Efficient Classical Simulation of Low-Rank-Width Quantum Circuits Using ZX-Calculus

Fedor Kuyanov and Aleks Kissinger

In this paper, we introduce an algorithm for contracting (i.e. numerically evaluating) ZX-diagrams whose computational complexity is governed by rank-width, a structural graph parameter that behaves nicely under ZX rewrite rules. Our method evaluates a graph-like ZX-diagram in $\tilde{O}(4^R)$ time, given its rank-decomposition of width R. By constructing suitable rank-decompositions, we provide a classical simulation algorithm for quantum circuits whose runtime is bounded by both $\tilde{O}(2^n)$ and $\tilde{O}(2^{t/2})$, where n is the number of qubits and t is the number of non-Clifford phase gates. Thus, our method is no slower than either the naïve state-vector simulation or the elementary stabiliser decomposition baseline, and in practice can be even faster when the reduced ZX-diagram has low rank-width. We benchmark our simulation algorithm against Quimb, a popular tensor contraction library, and observe substantial reductions in floating-point operations, often by several orders of magnitude, for random and structured non-Clifford circuits and for random ZX-diagrams.

1 Introduction

Classical simulation of quantum circuits has a variety of applications in designing, verifying, and benchmarking quantum computations. Furthermore, many techniques are transferable to related structures such as tensor networks, which can be applied to solve interesting problems in their own right, e.g. in constraint satisfaction or many-body physics. While it is widely believed that classical simulation of quantum circuits is computationally hard in general, improvements in classical simulation algorithms can significantly affect the feasibility of solving specific problems. For example, a good choice of classical simulation algorithm can make the difference between projected simulation time of 10,000 years [4] and 5 days [27].

While certain families of quantum circuits, such as Clifford circuits, are known to be classically simulable in polynomial time, all known algorithms for simulating computationally universal quantum circuits scale exponentially with respect to some metric of the circuit. However, different techniques rely on different metrics. For example, naïve state-vector techniques require space that grows exponentially with the number of qubits, matrix product state simulations grow with the amount of entanglement introduced by the circuit [33] (which typically grows with the circuit depth), sum-over-Clifford methods (aka stabiliser decompositions) grow exponentially in the number of non-Clifford gates [8, 18, 20], and sum-over-path methods grow exponentially in the number of non-diagonal gates [2].

Tensor network techniques, which generalise many of these methods, represent a quantum circuit (or a more general multilinear map) as a graph of smaller collections of complex numbers called *tensors* which can be composed, or *contracted*, by summing over shared indices. Markov and Shi showed that tensor contraction scales exponentially in the treewidth of the tensor network graph [22], so tensor networks that are relatively close to trees can be contracted efficiently. More general tensor networks are highly sensitive to the order in which smaller tensors are contracted together to form larger ones, and finding optimal contraction orders is known to be NP-hard. Various state-of-the-art tensor contraction techniques employ sophisticated heuristics to find graph decompositions that minimise treewidth or related metrics, such as edge/vertex congestion [15].

To appear in EPTCS.

The technique we introduce in this paper is similar in spirit to tensor network contraction methods, in that we start with a special kind of tensor network, namely a *ZX-diagram*, and show that computing the scalar value of a closed ZX-diagram scales with respect to a particular metric of the associated graph, namely its *rank-width* [25]. Unlike treewidth, rank-width can be small even for highly connected graphs: for example, the fully connected graph on n nodes has treewidth $n-1$ but rank-width 1.

The key observation is that, unlike for arbitrary tensor networks, the amount of entanglement across any bipartition of nodes in a ZX-diagram scales with the cut-rank of that bipartition. This property was first noted for graph states [16], which are closely related to ZX-diagrams (see e.g. [13]), and can be made explicit by simplifying ZX-diagrams using a set of graphical rewrite rules called the *ZX-calculus* [11, 19]. Most of the ZX-calculus rules simplify ZX-diagrams without increasing the rank-width, thus one can exploit low-rank connectivity to reduce the number of edges between graph components. Using this idea, we provide a tensor contraction routine that evaluates a graph-like ZX-diagram in $\tilde{O}(4^R)$ time, given a rank-decomposition of width R. We furthermore show that, for ZX-diagrams arising from quantum circuits, our routine's complexity is bounded by $\tilde{O}(2^n)$, where n is the number of qubits, and also by $\tilde{O}(2^{t/2})$, where t is the number of non-Clifford phase gates, for suitably chosen rank-decompositions. This matches both the scaling of the naïve state-vector simulation and the elementary stabiliser decomposition baseline. Our tensor contraction routine is implemented in PyZX.[1]

As in the case of treewidth, finding decompositions of minimal rank-width is NP-hard. So, we first introduce our methods for constructing rank-decompositions – see Section 3. We then introduce the ZX-diagram contraction algorithm in Section 4. Finally, in Section 5, we benchmark our methods against the naïve tensor contraction routine provided by the ZX-calculus library PyZX [17], as well as Quimb [15], a state-of-the-art tensor contraction tool. We show substantial reductions in floating-point operations for random CNOT+H+T circuits, two sets of structured non-Clifford circuits, and random ZX-diagrams.

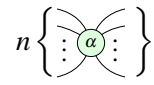
During the preparation of this paper, we became aware of independent work by Codsi and Laakkonen making use of treewidth, rank-width, and stabiliser decompositions to do classical simulation of ZX-diagrams [10]. Drawing explicit comparisons between the two approaches and/or combining them is a topic of future work.

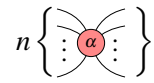
2 Preliminaries

2.1 ZX-calculus

ZX-calculus [19] is a graphical formalism for representing quantum circuits as a specific type of tensor networks, called *ZX-diagrams*. A ZX-diagram consists of building blocks called *spiders*, and there are only two types of them: Z- and X- spiders.

Definition 2.1 (Z- and X-spiders)**.**

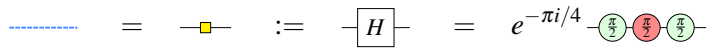
Z-spider: n  m $:=$ $\underbrace{|0\ldots0\rangle}_{m}\underbrace{\langle 0\ldots0|}_{n}+e^{i\alpha}\underbrace{|1\ldots1\rangle}_{m}\underbrace{\langle 1\ldots1|}_{n}$,

X-spider: n  m $:=$ $\underbrace{|+\cdots+\rangle}_{m}\underbrace{\langle +\cdots+|}_{n}+e^{i\alpha}\underbrace{|-\cdots-\rangle}_{m}\underbrace{\langle -\cdots-|}_{n}$.

The phase α is omitted when $\alpha=0$. The orthonormal bases $\{|0\rangle,|1\rangle\}$ and $\{|+\rangle,|-\rangle\}$ are the eigenbases of the Pauli Z and X matrices, respectively.

[1] https://github.com/zxcalc/pyzx/blob/5f5e409/pyzx/rank_width.py#L567

Hadamard gates interchange the Z- and X-bases and often appear in ZX-diagrams. Hence, it is convenient to introduce some special notation for them, either as a small yellow box or a dashed edge called the *Hadamard edge*:



(1)

Every quantum circuit can be compiled into a ZX-diagram, that is, a tensor network consisting of Z/X spiders and regular/Hadamard edges. Note that measurement in the standard basis corresponds to a post-selection with a single-legged X-spider having a parametrised phase $a\pi$, representing the measurement outcome $a \in \{0,1\}$. Then, by the Born rule, the probability of measuring a equals the absolute value squared of the value of the ZX-diagram.

ZX-calculus offers rewrite rules [19, Figure 3.1] which can be used for automated circuit simplification and simulation. For instance, they allow us to simulate Clifford circuits in polynomial time, providing an alternative proof for the Gottesman-Knill theorem [1]. Notably, every ZX-diagram can be efficiently transformed to a *graph-like form* [19, Definition 5.1.7], in which there are only green spiders and Hadamard edges. Using two additional rewrite rules derivable from the ZX-calculus ruleset, called *local complementation* and *pivoting* [19, Lemmas 5.2.9 and 5.2.11], one can further eliminate some Clifford spiders from the graph-like ZX-diagram and obtain a *reduced ZX-diagram* [19, Section 7.6], which in the case of scalar ZX-diagrams consists only of internal non-Clifford spiders and *phase gadgets* – structures at the top of Figure 1. Note that the reduced ZX-diagram can become arbitrarily dense; thus, conventional treewidth-based simulators are inefficient in this case.

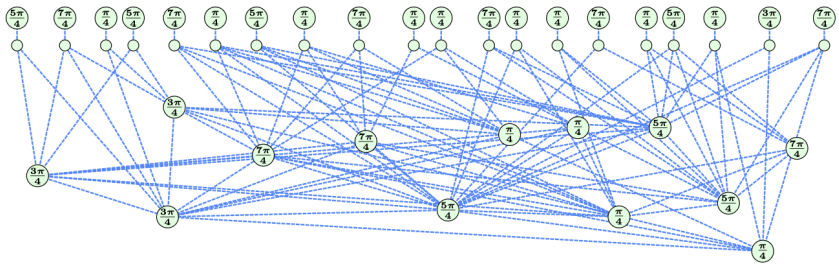


Figure 1: Example of a reduced Clifford+T ZX-diagram.

We also need the notion of parity maps [19, Chapter 4]. For the two-element finite field $\mathbb{F}_2$, consider an $\mathbb{F}_2$-linear map $\phi\colon \mathbb{F}_2^n \to \mathbb{F}_2^m$, mapping the basis states $|x_1 \dots x_n\rangle$ to the basis states $|y_1 \dots y_m\rangle$, where $x_i, y_j \in \{0,1\}$. We are interested in a graphical representation of ϕ in the ZX formalism. Let $A_\phi \in \mathrm{Mat}_{n\times m}(\mathbb{F}_2)$ be the matrix of ϕ acting on row vectors of $\mathbb{F}_2^n$. Consider a graph-like ZX-diagram whose spiders form a bipartite graph with biadjacency matrix A_ϕ, that is, the i-th spider on the left is connected to the j-th spider on the right iff $(A_\phi)_{ij} = 1$ (see Figure 2). One can verify this diagram corresponds to ϕ by plugging in the basis states $|x_1 \dots x_n\rangle$, which are single-legged red spiders with phases $x_1\pi, \dots, x_n\pi$, and applying the standard ZX-calculus rules such as state-copy, spider fusion, and colour change [19, Chapter 3]. Suppose we now have two parity maps $\phi\colon \mathbb{F}_2^n \to \mathbb{F}_2^m$ and $\psi\colon \mathbb{F}_2^m \to \mathbb{F}_2^k$, given by the matrices A_ϕ and A_ψ, respectively. Recall that the parity matrix of the composition is the product of the parity matrices: $A_{\psi\circ\phi} = A_\phi A_\psi$. This identity can be expressed graphically as in Figure 3.

2.2 GFlow

We use the notion of *extended gflow* to bound the rank-width of certain ZX-diagrams. In general, extended gflow provides sufficient conditions under which a measurement pattern on a graph state, with

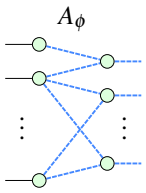


Figure 2: a parity map

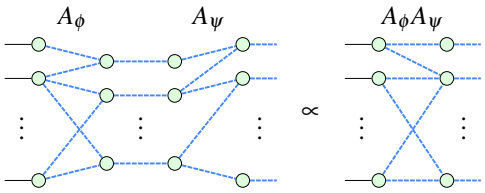


Figure 3: composition of parity maps

single-qubit measurements in the three Pauli planes, can be executed deterministically. Although originally developed for the one-way model of *measurement-based quantum computing* (MBQC) [9], it has since found other applications, most notably in performing efficient circuit extraction from ZX-diagrams that have been simplified using the rules of ZX-calculus [5, 31]. In particular, extended gflow always exists for reduced ZX-diagrams originating from quantum circuits, and there is a polynomial-time algorithm for finding it [5, Theorems 3.11 and 4.21].

Extended gflow is defined on an *open graph*, that is, a triple (G,I,O) where $G=(V,E)$ is an undirected graph, and $I,O\subset V$ are respectively called input and output vertices. Extended gflow consists of a *correction function* g associating each non-output vertex v with its correction set (in MBQC, it is used to handle the measurement outcomes of v), a strict partial order $\prec$ over V denoting the vertex measurement order ($i\prec j$ if i is measured before j), and a *labelling function* λ assigning to each non-output vertex one of the three Pauli planes (XY, YZ, or XZ).

Definition 2.2 (Extended gflow)**.** *Given an open graph (G,I,O), an* extended gflow *is a triple $(g,\prec,\lambda)$, where $g\colon V(G)\setminus O\to\mathscr{P}(V(G)\setminus I)\setminus\{\varnothing\}$, $\prec$ is a strict partial order over $V(G)$, $\lambda\colon V(G)\setminus O\to\{\mathrm{XY},\mathrm{XZ},\mathrm{YZ}\}$, which satisfy the following conditions:*

1. *if $j\in g(i)$ then $i\preceq j$,*
2. *if $j\in\mathrm{Odd}(g(i))$ then $i\preceq j$,*
3. *if $\lambda(i)=\mathrm{XY}$ then $i\notin g(i)$ and $i\in\mathrm{Odd}(g(i))$,*
4. *if $\lambda(i)=\mathrm{XZ}$ then $i\in g(i)$ and $i\in\mathrm{Odd}(g(i))$,*
5. *if $\lambda(i)=\mathrm{YZ}$ then $i\in g(i)$ and $i\notin\mathrm{Odd}(g(i))$,*

where $\mathrm{Odd}(S):=\{u\in V(G)\colon |N(u)\cap S|\equiv 1 \pmod 2\}$.

2.3 Rank-width

Rank-width [25] is a graph measure that captures the behaviour of its cut-ranks over $\mathbb{F}_2$. In particular, rank-width is the lowest possible width of the graph's *rank-decomposition*, which is defined as a recursive partitioning process of its vertices (called *branch-decomposition*) along with cut-ranks assigned to each edge of the decomposition tree. Rank-width is closely related to other structural graph parameters such as clique-width [12] and treewidth [30]. Before defining rank-width, we must introduce the following notions of *branch-decomposition* and *cut-rank*.

Definition 2.3. *Let M be a finite set, called the* ground set *of the decomposition. A* branch-decomposition *is a pair (T,L), where T is a tree with each non-leaf node having degree 3, and L is the bijection from M to the leaves of T.*

Definition 2.4. *Given an undirected graph $G=(V,E)$ and a subset $X\subset V$, the* cut-rank $\rho_G(X)$ *is the rank of the adjacency matrix between X and $V\setminus X$, viewed as a matrix over $\mathbb{F}_2$.*

Combining the two notions above, we get the definition of the *rank-decomposition* and *rank-width*. Refer to Figure 4 for illustration.

Definition 2.5. *Let $G = (V, E)$ be an undirected graph. A* rank-decomposition *is a branch-decomposition with the ground set V, where each edge (u, v) of the tree is assigned the cut-rank $\rho_{u,v} := \rho_G(X)$, where X is the set of leaves in the same component as u after deleting (u, v). The* width *of the rank-decomposition is the maximum $\rho_{u,v}$ over all edges (u, v).*

Definition 2.6. *The* rank-width *of an undirected graph G, written as* $\mathrm{rw}(G)$*, is the minimum width of its rank-decomposition.*

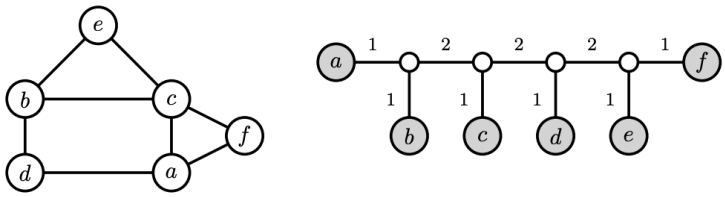


Figure 4: [24, Figure 3.2] Optimal rank-decomposition of G, showing $\mathrm{rw}(G) = 2$.

In fact, the rank-decomposition presented in Figure 4 has a special property: its cut-ranks are taken over the partitions that exhibit a linear structure. This allows for the following simplified notion of *linear rank-decomposition*.

Definition 2.7. *A rank-decomposition is called* linear *if the non-leaf nodes of its branch-decomposition form a path graph. Consequently, a linear rank-decomposition is given by a permutation $(p_i)_{i=1,\dots,|V|}$ of the graph vertices, with the cut-ranks r_i taken between $\{p_1, \dots, p_i\}$ and $\{p_{i+1}, \dots, p_{|V|}\}$ for each $i \in [1; |V| - 1]$.*

Let us discuss some hardness results related to rank-width. In [26], it is shown that computing the rank-width of a given graph is NP-hard, and the decision problem of determining whether the rank-width does not exceed k is NP-complete. So, one can only hope to compute the rank-width approximately. In [6], there is an attempt to calculate upper and lower bounds for rank-width. The algorithm for the upper bound iteratively improves the decomposition using "a mix of greedy and random decisions". Another approach [21, 24] produces the upper bound via simulated annealing. In this paper, we present our contribution to this problem in Section 3. In particular, we construct a rank-decomposition from gflow with an explicit rank-width upper bound and introduce several heuristics that produce good rank-decompositions in practice.

3 Constructing rank-decompositions

In this section, we present our methods for generating rank-decompositions of a graph G. Refer to Appendix A for more details regarding proofs, implementation, and complexity analysis. Our first method, "rw-flow", generates a linear rank-decomposition from the extended gflow with an explicit rank-width upper bound. This upper bound helps us to show that our simulation algorithm, presented in Section 4, matches the scaling of the naïve state-vector simulation. We proceed to describe other methods that do not yet have rigorous performance guarantees. The first heuristic, "rw-greedy-linear", generates a linear rank-decomposition by greedily appending vertices to it. The second heuristic, "rw-greedy-b2t", on the other hand, produces highly nonlinear structures by greedily constructing the decomposition tree from the bottom up.

3.1 rw-flow

In the context of this method only, we assume that G is an open graph with $|I| = |O| = n$. Our method starts by finding an extended gflow of G, that is, a tuple $(g, \prec, \lambda)$ where $\prec$ is the partial order and $\lambda(v) \in \{\mathrm{XY}, \mathrm{XZ}, \mathrm{YZ}\}$. This step takes polynomial time by [5, Theorem 3.11]. Next, we compute a linear extension of $\prec$, i.e., a permutation of vertices $(p_i)_{i=1,\ldots,|V(G)|}$ such that for each $i < j$ we have $p_i \not\succ p_j$. We can do this by topological sorting of the directed graph induced by $\prec$. Then, p constitutes the desired linear rank-decomposition of width at most n according to Lemma 3.1 below. The proof of Lemma 3.1 is given in Appendix A.1.

Lemma 3.1. *Let (G, I, O) be an open graph, $(g, \prec, \lambda)$ be its extended gflow, and let $(p_i)_{i=1,\ldots,|V(G)|}$ be a permutation of vertices satisfying $p_i \not\succ p_j$ for each $i < j$. Then, p constitutes a linear rank-decomposition of G, whose width does not exceed $|O|$.*

3.2 rw-greedy-linear

Our greedy method for generating linear rank-decompositions is inspired by the Goemans-Soto technique for minimising symmetric submodular functions [14], based on the earlier Queyranne's work [29]. These works establish that a certain greedy algorithm correctly minimises a symmetric submodular function defined on subsets. Our method is a simplified version of their algorithm applied to the function $f\colon S \subset V \mapsto \rho_G(S) - \lambda|S|$ for some $\lambda \in \mathbb{R}$, which is submodular by [26, Eq. 2].

Given a simple undirected graph $G = (V, E)$, our heuristic generates a linear rank-decomposition $(p_i)_{i=1,\ldots,|V|}$ by iteratively appending vertices to it. We start from an empty p. For each $i = 1, \ldots, |V|$ we pick p_i using the following rule. If the biadjacency matrix M between $S := \{p_1, \ldots, p_{i-1}\}$ and $V \setminus S$ is zero, then we set p_i to an arbitrary vertex from $V \setminus S$. Otherwise, we iterate over all pivot columns u of M and set $p_i := \operatorname{argmin}_u \rho_G(S \cup \{u\})$. Recall that a pivot column of M is a column such that in the row-echelon form of M, it contains the leftmost non-zero entry in some row. Refer to Algorithm A.1 for the pseudocode of this heuristic.

The choice of limiting our search to the pivot columns of M was not arbitrary. As verified by our experiments on numerous examples, it not only reduces the complexity of the algorithm but also significantly improves the quality of the rank-decomposition. A rigorous theoretical explanation for this phenomenon remains open.

3.3 rw-greedy-b2t

Another heuristic is inspired by the "Hyper-Greedy" contraction tree optimiser [15]. In contrast to previous methods, it produces generic rank-decompositions by constructing the decomposition tree from its leaves to the root. We maintain a forest of partial decompositions $T = (V_T, E_T)$ and a partition of graph vertices $\{S_i\}_{i \in \mathrm{roots}(T)}$ corresponding to the set of leaves of each tree in T. We start from a collection of subtrees having a single leaf: $V_T = V$, $E_T = \varnothing$, and $S_v = \{v\}$ for each $v \in V$. Our method iteratively merges two trees in T by adding a new vertex to T as their common ancestor. As in the previous case, we consider a subset of *valid* root pairs as candidates for this operation. Namely, a pair (i, j) of roots in T is considered valid if S_i intersects the pivot columns of the biadjacency matrix between S_j and $V \setminus S_j$. On each iteration, pick a valid pair (i, j) minimising $\rho_G(S_i \cup S_j)$ (if there are no valid pairs, pick an arbitrary one). Merge S_i and S_j by adding to T a new vertex k with children i, j, setting $S_k := S_i \cup S_j$, and discarding S_i, S_j (since i, j are no longer roots). Repeat this procedure $|V| - 1$ times, so that only one tree remains, corresponding to the desired rank-decomposition tree. Refer to Algorithm A.2 for the pseudocode of this algorithm.

4 Classical simulation

We focus on computing a single measurement amplitude $\langle y|C|x\rangle$ for a quantum circuit C, which is an important subroutine in most classical simulation algorithms. Solving this problem suffices to perform *strong simulation* of quantum circuits, that is, computing arbitrary measurement probabilities and marginal probabilities (the latter by means of norm computations, a.k.a. "doubling"). We can also use amplitude computations to perform weak simulation, i.e. sampling individual measurement outcomes, via techniques such as gate-by-gate circuit sampling [7].

Generally, computing measurement amplitudes in polynomial time is infeasible, as it is just as hard as contracting an arbitrary tensor network, which is known to be #P-complete [23]. In this section, we present an algorithm for contracting ZX-diagrams with complexity $\tilde{O}(4^R)$, where R is the width of a rank-decomposition of the reduced ZX-diagram. Additionally, if R is the width of a linear rank-decomposition, then our algorithm runs in $\tilde{O}(2^R)$ time (Corollary 4.2). In the case of quantum circuits, our method's complexity is bounded by $\tilde{O}(2^n)$ for a certain rank-decomposition (Corollary 4.3). Furthermore, for circuits consisting of t non-Clifford phase gates, we can construct a rank-decomposition that yields the simulation complexity of $\tilde{O}(2^{t/2})$ (Corollary 4.4). Our method is summarised by Algorithm 4.1.

Algorithm 4.1 Rank-width–based classical simulation of a quantum circuit C

1. Convert C into a ZX-diagram and simplify it as discussed in Section 2.1, to obtain a reduced ZX-diagram D.
2. Construct a rank-decomposition T of D.
3. Run the tensor contraction routine from Proposition 4.1 along T.

The first step is standard for ZX-based simulation methods (e.g. stabiliser decompositions [18, 20, 28, 32]). We assume the circuit's inputs and outputs are plugged in so that D is closed, except when we run the "rw-flow" routine in step 2, which works only for open graphs (as the extended gflow does not exist for closed ZX-diagrams). In practice, we may use all the techniques from Section 3 and choose the best rank-decomposition with respect to $\mathrm{flops}(T)$, defined below, which is a measure similar to rank-width characterising the cost of our tensor contraction routine.

4.1 Tensor contraction routine

Proposition 4.1. *Let $D = (G, \alpha)$ be a graph-like ZX-diagram with no inputs and outputs, where $G = (V, E)$ is an undirected graph and $\alpha_v \in [0; 2\pi)$ denotes the phase of the vertex v. Let $T = (V_T, E_T)$ be a rank-decomposition of G with cut-ranks $\rho_{u,v}$. Then, there exists an algorithm for contracting D with complexity $\tilde{O}(\mathrm{flops}(T))$, where*

$$\mathrm{flops}(T) := \sum_{v \in V_T} 2^{w_v}, \quad w_v := \begin{cases} \min(\rho_{v,u_1} + \rho_{v,u_2}, \rho_{v,u_1} + \rho_{v,u_3}, \rho_{v,u_2} + \rho_{v,u_3}) & \text{for } N_T(v) = \{u_1, u_2, u_3\}, \\ 0 & \text{for } |N_T(v)| = 1. \end{cases}$$

Proof. First, we *root* T by deleting an arbitrary edge $(u_0, v_0) \in E_T$, orienting all remaining edges towards u_0 and v_0, adding a dummy vertex 0 as a parent of u_0 and v_0, and setting the cut-ranks $\rho_{u_0,0} = \rho_{v_0,0} := \rho_{u_0,v_0}$. For each vertex u in the rooted decomposition tree, consider the set of leaves S_u from its subtree. Denote $r_u := \rho_G(S_u)$. Note that for each $u \neq 0$ we have $r_u = \rho_{u,p}$, where p is the parent of u, and also $r_0 = 0$.

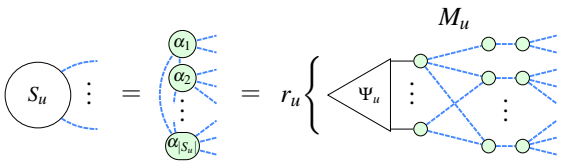


Figure 5: Conventions for our simulation routine. Ψ_u is a r_u-qubit state, and M_u is a parity matrix of size $r_u \times |S_u|$. The rightmost edges (between S_u and $V \setminus S_u$) are unchanged.

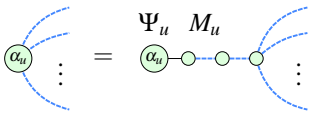


Figure 6: Simulation of $D|_{S_u}$: leaf case

Our contraction routine is a recursive procedure that takes a vertex u of T and "simulates" the subdiagram $D|_{S_u}$. Namely, it returns the state Ψ_u and the parity matrix M_u satisfying the equation in Figure 5. The state Ψ_u could be thought of as an "encoded version" of the subdiagram $D|_{S_u}$, which, in composition with the "decoder" M_u, would give the actual graph-like state induced by the spiders and edges of D in S_u (which has $|S_u|$ outputs). Note that our algorithm below never computes this state, as it would take $2^{\Omega(|V|)}$ time; instead, we work exclusively with the encoded versions, which takes only $2^{O(R)}$ time. To contract D, we call our procedure for $u = 0$ and return Ψ_0, which is a scalar since $r_0 = 0$.

Our recursive procedure works as follows. If u is a leaf of T, then Ψ_u is just a single-legged spider and M_u is an identity matrix – see Figure 6. Otherwise, we follow the course of reasoning presented in Figure 7:

Step 1. Denote by v, w the children of u in T, and let A, B be the biadjacency matrices defined by the pairs of vertex sets $(S_u, V \setminus S_u)$ and (S_v, S_w), respectively. Note that $S_u = S_v \sqcup S_w$.

Step 2. Recursively simulate $D|_{S_v}, D|_{S_w}$ to obtain Ψ_v, Ψ_w, M_v, M_w.

Step 3. B now corresponds to a layer of CZ gates after the parity map defined by $M_v \oplus M_w$. We can shift this to a layer of CZ gates before the parity map given by $E_{vw} := M_v B M_w^T$.

Step 4. Find $|S_u| \times r_u, r_u \times |S_u|$ matrices U, V such that $A = UVA$. This is possible because A has rank r_u, as guaranteed by the rank-decomposition. That is, we can write $A = PA$, where P is the projector onto the image of A. We can then split the projector P as a pair of matrices U, V such that $P = UV$ and $VU = I_{r_u}$.

Step 5. Let U_v be the first $|S_v|$ rows of U, and let U_w be the last $|S_w|$ rows. Composing the parity maps again, compute $E_{vu} := M_v U_v$ and $E_{wu} := M_w U_w$.

Step 6. Finally, perform the *convolution*, further detailed below, to contract the left half of the diagram and obtain Ψ_u. Setting $M_u := V$, we obtain Ψ_u, M_u in the required form for the recursion at **Step 2**. If u is the root, then Ψ_u is just a scalar, so we return it as the result.

The convolution subroutine contracts the tripartite ZX-diagram as shown in Eq. (2). Recall that Ψ_v, Ψ_w are r_v- and r_w-qubit states, respectively, and the output is a r_u-qubit state $\widehat{\Psi}_u$ (which relates to Ψ_u via Hadamard transform). The parity matrices E_{vu}, E_{wu}, E_{vw} denote the edges between the three parts. We show that the convolution can be performed in $\tilde{O}\left(2^{\min(r_v+r_w, r_u+r_v, r_u+r_w)}\right)$ time using the cheapest of three equivalent methods described below. Since steps 1, 3, 4, and 5 take polynomial time, by induction over u it will follow that the overall complexity of our simulation routine is $\tilde{O}(\text{flops}(T))$.

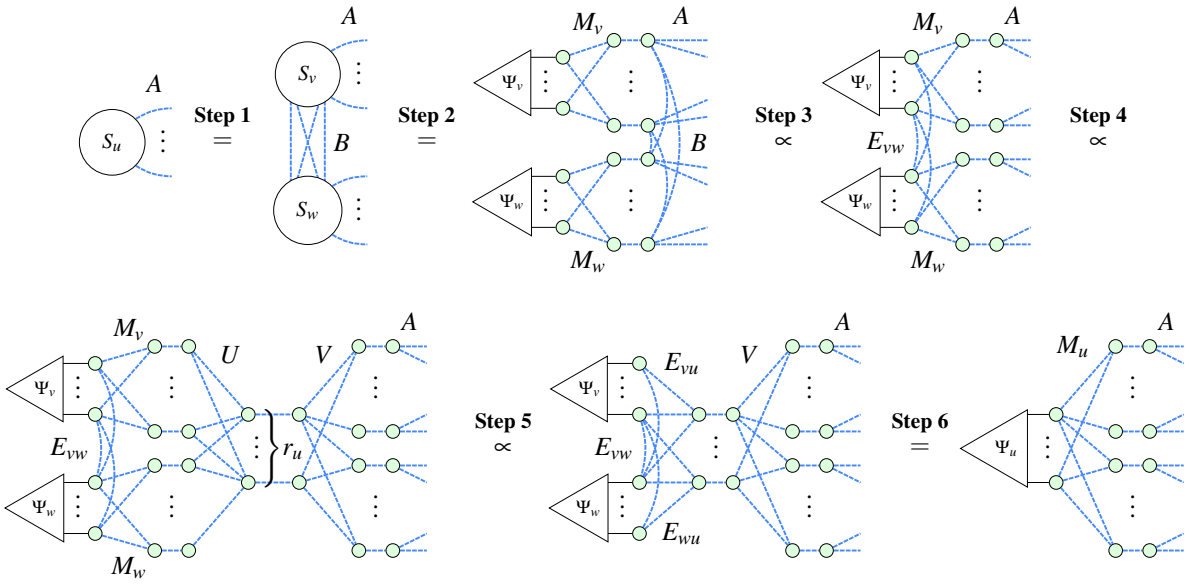
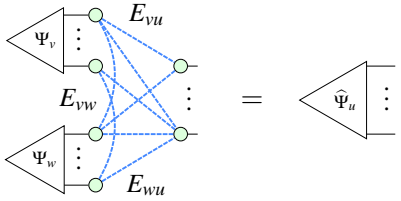


Figure 7: Simulation of $D|_{S_u}$: general case



(2)

The first approach for the convolution is summarised in Figure 8. We start by computing $\Psi_v \otimes \Psi_w$. Next, we apply the diagonal CZ-only operator corresponding to the E_{vw}-edges. Finally, we apply the parity map (E_{vu}, E_{wu}). Note that all these steps take $\tilde{O}(2^{r_v+r_w})$ time, since the inequality $r_u \leq r_v + r_w$ follows from the submodular inequalities for cut-ranks [26, Eq. 2].

The second method, summarised in Figure 9, first rotates the Ψ_v state (so that it becomes a post-selection) and then unfuses the E_{vu}-edges. We start by applying the parity map (E_{vw}^T, E_{wu}) to Ψ_w. Next, we apply the diagonal CZ-only operator corresponding to the E_{vu}-edges. Finally, we perform the post-selection with Ψ_v. Note that all these steps take $\tilde{O}(2^{r_u+r_v})$ time, since $r_w \leq r_u + r_v$ by [26, Eq. 2].

The third method with complexity $\tilde{O}(2^{r_u+r_w})$ is just the second one with Ψ_v and Ψ_w interchanged.

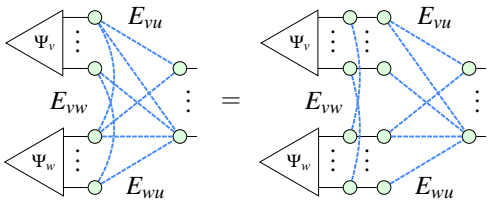


Figure 8: Convolution with complexity $\tilde{O}(2^{r_v+r_w})$

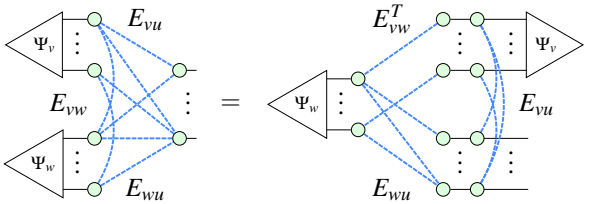


Figure 9: Convolution with complexity $\tilde{O}(2^{r_u+r_v})$

□

Corollary 4.1. *Under the assumptions from Proposition 4.1, the simulation complexity can be upper bounded by $\tilde{O}(4^R)$, where R is the width of the rank-decomposition T.*

Proof. Note that we have $w_v \leq 2R$ for each $v \in V_T$. Hence, $\text{flops}(T) \leq |V| \cdot 2^{2R} = \tilde{O}(4^R)$. □

Corollary 4.2. *Under the assumptions from Proposition 4.1, if additionally T is linear, then the simulation complexity can be upper bounded by $\tilde{O}(2^R)$, where R is the width of T.*

Proof. Since T is linear, we have $w_v \leq R+1$. Hence, $\text{flops}(T) \leq |V| \cdot 2^{R+1} = \tilde{O}(2^R)$. □

Corollary 4.3. *Algorithm 4.1 simulates an n-qubit quantum circuit in $\tilde{O}(2^n)$ time for a certain efficiently constructible decomposition T.*

Proof. The reduced ZX-diagram of a quantum circuit is computable in polynomial time; see e.g. [19, Chapter 7]. Let $(p_i)_{i=1,\dots,|V(D)|}$ denote the linear rank-decomposition generated by "rw-flow", which also takes polynomial time; see Section 3.1. By Lemma 3.1, the width of p does not exceed n. The complexity of Algorithm 4.1 is then dominated by the last step, which costs $\tilde{O}(2^n)$ by Corollary 4.2. □

Corollary 4.4. *Let C be a quantum circuit consisting of Clifford gates and t non-Clifford phase gates. Then, Algorithm 4.1 simulates C in $\tilde{O}(2^{t/2})$ time for a certain efficiently constructible decomposition T.*

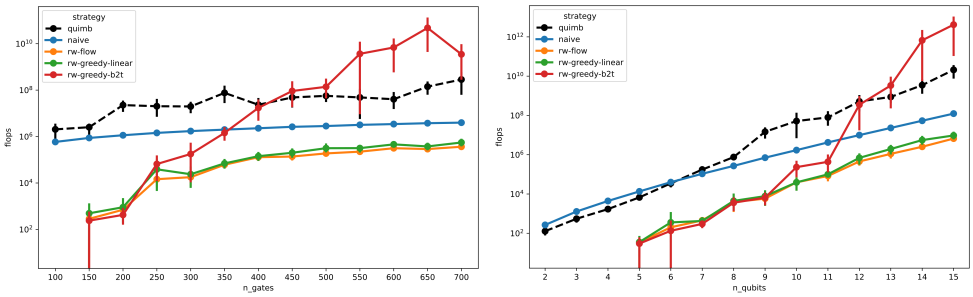
Proof. First, note that the ZX-diagram of C has t non-Clifford spiders. There are $t' \leq t$ non-Clifford spiders in its reduced ZX-diagram D because the ZX simplifications can only eliminate them. Since D is closed, each Clifford spider is a part of a phase gadget. Consider an arbitrary linear rank-decomposition $(p_i)_{i=1,\dots,t'}$ of the internal non-Clifford spiders and the phase gadgets (treated as single nodes). It extends to a (nonlinear) rank-decomposition T of D by adding two children to each phase gadget node. Note that the width of (p_i) is at most $t'/2$, since its cut-ranks r_i satisfy $r_i \leq \min(i, t'-i)$. Thus, the width of T is also bounded by $t'/2$. A direct estimate of $\text{flops}(T)$, analogous to the proof of Corollary 4.2, shows that Algorithm 4.1 simulates D in $\tilde{O}(2^{t'/2})$ time. □

5 Benchmarks

We benchmark our tensor contraction routine from Section 4.1 with different rank-decomposition strategies from Section 3 on certain families of quantum circuits and ZX-diagrams. The pipeline is essentially the same as in Algorithm 4.1, with one slight modification: instead of running the contraction routine directly, we estimate the number of *flops* – floating-point operations required for the contractions. This number is given by a formula similar to that of Proposition 4.1, with each term multiplied by a certain linear combination of the cut-ranks. We also estimate the number of flops for the naïve statevector-like simulation routine `tensorfy` provided by the PyZX library. As a baseline, we use the Quimb library, which contains analogous tensor contraction routines for hypergraphs, for which the number of flops can also be measured in advance. Similar to the ZX-based methods, Quimb first simplifies the given tensor network. Then it runs an optimiser to find an appropriate sequence of tensor contractions (aka a contraction tree). We use the high-quality "auto-hq" optimiser, which takes a longer time to converge on larger instances but aims to produce efficient contraction trees. Both `tensorfy` and Quimb routines are run on the original (non-reduced) ZX-diagram. We have also verified the correctness of all the compared routines on the instances below, where the number of flops is not too large.

5.1 Random CNOT + H + T circuits

We generate a family of random CNOT + H + T circuits by iteratively appending CNOT, Hadamard, and T gates with probabilities 0.6, 0.2, and 0.2, respectively. First, we fix the qubit count to 10 and vary the number of gates (Figure 10a). Then, we set the gate count to $3 \cdot \text{n\_qubits}^2$ and vary the number of qubits (Figure 10b). This scaling was chosen to ensure circuits do not become too sparse as the qubit count is increased, trivialising the classical simulation task. For each configuration of $(\text{n\_qubits}, \text{n\_gates})$, the results were averaged over five independent runs.



(a) for $\text{n\_qubits} = 10$ (b) for $\text{n\_gates} = 3 \cdot \text{n\_qubits}^2$

Figure 10: Benchmark on random CNOT + H + T circuits

When the number of qubits is fixed, the naive simulation appears to outperform Quimb by a constant factor of 10 on almost all instances. Our routines are very efficient for the shallow circuits since their T-count is small. For the deeper ones, "rw-flow" and "rw-greedy-linear" level off at the 100-times-faster mark compared to Quimb, but "rw-greedy-b2t" starts to slow down rapidly. This is likely due to the large cut-ranks appearing at the top of the rank-decomposition tree, since "rw-greedy-b2t" does not take them into account while building the decomposition from the bottom up.

When the number of qubits is varied, we observe that each simulation routine slows down exponentially. This is because for sufficiently dense random circuits, their rank-width appears to be close to the number of qubits. Thus, rank-width–based methods do not constitute a significant improvement over the naive method in this case.

5.2 Structured circuits

We run our routines on a set of structured circuits taken from the PyZX repository[2], which are mainly based on the T-Par [3] benchmark circuits[3]. Since the majority of circuits represent a classical computation, we set the inputs and outputs of these circuits to the T-states. Figure 11 illustrates the performance of our methods relative to the baseline.

On every circuit in the dataset, our methods in combination perform similarly to or faster than Quimb. In most cases, "rw-greedy-b2t" performs best, and in some cases, it is up to 10,000 times faster than Quimb. The "rw-flow" routine complements "rw-greedy-b2t": it is around the baseline when the greedy routine performs poorly, and is around 10 times slower than the baseline when the greedy routine dominates. This benchmark demonstrates that a good choice of a rank-decomposition can significantly impact

[2] `https://github.com/zxcalc/pyzx`
[3] `https://github.com/meamy/t-par`

the performance of our contraction routine, and the combination of "rw-greedy-b2t" and "rw-flow" is a robust choice in these benchmarks.

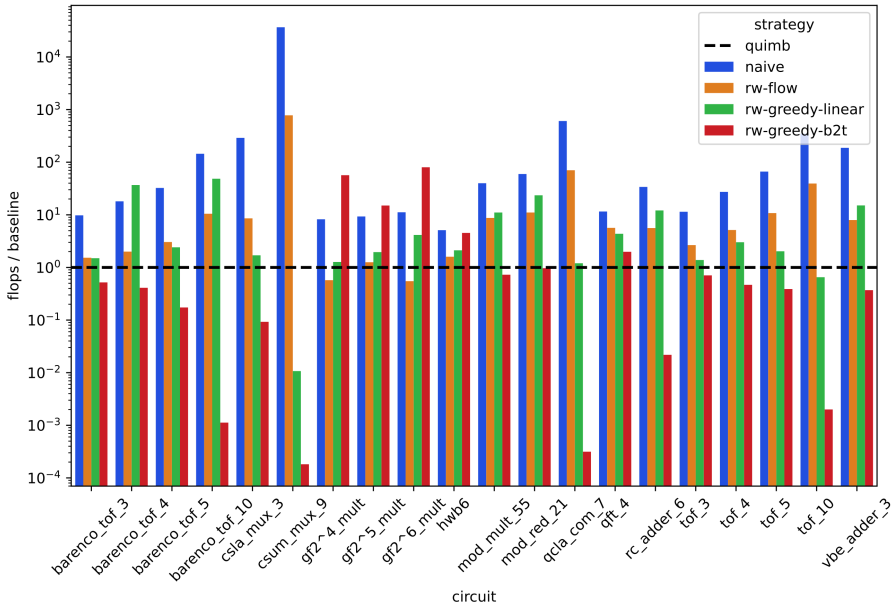


Figure 11: Benchmark on structured circuits from the PyZX repository

5.3 Multi-qubit Toffoli

The benchmark above motivates us to consider the family of circuits implementing an n-qubit Toffoli gate (also known as the multi-controlled NOT gate). It is well-known that these circuits have stabiliser rank 2 and are thus efficiently simulable. As before, we set the circuit inputs and outputs to T-states to avoid total simplification. The results are shown in Figure 12.

Notably, the performance of "rw-greedy-b2t" appears to scale polynomially with n, while other methods result in exponential scaling. This is quite unexpected and suggests the existence of a contraction-based polynomial-time simulation algorithm in this setup. An explicit description of such an algorithm could be interesting in its own right, and we leave this for future work.

5.4 Random ZX-diagrams

We simulate Erdős-Rényi random ZX-diagrams for different edge probabilities p (Figure 13). In particular, these ZX-diagrams are generated by creating n green spiders with $\pi/4$ phases and adding Hadamard edges between each pair of them with probability p. Since these ZX-diagrams do not originate from a circuit and thus have no causal flow/gflow, we can only run "rw-greedy-linear" and "rw-greedy-b2t" on these instances.

As the density of the ZX-diagram increases, Quimb's flops approach 2^n. Notably, the performance of our rank-width–based methods is symmetric with respect to p. This is because cut-ranks are invariant under edge complement, which corresponds to flipping edge probabilities and switching between $G(n,p)$ and $G(n,1-p)$. Note that the maximum value of the blue curve is approximately $2^{n/2}$, which aligns with the argument in Corollary 4.4 about the simulation complexity in terms of T-count.

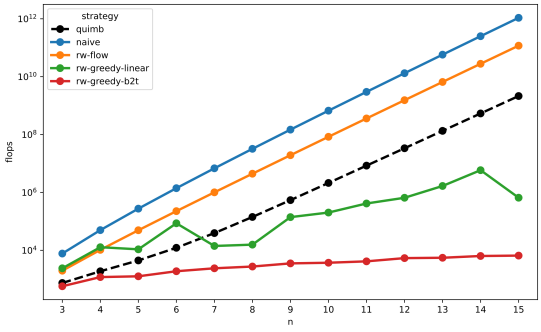


Figure 12: Simulating n-qubit Toffolis

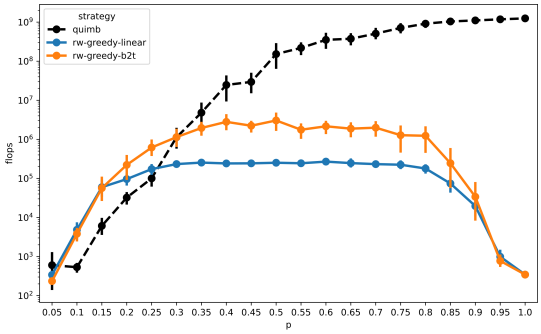


Figure 13: Simulating $G(30, p)$ ZX-diagrams

6 Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council grant number EP/Z002230/1, *(De)constructing quantum software (DeQS)*. The authors would like to thank Julien Codsi and Tuomas Laakkonen for useful discussions on rank-decompositions and classical simulation.

References

[1] Scott Aaronson & Daniel Gottesman (2004): *Improved simulation of stabilizer circuits*. *Phys. Rev. A* 70, p. 052328, doi:10.1103/PhysRevA.70.052328. Available at `https://link.aps.org/doi/10.1103/PhysRevA.70.052328`.

[2] Matthew Amy (2019): *Towards large-scale functional verification of universal quantum circuits*. In Peter Selinger & Giulio Chiribella, editors: *Proceedings of the 15th International Conference on Quantum Physics and Logic, Electronic Proceedings in Theoretical Computer Science* 287, doi:10.4204/EPTCS.287.1.

[3] Matthew Amy, Dmitri Maslov & Michele Mosca (2014): *Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33(10), pp. 1476–1489.

[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell et al. (2019): *Quantum supremacy using a programmable superconducting processor*. *Nature* 574(7779), pp. 505–510, doi:10.1038/s41586-019-1666-5.

[5] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski & John van de Wetering (2021): *There and back again: A circuit extraction tale*. *Quantum* 5, p. 421, doi:10.22331/q-2021-03-25-421. Available at `http://dx.doi.org/10.22331/q-2021-03-25-421`.

[6] Martin Beyß (2013): *Fast Algorithm for Rank-Width*. In Antonín Kučera, Thomas A. Henzinger, Jaroslav Nešetřil, Tomáš Vojnar & David Antoš, editors: *Mathematical and Engineering Methods in Computer Science*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 82–93.

[7] Sergey Bravyi, David Gosset & Yinchen Liu (2022): *How to simulate quantum measurement without computing marginals*. *Physical Review Letters* 128(22), p. 220503.

[8] Sergey Bravyi, Graeme Smith & John A Smolin (2016): *Trading classical and quantum computational resources*. *Physical Review X* 6(2), p. 021043.

[9] Daniel E Browne, Elham Kashefi, Mehdi Mhalla & Simon Perdrix (2007): *Generalized flow and determinism in measurement-based quantum computation*. *New Journal of Physics* 9(8), pp. 250–250.

[10] Julien Codsi & Tuomas Laakkonen (2026): *Unifying Graph Measures and Stabilizer Decompositions for the Classical Simulation of Quantum Circuits*. Personal communication.

[11] Bob Coecke & Ross Duncan (2008): *Interacting quantum observables*. In: *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, doi:10.1007/978-3-540-70583-3_25.

[12] Bruno Courcelle, Joost Engelfriet & Grzegorz Rozenberg (1993): *Handle-rewriting hypergraph grammars*. *Journal of Computer and System Sciences* 46(2), pp. 218–270, doi:https://doi.org/10.1016/0022-0000(93)90004-G. Available at `https://www.sciencedirect.com/science/article/pii/002200009390004G`.

[13] Ross Duncan & Simon Perdrix (2010): *Rewriting measurement-based quantum computations with generalised flow*. In: *International Colloquium on Automata, Languages, and Programming*, Springer, pp. 285–296, doi:10.1007/978-3-642-14162-1_24.

[14] Michel X. Goemans & José A. Soto (2013): *Algorithms for Symmetric Submodular Function Minimization under Hereditary Constraints and Generalizations*. *SIAM Journal on Discrete Mathematics* 27(2), pp. 1123–1145, doi:10.1137/120891502. Available at `http://dx.doi.org/10.1137/120891502`.

[15] Johnnie Gray & Stefanos Kourtis (2021): *Hyper-optimized tensor network contraction*. *Quantum* 5, p. 410.

[16] Marc Hein, Wolfgang Dür, Jens Eisert, Robert Raussendorf, M Nest & H-J Briegel (2006): *Entanglement in graph states and its applications*. In: *Proceedings of the International School of Physics "Enrico Fermi" on Quantum Computers, Algorithms and Chaos*.

[17] Aleks Kissinger & John van de Wetering (2020): *PyZX: Large Scale Automated Diagrammatic Reasoning*. In Bob Coecke & Matthew Leifer, editors: Proceedings 16th International Conference on *Quantum Physics and Logic*, Chapman University, Orange, CA, USA., 10-14 June 2019, *Electronic Proceedings in Theoretical Computer Science* 318, Open Publishing Association, pp. 229–241, doi:10.4204/EPTCS.318.14.

[18] Aleks Kissinger & John van de Wetering (2022): *Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions*. *Quantum Science and Technology* 7(4), p. 044001, doi:10.1088/2058-9565/ac5d20. Available at `http://dx.doi.org/10.1088/2058-9565/ac5d20`.

[19] Aleks Kissinger & John van de Wetering (2024): *Picturing Quantum Software: An Introduction to the ZX-Calculus and Quantum Compilation*. Preprint.

[20] Aleks Kissinger, John van de Wetering & Renaud Vilmart (2022): *Classical Simulation of Quantum Circuits with Partial and Graphical Stabiliser Decompositions*. In: *17th Conference on the Theory of Quantum Computation, Communication and Cryptography*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, doi:10.4230/LIPICS.TQC.2022.5. Available at `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TQC.2022.5`.

[21] Fedor Kuyanov (2025): *A Rank-Width–Based Approach to Quantum Circuit Simulation via ZX-Calculus*. Master's thesis, University of Oxford.

[22] Igor L Markov & Yaoyun Shi (2008): *Simulating quantum computation by contracting tensor networks*. *SIAM Journal on Computing* 38(3), pp. 963–981.

[23] Ramis Movassagh (2023): *The hardness of random quantum circuits*. *Nature Physics* 19(11), pp. 1719–1724, doi:10.1038/s41567-023-02131-2. Available at `https://doi.org/10.1038/s41567-023-02131-2`.

[24] Florian Nouwt (2022): *A simulated annealing method for computing rank-width*. Master's thesis, Utrecht University.

[25] Sang-il Oum (2003): *Approximation algorithm for the Clique-width*. https://mathsci.kaist.ac.kr/ sangil/pdf/rwdslide.pdf.

[26] Sang-il Oum (2017): *Rank-width: Algorithmic and structural results*. *Discrete Applied Mathematics* 231, pp. 15–24, doi:https://doi.org/10.1016/j.dam.2016.08.006. Available at `https://www.sciencedirect.com/science/article/pii/S0166218X16303705`. Algorithmic Graph Theory on the Adriatic Coast.

[27] Feng Pan & Pan Zhang (2021): *Simulating the Sycamore Quantum Supremacy Circuits*. *arXiv:2103.03074 [physics, physics:quant-ph]*. arXiv:2103.03074.

[28] David Philipps (2025): *GNNs for Fast Classical Simulation of Quantum Circuits through Optimizing ZX-Graph Decompositions*. Master's thesis, University of Oxford.

[29] Maurice Queyranne (1998): *Minimizing symmetric submodular functions*. Mathematical Programming 82(1), pp. 3–12, doi:10.1007/BF01585863. Available at `https://doi.org/10.1007/BF01585863`.

[30] Neil Robertson & P.D Seymour (1984): *Graph minors. III. Planar tree-width*. Journal of Combinatorial Theory, Series B 36(1), pp. 49–64, doi:https://doi.org/10.1016/0095-8956(84)90013-3. Available at `https://www.sciencedirect.com/science/article/pii/0095895684900133`.

[31] Will Simmons (2021): *Relating Measurement Patterns to Circuits via Pauli Flow*. Electronic Proceedings in Theoretical Computer Science 343, pp. 50–101, doi:10.4204/eptcs.343.4. Available at `http://dx.doi.org/10.4204/EPTCS.343.4`.

[32] Matthew Sutcliffe (2025): *Novel Methods for Classical Simulation of Quantum Circuits via ZX-Calculus*. Master's thesis, University of Oxford.

[33] Guifré Vidal (2003): *Efficient classical simulation of slightly entangled quantum computations*. Physical review letters 91(14), p. 147902.

A Rank-decomposition routines

Here we provide the proofs, pseudocode, and complexity analysis of our rank-decomposition routines. Section A.1 contains the proof of Lemma 3.1. In Sections A.2, A.3, we provide the pseudocode of "rw-greedy-linear" and "rw-greedy-b2t" and analyse their complexity.

A.1 rw-flow

Proof of Lemma 3.1. We prove that p_i constitutes a linear rank-decomposition of width $\leq |O|$ by induction from right to left. Let $M^{(i)}$ denote the adjacency matrix between $\{p_{i+1},\dots,p_{|V|}\}$ and $\{p_1,\dots,p_i\}$, respectively, and set $r_i := \mathrm{rk}(M^{(i)})$. Let us show that for each $i \in [2;|V|]$, we have $r_{i-1} \leq r_i$ for $p_i \notin O$ and $r_{i-1} \leq r_i + 1$ for $p_i \in O$.

Note that $M^{(i-1)}$ can be obtained from $M^{(i)}$ by deleting its last column and appending a row, representing the edges between p_i and $p_1,\dots,p_{i-1}$. It follows that $r_{i-1} \leq r_i + 1$. We now assume $p_i \notin O$ and show $r_{i-1} \leq r_i$ by considering the following two cases. It then follows that $\max(r_i) \leq |O|$, as desired.

1. $\lambda(p_i) = \mathrm{XY}$. By Definition 2.2, we have $p_i \notin g(p_i)$, $p_i \in \mathrm{Odd}(g(p_i))$, and for each $j < i$ we have $p_j \notin \mathrm{Odd}(g(p_i))$. Take some $u \in g(p_i)$ and add all the rows from $g(p_i) \setminus \{u\}$ to the row u, so that it has a single one at the i-th column. Then, zero out the entire i-th column so that $M^{(i)}$ becomes block-diagonal. See Figure 14 for illustration. Now consider the submatrix $\widetilde{M}$ between $\{p_{i+1},\dots,p_{|V|}\}$ and $\{p_1,\dots,p_{i-1}\}$, which corresponds to the 'after–before' section in the first diagram of Figure 14. Note that the row operations preserve the ranks of $\widetilde{M}$, $M^{(i)}$, and $M^{(i-1)}$. Due to the diagonalisation of $M^{(i)}$ as shown in the last diagram of Figure 14, we have $\mathrm{rk}(\widetilde{M}) = \mathrm{rk}(M^{(i)}) - 1$. As $M^{(i-1)}$ differs from $\widetilde{M}$ by one extra row, we have $\mathrm{rk}(M^{(i-1)}) \leq \mathrm{rk}(\widetilde{M}) + 1 = \mathrm{rk}(M^{(i)})$.

2. $\lambda(p_i) \in \{\mathrm{XZ},\mathrm{YZ}\}$. By Definition 2.2, we have $p_i \in g(p_i)$, and for each $j < i$ we have $p_j \notin \mathrm{Odd}(g(p_i))$. Hence, by adding the rows from $g(p_i) \setminus \{p_i\}$ to the row p_i, we can zero out its 'before' part – see Figure 15. Define $\widetilde{M}$ as in the previous case. Clearly, $\mathrm{rk}(M^{(i-1)}) = \mathrm{rk}(\widetilde{M}) \leq \mathrm{rk}(M^{(i)})$.

□

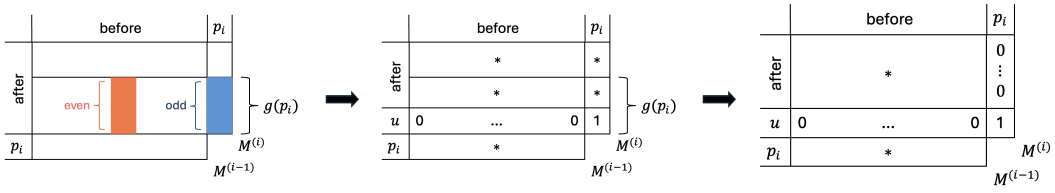


Figure 14: Block diagonalisation of the adjacency matrix in the XY case.

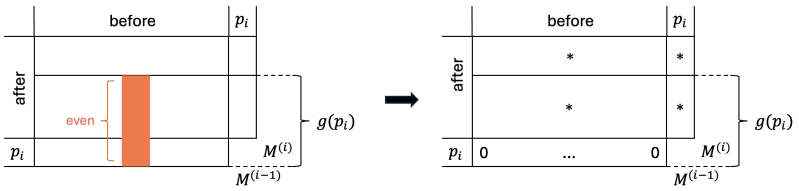


Figure 15: Matrix row reduction in the XZ and YZ cases.

A.2 rw-greedy-linear

The pseudocode of "rw-greedy-linear" is given by Algorithm A.1. Note that each cut-rank calculation costs $O(|V|^3)$ operations (e.g. by doing Gaussian elimination). Since at each iteration there are $O(|V|)$ choices for u, the complexity of Algorithm A.1 is $O(|V|^5)$. It is possible to reduce the complexity to $O(|V|^4)$, and we have implemented the optimised routine in PyZX.[4]

Algorithm A.1 rw-greedy-linear

Input: $G = (V, E)$
Output: $(p_i)_{i=1,\ldots,|V|}$ – linear rank-decomposition

```
p ← list()
for i = 1...|V| do
    S ← {p_1, ..., p_{i-1}}
    M ← Mat_G(S, V \ S)
    if M = 0 then
        p_i ← min(V \ S)
    else
        p_i ← argmin_{u ∈ pivotCols(M)} ρ_G(S ∪ {u})
    end if
end for
```

A.3 rw-greedy-b2t

The pseudocode of "rw-greedy-b2t" is given by Algorithm A.2. Note that for each pair of subtrees (S_u, S_v), computing the cut-rank $\rho_G(S_u \cup S_v)$ costs $O((|S_u| + |S_v|)|V|^2)$ operations. Summing over all u, v,

[4] https://github.com/zxcalc/pyzx/blob/5f5e409/pyzx/rank_width.py#L172

each iteration cost is bounded by

$$\sum_{u,v} O((|S_u| + |S_v|)|V|^2) = O\left(\sum_u |S_u| \cdot |V|^3\right) = O(|V|^4),$$

hence the complexity of Algorithm A.2 is $O(|V|^5)$. Similarly to "rw-greedy-linear", it is possible to reduce the complexity to $O(|V|^4)$, and we have implemented the optimised routine in PyZX.[5]

Algorithm A.2 rw-greedy-b2t

Input: $G = (V, E)$
Output: T – rank-decomposition tree

```
S ← dict()                          ▷ for each tree root in T, set of leaves in its component
T ← tree()                          ▷ forest of partial decompositions
for i = 1 ... |V| do
    S_i ← {i}
    T.newVertex()                   ▷ initially, T has |V| isolated vertices numbered 1, ..., |V|
end for
for t = 1 ... |V| − 1 do
    i, j ← argmin_{u,v: S_u ∩ pivotCols(Mat_G(S_v, V\S_v)) ≠ ∅} ρ_G(S_u ∪ S_v)
    k ← T.newVertex()               ▷ create a new root k with children i, j
    T.addEdge(k, i)
    T.addEdge(k, j)
    S_k ← S_i ∪ S_j                 ▷ update sets of leaves
    S.remove(i)
    S.remove(j)
end for
```

[5] https://github.com/zxcalc/pyzx/blob/5f5e409/pyzx/rank_width.py#L203