is not a function over  $V$  anymore, but over  $\mathcal{P}(V)$ . That said, we are always guaranteed that there will be a solution. i.e. if we build a bipartite graph  $B = (T, V, E_B)$ , where  $E_B = f_{tgt}$ , we would always have a perfect matching.

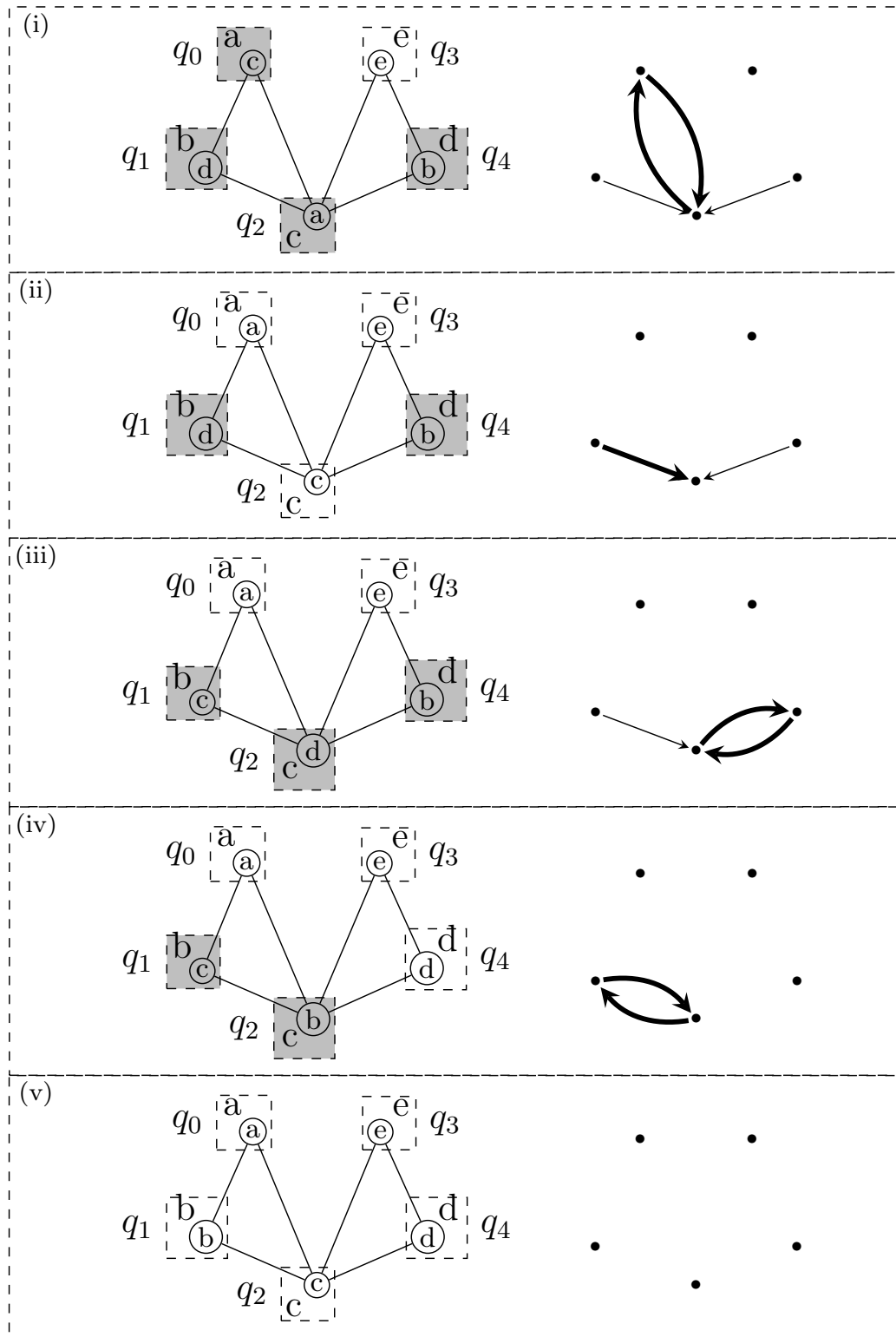
Fortunately, Miltzow et al. [2016] propose a solution to this problem, a 4-approximative one. The only change from the 4-approximative algorithm described above is that there is a preprocessing phase. In this phase, they transform the problem into the non-colored version of the problem by applying a minimum weighted matching algorithm, such as Kuhn [1955] solution, over a bipartite graph  $B$  (Definition 16). Example 24 illustrates this structure.

**Definition 16** (Colored Bipartite Graph). *the colored bipartite graph of a graph  $G = (V, E)$  is a weighted graph  $B = (T, V, E_B)$ , such that  $(t, v) \in E_B$  iff the token  $v \in f_{tgt}(t)$ . i.e.  $E_B = \{(t, v) | v \in f_{tgt}(t)\}$ . The weight of each edge  $(t, v) \in E_B$  is given by the distance from the vertex  $t$  is currently in, and  $v$ . i.e.  $w(t, v) = \text{dist}(f_{src}(t), v)$ .*

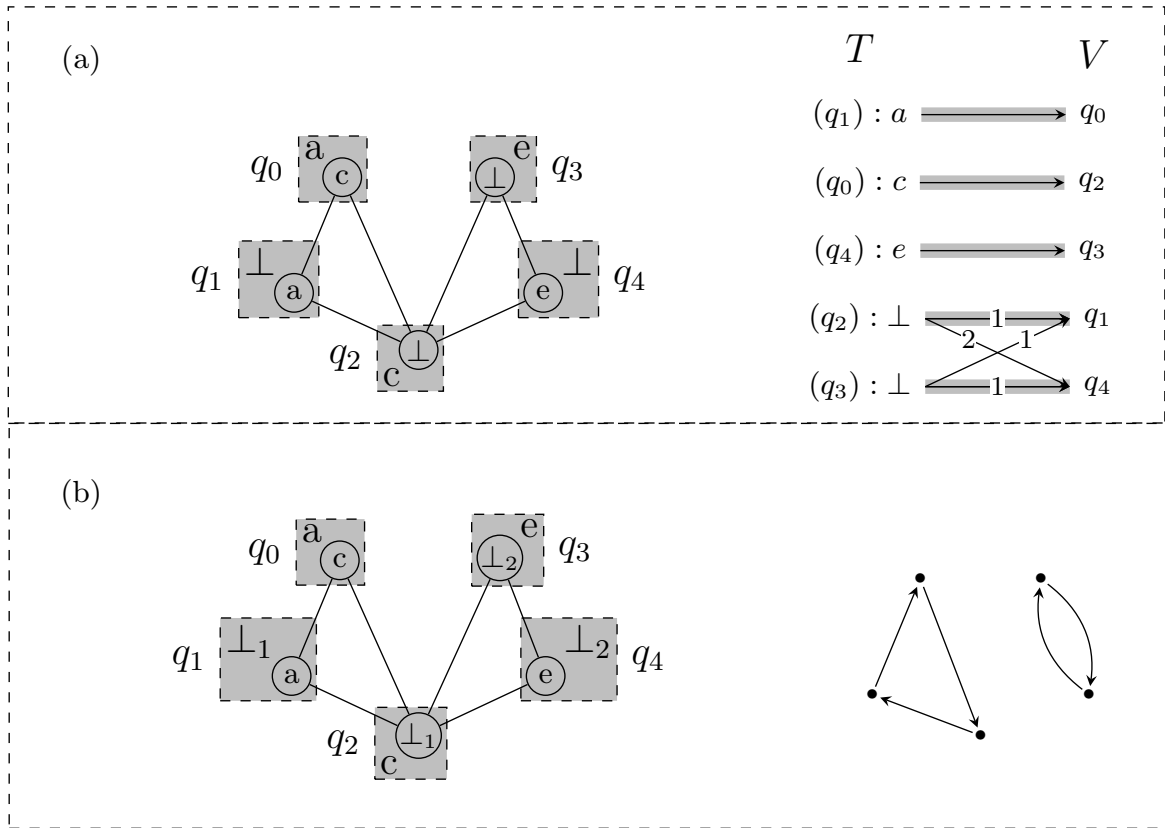
**Example 24.** *Figure 4.6 (a) shows an illustration of the bipartite graph described above. In this case, there are two tokens  $t$ , in vertices  $q_3, q_2$  such that  $|f_{tgt}(t)| > 1$ , denoted by  $\perp$ . Their set of target vertices  $f_{tgt}(t)$  consists in  $\{q_1, q_4\}$ , also denoted by  $\perp$ . On the right, we have the bipartite graph  $B$ , showing the final matching highlighted after the application of the minimal matching algorithm.*

After the minimum weighted matching, we will have a matching  $M = \{t_1 \mapsto v_1, \dots, t_n \mapsto v_n\}$ . This yielded matching will be the new  $f_{tgt}$  of the problem. In the end, each token will have a unique target vertex. Thus, we will have an instance of the narrower problem discussed earlier.

**Example 25.** *Figure 4.6 (b) shows an the modified initial token configuration, where each token has only one possible target vertex. We distinguish the formerly introduced two  $\perp$  tokens, renaming them to  $\perp_1, \perp_2$ , respectively. From this modified token configuration on the left, it is straightforward building the swap graph  $S$  on the right.*



**Figure 4.5.** (i)~(v) iterations of the execution of Miltzow et al. [2016] algorithm. On the left, there are the inputs to the algorithm: ( $T$ ) the token set  $a, \dots, e$ ; ( $f_{src}$ ) the current configuration of tokens – inside the circles; ( $f_{tgt}$ ) the target configuration of tokens – outside the circles. On the right, there is the respective Swap Chain Graph to each iteration. Bold edges represent the swaps taken.



**Figure 4.6.** (a) initial token configuration on the left, and the bipartite graph  $B$  on the right. Unused physical qubits and tokens  $t$  that  $|f_{tgt}(t)| > 1$  are represented as  $\perp$ . (b) the modified initial token configuration on the left, where we replaced the two  $\perp$  tokens for  $\perp_1, \perp_2$ , so that they have only one target vertex. On the right, there is the resulting swap graph  $S$ .



# Chapter 5

## Conclusion

This dissertation has introduced a new model for Qubit Allocation, based on two well-known problems: subgraph isomorphism and token swapping. We believe that these two problems provide a new principled approach to solve Qubit Allocation, in a way similar to what graph coloring has done to classic register allocation. And, just like there exist many solutions to register allocation based on graph coloring, several solutions to qubit allocation based on this combination of subgraph isomorphism and token swapping are possible. In this paper we have explored one among such possibilities; hence, implementing an algorithm that outperforms all previous state-of-the-art solutions to this problem in all three dimensions: weighted cost (20%); depth (25%); and number of gates (14%). The design and implementation of other solutions to Qubit Allocation, also based on the Isomorphism-Swapping combination, is an interesting research direction that we hope to explore in the future.





# Bibliography

- Amy, M., Maslov, D., Mosca, M., and Roetteler, M. (2013). A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 32(6):818--830. ISSN 0278-0070.
- Balensiefer, S., Kregor-Stickles, L., and Oskin, M. (2005). An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures. In *ISCA*, pages 186--196, Washington, DC, USA. IEEE.
- Barenco, A., Bennett, C. H., Cleve, R., DiVincenzo, D. P., Margolus, N., Shor, P., Sleator, T., Smolin, J. A., and Weinfurter, H. (1995). Elementary gates for quantum computation. *Physical review A*, 52(5):3457.
- Bellman, R. (1958). On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87-90.
- Bellman, R. E. (2003). *Dynamic Programming*. Dover Publications, Inc., New York, NY, USA.
- Benioff, P. (1980). 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. ISSN 1572-9613.
- Bonnet, E., Miltzow, T., and Ryzewski, P. (2016). Complexity of token swapping and its variants. *CoRR*, arXiv:1607.07676.
- Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. (1981). Register allocation via coloring. *Computer Languages*, 6:47--57.
- Codina, J. M., Sánchez, J., and González, A. (2001). A unified modulo scheduling and register allocation technique for clustered processors. In *Parallel Architectures and Compilation Techniques*, pages 175--184, Los Alamitos, CA, USA. IEEE.

- Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC*, pages 151--158, New York, NY, USA. ACM.
- Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. (2004). A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367--1372. ISSN 0162-8828.
- Cross, A. W., Bishop, L. S., Smolin, J. A., and Gambetta, J. M. (2017). *Open Quantum Assembly Language*. IBM, Armonk, NY, USA.
- Deutsch, D. (1985). Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 400(1818):97--117. ISSN 0080-4630.
- Devitt, S. J. (2016). Performing quantum computing experiments in the cloud. *Phys. Rev. A*, 94(3):032329.
- Devoret, M. H., Wallraff, A., and Martinis, J. M. (2004). Superconducting qubits: A short review. *arXiv*, cond-mat/0411174:1--41.
- Gambetta, J. M., Chow, J. M., and Steffen, M. (2017). Building logical qubits in a superconducting quantum computing system. *NPJ Quantum Mechanics*, 3.
- Gay, S. J. (2006). Quantum programming languages: Survey and bibliography. *Mathematical Structures in Computer Science*, 16(4):581--600.
- Gil, D. (2017). The future of computing: Ai and quantum. Online video.
- Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., and Valiron, B. (2013). Quipper: a scalable quantum programming language. In *SIGPLAN Notices*, volume 48, pages 333--342, New York, NY, USA. ACM.
- Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing, STOC '96*, pages 212--219, New York, NY, USA. ACM.
- Han, W.-S., Lee, J., and Lee, J.-H. (2013). Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337--348, New York, NY, USA. ACM.
- Häner, T., Steiger, D. S., Svore, K. M., and Troyer, M. (2016). A software methodology for compiling quantum programs. *CoRR*, abs/1604.01401:1--14.

- He, H. and Singh, A. K. (2008). Graphs-at-a-time: Query language and access methods for graph databases. In *SIGMOD*, pages 405--418, New York, NY, USA. ACM.
- IBM (2018a). Ibm qx devices. <https://quantumexperience.ng.bluemix.net/qx/devices>. [Online; accessed 13-November-2018].
- IBM (2018b). Qiskit terra. <https://github.com/QISKit/qiskit-terra>. [Online; accessed 13-November-2018].
- Javadi-Abhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F. T., and Martonosi, M. (2014). ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Computing Frontiers*, page 1, New York, NY, USA. ACM.
- Kawahara, J., Saitoh, T., and Yoshinaka, R. (2017). The time complexity of the token swapping problem and its parallel variants. In *WALCOM*, pages 448--459, Switzerland. Springer.
- Koch, J., Yu, T. M., Gambetta, J., Houck, A. A., Schuster, D. I., Majer, J., Blais, A., Devoret, M. H., Girvin, S. M., and Schoelkopf, R. J. (2007). Charge-insensitive qubit design derived from the cooper pair box. *Phys. Rev. A*, 76(1):04319.
- Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1;2):83--97.
- Lao, L., van Wee, B., Ashraf, I., van Someren, J., Khammassi, N., Bertels, K., and Almudever, C. G. (2018). Mapping of lattice surgery-based quantum circuits on surface code architectures.
- Li, G., Ding, Y., and Xie, Y. (2018). Tackling the qubit mapping problem for nisq-era quantum devices.
- Lidar, D. A. and Brun, T. A. (2013). *Quantum error correction*. Cambridge University Press, Cambridge, UK.
- Lin, C. C., Sur-Kolay, S., and Jha, N. K. (2015). Paqcs: Physical design-aware fault-tolerant quantum circuit synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(7):1221--1234. ISSN 1063-8210.
- Lin, Y., Yu, B., Li, M., and Pan, D. Z. (2018). Layout synthesis for topological quantum circuits with 1-d and 2-d architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1574--1587. ISSN 0278-0070.
- Lomont, C. (2003). Quantum circuit identities. *CoRR*, arXiv:quant-ph/0307111:1--6.

- Maslov, D. (2017). Basic circuit compilation techniques for an ion-trap quantum machine. *New Journal of Physics*, 19(2):023035.
- Maslov, D., Falconer, S. M., and Mosca, M. (2008). Quantum circuit placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):752–763. ISSN 0278-0070.
- Miltzow, T., Narins, L., Okamoto, Y., Rote, G., Thomas, A., and Uno, T. (2016). Approximation and hardness of token swapping. In *ESA*, pages 66:1–66:15, Dagstuhl, Germany. Schloss Dagstuhl.
- Mohseni, M., Read, P., Neven, H., Boixo, S., Denchev, V., Babbush, R., Fowler, A., Smelyanskiy, V., and Martinis, J. (2017). Commercialize early quantum technologies. *Nature*, 543(7644):171.
- Nielsen, M. A. and Chuang, I. (2000). *Quantum computation and quantum information*. Cambridge University Press, Cambridge, UK.
- Pedram, M. and Shafaei, A. (2016). Layout optimization for quantum circuits with linear nearest neighbor architectures. *IEEE Circuits and Systems Magazine*, 16(2):62–74. ISSN 1531-636X.
- Petit, J. (2003). Experiments on the minimum linear arrangement problem. *J. Exp. Algorithmics*, 8:1–33.
- Selinger, P. (2004). A brief survey of quantum programming languages. In *Functional and Logic Programming*, pages 61–69, Heidelberg, Germany. Springer.
- Shafaei, A., Saeedi, M., and Pedram, M. (2014). Qubit placement to minimize communication overhead in 2D quantum architectures. In *ASP-DAC*, pages 495–500, Washington, D.C., USA. IEEE.
- Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509.
- Shrivastwa, R. R., Datta, K., and Sengupta, I. (2015). Fast qubit placement in 2d architecture using nearest neighbor realization. In *iNIS*, pages 95–100, New York, NY, USA. IEEE.
- Siraichi, M. Y., Santos, V. F. d., Collange, S., and Pereira, F. M. Q. (2018). Qubit allocation. In *CGO*, pages 113–125, New York, NY, USA. ACM.

- Smith, R. S., Curtis, M. J., and Zeng, W. J. (2017). A practical quantum instruction set architecture. *arXiv*, arXiv:1608.03355:1--15.
- Surynek, P. (2018). Finding optimal solutions to token swapping by conflict-based search and reduction to sat.
- Svore, K. M., Aho, A. V., Cross, A. W., Chuang, I., and Markov, I. L. (2006). A layered software architecture for quantum computing design tools. *Computer*, 39(1):74--83.
- Tucci, R. R. (1999). A rudimentary quantum compiler (2nd ed.). *arXiv*, quant-ph/9902062:1--25.
- Wecker, D. and Svore, K. M. (2014). LIQUi|>: A software design architecture and domain-specific language for quantum computing. *arXiv*, quant-ph:1402.4467:1--14.
- Wille, R., Große, D., Teuber, L., Dueck, G. W., and Drechsler, R. (2008). Revlib: An online resource for reversible functions and reversible circuits. In *ISMVL*, pages 220--225, New York, NY, USA. IEEE.
- Wolsey, L. A. (2008). Mixed integer programming. *Encyclopedia of Computer Science and Engineering*, Online(ecse244):-.
- Yamanaka, K., Demaine, E. D., Horiyama, T., Kawamura, A., Nakano, S.-i., Okamoto, Y., Saitoh, T., Suzuki, A., Uehara, R., and Uno, T. (2017). Sequentially swapping colored tokens on graphs. In Poon, S.-H., Rahman, M. S., and Yen, H.-C., editors, *WALCOM: Algorithms and Computation*, pages 435--447, Heidelberg, Germany. Springer.
- Yamanaka, K., Demaine, E. D., Ito, T., Kawahara, J., Kiyomi, M., Okamoto, Y., Saitoh, T., Suzuki, A., Uchizawa, K., and Uno, T. (2014). *Swapping Labeled Tokens on Graphs*, pages 364--375. Springer, Switzerland.
- Yamanaka, K., Horiyama, T., Kirkpatrick, D., Otachi, Y., Saitoh, T., Uehara, R., and Uno, Y. (2015). Swapping colored tokens on graphs. In Dehne, F., Sack, J.-R., and Stege, U., editors, *Algorithms and Data Structures*, pages 619--628, Heidelberg, Germany. Springer.
- Zhao, P. and Han, J. (2010). On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1-2):340--351. ISSN 2150-8097.
- Zulehner, A., Paler, A., and Wille, R. (2018). An efficient methodology for mapping quantum circuits to the ibm qx architectures. *Transactions on Computer Aided Design of Integrated Circuits and Systems*.

Zulehner, A. and Wille, R. (2018). Compiling  $\text{su}(4)$  quantum circuits to ibm qx architectures.

# Appendix A

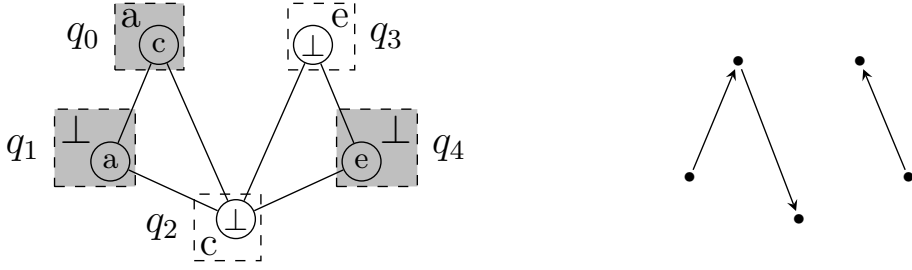
## Adapting the TWP Algorithm for BMT.

In the algorithm proposed in this work, *BMT*, we needed to merge two different mappings with swaps (Section 2.3.3). Therefore we decided to use the algorithm described in Section 4.4.1 for the colored version of the token swapping problem. That is because sometimes there will be “partially defined mapping functions”. i.e. pseudo-qubits mapped to  $\perp$ . In this context, every different pseudo-qubit is a token. Given two mappings  $h_{from}, h_{to} : P \rightarrow Q$  and the coupling graph  $G_q^u$  from *BMT*, the following equations describe how we transform them into the inputs for this problem (see Section 4.4 for more information).

$$\begin{aligned} G &= G_q^u \\ T &= P \\ U_{from} &= \{v \mid v \in Q \wedge \nexists t \in T(h_{from}(t) = v)\} \\ U_{to} &= \{v \mid v \in Q \wedge \nexists t \in T(h_{to}(t) = v)\} \\ f_{src}(t) &= \begin{cases} v \in U_{from} & \text{if } h_{from}(t) = \perp \\ h_{from}(t) & \text{else} \end{cases} \\ f_{tgt}(t) &= \begin{cases} U_{to} & \text{if } h_{to}(t) = \perp \\ h_{to}(t) & \text{else} \end{cases} \end{aligned}$$

### A.1 Optimizing for Partial Mappings

There is, however, room for optimization in this algorithm. In this case, specifically, the sets of every  $t \in T$  such that  $|f_{tgt}(t)| > 1$  are the same, i.e.  $U_{to}$ . That means



**Figure A.1.** On the left, we have the same initial configuration as Figure 4.6.  $(T)$  is the token set  $\{a, c, e, \perp, \perp\}$ ;  $(f_{src})$  is the current configuration of tokens – inside the circles;  $(f_{tgt})$  is the target configuration of tokens – outside the circles. On the right, we have the corresponding swap graph  $S$  ignoring the  $\perp$  tokens.

that we can skip the minimum weighted matching, since we do not care where the pseudo-qubits that are mapped to  $\perp$  are, as long as in the end, they are in one of the unused vertices  $U_{to}$ . Therefore, if every vertex that is not mapped to  $\perp$  is already in its target vertex, then it means that every other vertex is also in one of its target vertices. Summarizing, we can skip the execution of the minimum weighted matching, and ignore those pseudo-qubits in the swap graph  $S$  (defined in Section 4.4), turning them into edgeless vertices.

**Example 26.** Figure A.1 shows the swap graph  $S$  (on the right) that was built while ignoring tokens  $t$  such that  $|f_{tgt}(t)| > 1$ , denoted by  $\perp$  (on the left). As expected, on the left, we have the initial token configuration, where whenever (in vertices  $\{q_2, q_3\}$ ) there is a  $\perp$  token a white box is used. On the right, we lack the edges  $\{(q_2, q_1), (q_3, q_4)\}$ , since their source is one of the vertices that holds the  $\perp$  tokens.

**Theorem 5** (Optimization Correctness). *Given that the sets of every  $t \in T$  such that  $|f_{tgt}(t)| > 1$  are the same ( $U_{to}$ ) and that the swap graph  $S$  is built ignoring the vertices in it, the 4-approximative algorithm without the minimum weighted matching step is also 4-approximative.*

**Proof.** Given two vertices  $v_i, v_j \in V$  and the tokens in them  $t_i, t_j \in T$ , it is possible to divide the unhappy swaps into two different groups: (i) unhappy swaps  $(v_i, v_j)$  where  $|f_{tgt}(t_j)| = 1$ ; and (ii) unhappy swaps  $(v_i, v_j)$  where  $|f_{tgt}(t_j)| > 1$ . For the (i) unhappy swaps, it is easy to show that (Lemma 3):

$$|happy| \geq |unhappy_i|$$

For the (ii) unhappy swaps, they reduce 1 from  $L$ , but do not really add 1. The fact that the edge  $(v_i, v_j) \in E_S$  exists indicates that  $t_i$  will be closer to its



target after the swap (subtracting of 1 from  $L$ ). After the swap, even though  $v_j$  had out-degree 0 before, the vertex that  $t_j$  is now in ( $v_i$ ) will have out-degree 0, since, by definition, we are ignoring tokens that  $|f_{tgt}(t_j)| > 1$ . Therefore:

$$|happy| + |unhappy_{ii}| < L$$

Thus, the number of swaps is:

$$\begin{aligned} |swaps| &= |unhappy_i| + |happy| + |unhappy_{ii}| \\ |swaps| &= |unhappy_i| + L \\ |swaps| &= 2 * L \end{aligned}$$

Since the lower bound for the number of swaps is still  $\frac{L}{2}$ , we have that our solution is at most 4 times the optimal.

$$|swaps| = 2 * L = \frac{L}{2} * \alpha \implies \alpha = 4$$

□