# Qubit Allocation as a Combination of Subgraph Isomorphism and Token Swapping

MARCOS YUKIO SIRAICHI, Universidade Federal de Minas Gerais, Brazil
VINÍCIUS FERNANDES DOS SANTOS, Universidade Federal de Minas Gerais, Brazil
CAROLINE COLLANGE, Inria, Univ Rennes, CNRS, IRISA, France
FERNANDO MAGNO QUINTÃO PEREIRA, Universidade Federal de Minas Gerais, Brazil

In 2016, the first quantum processors have been made available to the general public. The possibility of programming an actual quantum device has elicited much enthusiasm. Yet, such possibility also brought challenges. One challenge is the so called Qubit Allocation problem: the mapping of a virtual quantum circuit into an actual quantum architecture. There exist solutions to this problem; however, in our opinion, they fail to capitalize on decades of improvements on graph theory. In contrast, this paper shows how to model qubit allocation as the combination of Subgraph Isomorphism and Token Swapping. This idea has been made possible by the publication of an approximative solution to the latter problem in 2016. We have compared our algorithm against five other qubit allocators, all independently designed in the last two years, including the winner of the IBM Challenge. When evaluated in "Tokyo", a quantum architecture with 20 qubits, our technique outperforms these state-of-the-art approaches in terms of the quality of the solutions that it finds and the amount of memory that it uses, while showing practical runtime.

CCS Concepts: • **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → **Compilers**; • **Theory of computation** → *Parameterized complexity and exact algorithms.*

Additional Key Words and Phrases: Quantum computing, Qubit allocation, Graph isomorphism, Token swapping

**ACM Reference Format:**
Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. 2019. Qubit Allocation as a Combination of Subgraph Isomorphism and Token Swapping. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 120 (October 2019), 29 pages. https://doi.org/10.1145/3360546

## 1 INTRODUCTION

The recent introduction of cloud access to quantum computers has made experimental quantum computing available to a wide community [Devitt 2016]. For instance, the IBM Quantum Experience program[1] 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]. 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

[1]http://research.ibm.com/ibm-q/

Authors' addresses: Marcos Yukio Siraichi, Universidade Federal de Minas Gerais, Brazil, yukio.siraichi@dcc.ufmg.br; Vinícius Fernandes dos Santos, Universidade Federal de Minas Gerais, Brazil, viniciussantos@dcc.ufmg.br; Caroline Collange, Inria, Univ Rennes, CNRS, IRISA, France, caroline.collange@inria.fr; Fernando Magno Quintão Pereira, Universidade Federal de Minas Gerais, Brazil, fernando@dcc.ufmg.br.

2475-1421/2019/10-ART120
https://doi.org/10.1145/3360546

5 and 16 qubits, although 20 and 50-qubit machines have 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.

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]; *circuit compilation* [Itoko et al. 2019] and *qubit allocation* [Siraichi et al. 2018; Tannu and Qureshi 2019]. Henceforth, we shall adopt the latter terminology. There exist different solutions to qubit allocation [Itoko et al. 2019; 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 modeled via graph coloring [Chaitin et al. 1981; Pereira and Palsberg 2005], 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]. In this paper, we show how to extend this model to general circuits. Such extension has been made possible by the recent contribution of Miltzow et al. [2016], who introduced approximations for a problem called *Token Swapping*. The key insight of this paper is the observation that the combination of subgraph isomorphism and token swapping completely models qubit allocation.

Based on this insight, this paper proposes a parameterized polynomial-time algorithm that deconstructs qubit allocation as the combination of subgraph isomorphism and token swapping. As we explain in Section 3, this deconstruction has two advantages: first, it simplifies the understanding of qubit allocation, as it gives us the opportunity to revisit it under well-known ground. Second, it gives us also the chance to use already well-established heuristics and approximations to solve qubit allocation. Notice, however, that although these two problems are key to solve qubit allocation, they are not enough: some algorithmic equipment is necessary to bind them together. In particular, we propose a dynamic programming approach to find the optimal solution to the combination of these two problems. This approach is parameterized, so as to bound the number of times we apply it; hence, making it practical. Additionally, we provide a way to estimate the solution of token swapping; thus, avoiding multiple applications of an expensive algorithm. Once we find a promising candidate solution, we use Miltzow et al. [2016] approximation of token swapping to produce a definitive solution to qubit allocation.

Section 4 presents an empirical evaluation of our ideas. This evaluation is another contribution of this paper, which compares our technique against five state-of-the-art qubit allocators. In this comparison, we include the algorithm currently used in the IBM Quantum Experience Toolkit; the exhaustive search of Zulehner et al. [2018], the hill-climbing algorithm of Li et al. [2018], the dedicated compiler of Zulehner and Wille, which won IBM's QISKit Developer Challenge [Zulehner and Wille 2019] in 2018, and the best-effort heuristic of Siraichi et al. [2018]. Experiments on quantum programs publicly available show that our algorithm is able to generate more efficient quantum circuits than other approaches. As an illustration, we produce code for the benchmarks collected by Zulehner et al. [2018] that is 20% more cost effective than Li et al.'s allocator –one of the best algorithms available today. To allow the reproducibility of our experiments, we have made all the implementations used in this paper accessible through an on-line interface, where they can be directly tried: http://cuda.dcc.ufmg.br/enfield/. This compiler, Enfield, contains several ready-to-use backends, including `ibmqx2` "Tenerife" and `ibmqx3` "Albatross". Additionally, users can enter new back-ends in a JSON file, if they use Enfield's standalone distribution.

## 2 BACKGROUND

This section introduces the three problems to be discussed throughout this paper: qubit allocation (Sec. 2.1); subgraph isomorphism (Sec. 2.2); and token swapping (Sec. 2.3).

**Qubits and Quantum Gates.** Quantum programs are made of qubits and reversible quantum gates, which receive qubits as inputs, and produce qubits as outputs. The semantics of quantum programs can be expressed in terms of linear algebraic operations, and is commonly represented through the visual abstraction of quantum circuits. Figure 1 shows an example of quantum circuit.

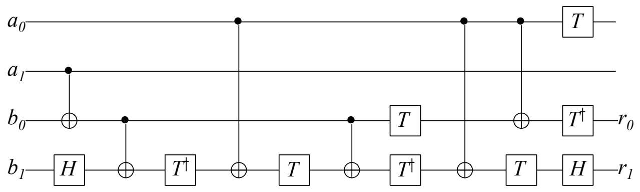


Fig. 1. Typical representation of a quantum program.

The circuit in Figure 1 has four pseudo qubits: $a_0$, $a_1$, $b_0$, represented as horizontal lines. It uses four different types of gates to operate on these qubits: $H$, $T$, $T^\dagger$ and CNOT. The gate CNOT$_{ab}$ is represented as a vertical line connecting $a$ to $b$. We say that a qubit $q$ encodes a superposition of the two classical states 0 and 1. Its state $|q\rangle$ consists of a two dimensional complex vector:

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

Values $|0\rangle$ and $|1\rangle$ are the basis states of a 2D vector space, and $\alpha$ and $\beta$ are complex numbers. A set of $n$ qubits encodes a superposition of $2^n$ classical bit vectors of size $n$, whose state consists of a $2^n$-dimension complex vector. In this terminology, quantum gates are understood as unitary matrix operations applied on vectors that describe quantum states. Example 2.1 illustrates this view.

*Example 2.1.* 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:

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

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 applies 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:

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

A *CNOT* gate applies 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 2-line vectors that represent each individual qubit. Example 2.2 illustrates how pairs of qubits are combined.

*Example 2.2.* If $|a\rangle$ and $|b\rangle$ are the states of qubits $a$ and $b$, then their combined state is given by the tensor product:

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

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 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. Example 2.3 illustrates this view.

*Example 2.3.* The first column in Figure 1 (*CNOT* and *H*) corresponds to the matrix product:

$$(I \otimes \text{CNOT} \otimes H)(|a_0a_1b_0b_1\rangle)$$

The empty wire over the qubit $a_0$ represents the identity matrix $I$, i.e. the absence of any gate.

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 paper.
**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 interactions between physically connected qubits [Devoret et al. 2004]. Thus, technology restricts the possible shapes of the *couplings graph*, a structure whose definition we revisit below:

*Definition 2.4 (Coupling Graph [Gambetta et al. 2017]).* 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 $q_1 \longrightarrow q_2 \in E_q$ if, and only if, $\text{CNOT}_{q_1q_2}$ is valid in $A$.

## 2.1 Qubit Allocation

Problem 2.5 (Qubit Allocation). ***Input:*** *a (directed) coupling graph $G_q^d = (Q, E_q)$, a list $\Psi = (P \times P)^n, n \geq 1$ of $n$ control relations between pseudo qubits $p \in P$, an integer $k \geq 0$, 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 $G_q^d$ with transformations whose total cost does not exceed $k$.*

In its simplest form [Siraichi et al. 2018, Def-2.4], qubit allocation is equivalent to the problem of finding an isomorphism between the coupling graph and the graph formed by the CNOT relations in the quantum circuit. However, no isomorphism might exist. When such is the case, the compiler must use *transformations* to fit a quantum circuit into the architecture. A quantum transformation consists of additional gates inserted in the circuit. Definition 2.6 shows the transformations $\Theta$ that we shall consider in this work. We consider only two operations: reversals and swaps, which Figure 2 illustrates. As Definition 2.5 states, transformations have a cost, which is ultimately determined by the gates necessary to implement them. In Section 4, we shall provide concrete examples of such costs. Example 2.7 shows one possible result for the allocation of the program shown in Figure 1.

*Definition 2.6.* A transformation is a sequence of quantum operations that changes the mappings of pseudo qubits onto physical qubits. We recognize two 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 (Fig. 2-a).
- **Swap:** exchanges two pseudo qubits $p_a$ and $p_b$, at the expense of three CNOT and four Hadamard gates (Fig 2-c).

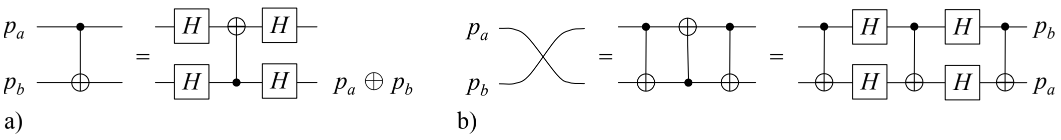


Fig. 2. (a) Reversal. (c) Swap.

*Example 2.7.* Given the coupling graph in Figure 3 (a), Figure 3 (c) shows a possible allocation of the program in Figure 1. Notice that when solving qubit allocation we can disregard single-qubit gates: only CNOT gates matter. These gates are represented as ordered pairs. Mapping $f = \{a_0 \mapsto q_0, a_1 \mapsto q_4, b_0 \mapsto q_2, b_1 \mapsto q_1\}$ plus two reversals yields the circuit in Fig. 3 (c), which has the same semantics as the input quantum program in Fig. 3 (b).

**Qubit Allocation and Register Allocation** bear similarities; however, the latter is about allocating the –unlimited– virtual registers in the program to the –limited– registers in the architecture. In contrast, in quantum computing virtual qubits are also limited: we can only have as many as there are physical qubits in the architecture. The constraint is rather on the allowed interactions between qubits. It is impossible to copy the state of qubits due to the no-cloning theorem [Wootters and Zurek 1982]; hence, we have to map and route them throughout the execution of the program.

## 2.2 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 2.9.

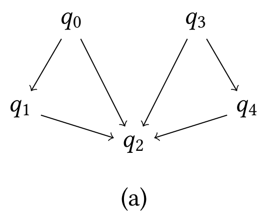
PROBLEM 2.8 (SUBGRAPH ISOMORPHISM – SIP). ***Input:*** *undirected graphs G and H.* ***Output:*** *yes, if we can find a subgraph H′ of H, plus a bijection* $f : V(G) \longrightarrow V(H')$*, where for every edge* $(u, v) \in E(G)$, $(f(u), f(v)) \in E(H')$.

*Example 2.9.* Figure 4 shows different solutions to an instance of subgraph isomorphism. Given the two undirected graphs in Figure 4 (a), $G$ and $H$ (left to right), Figure 4 (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.

SIP is NP-complete [Cook 1971]. Different heuristics have been proposed to solve it [Cordella et al. 2004; Han et al. 2013; Zhao and Han 2010]. As Section 3.1 will discuss, applying these heuristics would compromise the scalability of our algorithm. Therefore, we shall propose a parameterized



$(a_1, b_0)$
$(b_0, b_1)$
$(a_0, b_1)$
$(b_0, b_1)$
$(a_0, b_1)$
$(a_0, b_0)$

(b)

$(q_4, q_2)$
$rev(q_2, q_1)$
$(q_0, q_1)$
$rev(q_2, q_1)$
$(q_0, q_1)$
$(q_0, q_2)$

(c)

Fig. 3. (a) the coupling graph; (b) the input quantum circuit; (c) the allocated quantum circuit (output) with two uses of the reversal transformation (highlighted).

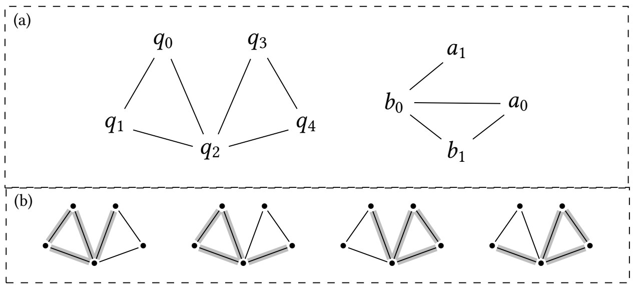


Fig. 4. (a) two undirected graphs $G$ and $H$ (left to right); (b) all possible subgraphs of $G$ isomorphic to $H$.

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 a optimal solution to SIP. In other words, parameterization will let us exchange optimality for time.

## 2.3 Token Swapping

Token Swapping is another central element to our solution to qubit allocation. This problem was introduced by Yamanaka et al. [2014] in 2014, and proven to be NP-hard by Miltzow et al. [2016]. For completeness, we restate the definition of this problem below:

PROBLEM 2.10 (TOKEN SWAPPING - TWP). ***Input:*** *a set of colors $C$, an integer $k$, an undirected graph $G = (V, E)$, a bijective function $f_v : V \longrightarrow C$ representing a coloring of $G$, plus a bijective function $f_t : V \longrightarrow C$ that assigns colored tokens to vertices.* ***Output:*** *yes, if we can match the colors of vertices and tokens with up to $k$ swap operations, where a swap exchanges tokens in two adjacent vertices.*

Research around TWP is recent; hence, it has not been as deeply studied as SIP. 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 2.11 illustrates TWP.

*Example 2.11.* Figure 5 (a) shows both $f_v = \{q_0 \mapsto a, q_1 \mapsto b, q_2 \mapsto c, q_3 \mapsto e, q_4 \mapsto d\}$ (bigger letters outside the circle), and $f_t = \{q_0 \mapsto c, q_1 \mapsto d, q_2 \mapsto a, q_3 \mapsto e, q_4 \mapsto b\}$ (smaller letters inside the circle). Thick edges in Figure 5 (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 $f_t$ to $f_v$ in 4 steps.

As we shall see in Section 3.2, token swapping lets us glue many different instances of the qubit allocation problem. We split the bigger problem using subgraph isomorphism, and then use token

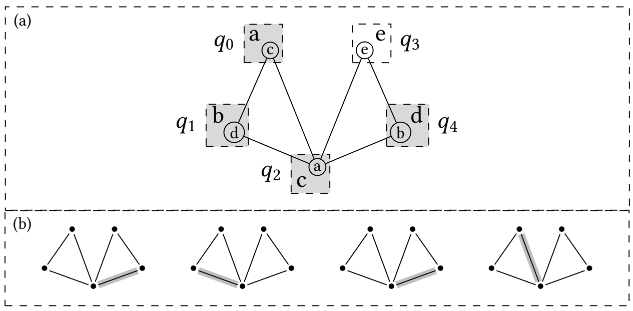


Fig. 5. (a) undirected graph of Figure 4 (a) with the labels of the vertices ($f_v$) outside the circle, the labels of the tokens ($f_t$) inside the circle. Grey boxes indicate vertices whose labels do not match tokens; (b) a sequence of swaps that solves the problem (highlighted edges).

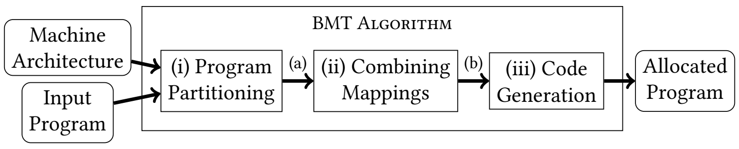


Fig. 6. Steps of the *BMT* algorithm. (a) the output of phase (i) is a partitioned program; (b) the output of phase (ii) is one mapping for each program partition.

swapping to stitch solutions together. The beauty of this approach is that this splitting meets Bellman's *Principle of Optimality* [Bellman 1958]. Hence, we can find an optimal solution to qubit allocation by uniting solutions of the smaller problems via dynamic programming.

## 3 THE ALGORITHM

This section introduces *Bounded Mapping Tree* (BMT). BMT divides qubit allocation into two subproblems: Subgraph Isomorphism (Problem 2.8) and Token Swapping (Problem 2.10). BMT searches a solution space bounded by two parameters: the *Maximum Number of Children* and the *Maximum Number of Partial Solutions*. We shall clarify in Section 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 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 6. Figure 7 summarizes the notation used in these sections.

### 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 Sublist, which Definition 3.1 introduces, and Example 3.2 illustrates.

*Definition 3.1.* [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,

- [$P$] the set of **Pseudo-Qubits** in a quantum circuit.
- [$Q$] the set of **Physical-Qubits** present in the architecture. i.e. the vertices of the coupling graph.
- [$f : P \rightarrow Q \cup \{\perp\}$] the **Mapping** from pseudo-qubits to physical-qubits. The symbol $\perp$ denotes pseudo qubits that are not mapped to any physical qubit.
- [$F : \mathcal{P}(P \rightarrow Q \cup \{\perp\})$] the **Set of Mappings**.
- [$G_q^d$] the **Directed** Coupling Graph, whose vertices ($Q$) are physical qubits. An edge from $q_1$ to $q_2$ means that $q_1$ can control $q_2$ via a CNOT gate.
- [$G_q^u$] the **Undirected** version of the Coupling Graph.
- [$\Psi : (P \times P)^k$] a list of **Control Relations**. We let $\Psi(i)$ denote the $i^{th}$ relation in this list, so that $\Psi(i) = (p_1, p_2)$ means that $p_1$ controls $p_2$, i.e., via a CNOT gate. We let $\Psi(i, j), 1 \leq i < j \leq k$ denote a sublist of $\Psi$ from $\Psi(i)$ to $\Psi(j)$. Finally, we call $\Psi_i$ the $i^{th}$ partition of $\Psi$.
- [$\mathcal{G}_\Psi$] 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 $(i, j) \in \Psi$.
- [$H \lesssim G$] a **Subgraph Isomorphism Relation** indicating that $H$ is isomorphic to some subgraph of $G$.

Fig. 7. Summary of notation used in this paper.

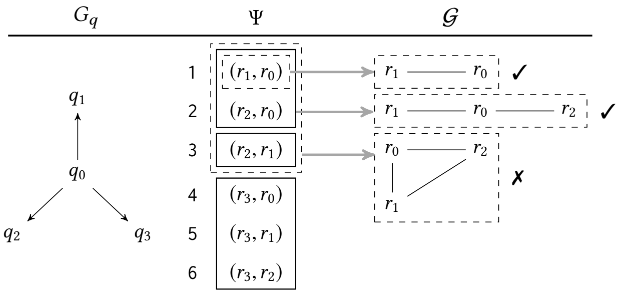


Fig. 8. From left to right, we have the coupling graph $G_q^d$, the list of control relations $\Psi$, partitioned into Maximal Isomorphic Sublists (solid boxes), and the graphs $\mathcal{G}_\Psi$ derived from sublists of $\Psi$ (dashed boxes). Next to each derived graph, we show if an isomorphism is possible (✓) or not (✗).

$\mathcal{G}_{\Psi(i,j)} \lesssim G_q^u$, 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 set of Maximal Isomorphic Sublists of $\Psi$ as $\mathcal{S}(\Psi) = \{\Psi_1, \ldots, \Psi_n\}$.

*Example 3.2.* Figure 8 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.

Problem 3.3 (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 $\Psi(1, i_1), \Psi(i_1 + 1, i_2), \ldots, \Psi(i_{k-1} + 1, i_k)$, 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 sublist 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, turning it into another partition of $\Psi$.

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 $\bot$. We then update $f$ successively for each instruction in $\Psi(i + 1, n)$, until 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 $f(p_1) = \bot$ and $f(p_2) = \bot$, then, for every free edge $(q_1, q_2) \in E(G^u_q)$, 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 $f(p_1) = \bot$ and $f(p_2) \neq \bot$ (or $f(p_1) \neq \bot$ and $f(p_2) = \bot$), then only one of the pseudo qubits needs to be mapped. Without loss of generality, let us assume that $f(p_1) = \bot$ and $f(p_2) \neq \bot$. For every $(f(p_1), q_2) \in E(G^u_q)$, 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 $f(p_1) \neq \bot$, $f(p_2) \neq \bot$ and $(f(p_1), f(p_2)) \in E(G^u_q)$, then no update is necessary. We continue recursively on $f$, $\Psi(i + x + 1, n)$.

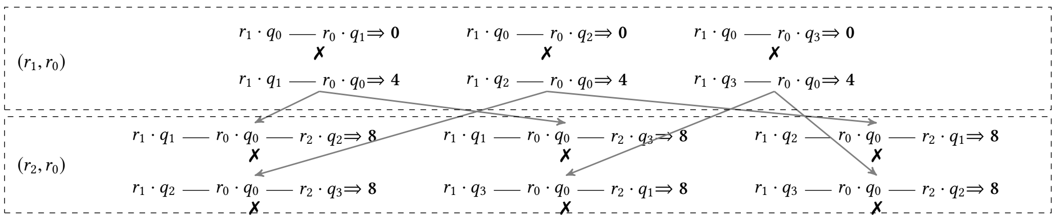
If none of these three actions is possible, then $\Psi(i, i+x-1)$ defines another partition as a maximal isomorphic sublist. In this case, we create a set of mappings $F_{k'+1}$ containing all mappings that satisfy $\mathcal{G}_{\Psi(i,i+x-1)} \lesssim G^u_q$, and invoke $S_{pcr}$ over the remaining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

**The cost of a maximal isomorphic sublist.** Function $S_{pcr}$ creates a sequence of sets of mappings $F_1, F_2, \ldots, F_m, m \leq k$. 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 $G^u_q$ can be adjusted onto $G^d_q$, because we can use *reversals*, a quantum transformation introduced by Definition 2.6, 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 3.4 formalizes this cost function:

*Definition 3.4 (Cost of Mapping).* If $f$ is a mapping from pseudo qubits to physical qubits, then its cost is given by a function $C : \Psi \times F \times G^d_q \to \mathbb{R}$. Given pseudo qubits $p_1, p_2 \in P$, if $(f(p_1), f(p_2)) \in G^d_q$, then the cost is zero; otherwise, it is a constant $C_{rev}$.

*Example 3.5.* Given the coupling graph $G^u_q$ and the list $\Psi$ shown in Figure 8, we computed the set of mappings for the first two instructions. Figure 9 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 $\bot$, and all the edges in the coupling graph are free. Thus, we map $(r_1, r_0)$ to any edge. There are 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:



$(r_1, r_0)$
$r_1 \cdot q_0$ — $r_0 \cdot q_1 \Rightarrow$ 0
$r_1 \cdot q_0$ — $r_0 \cdot q_2 \Rightarrow$ 0
$r_1 \cdot q_0$ — $r_0 \cdot q_3 \Rightarrow$ 0
$r_1 \cdot q_1$ — $r_0 \cdot q_0 \Rightarrow$ 4
$r_1 \cdot q_2$ — $r_0 \cdot q_0 \Rightarrow$ 4
$r_1 \cdot q_3$ — $r_0 \cdot q_0 \Rightarrow$ 4
$(r_2, r_0)$
$r_1 \cdot q_1$ — $r_0 \cdot q_0$ — $r_2 \cdot q_2 \Rightarrow$ 8
$r_1 \cdot q_1$ — $r_0 \cdot q_0$ — $r_2 \cdot q_3 \Rightarrow$ 8
$r_1 \cdot q_2$ — $r_0 \cdot q_0$ — $r_2 \cdot q_1 \Rightarrow$ 8
$r_1 \cdot q_2$ — $r_0 \cdot q_0$ — $r_2 \cdot q_3 \Rightarrow$ 8
$r_1 \cdot q_3$ — $r_0 \cdot q_0$ — $r_2 \cdot q_1 \Rightarrow$ 8
$r_1 \cdot q_3$ — $r_0 \cdot q_0$ — $r_2 \cdot q_2 \Rightarrow$ 8

Fig. 9. 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 ✗ 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.

$q_2$ 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 $S_{pcr}$ 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 start from each mapping. The other limits the number of mappings that we can consider. Figure 10 illustrates these two forms of pruning. Pruning happens whenever we try to add a new instruction onto the current partition. We use the weight of each mapping for pruning, by using a roulette weighted selection. In other words, we select the mappings randomly, using their weights as a probability mass function.

In the end of this first phase, we have the input program sliced into up to $k$ partitions $\mathcal{S} = \Psi(1, i_1), \Psi(i_1 + 1, i_2), \ldots, \Psi(i_{k-1} + 1, i_k)$, plus the corresponding sets of mappings $\mathcal{F} = F_1, F_2, \ldots, F_n$. 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 $C$ from Definition 3.4.

**Iterating the Quantum Program.** Instructions in a quantum circuit do not have to be processed in the order defined by that circuit. *Quantum gates over non-overlapping qubits may be executed in parallel;* hence, we can change their order without changing the semantics of the program. Nevertheless, we still have to keep the gate sequence applied on each qubit. Therefore, to iterate over quantum programs, we use a directed acyclic graph (DAG) whose vertices represent gates. An edge $(g_i, g_j)$ indicates that $g_i$ should be executed before $g_j$. A topological ordering of this graph determines the valid ways to iterate over the program. Whenever multiple orderings are possible, we select the next gate to be processed based on four criteria. To explain these predicates, suppose that $(a, b)$ are pseudo-qubits linked by a candidate gate:

(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.

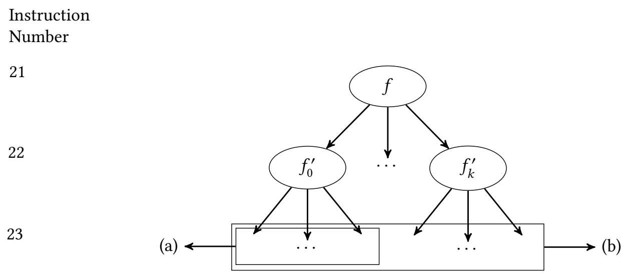


Fig. 10. Tree of mappings for one partition. The $i^{th}$ level represents the possible mappings once we add the $i^{th}$ instruction (on the left) to the mappings already in place. Since the number of leaves grows exponentially, we limit them with two parameters: $M_c$: maximum number of children of each mapping, and $M_p$: maximum number of mappings per partition.

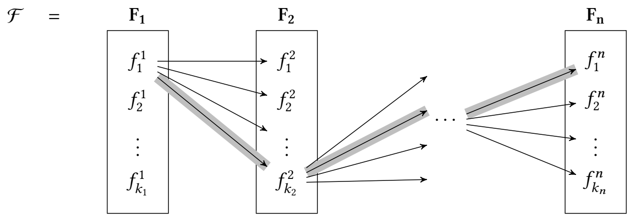


Fig. 11. Output from first phase $\mathcal{F}$ with some possible combinations represented by different paths. Each path in this figure represents one different solution. The path representing the optimal solution is highlighted.

**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 $O(M_p|\Psi||Q|)$

## 3.2 Combining Mappings via Token Swapping

In Section 3.1 we have split the list of control relations into 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 11 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 program. Each path consists of $n - 1$ *hops*. 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.

There exist $|F_i| \times |F_{i+1}|$ possible paths between two successive sets of candidates $F_i$ and $F_{i+1}$. Thus, the number of paths between $F_1$ and $F_n$ is exponentially large. Hence, to find the optimal path illustrated in Figure 11, we must solve an NP-complete problem –token swapping– an exponential number of times! Fortunately, there are ways to approximate Token-Swapping [Miltzow et al. 2016], as we discuss in Section 3.2.1; and we can handle the combinatorial explosion of paths via dynamic programming, as we explain in Section 3.2.2.

*3.2.1 Solving the Token Swapping Problem.* Recently, Miltzow et al. [2016] presented a solution for the Token Swapping Problem that is proven to be 4-approximative. 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. 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 3.6 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 to $2 \times \delta(f_{freq}, f)$. Thus, we use this upper bound as the estimation of the number of swap operations. Example 3.7 illustrates this estimate.

*Definition 3.6 (Cost of joining two successive mappings).* Let $d : Q \times Q \to \mathbb{N}$ be a function that yields the minimum number of edges between two vertices in the coupling 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 3.7.* Figure 12 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 8. 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.

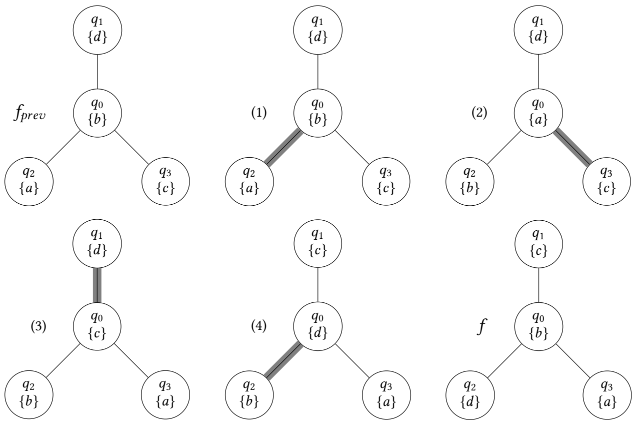


Fig. 12. Steps for transforming $f_{prev}$ into $f$, assuming the coupling graph seen in Figure 8. The pseudo qubit assigned to a physical qubit is shown in brackets. Gray edges indicate qubits that shall be swapped.

**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 to a physical qubit at every mapping $f$ related to a partition containing instructions from $\Psi(i, j)$. Otherwise, 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 3.8.* Pseudo qubit $r_0$ is alive at the second partition in Fig. 8; however, this partition only contains instruction $\Psi(3) = (r_2, r_1)$. A solution to PCR sets $f(r_0) = \perp$ 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 do it, lets 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 $f'(p) = \bot$. 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 always more pseudo qubits than physical qubits, and a pseudo qubit is never mapped onto two different physical qubits in the same partition.

*3.2.2 Dynamic Programming.* The approximations discussed in Section 3.2.1 give 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 11 illustrates. As we have already discussed, the number of paths is exponentially high, 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:

Problem 3.9 (Construction of a Complete Sequence of Transformations). ***Input:*** *a sequence of $n$ sets of mappings of pseudo to physical qubits $F_1, F_2, \ldots, F_n$, the function $C$ from Definition 3.4 and the function $\delta$ from Definition 3.6.* ***output:*** *a sequence $f_1, f_2, \ldots, f_n$, $f_i \in F_i$, which minimizes $\sum \delta(f_{i-1}, f_i) + \sum C(f_i)$.*

Problem 3.9 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 1958]. Definition 3.10 describes the dynamic programming subproblem, and Equation 1 shows its recurrence relation. In Theorem 3.11 we state and prove the correctness of our solution. Finally, Example 3.13 illustrates how we solve Problem 3.9.

*Definition 3.10.* (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 the $i$-th partition and the coupling graph) as the last mapping. i.e. it is the minimum cost using the $j$-th mapping generated for the $i$-th partition.

$$OPT(i,j) = \begin{cases} C(\Psi_i, f_j^i) & i = 1 \\ \min\limits_{0 \le k < |F_{i-1}|} \left(\delta(f_k^{i-1}, f_j^i) + OPT(i-1, k)\right) & \\ \qquad + C(\Psi_i, f_j^i) & i > 1 \end{cases} \tag{1}$$

Theorem 3.11. *The recurrence relation given by Equation 1 yields an optimal solution to Problem 3.9.*

Proof 3.12. *The proof is a case analysis on the two branches of function* OPT*:*

*(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, $\delta(f_{k'}^{i-1}, f_j^i) + OPT(i-1, k') < OPT(i,j) = \min\limits_k \delta(f_k^{i-1}, f_j^i) + OPT(i-1, k)$ must be true, a contradiction.*

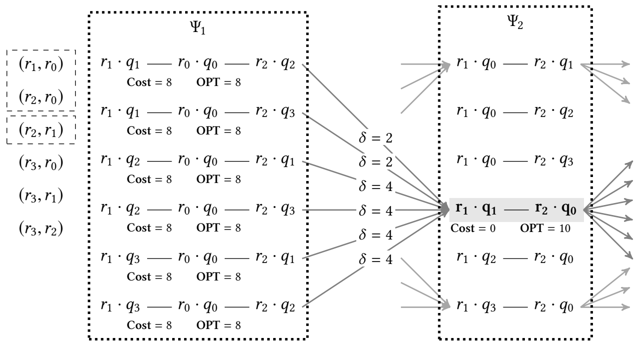


Fig. 13. Subproblem dependency for calculating *OPT* of the highlighted mapping. It shall be the minimum value of the sum of each dependency by its cost $\delta$ of transforming the previous one into the highlighted one.

*Example 3.13.* Figure 13 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 partition 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 1. The estimation of the swap cost takes $O(|Q|)$, while the cost for ensuring live qubits remain mapped is $O(|Q|(|Q| + E(G^u_q)))$, since we execute, in the worst case, one BFS for each qubit. Hence, the time complexity of this phase is $O((M_p)^2|\Psi||Q|(|Q| + E(G^u_q)))$. The space complexity is $O(M_p|\Psi||Q|)$: each subproblem is $O(|Q|)$, and we have $O(M_p|\Psi|)$ of them.

## 3.3 Code Generation

The dynamic programming algorithm discussed in Section 3.2.2 gives us a sequence of mappings $f_1, f_2, \ldots, f_n$, 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 14 shows the product of this phase.

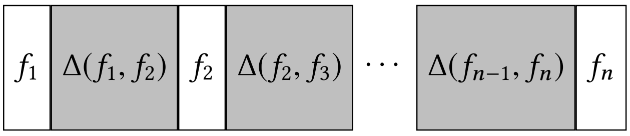


Fig. 14. Mappings $f$ and swapping sequences $\Delta$ (highlighted) give us all the information that is necessary to transform a virtual quantum circuit into a physical quantum circuit.

**From estimates ($\delta$) to concrete sequences ($\Delta$).** In Section 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 4-approximative algorithm used by Miltzow et al. [2016] to solve the Colored Token Swapping Problem. The $\delta$ function is an over-approximation of Miltzow et al.'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.'s approximation receives two mappings, $f_{prev}$ and $f$, and finds a sequence of swaps that transform $f_{prev}$ into $f$. In their original formulation, 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 3.2.1, we ensure that live qubits remain always mapped. To make provision to partially defined mappings, we introduce a projection operator $\triangledown$:

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

Instead of solving token swapping between $f_{prev}$ and $f$, we solve it between $f_{prev}$ and $f_{prev} \triangledown f$. In other words, we solve the problem only for virtual qubits which are defined in both mappings. Lemma 3.14 shows that this approach is sound. Example 3.16 illustrates this phase with the input program used in the previous sections.

Lemma 3.14. *Let $undef(f) = \{p \mid f(p) = \perp, p \in P\}$. Given the mapping $f' = f_{prev} \triangledown f$, 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 3.15. *Since $undef(f_{prev}) \supseteq undef(f)$, all pseudo-qubits mapped to a physical-qubit $q$, $q \neq \perp$, in $f_{prev}$ are defined in $f$. Thus, we have to allocate, at least, these pseudo-qubits. That is exactly what $f'$ is: a mapping of the pseudo-qubits mapped in $f_{prev}$ to their respective physical-qubits mapped in $f$.*

*Example 3.16.* Figure 15 (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 3.8. From these mappings, we are able to calculate the $\Delta$ function for each pair of consecutive partition. Figure 15 (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 [W. 1955] ($O(|Q|^3)$). Besides that, the algorithm's main loop will execute $O(|Q|)$ breadth first searches (BFS) for each swap. Thus, the complexity of the approximative algorithm is $O(|Q|^3 + |\Delta||Q|(|Q| + E(G^u_q)))$. $|\Delta|$ is bounded by the sum of the distance of misplaced qubits. Miltzow's algorithm runs once per partition ($|\Psi| - 1$ times). Thus, the complexity of this phase is $O(|\Psi|(|Q|^3 + |\Delta||Q|(|Q| + E(G^u_q))))$. 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 $O(|Q|^2)$.

## 4 EVALUATION

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

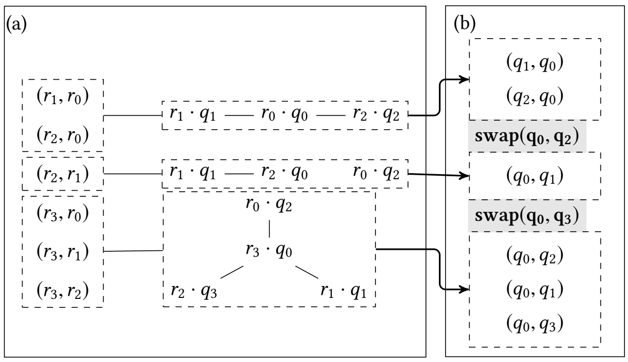


Fig. 15. (a) from left to right, we have the whole input program segmented into partitions (dashed box), and their mappings; (b) the swap operations (grey boxes) necessary to transform one mapping into another.

- **[RQ1]**: how do the parameters of our algorithm: maximum number of children and maximum number of partial solutions, affect the solution that it produces to qubit allocation?
- **[RQ2]:** how does our algorithm compare with state-of-the-art approaches in terms of quality of solution?
- **[RQ3]:** how efficient is our algorithm, in terms of space and time, compared with state-of-the-art approaches?
- **[RQ4]:** how does the target architecture influence the performance of BMT, compared to other qubit allocation algorithms?

**The algorithms used in this evaluation:** We compare *BMT* against five algorithms: IBM's python sdk (QISKit); Siraichi et al. [2018] weighted partial mapper (WPM); Zulehner et al. [2018] A* search; Li et al. [2018] SWAP-based BidiREctional heuristic (SABRE); and Zulehner and Wille [2019] dedicated SU(4) compiler, which won the IBM QISKit challenge in 2018. Henceforth, we shall refer to these algorithms as `ibm`, `wpm`, `jku`[2], `sbr` and `chw`. We emphasize that none of these algorithms is guaranteed to find a global optimum, because they all resort to some simplification of the problem, even those that run an exhaustive search, such as `jku`. In our case, we segment the program greedily, and use an approximative algorithm to solve token swapping.
**BMT Configuration:** We have selected the values for the two parameters in our algorithm empirically to obtain a good compromise between runtime and quality of solution. We shall consider two versions of *BMT*: `bmtF` and `bmtS`. The former is a *faster* version of *BMT*, which uses $M_c = 4$ and $M_p = 320$. The latter, the *slower* version, uses a more time consuming parametrization of *BMT*, featuring $M_c = 8$ and $M_p = 1,280$. As we shall discuss in Section 4.1, `bmtF` produces circuits about 1% more costly than `bmtS`; however, it runs approximately 70% faster.

For the sake of fairness when performing runtime comparisons, we have implemented all the allocators in our C++ OpenQASM open source compiler Enfield. The compiler has an online interface, publicly available at http://cuda.dcc.ufmg.br/enfield/, where users can try all the different algorithms used in our empirical evaluation. Our implementation follows the original Python version of each algorithm. We certify that results remain unchanged by comparing outputs. Our implementation is always faster and more memory frugal than the original version of each algorithm, because we use C++ instead of Python.
**Runtime Environment.** Most of the tests were executed on a dedicated server featuring an Intel(R) Xeon(TM) E5-2660 CPU @ 2.20GHz, with up to 32GB RAM, running Linux Debian Jessie

[2]We shall call Zulehner et al.'s first algorithm JKU because it was designed and tested at Johannes Kepler Universität Linz.

8.11. The experiments in Section 4.4 were executed in a commodity laptop, featuring an Intel i7 4700MQ @ 2.4GHz, with 8GB RAM, running Linux Debian Stretch 9.9. We chose a smaller machine for this last experiment to avoid paying the cost of the dedicated server.

**Benchmarks.** We used the 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 [Javadi-Abhari et al. 2014]. These suites are staple in papers on Qubit Allocation [Lao et al. 2018; Li et al. 2018; Lin et al. 2015, 2018; Pedram and Shafaei 2016; Shafaei et al. 2014; Zulehner et al. 2018]. We have opted to use these benchmarks, instead of those recently used in the IBM Challenge because whereas the latter are randomly generated, our benchmarks are actual quantum circuits. We believe that the randomly produced patterns from IBM are not a good reflex of typical quantum circuits, because they show regularity that we cannot find in human-produced programs. Our collection, on the other hand, is more varied, because it was taken from three different sources, and even within a single source, programs were written by more than one person.

**Target architecture.** The algorithms compared in this section work for any quantum architecture. In this paper we will use `ibmqx20`, "Tokyo" [IBM 2016]. To the best of our knowledge, Tokyo is the largest quantum computer publicly available as of February 2019. In Section 4.4, we shall experiment with the `ibmqx3`, "Albatross", which has 16 qubits, instead of 20.

**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[IBM 2016], we let the cost of each CNOT be 10, and of each single-qubit gate be 1. As discussed by Zulehner and Wille [Zulehner and Wille 2019, Sec.5], 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. 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 [Li et al. 2018; Pedram and Shafaei 2016; Shafaei et al. 2014; Shrivastwa et al. 2015; Zulehner et al. 2018].
- **Depth:** the depth is closely related to the time a program takes to terminate. It is measured as the longest path that must be traversed to run a quantum circuit. This metric has also being used in previous work [Amy et al. 2013; Maslov et al. 2008; Zulehner et al. 2018]. Theoretically, given a quantum computer that executes operations simultaneously for disjoint sets of qubits, a program 2x deeper is likely to be 2x slower. The main benefit of running quantum programs faster is to minimize noise due to decoherence effects, which are exponential with respect to time. A 2x speedup would theoretically enable a quadratic improvement for decoherence-induced error.

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

- **Memory Consumption:** peak memory that each qubit allocation uses to process a quantum circuit, gauged by reading `/proc/self/status`.
- **Allocation Time:** the time that each allocator needs to process a quantum circuit.

**Summary of Results:** Figure 16 summarizes the comparison between our approach and the five algorithms we consider. Assuming that $R_{foo}$ and $R_{bmt}$ correspond respectively to a given algorithm and to `bmtS`, results are given by the ratio $R_{foo}/R_{bmt}$. Each algorithm is represented by a major row. Sub-rows (cost, depth, gates, mem and time) represent dimensions of quality and efficiency. In what follows, we shall analyze the five dimensions of efficiency that we have reported in Figure 16: cost, depth, gates, memory and time. The important message from Figure 16 is that `bmtS` outperforms all the other algorithms in terms of cost, depth and number of gates that it produces on average.

| Allocator. | Dim | G. Mean | G. Std. D. | B. Count | B. Mean | W. Count | W. 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 |
| 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 |
| | time | 0.4634 | 2.3381 | 125 (79.11%) | 0.3293 | 33 (20.89%) | 1.6889 |
| | mem | 0.9258 | 1.0778 | 134 (84.81%) | 0.9089 | 24 (15.19%) | 1.026 |
| 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 |
| | time | 0.0445 | 3.2652 | 155 (98.1%) | 0.04 | 3 (1.9%) | 10.8607 |
| | mem | 1.0201 | 1.762 | 105 (66.46%) | 0.9021 | 52 (32.91%) | 1.3081 |
| 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 |
| | time | 0.1193 | 2.3326 | 158 (100%) | 0.1193 | 0 (0%) | - |
| | mem | 1.0544 | 1.1246 | 65 (41.14%) | 0.9483 | 93 (58.86%) | 1.1356 |
| 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 |
| | time | 0.0112 | 3.106 | 158 (100%) | 0.0112 | 0 (0%) | - |
| | mem | 0.971 | 1.0903 | 101 (63.92%) | 0.9233 | 56 (35.44%) | 1.0628 |

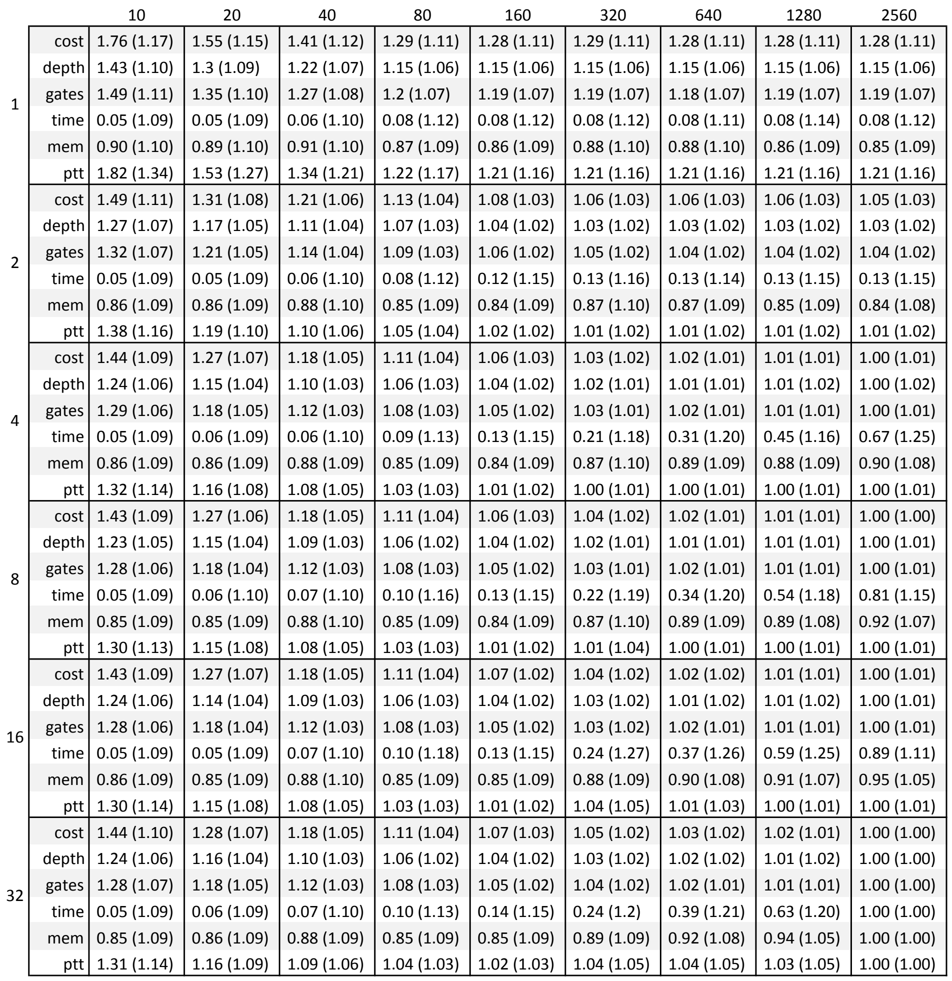
Fig. 16. Summary of the comparison between bmtS and other algorithms. **Allocator:** algorithm that we compare with bmtS. **Dim:** dimensions of efficiency and quality. **G. Mean:** geometric mean of ratios, taking bmtS as baseline. **G. Std. D.:** the geometric standard deviation of the ratios. The closer to 1, the smaller the spread of the data. **B.[etter] (W.[orse] Count:** the number of benchmarks where the algorithm was better (worse) than bmtS. We use a hyphen ("–") to mark cases where no benchmark exists. **B.[etter] (W.[orse]) G. Mean:** the geometric mean of the ratios of the benchmarks where the algorithm was better (worse) than bmtS. This column answers the following question: "what would be the G. Mean for this allocator, considering only the benchmarks where it performed better (worse) than bmtS?" Except for Better Count (**B. Count**), the higher the reported value, the better for bmtS.

Moreover, its closest competitor is bmtF –another contribution of this paper. Li et al.'s SABRE (sbr) yields the next best results; however, its cost is still 25% higher than bmtS.

## 4.1 RQ1: The Influence of Parameters

As we explained in Section 3.1, the search that BMT carries out on the universe of solutions is bounded by two parameters: $M_c$, the maximum number of children; and $M_p$, the maximum number of partial solution. To observe how these parameters impact the quality of BMT's solutions, and the runtime of this algorithm, we have performed an exhaustive batch of experiments with seven different values of $M_c$ and nine different values of $M_p$, for a total of 54 combinations. The values that we chose for each parameter are: $M_c$: 1, 2, 4, 8, 16, 32, and $M_p$: 10, 20, 40, 80, 160, 320, 640, 1280, 2560.

We run BMT three times for each combination of parameters, for each one of the 158 available benchmarks. Figure 17 summarizes this result. In this figure, we compare each combination of parameters against our BMT with these two parameters, e.g., $M_p$ and $M_c$, set to 32 and 2,560, respectively. Larger values are possible, but operating with them becomes very time consuming, and we chose not to do it. We shall call this version of our algorithm $BMT_{top}$. In other words, Figure 17 reports ratios in which each dimension of efficiency is compared against that same metric when $M_c = \infty$ and $M_p = \infty$. Each cell of Figure 17 contains two values: a ratio (taking $BMT_{top}$ as the baseline) and that ratio's standard deviation



| | | 10 | 20 | 40 | 80 | 160 | 320 | 640 | 1280 | 2560 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | cost | 1.76 (1.17) | 1.55 (1.15) | 1.41 (1.12) | 1.29 (1.11) | 1.28 (1.11) | 1.29 (1.11) | 1.28 (1.11) | 1.28 (1.11) | 1.28 (1.11) |
| | depth | 1.43 (1.10) | 1.3 (1.09) | 1.22 (1.07) | 1.15 (1.06) | 1.15 (1.06) | 1.15 (1.06) | 1.15 (1.06) | 1.15 (1.06) | 1.15 (1.06) |
| | gates | 1.49 (1.11) | 1.35 (1.10) | 1.27 (1.08) | 1.2 (1.07) | 1.19 (1.07) | 1.19 (1.07) | 1.18 (1.07) | 1.19 (1.07) | 1.19 (1.07) |
| | time | 0.05 (1.09) | 0.05 (1.09) | 0.06 (1.10) | 0.08 (1.12) | 0.08 (1.12) | 0.08 (1.12) | 0.08 (1.11) | 0.08 (1.14) | 0.08 (1.12) |
| | mem | 0.90 (1.10) | 0.89 (1.10) | 0.91 (1.10) | 0.87 (1.09) | 0.86 (1.09) | 0.88 (1.10) | 0.88 (1.10) | 0.86 (1.09) | 0.85 (1.09) |
| | ptt | 1.82 (1.34) | 1.53 (1.27) | 1.34 (1.21) | 1.22 (1.17) | 1.21 (1.16) | 1.21 (1.16) | 1.21 (1.16) | 1.21 (1.16) | 1.21 (1.16) |
| 2 | cost | 1.49 (1.11) | 1.31 (1.08) | 1.21 (1.06) | 1.13 (1.04) | 1.08 (1.03) | 1.06 (1.03) | 1.06 (1.03) | 1.06 (1.03) | 1.05 (1.03) |
| | depth | 1.27 (1.07) | 1.17 (1.05) | 1.11 (1.04) | 1.07 (1.03) | 1.04 (1.02) | 1.03 (1.02) | 1.03 (1.02) | 1.03 (1.02) | 1.03 (1.02) |
| | gates | 1.32 (1.07) | 1.21 (1.05) | 1.14 (1.04) | 1.09 (1.03) | 1.06 (1.02) | 1.05 (1.02) | 1.04 (1.02) | 1.04 (1.02) | 1.04 (1.02) |
| | time | 0.05 (1.09) | 0.05 (1.09) | 0.06 (1.10) | 0.08 (1.12) | 0.12 (1.15) | 0.13 (1.16) | 0.13 (1.14) | 0.13 (1.15) | 0.13 (1.15) |
| | mem | 0.86 (1.09) | 0.86 (1.09) | 0.88 (1.10) | 0.85 (1.09) | 0.84 (1.09) | 0.87 (1.10) | 0.87 (1.09) | 0.85 (1.09) | 0.84 (1.08) |
| | ptt | 1.38 (1.16) | 1.19 (1.10) | 1.10 (1.06) | 1.05 (1.04) | 1.02 (1.02) | 1.01 (1.02) | 1.01 (1.02) | 1.01 (1.02) | 1.01 (1.02) |
| 4 | cost | 1.44 (1.09) | 1.27 (1.07) | 1.18 (1.05) | 1.11 (1.04) | 1.06 (1.03) | 1.03 (1.02) | 1.02 (1.01) | 1.01 (1.01) | 1.00 (1.01) |
| | depth | 1.24 (1.06) | 1.15 (1.04) | 1.10 (1.03) | 1.06 (1.03) | 1.04 (1.02) | 1.02 (1.01) | 1.01 (1.01) | 1.01 (1.02) | 1.00 (1.02) |
| | gates | 1.29 (1.06) | 1.18 (1.05) | 1.12 (1.03) | 1.08 (1.03) | 1.05 (1.02) | 1.03 (1.01) | 1.02 (1.01) | 1.01 (1.01) | 1.00 (1.01) |
| | time | 0.05 (1.09) | 0.06 (1.09) | 0.06 (1.10) | 0.09 (1.13) | 0.13 (1.15) | 0.21 (1.18) | 0.31 (1.20) | 0.45 (1.16) | 0.67 (1.25) |
| | mem | 0.86 (1.09) | 0.86 (1.09) | 0.88 (1.09) | 0.85 (1.09) | 0.84 (1.09) | 0.87 (1.10) | 0.89 (1.09) | 0.88 (1.09) | 0.90 (1.08) |
| | ptt | 1.32 (1.14) | 1.16 (1.08) | 1.08 (1.05) | 1.03 (1.03) | 1.01 (1.02) | 1.00 (1.01) | 1.00 (1.01) | 1.00 (1.01) | 1.00 (1.01) |
| 8 | cost | 1.43 (1.09) | 1.27 (1.06) | 1.18 (1.05) | 1.11 (1.04) | 1.06 (1.03) | 1.04 (1.02) | 1.02 (1.01) | 1.01 (1.01) | 1.00 (1.00) |
| | depth | 1.23 (1.05) | 1.15 (1.04) | 1.09 (1.03) | 1.06 (1.02) | 1.04 (1.02) | 1.02 (1.01) | 1.01 (1.01) | 1.01 (1.01) | 1.00 (1.01) |
| | gates | 1.28 (1.06) | 1.18 (1.04) | 1.12 (1.03) | 1.08 (1.03) | 1.05 (1.02) | 1.03 (1.01) | 1.02 (1.01) | 1.01 (1.01) | 1.00 (1.01) |
| | time | 0.05 (1.09) | 0.06 (1.10) | 0.07 (1.10) | 0.10 (1.16) | 0.13 (1.15) | 0.22 (1.19) | 0.34 (1.20) | 0.54 (1.18) | 0.81 (1.15) |
| | mem | 0.85 (1.09) | 0.85 (1.09) | 0.88 (1.10) | 0.85 (1.09) | 0.84 (1.09) | 0.87 (1.10) | 0.89 (1.09) | 0.89 (1.08) | 0.92 (1.07) |
| | ptt | 1.30 (1.13) | 1.15 (1.08) | 1.08 (1.05) | 1.03 (1.03) | 1.01 (1.02) | 1.01 (1.04) | 1.00 (1.01) | 1.00 (1.01) | 1.00 (1.01) |
| 16 | cost | 1.43 (1.09) | 1.27 (1.07) | 1.18 (1.05) | 1.11 (1.04) | 1.07 (1.02) | 1.04 (1.02) | 1.02 (1.02) | 1.01 (1.01) | 1.00 (1.01) |
| | depth | 1.24 (1.06) | 1.14 (1.04) | 1.09 (1.03) | 1.06 (1.03) | 1.04 (1.02) | 1.03 (1.02) | 1.01 (1.02) | 1.01 (1.02) | 1.00 (1.01) |
| | gates | 1.28 (1.06) | 1.18 (1.04) | 1.12 (1.03) | 1.08 (1.03) | 1.05 (1.02) | 1.03 (1.02) | 1.02 (1.01) | 1.01 (1.01) | 1.00 (1.01) |
| | time | 0.05 (1.09) | 0.05 (1.09) | 0.07 (1.10) | 0.10 (1.18) | 0.13 (1.15) | 0.24 (1.27) | 0.37 (1.26) | 0.59 (1.25) | 0.89 (1.11) |
| | mem | 0.86 (1.09) | 0.85 (1.09) | 0.88 (1.10) | 0.85 (1.09) | 0.85 (1.09) | 0.88 (1.09) | 0.90 (1.08) | 0.91 (1.07) | 0.95 (1.05) |
| | ptt | 1.30 (1.14) | 1.15 (1.08) | 1.08 (1.05) | 1.03 (1.03) | 1.01 (1.02) | 1.04 (1.05) | 1.01 (1.03) | 1.00 (1.01) | 1.00 (1.01) |
| 32 | cost | 1.44 (1.10) | 1.28 (1.07) | 1.18 (1.05) | 1.11 (1.04) | 1.07 (1.03) | 1.05 (1.02) | 1.03 (1.02) | 1.02 (1.01) | 1.00 (1.00) |
| | depth | 1.24 (1.06) | 1.16 (1.04) | 1.10 (1.03) | 1.06 (1.02) | 1.04 (1.02) | 1.03 (1.02) | 1.02 (1.02) | 1.01 (1.02) | 1.00 (1.00) |
| | gates | 1.28 (1.07) | 1.18 (1.05) | 1.12 (1.03) | 1.08 (1.03) | 1.05 (1.02) | 1.04 (1.02) | 1.02 (1.01) | 1.01 (1.01) | 1.00 (1.00) |
| | time | 0.05 (1.09) | 0.06 (1.09) | 0.07 (1.10) | 0.10 (1.13) | 0.14 (1.15) | 0.24 (1.2) | 0.39 (1.21) | 0.63 (1.20) | 1.00 (1.00) |
| | mem | 0.85 (1.09) | 0.86 (1.09) | 0.88 (1.09) | 0.85 (1.09) | 0.85 (1.09) | 0.89 (1.09) | 0.92 (1.08) | 0.94 (1.05) | 1.00 (1.00) |
| | ptt | 1.31 (1.14) | 1.16 (1.09) | 1.09 (1.06) | 1.04 (1.03) | 1.02 (1.03) | 1.04 (1.05) | 1.04 (1.05) | 1.03 (1.05) | 1.00 (1.00) |

Fig. 17. Exploration of the space of parameters. Rows group results by the maximum number of children ($M_c$), and columns group results by maximum number of partial solutions ($M_p$). Row ptt denotes the number of partitions in which we split the quantum circuit, given a certain set of parameters.

**Analysis of results.** As expected, the quality of the solution improves as we increase both $M_c$ and $M_p$. That is to say, the higher the values of $M_c$ and $M_p$, the lower the allocation cost produced by BMT. A bit smaller configuration, $M_c = 8$ and $M_p = 1,280$, yield allocation costs within 2% of

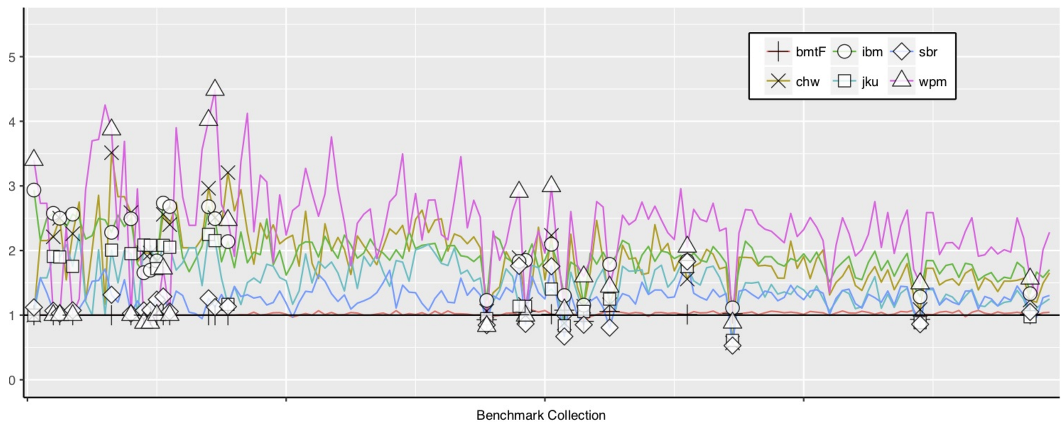


Fig. 18. (**Cost**) Ratio of the cost of circuits produced by the different allocators using the cost of the circuit produced by bmtS as the baseline. The Y-axis shows the ratio of the weighted cost. The X-axis shows the benchmarks –each tick represents a different quantum circuit. Less is better. If we sum up the costs of all the 158 circuits, then we obtain the following absolute numbers, in ascending order: bmtS=24.5M, bmtF=25.3M, jku=29.6M, sbr=30.3M, chw=36.8M, ibm=40.3M and wpm=50.0M. Figure 16 reports averages.

$BMT_{top}$. However, raising $M_p$ to 2,560 already gives results comparable to $BMT_{top}$ when $M_c = 4$, but in 67% of $BMT_{top}$'s time. If we accept to produce circuits that are 3% more costly, then we can reduce the runtime of our algorithm even more: Setting $M_c = 4$ and $M_p = 320$ we remain within this bound, but run in 21% of the time taken by $BMT_{top}$, on average.

## 4.2 RQ2: Quality of Allocation

Figure 18 compares the different allocators in terms of the circuits that they produce. Analyzing this figure in tandem with Figure 16, we can draw the following conclusions:

- **Weighted Cost:** bmtS yields the lowest cost. sbr, jku, chw, ibm, and wpm were 25%, 47%, 83%, 91%, and 125% worse than our algorithm, respectively. The faster version of BMT, e.g. bmtF, also outperforms the other algorithms. They were, respectively, 22%, 44%, 80%, 87% and 120% worse than bmtF. bmtF was only 2% worse than bmtS;
- **Depth:** bmtS yields the smallest depth on average. sbr, jku, chw, ibm, and wpm were 35%, 44%, 92%, 76%, and 116% worse than our algorithm, respectively. The faster version of BMT, e.g. bmtF, also outperforms the other algorithms. They were, respectively, 32%, 41%, 88%, 73% and 113% worse than bmtF. bmtF was only 1.7% worse than bmtS;
- **Gates:** bmtS yields the smallest number of gates. sbr, jku, chw, ibm, and wpm were 16%, 27%, 47%, 55%, and 75% worse than our algorithm, respectively. The faster version of BMT, e.g. bmtF, also outperforms the other algorithms. They were, respectively, 14%, 25%, 45%, 52% and 72% worse than bmtF. bmtF was only 1.8% worse than bmtS;

**Analysis of results.** bmtS wins in cost, depth and gates, followed closely by bmtF. Both algorithms outperform sbr, jku, ibm, and wpm by a large margin. BMT's parameters allow users to, given enough time and space, find better and better solutions. The fact that our algorithm outperforms the others in terms of cost, depth and gates indicates that we generate circuits with smaller error rates (given by the weighted costs), but also faster circuits (given by the depth). These results show that BMT is able to yield good programs even with the maximum number of children, $M_c$, limited

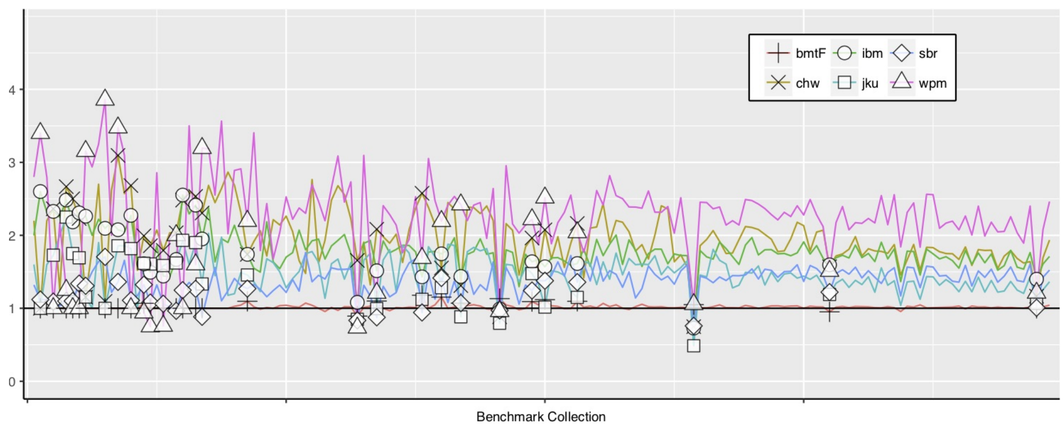


Fig. 19. (**Depth**) Ratio of the depth of circuits produced by the different allocators, using bmtS as the baseline. The Y-axis shows the ratio of the weighted cost. The X-axis shows the benchmarks. Less is better. If we sum up the depth of all the 158 circuits, then we obtain the following absolute sum of longest paths in the circuit, in ascending order: bmtS=2.2M, bmtF=2.3M, jku=2.9M, sbr=3.2M, ibm=3.7M, chw=3.8M and wpm=4.7M. See Figure 16 for the averages.

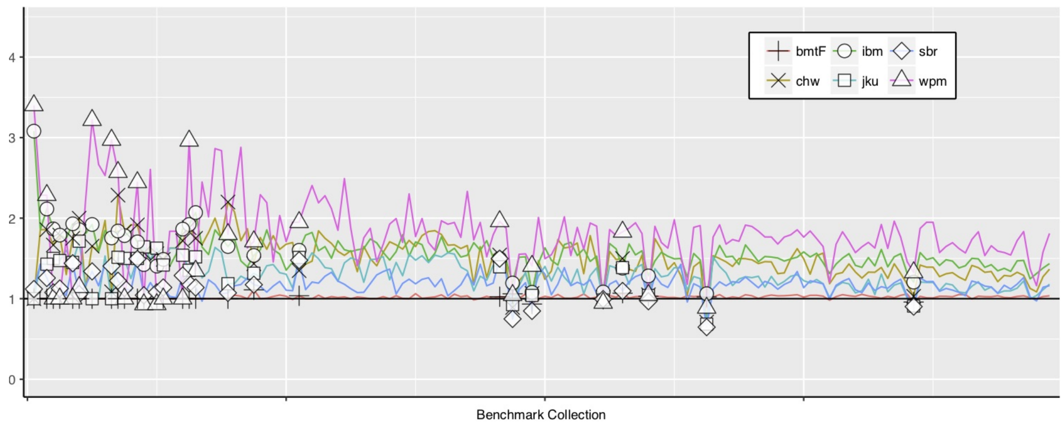


Fig. 20. (**Gates**) Ratio of the number of gates in the circuits produced by the different allocators by the cost of the circuit produced by bmtS. The Y-axis shows the ratio of the weighted cost. The X-axis shows the benchmarks. Less is better. If we sum up the gates used in all the 158 circuits, then we obtain the following absolute number of gates, in ascending order: bmtS=4.21M, bmtF=4.31M, jku=4.72M, sbr=4.75M, chw=5.38M, ibm=5.9M and wpm=6.79M. For the corresponding averages, see Figure 16.

to 8 (out of 96 possible children), and the maximum number of mappings, $M_p$, restricted to 1,280 (out of an exponentially large universe).

Figures 18, 19 and 20 show results for individual benchmarks, and provide absolute numbers for our entire collection of 158 benchmarks. Notice that bmtS and bmtF still outperform the other approaches to qubit allocation in terms of absolute numbers. However, the relative order between these allocators change, given that some are more or less susceptible to worst cases due to the influence of outliers. There are a few cases in which all the other algorithms can beat bmtS. even though BMT has the potential to find optimal solutions to qubit allocation, when left unbounded,

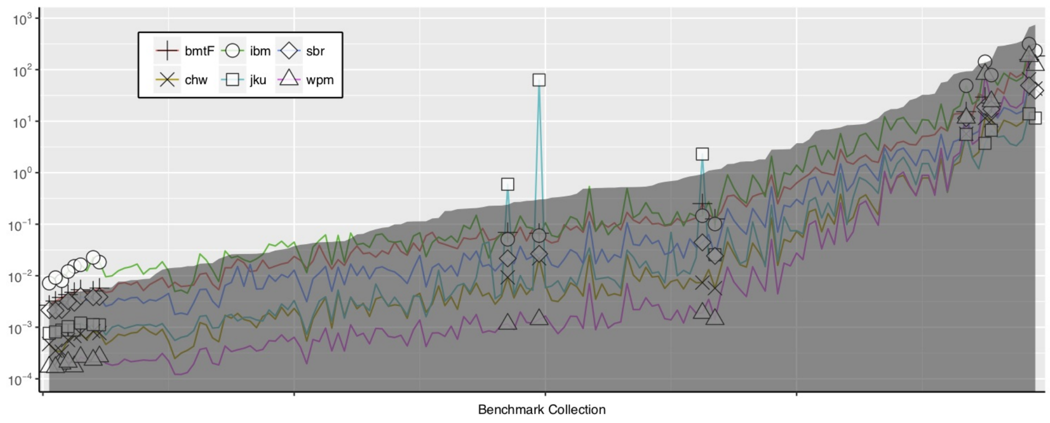


Fig. 21. (**Time**) Time spent by different allocators. The Y-axis shows time (seconds) in logarithmic scale. The X-axis shows benchmarks ordered in increasing order of the time spent by `bmt` (shaded area). If we sum up the time, in seconds, used during the allocation of all the 158 circuits, then we obtain the following absolute numbers, in ascending order: `jku`=157secs, `chw`=198secs, `sbr`=275secs, `wpm`=549secs, `bmtF`=883secs, `ibm`=1,405secs and `bmtS`=4,247secs. For the averages, see Figure 16.

we parameterize it to keep runtime practical. Once we need to prune off mappings, we use a random roulette, weighted by the cost of each mapping. This is a heuristic, so, there might be problems that do not benefit from this approach. The hypothesis behind our pruning approach is that the smaller the number of partitions in the program, the lower its overall cost will be, because there will be less points where swaps will be necessary. However, there are quantum circuits in which this hypothesis is false. In these cases, even though we are glueing together a smaller number of partitions, more swaps might be necessary between them.

## 4.3 RQ3: Efficiency of the Algorithm

Figure 21 shows the time that each allocator takes to map pseudo- into physical-qubits. Figure 22, in turn, compares the allocators in terms of memory usage. From these figures, plus Figure 16, we draw the following conclusions:

- **Allocation Time:** `bmtS` is the slowest algorithm. `sbr`, `jku`, `chw`, `ibm`, and `wpm` took 11%, 4.4%, 2.8%, 46%, and 1.1% the time needed by our algorithm, respectively. The faster version of BMT, e.g. `bmtF`, was the third slowest one. All the other allocators, except `bmtS` and `ibm` that took, respectively, 244% and 59% more, were faster. `sbr`, `jku`, `chw`, and `wpm` took 41%, 15%, 9.8%, and 3.8% the time needed by it, respectively;
- **Memory Consumption:** On average, there is not much variation memory-wise: `sbr` and `jku` consumed 5.4% and 2.0% more memory, respectively, and `chw`, `ibm`, `wpm` and `bmtF` consumed, on average, 6%, 8%, 3% and 3.6% less than `bmtS`. As expected, `bmtF` consumed a bit less memory than `bmtS`. `sbr`, `jku`, and `wpm` consumed 9.3%, 5.7%, and 0.6% more; and `ibm` and `chw` consumed 4.1% and 2.4% less.

**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 100 secs. The other algorithms stay under 10 secs per sample. The slowest part of BMT is phase two's dynamic programming search. Dynamic programming accounts for, on average (geometric mean), 50% of the time. The first phase

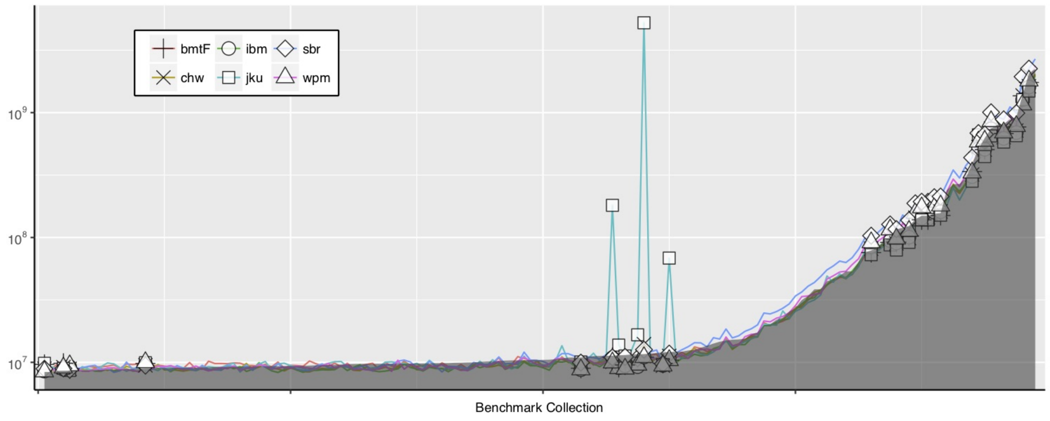


Fig. 22. (**Memory**) Memory used by different allocators. The Y-axis shows the memory (bytes) in logarithmic scale. The X-axis shows benchmarks. Benchmarks are ordered in increasing order based on the memory used by bmtS (shaded area). If we sum up the memory, in bytes, used during the allocation of all the 158 circuits, then we obtain the following absolute numbers, in ascending order: ibm=13.8GB, chw=14.1GB, bmtF=14.1GB, wpm=15.3GB, bmtS=15.7GB, jku=18.6GB and sbr=18.7GB. For the averages, see Figure 16.

comes next, with 37% of the time. In spite of its higher computational complexity, jku runs faster than bmtF and bmtS.

Unsurprisingly, wpm is the fastest algorithm, on average. This implementation uses a best-effort heuristic that is linear on the size of quantum circuits and on the number of qubits. However, once we consider absolute runtimes, wpm suffers from outliers, and takes longer than other several other algorithms to finish. For instance, wpm is faster than sbr, on average, but it takes twice its total time to run. The reason behind this apparently paradox is that the geometric mean hides these outliers (10 out of 158). The absolute time of these 10 benchmarks accounts for 95% of the total runtime of wpm, and for 77% of sbr's.

The jku implementation 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 22 indicates that the memory usage of wpm, sbr, Q_ibm, and both BMTs grows linearly with the number of control relations; the same cannot be inferred for jku, which contains several outliers.

We close this section summarizing the data produced to answer research questions **RQ2** and **RQ3**. Figure 23 subsumes these results. Interestingly, once we consider average runtime and costs, we find that all the algorithms, except for ibm fit into the optimal convex hull of time vs cost. In other words, following this path wpm $\longrightarrow$ chw $\longrightarrow$ jku $\longrightarrow$ sbr $\longrightarrow$ bmtF $\longrightarrow$ bmtS we consistently improve allocation cost, at the expense of runtime.

## 4.4 RQ4: The Influence of the Target Architecture

The experiments reported in Sections 4.1, 4.2 and 4.3 were executed in a target architecture with 20 qubits. Figure 24 (a) shows its coupling graph. In this section, we investigate how the algorithms fare when given a different coupling graph. For this experiment, we chose Albatross[3], a retired architecture with 16 qubits, whose coupling graph Figure 24 (b) outlines. In addition of having less qubits than Tokyo, Albatross is asymmetric: connections between qubits are one-way.

[3]Choosing a smaller architecture is not possible, because several of our benchmarks use at least 16 qubits.

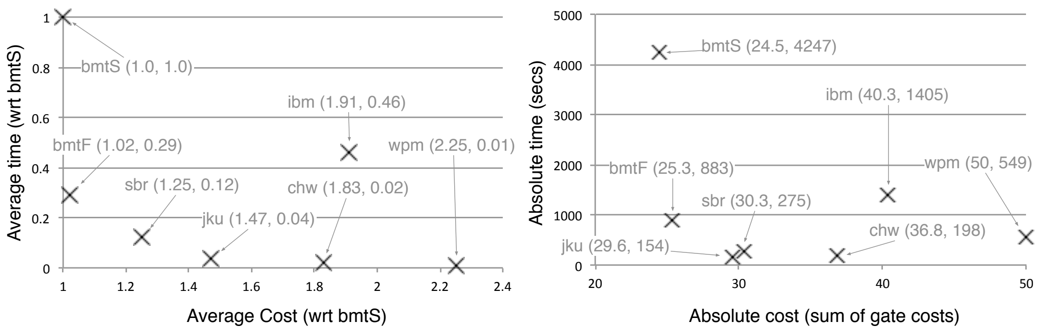


Fig. 23. Runtime vs Weighted Allocation Cost for all the algorithms compared in this evaluation. Allocation cost is measured as the sum of all the weighted costs of gates used to compose the quantum circuit after qubit allocation (result is given in millions).

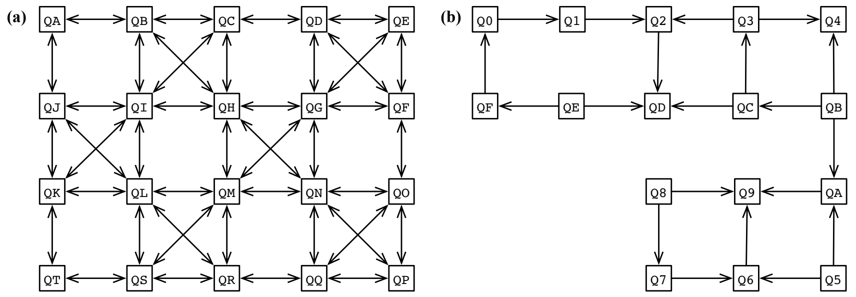


Fig. 24. Coupling graphs for: (a) ibmqx20 "Tokyo", with 20 qubits. (b) ibmqx3 "Albatross", with 16 qubits.

Figure 25 compares the performance of the seven algorithms considered in this paper, using Albatross' coupling graph. We run these experiments in a commodity laptop with 8GB of RAM, instead of the dedicated server (with 32GB of RAM) used in the three previous sections. Reason was cost: scheduling experiments in the server incurs into a monetary cost and follows a constrained time table. Furthermore, a simpler machine lets us show that our algorithms are practical.

We had to remove seven benchmarks from the experiments used to produce Figure 25, because jku could not handle them given the amount of available memory. The other six algorithms have been able to compile all the 158 benchmarks. If we include the missing seven benchmarks, then the cost of circuits produced by bmtS jumps from 37.6 million up to 58.3 million.

The jku and sbr allocators tend to produce circuits with lower cost than bmtS and bmtF, in Albatross. Adding back the seven excluded benchmarks does not change this scenario: sbr still outperforms the two instances of BMT. However, bmtS and bmtF produce circuits of shorter depth, on average, than the other algorithms, being amost 25% more efficient than sbr in this criterion. By construction, BMT is prone to produce circuits of short depth, because it tries to increase the size of each partition of the circuit. Less partitions do not necessarily, lead to lower costs, but tend to reduce circuit depth. The more connected the coupling graph, the better tends to be BMT's performance with regards to sbr. To support this statement, we have compared bmtS, bmtF, wpm and sbr using random graphs produced according to the Erdös-Rényi model [Erdös and Rényi 1959]. We omit jku, because some experiments could not terminate due to lack of memory. Graph generation takes a parameter $p$, corresponding to the probability of adding an edge. We pick up

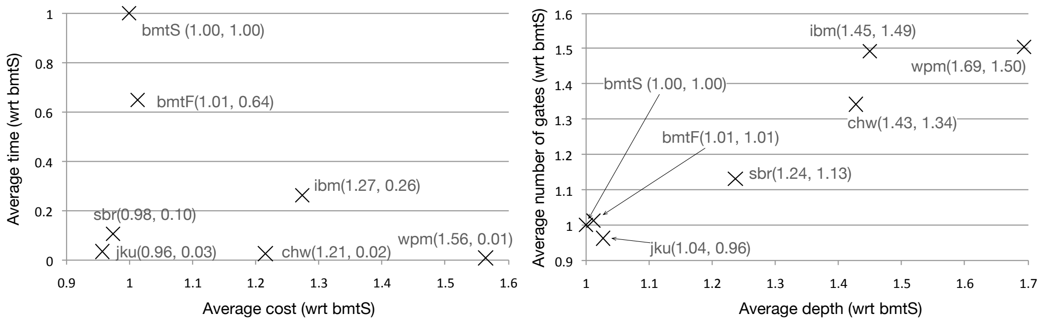


Fig. 25. Comparison between different qubit allocators in the Albatross architecture. We had to remove seven benchmarks from our collection of 158 circuits to perform this experiment, because jku requires more than 8GB of memory to compile them.

graphs that have a certain diameter, given a fixed $p$. Figure 26 shows the result of this comparison, in terms of cost and depth for graphs with 16, 20 and 24 vertices (qubits) and diameters of 2 ($p = 0.2$), 4 ($p = 0.1$), 6 ($p = 0.1$) and 8 ($p = 0.05$) edges. These results are congruous with the fact that BMT's quality is better on Tokyo than on Albatross (which has a sparser coupling graph), and also support the hypothesis that BMT reduces depth as connectivity grows.

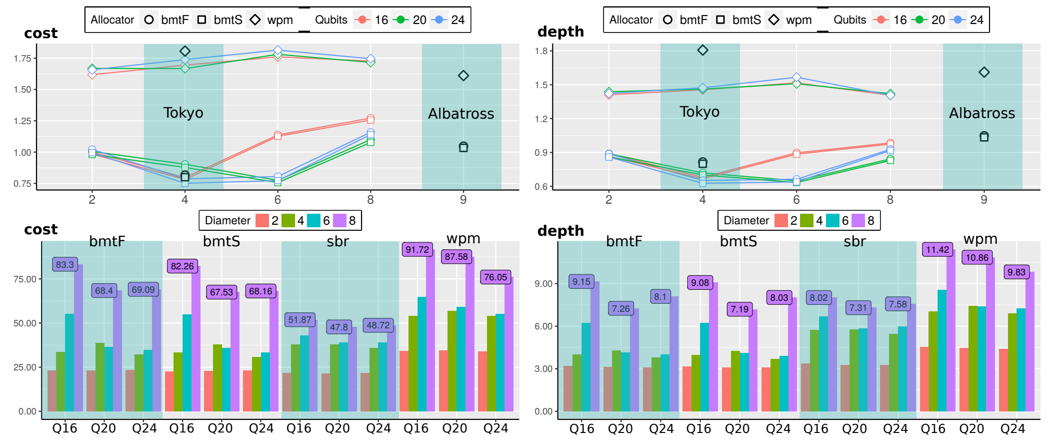


Fig. 26. Comparison of different qubit allocators in randomly produced graphs with 16, 20 and 24 qubits, and diameters of 2, 4, 6 and 8 nodes. The first row shows the results w.r.t. sbr. The black points in the highlighted areas locate the Tokyo and Albatross architectures. The second row shows the absolute results (in millions) for weighted cost (left), and depth (right) for all allocators. For each allocator, there are three groups (qubit number) of four bars (diameter). Each bar provides the result of using a given allocator on a different set of graphs.

## 4.5 Discussion: Qubit Allocation as a Tool Towards Quantum Supremacy

An important question that concerns the material covered in this paper is: “is the time required to compile and execute a circuit allocated with BMT on a quantum computer significantly less than the time required to solve the same problem with a digital computer?" In other words, we are interested in knowing if there exists a particular problem size at which point the overhead incurred by bounded mapping trees becomes negligible. The answer to this question is positive, as we will argue in this section. However, the current prototypes of quantum computers that we have access today do not let us show this benefit with numbers. Even obtaining the runtime of a quantum circuit in the IBM Qiskit is not trivial today, due to the infrastructure to access the quantum computer. Yet, we can estimate the time to compile a circuit with BMT and run it on a quantum computer versus the time to simulate the circuit.

The current record of simulation of a quantum program on a digital machine stays at 60 Qubits [Dang et al. 2019] –four more than the previous record, from October of 2018 [Pednault et al. 2018]. As mentioned by Dang *et al.*, the simulation of a 50-qubits random state might require up to 18 petabytes of classical computer memory. In both works, Dang's and Pednault's, simulation at such scale was only possible by exploring particular characteristics of the algorithm to be simulated. The consensus today is that above 50 entangled qubits, simulation of general quantum programs in classical hardware becomes an intractable problem, with prohibitively high costs. To give the reader some perspective on these numbers, Markov *et al.* simulate approximate sampling of a $7 \times 8$ qubits architecture [Markov et al. 2018]. At a fidelity rate of 0.5%, their cost (estimated as cloud service fees) to simulate a circuit with 1+40+1[4] layers of depth would be around 35 thousand dollars. A deeper circuit, e.g., with 1+48+1 layers would add one million US dollars to this baseline cost.

Execution of the same circuit on a quantum computer is likely faster. Markov et al. provide the following estimates: "*When sampling outputs on a quantum computer based on superconducting qubits, reported times are around 45ns per cycle with CNOT gates along the Z axis, 25ns per cycle with only single-qubit gates [Kelly et al. 2014]. Readout can be as fast as 140ns [Sank et al. 2014], and qubits can be initialized to a known state in a few hundred nanoseconds [Magnard et al. 2018].*"

Similarly, the time to run qubit allocation on a quantum circuit is much lower than the time to simulate it. We have compiled circuits with up to 100 qubits in commodity hardware. As an example, an 100-qubit Quantum Fourier Transform algorithm takes up to 150 and 500 seconds to be compiled by `bmtF` and `bmtS`, respectively. Thus, even though BMT runs in exponential time (when left unbounded), parameterization ensures that its execution time remains practical. Furthermore, due to material constraints like cooling to cryogenic temperature, we expect quantum computer time to remain much more expensive than classical computer time in the near future. So it is still a reasonable tradeoff to spend several seconds on commodity hardware to save milliseconds on a very expensive equipment. Finally, the benefit of reducing gate count or circuit depth –something that qubit allocation accomplishes– is not just a shorter execution time, but much more importantly, a lowered exposure to noise and decoherence, which leads to better accuracy of quantum programs.

## 5 RELATED WORK

Qubit allocation is a relatively recent issue. In 2004 Svore et al. mentioned the need to insert movement instructions, e.g., swaps, as a step in a general design flow to map a program representing a quantum computing algorithm into a technology-specific hardware. That early report did not provide an algorithm for solving qubit allocation, although the authors have addressed this challenge later [Häner et al. 2016; Svore et al. 2006]. We believe that the first formal treatment of the problem, with definitions, complexity analyses and heuristics, was proposed by Maslov et al. in 2008. Earlier works would either solve qubit allocation manually, or present allocators only for specific quantum algorithms [Blais 2001; Copsey et al. 2003; Oskin et al. 2002; Saito et al. 2000]. Qubit allocation has several variations, all of which are NP-complete [Siraichi et al. 2018]. Such diversity has led to an

[4]Markov *et al.* explicitly denote layers of Hadamard gates with "1+" at the beginning of the circuit, and "+1" at the end. The other layers include other one-qubit gates and controlling CNOT gates.

equally varied number of solutions. Our compiler, Enfield, implements the algorithms that we were aware of, and whose source code we could easily find, or whose authors were kind enough to help us. Some algorithms we decided not to implement, for the following reasons:

- **Restricted design**: some heuristics target specific quantum architectures. Architectures are determined by the shape of the coupling graphs; e.g.: 2D grids and lines [Lin et al. 2015, 2018; Pedram and Shafaei 2016; Shafaei et al. 2014; Shrivastwa et al. 2015].
- **Runtime complexity**: several heuristics suffer from a worst case exponential execution time. Contrary to our design, this runtime penalty is mandatory, i.e., it cannot be bounded via parameters [Itoko et al. 2019; Lao et al. 2018; Lin et al. 2018; Maslov et al. 2008; Shafaei et al. 2014; Zulehner et al. 2018].
- **Swaps only**: most of the solutions to qubit allocation rely on swaps as the transformation available to fit logical circuit into physical coupling graphs [Lao et al. 2018; Li et al. 2018; Lin et al. 2015; Maslov et al. 2008; Pedram and Shafaei 2016; Shafaei et al. 2014]; hence, reversals are not considered.

**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 [Bonnet et al. 2016; Kawahara et al. 2017; Miltzow et al. 2016; Yamanaka et al. 2017, 2014]. We know two concrete implementations of algorithms that solve 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. [2016], which runs in polynomial time. In contrast, subgraph isomorphism is an old problem. There are several heuristics and exact algorithms to solve it [Cordella et al. 2004; Han et al. 2013; Zhao and Han 2010]. We use a best-effort approach. Thus, the search discussed in Section 3.1 might not find a solution, even though a solution might exist.

## 6 CONCLUSION

This paper has introduced a new model for Qubit Allocation, based on 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. Our algorithm outperforms several state-of-the-art solutions to this problem along three different dimensions: weighted cost; depth; and number of gates, in a quantum architecture with 20 qubits. When we consider these metrics, our algorithm delivers better results than previous work, both in terms of averages and in terms of absolute numbers, at the expense of a longer runtime. 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.

## ACKNOWLEDGEMENT

This project was funded by grants from FAPEMIG (Grant APQ-03832-14 “Cooperation FAPs-INRIA-CNRS"), CNPq (Grant 406377/2018-9) and CAPES. We thank the referees for the time and expertise they have put into reviewing our work. Their suggestions have greatly improved our paper.

## REFERENCES

Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 32, 6 (jun 2013), 818–830.

Adriano Barenco, Charles H Bennett, Richard Cleve, David P DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Physical review A* 52, 5 (1995), 3457.

Richard Bellman. 1958. On a Routing Problem. *Quart. Appl. Math.* 16 (1958), 87–90.

Alexandre Blais. 2001. Quantum network optimization. *Phys. Rev. A* 64 (Jul 2001), 022312. Issue 2.

Édouard Bonnet, Tillmann Miltzow, and Pawel Rzazewski. 2016. Complexity of Token Swapping and its Variants. *CoRR* arXiv:1607.07676, Article 2 (2016), 23 pages.

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation via Coloring. *Comput. Lang.* 6, 1 (1981), 47–57.

Stephen A. Cook. 1971. The Complexity of Theorem-proving Procedures. In *STOC*. ACM, NY, USA, 151–158.

Dean Copsey, Mark Oskin, Tzvetan Metodiev, Frederic T. Chong, Isaac Chuang, and John Kubiatowicz. 2003. The Effect of Communication Costs in Solid-state Quantum Computing Architectures. In *SPAA*. ACM, New York, NY, USA, 65–74.

L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *TPAMI* 26, 10 (Oct 2004), 1367–1372.

Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. *Open Quantum Assembly Language*. IBM, Armonk, NY, USA.

Aidan Dang, Charles D. Hill, and Lloyd C. L. Hollenberg. 2019. Optimising Matrix Product State Simulations of Shor's Algorithm. *CoRR* 3 (2019), 116–125.

Simon J. Devitt. 2016. Performing quantum computing experiments in the cloud. *Phys. Rev. A* 94, 3 (2016), 032329.

Michel H Devoret, Andreas Wallraff, and John M Martinis. 2004. Superconducting qubits: A short review. *arXiv* 0411174 (2004), 1–41.

P. Erdös and A. Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae* 6 (1959), 290–297.

Jay M Gambetta, Jerry M Chow, and Matthias Steffen. 2017. Building logical qubits in a superconducting quantum computing system. *NPJ Quantum Mechanics* 3, Article 2 (2017), 7 pages.

Dario Gil. 2017. The Future of Computing: AI and Quantum. Online video.

Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *SIGPLAN Notices*, Vol. 48. ACM, NY, USA, 333–342.

Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *SIGMOD*. ACM, NY, USA, 337–348.

Thomas Häner, Damian S. Steiger, Krysta M. Svore, and Matthias Troyer. 2016. A Software Methodology for Compiling Quantum Programs. *CoRR* abs/1604.01401 (2016), 1–14.

IBM. 2016. IBM QX Devices. https://quantumexperience.ng.bluemix.net/qx/devices

Toshinari Itoko, Rudy Raymond, Takashi Imamichi, Atsushi Matsuo, and Andrew W. Cross. 2019. Quantum Circuit Compilers Using Gate Commutation Rules. In *ASPDAC*. ACM, New York, NY, USA, 191–196. https://doi.org/10.1145/3287624.3287701

Ali Javadi-Abhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. 2014. ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Computing Frontiers*. ACM, NY, USA, 1.

Jun Kawahara, Toshiki Saitoh, and Ryo Yoshinaka. 2017. The Time Complexity of the Token Swapping Problem and Its Parallel Variants. In *WALCOM*. Springer, Heidelberg, Germany, 448–459.

J. Kelly, R. Barends, A. G. Fowler, A. Megrant, E. Jeffrey, T. C. White, D. Sank, J. Y. Mutus, B. Campbell, Yu Chen, Z. Chen, B. Chiaro, A. Dunsworth, I.-C. Hoi, C. Neill, P. J. J. O'Malley, C. Quintana, P. Roushan, A. Vainsencher, J. Wenner, A. N. Cleland, and John M. Martinis. 2014. State preservation by repetitive error detection in a superconducting quantum circuit. *CoRR* arXiv:1411.7403 (2014), 1–30.

L. Lao, B. van Wee, I. Ashraf, J. van Someren, N. Khammassi, K. Bertels, and C. G. Almudever. 2018. Mapping of Lattice Surgery-based Quantum Circuits on Surface Code Architectures. arXiv:arXiv:1805.11127

Gushu Li, Yufei Ding, and Yuan Xie. 2018. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. arXiv:arXiv:1809.02573 To appear in ASPLOS'19.

C. C. Lin, S. Sur-Kolay, and N. K. Jha. 2015. PAQCS: Physical Design-Aware Fault-Tolerant Quantum Circuit Synthesis. *TVLSI* 23, 7 (2015), 1221–1234.

Y. Lin, B. Yu, M. Li, and D. Z. Pan. 2018. Layout Synthesis for Topological Quantum Circuits With 1-D and 2-D Architectures. *TCAD* 37, 8 (2018), 1574–1587.

Paul Magnard, Philipp Kurpiers, Baptiste Royer, Theo Walter, Jean-Claude Besse, Simone Gasparinetti, Marek Pechal, Johannes Heinsoo, Simon Storz, Alexandre Blais, and Andreas Wallraff. 2018. Fast and Unconditional All-Microwave Reset of a Superconducting Qubit. *CoRR* arXiv:1801.07689 (2018), 1–9.

Igor L. Markov, Aneeqa Fatima, Sergei V. Isakov, and Sergio Boixo. 2018. Quantum Supremacy Is Both Closer and Farther than It Appears. *CoRR* arXiv:1807.10749 (2018), 1–32.

D. Maslov, S. M. Falconer, and M. Mosca. 2008. Quantum Circuit Placement. *TCAD* 27, 4 (2008), 752–763.

Tillmann Miltzow, Lothar Narins, Yoshio Okamoto, Günter Rote, Antonis Thomas, and Takeaki Uno. 2016. Approximation and Hardness of Token Swapping. In *ESA*. Schloss Dagstuhl, Dagstuhl, Germany, 66:1–66:15.

M. Oskin, F. T. Chong, and I. L. Chuang. 2002. A practical architecture for reliable quantum computers. *Computer* 35, 1 (Jan 2002), 79–87. https://doi.org/10.1109/2.976922

Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, Erik W. Draeger, Eric T. Holland, and Robert Wisnieff. 2018. Breaking the 49-Qubit Barrier in the Simulation of Quantum Circuits. *CoRR* arXiv:1710.05867 (2018), 1–29.

M. Pedram and A. Shafaei. 2016. Layout Optimization for Quantum Circuits with Linear Nearest Neighbor Architectures. *Circuits and Systems Magazine* 16, 2 (2016), 62–74.

Fernando Magno Quintão Pereira and Jens Palsberg. 2005. Register Allocation Via Coloring of Chordal Graphs. In *APLAS*. Springer, Heidelberg, Germany, 315–329.

A. Saito, K. Kioi, Y. Akagi, N. Hashizume, and K. Ohta. 2000. Actual computational time-cost of the Quantum Fourier Transform in a quantum computer using nuclear spins. arXiv:quant-ph/0001113

Daniel Sank, Evan Jeffrey, J.Y. Mutus, T.C. White, J. Kelly, R. Barends, Y. Chen, Z. Chen, B. Chiaro, A. Megrant A. Dunsworth, P.J.J. O'Malley, C. Neill, P. Roushan, A. Vainsencher, J. Wenner, A.N. Cleland, and J.M. Martinis. 2014. Fast Scalable State Measurement with Superconducting Qubits. *CoRR* arXiv:1401.0257 (2014), 1–9.

A. Shafaei, M. Saeedi, and M. Pedram. 2014. Qubit placement to minimize communication overhead in 2D quantum architectures. In *ASP-DAC*. IEEE, Washington, DC, USA, 495–500.

R. R. Shrivastwa, K. Datta, and I. Sengupta. 2015. Fast Qubit Placement in 2D Architecture Using Nearest Neighbor Realization. In *iNIS*. IEEE, NY, USA, 95–100.

Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintao Pereira. 2018. Qubit Allocation. In *CGO*. ACM, NY, USA, 113–125.

Pavel Surynek. 2018. Finding Optimal Solutions to Token Swapping by Conflict-based Search and Reduction to SAT. arXiv:arXiv:1806.09487

Krysta M. Svore, Alfred V. Aho, Andrew W. Cross, Isaac Chuang, and Igor L. Markov. 2006. A Layered Software Architecture for Quantum Computing Design Tools. *Computer* 39, 1 (2006), 74–83.

Krysta Marie Svore, A. Cross, and I-Hsun Chuang. 2004. Toward a Software Architecture for Quantum Computing Design Tools.

Swamit Tannu and Moinuddin Qureshi. 2019. A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. In *ASPLOS*. ACM, NY, USA, To appear.

Kuhn H. W. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, 1-2 (1955), 83–97.

R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *ISMVL*. IEEE, NY, USA, 220–225.

W. K. Wootters and W. H. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299 (oct 1982), 802–803. https://doi.org/10.1038/299802a0

Katsuhisa Yamanaka, Erik D. Demaine, Takashi Horiyama, Akitoshi Kawamura, Shin-ichi Nakano, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Ryuhei Uehara, and Takeaki Uno. 2017. Sequentially Swapping Colored Tokens on Graphs. In *WALCOM: Algorithms and Computation*, Sheung-Hung Poon, Md. Saidur Rahman, and Hsu-Chun Yen (Eds.). Springer, Heidelberg, Germany, 435–447.

Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. 2014. *Swapping Labeled Tokens on Graphs*. Springer, Heidelberg, Germany, 364–375.

Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3, 1-2 (2010), 340–351.

Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. Efficient mapping of quantum circuits to the IBM QX architectures. In *DATE*. IEEE, NY, USA, 1135–1138.

Alwin Zulehner and Robert Wille. 2019. Compiling SU(4) Quantum Circuits to IBM QX Architectures. In *ASPDAC*. ACM, New York, NY, USA, 185–190.