IOP Publishing Mach. Learn.: Sci. Technol. 5 (2024) 025029 https://doi.org/10.1088/2632-2153/ad4252

MACHINE LEARNING
Science and Technology

OPEN ACCESS

RECEIVED
30 September 2023

REVISED
24 March 2024

ACCEPTED FOR PUBLICATION
23 April 2024

PUBLISHED
3 May 2024

PAPER

Discovering quantum circuit components with program synthesis

Leopoldo Sarra[1,2,*], **Kevin Ellis**[3] and **Florian Marquardt**[1,2]

[1] Max Planck Institute for the Science of Light, Staudtstraße 2, 91058 Erlangen, Germany
[2] Department of Physics, Friedrich-Alexander Universität Erlangen-Nürnberg, Staudtstraße 5, 91058 Erlangen, Germany
[3] Cornell University, Ithaca, NY, United States of America
[*] Author to whom any correspondence should be addressed.

E-mail: leopoldo.sarra@gmail.com

Keywords: quantum circuits, machine learning, program synthesis, quantum physics

Abstract

Despite rapid progress in the field, it is still challenging to discover new ways to leverage quantum computation: all quantum algorithms must be designed by hand, and quantum mechanics is notoriously counterintuitive. In this paper, we study how artificial intelligence, in the form of program synthesis, may help overcome some of these difficulties, by showing how a computer can incrementally learn concepts relevant to quantum circuit synthesis with experience, and reuse them in unseen tasks. In particular, we focus on the decomposition of unitary matrices into quantum circuits, and show how, starting from a set of elementary gates, we can automatically discover a library of useful new composite gates and use them to decompose increasingly complicated unitaries.

It has been theorized for decades that quantum computers can perform tasks more efficiently than classical ones [1]. Consider, for example, Shor's algorithm for factorization [2] or Grover's algorithm for search [3], the variational quantum eigensolver [4] or other quantum machine learning algorithms [5], which can all have a significant impact on important problems. Although those algorithms are very promising and justify the current effort in the development of large-scale quantum computers [6], it is difficult to exploit the advantages offered by quantum computing automatically. Indeed, each of those algorithms has been invented specifically for the task it solves, and often their principles do not generalize easily to other tasks. Currently barely two hundred algorithms exist [7]. While there is strong evidence of a quantum advantage [8], i.e. quantum computers can be more powerful than classical ones, and this advantage on near-term devices has been shown experimentally for specific purpose-designed tasks [9–11], we do not have a way to automatically make use of quantum principles such as superposition or entanglement to speed up classical algorithms: each algorithm must be designed from scratch and it is not known *a priori* whether a corresponding faster quantum algorithm could exist. Compared to the classical regime, quantum mechanics is generally counterintuitive, and thus a significant effort and imagination is required to conceive quantum algorithms. Hence, a technique for understanding the laws of the quantum world and finding efficient approaches to solve a given task on a quantum computer would be an invaluable tool in the development of new quantum algorithms and general principles which can grant quantum speed up. This long-standing goal is currently out of reach, but worthy of exploration to develop ingredients potentially useful for that purpose.

This paper takes a step in that direction by considering a simpler core subproblem: instead of working on entire quantum algorithms, we focus on the purely quantum part, excluding measurements and classical processing. We develop a machine learning technique to automatically produce a quantum circuit which performs a desired unitary transformation of a quantum state. The main innovation of our approach to this general problem is the gradual discovery of new composite gates, which can subsequently be used to decompose increasingly complicated unitary matrices. In this way, we can build a self-learning compiler, which can express a given unitary matrix into a given set of gates and satisfy given qubit connectivity constraints, without providing explicit transformation rules. Instead, we propose a training set of unitary matrices to decompose, with varying degrees of difficulty.

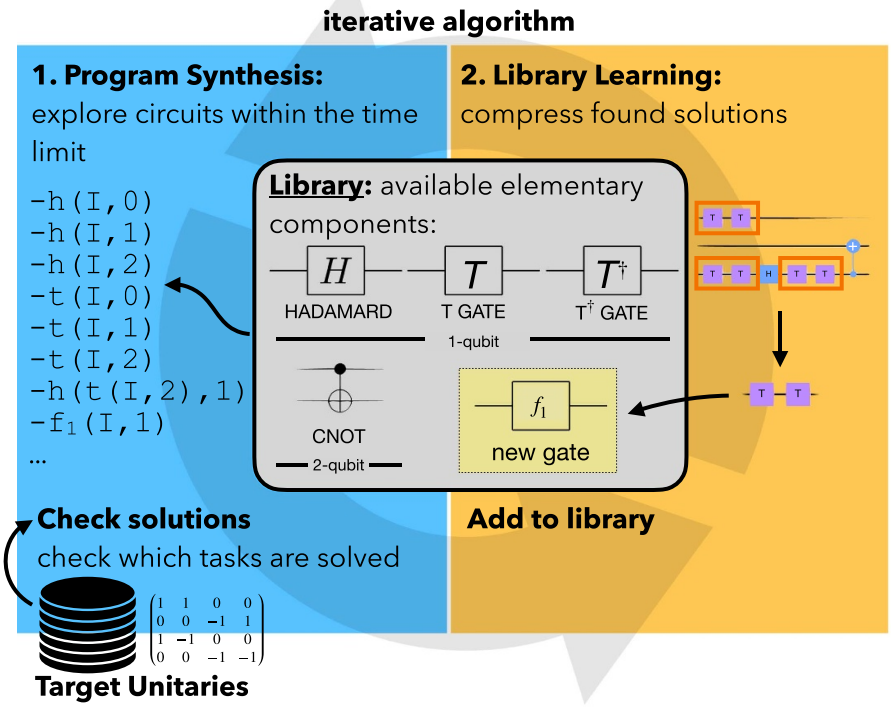


Figure 1. Synthesis of quantum unitaries. Given a dataset of target unitary matrices, we enumerate quantum circuits for a given timeout interval, using the elementary gates in the library as components. After some matrix decompositions have been found, the solutions are analyzed and the most useful components are added to the library as new available gates. The procedure is repeated for a given number of iterations.

At a high level, our method works by iteratively (1) searching for correct decompositions, assembling gates from our current set of components, and (2) extracting *new* components by analyzing the solutions found in the previous step. In doing so, our system gradually learns increasingly useful quantum operations, which it uses to decompose more complex matrices (figure 1). At first, our method automatically rediscovers suitable representations of common gates such as SWAP and CZ, even when they are not initially provided to the system. Importantly, it also proposes new unexpected combinations of gates which prove to be valuable ingredients in the construction of more sophisticated unitaries. Circuits too difficult to synthesize by randomly assembling the initial elementary components can be tractably found using new gates proposed by the system itself while building simpler circuits. In that way, the system bootstraps a set of new gates from solving simpler problems, which unlocks solving more complex problems, leading to new gates, and so on.

The technique presented here introduces concept extraction and program synthesis to the field of quantum computing and shows proof-of-concept applications to the domain of unitary matrix decomposition. We demonstrate the importance of bootstrapping advanced concepts from simpler ones, and how these concepts can be used to solve increasingly complex problems. The choice to express concepts as small 'programs' also allows better interpretability, in comparison to other possible black-box approaches such as neural networks [12]. Extensions of these ideas will allow significant improvements to existing machine learning methods in quantum computing.

1. Related works

The synthesis of quantum unitary matrices, to which we will also just refer as 'unitaries' in the following, is the process of building a quantum circuit by placing gates one after the other, acting on selected qubits, to reproduce the effect of the given unitary. The problem of building a circuit that reproduces the action of a given unitary, or of another quantum circuit, using only a given set of gates, or respecting some given constraints, is well-known in the literature, and is called compilation [13]. On the other hand, if the unitary is given by a circuit expressed in a set of gates, and we want to find an equivalent circuit which uses another set of gates, the process is called transpilation. Many algorithms already exist to compile given unitaries into circuits [14] that only use a given set of gates, e.g. the ones that can be physically implemented, some even using machine learning and reinforcement learning techniques [15]. Many other works explore different

directions for optimizing the total number of gates in a circuit [16] or reducing the number of a given kind of gate, which may be more expensive or noisy in the considered implementation [17]. For example, the Solovay–Kitaev algorithm [18, 19] can approximate with arbitrary error any given unitary matrix with a given finite set of gates, as long as this set can approximate any one-qubit gate. In our case, we build a self-learning compiler, which changes its behavior according to the experience, until converging to the optimal solution for the given architecture. This kind of compiler does not need explicit rules about how to decompose the given unitary matrix, but just a series of matrices to decompose with increasing difficulty. We emphasize that the main interest in our case is not in reproducing the performance of a state-of-the-art compiler, or transpiler, but to investigate how to imitate the ability of scientists to learn new concepts and reason with them, for example by building more and more complicated circuits from elementary components whose behavior is known. In our examples, we mainly optimize for conceptual efficiency, i.e. number of used high-level operations to express a quantum operation, rather than for the effective experimental implementation cost (e.g. total gate number minimization). Different constraints can potentially be chosen nonetheless.

After expressing quantum circuits more conveniently, we can take advantage of machine learning techniques to assemble them until the requested unitary is produced. We build on machine learning techniques for program synthesis [20]. Program synthesis methods automatically construct source code, and our work exploits the fact that a quantum circuit can be easily expressed as a simple program. Recent program synthesis techniques use neural networks to learn how to generate source code ([21], *inter alia*). Our work uses a slightly different family of learning methods which casts program synthesis as Bayesian inference [22]. This probabilistic Bayesian framing allows searching for the most likely programs that solve a given unitary, and also learning to generate good programs via hierarchical Bayesian methods. These Bayesian program learning methods were developed in a series of works [23–25]. We directly build upon DreamCoder [26], a recent work in this family.

Coming back to the much higher conceptual level, regarding the longer-term idea of an 'artificial scientist', the first proof-of-principle of an agent capable of conducting research on its own has been shown in [27], to automate functional genomics experiments. The idea of automatic concept discovery from experience is a helpful ingredient for this long-term goal, as it is a first step towards reasoning. Indeed, there has already been a large interest also in other fields of physics ranging from the design of new quantum optics setups [28], the use of symbolic regression and graph neural networks to find new laws of astrophysics [29], to the general development of algorithms capable to formulate scientific laws [30]. In [31], the idea of extraction of building blocks is used in a reinforcement learning setting to produce quantum entangled states. Also, projective simulation [32] allows employing reinforcement learning agents to explore novel algorithms for quantum communication [33]. While these techniques share the idea of concept extraction to solve a specific quantum task, in the usual reinforcement learning setting the discovery of new components is an incidental effect, obtained while achieving the task. In our case, the goal of circuit decomposition itself is the discovery of new useful gates, by minimizing the overall complexity of the solutions, quantified in terms of description length [34, 35].

2. Methods

In this section, we explain how our algorithm for unitary synthesis and gate extraction works. As shown in figure 2, quantum circuits can be seen as programs that subsequently apply operations to a given state: starting from the identity matrix, representing a circuit without any gates, each operation corresponds to a multiplication by the unitary matrix associated to the applied gate. The final unitary matrix is built up by sequentially multiplying all the unitary matrices. By working on programs that build up the sequence of operations that make a circuit, we can explore the space of possible unitary matrix decompositions with program synthesis.

To begin, we define a probability distribution over quantum circuits c. Our learning algorithm works by adjusting the distribution over c to make useful circuits more likely. The distribution depends on the set of allowed gates, G, together with the probability of each gate $g \in G$, which we write θ_g. We decompose it as a product of the probability of choosing the specific gates and applying those gates to the selected qubits. We assume gates are generated independently at random, and that they attach to wires (i.e. qubits) drawn uniformly and independently at random:

$$P(c|G,\theta) = \prod_{g\in c} \mathbb{1}\left[g \in G\right] \theta_g \chi\left(c,g\right), \tag{1}$$

where $\mathbb{1}$ is the indicator function, yielding one iff the condition is fulfilled, and $\chi(c,g)$ the probability of connecting the gate to the specific qubits. For example, we use uniform probability of associating a gate to

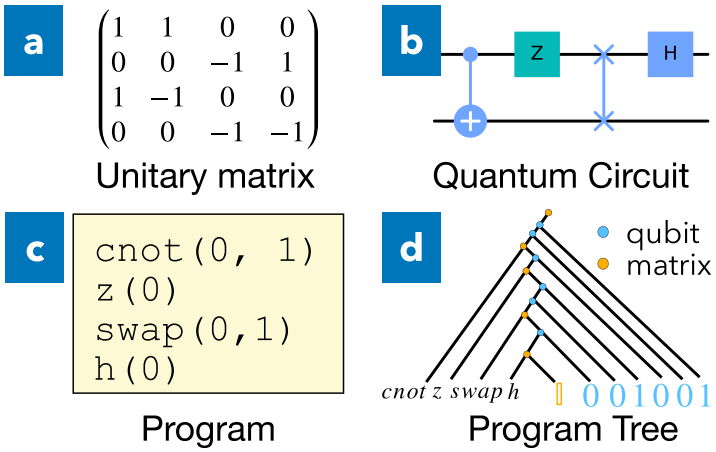


Figure 2. Unitary matrix representations. A quantum unitary matrix (a) can be represented as a quantum circuit (b), a sequence of instructions about the gates and the qubits on which they act (c), or a tree representation of a program (d). When enumerating new programs, the input identity matrix ($\mathbb{I}$) is replaced with another gate applied to other qubits, creating a new branch.

any of the possible qubits: $\chi(c,g) = N_c^{-n_g}$, with n_g is the number of inputs to gate g, N_c is the number of qubits in the circuit c. If the resulting circuit is not valid, for example if the inputs of a CNOT gate are repeated, it is automatically discarded, which leads to a small modification of the probability χ whose discussion we omit since it is not crucial for understanding the workings of the algorithm.

Notice that by optimizing the choice of the set of gates G, it is possible to make more complex circuits more likely.

Ultimately our aim is not to probabilistically score specific circuits but to learn a collection of gates that are valuable for solving a broad range of unitary synthesis problems. To that end, we assume that we have a training set of unitary matrices to decompose, collectively written U. Our algorithm tries to maximize the posterior probability of the gate set, given that it must solve every unitary in U: it finds the optimal gate set and optimal gate probabilities as

$$(G^*, \theta^*) = \arg\max_{G,\theta} P(G, \theta | U), \tag{2}$$

where we employ Bayesian reasoning to write

$$P(G, \theta | U) \propto \tag{3}$$

$$P(G) P(\theta | G) \prod_{u \in U} \sum_{c} \mathbb{1}\left[\mathcal{U}(c) = u\right] P(c | G, \theta), \tag{4}$$

where $P(G)$ is the prior over gate sets, $P(\theta|G)$ is the prior of gate weights of a given gate set, and $\mathcal{U}(c)$ is the operator that gives the unitary matrix associated to a given circuit. The above equation is computationally intractable because it includes summing over the infinite space of all circuits (inner sum over c). We introduce a tractable lower bound on equation (4) by only summing over a small set of possible circuits for each unitary. Writing $\mathcal{B}_u$ for the small set of circuits we consider for unitary u, our objective function becomes lower-bounded by

$$P(G) P(\theta | G) \prod_{u \in U} \sum_{c \in \mathcal{B}_u} \mathbb{1}\left[\mathcal{U}(c) = u\right] P(c | G, \theta). \tag{5}$$

Equation (5) serves as our core objective function for learning a library of gates. A more detailed derivation of this objective is given in the appendix B, and in the original work [26]. Maximizing it with respect to (G, θ) corresponds to updating our gate set to increase the probability of a circuit solving each unitary. Maximizing it with respect to $\mathcal{B}_u$ corresponds to program synthesis: finding a handful of likely circuits that evaluate to a given unitary.

More precisely, our system takes as inputs the example unitary matrices to learn to decompose, U, together with a set of initial elementary gates, G_0. The unitaries provided as examples in U determine which assemblies of gates are the most useful, thus the optimal set of learned gates G^*. The algorithm iterates many times through two phases: program synthesis, where circuits that decompose target matrices are proposed, and library learning, where concepts are extracted from the found circuits and the most useful sequences of

gates are added to the set of elementary gates G as a composite gate. It can then use the new gate as a single block in the subsequent iterations of program synthesis. Mathematically, program synthesis corresponds to maximizing equation (5) w.r.t. $\mathcal{B}_u$, while library learning corresponds to maximizing w.r.t. (θ, G). A sketch of the algorithm is shown in figure 1.

2.1. Program synthesis

During this phase of the algorithm, we seek the top k most likely programs solving each unitary:

$$\mathcal{B}_u \leftarrow \arg k - \max_c P(c|G,\theta)\,\mathbb{1}\left[\mathcal{U}(c) = u\right], \tag{6}$$

where $\arg k - \max_c$ is the function that returns the arguments with the largest k values. To find those top k circuits, we enumerate programs in order of decreasing probability under $P(\cdot|G,\theta)$ until k solutions have been found or we reach a timeout. We construct the syntax trees of candidate programs bottom-up, with higher probability expressions being generated first, using recent algorithms for probabilistic program enumeration [36, 37]. As an optimization, we discard any programs containing subexpressions that are semantically equivalent to higher-probability subexpressions, meaning they evaluate to the same unitary (in the literature called pruning by observational equivalence [38]). Within our implementation, we search for a maximum budget of 200 s and collect the top $k = 2$ programs for each unitary.

In practice, to accelerate the convergence in the algorithm, we do not update the programs for every unitary at each iteration. Instead, we sample a small batch of unitaries and only synthesize programs for those tasks. This is analogous to the use of mini batching for training neural networks using gradient descent [12]. Essentially, it allows taking fast, small learning steps (updating the set of available gates G) without examining and analyzing the entire training set.

Although enumeration may seem like a very basic program synthesis strategy, our goal is to learn a sophisticated set of gates G such that even a simple enumerative search can quickly uncover interesting unitaries. Thus, the ability of the program synthesis to succeed hinges critically on learning a good gate library G, which we describe next.

2.2. Library building

During the library building step, we augment the library of gates by adding new compositions of gates that the system itself proposes. We do this by analyzing the circuits found during the program synthesis phase and extracting commonly occurring patterns of gates. Adding these new patterns of gates to G increases the probability of generating circuits that use them.

On the other hand, the goal of library building is not to simply memorize every successful circuit, even though memorizing would most increase the probability of the programs found so far. Instead, we want to find new gates that generalize the patterns found in the synthesized programs. Striking the right balance is accomplished by prioritizing gate sets that are compressive, i.e. have small description length [34, 35]. Remembering that our goal is to maximize equation (4), we see that we need to not just make the circuits likely under G, but also have a G with a high prior probability $P(G)$. Our system uses a prior that assigns less probability to larger sets of gates and to gates with many subcomponents, which exerts pressure for proposing new gates that are small, yet broadly useful across many tasks.

Algorithmically, our system proposes new gates by extracting fragments of program syntax trees discovered during the previous program synthesis phase. Given a set of candidate new gates, G', it then constructs a set of candidate new libraries that extend the old library by exactly one gate: $\{G \cup \{g'\} \,:\, g' \in G'\}$. For each such $G \cup \{g'\}$, the system estimates a new θ using expectation-maximization [35]. The system finally computes the objective function in equation (5), and takes the gate which most increases it. This entire process repeats until equation (5) fails to improve, and then another round of program synthesis begins. See [26] for details.

Despite the apparent simplicity of the synthesis step, the overall algorithm is substantially more efficient than simple brute-force enumeration, as each component is used according to its assigned probability, and branches of the tree are pruned as they are discovered to be equivalent to already known branches. Also, the addition of the extracted gates to the set of elementary gates increases the breadth of the search tree (more components to choose from at each step) but reduces the required depth of the search (number of components to put together one after the other). Without these tricks, there would be a combinatorial explosion with the depth of the tree (e.g. $\mathcal{O}((gn)^d)$ with g elementary gates, n qubits and depth of the tree d, considering only 1-qubit gates in this rough estimate). It would be just unfeasible to decompose very long circuits, and this is why reducing the depth to explore is so helpful. Using the probabilistic guidance of the learned (G,θ), we can discover a circuit c in at most $\mathcal{O}(1/P(c|G,\theta))$, which may be much better than $\mathcal{O}((gn)^d)$ if the target circuit c employs similar computational motifs to the training data. In some sense, this

algorithm allows us to learn a domain-specific language for quantum circuits, by discovering a good prior to guide the circuit synthesis.

2.3. Convergence of the algorithm

We maximize the posterior in equation (4) by repeating two steps: (1) estimating a small collection of circuits that have high probability, and (2) improving the set of gates used to build those circuits (along with their probabilities θ). Both steps are approximations. We now discuss convergence properties of these approximations.

During the program synthesis phase we approximate $P(u|G,\theta)$, whose exact computation demands summing over every possible program that could generate u. Rather than sum over all programs, we consider the top k most likely that we can assemble in a given timeout interval t, effectively ignoring circuits with low prior probability (even if they might have high posterior probability). Formally, it is well known that quantities like $P(u|G,\theta)$ are lower-semicomputable [39], and in the limit as the timeout budget $t \to \infty$ and the program budget $k \to \infty$, our bounded approximation becomes exact. In practice, as we learn the prior over successive iterations, the approximation should become increasingly accurate as the system learns to solve more circuits: Initially we receive no learning signal from unitaries that cannot be solved within the given timeout, but as long as some new matrices are decomposed at each iteration, the prior (G,θ) should shift such that our bounded approximation covers a larger share of the posterior probability mass.

The other approximation that affects the convergence of the algorithm takes place during the library building step. Starting with an initial set of elementary gates, new gates are added incrementally at each step. This procedure is greedy—the next gate library is built on top of the one from the previous step—so this optimization may be suboptimal. In addition, since we sample a small batch of unitaries to decompose at each iteration, the batching order can affect the final outputs. After the first batch, some gate sequences may be selected and added to the elementary gate set. This will change the circuits enumerated during the next iteration. More computationally expensive alternatives include building a new gate set from scratch according to the enumerated solutions up to that particular iteration, or additional mechanisms such as pruning seldom-used gates. Larger batches of unitary matrices reduce the randomness of the algorithm.

As a consequence of these approximations, our algorithm does not necessarily converge to the global maximum of the likelihood in equation (2), but may get stuck in a local maximum. While the enumeration of programs that decompose a given matrix can be easily improved by increasing the timeout, and thus convergence to the correct circuits can be guaranteed, it is not possible to guarantee the convergence to the optimal library. However, it is not obvious that optimality on the training set is truly the goal: like many machine learning algorithms, the global optimum of the loss function depends on the training set (in our case, the unitary matrices to be decomposed), but performance is evaluated on the test set, which may have a different global maximum. Since we want to build a library of gates that can be used to efficiently decompose new (unseen) matrices, generalization properties are desirable.

3. Results and discussion

In this section, we show the application of our unitary matrix decomposition algorithm using an elementary set of gates, which can theoretically approximate any circuit. We show results when enforcing either full connectivity between qubits, or only allowing gates between nearest-neighboring qubits (e.g. between qubit 0 and 1 but not 0 and 2). To make the search faster and focus on the proof of concept, we limit ourselves to discrete gates, i.e. gates that do not depend on a tunable real parameter: this would require an optimization over the parameter of the gates, in addition to the search among the possible programs. For simplicity, we also fix the number of qubits to be the same in the entire set. In particular, to avoid the combinatorial explosion due to input qubit combinations (a n-qubit gate should be tested on all permutations of qubit inputs), we limit our examples to circuits with only 3 qubits, but generalizations to larger circuits are of course possible.

We choose $G_0 = \{\mathrm{H}, \mathrm{T}, \mathrm{T}^\dagger, \mathrm{CNOT}\}$ as the elementary gate set. This essentially corresponds to the Clifford gate set, plus T gates to make it a universal approximator [40, 41]. We also include the $\mathrm{T}^\dagger$ gate, which itself corresponds to seven T gates, to spifeed up the search. The choice of the target unitary set U is important, as the extracted gates will be selected to maximize the decomposition efficiency over those tasks. In general, some unitaries are much harder to approximate because they require many elementary gates. Sampling from the space of unitary matrices would generally produce matrices that are too hard to decompose when starting from our elementary set of gates and trying combinations of them. In our experiments, to be sure that it can be decomposed in a finite amount of search time, we build U by defining another set of gates, G_{tasks}, which uses more high-level operations. We sample circuits from G_{tasks} by randomly putting gates on the circuit. The unitaries associated with the sampled circuits will be the target for our algorithm. To keep the decomposition difficulty under control, also the gates in G_{tasks} have no continuous parameters, in particular

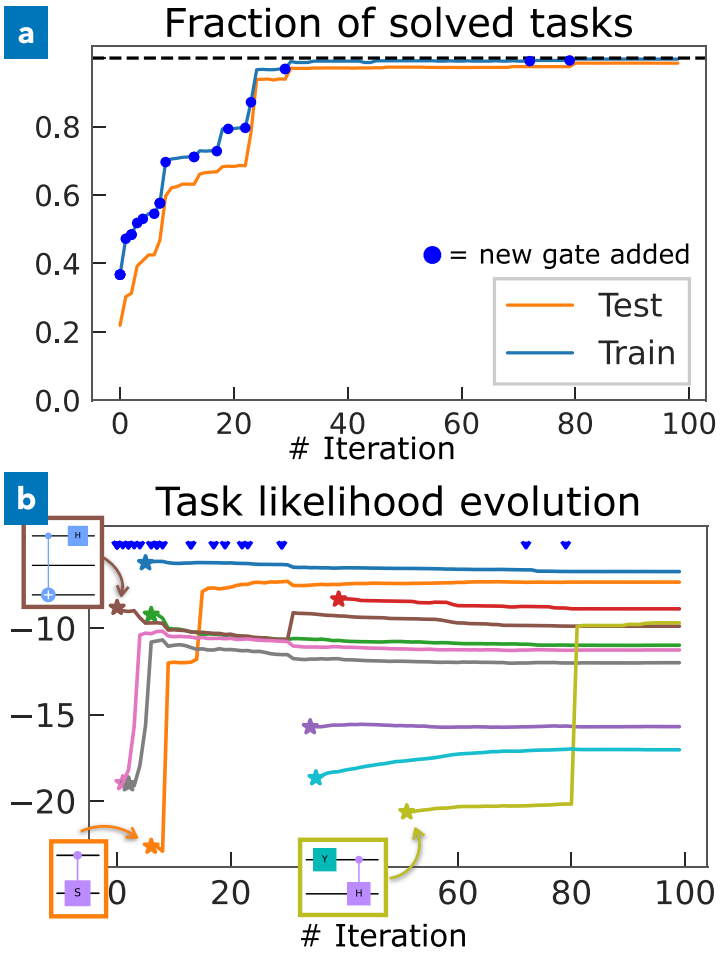


Figure 3. Unitary synthesis with full connectivity among qubits. (a) Fraction of solved tasks at each iteration. (b) Likelihood to decompose some target unitaries during the algorithm iterations ($P(u|G,\theta)$). The star symbol shows the first iteration in which the target matrix has been successfully decomposed. As new gates are discovered, some matrices become easier to decompose. For three tasks, we show the circuit that generates the target matrix in the insets. In both figures, the blue symbols mark the iterations at which a new gate has been extracted.

$G_{\text{tasks}} = \{\text{H, T, T}^\dagger\text{, S, X, Y, Z, SX, SX}^\dagger\text{, CNOT, CY, CZ, CS, CH, SWAP, iSWAP}\}$. We first generate a set of matrices by enumerating all the possible circuits given by this set within 50 s. From this set, we select 1000 matrices to build U. To make sure that the train set always contains a significant fraction of both easy and difficult decomposition tasks, we choose them with uniform probability in the number of gates of the initial circuits, taking into account only circuits generated within the timeout. All other generated unitaries are included in our test set T, and they will be used to assess the performance of the algorithm. The target dataset thus contains tasks with different levels of difficulty, allowing the algorithm to gradually learn to solve more and more complicated tasks. It is important to include unitaries with different decomposition lengths. Indeed, only after some tasks are solved it is possible to extract gates that can be used to solve other tasks since we need at least some elements in B_k in equation (5). If all tasks are too complicated, the learning procedure will not start and each iteration will not provide any benefit, since it will always propose the same programs. In that case, the enumeration timeout should be increased until some solutions are found.

We perform 100 algorithm iterations, each time considering batches of 25 tasks and enumerating for 150 s (in parallel with 32 CPUs). We see that after some iterations we can solve most of the about 50 000 of tasks in the test set. Every time we learn a new gate, the algorithm starts using it to explore new circuits thus solving more tasks. To evaluate the performance of the algorithm, we can consider the test set T (which the algorithm has never seen during the previous iterations) and check how many unitaries can be decomposed into a circuit. Results are shown in figure 3. After about 50 iterations, the algorithm can decompose almost all the proposed matrices. The algorithm rediscovers suitable decompositions of gates into the available elementary gates. For example, it finds useful elementary decompositions of high-level gates like the SWAP gate and the Pauli gates, adding them as building blocks to its library. Importantly, it goes beyond those simpler examples and discovers more complicated composite gates, which also seem to be useful to reuse. By discovering useful building blocks, matrices that are initially impossible to decompose in the given time budget because of the

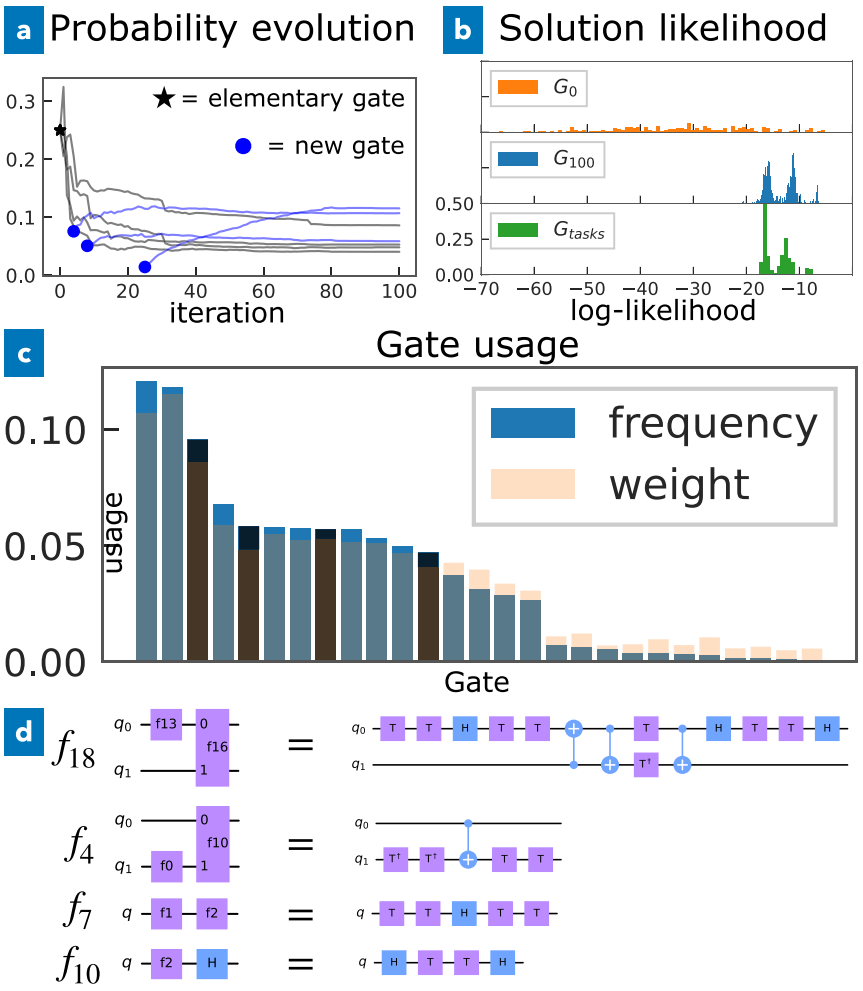


Figure 4. Analysis of extracted gates. (a) Evolution of the probability of using a certain gate (θ). Some extracted gates (in blue) become more important during the iterations, while some elementary gates (in black) become rarer. (b) Likelihood of decomposing the target unitaries with different elementary sets: the initial set of gates, the final one (which also includes the extracted gates), and the set of gates with which the target dataset has been generated. (c) Final probability of using a certain gate in G_{100} ('weight') and in the found solutions ('frequency'). In blue are shown the extracted gates, in black the initial ones (in decreasing order: CNOT, T, Hadamard, and $T^\dagger$). (d) The first few most useful extracted gates.

high number of required components are easily decomposed into short sequences of the newly extracted components. Indeed, as soon as a new gate is extracted, many new unitary matrices can immediately be decomposed. The 'train' and 'test' curves are obtained by checking all the found programs at a given iteration against each unitary matrix in the target set U and test set T. We recall that here we consider the whole target set for evaluation purposes, but the algorithm only has access to a small random batch of matrices at each iteration. As figure 3(b) shows, the difficulty to decompose a given matrix changes during the algorithm iterations: when new gates are added, some unitaries suddenly become easier to be decomposed, while other matrices, which mainly use the initial elementary components, become less likely to occur because those components become less frequent in the enumeration. Overall, all matrices become easier to decompose.

It is also interesting to inspect the library of extracted gates and try to interpret their behavior. As seen in figure 4, after discovering composite gates, initial gates like T become less useful and are used less often, while some new gates like 'f18', which corresponds to a controlled Hadamard gate suitably decomposed into elementary gates, and 'f4', which corresponds to a controlled Z gate, are used more often than CNOT. By looking at figure 4(b), we see that, initially, it is generally very hard to find decompositions for the target matrices (in orange). After running the algorithm, we have a new set of gates (in blue) that allows decomposing most of the matrices. We go from problems with probability $\simeq e^{-60}$ to be solved to $\simeq e^{-20}$, about 17 orders of magnitude larger probability, which make it feasible to find the decomposition in a finite amount of time. It is interesting to observe how a different choice of elementary components can make the decomposition easier. In particular, the extracted set of gates at the final iteration, G_{100}, makes the decomposition of the target matrices even easier than when using exactly the same set we used to generate

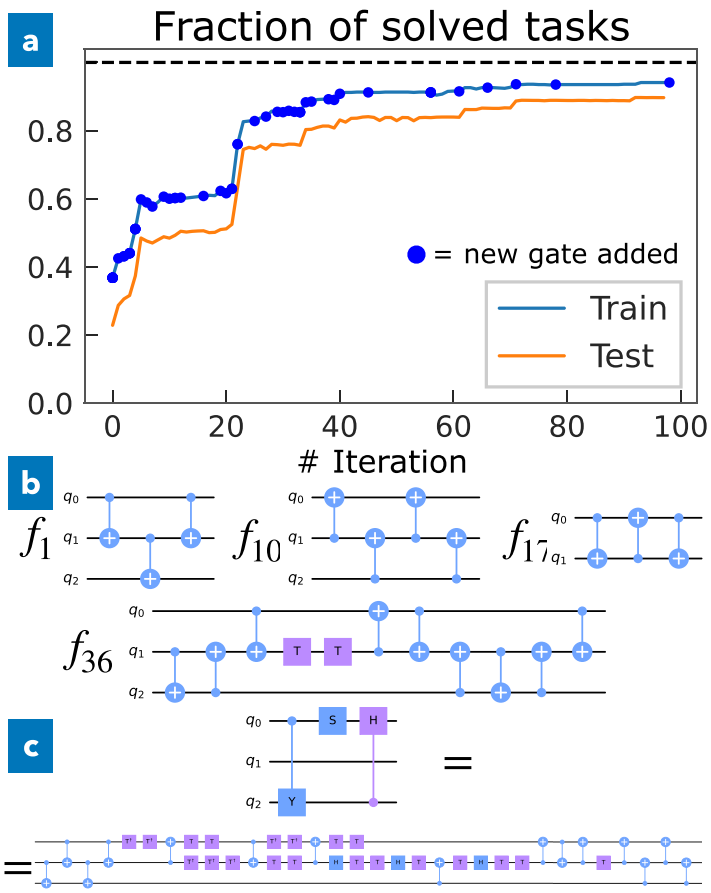


Figure 5. Unitary synthesis with only nearest-neighbor connectivity among qubits. (a) Percentage of solved tasks at each iteration. (b) Some extracted gates. (c) Example decomposition of a high-level circuit into the requested gate set.

the target dataset itself (in green). In other words, our algorithm discovers a set of quantum gates to describe the target dataset that is even better than the set that we used to generate it, G_{tasks}. For example, it turns out that it is much more useful to have two-qubit gates like CH and CZ than CNOT.

We also performed another experiment with the same parameters, but this time we constrained the 2-qubit gates to only act on neighboring qubits. This configuration resembles a linear array of qubits, where interactions are constrained to nearest neighbors. Also in this case, the algorithm learns to decompose more and more matrices with experience. We notice that this time a larger fraction cannot be decomposed yet even after 100 iterations. This is due to the larger difficulty of this problem, and with more iterations and larger enumeration timeout results would keep improving. Again, the algorithm learns more complicated gates and finds similar results as in the previous example. In addition, it also learns gates that allow it to efficiently handle the enforced connectivity constraint, like the SWAP gate between the first and the third qubit (by swapping with the middle one) and similar two and three-qubit gates. The automatic extraction of composite gates allows for expressing concisely very long sequences of gates. Results are shown in figure 5.

The final outcome of the algorithm depends of course on the chosen definition of being a better set of gates: in this case, we wanted to minimize the number of components to put in a circuit so that all target matrices could be decomposed with some high-level gates. However, different constraints can be considered, for example, to include the overall length of the circuit in terms of elementary components or a different cost for the use of each component.

4. Outlook

In this paper, we have shown how concept extraction and program synthesis techniques can potentially help quantum computing, by providing tools to work and reason with the quantum world. In particular, we have shown a procedure to discover useful quantum gates (in terms of reusability) by just giving a set of unitary matrices to decompose. This can be seen as a first step toward the longer-term goal of enabling the discovery of new quantum algorithms.

The extension to larger qubit numbers will require careful optimization of the performance of the different parts of the algorithm (search and library building), but we anticipate there is a lot of room for improvement here. Indeed, our experiments take about 20 min per iteration to run, and systems with more qubits would require much more time. It is also possible to test the generalization capabilities by running first on smaller systems and then trying to solve more complicated tasks on larger systems. To improve performance, it would be possible to restrict the decomposition to the Clifford gate set, so that calculation would be faster, e.g. by exploiting highly optimized Clifford simulators that can deal with large qubit numbers [42]. To tackle the combinatorial explosion due to the larger number of possible qubits a gate can be applied to, more advanced approximations for the circuit distribution $P(c)$ in equation (1) may be employed. For example, instead of factorizing the circuit distribution as the product of the probability of its gates, we could condition the probability of a gate to the previous ones, improving the precision of the enumerator.

One of the most important future extensions would concern the choice of the set of target unitaries. While in our case these were generated as random circuits from a high-level gate set, the application to more structured training sets would greatly increase the power of the approach. We are thinking, in particular, of circuits generated from a library of quantum algorithms.

Different connectivity constraints may also be enforced, or one could extend the programs that generate the circuits to also include programming constructs like conditions and loops. In the long run, the presented library bootstrapping procedure can be part of future algorithms to automatically extract components and reuse them in a curriculum-learning approach [43]. Also, it would be possible to adopt this concept extraction algorithm as an additional step of a reinforcement learning agent [44] that tries to decompose a unitary matrix. In that case, the goal would be to train an agent (i.e. a probability distribution of putting a certain gate given the current circuit) to synthesize the unitary, where the state would be the current circuit, and the possible actions would be the allowed elementary gates. It would be possible to extend the action space by adding the extracted gates, thus facilitating the exploration of the circuit space, as in hierarchical reinforcement learning [45].

Finally, on a more general level, the ability to extract concepts and to use them in further exploration is also interesting for the development of future 'artificial scientist' algorithms, here applied to the quantum domain and specifically quantum computation, aimed at reasoning and developing scientific models similarly to humans: the possibility to define concepts and reason about them is reasonably a necessary skill for this purpose. Similar techniques can be a useful addition to existing machine learning algorithms for quantum circuit design, and, in the long run, they may help to develop new quantum algorithms.

The code of the algorithm and the instructions to reproduce the presented examples are open-sourced on GitHub [4].

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://zenodo.org/record/8213680 [46].

Acknowledgments

This work was supported by the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus, and by the Max Planck Society.

Appendix A. The program synthesis algorithm

To apply the program synthesis framework presented in [26], we need to express quantum circuits as programs. The specific formalism that we adopt is that of functional programming, typed-λ-calculus [47] in particular. Each quantum gate becomes a function that takes as input a quantum circuit and the sequence of qubits on which it should act, and returns a quantum circuit with the requested gate applied on the right. In this way, a program is simply the sequential application of many gates, starting from the empty circuit $\mathbb{I}$.

For example, the circuit in figure A1 can be expressed as

```
f = cnot(x(I,0),0,1).
```

Lambda calculus allows representing and working with these kinds of expressions efficiently, so that each function and its arguments are associated to the leaves of a tree and new programs can be obtained by modifying existing trees. In this language, the previous function is expressed as

[4] https://github.com/ellisk42/ec/tree/quantum_algorithms-simplified.

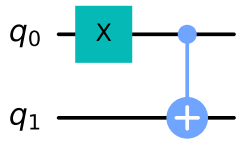


Figure A1. Example circuit.

```
(lambda (lambda (lambda (cnot (x $0 $1) $1 $2))))
```

where $0 is the initial input circuit and $i the qubit index (increased by one). For the technical description of the lambda calculus programs as trees and their advantages, we refer to the DreamCoder paper [26].

Appendix B. Probabilistic framing

In this section, we give some more details about the probabilistic framing that yields the optimization objective of our algorithm, while still referring to [26] for the full treatment. We want to find the optimal set of gates G^* and the optimal single gate probabilities $\theta^* = \{\theta_g, \forall g \in G\}$. For this purpose, we perform Bayesian inference using Bayes theorem [48]. The posterior reads

$$P(G,\theta|U) = \frac{P(U|G,\theta)\,P(G,\theta)}{P(U)}, \tag{B.1}$$

where $P(G,\theta)$ is the prior, $P(U|G,\theta)$ is the likelihood, and $P(U)$ the marginal probability of the evidence. We can consider U as constant because it does not depend on this set of gates, hence we drop the denominator from the above equation (equation (2)):

$$(G^*,\theta^*) = \arg\max_{G,\theta} P(G,\theta|U)\,. \tag{B.2}$$

By definition, we can factorize the joint distribution as $P(G,\theta) = P(G)P(\theta|G)$, and write the likelihood as

$$P(U|G,\theta) = \prod_u P(u|G,\theta)\,, \tag{B.3}$$

where $P(u|G,\theta)$ is the probability of a specific unitary u given G and θ, because of the independence between different unitary matrices in the dataset. The probability of a specific unitary can in turn be written as a function of the probability of generating it from sequences of gates in G:

$$P(u|G,\theta) = \sum_c P(u|c)\,P(c|G,\theta)\,, \tag{B.4}$$

where the sum over c is intended over all the circuits that can be generated by assembling sequences of gates in G. In our case, each unitary is deterministically induced by a circuit, thus

$$P(u|c) = \mathbb{1}\left[\mathcal{U}(c) = u\right]. \tag{B.5}$$

The probability of a circuit given a gate set can be rewritten as a function of the gate probabilities θ. Indeed, a circuit is a sequence of gates, each applied to a set of qubits. In general, the probability of the ith gate could depend on all the previous gates, thus the probability of a sequence of gates would read $P(g_0)P(g_1|g_0)P(g_2|g_1,g_0)\ldots$. In our case, we made the roughest approximation to consider these probabilities to be independent: $\prod_g \theta_g$. The probability of a circuit is thus obtained by multiplying the gate sequence probability and the multiplicity factor that takes into account the number of inputs of the gate and the number of possible qubits to connect it to. Therefore, we obtain equation (1)

$$P(c|G,\theta) = \prod_{g\in c} \mathbb{1}\left[g \in G\right] \theta_g \chi(c,g)\,. \tag{B.6}$$

If the resulting circuit is not valid, for example, if the inputs of a CNOT gate are repeated, it is automatically discarded. In this case, the χ factor would include an extra characteristic function that selects only the accepted circuits. In particular, to obtain gate sequences of variable length, when sampling we add a terminating gate to G with some probability θ_{end} and stop the sequence as soon as the gate gets extracted.

By putting it all together, we obtain the optimization objective in equation (4)

$$P(G,\theta|U) \propto P(G)\,P(\theta|G)\prod_{u\in U}\sum_{c}\mathbb{1}\left[\mathcal{U}(c)=u\right]P(c|G,\theta)\,. \tag{B.7}$$

Our program synthesis algorithm systematically enumerates expression trees sequentially in decreasing probability order. Therefore, instead of finding all the possible circuits that can produce a given unitary, we consider only the most likely k (given the current set of gates G and θ). We obtain the lower bound in equation (5)

$$P(G)\,P(\theta|G)\prod_{u\in U}\sum_{c\in\mathcal{B}_u}\mathbb{1}\left[\mathcal{U}(c)=u\right]P(c|G,\theta) \tag{B.8}$$

just because we are neglecting smaller positive terms. The maximization of this lower bound also implies the indirect maximization of the posterior.

The specific details of the enumeration algorithms are in the original DreamCoder paper [26] and in the GitHub repository. After we enumerated programs within a certain timeout, we start checking whether some of those programs can produce a circuit that is associated to one of the unitary matrices we want to decompose. At each iteration, we consider a minibatch of 25 unitaries from the whole training set and update the sets $\mathcal{B}_u$ for those unitaries.

Appendix C. Implementation details

The algorithm workflow is described in pseudocode in agorithm 1. It takes as input the set of target unitary matrices to decompose, the initial set of elementary gates to use in the assembled circuits, the batch size of each iteration, the enumeration timeout in seconds, and the number of iterations. The output is the final list of extracted gates. To run the algorithm again and test the performance, it is enough to enumerate programs again for a given timeout and check against the test set.

Algorithm 1. Unitary matrix decomposition algorithm.

Data: *target_unitaries*, *gate_library*, *batch_size*, *timeout*, *N_iterations*
Result: *gate_library*
1 for $n <$*N_iterations* **do**
2 *target_unitaries_batch* ← get_batch(*target_unitaries*, *batch_size*);
3 *enumerated_programs* ← enumerate(*timeout*, *gate_library*);
4 *solutions* ← check_solutions(*enumerated_programs, target_unitary_batch*);
5 *gate_library* ← update_library(*elementary_set, solutions*);
6 end

We ran the search in parallel on 32 Xeon Gold 6130 CPUs. Each of them explores disjoint subsets of the program space. The enumeration runs for a fixed amount of time (150s). The library building step can take a longer amount of time, according to the number of found solutions and the number of tasks in the considered batch of tasks (in our case, 25). This phase can last up to about 15 minutes and is not parallelized. The effect of checking only against a few tasks at each iteration allows speeding up the library building phase since we limit the number of unitary matrices each enumerated circuit should be tested against. Of course, an additional consequence is that more iterations are overall needed, since it takes N/N_{batch} iterations on average just to check against all the tasks. The algorithm learns to decompose new matrices, but it accounts for that only when those are selected as target tasks. As shown in figure C1, the performance that would be inferred at train time is, therefore, lower than the effective one on the train set, just because we do not check all the tasks at each iteration. The 'seen train' curve (calculated by considering only the decomposed matrices in the iteration batch) takes more time to take advantage of the discovered gates since at each iteration it is only evaluated on a subset of the total elements. The same set, but completely evaluated at each iteration of the algorithm, is shown as 'train' set (this is the same curve as in figure 3). We see that, even if the algorithm can decompose all the matrices (for example after iteration 30), it can still invent new gates to make the decomposition easier in terms of the number of used gates.

To produce plots of the circuits we use the qiskit library [49]. The complete code to run the algorithm and reproduce the experiments is available on GitHub[5].

[5] https://github.com/ellisk42/ec/tree/quantum_algorithms-simplified.

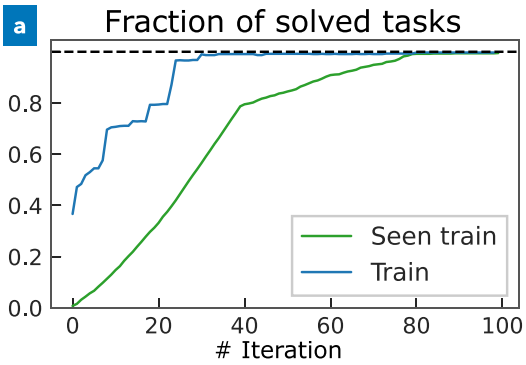


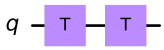
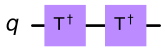
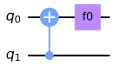
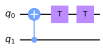
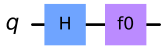
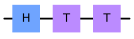
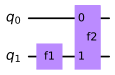
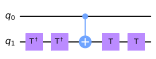
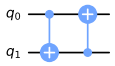
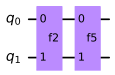
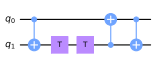
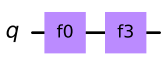
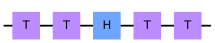
Figure C1. Estimated performance on the target set of matrices during algorithm iterations, calculated on the effectively seen matrices (in green), and on the complete set (in blue). Using batches of tasks adds some stochasticity to the algorithm (which helps to make it more robust) and speeds up the library building routine, but increases the number of required iterations.

Appendix D. Experiments

In this appendix, we show more details about the experiments we presented in the main text. In table D1, we show the list of all the gates that the algorithm extracted to solve the proposed tasks, in the case where no connectivity constraints were enforced (i.e. figure 3 in the main text).

Table D2 shows some examples of unitary matrices in our set (expressed in terms of the circuit that we used to generate the matrix) and the decompositions proposed by our algorithm. We see that the algorithm also finds unexpected ways to decompose the associated unitary matrix, which does not necessarily involve the use of exactly the same blocks we used to build it.

Table D1. List of extracted gates after 100 iterations, with no connectivity constraints between qubits.

#	Gate	Expanded circuit	Program
0			`(lambda (lambda (t (t $0 $1) $1)))`
1			`(lambda (lambda (tdg (tdg $0 $1) $1)))`
2			`(lambda (lambda (lambda (f0 $0 (cnot $1 $2 $0)))))`
3			`(lambda (lambda (f0 $0 (h $1 $0))))`
4			`(lambda (lambda (lambda (f2 $0 (f1 $1 $2) $1))))`
5			`(lambda (lambda (lambda (cnot (cnot $0 $1 $2) $2 $1))))`
6			`(lambda (lambda (lambda (f5 $0 $1 (f2 $0 $2 $1)))))`
7			`(lambda (lambda (f3 (f0 $0 $1) $0)))`

(Continued.)

Table D1. (Continued.)

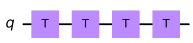
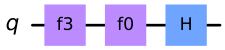
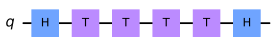
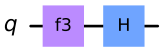
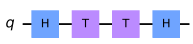
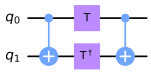
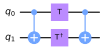
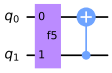
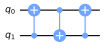
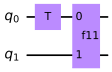
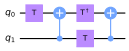
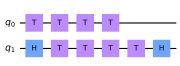
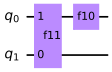
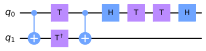
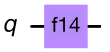
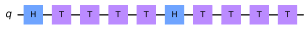
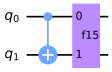
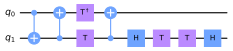
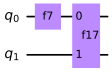
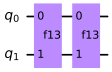
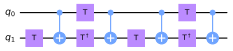
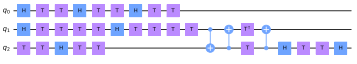
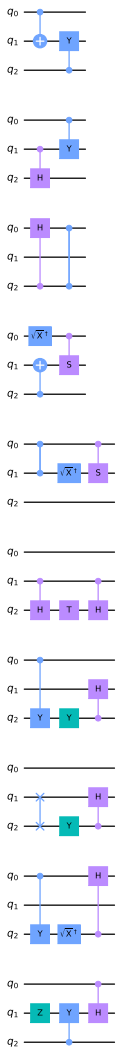
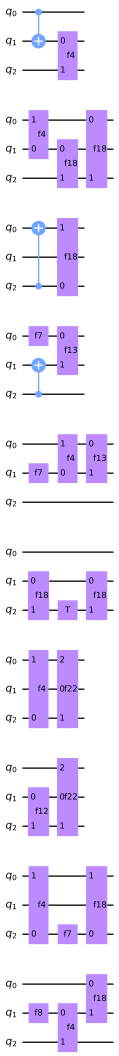
#	Gate	Expanded circuit	Program
8			`(lambda (lambda (f0 $0 (f0 $0 $1))))`
9			`(lambda (lambda (h (f0 $0 (f3 $1 $0)) $0)))`
10			`(lambda (lambda (h (f3 $0 $1) $1)))`
11			`(lambda (lambda (lambda (cnot (t (tdg (cnot $0 $1 $2) $2) $1) $1 $2))))`
12			`(lambda (lambda (lambda (cnot (f5 $0 $1 $2) $1 $0))))`
13			`(lambda (lambda (lambda (f11 $0 $1 (t $2 $0)))))`
14			`(lambda (lambda (f8 (f9 $0 $1))))`
15			`(lambda (lambda (lambda (f10 $0 (f11 $1 $0 $2)))))`
16			`(lambda (lambda (f14 $0 $1 $0)))`
17			`(lambda (lambda (lambda (f15 (cnot $0 $1 $2) $1 $2))))`
18			`(lambda (lambda (lambda (f17 $0 $1 (f7 $2 $0)))))`
19			`(lambda (lambda (lambda (f13 (f13 $0 $1 $2) $1 $2))))`
20			`(lambda (lambda (lambda (h (cnot $0 $1 $2)))))`
21			`(lambda (lambda (lambda (f8 (cnot $0 $1 $2)))))`
22			`(lambda (lambda (lambda (lambda (f7 (f10 $0 (f18 (f16 $1 $2) $2 $3)) $0)))))`

Table D2. Examples of decomposed matrices.

Target	Decomposition	Expanded decomposition
		

ORCID iDs

Leopoldo Sarra https://orcid.org/0000-0001-7504-8656
Kevin Ellis https://orcid.org/0000-0001-6586-0632
Florian Marquardt https://orcid.org/0000-0003-4566-1753

References

[1] Feynman R P 1982 *Int. J. Theor. Phys.* **21** 467–88
[2] Shor P 1994 Algorithms for quantum computation: discrete logarithms and factoring *Proc. 35th Annual Symp. on Foundations of Computer Science (Santa Fe, NM, USA)* (IEEE Computer Society Press) pp 124–34 (available at: http://ieeexplore.ieee.org/document/365700/)
[3] Grover L K 1996 A fast quantum mechanical algorithm for database search *Proc. 28th Annual ACM Symp. on Theory of Computing (STOC 1996) (Philadelphia, Pennsylvania, United States)* (ACM Press) pp 212–9 (available at: http://portal.acm.org/citation.cfm?doid=237814.237866)
[4] Peruzzo A, McClean J, Shadbolt P, Yung M-H, Zhou X-Q, Love P J, Aspuru-Guzik A and O'Brien J L 2014 *Nat. Commun.* **5** 4213
[5] Biamonte J, Wittek P, Pancotti N, Rebentrost P, Wiebe N and Lloyd S 2017 *Nature* **549** 195–202
[6] Preskill J 2021 arXiv:2106.10522
[7] Montanaro A 2016 *npj Quantum Inf.* **2** 15023
[8] Preskill J 2012 arXiv:1203.5813
[9] Aaronson S and Arkhipov A 2011 The computational complexity of linear optics *Proc. 43rd Annual ACM Symp. on Theory of Computing (San Jose California USA)* (ACM) pp 333–42 (available at: https://dl.acm.org/doi/10.1145/1993636.1993682)
[10] Arute F *et al* 2019 *Nature* **574** 505–10
[11] Zhong H-S *et al* 2020 *Science* **370** 1460–3
[12] Goodfellow I, Bengio Y and Courville A 2016 *Deep Learning* (MIT Press)

[13] Benenti G, Casati G and Strini G 2004 *Principles of Quantum Computation and Information* (World Scientific)
[14] Ostaszewski M, Grant E and Benedetti M 2021 *Quantum* **5** 391
[15] Moro L, Paris M G A, Restelli M and Prati E 2021 *Commun. Phys.* **4** 178
[16] Fösel T, Niu M Y, Marquardt F and Li L 2021 Quantum circuit optimization with deep reinforcement learning (arXiv:2103.07585)
[17] Wang P Y *et al* 2022 Automated quantum circuit design with nested Monte Carlo tree search (arXiv:2207.00132)
[18] Kitaev A Y 1997 *Russ. Math. Surv.* **52** 1191–249
[19] Dawson C and Nielsen M 2006 *Quantum Inf. Comput.* **6** 81–95
[20] Gulwani S, Polozov O and Singh R 2017 *Program Synthesis* (*Foundations and Trends in Programming Languages* (Now Publishers)
[21] Li Y *et al* 2022 *Science* **378** 1092–7
[22] Saad F A, Cusumano-Towner M F, Schaechtle U, Rinard M C and Mansinghka V K 2019 *Proc. ACM Program. Lang.* **3** 1–32
[23] Dechter E, Malmaud J, Adams R P and Tenenbaum J B 2013 Bootstrap learning via modular concept discovery *Proc. Twenty-Third Int. Joint Conf. on Artificial Intelligence* (*IJCAI 2013*) (AAAI Press) pp 1302–9 (available at: www.scopus.com/inward/record.uri?eid=2-s2.0-84896061120&partnerID=40&md5=0002fb391824bcd83d441b26339c5cef)
[24] Liang P, Jordan M I and Klein D 2010 Learning programs: a hierarchical bayesian approach *Proc. 27th Int. Conf. on Int. Conf. on Machine Learning ICML'10 (Madison, WI, USA)* (Omnipress) pp 639–46 (available at: https://icml.cc/Conferences/2010/papers/568.pdf)
[25] Ellis K *et al* 2018 Library learning for neurally-guided Bayesian program induction *Proc. 32nd Int. Conf. on Neural Information Processing Systems NIPS'18 (Red Hook, NY, USA)* (Curran Associates Inc.) pp 7816–26 (available at: https://proceedings.neurips.cc/paper_files/paper/2018/file/7aa685b3b1dc1d6780bf36f7340078c9-paper.pdf)
[26] Ellis K *et al* 2021 DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning *Proc. 42nd ACM SIGPLAN Int. Conf. on Programming Language Design and Implementation (Virtual Canada)* (ACM) pp 835–50 (available at: https://dl.acm.org/doi/10.1145/3453483.3454080)
[27] King R D *et al* 2009 *Science* **324** 85–89
[28] Arlt S, Ruiz-Gonzalez C, and Krenn M 2022 arXiv:2210.09981
[29] Cranmer M *et al* 2020 Discovering symbolic models from deep learning with inductive biases *Proc. 34th Int. Conf. on Neural Information Processing Systems NIPS'20 (Red Hook, NY, USA)* (Curran Associates Inc.)
[30] Wu T and Tegmark M 2019 *Phys. Rev.* E **100** 033311
[31] Trenkwalder L M, *et al* 2022 Automated gadget discovery in science (arXiv:2212.12743 [quant-ph])
[32] Briegel H J and De las Cuevas G 2012 *Sci. Rep.* **2** 400
[33] Wallnöfer J, Melnikov A A, Dür W and Briegel H J 2020 *PRX Quantum* **1** 010301
[34] Rissanen J 1978 *Automatica* **14** 465–71
[35] Bishop C M 2006 *Pattern Recognition and Machine Learning Information Science and Statistics* (Springer)
[36] Barke S, Peleg H and Polikarpova N 2020 *Proc. ACM Program. Lang.* **4** 1–29
[37] Fijalkow N *et al* 2022 *Proc. AAAI Conf. on Artificial Intelligence* vol 36 pp 6623–30
[38] Udupa A *et al* 2013 TRANSIT: specifying protocols with concolic snippets *Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (Seattle Washington USA)* (ACM) pp 287–96 (available at: https://dl.acm.org/doi/10.1145/2491956.2462174)
[39] Li M *et al* 2008 *An Introduction to Kolmogorov Complexity and its Applications* vol 3 (Springer)
[40] Kliuchnikov V, Maslov D and Mosca M 2012 Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates (arXiv:1206.5236)
[41] Giles B and Selinger P 2013 *Phys. Rev.* A **87** 032332
[42] Gidney C 2021 *Quantum* **5** 497
[43] Wang X, Chen Y and Zhu W 2021 *IEEE Trans. Pattern Anal. Mach. Intell.* **43** 1–1
[44] Sutton R S and Barto A G 2018 *Reinforcement Learning: An Introduction* (*Adaptive Computation and Machine Learning Series*) 2nd edn (The MIT Press)
[45] Pateria S, Subagdja B, Tan A-H and Quek C 2022 *ACM Comput. Surv.* **54** 1–35
[46] Sarra L, Ellis K and Marquardt F 2023 Discovering quantum circuit components with program synthesis (https://doi.org/10.5281/zenodo.8213680)
[47] Pierce B C 2002 *Types and Programming Languages* (MIT Press)
[48] Papoulis A and Pillai S U 2009 *Probability, Random Variables and Stochastic Processes* 4th edn (McGraw-Hill)
[49] Qiskit contributors 2023 Qiskit: an open-source framework for quantum computing (available at: https://github.com/Qiskit/qiskit/blob/main/CITATION.bib) (https://doi.org/10.5281/zenodo.2573505)