Automatic quantum function parallelization and memory management in Qrisp

Raphael Seidel

Fraunhofer Institute for Open Communication Systems, Berlin, Germany

Automated optimization of quantum programs has gathered significant attention amidst the recent advances of hardware manufacturers. In this work we introduce a novel datastructure for representing quantum programs called permeability DAG, which captures several useful properties of quantum programs across multiple levels of abstraction. Operating on this representation facilitates a variety of powerful transformations such as automatic parallelization, memory management and synthesis of uncomputation. More potential use-cases are listed in the outlook section. At the core, our representation abstracts away a class of non-trivial commutation relations, which stem from a feature called permeability. Both memory management and parallelization can be made sensitive to execution speed details of each particular quantum gate, implying our compilation methods are not only retargetable between NISQ/FT but even for individual device instances.

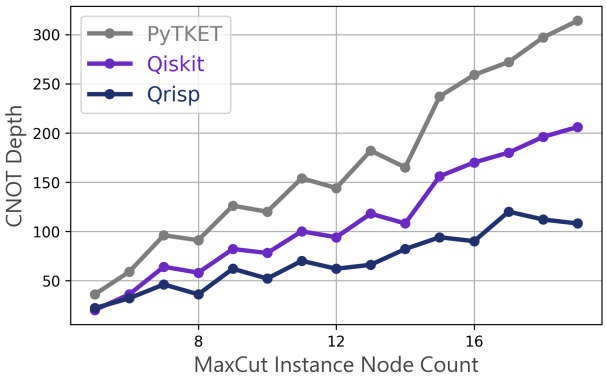


Figure 1: A plot of the resulting CNOT depth after applying several circuit optimizers (including the presented parallelization algorithm) to MaxCut problem circuits [5]. The problem instances are constructed by sampling a random Erdös–Rényi graph [6] with $p = 0.5$ for each node count. To optimize the circuits in Qiskit, we leverage the `transpile` function with optimization level 3. For PyTKET, we used the `FullPeepholeOptimise` optimization pass, which is advertised as a one fits all procedure. We note that our method works particularly well for MaxCut problems, implying general algorithms will most likely see a more moderate improvement in circuit depth.

1 Introduction

The promising advances of recent quantum hardware manufacturers [1–3] spark realistic hopes of achieving large scale fault tolerant quantum computations within less than a decade. As the scale of treatable problems grows, the implementation of more elaborate algorithms faces a variety of challenges. One of them is the lack of systematic programming abstractions, which are indispensable to all of modern software engineering (such as modular development, systematic testing or code introspection for debugging). Within a recent work [4] we gave an overview over our quantum programming framework *Qrisp*, which tackles these challenges. A core advantage of *Qrisp* is the possibility to modularize algorithm development, which is rooted in the feature of automatic quantum memory management. The present work introduces the structures and algorithms behind this feature. Additionally, we present a related algorithm, which automatically reduces the circuit runtime by establishing parallelism between quantum function calls. Both of these algorithms are based on a novel DAG representation called *permeability DAG*, which leverages nontrivial commutation relations among quantum function calls. In its essence, the permeability DAG captures information about which functions leave their inputs "constant" in a rigorously defined notion of "constant", which we call *permeability*. We show that if there are several functions operating in a permeable way on the same set of qubits, these functions can be interchanged. The permeability DAG therefore represents quantum algorithms with the mentioned commutation relations hard coded into its internal structure. As such, a much broader class of transformations is possible by operating on this representation.

A subset of the mentioned type of commutation relations have already been realized in the form of a DAG within the context of automatic synthesis of uncomputation circuits [7] ("Unqomp"). In this work we refine the Unqomp DAG to facilitate more general uncomputations and also the above mentioned compilation algorithms for memory management and parallelization.

Raphael Seidel: raphael.seidel@fokus.fraunhofer.de

2 Permeability

Permeability is a property of (composite) quantum gates, that was initially introduced by us in [8] and gives a formal meaning of what "constant" could mean for a quantum function. Even though the original purpose of the concept was restricted to automatic synthesis of uncomputations, we realized that the scope of applications is much broader. At the core, permeability information allows the compiler to leverage non-trivial commutation relations. To understand how this works in detail, we recall the definition given in [8].

Definition 2.1 (Permeability)**.** A (composite) quantum gate U is called Z-permeable in qubit i, if it commutes with the Z operator on this qubit.

$$\text{U is permeable on qubit i} \Leftrightarrow \text{UZ}_i = \text{Z}_i\text{U} \quad (1)$$

Similarly, U is called X-permeable in qubit i if it commutes with the X operator on this qubit.

For Z-permeability we gave a proof of the following theorem in [8]. For the readers convenience, it can be found in the appendix. In this work we expand the statement to X-permeability.

Theorem 1 (Commutativity theorem)**.** *Let U and V be n and m qubit operators, respectively. If U is Z-permeable (X-permeable) in its last p qubits and V is Z-permeable (X-permeable) in its first p qubits, then the two operators commute if they intersect only on these qubits:*

$$\begin{gathered}(U \otimes \mathbb{1}^{\otimes m-p})(\mathbb{1}^{\otimes n-p} \otimes V) \\ = \\ (\mathbb{1}^{\otimes n-p} \otimes V)(U \otimes \mathbb{1}^{\otimes m-p})\end{gathered} \quad (2)$$

The proof for the statement in the case of X-permeability can be found in Appendix A.
According to the results proven in [8], determining the permeability status for arbitrary gates is possible in linear time (in the unitary size) if the unitary is known. If the gate U is composite and only acting via Z-permeable (X-permeable) gates on the qubit q, Z-permeability (X-permeability) can be inferred without the unitary. Furthermore, in *Qrisp* it is possible for the programmer to annotate function definitions regarding their permeability status, implying in practice permeability information very rarely has to be inferred via unitary calculation. To highlight the prevalence of permeability features, we now list some common gates, which are permeable.

- Any RZ (RX) gate is Z-permeable (X-permeable).
- Z, P, CZ, CP, CRZ, RZZ gates are Z-permeable in all qubits.
- Any Pauli-X-gadget [9] ($U(\phi) = \exp(i\phi \otimes_{i=0}^{n} X_i)$) is X-permeable in all qubits. The same applies to Pauli-Z-gadgets.
- Global Mølmer-Sørensen gates [10] are X-permeable (or Z depending on the hardware [11]) in all qubits.
- Any phase polynomial [12] and any diagonal Hamiltonian evolution is Z-permeable in all inputs.
- Any (multi-)controlled gate is Z-permeable in any control qubit. If the base gate is Z-permeable (X-permeable), the controlled version is also Z-permeable (X-permeable) on that qubit.
- Any gate U_f that realizes an out-of-place classical function f in superposition, by acting as

$$U_f \left|x_0\right\rangle \cdots \left|x_n\right\rangle \left|0\right\rangle = \left|x_1\right\rangle \cdots \left|x_n\right\rangle \left|f(x_0, \ldots, x_n)\right\rangle \quad (3)$$

  is Z-permeable in all input qubits. This especially applies to all arithmetic functions.
- Quantum-quantum (modular) in-place adders are permeable in the input that is not operated on.

3 The permeability DAG

DAGs (directed acyclic graphs) have found manifold application in quantum compilation [13–15]. In their essence, DAGs are so useful because they can capture equivalence classes of reorderings of a given sequence. The DAG representation we construct here is based on the ideas presented in the Unqomp [7], however enriched by the concepts of Z/X-permeability. This enables the permeability DAG to express equivalence classes of quantum circuits, that are equivalent according to the commutation relations induced by Theorem 1. To leverage these, we extract circuit reorderings (with an invariant unitary) by determining a topological sort of the DAG. Topological sorting algorithms are however a wide class of techniques, which allow us to select a procedure, that favors certain characteristics of the circuit (like low circuit depth).

To elaborate the permeability DAG construction procedure, we start by introducing the types of nodes that can appear.

- **Instruction nodes** represent quantum instructions (like a CX gate).
- **Allocation nodes** represent qubit resources being allocated.

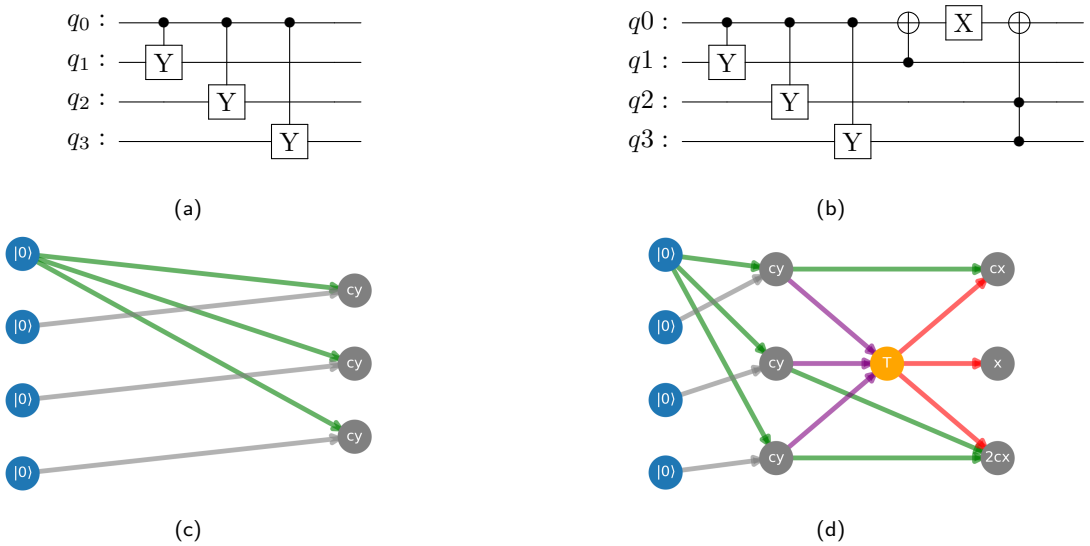


Figure 2: 2a, 2b Simple example circuits to demonstrate how the permeability DAG is built up. 2c The DAG constructed according to the rules given in Section 3. The CY gates are Z-permeable on their control, so they form a streak. Note that these gates commute according to Theorem 1, so any reordering of these gates result in a circuit with the same semantics (i.e. unitary). A reordered version of the circuit however induces the same DAG, therefore the DAG indeed represents an equivalence class of reorderings. 2d The Z-permeability streak of the CY gates is ended by applying a CX gate on q_0, which is X-permeable on that qubit. According to the DAG construction rules, the CX gate induces a terminator node connected with (purple) anti-depedency edges. The following streak of X-permeable operations will therefore be guaranteed to end up behind the Z-streak in a topological sort. For more examples please check Appendix B for a simple snippet producing pictures like these.

- **Deallocation nodes** represent qubits being deallocated. The information about which qubits can be deallocated is collected during *Qrisp* execution.
- **Terminator nodes** represent the end of a so called streak which will be elaborated shortly.

To construct a permeability DAG from a quantum circuit, we begin by adding an allocation node for each qubit in the quantum circuit. Furthemore, for every (composite) quantum gate and every deallocation we add another node to the graph. To connect the nodes we now introduce the types of edges:

- **Z-edge** (green).
- **X-edge** (red).
- **Neutral edge** (grey).
- **Anti-dependency edge** (purple).

For each instruction node, we check the permeability status on the corresponding qubits ((de)allocation nodes are treated to be neutral) and connect the edges. Connecting the edges is done according to the following rules. Let q be a qubit, that the subsequent gates U_1, U_2 are operating on.

1. If U_1 and U_2 have differing permeability status in q, a Z-edge (X-edge) is created from U_1 to U_2 if U_2 is Z-permeable (X-permeable) in q. If neither is the case, a neutral edge is used instead.
2. If U_1, U_2 are both Z-permeable (X-permeable), instead inserting a Z-edge (X-edge) from U_1 to U_2, we let the edge start in U_0, where U_0 is the node, which starts the edge to U_1 (the "parent"). A sequence of more than one gate $U_1, ..U_n$ acting on q with the same permeability is therefore connected to the same node, which we call a "streak".
3. If U_{n+1} ends the streak $U_1, ..U_n$, that is U_{n+1} has a different permeability type in q (or is neutral), a terminator node is inserted and every node in the streak receives an anti-dependency edge pointing towards the terminator node. Finally an edge representing the permeability of U_{n+1} in q is inserted pointing away from the terminator.

To get a thorough understanding, please check the example in Fig. 2. We will now discuss the motivation behind these rules. Since the DAG will undergo a topological sorting process, rule 1 enforces that two subsequent gates acting on the same qubit will appear in the correct order upon linearization. Rule 2 is designed to reflect the commutation relations from Theorem 1. Since all gates of the streak commute, they are all connected to the same initial node U_0. Therefore, any valid reordering of the streak would yield the same DAG. The anti-depedency edges in rule 3 ensure that in a linearization, the final gate ending the streak will indeed always be executed last.

3.1 Comparison to the ZX-Calculus

We see the following similarities/differences between our construction and the ZX-Calculus [16]:

- Both representations are based around certain properties of quantum gates in the Z/X-base.
- The commutativity of primitive operations expressed by the permeability DAG is a special case of the Spider Fusion Rule.
- ZX-Calculus expressions are undirected graphs and can contain loops.
- The permeability DAG contains structural information in the form of (de)allocation nodes and terminator nodes that are of a fundamentally different type compared to the instruction node.
- The permeability DAG can abstract low level implementations and represent properties of high-level quantum functions. For instance, the commutativity in the input of subsequent adders is not immediately obvious in the ZX-calculus.

3.2 Comparison to the Unqomp-Graph

The following similarities/differences are found with respect to the Unqomp DAG [7]:

- The Unqomp DAG only differentiates between control, target and anti-depedency edges. Conceptually, Z-edges can be identified with control edges, target edges are either X or neutral. Anti-depedency edges serve a similar purpose.
- With the above identifications, the permeability DAG can be used to synthesize automatic uncomputation.
- The permeability DAG contains the terminator node. The purpose of this node is mainly to separate two subsequent streaks from each other, which can't happen in the Unqomp DAG, since only the control edge type can have streaks.
- The Unqomp DAG contains allocation nodes but no deallocation nodes. For the permeability DAG, deallocation nodes play an essential role when it comes to memory management tasks.

4 Topological Sorting

Now that we have the tools to build up the permeability DAG from arbitrary quantum circuits, we can start extracting equivalent reorderings of the circuit. We do this by applying a topological sorting algorithm, which is designed in such a way that certain characteristics of the circuit are improved. In this section, we describe three such algorithms for the following purposes:

1. Automatic circuit parallelization in Section 4.1.
2. Memory management in Section 4.2.

4.1 Parallelization

The topological sorting algorithm for circuit parallelization is based on a adapted version of Kahn's algorithm [18]. For reader convenience, we describe the algorithm shortly:

Algorithm 1: Kahn's Algorithm

Input: A directed acyclic graph $G = (V, E)$
Result: A list L of nodes in topologically sorted order

Initialize the Queue;
Identify all nodes with an in-degree of 0;
Add these nodes to a queue Q;

Process the Queue;
while Q *is not empty* **do**
  node ← `Dequeue(`Q`)`;
  for *each neighbor connected by an outgoing edge from node* **do**
    Decrease the in-degree of *neighbor* by 1;
    if *in-degree of neighbor becomes 0* **then**
      `Enqueue(`Q, *neighbor*`)`;

Check for Cycles;
if *any nodes still remain in the graph with a non-zero in-degree* **then**
  Error: The graph has a cycle, topological sort not possible;

Output the Result;
The sequence of nodes removed from the queue represents the topological order of the graph;

Our hook to shaping the algorithm for our purpose lies in the dequeuing step. Since we can choose an arbitrary node from the queue, we can now deploy a heuristic for picking a node that favors circuit depth. Our strategy here is to rate all dequeuing options according to a certain cost metric $\mathcal{C}$ and choose the node with the lowest cost. We evaluated several cost metrics but focus our description on the one which produced the best results.
To describe the cost metric, we first need an additional concept:

Definition 4.1 (Dynamic Qubit Depth)**.** Given is a quantum circuit Q described by a sequence of gates $(U_i)_{0 \leq i < n}$ and a sequence of durations $(t_i)_{0 \leq i < n}$, which indicates how long each gate would take to execute on a given physical device. The **dynamic qubit depth** of the qubit k is the time $D_k(Q) \in \mathbb{R}$, which is required to execute the gate sequence until the last gate operating on qubit k.

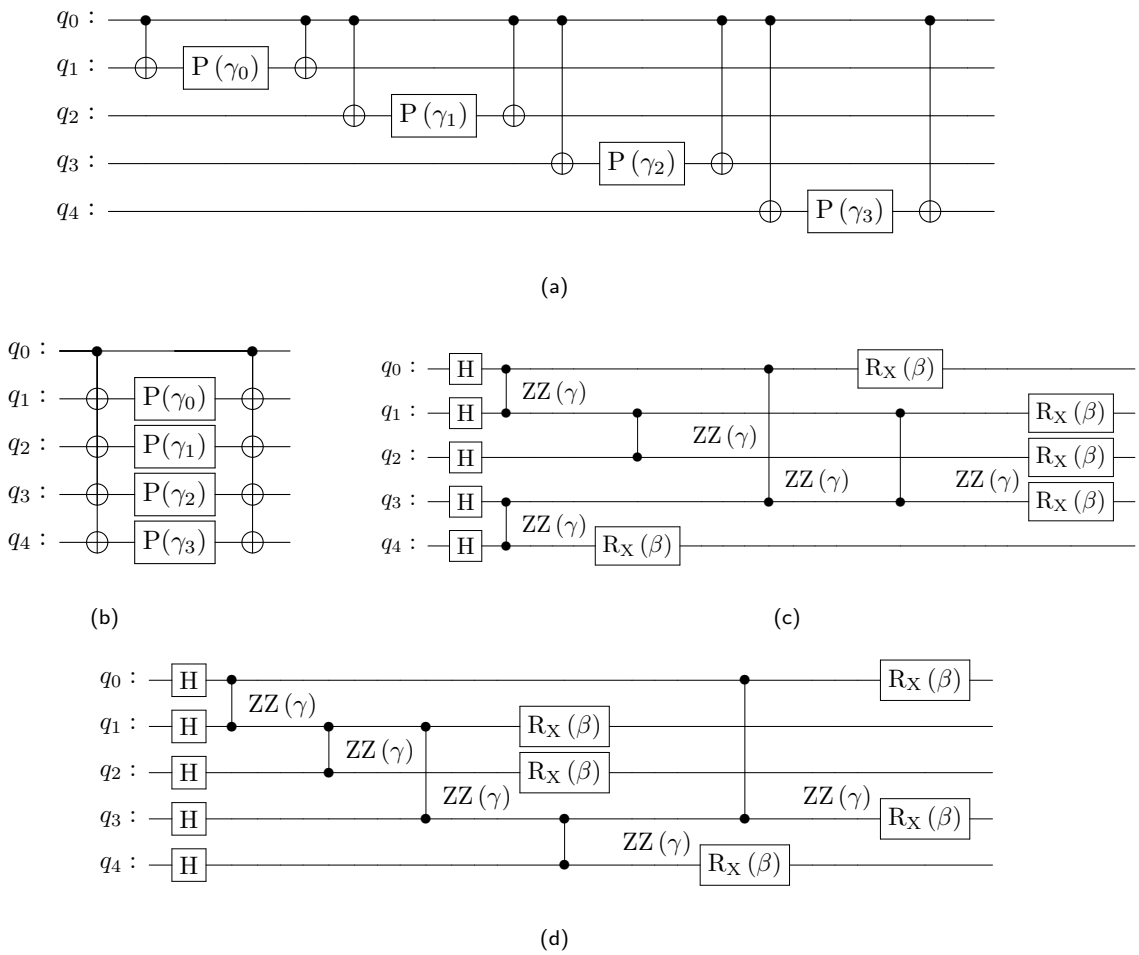


Figure 3: 3a A circuit describing a descending chain of RZZ gates, where the T-depth scales like $\mathcal{O}(n)$ in circuit size. Such circuits are a common occurence in QFT implementations or draper style adders [17]. 3b The resulting circuit after applying our parallelization procedure optimizing for T-depth, leveraging the commutation relations among the CNOT gates. The T-depth is now constant regardless of the circuit scale. 3d An example describing a $p = 1$ Max-Cut QAOA circuit. The CNOT depth is 10. 3c The QAOA circuit from 3d after applying the parallelization algorithm. The resulting CNOT depth is 6, which amounts to a 40% improvement. Note that our implementation also works for gates with unspecified parameters, since the permeability features of the RZZ gates are independent of the particular choice of parameters. In a VQA setting, our algorithm can therefore be applied once (before the optimization loop) instead of multiple times (before each iteration).

Determining the dynamic qubit depth can be achieved with a tetris-like construction and is a straightforward task. For the purpose of parallelization, the values for each qubits can be cached and reused in the next iteration, reducing classical resource requirements significantly.
Building on dynamic qubit depth allows our algorithm to optimize quantum circuits not only towards a certain device class (e.g. NISQ/FT) but even towards the particular device instances by feeding the gate-timings.
Using the above concepts, our cost function is

$$\begin{aligned}\mathcal{C}(i, t_i, Q) =& \max(D_k(Q) | k \in \mathrm{QB}(U_i)) \\ &+ t_i/(t_{max} + 1)\end{aligned} \tag{4}$$

where

- Q is the circuit that has been produced by the previous dequeuing events.
- $\mathrm{QB}(U_i)$ is the set of qubit indices that the gate U_i is operating on.
- $t_{max} = \max(t)$ is the maximum amount of time a gate can take.

We motivate the choice of these terms. Since all participating qubits of U_i need to be finished with their final gate, the first term essentially determines the time when U_i can be executed. In a situation where this time is equivalent for two separate gates, the second term makes sure that the faster gate is executed first. For an example of the effect of the parallelization procedure please refer to Fig. 3.

4.1.1 Performance analysis

The complexity of Kahn's algorithm is $\mathcal{O}(V+E)$ [18] where V is the amount of vertices and E is the number of edges. Applied to the permeability DAG, we infer that every qubit port for each gate can amount for at most two edges (one red/green/grey edge and one anti-dependency edge). Assuming that the amount of qubit ports per gate is bounded, we therefore deduce that the complexity for the parallelization procedure is $\mathcal{O}(N)$ where N is the amount of gates, which gives our method a very favorable scaling even for problem instances with a practically relevant scale.
The *Qrisp* implementation of the algorithm is based on the high performance computing framework *Numba* [19], which ensures that even for very large circuits the parallelization step can be executed with barely any delay. Please refer to Fig. 1 for a benchmark of the algorithm applied to QAOA circuits. Apart from the smallest instance our method improves the second best method by approximately 33%. Furthermore, our implementation is faster by approximately one order of magnitude compared to the second fastest implementation. For further benchmarks of the technique please check the Appendix of [4] where a Shor implementation has been optimized for T-depth.

4.2 Memory Management

This section treats the problem of reordering the operation sequence such that the amount of required Qubits is optimized. To present this feature, the section is divided into three subsections.

- In subsection 4.2.1, we elaborate how and why reordering a circuit can have influence on the amount of required qubits.
- Similar to the section on parallelization, in subsection 4.2.2, we construct a specialized algorithm for topological sorting called Flex-Sort, which caters to our requirements.
- In subsection 4.2.3, we demonstrate how Flex-Sort is applied to the permeability DAG.

4.2.1 Why order matters

Within *Qrisp* the allocation events are triggered by calling the `QuantumVariable` constructor and deallocations can be achieved by calling either the `.delete` or `.uncompute` method. On the level of the intermediate representation, (de)allocations are treated as a particular kind of operation among the regular quantum gates. To elaborate how topological sorting can help to acquire a good allocation strategy, we introduce the following concepts:

- A **algorithmic qubit** is a qubit which takes a particular role in an algorithm. Algorithmic qubits are (de)allocated once.
- An **execution qubit** is a qubit in the optimized quantum circuit and can host multiple algorithmic Qubits if their lifetime falls in distinct steps of the algorithm.

To map a sequence of gates S operating on algorithmic qubits to a quantum circuit with execution qubits, we apply the following procedure:

1. Analyze S regarding how many algorithmic qubits are allocated during peak load.
2. Create a new quantum circuit with that many execution qubits and successively insert the corresponding quantum gates. During this procedure the pool of available execution qubits is managed in the following way[1]
   (a) The **allocation** of an algorithmic qubit q_{alg} chooses an execution qubit q_{ex} from the pool according to a certain heuristic[2]. All subsequent operations on q_{alg} are executed on q_{ex} until deallocation.
   (b) The **deallocation** of an algorithmic qubit q_{alg} returns the corresponding q_{ex} to the pool.

From this construction we see that the amount of required qubits can be influenced by changing the order of allocations. In particular, if a deallocation event can be "pulled in front" of an allocation event, which would trigger peak load, the amount of required qubits decreases by one.

4.2.2 Flex-Sort

Our goal is therefore to find a reordering of the gate sequence such that deallocations are executed as early as possible and allocations are executed as late as possible. We achieve this reformulating another strategy for DAG linearization: Depth-first traversal [20].

[1] By creating a circuit with even more than the required execution qubits, it is possible to give the compiler the opportunity to choose the allocated qubit from a bigger pool of choices. This can significantly reduce circuit depth as the load can be distributed. This feature is described as "workspace" within [4].

[2] To decide about the most suitable execution qubit, we leverage the concept of dynamic qubit depth (defined earlier) to determine which of the available execution qubits would be free the earliest in an actual execution of the quantum circuit that has been compiled so far.

Algorithm 2: Depth First Topological Sort

Data: A directed acyclic graph G = (E,V)
Result: A list L of nodes in topologically sorted order
$L \leftarrow$ empty list to store the sorted nodes;
while *nodes without a permanent mark exist* **do**
  select an arbitrary unmarked node n;
  `visit`(n)
return L
Function `visit`(*node* n)**:**
  if n *has a permanent mark* **then**
    return
  if n *has a temporary mark* **then**
    stop graph contains a cycle;
  mark n with a temporary mark;
  for *each* $m \in V, (n, m) \in E$ **do**
    `visit`(m);
  mark n with a permanent mark;
  add n to head of L;

To make this algorithm useful for our purposes, we note that the `visit` function in its essence creates a linearization of the subgraph $\texttt{desc}(G, n)$[3] and inserts it at the head of the result list L. Instead of using multiple recursions of the `visit` function, we now generalize the algorithm such that an arbitrary blackbox topological sorting (TS) algorithm can be used as a "backend".

Algorithm 3: Flex-Sort

Data: A directed acyclic graph G = (E,V)
An arbitrary topological sort algorithm TS
Result: A list L of nodes in topologically sorted order
$L \leftarrow$ empty list to store the sorted nodes;
while G *contains at least one node* **do**
  select an arbitrary node n;
  $K \leftarrow$ TS(`anct`(G, n));
  extend L by K;
  remove all K from G;
return L

Note that instead of using the descendants subgraph, we used the ancestors, which is essentially the descendants of the transposed graph[4]. To make up for this deviation we **extend** the resulting list L instead of inserting the entries at the head of L. We denote this algorithm "Flex-Sort" because it gives us the following flexibility during execution:

- We can choose the backend TS.
- For each iteration, we can choose a suitable initial node n.

An obvious choice for TS is the parallelization algorithm[5], however any other topological sorting algorithm is possible too. As elaborated above, it is our goal to execute deallocation nodes as early as possible, which is why we select this node type as the prioritized starting nodes.

4.2.3 Deallocation order

But how to choose the order of deallocation nodes that suits our efforts the most? For that we investigate the DAG a bit more (prior to sorting). In particular, we generate the *ancestors* subgraph for each *deallocation* node and count how many *allocations* are required to execute that particular deallocation. The deallocation nodes are then ranked according to the amount of allocations that are required to execute them.

We summarize our procedure: Given is a sequence of operations S, a set of allocation events $A \subset S$, a set of deallocation events $D \subset S$. The following steps are taken to generate an equivalent reordering, which executes the deallocations as early as possible and the allocations as late as possible.

1. Generate the permeability DAG G of S
2. For each deallocation node d determine the ancestors subgraph $\texttt{anct}(G, d)$ and count how many allocations are contained in that subgraph. Denote this amount $|A(d)|$.
3. Create a list of deallocation nodes and sort it according to the sorting key $|A(d)|$.
4. Execute the Flex-Sort algorithm by picking the nodes of the list from the previous step as initial nodes.
5. Iterate through the sequence generated in the previous step and dynamically (de)allocate execution qubits as described in Section 4.2.1.

For an example of this procedure please consider Fig. 4. Benchmarking this algorithm is difficult since benchmark sets like [21] contain no deallocation information. In practice we could verify for a variety of examples that our algorithm finds an allocation strategy requiring the optimal amount of qubits. This is however not always the case.

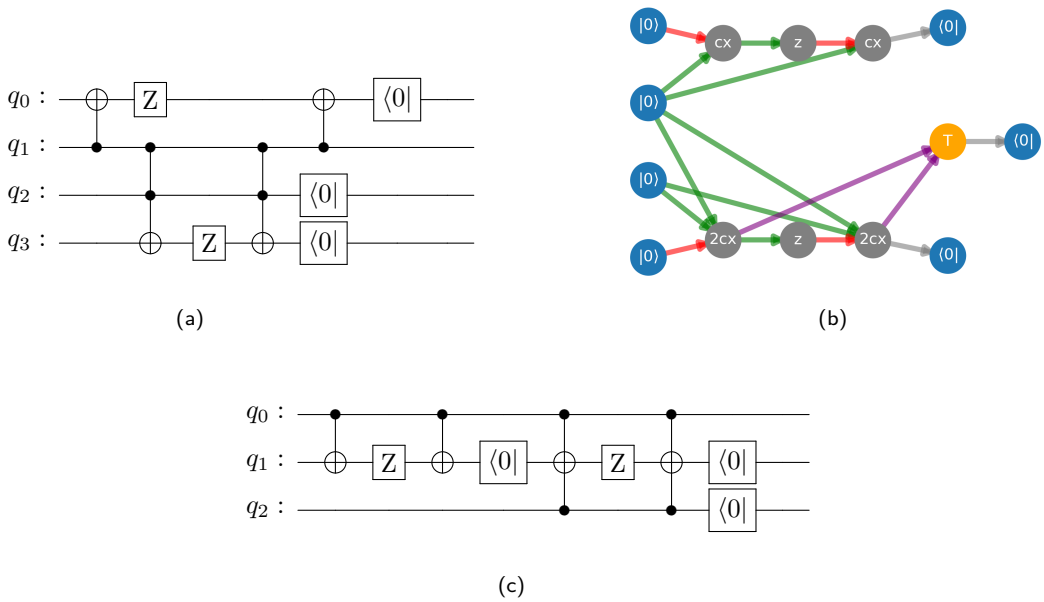
4.2.4 Performance analysis

The above described procedure for establishing a memory management strategy is more demanding than the parallelization technique. It is however not

[3] $\texttt{desc}(G, n)$ stands for the descendants subgraph of G in n and comprises all nodes that are reachable starting in n

[4] The transpose of a directed graph G is the graph that is acquired when flipping all the edges in the opposite direction.

[5] In our implementation we avoid callig the parallelization algorithm $|D|$ times. Instead we call the parallelization algorithm before memory management once and assign each node an integer label corresponding to the nodes position in the linearization. During memory management, this label is then used as a sorting key for a regular sorting algorithm to emulate a topological sort.



(a) (b)

(c)

Figure 4: 4a A simple circuit containing some deallocations (marked by the $\langle 0|$ gates). 4b The permeability DAG of 4a. Note that the ancestors of the highest deallocation node contain only two allocation nodes. The ancestors of the lower two contain three. Based on this information, our compiler chooses to perform the necessary quantum gates to perform the upper allocation as early as possible, allocating only the minimal amount of qubits. 4c The resulting circuit after applying the memory management topological sorting algorithm to the permeability DAG in 4b.

the topological sorting itself, which is more expensive[6] but instead determining the order of deallocation nodes, because this requires computing the ancestors graph for every deallocation node. In the worst case the complexity is therefore $\mathcal{O}(|S| \cdot |D|)$. However this task can be parallelized (one thread for each deallocation node). Within *Qrisp* this bottleneck is remedied by a parallel Numba [19] kernel, which performs our technique with barely any delay for most mid-term quantum algorithms. Additionally, the ancestors function is available in the cuGraph framework [22], which allows to outsource this task to clusters of GPU hardware, implying even circuits with billions of gates might be treatable.

5 Outlook

Even though the present article describes the permeability DAG in terms of parallelization and memory management, more use-cases are possible:

1. **Light-cone reduction**. Viewing the permeability DAG as a representation of causally related events, it is possible to filter out gates which do not influence measurement results. In a different words: It doesn't matter whether a gate outside of the light-cone of a measurement is executed. We can use this fact to optimize quantum circuits in that we remove every node from the permeability DAG that is outside of the light-cone of the intended measurements.

2. **Peephole optimizations**. The permeability DAG can help with identifying peephole optimizations that require adjacency. One example of this is the simple gate sequence (Z, S, Z). The Z gates cancel out but only because the sequence can be arbitrarily reordered due to permeability features. Less trivial examples could include cancellation/merging of CX/CP gates or even higher level semantics such as fusing two quantum adders.

3. **Dynamic parallelization**. For circuits that contain very long Z-permeable streaks[7] it is possible to "quasi-copy" the value of the streak qubit and therefore enable the access to the streak qubit value via multiple qubits, facilitating an improved circuit depth. Quasi-copy means duplicating a computational basis state by applying a CX gate into a freshly allocated qubit.

Apart from the mentioned potential use-cases, further questions regarding the permeability DAG remain open:
Within Section 6 of [4], we describe our efforts to restructure *Qrisp* to remedy compilation speed bottlenecks but also to enable seamless hybrid algorithm compilation. For this we leverage the Jax [23]

[6] This is because the overall amount of edges can only be lower if the graph is cut into several ancestor subgraphs.

[7] An example for this phenomenon is the controlled call of a composite quantum function. Every subroutine of this function is individually controlled such that control qubit access is a bottleneck.

framework, which exposes an interface for an extensible dynamic intermediate representation called *Jaxpr*. Quantum algorithms expressed through Jaxprs can not only contain classical control flow (such as loops or conditionals) but also purely classical functions. As such we are faced with the question how these Jaxpr features can be embedded into the permeability DAG framework to facilitate the described compilation algorithms also for hybrid settings.

6 Summary

Within the present work we gave a rigorous definition of the concepts of Z/X-permeability, including a proof that permeable gates commute if they only intersect on permeable inputs. To make use of the induced commutation relations, we constructed the permeability DAG, which resembles the DAG defined in the Unqomp algorithm [7]. Next to the synthesis of uncomputation, the permeability DAG can be used to find equivalent reorderings of the circuit by applying a topological sorting algorithm. We describe two such sorting algorithms, which are constructed to optimize certain features of the resulting quantum circuit. The first algorithm parallelizes quantum function calls to reduce the depth of the overall circuit, whereas the second method reorders the program instructions to reduce peak quantum memory consumption. For both of these algorithms we gave a rough complexity analysis. Our implementation in the *Qrisp* framework shows that for near/mid-term quantum algorithms, the introduced delay of the procedures is negligible. Finally, we elaborated on further applications of the permeability DAG and discussed its implementation based on a more dynamic IR.

Code availability

Qrisp is an open-source Python framework for high-level programming of quantum computers. The source code is available in https://github.com/eclipse-qrisp/Qrisp.

References

[1] Dolev Bluvstein, Simon J. Evered, and Alexandra A. et al. Geim. "Logical quantum processor based on reconfigurable atom arrays". Nature **626**, 58–65 (2023).

[2] M. P. da Silva, C. Ryan-Anderson, and J. M. Bello-Rivas et al. "Demonstration of logical qubits and repeated error correction with better-than-physical error rates" (2024). arXiv:2404.02280.

[3] C. M. Löschnauer, J. Mosca Toba, and A. C. Hugheset et al. "Scalable, high-fidelity all-electronic control of trapped-ion qubits" (2024). arXiv:2407.07694.

[4] Raphael Seidel, Sebastian Bock, and René Zander et al. "Qrisp: A framework for compilable high-level programming of gate-based quantum computers" (2024). arXiv:2406.14792.

[5] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. "A quantum approximate optimization algorithm" (2014). arXiv:1411.4028.

[6] E. N. Gilbert. "Random Graphs". The Annals of Mathematical Statistics **30**, 1141 – 1144 (1959).

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

[8] Raphael Seidel, Nikolay Tcholtchev, and Sebastian et al. Bock. "Uncomputation in the Qrisp high-level quantum programming framework". Page 150–165. Springer Nature Switzerland. (2023).

[9] Alexander Cowtan, Silas Dilkes, and Ross et al. Duncan. "Phase gadget synthesis for shallow circuits". Electronic Proceedings in Theoretical Computer Science **318**, 213–228 (2020).

[10] Dmitri Maslov and Yunseong Nam. "Use of global interactions in efficient quantum circuit constructions". New Journal of Physics **20**, 033018 (2018).

[11] Sabine Wölk and Christof Wunderlich. "Quantum dynamics of trapped ions in a dynamic field gradient using dressed states". New Journal of Physics **19**, 083021 (2017).

[12] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. "On the controlled-not complexity of controlled-not–phase circuits". Quantum Science and Technology **4**, 015002 (2018).

[13] Joseph Clark, Travis Humble, and Himanshu Thapliyal. "TDAG: Tree-based directed acyclic graph partitioning for quantum circuits". In Proceedings of the Great Lakes Symposium on VLSI 2023. Page 587–592. GLSVLSI '23New York, NY, USA (2023). Association for Computing Machinery.

[14] Panagiotis Promponas, Akrit Mudvari, and Luca Della Chiesa et al. "Compiler for distributed quantum computing: a reinforcement learning approach" (2024). arXiv:2404.17077.

[15] Giulia Meuli, Mathias Soeken, and Giovanni Micheli. "Xor-and-inverter graphs for quantum compilation". npj Quantum Information**8** (2022).

[16] Bob Coecke and Ross Duncan. "Interacting quantum observables: categorical algebra and diagrammatics". New Journal of Physics **13**, 043016 (2011).

[17] Thomas G. Draper. "Addition on a quantum computer" (2000). arXiv:quant-ph/0008033.

[18] A. B. Kahn. "Topological sorting of large networks". Commun. ACM **5**, 558–562 (1962).

[19] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based python JIT compiler". In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. LLVM '15New York, NY, USA (2015). Association for Computing Machinery.

[20] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. et al. Rivest. "Introduction to algorithms, third edition". The MIT Press. (2009). 3rd edition.

[21] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. "MQT Bench: Benchmarking software and design automation tools for quantum computing". Quantum (2023).

[22] Seunghwa Kang, Joseph Nke, and Brad Rees. "Analyzing multi-trillion edge graphs on large gpu clusters: A case study with pagerank". In 2022 IEEE High Performance Extreme Computing Conference (HPEC). Pages 1–7. (2022).

[23] Roy Frostig, Matthew Johnson, and Chris Leary. "Compiling machine learning programs via high-level tracing". In SYSML'18, Stanford, CA USA. (2018). url: https://mlsys.org/Conferences/doc/2018/146.pdf.

Appendix

A Proof of Theorem 1

This appendix contains the proofs for Theorem 1 from Section 2. In order to prove Theorem 1, we first need the following theorem.

Theorem 2. *Let $U \in U(2^n)$ be an n-qubit operator. If U is Z-permeable on the first p qubits, there are operators $\tilde{U}_0, \tilde{U}_1, \ldots, \tilde{U}_{2^p-1}$ such that:*

$$U = \sum_{i=0}^{2^p-1} |i\rangle\langle i| \otimes \tilde{U}_i. \tag{5}$$

Proof. We will treat the case $p = 1$ first and generalize via induction afterwards.
We start by inserting identity operators $\mathbb{1} = \sum_{i=0} |i\rangle\langle i|$:

$$\begin{aligned} U =& \mathbb{1}U\mathbb{1} && (6)\\ =& \sum_{i,j=0}^{1} |i\rangle\langle i|U|j\rangle\langle j| && (7)\\ =& \sum_{i,j=0}^{1} |i\rangle\langle j| \otimes \hat{U}_{ij}. && (8)\end{aligned}$$

where $\hat{U}_{ij} = \langle i|U|j\rangle$. Due to the Z-permeability condition, we have

$$\begin{aligned} 0 =& Z_0U - UZ_0\\ =& \left(\sum_{k=0}^{1}(-1)^k |k\rangle\langle k| \otimes \mathbb{1}^{\otimes n-1}\right)\left(\sum_{i,j=0}^{1} |i\rangle\langle j| \otimes \hat{U}_{ij}\right)\\ &- \left(\sum_{i,j=0}^{1} |i\rangle\langle j| \otimes \hat{U}_{ij}\right)\left(\sum_{k=0}^{1}(-1)^k |k\rangle\langle k| \otimes \mathbb{1}^{\otimes n-1}\right)\\ =& \sum_{i,j,k=0}^{1} (-1)^k \left(|k\rangle\langle k|i\rangle\langle j| \otimes \hat{U}_{ij} - |i\rangle\langle j|k\rangle\langle k| \otimes \hat{U}_{ij}\right)\\ =& \sum_{i,j,k=0}^{1} (-1)^k \left(|k\rangle\langle k|i\rangle\langle j| - |i\rangle\langle j|k\rangle\langle k|\right) \otimes \hat{U}_{ij}\\ =& \sum_{i,j=0}^{1} \left((-1)^i |i\rangle\langle j| - (-1)^j |i\rangle\langle j|\right) \otimes \hat{U}_{ij}. \end{aligned} \tag{9}$$

From this form, we see that the index constellations, where $i = j$ cancel out. We end up with

$$0 = 2(|0\rangle\langle 1| \otimes \hat{U}_{01} - |1\rangle\langle 0| \otimes \hat{U}_{10}). \tag{10}$$

Since both summands act on disjoint subspaces, we conclude

$$\hat{U}_{01} = 0 = \hat{U}_{10}. \tag{11}$$

Finally, we set

$$\begin{aligned} \tilde{U}_0 &= \hat{U}_{00}\\ \tilde{U}_1 &= \hat{U}_{11} \end{aligned} \tag{12}$$

yielding the claim for p = 1. To complete the proof we give the induction step, that is, we prove the claim for $p = p_0 + 1$ under the assumption that it is true for $p = p_0$: Since U is permeable on qubit $p_0 + 1$, we have

$$0 = Z_{p_0+1}U - UZ_{p_0+1} \tag{13}$$

As the claim is true for $p = p_0$, we insert

$$U = \sum_{i=0}^{2^{p_0}-1} |i\rangle\langle i| \otimes \tilde{U}_i \tag{14}$$

yielding

$$0 = \sum_{i=0}^{2^{p_0}-1} |i\rangle\langle i| \otimes (Z_{p_0+1}\tilde{U}_i - Z_{p_0+1}\tilde{U}_i) \tag{15}$$

Since each of the summand operators acts on disjoint subspaces, we conclude

$$0 = Z_{p_0+1}\tilde{U}_i - Z_{p_0+1}\tilde{U}_i \tag{16}$$

This, as shown above, implies

$$\tilde{U}_i = \sum_{j=0}^{1} |j\rangle\langle j| \otimes (\tilde{U}_i)_j. \tag{17}$$

Finally, we insert this form into eq. 14 to retrieve the claim for $p = p_0 + 1$:

$$U = \sum_{i=0}^{2^{p_0+1}-1} |i\rangle\langle i| \otimes \tilde{U}_i \tag{18}$$

□

Having proved the above theorem, the next step is to employ it in the proof of Theorem 1 for Z-permeability.

Proof. According to Theorem 2 we can write

$$(U \otimes \mathbb{1}^{\otimes m-p}) = \sum_{i=0}^{2^p-1} \tilde{U}_i \otimes |i\rangle\langle i| \otimes \mathbb{1}^{\otimes m-p} \tag{19}$$

$$(\mathbb{1}^{\otimes n-p} \otimes V) = \sum_{j=0}^{2^p-1} \mathbb{1}^{\otimes n-p} \otimes |j\rangle\langle j| \otimes \tilde{V}_j \tag{20}$$

Multiplying these operators gives

$$\begin{aligned}
&(U \otimes \mathbb{1}^{\otimes m-p})(\mathbb{1}^{\otimes n-p} \otimes V) \\
&= \left(\sum_{i=0}^{2^p-1} \tilde{U}_i \otimes |i\rangle\langle i| \otimes \mathbb{1}^{\otimes m-p}\right) \\
&\quad \left(\sum_{j=0}^{2^p-1} \mathbb{1}^{\otimes n-p} \otimes |j\rangle\langle j| \otimes \tilde{V}_j\right) \\
&= \sum_{i,j=0}^{2^p-1} \tilde{U}_i \otimes |i\rangle\langle i|j\rangle\langle j| \otimes \tilde{V}_j \\
&= \sum_{i=0}^{2^p-1} \tilde{U}_i \otimes |i\rangle\langle i| \otimes \tilde{V}_i
\end{aligned} \tag{21}$$

Multiplication in reverse order yields the same result:

$$\begin{aligned}
&(\mathbb{1}^{\otimes n-p} \otimes V)(U \otimes \mathbb{1}^{\otimes m-p}) \\
&= \left(\sum_{j=0}^{2^p-1} \mathbb{1}^{\otimes n-p} \otimes |j\rangle\langle j| \otimes \tilde{V}_j\right) \\
&\quad \left(\sum_{i=0}^{2^p-1} \tilde{U}_i \otimes |i\rangle\langle i| \otimes \mathbb{1}^{\otimes m-p}\right) \\
&= \sum_{i,j=0}^{2^p-1} \tilde{U}_i \otimes |j\rangle\langle j|i\rangle\langle i| \otimes \tilde{V}_j \\
&= \sum_{i=0}^{2^p-1} \tilde{U}_i \otimes |i\rangle\langle i| \otimes \tilde{V}_i
\end{aligned} \tag{22}$$

From this we conclude the claim. □

Finally, we generalize the proof from just Z-permeability to X-permeability.

Proof. Let U, V satisfy the given conditions for X-permeability. We define two auxiliary operators

$$\tilde{U} = H_{\geq n-p} U H_{\geq n-p} \tag{23}$$

$$\tilde{V} = H_{<p} V H_{<p} \tag{24}$$

The notation $H_{<p}$ here implies that Hadamard gates are applied to all qubits with an index $< p$. We observe that $\tilde{U}, \tilde{V}$ are both Z-permeable on the relevant qubits. To see this, let $0 \leq k < p$:

$$\begin{aligned}
Z_k\tilde{V} &= H_k X_k H_k \tilde{V} \\
&= H_k X_k H_{<p,\neq k} V H_{<p} \\
&= H_k H_{<p,\neq k} X_k V H_{<p} \\
&= H_{<p} V X_k H_k H_{<p,\neq k} \\
&= H_{<p} V H_{<p} H_k X_k H_k \\
&= H_{<p} V H_{<p} Z_k \\
&= \tilde{V} Z_k
\end{aligned} \tag{25}$$

Obviously, a similar reasoning holds vor $\tilde{U}$. Using the Z-permeability commutativity theorem, we deduct that $\tilde{U}, \tilde{V}$ commute if they only intersect on the relevant qubits.

$$\begin{aligned}
&(\tilde{U} \otimes \mathbb{1}^{\otimes m-p})(\mathbb{1}^{\otimes n-p} \otimes \tilde{V}) \\
&= \\
&(\mathbb{1}^{\otimes n-p} \otimes \tilde{V})(\tilde{U} \otimes \mathbb{1}^{\otimes m-p})
\end{aligned} \tag{26}$$

By wrapping both sides of the equation into Hadamard gates, we obtain the statement:

$$\begin{aligned}
&H_{\geq n-p,<n}(\tilde{U} \otimes \mathbb{1}^{\otimes m-p})(\mathbb{1}^{\otimes n-p} \otimes \tilde{V})H_{\geq n-p,<n} \\
&= \\
&H_{\geq n-p,<n}(\mathbb{1}^{\otimes n-p} \otimes \tilde{V})(\tilde{U} \otimes \mathbb{1}^{\otimes m-p})H_{\geq n-p,<n} \\
&\Leftrightarrow \\
&(U \otimes \mathbb{1}^{\otimes m-p})(\mathbb{1}^{\otimes n-p} \otimes V) \\
&= \\
&(\mathbb{1}^{\otimes n-p} \otimes V)(U \otimes \mathbb{1}^{\otimes m-p})
\end{aligned} \tag{27}$$

□

B Permeability Graph Code

In the following, we provide some *Qrisp* code to reproduce or alter the plots in Fig. 2.

```
from qrisp import *
qc = QuantumCircuit(4)

# Z-permeable streak
qc.cy(0,1)
qc.cy(0,2)
qc.cy(0,3)

# X-permeable streak
qc.cx(1,0)
qc.x(0)
qc.mcx([2,3],0)

dag = PermeabilityGraph(qc)
dag.draw()
```

As a subclass of `networkx.DiGraph`, the permeability graph object can be processed by a variety of Networkx algorithms.