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

Bachelor Thesis

Design and Implementation of Efficient Algorithms for Operations on Partitions of Sets

submitted by
Sebastian Volz

under supervision of
Nicolas Faroß
Prof. Dr. Moritz Weber

submitted

Reviewers
Prof. Dr. Moritz Weber
Prof. Dr. Markus Bläser

Erklärung
Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik/Mathematik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science/Mathmatics Department.

Saarbrücken

Abstract

Easy quantum groups are associated with categories of partitions. These categories were initially defined by Teodor Banica and Roland Speicher in 2009 and their classification was completed by Sven Raum and Moritz Weber in 2016. We design and implement efficient algorithms and data structures for finding and verifying different properties related to partitions and their categories.

More precisely, we propose algorithms for operations on partitions, constructing categories, and linear combinations of partitions. As an extension, we introduce a tracing algorithm for constructing categories, as well as algorithms for generalizations of classical partitions, such as colored- and spatial partitions, which are not yet fully classified. Through tracing, we explore specific properties related to classical, colored, and spatial partitions.

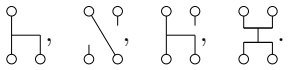
In addition, after completing the design process, we present a concrete implementation.

Contents

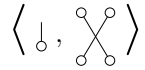
1 Introduction

In this thesis, we study algorithms and data structures in the domain of partitions of sets. More precisely, we consider operations and categories of partitions. Categories of partitions are related to easy quantum groups, as defined in [BS09] and their classification was completed in [SR16]. Furthermore, partitions can be generalized by colored partitions and spatial partitions. The classification of colored partitions is almost completed in [TW15b] and [MW19]. In contrast, spatial partitions, introduced in [CW16] and further investigated in [Far22], are still relatively unknown. However, in the context of this thesis, we focus on partition problems and not the relation to easy quantum groups. Overall, we operate at the interface of mathematics and computer science, combining these domains to obtain an interdisciplinary perspective.

A partition $p \in P(k,l)$ is a partition of a set $S = \{1, ..., k+l\}$ into disjoint subsets. These partitions can be represented by k upper and l lower points, where each point is an element of S. Points in the same subset are connected and form a resulting block. This can be visualized as follows:



Furthermore, we can apply operations on them, namely the tensor product, the involution and the composition. A category $\mathcal{C}$ of partitions is a set of partitions, which is closed under these operations and contains the base partitions  $\in P(1,1)$ and  $\in P(0,2)$. A category $\mathcal{C}$ can also be defined by a set of generating partitions. For example, the category $P^{1,2}$, representing all partitions with a block size of one or two (with the associated easy quantum group B_n) can be depicted by the following generating partitions:



This category contains every partition that can be constructed using both the generating partitions and the base partitions, considering the operations on partitions.
After designing algorithms for the operations on partitions, we can find an algorithm, which produces subsets of categories, given a partition size constraint.

Theorem 1.1 (Section 3.2, Theorem 3.14., 3.15., 3.16.)**.** *The time complexity of the tensor product is $O(n)$, of the involution $O(1)$ and of the composition $O(n \log(n))$.*

Given a set of generating partitions representing a category $\mathcal{C}$, and a number $n \in \mathbb{N}$, the algorithm for constructing categories has the goal to provide $P(n) \cap \mathcal{C}$ as an output, where $P(n)$ is the set of all partitions of size n.

Theorem 1.2. *Let G be a set of generating partitions of a category $\mathcal{C}$ and $n \in \mathbb{N}$. Then the algorithm for constructing categories in Section 4 computes a set $S \subseteq P(n) \cap \mathcal{C}$.*

The general application can be to retrieve properties of specific categories using the data produced by the construction algorithm.

Additional research in the scope of this thesis involves the development of extension algorithms for generalizations of partitions, specifically of colored and spatial partitions. Additionally, we aim to devise algorithms for handling linear combinations of partitions.

Extending the constructing categories algorithms for classical, colored and spatial partitions has the purpose of tracing every necessary partition in the constructed category. As a result, we get insights into the construction process of categories, potentially revealing noteworthy properties.

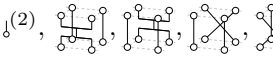
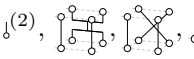
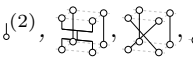
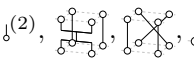
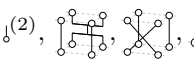
An example property for classical partitions, found during the construction process, is explained Example 4.12.1.

Theorem 1.3 (Example 4.19.1.)**.** *It is possible to construct* $\in P(4)$ *with* $\langle$,  $\rangle$*, without using partitions of size > 4.*

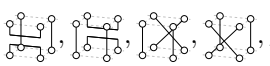
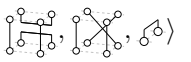
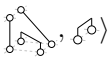
For spatial partitions, we were able to get the properties described in Example 4.12.2., Theorem 4.13. and Theorem 4.14.

Theorem 1.4 (Example 4.19.2.)**.** *The sets* $\langle$  , ,  ,  $\rangle$ *and* $\langle$, , $\rangle$ *generate the same category.*

Theorem 1.5 (Theorem 4.20.). *The following sets all generate $P^{(2)}$:*

- $\langle$  $\rangle$
- $\langle$  $\rangle$
- $\langle$  $\rangle$
- $\langle$  $\rangle$
- $\langle$  $\rangle$

Theorem 1.6 (Theorem 4.21.). *The following sets all generate $P_2^{(2)}$:*

- $\langle$  $\rangle$
- $\langle$  $\rangle$
- $\langle$  $\rangle$
- $\langle$  $\rangle$
- $\langle$  $\rangle$
- $\langle$  $\rangle$

The concrete implementations resulting from this thesis are available in [Vol23].

Overview of the thesis

We begin by laying out the mathematical foundations in Section 2.1, by introducing the core concepts of partitions, encompassing their definitions, operations, and categorizations. This is then followed by the computer science background in Section 2.2. The computer science background includes notation to analyse algorithms with respect to their complexity. Furthermore, we present some optimization techniques and data structures used in the algorithms.

In order to get a solid foundation for algorithms in the domain of categories and linear combinations of partitions, we start by designing efficient algorithms for operations on partitions in Section 3.

Afterwards, we propose algorithms for constructing categories in Section 4, including the algorithm for classical partitions and the algorithms for their generalizations, namely colored and spatial partitions. As there are categories for which understanding the partition construction process can be meaningful, we also present an algorithm capable of tracing each partition in terms of its construction process.

For handling linear combinations of partitions, we design algorithms for adding and multiplying polynomial partitions terms, followed by an application example in Section 5.

Finally, we discuss the limitations and prospects for future work within the scope of this thesis in Section 6.

2 Background

In this section, we start with some preliminaries regarding both the mathematical background, as well as the computer science background. First, we define some aspects in the area of partitions, followed by computer science oriented techniques, such as Big-O Notation, path compression and data structures.

2.1 Mathematical Background

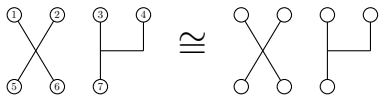
At first, we introduce some concepts from combinatorics, namely partitions.

2.1.1 Partitions

In the following section, we introduce partitions. As will be explained in Section 2.1.2 and Section 2.1.3, we can operate on them as well as classify their so called categories. All definitions proposed in this section can be found in [BCS09], [BBC07], [Sta99] and [TW15a].

Definition 2.1 (Partition)**.** Let $k, l \in \mathbb{N}$. A *partition* p is given by k lower and l upper points. It partitions a set $\{1, ..., k, k+1, ..., k+l\}$ into disjoint subsets, called blocks. Partitions can also be visualized. Given $p \in P(k, l) \subseteq P(n)$ with $n = k + l$, we first draw a row of k upper points followed by l lower points. After numbering these points from 1 to $k + l$ (all elements from the set $\{1, ..., k, ..., k + l\}$) we connect them according to the blocks defined in p.

Example 2.2. Given $p \in P(4, 3)$, where $s = \{1, 2, 3, 4, 5, 6, 7\}$ is the set we partition into $p = \{\{1, 6\}, \{2, 5\}, \{3, 4, 7\}\}$. We can visualize p as follows:

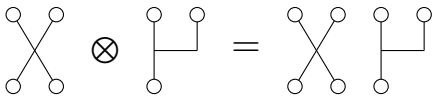
$\{\{1,6\},\{2,5\},$
$\{3,4,7\}\}$ $\cong$  $\cong$

2.1.2 Operations on Partitions

In order to operate on partitions, as defined in [BCS09], there are three main operations, namely the *tensor product*, the *involution* (also known as *horizontal reflection*) and the *composition*. From these three operations, combined with the base partitions, we also can derive the *rotation* and the *vertical reflection*.

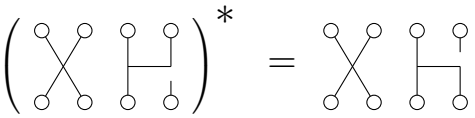
Definition 2.3 (Tensor Product)**.** Let $p \in P(k_1, l_1)$ and $q \in P(k_2, l_2)$. Then the *tensor product* can be obtained by concatenating p and q in form of $p \otimes q \in P(k_1 + k_2, l_1 + l_2)$.

Example 2.4 (Tensor Product)**.**



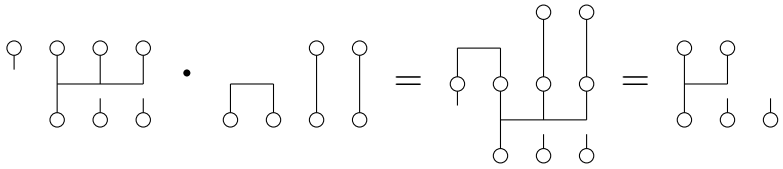
Definition 2.5 (Involution)**.** Let $p \in P(k, l)$. Then the unary operation *involution* $p^* \in P(l, k)$ can be obtained by swapping the upper and lower points.

Example 2.6 (Involution)**.**



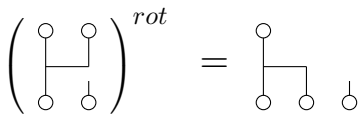
Definition 2.7 (Composition)**.** Let $p \in P(k_1, l_1)$ and $q \in P(k_2, l_2)$ and $k_1 = l_2$. Then the *composition* $pq \in P(k_2, l_1)$ can be obtained by connecting p and q by writing q above p and joining each lower point of q with the respective upper point of p so that we connect every point in l_2 to a point in k_1 with respect to the order. After this process, any intermediate points and loops are removed, such that only the upper points of q and the lower points of p are left.

Example 2.8 (Composition)**.**



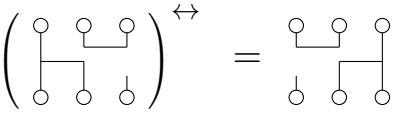
Definition 2.9 (Rotation)**.** Let $p \in P(k,l)$ and $k > 0$. Then the unary operator *rotation* $p^{rot} \in P(k-1, l+1)$ can be obtained by shifting the very left upper point to the very left of the lower points. Note that all points belong to the same blocks as before. The result of this operation is called *rotated version*. Analogical, we can also rotate from the lower points to the upper points and from the left side or from the right side of the partition p.

Example 2.10 (Rotation)**.** The following figure shows an example of a *left-top rotation*:



Definition 2.11. Let $p \in P(k,l)$. Then the unary operator *vertical reflection* $p^{\leftrightarrow} \in P(k,l)$ can be obtained by reversing the upper points and also reversing the lower points in p.

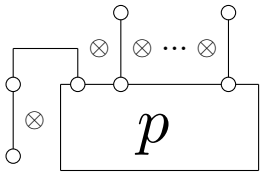
Example 2.12 (Vertical Reflection)**.**



Definition 2.13 (Category operation)**.** An operation is called *category operation* if and only if it is part of the three main operations (*tensor product*, *involution*, *composition*) or it can be constructed with a combination of the main operations and *base partitions* $\in P(1,1)$ and $\sqcap \in P(0,2)$. Because we can construct the *rotation* and the *vertical reflection* with the rules defined above, they both are also *category operations*.

Lemma 2.14. *The rotation is a category operation.*

Proof. Let $p \in P(n)$ be a partition with $n \in \mathbb{N}$. Then we can produce the left-top rotation with $(\otimes p) \cdot (\sqcap \otimes \otimes ... \otimes)$, which can be visualized as follows:



As a result we can see that the rotation can be produced with the operations tensor product and composition and the base partitions $\in P(1,1)$ and $\sqcap \in P(0,2)$. □

Lemma 2.15. *The vertical reflection is a category operation.*

Proof. Let $p \in P(k,l)$ be a partition with $k, l \in \mathbb{N}$. Then we can produce the vertical reflection by k top-left rotations, followed by l bottom-right rotations and one involution operation.

$$p^{\leftrightarrow} \Leftrightarrow \left(p^{\mathrm{rot}\,(\ldots)^{\mathrm{rot}}}\right)^{*}$$

□

2.1.3 Categories of Partitions

Definition 2.16 (Categories of Partitions)**.** A *category of partitions* is defined by a subset $\mathcal{C} \subseteq P$, where $P := \bigcup_{k,l} P(k,l)$ is the set of all partitions, such that C contains the base partitions $\{\{\}\} \in P(1,1)$ and $\{\{\},\{\}\} \in P(0,2)$ and is closed under the *category operations*. This results in the property $C(k,l) := C \cap P(k,l)$. We can also specify categories of partitions using a set S of so called *generator partitions*, where a category is closed under the set of generator partitions $\langle S \rangle$ with the base partitions.

Definition 2.17 (Non-crossing Partitions)**.** Let p be a partition. Then we say p is a *non-crossing partition*, if for every pair of blocks (b_1, b_2) in p all upper/lower points of b_2 have exclusively smaller or exclusively higher indices than every index of the upper/lower points in b_1.

Definition 2.18 (Balanced Partitions)**.** Let p be a partition. Then we say p is a *balanced partition*, if every block, that is not a singleton, has the same amount of points with even and odd indices. Additionally, within a balanced partition, the amount of singletons with odd indices matches that of even indices.

Example 2.19 (Categories of Partitions)**.** The following sets are examples of categories of partitions which are relevant for the subject of this thesis.

Below, we can see some example categories, followed by the generator partitions which can be used to construct those.

1. The set P of all partitions $= \langle$  $\rangle$
2. The set NC of all non-crossing partitions $= \langle$  $\rangle$
3. The set P_2 of all pair partitions (every block has a size of two), $= \langle$  $\rangle$
4. The set $P_{even\ BS}$ of all partitions with even block size (every block has an even size), $=$ $\langle$  $\rangle$
5. The set $P^{1,2}$ of all partitions with block size one or two, $= \langle$  $\rangle$
6. The set P_{even} of all partitions with an even size, $= \langle$  $\rangle$
7. The set $P^{1,2}_{even}$ of all partitions with an even size and block size one or two, $= \langle$  $\rangle$
8. The set NC_2 of all non-crossing partitions with block size two, $= \langle\ \rangle$ (i. e. can be produced by only the base partitions)
9. The set $NC_{even\ BS}$ of all non-crossing partitions with even block size, $= \langle$  $\rangle$
10. The set $NC^{1,2}$ of all non-crossing partitions of block size one or two, $= \langle$  $\rangle$
11. The set NC_{even} of all non-crossing partitions with even size, $= \langle$  $\rangle$
12. The set $NC^{1,2}_{even}$ of all non-crossing partitions of even size and with block size one and two, $= \langle$ $\rangle$
13. The set B of all balanced partitions, $= \langle$  $\rangle$
14. The set B_2 of all balanced partitions of block size two, $= \langle$  $\rangle$
15. The set NCB_{even} of all non-crossing partitions of even size, which are balanced pairs and have an even number of singletons (blocks of size one), $= \langle$  $\rangle$
16. The set $B^{1,2}_{even}$ of all balanced partitions of block size one and two with even number of singletons, $= \langle$  $\rangle$

Furthermore, the intersection of two categories forms a new category: Let C_1 and C_2 be categories, then $C_1 \cap C_2$ is also a category.
For example,

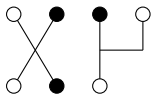
$$NC_2 := NC \cap P_2$$

.
An extensive overview of categories and the respective quantum groups can be found in Appendix A.

2.1.4 Generalization: Colored Partitions

In this section, we briefly define colored partitions in order to introduce a construction algorithm for their categories in Section 4. The classification of colored partitions is almost completed in [TW15b] and [MW19]. In this and the following sections, we use the definitions from [TW15b] and [MW19].

Definition 2.20 (Colored Partitions)**.** *Colored partitions* are a generalization of partitions. In order to get a colored partition, we have to assign every point in a partition either black or white. As a result, we get a binary coloring, for example:



Remark 2.21. For colored partitions, the operations involution and tensor product remain the same.

Let p, q be partitions. If we want to apply the composition pq, we have to ensure, that the coloring of the upper points of p and the lower points of q is the same. The rest of the operation does not differ from Definition 2.7.

Furthermore, the base partitions of colored partitions are  ,  $\in P(1,1)$ and  ,  $\in P(0,2)$.

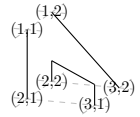
2.1.5 Generalization: Spatial Partitions

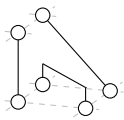
In this section we define spatial partitions, in order to introduce a construction algorithm for their categories in Section 4. Spatial partitions, introduced in [CW16], are still relatively unexplored. In the following section, we use the definitions from [CW16].

Definition 2.22 (Spatial Partitions)**.** Let $k, l \in \mathbb{N}$ and $m \in \mathbb{N}$. A *spatial partition* is a partition of the set $\{1, ..., k, k+1, ..., k+l\} \times \{1, ..., m\}$ into disjoint subsets. Each subset forms a called block. The first component $\{1, ..., k, k+1, ..., k+l\}$ is divided into k upper and l lower points, where the second component {1,..., m} consists of m levels. We say $P^{(m)}(k,l) \subseteq P^{(m)}(n)$ is the set of all spatial partitions with k upper points, l lower points, m levels and $k+l = n \in \mathbb{N}$. For simplicity, we write $P^{(m)}$ instead of $P^{(m)}(k,l)$ or $P^{(m)}(n)$. Spatial partitions are a generalization of classical partitions.

Example 2.23. The following example shows a visualization of a spatial partition $p \in P^{(2)}(1,2)$. In contrast to the classical partitions, we add dimensions to the points.

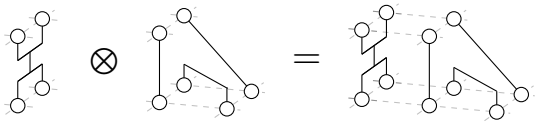
$$\{\{(1,1),(2,1)\},\ \{(1,2),(3,2)\},\ \{(2,2),(3,1)\}\} \cong$$



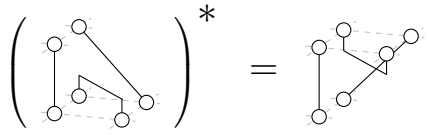
$\cong$ 

Example 2.24 (Operations on Spatial Partitions)**.** In the domain of spatial partitions, we have the operations tensor product, involution and composition. These operations on spatial partitions follow the same principles as on classical partitions. For example:

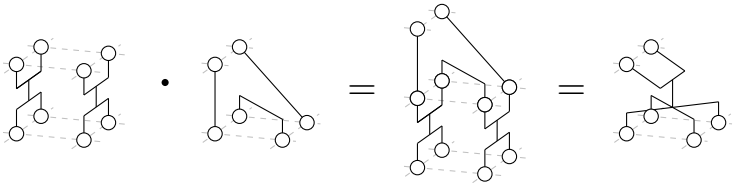
1. Tensor product



2. Involution



3. Composition



Definition 2.25 (Partition Amplification)**.** Let $p \in P(k,l)$ be a partition with $k, l \in \mathbb{N}$. Then we say $p^{(m)} \in P^{(m)}(k,l)$ is the *amplified version of p (on m levels)*, if $p^{(m)}$ is given by repeating p on each level d with $1 \leq d \leq m$.

Example 2.26. Let $p =$  , then $p^{(3)}$ is the amplified version of p, which looks as follows:

Definition 2.27 (Amplification of a Set of Partitions)**.** Let $\mathcal{C}$ be a set of partitions. Similar to Definition 2.25, we define $[\mathcal{C}]$ as the amplification of $\mathcal{C}$, if

$$[\mathcal{C}]^{(m)} := \{p^{(m)} \mid p \in \mathcal{C}\} \subseteq P^{(m)}$$

Definition 2.28 (Category of Spatial Partitions)**.** A *category of spatial partitions* $\mathcal{C}$ be a , then $\mathcal{C} \subseteq P^{(m)}$ is a set $\subseteq P^{(m)}$, which is closed under the operations tensor product, involution and composition with the base partitions  $^{(m)} \in P^{(m)}(1,1)$ and $^{(m)} \in P^{(m)}(0,2)$. Similar to classical partitions, we can represent categories of spatial partitions by a set S of generator spatial partitions with $\langle S \rangle$.

Remark 2.29. Note that, if $\mathcal{C} \subseteq P$ is a category of partitions, then $[\mathcal{C}]^{(m)} \subseteq P^{(m)}$ is a category of spatial partitions.

Example 2.30. The following sets are examples of categories of spatial partitions defined in [CW16], which are also relevant for the subject of this thesis.

1. The set $[P]^{(m)} = \langle$  $^{(m)}$,  $^{(m)} \rangle$
2. The set $[P_2]^{(m)} = \langle$  $^{(m)} \rangle$
3. The set $[NC_2]^{(m)} = \langle$  $^{(2)}$,  $^{(2)} \rangle$ of NC_2 as the minimal category of all spatial partitions
4. The set $P^{(2)} = \langle$  $^{(2)}$,  ,  , ,  ,  $\rangle$
5. The set $P_2^{(2)} = \langle$  ,  ,  ,  ,  $\rangle$

2.2 Computer Science Background

2.2.1 Asymptotic Analysis and Notation

In order to analyze an algorithm regarding the efficiency and the resulting runtime, there are several types of asymptotic notation which are used to compare the relative performance. Particularly, we will focus on the so called "Big-O notation" and "Ω-notation".

Definition 2.31 (Big-O notation)**.** Let $g : \mathbb{N} \to \mathbb{N}$ be functions. We define $O(g)$ as the following set:

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{N} \mid \exists_{c>0}, \exists_{n_0>0}, \forall_{n \geq n_0} : 0 \leq f(n) \leq c \cdot g(n)\}$$

As a result, this simply means $f \in O(g)$ if and only if we can find a c and a n_0 such that for all n greater than n_0, we satisfy $0 \leq f(n) \leq c \cdot g(n)$. In other words, g grows "faster" than f.

Note, that this leads to the following property:

$$\limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

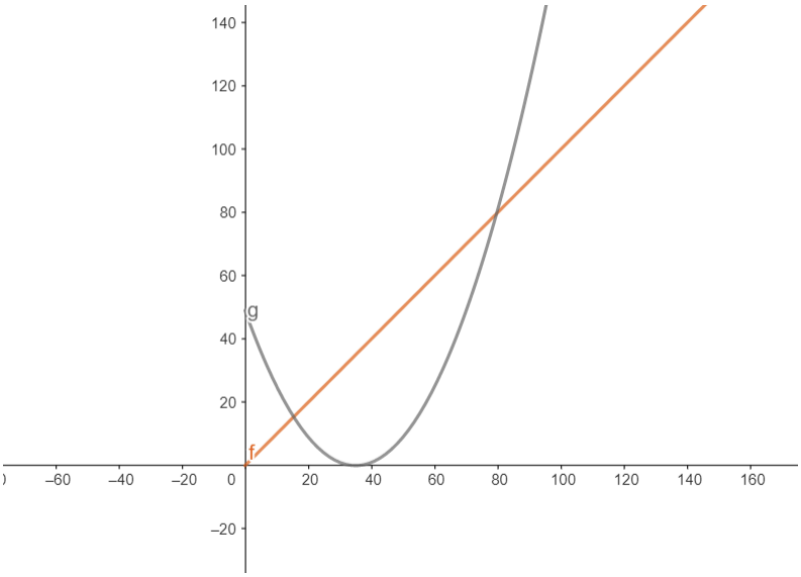


Figure 1: Example of two function g and f with $f \in O(g)$

In Figure 1, we can see an example in a geometrical point of view, in which $f \in O(g)$. We can clearly see, that there is a point n_0 from which on f is smaller than g.

Definition 2.32 (Ω-notation)**.** Let $f, g : \mathbb{N} \to \mathbb{N}$ be functions. We define $\Omega(g(n))$ as the following set:

$$\Omega(g(n)) = \{f \mid \exists_{c>0}, \exists_{n_0>0}, \forall_{n \geq n_0} : 0 \leq c \cdot g(n) \leq f(n)\}$$

The Ω-notation is used for assigning a problem a lower bound, regarding the runtime complexity. Its application is to make predictions about how efficient can an algorithm be in order to solve the underlying problem.

Example 2.33. The following examples apply.

1. Let $f : \mathbb{N} \to \mathbb{N},\ n \mapsto 1000$. Then
$$f \in O(1).$$
If $f : \mathbb{N} \to \mathbb{N},\ n \mapsto n$. Then
$$f \notin O(1).$$
2. Let $g : \mathbb{N} \to \mathbb{N},\ n \mapsto 1000n$. Then
$$g \in O(n).$$
If $g : \mathbb{N} \to \mathbb{N},\ n \mapsto n \cdot log(n)$. Then
$$g \notin O(n).$$

3. Let $g : \mathbb{N} \to \mathbb{N},\ n \mapsto n^k$, where $k \in \mathbb{N}$ and let $f(n) = \sum_{i=0}^{k} a_i n^i$ be a polynomial of degree at most k. Then
$$f \in O(g).$$

Proof. To prove the statement $f \in O(g)$, we show that
$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty. \tag{1}$$
If we insert the functions, we get
$$\limsup_{n\to\infty} \frac{\sum_{i=0}^{k} a_i n^i}{n^k} < \infty. \tag{2}$$
Since $\limsup_{n\to\infty} \frac{n^l}{n^k} = 0$, for $l < k$, we can reduce the equation to
$$\limsup_{n\to\infty} \frac{a_k n^k}{n^k} = a_k < \infty. \tag{3}$$
□

Remark 2.34. Let $k \in \mathbb{N}_0$. The most commonly used algorithmic runtimes are
$$O(n^k),\ O(n^k \cdot \log(n)),\ O(k^n),\ O(n!).$$

2.2.2 Data structure: Dictionary

The *dictionary* is a frequently used data structure to optimize different operations. It is used for storing a group of objects. Because *sets* and dictionaries have a strong relation, we first introduce them.

Data Structure Set. The *set* in data type theory is a computer implementation of the mathematical concept of *sets*. It stores unique objects without indices.
In order to guarantee the uniqueness of the elements in the set, so called *hash functions* and *collision resolution techniques* (see [MST20]) are used. Hash functions map data to a fixed-size value. Optimally we have the property, that two inputs of a hash function are equal, if and only if the resulting output value is equal. If our hash function does not satisfy this property, we use a collision resolution technique. Since the hash function can be calculated in $O(1)$, we are able to access a set in an average runtime of $O(1)$.

Data Structure Dictionary. A *dictionary* is a data structure for storing pairs of values in a key-value relationship. Every key in a dictionary points to a specific value. Analogical to a set, the dictionary transforms the keys to a hash value. Consequently, the keys are unique like the elements in a set. For example,
$$a = \{4 : 5, -14 : 6, "example" : 10, 6 : 6\}$$
is a valid dictionary, while
$$b = \{4 : 5, -14 : 6, "example" : 10, 4 : 6\}$$
is an invalid dictionary. Similarly to the set, we have the advantage of accessing the key elements in a dictionary in an average runtime of $O(1)$.

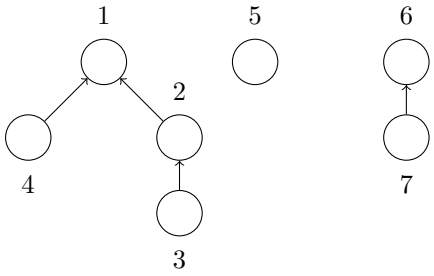
2.2.3 Path Compression

Path compression is a technique frequently used in the *Union-Find* data structure, see [Tar79]. We will use this technique to optimize the runtime of the composition operation, discussed in Section 2.1.2, regarding the runtime.
Because we use the *path compression* technique on a *directed forest*, we first define the terms *directed tree* and *directed forest*, see [Deo12].

Definition 2.35 (Directed tree)**.** A *directed tree* is a directed graph with exactly one root r. For every node v there is exactly one path from v to r. The definition of a *directed tree* is equivalent to a directed rooted connected acyclic graph.

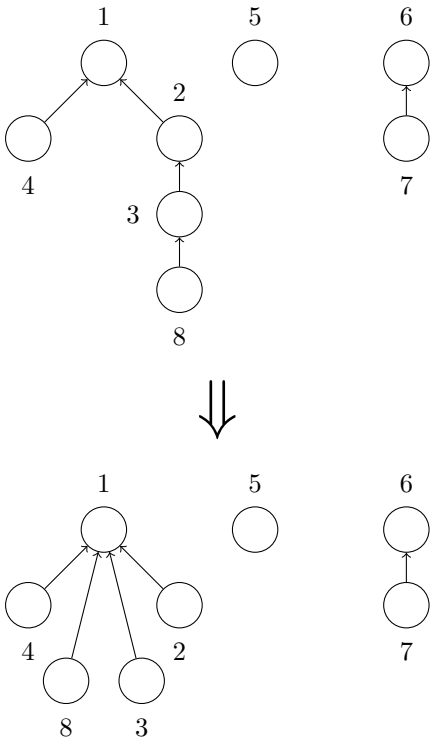
Definition 2.36 (Directed forest)**.** A *forest* is a directed graph which allows multiple roots. For every node v there is at most one path to a root. The definition of a *directed tree* is equivalent to an directed rooted (not necessarily connected) acyclic graph. In contrast to a *directed tree*, the directed forest can consist of more than one connected component.

Example 2.37 (Directed forest)**.**



Note that each connected component in a forest is a *directed tree*.

Path Compression. Let $\mathcal{F} = (V, E)$ be a forest of a set V representing the nodes and a set E representing the edges in which $\forall_{e \in E} : e \in V \times V$. If there is a path $v \to p \to r$, where r is the root, p is a path and p_{nodes} are all nodes in p, then we assign the set $v \cup p_{nodes}$ as children of r, while deleting all edges in $v \to p$ from E. For example:



In this example, we can observe, that the path $8 \to 3 \to 2 \to 1$ has been compressed, following the definition above.

```
for start_node in new_ids:
    a = new_ids.get(start_node)
    path = [start_node]
    while a in new_ids:
        path.append(a)
        a = new_ids.get(a)
    path.append(a)
    for node in path:
        if node != path[-1]:
            new_ids[node] = path[-1]
```

The code above shows an implementation example, which is used in the code [Vol23] to improve the composition operations, as already stated above. The code compresses every path in a forest containing edges, which have not yet been compressed. The variable *new_ids* is a dictionary from node to node.
In general, the dictionary represents the directed forest we want to compress. Every key and value in the dictionary stands for a node $v \in V$ in the forest $F = (V, E)$. Every two nodes v and u, which are in a key-value relationship in the dictionary, describe an edge $(v, u) \in E$. This algorithm has an average runtime of $O(n)$ with $n = |E|$, since we either compress or iterate over every edge at most once and the dictionary access is in average $O(1)$, see Section 2.2.2.

2.2.4 SymPy Python Package

In this section, we will introduce the SymPy package, which is a Python library for symbolic mathematics. It is an open-source computer algebra system written in pure Python. This package is used in Section 5 in order to implement the operations for linear combinations of partitions. In particular, we need to briefly introduce *symbolic variables*, the *solve function* and the *expand function*. More information about this package can be found in [MSP+17].

Symbolic Variables

The SymPy library offers the functionality of symbolic variables. We can define such a variable, by importing the sympy.abc module as follows:

```
from sympy.abc import d
```

By doing so, we can represent terms and expressions that depend on the variable d.

The solve function

The *solve* function is a powerful tool in sympy for solving algebraic equations and systems of equations. It takes an expression or a list of expressions as input and calculates the values of the variables that satisfy the equation. For example, to solve the equation $d^3 + d^2 - 6d = 0$, we can use the *solve* function as follows:

$$solve(d^3 + d^2 - 6d, d).$$

This will return the solutions $\{-3, 0, 2\}$. In other words, we can use it to find the roots of a polynomial.

The expand function

The expand function is used to simplify algebraic expressions by expanding them. It takes an expression as input and returns its expanded form. For example, to expand the expression $(d + 3) \cdot (d - 2) \cdot d$, we can use the expand function as follows:

$$expand((d + 3) \cdot (d - 2) \cdot d).$$

This will return $d^3 + d^2 - 6d$.

3 Algorithms and Data Structures for Partitions and Their Operations

In this section, we will explain the design choices of the implementation of partitions and their operations (see Section 2.1.1 and Section 2.1.2). The whole implementation, including the algorithms discussed in this chapter, can be found in [Vol23].

3.1 Design Description and Implementation Details

Prior to implementing the operations, we must first find an efficient data structure to represent a partition. This allows the computer to operate on the partition effectively. When implementing the operations, it is crucial to consider the chosen data structure's suitability and impact.

Data Structure for Partitions

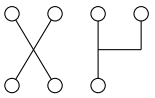
Since partitions are characterized by a set of upper and lower points, it is evident that we shall use a list of two lists $(List[List[], List[]])$. The first list stands for the upper points, while the second stands for the lower points.
A point of a partition is represented by a number $n \in \mathbb{N}$, with the following rule

$$connected(a, b) \Leftrightarrow a = b$$

for all points a and b in a partition, where *connected* returns *true* if the inputs are in the same block.

Example 3.1.

$$\begin{matrix}[[1,2,3,3],\\ [2,1,3]]\end{matrix} \cong$$



Algorithm 3.2 (Normal Form Algorithm)**.** We clearly need an efficient method to distinguish partitions. It is often the case, that two partition objects with a different syntax has the same semantic. For example,

$$\begin{matrix}[[1,2,3,3],\\ [2,1,3]]\end{matrix} =$$

$$= \begin{matrix}[[2,3,4,4],\\ [3,2,4]]\end{matrix}$$

.

Because of that, we want to ensure that, two partitions are equal, if and only if they have the same syntax. As a result, the goal of this algorithm is to assign an input partition new point values from 1 to the number of blocks in the partition, while we only change the syntax but not the semantics of the partition. With that property, we can simply assume that partitions with equal lists are equal.

Algorithm 3.3 (Helper Function Algorithm)**.** Before proceeding with the operations on partitions, we require a helper function that assigns new point values to a partition without altering its structure. Let p, q be partitions with $p \in P(k_1, l_1)$ and $q \in P(k_2, l_2)$. Then we assign q new point values greater than the greatest point value in p. After this step, the point values in p and q are disjoint, since

$$\forall_{point_p \in p} \forall_{point_q \in q} : point_p > point_q$$

.

Now we will proceed by introducing the algorithms for the operations on partitions (see Section 2.1.2).

Algorithm 3.4 (Tensor Product Algorithm)**.** Let p, q be partitions with $p \in P(k_1, l_1)$ and $q \in P(k_2, l_2)$ and $List[List[k_1], List[l_1]]$, $List[List[k_2], List[l_2]]$ be the respective representations of p and q, regarding the data structure introduced above. Then the tensor product $p \otimes q$ can be implemented by the concatenation of the upper lists $list[k_1]$ and $list[k_2]$ and the lower lists $list[l_1]$ and $list[l_2]$. In order to preserve the blocks, we have to apply the helper function algorithm (Algorithm 3.3) before concatenating.

Example 3.5 (Tensor Product Algorithm)**.** In this example, we can see, that the algorithm first applies the helper function algorithm (Algorithm 3.3) followed by the concatenation process.

$$\begin{matrix}[[1,2],\\ [2,1]]\end{matrix} \otimes \begin{matrix}[[1,1],\\ [1]]\end{matrix} = \begin{matrix}[[1,2],\\ [2,1]]\end{matrix} \otimes \begin{matrix}[[3,3],\\ [3]]\end{matrix} = \begin{matrix}[[1,2,3,3],\\ [2,1,3]]\end{matrix}$$

Algorithm 3.6 (Involution Algorithm)**.** The involution operation, for the data structure introduced above, can be implemented by inverting the outer list, i.e. swapping the first list standing for the upper points with the second list standing for the lower points.

Example 3.7 (Involution Algorithm)**.** In this example, we can see, that the algorithm simply swaps the two inner lists.

$$\begin{pmatrix}[[1,2,3,3],\\ [2,1,3,4]]\end{pmatrix}^* = \begin{matrix}[[2,1,3,4],\\ [1,2,3,3]]\end{matrix}$$

Algorithm 3.8 (Composition Algorithm)**.** Since the composition operation is one of the main operations, it is evident, that our goal is to design an adequate algorithm. In order to get an efficient implementation of the composition operation, we make use of dictionaries (Section 2.2.2) and path compression (Section 2.2.3). Given a partition p and a partition q, we implement the composition pq as represented by the following Python pseudo-code

```
1  q = p.helper_function(q)
2
3  new_ids = dict()
4
5  for index, n in enumerate(q[1]):
6      if n not in new_ids:
7
8          new_ids[n] = p[0][index]
9      else:
10
11         if p[0][index] in new_ids and new_ids.get(n) in new_ids:
12
13             Do path compression from node n and p[0][index]
14             """connect the two components"""
15             new_ids[new_ids.get(n)] = p[0][index]
16         else:
17             if new_ids.get(n) not in new_ids:
18                 new_ids[new_ids.get(n)] = p[0][index]
19             else:
20                 new_ids[p[0][index]] = new_ids.get(n)
21
22 Do path compression
23
24 """giving the top part new values"""
25 for index, n in enumerate(q[0]):
26     if n in new_ids:
27         q[0][index] = new_ids.get(n)
28
29 """giving the bottom part new values"""
30 for index, n in enumerate(self.partition[1]):
31     if n in new_ids:
32             p[1][i] = new_ids.get(n)
33
34 return new partition with q[0] as upper, p[1] as lower points
```

Let us explain the algorithm above.
As a first step, we apply our helper function in Line 1, introduced in the *tensor product* paragraph, to q, so that we can guarantee that the point numbers of p and q are disjoint.

The algorithm iterates through the lower points in q and the upper points in p, while storing the information about the changed point values (representing the blocks) into the dictionary *new_ids*,

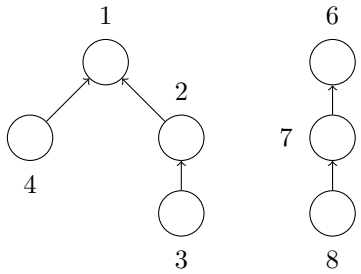
mapping each point value to its updated value (i.e. the point values from p to the respective point values from q). With that information, we adjust the two partitions and unite the overlapping disjoint subsets, like described in Section 2.1.2. As a result, the dictionary *new_ids* goes from old to new point value.

After that step we perform path compression in Line 21 with our dictionary *new_ids* as *forest*, as shown in Section 2.2.3. As a last step we just need to explicitly change the point numbers and return them as a new partition.

We also need to do path compression in Line 13 due to the dictionary property, that the keys have to be unique. We can see this special case in the example below.
A dictionary can be seen as a *forest* as follows:
Given the following dictionary.

$$dict\_a = \{4:1, 2:1, 3:2, 7:6, 8:7\}$$

We can now build the following *forest* from it.



If we would now need to connect the nodes 2 and 8, we would either lose the information that 7 and 8 are connected or that 2 and 1 are connected. Because of this conflict, we compress the paths from 2 to the root and 8 to the root, so that we are able to connect the two trees by adding an edge from the value of 2 to the value of 8, namely from 1 to 6.

Example 3.9 (Composition Algorithm)**.** In this example, we want to apply the composition algorithm on $p \cdot q$, where

$$p = \begin{matrix}[[1,2,2,2],\\ [2,3,4]]\end{matrix} \qquad q = \begin{matrix}[[5,6],\\ [7,7,5,6]]\end{matrix}.$$

According to the algorithm, we iterate through the upper points of p and the lower points of q. During the iteration process, we maintain a record of the specific point values in q that need to be transformed into corresponding point values from p, in form of a dictionary. This allows us to merge these two partitions together. As a result we get the dictionary

$$\{7:1, 1:2, 5:2, 6:2\}.$$

The dictionary will then be processed with path compression, so that we get

$$\{7:2, 1:2, 5:2, 6:2\}.$$

Now, the algorithm replaces the lower point values of p and the upper point values of q with the respective values in the dictionary. At the end, the algorithm outputs the partition

$$p \cdot q = \begin{matrix}[[2,2],\\ [2,3,4]]\end{matrix}.$$

Algorithm 3.10 (Rotation Algorithm)**.** In order to get an algorithm for the rotation, we proceed as follows. Given a partition p, represented by our partition data structure, with the goal of getting a rotated version, we can simply remove one of the first or last element of one of the lists and append it to the respective first or last element of the other list, depending on whether we want to perform either a left or right or either a top or bottom rotation.

Example 3.11 (Rotation Algorithm top-left)**.**

$$\left(\begin{matrix}[[1,1],\\ [1,2]]\end{matrix}\right)^{rot} = \begin{matrix}[[1],\\ [1,1,2]]\end{matrix}$$

Algorithm 3.12 (Vertical Reflection Algorithm)**.** To obtain a vertically reflected form of the input partition p, it is sufficient to reverse the two lists in the outer list of the partition data structure.

Example 3.13 (Vertical Reflection Algorithm)**.**

$$\begin{pmatrix} [[1,2,2], \\ [1,1,3]] \end{pmatrix}^{\leftrightarrow} = \begin{matrix} [[2,2,1], \\ [3,1,1]] \end{matrix}$$

3.2 Evaluation and Results

In order to evaluate the runtime of the presented techniques above, we will use the *Big-O Notation* introduced in Section 2.2.1.

Space Complexity

Let n be the number of points in our partition p. The space complexity of the chosen data structure is $O(n)$, because we store for every point a value in a list.

Time Complexity

Theorem 3.14. *The time complexity of the tensor product is $O(n)$.*

Proof. In order to proof this, we show that the runtime of the algorithm in Section 3.1 is $O(n)$. At first sight, one might guess that, the runtime of the tensor product should be $O(1)$, by using linked lists, because we simply need to concatenate two lists. However, because we need to adjust the point values of one of the partitions as described in Section 3 with the helper function, we have a time complexity of $O(n)$.

Indeed, the helper function algorithm (introduced in Algorithm 3.3) has a runtime of $O(n)$, because we need to find the maximal point value in one of the partitions, which has for obvious reasons a worst case runtime of $O(n)$. After that, we only need to assign new values greater than the calculated maximal number as follows:

```
1        new_ids = dict()
2        for n in p:
3            new_ids[n] = max_id
4            max_id += 1
5
6        for i, n in enumerate(p[0]):
7            p[0][i] = new_ids.get(n)
8
9        for i, n in enumerate(p[1]):
10           p[1][i] = new_ids.get(n)
11
12       return p
```

Here, *max_id* is the maximal point value and p is the partition we want to assign new values. Since we iterate a constant amount of times through the n point partition p, namely three, we also have a time complexity of $O(n)$.

□

Theorem 3.15. *The time complexity of the involution is $O(1)$.*

Proof. Since we only swap two elements in a list, namely the upper list and the lower list in the outer list of the data structure, where each list is considered as one element, we have a constant time complexity $O(1)$. □

Theorem 3.16. *The time complexity of the composition is $O(n \log(n))$.*

Proof. In order to proof this, we show that the runtime of the algorithm for the composition in Section 3.1 is $O(n \log(n))$. Let p and q be partitions. After initializing q with new point values, by calling the helper function, we iterate through every lower point of q and upper point of p. In every iteration we check whether we already have the point we are operating on in our dictionary from old to new point values. Since we use dictionaries (see Section 2.2.2) and the helper function, which has a time complexity of $O(n)$, we have an overall runtime of $O(n)$ until this state. After we are done iterating through all the points, we perform path compression, as shown in Section 2.2.3, and assign the old points the new point values. Both of these steps have a time complexity of $O(n)$.

However as illustrated in Section 3.1, there is a special case, in which we need to perform one path compression operation during the iteration process to prevent information loss. Since we merely do this compression technique for two paths and in as much as the fact that our work is not

entirely redundant, we get a worst case runtime of $O(n \log(n))$. This is because we can compare our algorithm with the *find* operation of the *union find* data structure with path compression, which has a worst case runtime of $O(n \log(n))$, see [Fis72].

As a result the overall runtime of the composition operation is $O(nlog(n)))$ □

Remark 3.17. This remark discusses the extent in practise of performing path compression, as described above, and performing path compression in every iteration step. As a result, we can see the benefit of introducing the special case, instead of naively running path compression in every for iteration.

partition size	6	8	10	12	14	16
steps w/o redundancy	912	13.950	206.161	3.027.009	44.507.030	657.155.024
steps w redundancy	988	19.425	348.164	5.985.883	100.507.030	1.666.412.924

In the table above, we can observe the extent of performing path compression as described above (see Row 1) and the extent of performing path compression in every for iteration (see Row 2).

This data comes from counting the path compression steps while constructing every possible partition in NC_2 (see Section 2.1.3) with the algorithm represented in Section 4. Every time the algorithm checks whether a node needs to be compressed and every time we compress a node along one edge, the counter is incremented. The numbers $6, 8, 10, 12$ and 14 are the sizes n with $NC_2 \cap P(n)$.

The larger our n gets, the greater the gap between these two different approaches becomes. This illustrates, that performing path compression after the for iteration and in the case discussed and applied in Algorithm 3.7, is more efficient, than performing it in every for iteration step.

Theorem 3.18. *The time complexity of the rotation is $O(n)$.*

Proof. Because we can append and remove an element to or from a list in $O(n)$, we have an overall runtime of $O(n)$. □

Theorem 3.19. *The time complexity of the vertical reflection is $O(n)$.*

Proof. To obtain a reversed version of a list, we have to iterate through the whole list. Since our partition is of size n, we have a time complexity of $O(n)$. □

Remark 3.20. Note, that by adding the *Normal Form Algorithm*, explained in Section 3.1, we add an extra runtime of $O(n)$ to every operation. This only effects the runtime of the involution and rotation algorithm.

4 Algorithm 1: Constructing Categories

In the following chapter, we will present and evaluate an algorithm for constructing categories of partitions. Let $\mathcal{C}$ be a category, then we are interested in $\mathcal{C} \cap P(n)$ (see Section 2.1.3). The whole implementation, including the algorithms discussed in this chapter, can be found in [Vol23].

4.1 Design Description and Implementation Details

The goal of designing and implementing an algorithm for constructing categories is to get a tool, which is able to retrieve the size of specific categories as well as to output all partitions contained in a specific category. The input is a set of partitions (represented by the data structure shown in Section 3.1) and a parameter $n \in \mathbb{N}$. The output is desired to be a set of all partitions, generated by our input set with all possible operations (see Section 2.1.2), of size n.

More precisely, we want to compute the following:
Let $\mathcal{P}$ be a set of partitions, we are interested in

$$\langle \mathcal{P} \rangle \cap P(n) \tag{1}$$

and

$$\#(\langle \mathcal{P} \rangle \cap P(n)) \tag{2}$$

Note, that (2) can simply be retrieved with (1), but not necessarily the opposite. Because of this property we concentrate on finding an algorithm for (1) and then calculate the length of the output to get to a solution for (2). However, the algorithm presented in this chapter does not guarantee finding every partition in $\langle \mathcal{P} \rangle \cap P(n)$. Its goal, however, is to achieve this. The reasons for this circumstance will also be discussed in this section.

Basic Algorithm Structure

The figure *Algorithm 1* below shows the basic structure of our approach in a naive setting, namely without optimization techniques and the restriction on extending a certain number of points in the partitions we operate on.

Algorithm 1: Basic structure of the algorithm without optimization techniques

```
   Input: List of partitions p and partitions size n
   Output: Set of partitions ⟨p⟩ ∩ P(n)
 1 Initialize variables
 2 while new partitions found in last iteration or at beginning do
 3 |   while new partitions found in last iteration do
 4 |   |   do every possible unary operation and store new found partitions
 5 |   end
 6 |   while new partitions found in last iteration do
 7 |   |   do every possible tensor product operation and store new found partitions
 8 |   end
 9 |   while new partitions found in last iteration do
10 |   |   do every possible composition operation and store new found partitions
11 |   end
12 end
13 return all found partitions of size n;
```

We will use this structure with some optimization techniques, introduced in this chapter below, to prevent redundant computation. We will begin by examining Line 4, where we initialize the required variables. In succession, we concentrate on the outer while loop, where the preparation for the inner while loops take place. Subsequently, we discuss the functionality of the inner while loops. During the whole explanation, we compare our techniques with more naive approaches.

Used variables

First of all, we will examine the variables which are used. For a naive algorithm, we would only need to initialize the following variables:

```
#store all candidates found
    all_partitions = {Partition([1, 1], []), Partition([1], [1])}

#end output: All partitions found of size n
    all_partitions_of_size_n = set()

#compare allowed expansion size with max(n, max_length)
max_length = n
```

Note that we use not only these variables but also additional optimization variables explained in the upcoming paragraph. Let n be the input size and p_{max} be the greatest partition in the input set p, regarding the number of points. The *all_partitions* variable is our container set for storing all found partitions of size $\leq \max(n, \text{size}(p_{max}))$. Through generating the categories listed in Section 4.2, it seems to be sufficient to not let any partitions expand greater than $\max(n, size(p_{max}))$, represented by the variable *max_length*, in order to get to the desired output for the listed categories in Section 2.1.3. It must be emphasized, that this property does not apply for all generating partition sets. The variable *all_partitions_of_size_n* is our output set, in which we store all partitions from the set *all_partitions* with size equal n. This is done at the end of the algorithm.

To avoid computation which has already happened, we introduce the following extra variables:

```
    #all candidates stored in dict from size to partition
    all_partitions_by_size = dict()

    #all candidates stored in tuple with a dict for top size to
        partition and bottom size to partition
    all_partitions_by_size_top_bottom = (dict(), dict())

    #store partitions already unary operated
    already_u = set()

    #store partitions already tensor product operated
    already_t = set()

    #store partitions already composition operated
    already_c = set()

    #all candidates for unary operations
    to_unary = set()

    #store pairs that are candidates to get tensor product operated
    to_tens = set()

    #store pairs that are candidates to get composition operated#
    to_comp = set()
```

The variables *already_u*, *already_t*, and *already_c* keep track of the partitions that have already undergone the corresponding operations. The variable *already_u* contains all partitions processed by the unary operations. In the variables *already_t* and *already_c*, we store the pairs of partitions, which have already been operated regarding tensor product and composition operations. Since these binary operations are not commutative, the order has to be taken into account. We use the variables *to_unary*, *to_tens*, and *to_comp* to store the partitions and partition pairs intended for processing with their corresponding operations. These three variables are emptied every time a new iteration of the outer while loop starts.

To optimize the retrieval of candidates added to the sets *to_unary*, *to_tens*, and *to_comp*, we introduce the dictionary *all_partitions_by_size* and the variable *all_partitions_by_size_top_bottom*. This approach helps to avoid iterating through the entire *all_partitions* set repeatedly.

As the name suggest, the variable *all_partitions_by_size* contains every partition with the corresponding size in a dictionary from size to a set of partitions. This variable is used for getting the candidates for the tensor product operation in advance, since we do not want to exceed the *max_length*.

Similarly, the variable *all_partitions_by_size_top_bottom* incorporates two dictionaries: one from size of the top part to a set of partitions and the other from size of the bottom part to a set of partitions. This variable is of practical value for getting candidates for the composition operation in advance, since we do not want to exceed the *max_length* and since we have to keep a specific format for the composition operation (see Section 2.1.2).

Note that, without these dictionaries, we would need to iterate through every partition in *all_partitions* for each partition in *all_partitions*, resulting in a significantly less efficient process. But with these dictionaries, we only iterate for every partition in *all_partitions* through the fitting partitions in *all_partitions*, considering constraints like size and format.

After adding the base partitions and the input list of partitions to the respective sets, we start the operation process.

Outer while-loop

In preparation for the inner while-loops, we need to calculate the candidates which are going to be operated on and processed, in order to add them into the respective sets *to_unary*, *to_tens* and *to_comp*. Take into account that we are now looking at the while loop of Line 2 to 12 of the basic algorithm code.

The preparation for the unary operations is remarkably intuitive, by simply iterating through the *all_partitions* set and adding every partition to the *to_unary* set, if it is not contained in the *already_u* set.

In the process of the tensor product, we also iterate through the *all_partitions* set. Since we now have a binary operator, we have to iterate again through a set of partitions in order to get pairs that we can add into the *to_tens* set.
The naive implementation would look as follows

```
1    for i in all_partitions:
2        for ii in all_partitions:
3            if (i, ii) not in already_t:
4                to_tens.add((i, ii))
5                already_t.add((i, ii))
```

Since the *all_partitions* set can become exceedingly large, we would get a long processing time for just iterating through every pair. This process is fairly inefficient since most of the candidates would exceed the *max_length* limit by applying the tensor product. The solution is to use the *all_partitions_by_size* dictionary, by considering for every partition in the *all_partition* set only the partitions, which are not resulting in a partition being greater than *max_length*.
This can be implemented as follows

```
1    for i in all_partitions:
2        all_partitions_temp_tensor = set()
3        for key in all_partitions_by_size.keys():
4            if tuple_to_partition(i).size() + int(key) <= max_length:
5                all_partitions_temp_tensor = all_partitions_temp_tensor
                     .union(all_partitions_by_size.get(key))
6        for ii in all_partitions_temp_tensor:
7            if (i, ii) not in already_t:
8                to_tens.add((i, ii))
9                already_t.add((i, ii))
```

This analogically applies for the composition operation as follows

```
1     for i in all_partitions:
2     all_partitions_temp_comp =
3     all_partitions_by_size_top_bottom[1].get(len(i[0]))
4         for ii in all_partitions_temp_comp:
5             if (i, ii) not in already_c:
6                 if len(i[1]) + len(ii[0]) <= max_length:
7                     to_comp.add((i, ii))
8                     already_c.add((i, ii))
```

For the composition, we additionally check whether the format is fitting.

Applying unary operations

Now we are in Lines 3 to 5 of the basic structure pseudo-code of our algorithm.

In general, we will distinguish between three different unary operations for our algorithms: the involution, the rotation, and the vertical reflection (see Section 2.1.2).

For the purpose of getting an algorithm which performs every possible unary operation, we can formulate two main approaches

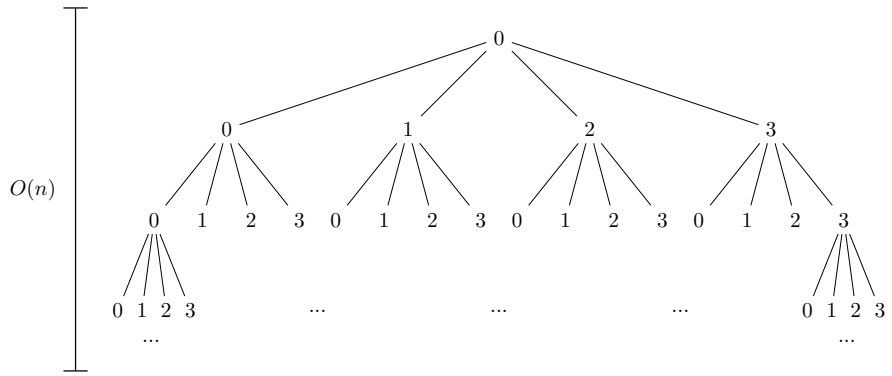
Naive Approach

The rotation and the vertical reflection can be represented by a combination of the three main operations: the involution, the tensor product and the composition. Nevertheless, we will include them, because it safes computation time in practise. In theory, when evaluating the runtime of our algorithm using the Big-O notation, the difference would be negligible, as both the simulation of the rotation and the vertical reflection require only a constant number of steps.

With the goal of obtaining every possible combination of the three unary operations, we encode them to integer values as follows

0 stands for do nothing,
1 stands for rotation,
2 stands for involution,
3 stands for vertical reflection.

Now, we can construct a tree of degree 4 with the operation values as nodes in the following manner:



The tree above has the height $O(n)$, because we can rotate a partition of size n into n different states. Furthermore, with the involution and the vertical reflection, we are able to transform a partition to one other state. As a result, we can apply every possible combination to a partition, if we follow the tree structure above by traversing every path of the tree from the root until we end up at a leaf. While passing an edge, we apply the corresponding operation, encoded via integers with the rules described above.

With this technique we can ensure to transform a partition into every other state, which can be reached by the defined unary operations. However, this approach has several disadvantages. Since we have a tree of degree four and a depth of $O(n)$, we need to traverse $4^{O(n)}$ different paths, which leads to a massive time complexity.

This can be attributed to the fact, that the tree contains numerous redundancy. For example, the following paths have all the same effect on a partition:

$$0 \to 0 \to 1 \to 0 \to 0,$$
$$0 \to 0 \to 0 \to 0 \to 1,$$
$$0 \to 1 \to 0 \to 0 \to 0,$$

as well as

$$0 \to 2 \to 1,$$
$$0 \to 1 \to 2,$$

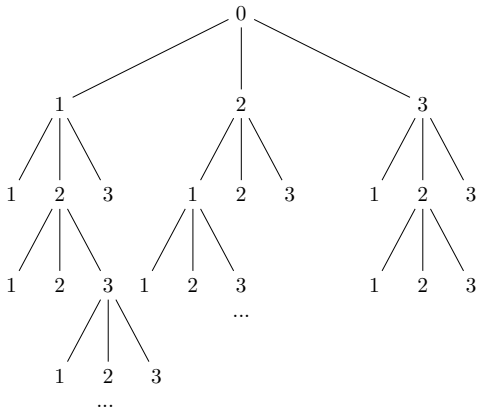
because the unary operations are in contrast to the binary operations commutative. As a result, even though this approach seems intuitive, we see that the redundancy is too great in order to design an efficient algorithm for our domain.

Used approach

To avoid the described redundancy, we use a method that includes a while loop. This loop terminates if there are no more possibilities to generate a partition not contained in *all_partitions*. This technique can be illustrated with the following pseudo-code.

```
1  while new partitions found in last iteration do
2      for partition in to_unary do
3          partition_rotated = rotation(pp)
4          add partition_rotated to all_partitions and to to_unary if not already in
             all_partitions
5          partition_involution = involution(pp)
6          add partition_involution to all_partitions and to to_unary if not already in
             all_partitions
7          partition_vertical_reflection = vertical_reflection(pp)
8          add partition_vertical_reflection to all_partitions and to to_unary if not already in
             all_partitions
9          remove partition from to_unary add partition to already_u
10     end
11     set while condition to false if to_unary is empty
12 end
```

Because we do not extend paths if they would get superfluous, we reduce the size of our tree substantially. In contrast to Approach 1, the structure of the tree could look as follows



Note, that we reduce the maximal degree to three, since we do not need the option of "doing nothing" represented by 0 anymore. In addition, we get a tree with significantly less paths as well. As a result, we use this approach for our algorithm.

Applying binary operations

Now, we are in Lines 6 to 11 of the basic structure pseudo code of our algorithm.

Since the methods for applying the tensor product and the composition are relatively comparable, it is more intuitive to elucidate them simultaneously.

The overall strategy looks as follows

```
while new partitions found in last iteration do
    for p in partitions found in last iteration do
        for pp in partitions found in last iteration with size ≤ max_length − size(p) for
          tensor product or with fitting format for composition do
            if (p, pp) not in already_t/c then
                add (p, pp) to to_tens/comp
            end
        end
    end
    for (p, pp) in to_tens do
        apply tensor product or composition
        store result for next iteration
        remove (p, pp) in to_tens/comp
        add (p, pp) in already_t/c
    end
end
```

Similar to the outer while loop, we also optimize the selection of candidates for the second entry in the tuples within *to_tens/comp*. This is achieved by avoiding extensions beyond the *max_length* value when performing the tensor product, or by only considering candidates with the appropriate format during the composition process. By doing so, we efficiently reduce the number of candidates for further operations, ensuring an effective algorithm. As already described above, we can use a dictionary from size to partitions or rather from size of the upper/lower point to partition for that domain.

With the combination of optimizing the process of choosing the candidates both in the outer and in the respective inner while loops and additionally simultaneously introducing constrains, we are able to avoid computations which are redundant.

Nevertheless, not every operation we perform leads to a new partition, since a partition can be constructed by more than one combination. As a result we still have superfluous work. However, this kind of redundancy is not preventable without using more resources than we would save.

4.2 Evaluation and Results

After designing the algorithm presented in Section 4.1, we are prepared to construct the categories defined in Section 2.1.3. After this process, we can evaluate the results, regarding the paper [Web13] and [BCS09], and find resulting sequences.

partition size	0	1	2	3	4	5	6	7	8
P	1	1	2	5	15	52	203	877	4140
NC	1	1	2	5	14	42	132	429	1430
$P^{1,2}$	1	1	2	4	10	26	76	232	764
$NC^{1,2}$	1	1	2	4	9	21	51	127	323

Table 1: Comparison and sizes of categories with not necessarily even block size.

Table 1 shows $\#(\langle\mathcal{P}\rangle \cap P(0,n))$ for $n \in \{0,1,2,3,4,5,6,7,8\}$, regarding the categories which contain both even and odd partitions. In contrast, Table 2 includes the categories with only partitions of an even size. These values are exclusively calculated from the algorithm presented in Section 4.1 and implemented in [Vol23]. Entries containing a hyphen ("-") signify extended computation durations on my computer.

Comparison of the results

In order to evaluate the results in Table 1 and Table 2, we compare our values with the moments of the laws of characters stated in [BS09], [BCS09] and [Web13]. The moments of a law of character count the partitions in a category. The laws of characters are a generalization of random variables

partition size	0	2	4	6	8	10
P_2	1	1	3	15	105	945
P_{even}	1	2	15	203	4140	-
$P_{even}^{1,2}$	1	2	10	76	764	-
$P_{even\ BS}$	1	1	4	31	379	-
NC_2	1	1	2	5	14	42
$NC_{even\ BS}$	1	1	3	12	55	273
NC_{even}	1	2	14	132	1430	-
$NC_{even}^{1,2}$	1	2	9	51	323	2188
B	1	1	3	16	131	1496
B_2	1	1	2	6	24	120
NCB_{even}	1	2	7	30	143	728
$B_{even}^{1,2}$	1	2	7	34	209	1546

Table 2: Comparison and sizes of categories with even block size.

in the free probability theory. An overview of the categories with their corresponding quantum groups can be found in Appendix A.

Theorem 4.1. *The outputs from the algorithm for P_2, NC_2 and B_2 match the laws of characters from [Web13].*

Proof. We get the followings laws from Proposition 4.1 in [Web13]:

- The law of P_2 is the real Gaussian, the moments can be calculated with the formula

$$g(n) = \begin{cases} 0 & \text{if } n \text{ odd}, \\ n!! & \text{if } n \text{ even}. \end{cases}$$

- The law of NC_2 is the semicircle, which corresponds to the Catalan numbers (see [Sta15])
- the law of B_2 is the squeezed complex Gaussian, the moments can be calculated with the series g of the real Gaussian as follows:

$$(squeezed\_complex(g_n))_k = \begin{cases} 0 & \text{if } k \text{ odd}, \\ \sum_{i=0}^{\frac{k}{2}} \binom{\frac{k}{2}}{i} \cdot g_{2i} \cdot g_{k-2i} & \text{if } k \text{ even}. \end{cases}$$

The corresponding implementations of the functions that yield the matching outputs can be found in [Vol23]. □

Theorem 4.2. *The outputs from the algorithm for $P^{1,2}, P_{even}^{1,2}, NC^{1,2}, NC_{even}^{1,2}, NCB_{even}$ and $B_{even}^{1,2}$ match the laws of characters from [Web13].*

Proof. We get the followings laws from Proposition 4.2 in [Web13], :

- The law of $P^{1,2}$ is the shifted real Gaussian, the moments can be calculated with the series g of the real Gaussian as follows:

$$(shifted(g_n))_k = \sum_{i=0}^{k} \binom{k}{i} \cdot g_i.$$

- The law of $P_{even}^{1,2}$ is the symmetry shifted real Gaussian, the moments can be calculated with the series g of the real Gaussian as follows:

$$(shifted(g_n))_k = \begin{cases} 0 & \text{if } k \text{ odd}, \\ \sum_{i=0}^{k} \binom{k}{i} \cdot g_i & \text{if } k \text{ even}. \end{cases}$$

- The law of $NC^{1,2}$ is the shifted semicircle, the moments can be calculated with the series sc of the semicircular as follows:

$$(shifted(sc_n))_k = \sum_{i=0}^{k} \binom{k}{i} \cdot sc_i.$$

- The law of $NC^{1,2}_{even}$ is the symmetry shifted semicircle, the moments can be calculated with the series sc of the semicircular as follows:

$$(shifted(c_n))_k = \begin{cases} 0 & \text{if } k \text{ odd}, \\ \sum_{i=0}^{k} \binom{k}{i} \cdot c_i & \text{if } k \text{ even}. \end{cases}$$

- The law of NCB_{even} is the squeezed shifted circle, which is related to the Fuss-Catalan numbers (see Proposition 4.3. in [Web13]) and can be calculated with the formula

$$f(n) = \frac{1}{k+1}\binom{3k+1}{k}.$$

- The law of $B^{1,2}_{even}$ is the squeezed shifted complex Gaussian $\sqrt{(1+\widetilde{g})(1+\widetilde{g})^*}$, the moments can be calculate using the following formula with the series g of the real Gaussian:

$$(squeezed(shifted(complex(g_n))))_k = \sum_{i_1+i_2+i_3+i_4=k} \binom{k!}{i_1! \cdot i_2! \cdot i_3! \cdot i_4!} \cdot (\sqrt{2})^{i_2-2i_3-2i_4} \cdot g_{i_2+2_{i_3}} \cdot g_{2i_4}.$$

The corresponding implementations of the functions that yield the matching outputs can be found in [Vol23]. □

Theorem 4.3. *The output from the algorithm for $P_{even\ BS}$ matches the law of character from [BS09].*

Proof. Comparing the values from Table 2 with [OEI], we get the sequence A005046, which corresponds to *number of partitions of a 2n-set into even blocks*. Accordingly, Proposition 2.5. from [BS09] says that the quantum group of P_{evenBS} are the partitions with blocks of size 2. □

Theorem 4.4. *The outputs from the algorithm for $P, P_{even}, NC, NC_{even}$ and B match the laws of characters from [BCS09].*

Proof. We get the followings laws from Theorem 7.3. in [BCS09]:

- The law of P is the Poisson, the moments can be calculated with the Bell numbers.
- The law of P_{even} is the symmetry Poisson.
- The law of NC is the free Poisson.
- The law of NC_{even} is the symmetry free Poisson.
- The law of B is the ∞-Bessel, which can be calculated with the formula

$$b(n) = \begin{cases} 1 & \text{if } n = 0, \\ \sum_{i=0}^{n-1} \binom{n}{i} \cdot \binom{n-1}{i} \cdot b(i) & \text{if } k \text{ even}. \end{cases}$$

The moments of the (symmetry) poisson distribution can be calculation from the Bell numbers and the moments from the (symmetry) free poisson distribution can be calculation from the Catalan numbers. The implementations can be found in [Vol23]. □

Theorem 4.5. *The output from the algorithm for $NC_{even\ BS}$ matches the law of characters from [BBC07].*

Proof. We get the following law from [BBC07]:

- The law of $NC_{even\ BS}$ is the free Bessel, which can be calculated with the following formula

$$b(n) = \sum_{b=1}^{k} \frac{1}{b} \cdot \binom{k-1}{b-1} \cdot \binom{2k}{b-1}.$$

The corresponding implementation of the free Bessel that yield the matching outputs can be found in [Vol23]. □

Runtime

Evidently, the property, the greater the category, the more resources we need, applies regarding time and space complexity. As a result, we can derive the following assumption.

Theorem 4.6. *Let $(a_k)_{k\in\mathbb{N}}$ be a sequence, describing the size distribution of a category and $n \in \mathbb{N}$ the input of the algorithm standing for the partition size (see Section 4.1). Then the algorithm presented in Section 4.1 has a worst case lower bound runtime of*

$$\Omega\left(\left(\sum_{i=1}^{k}(a_i)\right)_{k\in\mathbb{N},k\leq n} \cdot n\log(n)\right)$$

assuming, we use the operations on partition approximately constant equally often. However, in any case, the lower bound

$$\Omega\left(\left(\sum_{i=1}^{k}(a_i)\right)_{k\in\mathbb{N},k\leq n}\right)$$

applies.

Proof. Because $(a_k)_{k\in\mathbb{N}}$ describes the size distribution of a category and $n \in \mathbb{N}$ the input of the algorithm standing for the partition size (see Section 4.1). We have to construct at least

$$\left(\sum_{i=1}^{k}(a_i)\right)_{k\in\mathbb{N},k\leq n}$$

partitions $\leq max\_length$ in the category in order to get to an output.
Since the composition is the operation with the greatest time complexity of $O(n\log(n))$, we get a lower bound worst case runtime of

$$\Omega\left(\left(\sum_{i=1}^{k}(a_i)\right)_{k\in\mathbb{N},k\leq n} \cdot n\log(n)\right)$$

□

But as already discussed in Section 4.1, we cannot avoid having redundancy in the construction process. Because of that, it is hard to make precise predictions about the runtime in general.

For our algorithm, experience has shown, that for $n \leq 8$, we are able to get an output for every category within at most a few hours, even for the biggest category P. For smaller categories, like for example NC_2, we can go even slightly higher with the input n. These measurements are, of course, influenced by the performance of the computer being used.

The following table shows the comparison of the smallest category NC_2, only containing the base partitions, and the greatest category P, containing all partitions. We compare the two categories based on the number of operations that are performed until the algorithm terminates.

partition size	1	2	3	4	5	6	7	8
P	1717	1717	1717	1717	16.480	246.775	4.712.101	114.603.058
NC_2	-	10	-	97	-	499	-	3104

We can clearly see the enormous difference between the resources needed for constructing P and NC_2, and on the other hand the general amount of operations, which is significantly greater than the respective sums of the sequences as described above. This is the result of the already discussed redundancy, which is nearly inevitable.

Note that for P, we perform the same amount of operations in each computation until partition size 4.

Proof. Let p_{max} be the greatest generator partition and n be the input partition size. Then the algorithm produces every partition $\leq \max(n, \text{size}(p_{max}))$. For $n \leq 4$, $\max(n, size(p_{max}))$ is equal to 4, since the generator partitions of P are $\in P(4)$ and $\in P(3)$. Therefore, we perform the same amount of operations for $n \leq 4$, namely 1717. □

In practise with the *SAMSUNG Galaxy Book Pro EVO* featuring an *Intel® Core™ i7 Processor* and 16 GB of RAM, I observed the following approximate construction times associated with Table 1 and Table 2:

partition size	0	1	2	3	4	5	6	7	8
P	$0.02s$	$0.03s$	$0.03s$	$0.03s$	0.03	$0.35s$	$10s$	$2m24s$	$6.5h$
NC	$0.005s$	$0.005s$	$0.005s$	$0.005s$	$0.03s$	$0.2s$	$2s$	$35s$	$8m55s$
$P^{1,2}$	$0.017s$	$0.02s$	$0.02s$	$0.02s$	$0.02s$	$0.12s$	$1s$	$11s$	$2m24s$
$NC^{1,2}$	$< 0.001s$	$< 0.001s$	$< 0.001s$	$0.015s$	$0.05s$	$0.3s$	$1.8s$	$11s$	$1m30s$

partition size	0	2	4	6	8	10
P_2	$0.001s$	$0.002s$	$0.003s$	$0.05s$	$2.1s$	$3m13s$
P_{even}	$0.02s$	$0.03s$	$0.03s$	$10s$	$6.5h$	$> 8h$
$P^{1,2}_{even}$	$0.017s$	$0.02s$	$0.02s$	$1s$	$2m24s$	$> 8h$
$P_{even\ BS}$	$0.003s$	$0.003s$	$0.003s$	0.1	$19s$	$> 8h$
NC_2	$< 0.001s$	$< 0.001s$	$0.002s$	$0.01s$	$0.07s$	$0.7s$
$NC_{even\ BS}$	$0.008s$	$0.016s$	$0.016s$	$0.13s$	$1.7s$	$56s$
NC_{even}	$0.005s$	$0.005s$	$0.03s$	$2s$	$8m55s$	$> 8h$
$NC^{1,2}_{even}$	$0.03s$	$0.03s$	$0.03s$	$1s$	$50s$	$43m$
B	$0.06s$	$0.06s$	$0.06s$	$0.06s$	$3s$	$8m$
B_2	$0.01s$	$0.01s$	$0.01s$	$0.01s$	$0.1s$	$4s$
NCB_{even}	$< 0.001s$	$< 0.001s$	$0.007s$	$0.1s$	$4s$	$1m12s$
$B^{1,2}_{even}$	$0.2s$	$0.2s$	$0.2s$	$0.2s$	$8s$	$9m7s$

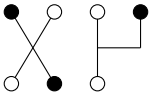
4.3 Algorithm Extensions

In Section 2.1.5 and in Section 2.1.4, we introduced generalizations of the classical partitions, namely the spatial partitions and the colored partitions. In the following section, we briefly discuss the algorithm presented and analyzed in Section 4.1 and Section 4.2 for the domain of spatial and colored partitions. Furthermore, we propose an helpful extension tool, called tracing, for our constructing categories algorithms. The whole implementation, including the algorithms discussed in this chapter, can be found in [Vol23].

4.3.1 Algorithms for Colored Partitions

We remain consistent in our data structure for partitions by using lists, similar to our previous approach. For classical partitions, we operate on a list of two lists, in which the first list represents the upper points and the second list the lower points. Since we are now incorporating colors, we use a list of lists, specifically four lists in the form of $List[List[], List[], List[], List[]]$. The first and second lists retain their purpose of representing the upper and lower points, respectively. The third list describes the colors of the upper points, while the fourth list denotes the colors of the lower points. Because we have a binary coloring, we use zeros and ones for the colors.

Example 4.7.

$[[1,2,3,3],[2,1,3],$
$[1,0,0,1],[0,1,0]]$ $\cong$ 

Since the vertical reflection is not defined in the domain of colored partitions, we only have to adjust the tensor product, the involution, the composition and the rotation. In general, we can simply overwrite the respective functions by running the classical operation algorithms with the lists representing the upper and lower points. In addition, for the composition, we have to check the colors of the input partitions as described in Section 2.1.2.
We operate on the colors as follows:

Algorithm 4.8 (Tensor Product.)**.** Let p, q be the input partitions of size n, m and $p_{color} \in \{0,1\}^n$, $q_{color} \in \{0,1\}^m$ the respective point colors, then we simply concatenate the lists of the upper point colors and the lists of the lower point colors.

Algorithm 4.9 (Involution.)**.** Let p be the input partition of size n and $p_{color} \in \{0,1\}^n$ the respective point colors. Then we simply swap the lists of the upper point colors and the lists of the lower point colors.

Algorithm 4.10 (Composition.)**.** Let p, q be the input partitions of size n, m and $p_{color} \in \{0,1\}^n$, $q_{color} \in \{0,1\}^m$ the respective point colors. For the final output colored partition, we simply use the upper point colors of q_{color} as the upper point colors and the lower point colors of p_{color} as the lower point colors.

Algorithm 4.11 (Rotation.)**.** Let p be the input partition of size n and $p_{color} \in \{0,1\}^n$ the respective point colors. Then we rotate p_{color}, while inverting the color of the rotated point.

Theorem 4.12. *The time complexities of the operations on colored partitions are equal to the operations on classical partitions.*

Proof. Follows directly from the fact, that we reuse the classical operation algorithms, while not adding greater extra complexity. □

With the algorithms for the operations of colored partitions as a foundation and the base partitions  ,  $\in P(1,1)$ and  ,  $\in P(0,2)$, we are able to run the constructing categories algorithm as presented in Section 4.1.

In order to get a brief application for this variation of the algorithm, we define the following colored partition categories (see [TW15b]) in Example 4.2.

Example 4.13 (colored Partition Categories)**.**

1. The set $\mathcal{O}_{loc} = \langle\rangle$, constructed from the base partitions
2. The set $\mathcal{O}_{grp,loc} = \langle$ $\rangle$
3. The set $\mathcal{O}_{glob}(2) = \langle$  $\rangle$

Now, as a demonstration, we can run the algorithm with the defined categories and get:

partition size	0	1	2	3	4	5	6	7	8
$\mathcal{O}_{loc}$	1	0	6	0	40	0	280	0	2016
$\mathcal{O}_{grp,loc}$	1	0	6	0	60	0	840	0	15.120
$\mathcal{O}_{glob}(2)$	1	0	12	0	160	0	2240	0	-

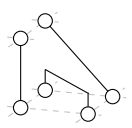
The table shows the number of partitions of size $0, 1, 2, 3, 4, 5, 6, 7$ and 8 in the corresponding categories.

Note, that categories containing the generator partition  are more extensive, because they contain for every partition $p \in P(n)$ all different coloring combinations (see [TW15b]). As a consequence, the computation of $\mathcal{O}_{\text{glob}}(2)$ for $n = 8$ took too long on my PC.

4.3.2 Algorithms for Spatial Partitions

Similar to colored partitions, we want to maintain the data structures consisting of lists. Like for classical partitions, we use the data structure $List[List[], List[]]$. But instead of storing the point values in the lists representing the lower and upper point, we store lists containing the point values.

Example 4.14.

$[[[1,2]],$
$[[1,3],[3,2]]]$ $\cong$ 

Since we are not defining the vertical reflection and the rotation for spatial partitions, we only have to concentrate on the three main operations the tensor product, the involution and the composition.

The general approach for designing algorithms for operations on spatial partitions is to use the techniques from the operations on classical partitions, we already discussed in Section 3.

Algorithm 4.15 (Tensor Product.)**.** Let p, q be the input spatial partitions. Similar to the classical partition tensor product, we simply concatenate the first list of p with the first list of q and the second list of p with the second list of q.

Algorithm 4.16 (Involution.)**.** Let p be the input spatial partition. Similar to the classical partition involution, we swap the upper with the lower points, be inverting the outer list.

Algorithm 4.17 (Composition)**.** Let p, q be the input spatial partitions. In order to design an algorithm for the composition, we reuse the techniques from the algorithm we already have for the classical partition composition. Initially we unfold the spatial partition, by resolving the lists in the two lists representing the upper and lower points. For example:

$$[[[1,2,1],[2,3,3]],[[3,4,4],[5,5,1],[6,6,6]]]$$

$$\Downarrow$$

$$[[1,2,1,2,3,3],[3,4,4,5,5,1,6,6,6]]$$

After this step, we run the classical composition algorithm in order to get the *new_ids* dictionary, which contains the information of old point values to new point values as described in Section 3. Afterwards, we iterate through the point values of the input spatial partition to change the them according to the *new_ids* dictionary. As a result we get our output pq.

Theorem 4.18. *The time complexities of the operations on spatial partitions are equal to the time complexities of the operations on classical partitions.*

Proof. Follows directly from the fact, that we reuse the classical operation algorithms, while not adding greater extra complexity. □

With the algorithms for the operations of spatial partitions as a basis and the base partitions $\overset{\circ}{|}^{(m)} \in P^{(m)}(1,1)$ and $\sqsupset^{(m)} \in P^{(m)}(0,2)$, we are able to run the constructing categories algorithm similar to the algorithm presented in Section 4.1.

While running the algorithm to construct the categories defined in Section 2.1.5, it appears that for spatial partitions, it is more common, that we have to let the intermediate step partitions exceed the *max_length* constraint. We saw in Section 4.2 that increasing the value of *max_length* leads to a substantial growth in runtime. As a result, the construction algorithm for spatial partition categories does not work as well as for classical partitions. Without allowing the intermediate step partitions grow beyond *max_length*, we get the following results

partition size	0	2	4	6	8	10	12
$[P]^{(2)}$	1 (1)	2 (2)	6 (6)	16 (20)	45 (75)	194 (312)	919 (1421)
$[NC_2]^{(2)}$	1 (1)	0 (0)	3 (3)	0 (0)	8 (10)	0 (0)	31 (35)
$[P_2]^{(2)}$	1 (1)	0 (0)	3 (3)	0 (0)	9 (15)	0 (0)	69 (105)
$P^{(2)}$	1 (1)	4 (6)	45 (75)	510 (1421)	6666 (37.260)	-	-
$P_2^{(2)}$	1 (1)	2 (3)	8 (15)	22 (105)	177 (945)	2374 (10.395)	-

Note that in this table, we consider every spatial partition in contrast to Section 4.2, where we only consider every $p \in P(0,n)$.

Following each entry in the table, we find the corresponding desired value enclosed within brackets, which can be retrieved from the sizes of the categories of the classical partitions. In other words, the brackets enclose the actual size, which may deviate from our calculations. This relation shows, that the problem of constructing categories is much more complex for spatial partitions. Nevertheless, the algorithm retains its significance, as demonstrated in Example 4.19.2 and in Theorem 4.20./4.21.

4.3.3 Tracing

Tracing is a helpful method for getting insights about the history of the emergence of the constructed partitions. Essentially, we use a dictionary to store for every partition how it was constructed.

At that time, the input for our construction algorithm is a list of partitions and the output partition size $n \in \mathbb{N}$. By introducing tracing, we add an extra optional input variable called $tracing \in \{true, false\}$. To enable tracing, we have to set the input variable tracing to *true*. By default tracing is disabled.

If tracing is enabled, we additionally get a dictionary containing the trace as an output, besides the constructed partitions of size n. Let p, p_1 and p_2 be partitions. Then an element of the tracing dictionary has the form

$$p : ((p_1), unary\ operation)$$

or

$$p : ((p_1, p_2), binary\ operation),$$

where the key p is the result of the corresponding value. The value is a tuple of the partitions and the operation. The operation is encoded as the first letter of the name of the operation. So in general, the key is the result of the value in our tracing dictionary.

Every time we construct a partition, which is not already contained in *all_partitions*, we add it to the dictionary as a key plus the partitions and the operation which produced it as the corresponding value.

In order to get the whole trace of a specific partition, we propose a function called *get_trace*, which iterates through the dictionary called *trace*. As a result, the algorithm prints out the whole trace from a specific partition down to the generator and base partitions via breath first search. This is implemented as follows.

```
def get_trace(trace, start):
"""track the trace with breath first search"""

    if start not in trace:
        print(f"Partition {start} not found in trace")

    track = [start]
    for p in track:
        if p in trace:
            print(p, " : ", trace.get(p))
            for i in trace.get(p)[0]:
                if i not in track:
                    track.append(i)
```

Here, the variable *trace* is the dictionary containing the trace from the construction algorithm and *start* is the partition we want to trace.
Note that we use breath first search, since it improves legibility compared to depth first search.

Example 4.19 (Applications)**.** Tracing has some applications in giving insights about the construction process. Based on these insights we can derive certain properties. For example:

1. **Finding different ways for constructing a partition**

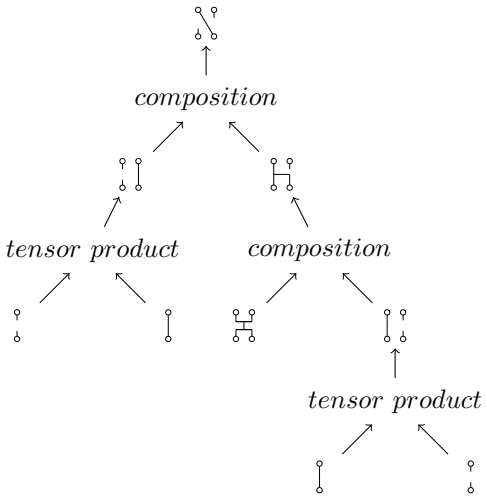
   Lemma 2.6. (c) in [Web13] says, that  $\in \langle$  $\rangle$. The proof contains a construction example of a rotated version of  , in which the intermediate step partitions become greater than the generator partitions. If we now want to know, whether, and particularly how, we can construct  without letting any intermediate step partitions expand greater than *max_length* (in this case 4), we can run our algorithm with tracing enabled.

   As a result, the function *get_trace* gives us the following trace:

   ((1, 2), (3, 1)) : ((((1, 2), (3, 2)), ((1, 2), (1, 1))), $'c'$)
   ((1, 2), (3, 2)) : ((((1,), (2,)), ((1,), (1,))), $'t'$)
   ((1, 2), (1, 1)) : ((((1, 1), (1, 1)), ((1, 2), (1, 3))), $'c'$)

((1,), (2,)) : ((((1,), (2,)), $'i'$),)
((1, 1), (1, 1)) : ((((1, 1), (1, 1)), $'i'$),)
((1, 2), (1, 3)) : ((((1,), (1,)), ((1,), (2,))), $'t'$)

We can visualize (the simplified version of) this trace in form of a tree as follows:



In this case, there is indeed a way to construct  with ⟨  ⟩, without expanding any intermediate step partitions greater than 4.

2. **Checking whether a set of generator partitions is minimal**

Let S be a set of generator partitions, representing a category $\mathcal{C}$. We want to know, whether there is a strict subset $s \subsetneq S$, which generates $\mathcal{C}$. If so, how can we produce $S \setminus s$ with s? As a human, this problem can be remarkably difficult.

With our construction algorithm combined with tracing, we can use the following method.

The first question, whether there is a subset $s \subsetneq S$, can be answered by running the construction algorithm with S and all strict subsets of s. If there is a set s, which generates for an input partition size of $n \in \mathbb{N}$ the same partitions as S for n, then we know that the generator partitions $S \setminus s$ can be removed. Furthermore, we can use tracing, in order to solve the problem of how $S \setminus s$ is produced with s. For example,

Let $\mathcal{C}$ be a category defined with $S = \langle$  ,  ,  , $\rangle$ as the set of generator partitions. If we apply the method described above, we can see that we can remove  without modifying $\mathcal{C}$. By enabling tracing, we get a counterexample, which illustrates why can be removed. As a result we get the following trace:

(((1, 2), (2, 3)), ((1, 2), (2, 3))) :
(((((1, 2), (2, 2), (3, 4)), ((1, 2), (3, 4))),
(((1, 2), (3, 4)), ((1, 2), (3, 5), (5, 4)))), $'c'$)

(((1, 2), (2, 2), (3, 4)), ((1, 2), (3, 4))) :
(((((1, 2), (2, 2)), ((1, 2),)), (((1, 2),), ((1, 2),))), $'t'$)

(((1, 2), (3, 4)), ((1, 2), (3, 5), (5, 4))) :
(((((1, 2),), ((1, 2),)), (((1, 2),), ((1, 3), (3, 2)))), $'t'$)

(((1, 2), (2, 2)), ((1, 2),)) :
(((((1, 1), (2, 3), (3, 4)), ((2, 4),)),
(((1, 2), (3, 3)), ((1, 4), (4, 2), (3, 3)))), $'c'$)

(((1, 2),), ((1, 2),)) :
(((((1, 2), (2, 3), (4, 4)), ((1, 3),)),(((1, 2),), ((3, 3), (1, 4), (4, 2)))), $'c'$)

(((1, 2),), ((1, 3), (3, 2))) : (((((1, 3), (3, 2)), ((1, 2),)), $'i'$))

(((1, 1), (2, 3), (3, 4)), ((2, 4),)) :
(((((1, 1),), ()), (((1, 2), (2, 3)), ((1, 3),))), $'t'$)

(((1, 2), (3, 3)), ((1, 4), (4, 2), (3, 3))) :
(((((1, 2),), ((1, 3), (3, 2))), (((1, 1),), ((1, 1),))), $'t'$)

(((1, 2), (2, 3), (4, 4)), ((1, 3),)) :
(((((1, 2), (2, 3)), ((1, 3),)), (((1, 1),), ())), $'t'$)

(((1, 2),), ((3, 3), (1, 4), (4, 2))) :
((((), ((1, 1),)), (((1, 2),), ((1, 3), (3, 2)))), $'t'$)

(((1, 1),), ()) : ((((), ((1, 1),)), $'i'$),)

(((1, 2), (2, 3)), ((1, 3),)) : (((((1, 3),), ((1, 2), (2, 3))), $'i'$),)

(((1, 1),), ((1, 1),)) : (((((1, 1),), ((1, 1),)), $'i'$),)

((), ((1, 1),)) : (((((1, 1),), ()), $'i'$),)

Note that the trace shows the first construction process, where the partition was generated, which does not represent the smallest possible way to construct the partition. As a result, a trace can also be fairly exhausting to retrace, as we can see above.

Further improvements can consist of implementing a method that is able to visualize a trace for improving the retracing process, as well as the readability. The tracing tree in the first application example could be a possible visualization choice.

Applying the approach demonstrated in Example 4.19.2, we can further simplify Theorem 3 in [CW16].

Theorem 4.20. *The following sets all generate $P^{(2)}$:*

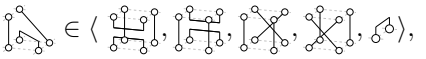
- $\langle \downarrow^{(2)}, \ldots \rangle$
- $\langle \downarrow^{(2)}, \ldots \rangle$
- $\langle \downarrow^{(2)}, \ldots \rangle$
- $\langle \downarrow^{(2)}, \ldots \rangle$
- $\langle \downarrow^{(2)}, \ldots \rangle$

Proof. Traces can be found in [Vol23] in the file *traces.txt*. As demonstrated in Example 4.19.2, we use the algorithm to generate traces, revealing the process of constructing each partition within the generating set from Theorem 3 in [CW16] for each new set. □

Theorem 4.21. *The following sets all generate $P_2^{(2)}$:*

- $\langle \ldots \rangle$
- $\langle \ldots \rangle$
- $\langle \ldots \rangle$
- $\langle \ldots \rangle$
- $\langle \ldots \rangle$
- $\langle \ldots \rangle$

Proof. Traces can be found in [Vol23] in the file *traces.txt*. As demonstrated in Example 4.19.2, we use the algorithm to generate traces, revealing the process of constructing each partition within the generating set from Theorem 3 in [CW16] for each new set. For the last generating set, we also produced a trace, showing the opposite direction



in order to ensure, that it only produces partitions included in $P_2^{(2)}$. □

5 Algorithm 2: Linear Combinations of Partitions

In this section, we will present an implementation for linear combinations of partitions. In general, we are interested in calculating polynomial terms, involving partitions. In order to achieve this goal, we will define and implement calculation rules for partitions, similar to the definitions in [GW21] and [GW19], but slightly adapted and simplified to our domain. The whole implementation, including the algorithms discussed in this chapter, can be found in [Vol23].

Definition 5.1. Let $p_1, ..., p_n$ be partitions and $f_1, ..., f_n$ be polynomials of the form $f_i(d) = \sum_{j=0}^{k} a_j d^j$ with degree at most $k \in \mathbb{N}$. Then we say that, the sum $f_1 \cdot p_1 + ... + f_n \cdot p_n$ is a polynomial partition term (i.e. a linear combination of partitions) of length n and degree at most k.

Definition 5.2. Let p_1, p_2 be partitions and $f(d) = \sum_{i=0}^{k} a_i d^i$ and $g(d) = \sum_{i=0}^{k} b_i d^i$ be polynomials of degree at most $k \in \mathbb{N}$. We define $f \cdot p_1 + g \cdot p_2$ as follows:

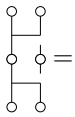
$$f \cdot p_1 + g \cdot p_2 = \begin{cases} (f+g) \cdot p_1 & \text{if } p_1 = p_2, \\ f \cdot p_1 + g \cdot p_2 & \text{if } p_1 \neq p_2. \end{cases}$$

Definition 5.3. Let $p_1 \in P(k,l), p_2 \in P(m,k)$ with $k, l, m \in \mathbb{N}$ and $f(d) = \sum_{i=0}^{k} a_i d^i$ and $g(d) = \sum_{i=0}^{k} b_i d^i$ be polynomials of degree at most $k \in \mathbb{N}$. We define $(f \cdot p_1) \cdot (g \cdot p_2)$ as follows:

$$(f \cdot p_1) \cdot (g \cdot p_2) = f(d) \cdot g(d) \cdot d^{\,loops} \cdot (p_1 \cdot p_2)$$

The multiplication between two partitions is defined by the composition operation and $loops \in \mathbb{N}$ is the number of omitted blocks (when composing p_1 and p_2), called loops.

Example 5.4.

$3 \cdot d \cdot$  $\cdot\, 4 \cdot d \cdot$  $= 3 \cdot d \cdot 4 \cdot d \cdot$  $= 12 \cdot d^3 \cdot$ 

5.1 Design Description and Implementation Details

Data Structure

In order to operate on polynomial partition terms, we need a suitable data structure. For the implementation, we import the package for classical partitions discussed in Section 3.1.

Let $f_1(d) \cdot p_1 + f_2(d) \cdot p_2 + ... + f_{n-1}(d) \cdot p_{n-1} + f_n(d) \cdot p_n$ be a polynomial partition term, where $\{f_1, f_2, ..., f_{n-1}, f_n\}$ are polynomials in d, $\{p_1, p_2, ..., p_{n-1}, p_n\}$ are partitions and $n \in \mathbb{N}$. As a data structure, we write every summand of the term in a list of size two. The polynomial coefficient is the first element and the respective partition is the second element. Every summand, represented by a list of size two, is contained in an outer list representing the whole term. As a result, we get the following structure:

$$List[List[f_1(d), p_1], ..., List[f_n(d), p_n]].$$

Note that the order of the outer list does not matter, since the addition is commutative. Since the order of the summands (i.e. the elements in the outer list) does not have to be taken into account, and each element in the list consists of a list of two entries, it is more efficient to use a data structure like dictionaries, in order to perform operations on the object. But as already discussed in Section 2.2.2, we also need unique keys. So initially, we simplify p as described in Definition 5.6, so that the data structure can easily transformed into a dictionary from partition part to term part.

Example 5.5.

$3 \cdot d \cdot$  $+ 4 \cdot d \cdot$  $\quad \cong \quad [[3 \cdot d, [[1,2],[1,1]]], [4 \cdot d, [[1,1],[1,2]]]]$

By initializing a polynomial partition term, the data structure simplifies the term automatically regarding zero summands and equal partition sums.

Example 5.6. The following example shows the simplification of the data structure by initializing a term.

$$0 \cdot \sqcup + 6 \cdot d \cdot \sqcup\!\sqcup + 10 \cdot d^{\,4} \cdot \sqcup + 5 \cdot d \cdot \sqcup\!\sqcup$$

becomes

$$11 \cdot d \cdot \sqcup\!\sqcup \cdot 10 \cdot d^{\,4} \cdot \sqcup.$$

Algorithms

For implementing the addition/subtraction and multiplication as defined above, we can override the addition and multiplication in Python with the following algorithms.

Algorithm 5.7 (Addition Algorithm)**.** Let p, q be terms, represented by the data structure above. Initially, we use a dictionary form of the partition term, namely *dict_term* from partition to the respective polynomial term. In succession, we iterate through every element e_1 in the outer list of q and check, whether the partition part of e_1 is in *dict_term*. If this is the case, we add the polynomial term part of e_1 to the value of the partition part of e_1 in *dict_term*. If the partition part of e_1 is not in *dict_term*, we add it in *dict_term* with the respective polynomial term part as value. After this process we can transform *dict_term* back to our data structure and output the result. This algorithm can be implemented as represented by the following Python pseudo-code

```
    def __add__(self, q):

        self.simplify()
        dict_term = dict()
        for i in self.partition_sum.copy():
            dict_term[i[1]] = i[0]

        for i in q:
            if i[1] in dict_term:
                add i[0] to dict_term.get(i[1])
            else:
                dict_term[i[1]] = i[0]

        out = [reversed(list(i)) for i in dict_term.items()]

        return Operations(out)
```

Note, that in this codesnipped *self* represents the term p.

Algorithm 5.8 (Multiplication Algorithm)**.** Let p, q be terms represented by the data structure above and *out* be an empty list. Since we need a method to get the *loop* value for the multiplication, we first need to modify the composition operation algorithm, introduced in Section 3. In order to get the loop value for the multiplication algorithm, we add an optional input *loop* to the composition algorithm. The *loop* input is set to *false* by default. If set to *true*, the algorithm outputs the composition result as well as the loop level. We can implement this by iterating over the points that are merged together in the composition process. If we encounter a point that is not part of the resulting partition, we can increase the loop level and proceed accordingly. This does not effect the runtime of the algorithm. Implementation details of this technique can be found in [Vol23].

With the modified composition, we iterate for every element e_1 in the outer list of p through every element e_2 of q. In every iteration, we perform a composition of the partition element in e_1 and the partition element in e_2, while setting the optional *loop* input to *true*. After this step, we append the multiplication of the coefficient part of e_1 and e_2 times d^{loop} with the result of the composition in the list *out*. The output of the algorithm is the out list as term object. This can be implemented as follows

```
    def __mul__(self, q):
    out = []

    for i in self:
```

```
5           for ii in q:
6               composition, loop = i[1].composition(ii[1], True)
7               out += [[expand(i[0]*ii[0]*(d**loop)), composition]]
8
9       return Term(out)
```

Note that *self* represents the term p. The *expand* function is part of the SymPy package as described in Section 2.2.4.

5.2 Evaluation

In this section, we first analyse the algorithms in Section 5.1 regarding their time complexity.

Time Complexity

As discussed in Section 5.1, the data structure simplifies the term, when creating an object. For this purpose, we use a *simplify* helper method, which iterates through the initial data structure, in order to check for zero summands and equal partitions.

Theorem 5.9. *Let a, b be polynomial terms and p be a partition. Then simplifying a polynomial partition term regarding zero summands and the distributivity rule*

$$a \cdot p + b \cdot p = (a + b) \cdot p$$

has the time complexity of $O(n)$, where $n \in \mathbb{N}$ is the number of summands in a term p

Proof. We iterate through the n summands of the input term p and store the already seen partitions in a dictionary from partition to coefficient. In every new *for* iteration we check, whether the partition of the summand is already in our dictionary. If so, we add the respective coefficients, which we assume to be $O(1)$. Since the access time complexity of a dictionary is in average $O(1)$ (see Section 2.2.2), we get an overall runtime of $O(n)$. □

Theorem 5.10. *Let p, q be polynomial partition terms and $n \in \mathbb{N}$ be the number of summands in p and $m \in \mathbb{N}$ be the number of summands in q. The average time complexity of the addition between p and q is $O(n + m)$.*

Proof. We know from Definition 5.9 that simplifying p has a time complexity of $O(n)$. Transforming p to a dictionary has also a time complexity of $O(n)$, since we iterate through p and perform in each iteration a constant time step, i.e. adding an item in a dictionary. The same applies for the other direction.

The main part of the algorithm consists of iterating through q and checking, whether the partition part of q is in the dictionary representation of p, followed by adding/overwriting an element to/in the dictionary representation of p. Accessing and adding an element in and to a dictionary has an average time complexity of $O(1)$ (see Section 2.2.2). As a result, we get an overall runtime of $O(n + m)$. □

Remark 5.11. Note that the problem of forming a sum of two polynomial partition terms has a linear lower bound time complexity of $\Omega(n)$, since we have to consider at least every element in the polynomial partition terms.

Theorem 5.12. *Let p, q be polynomial partition terms, $n \in \mathbb{N}$ be the number of summands in p, $m \in \mathbb{N}$ be the number of summands in q and $k \in \mathbb{N}$ be the size of the greatest partition in p and q. Then the time complexity of the multiplication of p and q is $O(nm \cdot k \log(k))$.*

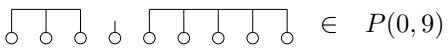
Proof. This follows directly from the fact, that we need to apply distributivity, where we perform for every pair in p one composition between the partition parts. Note that the runtime of the composition is $O(k \log(k))$ (see Section 3.2). □

5.3 Application with Interval Partitions

In this section, we will apply the techniques introduced in Section 5.1 to a specific problem that arose in some yet unpublished work by my supervisor Moritz Weber. In the extend of this thesis, we will only use it as an example application. First, we define interval partitions (see [Web17]).

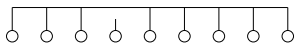
Definition 5.13 (Interval Partition)**.** Let $p \in P(0, l)$ and $i, j, m \in \mathbb{N}$. Then p is an *interval partition*, if it consists of blocks that satisfy the condition that if the points $1 \leq i < j \leq l$ belong to a particular block of p, then all points $i < m < j$ are also part of the same block.

Example 5.14.

1.  $\in P(0, 9)$

   is an interval partition.

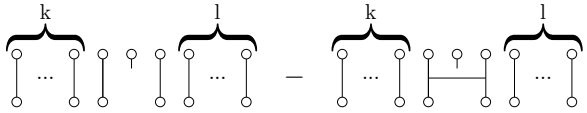
2.

 $\in \quad P(0,9)$

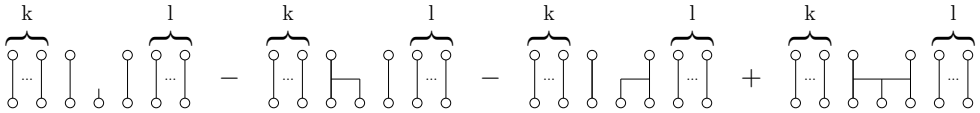
is not an interval partition.

Now, we will use our algorithms to analyse the zeros of some polynomial partition terms in connection with interval partitions. We define the problem space as follows:

Let p_m be the sum of all interval partitions of size m of the form $p_m := \sum_{p \in I_m} (-1)^b p$ with $b(p) \in \mathbb{N}$ the number of blocks in p and I_m as the set of all interval partitions of length m. Further, let $A_{k,l}$ be a polynomial partition term of the form



and $B_{k,l}$ be a polynomial partition term of the form



where l is the number of tensor products with from the left side and k the number of tensor products with from the right side of all partitions in $A_{k,l}$ and $B_{k,l}$.

We are interested in computing the zeros of

$$\circ \ldots \circ \cdot \prod_{i=1}^{n} S_i \cdot p_m,$$

where S is a list of combinations of A_{k_1,l_1} and B_{k_2,l_2} of length at most n, with variable values for k_1, l_1, k_2, l_2 for every summand, and m is the number of upper points of the partition parts of the last element in S.

Remark 5.15. Let $k \in \mathbb{N}$ be the upper bound for k_1, k_2 and $l \in \mathbb{N}$ be the upper bound for l_1, l_2. Then, we have

$$\sum_{i=1}^{n} 2^i \cdot k^i \cdot l^i$$

different possible combinations for $\circ \ldots \circ \cdot \prod_{i=1}^{n} S_i \cdot p_m$.

Remark 5.15 shows how large the different combinations of this problem can be regarding n, k and l. As a result, it is not possible to calculate the zeros for every polynomial partition term of this form even for small values for n, k and l. Nevertheless, it is possible to calculate the zeros for a few terms with the goal of finding a pattern.

Hence, we use a random generator function, which receives n, k and l as input plus a variable m, standing for the number of terms to generate.
The implementation and the list of results can be found in [Vol23]. The list contains:

10.000 different combinations with the upper bounds $n = 10, k = 3, l = 3$
100 different combinations with the upper bounds $n = 3, k = 5, l = 5$
25 different combinations with the upper bounds $n = 3, k = 6, l = 6$
50 different combinations with the upper bounds $n = 3000, k = 4, l = 4$

By using the SymPy package introduced in Section 2.2.4, we obtained the following results: All the factorizations of the resulting polynomial terms have the form

$$d(d-2)^a(d-1)^b, a, b \in \mathbb{N}.$$

As a result, all terms of the list have the zeros $\{0, 1, 2\}$. Furthermore, a is always equal to $k+l+1$ of S_1. It must be emphasized, that this data does not prove any of these properties. However, the data can be useful to get an idea about possible characteristics.

Remark 5.16. Note that the runtime complexity of this process is exponential with respect to the size of k and l, as m of p_m has to be of size $O(k+l)$. As a consequence, we need to perform $O(2^{k+l})$ compositions, which is the size of the summand $p_{O(k+l)}$.

Theorem 5.17. *Let S be a list of combinations of A_{k_1,l_1} and B_{k_2,l_2} of length n with variable values for k_1, l_1, k_2, l_2 for every summand and let p_m be the set of all interval partitions of size m of the form $\sum_{p\in I}(-1)^b p$ with $b \in \mathbb{N}$ as the number of blocks in p and I as the set of all interval partitions of length m. Then generating a list of the zeros of $u \in \mathbb{N}$ different versions of*

$$\circlearrowleft\cdots\circlearrowleft \cdot \prod_{i=1}^{n} S_i \cdot p_m$$

has a time complexity of $O(2^m \cdot n\log(n)\cdot u)$, with $m = O(k+l) = w$, where w is equal to the upper points of the partition parts of S_n.

Proof. Results from Definition 5.16, Definition 3.16 and the fact that there are 2^c combinations for a binary number of length c. □

6 Discussion and Outlook

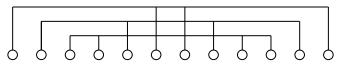
In this section, we bring this thesis to an end by presenting a brief discussion, including the algorithm's limitations and potential future research of this kind.
Overall, this thesis comprises the following contents:

1. Algorithms for Operations on Partitions (Section 3)
   (a) tensor product
   (b) involution
   (c) composition
2. Algorithms for Constructing Categories (Section 4)
   (a) classical partitions
   (b) colored partitions (Section 4.3.1)
   (c) spatial partitions (Section 4.3.2)
   (d) tracing (Section 4.3.3)
3. Algorithms and Data Structures for Linear Combinations of Partitions (Section 5)
   (a) addition
   (b) multiplication
   (c) application example (Section 5.3)

6.1 Limitations

The main limitation is the runtime of the algorithm for constructing categories, see Item 2. The origin for this lies in redundant computation, which is difficult to avoid without spending more resources than the redundancy costs. But because of the fast growth of the number of partitions in a category, even without redundancy we are not able to allow a significantly higher input partition size n.

Example 6.1. It would have been interesting to construct $\langle \Pi_3 \rangle$ (see [SR16]), where

$\Pi_3 \quad =$  $\in P(12).$

Even for small partition sizes n, it takes as much computation to construct $\langle \Pi_3 \rangle \cap P(n)$ as for $n = 12$, because we produce all partitions of size $\leq \max(n, \text{size}(\Pi_3)) = 12$.

This limitation becomes even more pronounced for colored partitions, since the coloring leaves room for more combinations. In the case of spatial partitions, a notable challenge arises as the generator partitions tend to be larger in size.

In addition, it seems that for spatial partitions, it is more common, that we have to let the intermediate partitions exceed the *max_length* constraint. As a result, constructing categories for spatial partitions is much more complex (see Section 4.3).

6.2 Future Work

Potential for future research lies in Item 1c, the composition algorithm.

Proposition 6.2. *Let $n \in \mathbb{N}$ be the number of nodes in a forest $\mathcal{F}$. Then a sequence of n union or find operations on $\mathcal{F}$ has a runtime of $O(n\alpha^{-1}(n))$. Where $\alpha^{-1}(n)$ is the inverse Ackermann function.*

Proof. See [Fis72]. □

Since the inverse Ackermann function grows slower than the logarithm function, $O(n\alpha^{-1}(n))$ is less complex as $O(n\log(n))$.

The improved runtime is achieved through the implementation of path compression and a technique known as *union by rank* (see [Fis72]). An integration of this technique in our composition algorithm could result in a runtime improvement for the whole algorithm. But note that, the construction algorithm would hardly benefit from it, since we are not dealing with partitions of a large size.

Additionally, with the algorithm presented in Section 4 and implemented in [Vol23], it would be interesting to investigate and potentially extend Theorem 5.8. in [CW16].

Theorem 6.3 (Theorem 5.8. from [CW16])**.** *The following subcategories of* $P_2^{(2)}$ *are all distinct:*

$\langle\rangle, \langle$  $\rangle$, $[P_2]^{(2)}$, $\langle$  $\rangle, \langle$  $\rangle, \langle$  $\rangle, \langle$  $\rangle, \langle$ $\rangle$, $P_2^{(2)}, \mathcal{C}_1 \times \mathcal{C}_2$ *with* $C_i = \{NC_2, \langle$  $\rangle, P_2\}$

In a broader context, tracing could be utilized to demonstrate that various sets of generator partitions representing a given category are not minimal.

Furthermore, it would be beneficial to create a program that visualizes a trace, with the aim of improving readability, as done in Example 4.19.2.

References

[BBC07] Teodor Banica, Julien Bichon, and Benoit Collins. The hyperoctahedral quantum group. *J. Ramanujan Math. Soc.*, 22(4):345–384, 2007.

[BCS09] Teodor Banica, Stephen Curran, and Roland Speicher. Classification results for easy quantum groups. *Pacific J. Math*, 247:1–26, 2009.

[BS09] Teodor Banica and Roland Speicher. Liberation of orthogonal lie groups. *Advances in Mathematics*, 222(4):1461–1501, 2009.

[CW16] Guillaume Cébron and Moritz Weber. Quantum groups based on spatial partitions. *Annales de la Faculté des Sciences de Toulouse*, 2016.

[Deo12] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science*. PHI Learning, New Delhi, [reprint.] edition, 2012. Print Book.

[Far22] Nicolas Faroß. Spatial pair partitions and applications to finite quantum spaces. *Saarland University - Department of Mathematics and Computer Science - Master's thesis*, 2022.

[Fis72] Michael J. Fischer. *Efficiency of Equivalence Algorithms*, pages 153–167. Springer US, Boston, MA, 1972.

[GW19] Daniel Gromada and Moritz Weber. Intertwiner spaces of quantum group subrepresentations. *Communications in Mathematical Physics*, 376(1):81–115, may 2019.

[GW21] Daniel Gromada and Moritz Weber. Generating linear categories of partitions. *Kyoto J. Math.*, 62(4):865–909, 2021.

[MSP+17] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017.

[MST20] Aye Aye Moe, Tin Tin Soe, and Moe Moe Thein. Review and comparison for collision resolution in a hash table. *International Journal of Research Publications*, 49, 2020.

[MW19] Alexander Mang and Moritz Weber. Non-hyperoctahedral categories of two-colored partitions, part i: New categories. *Journal of Algebraic Combinatorics*, 54:475–513, 2019.

[OEI] OEIS Foundation Inc. The on-line encyclopedia of integer sequences. Published electronically at `http://oeis.org`.

[SR16] Moritz Weber Sven Raum. The full classification of orthogonal easy quantum groups. *Communications in Mathematical Physics*, 341:751–779, 2016.

[Sta99] Richard P. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University Press, 1999.

[Sta15] Richard P. Stanley. *Catalan Numbers*. Cambridge University Press, 2015.

[Tar79] Robert Endre Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, oct 1979.

[TW15a] Pierre Tarrago and Moritz Weber. The classification of tensor categories of two-colored noncrossing partitions. *J. Comb. Theory, Ser. A*, 154:464–506, 2015.

[TW15b] Pierre Tarrago and Moritz Weber. Unitary easy quantum groups: the free case and the group case. *International Mathematics Research Notices*, 2017(18):5710–5750, 2015.

[Vol23] Sebastian Volz. Github repository. `https://github.com/sebvz777/Bachelorthesis`, 2023.

[Web13] Moritz Weber. On the classification of easy quantum groups. *Advances in Mathematics*, 245:500–533, 2013.

[Web17] Moritz Weber. Partition C*-algebras. `https://arxiv.org/abs/1710.06199`, 2017.

A Overview Categories

Category	Quantum Group	generators	Description	Law	Sizes	OEIS of the first 8 numbers ([OEI])
P	S_n		all partitions	Poisson	1, 1, 2, 5, 15, 52, 203, 877, 4140	Bell or exponential numbers (A000110)
P_2	O_n		partitions with blocksize 2	Gaussian	1, 0, 1, 0, 3, 0, 15, 0, 105, 0, 945	$a(n) = (n-1) \cdot a(n-2), a(0) = 1, a(1) = 0$ (A123023)
$P^{1,2}$	B_n		partitions with blocksize 1 and 2	shifted real Gaussian	1, 1, 2, 4, 10, 26, 76, 232, 764	Number of self-inverse permutations on n letters (A000085)
$P^{1,2}_{even}$	B'_n		even part of partitions with blocksizes 1 or 2	symmetry shifted real Gaussian	1, 0, 2, 0, 10, 0, 76, 0, 764	Number of self-inverse permutations on n letters (A000085) for even n, else 0
$P_{even\ BS}$	H_n		partitions with even blocksize	2-Bessel	1, 0, 1, 0, 4, 0, 31, 0, 379	Number of partitions of a $2n$-set into even blocks (A005046) with 0 if odd
NC_2	O^+_n	-	non-crossing partitions with blocksize 2	Semicircle	1, 0, 1, 0, 2, 0, 5, 0, 14, 0, 42	Catalan numbers (A000108) interpolated with 0's
P_{even}	S'_n		even part of all partitions	symmetry Poisson	1, 0, 2, 0, 15, 0, 203, 0, 4140	Bell or exponential numbers (A000110) if even, else 0
NC	S^+_n		non-crossing partitions	free Poisson	1, 1, 2, 5, 14, 42, 132, 429, 1430	Catalan numbers (A000108)
$NC_{even\ BS}$	H^+_n		non-crossing partitions with even blocksize	free Bessel	1, 0, 1, 0, 3, 0, 12, 0, 55, 0, 273	$a(n) = \frac{binomial(3n,n)}{2n+1}$ if n is even, else 0 (A001764)
$NC^{1,2}$	B^+_n		non-crossing partitions with blocksize 1 and 2	shifted Semicircle	1, 1, 2, 4, 9, 21, 51, 127, 323	Motzkin numbers (A001006)
NC_{even}	S'^+_n		even part of non-crossing partitions	symmetry free Poisson	1, 0, 2, 0, 14, 0, 132, 0, 1430	Catalan numbers (A000108) if even else 0
$NC^{1,2}_{even}$	B'^+_n		even part of non-crossing partitions with blocksize 1 and 2	symmetry shifted Semicircle	1, 0, 2, 0, 9, 0, 51, 0, 323, 0, 2188	Motzkin numbers (A001006) if even else 0
NCB_{even}	$B^{\#+}_n$		non-crossing partitions with balanced pairs and even number of singletons	squeezed shifted Circle	1, 0, 2, 0, 7, 0, 30, 0, 143, 0, 728	$a(n) = \frac{binomial(3n+1,n)}{n+1}$ (A006013)
B_2	O_n*		balanced partitions with blocksize 2	squeezed complex Gaussian	1, 0, 1, 0, 2, 0, 6, 0, 24, 0, 120	The aerated factorial numbers (A361522)
B	H_n*		balanced partitions	∞-Bessel law	1, 0, 1, 0, 3, 0, 16, 0, 131, 0, 1496	Number of block permutations on an n-set which are uniform (A023998) for even and 0 for odd
$B^{1,2}_{even}$	$B^{\#}_n*$		all balanced partitions of block size one and two with even number of singletons	squeezed shifted complex Gaussian	1, 0, 2, 0, 7, 0, 34, 0, 209, 0, 1546	number of $n \times n$ binary matrices with at most one 1 in each row and column (A002720) for even else 0