Efficient template matching in quantum circuits

Raban Iten†*, David Sutter*, and Stefan Woerner†

†IBM Research — Zurich, Switzerland
*Institute for Theoretical Physics, ETH Zurich, Switzerland

Abstract

Given a large and a small quantum circuit, we are interested in finding all maximal matches of the small circuit, called template, in the large circuit under consideration of pairwise commutation relations between quantum gates. In this work we present a classical algorithm for this task that provably finds all maximal matches with a running time that is polynomial in the number of gates and the number of qubits of the circuit for a fixed template size. Such an algorithm finds direct applications in quantum circuit optimization. Given a template circuit for which a lower-cost implementation is known, we may search for all instances of the template in a large circuit and replace them with their optimized version.

1 Introduction

For certain tasks quantum algorithms offer a provable computational advantage compared to the best possible classical methods [1]. For other problems such as finding the prime factors of a large integer [2] or solving linear systems of equations [3], there is no mathematical proof but good evidence that quantum computers can solve them considerably faster than every classical computer. This raises hope that for many practically relevant problems quantum algorithms can outperform existing classical methods, see e.g., [4, 5].

To profit from the power of quantum computing as much as possible, it is crucial to further optimize the implementation cost of existing quantum algorithms. One building block for this task is finding small circuits (or parts thereof) efficiently in a larger circuit — a process called template matching. More precisely, suppose we are given a (potentially large) quantum circuit $C$ consisting of $|C|$ gates and $n_C$ qubits and a template which is another (small) quantum circuit $T$ with $|T|$ gates and $n_T$ qubits. Template matching describes the problem of finding all possible maximal matches of $T$ within $C$.1 The idea for optimizing quantum circuits with templates was introduced in [6], where a template is defined as a sequence of unitary gates $U_i$, such that $U_{|T|} \ldots U_1 = I$.2 Let us now assume that the template matching algorithm finds the gate sequence $U_a \ldots U_b$ in the circuit $C$ for some $1 \leq a \leq b \leq |T|$ that matches the template. Since the full template implements the identity operator and since each unitary $U_i$ has an inverse $(U_i)^{-1} = U_i^\dagger$, we find an alternative representation of this gate sequence as $U_a \ldots U_b = U_{a-1}^\dagger \ldots U_1^\dagger U_{|T|}^\dagger \ldots U_{b+1}^\dagger$. If this gate sequence has lower implementation cost than the original one, we may replace the found match with the gate sequence $U_{a-1}^\dagger \ldots U_1^\dagger U_{|T|}^\dagger \ldots U_{b+1}^\dagger$ in the circuit. Assuming that each additional quantum gate that we have to implement increases the implementation cost, motivates the search for maximal matches.

1Definition 4.5 explains mathematically precisely what a maximal match of $T$ within $C$ is. In particular, a maximal match does not necessarily need to be complete, i.e., the match does not need to cover all gates of the template circuit.
2We do not require a template to implement the identity for this work.
Efficient template matching is a non-trivial task. The classical analog of this problem, where none of the gates commute, is well studied (see for example [7, 8, 9, 10]) and found many applications in the context of computer-aided design (see [10] and references therein). The classical matching problem was reduced to the subgraph isomorphism problem in [10]. However, template matching in quantum circuits faces additional difficulties (as discussed in Section 2) due to the fact that certain gates in a circuit commute with other gates and others do not commute. The commutation relations could be mapped to the subgraph isomorphism problem, by introducing relations between the vertices in the graph that determine if it is allowed to interchange two vertices with each other to find a maximal match. As far as we know, this more complex task has not been studied in the general context of graph matching algorithms so far, but the two extreme cases for commutation relations are well understood. In case all gates commute, efficient template matching is straightforward as we essentially need to check if all gates in the template can be found in the circuit. In the other extreme case where none of the gates commute template matching is not expected to be possible in polynomial time due to close relations with the subgraph isomorphism problem which is NP-complete [11]. Nevertheless for a fixed template (and under the assumption that no gates commute) a polynomial time algorithm for template matching is possible since for a fixed subgraph the subgraph isomorphism problem can be solved efficiently [12]. However, allowing certain gates to commute, it is not clear a priori if the template matching problem is still efficiently solvable for fixed template size, since the number of possible permutations of the gates in the circuit $C$ can grow exponentially in $|C|$. In this work we present an algorithm that demonstrates that it is indeed possible to find all maximal matches in a quantum circuit efficiently.

**Previous work.** In [6, 13], heuristic template matching algorithms were introduced. The algorithm in [6] was then applied in [14, 15] for reversible logic synthesis and achieves very low runtimes. In [16], a template matching algorithm is presented that provably finds all matches. It is based on mapping the circuit to a satisfiability modulo theory problem and applying a specific solver to it. Moreover, it is shown that finding all the matches indeed helps to significantly reduce the gate counts further compared to heuristic approaches. Unfortunately, this improvement comes in tradeoff with the runtime of the algorithm. The algorithm is not efficient, i.e., its worst-case time complexity is exponentially in $|C|$ and $n_C$, and its practical runtimes are significantly higher than for the heuristic approaches (see [16] for a comparison with [14]). We provide a step to overcome the tradeoff between not finding all the matches and long runtimes, by introducing an efficient template matching algorithm that provably finds all maximal matches.

**Result.** We fix an arbitrary set of quantum gates and assume that we can check in constant time if these gates commute with each other. Then, for any circuit $C$ with $|C|$ gates (from the fixed gate set) and $n_C$ qubits and any template $T$ with $|T|$ gates and $n_T$ qubits, we present an algorithm that provably finds all maximal matches of $T$ in $C$. Furthermore the algorithm is efficient in $n_C$ and $|C|$, however inefficient in $n_T$ and $|T|$. More precisely, its running time is

$$O\left(|C|^{|T|+3}|T|^2n_C^{n_T}\right). \tag{1}$$

We refer to Algorithm 3 for a detailed description of the algorithm and to Theorem 4.8 and Theorem 4.9 for a precise statement about the correctness and the complexity of the algorithm. We note that (1) is a theoretical worst-case bound on the running time of our algorithm whose purpose is to prove the efficiency in $|C|$ and $n_C$. In practice, we expect the algorithm to run much faster on most instances.

In general there is a trade-off between speeding up the runtime of the algorithm using heuristics and finding all maximal matches. There are situations where it is beneficial to speed up the runtime of the algorithm at the cost of missing certain maximal matches. For such scenarios our algorithm should be combined with certain heuristics as discussed in Section 4.6. We also give a worst-case complexity

\footnote{A quantum circuit may be viewed as a labeled directed acyclic graph (DAG) and the template would correspond to the subgraph.}
analysis using the heuristics considered in Section 4.6. On the other hand there are situations, such as optimizing a small quantum circuit for current experimental architectures, where one would like to find all the matches to fully optimize the circuit and where the classical computational time is not the bottleneck. For such scenarios, Algorithm 3 should be run without heuristics such that it always finds all the maximal matches while still being efficient. We note that also in intermediate scenarios, where it is not obvious whether a low runtime or finding as many matches as possible is more effective, our algorithm offers a flexible solution. It is possible to change the parameters used for the heuristics such that we can fine-tune the trade-off between lowering the runtime and the number of found matches. Alternatively, one may adapt the search in the algorithm to first look for the most promising matching scenarios and stop the algorithm as soon as one has found sufficient matches for the considered task.

**Notation.** We write a circuit $C$ as a gate list $C = (C_1, \ldots, C_{|C|})$, where the unitary performed by the circuit is given by $U = U_{|C|}U_{|C|-1} \cdots U_1$, where $U_i$ denotes the unitary corresponding to the gate $C_i$. A gate can be any description of a unitary operation together with an ordered list of qubits it acts on, e.g., the gate C-NOT(1, 4) represents a C-NOT gate controlling on the qubit with label 1 and acting on the qubit with label 4. If two gates $A$ and $B$ perform the same operation, eventually on qubits with different labels, we write $A \cong B$, e.g., C-NOT(1, 4) $\cong$ C-NOT(2, 1). If the unitaries that represent the circuits $C$ and $D$ are equal up to a global phase shift, we say that the two circuits are represented by the same operator, and we write $C \simeq D$. The concatenation of two circuits $C = (C_1, \ldots, C_{|C|})$ and $D = (D_1, \ldots, D_{|D|})$ is denoted by $(C, D) = (C_1, \ldots, C_{|C|}, D_1, \ldots, D_{|D|})$. We denote the commutator of the unitaries corresponding to two gates $A$ and $B$ by $[A, B]$. Moreover, we write $[i, j]_C = 0$ if and only if $i = j$ or if we can pairwise commute gates in the circuit $C$ such that the order of the gates $C_i$ and $C_j$ is interchanged, i.e., if $i < j$ the gate $C_i$ can be moved before the gate $C_j$ and vice versa for the case $j < i$. The set $\text{Perm}(1, 2, \ldots, n) = \{(1, 2, 3, \ldots, n), (2, 1, 3, \ldots, n), \ldots\}$ denotes the set of all possible permutations of $(1, 2, \ldots, n)$.

In the circuit model of quantum computation, information carried in qubit wires is modified by quantum gates which mathematically are described by unitary operations. For the examples in this work, we use C-NOT gates, Toffoli gates and single-qubit rotations, however, our algorithm works for any arbitrary gate set. For the single-qubit rotations, we will use the following convention

$$R_x(\theta) = \begin{pmatrix} \cos[\theta/2] & -\sin[\theta/2] \\ -\sin[\theta/2] & \cos[\theta/2] \end{pmatrix}, \quad R_y(\theta) = \begin{pmatrix} \cos[\theta/2] & -\sin[\theta/2] \\ \sin[\theta/2] & \cos[\theta/2] \end{pmatrix}, \quad R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix},$$

which correspond to rotations by an angle $\theta$ around the $x$-, $y$- and $z$-axes of the Bloch sphere. One important special case is the NOT gate, $\sigma_x = iR_x(\pi)$ in terms of which the C-NOT gate can be written as $|0\rangle|0\rangle \otimes 1 + |1\rangle|1\rangle \otimes \sigma_x$. Similarly, the non trivial part of the action of a Toffoli gate is given by $|1\rangle|0\rangle \otimes [1\rangle|1\rangle \otimes \sigma_z$.

It is convenient to represent quantum circuits diagrammatically. Each qubit is represented by a wire and gates are shown using a variety of symbols. Conventionally time flows from left to right. The circuit depicted in Figure 1 shows a C-NOT gate controlling on the most significant qubit (with label 1) and acting on the least significant qubit (with label 4), a Toffoli gate controlling on the qubits 1, 2 and acting on the qubit 3 and finally a $R_x$ rotation with rotation angle $\pi$ acting on qubit 1.

We will also work with a canonical form of quantum circuits that is independent on how the gates are commuted. This form is represented by directed acyclic graphs. We denote the set of successors of a vertex $v_i$ in such a graph $G$ by $\text{Succ}(v_i, G)$, i.e., $\text{Succ}(v_i, G)$ contains all the vertices $v_j$ for which there is a (forward directed) path from vertex $v_i$ to vertex $v_j$. On the other hand, we denote the set of predecessors of a vertex $v_i$ in a graph $G$ by $\text{Pred}(v_i, G)$, i.e., the set $\text{Pred}(v_i, G)$ contains all the vertices $v_j$ such that there is a path from vertex $v_j$ to vertex $v_i$. The direct successors and predecessors are the ones that are connected through only one edge to the considered vertex $v_i$, and we call these sets $\text{DirectSucc}(v_i, G)$ and $\text{DirectPred}(v_i, G)$, respectively.

\footnote{It should always be clear for the context if we mean a tuple of two circuits or the concatenation of them.}
Structure. We start in Section 2 with a discussion about difficulties for efficient template matching due to the fact that quantum gates may commute. In Section 3 we introduce a canonical form of quantum circuits as directed acyclic graphs. We then present and analyze the matching algorithm in Section 4. An overview of the algorithm is given in Section 4.1, followed by the pseudocode in Section 4.2. Section 4.3 presents a pedagogical example that illustrates how the template matching algorithm works in practice. Understanding this example might be simpler than reading through the pseudocode of the algorithm. We discuss the correctness and complexity of the algorithm in Section 4.4 and Section 4.5, where the details of the correctness proof are shifted to Appendix A for readability. Further, we provide some suggestions for heuristics to lower the runtime of practical implementations in Section 4.6.

2 Difficulties for matching quantum circuits

We start with a discussion about difficulties in constructing an efficient template matching algorithm. Let us list some different kind of problems that arise due to the commutative nature of certain gates. This list will help us to understand the structure of the template matching algorithm given in Section 4, which handles all of the following problems efficiently.

1. **Ordering:** The simplest problem that appears due to commuting gates is illustrated in Figure 2. If we just start matching the first gate of the template with the first gate of the circuit, we assign the second qubit of the template with the third one of the circuit and hence the third gate of the template will not match. However, clearly the two circuits could be fully matched by commuting the first two gates in the circuit.\(^5\)

![Figure 2: Ordering problem when matching a template with a circuit.](image)

2. **Additional matching gates:** Consider the case where some gates could be matched but they should not in order to find the maximal match. Hence a straightforward greedy approach is not necessarily optimal. Let us consider the following template and circuit depicted in Figure 3. If we match the first two gates, the third gate will not match. Furthermore, it is not possible to

\(^5\)We recall that the target as well as the control nodes of different C-NOT gates commute. However a target node does not commute with a control node.
commute the matched gates next to the last gate in the circuit (which matches the third gate of the template) or vice versa. However, there exists a full match that one can find by matching the first gate of the circuit (and ignore the second one) and commute it through the second and the third one to match the full template.

![Figure 3: Greedy matching does not always lead to the maximal match.](image)

3. **Disturbing gates:** We consider a template and a circuit as given in Figure 4. The second gate in the circuit “disturbs” the match. The maximal match is found by commuting it as far as possible to the left or the right. In the considered case, we can match three gates (instead of two) by not commuting the second gate to the right. The disturbing gates are difficult to handle in general, since it is a priori unclear if one should try to move them to the right or to the left in the circuit. If one always considers both options, the time complexity of such an algorithm would be exponential in the number of disturbing gates.

![Figure 4: It might be unclear to which position we should move a disturbing gate (marked by a solid box in the circuit) to find a maximal match.](image)

The reader may find it helpful to have these problems in mind while reading the proof of the correctness of the template matching algorithm given in Section 4. We close this section with a remark about a possibility to go beyond pairwise commutations.

**Remark 2.1** (Beyond pairwise commutations). We only allow to commute single gates with each other throughout this work. In general, it could happen that in a circuit $C = (C_1, C_2, C_3)$, no gates commute pairwise, however, the unitary corresponding to $(C_1, C_2)$ could commute with the unitary corresponding to $C_3$. Hence, one could bring the circuit $C$ into the form $(C_3, C_1, C_2)$, which could help matching in principle. However, multiplying gates to check commutation relations is computationally expensive and we do not take such commutation relation into account in this work. Nonetheless, such relations could be used to improve an implementation of the matching algorithm in practice.
3 Canonical form for quantum circuits

Representations of quantum circuits are not unique in general, because various gates may commute. For example, the two circuits represented in Figure 5a and Figure 5b represent the same operator.

![Figure 5: We can represent circuits in a canonical form which stays the same under gate commutations in a circuit C. The canonical form is a directed acyclic graph where all the gates that can be commuted next to each other and do not commute are connected with an edge. The edge points from the gate that comes first in the circuit to the one that appears later on.](image)

For some applications, it is desirable to work with a canonical form of quantum circuits. The canonical form that we use (and which was introduced in [13]) is a directed acyclic graph where the vertices are labeled with the gate indices and where all the vertices corresponding to gates that can be commuted next to each other and do not commute are connected with an edge. The edges point from the gate that comes first in the circuit to the one that appears later on (see Figure 5c for an example). Using such a representation is not strictly necessary for our algorithm to be efficient, but it will simplify its description and lower the runtime for some subroutines.

We will allow to add different attributes to vertices in the pseudocode, which we access by “vertex.attribute”. In particular, as mentioned above, we always add an attribute “label” referring to the gate index corresponding to the vertex. Further, we use $G_i$ to access the vertex with label $i$ in the graph $G$ and we denote the number of vertices in $G$ by $|G|$. Note that we can store the vertices of the graph in an array at positions according to their index to have constant time access to any vertex with known label. Storing incoming and outgoing edges together with each vertex (as pointers to the direct successors and predecessors), we also have constant time access to all direct successors and predecessors of any given vertex.

An algorithm that constructs the canonical representation of any quantum circuit $C$ with time complexity $O(|C|^2)$ was given in [13] and is described in Algorithm 1 for completeness.

**Algorithm 1 CreateCanonicalForm:** Creates the canonical form of a quantum circuit

1: Input: Quantum circuit $C$ with $|C|$ gates
2: Initialize an empty directed acyclic graph $G$
3: for $j \in \{1, 2, \ldots, |C|\}$ do
4:   Set the attribute isReachable to true for all vertices in $G$
5:   Add a vertex with label $i$ to the graph $G$
6:   for $i \in \{j - 1, j - 2, \ldots, 1\}$ do
7:     if $G_i$ isReachable and $[C_i, C_j] \neq 0$ then
8:       Add an (directed) edge from vertex $G_i$ to vertex $G_j$ in $G$
9:       for preDec $\in \text{Pred}(G_i, G)$ do
10:          preDec.isReachable $\leftarrow$ false
11:       end for
12:     end if
13:   end for
14: end for
15: Output: the canonical form $G$ of the circuit $C$
Remark 3.1. If we account worst-case time complexity $\mathcal{O}(|C|^3)$ to create the canonical form $G$ of a quantum circuit $C$, we can assume constant time access to the list of all possible successors (or predecessors) of a vertex in $G$. Moreover, we may assume that this list is ordered in increasing order according to the labels of the vertices. Indeed, Algorithm 2 sets an attribute $\text{Successors}$ that provides the successor of the corresponding vertex ordered according to their labels for all vertices in $G$ in time $\mathcal{O}(|C|^2)$. To see this, it is enough to notice that the merging process of the ordered lists in Algorithm 2 has worst-case time complexity $\mathcal{O}(|C|^2)$. Indeed, consider $k$ ordered lists, each of which containing at most $n$ numbers between 1 and $n$. Merging two of the ordered lists of length $n$ can be done in time $\mathcal{O}(n)$. Since we remove doublings during the merging process, we end up again with a list of length at most $n$ (because there are only $n$ different numbers). Therefore, merging all pairs of the $k$ lists has time complexity $\mathcal{O}(kn)$, and we end up with $\lceil k/2 \rceil$ lists of length at most $n$. Going on recursively like this, gives a time complexity of $\sum_{i=0}^{\lceil \log_2(k) \rceil - 1} \frac{1}{2} \mathcal{O}(kn) = \mathcal{O}(kn)$. In our case, we have at most $|G|$ direct successors $v_i$ for each vertex and each list $(v_i).\text{Successors}$ can contain at most $|G|$ entries, i.e., $k = n = |G| = |C|$.

Algorithm 2 $\text{InitializeSuccessors}$: Sets an attribute $\text{Successors}$ of each vertex equal to the list of its successors (ordered according to their labels)

1. Input: Canonical form $G$ of a quantum circuit
2. Initialize an attribute $\text{Successors} \leftarrow \text{null}$ for all vertices in $G$
3. for $i \in \{|G|, |G| - 1, \ldots, 1\}$ do
4. \quad $(G_i).\text{Successors} \leftarrow (\text{Sorted})$ merge of the lists $v.\text{Successors}$ for all $v \in \text{DirectSucc}(G_i, G)$
5. end for

In the following, we will show some properties of the canonical form, which will turn out useful for proving the correctness of our matching algorithm.

Lemma 3.2 (Independence on pairwise gate commutations). Given two circuit $C \simeq C'$, where $C'$ equals $C$ up to a changed gate order (respecting the commutation relations between the quantum gates) and keeping the gate labels from circuit $C$. Then, the canonical form of $C$ and $C'$ are given by the same directed acyclic graph.

Proof. In the construction in Algorithm 1, we only add edges for non commuting gates. The order of these gates could hence not be interchanged in the circuit $C'$ and one can see that one ends up with the same canonical form (however, the vertices might have been added to the graph in different orders).

Lemma 3.3 (Necessary and sufficient condition for interchanging gates). Given a circuit $C$ with a canonical form $G$ and two indices $i < j$, then the following two statements are equivalent:

1. $[i, j]_C = 0$,
2. $G_j \notin \text{Succ}(G_i, G)$.

Proof. From the construction in Algorithm 1 it is clear that if $G_j \in \text{Succ}(G_i, G)$, we have that the gate $C_j$ can not be moved before the gate $C_i$ and hence $[i, j]_C \neq 0$, because the edges represent non-zero commutation relations. It thus remains to show that $[i, j]_C \neq 0$ implies $G_j \in \text{Succ}(G_i, G)$. We show the claim by induction over the number of vertices in the graph $G$. The claim is clear for a graph consisting of two vertices, since we add an edge between them in Algorithm 1 if and only if the two corresponding gates in the circuit $C$ do not commute. Let us assume that the claim is true for all possible pairs of indices $i < j$ in a circuit with $n$ vertices. We have to show the claim for $j = n + 1$ and an arbitrary $i < n + 1$. Clearly, $[i, n + 1]_C \neq 0$ implies that there exists an index $i \leq k \leq n$ with $[C_{n+1}, C_k] \neq 0$ and with $[i, k]_C \neq 0$ if $k \neq i$. Let us choose the largest such index $k$. Then, by the induction assumption, $G_k \in \text{Succ}(G_i, G) \cup \{G_i\}$. By the construction in Algorithm 1, the vertex $G_k$ is
still accessible if it is visited in the inner loop with respect to \( i = n + 1 \) in the outer loop, and hence, an edge from \( G_k \) to \( G_{n+1} \) is added and since \( G_k \in \text{Succ}(G_i, G) \cup \{G_i\} \), we find \( G_{n+1} \in \text{Succ}(G_i, G) \). Indeed, to see that the vertex \( G_k \) is still accessible when it is visited in Algorithm 1, assume by contradiction that it would not be accessible. Then, \( G_k \) must be a predecessor of a vertex \( G_{k'} \) with \( i \leq k < k' < n + 1 \) such that \([G_k, G_{n+1}] \neq 0\). Since \( G_k \in \text{Succ}(G_i, G) \cup \{G_i\} \), this would imply that \( G_{k'} \in \text{Succ}(G_i, G) \). By the induction assumption, this means \([i, k']_C \neq 0\). However, we assumed that \( k \) is the largest index with these properties, and hence this leads to \( k = k' \), which contradicts \( k < k' \). □

4 Template matching algorithm

For any given circuit \( C \) and any template \( T \), TempMatch (see Algorithm 3) finds all maximal matches of \( T \) in \( C \).⁶ The intuition how TempMatch works is visible best by reading the example presented in Section 4.3. Let us first give a rough overview of the idea behind the algorithm, and the detailed pseudocode afterwards.

4.1 Overview of the template matching algorithm

The algorithm TempMatch loops over the gates in the template \( T \) and for each such gate \( T_i \) for \( 1 \leq i \leq |T| \), we search for matching gates \( C_r \) in the circuit \( C \) acting on qubits listed in \( L_q = \{q_1, \ldots, q_l\} \) for \( 1 \leq l \leq n_T \). Then, we also consider all possibilities for choices of \( n_T - l \) additional qubits in \( C \) that can be assigned with the qubits of the template \( T \). Afterwards, the main subroutines ForwardMatch and BackwardMatch (see Algorithm 4 and Algorithm 5) are called to find the maximal matches of the partial template \( T' = (T_1, \ldots, T_{|T|}) \) in the circuit \( C \) under the condition that the gate \( T_i \) is matched with \( C_r \).

We think of splitting the template \( T' \simeq (T_{\text{backward}}, T_{\text{forward}}) \) into a left part \( T_{\text{backward}} \) and a right part \( T_{\text{forward}} \) (with \( T_{\text{forward}} = T_i = T'_i \)) of the starting gate, such that all the gates in \( T_{\text{forward}} \) cannot be moved to the left of the initially matched gate \( T_i \), i.e., \([i, j]_{T_{\text{forward}}} \neq 0 \) for all \( j \in \{2, 3, \ldots, |T_{\text{forward}}|\} \). The very efficient algorithm ForwardMatch finds the maximal match of \( T_{\text{forward}} \) in the circuit \( C \) under the condition that we match \( C_r \) with \( T_i \) in quadratic time in the the number of gates \( |C| \) in the circuit \( C \) (see Lemma 4.10 for the derivation of the worst-case complexity). The algorithm BackwardMatch can then be used to expand the match found by ForwardMatch to maximal matches of the template \( T' \). The algorithm BackwardMatch thereby tries to add as many matches as possible with \( T_{\text{backward}} \). However, matching more of \( T_{\text{backward}} \) might destroy some matches with \( T_{\text{forward}} \). To handle this tradeoff, BackwardMatch has to go through all matching situations that could lead to maximal matches and is hence computationally more costly (but still efficient, see Lemma 4.11 for the worst-case time complexity) than ForwardMatch. In practice, if not nearly all of the gates commute in the template, we expect that several matching scenarios can directly be ignored, since there are more matches destroyed with \( T_{\text{forward}} \) than could possibly be added by matching all the gates in \( T_{\text{backward}} \). Hence, the average case complexity of BackwardMatch is expected to be much lower than the worst-case complexity.

4.2 Pseudocode for the matching algorithm

In this section, we describe the pseudocode of the template matching algorithm. We stress that the focus of the code is readability and we do not optimize the constants of the runtime. We usually think of working with pointers to circuit or graph objects. Hence, an object might be modified by a method call, however it is not given back as an output. As a result, we may sometimes have to copy an object \( o \) by calling \( o.copy \).

⁶The mathematically precise definition of a maximal match is given in Definition 4.5.
Algorithm 3 TempMatch: Templating matching algorithm

1: Input: \((C, T)\)
   - Circuit \(C\) with \(n_C\) qubits and \(|C|\) gates
   - Template \(T\) with \(n_T\) qubits and \(|T|\) gates

2: Initialize a list \(L_M\) to store matches
3: \(G^C \leftarrow \text{CreateCanonicalForm}(C)\)
4: \(G^T \leftarrow \text{CreateCanonicalForm}(T)\)
5: \(L_q \leftarrow \{1, 2, \ldots, n_C\}\)
6: for \(i \in \{1, 2, \ldots, |T|\}\) do
   \[\text{loop through all gate indices of } T\text{ for starting a match at } T_i\]
7:   for \(r \in \{k \in \{1, \ldots, |C|\} : C_k \cong T_i\}\) do
      \[\text{loop through the indices of gates in } C\text{ with } C_r \cong T_i\]
8:     \(L^\text{action}_q \leftarrow \{s \in L_q : \text{the gate } C_r \text{ is acting non trivially on the qubit with label } s\}\)
9:   for \(L^\text{sel}_q \in \{L \in (L^\text{action}_q) : L^\text{action}_q \subset L\}\) do
   \[\text{loop through all possible choices of qubits}\]
10:      \(L^\text{sel}_q \leftarrow \text{Sort the qubits in } L^\text{sel}_q\) in increasing order according to their labels (store as list)
11:     for \(p \in \text{Perm}(1, 2, \ldots, n_T)\) do
   \[\text{loop through all possible qubit orderings}\]
12:        \(\tilde{T} \leftarrow \text{Label the qubits in } T\text{ with the labels in } L^\text{sel}_q\) using the mapping \(t \mapsto (L^\text{sel}_q)_p t\)
13:           \# Note that the canonical form of \(T\) and \(\tilde{T}\) is the same
14:        if \(C_r = \tilde{T}_i\) then
   \[\text{Check if the qubit permutation is such that } C_r\text{ matches } \tilde{T}_i\]
15:           \# Rooted template matching: Find the maximal matches of the template
16:           \# \((\tilde{T}_i, \ldots, \tilde{T}_{|T|})\) in the circuit \(C\) under the restriction that \(\tilde{T}_i\) is matched with \(C_r\)
17:           \# We match the maximal part in forward direction of \(\tilde{T}_i:\)
18:           \(M^\text{forward} \leftarrow \text{ForwardMatch}(C, G^C, \tilde{T}, G^T, L^\text{sel}_q, r, i)\)
19:           \# Expand forward match to maximal ones with partial template \((\tilde{T}_i, \ldots, \tilde{T}_{|T|})\):
20:               \(L^\text{rooted} \leftarrow \text{BackwardMatch}(C, G^C, \tilde{T}, L^\text{sel}_q, r, i, M^\text{forward})\)
21: Add the matches in \(L^\text{rooted}\) to \(L_M\)
22: end if
23: end for
24: end for
25: end for
26: end for
27: Output: the list \(L_M\) of matches

Note that the elements \(M\) in the output \(L_M\) in Algorithm 3 are sets containing index pairs \((i, j)\) of matched gates, i.e., if the match \(M\) contains \((i, j)\), the gate \(T_i\) from the template was matched with the gate \(C_j\) from the circuit. The qubit mapping can then be recovered from the matched gates.

Remark 4.1. The output of TempMatch indicates how to match the gates of the template with the gates in the circuit. However, the information of how to commute the gates in the circuit to move the matched gates next to each other is not contained for simplicity. This information can be restored efficiently by commuting the gates in between the matched gates to the left or to the right of the match.
**Algorithm 4 ForwardMatch:** Find the maximal match in forward direction

1: Input: $(C, G^C, T, G^T, L_q, r, i)$
   - Circuit $C$ with canonical form $G^C$
   - Template $T$ with canonical form $G^T$
   - List $L_q$ of qubit labels we are matching on
   - Gate indices $r$ in $C$ and $i$ in $T$ (where we start matching)

2: # Initialization:
3: Initialize a set $M \leftarrow \{(i, r)\}$ to store matched gate indices
4: For all vertices in $G^C$, initialize attributes
   - $SuccessorsToVisit = ()$ with $G^C_r$.
     $SuccessorsToVisit \leftarrow \text{list containing } \text{DirectSucc}(G^C_r, G^C)$ ordered in increasing order according to theirs labels,
   - $matchedWith \leftarrow \text{null}$ and set $G^C_r.matchedWith \leftarrow G^T_i$ and
   - $isBlocked \leftarrow false$.
5: Initialize a list $MatchedVertexList$ and add $G^C_r$ as a first element. The list is ordered in increasing order according to the label of the first element of $SuccessorsToVisit$ of each vertex.
6: # Forward matching process:
7: while $MatchedVertexList$ is not empty do
8:  $v_0 \leftarrow \text{MatchedVertexList.get(1)}$ \smaller{matched vertex as a root for further matching}
9:  if $v_0.SuccessorsToVisit$ is empty then
10:    GoTo “EndOfWhileLoop”
11:  end if
12:  $v \leftarrow v_0.SuccessorsToVisit.get(1); s \leftarrow v.label$ \smaller{vertex to consider for matching}
13:  Insert $v_0$ into $MatchedVertexList$ \smaller{Put vertex back with modified $SuccessorsToVisit$}
14:  if $v.isBlocked$ or $v.matchedWith \neq \text{null}$ then
15:    GoTo “EndOfWhileLoop”
16:  end if
17:  # We try to expand the match with the vertex $v$ in the following.
18:  $CandidateIndices \leftarrow \text{FindForwardCandidates}(G^T, v_0, matchedWith, M)$
19:  if There exist a $j \in CandidateIndices$ with $C_s = T_j$ then
20:    # We found a match with $v$:
21:    $j \leftarrow \text{Choose the minimal } j \in CandidateIndices$ with $C_s = T_j$
22:    $v.matchedWith \leftarrow G^T_j$; Add $(j,s)$ to $M$
23:    $S \leftarrow \{w \in \text{DirectSucc}(v, G^C) : w.isBlocked = false; w.matchedWith = \text{null}\}$
24:    $L \leftarrow \text{Order vertices in } S$ in increasing order according to teir labels (and store as a list)
25:    $v.SuccessorsToVisit \leftarrow L$
26:    $MatchedVertexList.Insert(v)$
27:  else
28:    # No match with $v$ was found:
29:    Set the attribute $isBlocked$ equal to $true$ for the vertex $v$ and all of its successors
30:  end if
31:  Label “EndOfWhileLoop”
32: end while
33: Output: $M$

---

7The method $SuccessorsToVisit.Get(i)$ returns the ith vertex from $SuccessorsToVisit$ and removes it from the list.
8The method $MatchedVertexList.Get(i)$ returns the ith vertex from $MatchedVertexList$ and removes it from the list.
The method $MatchedVertexList.Insert(v)$ adds a vertex $v$ at the position according to the label of the first vertex listed in $v.SuccessorsToVisit$. 
Algorithm 5 BackwardMatch: Find maximal expansions of the forward match in backward direction

1: Input: \((C, G^C, T, L_q, r, i, M_{\text{forward}})\)
   - Circuit \(C\) with canonical form \(G^C\) with assigned attributes “matchedWith” and “isBlocked”
   - Template \(T\)
   - List \(L_q\) of \(n_T\) qubit labels we are matching on
   - Gate indices \(r\) in \(C\) and \(i\) in \(T\) (where we start matching)
   - Set of matched index pairs \(M_{\text{forward}}\)

2: # Initialization:
3: \(L_M \leftarrow ()\) \(\triangleright\) List to store matchings
4: \(\text{GateIndices} \leftarrow \{l \in \{1, 2, \ldots, |G^C|\} : G^C_l.\text{isBlocked} = \text{false} \text{ and } G^C_l.\text{matchedWith} = \text{null}\}\)
5: \(\text{GateIndices} \leftarrow \text{Order}\ \text{GateIndices}\ \text{in decreasing order (and store as list)}\)
6: \(\text{counter} \leftarrow 1; \text{numberOfGatesLeftToMatch} \leftarrow |T| - (i - 1) - |M_{\text{forward}}|\)
7: # Initialize a stack to save all matching scenarios that should be considered for expansion:
8: Initialize a stack \(\text{MatchingScenarios}\) and call \(\text{MatchingScenarios}.\text{Push}(G^C, M_{\text{forward}}, \text{counter})\)
9: # Start the matching process:
10: \textbf{while} \(\text{MatchingScenarios}\) is not empty \textbf{do}
11:  \((G^C, M, \text{counter}) \leftarrow \text{MatchingScenarios}.\text{Pop}\) \(\triangleright\) We consider \(G_s\) for matching
12:  \(s \leftarrow (\text{GateIndices})_{\text{counter}}\)
13:  \(v \leftarrow G^C_s\)
14:  # Trivial cases
15:  \(M_{\text{backward}} \leftarrow M \setminus M_{\text{forward}}\)
16:  \textbf{if} \(\text{counter} = |\text{GateIndices}|\) or \(|M_{\text{backward}}| = \text{numberOfGatesLeftToMatch}\) \textbf{then}
17:      Add \(M\) to \(L_M\)
18:      \text{GoTo “EndOfWhileLoop”}
19:  \textbf{end if}
20:  \textbf{if} \(v.\text{isBlocked}\) \textbf{then}
21:      \text{MatchingScenarios}.\text{Push}(G^C, M, \text{counter} + 1)
22:      \text{GoTo “EndOfWhileLoop”}
23:  \textbf{end if}
24:  # Consider the gate \(G_s\) corresponding to vertex \(v\) for matching:
25:  \(\text{CandidateIndices} \leftarrow \text{FindBackwardCandidates}(T, M, i)\)
26:  \textbf{if} \(G_s \in \{T_j : j \in \text{CandidateIndices}\}\) \textbf{then}
27:      # We found a match with the gate \(G_s\)
28:      \(G' \leftarrow G^C.\text{copy}\) and \(M' \leftarrow M.\text{copy}\)
29:      # Option 1: we match
30:      Choose a vertex \(G^T_j\) with a label \(j \in \text{CandidateIndices}\), such that \(G_s = T_j\)
31:      \(M \leftarrow M \cup \{(j, s)\}\)
32:      \(v.\text{matchedWith} \leftarrow G^T_j\) \(\triangleright\) Updates attribute of vertex \(v\) in graph \(G^C\)
33:      \text{MatchingScenarios}.\text{Push}(G^C, M, \text{counter} + 1)
34:      # Option 2: we block \(\triangleright\) Greedy matching is not always optimal
35:      \(v' \leftarrow G'_s\)
36:      \(v'.\text{isBlocked} \leftarrow \text{true}\)
37:      Block all successors of \(v'\) and update \(M'\) if matched gates are blocked
38:      \textbf{if} \((i, r) \in M'\) and \(M_{\text{backward}} \subseteq M'\) \textbf{then} \(\triangleright\) Check if we blocked “fixed” matches
39:         \text{MatchingScenarios}.\text{Push}(G', M', \text{counter} + 1)
40:  \textbf{end if}
else
  # There is no gate in the template that matches the gate $C_s$
  v.isBlocked ← true
  followingMatches ← \{ w ∈ Succ(v, GC) : w.matchedWith ≠ null \}
  if Pred(v, GC) = ∅ or followingMatches = ∅ then
    # The gate $C_s$ corresponding to $v$ can be moved to the start of the circuit $C$
    # or after the end of the already matched gates:
    MatchingScenarios.Push(GC, M, counter + 1)
  else
    # The gate $C_s$ corresponding to $v$ might disturb the expansion of the match.
    # We have to consider two options.
    G′ ← GC.copy and M′ ← M.copy
    # Option 1: Move the disturbing gate $C_s$ to the left
    MatchingScenarios.Push(GC, M, counter + 1)
    # Option 2: Move the disturbing gate $C_s$ to the right
    v′ ← G′s
    Block all successors of the vertex v′ and update M′ if matched gates are blocked
    if (i, r) ∈ M′ and Mbackward ⊂ M′ then  \(\triangleright\) Check if we block “fixed” matches\(^{10}\)
      MatchingScenarios.Push(G′, M′, counter + 1)
  end if
end if
Label “EndOfWhileLoop”
end while
maxLength ← max{|M| : M ∈ LM} 
Remove all the matches in LM that have smaller length than maxLength
Output: LM

---

**Algorithm 6 FindForwardCandidates:** Finds the indices of the gates that might match next for forward matching

1: Input: \((G, v, M)\)
   - Canonical form $G$
   - Vertex $v$ in $G$
   - Set of matched gate indices $M$, where the first index corresponds to the labels in $G$
2: Match ← \{Gi : i ∈ \{1, ..., |G|\} \} such that there exists a $j$ with \((i, j) ∈ M\)
3: CandidatesV ← DirectSucc(v, G) \ Match
4: CandidateIndices ← \{w.label : w ∈ CandidatesV\}
5: Output: CandidateIndices

\(^{10}\)To improve the runtime, we could in addition check if the length of the match $M′$ plus the number of gates in the template that could possibly be matched in the further backwards matching process, is smaller than the length of the initial forward match $M_{forward}$. If this is the case, we could ignore this matching scenario because it cannot lead to a match that is at least as long as the forward match, and hence it cannot lead to a maximal match.
Algorithm 7 FindBackwardCandidates: Finds the indices of the gates $T_{i+1}, \ldots, T_{|T|}$ that might match next for backward matching

1: Input: $(T, M, i)$
   - Circuit $T$
   - Set of matched gate indices $M$, where the first index labels the gates in $T$
   - Start index $i$ in the template
2: $Match \leftarrow \{l \in \{1, \ldots, |T|\} \mid \text{there exists a } j \text{ with } (l, j) \in M\}$
3: $lastMatch \leftarrow \text{minimal index in } Match$
4: $S \leftarrow \{l \in \{i+1, \ldots, |T|\} : T_l \text{ can be moved next to } T_{lastMatch} \text{ on the left}\}$
5: $CandidateIndices \leftarrow S \setminus Match$
6: Output: $CandidateIndices$

4.3 Example for a template match using the algorithm TempMatch

To demonstrate the working of our template matching algorithm TempMatch (Algorithm 3), let us consider an example. Suppose we are given a template $T$ and a circuit $C$ as shown in Figure 6. For simplicity, we represent some gates in parallel. However, we may think of them as being stored in an ordered list and each gate having its own index. We recall that target as well as control nodes of different C-NOT gates (or Toffoli gates) commute with each other. In addition, $R_x$ and $R_z$ gates commute with target and control nodes of C-NOT gates, respectively. Note also that we use a restricted gate set in the example for simplicity, however, the algorithm works for arbitrary sets of gates.

Let us consider a fixed starting point for the matching, and for simplicity, let us assume that we want to start matching with the first gate of the template.\footnote{In the algorithm, we loop over all the gates in the template and consider them as starting gates. The given choice does not restrict the generality of the example, since you may consider the given template as a part of a larger template, where the first gate corresponds to a gate with index $i$ of the larger template.} In the circuit, the algorithm TempMatch loops over all possible starting points for a match of the first gate and over all choices and orders of five qubits out of the eight qubits of the given circuit.\footnote{This loop leads to the term $\frac{n_G!}{(n_C-n_T)!}$ in the complexity of the algorithm TempMatch (see Theorem 4.9).} Let us consider $C_8$ as the starting gate for a match (as shown in Figure 6b) and the case where qubits $3-7$ have been chosen with the order given by the mapping of the qubit labels $(1, 2, 3, 4, 5)$ of the template to the qubit labels $(7, 6, 5, 4, 3)$ in the circuit. For simplicity, let us denote the template with relabeled qubit (which is denoted by $\tilde{T}$ in Algorithm 3) again by $T$ in the following.

Figure 6: Template $T$ that should be maximally matched with a connected part of the circuit $C$. We start with matching the two marked gates. The numbers at the top denote the indices of the individual gates.
The algorithm **TempMatch** first runs Algorithm 1 twice to construct the canonical forms $G^T$ and $G^C$ of the template $T$ and the circuit $C$, respectively. The resulting graphical representations of $G^T$ and $G^C$ are depicted in Figure 7.

The algorithm **TempMatch** next starts its two main subroutines, first **ForwardMatch** and afterwards **BackwardMatch** to determine the maximal matches of $T$ in $C$ with the chosen starting gates and on the chosen qubits (in the fixed order).

**Forward matching.** The subroutine **ForwardMatch** (Algorithm 4) is an algorithm that finds the maximal match of $T$ in $C$ in forward direction, i.e., with the gates that must follow the starting gate. In the graph representation this corresponds to vertices that are successors of the starting vertex. Since the gates considered by **ForwardMatch** cannot be commuted to the left of the starting gate, it can be proven that a greedy matching strategy is optimal (see Lemma A.1).

The algorithm starts with initializing a list $MatchedVertexList = (G^C_8)$ of matched vertices that have direct successors left to consider for matching. The attribute $SuccessorsToVisit$ of $G^C_8$ is then set equal to $(G^C_9, G^C_12, G^C_14, G^C_16)$ which are the direct successors of the starting vertex.

First, the vertex with lowest label in $SuccessorsToVisit$, i.e., the vertex $G^C_9$, is considered for matching (and removed from the list $SuccessorsToVisit$). The candidates of the template for a match with $G^C_9$ are the direct successors of the starting vertex $G^T_1$, i.e., the gates with labels $(4, 6)$ in the template (which are found by Algorithm 6). Since $T_6 = C_9$, we found a match, and we set the attribute $matchedWith$ of vertex $G^C_9$ equal to $G^T_6$ (see the code after line 20 in Algorithm 4). Once in the further matching process we have to consider all the direct successors of $G^C_9$ as candidates for a match. Hence, we set the attribute $SuccessorsToVisit$ of $G^C_9$ equal to $(G^C_{10})$ and add the vertex $G^C_9$ to the list of matched vertices $MatchedVertexList$. We order the list $MatchedVertexList$ in increasing order according to the smallest label in the attribute list $SuccessorsToVisit$ of the vertices. In the considered case, this label is equal to 10 for $G^C_9$ and 12 for $G^C_9$, hence we end up with $MatchedVertexList = (G^C_9, G^C_9)$.

In the second cycle of the while-loop in Algorithm 4, we consider the direct successors of the first vertex in $MatchedVertexList$ for further matches, i.e., we consider the vertex $G^C_{10}$ from $G^C_9$.successorsToVisit. Since $G^C_9$ is matched with $G^T_6$, the possible matching candidates are the direct successors of $G^T_6$, i.e., the vertex $G^T_7$. Again we find that $T_7 = C_{10}$ and we can match the two gates, set $G^C_{10}.successorsToVisit = G^C_{13}$ and add $G^C_{10}$ to $MatchedVertexList$.

Similarly, in the next few cycles of the while loop the vertex $G^T_12$ is matched with $G^C_{12}$ and the vertex $G^T_13$ is matched with $G^C_{13}$. Then, vertex $G^T_{14}$ (as a direct successor of vertex $G^C_9$) is considered for matching with direct successors of the vertex $G^T_7$ (which is matched with $G^C_{15}$). However, both direct successor $G^T_7$ and $G^T_9$ of $G^T_7$ have already been matched, and hence, there are no candidates for matching in the template and we block vertex $G^C_{14}$ (see the code after line 28 in Algorithm 4). Note that blocked vertices will not be considered further in the matching process. The resulting states of the graphs $G^T$ and $G^C$ are shown in Figure 8.

The matching process then goes on as follows: the vertex $G^T_7$ is matched with $G^C_{15}$; the vertex $G^C_{16}$ is blocked; the vertex $G^C_{17}$ and all its successors $G^C_{19}, G^C_{20}, G^C_{21}$ are blocked; the vertex $G^C_{10}$ is matched.
with $G^T_{11}$; the vertex $G^C_{11}$ is blocked (note that its successor $G^C_{21}$ is already blocked); the vertex $G^T_{12}$ is matched with $G^C_{22}$; and finally, the vertex $G^T_{11}$ is matched with $G^C_{24}$. The resulting match after finishing the run of ForwardMatch is shown in Figure 9.

Figure 8: Matched gates in the middle of running ForwardMatch (we have just blocked vertex $G^C_{14}$ and the next vertex to consider is $G^C_{15}$). The already matched gates are marked in lime and the starting gates are marked in gray. The blocked vertex is marked in black. At this stage of the algorithm, we have $MatchedVertexStack = (G^C_{12}, G^C_{13}, G^C_{8})$ and the successor to visit from the point of view of $G^T_{12}$ and $G^C_{13}$ is $G^C_{15}$, where the successor to visit from the point of view of $G^C_{8}$ is $G^C_{16}$. The vertices in the union of the successors left to visit from any point of view of vertices in $MatchedVertexStack$ are marked in blue.

Figure 9: Situation after ForwardMatch, where the matched vertices are marked in lime and the starting vertices are marked in gray. The blocked vertices (which will never match anymore) are marked in black.

Backward matching. It remains to find all maximal matches by also considering the vertices that are are not successors of the starting vertex $G^C_{8}$. In general, it could help to move gates that disturb this matching process as far as possible to the right (which corresponds to blocking the corresponding vertex and its successors). However, this might block some gates that have already been matched in the forward matching process. Hence, there is a tradeoff between matching more gates on the left or on the right of the starting gate. This tradeoff makes the matching process costly in general, since one has to go through all the possibilities of moving disturbing gates to the left or to the right. However, based on the fact that we have already found a maximal match in the forward direction of the starting gate (by running ForwardMatch), we can show that this process is still efficient in the circuit size (but not in the template size in the worst-case) as discussed in the proof of Lemma 4.11 given in Section 4.5.

We now start the method BackwardMatch (Algorithm 5). The list of all vertices in $G^C$ that have not been matched and are not blocked is given by $GateIndices = (11, 7, 6, 5, 4, 3, 2, 1)$ (ordered in decreasing order). We start by picking the largest index, i.e., we consider vertex $G^C_{11}$ for matching. The index 5 in the template is then found by FindBackwardCandidates, i.e., the only candidate in the template for a next match (in backwards direction) is the direct predecessor $G^T_{5}$ of vertex $G^T_{12}$. 


Since $T_5 \neq C_{11}$, we block the vertex $G_{11}^T$ according to line 43 in Algorithm 5. Similarly, we block the vertices $G_5^T$ and $G_6^T$ in the following.

Then, vertex $G_5^C$ is considered for matching. We find that it matches the candidate vertex $G_5^T$, however, as we will see in the following, greedy matching is not always the optimal strategy. According to the code after line 27 we have to consider the option that we match the vertices (Figure 10a), but also the one of blocking $G_5^C$ and all of its successors (Figure 10b). Although the second option destroys the already matched gate $G_{22}^C$, blocking $G_5^C$ might help to match further gates on the left (which will turn out to be the case in the considered example). We have to consider both scenarios in the further matching process and we put both of them to a stack $MatchingScenarios$ to keep track of them.

In the case where we matched the vertex $G_5^C$, one finds that no further gates can be matched (without blocking the matched gate $G_5^C$, what we do not have to consider, since it would lead to the same scenario as the non-matching case already added to the stack $MatchingScenarios$). Hence, we could match 10 gates in total in this scenario.

In the case where we do not match vertex $G_5^C$, one finds (by going through some different matching scenarios not listed here for readability) that the vertices $G_2^T$, $G_3^T$, $G_5^T$ can be matched with $G_1^C$, $G_2^C$, $G_3^C$ (Figure 11). Hence, we can match 11 gates in total in this matching scenario, which turns out to be the maximal match. The maximal match in the circuit picture is shown in Figure 12.

4.4 Correctness of the algorithm

In this section we formally prove the correctness of TempMatch (Algorithm 3), i.e., that for any circuit $C$ and any template $T$ the algorithm finds all maximal matches. Let us first formally define what we
Definition 4.2 (Connected part of a circuit). We say that $E = (E_1, \ldots, E_{|E|})$ is a connected part of a circuit $C = (C_1, \ldots, C_{|C|})$ if, by commuting gates pairwise with each other, one can bring the circuit to the form $C \simeq (D, E, F)$, where the circuits $D$ and $F$ consist of all the gates of the circuit $C$ except the ones listed in $E$.

In terms of the canonical form, a part $E$ of a circuit $C$ is connected if and only for all the vertices corresponding to the gates in $E$ in the canonical form of $C$ we have the following property: if two vertices are connected by a path, then all the vertices that lie on the path have also to correspond to gates in $E$.

Definition 4.3 (Equivalence of circuits up to qubit relabeling). A circuit $C$ is equivalent to a circuit $E$ up to qubit relabeling if and only if there exists a bijective mapping from the qubit labels in circuit $C$ to the ones in circuit $E$, such that for the resulting circuit $C'$ (that one gets by relabling the qubits in circuit $C$) we have that $C' = E$ (i.e., $C'_i = E_i$ for all $i \in \{1, 2, \ldots, |C| = |E|\}$).

Definition 4.4 (Template match). We say that a template $T$ has a match of length $m$ in a circuit $C$ if there exists a connected part $E^T$ of $T$ of length $|E^T| = m$ that is equal up to qubit relabeling to a connected part $E^C$ of $C$. We refer to such a match $M$ by a set of tuples of gate indices, where a tuple $(i, j)$ means that we matched the gate $T_i$ with the gate $C_j$.

Definition 4.5 (Maximal template match). We say that a match $M$ is maximal, if there are no matches $\tilde{M}$ in the circuit, such that

- $|\tilde{M}| > |M|$, and
- $M \cap \tilde{M} \neq \emptyset$, i.e., $M$ and $\tilde{M}$ have at least one element of matched gate indices $(i, j)$ in common.

Definition 4.6 (Equivalence of sub-circuits). Let us consider a circuit $C$ and two subset of gate indices $A, B \subset \{1, \ldots, |C|\}$ with $|A| = |B|$. We say that the subsets $A$ and $B$ describe equivalent sub-circuits of $C$ if and only if there exists a bijective mapping $f : A \mapsto B$, such that for all $i \in A$ we have

- $C_i = C_{f(i)}$, and
- the gates in the circuit $C$ can be commuted, such that in the resulting circuit we have the gate with index $f(i)$ at the positions $i$. 

Figure 12: Maximal match of template $T$ found in the circuit $C$ by TempMatch with initially matching $T_1$ with $C_8$. The gate indices are shown on the top. The indices at the bottom of the circuit indicate with which gate of the template the gate in the circuit is matched. The only gate in the template that cannot be matched is $T_{12}$. Consider to be a “template match” and when it is called maximal.
Definition 4.7 (Equivalence of template matches). For a match \( Q \), let us denote the set of matched indices in the template by \( Q^T := \{ (i, j) \in Q \} \) and the set of matched indices in the circuit by \( Q^C := \{ (i, j) \in Q \} \). We say that two matches \( M \) and \( \tilde{M} \) are equivalent if and only if

- \( M^T \) and \( \tilde{M}^T \) describe equivalent subcircuits of the template \( T \) and
- \( M^C \) and \( \tilde{M}^C \) describe equivalent subcircuits of the circuit \( C \).

We are now ready to state and prove the formal statement ensuring the correctness of TempMatch. This ensures that Algorithm 3 always succeeds, i.e., there are no situations where the algorithm does not deliver the desired output.

Theorem 4.8 (Correctness of TempMatch). Given a circuit \( C \) and a template \( T \). Then Algorithm 3 finds all maximal template matches (up to equivalent matches) of \( T \) in \( C \).

We note that not all the matches given as an output of Algorithm 3 might be maximal and there may be equivalent matches in the output. We ignored this for simplicity of the pseudocode and since for certain applications, it might be more efficient to work with this output instead of removing the non maximal and equivalent matches from it. The proof of Theorem 4.8 is given in Appendix A.

4.5 Complexity of the algorithm

In the previous section we have learned that Algorithm 3 is correct in the sense that it always finds all the maximal matches for any given template and any given circuit. Thus what remains to be understood is how efficient Algorithm 3 finds these matches. This is settled by the following theorem.

Theorem 4.9 (Complexity of TempMatch). The worst-case time complexity of Algorithm 3 for a circuit \( C \) and a template \( T \) is \( \mathcal{O} \left( |C|^{|T| + 3} |T|^2 \frac{n_C!}{(n_C - n_T)!} \right) \).

We note that the running time stated in Theorem 4.9 may be simplified as

\[
\mathcal{O} \left( |C|^{|T| + 3} |T|^2 \frac{n_C!}{(n_C - n_T)!} \right) \leq \mathcal{O} \left( |C|^{|T| + 3} |T|^2 n_C^{n_T} \right). 
\]

From this we see immediately that Algorithm 3 is efficient (i.e., polynomial) in \( |C| \) and \( n_C \) and inefficient (i.e., exponential) in \( |T| \) and \( n_T \).

To prove the assertion of Theorem 4.9 we first need to understand the running time of the two subroutines ForwardMatch and BackwardMatch. This is done in the following lemmas, where we assume that \( |T| \leq |C| \). Furthermore, we assume that we can check if two gates commute in constant time. For a fixed (finite) gate set, one possibility to achieve this, is by storing the commutation relations between all the gates in a table. Since we will account time complexity \( \mathcal{O}(|D|^3) \) to create the canonical form of a circuit \( D \) in line 3 and line 4 in Algorithm 3, we can assume constant time access to an ordered list of successors or predecessors of all vertices in the canonical form (see Remark 3.1).

Lemma 4.10 (Complexity of ForwardMatch). The worst-case time complexity of Algorithm 4 for a circuit \( C \) and a template \( T \) is \( \mathcal{O}(|T||C|^2) \), under the assumption that we have constant time access to an ordered list of all successors for any vertex in the canonical forms \( G^C \) and \( G^T \) of the circuit \( C \) and the template \( T \), respectively.

Proof. The while-loop runs at most \( |T||C| \) times, since there are at most \( |T| \) vertices that can be matched and added to MatchedVertexList and for each vertex, there are at most \( |C| \) successors to visit. The method MatchedVertexList.insert has worst-case time complexity \( \mathcal{O}(|T|) \) (since the attribute SuccessorsToVisit is an ordered list). Finding the candidates for further matches by running FindForwardCandidates has worst time complexity \( \mathcal{O}(|T|) \), since we have at most \( |T| \) successors to
consider (and we have constant time access to a list that contains all the successors). Let us now analyze the complexity of the two cases of the if-condition in line 19 in Algorithm 4 separately.

In the case, where we found a match (see line 20 in Algorithm 4), the most expensive part is to order the set \( S \) that contains at most \( |C| \) successors that we have left to visit from the point of view of the vertex \( v \). Ordering the elements in \( S \) has worst time complexity \( O(|C| \log |C|) \). Since the case where we found a match can occur at most \( |T| \) times, the full complexity of this case is \( O(|T| |C| \log |C|) \).

In the case, where we cannot match (see line 28 in Algorithm 4), we have to block the vertex \( v \) and all of its successors. Since we assume constant time access to a list containing all successors of \( v \), this can be done in time \( O(|C|) \). The case where we cannot match can occur at most as many times, as we have to run the while-loop. Hence, the full complexity of this case is \( O(|T| |C|^2) \).

We conclude that the worst-case complexity of ForwardMatch is given by \( O(|T| |C|^2 + |T|^2 |C|) = O(|T| |C|^2) \), where the second term \( |T|^2 |C| \) arises from line 13 and line 18 in Algorithm 4.

**Lemma 4.11** (Complexity of BackwardMatch). The worst-case time complexity of Algorithm 5 for a circuit \( C \) and a template \( T \) is \( O(|T| |C|^{2+|T|}) \), under the assumption that we have constant time access to an ordered list of all successors and predecessors for any vertex in the canonical forms \( G^C \) of the circuit \( C \).

**Proof.** The complexity of BackwardMatch depends on the number of possible matching conditions that are added to the stack MatchingScenarios during the while-loop. We consider the tree of possible matching conditions, i.e., we start from an initial match \( M_{\text{forward}} \) and consider the branching defined by the two options considered in both cases of the if-condition in line 26 in Algorithm 5. First, we calculate an upper bound on the number of vertices of this tree, i.e., an upper bound on the number \( \text{It}(C, T) \) of iterations in the while-loop. The branching that happens if we can match (see line 27 and the following lines in Algorithm 5), adds at most one matching condition to the stack with a counter increased by one and one matching condition where in addition a further gate of the template is matched. The branching that happens if we cannot match (see line 42 and the following lines in Algorithm 5), adds at most one matching condition to the stack with a counter increase by one and one matching condition where in addition at least one of the initially matched gates (listed in \( M_{\text{forward}} \)) was disturbed. Note that the gates matched during the run of BackwardMatch (and stored in \( M_{\text{forward}} \)) will not be disturbed again later on (see the condition to add a matching condition to the stack in line 38 and line 59 in Algorithm 5).

Let us denote the number of cases where we disturb an initially matched gates in a certain branch \( \beta \) of the binary tree by \( t_d(\beta) \), and the number of cases where we add an additional match by \( t_m(\beta) \). Then, each branch \( \beta \) of the binary tree must satisfy \( t(\beta) := t_d(\beta) + t_m(\beta) \leq |T| \), since the matching process ends if all gates of the template are matched or were disturbed (and excluded for further matching). Each branch of the binary tree can contain at most \( |C| \) vertices, since each vertex reduces the size of the circuit that is left to consider for matching by one (since counter is increased by one). Therefore, and since at each branching either \( t_d \) or \( t_m \) is increased by 1 for one of the two branches, the number of branches of the binary tree is bounded by \( \sum_{i=0}^{|T|} \binom{|C|}{i} \leq (1 + |C|)^{|T|} \). Since each branch is of length at most \( |C| \), we find

\[
\text{It}(C, T) \leq |C|(1 + |C|)^{|T|}. \tag{3}
\]

Let us now consider the complexity of the computation at each vertex of the tree. The complexity of FindBackwardCandidates is \( O(|T|^2) \), since, in the worst-case, we have to check for each gate in \( T \) if it can be commuted to the current matching position. By assumption, we have constant time access to all predecessors of a vertex in the canonical form \( G^C \) of the circuit \( C \). Hence, the calculation at each vertex has time complexity \( O(|C| + |T|^2) = O(|C||T|) \). This finishes the proof.

**Proof of Theorem 4.9.** The assertion of Theorem 4.9 now follows from Lemma 4.10 and Lemma 4.11 and inspection of the loop structure in TempMatch.
4.6 Heuristics to reduce the runtime

Since the worst-case complexity of the algorithm can be a high degree polynomial in the circuit size and in the number of qubits in the circuit for large templates (see Theorem 4.9), one might want to introduce some heuristics to lower the runtime. This is however in tradeoff with finding all the matches. For the complexity analysis here we assume a fixed template size. The computationally expensive steps in the matching algorithm are:

1. Choosing the qubits in a circuit $C$ on which we would like to search for a match with a template $T$ (see line 9 in Algorithm 3).

2. Considering all possible matching conditions during the run of BackwardMatch (see the two options in the two cases starting in line 27 and line 42 in Algorithm 5).

Possible heuristics to reduce the runtime of each of these cases could be:

1. We fix a maximal “looking forward” length $l \in \mathbb{N}$ and start from the first matched gate $C_r$ in the circuit. Then, we take the first $l$ successors of $C_r$ with the minimal labels and store them in a gate list $L$. We consider only the qubits in a set $Q_L$ as possible candidates for the matching, where the set $Q_L$ contains all the qubits for which there is a gate in $L$ that acts non trivially on the qubit. This reduces the theoretical complexity term $O(n_C n_T)$ appearing in Theorem 4.9 to a constant, i.e., $O(1)$ under the assumption that all the gates in the considered gate library act on a bounded number of qubits. (The additional complexity for searching the successors stored in $L$ is $O(1)$ by Remark 3.1).

Furthermore, we may use some heuristics to choose “promising” subsets of the qubits listed in $Q_L$ during the matching process. For example, one may consider the first gate $T_a$ in the template that connects the already matched gates with a new qubit. Then we search for all gates of type $T_a$ in the gate list $L$ whose relation to the qubits on which $C_r$ (and the other matched gates in the circuit) are acting on corresponds to the relation of the qubits in the template. The qubits connected through the so found gates might be chosen as candidates for the corresponding qubit in the template.

2. We fix a maximal branch length $b \in \mathbb{N}$ and a maximal number of survivors $s \in \mathbb{N}$. In the algorithm BackwardMatch, we add all the matching conditions to the stack until each of them has undergone $b$ iterations of the while-loop. Then, we evaluate how promising each of the matching condition is under some chosen metric. For example, one may choose the number of already matched gates for such a metric. We remove all the matching conditions from the stack up to $s$ remaining ones with the highest value with respect to the chosen metric. Then we evolve these matching conditions further, and so on. Note that such a heuristic reduces the complexity of BackwardMatch from $O(|C|^{2+|T|})$ (see Lemma 4.11 and recall that we have a fixed template size) to $O(|C|^3)$.

Alternatively, one may also consider a more clever order to visit the gates in the circuit. The current algorithm traverses the gates in decreasing order according to their labels (by using a counter over an ordered list GateIndices of gate indices, see Algorithm 5). However, changing the order to higher the chances that gates that matches are considered earlier, may significantly lower the runtime of an practical implementation.

We conclude that under the two mentioned heuristics the full matching algorithm has a worst-case time complexity of $O(|C|^3)$ for a fixed template size.

Remark 4.12. Let us assume that an application uses a certain “stopping criterion” for the algorithm TempMatch, for example, it might be fine to finish the search for maximal matches if a match of a certain length is found. For this use case, it is important to traverse the tree of matching scenarios as discussed in 2 above, in a way that is adapted to the class of circuits one considers. For example, one could start with a greedy strategy and always match gates if possible. Then one could consider the options where only once we do not match and so on.
5 Outlook

In this paper we presented an efficient algorithm for template matching in quantum circuits and hence, in particular, also for sub-circuit searches in reversible logic circuits. This provides a step to overcome the tradeoff existing today between not finding all the matches and long runtimes.

Currently, we are working on an implementation of the matching algorithm in the IBM software library Qiskit \([17]\) to test its runtime for practical applications. As mentioned, we expect considerably better scaling on average than given by the theoretical worst-case analysis (see Theorem 4.9). Furthermore, we plan to implement the heuristics explained in Section 4.6 to investigate how much they can speed up the algorithm and how many matches are missed in the average case, depending on the choice of the parameters for the heuristics. It would be interesting to see how many of the maximal matches can be found in a fraction of the runtime of the full algorithm using search techniques adapted to certain classes of circuits (see Remark 4.12). Plotting the number of found matches against the runtime of the algorithm for different greedy search heuristics might lead to insights in the structure of common quantum circuits.

To use our algorithm for circuit optimization, it would be interesting to find further templates. In general, not too many templates are known. Universal decomposition schemes (such as the one given in \([18]\) and implemented in \([19]\)) can in principle produce whole parameter groups of templates. It remains to be investigated how to make use of such templates in practice.

Another application of our algorithm is peephole optimization of quantum circuits. There, instead of searching for a template \(T\) in a circuit \(C\), we may search for the longest connected parts in \(C\) on a chosen subset of qubits \(L_q\) (i.e., every gate that only acts on these qubits can be considered as a match). If one finds a connected part \(E\) in \(C\) that requires more gates than a generic unitary on \(|L_q|\) qubits (see \([18]\) for the best known gate counts for arbitrary isometries), one may multiply the gates listed in \(E\) to find the unitary \(U_E\) that describes the whole operation performed by the \(E\). The unitary \(U_E\) could then be synthesized with the best known methods, and the circuit \(E\) can be replaced with the newly synthesized circuit. A version of peephole optimization on two qubits is already implemented in the IBM software library Qiskit \([17]\) and turned out to be very useful for circuit optimizations.

Another interesting direction is to generalize the template matching algorithm to arbitrary DAGs, where vertices can be interchanged under certain predefined rules (corresponding to the commutation relations between the different quantum gates in the circuit picture). Representing quantum circuits as DAGs, our current algorithm could be considered as a subgraph search algorithm. However, it requires some additional structure, such as that the numbers of incoming edges to a vertex is equal to the number of outgoing edges (since quantum gates are reversible). It would be interesting to see if all such restrictions could be removed or if some of them are necessary for the efficiency of the template matching algorithm. Searching for patterns in graph-structured data has applications in a broad range of areas, such as biology, computer vision, electronics, computer aided design, social networks, intelligence analysis, and artificial intelligence (see \([20]\) for an overview). Hence, we would expect our algorithm to find further applications in different fields, where it is natural that certain vertices of the graph can be interchanged.

Acknowledgements. We thank Roger Colbeck, Ali Javadi-Abhari, Dmitri Maslov, Mathias Soeken and Tony Metger for helpful discussions. We would like to thank Luca Mondada for working on a template matching algorithm as a semester thesis at ETH Zürich \([21]\). RI and DS acknowledges support from the Swiss National Science Foundation through SNSF project No. 200020-165843 and through the National Centre of Competence in Research Quantum Science and Technology (QSIT).

A Correctness (proof of Theorem 4.8)

To prove the assertion of Theorem 4.8 we need a couple of lemmas that ensure the correctness of the individual subroutines used by the TempMatch algorithm.
Lemma A.1 (Correctness of \texttt{ForwardMatch} (Algorithm 4)). Given a circuit $C$, a start index $r$, a set of qubits $L_q$, and a template $T$, such that $(T_1, T_{i+1}, \ldots, T_{|T|}) \simeq (T_{\text{backward}}^1, T_{\text{forward}}^r)$ for some $i \in \{1, \ldots, |T|\}$ and with $T_{\text{forward}}^1 = T_i$. Let us further assume that the circuit $T_{\text{forward}}$ satisfies

$$[1, j]_{T_{\text{forward}}} \neq 0 \quad \text{for all } j \in \{2, 3, \ldots, |T_{\text{forward}}|\}. \quad (4)$$

Then Algorithm 4 finds one of the equivalent maximal matches of the template $T_{\text{forward}}$ on the qubits with labels in $L_q$ in $C$ that starts by matching the gate $C_r$ with the gate $T_{\text{forward}}^1$ and keeps the qubit order of the template.\footnote{This means that the first qubit in the template is mapped to the qubit with the smallest label in $L_q$, the second qubit is mapped to the second smallest label in $L_q$, and so on.} Furthermore, \texttt{ForwardMatch} only blocks vertices in the canonical form of the circuit $C$ that correspond to gates that could never be matched with gates in the full template $T$ (or which would lead to equivalent matches).

\textbf{Proof.} Let us denote the canonical form of the circuit $C$ and of the template $T$ by $G^C$ and $G^T$, respectively. We have only to consider gates $C_s$ with $G^C_s \in \text{Succ}(G^C_r, G^C)$ for matching with $T_{\text{forward}}$. Indeed, let us show this rigorously.

\textbf{Matching gates must correspond to vertices in $G^C$ that are successors of $G^C_r$:} Assume by contradiction that we have given a proper match (see Definition 4.4) where a gate $T_j \in T_{\text{forward}}$ is matched with a gate $C_s$ (with $s \neq r$), such that $G^C_s \notin \text{Succ}(G^C_r, G^C)$. By (4) and Lemma 3.3, we have $G^T_j \in \text{Succ}(G^T_r, G^T)$. Choose a path $p$ from the vertex $G^T_r$ to $G^T_j$. Since a proper match is connected, we find that all the gates with labels corresponding to the vertices on the path $p$ must also be matched with gates in the circuit. Therefore, we find a corresponding path from $G^C_r$ to $G^C_s$ in the circuit, and hence $G^C_s \in \text{Succ}(G^C_r, G^C)$, which contradicts the initial assumption.

Let us now go on with the proof of the correctness of \texttt{ForwardMatch}. By construction, the algorithm goes through all the successors of $G^C_r$ in the while-loop, and hence it visits all gates that possibly could be matched with gates in $T_{\text{forward}}$. Indeed, the successors of $G^C_r$ are either considered as a direct successor of a matched vertex that was inserted into the list \texttt{MatchedVertexList}, or as a successor of a vertex that is blocked in a cycle of the while-loop (see line 29 in Algorithm 4). Moreover, the order of considering the successor vertices of $G^C_r$ that might be matched corresponds to the order of the corresponding gates in the circuit (from left to right). This is ensured by always considering the successors with minimal label that we have to visit first in the while-loop.

It remains to be shown that \texttt{ForwardMatch} handles each successor of $G^C_r$ in a way that leads to one of the (equivalent) longest matches. Since a proper match must be connected and we start matching $C_r$ with $T_{\text{forward}}^1$ from the left to the right in the circuit, the indices \texttt{CandidateIndices} of the gates in the template that could be matched next must be direct successors of the already matched gates. These successors are found in Algorithm 6. We have the following two cases in the while-loop for a matched root vertex $v_0$ and a direct successor $v$ of $v_0$ with label $s$ that we consider for matching in the canonical form $G^C$ of the circuit $C$.

1. \textbf{Gate $C_s$ matches with a gate $T_j$ with $j \in \texttt{CandidateIndices}$.} In the following we show that the optimal strategy in this case is indeed to greedy match the two gates. There are two cases:

   \begin{enumerate}
   \item[(a)] If the vertex $v = G^C_s$ is the only successor of $G^C_r$, such that $C_s = T_j$, we should match the two gates, since matching them will not disturb any possible further matches (but can only lead to longer matches). Indeed, not matching would reduce the possible candidates in the template for further matches and might block vertices in $G^C$ that are successors of the not matched gate. Note that the order in which the gates are visited in the while-loop ensures that all predecessors of $v$ that are also successors of the starting vertex $G^C_r$ are already matched if we end up in line 20 in Algorithm 4, and hence, matching leads to a proper connected match.
   \item[(b)] However if we have a further vertex $G^C_i \in \text{Succ}(G^C_r, G^C)$ and with $C_i = T_j$, it could a priori happen that we should not match, because matching $G^C_i$ instead of $v$ could lead to a longer
   \end{enumerate}
match (see point 2 in Section 2). In the following we show that such a vertex $G'^C_r$ would have to be a direct successor of the vertex $v_0$ to be properly matched (i.e., to lead to a connected match including the starting gate $G'^C_1$), and hence, matching $G'^C_r$ instead of $v$ would lead to equivalent matches. Indeed, if $G'^C_r \notin \text{DirectSucc}(v_0, G'^C_0)$, it has to be a successor of $v_0$ with at least one vertex in between (because $|C_{v_0, \text{label}}, C_i| \neq 0$, since $C_i = T_j = C_s$ and $G'^C_r \in \text{DirectSucc}(v_0, G'^C_0)$). In this case, the vertex in between cannot be matched, and hence this scenario would not lead to a connected match (as long as we do not block the matched predecessors of $G'^C_r$, which is not allowed since it would also block the starting vertex $G'^C_1$).

2. Gate $C_s$ does not match with a gate $T_j$ with $j \in \text{CandidateIndices}$. Similarly as above, one can see that matching the gate $C_s$ with any gate in $T_{\text{forward}}$ or in $T_{\text{backward}}$ cannot lead to a connected match. Since $G'^C_s$ is a successor of the starting vertex $G'^C_1$, all the successors of $G'^C_s$ can also not be matched, because matching them would not lead to a connected match.

This finishes the proof. \qed

For the proof of the correctness of $\text{BackwardMatch}$, we will further use the following fact related to the construction of the algorithm $\text{ForwardMatch}$.

**Definition A.2.** A match $S$ (i.e., a set $S$ of matched gate indices) is called sub-match of a match $M$, if there exists a match $\hat{S}$ that is equivalent to the match $S$ and such that $\hat{S} \subseteq M$.

**Lemma A.3** (Characterization of non maximal forward matches). Given a circuit $C$, a start index $r$, a set of qubits $L_q$, and a template $T_{\text{forward}}$ that satisfies the property (4). Then all possible (and not necessarily maximal) matches $M$ of the template $T_{\text{forward}}$ on the qubits with labels in $L_q$ in $C$ that start by matching the gate $C_r$ with the gate $T_{\text{forward}}^i = C_r$ and keep the qubit order of the template are sub-matches of the maximal match $M_{\text{max}}$ found by $\text{ForwardMatch}$.

*Proof.* Assume by contradiction that there is a match $M$ of $T_{\text{forward}}$ in the circuit $C$ (matching $T_{\text{forward}}^i$ with $C_r$), such that there is no sub-match in $M_{\text{max}}$ that is equivalent to $M$. Since the algorithm $\text{ForwardMatch}$ goes through all gates that could possibly be matched with $T_{\text{forward}}$ (see the proof of Lemma A.1), there is a first gate $C_s$ considered in this process that is not matched in the maximal match $M_{\text{max}}$ found by $\text{ForwardMatch}$ (and there is also no equivalent match). Since matching a gate in this process can never disturb future matches (apart from equivalent ones) as shown in case 1 in the proof of Lemma A.1, we conclude that the gate $C_s$ could also be matched in the forward matching process. However, according to the algorithm $\text{ForwardMatch}$, the gate would then indeed be matched, leading to a contradiction with the assumption that $C_s$ is not matched in the maximal match $M_{\text{max}}$ found by $\text{ForwardMatch}$. \qed

**Lemma A.4** (Correctness of $\text{BackwardMatch}$ (Algorithm 5)). Given a template $T$, such that for some $i \in \{1, \ldots, |T|\}$ we have $(T_i, T_{i+1}, \ldots, T_T) \simeq (T_{\text{backward}}, T_{\text{forward}})$ and such that $T_{\text{forward}}$ satisfies the property (4). Further, given a circuit $C$ together with one of the equivalent maximal matches $M_{\text{forward}}$ of $T_{\text{forward}}$ in $C$ on the qubits with labels given in a set $L_q$ and starting with matching $C_r$ with $T_{\text{forward}}^i$.

Then, Algorithm 5 finds all maximal matches of $T$ on the qubits with labels in $L_q$ in $C$ that starts with matching the gate $C_r$ with the gate $T_{\text{forward}}^i$ and keeping the qubit order of the template.

Let us first give an overview of the proof idea. Since we have already given a maximal match of $T_{\text{forward}}$ in $C$, it remains to verify how many gates of $T_{\text{backward}}$ can be matched. In general, it can happen that a gate that disturbs the match of $T_{\text{backward}}$ can be moved to the right in $C$. This may allow us to match further gates in $T_{\text{backward}}$, but may also disturb the maximal match of $T_{\text{forward}}$. To handle this tradeoff, we have to consider both possibilities: (i) moving the disturbing gate as far as possible to the right, and (ii) moving it as far as possible to the left. These options correspond to blocking the successors or the predecessors of the vertex corresponding to the disturbing gate in the canonical form of the circuit $C$. We then have to keep track of both possibilities and go on with
matching in both cases building up a stack of possible matching scenarios that could lead to a maximal match. Let us now give a detailed proof of the correctness of the algorithm.

Proof of Lemma A.4. The matching process works with the canonical representations $G^C$ of the circuit $C$, where the matched gates listed in $M^\text{forward}$ are marked as matched in $G^C$, and their successors are blocked (corresponding to the state after running **ForwardMatch**). Let us first show that the strategy of first maximally forward match and afterwards start the backward match leads to all possible maximal matches of the full template. A priori it could happen that a non maximal forward match could lead to a maximal match of the full template. However, by Lemma A.3, different forward matches could only be sub-matches of the maximal match. Since in the process of backwards matching (as analyzed in detail below) further matched gates on the right of the starting gate $C_r$ with gates in $T^\text{forward}$ can only lead to longer matches, we conclude that the strategy of first finding the maximal forward match is indeed optimal.

Let us now consider the backward matching process. The while-loop of the matching process in Algorithm 5 goes through all the vertices in the canonical form $G^C$ of the circuit that are not blocked or already matched (as long as there are gates left to consider in $T^\text{backward}$ that could possibly be matched). Since the blocked gates will never match (see Lemma A.1), we loop through all vertices corresponding to gates that could possibly be matched with $T^\text{backward}$. The indices of these vertices are stored in a list $\text{GateIndices}$ in decreasing order (see line 4 and line 5 in Algorithm 5). The variable $\text{counter}$ keeps track of the number of gates with indices listed in $\text{GateIndices}$ that we have already considered in the backwards matching process. During the matching, we create a stack of possible matching scenarios that may lead to a maximal match. All of these scenarios are then considered for further matching in the next steps of the while-loop. Therefore, it remains to show that each step in the while-loop with parameters $(G^C, \ M, \ \text{counter})$ puts all the matching possibilities for the gate with index $s = (\text{GateIndices})_{\text{counter}}$ that might lead to a maximal match on the stack $\text{MatchingScenarios}$ for further investigation. The indices $\text{CandidateIndices}$ of the gates in the template that could possibly match are the ones that can be moved to the left of the already matched gates in the template. These gates are found in Algorithm 7. To show the correctness of one cycle of the while-loop, let us consider two cases separately.

1. **Gate $C_s$ matches with a gate in the template with index $s \in \text{CandidateIndices}$.** First we note that if we decide to match the gate $C_s$, then matching it with another gate $T_k = T_j$ with $k \in \text{CandidateIndices}$ would lead to equivalent matches (since the gates $T_j$ and $T_k$ could in this case be commuted to the same place in the template). If we do not match, the gate $C_s$ is moved as far as possible to the right in the circuit $C$ in Algorithm 5 and the gates that cannot be commuted to the left of it are removed, i.e., we block the vertex $G^C_s$ and all its successors. If we disturb the match of the starting gate by doing so, we can ignore this matching scenario. Moreover, if we block a vertex corresponding to a gate $C_t$ that is matched with a gate in $T^\text{backward}$ by blocking the successors of $G^C_s$, we can also ignore this scenario, since it is already on the stack (because in an earlier cycle of the while-loop, we considered the case of not matching $C_t$). Hence, we have left to show that if we do not match, it is never necessary to move the gate $C_s$ as far as possible to the left in the circuit to find a maximal match. Indeed, moving the gate to the left (and ignoring it for the further matching process) could lead to one of the following two cases:

   (a) we match the gate $T_j$ later on in the matching process with a different gate in the circuit,
   (b) we do never match the gate $T_j$.

   It can be verified that case (a) leads to matches that are equivalent to the ones where we match $C_s$ with $T_j$. In case (b), not matching $T_j$ with $C_s$ can only disturb the backwards matching process later on (since it blocks gates that do not commute to the right of $C_s$ in the circuit and reduces the possible matching candidates from the template), but cannot increase the match. Hence, we can ignore the case where we move $C_s$ as far as possible to the left.
2. **Gate $C_s$ does not match with any gate in the template with index $s \in \text{CandidateIndices}$**

One can see that the gate $C_s$ can never match and may disturb the matching. We consider both options: moving it as far as possible to the left and to the right in the circuit and remove the trailing gates. If we can move the gate $C_s$ to the right of all the matched gates in $M_{\text{forward}}$ with $T_{\text{forward}}$ or to the start of the circuit $C$ (see the condition in line 45 in Algorithm 3), doing so will lead to a maximal match, since the gate $C_s$ does not disturb the following matching process or the current match. Hence, in this case we only add this possibility to the stack $\text{MatchingScenarios}$, where otherwise, we have to add both options of moving the gate $C_s$ as far as possible to the left or to the right. As in case 1, we do not have to add matching conditions to the stack if we disturb the initial match or any match with gates in $T_{\text{backward}}$.

This finishes the proof.

We are finally ready to prove the assertion of Theorem 4.8.

**Proof of Theorem 4.8.** Assume that there is a maximal match $M$ of a template $T$ in the circuit $C$. We have to show that this match is found by Algorithm 3. The algorithm loops over all possible choices of $n_T$ “matching”-qubits out of the $n_C$ qubits of the circuit, as well as over all possible permutations of them. The qubit relabeling in the template in line 12 in Algorithm 3 then ensures that there is a run of the loop with a template $\hat{T}$, such that for all the index pairs $(j, s) \in M$, we have $\hat{T}_j = C_s$ (which in particular means that the two gates are acting on qubits with the same labels). Assume that the indices of the gates $\hat{T}_1, \hat{T}_2, \ldots, \hat{T}_k$ are not listed (as a first entry of an element) in $M$, but $\hat{T}_{k+1}$ is the first matched gate in the template, i.e., there is a tuple $(k + 1, r) \in M$ for some $r \in \{1, 2, \ldots, |C|\}$. Then, once in the loop of the algorithm $\text{TempMatch}$, we set the start index of the template $i := k + 1$ and the start index in the circuit to $r$. By the correctness of $\text{ForwardMatch}$ and $\text{BackwardMatch}$ (see Lemma A.1 and Lemma A.4), all the maximal matches (up to equivalent ones) of $(\hat{T}_i, \ldots, \hat{T}_{|\hat{T}|})$ in the circuit $C$ are found starting with matching the gate $T_i$ with $C_r$. Since the gates $\hat{T}_1, \hat{T}_2, \ldots, \hat{T}_{i-1}$ are not contained in the match we are searching for, $\text{TempMatch}$ hence find the maximal match $M$ (or an equivalent one).

**References**

[1] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC ’96, pages 212–219, New York, NY, USA, 1996. ACM. doi:10.1145/237814.237866.

[2] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997. doi:10.1137/S0097539795293172.

[3] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.*, 103:150502, Oct 2009. doi:10.1103/PhysRevLett.103.150502.

[4] F. G. S. L. Brandao and K. M. Svore. Quantum speed-ups for solving semidefinite programs. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 415–426, Oct 2017. doi:10.1109/FOCS.2017.45.

[5] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549, Sep 2017. doi:10.1038/nature23474.

[6] D. Maslov, G. W. Dueck, D. M. Miller, and C. Negrevergne. Quantum circuit simplification and level compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, March 2008. doi:10.1109/TCAD.2007.911334.
[7] T. Watanabe, Makoto Endo, and N. Miyahara. A New Automatic Logic Interconnection Verification System for VLSI Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(2):70–82, April 1983. doi:10.1109/TCAD.1983.1270023.

[8] F. Luellau, T. Hoepken, and E. Barke. A Technology Independent Block Extraction Algorithm. In *21st Design Automation Conference Proceedings*, pages 610–615, June 1984. doi:10.1109/DAC.1984.1585862.

[9] M. Boehner. LOGEX—an automatic logic extractor from transistor to gate level for CMOS technology. In *25th ACM/IEEE, Design Automation Conference. Proceedings 1988.*, pages 517–522, June 1988. doi:10.1109/DAC.1988.14809.

[10] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather. SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm. In *30th ACM/IEEE Design Automation Conference*, pages 31–37, June 1993. doi:10.1145/157485.164556.

[11] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC ’71, pages 151–158, New York, NY, USA, 1971. ACM. doi:10.1145/800157.805047.

[12] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976. doi:10.1145/321921.321925.

[13] Md. Mazder Rahman, Gerhard W. Dueck, and Joseph D. Horton. An algorithm for quantum template matching. *J. Emerg. Technol. Comput. Syst.*, 11(3):31:1–31:20, December 2014. URL: http://doi.acm.org/10.1145/2629537, doi:10.1145/2629537.

[14] D. M. Miller, D. Maslov, and G. W. Dueck. A transformation based algorithm for reversible logic synthesis. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pages 318–323, June 2003. doi:10.1109/775832.775915.

[15] D. Maslov, G. W. Dueck, and D. M. Miller. Synthesis of Fredkin-Toffoli reversible networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(6):765–769, June 2005. doi:10.1109/TVLSI.2005.844284.

[16] N. Abdessaied, M. Soeken, R. Wille, and R. Drechsler. Exact template matching using Boolean satisfiability. In *IEEE 43rd International Symposium on Multiple-Valued Logic*, pages 328–333, May 2013. doi:10.1109/ISMVL.2013.26.

[17] Héctor Abraham et al. Qiskit: An open-source framework for quantum computing, 2019. doi:10.5281/zenodo.2562110.

[18] Raban Iten, Roger Colbeck, Ivan Kukuljan, Jonathan Home, and Matthias Christandl. Quantum circuits for isometries. *Physical Review A*, 93(3):032318, March 2016. arXiv: 1501.06911. doi:10.1103/PhysRevA.93.032318.

[19] Raban Iten, Oliver Reardon-Smith, Luca Mondada, Ethan Redmond, Ravjot Singh Kohli, and Roger Colbeck. Introduction to UniversalQCompiler, April 2019. URL: http://arxiv.org/abs/1904.01072.

[20] Brian Gallagher. The state of the art in graph-based pattern matching. Technical report, Lawrence Livermore National Lab, Livermore, CA, 2006.

[21] Luca Mondada. UniversalQCompiler: a quantum compiler from arbitrary isometries to elementary gates, 2018. Semester Thesis at ETH Zurich.