Saarland University
Faculty of Mathematics and Computer Science
Departments of Mathematics and Computer Science

Master's thesis

An algorithmic approach of transferring zero relations in quantum permutation groups

submitted by
Matias Klimpel Akahoshi

submitted
02.03.2026

Reviewers:
Professor Dr. Moritz Weber
Professor Dr. Gabriela Weitze-Schmithüsen

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit keine mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

Declaration of Original Authorship

I hereby declare that this thesis is my own original work and was completed independently, without unauthorized assistance or unacknowledged sources. Any content derived from publications or other external sources, whether quoted verbatim or paraphrased, has been duly acknowledged and clearly marked as such. I hereby confirm that, in preparing the following thesis, I have not used any artificial intelligence-based software (such as ChatGPT) to address the core research questions presented in this thesis. I acknowledge that any breach of this declaration may result in failing the examination and, in severe cases, the loss of the right to be examined.

Ort/Place, Datum/Date	Unterschrift/Signature

Contents

Introduction

A finite graph Γ consists of a set of n vertices and a set of edges encoding connections between its vertices. One of the natural things to look at when examining a graph are its symmetries which are operations on the set of vertices under which the set of edges is invariant. All of the symmetries of a graph form the group $\mathrm{Aut}(\Gamma)$ called the automorphism group of Γ. This automorphism group is a subgroup of the symmetric group S_n. If A_Γ denotes the adjacency matrix of Γ then $\mathrm{Aut}(\Gamma)$ can be written as $\{\sigma \in S_n \text{ permutation matrix} \mid \sigma A_\Gamma = A_\Gamma \sigma\}$.

There has been a movement in mathematics started by Murray and von Neumann in the 1930s with von Neumann algebras [20] and furthered by Gelfand and Naimark in the 1940s with their work on C^*-algebras [13] of considering "non-commutative"—or "quantum"—versions of known concepts. Here, the theory of von Neumann algebras can be seen as non-commutative measure theory and the theory of C^*-algebras can be seen as non-commutative topology. The term "quantum" comes from the fact that quantum mechanical systems are modelled using bounded operators acting on Hilbert spaces which in general do not commute with each other. We are particularly interested in the theory of quantum groups introduced by Woronowicz in the 1980s as a non-commutative version of group theory in order to study the "quantum symmetries" of a graph. Further notable examples of quantized theories include among others quantum probability theory introduced by Accardi in the 1970s (with an overview published in 1981 [1]) and free probability introduced by Voiculescu in 1985 [31] as non-commutative versions of probability theory, free analysis introduced by Taylor in 1972 [28] as a non-commutative version of complex analysis and non-commutative geometry introduced by Connes in 1985 [9].

In order to formulate what the "quantum automorphism group of a graph" should be, we first take a look at the quantized version of the permutation group on n points. It is given by the quantum permutation group S_n^+ proposed by Wang in 1998 [32] which is a compact matrix quantum group $(C(S_n^+), u)$ where $C(S_n^+)$ is the universal C^*-algebra defined by

$$C(S_n^+) = (u_{ij}, i, j \in [n], 1 \mid u_{ij} = u_{ij}^* = u_{ij}^2, \sum_{s=1}^{n} u_{is} = \sum_{s=1}^{n} u_{si} = 1).$$

The concept of a compact matrix quantum group (CMQG) was introduced by Woronowicz in 1987 [34]. As the classical automorphism group of a graph with n vertices is a subgroup of S_n, the quantum automorphism group analogously should be a compact matrix quantum subgroup of the quantum permutation group S_n^+. In fact, two definitions for the quantum automorphism group of a graph have been proposed—one denoted by $G_{aut}^*(\Gamma)$ introduced by Bichon in 2003 [5] and one denoted by $G_{aut}^+(\Gamma)$ introduced by Banica in 2005 [2]. Both definitions are compact quantum matrix subgroups of S_n^+—and even further we have $G_{aut}^*(\Gamma) \subseteq G_{aut}^+(\Gamma) \subseteq S_n^+$ as CMQGs. However, in this thesis we are concerned only with $G_{aut}^+(\Gamma) = (C(G_{aut}^+(\Gamma)), u)$ where $C(G_{aut}^+(\Gamma))$ is the universal C^*-algebra defined by

$$C(G_{aut}^+(\Gamma)) = (u_{ij}, i, j \in [n], 1 \mid u_{ij} = u_{ij}^* = u_{ij}^2, \sum_{s=1}^{n} u_{is} = \sum_{s=1}^{n} u_{si} = 1, A_\Gamma u = u A_\Gamma).$$

This definition clearly shows the similarity to the classical automorphism group as a subgroup of the symmetrical group respecting the graph structure.

Banica and Bichon then went on to define the notion of quantum symmetries in 2006 [3]. A graph is said to have no quantum symmetries if its quantum automorphism group is commutative or equivalently if we have $C(G^+_{aut}(\Gamma)) = C(\mathrm{Aut}(\Gamma))$. The question of which graphs do or do not have quantum symmetries has been studied for specific examples and classes of graphs in the past twenty years. In 2018, Schmidt and Weber determined the quantum automorphism groups of all simple graphs with four vertices [25]. In 2019, Eder, Levandovskyy, Schanz, Schmidt, Steenpass and Weber algorithmically computed the existence of quantum automorphisms for all connected, simple graphs on up to six vertices and on seven vertices for graphs whose automorphism groups have order up to two [17]. The quantum automorphism groups of vertex-transitive graphs of order up to 13 have been determined in the years 2006 to 2024 by Banica and Bichon [3], Schmidt [23], Chassaniol [7] and Schanz [22]. In 2025, van Dobben de Bruyn, Kar, Roberson, Schmidt and Zeman gave a characterization of quantum automorphism groups of trees [29]. Additionally, in 2025, van Dobben de Bruyn, Roberson and Schmidt also constructed a graph with trivial classical automorphism group that admits quantum symmetries [30] which answered a previously open question.

These results have been found using a wide variety of tools. However, a central part of almost all of these computations is using the defining relations of $C(G^+_{aut}(\Gamma))$ or relations derived from them. In particular, it often comes to pass that these algebraic transformations show that certain monomials are equal to zero which in turn then can be used to find that certain pairs of generators of $C(G^+_{aut}(\Gamma)$ commute. If this can be done for all pairs of generators the graph has no quantum symmetries.

So far, we are only aware of the approaches taken in [17] and [22] that tried generalized algorithmic approaches in order to compute the quantum automorphism groups of graphs. They both rely on the same implementation of a non-commutative version of Buchberger's algorithm introduced in [17] in order to compute the Gröbner bases of the quantum automorphism groups. However, unlike in the commutative case, Buchberger's algorithm is not guaranteed to terminate here.

In this thesis, we propose a different algorithmic approach that mimics the manual approach of using the relations of the quantum automorphism group of a graph to find that certain monomials are zero and using this information in order to try to conclude whether the quantum automorphism group is commutative. Our approach relies on propagating zeros among monomials according to specified rules derived from the defining relations of the quantum automorphism group of the graph.

The Algorithm

We propose an algorithm that takes a graph Γ with n-many vertices and the maximal length k of monomials one wants to allow as an input. It then computes a number of zeros in the quantum automorphism group of the graph Γ and tries to use them to determine whether the graph has quantum symmetries. Limiting the maximal length of monomials ensures termination, but as a trade-off the output of the algorithm becomes ambiguous in the following sense: If the algorithm outputs that $G^+_{aut}(\Gamma)$ is commutative this is true and the output of the algorithm serves as a proof. However, if the algorithm does not it is uncertain if increasing k would yield a different result

or if Γ does have quantum symmetries. Even in this case the algorithm can give a list of monomials that have been determined to be zero, and that can then be used in further manual computations. Our approach can also be an additional tool to automatically check large volumes of graphs in conjunction with other methods.

The algorithm itself can be split into four different sub-algorithms:

(1) *Monomial graph creation*: This sub-algorithm creates a directed hypergraph called the *monomial graph* with vertices corresponding to monomials up to a predetermined length k. The edges of the graph are chosen such that if all vertices in the source set of an edge are zero so are the vertices in the range set of the edge. The monomial graph is constructed inductively by increasing k. The induction step is the main part of this sub-algorithm and is shown in Algorithm 7.

(2) *Finding start zeros*: This sub-algorithm determines monomials that can be seen to be zero from the structure of the graph and the relations of the quantum automorphism group. This step is designed to be modular such that additional criteria may be added.

(3) *Monomial graph traversal*: This sub-algorithm is the main part of the entire algorithm and is shown in Algorithm 10. It iteratively traverses the monomial graph along its edges in order to propagate zeros starting with the zeros found in the previous step. This is repeated until the monomial graph cannot be explored further. While traversing the monomial graph, simple commutation relations are also noted down and used in the exploration.

(4) *Post-processing*: This sub-algorithm finds additional commutation relations from the zeros found in the previous steps. As with the second step, this step is designed to be modular such that additional criteria may be added.

Finally, the sub-algorithms (2) to (4) are iterated until either the input graph Γ is found to have a commutative quantum automorphism group or until no additional information is found.

For example, let $\Gamma = K_{1,3}$ be the claw graph with $n = 4$ vertices.

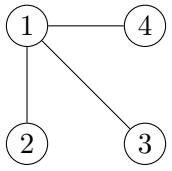


Figure 1: The claw graph $K_{1,3}$

$C(G^+_{aut}(\Gamma))$ has the generators u_{ij} for $i, j \in \{1, 2, 3, 4\}$. The most straight forward case for propagating a zero is multiplication: If we know that the monomial $u_{11}u_{12}$ is zero multiplying it with any generator u_{kl} obviously results in another zero. One of the more interesting rules makes use of the fact that we have $\sum_{s=1}^{4} u_{is} = 1$ for any $i \in \{1, 2, 3, 4\}$ as one of the defining relations of $C(G^+_{aut}(\Gamma))$. Thus, if we know

that the monomials $u_{1i}u_{11}u_{12}$ are equal to zero for every i then it follows that

$$0 = \sum_{s=1}^{4} u_{1s}u_{11}u_{12} = \left(\sum_{s=1}^{4} u_{1s}\right) u_{11}u_{12} = u_{11}u_{12}.$$

In this way we have both methods for propagating zeros to monomials of longer and shorter length. The entirety of our propagation rules are laid out in Section 4. The first sub-algorithm creates the (n, k)-*monomial graph* according to these rules.

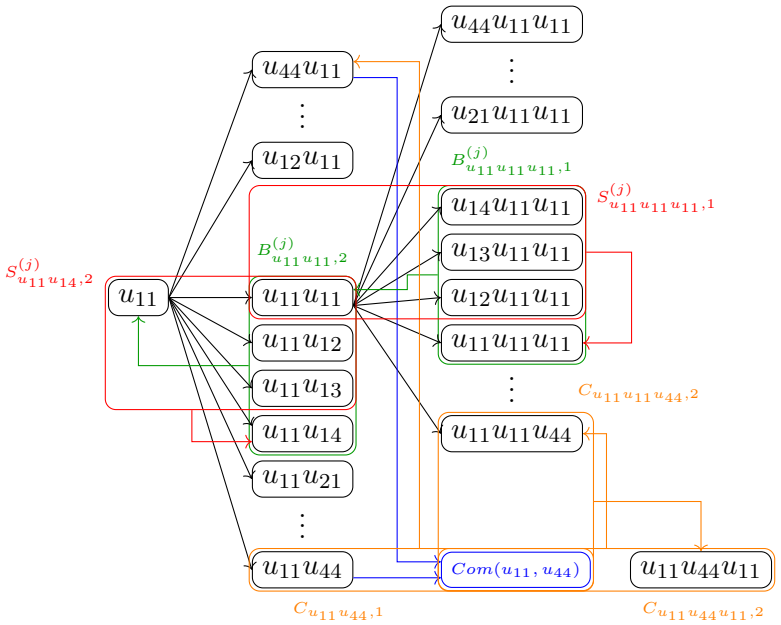


Figure 2: Part of the $(4, 3)$-monomial graph with edges coloured according to the propagation rules they are derived from.

Having created the (n, k)-monomial graph, we next aim to find initial zeros. In the case of the claw graph we can use among others a criterion that states that for two vertices i and j with differing degrees, the generator u_{ij} is equal to zero. Thus, we would be able to add the monomials $u_{12}, u_{13}, u_{14}, u_{21}, u_{31}$ and u_{41} to the list of start zeros. The list of criteria and their proofs are shown in Section 2. For the claw graph, using the parameter $k = 3$, we are able to find 174 start zeros using the different criteria.

Next, the traversal sub-algorithm is a graph traversal algorithm that explores the monomial graph starting from the start zeros. In our example with the claw graph and $k = 3$, the traversal finds 3804 additional zeros bringing the number of all found zeros up to 3978. Further, in this step we also already mark those commutation relations that we can deduce by a product of two generators being zero. In our example, we find 186 such commutation relations.

Finally, the post-processing sub-algorithm uses the found zeros to extract additional commutation relations. The criteria used in this step are laid out in Sections 2 and 4. In our example, we find an additional 54 commutation relations.

Thus, at the end of one iteration of all of our sub-algorithms, we find 16 trivial commutation relations (each generator trivially commutes with itself), 186 commu-

tation relations during the traversal step and 54 commutation relations during the post-processing step for a total of $16^2 = 256$ commutation relations. Therefore, any two generators commute with each other and we can determine that $C(G^+_{aut}(\Gamma))$ is commutative.

Results

Using this algorithm, we are able to find all graphs on 4 vertices without quantum symmetries in around one seconds, all graphs on 6 vertices without quantum symmetries in under five minutes, and 507 graphs without quantum symmetries on connected graphs on 7 vertices in 45 minutes, improving on a result in [17] that only considered connected graphs on 7 vertices whose classical automorphism group is either trivial or $\mathbb{Z}_2$. Combining our tool with another criterion that can show that certain graphs have quantum symmetries, we are able to show that these 507 graphs without quantum symmetries are in fact all connected graphs on 7 vertices that do not have quantum symmetries. Further, our algorithm is also able to show that the Petersen graph does not admit quantum symmetry which has been shown previously in [23] but as far as we are aware could not be found algorithmically. These results are shown in more detail is Section 5.

Table 1: Results of running our algorithm on different classes of graphs

Graph/Class of Graphs	# of graphs	# of graphs without quantum symmetries	# of graphs without quantum symmetries found by our algorithm	Run time
Graphs on 4 vertices	11	5	5	1.116 seconds
Graphs on 6 vertices	156	68	68	250 seconds
Connected graphs on 7 vertices	853	507	507	45 minutes
Petersen Graph	1	1	1	5.95 hours

1 Preliminaries

In this section we present the terminology for graphs used in this thesis, the C^*-algebraic objects studied with a focus on quantum groups and introduce the computational model underlying the algorithms in this thesis.

1.1 Notation. In this thesis we write $[n]$ as a shorthand for the set $\{1, \ldots, n\}$.

1.1 Graphs

1.2 Definition. A *finite graph* $\Gamma = (V, E)$ is a pair consisting of a finite non-empty set V of vertices and a set $E \subseteq V \times V$ of edges. A graph is called *undirected* if for any edge $(u, v) \in E$ the edge (v, u) is also in E. A undirected graph is called *simple* if it contains no loops or multiple edges. In this thesis we consider only finite simple graphs unless otherwise specified. We say that vertices u and v are adjacent if (u, v) is an edge. In this case we also write $u \sim v$. Otherwise we write $u \not\sim v$.

1.3 Definition. Let $\Gamma = (V, E)$ be a graph and $u, v \in V$. The *degree of* u is defined as $\deg(u) = \#\{v \in V \mid v \sim u\}$. A *path* p *of length* k *from* u *to* v is a sequence of vertices u_i for $i \in \{0, \ldots, k\}$ such that $u_0 = u$ and $u_k = v$ and $u_{i-1} \sim u_i$ for all $i \in \{1, \ldots, k\}$. The *distance* $d(u, v)$ *of* u *and* v is defined as $d(u, v) = \min\{k \in \mathbb{N}_0 \mid \exists \text{ path of length } k \text{ from } u \text{ to } v\}$ and is set to ∞ if u and v are not in the same connected component. The *adjacency matrix* A_Γ is the $|V| \times |V|$-matrix defined by

$$(A_\Gamma)_{u,v} = \begin{cases} 1 & \text{if } u \sim v, \\ 0 & \text{if } u \not\sim v. \end{cases}$$

and the *distance matrix* D_Γ is the $|V| \times |V|$-matrix defined by

$$(D_\Gamma)_{u,v} = d(u, v).$$

1.4 Example. Consider the graph Γ with $V = [6]$ and adjacency matrix

$$A_\Gamma = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Its graphical representation is shown in Figure 3.

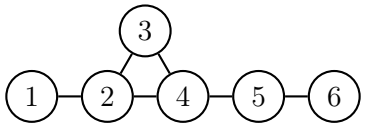


Figure 3: Graph with 6 points

Its distance matrix is given by

$$D_\Gamma = \begin{pmatrix} 0 & 1 & 2 & 2 & 3 & 4 \\ 1 & 0 & 1 & 1 & 2 & 3 \\ 2 & 1 & 0 & 1 & 2 & 3 \\ 2 & 1 & 1 & 0 & 1 & 2 \\ 3 & 2 & 2 & 1 & 0 & 1 \\ 4 & 3 & 3 & 2 & 1 & 0 \end{pmatrix}.$$

1.5 Definition. Let $\Gamma = (V, E)$ be a graph. An *automorphism of* Γ is a bijective map $\sigma : V \to V$ such that

$$u \sim v \iff \sigma(u) \sim \sigma(v).$$

The set of all automorphisms of Γ forms the group $\mathrm{Aut}(\Gamma)$ and is a subgroup of the symmetric group $S_{|V|}$. As such it may be represented through permutation matrices which allow for the following characterization using the adjacency matrix A_Γ

$$\mathrm{Aut}(\Gamma) = \{\sigma \in S_{|V|} \mid \sigma A_\Gamma = A_\Gamma \sigma\}.$$

1.6 Definition. A *(directed) hypergraph* is a pair (V, E) consisting of a finite non-empty set of vertices V and a set E of pairs of subsets of V. We call the elements $(S, R) \in E$ *hyperedges* and the sets S and R the *source set* and *range set* respectively. If it is clear from context we may omit the prefix *hyper* and the word *set* and speak of the *edges* of a hypergraph with their *sources* and *ranges*.

1.2 Universal C^*-Algebras

In this section we broadly introduce C^*-algebras. The contents of this section have been primarily sourced from [33].

1.7 Definition. A *C^*-algebra* is a complex Banach algebra A with an involution, i.e. an anti-linear map

$$\begin{aligned} {}^* : A &\longrightarrow A \\ x &\longmapsto x^* \end{aligned}$$

such that $x^{**} = x$, $(xy)^* = y^*x^*$ and $\|x^*x\| = \|x\|^2$. We say that A is *unital* if $1 \in A$.

If $\mathcal{H}$ is a Hilbert space then $B(\mathcal{H})$, the space of bounded operators acting on $\mathcal{H}$, is a unital C^*-algebra. In fact, the following theorem by Gelfand and Naimark shows that bounded operators acting on Hilbert spaces can be considered the canonical representations of C^*-algebras. See [13] for the original proof by Gelfand and Naimark or Theorem II.6.4.10 in [6] for a modern version.

1.8 Theorem (Second Gelfand-Naimark Theorem)**.** *Every C^*-algebra A admits a faithful representation $\pi : A \hookrightarrow B(\mathcal{H})$, i.e. an injective $*$-homomorphism, on a Hilbert space $\mathcal{H}$. Hence, A is isomorphic to a C^*-subalgebra of $B(\mathcal{H})$.*

1.9 Definition. Let I be an index set and $E = \{x_i \mid i \in I\}$ a set of generators. Further, let $P(E)$ bet the involutive $\mathbb{C}$-algebra of non-commutative $*$-polynomials in E and let $R \subseteq P(E)$ be a set of relations. Let $J(R) \subseteq P(E)$ be the two-sided ideal in $P(E)$ generated by R. Define

$$A(E,R) \coloneqq P(E)/J(R)$$

as the universal *-algebra with generators E and relations R. For $x \in A(E,R)$ put

$$\|x\| \coloneqq \sup\{p(x) \mid \ p \text{ is a } C^*\text{-seminorm on } A(E,R)\}.$$

If now $\|x\| < \infty$ for all $x \in A(E,R)$ define

$$C^*(E \mid R) \coloneqq \overline{A(E,R)/\{x \in A(E,R) \mid \|x\| = 0\}}^{\|\cdot\|}$$

as the *universal C^*-algebra with generators E and relations R.*

In particular, a universal C^*-algebra admits the following universal property

1.10 Proposition. *Let $A = C^*(E \mid R)$ be a universal C^*-algebra as in Definition 1.9 and let B be a C^*-algebra such that $E' = \{y_i \in B \mid i \in I\} \subseteq B$ is a subset satisfying the relations R. Then there is a unique $*$-homomorphism $\varphi : A \to B$ with $\varphi(x_i) = y_i$.*

1.11 Example. Consider the following universal C^*-algebra

$$\begin{aligned} A \coloneqq C^*(u_{ij}, i,j \in [n], 1 \mid\, & u_{ij} = u_{ij}^* = u_{ij}^2, \\ & \sum_{s=1}^{n} u_{is} = \sum_{s=1}^{n} u_{sj} = 1, \\ & u_{ij}u_{kl} = u_{kl}u_{ij}\ \forall i,j,k,l \in [n]) \end{aligned}$$

defined by projections u_{ij}. In particular, this universal C^*-algebra does exist since for a projection u_{ij} we observe that

$$\|u_{ij}\| = \|u_{ij}^* u_{ij}\| = \|u_{ij}\|^2 \in \{0,1\}.$$

It turns out that this universal C^*-algebra is isomorphic to the continuous functions on the symmetric group S_n. To show this, we need the following theorem by Gelfand and Naimark.

1.12 Theorem (Commutative Gelfand-Naimark Theorem, Theorem II.2.2.4 in [6])**.** *Let A be a commutative unital C^*-algebra. Then there exists a compact topological space X such that $A \cong C(X)$. In fact, $X = \mathrm{Spec}(A)$ is the set of all characters of A.*

To see that the universal C^*-algebra A from Example 1.11 is, in fact, isomorphic to the continuous functions on the symmetric group, we show that the spectrum $\mathrm{Spec}(A)$ is homeomorphic to S_n. Then, Theorem 1.12 yields that $A \cong C(\mathrm{Spec}(A)) \cong C(S_n)$. To this end, we consider the mapping

$$\Psi : S_n \to \mathrm{Spec}(A), \sigma \mapsto \varphi_\sigma$$

where φ_σ is the character defined by $\varphi_\sigma(u_{ij}) \coloneqq \delta_{j,\sigma(i)}$. Then, Ψ defines a homeomorphism showing the claim.

1.3 Quantum Groups

In Theorem 1.12 we have seen that any commutative unital C^*-algebra is isomorphic to the continuous functions on a compact topological space X. Conversely, for such a compact space X, the pair $(C(X), \|\cdot\|_\infty)$ is a commutative unital C^*-algebra again. Thus, the theory of such C^*-algebras may be considered as the theory of continuous functions on compact groups. By omitting the commutativity of a C^*-algebra one may thus think of "continuous functions on a quantum group". This motivated Woronowicz to formulate a definition for compact quantum groups.

1.13 Definition ([35])**.** A *compact quantum group (CQG)* is a pair $G = (A, \Delta)$ where A is a separable unital C^*-algebra and $\Delta : A \to A \otimes A$ is a unital *-homomorphism such that

(a) $(\Delta \otimes \mathrm{Id}) \circ \Delta = (\mathrm{Id} \otimes \Delta) \circ \Delta$ and

(b) the sets

$$\{(b \otimes I)\Delta(c) \mid b, c \in A\}$$
$$\{(I \otimes b)\Delta(c) \mid b, c \in A\}$$

are linearly dense subsets of $A \otimes A$.

In this thesis we are only concerned with a special class of CQGs also introduced by Woronowicz in 1987 eight years before general CQGs.

1.14 Definition ([34])**.** A *compact matrix quantum group (CMQG)* is a pair $A = (C(A), u)$ where

- $C(A)$ is a unital C^*-algebra generated by elements $u_{ij}, i, j \in [n]$ which are the entries of the matrix u,
- $u = (u_{ij})_{i,j}$ and $u^t = (u_{ji})_{i,j}$ are invertible and
- the map $\Delta : C(A) \to C(A) \otimes C(A)$ defined by $\Delta(u_{ij}) = \sum_{k=1}^n u_{ik} \otimes u_{kj}$ is a *-homomorphism.

If we have two compact matrix quantum groups $A = (C(A), u)$ and $B = (C(B), v)$ we say that $B \subseteq A$ is a *compact matrix quantum subgroup of* A if there is a surjective $*$-homomorphism $\varphi : C(A) \to C(B)$ mapping generators to generators. We further say $A = B$ as CMQGs if $A \subseteq B$ and $B \subseteq A$ holds.

With this definition we see that the C^*-algebra $A = C(S_n)$ from Example 1.11 is in fact a CMQG with the matrix $u = (u_{ij})_{i,j}$. Since we set out to consider the quantum version of S_n, we get the CMQG $S_n^+ = (C(S_n^+), u)$ defined by

$$C(S_n^+) = (u_{ij}, i, j \in [n], 1 \mid u_{ij} = u_{ij}^* = u_{ij}^2, \sum_{s=1}^n u_{is} = \sum_{s=1}^n u_{si} = 1).$$

This CMQG was introduced by Wang in 1998 in [32] and is indeed the non-commutative version of $C(S_n)$, i.e. it holds that

$$C(S_n) \cong {}^{C(S_n^+)}\!/_{<u_{ij}u_{kl} = u_{kl}u_{ij}>}.$$

In terms of CMQGs, we have $S_n \subseteq S_n^+$ where the surjective $*$-homomorphism required is given directly by the universal property of $C(S_n^+)$. This also means that we have $S_n = S_n^+$ whenever $C(S_n^+)$ is commutative and we are thus particularly interested in the case where $C(S_n^+)$ is non-commutative. The following two propositions show when this is the case.

1.15 Proposition (Proposition 4.3.3 in [15]). *Let $n \in \{1, 2, 3\}$. Then $C(S_n^+)$ is commutative.*

Proof. We need to show that the generators of $C(S_n^+)$ commute with each other. For $n = 1$ this is trivially true. For $n = 2$, observe that the rows and columns of u summing to one means that we have

$$u = \begin{pmatrix} u_{11} & 1 - u_{11} \\ 1 - u_{11} & u_{11} \end{pmatrix}$$

and thus $C(S_2^+)$ is commutative. For $n = 3$, recall first that projections p_i in a unital C*-algebra satisfying $\sum_i p_i \leq 1$ are mutually orthogonal, i.e. $p_i p_j = 0$ for $i \neq j$. By the rows and columns summing to one, we thus already have $u_{ij}u_{kl} = 0 = u_{kl}u_{ij}$ if either $i = k$ or $j = l$. It remains to show that the generators u_{ij} and u_{kl} commute for $i \neq k$ and $j \neq l$. We show this now for u_{11} and u_{22}.

$$\begin{aligned} u_{11}u_{22} &= u_{11}u_{22}(\sum_{k=1}^{3} u_{1k}) \\ &= u_{11}u_{22}u_{11} + u_{11}u_{22}u_{12} + u_{11}u_{22}u_{13} \\ &= u_{11}u_{22}u_{11} + u_{11}u_{22}u_{13} \\ &= u_{11}u_{22}u_{11} + u_{11}(1 - u_{21} - u_{23})u_{13} \\ &= u_{11}u_{22}u_{11} + u_{11}u_{13} - u_{11}u_{21}u_{13} - u_{11}u_{23}u_{13} \\ &= u_{11}u_{22}u_{11}. \end{aligned}$$

Here, we also repeatedly used that the product of two generators from the same row or column of u is zero. Next, we can analogously show that $u_{22}u_{11} = u_{11}u_{22}u_{11}$ by multiplying from the left rather than from the right and get

$$u_{11}u_{22} = u_{22}u_{11}.$$

Similar computations can be done for the remaining pairs of generators and thus $C(S_3^+)$ is commutative. □

1.16 Proposition (Proposition 4.3.5 in [15]). *Let $n \geq 4$. Then $C(S_n^+)$ is non-commutative.*

Proof. Let $\mathcal{H}$ be a Hilbert space and let $p, q \in B(\mathcal{H})$ be projections with $pq \neq qp$. Call A the C*-subalgebra of $B(\mathcal{H})$ that is generated by p, q and 1. Further, let v be

the $n \times n$-matrix defined by

$$v = \begin{pmatrix} p & 1-p & & & & & \\ 1-p & p & & & & & \\ & & q & 1-q & & & \\ & & 1-q & q & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

Then, by the universal property of $C(S_n^+)$ there is a *-homomorphism $\varphi : C(S_n^+) \to A$ mapping u_{ij} to v_{ij}. In particular, we have $pq = \varphi(u_{11}u_{33})$ and $qp = \varphi(u_{33}u_{11})$ and by $pq \neq qp$ we get that u_{11} and u_{33} do not commute. □

1.4 Computational Model and Asymptotic Notation

In this section we present the notations of and assumptions on the computational model underlying the algorithms and their analysis in this thesis. In general, we follow the notions given in [10] which acts as a source for this section. This means in particular, that we assume a *random-access machine model* in which

- we have atomic data types such as *Boolean* (i.e. True or False), *Integer*, *Character*,
- we have lists in the form of *Arrays* and *Strings*, which are arrays of characters,
- we have composite data types in the form of *Objects*.
- Each instruction is executed after the previous one (in particular this rules out concurrent computation),
- data accesses (including access into an array) take constant time,
- common arithmetic instructions, i.e. addition, subtraction, multiplication, division and remainder (denoted by mod), and logic operations, i.e. conjunction, disjunction and negation (denoted by **and**, **or** and **not** respectively) take constant time,
- comparisons of atomic size objects take constant time and
- control flow statements, i.e. branching instructions, subroutine calls and return, take constant time.

1.17 Remark. We assume that each atomic datatype has an upper bound on the amount of information it may represent. This is clear for boolean values but for integers this means that there is a maximal value that may be represented as one integer in our model and for characters this means that the characters come from a finite alphabet. Because we are ultimately interested in running algorithms on computers in the real world, this upper bound on the size of integers is required, as it is a reality in modern computing architecture. Further, if one were to remove this constraint, we could encode arbitrarily large data in a single integer and transform it in constant time. This would render run time analysis meaningless.

In order to compute run times of algorithms, we count the number of constant-time steps an algorithm takes on a given input instance before terminating. As inputs grow in size, we also expect an algorithm's run time to increase. Therefore, we are interested in expressing the run time of an algorithm as a function of the input size. In order to compare algorithms and find general statements about the performance of an algorithm, we use asymptotic notation for run times; that is we want to know how an algorithm behaves as the inputs grow arbitrarily large. In order to do so, we use the following notation.

1.18 Definition. Let f be a real-valued function defined on a subset D of $\mathbb{R}$. The *set $\mathcal{O}(f(x))$ of functions asymptotically bounded by f* is defined as

$$\mathcal{O}(f(x)) = \{g(x) \text{ real-valued function } | \\ \exists c > 0, x_0 \in \mathbb{R} \text{ such that } 0 \leq g(x) \leq cf(x)\ \forall x \in D, x \geq x_0\}.$$

1.19 Remark. While $\mathcal{O}(f(x))$ is a set, it has established itself to write $g(x) = \mathcal{O}(f(x))$ (pronounced "g is big-Oh of f" or just "g is Oh of f") rather than $g(x) \in \mathcal{O}(f(x))$ (pronounced "g is *in* (big-)Oh of f"). In this thesis we usually write that something happens in time $\mathcal{O}(f(x))$ or that an algorithm has a run time of $\mathcal{O}(f(x))$. Since $\mathcal{O}(f(x))$ is an upper bound, we have the transitivity property that if $g(x) = \mathcal{O}(f(x))$ and $h(x) = \mathcal{O}(g(x))$, then $h(x) = \mathcal{O}(f(x))$ holds as well. In such a case we might say "h is in $\mathcal{O}(g(x))$ which is in $\mathcal{O}(f(x))$" or "h is in $\mathcal{O}(g(x))$ which is dominated by $\mathcal{O}(f(x))$".

The big-Oh notation is not reserved for run times but could also be used for expressing the asymptotic size of some other parameter like the number of elements in a list where only the rate of growth is of importance for us and not the exact size of the list. In this thesis we usually give f via the term used to define it as in $\mathcal{O}(n^2)$ where we mean $\mathcal{O}(f(n))$ with $f(n) = n^2$. In the cases that a function specification could be interpreted in two or more ways it is either clear from context which variable is meant to be the function's argument or it will be mentioned explicitly. The notation naturally extends to multivariate functions if an input is defined in terms of several variables (for example, an algorithm whose run time is linear in the number of vertices n and edges m of a graph would have a run time of $\mathcal{O}(n+m)$). For more details and an overview how this notation is typically used we refer to Chapter 3.2 in [10]. There are other classes of asymptotically-bounded functions like $\Omega(f(x))$ and $\Theta(f(x))$ which we do not consider in this thesis. They correspond to asymptotic lower and tight bounds respectively.

We give algorithms using *pseudocode*, largely following the conventions from [10]. The pseudocode abstracts away details specific to a particular programming language but shares its overall structure with modern programming languages like C++, Java or Python. Therefore, it should also be understandable by anyone that is familiar with such a programming language. Importantly, the pseudocode can also include natural language to convey its meaning. This will be the case when it is helpful for the understanding of the ideas involved and when being closer to a programming language does not yield more insights. On the other hand, sometimes the pseudocode will appear more technical. This will be the case when the specificity is required in the analysis of the algorithm. We now give a few notational choices we made for our pseudocode.

- Assignments to variables are indicated by $\leftarrow$ and not by $=$ which instead is used to test for equality.

- Arrays are indexed into using brackets, i.e. $A[i]$ is the i-th element of the array A. An array of length l is generated using the function call `new_array(`l`)`, whereupon the memory for the array is allocated. The time to create an array is therefore proportional to its length. Once created, assigning an array to a variable happens in constant time because we assume it is passed by reference rather than by value. A "real" copy of the array A is generated by `copy(`A`)` which takes time proportional to the length of A. The function `copy` is assumed to work recursively if the array it is called on is an array of arrays. We assume that upon creation each array cell contains the special value $undefined$. We assume that all arrays are 1-origin indexing, i.e. the first element of the array A is $A[1]$ and not $A[0]$. We use brackets and colons to indicate subarrays, i.e. $A[i:j]$ denotes the subarray of A whose first entry is $A[i]$ and whose last entry is $A[j]$. Arrays can be extended by using the methods `push(`A`,`x`)` and `pushfirst(`A`,`x`)`, which insert the element x at the end or at the beginning of the array A respectively. They also take time proportional to the length of the array A. We also assume that the array A keeps track of its length which can be accessed using `length(`A`)` in constant time.

- Matrices are denoted by 2D-Array and also indexed into using brackets, i.e. $A[i,j]$ is the (i,j)-th component of the matrix A.

- Objects are defined using the keyword **structure** and contain the names of the fields such an object has as well as the field's data types. If an object a has a field b, this field is accessed via $a.b$. Further, we allow for an object oriented approach, where objects may have functions which can then access the object's fields. If a structure a has a function f, calling that function with the argument x is done via $a.f(x)$. This object oriented flavour is not required for the functionality of the algorithms in this thesis but is rather done with the intention of improving readability. As such, one can write $a.f(x)$ rather than $f(a,x)$. However, as can be seen for example with the method `push`, not all functions or methods belong to an object.

- Function and method definitions have their input arguments marked using the keyword **Input** and their output using the keyword **return**. Upon encountering the keyword **return**, a routine immediately stops. Further, arguments to functions and methods are passed by value if they are atomic data types and passed by reference otherwise.

1.20 Remark. Generally, when we speak of an algorithm's run time we consider its worst-case run time. This means, we assume the input instance to an algorithm to be such that it has to do the maximal amount of work. We could also consider the best-case or average-case run time, but because we want to get upper bounds on the run time of an algorithm, we use the worst-case run time. However, there is one case in this thesis where we consider the average-case run time which is when using *dictionaries*.

1.21 Definition. A *dictionary* is a data structure for a dynamic set that allows for insertion and deletion of elements as well as for membership tests. We assume that our dictionaries are implemented using *hash tables with chaining* which yields an average run time of $\mathcal{O}(1)$ for the dictionary operations. For more details on dictionaries and hash tables we refer to Chapter 11 in [10].

1.22 Remark. We rarely also require exponentiation and square root operations in this thesis. For arbitrary inputs, these operations are not constant time. Instead, computing a^b takes time $\mathcal{O}(\log b)$ using repeated squaring and computing $\sqrt{b}$ takes time $\mathcal{O}(M(\beta))$ using the Newton-Raphson method where b is a β-digit number and $M(\beta)$ is the time it takes to multiply two β-digit numbers. However, as we remarked previously, we consider integers of bounded size and therefore assume these operations to take constant time.

Another data structure we will need for our algorithm is a stack.

1.23 Definition. A *stack* is a data structure for a dynamic set that allows for insertion and deletion of elements according to a *last-in, first-out* or *LIFO* policy. The insertion operation is called `push` and the deletion operation is called `pop`. Additionally, a stack supports a query for whether it is empty or not. Both `push`, `pop` as well as the query for emptiness take constant time. A stack may be internally represented through arrays or linked lists. For more details we refer to Chapter 10 in [10].

2 Quantum Automorphism Groups of Graphs

For this section and the remainder of the thesis, we assume that $\Gamma = (V, E)$ is a graph with $|V| = n$ unless otherwise specified. The automorphism group $\mathrm{Aut}(\Gamma)$ of Γ is a subgroup of the symmetric group S_n consistent with the graph structure. Analogously, the quantum automorphism group $G^+_{aut}(\Gamma)$ is a quotient of the CMQG S^+_n respecting the graph structure.

2.1 Definition. The *quantum automorphism group* $G^+_{aut}(\Gamma)$ of the graph $\Gamma = (V, E)$ with $|V| = n$ is given by the CMQG $G^+_{aut}(\Gamma) = (C(G^+_{aut}(\Gamma)), u)$ where $C(G^+_{aut}(\Gamma))$ is the universal C^*-algebra defined by

$$\begin{aligned} C(G^+_{aut}(\Gamma)) = C^*(u_{ij}, i, j \in [n], 1 \mid\, & u_{ij} = u^*_{ij} = u^2_{ij}, \\ & \sum_{k=1}^{n} u_{ik} = \sum_{k=1}^{n} u_{kj} = 1, \\ & uA_\Gamma = A_\Gamma u). \end{aligned} \tag{1}$$

Here, $uA_\Gamma = A_\Gamma u$ is a shorthand for $\sum_{k=1}^n u_{ik}(A_\Gamma)_{kj} = \sum_{l=1}^n (A_\Gamma)_{il} u_{lj}$ for all $i, j \in [n]$. We say that Γ has *no quantum symmetries* if $C(G^+_{aut}(\Gamma))$ is commutative, i.e. $C(G^+_{aut}(\Gamma)) = C(\mathrm{Aut}(\Gamma))$.

The previous definition of $G^+_{aut}(\Gamma)$ is due to Banica in 2005 [2] and the definition of quantum symmetries is due to Banica and Bichon in 2006 [3]. In 2003, Bichon gave a definition for a quantum automorphism group called $G^*_{aut}(\Gamma)$ [5]. It holds that

$$\mathrm{Aut}(\Gamma) \subseteq G^*_{aut}(\Gamma) \subseteq G^+_{aut}(\Gamma)$$

as CMQGs as described in more detail in [27] and with

$$C(G^+_{aut}(\Gamma)) \cong {}^{C(S^+_n)}\!/_{<uA_\Gamma = A_\Gamma u>}$$

we further get

$$\mathrm{Aut}(\Gamma) \subseteq G^*_{aut}(\Gamma) \subseteq G^+_{aut}(\Gamma) \subseteq S^+_n.$$

In this thesis we only consider the quantum automorphism group due to Banica.

2.2 Lemma. *The relations given in Equation* (1) *imply that*

$$u_{ik}u_{il} = \delta_{k,l} u_{ik} \tag{R1}$$

$$u_{kj}u_{lj} = \delta_{k,l} u_{kj} \tag{R2}$$

$$u_{ij}u_{kl} = 0 \text{ if } i \sim k, j \not\sim l \tag{R3}$$

$$u_{ij}u_{kl} = 0 \text{ if } i \not\sim k, j \sim l \tag{R4}$$

Proof. Equations (R1) and (R2) follow immediately from the fact that the generators $u_{ik}, k \in [n]$ and $u_{kj}, k \in [n]$ are mutually orthogonal projections, where the orthogonality follows from $\sum_{k=1}^n u_{ik} = 1$ and $\sum_{k=1}^n u_{kj} = 1$ respectively.

For equation (R3), take i, k, j, l such that $i \sim k$ and $j \not\sim l$. The relation $uA_\Gamma = A_\Gamma u$ implies in particular that

$$\sum_{r=1}^{n} (A_\Gamma)_{rl} u_{ir} = \sum_{r=1}^{n} (A_\Gamma)_{ir} u_{rl}.$$

Multiplying with u_{ij} from the left yields

$$\sum_{r=1}^{n}(A_\Gamma)_{rl}u_{ij}u_{ir} = \sum_{r=1}^{n}(A_\Gamma)_{ir}u_{ij}u_{rl}$$

which due to equation (R1) simplifies to

$$\begin{aligned}(A_\Gamma)_{jl}u_{ij} &= \sum_{r=1}^{n}(A_\Gamma)_{ir}u_{ij}u_{rl}\\ \Leftrightarrow 0 &= \sum_{r\in[n],r\sim i} u_{ij}u_{rl}.\end{aligned}$$

Multiplying with u_{kl} from the right finally yields $0 = u_{ij}u_{kl}$. Equation (R4) is shown analogously. □

The relations given in the previous lemma are useful to determine relations in the quantum automorphism group of Γ. However, even further than that, Relations (R3) and (R4) give an equivalent characterization for $C(G^+_{aut}(\Gamma))$ as follows.

2.3 Proposition. *Let $\Gamma = (V, E)$ with $|V| = n$ be a graph and consider the universal C^*-algebra B defined by*

$$\begin{aligned}B = C^*(v_{ij}, i, j \in [n], 1 \mid\ & v_{ij} = v_{ij}^* = v_{ij}^2,\\ & \sum_{k=1}^{n} v_{ik} = \sum_{k=1}^{n} v_{kj} = 1,\\ & v_{ij}v_{kl} = 0 \text{ if } i \sim k, j \not\sim l,\\ & v_{ij}v_{kl} = 0 \text{ if } i \not\sim k, j \sim l).\end{aligned}$$

Then $B \cong C(G^+_{aut}(\Gamma))$.

Proof. By Lemma 2.2, the universal property of B gives a $*$-homomorphism $\varphi : B \to C(G^+_{aut}(\Gamma))$ mapping v_{ij} to u_{ij}. On the other hand, let $v = (v_{ij})$ denote the matrix with entries v_{ij}. For any i and j and using $\sum_{l=1}^{n} v_{il} = 1$, we get

$$\begin{aligned}\sum_{k=1}^{n}(A_\Gamma)_{ik}v_{kj} &= \sum_{l=1}^{n} v_{il}\sum_{k=1}^{n}(A_\Gamma)_{ik}v_{kj}\\ &= \sum_{k=1}^{n}\sum_{l=1}^{n}(A_\Gamma)_{ik}v_{il}v_{kj}.\end{aligned}$$

With $(A_\Gamma)_{ik} = 0$ for $k \not\sim i$ this can be rearranged as

$$\sum_{k=1}^{n}(A_\Gamma)_{ik}v_{kj} = \sum_{k\in[n],k\sim i}\sum_{l=1}^{n} v_{il}v_{kj}$$

and with $v_{il}v_{kj} = 0$ for $i \sim k, l \not\sim j$ we get

$$\sum_{k=1}^{n}(A_\Gamma)_{ik}v_{kj} = \sum_{k\in[n],k\sim i}\sum_{l\in[n],l\sim j} v_{il}v_{kj}.$$

On the other hand, using symmetric arguments, we get

$$\begin{aligned}\sum_{l=1}^{n} v_{il}(A_\Gamma)_{lj} &= \sum_{l=1}^{n} v_{il}(A_\Gamma)_{lj} \sum_{k=1}^{n} v_{kj} \\ &= \sum_{k=1}^{n}\sum_{l=1}^{n} (A_\Gamma)_{lj} v_{il} v_{kj} \\ &= \sum_{k=1}^{n} \sum_{l\in[n], l\sim j} v_{il} v_{kj} \\ &= \sum_{k\in[n], k\sim i} \sum_{l\in[n], l\sim j} v_{il} v_{kj}.\end{aligned}$$

Thus, we have $A_\Gamma v = v A_\Gamma$ and the universal property of $C(G_{aut}^+(\Gamma))$ gives a $*$-homomorphism $\psi : C(G_{aut}^+(\Gamma)) \to B$ mapping u_{ij} to v_{ij}. Since φ and ψ are inverse to each other, B and $C(G_{aut}^+(\Gamma))$ are isomorphic to each other. □

The previous lemma and proposition are close to the results presented in Section 3.1 in [12] but are translated into a more C^*-algebraic language. In the remainder of this section we give some tools that help with finding relations in $C(G_{aut}^+(\Gamma))$.

2.4 Remark. Denote by R the defining relations of $C(G_{aut}^+(\Gamma))$. Since the $*$-algebra $A = A(u_{ij}, i, j \in [n], 1|R)$ is dense in $C(G_{aut}^+(\Gamma))$ it suffices to show that A is commutative in order to conclude that $C(G_{aut}^+(\Gamma))$ is commutative.

2.5 Lemma. *Let A be a C^*-algebra and $a, b \in A$ projections satisfying $ab = aba$. Then $ab = ba$ holds as well.*

Proof. This follows immediately as

$$ab = aba = a^*b^*a^* = (aba)^* = (ab)^* = b^*a^* = ba.$$

□

2.6 Remark. Lemma 2.5 is not specific to $C(G_{aut}^+(\Gamma))$ but will be useful in this setting because the generators u_{ij} are projections and we are interested in finding out whether a graph has quantum symmetries, i.e. whether $C(G_{aut}^+(\Gamma))$ is commutative.

2.7 Lemma. *Let $i, j \in V$ such that $\deg(i) \neq \deg(j)$. Then $0 = u_{ij} \in C(G_{aut}^+(\Gamma))$ holds.*

Proof. This follows from straightforward computation as

$$\begin{aligned}
\deg(i) \cdot u_{ij} &= \sum_{k\in[n],k\sim i} u_{ij} \\
&= \sum_{k\in[n],k\sim i} \left(u_{ij} \sum_{l=1}^{n} u_{kl} \right) \\
&= \sum_{k\in[n],k\sim i} \left(u_{ij} \sum_{l\in[n],l\sim j} u_{kl} \right) \\
&= \sum_{k\in[n],k\sim i} \left(\sum_{l\in[n],l\sim j} u_{ij}u_{kl} \right) \\
&= \sum_{l\in[n],l\sim j} \left(\sum_{k\in[n],k\sim i} u_{ij}u_{kl} \right) \\
&= \sum_{l\in[n],l\sim j} \left(u_{ij} \sum_{k\in[n],k\sim i} u_{kl} \right) \\
&= \sum_{l\in[n],l\sim j} \left(u_{ij} \sum_{k=1}^{n} u_{kl} \right) \\
&= \sum_{l\in[n],l\sim j} u_{ij} \\
&= \deg(j) \cdot u_{ij}
\end{aligned}$$

□

2.8 Lemma. *Let $i, j, k, l \in V$ such that $d(i,k) \neq d(j,l)$. Then $0 = u_{ij}u_{kl} \in C(G^{+}_{aut}(\Gamma))$ holds.*

Proof. Assume $d(i,k) < d(j,l)$ without loss of generality and let $i \sim i_1 \sim \ldots \sim i_{d(i,k)-1} \sim k$ be a path of minimal length $d(i,k)$ from i to k. Then we get

$$\begin{aligned}
u_{ij}u_{kl} &= u_{ij} \left(\sum_{j_1=1}^{n} u_{i_1 j_1} \right) u_{kl} \\
&= u_{ij} \left(\sum_{j_1\in[n],j_1\sim j} u_{i_1 j_1} \right) u_{kl} \\
&= \ldots \\
&= u_{ij} \left(\sum_{j_1\in[n],j_1\sim j} u_{i_1 j_1} \right) \ldots \left(\sum_{j_{d(i,k)-1}\in[n],j_{d(i,k)-1}\sim j_{d(i,k)-2}} u_{i_{d(i,k)-1} j_{d(i,k)-1}} \right) u_{kl}.
\end{aligned}$$

However, since we assumed that $d(i,k) < d(j,l)$, there must be an index $r \in [d(i,k)-1]$ with $j_r \not\sim j_{r+1}$. Therefore the sum is equal to zero as required. □

Note that the previous lemma generalizes the relations mentioned in Lemma 2.2.

2.9 Definition. Let $\Gamma = (V, E)$ be a graph. We say that two automorphisms $\sigma, \tau \in \mathrm{Aut}(\Gamma)$ are *disjoint automorphisms* if and only if $\sigma(i) \neq i$ implies $\tau(i) = i$ and if $\tau(i) \neq i$ implies $\sigma(i) = i$ for all $i \in V$.

2.10 Lemma (Lemma 2.5 in [17], Theorem 2.2 in [24])**.** *Let Γ be a graph and let $\sigma, \tau \in \mathrm{Aut}(\Gamma)$ be two non-trivial, disjoint automorphisms of Γ. Then Γ has quantum symmetries.*

Proof. In order to show that Γ has quantum symmetries, we find a surjective $*$-homomorphism from $C(G^+_{aut}(\Gamma))$ to the C^*-algebra $B \coloneqq C^*(p, q, 1 \mid p = p^* = p^2, q = q^* = q^2)$ generated by two projections which is not commutative.

We define the $(n \times n)$-matrix u' via

$$u'_{ij} = \delta_{\sigma(i)j}p + \delta_{\tau(i)j}q + \delta_{ij}(1 - p - q) \in B$$

for $i, j \in [n]$. By distinguishing different cases for i and j, we see that the entries u'_{ij} are all projections again:

(1) Let $i = j$.

  (a) If $\sigma(i) = i$ and $\tau(i) = i$, then $u'_{ij} = 1$.

  (b) If $\sigma(i) \neq i$, then $\tau(i) = i$ since σ and τ are disjoint automorphisms and then $u'_{ij} = 1 - p$.

  (c) If $\tau(i) \neq i$, then $\sigma(i) = i$ since σ and τ are disjoint automorphisms and then $u'_{ij} = 1 - q$.

(2) Let $i \neq j$.

  (a) If $\sigma(i) = i$ and $\tau(i) = i$, then $u'_{ij} = 0$.

  (b) If $\sigma(i) \neq i$, then $\tau(i) = i$ since σ and τ are disjoint automorphisms and then $u'_{ij} = \delta_{\sigma(i)j}p$.

  (c) If $\tau(i) \neq i$, then $\sigma(i) = i$ since σ and τ are disjoint automorphisms and then $u'_{ij} = \delta_{\tau(i)j}q$.

Further, the rows and columns of u' sum to one:

$$\sum_{k=1}^{n} u'_{ik} = \sum_{k=1}^{n} \delta_{\sigma(i)k}p + \delta_{\tau(i)k}q + \delta_{ik}(1 - p - q) = 1$$

and

$$\sum_{k=1}^{n} u'_{kj} = \sum_{k=1}^{n} \delta_{\sigma(k)j}p + \delta_{\tau(k)j}q + \delta_{kj}(1 - p - q) = 1.$$

Finally, since σ and τ are automorphisms of Γ, we get $\sigma A_\Gamma = A_\Gamma \sigma^{-1}$ and $\tau A_\Gamma = A_\Gamma \tau^{-1}$ when considering σ and τ as permutation matrices. This means that

$$(A_\Gamma)_{\sigma(i)j} = (A_\Gamma)_{i\sigma^{-1}(j)}$$

and

$$(A_\Gamma)_{\tau(i)j} = (A_\Gamma)_{i\tau^{-1}(j)}$$

for all $i, j \in [n]$ and thus

$$\begin{aligned}
(A_\Gamma u')_{ij} &= \sum_{k=1}^{n} (A_\Gamma)_{ik} u'_{kj} \\
&= \sum_{k=1}^{n} (A_\Gamma)_{ik}\delta_{\sigma(k)j} p + (A_\Gamma)_{ik}\delta_{\tau(k)j} q + (A_\Gamma)_{ik}\delta_{kj}(1-p-q) \\
&= (A_\Gamma)_{i\sigma^{-1}(j)} p + (A_\Gamma)_{i\tau^{-1}(j)} q + (A_\Gamma)_{ij}(1-p-q) \\
&= (A_\Gamma)_{\sigma(i)j} p + (A_\Gamma)_{\tau(i)j} q + (A_\Gamma)_{ij}(1-p-q) \\
&= \sum_{k=1}^{n} \delta_{\sigma(i)k}(A_\Gamma)_{kj} p + \delta_{\tau(i)k}(A_\Gamma)_{kj} q + \delta_{ik}(A_\Gamma)_{kj}(1-p-q) \\
&= \sum_{k=1}^{n} u'_{ik}(A_\Gamma)_{kj} \\
&= (u' A_\Gamma)_{ij}
\end{aligned}$$

for all $i, j \in [n]$ and therefore $A_\Gamma u' = u' A_\Gamma$.

The universal property of $C(G^+_{aut}(\Gamma))$ then yields a $*$-homomorphism

$$\varphi : C(G^+_{aut}(\Gamma)) \to B$$

mapping u_{ij} to u'_{ij}. In particular, the cases (2) (b) and (c) show that φ is surjective and thus $C(G^+_{aut}(\Gamma))$ is not commutative. □

3 Manual Propagation of Zeros

In this section, we give concrete examples of graphs and how their quantum automorphism group is computed. We will see that by applying the relations of Lemma 2.2 and by multiplying with specific sums that equal 1, we can already determine a lot of the structure of the quantum automorphism groups. More precisely, these relations will allow us to show that certain monomials are equal to zero which in turn will allow us to conclude that some—or all—of the generators commute. A takeaway from this section should be that choosing the "correct" next relation—by which we mean a relation that will yield new insights—to apply may be non-obvious but since the relations follow a strict scheme this approach lends itself to being automated.

The manual approach starts by examining a monomial of interest and shows that it is equal to zero through algebraic transformations. On the other hand, the automated approach will start with all known zeros and propagate them among the elements of the quantum automorphism group. In this sense, the manual method is a more targeted method but relies on experience or experimentation on the user's part whereas the automated method is untargeted but applies all relations without prejudice.

3.1 Asymmetric Graph with Six Vertices

Consider the graph Γ from Example 1.4.

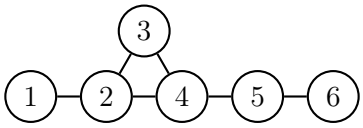


Figure 4: The graph from Example 1.4 is the smallest non-trivial graph with trivial automorphism group.

It is the smallest non-trivial graph with trivial automorphism group. Since the classical automorphism group of a graph is a CMQ-subgroup of its quantum automorphism group, this is also the smallest non-trivial graph that might have a trivial quantum automorphism group. Indeed, we have the following result.

3.1 Proposition. *The quantum automorphism group $G^+_{aut}(\Gamma)$ of the graph Γ from Example 1.4 is trivial.*

We can use the structure of the graph to immediately conclude that many of the generators of $C(G^+_{aut}(\Gamma))$ are equal to zero.

3.2 Lemma. *Let $\Gamma = (V, E)$ be the graph from Example 1.4 and let $(i, j) \in V \times V \setminus \{(1, 6), (6, 1), (2, 4), (4, 2), (3, 5), (5, 3)\}$ be a pair of distinct vertices in Γ. Then $u_{ij} = 0$.*

Proof. The graph Γ is such that for any $k \in \{1, 2, 3\}$ there are exactly two vertices $i \neq j$ with $\deg(i) = \deg(j) = k$. For $k = 1$ they are the vertices 1 and 6, for $k = 2$ they are the vertices 3 and 5 and for $k = 3$ they are the vertices 2 and 4. Therefore,

by excluding the pairs of vertices with equal degrees and by also requiring $i \neq j$, we have $\deg(i) \neq \deg(j)$. By Lemma 2.7 the claim follows. □

With this lemma, we can now conclude the proof of the previous proposition.

Proof of Proposition 3.1. In order to show that $G_{aut}^{+}(\Gamma)$ is trivial, we show that $u_{ij} = 0$ for every generator u_{ij} with $i \neq j$. Then, $u_{ii} = 1$ for all $i \in \{1, \ldots, 6\}$ follows from $\sum_{k=1}^{6} u_{ik} = 1$ and therefore $C(G_{aut}^{+}(\Gamma)) \cong \mathbb{C}$. By Lemma 3.2, we only have to consider the six elements $u_{16}, u_{24}, u_{35}, u_{42}, u_{53}$ and u_{61}.

(1) u_{16}: We use the fact that $\sum_{k=1}^{6} u_{2k} = 1$ to get

$$\begin{aligned} u_{16} &= u_{16} \sum_{k=1}^{6} u_{2k} \\ &= u_{16}u_{21} + u_{16}u_{22} + u_{16}u_{23} + u_{16}u_{24} + u_{16}u_{25} + u_{16}u_{26} \end{aligned}$$

Again by Lemma 3.2, $u_{2k} = 0$ for $k \notin \{2, 4\}$. Thus, the only surviving summands are $u_{16}u_{22}$ and $u_{16}u_{24}$. Since 1 is adjacent to 2 in Γ but 6 is neither adjacent to 2 nor to 4, Relation (R3) from Lemma 2.2 then gives $u_{16}u_{22} = 0$ and $u_{16}u_{24} = 0$ and therefore $u_{16} = 0$. We also get $u_{11} = \sum_{k=1}^{6} u_{1k} = 1$.

(2) u_{24}: We multiply with $1 = u_{11}$ to get $u_{24} = u_{24}u_{11}$. Because $2 \sim 1$ and $4 \not\sim 1$, Relation (R3) yields $u_{24}u_{11} = 0$. We also get $u_{22} = 1$.

(3) u_{35}: We multiply with $1 = u_{22}$ to get $u_{35} = u_{35}u_{22} = 0$. Because $3 \sim 2$ and $5 \not\sim 2$, Relation (R3) yields $u_{24}u_{11} = 0$. We also get $u_{33} = 1$.

(4) u_{42}: We observe that $1 = \sum_{k=1}^{6} u_{k2}$ and by Lemma 3.2 this is equal to $u_{22}+u_{42}$. However, we have previously seen that $u_{22} = 1$ and therefore $u_{42} = 0$.

(5) u_{53}: We conclude analogously to u_{42}.

(6) u_{61}: We conclude analogously to u_{42}.

□

3.3 Remark. We have relied heavily on Lemma 2.7 to simplify the computation of $C(G_{aut}^{+}(\Gamma))$ in the proof of Proposition 3.1. However, using only the relations from Lemma 2.2 and multiplying with specific sums that sum to 1 we can also show the same result. Doing so requires to use monomials of length 3 during the computations. This will be shown again in the coming section so we refrain from showing an example at this point.

3.2 Graphs with Four Vertices

We know that the graph from Example 1.4 with six vertices is the smallest graph with trivial automorphism group. Therefore, all graphs with fewer vertices have non-trivial automorphism groups and we know that their quantum automorphism groups are also non-trivial. Further, we know from Definition 2.1 that $C(G_{aut}^{+}(\Gamma))$ is a quotient of $C(S_n^{+})$ and from Propositions 1.15 and 1.16 that S_n^{+} is non-commutative if and only if $n \geq 4$. Thus, we know that the first candidates for graphs admitting quantum symmetries are graphs with four vertices.

3.4 Definition. Let $\Gamma = (V, E)$ be a simple graph. The *graph complement* of Γ is the graph $\Gamma^c = (V, E^c)$ that has the same vertices as Γ and whose edge set is given by $E^c = \{(u, v) \in V \times V \mid u \neq v, (u, v) \notin E\}$. This means that for any two vertices $u, v \in V$ with $u \neq v$ we have $u \sim v$ in Γ if and only if $u \not\sim v$ in Γ^c.

There are eleven different simple graphs with four vertices—ten of which are pairs of graphs and their graph complements and the eleventh graph whose compliment is itself again. In [25], the authors computed both $G_{aut}^+(\Gamma)$ and $G_{aut}^*(\Gamma)$ for each of these graphs. We summarize their findings in Table 2. Since we only consider the quantum automorphism group due to Banica in this thesis, we omit the results concerning Bichon's version of the quantum automorphism group. Each graph in the table is clustered with its complement. This is due to the following observation.

3.5 Lemma. *Let $\Gamma = (V, E)$ be a graph and $\Gamma^c = (V, E^c)$ its complement. Then $\mathrm{Aut}(\Gamma) = \mathrm{Aut}(\Gamma^c)$ as groups and $G_{aut}^+(\Gamma) = G_{aut}^+(\Gamma^c)$ as CMQGs.*

Proof. Let $n = |V|$ be the number of vertices in Γ, Id_n the $n \times n$-identity matrix and call B the $n \times n$-matrix with all one entries. Observe that A_{Γ^c} is given by $A_{\Gamma^c} = B - \mathrm{Id}_n - A_\Gamma$.

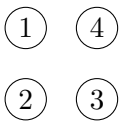
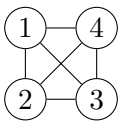
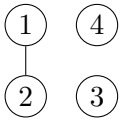
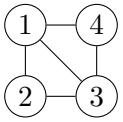
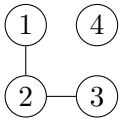
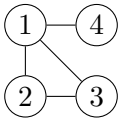
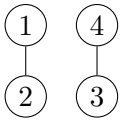
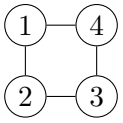
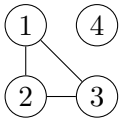
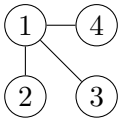
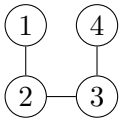
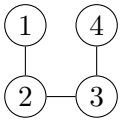
For the classical case, take $\sigma \in \mathrm{Aut}(\Gamma)$. By the definition of the automorphism group, σ commutes with A_Γ. Further, since σ is a permutation matrix, both its rows and columns sum to one and therefore $\sigma B = B = B\sigma$. Thus,

$$\sigma A_{\Gamma^c} = \sigma B - \sigma\,\mathrm{Id}_n - \sigma A_\Gamma = B\sigma - \mathrm{Id}_n\,\sigma - A_\Gamma\sigma = A_{\Gamma^c}\sigma$$

and therefore $\sigma \in \mathrm{Aut}(\Gamma^c)$. With $(\Gamma^c)^c = \Gamma$ we get the first claim.

Next, take $G_{aut}^+(\Gamma) = (C(G_{aut}^+(\Gamma)), u)$ and $G_{aut}^+(\Gamma^c) = (C(G_{aut}^+(\Gamma^c)), v)$ as the quantum automorphism groups of Γ and its complement. Since the rows and columns of u sum to one again, u also commutes with A_{Γ^c}. The universal property of $C(G_{aut}^+(\Gamma))$ then gives a $*$-homomorphism mapping u_{ij} to v_{ij} which is obviously surjective. Therefore, $G_{aut}^+(\Gamma^c) \subseteq G_{aut}^+(\Gamma)$ and using the same symmetry argument as in the classical case, we get the second claim. □

Table 2: Automorphism groups and quantum automorphism groups of graphs without loops and without multiple edges on four points.

Γ	Γ^c	$\mathrm{Aut}(\Gamma) = \mathrm{Aut}(\Gamma^c)$	$G^+_{aut}(\Gamma) = G^+_{aut}(\Gamma^c)$
		S_4	S_4^+
		$\mathbb{Z}_2 \times \mathbb{Z}_2$	$\widehat{\mathbb{Z}_2 * \mathbb{Z}_2}$
		$\mathbb{Z}_2$	$\mathbb{Z}_2$
		D_4	H_2^+
		S_3	S_3
		$\mathbb{Z}_2$	$\mathbb{Z}_2$

For concrete computations and more details on the quantum automorphism groups in Table 2 we refer back to the original paper. However, we want to highlight the way the authors determined the graphs in the third and fifth row to not have quantum symmetries. In either case, the authors determined that

$$u_{i4} = 0 = u_{4i}$$

in $C(G^+_{aut}(\Gamma))$ with $i \in \{1, 2, 3\}$ using a statement that follows directly from the relations given in Lemma 2.2 and which we generalized in Lemma 2.7. From this they infer that $G^+_{aut}(\Gamma)$ is a compact matrix quantum subgroup of S_3^+ which is commutative. While this certainly works, we can also arrive at $G^+_{aut}(\Gamma)$ being commutative through purely algorithmic means. For example, take the graph Γ^c in the fifth row of Table 2, which is sometimes referred to as the full bipartite graph $K_{1,3}$ or as the *claw graph*.

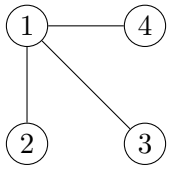


Figure 5: The claw graph $K_{1,3}$

3.6 Proposition. *The claw graph $K_{1,3}$ does not admit quantum symmetries.*

Proof. We need to show that $C(G^+_{aut}(K_{1,3}))$ is commutative. We partition the set of pairs of generators into four sets P_1, P_2, P_3, P_4 with

(1) $P_1 = \{(u_{ij}, u_{kl}) \mid i,j,k,l \in \{1,2,3,4\}, 1 \in \{i,j,k,l\}\}$

(2) $P_2 = \{(u_{ij}, u_{ij}) \mid i,j \in \{2,3,4\}\}$

(3) $P_3 = \{(u_{ij}, u_{ik}), (u_{ji}, u_{ki}) \mid i,j,k \in \{2,3,4\}, j \neq k\}$

(4) $P_4 = \{(u_{ij}, u_{kl}) \mid i,j,k,l \in \{2,3,4\}, k \neq i, l \neq j\}$

Clearly, P_1 is disjoint with all other sets because one of the indices is required to be 1 and all other sets do not allow 1 as an index. The set P_2 is disjoint with the sets P_3 and P_4 because it is made up of diagonal pairs and P_3 and P_4 do not contain diagonal pairs by definition. Finally, the sets P_3 and P_4 are disjoint because the generators in the pairs in P_3 either have the same first or second indices and this is prohibited in P_4. Therefore the sets are pairwise disjoint.

Next, we note that there are $16^2 = 256$ pairs of generators and

(1) P_1 has 175 elements:

- Fixing $i = 1$, there are $4^3 = 64$ elements.
- Fixing $i \neq 1, j = 1$, there are $3 \cdot 4^2 = 48$ elements.
- Fixing $i, j \neq 1, k = 1$, there are $3^2 \cdot 4 = 36$ elements.
- Fixing $i, j, k \neq 1, l = 1$, there are $3^3 = 27$ elements.

(2) P_2 has $3^2 = 9$ elements because we have a choice of $i, j \in \{2,3,4\}$.

(3) P_3 has $3^2 \cdot 2^2 = 36$ elements because we have a choice of $i, j \in \{2,3,4\}$ and $k \in \{2,3,4\} \setminus \{j\}$ and we have two elements for each such choice of i, j, k.

(4) P_4 has $3^2 \cdot 2 \cdot 2 = 36$ elements because we have a choice of $i, j \in \{2,3,4\}$ and $k \in \{2,3,4\} \setminus \{i\}, l \in \{2,3,4\} \setminus \{j\}$.

Therefore $\bigsqcup_{i \in \{1,2,3,4\}} P_i$ has $175 + 9 + 36 + 36 = 256$ elements and is a partition of the set of all pairs of generators indeed.

We now show for each set P_i that its elements are pairs of commuting generators.

(1) P_1: We have $\deg(1) = 3$ and $\deg(i) = 1$ for $i \in \{2,3,4\}$. By Lemma 2.7 we get $u_{1i} = 0$ and $u_{i1} = 0$ for $i \in \{2,3,4\}$ and therefore also $u_{11} = 1$. Thus, u_{1i} and u_{i1} are zero or one for $i \in \{1,2,3,4\}$ and commute with all generators. Since each element in P_1 contains u_{1i}, u_{i1}, or both, these elements are pairs of commuting generators.

(2) P_2: The elements of P_2 consist of pairs of the same generator and therefore trivially commute.

(3) P_3: Take an element of the form $(u_{ij}, u_{ik}) \in P_3$. By definition we have $j \neq k$ and therefore Relation (R1) yields $u_{ij}u_{ik} = 0 = u_{ik}u_{ij}$. For elements of the form $(u_{ji}, u_{ki}) \in P_3$, Relation (R2) gives the same result.

(4) P_4: Let $(u_{ij}, u_{kl}) \in P_4$ and consider the monomial $u_{ij}u_{kl}$. By definition $i \in \{2,3,4\}$ and $k \in \{2,3,4\} \setminus \{i\}$ and we denote by $s \in \{2,3,4\} \setminus \{i,k\}$ the third element of the set. Then, by multiplying with $1 = \sum_{p=1}^{4} u_{pj}$, we compute

$$\begin{aligned} u_{ij}u_{kl} &= u_{ij}u_{kl}\sum_{p=1}^{4} u_{pj} \\ &= u_{ij}u_{kl}u_{1j} + u_{ij}u_{kl}u_{ij} + u_{ij}u_{kl}u_{kj} + u_{ij}u_{kl}u_{sj}. \end{aligned}$$

We examine the summands individually.

- $u_{ij}u_{kl}u_{1j}$: We have $\deg(1) = 3$ and $\deg(j) = 1$, so Lemma 2.7 gives $u_{1j} = 0$ and therefore $u_{ij}u_{kl}u_{1j} = 0$.
- $u_{ij}u_{kl}u_{kj}$: We have $l \neq j$ by definition, so Relation (R1) gives $u_{kl}u_{kj} = 0$ and therefore $u_{ij}u_{kl}u_{kj} = 0$.
- $u_{ij}u_{kl}u_{sj}$: Recall that $i \neq s$ and therefore $u_{ij}u_{sj} = 0$ by Relation (R2). Then, by inserting $1 = \sum_{p=1}^{4} u_{pl}$, we get

$$\begin{aligned} 0 &= u_{ij}u_{sj} \\ &= u_{ij}\sum_{p=1}^{4} u_{pl}u_{sj} \\ &= u_{ij}u_{1l}u_{sj} + u_{ij}u_{il}u_{sj} + u_{ij}u_{kl}u_{sj} + u_{ij}u_{sl}u_{sj}. \end{aligned}$$

  Again, we examine these summands individually.
  - $u_{ij}u_{1l}u_{sj}$: As before, $u_{1l} = 0$ by Lemma 2.7, so $u_{ij}u_{1l}u_{sj} = 0$.
  - $u_{ij}u_{il}u_{sj}$: As before, $u_{ij}u_{il} = 0$ by Relation (R1), so $u_{ij}u_{il}u_{sj} = 0$.
  - $u_{ij}u_{sl}u_{sj}$: As before, $u_{sl}u_{sj} = 0$ by Relation (R1), so $u_{ij}u_{sl}u_{sj} = 0$.

  Thus, we have $0 = u_{ij}u_{sj} = u_{ij}u_{kl}u_{sj}$.

Going back to our initial element $u_{ij}u_{kl}$ we thus have

$$u_{ij}u_{kl} = u_{ij}u_{kl}\sum_{p=1}^{4} u_{pj} = u_{ij}u_{kl}u_{ij}$$

and Lemma 2.5 yields that u_{ij} and u_{kl} commute.

We have thus shown that any two generators of $C(G_{aut}^{+}(K_{1,3}))$ commute and so $K_{1,3}$ does not have quantum symmetries. □

3.3 Petersen Graph

The Petersen graph P is a vertex-transitive graph with 10 vertices whose importance in graph theory can be summarized by the following quote by Donald Knuth [16]: It is "a remarkable configuration that serves as a counterexample to many optimistic predictions about what might be true for graphs in general.". For general statements regarding the Petersen graph we refer to [14].

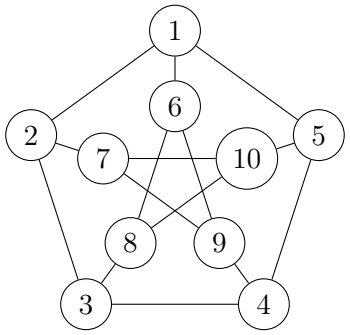


Figure 6: The Petersen graph P

Being vertex-transitive, we cannot apply Lemma 2.7 to find generators that are equal to zero. Still, using the only the defining relations from the quantum automorphism group, the relations from Lemma 2.2, and Lemma 2.5, we can compute that the Petersen graph has no quantum symmetries. We give a short review of the proof from [23] here. The proof is split in to two parts. First, we show that u_{ij} and u_{kl} commute for $i \sim k$ and $j \sim l$ and second, that u_{ij} and u_{kl} commute for $i \not\sim k$ and $j \not\sim l$. All other cases are already covered by the relations from Lemma 2.2. The proof also relies on the fact that the Petersen graph is 3-*regular*, meaning that each vertex has degree 3, and even *strongly regular* with $\lambda = 0$ and $\mu = 1$, meaning that any two adjacent vertices have no common neighbours and any two non-adjacent vertices have exactly one common neighbour.

3.7 Proposition (Theorem 3.2 in [23])**.** *Let i, j, k, l be vertices of the Petersen graph P with $i \sim k$ and $j \sim l$. Then $u_{ij}u_{kl} = u_{kl}u_{ij}$ in $C(G^+_{aut}(P))$.*

Proof. (1) Let s be a vertex with $s \sim l, s \neq j$. Then, vertex l is a common neighbour of both j and s by assumption. Using the strong regularity of the Petersen graph, this yields that j and s are not adjacent and that l is their unique common neighbour. Thus, for $r \neq l$, either $u_{ij}u_{kr} = 0$ or $u_{kr}u_{is} = 0$ by Relation (R3). Further, $u_{ij}u_{is} = 0$ holds by Relation (R1) since we assumed that $s \neq j$. Using $\sum_{r=1}^{10} u_{kr} = 1$ then gives

$$0 = u_{ij}u_{is} = u_{ij}\sum_{r=1}^{10} u_{kr}u_{is} = u_{ij}u_{kl}u_{is}.$$

(2) Using $\sum_{s=1}^{10} u_{is} = 1$ we get

$$u_{ij}u_{kl} = u_{ij}u_{kl}\sum_{s=1}^{10} u_{is}$$

and, because we have $i \sim k$, Relation (R3) yields that $u_{kl}u_{is} = 0$ for all $s \not\sim l$. This gives

$$u_{ij}u_{kl} = \sum_{s\in[10],s\sim l} u_{ij}u_{kl}u_{is}.$$

Now, using Step (1), we get

$$u_{ij}u_{kl} = u_{ij}u_{kl}u_{ij}.$$

Then, Lemma 2.5 yields the proof.

□

3.8 Proposition (Theorem 3.3 in [23]). *Let i, j, k, l be vertices of the Petersen graph P with $i \not\sim k, j \not\sim l$ and $i \neq k, j \neq l$. Then $u_{ij}u_{kl} = u_{kl}u_{ij}$ in $C(G^+_{aut}(P))$.*

Proof. (1) By the strong regularity of the Petersen graph, there is exactly one common neighbour s of i and k and one common neighbour t of j and l. Since the Petersen graph is 3-regular, t has exactly one remaining neighbour which we denote by q. Using Proposition 3.7, we have

$$u_{ij}u_{st} = u_{st}u_{ij} \tag{1*}$$

and

$$u_{st}u_{kl} = u_{kl}u_{st}. \tag{2*}$$

(2) By $\sum_{r=1}^{10} u_{sr} = 1$, we have

$$u_{ij}u_{kl} = u_{ij}\sum_{r=1}^{10} u_{sr}u_{kl}.$$

Let $r \neq t$. Since t is the only common neighbour of j and l, either $r \not\sim j$ or $r \not\sim l$ and thus $u_{ij}u_{sr} = 0$ or $u_{sr}u_{kl} = 0$ by Relation (R3). Thus, we have

$$u_{ij}u_{kl} = u_{ij}\sum_{r=1}^{10} u_{sr}u_{kl} = u_{ij}u_{st}u_{kl}.$$

(3) By using Step (2) and Equation (2*), we have

$$u_{ij}u_{kl} = u_{ij}u_{st}u_{kl} = u_{ij}u_{kl}u_{st}. \tag{3*}$$

Multiplying with $\sum_{p=1}^{10} u_{ip} = 1$ yields

$$u_{ij}u_{kl} = u_{ij}u_{kl}u_{st}\sum_{p=1}^{10} u_{ip}.$$

Now, observe that $s \sim i$. Therefore, $u_{st}u_{ip} = 0$ for all $p \not\sim t$ by Relation (R3). By Step (1), we know that the neighbours of t are j, l and q. Therefore, we now have

$$u_{ij}u_{kl} = u_{ij}u_{kl}u_{st}(u_{ij} + u_{il} + u_{iq})$$

and by applying Equation (2*) again, we have

$$u_{ij}u_{kl} = u_{ij}u_{st}u_{kl}(u_{ij} + u_{il} + u_{iq}).$$

We examine the summands individually:

- $u_{ij}u_{st}u_{kl}u_{il}$: Since we have assumed $i \neq k$, Relation (R2) yields

$$0 = u_{kl}u_{il} = u_{ij}u_{st}u_{kl}u_{il}.$$

- $u_{ij}u_{st}u_{kl}u_{iq}$: Consider $u_{ij}u_{st}u_{iq}$. Multiplying with $\sum_{a=1}^{10} u_{ka} = 1$ yields

$$u_{ij}u_{st}u_{iq} = u_{ij}u_{st}\sum_{a=1}^{10} u_{ka}u_{iq}.$$

  As before, k and s are adjacent and therefore $u_{st}u_{ka} = 0$ for all $a \not\sim t$ by Relation (R3). Again, the neighbours of t are j, l and q. Thus, we have

$$u_{ij}u_{st}u_{iq} = u_{ij}u_{st}(u_{kj} + u_{kl} + u_{kq})u_{iq}.$$

  We now examine these summands individually:

  - $u_{ij}u_{st}u_{kj}u_{iq}$: Since $s \sim k$ and $t \sim j$, Proposition 3.7 yields $u_{st}u_{kj} = u_{kj}u_{st}$ and thus

$$u_{ij}u_{st}u_{kj}u_{iq} = u_{ij}u_{kj}u_{st}u_{iq}.$$

    Since we have assumed $i \neq k$, Relation (R2) then yields

$$0 = u_{ij}u_{kj} = u_{ij}u_{kj}u_{st}u_{iq}.$$

  - $u_{ij}u_{st}u_{kq}u_{iq}$: As with the previous summand, Relation (R2) directly yields

$$0 = u_{kq}u_{iq} = u_{ij}u_{st}u_{kq}u_{iq}.$$

  Therefore, we get

$$u_{ij}u_{st}u_{iq} = u_{ij}u_{st}u_{kl}u_{iq}.$$

  Now, since $j \neq q$, using Relation (R1) and applying Equation (1*) again, we get

$$0 = u_{ij}u_{iq} = u_{ij}u_{st}u_{iq} = u_{ij}u_{st}u_{kl}u_{iq}$$

  for the summand we were examining.

Thus, we now have

$$u_{ij}u_{kl} = u_{ij}u_{st}u_{kl}u_{ij}.$$

Finally, applying Equation (3*) yields

$$u_{ij}u_{kl} = u_{ij}u_{kl}u_{ij}$$

and then Lemma 2.5 gives the desired result.

□

3.9 Remark. Unlike in the examples from Sections 3.1 and 3.2, the proof of Proposition 3.8 did not exclusively rely on propagating zeros and optionally applying one of the lemmas of Section 2, even though it appears very similar to the other proofs. This proof uses the fact, that the non-zero monomial $u_{ij}u_{kl}$ is equal to certain other monomials to ultimately conclude that it is equal to $u_{ij}u_{kl}u_{ij}$ and is thus not suited for the automated approach we hinted at in the introduction of this section.

This automated approach would instead have to find that for example all monomials $u_{ij}u_{kl}u_{ir}$ are equal to zero for $r \neq j$ and could then also conclude that $u_{ij}u_{kl}$ is equal to $u_{ij}u_{kl}u_{ij}$ with Lemma 2.5. It is in general not clear that this can be achieved using only this propagation of zeros but finding that such an approach is fruitful for the Petersen graph would be a good indicator that the approach may also be powerful enough for many other graphs of interest.

4 Algorithm Design

As mentioned previously, there are already algorithmic methods used for working with quantum automorphism groups of graphs. They rely on Gröbner bases of the ideal of relations used to define the C^*-algebra $C(G^+_{aut}(\Gamma))$ which then can be used to answer for example membership in that ideal. In order to compute the Gröbner basis in a non-commutative setting a generalized Buchberger's algorithm (or a more optimized algorithm like F4 or F5) are used. Unlike in the commutative case, the algorithm is only guaranteed to terminate if the ideal has a finite Gröbner basis. However, the problem of deciding whether an ideal has a finite Gröbner basis is undecidable. Further, the problems of the classical Buchberger's algorithm remain—mainly that intermediate monomials may grow very large. For an overview on Gröbner bases we refer to [19].

Instead of following in this path, we propose an algorithmic approach that mimcs the manual approach to computing commutation relations seen in Section 3. The general idea of the algorithm is "propagating zeros" using relations and properties of the quantum automorphism group of a graph. More precisely, this means that given the knowledge that a—or some—monomials are equal to zero certain others are also zero. The propagation rules in question are the following:

Propagation rules

(Pi) Let $u = u_{i_1j_1} \dots u_{i_mj_m} \in C(G^+_{aut}(\Gamma))$ be a monomial of length m such that $u = 0$. Then for any generator u_{kl} of $C(G^+_{aut}(\Gamma))$ the monomials uu_{kl} and $u_{kl}u$ are also equal to zero.

(Pii) Let $r \in [m]$ and $i_1, \dots, i_{r-1}, i_{r+1}, \dots, i_m \in [n]$ and $j_1, \dots, j_m \in [n]$ be fixed indices such that $u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{kj_r} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m} = 0$ for all $k \in [n]$. Then the monomial $u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m}$ is equal to zero.

(Piii) Let $s \in [m]$ and $i_1, \dots, i_m \in [n]$ and $j_1, \dots, j_{s-1}, j_{s+1}, \dots, j_m \in [n]$ be fixed indices such that $u_{i_1j_1} \dots u_{i_{s-1}j_{s-1}} u_{i_sl} u_{i_{s+1}j_{s+1}} \dots u_{i_mj_m} = 0$ for all $l \in [n]$. Then the monomial $u_{i_1j_1} \dots u_{i_{s-1}j_{s-1}} u_{i_{s+1}j_{s+1}} \dots u_{i_mj_m}$ is equal to zero.

(Piv) Let $u_{i_1j_1} \dots u_{i_mj_m} = 0$ and $r \in [m-1]$ such that $u_{i_rj_r} u_{i_{r+1}j_{r+1}} = u_{i_{r+1}j_{r+1}} u_{i_rj_r}$. Then $u_{i_1j_1} \dots u_{i_{r+1}j_{r+1}} u_{i_rj_r} \dots u_{i_mj_m}$ is equal to zero.

(Pv) Let $r \in [m]$ and $i_1, \dots, i_{r-1}, i_{r+1}, \dots, i_m \in [n]$ and $j_1, \dots, j_m \in [n]$ be fixed indices such that $u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{kj_r} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m} = 0$ for all $k \in [n]$ but one k' and such that $u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m} = 0$. Then the monomial $u_{i_1j_1} \dots u_{k'j_r} \dots u_{i_mj_m}$ is equal to zero.

(Pvi) Let $s \in [m]$ and $i_1, \dots i_m \in [n]$ and $j_1, \dots j_{s-1}, j_{s+1}, \dots, j_m \in [n]$ be fixed indices such that $u_{i_1j_1} \dots u_{i_{s-1}j_{s-1}} u_{i_sl} u_{i_{s+1}j_{s+1}} \dots u_{i_mj_m} = 0$ for all $l \in [n]$ but one l' and such that $u_{i_1j_1} \dots u_{i_{s-1}j_{s-1}} u_{i_{s+1}j_{s+1}} \dots u_{i_mj_m} = 0$. Then the monomial $u_{i_1j_1} \dots u_{i_sl'} \dots u_{i_mj_m}$ is equal to zero.

We also add the following rule that does not propagate a zero but rather knowledge of commutation.

(Ci) Let $u = u_{ij}u_{kl} \in C(G^+_{aut}(\Gamma))$ be a monomial of length two such that $u = 0$. Then u_{ij} and u_{kl} commute.

4.1 Remark. The rules follow directly from the relations of $C(G^+_{aut}(\Gamma))$. In order to make sense of them in a more mathematical sense, we give the following explanations.

(1) Rules (Pi), (Piv) and (Ci) are trivially true.

(2) Rule (Pii) codifies the following. Let $u_{i_1j_1} \dots u_{kj_r} \dots u_{i_mj_m} = 0$ for all $k \in [n]$. Then, with $\sum_{k=1}^n u_{kj_r} = 1$ we compute

$$\begin{aligned} 0 &= \sum_{k=1}^{n} u_{i_1j_1} \dots u_{kj_r} \dots u_{i_mj_m} \\ &= u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} \left(\sum_{k=1}^{n} u_{kj_r} \right) u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m} \\ &= u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m}. \end{aligned}$$

Rule (Piii) holds analogously for the second index.

(3) Rule (Pv) codifies the following. Let $u_{i_1j_1} \dots u_{kj_r} \dots u_{i_mj_m} = 0$ for all $k \in [n]$ but one k' and let $u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m} = 0$. Then, with $\sum_{k=1}^n u_{kj_r} = 1$ we compute

$$\begin{aligned} 0 &= u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m} \\ &= u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} \left(\sum_{k=1}^{n} u_{kj_r} \right) u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m} \\ &= u_{i_1j_1} \dots u_{k'j_r} \dots u_{i_mj_m}. \end{aligned}$$

Rule (Pvi) holds analogously for the second index.

As one can see, both Rule (Pii) and (Pv) use the same relation at their core but "in opposite directions".

The propagation rules are the integral part of this approach. However, in order to apply them for once and in order to find commutation relations (other than the ones covered by Rule (Ci)) we also need additional building blocks. The first one are monomials we can determine to be zero due to some properties of the graph. We denote these monomials as *start zeros*. The second building block we need are additional criteria that take into account some monomials being zero and yield commutation relations. Since this second step requires knowledge of all (or at least many) zeros found by the application of the propagation rules it cannot happen during the propagation but must rather happen afterwards. Therefore, we refer to it as the *post-processing* step. Both of these will use the criteria introduced in Section 2. With this, the broad outline of our algorithm is the following:

(1) Find start zeros.

(2) Apply propagation rules.

(3) Find commutations in post-processing.

In a more technical sense, the propagation of zeros is realized by building and traversing a hypergraph called the *monomial graph* where the vertices of the monomial graph correspond to monomials in $C(G_{aut}^{+}(\Gamma))$ and the hyperedges of the monomial graph correspond to the propagation rules. There is therefore an additional first step in the outline of our algorithm.

(1) Create monomial graph.

(2) Find start zeros.

(3) Apply propagation rules.

(4) Find commutations in post-processing.

Finding start zeros then can be thought of as selecting vertices in the monomial graph and applying the propagation rules corresponds to a hypergraph traversal algorithm with the selected vertices as start points.

4.2 Remark. When examining the propagation rules, we notice that they rely only on the relations from S_n^+ and do not take the relation $uA_\Gamma = A_\Gamma u$ into consideration. This second relation instead is used when selecting the start zeros.

4.3 Remark. If one were to simply apply the propagation rules freely and unguided, the monomials in question may become arbitrarily large due to Rule (Pi). Then, the monomial graph would also become infinitely large and a traversal algorithm would never terminate. This would negate any advantage to Gröbner basis approaches. Therefore, we need to impose additional restrictions.

A natural way for this is to limit the size of monomials we want to allow. The monomial graph therefore only relies on the number of vertices of Γ and the maximal length of monomials. This limit comes with its own trade off—the algorithm is guaranteed to terminate and, depending on the chosen maximal length, very fast but the results have to be interpreted in the following way: If the algorithm returns that the quantum automorphism group of the graph is commutative this is true. If the algorithm does not return that the quantum automorphism group is commutative it is not clear whether choosing a larger maximal length of monomials would yield a different result or whether the graph does indeed have quantum symmetries.

The remainder of this section is dedicated to the design of the algorithm with the data structures used in Section 4.1, the creation of the monomial graph in Section 4.2, finding start zeros in Section 4.3, applying the propagation rules in Section 4.4 and the post-processing in Section 4.5.

4.1 Data Structures

We first give a precise definition for the monomial graph which we will then use to translate it into a data structure.

4.4 Definition. Let $n, k \in \mathbb{N}$. The (n,k)-monomial graph is a tuple $M_{n,k} = (V, Com, E_F, E_B, E_C, E_S, F_C)$ where

- V is a set of vertices such that there is a vertex associated to any monomial made up of the n^2 generators of $C(G^+_{aut}(\Gamma))$ of length up to k. We use "the monomial u" as a shorthand for "the vertex associated to monomial u" whenever it is clear from context.

- Com is a set of vertices such that there is a vertex $Com(u_{ij}, u_{kl})$ for any unordered pair (u_{ij}, u_{kl}) of generators of $C(G^+_{aut}(\Gamma))$. Thus, we implicitly identify $Com(u_{ij}, u_{kl})$ with $Com(u_{kl}, u_{ij}$. We call such a vertex a *commutation vertex*.

- $E_F \subseteq V \times V$ is the set of all edges (u, uu_{kl}) and $(u, u_{kl}u)$ where u is a monomial of length $m < k$ and u_{kl} is a generator of $C(G^+_{aut}(\Gamma))$.

- $E_B \subseteq 2^V \times V$ is the set of all hyperedges $(B^{(i)}_{u,r}, \hat{u})$ and $(B^{(j)}_{u,r}, \hat{u})$ where u is a monomial $u_{i_1j_1} \dots u_{i_mj_m}$ of length $2 \leq m \leq k$, $r \leq m$ denotes a position, $\hat{u} = u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m}$ is u without the r-th generator and $B^{(i)}_{u,r} = \{u_{i_1j_1} \dots u_{i'j_r} \dots u_{i_mj_m} \mid i' \in [n]\}$ and $B^{(j)}_{u,r} = \{u_{i_1j_1} \dots u_{i_rj'} \dots u_{i_mj_m} \mid j' \in [n]\}$ are sets of monomials that correspond to u except for the i or j component of the r-th generator respectively (including u).

- $E_C \subseteq 2^{V \cup Com} \times V$ is the set of all hyperedges $(C_{u,r}, u^{(C)}_r)$ where u is a monomial $u_{i_1j_1} \dots u_{i_mj_m}$ of length $2 \leq m \leq k$, $r < m$ denotes a position, $u^{(C)}_r = u_{i_1j_1} \dots u_{i_{r+1}j_{r+1}} u_{i_rj_r} \dots u_{i_mj_m}$ is u with the r-th and $(r+1)$-th generator swapped and $C_{u,r} = \{u, Com(u_{i_rj_r}, u_{i_{r+1}j_{r+1}})\}$ are sets consisting of a monomial u and the commutation vertex corresponding to the generators at positions r and $r+1$ in u.

- $E_S \subseteq 2^V \times V$ is the set of all hyperedges $(S^{(i)}_{u,r}, u)$ and $(S^{(j)}_{u,r}, u)$ where u is a monomial $u_{i_1j_1} \dots u_{i_mj_m}$ of length $2 \leq m \leq k$, $r \leq m$ denotes a position and $S^{(i)}_{u,r} = \{u_{i_1j_1} \dots u_{i'j_r} \dots u_{i_mj_m} \mid i' \in [n] \setminus i_r\} \cup \{\hat{u}\} = (B^{(i)}_{u,r} \setminus \{u\}) \cup \{\hat{u}\}$ and $S^{(j)}_{u,r} = \{u_{i_1j_1} \dots u_{i_rj'} \dots u_{i_mj_m} \mid j' \in [n] \setminus j_r\} \cup \{\hat{u}\} = (B^{(j)}_{u,r} \setminus \{u\}) \cup \{\hat{u}\}$ are sets of monomials that correspond to u except for the i or j component of the r-th generator respectively (excluding u) united with the monomial $\hat{u} = u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m}$.

- $F_C \subseteq V \times Com$ is the set of all edges $(u_{ij}u_{kl}, Com(u_{ij}, u_{kl}))$ where $u_{ij}u_{kl}$ is a monomial of length 2.

We call $f \in E_F$ a *forward edge*, $b \in E_B$ a *backward edge*, $c \in E_C$ a *commutation edge* and $s \in E_S$ a *sum edge*. We call $f_C \in F_C$ a *commutation propagation edge*. For the sets $S^{(i)}_{u,r}$ and $S^{(j)}_{u,r}$, we call the unique monomial $\hat{u}$ that has length $m-1$ the *short monomial* and all other monomials *long monomials* of the respective set. Further, we call the sets $B^{(i)}_{u,r}$ and $B^{(j)}_{u,r}$ *backward vertices* and the sets $S^{(i)}_{u,r}$ and $S^{(j)}_{u,r}$ *sum vertices*.

4.5 Remark. The backward vertices are ambiguous in their naming in the sense that by their definition any monomial $u' \in B^{(i)}_{u,r}$ may be used to name them. More precisely, this means that for monomials $u_1, \dots, u_n \in B^{(i)}_{u,r}$ we have that

$$B^{(i)}_{u,r} = B^{(i)}_{u_1,r} = \dots = B^{(i)}_{u_n,r}.$$

The same holds for the sets $B^{(j)}_{u,r}$ but not for the sum vertices $S^{(i)}_{u,r}$ and $S^{(j)}_{u,r}$. Here, the monomial u in the index is the one monomial *not* included. However, using the equality from above, we can write the sum vertex as

$$S^{(i)}_{u,r} = (B^{(i)}_{u,r} \setminus \{u\}) \cup \{\hat{u}\} = (B^{(i)}_{u_1,r} \setminus \{u\}) \cup \{\hat{u}\} = \ldots = (B^{(i)}_{u_n,r} \setminus \{u\}) \cup \{\hat{u}\}$$

where $\hat{u}$ is the short monomial.

4.6 Remark. The monomial graph is a hypergraph in the sense that edges may connect sets of vertices to other sets of vertices (although the ranges of edges in the monomial graph always consist of just single vertices). However, it does not conform precisely to our definition of a hypergraph given in Definition 1.6. In order to fit this definition we would for example have to write $(\{u\}, \{uu_{kl}\})$ for the forward edge (u, uu_{kl}) and $(B^{(i)}_{u,r}, \{\hat{u}\})$ for the backward edge $(B^{(i)}_{u,r}, \hat{u})$ which would only reduce legibility further. This can be thought of as a notational simplification.

4.7 Remark. The (n, k)-monomial graph encodes the propagation rules (Pi)–(Ci) in a natural fashion. Each monomial and commutator of generators corresponds to a vertex or commutation vertex respectively and each rule corresponds to a type of edge in the monomial graph. We have the following correspondence between propagation rules and types of edges in the monomial graph.

(1) Rule (Pi) corresponds to the edges (u, uu_{kl}) and $(u, u_{kl}u)$ in E_F.

(2) Rules (Pii) and (Piii) correspond to elements of E_B.

(3) Rule (Piv) corresponds to elements of E_C.

(4) Rules (Pv) and (Pvi) correspond to elements of E_S.

(5) Rule (Ci) corresponds to propagation edges in F_C.

Therefore, propagating zeros according to the rules outlined is translated to graph exploration of the (n, k)-monomial graph.

4.8 Example. The (n, k)-monomial graph grows large quickly and has lots of connections and showing it in its entirety graphically is not practical even for small values of n and k. Therefore, we show only some representative parts of such a monomial graph to give an intuitive idea of its structure here. Consider the $(4, 3)$-monomial graph. Starting with the monomial u_{11}, we first show all of the vertices reached by forward edges and some of the vertices subsequently reached by forward edges from the monomial $u_{11}u_{11}$ in Figure 7.

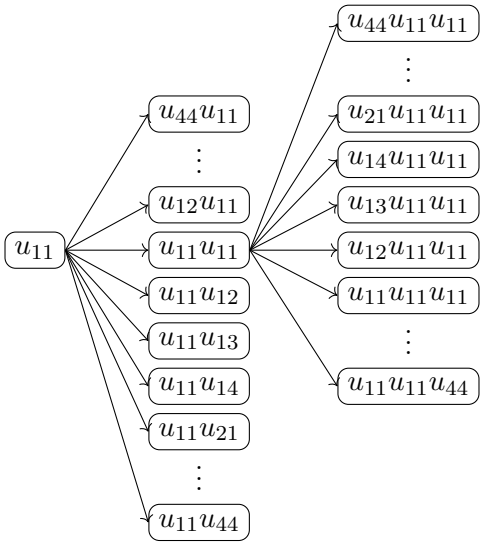


Figure 7: Part of the $(4, 3)$-monomial graph, showing only forward edges in black.

Next, we add the commutation vertex $Com(u_{11}, u_{44})$ and the commutation propagation edges $(u_{11}u_{44}, Com(u_{11}, u_{44}))$ and $(u_{44}u_{11}, Com(u_{44}, u_{11}))$ going toward it in Figure 8.

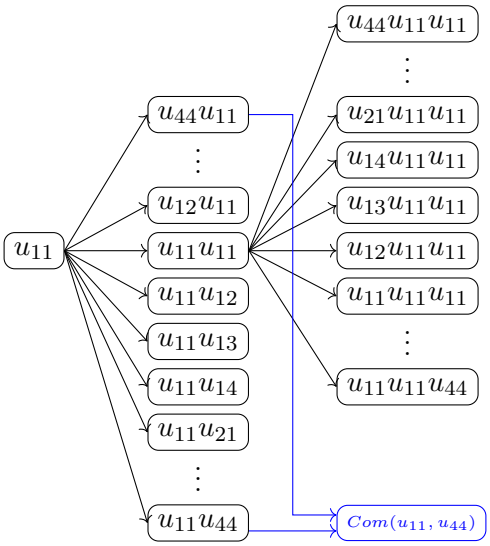


Figure 8: Part of the $(4, 3)$-monomial graph, now also showing a commutation vertex and the corresponding commutation propagation edges in blue.

Next, we mark the backward vertices $B^{(j)}_{u_{11}u_{11},2}$ and $B^{(j)}_{u_{11}u_{11}u_{11},1}$ and the backward edges $(B^{(j)}_{u_{11}u_{11},2}, u_{11})$ and $(B^{(j)}_{u_{11}u_{11}u_{11},1}, u_{11}u_{11})$ in Figure 9.

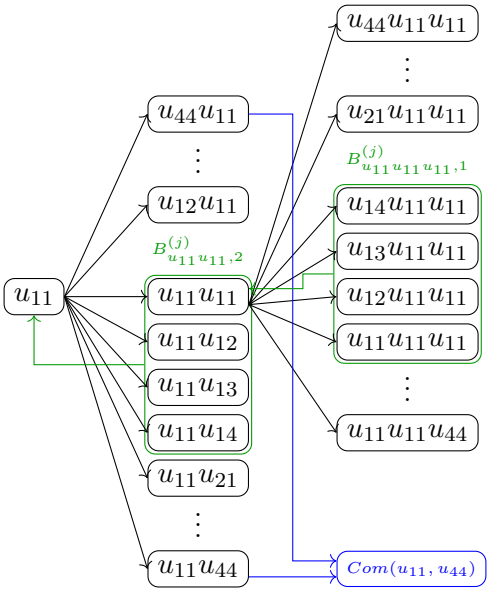


Figure 9: Part of the $(4, 3)$-monomial graph, now also showing two backward edges and backward vertices in green.

Next, we add the vertex $u_{11}u_{44}u_{11}$ to our figure and mark the commutation edges $(C_{u_{11}u_{44},1}, u_{44}u_{11})$, $(C_{u_{11}u_{11}u_{44},2}, u_{11}u_{44}u_{11})$ and $(C_{u_{11}u_{44}u_{11},2}, u_{11}u_{11}u_{44})$ in Figure 10.

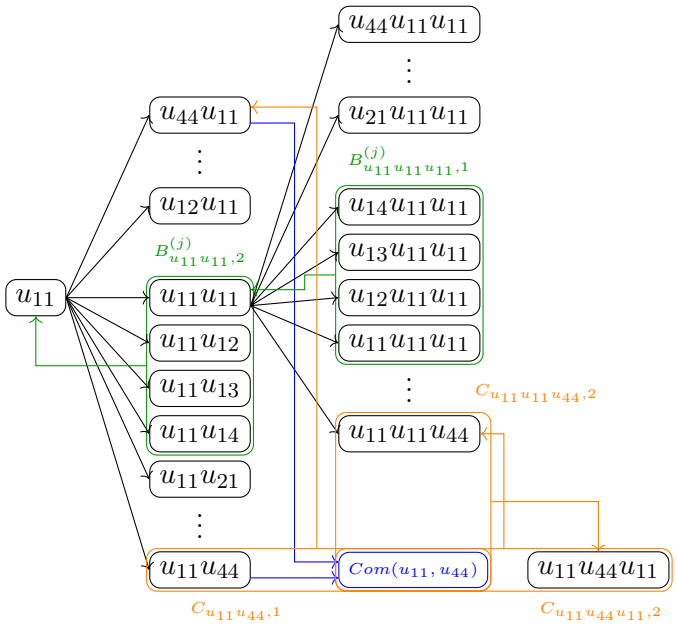


Figure 10: Part of the $(4, 3)$-monomial graph, now also showing three commutation edges in orange.

Finally, we also mark the sum vertices $S^{(j)}_{u_{11}u_{14},2}$ and $S^{(j)}_{u_{11}u_{11}u_{11}}$ as well as the sum edges $(S^{(j)}_{u_{11}u_{14},2}, u_{11}u_{14})$ and $(S^{(j)}_{u_{11}u_{11}u_{11}}, u_{11}u_{11}u_{11})$ in Figure 11. Note that in any

of these figures we have only marked representatives of the different types of edges rather than all of them in order to minimize visual occlusion.

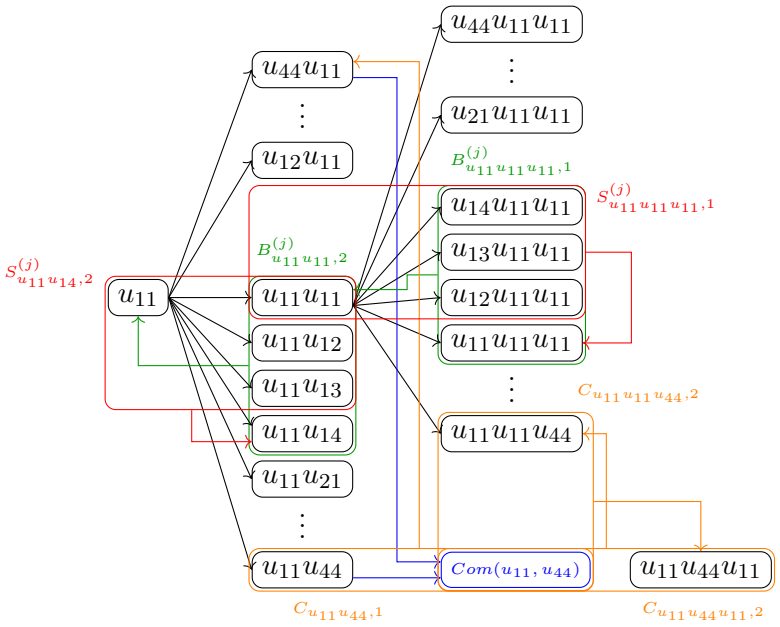


Figure 11: Part of the $(4, 3)$-monomial graph, now also showing two sum edges in red.

The figures from this example only showcase a very small part of the $(4, 3)$-monomial graph which also has the relatively small parameters $n = 4$ and $k = 3$. This might give an intuition for the size of the (n, k)-monomial graph. While the figures might seem hard to follow, they should give a feeling for the types of (hyper-)edges given in the monomial graph. They illustrate how the vertices can be thought of as being "in layers" where the length of the monomials determine which layer they are a part of. Then, one can quickly see that forward edges go from a vertex in one layer to one in the next layer, commutation edges require an additional commutation vertex and stay within a layer, backward edges go from several vertices in one layer to one in the previous layer and sum edges go from vertices in subsequent layers to one in the higher of the two.

4.9 Remark. The layering of the (n, k)-monomial graph also has another effect. If one discards the vertices associated to monomials of lengths l for $k' < l \leq k$ and any edges they are a part of—that is the layers $k' + 1$ to k—what remains is the (n, k')-monomial graph. Thus, the (n, k)-monomial graph contains all (n, k')-monomial graphs as subhypergraphs for $k' \leq k$. On the other hand, this gives a natural way to construct the (n, k)-monomial graph iteratively by subsequently adding layers.

We now need to find a suitable data structure for the (n, k)-monomial graph that later facilitates an algorithm for the traversal of the monomial graph. Here, we will differentiate between the operations such a data structure should offer and a concrete implementation. First however, we define an index that helps enumerate

the monomials and vertices of the (n,k)-monomial graph. This in turn will make it easier to access a specific monomial using this identifier.

4.10 Definition. Let $M_{n,k}$ be the (n,k)-monomial graph and v the vertex associated to the monomial $u = u_{i_1 j_1} \dots u_{i_m j_m}$ of length $m \leq k$. Then the *monomial index* of u is defined as

$$MI(u) = 1 + \sum_{r=1}^{m} (i_r - 1) n^{2(m-r)+1} + (j_r - 1) n^{2(m-r)}$$

and the *vertex index* of v is defined as the pair $VI(\mathsf{v}) = (m, MI(u))$.

4.11 Remark. The monomial index simply enumerates the monomials of a given length by considering the indices i_r and j_r as digits in base n shifted by 1, i.e. the digits run from 1 to n rather than from 0 to $n-1$. Thus, given the monomial $u = u_{i_1 j_1} \dots u_{i_m j_m}$, the shifted indices $(i_1 - 1)(j_1 - 1) \dots (i_m - 1)(j_m - 1)$ form the base n representation of the base 10 number $MI(u) - 1$ and so the mapping MI restricted to monomials of length m is a bijection onto $[n^{2m}]$. Since the monomial index of u is clearly unique among all monomials of length m, the vertex index uniquely identifies the vertex v.

Since the monomial index maps the monomials to natural numbers, this also induces an ordering on the monomials of the same length. In fact, this is exactly the lexicographical order of the monomial's i and j indices. Using the vertex index, one can define an order that first compares the length of the monomial and in case of equality then uses the previous order. This turns out to be the shortlex order.

The monomial index of the vertex v associated to the monomial u of length m may be computed using the function `compute_monomial_index(`v`, `n`)` in time $\mathcal{O}(m)$ which is upperbounded by $\mathcal{O}(k)$. Additionally, the function is overloaded to also accept the arguments (u, n).

To encode a monomial graph, we propose the following data structure interfaces for the monomial graph itself as well as substructures such as vertices.

4.12 Proposition. *Let $n, k \in \mathbb{N}$. The (n,k)-monomial graph interface supports the following operations:*

- `get_vertex(`$VI(u)$`)`*: Return the vertex associated to monomial u via its vertex index $VI(u)$.*
- `get_vertexlist(`m`)`*: Return a list of all vertices of length m.*
- `get_commutation(`u_{ij}, u_{kl}`)`*: Return* True *if the generators u_{ij} and u_{kl} are known to commute.*
- `set_commutation(`u_{ij}, u_{kl}`)`*: Set the commutation status of the generators u_{ij} and u_{kl} to commuting.*

4.13 Proposition. *The vertex interface for the vertex associated to the monomial u supports the following operations:*

- `get_visited()`*: Return* True *if the vertex has been visited while traversing the monomial graph. This means that the associated monomial is equal to zero.*

- `set_visited(`val`)`: *Set the visited status to* val.
- `monomial()`: *Return the monomial u the vertex is associated to.*
- `length()`: *Return the length m of the associated monomial u.*
- `forwardedges()`: *Return a list of all forward edges $(u, v) \in E_F$.*
- `backwardedges()`: *Return a list of all backward edges $(B_{u,r}^{(c)}, \hat{u}) \in E_B$ where $c \in \{$`'i'`, `'j'`$\}$ is a character and $r \in [$`length(u)`$], B_{u,r}^{(i)}, B_{u,r}^{(j)}$, and $\hat{u}$ are as described in Definition 4.4, i.e. a list of backward edges that have u in their source set.*
- `commutationedges()`: *Return a list of all commutation edges $(C_{u,r}, u_r^{(C)}) \in E_C$ where $r \in [$`length(u)`$-1], C_{u,r}$ and $u_r^{(C)}$ are as described in Definition 4.4, i.e. a list of commutation edges that have u in their source set.*
- `sumedges()`: *Return a list of all sum edges $(S_{u',r}^{(c)}, u')$ where $c \in \{$`'i'`, `'j'`$\}$ is a character, $r \in [$`length(u)` $+ 1]$ is a position, u' is a monomial such that removing the r-th generator from u' yields the monomial u, and $S_{u',r}^{(c)}$ is as described in Definition 4.4, i.e. a list of sum edges where u is the short monomial.*

4.14 Remark. The operation u.`sumedges()` only returns the sum edges $(S_{u',r}^{(c)}, u')$ where $u \in S_{u',r}^{(c)}$ is the short monomial. This is because in all other cases where u is one of the long monomials, we can construct the sum edge from a backward edge obtained by the operation u.`backwardedges()`. If we want to get the sum edge $(S_{u'',r}^{(c)}, u'')$ where we know that u is a long monomial of $S_{u'',r}^{(c)}$ and $u'' \neq u$ is any monomial in $B_{u,r}^{(i)}$ we can use the formula from Remark 4.5

$$S_{u'',r}^{(c)} = (B_{u'',r}^{(c)} \setminus \{u''\}) \cup \{\hat{u}\} = (B_{u,r}^{(c)} \setminus \{u''\}) \cup \{\hat{u}\}$$

where $\hat{u}$ is the short monomial. Thus, the backward edge $(B_{u,r}^{(c)}, \hat{u})$ obtained from u.`backwardedges()` gives the option of constructing $(n-1)$-many sum edges with u as the long monomial—one for each element of $B_{u,r}^{(c)}$ other than u. In Section 4.4 we show how this is done in more detail.

4.15 Remark. In Proposition 4.13, we have kept it deliberately ambiguous, how the monomial, forward, backward, commutation and sum edges returned by the respective functions of the vertex interface are encoded. This will depend on the chosen data structure used to implement the interface. The interfaces' operations have been designed with the goal of traversing the monomial graph in mind. Thus, both these and the following implementations will ultimately make sense in the context of the traversal algorithm defined in Section 4.4.

4.16 Definition. To implement the (n, k)-monomial graph interface we propose a composite data structure combining a boolean matrix for the commutation vertices and a list of lists storing the vertices. Here, the m-th list contains all vertices associated to monomials of length $m \leq k$ and is sorted by the monomial index. The boolean

matrix commutationtable is of size $n^2 \times n^2$ and encodes the commutation status of the generators u_{ij} and u_{kl} at the position $(MI(u_{ij}), MI(u_{kl}))$. Additionally, both n and k are stored by the monomial graph data structure for future computations.

Data structure 1: Monomial graph

```
1 structure Monomialgraph
2     n::Integer
3     k::Integer
4     vertexlist::Array
5     commutationtable::2D-Array
```

This data structure implements the (n, k)-monomial graph interface as follows.

- `get_vertex` $(VI(u))$: Let $VI(u) = (m, i)$. Return vertexlist$[m][i]$.
- `get_vertexlist` (m): Return vertexlist$[m]$.
- `get_commutation` (u_{ij}, u_{kl}): Return commutationtable$[MI(u_{ij}),MI(u_{kl})]$.
- `set_commutation` (u_{ij}, u_{kl}): Set commutationtable$[MI(u_{ij}),MI(u_{kl})]$ and commutationtable$[MI(u_{kl}),MI(u_{ij})]$ to True.
- A new monomial graph is instantiated using the function `new_monomial_graph` (vertexlist, commutationtable).

4.17 Remark. The constructor `new_monomial_graph` does not need n and k explicitly as arguments. Instead, it extracts them as $n =$ `length` (vertexlist[1]) $^{\frac{1}{2}}$ and $k =$ `length` (vertexlist). This is due to vertexlist being a list of length k with its m-th entry being a list of vertices associated to monomials of length m. Therefore, vertexlist[1] is a list of all monomials of length 1 which is a list of the n^2 generators of $C(G^+_{aut}(\Gamma))$.

4.18 Proposition. *With the data structure from Definition 4.16 the functions* `get_vertexlist`, `get_vertex`, `get_commutation`, `set_commutation` *and the constructor* `new_monomial_graph` *have constant run time.*

Proof. Both `get_vertexlist` and `get_vertex` are getters that simply access arrays. This happens in constant time. The function `get_commutation` and the method `set_commutation` with arguments u_{ij} and u_{kl} both require the monomial indices of the generators u_{ij} and u_{kl}. Since `compute_monomial_index` runs in time proportional to the length of the monomial passed as an argument, these calls also happen in constant time and so `get_commutation` and `set_commutation` do as well. The constructor `new_monomial_graph` does not copy the arrays vertexlist and commutationtable but rather stores them by reference. It runs in the time that the computation of $\sqrt{n^2}$ takes which is $\mathcal{O}(1)$ as mentioned in Remark 1.22. □

4.19 Remark. The edges of the monomial graph seem to be missing in this data structure. They are instead encoded within the vertices.

4.20 Definition. To implement the vertex interface, we propose a composite data structure combining a monomial $u = u_{i_1 j_1} \dots u_{i_m j_m}$ with a list encoding the forward edges $(u, v) \in E_F$, a dictionary encoding the backward edges $(B^{(i)}_{u,r}, \hat{u}), (B^{(j)}_{u,r}, \hat{u}) \in$

E_B, a dictionary encoding the sum edges $(S^{(i)}_{u',r}, u'), (S^{(j)}_{u',r}, u')$ that have u in its sources as the short monomial, a list encoding the commutation edges $(C_{u,r}, u_r^{(C)})$ and a field denoting whether the monomial has been visited during the traversal algorithm.

The monomial $u_{i_1 j_1} \dots u_{i_m j_m}$ is encoded as a list of pairs of integers as

$$[(i_1, j_1), \dots, (i_m, j_m)].$$

The list encoding the forward edges is simply a list of the ranges $v \in V$ such that $(u, v) \in E_F$.

The dictionary of backward edges maps a pair (r, c) to a pair $(VI(u_{next}), VI(\hat{u}))$. Here, $r \in [m]$ denotes a position in u and $c \in \{\texttt{'i'}, \texttt{'j'}\}$ distinguishes between the i and the j index. Again, $\hat{u}$ is defined as in 4.4 and $u_{next} = u_{i_1 j_1} \dots u_{i'_r j'_r} \dots u_{i_m j_m}$ is understood via

$$i'_r = \begin{cases} (i_r \bmod n) + 1 & \text{if } c = \texttt{'i'}, \\ i_r & \text{otherwise} \end{cases}$$

and

$$j'_r = \begin{cases} j_r & \text{if } c = \texttt{'i'}, \\ (j_r \bmod n) + 1 & \text{otherwise.} \end{cases}$$

Thus, u_{next} increases one the index specified by c at position r by one with the modular arithmetic being used to "wrap around" from the index n back to 1.

Similarly, the dictionary of sum edges maps a tuple (r, c, s) to a vertex index $VI(u')$. Here, the monomial u' is of length `length(u)` $+1$, $r \in$ `[length(u')]` is a position, $c \in \{\texttt{'i'}, \texttt{'j'}\}$ is a character and $s \in [n]$ is used as an index of a generator. Further, u' is of the form that

(a) its r-th generator is of the form $u_{i's}$ if $c = \texttt{'i'}$ and $u_{sj'}$ if $c = \texttt{'j'}$ with $i' \in [n]$ and $j' \in [n]$ arbitrary and

(b) u is obtained from u' by removing the r-th generator.

This simply means, that u' is an element of $B^{(c)}_{u',r}$.

Finally, the list of commutation edges is a list of length `length(u)` -1 where the r-th element of the list is $VI(u_r^{(C)})$.

Data structure 2: Vertex

```
structure Vertex
    monomial::Array
    forwardedges::Array
    backwardedges::Dictionary
    sumedges::Dictionary
    commutationedges::Array
    visited::Boolean
```

This data structure implements the vertex interface as follows.

- `get_visited()`: Return visited.

- `set_visited(val)`: Set visited to val.
- `monomial()`: Return `monomial`.
- `length()`: Return `length(monomial)`.
- `forwardedges()`: Return `forwardedges`.
- `backwardedges()`: Return `backwardedges`.
- `commutationedges()`: Return `commutationedges`.
- `sumedges()`: Return `sumedges`.
- A new vertex is instantiated using the function `new_vertex(monomial)`.

4.21 Remark. A newly instantiated vertex' `forwardedges`, `backwardedges`, `commutationedges` and `sumedges` are empty and have to be added in a further step.

4.22 Proposition. *With the data structure from Definition 4.20 the functions from the vertex interface and the constructor* `new_vertex` *run in constant time.*

Proof. The functions `get_visited`, `set_visited`, `monomial`, `forwardedges`, `backwardedges`, `commutationedges` and `sumedges` simply return a stored value which happens in constant time. The function `length` queries the length of `monomial` which takes constant time as mentioned in Section 1.4. The constructor `new_vertex` also runs in constant time because the array `monomial` is not copied but rather saved by reference. □

4.23 Remark. The dictionary `backwardedges` may seem unintuitive but will make sense in the context of the traversal step. For now, we remark that a backward vertex $B_{u,r}^{(c)}$ with r and c as in Definition 4.20 is determined entirely by u, r and c. Further, given a fixed pair (r, c) and a first vertex $\mathsf{v} \in B_{u,r}^{(c)}$, the backward vertex $B_{u,r}^{(c)}$ may be obtained by iteratively taking the "next" vertex $\mathsf{v}_{next} =$ `M.get_vertex(v.backwardedges()[(r,c)][1])` n-times.

Similarly, the dictionary `sumedges` does not directly return a sum vertex $S_{u',r}^{(c)}$, but rather also a representative of $B_{u',r}^{(c)}$, which may then be used to obtain the entire set as mentioned in Remark 4.14.

The concrete logic giving access to the entire backward and sum edges is thus not directly stored in the data structure but rather happens during the traversal. This is shown explicitly in Algorithm 9.

4.2 Monomial Graph Creation

In this section we show the first step of our algorithm which is creating the (n, k)-monomial graph. Because the (n, k)-monomial graph is contained in the $(n, k+1)$-monomial graph as a subhypergraph, we opt to use an iterative approach for the construction of the monomial graphs. First the $(n, 1)$-monomial graph is created as follows:

Algorithm 3: Create $(n, 1)$-monomial graph

Input: n

```
1 commutationtable ← I_{n²×n²};
2 vertexlist ← new_array(1);
3 vertexlist[1] ← new_array(n²);
4 for i ← 1 to n do
5 |   for j ← 1 to n do
6 |   |   v ← new_vertex([(i, j)]) ;
7 |   |   vertexlist[1][compute_monomial_index(v, n)] ← v;
8 return new_monomial_graph(vertexlist, commutationtable)
```

4.24 Proposition. *Algorithm 3 creates the* $(n, 1)$*-monomial graph* M *in time* $\mathcal{O}(n^4)$.

Proof. The $(n, 1)$-monomial graph $\mathsf{M} = (V, Com, E_F, E_B, E_C, E_S, F_C)$ contains n^2 vertices in V. Each vertex $\mathsf{v} \in V$ corresponds to a different monomial of length 1 which is simply one of the n^2 generators $u_{i,j}$. The algorithm creates all of these vertices in the nested for-loops. Since all edges and hyperedges require monomials of length $m \geq 2$, the sets E_F, E_B, E_C, E_S and F_C are all empty. Therefore, the newly created vertices are already complete. By default, we only know that each generator commutes with itself. Therefore, the commutation vertices $Com(u_{i,j}, u_{k,l}) \in Com$ are represented as a $(n^2 \times n^2)$-identity matrix.

The call to `compute_monomial_index` takes time $\mathcal{O}(1)$ because we know that the length of the monomials passed as an argument is 1. Therefore, both the creation of the array in line 3 as well as the nested for-loop take time $\mathcal{O}(n^2)$. However, the run time is dominated by the creation of the $(n^2 \times n^2)$-identity matrix in line 1 which takes time $\mathcal{O}(n^4)$. □

Before we can give the algorithm that creates the $(n, k+1)$-monomial graph given the (n, k)-monomial graph, we require three additional helpful functions that compute specific monomial indices used for the backward, sum and commutation edges.

Algorithm 4: `compute_target_index`: Compute the monomial index of $\hat{u}$

Input: monomial $u = u_{i_1 j_1} \dots u_{i_m j_m}$ of length m, position $r \in [m]$, n

```
1 target ← new_array(m − 1);
2 target[1 : r − 1] ← u[1 : r − 1];
3 target[r : m − 1] ← u[r + 1 : m];
4 return compute_monomial_index(target, n);
```

4.25 Corollary. *Let* u *be a monomial of length* $m \leq k$ *and* $r \in [m]$ *a position in* u. *Further, let* $\hat{u}$ *be as in Definition 4.4. Algorithm 4 computes the monomial index of* $\hat{u}$ *in time* $\mathcal{O}(m)$ *which is in* $\mathcal{O}(k)$.

Algorithm 5: `compute_next_index`: Compute the monomial index of u_{next}

Input: monomial $u = u_{i_1 j_1} \dots u_{i_m j_m}$ of length m, position $r \in [m]$, character $c \in \{\texttt{'i'}, \texttt{'j'}\}$, n

```
1 if c = 'i' then
2 |  numC ← 1;
3 else
4 |  numC ← 2;
5 currentIndex ← compute_monomial_index(u, n);
6 if u[r][numC] ≠ n then
7 |  return currentIndex + n^(2(m−r)+(numC mod 2));
8 else
9 |  return currentIndex − (n − 1) * n^(2(m−r)+(numC mod 2));
```

4.26 Proposition. *Let $u = u_{i_1 j_1} \dots u_{i_m j_m}$ be a monomial of length $m \leq k$. Further, let $r \in [m]$ be a position in u, $c \in \{\texttt{'i'}, \texttt{'j'}\}$ a character and u_{next} be defined as in Definition 4.20. Then Algorithm 5 computes the monomial index of u_{next} in time $\mathcal{O}(m)$ which is in $\mathcal{O}(k)$.*

Proof. Recall from Definition 4.10 that the formula for the monomial index of u is given by

$$MI(u) = 1 + \sum_{l=1}^{m} (i_l - 1) n^{2(m-l)+1} + (j_l - 1) n^{2(m-l)}$$

and that $u_{next} = u_{i_1 j_1} \dots u_{i'_r j'_r} \dots u_{i_m j_m}$ is identical to u except for the r-th component. For the r-th component, the index corresponding to c is increased by 1, wrapping around back to 1 if it is already equal to n. Thus, if c is equal to `'i'` and $i_r \neq n$ we get that $u_{next} = u_{i_1 j_1} \dots u_{i_r+1 j_r} \dots u_{i_m j_m}$ and thus

$$MI(u_{next}) = MI(u) + n^{2(m-r)+1}.$$

If c is again equal to `'i'` but $i_r = n$, we get $u_{next} = u_{i_1 j_1} \dots u_{1 j_r} \dots u_{i_m j_m}$ and then

$$MI(u_{next}) = MI(u) - (n-1) n^{2(m-r)+1}.$$

Similarly, if c is equal to `'j'`, we again distinguish whether j_r is equal to n or not and get

$$MI(u_{next}) = \begin{cases} MI(u) + n^{2(m-r)} & \text{if } j_r \neq n, \\ MI(u) - (n-1) n^{2(m-r)} & \text{otherwise.} \end{cases}$$

Comparing the exponents of n, we see that assigning numC as we do in lines 1–4 yields the correct result.

The run time comes entirely from the call to `compute_monomial_index` and is therefore $\mathcal{O}(m)$ which is dominated by $\mathcal{O}(k)$. □

Algorithm 6: `compute_switch_index`: Compute the monomial index of $u_r^{(C)}$

Input: monomial u of length m, position $r \in [m-1]$ in u, n

```
1 mon ← copy(u);
2 switch(mon, r, r + 1);
3 return compute_monomial_index(mon, n);
```

4.27 Remark. The method `switch(arr, i, j)` switches the entries i and j of the array *arr* in constant time.

4.28 Corollary. *Let u be a monomial of length $m \leq k$ and $r \in [m-1]$ a position in u. Further, let $u_r^{(C)}$ be as in Definition 4.4. Algorithm 6 computes the monomial index of $u_r^{(C)}$ in time $\mathcal{O}(m)$ which is in $\mathcal{O}(k)$.*

With these functions and given the (n,k)-monomial graph, the $(n,k{+}1)$-monomial graph is created as follows:

Algorithm 7: Create $(n, k+1)$-monomial graph

Input: (n,k)-monomial graph M

```
1  vertexlist' ← copy(M.vertexlist);
2  leaflist ← new_array(n^{2(k+1)});
3  foreach v in vertexlist'[k] do
4      for i, j ← 1 to n do
5          m' ← push(copy(v.monomial), (i, j));
6          index ← compute_monomial_index(m', n);
7          if leaflist[index] is undefined then
8              v' ← new_vertex(m');
9              leaflist[index] ← v';
10         else
11             v' ← leaflist[index];
12         push(v.forwardedges, v');
13     for i, j ← 1 to n do
14         m' ← pushfirst(copy(v.monomial), (i, j));
15         index ← compute_monomial_index(m', n);
16         if leaflist[index] is undefined then
17             v' ← new_vertex(m');
18             leaflist[index] ← v';
19         else
20             v' ← leaflist[index];
21         push(v.forwardedges, v');
22 push(vertexlist', leaflist);
```

Algorithm 7: Create $(n, k+1)$-monomial graph, continued

```
23 foreach v in leaflist do
24     for l ← 1 to k + 1 do
25         v̂ ← compute_target_index(v.monomial, l, n);
26         v_i' ← compute_next_index(v.monomial, 'i', l, n);
27         v_j' ← compute_next_index(v.monomial, 'j', l, n);
28         v.backwardedges[(l, 'i')] ← ((k + 1, v_i'), (k, v̂));
29         v.backwardedges[(l, 'j')] ← ((k + 1, v_j'), (k, v̂));
30         vertexlist'[k][v̂].sumedges[(l, 'i', v.monomial[l][2])] ← (k + 1, v_i');
31         vertexlist'[k][v̂].sumedges[(l, 'j', v.monomial[l][1])] ← (k + 1, v_j');
32     v.commutationedges ← new_array(k);
33     for r ← 1 to k do
34         v^(C) ← compute_switch_index(v.monomial, r, n);
35         v.commutationedges[r] ← (k + 1, v^(C));
36 return new_monomial_graph(vertexlist', copy(M.commutationtable));
```

4.29 Proposition. *Given the (n, k)-monomial graph* M*, Algorithm 7 creates the $(n, k+1)$-monomial graph* M' *in expected time $\mathcal{O}(n^{2(k+1)}(n^4 + n^2k + k^2))$.*

Proof. The $(n, k+1)$-monomial graph differs from the (n, k)-monomial graph only in the vertices corresponding to monomials of length k and of length $k+1$. The former now also have new forward edges and sum edges connecting them to vertices of the latter kind which are completely new and have backward and commutation edges. Thus, the algorithm has to create all new vertices corresponding to monomials of length $k+1$ and add all types of edges where appropriate.

First, we note that we call `copy` on M.vertexlist in line 1 in order to not alter the input monomial graph M. We do the same with M.commutationtable in the call to the constructor of the new monomial graph in line 36.

There are $n^{2(k+1)}$ monomials of length $k+1$. The vertices corresponding to them will be stored in the array leaflist allocated in line 2. Crucially, as mentioned in Section 1.4, this array by default has entries of *undefined*. Next, the loop in line 3 iterates over all vertices corresponding to monomials of length k. Consider such a vertex v corresponding to the monomial $u = u_{i_1 j_1} \dots u_{i_k j_k}$. The nested loops in line 4 then create all monomials uu' where u' is any generator and the nested loops in line 13 create all monomials $u'u$. Because vertexlist$[k]$ contains all monomials of length k, every monomial of length $k+1$ gets created twice in this fashion—once via the prefix of length k and once via the suffix of length k. However, we only want to create the corresponding vertex once. This is done by checking the entry of the array leaflist at the corresponding position in lines 7 and 16. If this entry is still *undefined*, the vertex has not been created before. It is then created, added to leaflist and the corresponding forward edge is added to the vertex v. Otherwise, the previously created vertex is directly used to create the forward edge. In this way, all $n^{2(k+1)}$ new vertices are created and all new forward edges are created.

What remains is adding the backward and commutation edges to these new vertices as well as adding the sum edges to the vertices of length k. Note that a vertex corresponding to a monomial u of length $k+1$ is part of $(2(k+1))$-many

backward vertices $B_{u,r}^{(c)}$, where $c \in \{\texttt{'i'}, \texttt{'j'}\}$ is a character and $r \in [k+1]$ is a position in u. As mentioned in Definition 4.20, the backward edges are stored as a dictionary mapping the pair (r, c) to the pair $(VI(u_{next}), VI(\hat{u}))$, where u_{next} and $\hat{u}$ are again defined as in Definition 4.20. The functions `compute_target_index` and `compute_next_index` exactly compute $MI(\hat{u})$ and $MI(u_{next})$ respectively and the vertex indices are then given by $VI(u_{next}) = (k+1, MI(u_{next}))$ and $VI(\hat{u}) = (k, \hat{u})$. By iterating over all new vertices `v` and all positions $r \in [\texttt{v.length()}] = [k+1]$ in lines 23 and 24, we can compute all required indices and assign them to the new vertices.

The sum edges of the vertices of length k are also added in this step, due to the sum edges being very similar to the backward edges. Given the same r and c as before, the target of each backward edge computed in line 25 is a vertex associated to a monomial of length k. Therefore, it is the short monomial in the sum vertices $S_{u,r}^{(c)}$ and should have a sum edge assigned to the tuple (r, c, s) where s is either i_r if $c = \texttt{'j'}$ or j_r if $c = \texttt{'i'}$. The value assigned is a vertex index belonging to a vertex in $B_{u,r}^{(c)}$, so we can reuse the value $VI(u_{next})$. In this way, each tuple (r, c, s) gets assigned n times—once with each of the elements of $B_{u,r}^{(c)}$. This is superfluous as just assigning it once would suffice but also does no harm as any of the n assignments is correct to our specification.

Finally, all that remains is adding the commutation edges to the new vertices. In order to do so, for every new vertex u and position $r \in [u.\texttt{length()} - 1] = [k]$, the index of $u_r^{(C)}$ gets computed in line 34 and gets assigned to the correct position.

The run time breaks down as follows. The `copy` command in line 1 takes time proportional to the number of vertices in the (n, k)-monomial graph. There are n^{2l} different monomials of length l and therefore this `copy` command takes time $\mathcal{O}(\sum_{l=1}^{k} n^{2l}) = \mathcal{O}(n^{2k})$. The `copy` command in line 36 takes time $\mathcal{O}(n^4)$, the creation of leaflist in line 2 takes time $\mathcal{O}(n^{2(k+1)})$ and the call of `push` in line 22 also takes time $\mathcal{O}(k)$.

However, the most time is taken by the creation of the new vertices, the assignment of the new forward edges and of the new backward edges. The loop in line 3 iterates over n^{2k} elements. The nested loop in line 4 then iterates over n^2 elements. Both the `push` and the `copy` calls in line 5 as well as the computation of the monomial index in line 6 take time $\mathcal{O}(k)$ and the `push` call in line 12 takes time $\mathcal{O}(n^2)$. The remaining part of the loops body runs in constant time. This yields a run time of $\mathcal{O}(n^2(k + n^2)) = \mathcal{O}(n^2k + n^4)$ for the nested loop in line 4. Analogously, this also holds for the next nested loop in line 13. Overall, this yields a run time of $\mathcal{O}(n^{2(k+1)}(n^2k + n^4))$ for the loop in line 3.

Next, the loop in line 23 iterates over $n^{2(k+1)}$-many elements and the loop in line 24 iterates over $(k+1)$-many elements. As mentioned in Corollary 4.25 and Proposition 4.26, the calls to `compute_target_index` and `compute_next_index` in the loop body will run in time $\mathcal{O}(k)$ and as mentioned in Section 1.4, the dictionary assignments happen in expected constant time. Thus, the loop in line 24 runs in expected time $\mathcal{O}(k^2)$. Finally, creating the array of length k in line 32 takes time $\mathcal{O}(k)$, as does the computation of the switch index in line 34. Thus, the loop in line 33 takes time $\mathcal{O}(k^2)$ and the loop in line 23 runs in expected time $\mathcal{O}(n^{2(k+1)}k^2)$. Overall, the two big loops dominate the run time and we get an expected run time of $\mathcal{O}(n^{2(k+1)}(n^4 + n^2k + k^2))$. □

4.30 Remark. In our use cases, we typically expect n to be larger or equal than k. In this case, the run time of Algorithm 7 simplifies to $\mathcal{O}(n^{2k+6})$.

4.3 Start Zeros

So far, other than the size n of the input graph Γ, the graph structure has not been used further. In this section, the graph structure is used together with the relations from Lemma 2.2 and the derived results from Lemmas 2.7 and 2.8 in order to find monomials that are known to be 0 in $C(G^+_{aut}(\Gamma))$. We denote these monomials as "start zeros" since they will be the start points for the propagation in the next step. As mentioned before, Lemma 2.8 is stronger than the relations from Lemma 2.2 and we therefore do not specifically check them when also using the stronger criterion.

In order to apply Lemma 2.8, we need access to the distance matrix D_Γ. This is a well-known graph problem known as *All Pairs Shortest Path (APSP)* introduced by Shimbel in 1953 [26] and as such there are many published results on how to solve it efficiently. Since the input graphs Γ we consider tend to be small in context of the APSP problem, we simply compute it as the n-fold Min-Plus power of a modified adjacency matrix.

4.31 Definition. The *Min-Plus product* $C = A \star B$ of two $n \times n$-matrices $A = (A_{ij})$ and $B = (B_{ij})$ (also allowing entries of ∞) is the matrix defined via

$$C_{ij} = \min_{k \in [n]} (A_{ik} + B_{kj}).$$

For $k \in \mathbb{N}$, we define $\star^k A$ to mean the k-fold Min-Plus product of A with itself.

This corresponds to a "normal" matrix multiplication where one applies a sum instead of a multiplication and the min operator instead of a sum. It is, in fact, the matrix multiplication over the min tropical semiring. Naively, this yields a run time of $\mathcal{O}(n^3)$ for the computation of a Min-Plus product.

4.32 Definition. Let Γ be a graph with n vertices. We call the $n \times n$-matrix $D^{(k)}_\Gamma$ defined by

$$(D^{(k)}_\Gamma)_{u,v} = \begin{cases} d(u,v) & \text{if } d(u,v) \leq k, \\ \infty & \text{otherwise} \end{cases}$$

the *k-distance matrix of* Γ. It encodes the length of the shortest path of length at most k connecting any pair of vertices while the entry ∞ denotes that there is no path of length at most k connecting the pair of vertices.

4.33 Remark. The 1-distance matrix $D^{(1)}_\Gamma$ is closely related to the adjacency matrix A_Γ of the graph Γ. The shortest paths of length at most 1 between two vertices u and v is trivially either 0 if $u = v$ or 1 if $u \sim v$. Thus, one obtains $D^{(1)}_\Gamma$ from A_Γ by changing all off-diagonal 0 entries to ∞. Crucially, for this we assumed all graphs to have no loops and therefore the diagonal of A_Γ already consists of all zeros. On the other hand, the distance matrix D_Γ is exactly the n-distance matrix $D^{(n)}_\Gamma$.

4.34 Proposition. *Let Γ be a graph with n vertices and k be natural number. We get $D^{(k+1)}_\Gamma = D^{(k)}_\Gamma \star D^{(1)}_\Gamma$.*

Proof. Let u and v be vertices in Γ. Note that $\min_{w\in V}(d(u,w)+d(w,v)) \geq d(u,v)$ holds in general. Otherwise, let $\hat{w}$ be a vertex minimizing the sum. Then a shortest path from u to $\hat{w}$ concatenated with a shortest path from $\hat{w}$ to v would be a path connecting u and v of length shorter than $d(u,v)$ which contradicts the minimality of the distance function. Note also that for any $k' \in \mathbb{N}$ and any vertices $u', v' \in V$, that $(D_\Gamma^{(k')})_{u',v'} \geq d(u',v')$ and thus together this yields

$$\min_{w\in V}((D_\Gamma^{(k)})_{u,w} + (D_\Gamma^{(1)})_{w,v}) \geq d(u,v).$$

Next, we distinguish three cases. First, let $d(u,v) < k+1$. Then $(D_\Gamma^{(k)})_{u,v} = (D_\Gamma^{(k+1)})_{u,v} = d(u,v)$ and $(D_\Gamma^{(1)})_{v,v} = 0$. We get

$$\begin{aligned} d(u,v) &\leq \min_{w\in V}((D_\Gamma^{(k)})_{u,w} + (D_\Gamma^{(1)})_{w,v}) \\ &\leq (D_\Gamma^{(k)})_{u,v} + (D_\Gamma^{(1)})_{v,v} \\ &= d(u,v) + 0 \\ &= d(u,v) \end{aligned}$$

and thus $\min_{w\in V}((D_\Gamma^{(k)})_{u,w} + (D_\Gamma^{(1)})_{w,v}) = (D^{(k+1)})_{u,v}$.

Next, let $d(u,v) = k+1$. This means that there is a minimal path p of length $k+1$ connecting u and v. Denote by w' the second to last vertex of p. Then we get $d(u,w') = k$ and $d(w',v) = 1$ and therefore

$$\begin{aligned} d(u,v) &\leq \min_{w\in V}((D_\Gamma^{(k)})_{u,w} + (D_\Gamma^{(1)})_{w,v}) \\ &\leq (D_\Gamma^{(k)})_{u,w'} + (D_\Gamma^{(1)})_{w',v} \\ &= d(u,w') + d(w',v) \\ &= k+1 \\ &= d(u,v). \end{aligned}$$

This again means that $\min_{w\in V}((D_\Gamma^{(k)})_{u,w} + (D_\Gamma^{(1)})_{w,v}) = (D_\Gamma^{(k+1)})_{u,v}$.

Finally, let $d(u,v) > k+1$. Then, assume that $\min_{w\in V}((D_\Gamma^{(k)})_{u,w} + (D_\Gamma^{(1)})_{w,v}) = l < \infty$ and let w' be a vertex minimizing the sum. Since the sum is finite, so are the summands and by the definition of the k-distance matrix, we also get $(D_\Gamma^{(k)})_{u,w'} \leq k$ and $(D_\Gamma^{(1)})_{w',v} \leq 1$. Taken together, this would yield

$$\begin{aligned} d(u,v) &\leq \min_{w\in V}((D_\Gamma^{(k)})_{u,w} + (D_\Gamma^{(1)})_{w,v}) \\ &\leq (D_\Gamma^{(k)})_{u,w'} + (D_\Gamma^{(1)})_{w',v} \\ &\leq k+1 \\ &< d(u,v) \end{aligned}$$

which is a contradiction. Thus, $\min_{w\in V}((D_{u,w}^{(k)_\Gamma} + (D_\Gamma^{(1)})_{w,v}) = \infty = (D_\Gamma^{(k+1)})_{u,v}$.

In all three cases we get

$$(D_\Gamma^{(k+1)})_{u,v} = \min_{w\in V}((D_\Gamma^{(k)})_{u,w} + (D_\Gamma^{(1)})_{w,v})$$

and thus

$$D_\Gamma^{(k+1)} = D_\Gamma^{(k)} \star D_\Gamma^{(1)}.$$

□

4.35 Corollary. *The* 1*-distance matrix* $D_\Gamma^{(1)}$ *is obtained via*

$$(D_\Gamma^{(1)})_{u,v} = \begin{cases} 0 & \text{if } u = v, \\ 1 & \text{if } u \neq v, u \sim v, \\ \infty & \text{if } u \neq v, u \not\sim v. \end{cases}$$

Then, the distance matrix D_Γ *may be computed as* $D_\Gamma^{(n)} = \star^n D_\Gamma^{(1)}$ *—the* n*-fold Min-plus product of* $D_\Gamma^{(1)}$ *with itself—in time* $\mathcal{O}(n^4)$*.*

Alternatively, should one wish to improve the run time of this step, one may consider a more efficient algorithm for the APSP problem. An overview is given in [8]. However, since checking the condition from Lemma 2.8 requires n^4 comparisons as is shown in Algorithm 8, no significant speed up from this is expected.

For the condition given by Lemma 2.7, one needs the degrees of all vertices.

4.36 Remark. The degree of the vertex u is computed in time $\mathcal{O}(n)$ via

$$\deg(u) = \sum_{v \in V} (A_\Gamma)_{u,v}.$$

The degrees of all vertices are gathered in the vector *deg* with $deg_u = \deg(u)$ in time $\mathcal{O}(n^2)$.

Finally, given the distance matrix D_Γ and the degree vector *deg*, we may compute the start zeros using Lemmas 2.7 and 2.8.

Algorithm 8: Marking start zeros

Input: distance matrix D_Γ, degree vector *deg*, (n,k)-monomial graph M

```
for i, j, k, l ← 1 to n do
    if (D_Γ)_{i,k} ≠ (D_Γ)_{j,l} then
        index ← compute_monomial_index([(i, j), (k, l)], n);
        Mark the vertex M.get_vertexlist(2)[index] as zero;
for i, j ← to n do
    if deg_i ≠ deg_j then
        index ← compute_monomial_index([(i, j)], n);
        Mark the vertex M.get_vertexlist(1)[index] as zero;
```

4.37 Remark. Marking a vertex as zero means setting the field visited to True and adding it to a stack visited_vertices. This stack is used in the traversal step to keep track of all the next vertices used to propagate zeros from. As mentioned in Section 1.4, adding an element to a stack takes time $\mathcal{O}(1)$.

4.38 Proposition. *Algorithm 8 marks all vertices as zero whose associated monomials satisfy the conditions from Lemmas 2.7 and 2.8. The algorithm runs in time* $\mathcal{O}(n^4)$*.*

Proof. Lemma 2.7 requires comparing the degrees of all vertices in Γ and Lemma 2.8 requires comparing the distances between all pairs of vertices in Γ. Since both the degrees and pairwise distances are given already via the degree vector *deg* and the distance matrix D_Γ, the only thing that remains are the comparisons that happen in the loops in lines 1 and 5 respectively.

Notably, the calls to `compute_monomial_index` in both loops happen in constant time rather than in the usual time of $\mathcal{O}(k)$ because we know that that the length of the monomials are at most two. Therefore, the bodies of the loops take constant time and the run time is equal to the number of repetitions of the loops, which is n^4 and n^2 respectively. □

4.39 Remark. While we use the relations stemming from the pairwise distance of vertices and the degrees of the input graph Γ here to find our start zeros, this may of course be improved upon by finding further relations and adding them here. For example, Lemmas 3.2.2 and 3.2.3 in [12] are easy to implement and provide a lot of information for graphs with little regularity. In particular the second criterion applied to the graph from Example 1.4 directly yields that the graph has trivial quantum automorphism group.

In a similar vein, should one know of a specific monomial that is zero due to some other computations they can also be added here manually. The same holds true for commutation relations. During the creation of the $(n, 1)$-monomial graph we set the commutation matrix that encodes the commutation of the generators as the $(n^2 \times n^2)$-identity matrix. When creating the $(n, k+1)$-monomial graph, we also just transfer the known commutation relations from the (n, k)-monomial graph by copying the commutation matrix. However, if one is aware of some generators commuting this knowledge should be added before the traversal step.

4.4 Traverse

Given the (n, k)-monomial graph M and the stack $\mathsf{visited\_vertices}$, the traversal step is similar to other graph traversal algorithms like depth-first search in that from a visited vertex on the stack all reachable vertices are again recursively added to the stack. However, due to M being a hypergraph, regular vertices are checked several times and thus the algorithm performs significantly worse than depth-first search. More precisely, when the vertex associated to the monomial u is "being visited" its outgoing edges' ranges are added to the stack $\mathsf{visited\_vertices}$ provided that the edges' source has been completely visited. For the forward edges this corresponds to just this vertex, but for the backward, sum and commutation edges we need to check the backward, sum and commutation vertices respectively.

In order to check how many of the vertices of a backward or sum vertex have been visited we make use of the similarity between backward and sum vertices. Indeed, recall from Remark 4.14, that given a monomial u of length m, a position $r \in [m]$ in u and an index $c \in \{\texttt{'i'}, \texttt{'j'}\}$, the set $B^{(c)}_{u,r}$ gives rise to the sets $S^{(c)}_{u',r} = (B^{(c)}_{u,r} \setminus \{u'\}) \cup \{\hat{u}\}$ for all vertices $u' \in B^{(c)}_{u,r}$ and $\hat{u}$ as defined in Definition 4.4. Therefore, we are interested in how many elements of $B^{(c)}_{u,r}$ haven been visited—if that number is equal to n, the backward vertex $B^{(c)}_{u,r}$ has been visited completely and if that number is equal to $n-1$ and the vertex $\hat{u}$ has also been visited, a sum vertex

$S^{(c)}_{u',r}$ has been visited. The vertex $\hat{u}$ is already given in the vertex data structure. However, the non-visited vertex u' must be dynamically determined during the check. Therefore, we propose the following function `check_hypervertex`.

Algorithm 9: check hypervertex

Input: (n,k)-monomial graph M, vertex v associated to a monomial of length m, position $r \in [m]$, index $c \in \{\texttt{'i'}, \texttt{'j'}\}$

```
1  num_visited ← 0;
2  non_visited_vertex ← undefined;
3  current ← v;
4  for i ← 1 to n do
5      if current.get_visited is True then
6          num_visited ← num_visited + 1;
7      else
8          non_visited_vertex ← current;
9      if i − num_visited > 1 then
10         break;
11     current ← M.get_vertex(current.backwardedges()[(r, c)][1]);
12 return (num_visited, non_visited_vertex);
```

4.40 Proposition. *Given a vertex* v *associated to a monomial* u *of length* m*, a position* $r \in [m]$ *in* u *and an index* $c \in \{\texttt{'i'}, \texttt{'j'}\}$*, Algorithm 9 returns the pair* $(n, \textit{undefined})$ *if the vertex* $B^{(c)}_{u,r}$ *has been completely visited. If instead there is exactly one element* $u' \in B^{(c)}_{u,r}$ *that has not been visited, Algorithm 9 returns the pair* $(n-1, \mathsf{v}')$ *where* v' *is the vertex associated to* u'*. In all other cases, Algorithm 9 returns a pair* (k, v'')*, where* $k < n-1$ *and* v'' *is a non-visited vertex.*

The run time of Algorithm 9 is $\mathcal{O}(n-l)$*, where* l *is the number of non-visited vertices in* $B^{(c)}_{u,r}$*. Without any assumption on* l*, we get a run time of* $\mathcal{O}(n)$*.*

Proof. First, note that by construction of the monomial graph, starting with the assignment of v to the variable current and fixing a position $r \in [m]$ in u and an index $c \in \{\texttt{'i'}, \texttt{'j'}\}$, the up to n calls of line 11 iterate over all elements of $B^{(c)}_{u,r}$ by always getting the "next" monomial. Next, note that the variable num_visited is increased in every iteration if and only if the currently looked at vertex has been visited, it thus keeps a running total of the number of visited vertices. Conversely, the number of non-visited vertices checked up to the i-th iteration is given by $i -$ num_visited. The if clause in line 9 therefore stops the iteration as soon as the second non-visited vertex is met and num_visited will be equal to the number of visited vertices seen until the iteration in line 4 stops. Finally, note that non_visited_vertex gets assigned the currently looked at vertex current if and only if it has not been visited. Therefore, at the end of the iteration in line 4 it will be equal to the last non-visited vertex seen.

So, if $B^{(c)}_{u,r}$ has been completely visited, the iteration will not stop early, num_visited will be equal to n and non_visited_vertex will not have been assigned a new value. The algorithm will return the pair $(n, \text{undefined})$ as required. Similarly, if there is exactly one vertex $u' \in B^{(c)}_{u,r}$ that has not been visited, at the end of the iteration num_visited will be equal to $n-1$ and non_visited_vertex will have been set to the

vertex v' associated to u'. The algorithm will return the pair $(n-1, \mathsf{v}')$. Finally, in all other cases, we call $l \geq 2$ be the number of non-visited elements in $B^{(c)}_{u,r}$. The iteration will stop early and after the loop num_visited will be at most $n-l$ and non_visited_vertex will have been assigned a non-visited vertex, indeed.

The run time is equal to the number of iterations which in turn is equal to $n-l+2$. This is made up from the number of visited vertices $n-l$ and up to two non-visited vertices before aborting the loop. □

4.41 Remark. Let v be the vertex associated to a monomial u. Then, we can check whether $B^{(c)}_{u,r}$ for a position $r \in [m]$ in u and an index $c \in \{\texttt{'i'}, \texttt{'j'}\}$ has been entirely visited by calling `check_hypervertex` on v, r and c. This is the case if and only if the first returned value is n. In the upcoming pseudocode of the traversal algorithm we have marked this case with the comment "backward edges".

We can also check whether $S^{(c)}_{u',r}$ for a position $r \in [m]$ in u, an index $c \in \{\texttt{'i'}, \texttt{'j'}\}$ and a monomial $u' \in B^{(c)}_{u,r}$ with $u' \neq u$ has been completely visited. In order to do so we would call `check_hypervertex` on v, r and c, again. If the first return value is $n-1$ and the second returned value is the vertex associated to the monomial u' we have a potential candidate for a sum vertex $S^{(c)}_{u',r}$ being visited. What remains is that the short monomial $\hat{u}$ has also been visited. This can be examined using v.`backwardedges()`$[(r,c)][2]$ which yields the vertex index of the vertex associated to $\hat{u}$. This corresponds to the case where u is one of the long monomials of a sum vertex. In the upcoming pseudocode of the traversal algorithm we have marked this case with the comment "sum edges, long monomial".

In the case where u is the short monomial of a sum vertex we are interested in the set $S^{(c)}_{u'',r}$ for a position $r \in [m]$ in u, an index $c \in \{\texttt{'i'}, \texttt{'j'}\}$ and a monomial $u'' \in B^{(c)}_{u'',r}$ that is obtained by inserting a generator at position r in u. Such a set can be determined by calling `check_hypervertex` on v.`sumedges()`$[(r,c,s)]$, r and c. Here, $s \in [n]$ is an index of the inserted generator with c determining whether it is the first or second index. As before, if the first returned value is $n-1$ then the second returned value is the vertex associated to the monomial u''. In the upcoming pseudocode of the traversal algorithm we have marked this case with the comment "sum edges, short monomial".

4.42 Remark. The forward and commutation edges are more straightforward. When visiting the vertex v associated to the monomial u the forward edges can simply be followed by marking each element from v.`forwardedges()` as zero. In the upcoming pseudocode of the traversal algorithm we have marked this case with the comment "forward edges".

The commutation edges similarly only require checking the commutation of the generators at positions r and $r+1$ in u. If they are found to commute we can follow the commutation edge by marking the vertex whose vertex index is found at v.`commutationedges()`$[r]$ as zero. In the upcoming pseudocode of the traversal algorithm we have marked this case with the comment "commutation edges".

The last thing that remains to be mentioned, is that the commutation propagation edges also need to be followed if applicable. This is exactly the case, when the vertex one visits corresponds to a monomial $u_{ij}u_{kl}$ of length two. The commutation vertices are handled differently to the monomial vertices. They are not added to the

stack visited_vertices but are rather marked in the matrix commutationtable. In the upcoming pseudocode of the traversal algorithm we have marked this case with the comment "commutation propagation edges".

Since a commutation vertex is also part of the source $C_{u,r}$ of a commutation edge we would theoretically also have to check every monomial that contains the monomial $u_{ij}u_{kl}$ or $u_{kl}u_{ij}$ as a substring every time we mark a new commutation relation. However, due to the type of commutations we find during the traversal step we can omit this. This is detailed in the proof of Theorem 4.49.

Given all this, we can now give the traversal algorithm in Algorithm 10 , which iteratively takes vertices from the stack visited_vertices, adds all possible new vertices to the stack and repeats until visited_vertices is empty.

4.43 Remark. Analogously to Algorithm 8, marking a vertex as zero means setting the corresponding field visited to True and adding it to the stack visited_vertices. This again takes constant time.

Algorithm 10: Monomial graph traversal

Input: (n,k)-monomial graph M, stack visited_vertices

```
while visited_vertices is not empty do
    v ← pop(visited_vertices);
    if v.length() = 2 then   /* commutation propagation edges */
        M.set_commutation(v.monomial()[1], v.monomial()[2]);
    foreach w in v.forwardedges() do      /* forward edges */
        if w.get_visited() is False then
            Mark the vertex w as zero;
    for i ← 1 to v.length() − 1 do      /* commutation edges */
        if M.get_commutation(v.monomial()[i], v.monomial()[i + 1])
         is True then
            w ← M.get_vertex(v.commutationedges()[i]);
            if w.get_visited() is False then
                Mark the vertex w as zero;
    foreach (r, c) in keys(v.backwardedges()) do
        (num_visited, non_visited_vertex) ← check_hypervertex(M, v,
         r, c);
        target ← M.get_vertex(v.backwardedges()[(r, c)][2]);
        if num_visited = n and target.get_visited() is False then
         /* backward edges */
            Mark the vertex target as zero;
        else if num_visited = n − 1 and target.get_visited() is True
         then                    /* sum edges, long monomial */
            Mark the vertex non_visited_vertex as zero;
    foreach (r, c, s) in keys(v.sumedges()) do      /* sum edges,
     short monomial */
        (num_visited, non_visited_vertex) ← check_hypervertex(M,
         v.sumedges()[(r, c, s)], r, c);
        if num_visited = n − 1 then
            Mark the vertex non_visited_vertex as zero;
```

Before computing the run time of Algorithm 10, we first give some statistics about the number of vertices, edges and hyperedges in the (n,k)-monomial graph which we will then use for the run time analysis.

4.44 Lemma. *The number of vertices in the (n,k)-monomial graph is given by $\sum_{l=1}^{k} n^{2l} = \frac{n^2 - n^{2(k+1)}}{1-n^2}$.*

Proof. The sum $\sum_{l=1}^{k} n^{2l}$ follows directly from counting—for a given length l there are n^{2l} different monomials of that length and we have a vertex for every monomial of length up to k. For the closed form, we consider the sum as a finite geometric series for which the closed form may be derived via a telescoping sum. □

4.45 Lemma. *The number of forward edges in the (n,k)-monomial graph is given by $\sum_{l=1}^{k-1} 2n^{2(l+1)} = 2\frac{n^4-n^{2(k+1)}}{1-n^2}$.*

Proof. Again, the sum follows from direct counting. For every vertex associated to a monomial of length up to $k-1$, there are n^2 forward edges corresponding to multiplying the monomial with a generator from the left and n^2 forward edges corresponding to multiplying the monomial with a generator from the right. This gives $\sum_{l=1}^{k-1} n^{2l}(2n^2) = \sum_{l=1}^{k-1} 2n^{2(l+1)}$. Using the closed form from Lemma 4.44 in order to count the vertices of length up to $k-1$, the closed form follows from direct multiplication. □

4.46 Lemma. *The number of all commutation edges in the (n,k)-monomial graph is given by $\sum_{l=1}^{k} n^{2l}(l-1) = n^2\frac{(k-1)n^{2(k+1)}-kn^{2k}+n^2}{(n^2-1)^2}$.*

Proof. For every vertex associated to a monomial of length $l \in \{2,\dots,k\}$ switching any two consecutive generators gives rise to a commutation edge. Therefore, there are $(l-1)$-many commutation edges for each vertex of length l. Summing up over all vertices of length at least two then almost yields the sum from the claim and noting that the summand for $l=1$ is zero yields the claimed sum. The closed form of the sum is again derived using a telescoping sum. Since this might be less well known than the derivation used for Lemma 4.44, we show it here. Denote by s_k the sum $\sum_{l=1}^{k}(l-1)c^l$ for any constant $c \neq 1$. Next, consider the telescoping sum

$$\begin{aligned}
(c-1)^2 s_k &= c^2 s_k - 2cs_k + s_k \\
&= \sum_{l=1}^{k} (l-1)c^{l+2} - 2(l-1)c^{l+1} + (l-1)c^l \\
&= 0c^1 + c^2 + \left(\sum_{l=3}^{k} (l-3-2(l-2)+l-1)c^l\right) - kc^{k+1} + (k-1)c^{k+2} \\
&= c^2 - kc^{k+1} + (k-1)c^{k+2} \\
&= c((k-1)c^{k+1} - kc^k + c).
\end{aligned}$$

Dividing both sides by $(c-1)^2$ and putting $c = n^2$ yields the claim. □

4.47 Lemma. *The number of backward edges in the (n,k)-monomial graph is given by $\sum_{l=2}^{k} 2ln^{2l-1} = \frac{2n^3}{(n^2-1)^2}(kn^{2k} - (k+1)n^{2(k-1)} - n^2 + 2)$.*

Proof. In order to count the number of backward edges we count the number of sources of backward edges. Recall that for a vertex associated to a monomial u of length l with $l \in \{2,\dots,k\}$ each choice of character $c \in \{\texttt{'i'}, \texttt{'j'}\}$ and position $r \in [l]$ in u yields a backward vertex $B_{u,r}^{(c)}$ that vertex is part of. Thus, we count $2l$ backward edges per vertex of length l of which there are n^{2l}. However, recall also that each backward vertex $B_{u,r}^{(c)}$ has n elements and thus we have counted each backward edge n times. To remedy this, we have to divide by n and get the sum $\sum_{l=2}^{k} 2ln^{2l-1}$. The closed form follows again from a telescoping sum, this time multiplying the sum with $(n^2-1)^2$. The remaining derivation is analogous to the one in the proof of Lemma 4.46. □

4.48 Lemma. *The number of sum edges in the (n,k)-monomial graph is given by $\sum_{l=2}^{k} 2ln^{2l} = \frac{2n^4}{(n^2-1)^2}(kn^{2k}-(k+1)n^{2(k-1)}-n^2+2)$. The number of all representations of sum edges stored in the (n,k)-monomial graph data structure* M*, i.e. the sum of* `length(v.sumedges())` *over all vertices* v *in* M*, is given by $\sum_{l=1}^{k-1} 2(l+1)n^{2l+1} = \frac{2n^3}{(n^2-1)^2}(kn^{2k} - (k+1)n^{2(k-1)} - n^2 + 2)$.*

Proof. First, we count the actual number of sum edges. For every vertex v associated to a monomial u of length $l \in \{2,\dots,k\}$ and choice of position $r \in [l]$ in u and character $c \in \{\texttt{'i'},\texttt{'j'}\}$ we get exactly one sum edge $(S^{(c)}_{u,r}, u) \in E_S$. Summing over all vertices of length at least two yields the desired sum and using a telescoping sum again gives the closed form.

In order to count the number of all elements of `sumedges()` we similarly get that for every vertex v associated to a monomial u of length $l \in [k-1]$ and choice of position $r \in [l+1]$, character $c \in \{\texttt{'i'},\texttt{'j'}\}$ and index $s \in [n]$ there is a sum edge representation stored at `v.sumedges()[(r,c,s)]`. Again, summing over all vertices of length at most $k-1$ and using a telescoping sum yields the claims. □

4.49 Theorem. *Starting with the initial zeros on the stack* visited_vertices*, Algorithm 10 finds any zero in $C(G^+_{aut}(\Gamma))$ that a repeated application of the propagation rules (Pi)–(Ci) limited to monomials of maximal length k can produce. The algorithm terminates in time $\mathcal{O}(kn^{2k+1})$.*

Proof. For the correctness claim, note that when limiting the maximal length of monomials considered, there is only a finite number of possible applications of the propagation rules. Further, there may be several sequences leading to fulfilling the conditions for any particular rule. However, as the number of possible applications of rules is finite so is the length of any such sequence. In order to show that any zero produced by a sequence of the propagation rules is found by our algorithm we will show that for any rule whose prerequisites are met the algorithm finds the zero that this rule would yield. Applying this iteratively then yields the claim.

We now perform a case-by-case analysis of the different types of propagation rule. Note here that every vertex w gets added to the stack visited_vertices at most once which happens when v gets marked as a zero and which is always preceded by a check for `w.get_visited`. Thus, since there are only finitely many vertices in the (n,k)-monomial graph and since in every iteration of the outermost while loop one vertex is removed from the stack, the algorithm terminates. Further, every found zero will have been added to the stack at some point and will be removed from it to be the vertex v in the main while loop of the algorithm. We say that the vertex v is *being visited* when v is the vertex obtained by calling `pop(visited_vertices)` in line 2. The vertex associated to any known zero will thus be added to the stack visited_vertices at some point and later be visited.

For our analysis, we assume that the conditions for applying the given rule are met. The rule gets applied correctly if the resulting zero is marked as zero before the algorithm terminates. We assume that the resulting zero has not been found before as then the step would be correct by default.

(Pi) The prerequisites for applying the forward propagation rule is that the we have

$$u = u_{i_1 j_1} \dots u_{i_m j_m} = 0.$$

We want to show that for a generator u_{kl} the monomials uu_{kl} and $u_{kl}u$ are equal to zero.

Since $u = 0$ is known we will eventually visit its associated vertex `v`. By design of the monomial graph both uu_{kl} and $u_{kl}u$ are in the list `v.forwardedges()` and thus they get marked as zero (see lines 5–7).

(Pii) The prerequisites for applying the first backward propagation rule is that we have

$$u_{i_1j_1} \dots u_{kj_r} \dots u_{i_mj_m} = 0$$

for all $k \in [n]$. We want to show that $\hat{u} = u_{i_1j_1} \dots u_{i_{r-1}j_{r-1}} u_{i_{r+1}j_{r+1}} \dots u_{i_mj_m}$ is equal to zero.

The condition can be shortened by saying that every element of $B_{u,r}^{(i)}$ is equal to zero and therefore every associated vertex `v` will be visited. Recall, that `v.backwardedges()` is a dictionary mapping every pair (r, c) for $r \in [m], c \in \{\texttt{'i'}, \texttt{'j'}\}$ to data used to determine the backward edge $(B_{u,r}^{(i)}, \hat{u})$. The number of visited vertices of $B_{u,r}^{(i)}$—and if available the non-visited vertices—are then found using the function `check_hypervertex` as mentioned in Remark 4.41. In lines 13 and 14 every set $B_{u,r}^{(i)}$ of which the vertex `v` is a part of gets examined in this way. At the latest when visiting the last vertex of $B_{u,r}^{(i)}$ all n vertices must previously have been marked as zero and `num_visited` returned by `check_hypervertex` will be equal to n. By design of the monomial graph the vertex associated to the monomial $\hat{u}$ is obtained and marked as zero in lines 15–17.

(Piii) The second backward propagation rule follows analogously to the first one with the condition being that every element of $B_{u,r}^{(j)}$ is zero.

(Piv) The prerequisites for applying the commutation propagation rule is that we have

$$u = u_{i_1j_1} \dots u_{i_mj_m} = 0$$

and

$$u_{i_rj_r} u_{i_{r+1}j_{r+1}} = u_{i_{r+1}j_{r+1}} u_{i_rj_r}.$$

We want to show that $u_{i_1j_1} \dots u_{i_{r+1}j_{r+1}} u_{i_rj_r} \dots u_{i_mj_m}$ is equal to zero.

As our condition consists of two parts, we need to distinguish several cases again. Depending on which part of the condition was met first the resulting vertex is reached through differing ways.

(1) If the commutation of $u_{i_rj_r}$ and $u_{i_{r+1}j_{r+1}}$ is known by the time that `v` is visited, then lines 8–12 yield the desired result.

(1) If `v` is visited before the commutation of $u_{i_rj_r}$ and $u_{i_{r+1}j_{r+1}}$ is known we need to consider how the new information of the commutation is obtained. In the traversal algorithm the only way a new commutation is found is by noticing that a monomial of length two is equal to zero. Thus, we either require that $u_{i_rj_r}u_{i_{r+1}j_{r+1}} = 0$ or that $u_{i_{r+1}j_{r+1}}u_{i_rj_r} = 0$. We can assume, that the vertex `v'` associated to the monomial $u_{i_{r+1}j_{r+1}}u_{i_rj_r}$ is

the one being visited. Otherwise, when visiting the vertex associated to $u_{i_r j_r} u_{i_{r+1} j_{r+1}}$ line 3 marks the commutation and then lines 8–12 add the vertex v' to the stack visited_vertices. When visiting v' iterated applications of the forward propagation rule eventually yields that the monomial $u_{i_1 j_1} \dots u_{i_{r+1} j_{r+1}} u_{i_r j_r} \dots u_{i_m j_m}$ is equal to zero.

(Pv) The prerequisites for applying the first sum propagation rule is that we have

$$u_{u_{i_1 j_1} \dots u_{i_{kj_r}} \dots u_{i_m j_m}} = 0$$

for all $k \in [n], k \neq k'$ and that

$$\hat{u} = u_{i_1 j_1} \dots u_{i_{r-1} j_{r-1}} u_{i_{r+1} j_{r+1}} \dots u_{i_m j_m} = 0.$$

We want to show that $u = u_{i_1 j_1} \dots u_{k' j_r} \dots u_{i_m j_m}$ is equal to zero.

The condition can be shortened by saying that every element of $S^{(i)}_{u,r}$ is equal to zero and therefore every associated vertex v will be visited. Due to the nature of $S^{(i)}_{u,r} = (B^{(i)}_{u,r} \setminus \{u\}) \cup \{\hat{u}\}$, we again need to distinguish two cases.

(1) If the short monomial $\hat{u}$ is not the last element of $S^{(i)}_{u,r}$ that is visited then one of the long monomials is. Once it is visited the same examination as for the backward propagation rule in lines 13 and 14 will return num_visited $= n-1$ as mentioned in Remark 4.41. By design of the monomial graph the vertex associated to $\hat{u}$ is given by target and the vertex associated to u is given by non_visited_vertex. In line 18 the vertex target is then also examined to see that $\hat{u} = 0$ and the monomial u is found to be zero.

(1) Otherwise the short monomial $\hat{u}$ is the last element of $S^{(i)}_{u,r}$ to be visited. Recall that v.sumedges() is a dictionary mapping a tuple $(r, \texttt{'i'}, s)$ for $r \in [m], s \in [n]$ to an element $u' = u_{i_1 j_1} \dots u_{i' \mathsf{s}} \dots u_{i_m j_m}$ with $i' \in [n]$ arbitrary (that is an element of $B^{(i)}_{u,r}$). As mentioned in Remark 4.41 if calling check_hypervertex on this element u' returns num_visited $= n-1$ then both $\hat{u}$ and all elements of $B^{(i)}_{u,r}$ except for one are zero. This non-zero element is returned as non_visited_vertex. Due to the assumptions we have made we know this missing element to be u. Therefore, in line 23 u is marked as zero.

(Pvi) The second sum propagation rule follows analogously to the first sum one with the condition being that every element of $S^{(j)}_{u,r}$ is zero.

For the run time consider the following. Any vertex gets added to and removed from the stack at most once and is therefore visited at most once. Observe that commutation propagation edges take constant time to be followed. Further, for both forward and commutation edges note that each edge gets followed at most once while incurring only constant time costs. Next, each backward vertex gets checked up to n-times, once for each of its elements. Each of these checks incurs a cost of $\mathcal{O}(n)$ for the call to check_hypervertex in line 14. Following a backward edge in lines 16–17 only takes constant time as does following a potentially found sum edge

in lines 18–19. Finally, for each element stored the dictionaries `sumedges` we again incur a cost of $\mathcal{O}(n)$ for the call to `check_hypervertex` in line 21. Following the sum edge in line 23 happens in constant time.

Thus, the run time can be found as the sum of the number of vertices, the number of forward and commutation edges, the number of backward edges multiplied with $n\mathcal{O}(n)$ and the number of stored elements in the dictionaries `sumedges` multiplied with $\mathcal{O}(n)$. By Lemmas 4.44–4.48, this yields

$$\mathcal{O}(n^{2k} + n^{2k} + kn^{2k} + n^2kn^{2k-1} + nkn^{2k-1}) = \mathcal{O}(kn^{2k+1}).$$

□

4.50 Remark. As mentioned in the proof of Theorem 4.49 the only new commutation relations found are those that occur due to a monomial of length two being zero. Thus, the commutation edges will only yield "new" zeros if the commutation relation is known beforehand and is non-trivial. However, by default the only commutation relations we know are trivially those of each generator commuting with itself as can be seen by the commutation matrix being initialized as the identity matrix. There are still good reasons to both track the commutations of generators during the traversal algorithm and to include the commutation edges in the monomial graph. Firstly, if we know of a non-trivial commutation relation we can add it before running the traversal algorithm. Then, the commutation edges can find zeros that might otherwise not be discovered. Secondly, tracking the commutations of generators allows us to check after the traversal algorithm whether all generators commute. If so the quantum automorphism group $C(G^+_{aut}(\Gamma))$ commutes. By Definition 2.1 this means that Γ has no quantum symmetries. Lastly, after running the traversal algorithm we can extract additional commutation relations and rerun the algorithm. This is shown in the following section.

4.5 Post-processing

Having run the traversal algorithm, we may still find out additional structure from the traversed monomial graph by utilising further algebraic relations. They do not yield additional zeros as these should have been found during either the start zeros step or the traversal step to be used during the traversalstep, but rather give additional commutation relations. We present two such post-processing steps here. The first of them utilises Lemma 2.5. The second one uses the fact that if there are $i, r' \in [n]$ and $u_{ir} = 0$ for all $r \neq r'$, then we get

$$u_{ir'} = \sum_{r=1}^{n} u_{ir} = 1$$

and thus $u_{i,r'}$ commutes with all other generators. The same holds true for $u_{s'j}$ if there are $j, s' \in [n]$ with $u_{s'j} = 0$ for all $s \neq s'$. Analogously to the start zeros step, if one finds additional relations that yield further commutation relations one may add them here to find more commutations.

For the first post-processing step we consider the following algorithm that checks for a single monomial $u_{i,j}u_{r,s}$ of length 2 whether it is equal to $u_{i,j}u_{r,s}u_{i,j}$.

Algorithm 11: Criterion 1: Lemma 2.5

Input: (n,k)-monomial graph M, vertex v $= u_{ij}u_{rs}$ of length 2 such that v.get_visited() is False and M.get_commutation(u_{ij}, u_{rs}) is False

```
1  i_flag ← True;
2  j_flag ← True;
3  foreach w in v.forwardedges() do
4      if not v.monomial() = w.monomial()[1 : 2] then
5          continue;
6      if v.monomial()[1] = w.monomial()[3] then
7          if w.get_visited() is True then
8              return False;
9      else if v.monomial()[1][1] = w.monomial()[3][1] then
10         if w.get_visited() is False then
11             j_flag ← False;
12     else if v.monomial()[1][2] = w.monomial()[3][2] then
13         if w.get_visited() is False then
14             i_flag ← False;
15 return i_flag or j_flag;
```

4.51 Proposition. *Given the (n,k)-monomial graph* M *and a vertex* v *associated to a monomial $u_{ij}u_{rs}$ that is not known to be zero in* M *and where u_{ij} and u_{rs} are not known to commute, Algorithm 11 returns* True *if and only if either $u_{i,j}u_{r,s}u_{i,h} = 0$ is known for all $h \neq j$ or $u_{i,j}u_{r,s}u_{g,j} = 0$ is known for all $g \neq i$ or both in* M*. The algorithm runs in time $\mathcal{O}(n^2)$.*

Proof. Let v be a vertex in the (n,k)-monomial graph M associated to a monomial $u_{ij}u_{rs}$ such that v.get_visited() is False and such that u_{ij} and u_{rs} are not known to commute. Note that this already excludes monomials of the form $u_{ij}u_{ij}$ as any generator is known to commute with itself. Further, let w be a vertex in v.forwardedges(). Then w is either associated to a monomial of the form $u_{gh}u_{ij}u_{rs}$ or to a monomial of the form $u_{ij}u_{rs}u_{gh}$. In the first case the condition in line 4 is met and w is not further examined. In the second case the condition in line 4 is not met and we may distinguish more cases.

First, let w be the vertex associated to the monomial $u_{ij}u_{rs}u_{ij}$. Then the condition in line 6 is met. If now the monomial is known to be zero we have that $u_{ij}u_{rs} \neq 0 = u_{ij}u_{rs}u_{ij}$ in M. The condition in line 7 is also met and the algorithm returns False.

If instead w is associated to a monomial $u_{ij}u_{rs}u_{ih}$ with $h \neq j$ then the condition in line 9 is met and if the monomial is not known to be zero in M j_flag is set to False. Analogously, this applies to monomials $u_{ij}u_{rs}u_{gj}$ with $g \neq i$ and the variable i_flag.

In all other cases, that is if w is associated to a monomial $u_{ij}u_{rs}u_{gh}$ with both $g \neq i$ and $h \neq j$, nothing happens.

After the loop terminates in line 15 the variable i_flag only remains True, if all monomials $u_{ij}u_{rs}u_{ih}$ with $h \neq j$ are known to be zero in M and is set to false

otherwise. Again, the same applies to j_flag and monomials $u_{ij}u_{rs}u_{gj}$ with $g \neq i$. Thus, the algorithm returns True if either $u_{ij}u_{rs}u_{ih} = 0$ is known for all $h \neq j$ or if $u_{ij}u_{rs}u_{gj} = 0$ is known for all $g \neq i$ or both.

Since the body of the for loop in line 3 runs in constant time and there are $2n^2$ elements in v.forwardedges() the whole algorithm runs in time $\mathcal{O}(n^2)$. □

4.52 Corollary. *Algorithm 11 called on the vertex* v *associated to the monomial* $u_{ij}u_{rs}$ *returning* True *is a sufficient condition for* u_{ij} *and* u_{rs} *commuting.*

Proof. If the algorithm returns true, without loss of generality, we may assume that $u_{ij}u_{rs}u_{ih} = 0$ for all $h \neq j$. This gives

$$u_{ij}u_{rs} = u_{ij}u_{rs}\sum_{h=1}^{n} u_{ih} = u_{ij}u_{rs}u_{ij}$$

and with Lemma 2.5 this yields the claim. □

For the second post-processing step we want to check if $u_{ij} = 1$. Once again, we achieve this by examining how many elements of the set $\{u_{is} \mid s \in [n]\}$ or the set $\{u_{rj} \mid r \in [n]\}$ are zero. If $(n-1)$ elements of either set are zero then the remaining element is 1. We once again use the function compute_next_index to handle the "wrapping" modular arithmetic when iterating over the sets.

Algorithm 12: Criterion 2: $u_{ij} = 1$

Input: (n,k)-monomial graph M, vertex v associated to monomial u_{ij} of length 1

```
1  for index in {'i', 'j'} do
2      break_flag ← False;
3      w ← v;
4      for r ← 1 to n − 1 do
5          w ← M.vertexlist[1][compute_next_index(w.monomial(), index,
             1, n)];
6          if w.get_visited() is False then
7              break_flag ← True;
8              break;
9      if break_flag is False then
10         return True;
11 return False;
```

4.53 Proposition. *Algorithm 12 called on a vertex* v *associated to a monomial* u_{ij} *that has not been visited in* M *returns* True *if* v *is known to be equal to* 1 *in* M. *The algorithm runs in time* $\mathcal{O}(n)$.

Proof. Let v be a vertex associated to a monomial u_{ij} such that v.get_visited() is False. First, we consider the case where index is set to 'i'. The function compute_next_index(v.monomial(), 'i', 1, n) returns the index of $u_{i'j}$ where $i' = (i \mod n) + 1$. Thus, the up to $(n-1)$-many calls of line 5 iteratively return

the vertices w associated to the monomials u_{rj} where $r \neq i$. If all of these vertices are found to be zero in M we can infer that u_{ij} is equal to 1 due to the relation $\sum_{r=1}^{n} u_{rj} = 1$. This is handled in the algorithm by using the variable break_flag. If all of the vertices have been found to be equal to zero break_flag remains False until line 9 and the algorithm returns True. Otherwise, we are in the case where index is `'j'` and we repeat the check for the set of vertices associated to the monomials u_{is} where $s \neq j$. If at the end of this check u_{ij} could still not be determined to be equal to 1 the algorithm returns False.

The call to `compute_next_index` takes time $\mathcal{O}(\texttt{length}(\texttt{w.monomial}())) = \mathcal{O}(1)$ and therefore the run time depends only on the number of repetitions of the loops which is $2(n-1)$. Thus, the entire algorithm runs in time $\mathcal{O}(n)$. □

With these two functions, the post-processing step is given by the following algorithm.

Algorithm 13: Post-processing

Input: (n, k)-monomial graph M

```
   // Criterion 1
 1 if k > 2 then
 2     foreach v in M.vertexlist[2] do
 3         if v.get_visited() is False and
            M.get_commutation(v.monomial(1), v.monomial(2)) is
            False then
 4             if Criterion 1 returns True then
 5                 M.set_commutation(v.monomial()[1],
                    v.monomial()[2]);
   // Criterion 2
 6 foreach v in M.vertexlist[1] do
 7     if v.get_visited() is False then
 8         if Criterion 2 returns True then
 9             for i, j ← 1 to n do
10                 M.set_commutation(v.monomial()[1], (i, j));
```

4.54 Proposition. *Algorithm 13 marks all commutation relations that can be inferred from Algorithms 11 and 12 in* M*. It runs in time* $\mathcal{O}(n^6)$.

Proof. Algorithm 11 is called on every vertex corresponding to a monomial $u_{ij}u_{kl}$ of length two that is not known to be equal to zero and whose two generators u_{ij} and u_{kl} are not known to commute in lines 2–5. By Corollary 4.52 Algorithm 11 returning True is a sufficient condition for u_{ij} and u_{kl} commuting. This is set in line 5.

Algorithm 12 is called on every vertex corresponding to a generator u_{ij} that is not known to be equal to zero in lines 6–10. If Algorithm 12 returns True the generator u_{ij} is equal to 1 and commutes with all generators of $C(G_{aut}^{+}(\Gamma))$. This is set lines 9 and 10.

There are n^4 vertices in M.vertexlist[2]. Algorithm 11 may be called with each of them which by Proposition 4.51 takes time $\mathcal{O}(n^2)$. The remainder of the loop in lines 2–5 takes constant time. This yields a run time of $n^4\mathcal{O}(n^2) = \mathcal{O}(n^6)$ for the first loop. The loop in lines 6–10 iterates over all monomials of length one, of which there are n^2. For every one of them Algorithm 12 may be called which by Proposition 4.53 takes time $\mathcal{O}(n)$. Next, the nested loop in line 9 iterates over n^2 elements and its body runs in constant time. This yields a run time of $\mathcal{O}(n^4)$ for the second loop and thus the entire algorithm runs in time $\mathcal{O}(n^6)$. □

Finally, in order to utilize the commutation relations found in the post-processing step the traversal step is run again. In order to do so, all vertices in the monomial graph M are marked as non-visited and Algorithm 8 is called again to mark the start zeros. This is repeated until no new zeros or commutations are found or until the quantum automorphism group is found to be commutative.

Algorithm 14: Iterated traversal of (n, k)-monomial graph

Input: (n, k)-monomial graph M, input graph Γ

```
1 repeat
2 |   Reset all vertices in M to non-visited;
3 |   Compute start zeros of M using Γ;
4 |   Run traversal algorithm on M;
5 |   Run post-processing on M;
6 until C(G^+_aut(Γ)) is found to be commutative or the number of zeros and
    commutations is no longer increased;
```

4.55 Remark. The end conditions of Algorithm 14 are computed as follows. If all entries of the commutation matrix commutationtable are equal to one, $C(G^+_{aut}(\Gamma))$ is commutative. Checking this requires checking each entry which takes time $\mathcal{O}(n^4)$. The number of zeros is determined by iterating over all vertices and counting the visited vertices. This takes time $\mathcal{O}(n^{2k})$.

Importantly, since there are only finitely many vertices in the (n, k)-monomial graph Algorithm 14 is guaranteed to terminate.

5 Julia Implementation and Results

The algorithms and data structures presented in the previous section have been implemented in the Julia programming language [4]. We list the essential functions here.

- `createk1monomialgraph(A)`: Implementation of Algorithm 3. Takes an adjacency matrix `A` of a graph Γ, extracts the number of vertices n and returns the $(n, 1)$-monomial graph. The monomial graph is represented as a Julia object whose type is a **`struct`** according to Data structure 1.

- `createnextmonomialgraph(m)`: Implementation of Algorithm 7. Takes the (n, k)-monomial graph `m` and returns the $(n, k+1)$-monomial graph. We also provide the function `createmonomialgraph(A,k)` that automatically uses the functions `createk1monomialgraph` and `createnextmonomialgraph` in order to return the (n, k)-monomial graph.

- `startzeros!(q, m, A)`: Implementation of Algorithm 8. Takes the stack `q`, the (n, k)-monomial graph `m` and the adjacency matrix `A` of the graph Γ, then computes the distance matrix D_Γ and degree vector deg of Γ and marks the start zeros according to Section 4.3. The stack provided is simply an empty `Vector` of `Vertex` (created via `q = Vertex[]`) where `Vertex` is a **`struct`** according to Data structure 2. This is due to the fact that Julia's `Vector` already implements a stack using the functions `push!`, `pop!` and `isempty`. In order to compute the distance matrix we use the package `MaxPlus.jl` that provides the Min-plus product used in Corollary 4.35.

- `traverse!(q, m)`: Implementation of Algorithm 10. Takes the stack `q` of known zeros and the (n, k)-monomial graph `m` and runs the traversal algorithm outlined in Section 4.4. The function returns the number of zeros found.

- `postprocessing(m)`: Implementation of Algorithm 13. Takes the (n, k)-monomial graph `m` and runs the post-processing steps outlined in Section 4.5.

We also provide useful functions like `computemonomialindex(u, n)` that gives the monomial index of the monomial `u`, `getvertex(vi, m)` that takes a vertex index and a monomial graph and returns the vertex corresponding to the index, or `getcommute(v, m)` that takes a monomial `v` of length 2 and a monomial graph `m` and returns `true` if and only if the generators making up `v` are found to commute in `m`.

The workflow for creating the (n, k)-monomial graph, finding start zeros, running the traversal algorithm and running the post-processing steps on a graph Γ then is as follows. We assume that the graph is represented via its adjacency matrix stored in the variable `A`.

```
m = createmonomialgraph(A, k);
q = Vertex[];
startzeros!(q, m, A);
traverse!(q, m);
postprocessing!(m);
```

Afterwards one may for example inspect individual monomials on having been found to equal zero or directly check the commutation of any two generators. These two things would be achieved with the following code.

```
# u1 contains the monomial one is interested in
mi = computemonomialindex(u1, n);
vi = Vertexindex(mi, u1.length);
v = getvertex(vi, m);
@show(v.visited); # show whether u1 has been found to be zero

# u2 contains the monomial ab where one is interested in the commutation of
    generators a and b
@show(getcommute(u2, m)); # show whether a and b have been found to commute
```

If one wants to rerun the traversal algorithm after the post-processing step one may call the function `resetvisit!(m)` after which `startzeros!`, `traverse!` and `postprocessing!` may be called again. In order to simplify this, we have also implemented the function `iteratetraverse!(q, m, Γ)` that implements Algorithm 14, i.e. it automatically reruns the traversal and post-processing algorithms until either all generators are found to commute with each other or no new information is found. It returns `true` if and only if $\mathrm{Aut}(\Gamma)$ has been found to be commutative. Using this function the workflow is simplified to the following:

```
m = createmonomialgraph(A, k);
q = Vertex[];
iteratetraverse!(q, m, A);
```

In the following sections we will present the results running the algorithm on different input graphs with different choices of the parameter k. In general there is a trade off between the run time increasing exponentially with k on the one hand and the possibility to find more zeros and commutation relations on the other hand. As we know for many of the graphs in this section whether they have quantum symmetries or not this will serve as a way to validate the theoretical results from Section 4. Recall that Algorithm 14 showing that all generators commute pairwise is a sufficient condition for the graph not having quantum symmetries. However, if the algorithm does not give this result we cannot infer the converse. Rather, it is unclear whether the graph does have quantum symmetries indeed or if choosing a larger value for k would yield the desired result.

We have benchmarked these trials using the package `TimerOutputs.jl`. We have run the implementation on both a laptop (Lenovo Thinkpad 13 2nd Gen with Intel i7-7500U and 8Gb RAM) and a server cluster with state of the art hardware.

5.1 Simple Graphs with Four Vertices

As mentioned in Section 3.2, there are eleven different simple graphs with four vertices; six of which do have quantum symmetries and five of which do not. We enumerate them here again and give them names in order to reference them throughout this section.

(1)  : K_4^c

(2)  : K_4

(3)  : diamond^c

(4)  : diamond

(5)  : paw^c

(6)  : paw

(7)  : $C_4^c = K_{2,2}^c$

(8)  : $C_4 = K_{2,2}$

(9)  : claw$^c = K_{1,3}^c$

(10)  : claw $= K_{1,3}$

(11)  : P_4

The claw graph $K_{1,3}$ is one of the graphs that does not admit quantum symmetries. Recall from Proposition 3.6 that we can show that any two generators of $C(G_{aut}^+(K_{1,3}))$ commute through purely algebraic means. Doing so did require considering monomials up to length 3. Since Algorithms 8, 10 and 13 are modelled after this algebraic method we expect them to also get to the same result. And indeed, with $k = 3$ calling `iteratetraverse!` does indeed return `true` thus showing that the claw graph does not have quantum symmetries. Running the algorithm with $k = 2$ returns `false`, implying that our calculations from Proposition 3.6 are optimal in the sense that we cannot find an algebraic proof using only the relations we consider as propagation rules that does not require monomials of length at least 3.

Indeed, when considering all graphs with four vertices and using the parameter $k = 2$ the algorithm did not find any graph to have a commutative quantum automorphism group. We show the results for $k = 3$ in the Table 3 .

Table 3: Comparison of existence of quantum symmetries and output of `iteratetraverse!` with $k = 3$ for all simple graphs with four vertices

Graph name	Graph has quantum symmetries	`iteratetraverse!` returns `true`
K_4^c	Yes	No
K_4	Yes	No
diamond^c	Yes	No
diamond	Yes	No
paw^c	No	Yes
paw	No	Yes
$C_4^c = K_{2,2}^c$	Yes	No
$C_4 = K_{2,2}$	Yes	No
claw$^c = K_{1,3}^c$	No	Yes
claw $= K_{1,3}$	No	Yes
P_4	No	Yes

As one can immediately see, with the choice of $k = 3$ the values returned by `iteratetraverse!` correlate with the existence of quantum symmetries.

Running the implementation with $k = 3$ on the Lenovo Thinkpad, we got the following run times presented in Table 4.

Table 4: Run times of Julia implementation for graphs on four vertices ($n = 4, k = 3$) on the Lenovo Thinkpad. If there are multiple calls to `startzeros!`, `traversal!` and `postprocessing!` during `iteratetraverse!` we show each call of `traversal!` and `postprocessing!` but only the first call of `startzeros!` since the starting conditions for each call of `startzeros!` stay the same.

Algorithm step		Time in seconds	Graph has quantum symmetries
`createk1monomialgraph`		$15 * 10^{-6}$	–
`createnextmonomialgraph`		$11.5 * 10^{-3}$	–
`createnextmonomialgraph`		$229 * 10^{-3}$	–
K_4^c	`startzeros!`	$95 * 10^{-6}$	Yes
	`traversal!`	$40.1 * 10^{-3}$	
	`postprocessing!`	$900 * 10^{-6}$	
	`traversal!`	$40.7 * 10^{-3}$	
	`postprocessing!`	$1.07 * 10^{-3}$	
K_4	`startzeros!`	$99.1 * 10^{-6}$	Yes
	`traversal!`	$41.3 * 10^{-3}$	
	`postprocessing!`	$868 * 10^{-6}$	
	`traversal!`	$64.2 * 10^{-3}$	
	`postprocessing!`	$945 * 10^{-6}$	

Table 4: Continued

Algorithm step		Time in seconds	Graph has quantum symmetries
diamond^c	`startzeros!`	$113 * 10^{-6}$	Yes
	`traversal!`	$58.4 * 10^{-3}$	
	`postprocessing!`	$290 * 10^{-6}$	
	`traversal!`	$58.1 * 10^{-3}$	
	`postprocessing!`	$240 * 10^{-6}$	
diamond	`startzeros!`	$109 * 10^{-6}$	Yes
	`traversal!`	$58.8 * 10^{-3}$	
	`postprocessing!`	$299 * 10^{-6}$	
	`traversal!`	$58.4 * 10^{-3}$	
	`postprocessing!`	$238 * 10^{-6}$	
paw^c	`startzeros!`	$152 * 10^{-6}$	No
	`traversal!`	$61.4 * 10^{-3}$	
	`postprocessing!`	$203 * 10^{-6}$	
paw	`startzeros!`	$125 * 10^{-6}$	No
	`traversal!`	$181.4 * 10^{-3}$	
	`postprocessing!`	$215 * 10^{-6}$	
$C_4^c = K_{2,2}^c$	`startzeros!`	$144 * 10^{-6}$	Yes
	`traversal!`	$96.8 * 10^{-3}$	
	`postprocessing!`	$570 * 10^{-6}$	
	`traversal!`	$56 * 10^{-3}$	
	`postprocessing!`	$470 * 10^{-6}$	
$C_4 = K_{2,2}$	`startzeros!`	$151 * 10^{-6}$	Yes
	`traversal!`	$55.4 * 10^{-3}$	
	`postprocessing!`	$578 * 10^{-6}$	
	`traversal!`	$55.5 * 10^{-3}$	
	`postprocessing!`	$482 * 10^{-6}$	
claw^c = $K_{1,3}^c$	`startzeros!`	$141 * 10^{-6}$	No
	`traversal!`	$58.9 * 10^{-3}$	
	`postprocessing!`	$401 * 10^{-6}$	
claw = $K_{1,3}$	`startzeros!`	$128 * 10^{-6}$	No
	`traversal!`	$58.6 * 10^{-3}$	
	`postprocessing!`	$401 * 10^{-6}$	
P_4	`startzeros!`	$176 * 10^{-6}$	No
	`traversal!`	$61.3 * 10^{-3}$	
	`postprocessing!`	$219 * 10^{-6}$	

5.1 Remark. There are a few main takeaways from this data.

- Each individual step takes less than 0.25 seconds.
- The step that takes the longest is the call to `createnextmonomialgraph` that creates the $(4, 3)$-monomial graph from the $(4, 2)$-monomial graph. However, when running the traversal step for several graphs with the same number of vertices, the corresponding monomial graph may be reused by calling

`resetvisit!` instead of building the monomial graph up from scratch every time.

- The algorithm terminated after the first iteration for the graphs without quantum symmetries. This is due to the fact that the first iteration already got the result showing that the quantum automorphism group is commutative. For the graphs with quantum symmetries the second iteration does not yield any new commutations or zeros and thus leads to a termination of `iteratetraverse!`.
- The longest complete computation (excluding the creation of the monomial graph) was the computation of the paw graph with just over 0.181 seconds. Running `iteratetraverse!` for all simple graphs with four vertices takes 1.115 seconds. Given this minimal run time on the Lenovo Thinkpad, we did not test the graphs with four vertices on the server cluster.

5.2 Simple Graphs with Six and Seven Vertices

There are 156 simple graphs with six vertices of which 112 are connected. One of these graphs, the smallest non-trivial graph with trivial automorphism group, has been presented previously in this thesis.

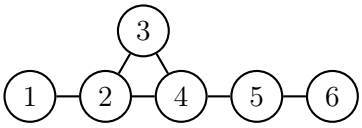


Figure 12: The smallest non-trivial graph with trivial automorphism group.

We showed in Proposition 3.1 that this graph does not have quantum symmetries. The proof again relied on the relations we used in the design of our traversal algorithm and did not require monomials of length greater than two. Indeed, calling `iteratetraverse!` with $k = 2$ on this graph confirms that the graph does not have quantum symmetries. Running the implementation with $k = 2$ on both the Lenovo Thinkpad and the server cluster, we got the following run times shown in Table 5.

Table 5: Run time of Julia implementation of `createmonomialgraph` and `iteratetraverse!` with $n = 6, k = 2$ for the smallest non-trivial graph with trivial automorphism group.

Algorithm step	Time in seconds on Lenovo Thinkpad	Time in seconds on server cluster
`createk1monomialgraph`	$19.2 * 10^{-6}$	$17.6 * 10^{-6}$
`createnextmonomialgraph`	$88.6 * 10^{-3}$	$16.2 * 10^{-3}$
`startzeros!`	$600 * 10^{-6}$	$460 * 10^{-6}$
`traversal!`	$24.2 * 10^{-3}$	$13.1 * 10^{-3}$
`postprocessing!`	$58.8 * 10^{-6}$	$58.0 * 10^{-6}$

As is expected, the server cluster is consistently faster compared to the laptop. However, for n and k as small as here even the laptop takes less than a second to

terminate.

In order to interpret the results further, we included a logging function in `traversal!` and `iteratetraverse!`. The log consists of several parts. The first part contains general information about the graph and statistics about the traversal. The second part contains run time information. The third part finally lists all zeros found during the traversal.

In case of this example graph (which here is called ECZG in the *graph6* format), the first part of the log of `iteratetraverse!` will look as follows.

```
Graph: ECZG

Adjacency Matrix:
 0  0  0  1  0  1
 0  0  0  0  1  1
 0  0  0  0  1  0
 1  0  0  0  0  0
 0  1  1  0  0  1
 1  1  0  0  1  0

Parameters: n = 6, k = 2

-------------------------------
Start iterated traversal of graph ECZG
There are currently 1000 zeros and 36 commutations known.

Iteration 1:
Traverse Monomialgraph with n = 6 and k = 2
Number of starting zeros: 1000
Visited 1290 vertices of which
    1000 are starting zeros,
    113 were added through forward edges,
    6 were added through backward edges,
    67 were added through commutation edges and
    104 were added due to sum rule.
Found 290 new zeros and 1260 new commutations,
of which 30 were added by post processing.
The quantum automorphism group is commutative! Stopping iteration.
Finished iterated traversal after 1 iterations,
finding 1290 zeros and 1296 commutations.

The quantum automorphism group is commutative!
```

In [17] the authors determined which simple connected graphs with six vertices have quantum symmetries using either the disjoint automorphism criterion we have shown in Lemma 2.10 to determine their existence or a Gröbner basis approach to prove their non-existence. They showed, that of the 112 graphs 55 have quantum symmetries. We can compare this result to the output of our algorithm.

We use the tool suite `gtools` from the `nauty` package [18] to generate all simple, connected graphs with six vertices. In order to be as efficient as possible we first call `iteratetraverse!` with $k = 2$ and only then call it with $k = 3$ for all graphs that have not already been found to have no quantum symmetries. Doing so, we obtain the following results.

- For $k = 2$ the algorithm finds 8 graphs that do not have quantum symmetries.
- For $k = 3$ the algorithm finds 49 additional graphs that do not have quantum symmetries.
- For the remaining 55 graphs the algorithm could not determine that they do not have quantum symmetries.

If we compare this result with the results in [17], we see that our algorithm finds all connected, simple graphs with 6 vertices that do not have quantum symmetries. In fact, the cited work shows that all 55 graphs with quantum symmetries have disjoint automorphisms. Combining an automated check for the disjoint automorphism criterion with our algorithm would thus give the same characterization as the previous work. Interestingly however, is the speed at which these results were obtained.

By reusing and resetting the $(6, k)$-monomial graphs between runs we could minimize the most costly step which is the creation of the monomial graph. We have already seen before that the creation of the $(6, 2)$-monomial graph took around 90 milliseconds on the laptop and around 16 milliseconds on the server cluster. For the $(6, 3)$-monomial graph we have a run time of around 3.5 seconds on the laptop and a run time of around 1 second on the cluster. For the entire trial we got a combined run time of 193 seconds on the laptop. Unfortunately, we do not have access to the implementation from [17], but typically the run time for Gröbner basis computation far exceeds this.

Next, we also computed not just all connected graphs on six vertices but all 156 graphs on six vertices. Running the algorithm with $k = 3$, we found 68 graphs without quantum symmetries. The run times on both the laptop and cluster are very close with 286 and 250 seconds respectively.

Finally, we also computed all connected graphs on seven vertices and ran our implementation on them using the parameter $k = 3$. There are 853 connected graphs on seven vertices. In [17] the authors already showed that all graphs on seven vertices with either the trivial group or $\mathbb{Z}_2$ as automorphism group do not admit quantum symmetries. This accounts for 461 of the 853 graphs. With our implementation, we found a total of 507 graphs without quantum symmetries. The automorphism groups of the additional 46 graphs without quantum symmetries can be seen in Table 6.

Table 6: Connected graphs on seven vertices without quantum symmetries and their classical automorphism groups

Classical automorphism group	Order	Amount
$\{e\}$	1	144
$\mathbb{Z}_2$	2	317
$\mathbb{Z}_2 \times \mathbb{Z}_2$	4	10
S_3	6	31
D_{10}	10	1
D_{12}	12	2
D_{14}	14	2

In particular, this shows that all connected graphs with seven vertices and automorphism group of order 6 do not have quantum symmetries as has been conjectured in [17]. However, the general conjecture that a graph's automorphism group having order 6 (or being a specific automorphism group in general) might be mutually exclusive with a graph having quantum symmetry has been disproved by van Dobben de Bruyn, Roberson and Schmidt in 2025 [30]. Further, we used the Julia package Oscar [21], [11] to implement the disjoint automorphism criterion. Running it in conjunction with our algorithm we found that all remaining 346 graphs admit disjoint automorphisms and thus they have quantum symmetries. Our algorithm thus correctly identifies all connected graphs on seven vertices without quantum symmetries with the parameter $k = 3$. Using the parameter $k = 2$ we only managed to determine that 144 of the 853 graphs have no quantum symmetries.

The run times for running the algorithm for all 853 graphs on the laptop where 49 minutes and on the cluster 45 minutes. Throughout this section, we have repeatedly seen that running the implementation on the laptop is only marginally slower compared to running it on the server cluster. This is due to the fact that the algorithm at this point is not optimized for parallelization. The main advantage of running the code on the server cluster will be seen in the following section when examining the Petersen graph.

5.3 Petersen Graph

Finally, we revisit the Petersen graph P.

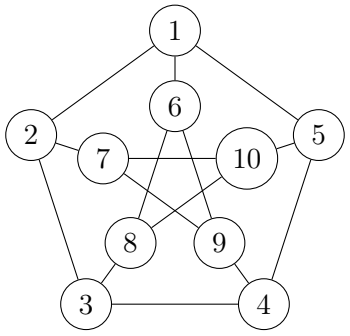


Figure 13: The Petersen graph P

As the Petersen graph has $n = 10$ vertices, Lemma 4.44 gives us the number of vertices in the $(10, k)$-monomial graph which grows exponentially with k. The $(10, 2)$-monomial graph has 10100 vertices, the $(10, 3)$-monomial graph has 1010100 vertices and the $(10, 4)$-monomial graph has 101010100 vertices. Since the entire monomial graph gets created by our algorithm this exponential growth poses a problem for the required memory. Indeed, on the Lenovo Thinkpad with 8 Gb of RAM we are only able to create the $(10, 2)$-monomial graph before running out of memory. Thus, running the algorithm on the server cluster is not necessary for the improved computational speed but for the increased available memory. Running the algorithm on the server cluster, we got the results shown in Table 7.

Table 7: Results of running the algorithm on the Petersen graph P for different k

k	Time for creating $(10, k)$-monomial graph	Time of running `iteratetraverse!`	Required memory	Output
2	5.7 milliseconds	1.5 milliseconds	8 Gb	`false`
3	37.6 seconds	88.9 seconds	40 Gb	`false`
4	1.99 hours	3.95 hours	400 Gb	`true`

Thus, we see that our algorithm with $k = 4$ is indeed capable of determining that the Petersen graph does not have quantum symmetry. To our knowledge this is the first time that this result has been found algorithmically. The required memory given in the previous table is an upper bound but is still an indication for the large cost of memory an increase in k can incur.

5.2 Remark. Even the output log of our algorithm is unwieldily large for $k = 4$: It is a text file with almost 10^8 lines and measures 3.2 Gigabytes. Examining this log file, we notice that the traversal algorithm is only called once and of the 10000 commutation relations 4500 were found in the post-processing step. This means that the commutation relations found in the post-processing step are required to determine that the quantum automorphism group is commutative but they are not needed for a second iteration of the traversal algorithm. In particular, this is interesting in the context of the known proof for the commutativity of the Petersen graph from [23] which we have reiterated in Section 3.3.

As mentioned in Remark 3.9, the proof of Proposition 3.8, which states that u_{ij} and u_{kl} commute for $i \not\sim k$ and $j \not\sim l$, relies on the commutation of generators u_{st} and u_{pq} where $s \sim p$ and $t \sim q$. This commutation in turn is the result of Proposition 3.7 which uses Lemma 2.5. However, our algorithm only applies Lemma 2.5 in the post-processing step which means that the commutations found this way are only available in the second iteration of the traversal step. Thus, the algorithm terminating after just one iteration of the traversal step and outputting that $C(G^+_{aut}(P))$ is commutative means that our algorithm has found an alternative proof for the commutativity of u_{ij} and u_{kl} with $i \not\sim k, j \not\sim l$. Examining the commutation relations found before and after the post-processing step shows that these commutation relations are also a result of Lemma 2.5. In order for it to be applicable in the post-processing step our algorithm must have found a way to show that certain monomials of the form $u_{ij}u_{kl}u_{st}$ are zero where we have $i \not\sim k \not\sim s$ and $j \not\sim l \not\sim t$.

In order to further examine this we modified the logs that are written by our algorithm to not only output the zeros that are found but to also add the propagation rule—or in other words the (hyper-)edge in the monomial graph—that has led to the zero being added.

5.3 Example. In the case of the Petersen graph, the part of the log showing the last zeros of length two and the first zeros of length three looks as follows:

```
VI(2,9998): [(10, 10), (10, 8)] due to
        Startzero with rule distance
VI(2,9999): [(10, 10), (10, 9)] due to
        Startzero with rule distance
```

```
length 3
VI(3,2): [(1, 1), (1, 1), (1, 2)] due to Backward edge
        from Vertexindex(4, 660002), pos=2, index=j
VI(3,3): [(1, 1), (1, 1), (1, 3)] due to Sum edge
        from long monomial Vertexindex(3, 10003), pos=1, index=j
```

We can interpret this in the following way.

- The monomial $u_{10\,10}u_{10\,8}$ has vertex index $(2, 9998)$. It has been found to be zero due to being a start zero. The criterion that determined it being a start zero was Lemma 2.8 which states that the monomial $u_{ij}u_{kl}$ is zero if the distance of vertices i and k is unequal to the distance of vertices j and l. The monomial $u_{10\,10}u_{10\,9}$ with vertex index $(2, 9999)$ has been found to be zero for the same reason.

- The monomial $u_{11}u_{11}u_{12}$ has vertex index $(3, 2)$. It has been found to be zero due to Propagation rule (Piii) which corresponds to a backward edge. The source of the backward edge is the set $B^{(j)}_{u,2}$ where $u = u_{11}u_{77}u_{11}u_{12}$ is the monomial with vertex index $VI(u) = (4, 660002)$. Recall that the set $B^{(j)}_{u,2}$ is defined as $\{u_{11}u_{7j}u_{11}u_{12} \mid i \in \{1, \ldots, 10\}\}$. Thus, we know that the monomial $u_{11}u_{11}u_{12}$ has been found to be zero due to all ten monomial in $B^{(j)}_{u,2}$ having been found to be zero. In particular, we also can deduce that the monomial $u_{11}u_{77}u_{11}u_{12}$ was the last monomial in $B^{(j)}_{u,2}$ that was found to be zero in this instance of running the algorithm.

- The monomial $u_{11}u_{11}u_{13}$ has vertex index $(3, 3)$. It has been found to be zero duo to Propagation rule (Pvi) which corresponds to a sum edge. The source of the sum edge is the set $S^{(j)}_{u,1}$ where $u = u_{12}u_{11}u_{13}$ is the monomial with vertex index $VI(u) = (3, 10003)$. Recall that the set $S^{(j)}_{u,1}$ is defined as $\{u_{1j}u_{11}u_{13} \mid j \in \{1, \ldots, 10\}, j \neq 2\} \cup \{u_{11}u_{13}\}$. Thus, we know that the monomial $u_{11}u_{11}u_{13}$ has been found to be equal to zero due to all ten monomials in $S^{(j)}_{u,1}$ having been found to be zero. In particular, we also know that u is one of the long monomials in $S^{(j)}_{u,1}$—so it is not the short monomial $u_{11}u_{13}$—and we can deduce that the monomial $u_{12}u_{11}u_{13}$ was the last monomial in $S^{(j)}_{u,1}$ that was found to be zero in this instance of applying the algorithm.

While not pictured here, we have similar outputs for forward edges and commutation edges.

Next, we created an additional interpretation program that, given the output log and a particular monomial one is interested in, examines the log and recursively finds all of the monomials that needed to be zero in order for the input monomial to be zero. This program either outputs a text file or can create an image of a graph showcasing the propagation of zeros.

5.4 Example. Take the monomial $u = u_{10\,10}u_{19}u_{45}$ in $C(G^{+}_{aut}(P))$. In the Petersen graph P we have that $10 \not\sim 1 \not\sim 4$ and $10 \not\sim 9 \not\sim 5$. Our algorithm states that $u = 0$. Using the interpretation program we could obtain a proof that we show in Table 8.

Table 8: Proof for $u = u_{10\,10}u_{19}u_{45} = 0$ as found by our algorithm. The rightmost column contains the reasoning why the monomials on the left are zero. In the case of a backward or sum edge, the rightmost column gives the source of the hyperedge. In the case of a forward edge it simply states this. If the monomial is a start zero the rightmost column gives the concrete application of one of the rules used to determine this. On the left side of the table, all of the monomials below and indented one column to the right relative to a certain monomial are the monomials that have to be zero in order for this monomial to be zero.

$u_{10\,10}u_{19}u_{45}$						$B^{(j)}_{u_{10\,10}u_{19}u_{54}u_{45},3}$
	$u_{10\,10}u_{19}u_{54}u_{45}$					$S^{(i)}_{u_{10\,10}u_{19}u_{54}u_{45},2}$
		$u_{10\,10}u_{54}u_{45}$				Forward
			$u_{10\,10}u_{54} = 0$			since $10 \sim 5, 10 \not\sim 4$
		$u_{10\,10}u_{29}u_{54}u_{45}$				Forward
			$u_{29}u_{54}u_{45}$			Forward
				$u_{29}u_{54} = 0$		since $2 \not\sim 5, 9 \sim 4$
		$u_{10\,10}u_{39}u_{54}u_{45}$				Forward
			$u_{39}u_{54}u_{45}$			Forward
				$u_{39}u_{54} = 0$		since $3 \not\sim 5, 9 \sim 4$
		$u_{10\,10}u_{49}u_{54}u_{45}$				Forward
			$u_{49}u_{54}u_{45}$			$S^{(j)}_{u_{49}u_{54}u_{45},2}$
				$u_{49}u_{45} = 0$		since $9 \neq 5$
				$u_{49}u_{51}u_{45}$		Forward
					$u_{49}u_{51} = 0$	since $4 \sim 5, 9 \not\sim 1$
				$u_{49}u_{52}u_{45}$		Forward
					$u_{52}u_{45} = 0$	since $5 \sim 4, 2 \not\sim 5$
				$u_{49}u_{53}u_{45}$		Forward
					$u_{53}u_{45} = 0$	since $5 \sim 4, 3 \not\sim 5$
				$u_{49}u_{55}u_{45}$		Forward
					$u_{55}u_{45} = 0$	since $5 \neq 4$
				$u_{49}u_{56}u_{45}$		Forward
					$u_{56}u_{45} = 0$	since $5 \sim 4, 6 \not\sim 5$
				$u_{49}u_{57}u_{45}$		Forward
					$u_{57}u_{45} = 0$	since $5 \sim 4, 7 \not\sim 5$
				$u_{49}u_{58}u_{45}$		Forward
					$u_{58}u_{45} = 0$	since $5 \sim 4, 8 \not\sim 5$
				$u_{49}u_{59}u_{45}$		Forward
					$u_{59}u_{45} = 0$	since $5 \sim 4, 9 \not\sim 5$
				$u_{49}u_{5\,10}u_{45}$		Forward
					$u_{49}u_{5\,10} = 0$	since $4 \sim 5, 9 \not\sim 10$
		$u_{10\,10}u_{59}u_{54}u_{45}$				Forward
			$u_{59}u_{54}u_{45}$			Forward
				$u_{59}u_{54} = 0$		since $9 \neq 4$
		$u_{10\,10}u_{69}u_{54}u_{45}$				Forward
			$u_{69}u_{54}u_{45}$			Forward
				$u_{69}u_{54} = 0$		since $6 \not\sim 5, 9 \sim 4$
		$u_{10\,10}u_{79}u_{54}u_{45}$				Forward

			$u_{79}u_{54}u_{45}$		Forward
				$u_{79}u_{54}=0$	since $7 \not\sim 5, 9 \sim 4$
		$u_{10\,10}u_{89}u_{54}u_{45}$			Forward
			$u_{89}u_{54}u_{45}$		Forward
				$u_{89}u_{54}=0$	since $8 \not\sim 5, 9 \sim 4$
		$u_{10\,10}u_{99}u_{54}u_{45}$			Forward
			$u_{99}u_{54}u_{45}$		Forward
				$u_{99}u_{54}=0$	since $9 \not\sim 5, 9 \sim 4$
		$u_{10\,10}u_{10\,9}u_{54}u_{45}$			Forward
			$u_{10\,10}u_{10\,9}u_{54}$		Forward
				$u_{10\,10}u_{10\,9}=0$	since $10 \neq 9$
	$u_{10\,10}u_{19}u_{55}u_{45}$				Forward
		$u_{19}u_{55}u_{45}$			Forward
			$u_{55}u_{45}=0$		since $5 \neq 4$
	$u_{10\,10}u_{19}u_{56}u_{45}$				Forward
		$u_{19}u_{56}u_{45}$			Forward
			$u_{56}u_{45}=0$		since $5 \sim 4, 6 \not\sim 5$
	$u_{10\,10}u_{19}u_{57}u_{45}$				Forward
		$u_{19}u_{57}u_{45}$			Forward
			$u_{57}u_{45}=0$		since $5 \sim 4, 7 \not\sim 5$
	$u_{10\,10}u_{19}u_{58}u_{45}$				Forward
		$u_{19}u_{58}u_{45}$			Forward
			$u_{58}u_{45}=0$		since $5 \sim 4, 8 \not\sim 5$
	$u_{10\,10}u_{19}u_{59}u_{45}$				Forward
		$u_{19}u_{59}u_{45}$			Forward
			$u_{59}u_{45}=0$		since $5 \sim 4, 9 \not\sim 5$
	$u_{10\,10}u_{19}u_{5\,10}u_{45}$				Forward
		$u_{19}u_{5\,10}u_{45}$			Forward
			$u_{19}u_{5\,10}=0$		since $1 \sim 5, 9 \not\sim 10$
	$u_{10\,10}u_{19}u_{51}u_{45}$				Forward
		$u_{19}u_{51}u_{45}$			Forward
			$u_{19}u_{51}=0$		since $1 \sim 5, 9 \not\sim 1$
	$u_{10\,10}u_{19}u_{52}u_{45}$				Forward
		$u_{19}u_{52}u_{45}$			Forward
			$u_{52}u_{45}=0$		since $5 \sim 4, 2 \not\sim 5$
	$u_{10\,10}u_{19}u_{53}u_{45}$				Forward
		$u_{19}u_{53}u_{45}$			Forward
			$u_{53}u_{45}=0$		since $5 \sim 4, 3 \not\sim 5$

Rather than having the data presented in a table we can also show it as a tree as follows.

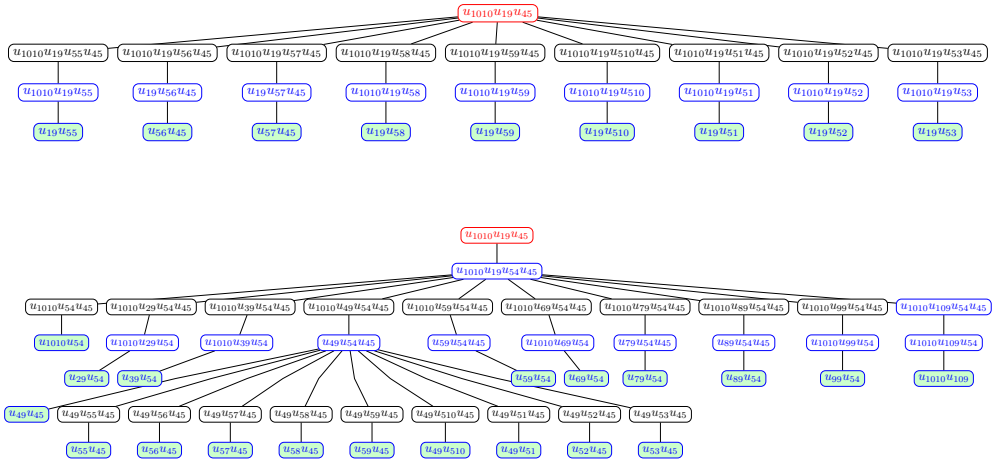


Figure 14: A graph showcasing the propagation of zeros from start zeros (green) to the monomial $u = u_{10\,10}u_{19}u_{45}$ (red). Marked in blue are the monomials that triggered the propagation (meaning they were the last monomial of the source of the corresponding (hyper-)edge to be visited). For reasons of legibility we have split the tree in two parts and the red monomials should be assumed to be combined.

Figure 14 corresponds to a partial spanning tree of the explored monomial graph rooted at $u = u_{10\,10}u_{19}u_{45}$. We call this partial spanning tree the *propagation tree of* u. As is clearly visible, for every backward or sum edge explored in the traversal algorithm we get $n = 10$ additional monomials in the propagation tree of u. Therefore, the tree structure leads to an exponential growth leading to the propagation trees soon also growing too large to display.

In fact, when simply using a stack in the algorithm as we have proposed so far in Section 4 the start zeros remain on the stack until the very end. This is rather unhelpful for following the propagation of zeros until one only finds start zeros. However, as we have shown in the proof of Theorem 4.49, the order in which one visits the monomials in the traversal step does not change which monomials will have been visited when the algorithm terminates. Thus, we have exchanged the stack with a list of stacks where each of the new stacks only takes monomials of a certain length. When calling `pop` on this list of stacks, we find the stack corresponding to the shortest monomials that is not empty and call `pop` on this stack. In this way we now prioritize shorter monomials to be visited earlier. As currently all our start zeros have length 1 or 2 they are now visited quicker leading to less deep propagation trees. Another approach one could take would be exchanging the stack with a *queue* which is a data structure for a dynamic set governed by a *first-in, first out* or *FIFO* policy. This would correspond to a traversal in the spirit of breadth-first search rather than depth-first search.

The monomial $u_{10\,10}u_{19}u_{45}$ was the first monomial of the form $u_{ij}u_{kl}u_{rs}$ with $i \not\sim k \not\sim r$ and $j \not\sim l \not\sim s$ the algorithm encountered and yet the propagation tree has depth six and has 76 vertices. This is the reason why we have chosen to display it at this point as opposed to another monomial. For example, the monomial $u_{11}u_{34}u_{12}$ was also found to be zero by our algorithm. This is an important intermediate result

in order to show that all monomials $u_{11}u_{34}u_{1j}$ for $j \in \{2, \ldots, 10\}$ are equal to zero so that we may conclude that $u_{11}u_{34} = u_{11}u_{34}u_{11}$ and thus by Lemma 2.5 the generators u_{11} and u_{34} commute. However, when attempting to create the propagation tree of $u_{11}u_{34}u_{12}$, we reached a depth of 98 before stopping to probe further. This computation was additionally made more complicated by the fact that even using the server cluster to compute the propagation tree we were limited to a depth of 6 per computation due to memory constraints.

6 Outlook

The primary and immediate continuation of the research on and with this algorithm lies within applying it to new classes of graphs whose quantum symmetries have not been found to exist or not exist yet. This is particularly interesting for highly regular and connected graphs where the disjoint automorphism criterion is less likely to immediately yield quantum symmetries. The fact that our algorithm was able to compute that the Petersen graph does not have quantum symmetries is an encouraging sign that it might also give results for other highly regular graphs.

Another avenue for continued research lies in improving the algorithm itself further. Here, some areas that come to mind are the following.

- Add additional criteria for finding start zeros and commutations in the post-processing step. As mentioned in Section 4.3 the criteria shown in Lemmas 3.2.2 and 3.2.3 in [12] good candidates for this.
- Improve the space efficiency of the algorithm. Even if the number of vertices in the (n, k)-monomial graph cannot be reduced it might be fruitful to improve the implementation. Since the number of vertices and edges grows exponentially with k even small improvements can pay off quickly here.
- Implement heuristics that determine the ordering in which the monomials are visited. If one wants to traverse the entire monomial graph this will not make a difference in the run time, but if one is interested in a specific monomial one might try to steer the algorithm towards it in the hopes of not having to blindly explore the entire monomial graph. Another use for this could be an improvement in the interpretation of the results. If "interesting" monomials are visited earlier in the traversal creating the propagation tree might be more feasible and give insights quicker.
- In a similar fashion to the previous point, it could be interesting to further examine the output the algorithm has given for the Petersen graph. It might be possible to generalize a proof in a more "mathematical" language from the computer-based proof that simply gives a list of zeros.
- Finally, in order to speed up the computation significantly the algorithm should be parallelized as much as possible.

References

1. Accardi, L. Topics in quantum probability. *Physics Reports* **77,** 169–192. ISSN: 0370-1573. https://www.sciencedirect.com/science/article/pii/0370157381900703 (1981).
2. Banica, T. Quantum automorphism groups of homogeneous graphs. *Journal of Functional Analysis* **224,** 243–280. ISSN: 0022-1236. https://www.sciencedirect.com/science/article/pii/S0022123604004070 (2005).
3. Banica, T. & Bichon, J. *Quantum automorphism groups of vertex-transitive graphs of order* ≤ 11 2006. arXiv: math/0601758 [math.QA]. https://arxiv.org/abs/math/0601758.

4. Bezanson, J., Edelman, A., Karpinski, S. & Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM Review* **59,** 65–98. https://epubs.siam.org/doi/10.1137/141000671 (2017).

5. Bichon, J. Quantum Automorphism Groups of Finite Graphs. *Proceedings of the American Mathematical Society* **131,** 665–673. ISSN: 00029939, 10886826. http://www.jstor.org/stable/1194467 (2025) (2003).

6. Blackadar, B. *Operator Algebras: Theory of C*-Algebras and von Neumann Algebras* (Springer Berlin Heidelberg, 2006).

7. Chassaniol, A. *Study of quantum symmetries for vertex-transitive graphs using intertwiner spaces* 2019. arXiv: 1904.00455 [math.QA]. https://arxiv.org/abs/1904.00455.

8. Cherkassky, B. V., Goldberg, A. V. & Radzik, T. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming* **73,** 129–174 (May 1996).

9. Connes, A. Non-commutative differential geometry. English. *Publ. Math., Inst. Hautes Étud. Sci.* **62,** 41–144. ISSN: 0073-8301. https://eudml.org/doc/104008 (1985).

10. Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. *Introduction to Algorithms, Third Edition* 3rd. ISBN: 0262033844 (The MIT Press, 2009).

11. *The Computer Algebra System OSCAR: Algorithms and Examples* 1st ed. (eds Decker, W., Eder, C., Fieker, C., Horn, M. & Joswig, M.) https://link.springer.com/book/9783031621260 (Springer, 2025).

12. Fulton, M. *The quantum automorphism group and undirected trees* PhD (Virginia, 2006).

13. Gel'fand, I. & Naĭmark, M. On the imbedding of normed rings into the ring of operators in Hilbert space. eng. *Matematiceskij sbornik* **54,** 197–217. http://eudml.org/doc/65219 (1943).

14. Holton, D. A. & Sheehan, J. *The Petersen Graph* (Cambridge University Press, 1993).

15. Jung, S. *Easy quantum groups: linear independencies, models and partition quantum spaces* PhD (Universität des Saarlandes, 2019).

16. Knuth, D. E. *The art of computer programming, volume 4, fascicle 0* (Addison-Wesley Educational, Boston, MA, Apr. 2008).

17. Levandovskyy, V. *et al. Existence of Quantum Symmetries for Graphs on Up to Seven Vertices: A Computer based Approach* in *Proceedings of the 2022 International Symposium on Symbolic and Algebraic Computation* (Association for Computing Machinery, Villeneuve-d'Ascq, France, 2022), 311–318. ISBN: 9781450386883. https://doi.org/10.1145/3476446.3535481.

18. McKay, B. D. & Piperno, A. Practical graph isomorphism, II. *Journal of Symbolic Computation* **60,** 94–112. ISSN: 0747-7171. https://www.sciencedirect.com/science/article/pii/S0747717113001193 (2014).

19. Mora, T. An introduction to commutative and noncommutative Gröbner bases. *Theoretical Computer Science* **134,** 131–173. ISSN: 0304-3975. https://www.sciencedirect.com/science/article/pii/0304397594902836 (1994).

20. Murray, F. J. & v. Neumann, J. On Rings of Operators. *Annals of Mathematics* **37,** 116–229. ISSN: 0003486X, 19398980. http://www.jstor.org/stable/1968693 (2026) (1936).

21. *OSCAR – Open Source Computer Algebra Research system, Version 1.6.0* The OSCAR Team, 2025. https://www.oscar-system.org.

22. Schanz, J. *Quantum Symmetries of Vertex-Transitive Graphs on 12 Vertices* 2024. arXiv: 2404.14976 [math.QA]. https://arxiv.org/abs/2404.14976.

23. Schmidt, S. The Petersen graph has no quantum symmetry. *Bulletin of the London Mathematical Society* **50,** 395–400. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/blms.12154. https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/blms.12154 (2018).

24. Schmidt, S. Quantum automorphisms of folded cube graphs. en. *Annales de l'Institut Fourier* **70,** 949–970. https://aif.centre-mersenne.org/articles/10.5802/aif.3328/ (2020).

25. Schmidt, S. & Weber, M. Quantum Symmetries of Graph C*-algebras. *Canadian Mathematical Bulletin* **61,** 848–864. ISSN: 1496-4287. http://dx.doi.org/10.4153/CMB-2017-075-4 (Nov. 2018).

26. Shimbel, A. Structural parameters of Communication Networks. *The Bulletin of Mathematical Biophysics* **15,** 501–507 (Dec. 1953).

27. Speicher, R. & Weber, M. Quantum Groups with Partial Commutation Relations. *Indiana University Mathematics Journal* **68,** pp. 1849–1883. ISSN: 00222518, 19435258. https://www.jstor.org/stable/26958357 (2025) (2019).

28. Taylor, J. L. A general framework for a multi-operator functional calculus. *Advances in Mathematics* **9,** 183–252. ISSN: 0001-8708. https://www.sciencedirect.com/science/article/pii/0001870872900175 (1972).

29. Van Dobben de Bruyn, J., Kar, P. N., Roberson, D. E., Schmidt, S. & Zeman, P. Quantum automorphism groups of trees. *J. Noncommut. Geom.* (Jan. 2025).

30. Van Dobben de Bruyn, J., Roberson, D. E. & Schmidt, S. Asymmetric graphs with quantum symmetry. *Proceedings of the London Mathematical Society* **131,** e70098. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms.70098. https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms.70098 (2025).

31. Voiculescu, D. *Symmetries of some reduced free product C*-algebras* in *Operator Algebras and their Connections with Topology and Ergodic Theory* (eds Araki, H., Moore, C. C., Stratila, Ş.-V. & Voiculescu, D.-V.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 1985), 556–588. ISBN: 978-3-540-39514-0.

32. Wang, S. Quantum Symmetry Groups of Finite Spaces. *Communications in Mathematical Physics* **195,** 195–211. ISSN: 1432-0916. http://dx.doi.org/10.1007/s002200050385 (July 1998).

33. Weber, M., Li, X. & Voigt, C. *lecture notes on ISEM24 C*-algebras and dynamics* https://www.math.uni-sb.de/ag/speicher/weber/ISem24/ISem24LectureNotes.pdf (2025).

34. Woronowicz, S. L. Compact matrix pseudogroups. *Communications in Mathematical Physics* **111,** 613–665 (1987).

35. Woronowicz, S. *Compact quantum groups. Symétries quantiques (Les Houches, 1995), 845–884* 1998.