Reqomp: Space-constrained Uncomputation for Quantum Circuits

Anouk Paradis, Benjamin Bichsel, and Martin Vechev

ETH Zurich, Switzerland

Quantum circuits must run on quantum computers with tight limits on qubit and gate counts. To generate circuits respecting both limits, a promising opportunity is exploiting *uncomputation* to trade qubits for gates.

We present Reqomp, a method to automatically synthesize correct and efficient uncomputation of ancillae while respecting hardware constraints. For a given circuit, Reqomp can offer a wide range of trade-offs between tightly constraining qubit count or gate count.

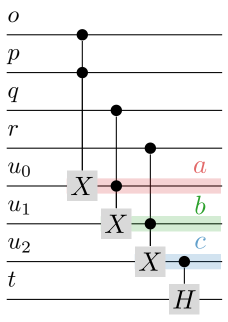
Our evaluation demonstrates that Reqomp can significantly reduce the number of required ancilla qubits by up to 96%. On 80% of our benchmarks, the ancilla qubits required can be reduced by at least 25% while never incurring a gate count increase beyond 28%.

1 Introduction

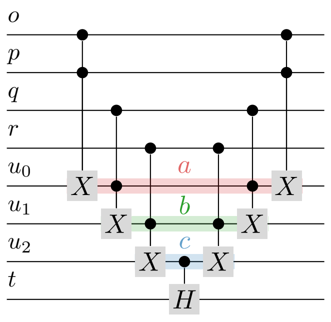
Quantum computers will remain tightly resource-constrained for the foreseeable future, both in terms of available qubits and number of operations applicable before an error occurs. Running quantum programs hence requires compiling them to circuits with a limited qubit and gate count. A promising opportunity to achieve this goal is to exploit the need for *uncomputation* as an opening to trade qubits for gates.

What is Uncomputation? Just as classical programs, quantum circuits often leverage temporary values, called *ancilla variables*. Whereas classical programs can discard temporary values whenever convenient, temporary values in quantum circuits must be carefully managed to avoid side-effects on other values through entanglement [1, §3]. Uncomputation is the process of preventing such side-effects by reverting ancilla variables to state $|0\rangle$ after their last use, thus ensuring that they are disentangled from the remainder of the state. For instance, Fig. 1 shows a circuit implementing $CCCCH$: the H gate on qubit t with four control qubits o, p, q, and r. Fig. 1a uses three ancillae variables a, b, c, stored in the respective *ancilla qubits* u_0, u_1, u_2. The first ancilla a holds $o \cdot p$,

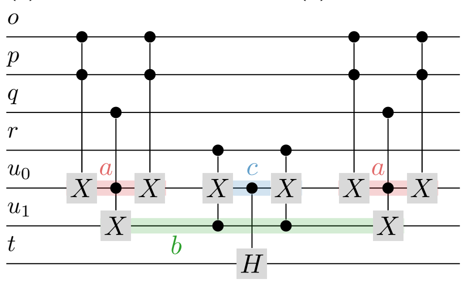
Anouk Paradis: anouk.paradis@inf.ethz.ch
Benjamin Bichsel: benjamin.bichsel@inf.ethz.ch
Martin Vechev: martin.vechev@inf.ethz.ch



(a) No uncomputation.



(b) 3-qubit uncomputation.



(c) 2-qubit uncomputation.

Fig.	#qb	#gt
1a	8	4
1b	8	7
1c	7	9

(d) Resources.

Figure 1: Two uncomputation strategies for $CCCCH$.

b holds $o \cdot p \cdot q$, and c holds $o \cdot p \cdot q \cdot r$. We then use this last ancilla c to control the H gate on t, only applying H if all of o, p, q, and r hold state $|1\rangle$. In Fig. 1a, these ancilla variables are not uncomputed, and may result in unexpected interactions if this circuit is used as part of a bigger computation. They must therefore be uncomputed, as shown in Fig. 1b: the operations applied to each of them are reverted at the end of the circuit, ensuring that all ancilla qubits are reset to $|0\rangle$.

Reducing Qubits. After uncomputing an ancilla variable, its qubit can be reused by another ancilla variable, therefore reducing the overall number of qubits used by the circuit. Sometimes, it is even beneficial to uncompute an ancilla variable (too) early, allowing its qubit to be reused at the cost of later *recomputing* the ancilla variable when it is needed again.

Fig. 1b simply uncomputes ancilla variables in the reverse order of their computation, namely c–b–a. As no ancilla qubit can be reused, Fig. 1b requires 8 qubits and 7 gates overall (see Fig. 1d). Fig. 1c shows an alternative implementation of $CCCCH$ leveraging *recomputation*. It uncomputes ancilla variable a early, making its qubit u_0 free for the computation of ancilla variable c. However, uncomputing b requires a again, forcing us to *recompute* it and subsequently uncompute it for a second time, at the cost of 2 additional gates. Overall, Fig. 1c thus trades qubits for gates compared to Fig. 1b, as summarized in Fig. 1d.

Correctness. Clearly, uncomputation is only useful if it correctly resets ancillae to $|0\rangle$ without modifying the remainder of the state. However, this is difficult to achieve, as uncomputing an ancilla may require some preprocessing on its controls, to ensure that they are in the right state (for details, see §7.1). Synthesizing the right gates to achieve uncomputation is thus a fundamental challenge, as evidenced by correctness issues in SQUARE [2] which attempts to automate the placement of programmer-defined computation and uncomputation blocks (see §7.1).

Our Work. We present Reqomp, a method to automatically synthesize and place correct yet efficient uncomputation while respecting hardware constraints. Reqomp takes as inputs a quantum circuit C without uncomputation (such as Fig. 1), its ancilla variables and a space constraint specifying the number of available ancilla qubits. If possible, Reqomp extends C to a circuit $\overline{C}$ which uncomputes all ancilla variables, using only the number of ancilla qubits specified.

To ensure Reqomp synthesizes correct uncomputation, it extends *circuit graphs* [1] (a graph representation of quantum circuits) by the new concept of *value indices* tracking the state of qubits. This allows Reqomp to determine which gates must be placed to reach the desired state.

Evaluation. Our experimental evaluation shows that Reqomp can significantly reduce the number of required ancilla qubits by up to 96% compared to the most relevant previous work Unqomp [1]. Many algorithms are amenable to a significant reduction: for 16 of 20 benchmarks, Reqomp can reduce the number of ancilla qubits by 25% compared to Unqomp, without incurring a gate count increase beyond 28%. For the remaining 4 examples, Reqomp strictly outperforms Unqomp, albeit by a smaller margin. In some cases, Reqomp achieves an impressive ancilla qubit reduction at very low cost: for one example, by 75% at the cost of increasing gate count by 17.6%.

Note that Unqomp already showed that manual uncomputation is both error-prone and less efficient than automatically synthesized uncomputation [1, §7].

Main Contributions. Our main contributions are:

- Reqomp, a method to synthesize and place uncomputation in circuits under space constraints (§3–§4);
- A correctness proof for Reqomp (§5);
- an implementation[1] and evaluation of Reqomp demonstrating it outperforms previous work (§6).

[1]Reqomp is publicly available at https://github.com/eth-sri/Reqomp.

2 Background

We now introduce the necessary background on quantum computation.

Quantum States. We write the quantum state φ of a system with qubits p and q as:

$$\sum_{j=0}^{1}\sum_{k=0}^{1}\gamma_{j,k}\,|j\rangle_p \otimes |k\rangle_q = \sum_{l\in\{0,1\}^2} \gamma_l\,|l\rangle_{pq} \in \mathcal{H}_2, \qquad (1)$$

where $\gamma_j, \gamma_k, \gamma_l \in \mathbb{C}$ and $\otimes$ is the Kronecker product. If φ factorizes into $\left(\sum_j \gamma'_j\,|j\rangle_p\right) \otimes \left(\sum_k \gamma''_k\,|k\rangle_q\right)$, p and q are *unentangled*, otherwise they are *entangled*. Whenever convenient, we omit $\otimes$ and write $|j\rangle$ instead of $|j\rangle_p$. We use latin letters $|j\rangle$ to denote computational basis states from the canonical basis $\{|0\rangle, |1\rangle\}$ and greek letters φ to denote arbitrary states.

Gates. A gate applies a unitary operation to a quantum state. Here, we only consider gates with a single target qubit in state φ and potentially multiple control qubits $C = \{c_1, ...\}$ in state $|j\rangle$ for $j \in \{0,1\}^m$, mapping $|j\rangle_C \otimes \varphi$ to $|j\rangle_C \otimes \phi$, where the mapping from φ to ϕ may depend on the control j. Specifically, only the value of the target qubit may be changed, while control qubits are preserved. Note that this mapping can be naturally extended to superpositions (i.e., linear combinations as in Eq. (1)) by linearity. Further, because any circuit can be decomposed into single-target gates, not considering multi-target gates is not a fundamental restriction.

A gate is *qfree* if its mapping can be fully described by operations on computational basis states, i.e., if for control qubits C and target qubit t it is of the form

$$|j\rangle_C\,|k\rangle_t \mapsto |j\rangle_C\,|F(j,k)\rangle_t\,,$$

for $F\colon \{0,1\}^m \times \{0,1\} \to \{0,1\}$. For example, the NOT gate X, the controlled NOT gate CX, and the Toffoli gate CCX are qfree, while the Hadamard gate H and the controlled Hadamard gate CH are not qfree. Qfree gates are known to be critical for synthesizing uncomputation [1, 3, 4].

Uncomputation. The task of uncomputation is to revert all ancilla variables in a circuit to their initial state $|0\rangle$, while preserving the circuit effect on the other variables. Formally, given a circuit C, we want to synthesize $\overline{C}$ which resets ancillae variables to $|0\rangle$ without affecting the remainder of the state:

Definition 2.1 (Correct Uncomputation, [1, 3])**.** *$\overline{C}$ correctly uncomputes the ancillae A in C if whenever*

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![C]\!]} \sum_{j\in\{0,1\}^{|A|}} \gamma_j\,|j\rangle_A \otimes \phi_j, \text{ then}$$

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![\overline{C}]\!]} \sum_{j\in\{0,1\}^{|A|}} \gamma_j\,|0\cdots0\rangle_A \otimes \phi_j.$$

Here, $[\![C]\!]$ denotes the semantics of circuit C acting on a given input state. We refer to [1] for a more thorough introduction to uncomputation.

3 Overview

We now showcase how Reqomp tackles the problem of ancilla variables uncomputation under space constraints. It takes as input a quantum circuit and a number of available ancilla qubits. If successful, it returns a quantum circuit where all ancilla variables from the original circuit are uncomputed and all other variables are preserved, using only the number of available ancilla qubits.

Example Circuit. Fig. 2 shows the algorithm of Reqomp and applies it to an example circuit with three ancilla variables, a, b and c. We note that while this circuit does not implement a relevant algorithm, it allows showcasing the key features of Reqomp on a simple example.

Reqomp Workflow. Fig. 2 highlights the steps performed by Reqomp, which we detail in §3.1–§3.4. First, Reqomp converts the circuit C into a circuit graph G (§3.1 and 🟥 in Fig. 2). Using this representation, Reqomp identifies the dependencies among ancilla variables in the circuit, and uses them to derive an uncomputation strategy respecting the number of available ancilla qubits (§3.2 and 🟦 in Fig. 2). Reqomp then applies this strategy to build a new circuit graph $\overline{G}$ containing uncomputation (§3.3 and 🟧 in Fig. 2). Finally, Reqomp converts the resulting circuit graph into a circuit $\overline{C}$ (§3.4 and 🟩 in Fig. 2).

3.1 Building the Circuit Graph

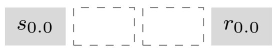
Reqomp does not work directly on circuits, but instead on an augmented version of the circuit graphs introduced in Unqomp [1], additionally tracking qubit values through *value indices*. For simplicity, we refer to those augmented circuit graphs simply as circuit graphs in the remainder of the text. The first step of Reqomp is hence to convert the circuit C in Fig. 2 to the circuit graph G. The Reqomp algorithm performs this step in Lin. 2 (see Fig. 2). This transformation also produces a value graph g^{val}, discussed below.

Vertices and Edges. The circuit graph G contains one *init vertex* per qubit (e.g., $s_{0.0}$ for qubit s), and one *gate vertex* per gate (e.g., $s_{1.0}$ for the first X gate on s). It also connects consecutive vertices on the same qubit by a *target edge*, e.g., $s_{0.0} \rightarrow s_{1.0}$. Further, as $a_{1.0}$ represents a CX gate controlled by qubit s, the circuit graph G also contains a *control edge* between the corresponding vertices on s and $a_{1.0}$: $s_{1.0} \bullet\!\!\rightarrow a_{1.0}$. Finally, the circuit graph G also contains *anti-dependency edges* to enforce the correct ordering between otherwise unordered vertices. For example, $a_{1.0} \dashrightarrow s_{0.1}$ ensures that the second X gate on s (represented by $s_{0.1}$) can only be applied after the CX gate targeting a (represented by $a_{1.0}$). Generally, circuit graphs must be acyclic, as translating them to a circuit requires applying its gate vertices in a topological order.

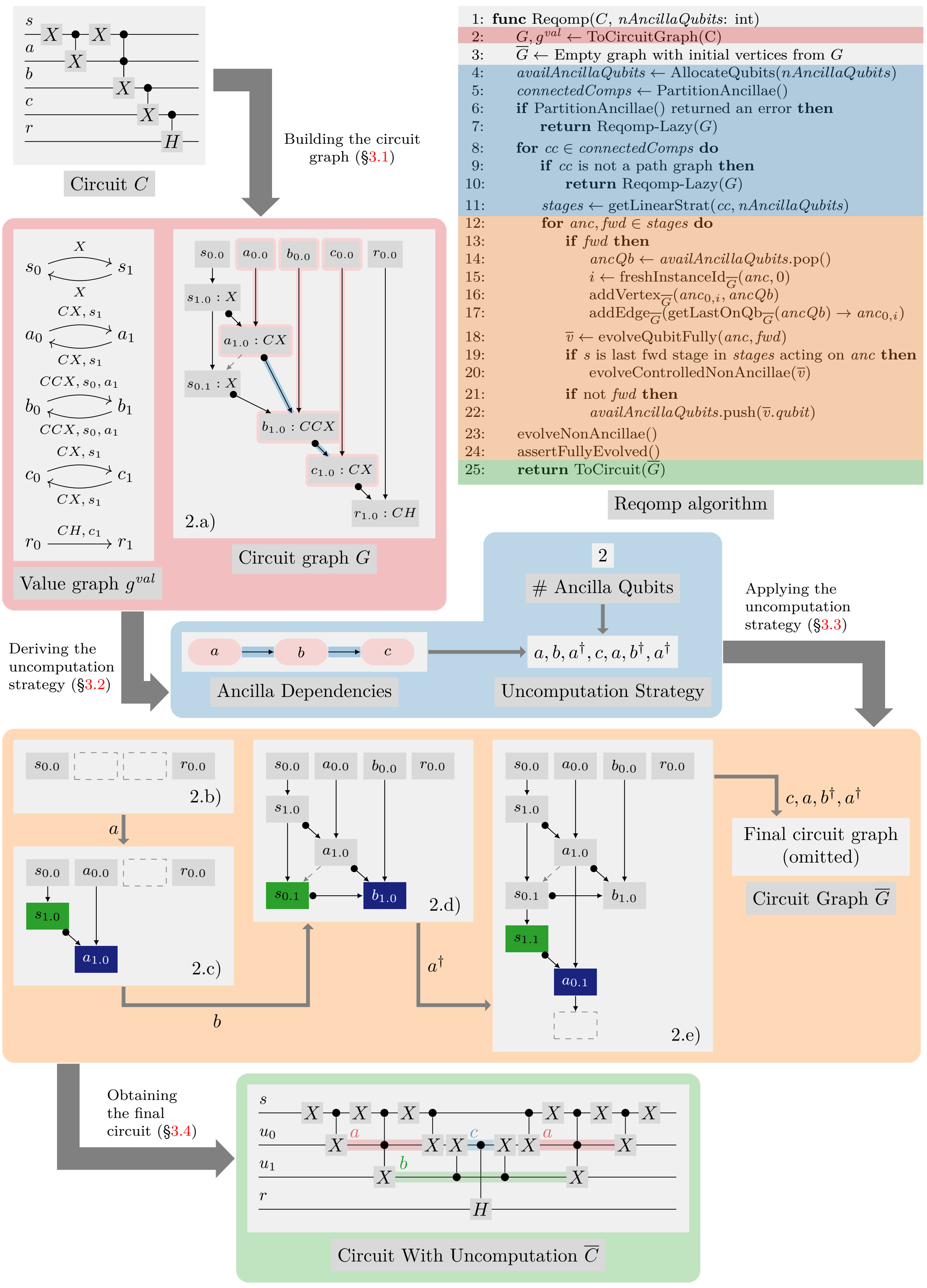
Tracking Values. While the above construction follows Unqomp [1], we additionally introduce a new vertex naming convention to track qubit values. Specifically, each vertex (e.g., $s_{1.0}$) is identified by its qubit (here s), its *value index* (here 1) and its *instance index* (here 0). The value index is chosen such that intuitively, two vertices with the same qubit and value index hold the same "value", even in the presence of entanglement. The instance index is used to ensure uniqueness of vertex names. For instance, as X is self-inverse, the value on qubit s is the same in the very beginning of the circuit ($s_{0.0}$) as after applying the two X gates to s ($s_{0.1}$). More precisely, if the input state to the circuit is $|0\rangle_s \otimes \varphi$, after the two X gates on s have been applied, the final state is $|0\rangle_s \otimes \varphi'$ for some φ'. Reflecting this in the circuit graph, vertices $s_{0.0}$ and $s_{0.1}$ share the same value index 0.

To track value indices during circuit graph construction and later during uncomputation, we rely on the value graph g^{val}, shown in Fig. 2. It records for each qubit and value index the possible value transitions: for instance a CX gate from a_0 with control s_1 yields a_1 (see Fig. 2). Note that we do not specify the instance index of s_1: as any vertex on qubit s with value index 1 carries the same value, any of them can be used as a control. The value graph g^{val} also records which operations can be safely uncomputed. For instance, as CX is a qfree gate, the CX on a_0 can be uncomputed: applying CX on a_1 with control s_1 yields a_0 (note that CX is self-inverse).

Preparing the New Circuit Graph. To apply the necessary uncomputation operations, Reqomp does not modify the circuit graph G built above, but instead creates a new empty circuit graph $\overline{G}$ where uncomputation and computation can be interleaved according to a chosen uncomputation strategy. In the Reqomp algorithm, Lin. 3 builds this new empty circuit graph $\overline{G}$ and Lin. 4 allocates *nAncillaQubits* qubits, used to store all the ancilla variables. [2] For our example, this results in a graph consisting of (i) init vertices for all non-ancillae variables and (ii) slots for all available ancillae qubits. Fig. 2b shows these slots as dashed blocks (Fig. 2 groups the new empty circuit graph with the section on applying uncomputation 🟧):



[2]Note that while *nAncillaQubits* does not constitute an upper limit on the physical qubits required overall (as non-ancillae are not taken into account), we can control the latter by decreasing or increasing *nAncillaQubits*.


Circuit C
Building the circuit graph (§3.1)
Value graph g^{val}
Circuit graph G
2.a)
Deriving the uncomputation strategy (§3.2)
Ancilla Dependencies
2
# Ancilla Qubits
$a, b, a^\dagger, c, a, b^\dagger, a^\dagger$
Uncomputation Strategy
Applying the uncomputation strategy (§3.3)
2.b)
2.c)
2.d)
2.e)
$c, a, b^\dagger, a^\dagger$
Final circuit graph (omitted)
Circuit Graph $\overline{G}$
Obtaining the final circuit (§3.4)
Circuit With Uncomputation $\overline{C}$
Reqomp algorithm

```
func Reqomp($C$, $nAncillaQubits$: int)
    $G, g^{val}$ ← ToCircuitGraph(C)
    $\overline{G}$ ← Empty graph with initial vertices from $G$
    $availAncillaQubits$ ← AllocateQubits($nAncillaQubits$)
    $connectedComps$ ← PartitionAncillae()
    if PartitionAncillae() returned an error then
        return Reqomp-Lazy($G$)
    for $cc \in connectedComps$ do
        if $cc$ is not a path graph then
            return Reqomp-Lazy($G$)
        $stages$ ← getLinearStrat($cc$, $nAncillaQubits$)
        for $anc, fwd \in stages$ do
            if $fwd$ then
                $ancQb$ ← $availAncillaQubits$.pop()
                $i$ ← freshInstanceId$_{\overline{G}}(anc, 0)$
                addVertex$_{\overline{G}}(anc_{0,i}, ancQb)$
                addEdge$_{\overline{G}}$(getLastOnQb$_{\overline{G}}(ancQb) \to anc_{0,i}$)
            $\overline{v}$ ← evolveQubitFully($anc, fwd$)
            if $s$ is last fwd stage in $stages$ acting on $anc$ then
                evolveControlledNonAncillae($\overline{v}$)
            if not $fwd$ then
                $availAncillaQubits$.push($\overline{v}.qubit$)
    evolveNonAncillae()
    assertFullyEvolved()
    return ToCircuit($\overline{G}$)
```

Figure 2: Overview of Reqomp.

3.2 Deriving the Uncomputation Strategy

Based on the circuit graph G in Fig. 2, Reqomp decides on an uncomputation strategy in two steps. First, it extracts relevant ancilla variables dependencies from the circuit graph. Second, it computes an optimal strategy based on those dependencies.

Identifying Ancilla Variable Dependencies. On the circuit graph G from Fig. 2, Reqomp identifies all ancilla variables vertices (highlighted in red) and their dependencies (highlighted in blue), and extracts the following ancilla dependencies (also shown in Fig. 2):

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \end{tikzcd}$$

Here, each vertex corresponds to an ancilla variable and each edge corresponds to a control edge among gate vertices between these respective ancilla variables. The Reqomp algorithm (Fig. 2) performs this step in Lin. 5. This line additionally partitions the ancilla dependency graph. Specifically, it identifies ancilla variables that do not interact with each other (i.e., lie in different connected components of the ancilla dependency graph), and can therefore be computed and uncomputed independently. This allows us to then process the individual partitions in a suitable order, deriving and applying a separate uncomputation strategy for each. We note that our running example in Fig. 2 only yields a single connected component—the general case is discussed in §4.2.

Failure and Fallback. Unfortunately, finding a suitable order among partitions may be impossible, in which case Lin. 6–7 of Reqomp fall back to our alternative algorithm Reqomp-Lazy, discussed in §4.5.

In general, uncomputation according to Def. 2.1 is not always physically possible [1, §6.2]. Because we cannot always achieve uncomputation, our algorithms apply heuristics to succeed as frequently as possible. However, we must accept that they may fail in some cases. When this happens, we can fall back to a different heuristic such as Reqomp-Lazy. If this also fails, it may indicate that no approach can achieve uncomputation, hinting at a possible implementation mistake or misconception by the programmer. If uncomputation is possible but no available approach can synthesize it automatically, a programmer can always uncompute manually instead.

Computing the Uncomputation Strategy. For each partition (i.e., connected component) identified above, Lin. 9 checks that the ancilla variables exhibit a linear dependency, i.e., if each ancilla variable only depends on its direct predecessor. If this was not the case in Fig. 2, our approach would fail, falling back to our alternative algorithm Reqomp-Lazy (Lin. 10). Lin. 11 then derives an optimal uncomputation strategy using dynamic programming [5]. For our example circuit with two ancilla qubits, the following optimal strategy is found:

$$a, b, a^\dagger, c, c^\dagger, a, b^\dagger, a^\dagger. \tag{2}$$

Here we write a to denote "computing ancilla a", and $a^\dagger$ to denote "uncomputing ancilla a".

3.3 Applying the Uncomputation Strategy

Now that the uncomputation strategy for the current connected component has been chosen, Lin. 12–22 of Reqomp apply each of its stages in turn, first computing a.

Applying stage a. We now describe the effect of the first computing stage, which computes a. As this is a compute stage, denoted by fwd = True in the Reqomp algorithm, Lin. 14 picks an available ancilla qubit to use and Lin. 15 gets a fresh instance index for a qubit on variable a and value index 0. Using those, Lin. 16 creates a new node in $\overline{G}$ on this chosen qubit ($a_{0.0}$ in our example) and Lin. 17 links it via target edge to the previous last node on this qubit if it exists (as target edges link nodes on the same qubit). Lin. 18 then evolves the ancilla variable a until its maximum value index in G, which is 1. To do so, it uses the value graph g^{val} as a guide: from a_0, applying CX controlled by s_1 gives a_1. Therefore, Reqomp first adds the gate vertex $a_{1.0}$ with gate CX to $\overline{G}$. Now $a_{1.0}$ must be controlled by some s_1, of which there are currently none in $\overline{G}$. Reqomp hence computes this s_1, again using g^{val} as a reference: applying gate X to some s_0 gives s_1. $s_{1.0}$ with gate X is hence added to $\overline{G}$. Reqomp finally adds the control edge $s_{1.0} \bullet\!\!\rightarrow a_{1.0}$, resulting in Fig. 2c and concluding the stage.

Applying stage b. The next stage computes b. Since fwd is true as before, Lin. 14 picks an ancilla qubit to use and Lin. 15–17 place the init vertex $b_{0.0}$ in it. Then, based on g^{val}, Reqomp determines that it should apply a CCX gate with controls a_1 and s_1 to go from b_0 to b_1. It thus adds $b_{1.0}$ to $\overline{G}$ as well as the control edge $a_{1.0} \bullet\!\!\rightarrow b_{1.0}$[3]. Additionally, Reqomp seemingly should control $b_{1.0}$ by $s_{0.0}$. However, this is impossible, since this would induce a cyclic dependency that prevents transforming $\overline{G}$ to a circuit: $s_{1.0} \bullet\!\!\rightarrow a_{1.0} \bullet\!\!\rightarrow b_{1.0}$ (from Fig. 2c), $b_{1.0} \dashrightarrow s_{1.0}$ (as $b_{1.0}$ must be applied before changing the value of s). Since $s_{0.0}$ is thus not available as a control, Reqomp instead uncomputes s to value index 0 by inserting a fresh vertex $s_{0.1}$ and using it as a control, as shown in Fig. 2e.

Applying stage $a^\dagger$. The next stage, denoted $a^\dagger$, uncomputes a. To this end, just as for computing b, Lin. 18 determines from the value graph g^{val} that applying a CX gate with control s_1 allows moving a from value index 1 to 0. Therefore, it introduces a CX gate vertex $a_{0.1}$. Further, again as above, $s_{1.0}$

[3]Reqomp always adds ancilla controls before non ancilla ones, as we found this to work best in practice.

is not available as a control for $a_{0.1}$, as this would create a cyclic dependency ($s_{0.1} \bullet\!\!\rightarrow b_{1.0} \dashrightarrow a_{0.1} \dashrightarrow s_{0.1}$). Reqomp thus recomputes s_1, creating the vertex $s_{1.1}$. Finally, as the value index of the introduced gate vertex $a_{0.1}$ is 0, a is now fully uncomputed and therefore in its original state $|0\rangle$. Thus, Lin. 22 adds its ancilla qubit back to the pool of available qubits, indicated with a dashed box in Fig. 2e.

Evolving Non-Ancilla Variables. As the stages described above only evolves ancilla variables, we make sure that we periodically evolve non-ancillae as well. To this end, whenever we perform the last step on a specific ancilla, we evolve all non-ancillae controlled by this ancilla to the point at which they need this control (Lin. 19–20). For instance, in Fig. 2, computing c also triggers to computation of r.

Final Steps. After all uncomputation and recomputation stages for all ancilla variables have been applied, Lin. 23 evolves any non-ancillae that may not yet be fully evolved, that is to say whose last value index in $\overline{G}$ is different from what it is in G.

Finally, Lin. 24 checks if all variables are fully evolved, either back to their initial state index 0 (for ancilla variables), or to their final state index in G (for non-ancillae). If not, Reqomp fails.

3.4 Obtaining the Final Circuit

If the above check succeeded, the final step of the algorithm converts the circuit graph $\overline{G}$ to a circuit $\overline{C}$. This step works analogously to Unqomp [1], by creating a qubit per init vertex and applying the gate vertices in any topological order. For our example, this results in the circuit $\overline{C}$ in Fig. 2. Importantly, the resulting circuit uses the same physical ancilla qubit to hold both a and c, saving one qubit at the cost of an extra uncomputation and recomputation of qubit a.

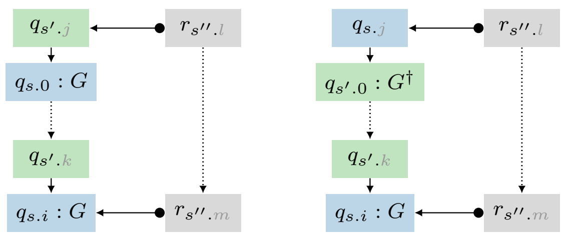
4 Reqomp

In this section, we formalize our main algorithm Reqomp. §4.1–§4.4 go into more details into each of its steps, mirroring §3.1–§3.4. §4.5 presents our fallback algorithm Reqomp-Lazy.

4.1 Building the Circuit Graph

We first present our augmented circuit graphs in more detail. The main novelty compared to the circuit graphs of Unqomp is the introduction and formalization of value indices to track qubit values.

Valid Circuit Graph. As discussed in §3.1 and consistently with Unqomp [1], a circuit graph consists of init vertices, gate vertices, target edges $\rightarrow$, control edges $\bullet\!\!\rightarrow$, and anti-dependency edges $\dashrightarrow$. Anti-dependency edges can be reconstructed from the target and control edges: whenever there are three vertices n, c, d such that $c \rightarrow d$ and $c \bullet\!\!\rightarrow n$, there is an edge $n \dashrightarrow d$ ensuring that n must be computed before d.



(a) Case (fwd) (b) Case (bwd)

Figure 3: Illustration of case (iv) in Def. 4.1. Dotted lines indicate a sequence of gate vertices, grayed out letters are unimportant.

A circuit graph is *valid* iff it corresponds to a valid circuit. Most importantly, all valid circuit graphs must be acyclic; we recall the precise definition of valid circuit graphs from [1] in Def. C.1. The semantics of a valid circuit graph G, denoted $[\![G]\!]$, are defined as the semantics of a circuit it can be converted to, as any of those circuits have equivalent semantics.

In contrast to Unqomp, we augment vertices by naming them using the naming convention $q_{s.i}$, where q is the qubit name, s its value index and i its instance index that we use to ensure each vertex is uniquely named.

Tracking Qubit Values. We use value indices to track when a qubit revisits a previous state. Intuitively, if a qubit is in some basis state at a value index, then if the qubit reaches the same value index at a later point in time, it will again be in this same basis state. We can translate the intuition above more formally as follows: for every pair of vertices $q_{s.i}$ and $q_{s.i'}$, applying all gates G' between these vertices should preserve q, in the following sense:

$$\begin{array}{l}\forall b \in \{0,1\}. \\ \forall \varphi \in \mathcal{H}_{n-1}.\end{array} \exists \psi \in \mathcal{H}_{n-1}. \, |b\rangle_q \otimes \varphi \xmapsto{[\![G']\!]} |b\rangle_q \otimes \psi, \quad (3)$$

where $\mathcal{H}_{n-1}$ denotes the set of quantum states over $n-1$ qubits. As we can write any state as a sum of computational basis states, Eq. (3) allows us to reason about any state.

Well-Valued Circuit Graph. To enforce the property above, we introduce the notion of *well-valued* circuit graphs, which describes a vertex naming scheme sufficient to ensure Eq. (3). Formally, we show in App. C that any well-valued circuit graph ensures Eq. (3) (more precisely, Lem. C.3 implies this).

Definition 4.1 (Well-valued Circuit Graph). *We say a valid circuit graph is well valued iff:*

(i) all vertex names are of the form $q_{s.i}$ where q is the name of the vertex qubit, s and i are natural numbers

(ii) there are no duplicate vertices

(iii) the init vertex on each qubit has name $q_{0.0}$ and for any $q_{s.i}$ in G, $q_{s.0}$ is in G

(iv) any gate vertex $q_{s.i}$ satisfies one of the following (see also Fig. 3):

(fwd) *$valIdx(pred(q_{s.i})) = valIdx(pred(q_{s.0}))$ and $q_{s.i}$ and $q_{s.0}$ have the same gate and same control vertices (up to their instance indices)*

(bwd) *if we denote $s' = valIdx(pred(q_{s.i}))$, we have that (i) $valIdx(pred(q_{s'.0})) = s$, (ii) $q_{s.i}.gate$ is qfree and equal to $q_{s'.0}.gate^\dagger$, and (iii) both $q_{s.i}$ and $q_{s'.0}$ have the same controls (up to instance indices).*

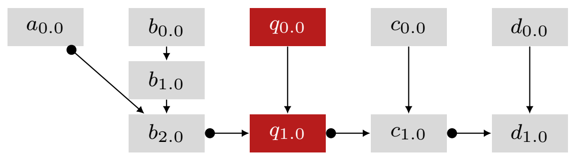
Here, pred(v) is the unique v' such that $v' \to v$ and valIdx(v) is the value index of v. We now give some intuition of the last condition (iv). Case **(fwd)** corresponds to a (forward) computation, for instance $s_{1.0}$ and $s_{1.1}$ in Fig. 2e. Here, case (fwd) ensures that $s_{0.0} \to s_{1.0}$ and $s_{0.1} \to s_{1.1}$ apply the same operation the "same" starting state. This is the case as both $s_{1.0}$ and $s_{1.1}$ have the same gate X, and their predecessors ($s_{0.0}$ and $s_{0.1}$) have the same value index 0. Case **(bwd)** corresponds to a (backward) uncomputation, for instance $a_{1.0}$ and $a_{0.1}$ in Fig. 2e. Here, case (bwd) ensures that that the operations $a_{0.0} \to a_{1.0}$ and $a_{1.0} \to a_{0.1}$ are exact inverses of each other. Specifically, it ensures that (i) $a_{0.1}$ and the predecessor of $a_{1.0}$ (here $a_{0.0}$) have the same value index 0, (ii) the gates of $a_{1.0}$ and $a_{0.1}$ are inverses of each other, and (iii) their controls ($s_{1.0}$ and $s_{1.1}$) have the same qubit and value index.

Building Well Valued Circuit Graphs. Reqomp works only with well-valued circuit graphs: the conversion from circuit to circuit graph ensures that the resulting graph is well valued, and later when building another circuit graph, Reqomp preserves well-valuedness at every step.

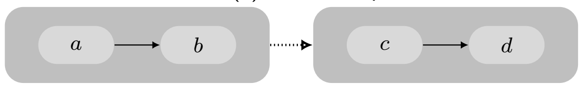
To help convert a circuit into a circuit graph, Reqomp builds in parallel a value graph g^{val}, as shown in Fig. 2. Initially, g^{val} contains one init vertex per qubit but without an instance index, for example s_0 for qubit s. When encountering a new gate, for example the first X gate on qubit s, we pick a fresh value index for this qubit and extend g^{val}. Here, we pick 1 for the value index and add vertex s_1 to g^{val}. As the last vertex on qubit s in G is currently $s_{0.0}$ with value index 0, we also add the edge $s_0 \xrightarrow{X} s_1$. Furthermore, as X is qfree s_1 can be uncomputed, which we materialize with the reverse edge $s_1 \to s_0$, giving:

$$\begin{tikzcd} s_0 \arrow[r, bend left, "X"] & s_1 \arrow[l, bend left, "X"] \end{tikzcd}$$

Later, when we encounter the second X gate on qubit s, we again check the value graph. The last vertex on s in G is now $s_{1.0}$, with value index 1. From s_1, g^{val} shows that an X gate brings back to s_0. Therefore, we do not update g^{val}, and know that the new gate vertex in G must have value index 0: this results in vertex $s_{0.1}$.



(a) Circuit Graph



(b) Ancilla Dependencies

Figure 4: Demonstration of PartitionAncillae (Fig. 13).

4.2 Deriving the Uncomputation Strategy

After converting the circuit to a circuit graph, Reqomp partitions the ancillae of the circuit graph and checks if those partitions have linear dependencies.

Why Partition Ancillae? Reqomp aims at balancing ancilla qubits and gates. For two ancillae that do not interact, such a trade-off is easy: we should always uncompute the first ancilla early, making its qubit available for the latter one. As the ancillae are independent, the latter one does not need the earlier one, so the early uncomputation will not induce extra gates, i.e., no recomputation is necessary. For instance, in Fig. 4a, ancillae $\{a, b\}$ and $\{c, d\}$ are independent. Therefore it is strictly better to uncompute a and b before computing c and d, thereby reusing the physical ancilla qubits initially holding a and b for c and d. This is in contrast to ancillae that are part of the same partition. For instance, in Fig. 1, we saw that for 3 linked ancillae, uncomputing early or late may yield different trade-offs.

Where to Partition? Fig. 4a shows an example graph with four ancillae a, b, c, and d. We can see that ancillae $\{a, b\}$ do not directly interact with ancillae $\{c, d\}$, and should therefore be two different partitions. We now explain how we identify such partitions based on the circuit graph.

To this end, we first collect all ancillae and record their direct dependencies (smaller, light gray vertices and edges $\longrightarrow$ in Fig. 4b). Then, we extract connected components to determine which ancillae must be processed in tandem (larger, dark gray components in Fig. 4b). Finally, we determine the order in which we should process these components, ensuring that if an ancilla c transitively depends on ancilla b, c's component is processed before b's component (captured by edges $\dashrightarrow$ in Fig. 4b). If ordering the connected components is impossible due to cycles, the partitioning fails, and Reqomp falls back to Reqomp-Lazy. For completeness, App. B.1 provides an implementation of PartitionAncillae.

Linear Dependencies. Once partitioned, we process each cluster of ancillae independently, completely computing and uncomputing its ancillae before moving on to the next cluster. The first processing step for each cluster is ensuring its ancilla are linearly dependent (Lin. 9 in Fig. 2). If not, Lin. 10 falls back to Reqomp-Lazy. We say ancillae $a^1, \ldots, a^n$ are linearly dependent if all gates targeting a^i for $i > 1$ are only controlled by a^{i-1} and non-ancillae. In Fig. 4a, ancillae a and b are linearly dependent, and ancillae c and d are linearly dependent. Lin. 9 checks this using the graph exemplified in Fig. 4b: each connected component must be a simple path.

For simple paths, the getLinearStrat function called in Lin. 11 then yields a correct uncomputation strategy. We note that we avoid solving the general problem of finding an uncomputation strategy for any ancilla dependency, as it is P-SPACE complete [6].

4.3 Applying the Uncomputation Strategy

We now go into the most central part of Reqomp, building up $\overline{G}$ step by step, following the uncomputation strategy. All modifications to $\overline{G}$ are done through function evolveVtx, shown in Fig. 5. This function is always called through the helper function evolveQubitFully (called in Lin. 18 of the Reqomp algorithm in Fig. 2).

Evolve Vertex. Function evolveVtx is used to evolve vertices, i.e., to bring qubits from one value index to another. It uses the value graph g^{val} as a guide, and iteratively modifies $\overline{G}$. We demonstrate this in detail on an example. Fig. 6 illustrates a possible call to evolveVtx, where $\overline{G}$ is already partially built, and we would like to revert a single gate on qubit a, whose latest vertex is $a_{1.0}$. Here, Fig. 6a shows the original graph G—for simplicity, we assume that graph $\overline{G}$ before the call to evolveVtx coincides with G. Further, Fig. 6b shows the value graph g^{val} we use as a guide and Fig. 6d shows $\overline{G}$ after the call. The example call is evolveVtx(a, 0, $\emptyset$), which uncomputes a single gate on qubit a: it will bring qubit a from its current value index 1 to 0. The argument $\emptyset$ indicates that no vertices on any other qubit are in the process of being added—this argument is needed to avoid infinite recursion (see also Lin. 29, discussed shortly).

Determine Reference. In Fig. 6, evolveVtx proceeds as follows. Lin. 30 gets the last vertex $\overline{last}$ on qubit x. Lin. 32 then checks that the $nVId$, that is the value index we want to add to the graph, here 1, can be reached in just one gate step. This is the case as a_0 is just one CX gate away from a_1, as evidenced by the edge $a_0 \xrightarrow{CX, b_0} a_1$ in g^{val}.

Because the new value index 0 is smaller than the current one 1, Lin. 33 sets *fwd* to **false**, indicating that we want to uncompute. Therefore, we enter the else branch starting in Lin. 37. Here, Lin. 38 determines that the first time a had the same state index as $\overline{last}$ in G was in vertex *ref*, i.e., in $a_{1.0}$. It then looks up the gate which produced *ref* (a CX gate) and records its inverse as *gt* (Lin. 39).

Uncomputing non-qfree gates could lead to unexpected results as shown by [1]. Therefore, evolveVertex only uncomputes qfree gates, as checked in Lin. 40.

Uncomputing a Gate. To uncompute *ref*, evolveVtx adds vertex $\overline{v}$ to $\overline{G}$, which applies gate *gt* (Lin. 41–42, see $a_{0.1}$ in Fig. 6d). However, to ensure that $\overline{v}$ indeed uncomputes *ref*, we must control it by vertices with the same state index as the controls of *ref*. To this end, Lin. 43 iterates over all controls c of *ref*. It then gets a (potentially fresh) vertex $\overline{c}$ (Lin. 45), which should have the same value index and qubit as c (Lin. 46), and be available as a control for $\overline{v}$ (Lin. 47). Then, it uses $\overline{c}$ as a control for $\overline{v}$ (Lin. 48). In Fig. 6c, we demonstrate why we cannot control $a_{0.1}$ by $b_{0.0}$ directly—this would induce a cycle $a_{0.1}$–$b_{1.0}$–$q_{1.0}$–$a_{0.1}$. Instead, Fig. 6d also uncomputes b, using the resulting $b_{0.1}$ to control $a_{0.1}$.

Here, both the iteration order and the strategy to obtain $\overline{c}$ are parametrized by a strategy *ctrlStrat* (see Lin. 43 and Lin. 45). We note that evolveVtx is quite versatile, and could be instantiated with various strategies. In this work, we show two possible strategies: the strategy for our main algorithm Reqomp, and a strategy Reqomp-Lazy, which closely follows [1].

Computing. Fig. 5 can also compute values by setting $nVId$ to a value greater than that of the last one on *qbit*. This works analogously to uncomputation, by selecting *ref* and *gt* appropriately (Lin. 34–36).

Avoiding Infinite Recursion. The assertion in Lin. 29 ensures that we never call evolveVtx recursively on the same qubit. This avoids infinite recursion where two qubits keep triggering recomputation of the other. To this end, we propagate the set I of qubits currently under construction through getAvailCtrl to potential recursive calls into evolveVtx (see Lin. 54 and Lin. 68).

Control Strategy of Reqomp. Fig. 5 also shows the control strategy employed by Reqomp. In order to get an available control (Lin. 62), it locates the last control with the same value index as c, and returns it if it is available to control $\overline{v}$ (Lin. 63–65). Here available means that adding this control edge $c \bullet\!\!\rightarrow \overline{v}$ and any anti-dependency edge $\dashrightarrow$ it induces does not create a cycle in G.

Otherwise, it evolves the latest vertex of the qubit until it matches the value index of c, and returns it if it is available for $\overline{v}$ (Lin. 67–70).

Using evolveVtx. Function evolveVtx is always called through evolveQubitFully (Lin. 56, called from Fig. 2), which in turn calls evolveVertexUntil (Lin. 50). Function evolveVertexUntil then choses appropriate intermediate steps to bring a qubit q from the value index of its last vertex to the given value

```
func evolveVtx($qbit$: Qubit, $nVId$: int, $I = \emptyset$: Set[Qubit])
        ▷ $nVId$: value index of the vertex to be added to $\overline{G}$
            ▷ $I$: set of in-progress qubits under construction
    assert $q \notin I$
    $\overline{last} \leftarrow$ getVertex$_{\overline{G}}$($qbit$, "last")
    $oVId \leftarrow \overline{last}$.valIdx
    assert there exists $qbit_{oVId} \rightarrow qbit_{nVId} \in g^{val}$
    $fwd \leftarrow (nVId > \overline{last}$.valIdx)
    if $fwd$ then
        $ref \leftarrow$ getVertex$_G$($qbit$, "first", $nVId$)
        $gt \leftarrow ref$.gate
    else
        $ref \leftarrow$ getVertex$_G$($qbit$, "first", $\overline{last}$.valIdx)
        $gt \leftarrow ref$.gate$^\dagger$
        assert $gt$ is qfree
    $\overline{v} \leftarrow$ addVertex$_{\overline{G}}$($gt$, $qbit$, $nVId$)
    addTargetEdge($\overline{last}$, $\overline{v}$)
    for $c \in ctrlStrat$.sort(getControls$_G$($ref$)) do
            ▷ getAvailCtrl may mutate $\overline{G}$ via evolveVertex
        $\overline{c} \leftarrow ctrlStrat$.getAvailCtrl($c$, $\overline{v}$, $I \cup \{qbit\}$)
        assert $\overline{c}$.valIdx $= c$.valIdx and $\overline{c}$.qbit $= c$.qbit
        assert isAvailable$_{\overline{G}}$($\overline{c}$, $\overline{v}$)    ▷ avoid trust in $ctrlStrat$
        addControl$_{\overline{G}}$($\overline{c}$, $\overline{v}$)
    return $\overline{v}$

func evolveVertexUntil($qbit$: Qubit, $valObj$: int, $I$:Set[Qb])
        ▷ Evolve $qbit$ from its last vertex to the given $valObj$
    $p \leftarrow$ getPath(getVertex$_{\overline{G}}$($qbit$, "last").valIdx, valObj)
    for ($valIdx$, $fwd$) in $p$ do
        evolveVtx($qbit$, $valIdx$, $I$)
func evolveQubitFully($qbit$: Qubit, $fwd$: bool)
                ▷ Fully evolve $qbit$ (forward/backward)
    if $fwd$ then
        $valIdx \leftarrow$ getVertex$_G$($qbit$, "maxValIdx").valIdx
    else
        $valIdx \leftarrow 0$
    return evolveVertexUntil($qbit$, $valIdx$, $\emptyset$)
func getAvailCtrl($c$: Vertex, $\overline{v}$: Vertex, $I$:Set[Qb])
    $\overline{c} \leftarrow$ getVertex$_{\overline{G}}$($c$.qbit, "last", $c$.valIdx)
    if isAvailable$_{\overline{G}}$($\overline{c}$, $\overline{v}$) then
        return $\overline{c}$
    else
        $\overline{c}' \leftarrow$ getVertex$_{\overline{G}}$($c$.qbit, "last")
        $\overline{c} \leftarrow$ evolveVertexUntil($\overline{c}'$, $c$.valIdx, $I$)
        assert isAvailable$_{\overline{G}}$($\overline{c}$, $\overline{v}$) ▷ always holds if $\overline{v}$ is fresh
        return $\overline{c}$
```

Figure 5: Basic building block for evolving values and helper functions.

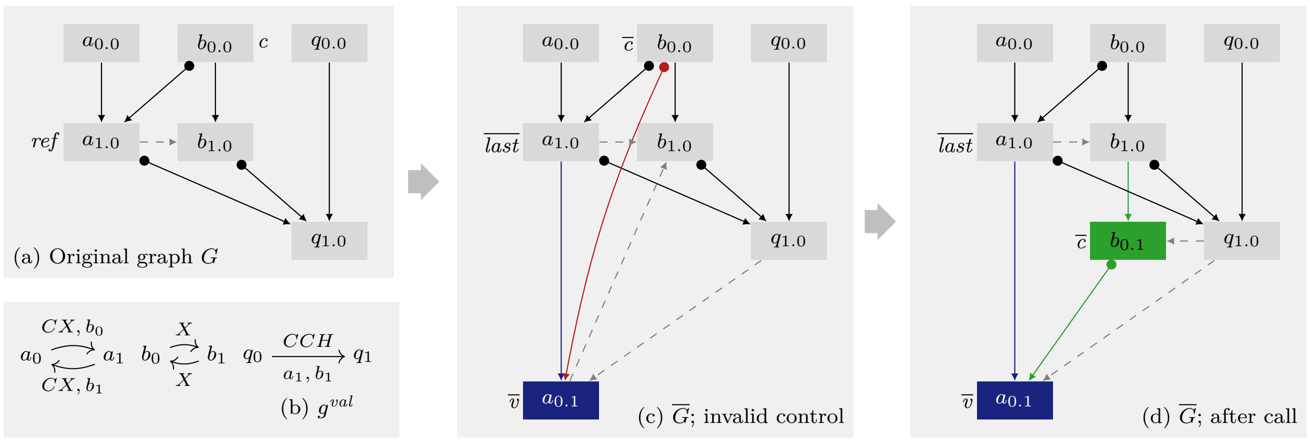


Figure 6: Demonstration of evolveVtx from Fig. 5.

```
func OptimizeRCCX()
    for $\overline{v} \in \overline{G}$ do
        if $\overline{v}$ is later uncomputed then
            $\overline{v}$.gate $\leftarrow RCCX$
        if $\overline{v}$ is an uncomputation step then
            $\overline{v}$.gate $\leftarrow RCCX^\dagger$
```

Figure 7: General-purpose optimization replacing CCX gates.

index $valObj$ by (i) finding the shortest path[4] in the value graph between the two value indices and (ii) calling evolveVtx for each step on this path. We formally describe getPath in App. B.2 (Fig. 14).

[4]As detailed in App. B.2 (Fig. 14), we may force extra steps in the path to ensure some values are computed at least once.

4.4 Obtaining the Final Circuit

We already discussed how to convert the final circuit graph $\overline{G}$ to a circuit in §3.4. Now, we describe a generic post-processing optimization we additionally perform during this step. Fig. 7 shows this optimization, which was previously discussed in [1]. Specifically, it replaces CCX gates which are later uncomputed by RCCX gates. While RCCX gates introduce an additional phase change, replacing pairs of CCX gates ensures that this phase change is also reverted.

As RCCX gates can be implemented more efficiently than CCX gates (the latter require more T gates), this can lead to a substantial efficiency improvement. This is particularly appealing in our setting, were we encounter many CCX gates, and most of them are uncomputed.

We note that Unqomp could only apply this opti-

```
77: func Reqomp-Lazy()
78:     $\overline{G} \leftarrow$ DeepCopy($G$)
79:     $U \leftarrow \left\{q_{s.i} \in \overline{G} \mid q \text{ ancilla in } \overline{G}.qbs, i = 0 \text{ and } s > 0\right\}$
80:     for $v \in$ reverse(topologicalSort($U$)) do
81:         $l \leftarrow \text{last}_{\overline{G}}(v.\text{qubit})$
82:         $p \leftarrow \text{predecessor}_{g^{val}}(v)$
83:         assert $l$.valIdx == $v$.valIdx
84:         assert $p$.valIdx == $v$.valIdx $- 1$
85:         evolveVertexUntil($l$.qbit, $p$.valIdx)
86:     assertFullyEvolved()
87:     reuseAncillaRegisters()
88: func getAvailableControl($c$: Vertex, $\overline{v}$: Vertex)
89:     $l_{\overline{c}} \leftarrow \text{getVertex}_{\overline{G}}(c.\text{qbit}, \text{"all"}, c.\text{valIdx})$
90:                                          ▷ Sorted according to target edges
91:     for $\overline{c} \in l_{\overline{c}}$ do
92:         if $\text{isAvailable}_{\overline{G}}(\overline{c}, \overline{v})$ then
93:             return $\overline{c}$
94:     $\overline{c}' \leftarrow \text{getVertex}_{\overline{G}}(c.\text{qbit}, \text{"last"})$
95:     $\overline{c} \leftarrow$ evolveVertexUntil($\overline{c}'$, $c$.valIdx)
96:     return $\overline{c}$
```

Figure 8: Fallback algorithm closely following Unqomp [1].

mization to gates it had itself uncomputed, whereas Reqomp can also identify uncomputation that is already in place in the original circuit, by leveraging value indices.

4.5 Reqomp-Lazy

While most quantum circuits we have investigated have linear ancilla dependencies, some are more complex. For those, Reqomp falls back to Reqomp-Lazy, as shown in Lin. 7 and Lin. 10 of Fig. 2. This fallback is inspired by Unqomp [1], but leverages the augmented circuit graphs and evolveVtx. In particular, Reqomp can uncompute and recompute controls for a vertex when they are not directly available. In contrast Unqomp would have returned an error anytime this happens. We provide an example in §6.

Overview. Fig. 8 shows our implementation of Reqomp-Lazy. Lin. 78 initializes $\overline{G}$ with a copy of G, to be extended by adding vertices that perform uncomputation. Then, Lin. 79 defines U as the set of all vertices to be uncomputed: it contains the first instance ($i = 0$) of each value index on each ancilla qubit. Then, Lin. 80–85 step through U in reverse topological order and revert all operations on ancillae by calling evolveVertexUntil (Lin. 85). Note that Lin. 83–84 ensure that the current last node on the qubit has the same value index as v and that its predecessor has value index exactly $v.\text{valIdx} - 1$, and therefore that the call to evolveVtxUntil will result in exactly one call to evolveVtx, uncomputing a single step on the given ancilla variable. Then, analogously to Reqomp, Lin. 85 evolves all non-ancillae (which may have been reverted by calls to evolveVtxUntil) and Lin. 86 asserts all qubits are fully evolved. Finally, as specified in the original version of Unqomp [1, §5.4], Lin. 87 allocates ancillae to the same physical qubits if their lifetimes do not overlap.

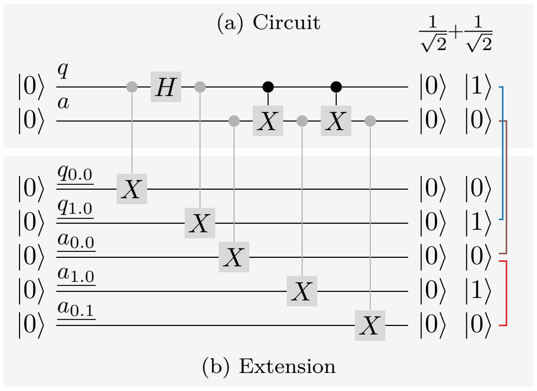


Figure 9: Intuition on the correctness of Reqomp.

Custom Control Strategy. While this implementation reuses evolveVtx, it leverages a custom control strategy, shown in Lin. 88. The goal of this strategy is to use controls that are as early (in terms of target edges) as possible, therefore keeping later controls available for later uncomputations.

Specifically, the strategy to get a control $\overline{c}$ available for a target $\overline{v}$ is stepping through all possible vertices with the correct state index, and picking the first that is available for $\overline{v}$. Only if none is available, analogously to Reqomp, Lin. 94–95 evolve the last vertex on the qubit of $\overline{c}$ until it has the correct state index.

5 Correctness

We prove in App. C that Reqomp synthesizes correct uncomputation. In this section, we provide an intuition of this proof.

Value Index Assertions. The correctness of Reqomp relies on value indices. At the end of the algorithm (Lin. 24), we assert that the last vertex on all ancilla qubits has value index 0, and that for any non-ancilla qubit, the value indices of the last vertex are the same for the original graph and the synthesized graph. Intuitively, this ensures that ancillae are reset to $|0\rangle$, while other qubits are preserved.

Correctness hence relies on the precise formal interpretation of value indices, Intuitively, we claim that two vertices on the same qubit with the same value index *hold the same value.*

Extended Circuits. To formally define this notion, we introduce the notion of an extended circuit. We conceptually *extend* a given circuit to allow us to compare the value of all vertices occurring in the circuit.

Fig. 9 exemplifies this by extending the example circuit in Fig. 9a, which applies an H gate and two controlled X gates to qubits q and a. Overall, the circuit in Fig. 9a yields state

$$\tfrac{1}{\sqrt{2}} |0\rangle_q |0\rangle_a + \tfrac{1}{\sqrt{2}} |1\rangle_q |0\rangle_a ,$$

which we write in a column-by-column format in Fig. 9a (right).

Fig. 9b shows our extension of Fig. 9a, copying[5] the value of each vertex from the corresponding circuit graph to a fresh qubit. The name of these *copy qubits* is the same as their corresponding vertex but underlined, e.g., $\underline{q_{0.0}}$ holds the initial state of q, corresponding to vertex $q_{0.0}$.

Value Index. Intuitively, copy qubits with the same value index and qubit hold the same value. More precisely, if we write the state produced by the extended circuit as a sum of computational basis states, in each summand (with a non-null coefficient), copy qubits with the same value index and qubit hold the same value. For example, in every summand (i.e., column) of the final state in Fig. 9b, $\underline{a_{0.0}}$ and $\underline{a_{0.1}}$ hold value $|0\rangle$ (see red bracket in Fig. 9).

Similarly, each qubit holds the same value as its last copy qubit. For example, in every summand (i.e., column) of the final state in Fig. 9b, q and $\underline{q_{1.0}}$ both hold either value $|0\rangle$ or $|1\rangle$ (see blue bracket in Fig. 9).

In Lem. C.3 (App. C) we formally prove that these two facts hold for any well-valued circuit graph, as defined in Def. 4.1.

We further show in App. C that any circuit graph built with evolveVtx (Fig. 5) is well-valued.

Final Values in the Extended Graph. Using the final assertion in Reqomp (Lin. 24), we have that at the end of the circuit, the last vertex on all ancilla qubits has value index 0. Hence those qubits hold the same value as the initial value of that qubit, i.e., $|0\rangle$. More precisely, consider a circuit graph G with ancilla qubits A and non ancilla qubits Q, and denote $\overline{G}$ the circuit graph after uncomputation. We then have that any summand in the final state after applying the extended version of $\overline{G}$ is of the form $|0...0\rangle_A \otimes |i\rangle_Q \otimes |...\rangle_{\overline{V}}$, where we use $\overline{V}$ to denote all the copy qubits in $E(G)$.

The assertion in Lin. 24 further checks that the value indices of non-ancilla qubits match their respective last vertices in G. As we show more formally in App. C, this means that if the effect of the extended version $E(G)$ of G on some initial state can be written as

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![E(G)]\!]} \sum_{\substack{j\in\{0,1\}^{|A|}\\ k\in\{0,1\}^{|Q|}}} \gamma_{jk} |j\rangle_A \otimes |k\rangle_Q \otimes |...\rangle_{\underline{V}},$$

then the effect of the extended version $E(\overline{G})$ of $\overline{G}$ on the same state can be written as:

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![E(\overline{G})]\!]} \sum_{\substack{j\in\{0,1\}^{|A|}\\ k\in\{0,1\}^{|Q|}}} \gamma_{jk} |0\cdots0\rangle_A \otimes |k\rangle_Q \otimes |...\rangle_{\underline{V'}},$$

where we denote $\overline{V'}$ the set of copy qubits in $E(\overline{G})$.

[5]Note that copying using a controlled X gate does not violate the no-cloning theorem.

Circuit Graph Semantics. Importantly, the semantics of the unextended circuit follows straightforwardly from the semantics of the extended circuit. In Fig. 9, simply ignoring the rows from Fig. 9b yields the correct final state. If we similarly ignore the values of the copy qubits $\underline{V}$ and $\underline{V'}$ in the two equations above, we recover the correct uncomputation theorem, for circuits C and $\overline{C}$:

Definition 2.1 (Correct Uncomputation, [1, 3])**.** *$\overline{C}$ correctly uncomputes the ancillae A in C if whenever*

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![C]\!]} \sum_{j\in\{0,1\}^{|A|}} \gamma_j |j\rangle_A \otimes \phi_j, \textit{ then}$$

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![\overline{C}]\!]} \sum_{j\in\{0,1\}^{|A|}} \gamma_j |0\cdots0\rangle_A \otimes \phi_j.$$

Multiple Graphs. Note that here we assumed that both G and $\overline{G}$ have the same effect, as they apply the same gates for the same value indices. Proving this formally requires extra work, done in Lem. C.4 (App. C).

6 Evaluation

We have evaluated Reqomp on an existing benchmark to answer the following research questions:

Q1 Circuit Efficiency: Can Reqomp create efficient circuits in terms of number of qubits and gates, while allowing to trade one for the other?

Q2 Usability: Is Reqomp fast and directly applicable to a wide range of circuits?

Implementation. We implemented Reqomp as a language extension of Qiskit, using Qiskit's built-in `AncillaRegister` type to mark ancilla variables in the circuit. As Qiskit, our extension is implemented in Python.

6.1 Benchmarks

To evaluate Reqomp, we used the benchmark from Unqomp [1]. The first column in Table 1 summarizes the circuits in our benchmark, separated into "small" and "big" circuits. While the "small" circuits were taken directly from Unqomp, we have generated the "big" circuits by re-parametrizing the original circuits to yield bigger circuits. This allows us to demonstrate the Reqomp also performs well on larger circuits.

For completeness, we provide the exact parameters for each circuit in App. D, including the resulting circuit sizes.

Circuits. To provide an intuition on our benchmark, we explain selected circuits (see [1, §7.1] for details).

IntegerComparator takes a constant parameter n and multiple input qubits encoding a value v, and flips its output qubit if and only if $v \geq n$. MCX flips its

Table 1: Reqomp results when targeting a specific ancilla qubit reduction compared to Unqomp (e.g., -66.7 indicates a reduction by 66.7%). Columns **Max** and **Min** report the results for the most aggressive settings, respectively optimizing only for number of qubits and optimizing only for number of gates. Columns **-75%**, **-50%**, and **-25%** report the gate counts when achieving the respective ancilla qubit reductions. Entries "x" indicate that a given ancilla qubit reduction was not achieved.

	Ancilla Reduction						
	Max		**-75%**	**-50%**	**-25%**	**Min**	
Algorithm	anc	gates	gates	gates	gates	anc	gates
Small							
Adder	-66.7	70.5	x	39.2	15.7	-8.3	0.0
Deutsch-Jozsa	-50.0	40.9	x	40.9	20.4	0.0	0.0
Grover	-33.3	12.9	x	x	12.9	0.0	0.0
IntegerComparator	-63.6	47.1	x	36.1	11.6	0.0	-5.2
MCRY	-63.6	80.0	x	53.3	26.7	0.0	0.0
MCX	-60.0	55.2	x	46.0	27.6	0.0	0.0
Multiplier	0.0	-5.1	x	x	x	0.0	-5.1
PiecewiseLinearR	-50.0	7.5	x	7.5	2.5	0.0	-3.3
PolynomialPauliR	-33.3	9.5	x	x	9.5	0.0	0.0
WeightedAdder	0.0	-9.7	x	x	x	0.0	-9.7
Big							
Adder	-93.0	314.9	64.7	42.9	21.0	-1.0	0.0
Deutsch-Jozsa	-92.9	327.1	69.0	45.7	23.3	0.0	0.0
Grover	-50.0	37.8	x	37.8	16.2	0.0	0.0
IntegerComparator	-92.9	310.8	60.2	37.6	14.7	0.0	-6.7
MCRY	-96.0	515.4	75.3	50.2	25.1	0.0	0.0
MCX	-96.0	509.9	74.9	49.8	25.1	0.0	0.0
Multiplier	0.0	-5.4	x	x	x	0.0	-5.4
PiecewiseLinearR	-85.0	47.1	17.6	10.2	2.8	0.0	-3.8
PolynomialPauliR	-50.0	14.4	x	14.4	1.5	0.0	0.0
WeightedAdder	0.0	-8.0	x	x	x	0.0	-8.0

output qubit if and only if all its input qubits are one. MCRY applies a rotation to its output qubit if and only if all its control qubits are one. PiecewiseLinearR applies a rotation $f(x)$ to its output qubit, where x is the value on its input qubits and f is piecewise linear. PolynomialPauliR works analogously, but for polynomial f. WeightedAdder takes as parameters a list of weights $\lambda_0, ...\lambda_n$ and outputs $\sum \lambda_i q_i$ where the q_i are the input values.

6.2 Q1: Circuit Efficiency

We now discuss the efficiency of circuits produced by Reqomp in terms of qubits and gates. To put our results into perspective, we compare them to Unqomp [1].

Approach. For each circuit, we ran Reqomp targeting all possible number of ancilla qubits `nAncillaQubits` (see Fig. 2). We then recorded, for all calls that terminated without error, the number of qubits and gates of the resulting circuit (with uncomputation).

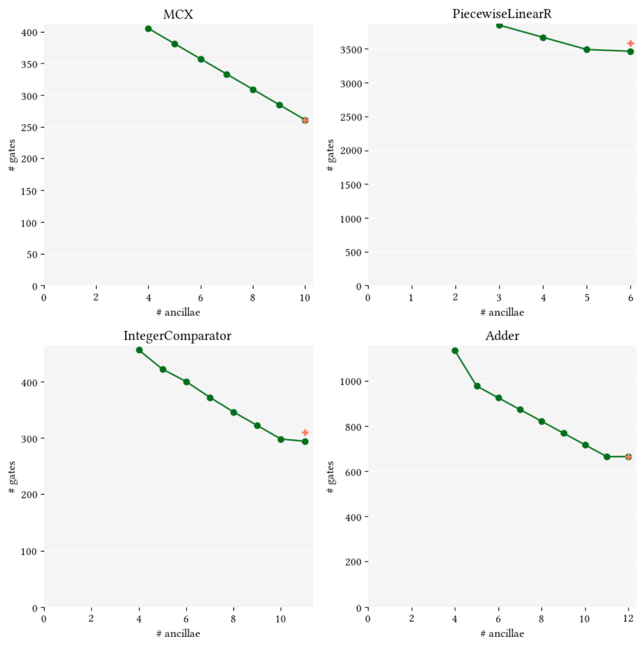
We note that Reqomp had to fall back to Reqomp-Lazy for circuits Multiplier and WeightedAdder, as the ancilla dependencies of these circuits are not linear. While Reqomp-Lazy succeeds on these circuits and even outperforms Unqomp, it cannot offer multiple space-time trade-offs.

Results. Table 1 summarizes our results. For all examples, using the maximum number of ancilla qubits (column **Min** as this is the *minimal reduction*) yields better results than Unqomp for 10 circuits, and equivalent results for the remaining 10 circuits. For example, Reqomp saves 5.2% of gates on circuit IntegerComparator, without requiring additional qubits. This is because Reqomp can identify uncomputation *already present in the original circuit*, allowing it to avoid unnecessary operates when uncomputing or recomputing an ancilla or even a control. Analogous effects occur for PiecewiseLinearR, WeightedAdder, and Multiplier, where the last two are handled by Reqomp-Lazy.

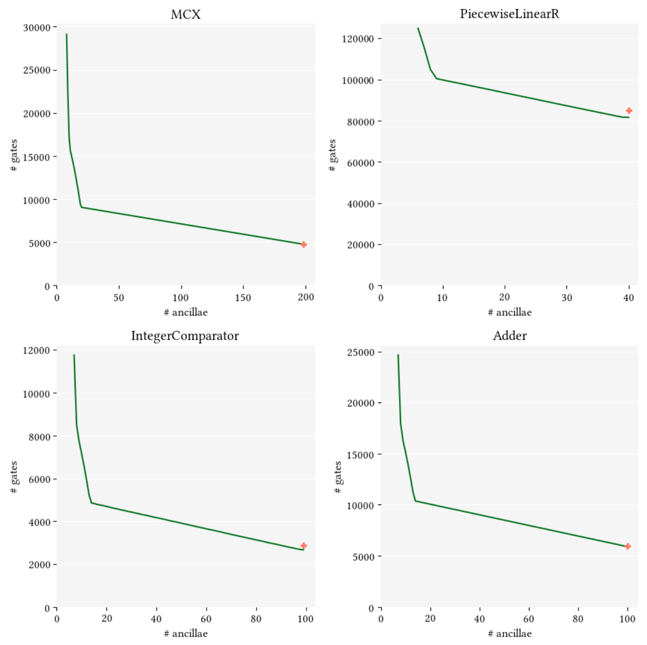
More importantly, Table 1 demonstrates that Reqomp can significantly reduce the number of ancilla qubits compared to Unqomp: by up to 96% for two examples, and by at least 25% for 16 out of 20 circuits. Importantly, this reduction comes at only a moderate cost in gate count, of below 28% for qubit reductions of 25%. As most quantum computers are more limited in terms of qubits than gates, these trade-offs are highly favorable. Further, for some examples the reduction in qubits comes at almost no cost in gates: for instance for PiecewiseLinearR, reducing the number of ancilla qubits by 75% only increases the number of gates by 17.6%.

Trade-Offs. To further demonstrate the gate count cost incurred by these reductions, Figs. 10a–10b show a more fine-grained visualization of the trade-offs between ancilla qubits reduction and gate count increase.

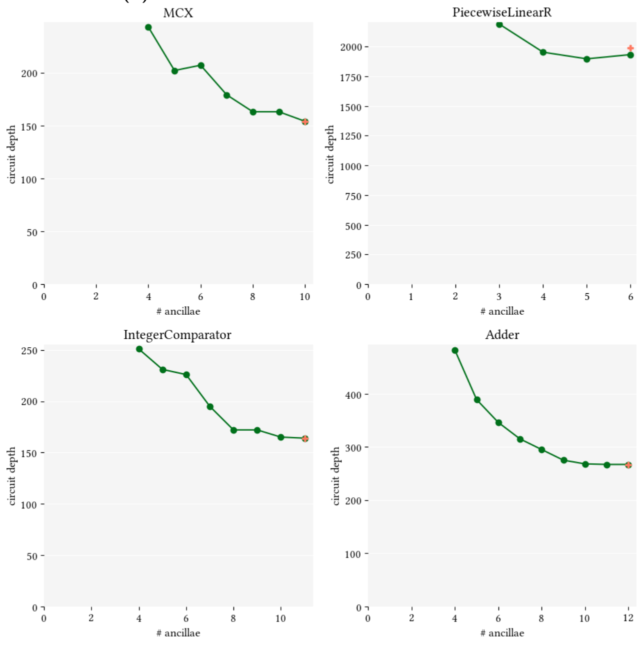
Overall, we immediately observe that on all circuits, reducing the number of available ancilla qubits can only increase (and never decrease) the gate count of the resulting circuit. However, the rate of this increase



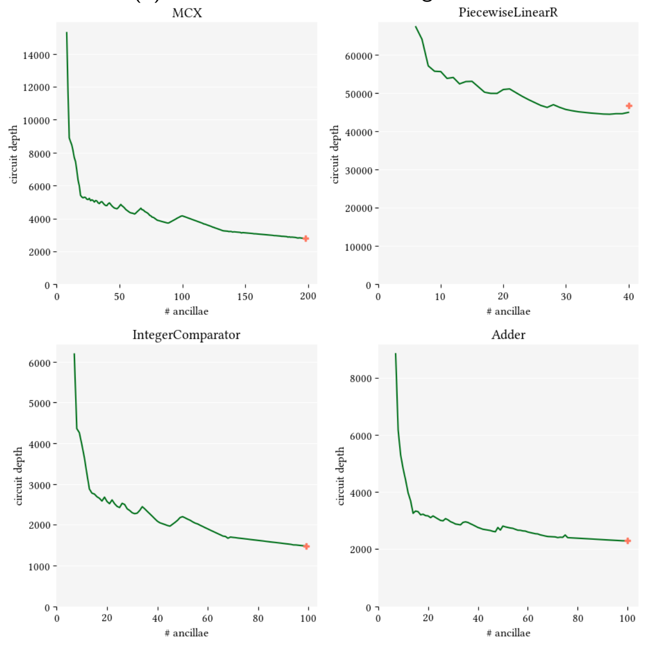
(a) Gate counts for selected small circuits.



(b) Gate counts for selected big circuits.



(c) Circuit depth for selected small circuits.



(d) Circuit depths for selected big circuits.

Figure 10: Gate counts (a–b) and circuit depths (c–d) for given numbers of ancillae, using Reqomp (▬) and Unqomp (+).

varies among the different circuits, as discussed next.

For some benchmarks such as PiecewiseLinearR (Figs. 10a–10b) and PolynomialPauliR (Table 1), Reqomp can drastically reduce the number of ancillae at almost no cost in terms of gates.

For other benchmarks such as MCX (Figs. 10a–10b) and MCRY (Table 1), Reqomp can still reduce the number of ancillae substantially, but at a significant cost in terms of gates. In such cases, the appropriate ancilla reduction depends on the available hardware—a programmer with access to Reqomp can then systematically select the right trade-off.

Other circuits fall somewhere between these two categories (Figs. 10a–10b and Table 1): Reqomp can reduce the number of ancilla qubits, at a non-negligible cost in terms of gates.

Very Small Number of Qubits. Fig. 10 further demonstrates that enforcing a very small number of ancillae typically increases the number of applied gates significantly. For instance, MCX with 200 controls can be implemented with only 8 ancilla qubits, but this requires a staggering 21 831 gates, compared to only 3579 when 200 ancillae are used.

Overall, we conclude that enforcing very small number of ancilla qubits is typically not a good approach.

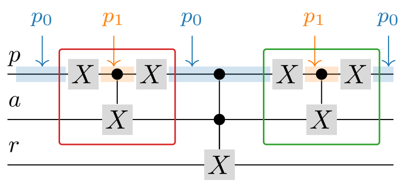


Figure 11: Gate requiring definition in Unqomp.

Depth. For completeness, Figs. 10c–10d shows the trade-off between ancilla qubits reduction and circuit depth. As we do not optimize for circuit depth, reducing the number of ancillae sometimes yields *shorter* circuits. Still, overall, circuit depth behaves analogously to gate count, generally increasing for reduced ancilla qubits counts, at different rates depending on the circuit.

Interestingly, in some cases, we can reduce the ancilla count at almost no cost in circuit depth, even though there is a cost in gate count. For example, reducing ancillae from 99 to 25 on Adder only increases depth by 31%, even though it increases the gate count by 64%.

6.3 Q2: Reqomp Usability

We also investigated the usability of Reqomp, showing that it is both fast and directly applicable to many quantum circuits.

Reqomp Runtime. Our evaluation indicated that Reqomp is fast: it synthesized uncomputation for all circuits in Table 1 within five seconds.

Furthermore, running Reqomp typically takes as much time as decomposing the resulting circuit to basic gates using Qiskit's built-in `decompose()` function. We hence believe that Reqomp can be integrated into the programmer's workflow without incurring a significant slowdown.

Applicability. Recall that even for a circuit where uncomputation is possible in principle, Reqomp may raise an error. We therefore investigated how frequently Reqomp succeeds in practice, comparing it to other tools:

	Qfree only	Unqomp	Reqomp
% examples covered	$\leq$ 50%	60%	100%

We find that Reqomp (with the fallback strategy Reqomp-lazy) finds a circuit with uncomputation for all input circuits. In contrast, Unqomp can only cover 60% of those circuits directly. We will explain shortly how we tweaked Unqomp to also cover the remaining 40%. Furthermore, only 50% of the circuits in our benchmark are purely classical, hence any tool that exclusively supports qfree gates can at most be used on 50% of the examples.

Unqomp Limitations. Unqomp can only handle 60% of the circuits in our evaluation directly, because it cannot accurately handle uncomputation occurring in the input circuit. Fig. 11 illustrates this on a circuit applying a $\overline{C}X$ gate (see red box on the left), where the bar over C indicates that the control is inverted. To invert the controls, the circuit applies an X gate to invert the control, and another X gate to restore the value of the control. To uncompute a it is hence necessary to track that after two X gates, p is back to its original value shown as p_0 in Fig. 11, and therefore applying a third X gate will bring its value to p_1 again, allowing to uncompute a. Value indices allow Reqomp to precisely track those value changes, and insert the uncomputation gates (in the green box). In contrast, Unqomp fundamentally cannot allow for recomputation, as its correctness relies on each operation being computed and uncomputed exactly once. It further does not recognize uncomputation or recomputation already present in the original circuit. Therefore in Fig. 11, Unqomp cannot recognize that the second X gate recovers the original value of p. Even if it did, it could not recompute p_1 to uncompute a. In our evaluation (Table 1), we bypassed this type of issue by defining the red block as a custom gate controlled by p. Unqomp then never decomposes this new gate, assumes it keeps p constant, and places it to uncompute a. Unfortunately, this approach makes Unqomp harder to use, and in some cases makes the resulting circuit less efficient.

7 Related Work

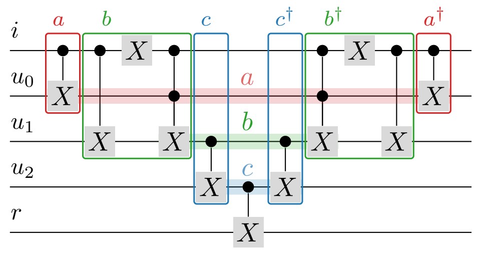
We now discuss works related to Reqomp.

7.1 SQUARE

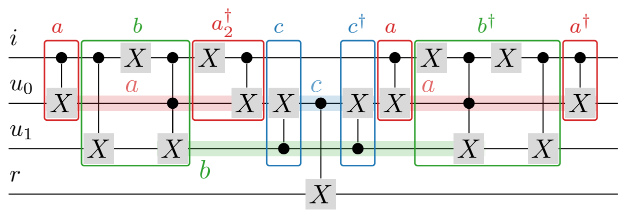
Even though it cannot synthesize uncomputation code, SQUARE [2] looks very closely related to Reqomp at first sight. Specifically, it presents "a compiler that automatically [places uncomputation] in order to manage the trade-offs in qubit savings and gate costs" [2, §1]. Unfortunately, SQUARE suffers from various shortcomings that prevent a meaningful comparison to Reqomp.

Square Problem Statement. SQUARE takes as input a program defining a qfree circuit (non qfree gates are not supported). In this program, each function consists of the three blocks *Compute* (indicating forward computation), *Store* (indicating computation of outputs), and *Uncompute* (indicating uncomputation). SQUARE then compiles this program to a circuit by arranging these blocks, possibly repeating blocks when recomputation is helpful.

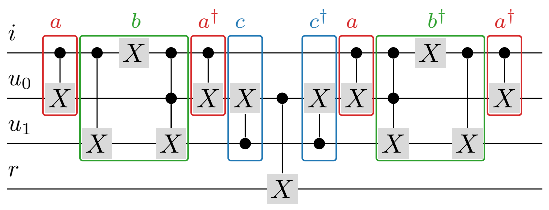
SQUARE defines three different strategies for interleaving the blocks. Lazy (uncompute as late as possible), Eager (uncompute as early as possible), and finally SQUARE itself, using a custom heuristic. For the example *CCCH* in Fig. 1, Lazy would correspond to the 3-qubit strategy shown in Fig. 1b and Eager to



(a) Uncomputation using 3 qubits.



(b) Correct uncomputation using 2 qubits.



(c) Incorrect uncomputation using 2 qubits.

Figure 12: Circuit with varying Uncomputation.

the 2-qubit strategy shown in Fig. 1c. We now present the main shortcomings of SQUARE.

Constant Compute/Uncompute Blocks. As mentioned in §1, the gates needed to uncompute an ancilla variable may depend on where this uncomputation occurs in the circuit. It is hence impossible to define fixed *Compute* and *Uncompute* blocks to be applied anywhere. For instance, consider the circuit in Fig. 12a. It uses three ancilla variables a, b, and c to compute the output variable r from the input i. Fig. 12a highlights the Compute and Uncompute blocks SQUARE would consider, namely blocks a, b and, c for computation and blocks $a^\dagger$, $b^\dagger$, and $c^\dagger$ for uncomputation. Note how the value of qubit i is changed by block b, and restored later by block $b^\dagger$, ensuring that qubit i has the same value for the CX gate in block $a^\dagger$ as it had in block a. Now, if we want to save one ancilla qubit by uncomputing ancilla variable a early, we get the circuit shown in Fig. 12b. Here, when uncomputing a for the first time, the value of i has been changed in block b and is not yet restored. To correctly uncompute a in the block $a_2^\dagger$ (different from the block $a^\dagger$), it is hence necessary to restore i using an X gate before using it as a control to uncompute a. Similarly, block $b^\dagger$ must change the value of i again.

Not accounting for the above, SQUARE assumes that no matter its placement, uncomputation code can be kept unchanged. In particular, its eager strategy would use the Compute and Uncompute blocks from Fig. 12a, yielding Fig. 12c. This is clearly incorrect as this circuit has different semantics than the one in Fig. 12a. For example, for input $|0\rangle_i |0\rangle_t$, Fig. 12a produces state $|0\rangle_i |0\rangle_t$ while Fig. 12c produces state $|0\rangle_i |1\rangle_t$ (assuming ancillae are in state $|0\rangle$).

We note that SQUARE does not exclude such patterns—in fact its `little-belle` benchmark contains an analogous pattern. [6]

Incomplete Uncomputation. Besides only supporting fixed uncomputation code, SQUARE may also skip uncomputation of some ancilla variables. For some examples evaluated in [2], the implementation of the lazy strategy does not insert any uncomputation code at all, leaving all ancilla variables dirty, while the eager strategy uncomputes all of them. Specifically, we believe that the reported differences between strategies in the SQUARE publication ([2, Tab. III]) on the reversible arithmetic benchmarks[7] RD53, 6SYM, 2OF5, and ADDER4 are only due to leaving some ancillae dirty—as these benchmarks do not contain nested uncomputation, the order of uncomputation should not make a difference.

Additional Parameters. Finally, the implementation of SQUARE is inconsistent with the system described in [2]. Specifically, using the interface to specify *Compute* blocks requires providing 7 parameters, and some benchmarks evaluated in [2] also contain *Unrecompute* and *Recompute* blocks not mentioned in the publication [2]. Even though the authors provided us with brief explanations of these parameters on request, we could not confidently derive correct parameters for new benchmarks.

7.2 Purely Classical Circuits

Most works synthesizing uncomputation cannot handle non-qfree gates [7, 8, 9, 10, 4]. [8] It has already been established [1] that using such works on quantum circuits by separating out the qfree subparts typically yields inefficient circuits, and is sometimes even impossible.

In the following, we discuss works which only support qfree gates, and define a custom strategy allowing

[6]Benchmark `little-belle` is available at `https://github.com/epiqc/Benchmarks/blob/master/bench/square-cirq/synthetic/little_belle.py`. We note that different uncomputation strategies do not yield different results on it, as it does not contain gates modifying the output and hence is semantically equivalent to the identity.

[7]Available at `https://github.com/epiqc/Benchmarks/tree/master/bench/square-cirq/application`.

[8][4] can verify uncomputation for non qfree circuits, but can synthesize it only for qfree ones.

to trade qubits for gates. We have already discussed SQUARE in §7.1.

Boolean Functions. Revs [7, 8] translates irreversible classical functions to reversible circuits. It focuses on optimization possibilities during the translation from boolean functions to reversible circuits, but also offers an uncomputation strategy, however without the option of trading qubits for gates.

Similarly, [11] also translates boolean specifications to reversible circuit. While it introduces another uncomputation heuristic, it also cannot trade qubits for gates.

We expect that both of those strategies could be incorporated into Reqomp, possibly yielding more efficient circuits.

Pebble Games. Multiple works present uncomputation strategies for classical reversible computation, which can be reduced to solving *pebble games* [12]. Importantly, while pebble games operate on dependency graphs on values, Reqomp operates on quantum circuits. In particular, pebble games assume all values can be uncomputed, which is incorrect for non-qfree gates. Further, a direct translation of circuits to such graphs would ignore repeated values, leading to issues analogous to Fig. 11. In contrast, conflating repeated values can lead to cyclic dependencies, which are not supported by pebble games.

Knill [5] provides an optimal yet efficient solution for linear dependencies. As most circuits we encounter in practice exhibit linear dependencies, Reqomp uses the same uncomputation strategy. Meuli et al. [13] suggest using a SAT-solver to handle arbitrary dependencies, which may be a possible extension of Reqomp.

7.3 Non-Qfree Circuits

We now discuss works offering uncomputation for non-qfree circuits.

Language Level. Quantum languages like Quipper [9] and Q# [14] offer convenience functions to automatically insert uncomputation. However, these functions are often tedious to use, and may insert incorrect uncomputation (see [1, §8] for details).

Silq [3] uses a type system to detect which variable can be safely uncomputed, but does not synthesize this uncomputation. Overall, none of those works can constrain the number of ancillae used.

Circuit Level. We are aware of only two works supporting uncomputation for non-qfree circuits. ReQWire [4] can only verify user supplied uncomputation (in the case of non-qfree circuits). Unqomp [1] allows to synthesize uncomputation for quantum circuits, but cannot trade qubits for gates. Further, as discussed in §6, it uses a notion of circuit graphs that does not allow to track qubit values and therefore is unable to uncompute directly many examples that Reqomp can handle.

8 Conclusion

We introduced Reqomp, a method to synthesize and place efficient uncomputation for quantum circuits with space constraints. Reqomp is proven correct and can easily be integrated into circuit based quantum languages such as Qiskit. We demonstrate in our evaluation that Reqomp is widely applicable and yields wide ranges of trade-offs in space and time, for instance allowing to generate tightly space constrained circuits by using only a few ancilla qubits.

References

[1] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. "Unqomp: synthesizing uncomputation in Quantum circuits". In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Pages 222–236. Association for Computing Machinery, New York, NY, USA (2021).

[2] Yongshan Ding, Xin-Chuan Wu, Adam Holmes, Ash Wiseth, Diana Franklin, Margaret Martonosi, and Frederic T. Chong. "Square: Strategic quantum ancilla reuse for modular quantum programs via cost-effective uncomputation". In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). Pages 570–583. IEEE (2020).

[3] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. "Silq: A High-level Quantum Language with Safe Uncomputation and Intuitive Semantics". In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. Pages 286–300. PLDI 2020New York, NY, USA (2020). Association for Computing Machinery.

[4] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. "ReQWIRE: Reasoning about Reversible Quantum Circuits". Electronic Proceedings in Theoretical Computer Science **287**, 299–312 (2019).

[5] Emanuel Knill. "An analysis of Bennett's pebble game" (1995). Number: arXiv:math/9508218 arXiv:math/9508218.

[6] Siu Man Chan, Massimo Lauria, Jakob Nordstrom, and Marc Vinyals. "Hardness of approximation in pspace and separation results for pebble games". In 2015 IEEE 56th Annual Symposium on Foundations of Computer Science. Pages 466–485. (2015).

[7] Alex Parent, Martin Roetteler, and Krysta M. Svore. "REVS: A Tool for Space-Optimized Reversible Circuit Synthesis". In Iain Phillips and Hafizur Rahaman, editors, Reversible Com-

putation. Pages 90–101. Lecture Notes in Computer ScienceCham (2017). Springer International Publishing.

[8] Alex Parent, Martin Roetteler, and Krysta M. Svore. “Reversible circuit compilation with space constraints” (2015). arXiv:1510.00377 [quant-ph].

[9] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. “Quipper: A scalable quantum programming language”. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Page 333–342. PLDI ’13New York, NY, USA (2013). Association for Computing Machinery.

[10] Matthew Amy, Martin Roetteler, and Krysta M. Svore. “Verified Compilation of Space-Efficient Reversible Circuits”. In Rupak Majumdar and Viktor Kunčak, editors, Computer Aided Verification. Volume 10427, pages 3–21. Springer International Publishing, Cham (2017).

[11] Debjyoti Bhattacharjee, Mathias Soeken, Srijit Dutta, Anupam Chattopadhyay, and Giovanni De Micheli. “Reversible Pebble Games for Reducing Qubits in Hierarchical Quantum Circuit Synthesis”. In 2019 IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL). Pages 102–107. (2019).

[12] Charles H. Bennett. “Time/Space Trade-Offs for Reversible Computation”. SIAM Journal on Computing **18**, 766–776 (1989).

[13] Giulia Meuli, Mathias Soeken, Martin Roetteler, Nikolaj Bjorner, and Giovanni De Micheli. “Reversible Pebbling Game for Quantum Memory Management” (2019). arXiv:1904.02121 [quant-ph].

[14] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. “Q#: Enabling scalable quantum computing and development with a high-level dsl”. In Proceedings of the Real World Domain Specific Languages Workshop 2018. RWDSL2018New York, NY, USA (2018). Association for Computing Machinery.

A Notational Conventions

Table 2 summarizes notational conventions used in this work.

Symbol	**Meaning**
i	Imaginary unit
$o, p, q, r, t, u, \dots$	Qubits
$a, a^{(0)}, b, c, d, \dots$	Ancilla qubits
n	Number of qubits
C	Circuit
$G = (V, E)$	Circuit graph
$g^{val} = (V^{val}, E^{val})$	Value graph
$\overline{G} = (\overline{V}, \overline{E})$	Synthesized circuit graph
$v, v', \overline{v}, w, \dots$	Vertex
s	State index
i	Instance index
$q_{s.i}, p_{0.1}, r_{1.0}, \dots$	Vertex with explicit q, s, i
$c, \overline{c}$	Control vertex
$\varphi, \phi, \psi, \sum_j \gamma_j \lvert j\rangle$	Quantum state
$\gamma, \gamma', \overline{\gamma}, \lambda, \lambda', \overline{\lambda}, \dots$	Complex coefficient (see above)
j, k, l	Variables to sum over (see above)
Q	Set of qubits
A	Set of ancilla qubits
R	Set of non-ancillae qubits (rest)
$F\colon \{0,1\}^{n+1} \to \{0,1\}$	Classical function defining qfree gate with n controls
U	(Unitary) gate (e.g., X or CX)
$\llbracket G \rrbracket$	Semantics of a circuit graph, as a function over quantum states
$E(G)$	Extended graph of G
$\underline{q_{s.i}}$	Qubit in $E(G)$ holding a copy of $q_{s.i}$
$(\!\lvert G \rvert\!)_p$	Coefficient for the projection p of the semantics of $E(G)$

Table 2: Notational conventions used throughout this work.

B Algorithms

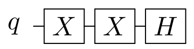
B.1 Partitioning

Fig. 13 shows the algorithm for partitioning the input graph.

B.2 Reqomp Convenience Methods

Fig. 14 shows convenience functions omitted from Reqomp in Fig. 2.

GetPath. The function getPath used by evolveVertexUntil is shown in Fig. 14. For ancilla variables, it simply returns the shortest path between the two values in the value graph. However for non ancilla variables, it forces the computation of intermediate values that may not have been computed yet. This could happend for a circuit such as:



Here the value graph is:

$$\begin{tikzcd} q_1 \arrow[r, bend left, "X"] & q_0 \arrow[l, bend left, "X"] \arrow[r, "H"] & q_2 \end{tikzcd}$$

$$q_1 \leftrightarrows q_0 \rightarrow q_2$$

Therefore, if we want to compute q_2 from q_0, the shortest path is simply $q_0 \rightarrow q_2$. However as H is not qfree, once q_2 has been computed, it can never be uncomputed again, and therefore, we can never compute

```
func PartitionAncillae()
    $G_{\text{anc}} \leftarrow$ AncillaDependencies($G$)
    $comps \leftarrow$ ConnectedComponents($G_{\text{anc}}$)                    ▷ Subgraphs of $G_{\text{anc}}$
    $G_{\text{ancdeps}} \leftarrow (comps, \{\})$                               ▷ Each comp is a vertex of $G_{\text{ancdeps}}$
    for $comp \in comps$ do
        for $comp' \in comps$ do
            if $\exists$ path from $c \in comp$ to $c' \in comp'$ in $G$ then
                addEdge$_{G_{\text{ancdeps}}}(comp, comp')$
    assert $G_{\text{ancdeps}}$ has no cycles
    return $comps$ in topological order according to $G_{\text{ancdeps}}$

func AncillaDependencies($G$)                                            ▷ Dependency graph on ancilla qubits
    $V_a \leftarrow \{v.\text{qbit} \mid v \in G, v.\text{isAnc}\}$
    $E_a \leftarrow \{(v.\text{qbit}, v'.\text{qbit}) \mid v \in G, v' \in \text{getControlled}_G(v)\} \cap V_a \times V_a$
    $G_a \leftarrow (V_a, E_a)$
```

Figure 13: Partitioning the uncomputation problem into independent subproblems.

```
func evolveControlledNonAncillae($\overline{v}$: Vertex)
    $v \leftarrow$ getVertex$_G(\overline{v}.\text{qbit}, \overline{v}.\text{valIdx})$
    for $t \in \text{getControlled}_G(v)$ do
        if not $t$.isAnc then
            if $\nexists t_{t.\text{valIdx}.i} \in \overline{G}$ then
                $\overline{t}' \leftarrow$ getVertex$_{\overline{G}}(t.\text{qbit}, \text{"last"})$
                evolveVertexUntil($\overline{t}'$, $t$.valIdx)

func evolveNonAncillae()
                                        ▷ Evolve remaining non-ancillas
    for $v \in$ final vertices in $G$ do
        if not $v$.isAnc then
            evolveQubitUntil($v$.qbit,$v$.valIdx)

func assertFullyEvolved()
                                        ▷ Abort if values were evolved incorrectly
    for $v \in$ final vertices in $G$ do
        $\overline{v} \leftarrow$ getVertex$_{\overline{G}}(v.\text{qbit}, \text{"last"})$
        if $\overline{v}$.isAnc then
            assert $\overline{v}.\text{valIdx} = 0$
        else
            assert $\overline{v}.\text{valIdx} = v.\text{valIdx}$

func getPath($q$: Qubit, $from$: int,$to$: int)
    $p \leftarrow$ shortestPathInValueGraph($q, from, to$)
    if $q$.isAnc then
        return $p$
    $p' \leftarrow []$
    $r \leftarrow [from + 1, to]$ if $from < to$ else $[to, from - 1]$
    $v \leftarrow from$
    for $i$ in $r$ do
        if $i \notin p$ and $\nexists j, q_{i.j} \in \overline{G}$ then
            $p' \leftarrow p' +$ shortestPathInValueGraph($q, v, i$)
            $v \leftarrow i$
    $p' \leftarrow p' +$ shortestPathInValueGraph($q, v, to$)
    return $p$
```

Figure 14: Convenience functions leveraged by Reqomp (Fig. 2).

q_1, which may be needed for some later computation. To correct this, we introduce q_1 (if it has not already been computed in $\overline{G}$) in the path, giving:

$$q_0 \rightarrow q_1 \rightarrow q_0 \rightarrow q_2$$

B.3 Linear Steps

Fig. 15 shows getLinearStrat. It is adapted from [5]: we added the *uncLast* parameters that allows us to apply it to ancillae only (that is we want all qubits to be computed once then uncomputed whereas the original algorithm did not uncompute the last qubit in the dependency line).

C Formal Correctness Proof

In the following, we provide a formal proof that Reqomp synthesizes correct uncomputation according to Def. 2.1.

C.1 Definitions and Helper Lemmas

We first define what we consider to be a valid circuit graph, following [1]:

Definition C.1 (Valid Circuit Graph). *A circuit graph is valid iff*

(i) its init vertices have no incoming target edge while gate vertices have exactly one,

(ii) all its vertices have at most one outgoing target edge

(iii) its anti-dependency edges can be reconstructed from its control and target edges according to the rule discussed in §4.1,

(iv) the number of incoming control edges of each gate vertex v matches the number of controls of the gate of v

(v) G is acyclic.

In a valid circuit graph, we can define for any non init vertex n its predecessor pred(n) as the only vertex m such that $m \rightarrow n$ (the target edge from m goes to n). We can also define for any qubit q its last vertex last(q): it is the only vertex on qubit q with no outgoing target edge.

```
func getLinearStrat($cc$: Qubit, $n_{\text{qbits}}$: int)
    $sortedAncillae \leftarrow$ topoSort($cc$)
    return [($sortedAncillae[i], b$) for $(i, b) \in$ stepsDP($|sortedAncillae|, n_{\text{qbits}}$, false)]

func stepsDP($n_{\text{anc}}$: int, $n_{\text{qbits}}$: int, $uncLast$: bool)
    if return value was computed previously then
        return previously computed value                                    ▷ memoization
    if $n_{\text{anc}} = 0$ then
        return []
    if $n_{\text{anc}} = 1$ then
        if $n_{\text{qbits}} = 0$ then
            return null
        if $uncLast$ then
            return [(0, true), (0, false)]
        else
            return [(0, true)]
    for $m \in \{1, \ldots, n_{\text{anc}} - 1\}$ do
        if $uncLast$ then
            $toM \leftarrow$ stepsDP($m, n_{\text{qbits}}$, false)                           ▷ $0 \rightarrow m$
            $fromM \leftarrow [(i + m, b)$ for $(i, b) \in$ stepsDP($n_{\text{anc}} - m, n_{\text{qbits}} - 1$, true)]   ▷ $m \rightleftarrows n_{\text{anc}}$
            $cleanM \leftarrow [(i, \neg b)$ for $(i, b) \in$ reverse(stepsDP($m, n_{\text{qbits}}$, false))]     ▷ $0 \leftarrow m$
        else
            $toM \leftarrow$ stepsDP($m, n_{\text{qbits}}$, false)                           ▷ $0 \rightarrow m$
            $fromM \leftarrow [(i + m, b)$ for $(i, b) \in$ stepsDP($n_{\text{anc}} - m, n_{\text{qbits}} - 1$, false)]  ▷ $m \rightleftarrows n_{\text{anc}}$
            $cleanM \leftarrow [(i, \neg b)$ for $(i, b) \in$ reverse(stepsDP($m, n_{\text{qbits}} - 1$, false))] ▷ $0 \leftarrow m$
        $steps_m \leftarrow toM + fromM + cleanM$
    return $\arg\min_{steps_m}$ cost($steps_m$)
```

Figure 15: Optimal uncomputation strategy for linear graphs.

We now restate the well-valued circuit graph definition, and illustrate it on an example.

Definition C.2 (Well-valued Circuit Graph)**.** *We say a valid circuit graph is well valued iff:*

(i) *all vertex names are of the form $q_{s.i}$ where q is the name of the vertex qubit, s and i are natural numbers*

(ii) *there are no duplicates*

(iii) *the init vertex on each qubit has name $q_{0.0}$ and for any $q_{s.i}$ in G, $q_{s.0}$ is in G*

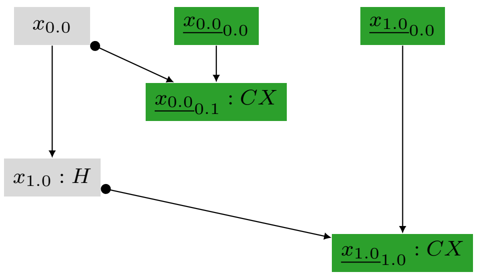
(iv) *any vertex $q_{s.i}$ verifies one of the following:*

(fwd) *$valIdx(pred(q_{s.i})) = valIdx(pred(q_{s.0}))$ and $q_{s.i}$ and $q_{s.0}$ have the same gate and same control vertices (up to their instance indices)*

(bwd) *if we denote $s' = valIdx(pred(q_{s.i}))$, we have that (i) $valIdx(pred(q_{s'.0})) = s$, (ii) $q_{s.i}.gate$ is qfree and equal to $q_{s'.0}.gate^\dagger$, and (iii)both $q_{s.i}$ and $q_{s'.0}$ have the same controls (up to instance indices).*

Vertices in a well-valued circuit graph are of the shape $q_{s.i}$, where we call s its value index (*valIdx* in the algorithms) and i its instance index. i is 0 for the first occurrence of q_s in the graph, but otherwise we only use its value to ensure uniqueness of the vertex names.

Due to the following lemma, it suffices to only consider valid and well-valued circuit graphs:



$x_{0.0}$
$\underline{x_{0.0}}_{0.0}$
$\underline{x_{1.0}}_{0.0}$
$\underline{x_{0.0}}_{0.1} : CX$
$x_{1.0} : H$
$\underline{x_{1.0}}_{1.0} : CX$

Figure 16: Extended graph example, copy vertices are shown in green.

Lemma C.1 (evolveVertex Correctness)**.** *For a valid and well-valued circuit graph G, any number of calls to evolveVertex results in a valid and well-valued circuit graph $\overline{G}$ such that (i) $\{q_{s.0} \in \overline{G}\}$ is a subset of $\{q_{s.0} \in G\}$ and (ii) for any $q_{s.0}$ in $G \cap \overline{G}$, it has the same gate and control vertices (up to instance index) in both graphs.*

Proof. By induction on the depth of calls to evolveVtx. □

We then define the extended graph $E(G)$ of a circuit graph G. Roughly, we want $E(G)$ to keep a copy of every vertex $q_{s.i}$ in G, saved on a fresh qubit $\underline{q_{s.i}}$. For a graph G with one qubit and two vertices, we show $E(G)$ in Fig. 16.

Definition C.3 (Extended Graph)**.** *For any circuit graph $G = (V, E)$, we define its extended graph*

$E(G) = (V_e, E_e)$ *as follows:*

$$\begin{aligned} V_e =& V \cup \left\{ \underline{q_{s.i}}_{0.0}, \underline{q_{s.i}}_{1.0} \mid q_{s.i} \in V \right\} \\ E_e =& E \cup \left\{ \underline{q_{s.i}}_{0.0} \rightarrow \underline{q_{s.i}}_{1.0} \mid q_{s.i} \in V \right\} \\ & \cup \left\{ q_{s.i} \bullet\!\!\rightarrow \underline{q_{s.i}}_{1.0} \mid q_{s.i} \in V \right\} \end{aligned}$$

For each $q_{s.i}$ in V, we have added a new qubit $\underline{q_{s.i}}$, with one init vertex and one gate vertex CX controlled by $q_{s.i}$. In the following we refer to those added qubits as $\underline{V}$. Note that while $\underline{q_{s.i}}_{1.0}$ is a vertex, $\underline{q_{s.i}}$ is qubit.

As the extended graph is a valid graph, it corresponds to a circuit and therefore its semantics $[\![E(G)]\!]$ is well defined. For a given input state φ to G, this allows us to define:

Definition C.4 (Projected Coefficients)**.** *For a fixed input state φ to the circuit graph $G = (V, E)$, we define the projected coefficients of G as the unique complex numbers $(\!|G|\!)_p$ such that:*

$$[\![E(G)]\!]\varphi \otimes |0...0\rangle_{\underline{V}} = \sum_{p:E(G).qbs \rightarrow \{0,1\}} (\!|G|\!)_p \, |p(G.qbs)\rangle_{G.qbs} \otimes |p(\underline{V})\rangle_{\underline{V}}$$

where $p(Q) = (p(q^{(1)}), ..., p(q^{(n)}))$ for qubits $Q = \{q^{(1)}...q^{(n)}\}$.

Using these coefficients, we can prove the following three lemmas. First, the semantics of the circuit graph G can be expressed in terms of its projected coefficients $(\!|G|\!)_p$:

Lemma C.2 (Projected Coefficients for Graph Semantics)**.** *For a circuit graph G we have:*

$$[\![G]\!]\varphi = \sum_{p:E(G).qbs \rightarrow \{0,1\}} (\!|G|\!)_p \, |p(G.qbs)\rangle$$

Proof. We can prove this by induction on the number of gates in G. □

Second, copies have consistent values. Specifically, for a given qubit q and valIdx s, all $\underline{q_{s.i}}$ hold the same value as $\underline{q_{s.i}}$, and the value of q is the same as the copy of the last vertex on q:

Lemma C.3 (Null Projected Coefficients)**.** *For a valid and well-valued circuit graph $G = (V, E)$ and $p : E(G).qbs \rightarrow \{0, 1\}$, we have $(\!|G|\!)_p = 0$ if*

(i) $p(\underline{q_{s.i}}) \neq p(\underline{q_{s.0}})$ for some $q_{s.i}$, or

(ii) $p(q) \neq p(\underline{last(q)})$ for some qubit q.

Proof. We prove Lem. C.3 in App. C.3. □

Finally, if $(\!|G|\!)_p \neq 0$, it depends only on the gates used for the first computation of each $q_{s.0}$.

Lemma C.4 (Projected Coefficients Values)**.** *For a circuit graph G and $p : E(G).qbs \rightarrow \{0, 1\}$, we have that if $(\!|G|\!)_p \neq 0$ then:*

$$(\!|G|\!)_p = \alpha_{p(q_{0.0}^{(0)} \cdots q_{0.0}^{(n)})} \prod_{\substack{q_{s.0} \in G \\ s \neq 0}} \gamma^{q_{s.0}}$$

Here the α describe the initial state:

$$\varphi = \sum_{k \in \{0,1\}^m} \alpha_k \, |k\rangle \, .$$

and the γ are gate coefficients defined such that $[\![g]\!] \, |c\rangle \, |t\rangle = \sum_{t'=0}^{1} \gamma^{g}_{t,c \rightarrow t'} \, |c\rangle \, |t'\rangle$ for a gate g and t in $\{0, 1\}$ and $c \in \{0, 1\}^m$. We have further shortened

$$\gamma^{q_{s.0}} = \gamma^{q_{s.0}.gate}_{p(\underline{pred}(q_{s.0})), p(\underline{ctrls}(q_{s.0})) \rightarrow p(\underline{q_{s.0}})}$$

Proof. We prove Lem. C.4 in App. C.3. □

C.2 Main Proof

Using Lem. C.1–C.4, we can prove the correctness of Reqomp:

Theorem C.1 (Correctness)**.** *Have G a circuit graph built from a circuit with n qubits, of which m are ancilla variables. Without loss of generality, we can assume that those ancilla variables $A = \left(a^{(1)}, \ldots, a^{(m)}\right)$ are the first m qubits of G. Let* Reqomp*$(G, A) = \overline{G}$. If*

$$|0 \cdots 0\rangle_A \otimes \varphi \xmapsto{[\![G]\!]} \sum_{k \in \{0,1\}^m} \lambda_k \, |k\rangle_A \otimes \phi_k, \text{ then} \tag{4}$$

$$|0 \cdots 0\rangle_A \otimes \varphi \xmapsto{[\![\overline{G}]\!]} \sum_{k \in \{0,1\}^m} \lambda_k \, |0 \cdots 0\rangle_A \otimes \phi_k. \tag{5}$$

Note that this is an equivalent rewrite of Def. 2.1.

Proof. We first make the values of the non-ancilla qubits explicit, and denote $R = G.qbs \backslash A$. This allows us to rewrite Eq. (4) as :

$$[\![G]\!] \, |0 \cdots 0\rangle_A \otimes \varphi = \sum_{\substack{k \in \{0,1\}^m \\ k' \in \{0,1\}^{n-m}}} \lambda_{kk'} \quad |k\rangle_A \, |k'\rangle_R \tag{6}$$

Similarly for $\overline{G}$ we can write:

$$[\![\overline{G}]\!] \, |0 \cdots 0\rangle_A \otimes \varphi = \sum_{\substack{k \in \{0,1\}^m \\ k' \in \{0,1\}^{n-m}}} \overline{\lambda_{kk'}} \quad |k\rangle_A \, |k'\rangle_R \tag{7}$$

Note that here we use $\overline{\lambda}$ to refer to a coefficient in $\overline{G}$, and not to the complex conjugate of λ.

To prove the theorem, it is hence enough to prove that for all k',

$$\overline{\lambda_{kk'}} = \begin{cases} 0 & \text{if } k \neq 0 \qquad \text{(i)} \\ \sum_k \lambda_{kk'} & \text{if } k = 0 \qquad \text{(ii)} \end{cases}$$

To do so, we first identify Eq. (7) with Lem. C.2. This gives us that:

$$\overline{\lambda_{kk'}} = \sum_{\substack{p:E(\overline{G}).qbs\to\{0,1\}\\ p(\overline{G}.qbs)=kk'}} \llparenthesis\overline{G}_p\rrparenthesis \tag{8}$$

The assertion at Lin. 24 in the Reqomp algorithm (Fig. 2) and Lem. C.3 then give that for any ancilla qubit $a^{(i)}$, if $p(a^{(i)}) \neq p(\underline{a^{(i)}_{0.0}})$, then $\llparenthesis\overline{G}_p\rrparenthesis$ is null. As $\underline{a^{(i)}_{0.0}}$ copies the initial state of the ancilla, we then get that if $k \neq 0$, then $\overline{\lambda_{kk'}} = 0$, proving (i).

To prove (ii), we first note that Eq. (8) holds analogously for G, allowing us to derive the following. Here, we denote $V_0 = \{q_{s.0} \in V\}$. We then have for any k' in $\{0,1\}^{n-m}$:

$$\begin{aligned}\sum_{k\in\{0,1\}^m} \lambda_{kk'} &= \sum_{k\in\{0,1\}^m} \sum_{\substack{p:E(G).qbs\to\{0,1\}\\ p(G.qbs)=kk'}} \llparenthesis G_p\rrparenthesis && (9)\\ &= \sum_{\substack{p:E(G).qbs\to\{0,1\}\\ p(R)=k'}} \llparenthesis G_p\rrparenthesis && (10)\\ &= \sum_{p_0:V_0\to\{0,1\}} \sum_{\substack{p:E(G).qbs\to\{0,1\}\\ p(R)=k'\\ p_{|V_0}=p_0}} \llparenthesis G_p\rrparenthesis && (11)\end{aligned}$$

Using Lem. C.3, we have that for any $p_0 : V_0 \to \{0,1\}$, there is a unique $p_0^+ : E(G).qbs \to \{0,1\}$ such that $p_{|V_0} = p_0$ and $\llparenthesis G\rrparenthesis_p$ is not known to be null. We can hence further rewrite Eq. (11):

$$\sum_{k\in\{0,1\}^m} \lambda_{kk'} = \sum_{\substack{p_0:V_0\to\{0,1\}\\ p_0(R_0)=k'}} \llparenthesis G\rrparenthesis_{p_0^+}, \tag{12}$$

where we denoted $R_0 = \{\underline{q_{s.0}} \mid \text{last}(q) = q_{s.i}, q \in R\}$. Similarly, we write the same equation for $\overline{G}$ and $\overline{\lambda_{kk'}}$:

$$\sum_{k\in\{0,1\}^m} \overline{\lambda_{kk'}} = \sum_{\substack{p_0:\overline{V_0}\to\{0,1\}\\ p_0(\overline{R_0})=k'}} \llparenthesis\overline{G}\rrparenthesis_{p_0^+} \tag{13}$$

Using that $\overline{\lambda_{kk'}}$ is null if $k \neq 0$, we finally get:

$$\overline{\lambda_{0k'}} = \sum_{\substack{p_0:\overline{V_0}\to\{0,1\}\\ p_0(\overline{R_0})=k'}} \llparenthesis\overline{G}\rrparenthesis_{p_0^+} \tag{14}$$

Now, if $V_0 = \overline{V_0}$, as gates are the same for any $q_{s.0}$ in G and $\overline{G}$, Eq. (12) and Eq. (14) combined with Lem. C.4 and Lem. C.1 give us (ii).

If this is not the case, there must exist some $q_{s.0}$ in $V_0 \backslash \overline{V_0}$. For instance this could happen if G contains $q_{0.0} \to q_{1.0} \to q_{0.1}$, and $\overline{G}$ only contains $q_{0.0}$: it was not necessary to compute $q_{1.0}$ to reach the same final state as in G. The crucial observation is that this vertex $q_{s.0}$ gate is qfree. If q is an ancilla, this is clear as Reqomp would have raised an error otherwise. Indeed, Reqomp computes all ancillae (in Lin. 18) and check that they are all later uncomputed (Lin. 24). All operations on ancillae are hence uncomputed, and therefore their gate must be qfree (this is checked in Lin. 40). If q is not an ancilla, it means then $q_{s.0}$ must have been uncomputed in G (as the final state of non ancilla qubits in both graphs is the same). As qfree gates coefficients γ are either 0 or 1, having an extra qfree gates does not change the result of the sum in Eq. (12), concluding this proof. □

C.3 Proofs of Helper Lemmas

We now prove Lem. C.3 and Lem. C.4 by induction on the number of gates in G.

Proof. For a circuit graph G with no gates, an immediate induction on the number of qubits gives both lemmas.

Now suppose both lemma holds for any G with at most l gates. Now have $G' = (V', E')$ with $l+1$ gates. We can write G' as $G' = G \cdot q_{s.i}$ where $G = (V, E)$ has l gates and $q_{s.i}$ can be applied last in G'. To simplify notations, we assume $q_{s.i}$ has only one control vertex $c_{t.j}$. If it has 0 or more controls, the reasoning is analogous.

By definition, we know that:

$$[\![E(G)]\!]\varphi = \sum_{p:E(G).qbs\to\{0,1\}} \llparenthesis G\rrparenthesis_p \left|p(G.qbs)\right\rangle_{G.qbs} \left|p(\underline{V})\right\rangle_{\underline{V}}$$

If we now apply $q_{s.i}$ and $\underline{q_{s.i}}_{0.1}$ (the CX gate copying $q_{s.i}$ to a new qubit) to one state of the sum above, we get for any $p : E(G).qbs \to \{0,1\}$:

$$[\![q_{s.i} \cdot \underline{q_{s.i}}_{0.1}]\!] \left|p(G.qbs)\right\rangle_{G.qbs} \left|p(\underline{V})\right\rangle_{\underline{V}} = \sum_{b\in\{0,1\}} \gamma^{q_{s.i}.gate}_{p(q),p(c)\to b} \left|p(G.qbs\backslash\{q\}), b\right\rangle_{G.qbs} \left|p(\underline{V}), b\right\rangle_{\underline{V'}}$$

Here b appears first as the value on the qubit q, and second as the value on the copy qubit $\underline{q_{s.i}}$.

As $E(G') = E(G) \cdot q_{s.i} \cdot \underline{q_{s.i}}_{0.1}$, we can use the above to compute $\llparenthesis G'\rrparenthesis_{p'}$ for any $p' : E(G').qbs \to \{0,1\}$. We first notice that if $p'(q) \neq p'(\underline{q_{s.i}})$, then $\llparenthesis G'\rrparenthesis_{p'} = 0$, giving us in Lem. C.3 (ii) for q. In the following, we hence only consider p' such that $p'(q) = p'(\underline{q_{s.i}})$. We then get:

$$\llparenthesis G'\rrparenthesis_{p'} = \sum_{b\in\{0,1\}} \llparenthesis G\rrparenthesis_{\substack{p'_{|E(G).qbs}\\ [q\mapsto b]}} \gamma^{q_{s.i}.gate}_{b,p'(c)\to p'(q)} \tag{15}$$

Using the recursion hypothesis, this immediately gives that if for any $q' \neq q$ if $p'(q') \neq p'(\underline{last(q')})$, then $(\!|G'|\!)_{p'} = 0$, giving us Lem. C.3 (ii) for $q' \neq q$. Together with the above, we hence get Lem. C.3 (ii).

We now work on proving both Lem. C.3 (i) and Lem. C.4. The recursion hypothesis gives us that for any $(q', s', i') \neq (q, s, i)$, if $p'(\underline{q'_{s'.i'}}) \neq p'(\underline{q'_{s'.0}})$, then again $(\!|G'|\!)_{p'} = 0$. We hence only need to establish that if $p'(q_{s.i}) \neq p'(q_{s.0})$ then $(\!|G'|\!)_{p'} = 0$, and the value of this coefficient when it is not null (that is to say Lem. C.4). To do so, we now consider p' consistent with what we have already proven, i.e., p' such that for any q' in $G.qbs$, $p'(q') = p'(\underline{last(q')})$ and for any $(q', s', i') \neq (q, s, i)$, $p'(q'_{s'.i'}) = p'(q'_{s'.0})$.

We first notice that the recursion hypothesis gives that if $b \neq p'(\underline{pred(q_{s.i})})$, then $(\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto b}} = 0$. Hence one of the summands in Eq. (15) is null:

$$(\!|G'|\!)_{p'} = (\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}} \gamma^{q_{s.i}.gate}_{p'(\underline{pred(q_{s.i})}),p'(c)\to p'(q)}$$

Using the constraints on p', we can rewrite this to:

$$(\!|G'|\!)_{p'} = (\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}} \gamma^{q_{s.i}.gate}_{p'(\underline{pred(q_{s.i})}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s.i}})}$$

Now we distinguish two cases. If this is the first occurence of q_s, that is to say $i = 0$, we immediately get Lem. C.3 (i), as $i = 0$. For Lem. C.4, by rewriting the equation above using that $i = 0$

$$(\!|G'|\!)_{p'} = (\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto p'(\underline{pred(q_{s.0})})}} \gamma^{q_{s.0}.gate}_{p'(\underline{pred(q_{s.0})}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s.0}})}$$

and using the induction hypothesis, we can conlude.

On the other hand, if $i \neq 0$, we again need to distinguish two possibilities: cases **fwd** and **bwd** in Item (iv) of Def. 4.1We focus on the later case, as the first is simpler. We denote $q_{s'.i'} = \mathrm{pred}(q_{s.i})$. We can hence rewrite:

$$\gamma^{q_{s.i}.gate}_{p'(\underline{pred(q_{s.i})}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s.i}})} = \gamma^{q_{s.i}.gate}_{p'(\underline{q_{s'.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s.i}})}$$

As we know that G' is well-valued, we have that $q_{s.i}.gate = q_{s'.0}.gate^\dagger$ and that both gates are qfree. Generally, the coefficient for the reverse of a qfree gate g is

$$\gamma^{g^\dagger}_{t,c\to t'} = \gamma^{g}_{t',c\to t},$$

as a qfree gate coefficient can only be 0 or 1.

We can hence again rewrite the above coefficient as:

$$\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.i}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}.$$

Here, we used that $\left(g^\dagger\right)^\dagger = g$. Overall this gives us that:

$$(\!|G'|\!)_{p'} = (\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto p'(\underline{pred(q_{s.0})})}} \gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.i}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$$

Now if $p'(\underline{q_{s.i}}) \neq p'(q_{s.0})$, let us prove that $(\!|G'|\!)_{p'}$ is null. If $(\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}}$ is null this is clear, otherwise using Lem. C.4 we get that $(\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}}$ contains $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$. We hence have that $(\!|G'|\!)_{p'}$ is a product which includes the factors $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.i}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$ and $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$. As $q_{s'.0}.gate$ is qfree, one of those coefficients is null, and hence so is $(\!|G'|\!)_{p'}$.

Finally, if $p'(\underline{q_{s.i}}) = p'(q_{s.0})$, we get Lem. C.3 (i) trivially. If $(\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}}$ is null, Lem. C.4 holds trivially. Otherwise, we use as above that $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$ is in $(\!|G|\!)_{\substack{p'_{|E(G).qbs}\\ \oplus q\mapsto p'(\underline{pred(q_{s.i})})}}$. As $\gamma^{q_{s'.0}.gate}_{p'(\underline{q_{s.0}}),p'(\underline{c_{t.0}})\to p'(\underline{q_{s'.0}})}$ is 0 or 1, it is equal to its squared value, and the recursion hypothesis allows us to conclude. $\square$

D Evaluation Values

We show in Table 4 the absolute numerical results on which the relative values in Table 1 are based. Table 3 further shows the exact parameters of each circuit used in our evaluation.

Table 3: Parameters for all examples in Table 1 and Table 4.

Algorithm	**Parameters**
Small	
Adder	12 qubits per operand
Deutsch-Jozsa	10 control qubits, with oracle MCX, returning true iff the value is 1111111111
Grover's algorithm	5 control qubits, with oracle MCX, returning true iff the value is 1111111111
IntegerComparator	12 control qubits, comparing to $i = 463$
MCRY	12 control qubits, with rotation angle $\theta = 4$
MCX	12 control qubits
Multiplier	5 qubits for each operand, and 5 for the result
PiecewiseLinearR	6 control qubits, function breakpoints are [10, 23, 42, 47, 51, 53, 63], slopes are [39, 32, 77, 27, 77, 4, 74] and offsets are [174, 40, 110, 163, 100, 185, 130]
PolynomialPauliR	5 control qubits, polynomial coefficients are [2, 2, 2, 2, 2]
WeightedAdder	10 controls qubits, values for sum are [0, 1, 1, 5, 2, 10, 4, 4, 9, 3]
Big	
Adder	100 qubits per operand
Deutsch-Jozsa	100 control qubits, with oracle MCX, returning true iff the value is 1111111111
Grover's algorithm	10 control qubits, with oracle MCX, returning true iff the value is 1111111111
IntegerComparator	100 control qubits, comparing to $i = 878234040205782925887743338143$
MCRY	200 control qubits, with rotation angle $\theta = 4$
MCX	200 control qubits
Multiplier	16 qubits for each operand, and 5 for the result
PiecewiseLinearR	40 control qubits, function breakpoints are [63870600266, 81180069351, 185076947411, 350818281077, 590566882159, 677977056232, 866030640015, 949186564661, 978976427282], offsets are [46, 59, 40, 48, 54, 67, 21, 71, 22] and coefficients are [60, 59, 6, 45, 83, 44, 34, 130, 130]
PolynomialPauliR	10 control qubits, polynomial coefficients are [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
WeightedAdder	20 controls qubits, values for sum are [9, 0, 9, 10, 2, 6, 10, 6, 8, 5, 8, 7, 8, 4, 0, 0, 5, 7, 5, 6]

Table 4: Reqomp results for the reductions presented in Table 1. We also report Unqomp results. Columns **Max** and **Min** report the results for the most aggressive settings, respectively optimizing only for number of qubits and optimizing only for number of gates. Columns **-75%**, **-50%**, and **-25%** report the gate counts when achieving the respective ancilla reductions. Entries "x" indicate that a given ancilla reduction was not achieved. Q is total number of qubits, A is number of ancillae, CX is number of CX gates, G is total number of gates and D is circuit depth.

	Ancilla Reduction														
	Max					**-75%**					**-50%**				
Algorithm	Q	A	CX	G	D	Q	A	CX	G	D	Q	A	CX	G	D
Small															
Adder	28	4	326	1132	482	x	x	x	x	x	30	6	270	924	346
Deutsch-Jozsa	15	4	78	331	162	x	x	x	x	x	15	4	78	331	162
Grover	8	2	192	839	456	x	x	x	x	x	x	x	x	x	x
IntegerComparator	17	4	110	456	251	x	x	x	x	x	18	5	100	422	231
MCRY	17	4	122	486	263	x	x	x	x	x	18	5	104	414	225
MCX	17	4	102	405	243	x	x	x	x	x	18	5	96	381	202
Multiplier	24	20	420	1500	550	x	x	x	x	x	x	x	x	x	x
PiecewiseLinearR	10	3	1000	3851	2188	x	x	x	x	x	10	3	1000	3851	2188
PolynomialPauliR	8	2	360	1381	846	x	x	x	x	x	x	x	x	x	x
WeightedAdder	25	40	689	2606	1184	x	x	x	x	x	x	x	x	x	x
Big															
Adder	207	7	6824	24664	8832	225	25	2820	9792	3005	250	50	2470	8492	2806
Deutsch-Jozsa	108	7	2700	10999	5632	125	24	1038	4351	2289	150	49	888	3751	2041
Grover	15	4	3600	15312	7734	x	x	x	x	x	15	4	3600	15312	7734
IntegerComparator	108	7	2720	11758	6190	125	24	1042	4584	2452	150	49	892	3938	2177
MCRY	209	8	7358	29430	15268	250	49	2096	8382	4646	300	99	1796	7182	4134
MCX	209	8	7278	29109	15283	250	49	2088	8349	4653	300	99	1788	7149	4141
Multiplier	79	64	4688	16768	5764	x	x	x	x	x	x	x	x	x	x
PiecewiseLinearR	47	6	31064	124834	67353	51	10	25252	99754	55636	61	20	23812	93512	50941
PolynomialPauliR	15	4	30322	119245	65640	x	x	x	x	x	15	4	30322	119245	65640
WeightedAdder	38	80	1857	7042	3259	x	x	x	x	x	x	x	x	x	x

	Ancilla Reduction										**Unqomp**				
	-25%					**Min**									
Algorithm	Q	A	CX	G	D	Q	A	CX	G	D	Q	A	CX	G	D
Small															
Adder	33	9	228	768	275	35	11	200	664	267	36	12	200	664	267
Deutsch-Jozsa	17	6	66	283	142	19	10	54	235	126	19	8	54	235	126
Grover	8	2	192	839	456	9	3	168	743	386	9	3	168	743	378
IntegerComparator	21	8	82	346	172	24	12	68	294	164	24	11	68	310	164
MCRY	21	8	86	342	177	24	12	68	270	161	24	11	68	270	161
MCX	20	7	84	333	179	23	12	66	261	154	23	10	66	261	154
Multiplier	x	x	x	x	x	24	20	420	1500	550	24	20	460	1580	573
PiecewiseLinearR	11	4	958	3671	1952	13	6	906	3465	1932	13	6	906	3583	1986
PolynomialPauliR	8	2	360	1381	846	9	3	330	1261	693	9	3	330	1261	735
WeightedAdder	x	x	x	x	x	25	40	689	2606	1184	25	40	749	2886	1322
Big															
Adder	275	75	2120	7192	2494	299	99	1784	5944	2291	300	100	1784	5944	2291
Deutsch-Jozsa	174	73	744	3175	1563	199	100	594	2575	1386	199	98	594	2575	1386
Grover	17	6	3000	12912	6279	19	8	2550	11112	5902	19	8	2550	11112	5852
IntegerComparator	175	74	742	3284	1662	200	100	596	2670	1484	200	99	596	2862	1484
MCRY	350	149	1496	5982	3145	400	200	1196	4782	2793	400	199	1196	4782	2793
MCX	349	148	1494	5973	3138	399	200	1194	4773	2786	399	198	1194	4773	2786
Multiplier	x	x	x	x	x	79	64	4688	16768	5764	79	64	5168	17728	5886
PiecewiseLinearR	71	30	22372	87224	45711	81	40	21050	81592	44966	81	40	21050	84852	46752
PolynomialPauliR	17	6	26962	105805	55664	19	8	26572	104245	50936	19	8	26572	104245	60488
WeightedAdder	x	x	x	x	x	38	80	1857	7042	3259	38	80	1989	7658	3458